Development of a Distributed Multi-Player Computer Game for Scientific Experimentation of Team Training Protocols

Sen Cao, sencao@cs.tamu.edu
Richard A. Volz, volz@cs.tamu.edu
Jamison Johnson, jcj0650@cs.tamu.edu
Maitreyi Nanjanath, maitreyi@neo.tamu.edu
Jonathan Whetzel, jwhetzel@neo.tamu.edu
Dianxiang Xu, xudian@cs.tamu.edu
Department of Computer Science
Texas A&M University
College Station, TX 77843

Abstract

To support our research on using intelligent agents in team training, we have developed a distributed multi-player game in Java that executes in real time. We based our development on the game Space Fortress, which has been widely used by cognitive psychologists for studying training protocols. Typical experiments involve in excess of 100 trials of the game under different training conditions. We incorporated features allowing flexible definition of experiment conditions (e.g., which players play which roles), mixed teams of human and intelligent software agents, synchronization of multiple player teams across a network, coordination of a formal experiment encompassing several different experimental conditions with many teams simultaneously playing the game, and the centralized maintenance of fine grained data on individual and team game activity. In this paper we describe the design of this game to achieve these features and real time performance in the Java environment.

1. Introduction

The basic thrust of our research is studying the use of intelligent agents in a variety of ways to enhance the training of teams, such as firefighters or NASA flight controllers, that must work together to accomplish a common goal. Our studies are conducted by a research team composed of both cognitive psychologists and computer scientists, and our results are validated by rigorously conducted formal experiments. As a vehicle for conducting these experiments, a computer game that requires both complex cognitive and motor skills is used. The particular game being used is named Space Fortress [1], and was developed more than a decade ago to reflect the kind of activity required in many real situations, e.g., piloting an aircraft in hostile surroundings. For a variety of reasons, this game had to be redeveloped to accommodate current research directions, incorporate many significant new features, and be compatible with advances in computer hardware and software.

The goals of our game development are non-traditional with respect to most of the computer game world. Rather than focusing on highly sophisticated graphical displays and intriguing game scenarios, we are primarily concerned with the ability to conduct scientific experiments in a rigorous manner that permits flexible experiment design, can capture extensive data on the conduct of the game, and can easily incorporate intelligent agents in various roles to facilitate training.

In this paper, we present the development of a new generation Space Fortress, which we call Revised Space Fortress (RSF). In Section 2, we describe the experimental procedures we wish to support and introduce the original Space Fortress game. In Section 3, we discuss the requirements for the system. Section 4 focuses on the design and system. In Section 5, we discuss validation of the implementation and initial experimental results. Summary and conclusion appears in Section 6.

2. Background

The use of computer games in research has a significantly different set of requirements than occurs for commercial computer gaming. In this section, we describe these differences and introduce the game Space Fortress that has been widely used in cognitive psychology research.

2.1 Typical Experimental Approach Supported

Our work on computer games is motivated by discovering intelligent agent techniques and protocols that can improve the process of training individuals and teams to perform tasks involving complex tasks, e.g., flight controllers. Such studies are normally conducted by

developing hypotheses on techniques that will improve training, and then testing them by having a significant number of human subjects perform the task being trained. There are normally at least two *experiment conditions* that are used. One, called the *control condition*, is used as a basis for comparison with the other conditions. The number of subjects needed for an experiment varies with the statistical parameters of the performance obtained, but 30 – 50 subjects for each condition is typical.

Since our objective is to study training, subjects must perform the task repeatedly, hopefully learning to perform better as they progress through their trials. Experiment conditions can be constructed differently, but with the Space Fortress experiments to date, e.g. [2-4], on the order of 100 trials is common, followed by another dozen or so follow on tests. Trials can be configured differently according to the experiment condition under study. For example, the team roles assigned may be changed from trial to trial. These trials are usually organized into sessions of a fixed number of trials and subjects usually only do one or two sessions at a time.

The impact of this kind of experimental structure is that our game design must

- include features to allow different experimental conditions to be easily defined,
- manage a number of teams to be simultaneously playing trials,
- manage the progress of each team through the trials ensuring that they play the proper configuration, and
- record all needed data ensuing from playing the game.

The impact of these and other considerations of this ilk will be discussed in detail in Sec. 3.

2.2 Space Fortress as a Research Vehicle for Training Protocols

Space Fortress (SF) was developed to facilitate research on training people to perform complex sensory/motor/ cognitive tasks similar to those occurring in many military situations. Mane and Donchin originally developed SF at the University of Illinois [1]. It was subsequently modified and ported to PC's by Gopher [5].

