Theses and Dissertations   Student Graduate Works

3-23-2017

A Sandbox in Which to Learn and Develop Soar Agents

Daniel Lugo

A SANDBOX IN WHICH TO LEARN AND DEVELOP SOAR AGENTS

THESIS

Daniel Lugo, Captain, USAF

AFIT-ENG-MS-17-M-047

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-17-M-047

A SANDBOX IN WHICH TO LEARN AND DEVELOP SOAR AGENTS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Daniel Lugo, BS

Captain, USAF

March 2017

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-17-M-047

A SANDBOX IN WHICH TO LEARN AND DEVELOP SOAR AGENTS

Daniel Lugo, BS

Captain, USAF

Committee Membership:

Douglas D. Hodson, PhD
Chair

Maj Logan O. Mailloux, PhD
Member

Maj Alan C. Lin, PhD
Member

Abstract

It is common for military personnel to leverage simulations (and simulators) as cost-effective tools to train and become proficient at various tasks (e.g., flying an aircraft and/or performing a mission, among thers). These training simulations often need to represent humans within the simulated world in a realistic manner. 'Realistic' implies creating simulated humans that exhibit behaviors which mimic real-world decision making and actions. Typically, to create the decision-making logic, techniques developed from the domain of artificial intelligence are used. Although there are several approaches to developing intelligent agents; we focus on leveraging and open source project called Soar, to define agent behavior.

Learning how to create agents with Soar is a bit challenging; its language, program syntax, and execution flow is quite different from typical computer programming languages. It can be viewed as a Domain Specific Language (DSL) for agent creation. Soar itself, is a C++ software-based decision making component that is interfaced to an existing system; it's not a complete solution, it's a tool to enhance and ease the agent development process. The software package itself includes examples which define simple environments for agents to make decision against, but they are intentionally simple and lack the richness of typical military simulations.

This research interfaced the off-the-shelf open-source software product that facilitates the creation of 3D virtual worlds (called the AI sandbox) to the Soar package.

Because the world created by the sandbox is rich in features, easily configurable using a simple scripting system, and visually engaging, it's ideal as a learning platform to develop Soar intelligent agents aligned with military simulations. In summary, this research develops a platform (or learning environment) to learn how to develop Soar-based agents.

Acknowledgments

Foremost, I would like to express my sincere gratitude to my faculty advisor, Dr. Douglas Hodson, for his support, patience, and knowledge. His guidance was invaluable in the writing of this thesis.

Besides my advisor, I would like to thank the rest of my thesis committee: Maj. Logan Mailloux and Maj. Alan Lin, for their encouragement, insightful comments, and questions.

Last but not the least; I would like to thank my wife and kids for all their love.

Daniel "Hocka" Lugo

Table of Contents

List of Figures

x

xi

List of Tables

A SANDBOX IN WHICH TO LEARN AND DEVELOP SOAR AGENTS

I. Introduction

Chapter I explains why the Department of Defense needs intelligent agents and how the Soar cognitive architecture fills this need. We define the need for a visual, virtual environment to learn to develop intelligent agents. It concludes with a brief discussion that suggests using an existing AI sandbox software package for learning how to program Soar agents.

Overview

Today's military has a wide variety of weapon systems. These weapon systems include the machines, and the humans that operate them. Because of the critical nature of military missions to the country, it is important to be able to practice what we learn, test what we use, and evaluate what we need [1]. Armed forces must train to develop and maintain the proficiency necessary to effectually carry out the duties with which they are entrusted by their nations [2]. Unfortunately, the very nature of military missions makes it hard to set up real-world training. Mimicking a military mission in the real world requires many people and resources [3]. Simulations can greatly help the military perform test, evaluation and training. They also mitigate the cost of operating weapons systems and the risk of equipment damage, injuries, or loss of life [4]. Simulations, especially the ones used in training, need to provide the humans being trained a sense of realism so that they can be effective in preparing trainees for real-world situations. One way to make training simulations more realistic is to use human operators that can

1

perform different roles in the simulation. Human operators help create a more diverse and realistic environment for training [3]; however using human operators might not be the most viable solution because skilled manpower is limited and costly. An alternative to train more effectively is to simulate the human operators that perform the various roles in the simulated environment [3]. This can be accomplished by creating artificial autonomous intelligent agents to perform the same tasks the human operators would in the virtual world. If these intelligent agents are able to produce realistic behavior similar to a real human operator, then the cost of having humans involved in training can be mitigated and we solve the manpower issue of not having sufficient skilled operators to make training successful.

Intelligent Agents

An agent is anything that perceives an environment through sensors and acts upon that environment through effectors [5]. The term agent is broad and can include agents from living things to machines. This is why the terms artificial, autonomous and intelligent are used to describe the agents of interest. In the case of the simulations mentioned above, the intelligent agent is software that can make decisions with the information it is given and has a certain degree of freedom to choose what and when to perform the actions [5]. Figure 1 shows a diagram of a simple agent interacting with the environment. It is the agent's task to sense and process information from the environment and utilize it to decide what to do. An intelligent agent might also use previous knowledge of the environment to make decisions, or create knowledge through

2

experience. These capabilities make them useful in many applications; our interest resides in military uses for training.
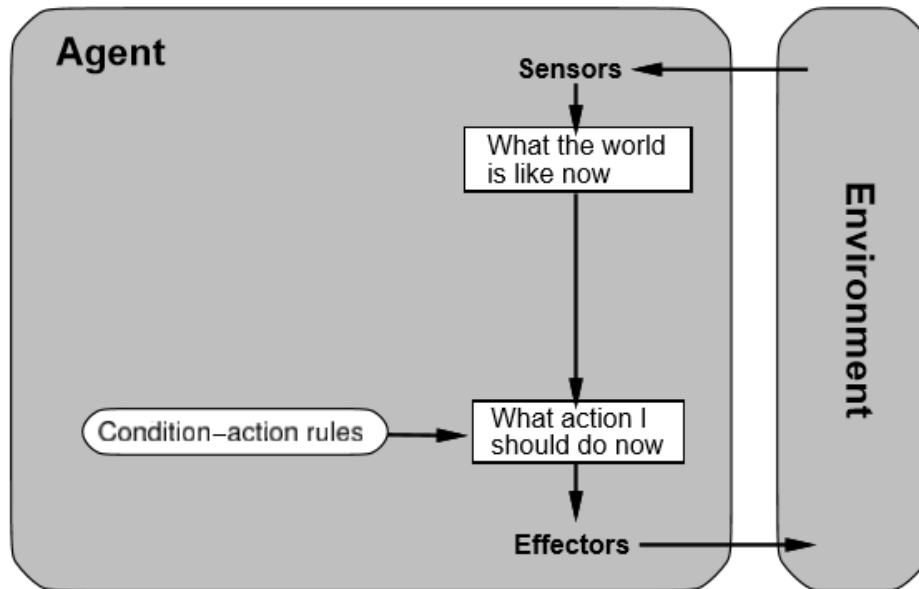
Figure 1. Simple agent interacting with environment [5].

Cognitive Architectures

One way to create an intelligent agent is to use a cognitive architecture. A cognitive architecture specifies the underlying infrastructure for an intelligent system; it provides us a definition of the computational structures required to store, retrieve, and process knowledge [6], [7]. For this research, we decided to use the Soar cognitive architecture. This architecture has a well defined mature framework, shown in Table 1. Moreover, it has been integrated with several systems including Joint Semi-Automated Forces (JSAF), which is a U.S. government owned and developed simulation system that is widely used in training and experimentation [8]. Soar is open source and available for Windows, iOS, Linux and Android operating systems. As an example, Soar has been

3

used to create intelligent agents that can model human behavior in military applications in the TacAir-Soar Human Behavior Model (TacAir-Soar). TacAir-Soar was built to determine if it was possible to replace human controllers with human behavior models for large-scale distributed training exercises [7]. TacAir-Soar agents were used in the Synthetic Theater of War-1997 (STOW-97) exercise to perform various missions flown by the US military and have been used in large-scale exercises and in training systems for pilots, ground controllers, and battlefield commanders [9], [10].

Table 1. Environments in which Soar has been integrated [7]

Category	Examples
Commercial and game-based environment simulations	SGI Flight Simulator, Descent 3, Quake II, Quake III, Unreal Tournament, Unreal 3, Crystal Space, LithTech, ORTS, FreeCiv, Gamebryo, OGRE, Torque, WildMagic, Empire, Zenilib, Planet Wars, Full Spectrum Command, Atari 2600 emulator
Military simulation environments	ModSAF, JSAF, OneSAF Testbed, OneSAF, VR Forces™, STAGE, simJr, JCATS
Robot simulation environments	Player/Stage, USARSIM, Microsoft Robotic Studio
Robot platforms	Early versions of Soar: Hero mobile robot, a PUMA robot arm, LEGO Mindstorms Recent versions of Soar: iRobot Create, Pioneer II robot platform, Splinterbot, iRobot's Packbot
Enterprise-software middleware	Soar has been interfaced to other software platforms via many different communication and interface protocols, including static libraries, shared object libraries, dynamically linked libraries, sockets, the Common Object Model, the Common Object Request Broker Architecture, the Distributed Interactive Simulation standard, High-Level Architecture (HLA), the Control of Agent-based Systems Grid, Apache Active MQ, and Java Message Service.
Academic research environments	General Game Playing server (Genesereth and Love 2005), RL-Glue (Tanner and White 2009), Robo-Cup soccer simulation (Marsella et al. 2001)

4

Virtual Environment

Intelligent agents need to gather information from an environment. Training in virtual environments provide lower cost than training in the real world. Virtual environments benefit from having a visual representation because it aids in understanding why the agent performs certain tasks. Since we can see the agent moving, acting upon the environment and responding to what other agents do, we are able to figure out how our agent sees the world and make decisions on its actions. Another aspect of virtual environments that is beneficial in testing intelligent agents, is the ability to be interactive. The environment can react and respond to what the agent does; this will give us the capability to create more complex agents that continually look at the world to decide what to do next. The ideal environment to learn to create intelligent agents should have various options for the agents to perform. These options can include how does the agent move, perception of the environment, communication with other agents, etc. The environment will dictate what the agent is able to do in the simulation.

Problem

We are interested in using Soar to create intelligent agents, due to the architecture's ability to create agents that can learn and resolve various problems, called impasses. But this is challenging, as Soar uses its own language to develop intelligent agents that is very different from other widely used programing languages like Java, C, C++, Python, C#, etc. Another issue with Soar is that the lack of example agents that have the tactical goals used in military simulations. In order to learn to create intelligent agents using Soar, we need to find a virtual, interactive and visual environment to test

5

agent behaviors. Having such an environment would help in learning how to program Soar agents because the user can see what the agents are doing and the interaction between the environment and the agents. During our research into Soar we encountered several environments that were being used to learn the basics of Soar programming. These environments consist of simple two-dimensional grid worlds where agents moved within. They are useful for introducing users to Soar agent programming but are limited to just simple behaviors. We would like to have more options on environments to aid in learning Soar agent development. Specifically, a dynamic environment where agents are capable of performing actions similar to military tactics would be beneficial. For example, moving at different speeds, following a leader, engaging an enemy, traveling to waypoints, and fleeing from danger are all actions that would make an environment more desirable.

Solution

While researching intelligent agents, we found a sandbox environment used to learn artificial intelligence game programming using a combination of C++ and Lua scripts. At first glance this seems to be a very good candidate environment to learn to program Soar agents because it has several of the characteristics mentioned above. We extended this sandbox environment to use Soar agents.

II. Background

Chapter Overview

The purpose of this chapter is to provide a background of the relevant components of the project. First, this chapter introduces Soar, its tools, and how it relates to established cognitive architectures. Next, this chapter presents examples of Soar applications including some used in military simulations and will discuss the current learning environments available for Soar agents. Finally, this chapter ends with a discussion of the potential sandbox to be extended so that users would be able to develop and test Soar agents.

The Soar Cognitive Architecture

Soar is a cognitive architecture, created by John Laird, Allen Newell, and Paul Rosenbloom at Carnegie Mellon University. It has been developed to be an architecture for constructing general intelligent systems. Soar has been in use since 1983, and has evolved through many different versions [11].

Soar Background

As opposed to many artificial intelligence systems that have been designed to excel at performing a single task, Soar was designed with a more general vision in mind [11]. Humans are able to perform many different tasks in an ever changing environment; we gather information and learn constantly. Because of this, creating an intelligent agent that has similar abilities to humans is very different than designing a system that is able to solve a certain type of problem very well. This is where cognitive architectures are beneficial. In the early 80s, Allen Newell and John Laird created a general cognitive

7

architecture that could use different methods to gain knowledge about a problem to complete a task. Soar was inspired by the human mind, its psychology and biology. Their goal was to make it possible for knowledge about a problem to be decomposed to smaller computational units that will then combine during problem solving phase, so that algorithms would be created from the interactions between the task needed to be completed and the knowledge of the problem [7]. This general approach to agent development covers a diverse set of domains and it is one of the reasons that Soar was found to be a good architecture for developing intelligent systems.

Capabilities of Soar

Figure 2 presents a structural view of the Soar cognitive architecture. First, Soar has the ability to receive inputs from an external environment using what is called the perception module. Changes to perception module are processed and sent to the working memory, which is similar to short-term memory. Working memory also initiates retrieval of long-term memories [7]. The different types of long-term memories are independent of each other and they have their own learning mechanisms which are discussed later in this chapter. With the input from the environment, and the knowledge that exist on working and long-term memories Soar can then make decisions. These decisions are then communicated to the environment through the action module. In summary, Soar can perceive information, use and create knowledge, learn from experience, and make decisions that in turn create behaviors to perform actions. With these fundamental capabilities, Soar supports planning, decision making and problem solving.

8
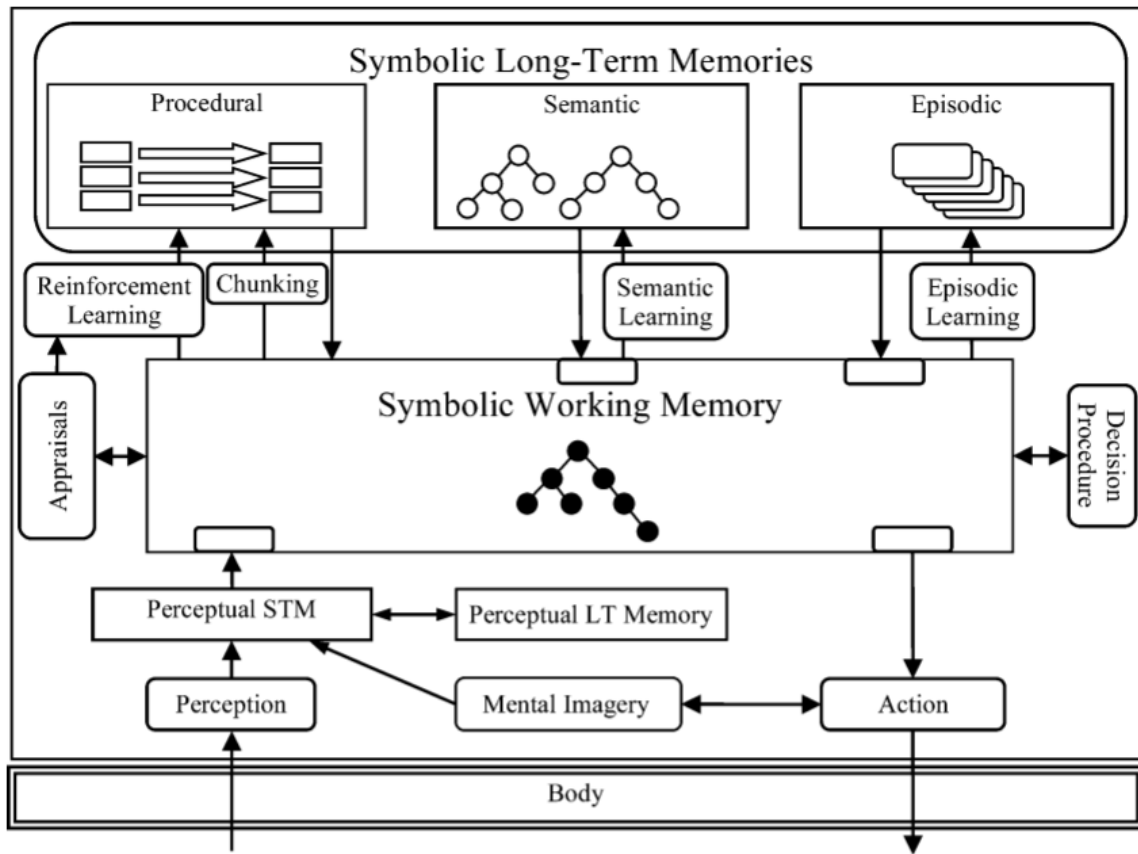
Figure 2. Soar Structure [12].

Soar Agents

The design of Soar is based on the hypothesis that all deliberate goal-oriented behavior can be cast as the selection and application of operators to a state [12]. When Soar is running, it is trying to match operators to the situation (current state) until the goal state is reached. This is why initially Soar was an acronym meaning, State, Operator and Result. Today the acronym is no longer used. The state is represented in working memory as a connected graph structure where the state is the root [13]. Figure 2 shows a graph representation of a state where you have two white blocks one named A the other named B.

9
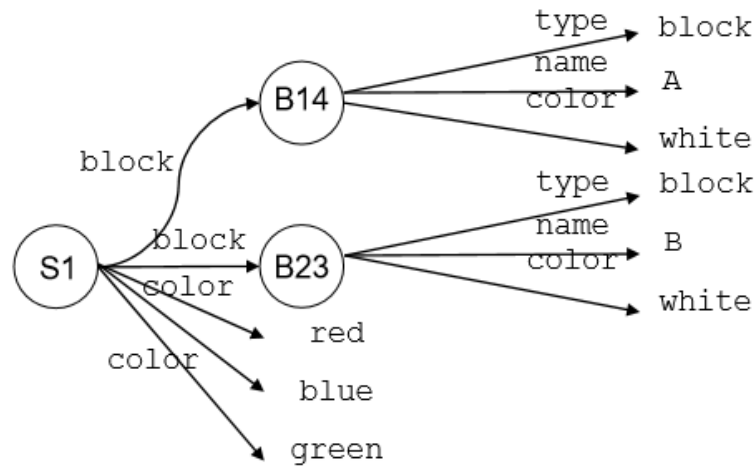
Figure 3. Representation of State in Working Memory [13]

Operators on the other hand are defined by rules. Rules are made of conditions and actions. An operator must have at least two rules, a proposal rule and an application rule [13]. The proposal rule will tell us when the operator should be checked and the application rule let us know what it does or how it changes the state. Operators are the ones that perform actions in the environment or in the agent's internal working memory. Once operators are selected, additional rules can match and apply the operator. This gives Soar a flexible representation of operators that makes it possible to have conditional, disjoint sets of preconditions and actions [7]. For a Soar agent to be able to work on a variety of problems it would have to include a lot of operators but if the problem was a simple one like deciding to go left or right then the number of operators would be less.

Knowledge, Memory and Learning

The various types of memories and learning mechanisms are discussed in more detail in this section. Working memory represents the current situation. For Soar agents

10

working memory consists of the perception of its world, results of intermediate calculations, active goals, and operators [12]. Working memory contains elements called working memory elements (WMEs), and each WME represents a specific piece of information [11]. An example of a WME would be "A1 is an agent" or "A1 is facing North". Each WME is an identifier-attribute-value triple, and all WMEs with the same identifier are part of the same object [11]. In the example "A1 is facing North", "A1" is the identifier, "facing" is the attribute and "North" is the value. Table 2 has an overview of memories and learning systems.

Table 2. Soar memories [7].

Memory and learning system	Source of knowledge	Representation of knowledge	Retrieval of knowledge
Procedural memory via chunking	Traces of rule firings in substates	Production rules	Exactly matches rule conditions, retrieves actions
Procedural memory via reinforcement learning	Reward and numeric preferences	Numeric preferences in rules	Exactly matches rule conditions, retrieves preference
Semantic memory	Working memory	Mirror of working-memory object structures	Exactly matches cue, retrieves object
Episodic memory	Working memory	Episodes: snapshots of working memory	Partially matches cue, retrieves episode
Perceptual memory	Perceptual short-term memories	Quantitative spatial and visual depictions	Deliberately recalls based on symbolic referent

Soar has three types of long-term memories: procedural, semantic and episodic. Procedural memory represents knowledge as production rules or numeric preferences in rules. These rules match a condition to an action. Production rules can modify working memory and generate preferences, which are used by the decision procedure to select an

11

operator [7]. Operators are where decisions are made and therefore where actions are chosen. Procedural memory has two learning mechanisms, chunking and reinforcement learning. Chunking is the process by which individual pieces of information are bound together into a meaningful whole [14]. It is an automatic process and is used to learn new production rules. Reinforcement learning utilizes a numerical reward signal to decide how to act in the world. The goal is to learn an action-selection policy such as to maximize expected receipt of future reward [15]. Reinforcement learning changes rules in procedural memory. These rules are then used to select the appropriate operator. Reinforcement learning is optional in Soar.

The second type of long-term memory is semantic memory. Soar's semantic memory is a repository for long-term declarative knowledge that supplements what is contained in short-term working memory and production memory [7]. It is beneficial in the case where there is a large amount of declarative knowledge because it does not have to be stored as rules like in procedural memory. It is also possible to preload knowledge from another source.

The last type of long-term memory is episodic memory. From Table 2 we can see that episodic memory is knowledge stored as episodes which are snapshots of working memory. In a biological sense, episodic memory represents our memory of experiences and specific events in time in a serial form, from which we can reconstruct the actual events that took place at any given point in our lives [16]. It includes all the information about these events. The difference between semantic and episodic memory can be

12

summarized as, semantic memory would only contain the facts that the intelligent system knows, and episodic would have all the information required to get to the facts.

Soar Processing Cycle (i.e. Execution)

The Soar processing cycle has five phases: input, proposal, decision, application and output. The Soar program will go through these cycles until Soar halts or the user interrupts the processing cycle [12]. Figure 4 shows the processing cycle: the normal rectangles are the processes that are performed by production rules and the rounded rectangles represent the task-independent processes [7]. The input phase is where new data comes into working memory. To get data from the environment and transfer it to working memory, a perception module (see Figure1) must be developed in a computer language that can interface with Soar. The interface to the perception module is via working memory through what is called the input-link [7]. This link is where the new structures are created for working memory.
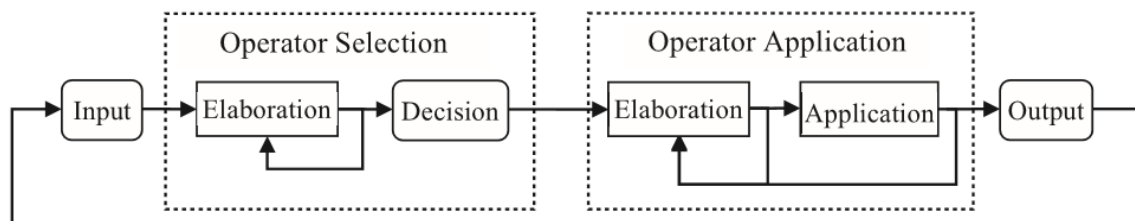
Figure 4. The Soar processing cycle [7].

The proposal state has several roles. This is where state elaboration, proposal of operators, and evaluation of operators occur. Soar combines these together so that rules for all of them fire in parallel as a wave [7]. In the elaboration state the production rules

13

fire and retract to interpret the new data from the perception module. During operator proposal, operators are proposed for the current situation and an acceptable preference is created. This means that the operator is a candidate for selection. After the operator is proposed it can be evaluated and compared to the other proposed operators in a process called operator evaluation. Evaluation rules match state structures and proposed operators, and create preferences for the proposed operators [7]. These preferences are then used in the decision procedure.

The decision procedure will use the preferences created earlier to select an operator. If there is not enough knowledge or conflicting preferences, then an impasse is detected and a new state is created [12]. When impasses are resolved Soar has the ability to learn what it did to solve it. If the decision procedure is possible then the current operator is changed in working memory. Decision making in Soar has several properties: they are made when there is knowledge, they are biased to the current operator, they are based on run-time integration of knowledge, agent can add more preferences to affect future decisions, and the computational complexity of the decision procedure is linear in the number of preferences [7].

After decision we have the application phase. Production rules will fire to apply the operator and they will fire and retract in parallel like in the proposal phase. Application phase has two sub phases, elaboration and application. Elaboration only happens if there are new rules that match the selected operator. Application is where progress is made, rules can change the state and create commands in the output-link. In

14

www.manaraa.com

the output phase the commands created before are processed and sent to the external environment.

Integrating Soar with External Environments

Many of the people interested in Soar want their current environment, either real or simulated, to be able to interact with Soar. For Soar to be able to interact with external environments it uses a mechanism that lets it receive inputs and send outputs to make changes to the external environment. This mechanism is defined by the Soar Markup Language (SML) interface. From within Soar, input and output are done by functions produced in the input and output phases of the Soar processing cycle. The structures for manipulating input and output in Soar are linked to a predefined attribute of the top-level state, called the io attribute [12]. These structures are called the input and output links and they either create WMEs or respond to WMEs that appear in the output structure. SML provides us with the interface to the outside so that the input and output data structures can be sent back and forth from the Soar Kernel to the external tool or environment. SML is based around sending and receiving commands packaged as XML packets [17].
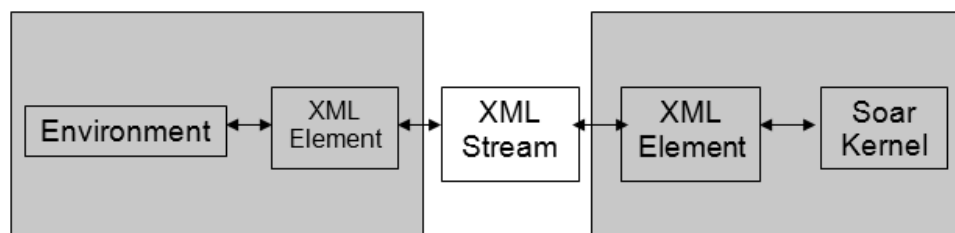
Figure 5. Sample communication diagram between external environment and Soar

15

The SML application programing interface (API) is implemented in C++ and is exported to Java and Python using SWIG. This gives the users some options to develop the client that communicates with Soar. In Figure 5 we have an example communication block diagram using SML between Soar and an external environment. In this example the environment creates an XML object that has information for Soar. That object is then converted into an XML stream (a sequence of characters) and it is sent in to Soar where it is converted back into an XML object and used by the Soar Kernel. The communication will work the same way from Soar to the environment. This is only one way of communication between Soar and an external tool or environment, if Soar and the tool or environment exist in a single process they will communicate back and forth using XML elements without converting them to streams. If Soar and the external tool or environment each run on a different machine they will communicate via sockets to send and receive an XML stream [18].

The SML API is only one part of integrating Soar with an external environment. Code needs to be written on both sides of the communication, the environment and the Soar agent. On the environment side the incoming XML object needs to be translated in something useful for the environment to process and in the Soar agent rules need to be created to work with the input from the environment.

Soar Debugger

The Soar Debugger is a Java based tool created using SML API to aid in the development and debugging of Soar agents. It is very useful because it has the capability of stepping through the Soar processing cycle and display information about working

16

memory, the operators that are selected, the rules that have been matched, and what is in the input and output links. The Soar Debugger can be used to run Soar agents locally or it could be connected to an external Soar kernel. Figure 6 shows a screenshot of the Soar Debugger.
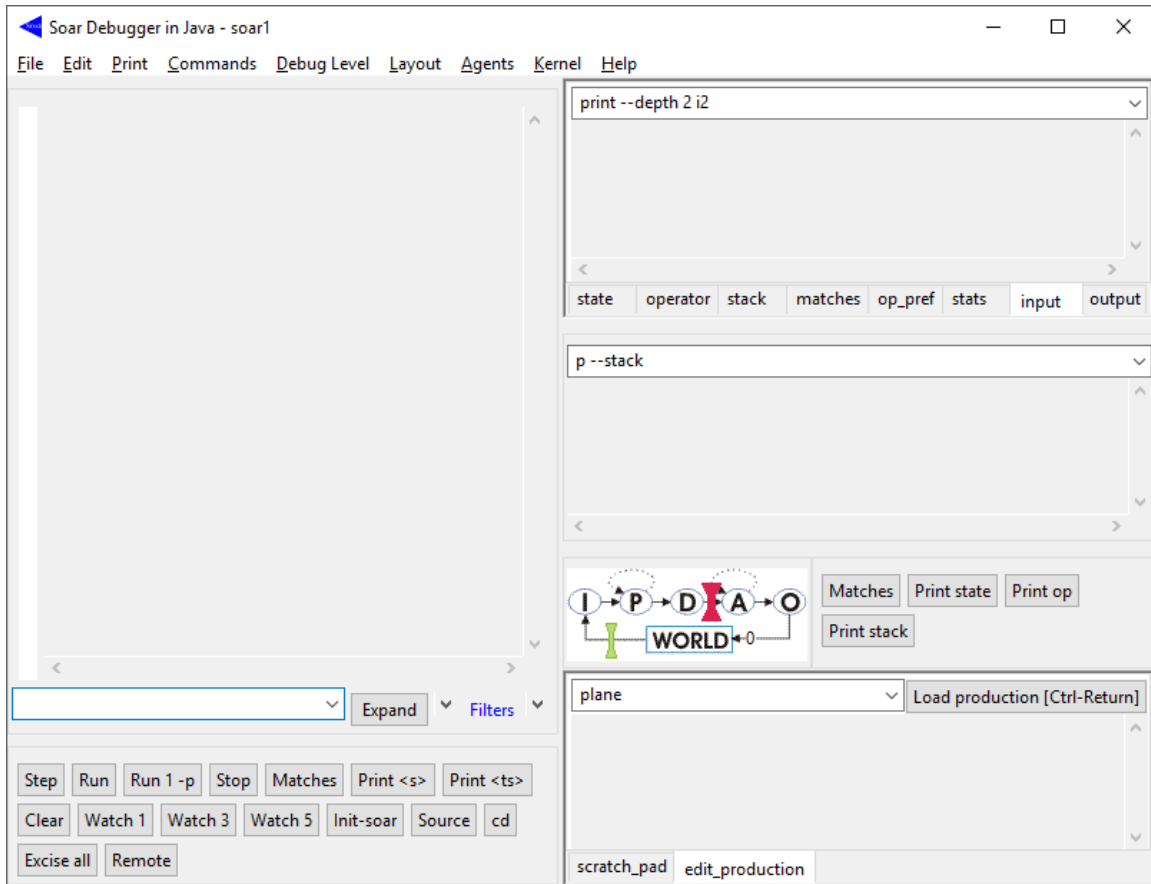
Figure 6. Screenshot of Soar Debugger

The tool is divided into multiple windows with various functions. The large window on the left is the Interaction window where the debugging information and print statements are displayed [12]. Bellow it is the command box where the user can either enter commands manually in the text box, or use the buttons below. These buttons are the common used commands and act as shortcuts. The right hand windows are used for

17

viewing and updating specific memory elements or mechanisms used in the Soar agent. A helpful option is located in the figure of the Soar processing cycle, where the user has the ability to add an artificial breakpoint between cycle phases. This aids in the debugging process because the user can view working memory after the decision or before the input phases.

Soar in Relation to Other Cognitive Architectures

There are different goals of cognitive architecture research; these goals are better explained by dividing them into three main categories [19]. The first category is research focused on biological modeling. This group of cognitive architectures are based on what we know about the brain: neurons, neural circuits, etc. These cognitive architectures are used to predict neural activity and cognitive behavior. Some examples are LEABRA and SPAUN. Next there are cognitive architectures that focus on psychological modeling, which are based on modeling human performance in a wide range of cognitive tasks and are used to predict human reaction time and error rates for psychological tasks. Some examples are ACT-R, EPIC, CLARION, LIDA, CHREST and 4CAPS. And the last category is cognitive architecture research geared toward AI functionality and is inspired by psychology and biology. Architectures from the last category emphasize in more complex cognitive processing and longer time scales. This is the category where Soar fits in. Other cognitive architectures in this category are Companions, Sigma, ICARUS and CogPrime [19].

Allen Newell classified human activity by different levels of processing and grouped them by time scales at twelve different orders of magnitude, starting with the

18

organelle level and extending through social activity measured in months [20]. Newell's Time Scale of Human Action can be used to distinguish between some of the cognitive architectures mentioned above. Table 3 shows where Soar fits in the Time Scale of Human Action compared to other architectures. Soar spans from the cognitive to the rational band making it a good option for creating intelligent agents.

Table 3. Newell's Time Scale of Human Action [19]

Scale (sec)	Time Units	System	Band
10^7	months		
10^6	weeks		Social
10^5	days		
10^4	hours	Task	
10^3	10 min	Task	Rational
10^2	minutes	Task	
10^1	10 sec	Unit task	
10^0	1 sec	Operations	Cognitive
10^{-1}	100 ms	Deliberate act	
10^{-2}	10 ms	Neural Circuit	
10^{-3}	1 ms	Neuron	Biological
10^{-4}	100 μs	Organelle	

Soar Applications

There are a variety of Soar applications as seen by the different environments that Soar has been interfaced to (Table 1). This demonstrates Soars ability to support the development of many different types of agents. Soar applications are usually grouped into four groups: expert systems, knowledge-intensive cognitive models, human behavior models, and autonomous agents [7]. These groups give us an idea of the type of problems Soar has been successfully used for. Expert systems use artificial-intelligence

19

methods to solve problems within a specialized domain that ordinarily requires human expertise. They utilize knowledge acquired from human experts which makes these types of systems excel at performing specific domain tasks. Soar has been used to develop expert systems in medical diagnosis [21], algorithm design [22], and antibody identification in immunohematology [23], [7]. In the cognitive models group of applications models are developed to study human performance in real-world tasks, Soar was used to develop a model to study human performance and learning in a simplified air traffic control task [24].

Human behavior models, like cognitive models are used to produce behavior similar to humans but human behavior models express theories at a higher level of abstraction and are usually used in simulated environments. They mostly focus on behavior generation in a realistic task environment [7]. Soar was used to develop pilot agents for a company of helicopters, a command agent that makes decisions and plans for the helicopter company and an approach to teamwork between the pilots [25]. This effort is called RWA-Soar. Other human behavior models have been used to model ground forces, air-traffic controllers and combat pilots (TacAir-Soar).

Tac-Air Soar is one of the largest applications of Soar, it has over 8000 rules. It integrates a wide variety of intelligent capabilities, including real-time hierarchical execution of complex goals and plans, communication and coordination with humans and simulated entities, maintenance of situational awareness, and the ability to accept and respond to new orders while in flight [10]. The successful use of TacAir-soar and RWA-Soar in a synthetic battlefield environment to perform missions with a large number of

20

aircraft demonstrates the potential to create real-time computer models that have the knowledge and tactics needed to be able to replace human operators in controlling these aircraft in a simulation. Figure 7 has more detail about RWA-Soar's architecture used in the modeling of the attack helicopter company. It shows the use of RWA-Soar in a real-time networked environment.
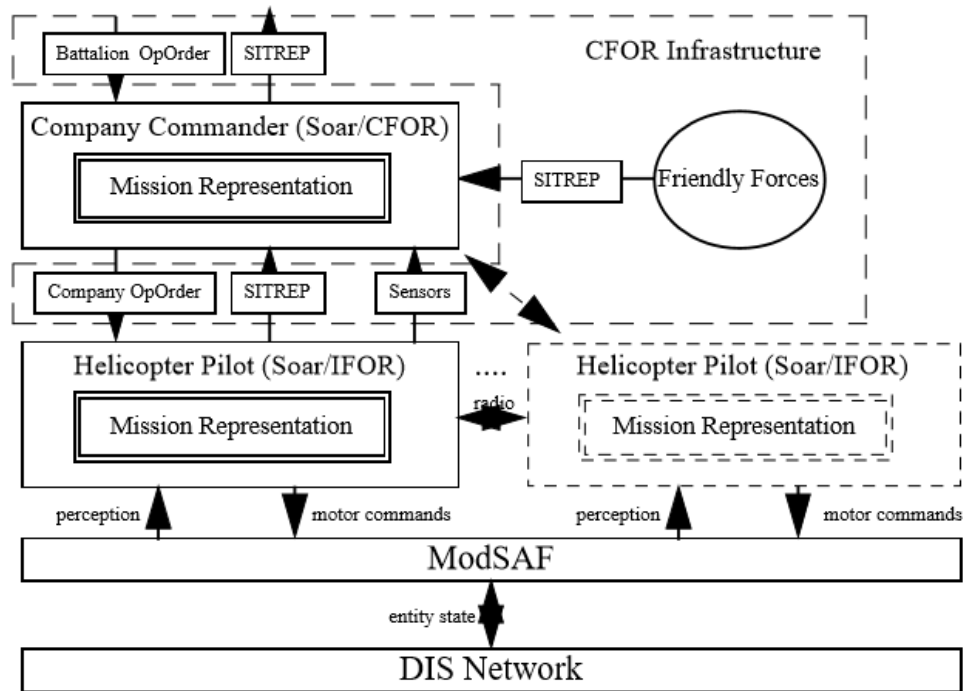
Figure 7. Architecture of the Attack Helicopter Company (Soar-RWA) [25]

Figure 8 is an example of TacAir-Soar operators, which correspond to the actions and goals human pilots perform during a mission. Abstract operators, such as "intercept an enemy", act as goals, which are dynamically decomposed by rules proposing more primitive operators to achieve the abstract actions. These rules test the current situation, including the available sensors and mission parameters, avoiding fixed, scripted responses [10].

21
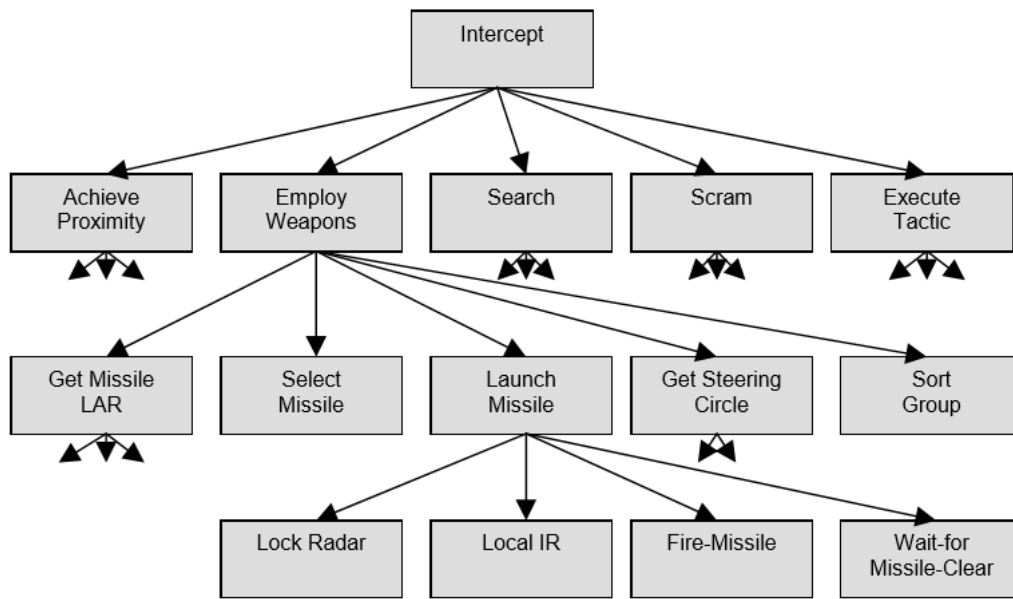
Figure 8. TacAir-Soar Example Operator Hierarchy Decomposition [10]

The last group of Soar applications that are mentioned above, are the autonomous agents. Autonomous agents are sometimes used in research and applications where building agents that communicate with external environments are desirable but they are not meant to model human behavior. Applications like controlling a robot or controlling a group of non-human characters in a first-person-shooter game are examples of autonomous agents. An example system is TacAir-Soar, an autonomous agent for the flight domain created to control a plane in a flight simulator that allows asynchronous control of the plane's throttle, ailerons, elevator and other control surfaces by an external system. Air-Soar controls the plane and is able to take off, level off and then follow a preset flight pattern including a series of turns and altitude changes, returning to land on (or near) the runway [26].

22

Soar Learning Environments

The main learning environment used to introduce users to the way Soar operators and rules are written is SimpleEaters. SimpleEaters is a game similar to PACMAN in which an agent represented as a yellow circle will consume food in a 2 dimensional grid world. It implemented using Java and interfaced with Soar via SML. The world consists of a square grid, 6 squares wide by 6 squares high. Figure 9 shows what SimpleEaters looks like.
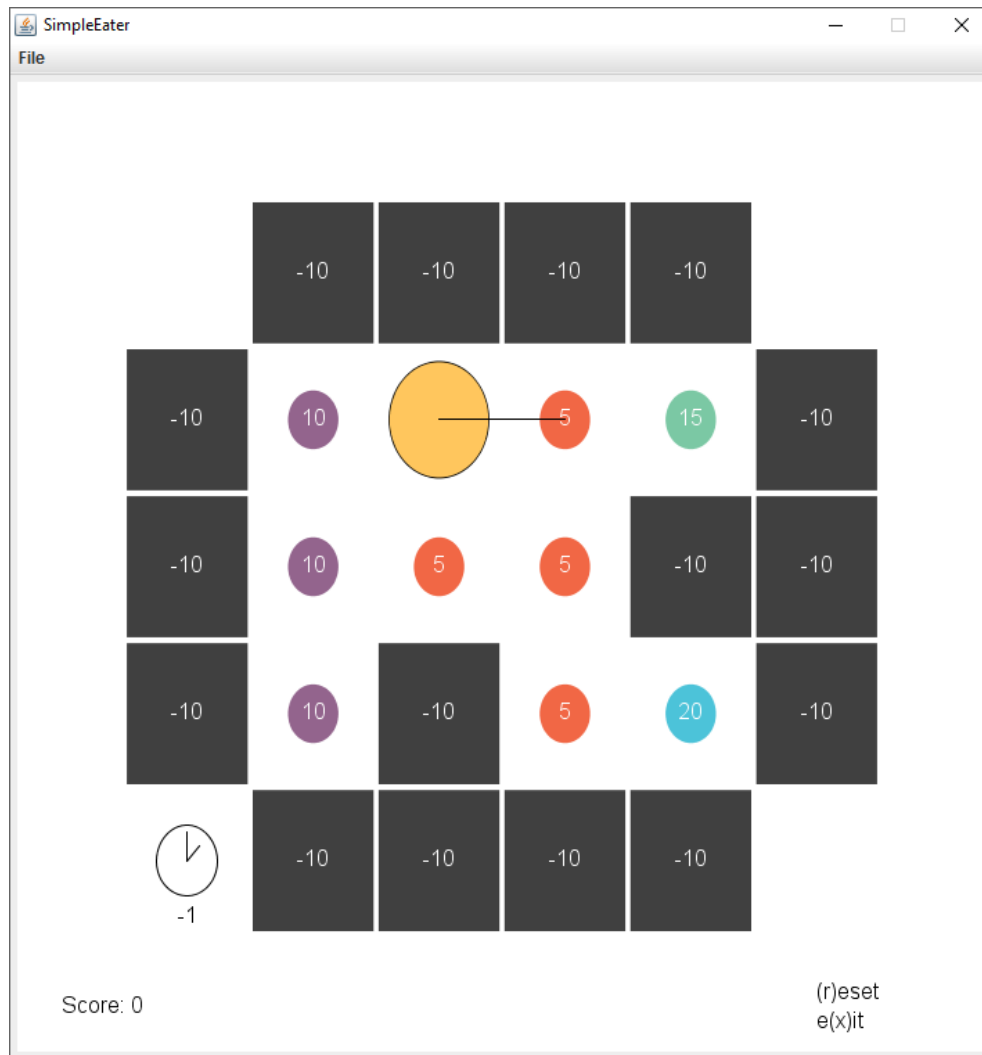
Figure 9. Screenshot of SimpleEater environment.