The most basic elements of the game are a fortress and a ship. The fortress tracks the ship and shoots at it. A player controls the ship and tries to destroy the fortress. Due to the speed at which the ship moves, it is actually moderately difficult to control it accurately. In addition, there are numerous features that require cognitive and motor skills. At random times during a game, mines appear and move toward the ship, causing damage if they hit it. However, there are two classifications of mines, and players must properly distinguish them or lose more points. The identity of the mine classifications is randomized at the being of each game, and a user has to memorize it before a game starts. A ship has a limited number of missiles it can shoot at the fortress. At random times, bonus opportunities appear, which give a player an opportunity to acquire either more missiles or an increase in score. If a player is not careful, the speed of a ship may cause it to wrap around the screen; each wrap costs a player from his/her score. The fortress is surrounded by two hexagons. The movement of the ship is to be kept in these hexagons; again, the score is reduced if the ship goes out of the region between the two hexagons.

There are numerous facets of the game of which a player must be aware, simulating situational awareness. The fortress can only be destroyed after being hit 10 times. The last shot must be a double shot fired within a specified time interval. Players must be aware of their missile supply to make good decisions on bonus opportunities. There can be a secondary interference task of tapping a board with one's feet at a specified rate, typically 2 hz.

Space Fortress has been widely used to examine issues such as gender difference, the effects of group discussion or strategies for practicing component skills. For optimal training effectiveness and efficiency, psychologists embedded their hypotheses into varied training protocols. Many training protocols have been applied to Space Fortress and conducted as experiments in last decades [2, 3, 6-8]. The effectiveness of a training protocol is shown by comparing its data with that of the control condition. Cumulatively, there is a large volume of SF experiment data. As it is a well understood research environment for which many experimental conditions have been explored and can be used for comparison, Space Fortress is still a useful tool for studying training of complex tasks.

3. Requirements for RSF

The research use of SF in a modern agent-oriented environment leads to a number of categories of requirements, flexible experiment description, fine-grained data acquisition, execution in a distributed environment and the ability to be run on standard, inexpensive platforms.

3.1 Experiment Description

A key requirement was that it be possible to define a large number of different experiments within RSF. To facilitate this, a multi-level experiment abstraction is used. A Space Fortress Experiment (SFE) is an organization of subjects repeatedly playing Space Fortress in a manner to test a training hypothesis. A SFE contains a number of

Space Fortress Experiment Conditions (SFEC). Each SFEC is composed of an ordered set of selected sessions. Each session is composed of an ordered set of selected trials. The number of sessions in a SFEC can vary, and different sessions in a SFEC are likely to have different numbers of trials and be present for different purposes, e.g., determining baseline performance before training, conduct of a training protocol, or testing retention after a training protocol has been completed[1]. A participant may play different roles in different trials and trials may be played either by teams or individuals. The definition of roles can be different in different trials. The definition of sessions, trials and roles does not depend upon knowing whether roles will be assumed by human subjects or intelligent agents. Instructions are given showing purpose, strategy, and functions before each trial. In some cases, it is desirable to dynamically select different configurations depending upon subject performance.

To facilitate testing different hypotheses, it is necessary to conduct different SFEC's without recoding Space Fortress program. Composable training protocols are desirable. Moreover, the specification of training protocols should be able to accommodate intelligent agents.

3.2 Data Requirements

Each SFEC in a Space Fortress experiment typically has at least ten sessions and usually lasts for a week. RSF is expected to keep track of experiment progress for all participants, advancing them to next trial or session automatically when appropriate and configuring trials dynamically. In order to accomplish this, RSF must store

- the participants in each team and which role each participant plays in a SFEC;
- the current session for each participant;
- the current trial within that session for each participant;
- the configuration of trials, including instructions and simulation characteristics; and
- scores to measure the participant's performance.

In addition to directly participating in trials to help human training, agents may serve as tutors by monitoring human subject behaviors and guiding them to make "better" decisions. In order to do this, fine-grained data about trial progress (individual actions) must also be acquired (and sometimes stored for after action reviews and trial replay).

For analyses of training results, experiment data is stored in a centralized location, though many trials may be in progress simultaneously on distributed computers. The representation of experiment data is also expected to be easily transformed into formats of data analysis programs.

3.3 Distribution of Activity

A Space Fortress experiment is normally conducted in a distributed manner. Participants must be able to join RSF, start trials, and quit RSF at any time. Participants (human subjects or intelligent agents) in a multiple player trial must be able to participate in their trials from different workstations, which may be connected by a local network or remotely by high speed Internet.