23

There are walls surrounding the playable area and the walls inside the playable area are generated randomly. The food is depicted as different color circles with different numbers that represent the amount of points the agent receives if that food is consumed. The eater consumes the food when it moves onto the square where the food is. There are 2 available moves the eater can perform, forward and rotate. The goal of the game is to consume all the food while getting the highest score. Score gets incremented by consuming food and decremented by 1 point for each move the agent makes.

Another learning environment available for Soar users is the Eaters game. Eaters world is similar to the Simple Eaters, it is a two-dimensional grid 15 by 15 squares. The goal in Eaters is the same but there are some expanded capabilities. The main difference is that there can be more than one eater in the world competing to consume the food. An eater can sense the contents of squares up to 2 squares away in all directions. On each turn, an eater can move one square north, south, east, or west. An eater can also jump two squares north, south, east, or west. An eater can jump over a wall or another eater. Whenever two eaters try to occupy the same cell at the same time, they collide. As a result of the collision, their scores are averaged and they are teleported to new, random locations on the board.

The two main learning environments are simple tools for learning the basics of Soar programming. Their limitations are good when starting to program because it gives the user the ability to create a simple agent that can complete a task. But once the user is comfortable with Eaters there is a gap in available training or testing environments. This

24

is where the need for a general purpose sandbox/environment is beneficial to learn how to develop Soar agents.

Potential Sandbox Environment

A solution to the lack of an interesting, rich learning environment for Soar would be to find an environment that would support all the graphics and physics needed while leaving the decision making for the agents to Soar. We have encountered an environment with high resolution graphics, a physics engine and a visual agent representation [29]. Figure 10 shows a screenshot of the sandbox environment with a simple agent.
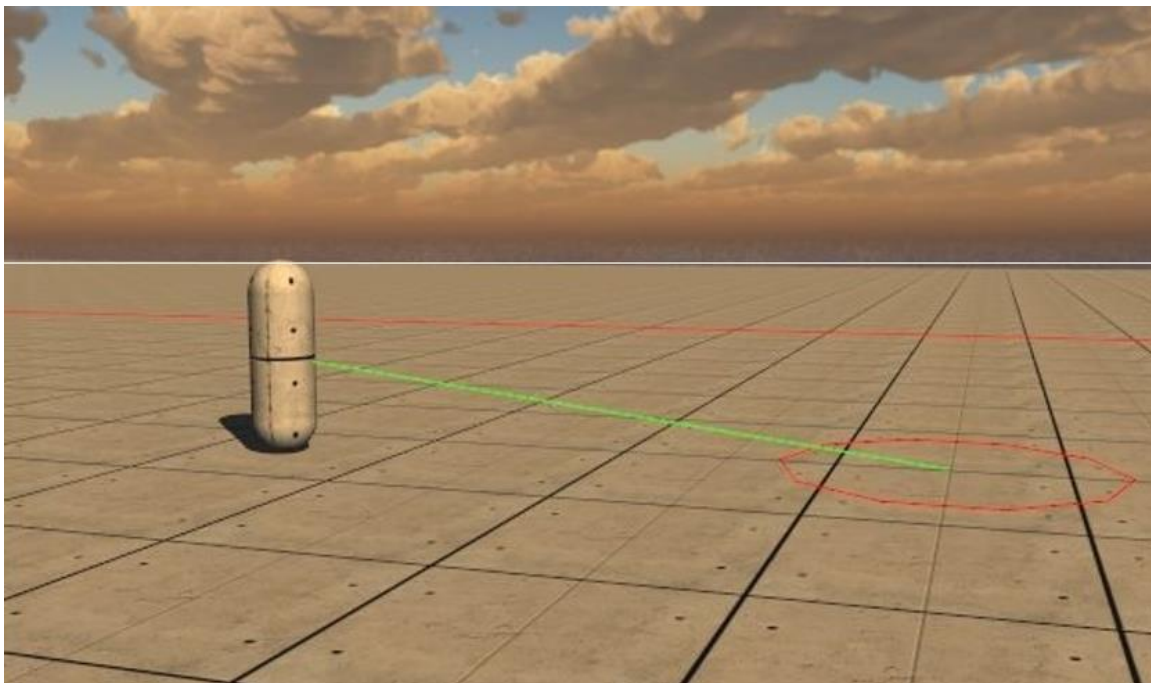
Figure 10. Screenshot of sandbox with agent

This sandbox environment is implemented with C++ and uses open source libraries. The code was packaged with a book named Learning Game AI Programing with Lua as a resource for the readers to learn to use Lua to control the agents [29]. The

book starts by giving us an overview of the sandbox environment and how the application is set up. The author then goes into creating agents and builds upon the agents by adding movement, navigation, decision making, perception and tactics. In Figure 11 we can see advanced agents in the sandbox environment. This book uses the sandbox environment to learn AI programing with Lua in a similar way that we would like to use it with Soar making this sandbox environment a natural choice to integrate Soar.
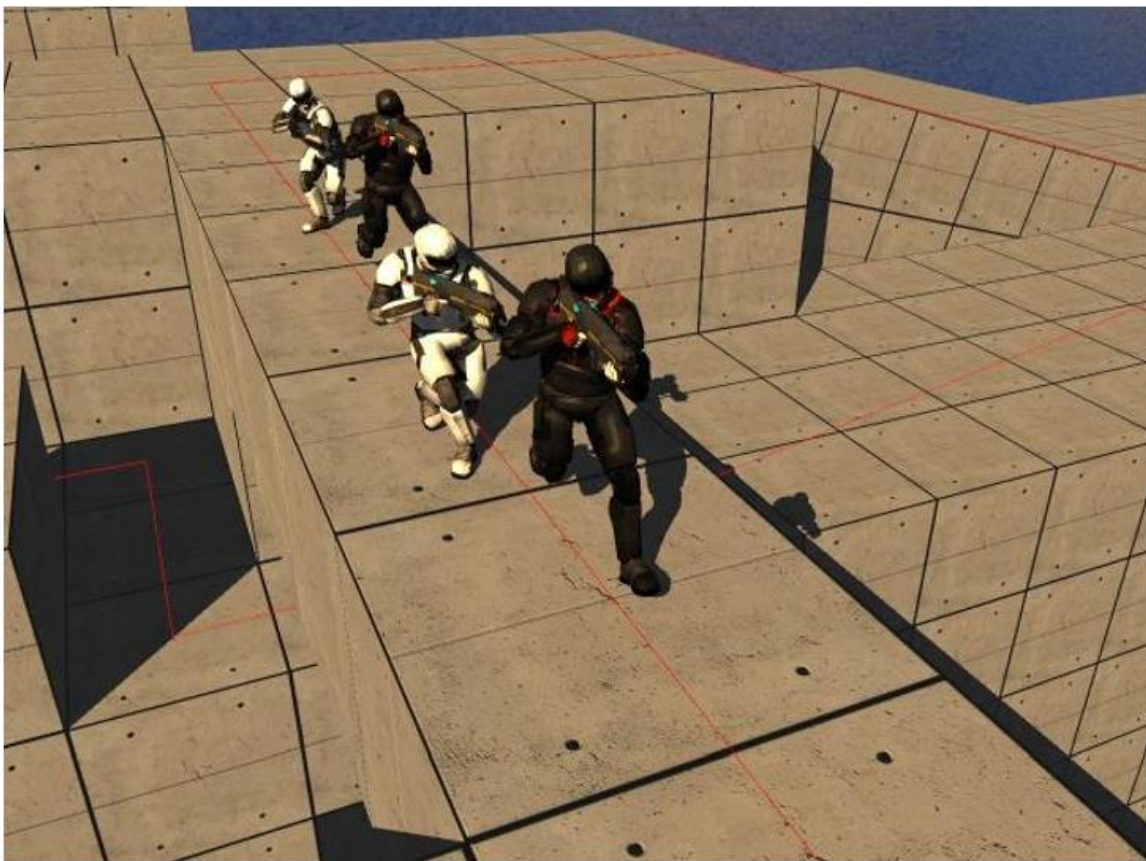
Figure 11. Sandbox with soldier agents

Lua is a scripting language that supports procedural programming, object-oriented programming, functional programming, data-driven programming, and data description [27]. It was created in the early 1990s in the Pontifical Catholic University of Rio de

26

Janeiro in Brazil and it is widely used in embedded systems, mobile devices, and games [28]. The Lua scripts are utilized in the sandbox environment to set up the visuals of the environment and to control the agents.

Some characteristics that make this sandbox environment an acceptable solution to learn Soar agent development are the ability to have multiple agents working in a team, having a three dimensional world, the agents have many movement options, and agents have the ability to shoot a projectile. These are important characteristics if we are to use Soar in military training applications. Table 4 shows the tools we have available to us through the sandbox to manipulate agents.

Table 4. Available agent properties in sandbox [29]

Agent Properties	Physics	Knowledge	Agent's Movement	Agent steering forces
Orientation	Mass	Target	Mass	Seeking
Location	Max force	Target Radius	Speed	Pursuit
Size	Max Speed	Path	Velocity	Fleeing
	Speed		Acceleration	Evasion
	Velocity		Force	Wandering
				Path following
				Avoid

27

As mentioned before this sandbox uses several open source libraries, these libraries provide debugging, graphics, input handling, physics, steering, and pathfinding. This is a list of the libraries and their functionality [29]:

- Ogre3D – Procedural geometry library that provides easy creation of objects such as spheres, planes, cylinders, and capsules

- OIS - Platform-agnostic library that is responsible for all the input handling and input device management

- Bullet Physics – The physics engine library that drives the agent movement and collision detection

- OpenSteer – Local steering library that is used to calculate the steering forces of agents

- Recast – Provides the sandbox with runtime navigation mesh generation

- Detour – Provides pathfinding on top of generated navigation meshes

Summary

In this chapter we reviewed the Soar cognitive architecture. Some of its capabilities include the ability to use knowledge to make decisions, the ability to learn from solved problems, and the use of various types of memory. These are characteristics that would make for robust and complex agents in simulations as we saw in the examples mentioned. The issue is that Soar's initial learning curve is steep and the learning environments available for the users, which were discussed in this chapter, do not have enough options to create and test agents that perform decisions similar to what they would be in a military simulation. This is why we propose the use of another sandbox

28

environment that has more options for the agents. In order to be able to use Soar in the sandbox we would need to extend the software's capabilities and build the communication between Soar and the sandbox environment.

III. Methodology

Chapter Overview

The purpose of this chapter is to describe the method used to extend an existing virtual environment to be able to use Soar to control agents in this environment. The chapter describes how to interface into Soar using SML. Next, it goes into detail on how the current sandbox environment is laid out and how the code works. Finally, it describes the work needed to extend the sandbox.

Interfacing with Soar using SML

Our approach to solving the problem starts with learning to develop the interface between Soar and the existing sandbox environment. First step is getting the Soar source code which is available online on https://github.com/SoarGroup/Soar or on the University of Michigan Soar page http://soar.eecs.umich.edu/Downloads. After Soar is built for the development environment, the user needs to include the SML client code, link the Soar libraries, and configure the environment for it to work properly. For C++, the user must include the header file "sml_Client.h" and link to the Soar shared library, named libSoar.so (Linux), libSoar.dylib (OSX), or Soar.dll(Windows). For Java the requirement for compiling a Java SML client is that the file sml.jar be in your class path.

For the client to communicate with Soar it first needs to create an instance of the Soar kernel or connect remotely to an existing Soar kernel. After creating the instance of the kernel the client can then create an agent. Once that is complete then there are several ways the client can use the SML API. These can be generalized as Soar control, input-output functions and registering for events. Examples of Soar control would be sending

30

specific commands for Soar to execute, loading agents or production rules to Soar, and synchronous run control of Soar. The client can ask Soar to run an agent for certain number of steps, run indefinitely or run until output from Soar is ready. This is useful especially for debuggers where synchronous run control aids in development and error tracing.

Basic input-output to Soar is achieved with the SML API and the input and output links. The client can create or edit WMEs in Soar's input-link. Graphs and trees of WMEs can be created using various SML API commands [17]. When completed the client can request the changes to be sent to Soar through the communication layer. The client can also read the contents of the output-link WMEs. This is where the client can evaluate what the agents' response was to the input, and use it to perform any actions the user deems necessary for that output.

The SML API can also be used to create an event driven client in which the program execution will be determined by events from Soar and the handling of those events in the code. There are various events the user can register for in the client. Some of the event handlers available for use are: production event handlers, system event handlers, run event handlers, agent event handlers, print event handlers, XML event handlers, and right hand side (RHS) event handlers [18]. This gives the developer various options for interacting with Soar on various parts of the processing cycle.

In interfacing Soar with an existing sandbox environment the SML API is used extensively to send the environment information and to receive commands from Soar. The code to extend the environment was written in C++ using Visual Studio. The Soar

31

Debugger, which is a SML client application, was used to look into the WMEs while Soar is performing its processing cycle. With this tool the user can see what is currently in the input link and what decisions the Soar agent proposed in the output link. The debugger will be connected to the remote Soar kernel that is being used by the C++ client for the Soar interface.

Before starting to work on the sandbox code tests were performed to verify if Soar SML would work in the sandbox environment. As a first step a small SML client application was created to test the development environment. This first application tested if the SML client would compile, create a kernel and be able to create a WME on the input link. After the first test application was completed successfully then we proceeded to create a second test application in which the communication to and from the kernel was tested. Using the Java based SimpleEater application as a reference the C++ test application was designed to have several components. These components were: a virtual world, an agent, and a goal. The world consisted of a long hallway with a wall at the end. The goal of the agent was to reach that wall. In this application the code needed to send the world information to Soar and receive a command. The agent after receiving the information about the world would decide to move or stop. This test was completed successfully and another test was planned to study how to read the output link by registering for Soar events using the SML code.

There are several ways the user can register for events using SML. These are registering for agent specific events or update events in the kernel [18]. For this test it was decided to register for the agent output event. By registering for this event we are

32

able to read the output phase after that phase is completed in the processing cycle. After the successful test of the SML event handlers, the next step was to study the sandbox environment to identify what parts need to change to extend it.

Sandbox Design

The sandbox code is designed as a framework or integrated set of libraries that share resources and functionality. One of the most important projects in the code is demo_framework which has all the core code used in creating the environment (i.e, the simulation or virtual world). The layout of the project as per the book, Learning Game AI Programing with Lua [29], is presented in Figure 12.

```
bin
    x32/debug
    x32/release
    x64/debug
    x64/release
build (generated folders)                 premake
    projects                                  premake
    Learning Game AI Programming.sln          SandboxDemos
decoda                                    src
lib (generated folders)                       chapter_1_movement (example)
    x32/debug                                     include
    x32/release                                   script
    x64/debug                                     src
    x64/release                               ogre3d (example)
media                                             include
    animations                                    src
    fonts                                     ...
    materials                             tools
    models                                    decoda
    packs                                     premake
    particle                              vs2008.bat
    programs                              vs2010.bat
    shaders                               vs2012.bat
    textures                              vs2013.bat
```

Figure 12. Layout of Sandbox project [29]

33

The src folder will be the one that needs to change, it contains the source code for the open source libraries and the sandbox itself. Each application created will live in its own folder in src. The applications will have its own header, src and script folders.

Before extending the sandbox we go into some detail on how the sandbox works. The sandbox uses C++ and Lua to create the environment and the agents. Lua scripts help set up the sandbox environment and initialize the agents. There is communication between the C++ code and the Lua scripts. C++ is able to call Lua functions and vice versa. The C++ code hooks into the Lua script through three predefined global Lua functions: Sandbox_Initialize, Sandbox_Cleanup, and Sandbox_Update [29]. Lua scripts take advantage of C++ capabilities to do all the "heavy lifting" and simply call C++ functions that are exposed to Lua through function binding. This function binding is created by defining a constant array that maps the function's name within Lua to the C++ function that should be called.

Demo Framework

The demo framework project is the core of the sandbox and contains all the important classes that make up the sandbox. It uses an update, initialize, and cleanup design. From this code the classes being used on the initialization and the update were identified and modified to extend the sandbox functionality. Figure 13 shows a class diagram of some important classes used when running the sandbox application. Following are a list of classes and their purpose:

1. BaseApplication – responsible of configuring the application window, process input commands and interface with the Ogre3D library. It implements the

34

Ogre::FrameListener interface for updating and drawing in the sandbox. It also implements Ogre::WindowEventListener to handle window events. BaseApplication also interfaces with the Object-Oriented Input System (OIS) library, which is responsible for all the keyboard and mouse handling events.

2. SandboxApplication – this is a child class of BaseApplication. When the user creates an application they will be creating an instance of this class. This class implements the Cleanup, Draw, Initialize and Update functions inherited from the parent class. A function of interest that is implemented by this class is CreateSandbox which creates an instance of the sandbox and connects the sandbox with the Lua script that has the sandbox information.

3. Sandbox – this class handles the calls to the Lua script and has the sandbox data. It utilizes the Ogre::SceneNode to place the sandbox in the world.

4. Agent – this class encapsulates the agent data and handles calls to the agent Lua scripts.

5. The following utility classes are implementing the utility pattern which separates the need for the agent and sandbox classes to know how to interact with the Lua VM. All the data manipulation interacting with the Lua VM is performed by the utility classes [29].

   a. AgentUtilities – handles all actions performed on AI agents from Lua

   b. SandboxUtilities – handles actions performed on the sandbox from Lua

   c. LuaScriptBindings – describes the C++ functions that are exposed to the Lua scripts.

35
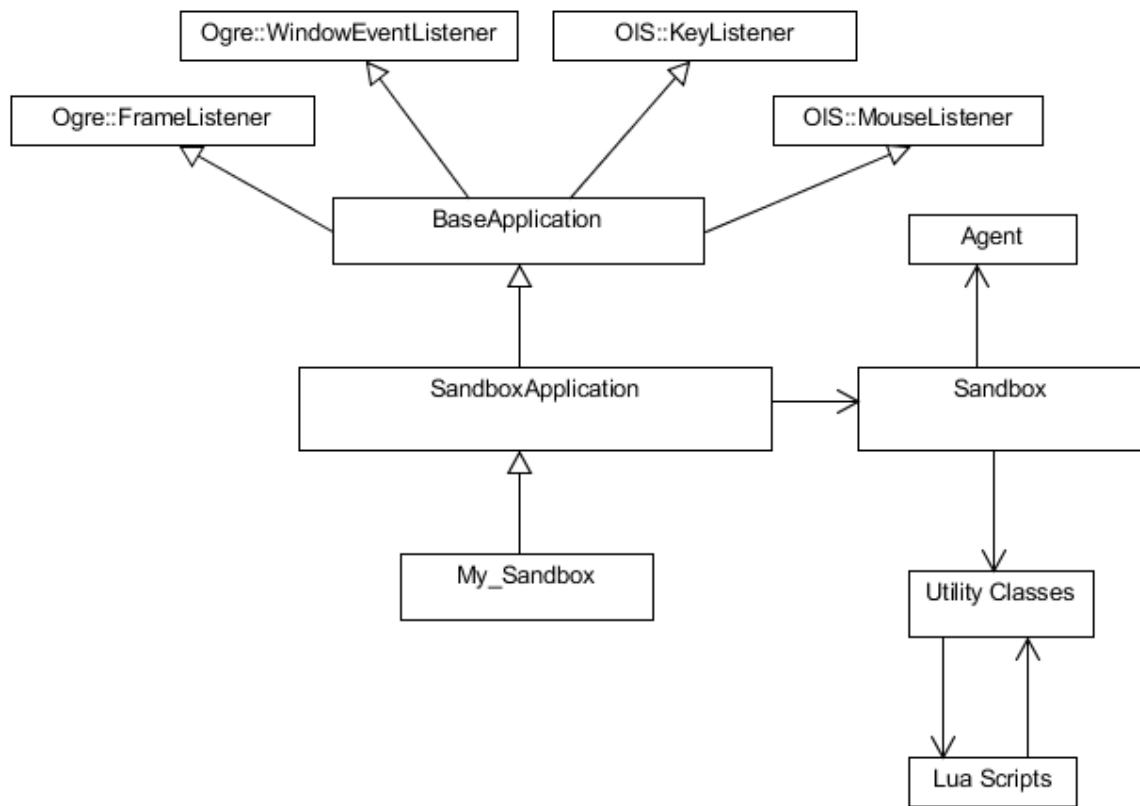
Figure 13. Class diagram of the important classes in the sandbox

Sandbox Initialization

The following is the process the application takes when initializing. It starts with a call to the run method which is part of the BaseApplication class this starts the setup process where Ogre plugins are initialized and the resource paths are configured. Then the configuration of the rendering window is processed where scene managers and the camera is set up. The instance of SandboxApplication is initialized. The Lua file manager sets the paths for the scripts and the sandbox is created by pairing it with the Lua script file that initializes some of the sandbox options.   This is where the utility classes perform the work that the Lua scripts demands. Some options set up by Lua in

36

the sandbox are sky, agents, floor, and lighting. The final step is to create the frame

listeners to start the update loop.