A variable number of teams in different SFEC's or even in different SFE's should be able to play their trials simultaneously. To distribute computation, a simulation server for each trial is delegated to workstation of some team member. RSF should configure dynamically simulation characteristics.

The data of experiment progress is stored centrally in RSF main server. Each simulation server also transfers trial progress after in the end of the trial.

3.4 System Considerations

It is desired that RSF be runable on current main steam platforms, i.e., PC's, with Windows, Linux or Unix. Java was selected as the programming language to make RSF useable on the above various platforms.

With the rapid advance of PC industry, PCs have a wide range of processor speeds. Almost all operating systems running on PC's are not real time systems. Despite of these disadvantages, real time performance is critical for RSF. Java just has rather coarse clock with a resolution of 55 ms.. Such accuracy is not precise enough to manage real time performance. Java does not provide the access to read joystick inputs either. Thus, the design must provide a higher resolution timing control.

4. RSF Design

In this section, we describe the representation of training protocols, overall system architecture, and major system components. We also explain how RSF flexibly manages experiments and accommodate both human subjects and intelligent agents.

4.1 Experiment Condition Specification

In order to realize the features mentioned in Section 3, we abstract the entities in SF experiments.

- Object: There are eight kinds of objects, ship, shell, fortress, missile, mine, bonus, hexagon, and tapper.

[1] Most commonly during the training process, a SFEC contains ten sessions and sessions contain ten trials.

- Device: the input devices used to operate objects. There are four kinds, keyboard, mouse, joystick and tapper. Device is defined in terms of the lower level action, e.g, joystick button 1 pressed, mouse key down, and so on.
- Operation: an operation is a single elementary (or atomic) action, which impacts some object. For example, ship thrust or firing a missile.
- Role component: a role component is a 3-tuple (*ob*, *op*, *dev*). It means using device *dev* to perform operation *op* on object *ob*.
- Role: a role is a set of role components performed by a single agent performing during a trial.
- Emphasis: SF has multiple score components, speed, control, point, velocity, and total. Subjects may be told to emphasize one of them.
- Instruction: These are displayed at the beginning of each trial.
- Trial: The execution of a SF game. It may be either for practice or test purposes. An emphasis and instruction may be associated with a trial.
- Session: An ordered set of trials. A session may associate with an emphasis, but possibly switch emphases based on trial performance.
- Abstract agent: An abstract entity playing a role. It is mapped to a role on a trial-by-trial basis.
- Space Fortress experiment condition: an ordered set of sessions with emphases. Multiple subjects are allowed to participate a SFEC at the same time.
- Abstract player: An abstract entity playing an abstract agent. The mapping is on a session basis.
- Team: A mapping from a group of agents (human or software) to a set of abstract players for a SFEC.
- Space Fortress experiment condition instance: The conduct of a SFEC by a team.

Based on these concepts, we proposed a four-level specification to express training protocols, which includes basic-level, trial-level, session-level, and SF experiment-condition-level specifications.

Basic-level specifications define legal devices, operations, combinations of devices and operations (i.e. which device is to do which operation), and instructions. Instructions can be in text or/and non-text format. Non-text instruction shows how some operations are to be performed.

Trial-level specifications define the roles and trial configuration. The number of roles determines whether a trial is a single or multiple player game. A role is composed of role components which specify what devices to perform what operations on what objects. The roles partition all functions in a trial. Trial configurations define the simulation characteristics to be used for the execution of a specific kind of trial, including objects and their properties, the operations on the objects, the duration of the trial, and other simulation parameters.

The session-level specification defines abstract agents, the structure of trials in a session, the structure of assignments from roles to abstract agents, and special instructions. The structure of trials shows a sequence of trials (defined in the trial-level specification) in a session, and whether the trials are for practice or test. The structure of assignments specifies abstract agents to play roles in each trial.

The SF experiment-condition-level specification defines abstract players, the structure of sessions, and the structure of assignments from abstract agents to abstract players in an experiment condition, and the structure of emphases. The structure of sessions shows a sequence of sessions (defined in a session-level specification) in a SFEC. The structure of assignments specifies the abstract players to play abstract agent mapping. The structure of emphases defines a scheme of instructions. The emphases in an experiment condition vary with session transitions. Advanced training protocols may allow emphasis advance to be adaptive to subject performance. To accommodate this kind of training protocol, special algorithms are specified in session level to advance emphasis.

4.2 Overall System Architecture

To realize the experiment condition structure given in the previous section, RSF has been developed in an object oriented fashion using Java. Three major components, the Central Control Module (CCM), the Simulation Engine (SE or SimEngine) and the Player/Viewer Module (PVM) interact as shown in Figure1.

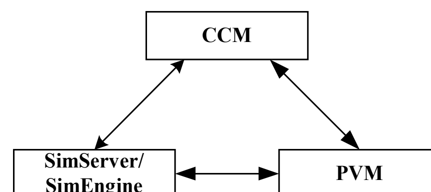

Figure 1: Overall organization of RSF

The CCM is the main server of RSF. It provides user management, experiment management, statistic services, and system maintenance. User management includes creating accounts and user login/logoff. Experiment management includes creating experiments and conditions, starting and stopping experiments and conditions, maintaining experiment progress, configuring simulation trials, coordinating the advances of individuals, and saving trial actions and results. Statistic services transform experiment results into proper formats for analysis tools. In case of unpredictable errors, system maintenance

provides a set of tools to return the CCM to "healthy" status and continue experiments without rebooting.

The SimEngine performs the actual simulation of a trial. The prior to the start of a trial, the SimEngine is configured for the particular trial to be conducted. During the conduct of a trial, the SimEngine performs operations according to inputs from PVM's and updates the simulation state. At the end of each cycle, the SimEngine publishes simulation states to each PVM. At the end of a trial, the SimEngine transfers scores and the sequence of actions to the CCM. The SimServer ensures that there is exactly one SimEngine running per team/individual trial.

The PVM is the terminal from which subjects participate in trials. Each PVM can manage one or more subjects. Before a trial starts, it obtains role information for each subject and instructions from the CCM. It is the interface to subjects during trials, both passing subjects' inputs to the SimEngine and displaying simulation status to subjects.

4.3 Subsystem View

4.3.1 Central Control Module (CCM)

A set of sub-components has been developed to achieve the functionalities of CCM. These sub-components are organized hierarchically as shown in Figure 2. For brevity, we do not include the statistical service, maintenance tools and other services in this section.

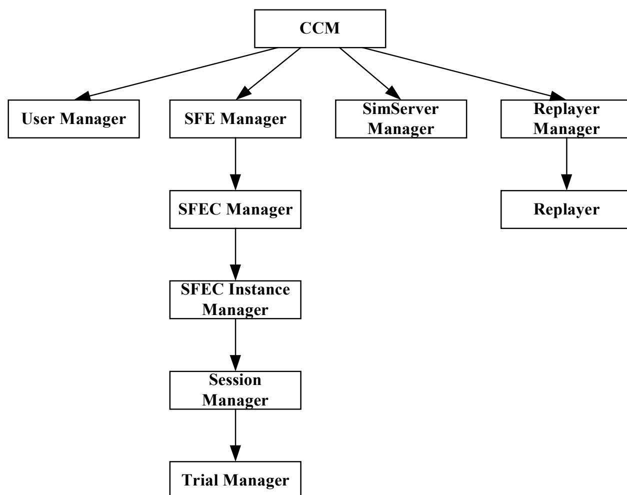

Figure 2: The structure of CCM

The User Manager provides the interfaces to create accounts and to log users on/off RSF system. When a user is trying to login to the RSF system, his/her password is checked.

The SimServer provides the service of running the SimEngine. Multiple SimServers are deployed in the RSF system. The SimServer manager manages all SimServers, including SimServer registration, SimEngine delegation, and starting and stopping SimEngine.

The Replayer manager coordinates trial replay. For each trial to be replayed, a Replayer is created to configure the SimEngine and coordinate replaying. Multiple trials can be replayed at the same time.

The hierarchy of CCM, SFE manager, SFEC manager, SFEC instance manager, session manager, and trial manager provide the services of experiment management, particularly maintaining experiment progress, configuring the SimEngine, and coordinating trial advance.

A SFE manager provides services for conducting an experiment. Since multiple experiments may be conducted at the same time, multiple instances of the SFE Manager may exist. SFE managers allow users so assigned to join the experiment they serve. It also instantiates a SFEC manager for each experiment condition.

A SFEC manager loads the experiment condition specification and allows the users so assigned to join the condition. The SFEC manager creates a SFEC Instance Manager for each online team and dispatches its members to the SFEC instance manager.

A SFEC instance manager manages its team's progress in an experiment condition. It creates a session manager for the team and sends it the information it needs such as current session, session-level specification and emphasis. When a session is over, SFEC instance manager upgrades its team's progress in the database.