Sandbox Rendering a Frame

To begin the rendering of the application the method Root::startRendering is

called from the OgreRoot class. Here is part of that code:

```
void Root::startRendering(void)
    {
        assert(mActiveRenderer != 0);

        mActiveRenderer->_initRenderTargets();

        // Clear event times
            clearEventTimes();

        // Infinite loop, until broken out of by frame listeners
        // or break out by calling queueEndRendering()
        mQueuedEnd = false;

        while( !mQueuedEnd )
        {
                //Pump messages in all registered RenderWindow windows
                WindowEventUtilities::messagePump();

                if (!renderOneFrame())
            break;
        }

    }
```

Inside the while loop we see the call to render one frame, this is an infinite loop which

the user can exit by calling queueEndRendering(). This rendering process first updates

all the frame listeners inside the BaseApplication. The BaseApplication calls the update

for the SandboxApplication object which also calls the Sandbox object to update. The

Sandbox object goes through the list of agents and calls each of their update methods.

From the Agent class then the Lua agent's update method is then called. With this

knowledge of the initialization and rendering process, a plan can be made to extend and

create classes to add the Soar agent functionality to the sandbox environment application.

37

Code Changes

To have Soar agents controlling the entities in the sandbox, changes had to be made in the Agent class. The Agent class became a parent class for a Soar agent class. A Lua agent class had to be created to make use of the Lua script functionality. This gave us the ability to have both Lua and Soar agents in the same environment and kept the current agents created for the sandbox working. We added the capability of having Soar agents without affecting how the sandbox works.

A parent class named BaseAgent was created. It has all the base functionality that the Agent class had. Child classes named LuaAgent and SoarAgent were also created so that the communication with the Lua scripts and Soar kernel will be handled by dedicated classes. From researching the code and looking at where is the Agent class was used, several classes were identified as the ones that had to be modified. Here is a list of the code files that were changed:

- o Agent.h – This class was kept in the project but functionality was replaced with BaseAgent, it includes some virtual methods for the SoarAgent class and LuaAgent class to implement.

- o Agent.cpp – Source code was replaced with BaseAgent.

- o Sandbox.h – The Sandbox class keeps track of the agents by maintaining a list. Code was added to be able to keep lists for BaseAgent, LuaAgent and SoarAgent class instances.

- o Sandbox.cpp – Source code was changed to keep lists for BaseAgent, LuaAgent and SoarAgent.

38

- AgentGroup.h – This class handles agent groups, the methods that contain instances of Agent class were changed to use the BaseAgent class.

- AgentGroup.cpp – Source code was modified replacing Agent instances with BaseAgent.

- AgentPath.h – Derived from OpenSteer, the methods that contain instances of Agent class were modfied to use the BaseAgent class.

- AgentPath.cpp – Source code modified to add BaseAgent functionality.

- AgentUtilities.h – Multiple methods to initialize, update and perform cleanup. This class had methods created for the SoarAgent class and LuaAgent class. This is the intermediary between Lua, Soar and the sandbox.

- AgentUtilities.cpp – Important class that has the functionality to communicate with Lua. Source code was added so that LuaAgent and SoarAgent can communicate with the Lua virtual machine.

- SandboxUtilities.h – This class has methods to create Lua and Soar agents in the sandbox. New methods were written to perform these tasks.

- SandboxUtilities.cpp – Source code added for the creation of Soar and Lua agents from the Lua script.

- LuaScriptBindings.h – This class has the list of methods that can be called from the Lua script. Current methods involving the Agent class were changed to use LuaAgent and SoarAgent classes.

- LuaScriptBindings.cpp – Source code modified to add the LuaAgent and SoarAgent capabilities.

- SandboxObject.h – The Agent class is friend class to SandboxObject. The new implementation uses BaseAgent as a friend class to SandboxObject.
- SandboxObject.cpp – Souce code changed BaseAgent for Agent as friend class.
- NavigationUtilities.cpp – The default configuration for the agent movement is set up here. Code was modified to use BaseAgent class.

Summary

This chapter described the method used to extend an existing virtual environment to use Soar to control the agents. It also described how to interface into Soar using SML. Next, it went into detail on how the current sandbox environment is laid out and how the code works. Finally, it discussed which code files were changed to extend the sandbox.

40

IV. Results

Chapter Overview

This chapter contains information about the initial tests performed with Soar SML in the development environment. It presents the design and code changes implemented to utilize Soar to control agents in the sandbox. Then, the chapter goes into the various agents created to test the solution. It ends with the summary of the results of the Soar agent tests.

Initial SML Testing

This section details the tests completed to verify Soar and SML in the development environment. The initial test was designed to create an instance of the Soar kernel and to populate a WME in the input link. To set up the development environment the header file "sml_Client.h" was included and the Soar shared library was linked. The instance of the Soar kernel was created utilizing the command Kernel:: CreateKernelInNewThread(). After testing that the kernel was created successfully the input link was created and information was committed to Soar's working memory. To verify if this was successful, we utilized the Soar debugger. By connecting the Soar debugger to the kernel (created in the SML application), we were able to inspect the contents of the input link. It was verified that the WME that was created in the initial test existed in the Soar kernel in the input link. Figure 14 shows a screenshot of the input link from the Soar debugger.
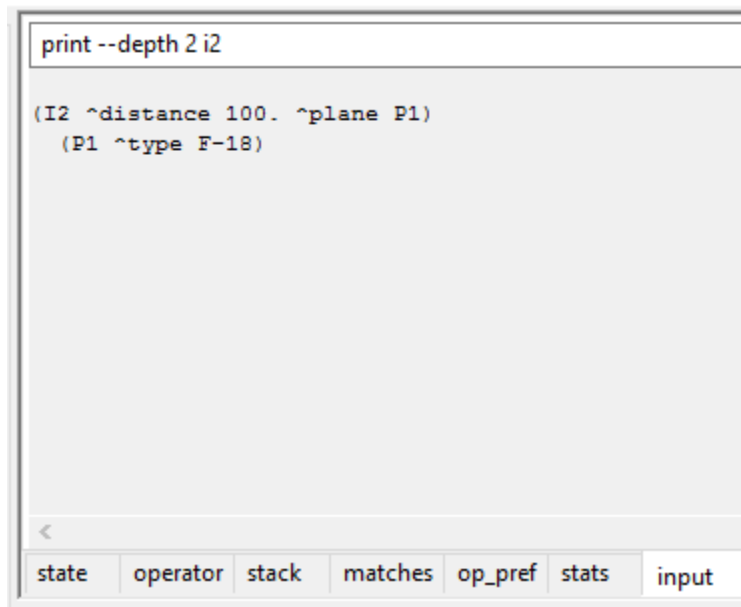
41

Figure 14. SML test, screenshot of Soar debugger

The second test was designed to check the communication between Soar and the SML client while utilizing a Soar agent's decision making capability. In this test the agent needed to decide if it wanted to move or stop. The SML client sent the information about the world to the Soar agent. The world consisted of a long hallway with a wall at the end. This was represented in the code as an array. The array was populated with the string "empty" representing empty cells and "wall" to represent the wall at the end of the hall. The Soar agent receives from the SML client what is in front of it, and it decides if it should move forwards or stop. While the agent makes the decisions we can use the Soar debugger to verify that the Soar kernel was receiving input and producing output decisions. The test also used a command prompt to display debugging output from the C++ SML client. The test was successful and the agent was able to move to the end of the hall and stop when it encountered a wall in front. Figure 15 shows part of the Soar

42

agent code utilized in the test. Figure 16 is a screenshot of the input link and figure 17

shows the output link, which is the communication from Soar to the client.

```
# Rule fires if there is an empty in front
sp {if-empty*propose*forward
    (state <s> ^name mover
            ^io.input-link.front empty)
-->
    (<s> ^operator <op> + )
    (<op> ^name forward)}

sp {apply*forward
    (state <s> ^operator <op>
                ^io.output-link <out>)
    (<op> ^name forward)
-->
    (<out> ^forward <f>)}


# Rule fires if there is a wall in front
sp {if-wall*propose*stop
    (state <s> ^name mover
            ^io.input-link.front wall)
-->
    (<s> ^operator <op> + = )
    (<op> ^name stop)}

sp {apply*stop
    (state <s> ^operator <op>
                ^io.output-link <out>)
    (<op> ^name stop)
-->
    (<out> ^stop <f>)}
```

Figure 15. Fragment Soar agent code for hallway test
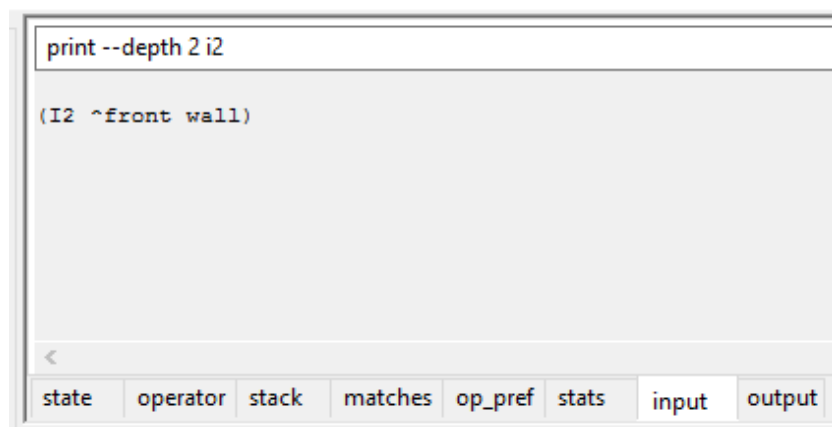
Figure 16. Soar debugger screenshot, simple environment test.

43

```
source {agent.soar}***********
Total: 11 productions sourced.
     6: O: O6 (forward)
     7: O: O7 (forward)
     8: O: O8 (forward)
     9: O: O9 (forward)
    10: O: O10 (forward)
    11: O: O11 (forward)
    12: O: O12 (forward)
    13: O: O13 (forward)
    14: O: O14 (forward)
    15: O: O15 (forward)
    16: O: O16 (forward)
    17: O: O17 (forward)
    18: O: O18 (forward)
    19: O: O19 (forward)
    20: O: O20 (forward)
    21: O: O21 (stop)
    22: O: O22 (stop)
    23: O: O23 (stop)
|
```

Figure 17. Soar agent output.

Once the major elements of the SML API were tested the next step was to investigate other ways in which the SML client could manage the input and output link. The SML API has the capability of utilizing an event model framework in which the client can register for Soar events. There are many events you can register for and with each version of Soar that list grows [17]. To test the SML client ability to register and handle Soar events a test was conducted. The concept of the world as a long hallway was utilized here. The C++ client code was modified by registering for the Soar run event and the event handlers were created. Figure 15 shows part of the C++ code used to register for events. A benefit of utilizing events was the ability to also control the simulation with the Soar debugger. Figure 19 shows the output from the agent as seen on the Soar debugger. Once these initial tests were complete it was time to study the sandbox environment and work on extending that application.

44

```
SoarRegisterEvent* worldData = new SoarRegisterEvent();


pAgent->RegisterForRunEvent(smlEVENT_BEFORE_INPUT_PHASE,
    InputPhaseHandler, worldData, NULL);
pAgent->AddOutputHandler("forward", ForwardEventHandler, worldData, NULL);
pAgent->AddOutputHandler("stop", StopEventHandler, worldData, NULL);
Identifier* pInputLink = pAgent->GetInputLink();
```

Figure 18. C++ SML client code to register for Soar event

Figure 19. Agent output screenshot from Soar debugger.

Code Changes to Demo Framework

Chapter III introduced a list of code files that needed to be changed in the demo
framework application of the sandbox. The new implementation of the Agent class is
called BaseAgent and it has a child class named LuaAgent. The LuaAgent class works
with the Lua scripts and the sandbox. The SoarAgent class is a child of LuaAgent and
inherits all the functionality that helps with the communication of the Lua scripts and the

45

sandbox. The SoarAgent class serves as the parent class for Soar agent classes. Figure 16 shows the diagram of the new implementation of the agent classes. Users that want to create Soar agents will need to inherit from SoarAgent class. Users can also have Lua agents and Soar agents in the sandbox at the same time. This design uses the Lua scripts to set up the sandbox and handle the physics and animation of the agents. The Soar agent code will be the decision maker part of the agent. Each Soar agent has a Lua script associated with it.
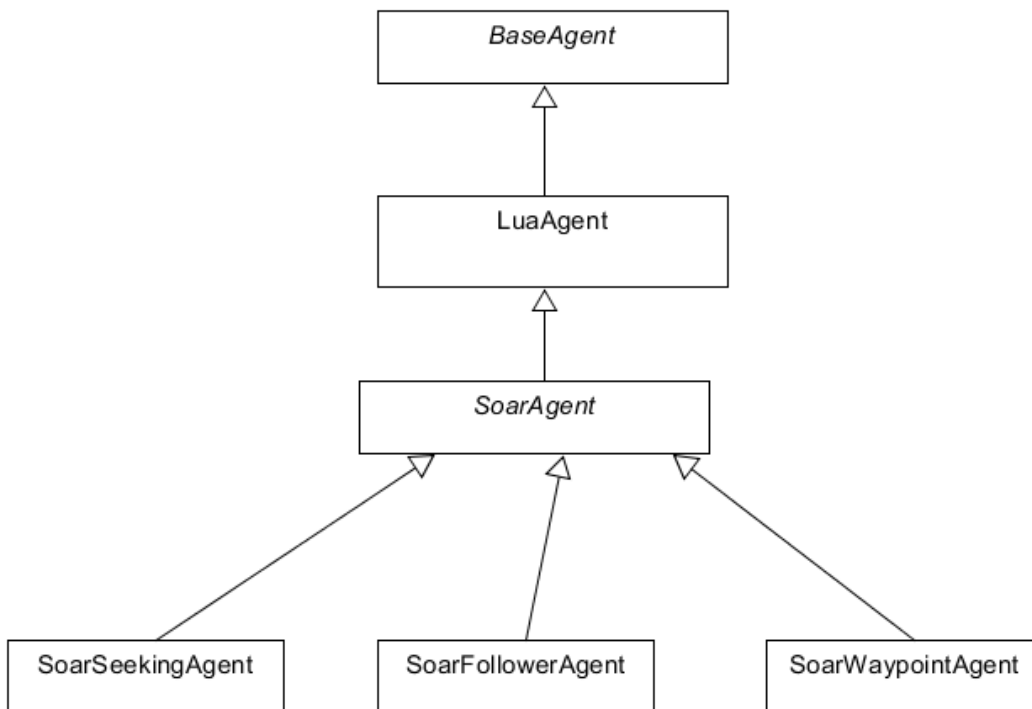
Figure 20. Diagram of Agent classes

46

Soar Agent Tests

After the completion of the code changes necessary to be able to use the new agent classes, the next step was to create agents in the sandbox controlled by Soar. First, we create the sandbox project for the agents to exist on. The project my_sandbox was created for this purpose. This sandbox project includes the MySandbox class and the Lua scripts necessary to set up the sandbox. These Lua scripts generate the floor, light the sky, and initialize the agents in the sandbox. The Soar agents created for the tests are based on Lua agents available in the sandbox project code [29]. Creating Soar agents that mimic existing agents was a way to test the new functionality without having to design agents from the ground up.

Seeking Agent

The Soar seeking agent moves to a target point in the environment and once it reaches that point it randomly selects another point in the environment to set as its target. To create a Soar seeking agent a SoarSeekingAgent class was created. This includes a header file and a source file. This new class is a child of SoarAgent class and becomes part of the demo framework project. The SoarSeekingAgent class creates an instance of the Soar kernel; it also sends the environment information to the Soar agent and receives the output from Soar. In the case of the seeking agent the code utilizes the current agent's position and the position of the agent's target to calculate a distance. This distance is then sent to the Soar kernel for the decision making process. Since the Lua scripts are responsible for the initialization of the sandbox, code needed to be added to the LuaScriptBindings and LuaScriptUtilities classes. This code makes it possible for the

47

Lua script to create a SoarSeekingAgent instance. After the C++ code is complete the agent needs a Soar file with the decision making code which runs on the Soar kernel, and a Lua script that handles the agent representation and movement. The Soar agent code will receive the distance parameter calculated by the SoarSekingAgent code and decide whether to continue heading towards the target point or choose a new target point. Having all the components ready the test was ran. The Soar Debugger was used to view the input and output on the Soar kernel side. Figure 17 shows a screenshot of the output from the Soar agent from the debugger, line 1711 shows the output "newtarget" which was used by the SoarSeekingAgent code to initiate the new target process. Figure 18 is a screenshot of the sandbox with the agent and the target point.

```
1704: O: O1704 (sametarget)
1705: O: O1705 (sametarget)
1706: O: O1706 (sametarget)
1707: O: O1707 (sametarget)
1708: O: O1708 (sametarget)
1709: O: O1709 (sametarget)
1710: O: O1710 (sametarget)
1711: O: O1711 (newtarget)
1712: O: O1712 (sametarget)
1713: O: O1713 (sametarget)
1714: O: O1714 (sametarget)
1715: O: O1715 (sametarget)
```

Figure 21. Screenshot of Soar Debugger (seeking agent)
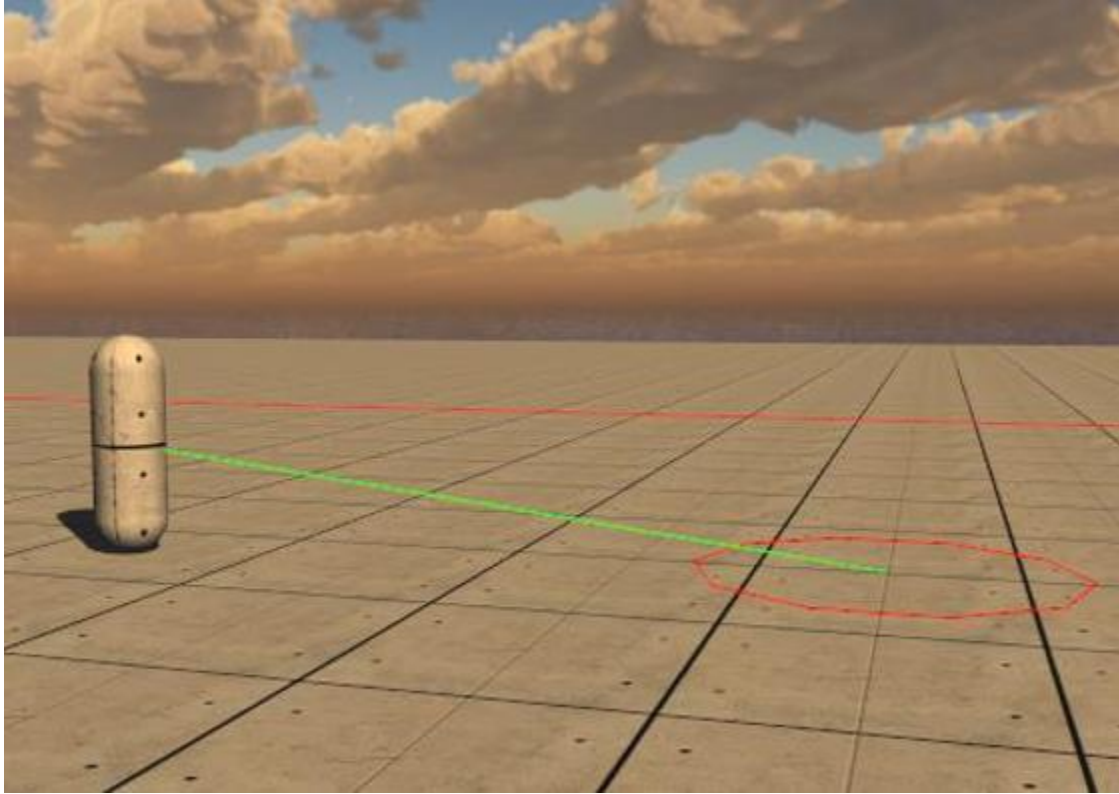
Figure 22. Screenshot of seeking agent

Follower Agent

The follower agent is similar to the seeking agent in because it needs to travel to a target point. In the case of the follower agent the target is another agent. This agent can be used to follow a leader or to pursue an enemy. The SoarFollowerAgent class was created as a child class of SoarAgent. This class adds a member named pLeader. The pLeader is a pointer to a BaseAgent element. Since Soar agents and Lua agents inherit from BaseAgent, this gives us the capability to have any type of agent as the leader. The utility classes and binding classes used by Lua scripts to call C++ functions were changed to add the capability of Lua to initialize follower agents. After updating the utility clases

49

the MySandbox class has to be updated to initialize the different types of agents. For this test a Soar seeking agent was utilized as the leader.