A Session manager loads its session specification and instructions, according to current experiment condition specification. It then makes the assignment from its team to abstract agents. It also monitors the team's progress in the session, including current trial and trial specification. A Session Manager creates a trial manager for the team (or individual). When the team completes its trial(s), a session manager updates team progress (and emphasis if the scheme of adaptive emphasis advance is being used).

A Trial Manager loads the specification of the trial to be conducted and configures the SimEngine that will be used for the trial. It decides the roles each user plays and then tells the users their role information. It also coordinates all PVM's used by the users and SimEngine. Inside a trial manager, a data recorder is associated to save users' trial actions, simulation states and scores.

The CCM uses a database system to store the four-level training protocol specifications, user information, experiment progress, and scores. It uses JDBC to access the database system.

4.3.2 Simulation Server and Simulation Engine

The SimServer and SimEngine components are organized hierarchically as shown in Figure 3. A SimEngine is the component that actually performs a trial simulation. The SimServer was created to provide flexibility in the location at which the SimEngine runs. There can be a number of SimServers in the RSF system. SimServers can be started in two ways, one on each machine with each PVM or as a separate pool of simulation servers. To ensure correct performance, the SimServers must run exactly one SimEngine for each team (or individual) executing a sequence of practice trials, but one SimEngine for each individual (even if in a team) executing a test trial. To achieve the necessary synchronization, the SimManager in the CCM manages all SimServers and directs them to create instances of the SimEngine as needed and directly by the experiment definition database.

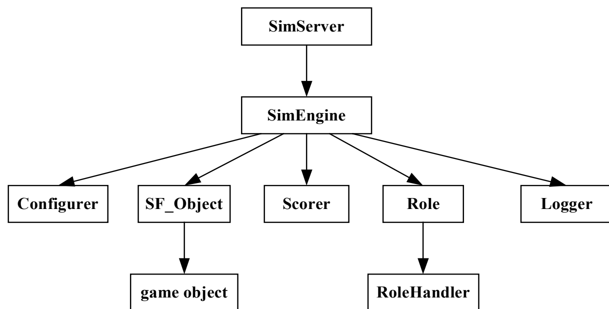

Figure 3: The structure of SimEngine

The SimEngine, in turn, creates instances of a number of utilities to help it do its job. The Configurer obtains trial specification including the simulation characteristics, the objects in the trial and their properties, and the roles from CCM, and initializes the SimEngine for a trial. It also obtains the assignment of participants (human subjects and intelligent agents) to the roles.

Recall that there are a number of different kinds of objects in the game, e.g., ship, fortress, missile, etc. The class SF_Object has been designed to the common functionality of all kinds of objects. Each specific type of object extends SF_Object. Some, e.g., fortress, are created during initialization for a trial. Others are created during the play of a trial, e.g., missiles or mines. Also, some objects may be removed, e.g., when a mine is destroyed.

The component Role was designed to implement the concept a role. There may be multiple instances. A RoleHandler is created for each role as the interface to a participant. Prior to the start of a trial, each PVM requests its RoleHandlers. According to the assignment of participants to roles, the SimEngine returns corresponding RoleHandlers. A RoleHandler is called by the PVM to pass inputs, which in turn triggers its Role to perform the corresponding operations. Through the RoleHandler, the SimEngine also publish simulation states to each PVM at the end of each simulation cycle.

The Scorer maintains participant scores during a trial. At the end of each cycle, SimEngine also sends trainee scores to that point to the PVMs via the RoleHandlers .

The Logger has a data buffer that records simulation states, user inputs, operations on objects, and final scores. At the end of a trial, the Logger transfers these data to a data recorder in trial manager (on the CCM), which saves these data in a log file and scores in the database.

The SimEngine is also used for replaying a trial. In this case, the Logger retrieves trial data from the CCM. It parses the data and regenerates events to the RoleHandlers..

4.3.3 Player-Viewer Module (PVM)

PVM contains a set of frames that interact with participants, as shown in Figure 4. Login frame allows participants to login and join their experiments. Once a PVM participates in a trial, it retrieves instructions from CCM and displays them to the participants. The PVM requests Rolehandlers (from the SimEngine) for the trainee(s). The PVM uses these to pass inputs to the SimEngine. It receives the simulation state at the end of each cycle, and passes it to the Game frame, which refreshes the display of simulation states. The Score frame shows trainee's scores at the end of a trial.