Each frame rendered on the sandbox environment calls the update method of the agents. The update code in SoarFollowerAgent gets the current position of the leader from the pointer and compares it with the agent's position. This distance between the follower and the leader is then sent to the Soar kernel and it is used by the Soar agent to decide to stop following or to continue to follow. Figure 19 shows the output from the Soar agent as seen from the Soar Debugger. Once the follower agent gets to a predefined distance from the leader, the follower agent stops and waits until that distance increases to continue following. Figure 20 shows a Soar seeking agent with one Soar follower agent. By creating more than one follower agent we can have a group of follower agents. When performing this test we can see how the group steering behaves by keeping the agents within the group a minimum distance from each other. Figure 21 shows a Soar seeking agents and a group of Soar follower agents.

```
432: O: O432 (follow)
433: O: O433 (follow)
434: O: O434 (follow)
435: O: O435 (stop)
436: O: O436 (stop)
437: O: O437 (stop)
438: O: O438 (stop)
439: O: O439 (stop)
440: O: O440 (stop)
441: O: O441 (stop)
442: O: O442 (stop)
443: O: O443 (stop)
444: O: O444 (stop)
445: O: O445 (stop)
446: O: O446 (stop)
447: O: O447 (follow)
448: O: O448 (follow)
449: O: O449 (follow)
450: O: O450 (follow)
```

Figure 23. Screenshot from Soar Debugger (Follower Agent)

50
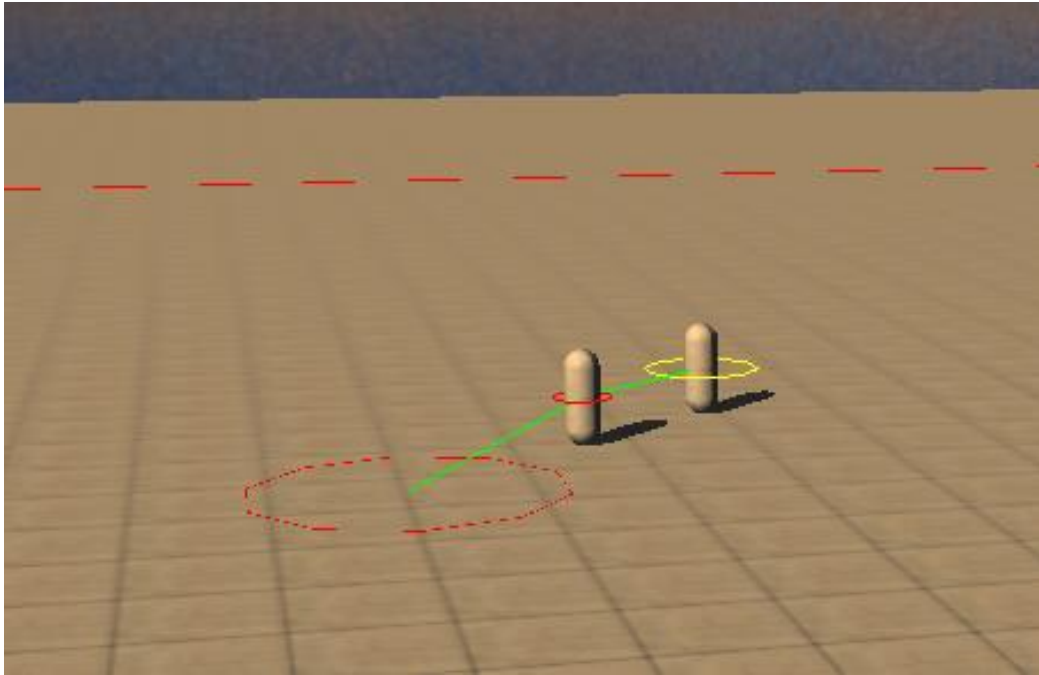
Figure 24. Leader agent with follower agent
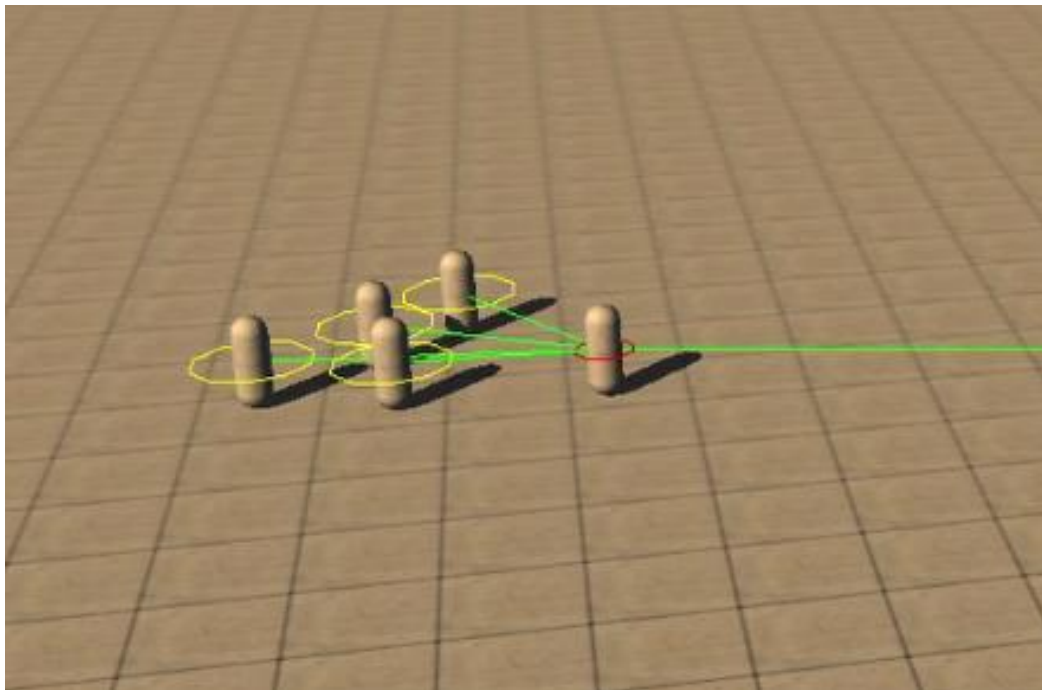
Figure 25. Leader agent with group of followers

51

Waypoint Agent

The purpose of the waypoint agent is to go through a list of predetermined points. Waypoints are used to aid in navigation. This test replicates a multi-waypoint mission where an agent needs to travel a certain path. Similar to the previous Soar agents a new class had to be created, named SoarWaypointAgent. The Lua utility classes were updated to be able to initialize these types of agents from the Lua script. The Lua script in the my_sandbox project was updated so that a Soar waypoint agent is initialized. The Lua script agent file that acts as the helper for the Soar agent will have the list of points. This gives us the ability to change the amount of waypoints and modify the list without having to compile the project. The SoarWaypointAgent gets the position of the current target and the position of the agent and calculates the distance between them. This distance is sent to the Soar agent's input link as a working memory element. The Soar agent goes through its decision cycle and outputs "samewaypoint" to keep it current heading or "nextwaypoint" if the distance from the agent and its target reaches a certain value. The SoarWaypointAgent code will take the output from the Soar agent and communicate it to the Lua script. The Lua script uses this to update the agent's target to the new waypoint if needed. This agent was tested by itself in the sandbox and later it was tested by initializing the 3 different types of Soar agents that had been developed so far. The screen shot of the test with all agents can be seen in Figure 22.
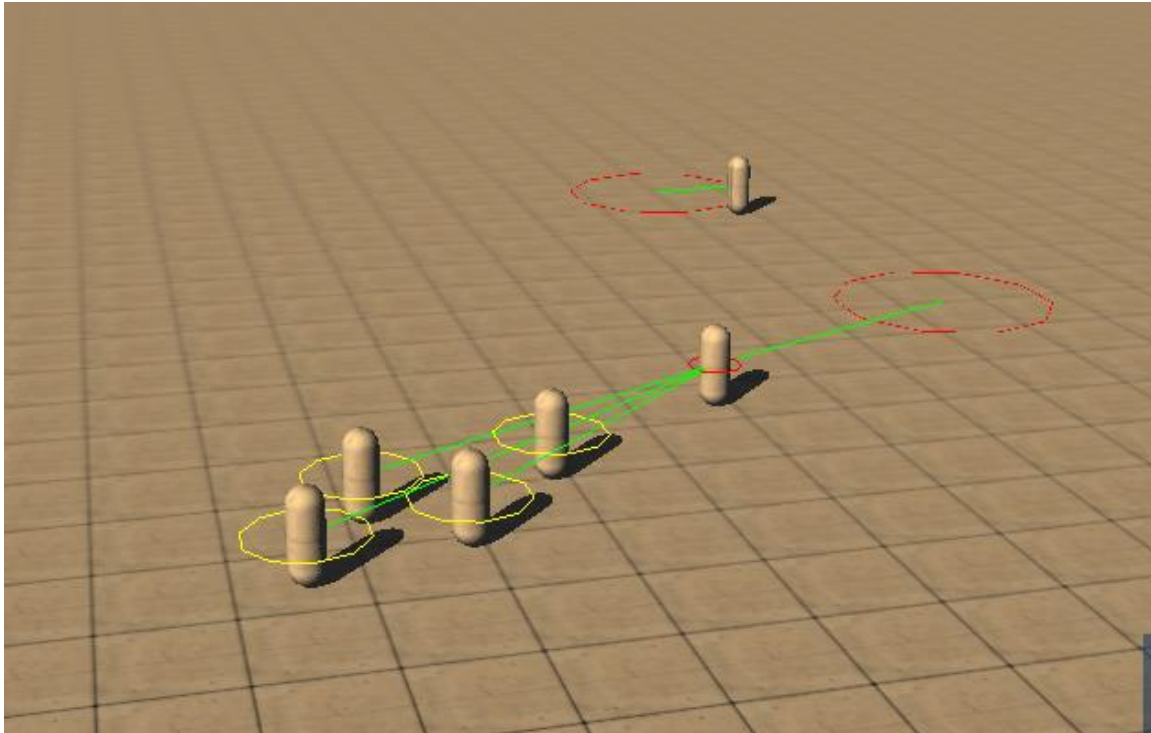
Figure 26. Soar agents in the sandbox, Waypoint Agent (top), Follower Agents (group of 4) and Seeking Agent (center right)

Soar and Lua agents

Two tests were used to verify the ability to have Lua agents and Soar agents working in the same sandbox environment. The first test added a Lua agent as a follower agent in a group. This group consisted of the Lua agent and three Soar follower agents. They were all following a Soar seeking agent. To create this test the sandbox Lua script was modified to initialize the Soar seeking agent and then to create the different types of follower agents. To be able to identify the Lua agent in the group the AgentUtilities Lua script was modified so that the Lua agent was visually recognizable. Figure 23 shows the result of the test. The Lua agent behaved as expected as part of the follower group. The second test added a Lua agent as the leader of the group. A script named Seeking-

53

LuaAgent was created based on the agent in the Lua book [29]. Figure 24 shows a screenshot of the test where the Lua agent is the leader.
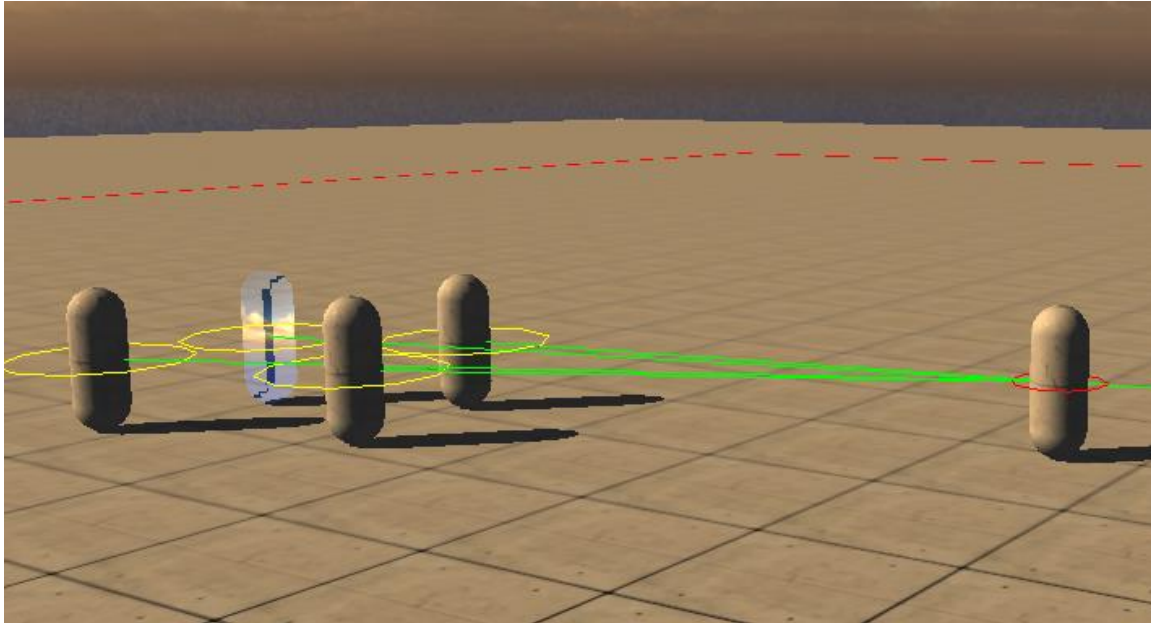
Figure 27. Screenshot Lua follower agent
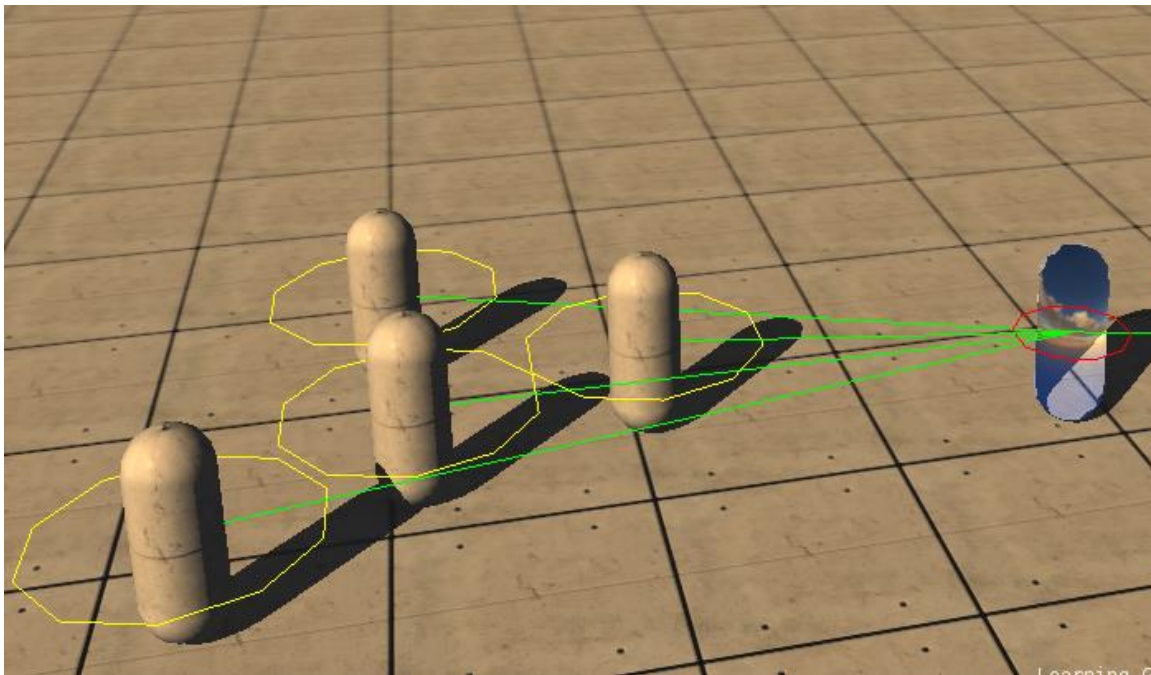
Figure 28. Screenshot Lua agent as leader

54

Agent Test Result Summary

Testing was performed throughout the development process. The first changes in the code were not to create Soar agents but to give the user the capability to do so by extending the software. After these changes, we performed regression testing by running the agents provided in the book Learning Game AI Programing with Lua [29]. By running these agents we verified that the code that was previously working still performed correctly. After regression testing, the Soar agent classes were created. These include the SoarSeekingAgent, SoarFollowerAgent, and the SoarWaypoint agent classes. All the Soar agents were tested visually and functionally with the Soar Debugger. The Soar agents were able to communicate with the sandbox successfully. The decision making capability of Soar combined with the Lua scripts was capable of controling the agents in the sandbox.

Summary

This chapter discussed the tests performed with Soar SML in the development environment. It presented the changes in the agent clases implemented to utilize Soar to control agents. This chapter also showed the different Soar agents that were created and some screenshots of the agents in the sandbox. Finally we have a brief discussion of the agent tests.

55

V. Conclusions and Recommendations

Chapter Overview

This chapter presents the conclusions and significance of the research performed. Additionally, recommendations for future work in extending the sandbox environment and some lessons learned about Soar are provided.

Conclusions of Research

This research successfully extended the sandbox, an off-the-shelf open-source software product, and developed a learning environment for creating Soar agents. The results of the tests performed with the Soar agents show that it is possible to extend the sandbox environment and give it the capability of using Soar and Lua to control intelligent agents. The Soar cognitive architecture requires effort to learn and time to practice in order to become proficient. If your goal is to create simple intelligent agents or you have a limited time to develop your system, Soar may not be the right tool to use. However, if your system needs to integrate learning with problem solving, have parallel reasoning, or capable of complex intelligent agents Soar could be an appropriate tool.

Significance of Research

By extending the sandbox environment and enabling it to use the Soar cognitive architecture to create intelligent agents, the Soar community and interested parties now have a customizable environment that is rich in features to learn and practice the Soar agent development process. This sandbox being similar to military simulations gives users chances to not only learn to create Soar agents, but also determine if Soar is the

56

right choice for their modelling needs. This reaserch also presents a blueprint on how to create Soar agents using the sandbox environment so that future research and more advanced agents can be developed.

Recommendations for Action

The next step in the process of extending the environment would be to update the visualization of the current Soar agents. This will give the ability of the Soar agents to have the appearance of human beings. Additionally, the sandbox environment has features that were not able to be used by the agents that were tested. These features, like having agents in different teams and communicate with each other, would be beneficial in creating advanced agents that perform simulated military engagements. It is recommended to utilize the sandbox environment to build Soar agents that could later be used as intelligent entities in military simulations.

Summary

This chapter presents the successful outcome of extending the sandbox environment to use the Soar cognitive architecture. It also talks about the significance of the research and how the environment bridges the gap between what was available for learning Soar and military simulations. It ends with the recommendations of utilizing this sandbox to bould advanced Soar agents that could be used in future simulations.

Project: my_sandbox
MySandbox Class

```
/**
* Author: Daniel Lugo
*
* Software written as part of Master's Thesis
*/

#include "my_sandbox/include/MySandbox.h"
#include <iostream>
#include "sml_Client.h"

MySandbox::MySandbox()
        : SandboxApplication("My Sandbox") {}
MySandbox::~MySandbox() {

}
void MySandbox::Cleanup()
{
        SandboxApplication::Cleanup();
}

void MySandbox::Draw()
{
        SandboxApplication::Draw();
}

void MySandbox::Update()
{
        SandboxApplication::Update();
}
void MySandbox::Initialize() {
        SandboxApplication::Initialize();

        AddResourceLocation("../../../src/my_sandbox/script");

        CreateSandbox("Sandbox.lua");

}
```

Project: demo_framework
BaseAgent Class
```
/**
*
* Author: Daniel Lugo
*
* Based on Agent class written by David Young
*
* Copyright (c) 2013 David Young dayoung@goliathdesigns.com
*
* This software is provided 'as-is', without any express or implied
* warranty. In no event will the authors be held liable for any damages
* arising from the use of this software.
*
* Permission is granted to anyone to use this software for any purpose,
* including commercial applications, and to alter it and redistribute it
* freely, subject to the following restrictions:
*
*  1. The origin of this software must not be misrepresented; you must not
*  claim that you wrote the original software. If you use this software
*  in a product, an acknowledgment in the product documentation would be
*  appreciated but is not required.
*
*  2. Altered source versions must be plainly marked as such, and must not be
*  misrepresented as being the original software.
*
*  3. This notice may not be removed or altered from any source
*  distribution.
*/

#include "PrecompiledHeaders.h"

#include "demo_framework/include/BaseAgent.h"
#include "demo_framework/include/AgentGroup.h"
#include "demo_framework/include/AgentUtilities.h"
#include "demo_framework/include/AnimationUtilities.h"
#include "demo_framework/include/LuaScriptUtilities.h"
#include "demo_framework/include/PhysicsUtilities.h"
#include "demo_framework/include/SandboxObject.h"
#include "demo_framework/include/SandboxUtilities.h"

namespace
{
```

59

```cpp
        inline Ogre::Quaternion BtQuaterionToQuaternion(const btQuaternion&
quaternion)
        {
                return Ogre::Quaternion(
                        quaternion.w(), quaternion.x(), quaternion.y(), quaternion.z());
        }

        inline btQuaternion QuaternionToBtQuaternion(
                const Ogre::Quaternion& quaternion)
        {
                return btQuaternion(quaternion.x, quaternion.y, quaternion.z,
quaternion.w);
        }

        inline OpenSteer::Vec3 Vector3ToVec3(const Ogre::Vector3& vector)
        {
                return OpenSteer::Vec3(
                        static_cast<float>(vector.x),
                        static_cast<float>(vector.y),
                        static_cast<float>(vector.z));
        }

        inline Ogre::Vector3 Vec3ToVector3(const OpenSteer::Vec3& vector)
        {
                return Ogre::Vector3(
                        Ogre::Real(vector.x), Ogre::Real(vector.y), Ogre::Real(vector.z));
        }

        inline btVector3 Vector3ToBtVector3(const Ogre::Vector3& vector)
        {
                return btVector3(
                        btScalar(vector.x), btScalar(vector.y), btScalar(vector.z));
        }
} // anonymous namespace

// Some default humanistic values.
const float BaseAgent::DEFAULT_AGENT_HEALTH = 100.0f;
const float BaseAgent::DEFAULT_AGENT_HEIGHT = 1.6f;  // meters (5.2 feet)
const float BaseAgent::DEFAULT_AGENT_MASS = 90.7f;  // kilograms (200 lbs)
const float BaseAgent::DEFAULT_AGENT_MAX_FORCE = 1000.0f;  // newtons
(kg*m/s^2)
const float BaseAgent::DEFAULT_AGENT_MAX_SPEED = 7.0f;  // m/s (23.0 ft/s)
const float BaseAgent::DEFAULT_AGENT_RADIUS = 0.3f;  // meters (1.97 feet)
const float BaseAgent::DEFAULT_AGENT_SPEED = 0.0f;  // m/s (0 ft/s)
```