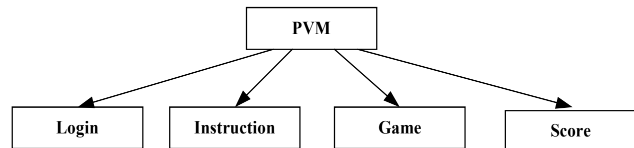

Figure 4: The structure of PVM

4.4 Flexible Experiment Management

In Sec. 4.1, a four-level specification was proposed to express various training protocols. We have designed a database system [10], which uses a set of tables to represent this four-level specification. These tables allow one to specify a broad range of experiments. Table values defining experiments and conditions may be entered either through the CCM or through separate database tools. Space does not permit detailed explanation of these tables in this document.

The database system contains a set of tables to record subject information and deploy subjects to experiments. Subjects register through PVM's and their information is saved by the CCM. The CCM randomly assigns subjects to teams and experiment, and records the assignment.

In addition, the database system contains a set of tables (maintained by the CCM) to keep track of experiment progress. The SFEC manager, the session manager, and the trial manager, introduced in Sec. 4.3.1, load and parse corresponding specifications, and update experiment progress at their corresponding levels. Experiment progress includes current session, trial, assignment of participants to abstract agents at the session level and to roles at the trial level, and scores. Figure 5 shows the relationships among training protocols, the CCM and experiment data.

RSF also has fine-grained data acquisition, including users' trial actions, simulation states and scores at the end of each cycle. The fine-grained data is transferred from a SimEngine to the CCM at the end of each trial, whereupon the CCM saves it in a file.

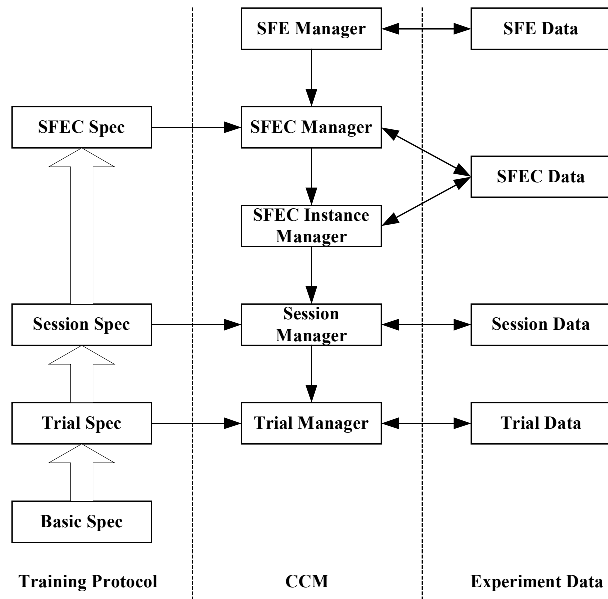

Figure 5: The relationships among training protocol, the CCM, and experiment data

4.5 Compatible Interfaces To Human Subjects And Intelligent Agents

An important RSF feature is that it provides a uniform interface to both human subjects and intelligent agents. This feature allows a team to be formed by a mixture of human subjects and intelligent agents. This makes it possible to design advanced training protocols in which intelligent agents help the training of human subjects. There are two aspects to interfacing to intelligent agents.

The SimEngine is compatible with intelligent agents. The concept of role is the basis of RSF. It does not distinguish what kind of entity is playing a role. In the SimEngine, a RoleHandler is the interface to a role. Either an intelligent agent or a human subject (through PVM) can obtain the RoleHandler and use it to send inputs to the SimEngine. At the end of each cycle, the SimEngine simply sends the simulation state to each PVM, without knowing whether a human or intelligent agent is using it.

The CCM is compatible with intelligent agents. The four-level specification uses the concepts of abstract player, abstract agent, and role. If an experiment uses intelligent agents to help training, they can be specified to take abstract players, abstract agents, and roles.

Thus, to incorporate an intelligent agent, one simply has a PVM variant that forwards the simulation state to the agent and has the agent implement the interface through which the PVM obtains data.

5. Implementation Issues

5.1 Real Time Performance

Achieving real-time performance with non-real-time platforms such as Windows requires careful treatment of issues such as time measurement, achieving fine resolution delays, managing system load, and controlling garbage collection. A detailed treatment of these issues is beyond the scope of this paper, but an extended treatment may be found in [9]. However, it is useful to show the stability of timing performance achieved and briefly summarize the techniques used.

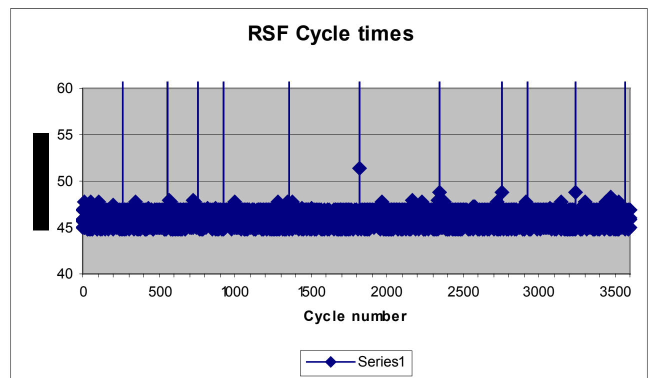

Figure 6. Achieved cycle times for SimEngine.

The standard PC only has a resolution of 55 ms., and times are rounded to the nearest 20 ms., totally inadequate. A high resolution timer is available, though hard to time, for modern PCs. Using this, clock routines were developed that have a resolution on the order of 1 μsec.; the actual resolution varies depending upon hardware, operating system and system clock speed, but is typically around this value. With Java, delay times with a resolution of 1 ms. can be achieved, which is adequate.

Memory allocation/deallocation can be a major inhibitor to real-time performance. We used a three phase approach

to control this to the point that acceptable performance is achieved. First, many of the larger objects were pre-allocated and reused during a trial. While this did not eliminate the problems associated with memory, it did reduce them to a level that could be managed through garbage collection control. Garbage collection can take a few hundred milliseconds in RSF. However, Java 1.3.x provides a two phase garbage collection in which there are a frequent, but short, collections, and less frequent, but longer collections. By adjusting command line parameters we were able to reduce the short collection times to approximate 1.7 milliseconds occurring only about once every 800 ms. Further, we were able to control the longer collections so that they only occurred between trials when their length of time does impede game progress.

Figure 6 shows the resulting cycle times in the SimEngine, which controls game timing. The nominal time is 46 ms. The resolution of delay times in Java is 1 ms., so we could only expect to achieve times between 45 and 47 ms. As can be seen, the times were rarely out of this range. The large spikes that occur correspond to explosions of either fortress or ship and are supposed to be longer.

5.2 Synchronization

5.2.1 Clock Synchronization

RSF is a real time multiplayer game. Multiple PVM's and a SimEngine communicate during a trial and may be located on different machines having different local clocks. In each cycle, the SimEngine processes inputs from all PVM's and publishes the simulation state to all PVM's; each PVM passes inputs to the SimEngine and displays the simulation state to the trainee. A uniform clock is important to keep their interactions going correctly.

Cristian's algorithm [10] is applied to synchronize the clocks in the PVM's and SimEngine. The SimEngine serves as the time server. The difference between the clock of the SimEngine and that of each PVM is calculated so that each PVM gets the uniform time by reading and adjusting local clock. A plenty of tests have proven the accuracy is less than 1 millisecond.

5.2.2 Mutual Exclusion

RSF uses one CCM, a variable number of SimEngines and PVM's accessing various data and resources concurrently. For example, multiple PVM's associated with a trial access a structure of users and their IP addresses concurrently. Mutual exclusion is widely applied to protect process to access the shared data without being interrupted by others and prevent deadlock and livelock. Mutual exclusion is implemented in Java by three ways: synchronized method, synchronized data, and locks. N-mutual exclusion to allow up n processes to shared data and resources concurrently is implemented by locks.

5.2.3 Multiple Phase Confirmation

RSF has special properties differing form most popular entertainment games, such as War Craft, Age of Empire, and Star Craft. In RSF, participants are randomly assigned to team before performing trials and session. A trial is not isolated, but it is just one of game sequences. A participant advances trials in an experiment. The sequence of trials could be either team trials or individual trials. It is necessary to synchronize trial advance for a whole team.

A multiple-phase confirmation protocol has been developed to accomplish such synchronization. The basic idea of the multiple-phase confirmation protocol is similar with the two-phase commit protocol of atomic transaction [11]. However, when one considers the possibility of team members finishing individual test games at different times and independently logging off and back on again, required us to extend this to multiple phases. The multiple-phase confirmation protocol allows a trial to be cancelled in any stage or starts the trial until all participant commit.

5.3 Memory Limitations

When a trial is over, the SimEngine transfers log data to the CCM, including user inputs and operations, scores and simulation states. This causes a large consumption of memory, up to 8 Mb. The data recorder takes a few seconds to save the data into a log file and the database. If a large number of trials end in a short period, JVM may not be able to provide enough memory. N-mutual exclusion is applied to control the number of concurrent log saving.