60

```cpp
const float BaseAgent::DEFAULT_AGENT_TARGET_RADIUS = 0.5f;  // meters (1.64
feet)
const float BaseAgent::DEFAULT_AGENT_WALKABLE_CLIMB =
DEFAULT_AGENT_RADIUS / 2.0f;
const float BaseAgent::DEFAULT_AGENT_WALKABLE_SLOPE = 45.0f;
const Ogre::String BaseAgent::DEFAULT_AGENT_TEAM = "team1";

BaseAgent::BaseAgent(
        const unsigned int agentId,
        Ogre::SceneNode* const sceneNode,
        btRigidBody* const rigidBody)
        : Object(agentId, Object::AGENT),
        sandbox_(NULL),
        health_(DEFAULT_AGENT_HEALTH),
        height_(DEFAULT_AGENT_HEIGHT),
        hasPath_(false),
        sceneNode_(sceneNode),
        maxForce_(DEFAULT_AGENT_MAX_FORCE),
        maxSpeed_(DEFAULT_AGENT_MAX_SPEED),
        mass_(DEFAULT_AGENT_MASS),
        radius_(DEFAULT_AGENT_RADIUS),
        speed_(DEFAULT_AGENT_SPEED),
        rigidBody_(rigidBody),
        targetRadius_(DEFAULT_AGENT_TARGET_RADIUS),
        team_(DEFAULT_AGENT_TEAM),
        target_(Ogre::Vector3::ZERO)
{
        assert(sceneNode_);
        // SceneNode shouldn't have attached entities already.
        assert(!sceneNode_->numAttachedObjects());
        // SceneNode shouldn't have attached children.
        assert(!sceneNode_->numChildren());




        if (rigidBody_)
        {
                PhysicsUtilities::SetRigidBodyMass(rigidBody_,
DEFAULT_AGENT_MASS);
                rigidBody->setUserPointer(this);
        }
```

61

```cpp
        SetForward(Ogre::Vector3::UNIT_Z);
}

BaseAgent::~BaseAgent()
{
        sandbox_ = NULL;
        sceneNode_ = NULL;

        if (rigidBody_)
        {
                PhysicsUtilities::DeleteRigidBody(rigidBody_);
        }

        rigidBody_ = NULL;


}


void BaseAgent::ClearPath()
{
        hasPath_ = false;
}

Ogre::Vector3 BaseAgent::ForceToAlign(
        const Ogre::Real maxDistance,
        const Ogre::Degree maxAngle,
        const AgentGroup& group)
{
        const float maxCosAngle = cosf(maxAngle.valueRadians());
        return Vec3ToVector3(steerForAlignment(maxDistance, maxCosAngle, group));
}

Ogre::Vector3 BaseAgent::ForceToAvoidAgents(
        const std::vector<BaseAgent*>& agents, const float predictionTime)
{
        const static float MIN_PREDICTION_TIME = 0.1f;

        OpenSteer::AVGroup group;
        std::vector<BaseAgent*>::const_iterator it;

        // TODO(David Young): Avoid copying into the AVGroup.
```

62

```
        for (it = agents.begin(); it != agents.end(); ++it)
        {
                BaseAgent* const agent = *it;
                group.push_back(agent);
        }

        return Vec3ToVector3(steerToAvoidNeighbors(
                std::max(MIN_PREDICTION_TIME, predictionTime), group));
}

Ogre::Vector3 BaseAgent::ForceToAvoidObjects(
        const std::map<unsigned int, SandboxObject*>& objects,
        const float predictionTime)
{
        const static float MIN_PREDICTION_TIME = 0.1f;
        const float timeToCollision = std::max(MIN_PREDICTION_TIME,
predictionTime);

        OpenSteer::ObstacleGroup group;
        std::map<unsigned int, SandboxObject*>::const_iterator it;

        OpenSteer::Vec3 avoidForce = OpenSteer::Vec3::zero;

        // TODO(David Young): Avoid copying into the ObstacleGroup.
        for (it = objects.begin(); it != objects.end(); ++it)
        {
                SandboxObject* const object = it->second;

                // Only avoid objects that aren't fixed.
                if (object->GetMass() > 0)
                {
                        avoidForce += object->steerToAvoid(*this, timeToCollision);
                }
        }

        return Vec3ToVector3(avoidForce);
}

Ogre::Vector3 BaseAgent::ForceToCombine(
        const Ogre::Real maxDistance,
        const Ogre::Degree maxAngle,
        const AgentGroup& group)
{
        const float maxCosAngle = cosf(maxAngle.valueRadians());
```

63

```cpp
        return Vec3ToVector3(steerForCohesion(maxDistance, maxCosAngle, group));
}

Ogre::Vector3 BaseAgent::ForceToFleePosition(const Ogre::Vector3& position)
{
        return Vec3ToVector3(steerForFlee(Vector3ToVec3(position)));
}

Ogre::Vector3 BaseAgent::ForceToFollowPath(const float predicitionTime)
{
        return ForceToFollowPath(path_, predicitionTime);
}

Ogre::Vector3 BaseAgent::ForceToFollowPath(
        AgentPath& path, const float predictionTime)
{
        const static int FORWARD_DIRECTION = 1;
        const static float MIN_PREDICTION_TIME = 0.1f;

        if (path.GetNumberOfPathPoints())
        {
                return Vec3ToVector3(steerToFollowPath(
                        FORWARD_DIRECTION,
                        std::max(MIN_PREDICTION_TIME, predictionTime),
                        path));
        }
        return Ogre::Vector3(0.0f);
}

Ogre::Vector3 BaseAgent::ForceToPosition(const Ogre::Vector3& position)
{
        return Vec3ToVector3(steerForSeek(Vector3ToVec3(position)));
}

Ogre::Vector3 BaseAgent::ForceToStayOnPath(const float predictionTime)
{
        return ForceToStayOnPath(path_, predictionTime);
}

Ogre::Vector3 BaseAgent::ForceToSeparate(
        const Ogre::Real maxDistance,
        const Ogre::Degree maxAngle,
        const AgentGroup& group)
{
```

64

```cpp
        const float maxCosAngle = cosf(maxAngle.valueRadians());
        return Vec3ToVector3(steerForSeparation(maxDistance, maxCosAngle, group));
}

Ogre::Vector3 BaseAgent::ForceToStayOnPath(
        AgentPath& path, const float predictionTime)
{
        const static float MIN_PREDICTION_TIME = 0.1f;

        if (path.GetNumberOfPathPoints())
        {
                return Vec3ToVector3(steerToStayOnPath(
                        std::max(MIN_PREDICTION_TIME, predictionTime), path));
        }

        return Ogre::Vector3(0.0f);
}

Ogre::Vector3 BaseAgent::ForceToTargetSpeed(const Ogre::Real speed)
{
        return Vec3ToVector3(steerForTargetSpeed(float(speed)));
}

Ogre::Vector3 BaseAgent::ForceToWander(const Ogre::Real deltaMilliseconds)
{
        return Vec3ToVector3(steerForWander(deltaMilliseconds));
}

OpenSteer::Vec3 BaseAgent::forward() const
{
        return Vector3ToVec3(GetForward());
}

Ogre::Vector3 BaseAgent::GetForward() const
{
        if (rigidBody_)
        {
                const btQuaternion quaterion = rigidBody_->getOrientation();
                return Ogre::Quaternion(
                        quaterion.w(), quaterion.x(), quaterion.y(), quaterion.z()).zAxis();
        }
        else if (sceneNode_)
        {
                return sceneNode_->getOrientation().zAxis();
```

65

```cpp
        }

        return Ogre::Vector3::UNIT_Z;
}

Ogre::Real BaseAgent::GetHealth() const
{
        return health_;
}

Ogre:Real BaseAgent::GetHeight() const
{
        return height_;
}

Ogre::Vector3 BaseAgent::GetLeft() const
{
        if (rigidBody_)
        {
                const btQuaternion quaterion = rigidBody_->getOrientation();
                return Ogre::Quaternion(
                        quaterion.w(), quaterion.x(), quaterion.y(), quaterion.z()).xAxis();
        }
        else if (sceneNode_)
        {
                return sceneNode_->getOrientation().xAxis();
        }

        return Ogre::Vector3::UNIT_X;
}


Ogre::Real BaseAgent::GetMass() const
{
        if (rigidBody_)
        {
                return Ogre::Real(PhysicsUtilities::GetRigidBodyMass(rigidBody_));
        }
        else
        {
                return mass_;
        }
}
```

66

```cpp
Ogre::Real BaseAgent::GetMaxForce() const
{
        return maxForce_;
}

Ogre::Real BaseAgent::GetMaxSpeed() const
{
        return maxSpeed_;
}

Ogre::Quaternion BaseAgent::GetOrientation() const
{
        if (rigidBody_)
        {
                return BtQuaterionToQuaternion(rigidBody_->getOrientation());
        }
        else if (sceneNode_)
        {
                return sceneNode_->_getDerivedOrientation();
        }

        return Ogre::Quaternion::ZERO;
}

AgentPath BaseAgent::GetPath()
{
        return path_;
}

const AgentPath& BaseAgent::GetPath() const
{
        return path_;
}

Ogre::Vector3 BaseAgent::GetPosition() const
{
        if (rigidBody_)
        {
                return PhysicsUtilities::BtVector3ToVector3(
                        rigidBody_->getCenterOfMassPosition());
        }
        else if (sceneNode_)
        {
```

```cpp
                return sceneNode_->_getDerivedPosition();
        }

        return Ogre::Vector3::ZERO;
}

Ogre::Real BaseAgent::GetRadius() const
{
        return radius_;
}

btRigidBody* BaseAgent::GetRigidBody()
{
        return rigidBody_;
}

Sandbox* BaseAgent::GetSandbox()
{
        return sandbox_;
}

Ogre::SceneNode* BaseAgent::GetSceneNode()
{
        return sceneNode_;
}

Ogre::Real BaseAgent::GetSpeed() const
{
        if (rigidBody_)
        {
                const btVector3 velocity = rigidBody_->getLinearVelocity();
                // Ignore movement along the y axis.
                // Gravity does not contribute to the speed of the Agent.
                return Ogre::Real(
                        Ogre::Vector3(velocity.x(), 0, velocity.z()).length());
        }
        else
        {
                return speed_;
        }
}

Ogre::Vector3 BaseAgent::GetTarget() const
{
```

68

```cpp
        return target_;
}

Ogre::Real BaseAgent::GetTargetRadius() const
{
        return targetRadius_;
}

const Ogre::String& BaseAgent::GetTeam() const
{
        return team_;
}

Ogre::Vector3 BaseAgent::GetUp() const
{
        if (rigidBody_)
        {
                const btQuaternion quaterion = rigidBody_->getOrientation();
                return Ogre::Quaternion(
                        quaterion.w(), quaterion.x(), quaterion.y(), quaterion.z()).yAxis();
        }
        else if (sceneNode_)
        {
                return sceneNode_->getOrientation().yAxis();
        }

        return Ogre::Vector3::UNIT_Y;
}

Ogre::Vector3 BaseAgent::GetVelocity() const
{
        if (rigidBody_)
        {
                const btVector3 velocity = rigidBody_->getLinearVelocity();

                return Ogre::Vector3(velocity.x(), velocity.y(), velocity.z());
        }
        return GetForward() * speed_;
}

OpenSteer::Vec3 BaseAgent::globalizeDirection(
        const OpenSteer::Vec3& localDirection) const
{
        (void)localDirection;
```

69

```cpp
        // not implemented
        assert(false);

        return OpenSteer::Vec3();
}
OpenSteer::Vec3 BaseAgent::globalizePosition(
        const OpenSteer::Vec3& localPosition) const
{
        (void)localPosition;
        // not implemented
        assert(false);

        return OpenSteer::Vec3();
}

OpenSteer::Vec3 BaseAgent::globalRotateForwardToSide(
        const OpenSteer::Vec3& globalForward) const
{
        (void)globalForward;
        // not implemented
        assert(false);

        return OpenSteer::Vec3();
}

bool BaseAgent::HasPath() const
{
        return hasPath_;
}

void BaseAgent::Initialize()
{
        //AgentUtilities::CreateRigidBodyCapsule(this);

        //AgentUtilities::CallLuaAgentInitialize(this);
}


OpenSteer::Vec3 BaseAgent::localizeDirection(
        const OpenSteer::Vec3& globalDirection) const
{
        // TODO(David Young): This is very slow, convert to native Ogre math.
        return OpenSteer::Vec3(
```

70

```cpp
                globalDirection.dot(Vector3ToVec3(GetLeft())),
                globalDirection.dot(Vector3ToVec3(GetUp())),
                globalDirection.dot(Vector3ToVec3(GetForward()))));
}

OpenSteer::Vec3 BaseAgent::localizePosition(
        const OpenSteer::Vec3& globalPosition) const
{
        OpenSteer::Vec3 globalOffset =
                globalPosition - Vector3ToVec3(GetPosition());

        return localizeDirection(globalOffset);
}

OpenSteer::Vec3 BaseAgent::localRotateForwardToSide(
        const OpenSteer::Vec3& side) const
{
        (void)side;
        // not implemented
        assert(false);

        return OpenSteer::Vec3();
}

float BaseAgent::mass() const
{
        return static_cast<float>(GetMass());
}

float BaseAgent::maxForce() const
{
        return static_cast<float>(GetMaxForce());
}

float BaseAgent::maxSpeed() const
{
        return static_cast<float>(GetMaxSpeed());
}

OpenSteer::Vec3 BaseAgent::position() const
{
        return Vector3ToVec3(GetPosition());
}
```

71

```cpp
OpenSteer::Vec3 BaseAgent::predictFuturePosition(
        const float predictionTime) const
{
        return Vector3ToVec3(PredictFuturePosition(Ogre::Real(predictionTime)));
}

Ogre::Vector3 BaseAgent::PredictFuturePosition(
        const Ogre::Real predictionTime) const
{
        return GetPosition() + GetVelocity() * std::max(Ogre::Real(0), predictionTime);
}

float BaseAgent::radius() const
{
        return static_cast<float>(GetRadius());
}

void BaseAgent::regenerateOrthonormalBasisUF(
        const OpenSteer::Vec3& newUnitForward)
{
        (void)newUnitForward;
        // not implemented
        assert(false);
}

void BaseAgent::regenerateOrthonormalBasis(const OpenSteer::Vec3& newForward)
{
        (void)newForward;
        // not implemented
        assert(false);
}

void BaseAgent::regenerateOrthonormalBasis(
        const OpenSteer::Vec3& newForward, const OpenSteer::Vec3& newUp)
{
        (void)newForward;
        (void)newUp;
        // not implemented
        assert(false);
}


void BaseAgent::RemovePath()
{
```

72

```cpp
        hasPath_ = false;
}

void BaseAgent::resetLocalSpace()
{
        // not implemented
        assert(false);
}

bool BaseAgent::rightHanded() const
{
        return true;
}

void BaseAgent::SetForward(const Ogre::Quaternion& orientation)
{
        if (rigidBody_)
        {
                PhysicsUtilities::SetRigidBodyOrientation(
                        rigidBody_, QuaternionToBtQuaternion(orientation));
        }

        if (sceneNode_)
        {
                sceneNode_->setOrientation(orientation);
        }
}

void BaseAgent::SetForward(const Ogre::Vector3& forward)
{
        Ogre::Vector3 up = Ogre::Vector3::UNIT_Y;

        const Ogre::Vector3 zAxis = forward.normalisedCopy();
        const Ogre::Vector3 xAxis = up.crossProduct(zAxis);
        const Ogre::Vector3 yAxis = zAxis.crossProduct(xAxis);

        Ogre::Quaternion orientation(xAxis, yAxis, zAxis);

        // Update both the rigid body and scene node.
        if (rigidBody_)
        {
                PhysicsUtilities::SetRigidBodyOrientation(
                        rigidBody_, QuaternionToBtQuaternion(orientation));
        }
```

73

```cpp
        if (sceneNode_)
        {
                sceneNode_->setOrientation(orientation);
        }
}

OpenSteer::Vec3 BaseAgent::setForward(OpenSteer::Vec3 forward)
{
        (void)forward;
        // not implemented
        assert(false);

        return OpenSteer::Vec3();
}

void BaseAgent::SetHealth(const Ogre::Real health)
{
        health_ = health;
}

void BaseAgent::SetHeight(const Ogre::Real height)
{
        height_ = std::max(Ogre::Real(0), height);

        if (rigidBody_)
        {
                //AgentUtilities::UpdateRigidBodyCapsule(this);
        }
}

float BaseAgent::setMass(float mass)
{
        SetMass(Ogre::Real(mass));
        return this->mass();
}

void BaseAgent::SetMass(const Ogre::Real mass)
{
        if (rigidBody_)
        {
                PhysicsUtilities::SetRigidBodyMass(
                        rigidBody_, std::max(Ogre::Real(0), mass));
        }
```

74

```cpp
        mass_ = mass;
}

float BaseAgent::setMaxForce(float force)
{
        SetMaxForce(Ogre::Real(force));
        return maxForce();
}

void BaseAgent::SetMaxForce(const Ogre::Real force)
{
        maxForce_ = std::max(Ogre::Real(0), force);
}

float BaseAgent::setMaxSpeed(float speed)
{
        SetMaxSpeed(Ogre::Real(speed));
        return maxSpeed();
}

void BaseAgent::SetMaxSpeed(const Ogre::Real speed)
{
        maxSpeed_ = std::max(Ogre::Real(0), speed);
}

void BaseAgent::SetPath(const AgentPath& path)
{
        path_ = path;
        hasPath_ = true;
}

OpenSteer::Vec3 BaseAgent::setPosition(OpenSteer::Vec3 position)
{
        SetPosition(Vec3ToVector3(position));
        return this->position();
}

void BaseAgent::SetPosition(const Ogre::Vector3 position)
{
        if (rigidBody_)
        {
                PhysicsUtilities::SetRigidBodyPosition(
                        rigidBody_, Vector3ToBtVector3(position));
```

75

```
        }

        if (sceneNode_)
        {
                sceneNode_->setPosition(position);
        }
}

float BaseAgent::setRadius(float radius)
{
        SetRadius(Ogre::Real(radius));
        return this->radius();
}

void BaseAgent::SetRadius(const Ogre::Real radius)
{
        if (sceneNode_)
        {
                radius_ = std::max(Ogre::Real(0), radius);
        }

        if (rigidBody_)
        {
                //AgentUtilities::UpdateRigidBodyCapsule(this);
        }
}

void BaseAgent::SetRigidBody(btRigidBody* const rigidBody)
{
        rigidBody_ = rigidBody;

        if (rigidBody_)
        {
                rigidBody_->setUserPointer(this);
        }
}

void BaseAgent::SetSandbox(Sandbox* const sandbox)
{
        sandbox_ = sandbox;
}

OpenSteer::Vec3 BaseAgent::setSide(OpenSteer::Vec3 side)
{
```

76

```cpp
        (void)side;
        // not implemented
        assert(false);

        return OpenSteer::Vec3();
}

float BaseAgent::setSpeed(float speed)
{
        SetSpeed(Ogre::Real(speed));
        return this->speed();
}

void BaseAgent::SetSpeed(const Ogre::Real speed)
{
        // TODO(David Young): Need to update rigidbody's velocity based on input.
        speed_ = speed;
}

void BaseAgent::SetTarget(const Ogre::Vector3& target)
{
        target_ = target;
}

void BaseAgent::SetTargetRadius(const Ogre::Real radius)
{
        targetRadius_ = std::max(Ogre::Real(0), radius);
}

void BaseAgent::SetTeam(const Ogre::String& team)
{
        team_ = team;
}

void BaseAgent::setUnitSideFromForwardAndUp()
{
        // not implemented
        assert(false);
}

OpenSteer::Vec3 BaseAgent::setUp(OpenSteer::Vec3 up)
{
        (void)up;
        // not implemented
```

77

```cpp
        assert(false);

        return OpenSteer::Vec3();
}

void BaseAgent::SetVelocity(const Ogre::Vector3& velocity)
{
        if (rigidBody_)
        {
                PhysicsUtilities::SetRigidBodyVelocity(
                        rigidBody_, Vector3ToBtVector3(velocity));
        }

        SetSpeed(Ogre::Vector3(velocity.x, 0, velocity.z).length());
}

OpenSteer::Vec3 BaseAgent::side() const
{
        return Vector3ToVec3(GetLeft());
}

float BaseAgent::speed() const
{
        return static_cast<float>(GetSpeed());
}

OpenSteer::Vec3 BaseAgent::up() const
{
        return Vector3ToVec3(GetUp());
}

void BaseAgent::update(const float currentTime, const float elapsedTime)
{
        (void)currentTime;
        (void)elapsedTime;
        // not implemented
        assert(false);
}


OpenSteer::Vec3 BaseAgent::velocity() const
{
        return Vector3ToVec3(GetVelocity());
}
```

LuaAgent Class
```
/**
*
* Author: Daniel Lugo
*
* Based on code written by David Young
*
* Copyright (c) 2013 David Young dayoung@goliathdesigns.com
*
*/

#include "PrecompiledHeaders.h"

#include "demo_framework/include/LuaAgent.h"
#include "demo_framework/include/AgentUtilities.h"
#include "demo_framework/include/AnimationUtilities.h"
#include "demo_framework/include/LuaScriptUtilities.h"
#include "demo_framework/include/PhysicsUtilities.h"
#include "demo_framework/include/SandboxObject.h"
#include "demo_framework/include/SandboxUtilities.h"


LuaAgent::LuaAgent(
        const unsigned int agentId,
        Ogre::SceneNode* const sceneNode,
        btRigidBody* const rigidBody)
        :BaseAgent( agentId, sceneNode,rigidBody)

{

        luaVM_ = LuaScriptUtilities::CreateVM();

        // Add general C functions.
        LuaScriptUtilities::BindVMFunctions(luaVM_);

        // Add Agent specific functions.
        AgentUtilities::BindVMFunctions(luaVM_);

        // Add Sandbox specific functions.
        SandboxUtilities::BindVMFunctions(luaVM_);

        // Add Animation specific functions.
        AnimationUtilities::BindVMFunctions(luaVM_);
```

79

```cpp
}

LuaAgent::~LuaAgent() //how do I clear the memory of the base class

{


        LuaScriptUtilities::DestroyVM(luaVM_);
}

void LuaAgent::Cleanup()
{
        AgentUtilities::CallLuaAgentCleanup(this);
}


lua_State* LuaAgent::GetLuaVM()
{
        return luaVM_;
}


void LuaAgent::Initialize()
{
        AgentUtilities::CreateRigidBodyCapsule(this);

        AgentUtilities::CallLuaAgentInitialize(this);
}

void LuaAgent::LoadScript(
        const char* const luaScript,
        const size_t bufferSize,
        const char* const fileName)
{
        AgentUtilities::LoadScript(this, luaScript, bufferSize, fileName);
}
```

```cpp
bool LuaAgent::ReloadScript(
        const char* const luaScript,
        const size_t bufferSize,
        const char* const fileName)
{
        (void)luaScript;
        (void)bufferSize;
        (void)fileName;

        return false;
}




void LuaAgent::Update(const int deltaMilliseconds)
{
        AgentUtilities::CallLuaAgentUpdate(this, deltaMilliseconds);
        AgentUtilities::UpdateWorldTransform(this);

        if (this->GetRigidBody())
        {
                const btVector3 velocity = this->GetRigidBody()->getLinearVelocity();
                SetSpeed(Ogre::Vector3(velocity.x(), 0, velocity.z()).length());
        }
}

int LuaAgent::AgentType(){
        return 1;
}
```

SoarAgent Class

```
/**
*
* Author: Daniel Lugo
*
*
*/
#include "PrecompiledHeaders.h"

#include "demo_framework/include/SoarAgent.h"
#include "demo_framework/include/AgentUtilities.h"
#include "demo_framework/include/AnimationUtilities.h"
#include "demo_framework/include/LuaScriptUtilities.h"
#include "demo_framework/include/PhysicsUtilities.h"
#include "demo_framework/include/SandboxObject.h"
#include "demo_framework/include/SandboxUtilities.h"


SoarAgent::SoarAgent(
        const unsigned int agentId,
        Ogre::SceneNode* const sceneNode,
        btRigidBody* const rigidBody)
        :LuaAgent(agentId, sceneNode, rigidBody)

{

}



SoarAgent::~SoarAgent()

{

}


void SoarAgent::Cleanup()
{
        AgentUtilities::CallSoarAgentCleanup(this);
}
```

82

SoarSeekingAgent Class

```cpp
/**
*
* Author: Daniel Lugo
*
*
*/
#include "PrecompiledHeaders.h"

#include "demo_framework/include/SoarSeekingAgent.h"
#include "demo_framework/include/AgentUtilities.h"
#include "demo_framework/include/AnimationUtilities.h"
#include "demo_framework/include/LuaScriptUtilities.h"
#include "demo_framework/include/PhysicsUtilities.h"
#include "demo_framework/include/SandboxObject.h"
#include "demo_framework/include/SandboxUtilities.h"

#include <string>


SoarSeekingAgent::SoarSeekingAgent(
        const unsigned int agentId,
        Ogre::SceneNode* const sceneNode,
        btRigidBody* const rigidBody)
        :SoarAgent(agentId, sceneNode, rigidBody)

{

        luaVM_ = LuaScriptUtilities::CreateVM();

        // Add general C functions.
        LuaScriptUtilities::BindVMFunctions(luaVM_);

        // Add Agent specific functions.
        AgentUtilities::BindVMFunctions(luaVM_);

        // Add Sandbox specific functions.
        SandboxUtilities::BindVMFunctions(luaVM_);

        // Add Animation specific functions.
        AnimationUtilities::BindVMFunctions(luaVM_);

        pAgent = nullptr;
```

83

```cpp
        pKernel = nullptr;

        pWME1 = nullptr;

        pWME2 = nullptr;

        pWME3 = nullptr;
}

SoarSeekingAgent::~SoarSeekingAgent()

{


        LuaScriptUtilities::DestroyVM(luaVM_);
}



lua_State* SoarSeekingAgent::GetLuaVM()
{
        return luaVM_;
}


void SoarSeekingAgent::Initialize()
{

        this->SetTarget(Ogre::Vector3(0, 0, 10));


        AgentUtilities::CreateRigidBodyCapsule(this);

        AgentUtilities::CallLuaAgentInitialize(this);


        pKernel = sml::Kernel::CreateKernelInNewThread(12121);

        // Check that nothing went wrong.  We will always get back a kernel object
        // even if something went wrong and we have to abort.
        if (pKernel->HadError())
        {
                //Error handling
```

84

```
        }


        pAgent = pKernel->CreateAgent("agent");
        pAgent->LoadProductions("C:/Learning-Game-AI-Prog-w-Lua-
Code/1336OS_Code/src/my_sandbox/src/seekingAgent.soar");
        pAgent->SetBlinkIfNoChange(TRUE);

        if (pKernel->HadError())
        {
                //Error handling
        }

        pInputLink = pAgent->GetInputLink();

        //Here I am creating a WME on the input link, it is based on the eaters example



        pWME1 = pAgent->CreateFloatWME(pInputLink, "distance", 0);
}

void SoarSeekingAgent::LoadScript(
        const char* const luaScript,
        const size_t bufferSize,
        const char* const fileName)
{
        AgentUtilities::LoadScript(this, luaScript, bufferSize, fileName);
}




bool SoarSeekingAgent::ReloadScript(
        const char* const luaScript,
        const size_t bufferSize,
        const char* const fileName)
{
        (void)luaScript;
        (void)bufferSize;
        (void)fileName;
```

85

```cpp
        return false;
}



void SoarSeekingAgent::Update(const int deltaMilliseconds)
{
        Ogre::Real distance = this->GetPosition().squaredDistance(this->GetTarget());

        pAgent->Update(pWME1, distance);

        pAgent->RunSelfTilOutput();
        int numberCommands = pAgent->GetNumberCommands();

        //Ogre::Vector3 target;
        //target.x = 50; target.y = 0; target.z = 50;
        //this->SetTarget(target);



        for (int i = 0; i < numberCommands; i++)
        {
                sml::Identifier* pCommand = pAgent->GetCommand(i);

                std::string name = pCommand->GetCommandName();

                AgentUtilities::CallLuaAgentUpdate(this, deltaMilliseconds, name);

                AgentUtilities::UpdateWorldTransform(this);



                // Then mark the command as completed
                pCommand->AddStatusComplete();


        }


        if (this->GetRigidBody())
        {
                const btVector3 velocity = this->GetRigidBody()->getLinearVelocity();
                SetSpeed(Ogre::Vector3(velocity.x(), 0, velocity.z()).length());
```

86

```cpp
        }
}

int SoarSeekingAgent::AgentType(){
        return 2;
}
```

SoarFollwerAgent Class
```cpp
/**
*
* Author: Daniel Lugo
*
*
*/

#include "PrecompiledHeaders.h"

#include "demo_framework/include/SoarFollowerAgent.h"
#include "demo_framework/include/AgentUtilities.h"
#include "demo_framework/include/AnimationUtilities.h"
#include "demo_framework/include/LuaScriptUtilities.h"
#include "demo_framework/include/PhysicsUtilities.h"
#include "demo_framework/include/SandboxObject.h"
#include "demo_framework/include/SandboxUtilities.h"

#include <string>

SoarFollowerAgent::SoarFollowerAgent(
        const unsigned int agentId,
        Ogre::SceneNode* const sceneNode,
        btRigidBody* const rigidBody)
        :SoarAgent(agentId, sceneNode, rigidBody)

{

        luaVM_ = LuaScriptUtilities::CreateVM();

        // Add general C functions.
        LuaScriptUtilities::BindVMFunctions(luaVM_);

        // Add Agent specific functions.
        AgentUtilities::BindVMFunctions(luaVM_);

        // Add Sandbox specific functions.
```

87

```cpp
        SandboxUtilities::BindVMFunctions(luaVM_);

        // Add Animation specific functions.
        AnimationUtilities::BindVMFunctions(luaVM_);

        pAgent = nullptr;

        pLeader = nullptr;

        pKernel = nullptr;

        pWME1 = nullptr;

        pWME2 = nullptr;

        pWME3 = nullptr;
}

SoarFollowerAgent::~SoarFollowerAgent()

{


        LuaScriptUtilities::DestroyVM(luaVM_);
}




lua_State* SoarFollowerAgent::GetLuaVM()
{
        return luaVM_;
}


void SoarFollowerAgent::Initialize()
{

        this->SetTarget(Ogre::Vector3(0, 0, 10));


        AgentUtilities::CreateRigidBodyCapsule(this);
```

88

```cpp
        AgentUtilities::CallLuaAgentInitialize(this);


        pKernel = sml::Kernel::CreateKernelInNewThread(12122);

        // Check that nothing went wrong.  We will always get back a kernel object
        // even if something went wrong and we have to abort.
        if (pKernel->HadError())
        {
                //Error handling
        }


        pAgent = pKernel->CreateAgent("agent");
        pAgent->LoadProductions("C:/Learning-Game-AI-Prog-w-Lua-
Code/1336OS_Code/src/my_sandbox/src/followerAgent.soar");
        pAgent->SetBlinkIfNoChange(TRUE);

        if (pKernel->HadError())
        {
                //Error handling
        }

        pInputLink = pAgent->GetInputLink();

        //Here I am creating a WME on the input link, it is based on the eaters example



        pWME1 = pAgent->CreateFloatWME(pInputLink, "distance", 0);
}

void SoarFollowerAgent::LoadScript(
        const char* const luaScript,
        const size_t bufferSize,
        const char* const fileName)
{
        AgentUtilities::LoadScript(this, luaScript, bufferSize, fileName);
}

void SoarFollowerAgent::SetLeader(BaseAgent* leader){
        this->pLeader = leader;
}
```

89

```cpp
bool SoarFollowerAgent::ReloadScript(
        const char* const luaScript,
        const size_t bufferSize,
        const char* const fileName)
{
        (void)luaScript;
        (void)bufferSize;
        (void)fileName;

        return false;
}




void SoarFollowerAgent::Update(const int deltaMilliseconds)
{


        Ogre::Real distance = this->GetPosition().squaredDistance(this->pLeader-
>GetPosition());

        pAgent->Update(pWME1, distance);



        pAgent->RunSelfTilOutput();
        int numberCommands = pAgent->GetNumberCommands();




        for (int i = 0; i < numberCommands; i++)
        {
                sml::Identifier* pCommand = pAgent->GetCommand(i);
```

90

```cpp
                std::string name = pCommand->GetCommandName();

                AgentUtilities::CallLuaAgentUpdate(this, deltaMilliseconds, name);

                AgentUtilities::UpdateWorldTransform(this);

                // Then mark the command as completed
                pCommand->AddStatusComplete();


        }


        if (this->GetRigidBody())
        {
                const btVector3 velocity = this->GetRigidBody()->getLinearVelocity();
                SetSpeed(Ogre::Vector3(velocity.x(), 0, velocity.z()).length());
        }
}

int SoarFollowerAgent::AgentType(){
        return 3;
}
```

SoarWaypointAgent

```
/**
*
* Author: Daniel Lugo
*
*
*/
#include "PrecompiledHeaders.h"

#include "demo_framework/include/SoarWaypointAgent.h"
#include "demo_framework/include/AgentUtilities.h"
#include "demo_framework/include/AnimationUtilities.h"
#include "demo_framework/include/LuaScriptUtilities.h"
#include "demo_framework/include/PhysicsUtilities.h"
#include "demo_framework/include/SandboxObject.h"
#include "demo_framework/include/SandboxUtilities.h"

#include <string>



SoarWaypointAgent::SoarWaypointAgent(
        const unsigned int agentId,
        Ogre::SceneNode* const sceneNode,
        btRigidBody* const rigidBody)
        :SoarAgent(agentId, sceneNode, rigidBody)

{

        luaVM_ = LuaScriptUtilities::CreateVM();

        // Add general C functions.
        LuaScriptUtilities::BindVMFunctions(luaVM_);

        // Add Agent specific functions.
        AgentUtilities::BindVMFunctions(luaVM_);

        // Add Sandbox specific functions.
        SandboxUtilities::BindVMFunctions(luaVM_);

        // Add Animation specific functions.
        AnimationUtilities::BindVMFunctions(luaVM_);
```

92

```cpp
        pAgent = nullptr;

        pKernel = nullptr;

        pWME1 = nullptr;

        pWME2 = nullptr;

        pWME3 = nullptr;
}

SoarWaypointAgent::~SoarWaypointAgent()

{


        LuaScriptUtilities::DestroyVM(luaVM_);
}




lua_State* SoarWaypointAgent::GetLuaVM()
{
        return luaVM_;
}

void SoarWaypointAgent::Initialize()
{

        this->SetTarget(Ogre::Vector3(0, 0, 10));

        AgentUtilities::CreateRigidBodyCapsule(this);

        AgentUtilities::CallLuaAgentInitialize(this);

        pKernel = sml::Kernel::CreateKernelInNewThread(12123);

        // Check that nothing went wrong.  We will always get back a kernel object
```

93

```cpp
		// even if something went wrong and we have to abort.
		if (pKernel->HadError())
		{
			//Error handling
		}


		pAgent = pKernel->CreateAgent("agent");
		pAgent->LoadProductions("C:/Learning-Game-AI-Prog-w-Lua-
Code/1336OS_Code/src/my_sandbox/src/waypointAgent.soar");
		pAgent->SetBlinkIfNoChange(TRUE);

		if (pKernel->HadError())
		{
			//Error handling
		}

		pInputLink = pAgent->GetInputLink();

		//Here I am creating a WME on the input link, it is based on the eaters example


		pWME1 = pAgent->CreateFloatWME(pInputLink, "distance", 0);
}

void SoarWaypointAgent::LoadScript(
		const char* const luaScript,
		const size_t bufferSize,
		const char* const fileName)
{
		AgentUtilities::LoadScript(this, luaScript, bufferSize, fileName);
}




bool SoarWaypointAgent::ReloadScript(
		const char* const luaScript,
		const size_t bufferSize,
		const char* const fileName)
{
```

94

```cpp
        (void)luaScript;
        (void)bufferSize;
        (void)fileName;

        return false;
}




void SoarWaypointAgent::Update(const int deltaMilliseconds)
{
        Ogre::Real distance = this->GetPosition().squaredDistance(this->GetTarget());

        pAgent->Update(pWME1, distance);

        pAgent->RunSelfTilOutput();
        int numberCommands = pAgent->GetNumberCommands();




        for (int i = 0; i < numberCommands; i++)
        {
                sml::Identifier* pCommand = pAgent->GetCommand(i);

                std::string name = pCommand->GetCommandName();

                AgentUtilities::CallLuaAgentUpdate(this, deltaMilliseconds, name);

                AgentUtilities::UpdateWorldTransform(this);



                // Then mark the command as completed
                pCommand->AddStatusComplete();


        }

        if (this->GetRigidBody())
```

95

```cpp
        {
                const btVector3 velocity = this->GetRigidBody()->getLinearVelocity();
                SetSpeed(Ogre::Vector3(velocity.x(), 0, velocity.z()).length());
        }
}

int SoarWaypointAgent::AgentType(){
        return 4;
}
```

96

Seeking Agent

```
# Initialization operator
sp {propose*initialize-seekingagent
   (state <s> ^superstate nil
          -^name)
-->
   (<s> ^operator <o> +)
   (<o> ^name seekingagent)
}

sp {apply*initialize-mover
   (state <s> ^operator <op>)
   (<op> ^name seekingagent)
-->
   (<s> ^name seekingagent)
}

# cleans the output-link once commands complete
sp {apply*cleanup*output-link
   (state <s> ^operator <op>
          ^superstate nil
          ^io.output-link <out>)
   (<out> ^<cmd> <id>)
   (<id> ^status)
-->
   (<out> ^<cmd> <id> -)
}

# misc useful elaboration rules
sp {elaborate*state*name
   (state <s> ^superstate.operator.name <name>)
-->
   (<s> ^name <name>)
}

sp {elaborate*state*top-state
   (state <s> ^superstate.top-state <ts>)
-->
   (<s> ^top-state <ts>)
}
```

97

```
sp {elaborate*top-state*top-state
  (state <s> ^superstate nil)
-->
  (<s> ^top-state <s>)
}

sp {elaborate*state*io
  (state <s> ^superstate.io <io>)
-->
  (<s> ^io <io>)
}

# Rule fires if target is near
sp {if-close*propose*newtarget
        (state <s> ^name seekingagent
                        ^io.input-link.distance < 3)
-->
        (<s> ^operator <op> + )
        (<op> ^name newtarget)}

sp {apply*newtarget
        (state <s> ^operator <op>
                                ^io.output-link <out>)
        (<op> ^name newtarget)
-->
        (<out> ^newtarget <f>)}

# Rule fires if target is not near
sp {if-far*propose*sametarget
        (state <s> ^name seekingagent
                        ^io.input-link.distance > 3)
-->
        (<s> ^operator <op> + )
        (<op> ^name sametarget)}

sp {apply*sametarget
        (state <s> ^operator <op>
                                ^io.output-link <out>)
        (<op> ^name sametarget)
-->
        (<out> ^sametarget <f>)}
```

98

Follower Agent

```
# Initialization operator
sp {propose*initialize-followeragent
  (state <s> ^superstate nil
        -^name)
-->
  (<s> ^operator <o> +)
  (<o> ^name followeragent)
}

sp {apply*initialize-mover
  (state <s> ^operator <op>)
  (<op> ^name followeragent)
-->
  (<s> ^name followeragent)
}

# cleans the output-link once commands complete
sp {apply*cleanup*output-link
  (state <s> ^operator <op>
        ^superstate nil
        ^io.output-link <out>)
  (<out> ^<cmd> <id>)
  (<id> ^status)
-->
  (<out> ^<cmd> <id> -)
}

# misc useful elaboration rules
sp {elaborate*state*name
  (state <s> ^superstate.operator.name <name>)
-->
  (<s> ^name <name>)
}

sp {elaborate*state*top-state
  (state <s> ^superstate.top-state <ts>)
-->
  (<s> ^top-state <ts>)
}

sp {elaborate*top-state*top-state
  (state <s> ^superstate nil)
```

99

```
-->
  (<s> ^top-state <s>)
}

sp {elaborate*state*io
  (state <s> ^superstate.io <io>)
-->
  (<s> ^io <io>)
}


#
sp {if-close*propose*follow
      (state <s> ^name followeragent
                    ^io.input-link.distance > 5)
-->
      (<s> ^operator <op> + )
      (<op> ^name follow)}

sp {apply*follow
      (state <s> ^operator <op>
                              ^io.output-link <out>)
      (<op> ^name follow)
-->
      (<out> ^follow <f>)}


#
sp {if-far*propose*stop
      (state <s> ^name followeragent
                    ^io.input-link.distance < 5)
-->
      (<s> ^operator <op> + )
      (<op> ^name stop)}

sp {apply*stop
      (state <s> ^operator <op>
                              ^io.output-link <out>)
      (<op> ^name stop)
-->
      (<out> ^stop <f>)}
```

100

Waypoint Agent

```
# Initialization operator
sp {propose*initialize-waypointagent
  (state <s> ^superstate nil
          -^name)
-->
  (<s> ^operator <o> +)
  (<o> ^name waypointagent)
}

sp {apply*initialize-mover
  (state <s> ^operator <op>)
  (<op> ^name waypointagent)
-->
  (<s> ^name waypointagent)
}

# cleans the output-link once commands complete
sp {apply*cleanup*output-link
  (state <s> ^operator <op>
          ^superstate nil
          ^io.output-link <out>)
  (<out> ^<cmd> <id>)
  (<id> ^status)
-->
  (<out> ^<cmd> <id> -)
}

# misc useful elaboration rules
sp {elaborate*state*name
  (state <s> ^superstate.operator.name <name>)
-->
  (<s> ^name <name>)
}

sp {elaborate*state*top-state
  (state <s> ^superstate.top-state <ts>)
-->
  (<s> ^top-state <ts>)
}

sp {elaborate*top-state*top-state
  (state <s> ^superstate nil)
```

101

```
-->
  (<s> ^top-state <s>)
}

sp {elaborate*state*io
  (state <s> ^superstate.io <io>)
-->
  (<s> ^io <io>)
}

# Rule fires if target is near
sp {if-close*propose*nextwaypoint
       (state <s> ^name waypointagent
                       ^io.input-link.distance < 3)
-->
       (<s> ^operator <op> + )
       (<op> ^name nextwaypoint)}

sp {apply*nextwaypoint
       (state <s> ^operator <op>
                              ^io.output-link <out>)
       (<op> ^name nextwaypoint)
-->
       (<out> ^nextwaypoint <f>)}

# Rule fires if target is not near
sp {if-far*propose*samewaypoint
       (state <s> ^name waypointagent
                       ^io.input-link.distance > 3)
-->
       (<s> ^operator <op> + )
       (<op> ^name samewaypoint)}

sp {apply*samewaypoint
       (state <s> ^operator <op>
                              ^io.output-link <out>)
       (<op> ^name samewaypoint)
-->
       (<out> ^samewaypoint <f>)}
```