6. Validation

We have done several categories of validation on RSF as follows:

- The basic properties of the game have been measured, such as object dimensions (ship, fortress, hexagons, mines, shells, and missiles), maximum object speeds (ship, missiles, shells, mines), the frequency at which mines appear, the frequency at which bonuses appear, etc. These properties were set identical to those in the original SF. The measurements on RSF and the original SF have been compared to ensure equivalency.
- The timing performance and memory utilization have been measured. These show that stable cycles have been achieved despite of a few unnoticeable

variances. Memory utilization grows to reasonable level and remains stable.

- Experts were asked to execute both simulations. Whenever discrepancies were found, both original and revised codes we examined to determine and correct the cause.
- Human subject experiments have been conducted. Each experiment had 8 or 16 subjects and lasted for a week. Each subject was required to conduct several sessions on both RSF and original version. Questionnaires were given to report discrepancies and their feelings.

Further human subject tests will be conducted shortly to check that there is not a statistically significant difference between scores obtained with RSF and the original.

7. Conclusions and Future Work

In this paper, we presented the requirements and design of RSF. A four-level specification was developed to express training protocols, which allows a broad range of experiments and conditions to be defined. RSF contains three components, the CCM, the SimEngine and the PVM. The CCM manages experiment progress; the SimEngine conducts the trial simulation; and the PVM interfaces to participants. We also explained how RSF can accommodate intelligent agents. Real time performance and the control of memory utilization have been achieved. RSF has also been validated in several different ways to ensure compatibility with the original version.

In the future, the game will be extended to actually include several different intelligent agents, e.g., a coaching agent that recognizes the strategies of players can gives advice on more effective ways to play the game. In addition, modifications will be made to allow several different component strategies to be played in isolation, both to measure what players have learned after a full training experiment and to provide training to other players on particularly difficult tasks, such as flying the ship slowly enough to allow the fortress to be targeted.

8. Acknowledgments

The work in this paper was funded by DoD MURI grant F49620-00-1-0326 administered through AFOSR.

9. References

[1] A. Mane and E. Donchin, "The Space Fortress Game," *Acta Psychologica*, vol. 71, pp. 17-22, 1989.

[2] D. Gopher, "The Skill of Attention Control: Acquisition and Execution of Attention Strategies," in *Attention and Performance: Synergies in Experimental Psychology, Artifical Intelligence, and Cognitive Neuroscience*, vol. XIV, E. Meyer and S. Korenblom, Eds. Hillsdale, NJ - Cambridge, MA: Erlbaum - MIT Press, 1993, pp. 299-322.

[3] W. Arthur Jr., E. A. Day, W. Bennett Jr., T. L. McNelly, and J. A. Jordan, "Dyadic Versus Individual Training Protocols: Loss and Reacquisition of a Complex Skill," *Journal of Applied Psychology*, vol. 82, pp. 783-791, 1997.

[4] W. L. Shebilske, J. W. Regian, W. Arthur Jr., and J. A. Jordan, "A dyadic protocol for training complex skills," *Human Factors*, vol. 34, pp. 369-374, 1992.

[5] D. Gopher, M. Weil, and T. Bareket, "Transfer of skill from a computer game trainer to flight," *Human Factors*, vol. 36, pp. 387-405, 1994.

[6] W. Shebilske, B. Goettl, and J. W. Regian, "Executive Control and Automatic Processes as Complex Skills Develop in Laboratory and Applied Settings," in *Attention and Performance XVII: Cognitive regulation of performance: Interaction of theory and application.*, D. Gopher and A. Koriat, Eds. Cambridge, MA: MIT Press, 1999, pp. 401-432.

[7] J. R. Frederiksen and B. Y. White, "An Approach to Training Based on Principled Task Decomposition.," *Acta Psychologica*, vol. 71, pp. 89-146, 1989.

[8] E. A. Day, W. Arthur Jr., and W. L. Shebilske, "Ability determinants of complex skill aquisition: Effects of training protocol," *Acta Psychologica*, vol. 97, 1997.

[9] R. A. Volz, J. C. Johnson, S. Cao, M. Nanjanath, J. Whetzel, T. R. Ioerger, B. Raman, W. L. Shebilske, and D. Xu, "Fine-Grained Data Acquisition and Agent Oriented Tools for Distributed Training Protocol Research: Revised Space Fortress," 2003.

[10] F. Cristian, "Probablistic Clock Synchronization," *Distributed Computing*, vol. 3, pp. 146-158, 1989.

[11] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design*, Third ed. Harlow, England: Addison-Wesley, 2001.