Appendix C: Lua Scripts

Sandbox.lua – Sets up the environment

```lua
--[[
  Author: Daniel Lugo
  Based on Sandbox script created by David Young
]]

require "DebugUtilities";
require "GUI";

local ui;
local displaySkeleton;
local function CreateSandboxText(sandbox)
   -- Create a UI component to display text on.
   ui = Sandbox.CreateUIComponent(sandbox, 1);
   local width = Sandbox.GetScreenWidth(sandbox);
   local height = Sandbox.GetScreenHeight(sandbox);
   local uiWidth = 300;
   local uiHeight = 180;

   UI.SetPosition(ui, width - uiWidth - 20, height - uiHeight - 35);
   UI.SetDimensions(ui, uiWidth, uiHeight);
   UI.SetTextMargin(ui, 10, 10);
   GUI_SetGradientColor(ui);

   UI.SetMarkupText(
      ui,
      GUI.MarkupColor.White .. GUI.Markup.SmallMono ..
      "W/A/S/D: to move" .. GUI.MarkupNewline ..
      "Hold Shift: to accelerate movement" .. GUI.MarkupNewline ..
      "Hold RMB: to look" .. GUI.MarkupNewline ..
      GUI.MarkupNewline ..
      "Space: shoot block" .. GUI.MarkupNewline ..
      GUI.MarkupNewline ..
      "F1: to reset the camera" .. GUI.MarkupNewline ..
      "F2: toggle the menu" .. GUI.MarkupNewline ..
      "F5: toggle performance information" .. GUI.MarkupNewline ..
      "F6: toggle camera information" .. GUI.MarkupNewline ..
      "F7: toggle physics debug");
end

function Sandbox_Cleanup(sandbox)
```

103

```
end

function Sandbox_HandleEvent(sandbox, event)
   -- Pass events into the UI system.
   GUI_HandleEvent(sandbox, event);

   if (event.source == "keyboard" and event.pressed) then
      if (event.key == "f1_key") then
         Sandbox.SetCameraPosition(sandbox, Vector.new(-30, 5, 7));
         Sandbox.SetCameraOrientation(sandbox, Vector.new(-131, -68, -133));
      elseif (event.key == "f2_key") then
         UI.SetVisible(ui, not UI.IsVisible(ui));
      elseif (event.key == "space_key" ) then
         local block = Sandbox.CreateObject(
            sandbox,
            "models/nobiax_modular/modular_block.mesh");

         local cameraPosition = Sandbox.GetCameraPosition(sandbox);

         -- Normalized forward camera vector.
         local cameraForward = Sandbox.GetCameraForward(sandbox);

         -- Offset the block's position in front of the camera.
         local blockPosition = cameraPosition + cameraForward * 2;

         local rotation = Sandbox.GetCameraOrientation(sandbox);

         Core.SetMass(block, 15);
         Core.SetRotation(block, rotation);
         Core.SetPosition(block, blockPosition);

         -- Applies instantaneous force for only one update tick.
         Core.ApplyImpulse(
            block, Vector.new(cameraForward * 15000));

         -- Applies instantaneous angular force for one update
         -- tick.  In this case blocks will always spin forwards
         -- regardless where the camera is looking.
         Core.ApplyAngularImpulse(
            block, Sandbox.GetCameraLeft(sandbox) * 10);
      end
   end
end
```

104

```lua
function Sandbox_Initialize(sandbox)
    -- Caching a resource prevents a slowdown when the mesh/material/etc is
    -- first encountered.
    Core.CacheResource("models/nobiax_modular/modular_block.mesh");

    -- Create the demo specific UI, and default UI.
    GUI_CreateUI(sandbox);
    CreateSandboxText(sandbox);

    -- Set an initial camera position and orientation.
    Sandbox.SetCameraPosition(sandbox, Vector.new(-30, 5, 7));
    Sandbox.SetCameraOrientation(sandbox, Vector.new(-131, -68, -133));

    Sandbox.CreateSkyBox(
        sandbox, "ThickCloudsWaterSkyBox", Vector.new(0, 180, 0));

    -- When creating a plane from the Sandbox module this will attach a physics
    -- representation to the graphics representation.
    -- NOTE: A plane in the physics world is infinite in all directions.
    local plane = Sandbox.CreatePlane(sandbox, 200, 200);
    Core.SetMaterial(plane, "Ground2");

    -- Set the ambient light color.
    Sandbox.SetAmbientLight(sandbox, Vector.new(0.3));

    -- Create a Directional light for the sun.
    local directional =
        Core.CreateDirectionalLight(sandbox, Vector.new(1, -1, 1));

    -- Color is represented by a red, green, and blue vector.
    Core.SetLightDiffuse(directional, Vector.new(1.8, 1.4, 0.9));
    Core.SetLightSpecular(directional, Vector.new(1.8, 1.4, 0.9));

    --Sandbox.CreateSoarSeekingAgent(sandbox, "SeekingAgent.lua");
    Sandbox.CreateAgent(sandbox, "SeekingLuaAgent.lua");

        for i=1, 4 do

            Sandbox.CreateSoarFollowerAgent(sandbox, "FollowerAgent.lua");

        end

        --Sandbox.CreateAgent(sandbox, "FollowerLuaAgent.lua");
```

105

```
            Sandbox.CreateSoarWaypointAgent(sandbox, "WaypointsAgent.lua");

end

function Sandbox_Update(sandbox, deltaTimeInMillis)
    -- Update the default UI.
    GUI_UpdateUI(sandbox);

    -- Grab all Sandbox objects, not including agents.
    local objects = Sandbox.GetObjects(sandbox);

    -- Draw debug bounding sphere representations for objects with mass.
    DebugUtilities_DrawDynamicBoundingSpheres(objects);
End
```

Seeking Agent Script

```
--[[
 Author: Daniel Lugo
 Based on Lua agents created by David Young

 Copyright (c) 2013 David Young dayoung@goliathdesigns.com

 This software is provided 'as-is', without any express or implied
 warranty. In no event will the authors be held liable for any damages
 arising from the use of this software.

 Permission is granted to anyone to use this software for any purpose,
 including commercial applications, and to alter it and redistribute it
 freely, subject to the following restrictions:

  1. The origin of this software must not be misrepresented; you must not
  claim that you wrote the original software. If you use this software
  in a product, an acknowledgment in the product documentation would be
  appreciated but is not required.

  2. Altered source versions must be plainly marked as such, and must not be
  misrepresented as being the original software.

  3. This notice may not be removed or altered from any source
  distribution.
]]
```

```lua
require "AgentUtilities";

function Agent_Cleanup(agent)
end

function Agent_HandleEvent(agent, event)
end

function Agent_Initialize(agent)
  AgentUtilities_CreateAgentRepresentation(
    agent, agent:GetHeight(), agent:GetRadius());

  -- Assign a default target and acceptable target radius.


  agent:SetTargetRadius(2.5);
end

function Agent_Update(agent, deltaTimeInMillis, soarCommand)
  local destination = agent:GetTarget();
  local deltaTimeInSeconds = deltaTimeInMillis / 1000;
  local avoidAgentForce = agent:ForceToAvoidAgents(1.5);
  local avoidObjectForce = agent:ForceToAvoidObjects(1.5);
  local seekForce = agent:ForceToPosition(destination);
  local targetRadius = agent:GetTargetRadius();
  local radius = agent:GetRadius();
  local position = agent:GetPosition();
  local avoidanceMultiplier = 3;

  -- Sum all forces and apply higher priorty to avoidance forces.
  local steeringForces =
    seekForce +
    avoidAgentForce * avoidanceMultiplier +
    avoidObjectForce * avoidanceMultiplier;

  -- Apply all steering forces.
  AgentUtilities_ApplyPhysicsSteeringForce(
    agent, steeringForces, deltaTimeInSeconds);
  AgentUtilities_ClampHorizontalSpeed(agent);

  local targetRadiusSquared =
    (targetRadius + radius) * (targetRadius + radius);
```

107

```
   -- Calculate the position where the Agent touches the ground.
   local adjustedPosition =
      agent:GetPosition() -
      Vector.new(0, agent:GetHeight()/2, 0);

   if (soarCommand == "newtarget") then

      -- New target is within the 100 meter squared movement space.
      local target = agent:GetTarget();
      target.x = math.random(-50, 50);
      target.z = math.random(-50, 50);

      agent:SetTarget(target);
   end

   -- Draw debug information for target and target radius.
   Core.DrawCircle(
      destination, targetRadius, Vector.new(1, 0, 0));
   Core.DrawLine(position, destination, Vector.new(0, 1, 0));

   -- Debug outline representing the space the Agent can move
   -- within.
   Core.DrawSquare(Vector.new(), 100, Vector.new(1, 0, 0));
End
```

Follower Agent Script

```
--[[
 Author: Daniel Lugo
 Based on Lua agents created by David Young

 Copyright (c) 2013 David Young dayoung@goliathdesigns.com

 This software is provided 'as-is', without any express or implied
 warranty. In no event will the authors be held liable for any damages
 arising from the use of this software.

 Permission is granted to anyone to use this software for any purpose,
 including commercial applications, and to alter it and redistribute it
 freely, subject to the following restrictions:

  1. The origin of this software must not be misrepresented; you must not
  claim that you wrote the original software. If you use this software
```

108

in a product, an acknowledgment in the product documentation would be appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.
]]

```lua
require "AgentUtilities";

local leader;

function Agent_Cleanup(agent)
end

function Agent_HandleEvent(agent, event)
end

function Agent_Initialize(agent)
  AgentUtilities_CreateAgentRepresentation(
    agent, agent:GetHeight(), agent:GetRadius());

  -- Randomly assign a position to the agent.
  agent:SetPosition(
    Vector.new(math.random(-50, 50), 0, math.random(-50, 50)));

  -- Assign the first valid agent as the leader to follow.
  local agents = Sandbox.GetAgents(agent:GetSandbox());
  for index = 1, #agents do
    if (agents[index] ~= agent) then
      leader = agents[index];
                    Agent.SetLeader(agent, leader);
      break;
    end
  end
end

function Agent_Update(agent, deltaTimeInMillis, soarCommand)

  local deltaTimeInSeconds = deltaTimeInMillis / 1000;
  local sandboxAgents = Sandbox.GetAgents(agent:GetSandbox());
```

109

```lua
if (soarCommand == "follow") then

    -- Calculate a combining force so long as the leader stays
    -- within a 100 meter range from the agent, and has less than
    -- 180 degree difference in forward direction.
    local forceToCombine =
        Agent.ForceToCombine(agent, 100, 180, { leader } );

    -- Force to stay away from other agents that are closer than
    -- 2 meters and have less than 180 degree difference in forward
    -- direction.
    local forceToSeparate =
        Agent.ForceToSeparate(agent, 2, 180, sandboxAgents );

    -- Force to stay away from getting too close to the leader if
    -- within 3 meters of the leader and having a maximum forward
    -- degree difference of less than 45 degrees.
    local forceToAlign =
        Agent.ForceToSeparate(agent, 3, 45, { leader } );

    -- Summation of all separation and cohension forces.
    local totalForces =
        forceToCombine + forceToSeparate * 1.15 + forceToAlign;

    -- Apply all steering forces.
    AgentUtilities_ApplyPhysicsSteeringForce(
        agent, totalForces, deltaTimeInSeconds);
    AgentUtilities_ClampHorizontalSpeed(agent);

end

    if (soarCommand == "stop") then

        local forcestay =
        Agent.ForceToSeparate(agent, 3, 45, { leader } );

    AgentUtilities_ApplyPhysicsSteeringForce(
        agent, forcestay, deltaTimeInSeconds);
    AgentUtilities_ClampHorizontalSpeed(agent);

end

local targetRadius = agent:GetTargetRadius();
local position = agent:GetPosition();
```

110

```lua
    local destination = leader:GetPosition();

    -- Draw debug information for target and target radius.
    Core.DrawCircle(
        position, 1, Vector.new(1, 1, 0));
    Core.DrawCircle(
        destination, targetRadius, Vector.new(1, 0, 0));
    Core.DrawLine(position, destination, Vector.new(0, 1, 0));
End
```

Waypoint Agent script

```lua
--[[
 Author: Daniel Lugo
 Based on Lua agents created by David Young
 Copyright (c) 2013 David Young dayoung@goliathdesigns.com

 This software is provided 'as-is', without any express or implied
 warranty. In no event will the authors be held liable for any damages
 arising from the use of this software.

 Permission is granted to anyone to use this software for any purpose,
 including commercial applications, and to alter it and redistribute it
 freely, subject to the following restrictions:

  1. The origin of this software must not be misrepresented; you must not
  claim that you wrote the original software. If you use this software
  in a product, an acknowledgment in the product documentation would be
  appreciated but is not required.

  2. Altered source versions must be plainly marked as such, and must not be
  misrepresented as being the original software.

  3. This notice may not be removed or altered from any source
  distribution.
]]

require "AgentUtilities";

-- Waypoint list for agents.
local waypoints = {
    Vector.new(0, 0, 0),
    Vector.new(20, 0, 0),
```

111

```lua
        Vector.new(20, 0, 30),
        Vector.new(-20, 0, 0),
        Vector.new(-20, 0, 10)};

local index = 0;

local number_waypoints = 5;

function Agent_Cleanup(agent)
end

function Agent_HandleEvent(agent, event)
end

function Agent_Initialize(agent)
    AgentUtilities_CreateAgentRepresentation(
        agent, agent:GetHeight(), agent:GetRadius());

    -- Assign a default target and acceptable target radius.


    agent:SetTargetRadius(2.5);
end

function Agent_Update(agent, deltaTimeInMillis, soarCommand)
    local destination = agent:GetTarget();
    local deltaTimeInSeconds = deltaTimeInMillis / 1000;
    local avoidAgentForce = agent:ForceToAvoidAgents(1.5);
    local avoidObjectForce = agent:ForceToAvoidObjects(1.5);
    local seekForce = agent:ForceToPosition(destination);
    local targetRadius = agent:GetTargetRadius();
    local radius = agent:GetRadius();
    local position = agent:GetPosition();
    local avoidanceMultiplier = 3;

    -- Sum all forces and apply higher priorty to avoidance forces.
    local steeringForces =
        seekForce +
        avoidAgentForce * avoidanceMultiplier +
        avoidObjectForce * avoidanceMultiplier;

    -- Apply all steering forces.
    AgentUtilities_ApplyPhysicsSteeringForce(
        agent, steeringForces, deltaTimeInSeconds);
```

112

```lua
      AgentUtilities_ClampHorizontalSpeed(agent);

   local targetRadiusSquared =
      (targetRadius + radius) * (targetRadius + radius);

   -- Calculate the position where the Agent touches the ground.
   local adjustedPosition =
      agent:GetPosition() -
      Vector.new(0, agent:GetHeight()/2, 0);

   if (soarCommand == "nextwaypoint") then

      -- Next waypoint code
              index = index +1;
              if (index > number_waypoints) then
                index = 1;
      end

              agent:SetTarget(waypoints[index]);


   end

   -- Draw debug information for target and target radius.
   Core.DrawCircle(
      destination, targetRadius, Vector.new(1, 0, 0));
   Core.DrawLine(position, destination, Vector.new(0, 1, 0));

   -- Debug outline representing the space the Agent can move
   -- within.
   Core.DrawSquare(Vector.new(), 100, Vector.new(1, 0, 0));
end
```

113

Bibliography

[1] Office of the Deputy Under Secretary of Defense for Acquisition and Technology, Systems and Software Engineering, Developmental Test and Evaluation (ODUSD(A&T)SSE/DTE). Modeling and Simulation Guidance for the Acquisition Workforce, Version 1.0. Washington, DC, 2008.

[2] J. E. Summers. *Simulation-based Military Training: An Engineering Approach to Better Addressing Competing Environmental, Fiscal, and Security Concerns*. Washington Academy of Sciences, 2012.

[3] A. Heuvelink et al. "Intelligent Agent Supported Training in Virtual Simulations". TNO Defense, Security, and Safety. Soesterberg, Netherlands, 2009.

[4] Army and Marine Corps Training. *Better Performance and Cost Data Needed to More Fully Assess Simulation-Based Efforts*(GAO-13-698). United States Government Accountability Office. 2013.

[5] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Inc., 1995.

[6] P. Langley et al. "Cognitive architectures: Research issues and challenges", *Cognitive Systems Research.*, 2008.

[7] J. E. Laird, *The Soar Cognitive Architecture*, The MIT Press, 2012.

[8] Acquisition Community Connection. (2012, Feb. 24). *Joint Semi-Automated Forces (JSAF)* [Online]. Available: https://acc.dau.mil/ CommunityBrowser.aspx? id=500289&lang=en-US

[9] J. E. Laird et al. *Integrating Intelligent Computer Generated Forces in Distributed Simulations: TacAir-Soar in STOW-97*[Online]. Available: http://ai.eecs.umich.edu/ people/laird/papers/siw.html

[10] R. M. Jones et al. "Automated intelligent pilots for combat flight simulation". *AI Magazine*, vol. 20, pp. 27–41, 1999.

[11] J. E. Laird, "The Soar User's Manual Version 9.5.0". The Regents of the University of Michigan, 2015.

[12] J. E. Laird. "The Soar 9 Tutorial" (Updated for Soar 9.5.0), 2015.

[13] J. E. Laird et al. (2016, Jun. 7). *Soar Basics* [Online]. Available: https://web.eecs.umich.edu/ ~soar/tutorial16/Tutorial-2016-SW-basic.pdf

[14] I. Neath and A. M. Surprenant, *Human Memory: An Introduction to Research, Data, and Theory*, Thomson/Wadsworth, 2003

[15] R. S. Sutton, A. G. Barto, *Reinforcement Learning: An Introduction*, The MIT Press, 1998

[16] L. Mastin. (2010). *Episodic & Semantic Memory* [Online]. Available: http://www.human-memory.net/types_episodic.html

[17] "SML Quick Start Guide - Soar Cognitive Architecture", Soar.eecs.umich.edu, 2017. [Online]. Available: http://soar.eecs.umich.edu/articles/articles/soar-markup-language-sml/78-sml-quick-start-guide.

[18] D. Pearson. *XML Interface to Soar (SML) Software Specification.* ThreePeny Software LLC., 2005.

[19] J. E. Laird et al. (2016, Jun. 7). *Soar Tutorial – Introduction* [Online]. Available: https://web.eecs.umich.edu/~soar/tutorial16/Tutorial-2016-SW-intro.pdf.

[20] A. Newell. *United Theories of Cognitio*n. Harvard University Press, Cambridge, MA, 1990.

[21] R. Washington and P. S. Rosenbloom. "Applying Problem Solving and Learning to Diagnosis". *The Soar Papers: Research on Integrated Intelligence,* vol. 1, pp. 674-687. Cambridge, MA: MIT Press, 1993.

[22] D. Steier and A. Newell. "Integrating Multiple Sources of Knowledge into Designer-Soar, an Automatic Algorithm Designer" *AAAI-88 Proceedings*. AAAI Press, 1988.

[23] K. A. Johnson el al. "RedSoar: A system for red blood cell antibody identification". *Fifteenth Annual Symposium on Computer Applications in Medical Care,* pp.664-668. McGraw Hill, 1991.

[24] R.S. Chong and R.E .Wray. "Inheriting constraint in hybrid cognitive architectures: Appyling the EASE architecture to performance and learning in a simplified air traffic control task". *Modeling Human Behavior with Integrated Cognitive*

115

Architectures: Comparison, Evaluation, and Validation, pp. 237-304. Lawrence Erlbaum Associates, 2005.

[25] R. Hill et al. "Intelligent Agents for the Synthetic Battlefield: A Company of Rotary Wing Aircraft" *Proceedings of Innovative Applications of Artificial Intelligence* (IAAI-97). Providence, RI, July 1997.

[26] D. J. Pearson et al. "AIR-SOAR: Intelligent Multi-Level Control". *Artificial Intelligence Laboratory.* The University of Michigan, 2004.

[27] "Lua: about", *Lua.org*, 2017. [Online]. Available: http://www.lua.org/about.html.

[28] R. Ierusalimschy *Programming in Lua* (Third Edition). PUC-Rio de Janeiro, Brazil, 2013.

[29] D. Young. *Learning Game AI Programming with Lua.* Packt Publishing Ltd., 2014.

116

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
23-03-2017	Master's Thesis	Sep 2015 – March 2017

TITLE AND SUBTITLE		5a. CONTRACT NUMBER
A Sandbox in Which to Learn and Develop Soar Agents		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)	5d. PROJECT NUMBER
Lugo, Daniel, Captain, USAF	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)	8. PERFORMING ORGANIZATION REPORT NUMBER
Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/ENY) 2950 Hobson Way, Building 640 WPAFB OH 45433-8865	AFIT-ENG-MS-17-M-047

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
Intentionally Left Blank	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
  DISTRUBTION STATEMENT A. APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES
This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

It is common for military personnel to leverage simulations (and simulators) as cost-effective tools to train and become proficient at various tasks (e.g., flying an aircraft and/or performing a mission, among thers). These training simulations often need to represent humans within the simulated world in a realistic manner. 'Realistic' implies creating simulated humans that exhibit behaviors that mimic real-world decision making and actions. Typically, to create the decision-making logic, techniques developed from the domain of artificial intelligence are used. Although there are several approaches to developing intelligent agents; we focus on leveraging and open source project called Soar, to define agent behavior. This research took an off-the-shelf open-source software product (called the AI sandbox) that facilitates the creation of 3D virtual worlds and interfaced it to the Soar package. Because the world created by the sandbox is rich in features, easily configurable using a simple scripting system, and visually engaging, it's ideal as a learning platform to develop Soar agents more aligned with military simulations. In summary, this research develops a platform (or learning environment) to learn how to develop Soar-based agents.

15. SUBJECT TERMS
  Intelligent Agents, Soar Cognitive Architectre, Training Simulation, Artificial Inteligence

16. SECURITY CLASSIFICATION OF: UNCLASSIFIED			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON Dr. Douglas Hodson, AFIT/ENG
a. REPORT	b. ABSTRACT	c. THIS PAGE	UU	130	19b. TELEPHONE NUMBER *(Include area code)* (937) 255-3636, ext 4719 (douglas.hodson@afit.edu)
U	U	U			

117