Network Clubhouse: A Constructive Learning Environment for Children

by

Loren C. Shih

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 28, 1996

Author .. ..
Department of Electrical Engineering and Computer Science
May 28, 1996

Certified by . .....
Professor Mitchel Resnick
Associate Professor of Media Arts and Sciences
Program in Media Arts and Sciences
Thesis Supervisor

Accepte .....
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Network Clubhouse: A Constructive Learning Environment for Children

by

Loren C. Shih

Submitted to the
Department of Electrical Engineering and Computer Science

May 28, 1996

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and
Engineering and Master of Engineering in Electrical
Engineering and Computer Science

ABSTRACT

Network Clubhouse is a constructive, graphical environment which allows children to collaborate in a shared virtual world using Logo-like control of objects. While Logo enables children to individually learn through design and construction, Network Clubhouse explores learning through communities of children jointly constructing projects in a cooperative environment.

Thesis Supervisor: Mitchel Resnick
Title: Associate Professor, Media Arts and Sciences

Table of Contents

Introduction

This thesis project involves the design and implementation of *Network Clubhouse*, a multi-user version of Logo. I have been involved with the Network Clubhouse project since Fall 95 and am submitting this document in partial fulfillment of a Masters of Engineering degree in Electrical Engineering and Computer Science. However, I will continue this project over Summer 96 and will substantially add to this document in my final version. I encourage readers to refer to that document when it is completed for more timely and useful information.

The document is designed to fulfill two goals. The first is to give researchers an idea behind the motivation and design of Network Clubhouse. As Network Clubhouse is somewhat unique and part of a new development of networked applications, I hope this document will clearly outline some of the potentials and limitations of those environments. The second is to provide future developers of Network Clubhouse with a technical background of the implementation of Network Clubhouse. I hope this document will clearly describe the project with enough detail to facilitate future work.

This thesis arranges to present Network Clubhouse in an increasing technical manner. The first major section outlines the background and motivation behind the project. The second section describes some of the higher level design challenges in Network Clubhouse. The third section addresses technical challenges in the implementation of Network Clubhouse.

Acknowledgments

The primary developers of the Network Clubhouse project were myself, Andrew Begel, professor Mitchel Resnick, and Brian Silverman.

Begel is an undergraduate student who was primarily in charge of the implementation of the Network Clubhouse client. He also provided much of the support I needed for the implementation of the server, interpreter, and compiler because of his superior background in language design. I am indebted to Begel both for his implementation of the client and continuing supervision of my server design.

Silverman is an MIT graduate working at Logo Computer Systems Inc., which produces projects such as "MicroWorlds Logo" and "My Make Believe Castle." He has a continuing affiliation with the Logo Lab and provided much expert advice and assistance on the design and implementation of Network Clubhouse. Silverman also designed the Cocoa language, discussed below. I thank Silverman for his advice on higher level design issues and help with low-level implementation details

Professor Resnick is the head of the Epistemology and Learning Group at the MIT Media Lab. In addition to acting as my advisor on this project, he also contributed most of the direction and goals of Network Clubhouse. Professor Resnick first introduced the Network Clubhouse idea as an extension from his earlier StarLogo project. I am grateful to Professor Resnick for his input on the goals and motivation of Network Clubhouse, help on higher level design issues, close supervision on my progress throughout the course of the project, provision of background and reference material, and involvement in the creation and revision of this thesis document.

Chapter 1

The Network Clubhouse Project

1.1 Overview

Network Clubhouse is based on a graphical multi-user environment similar to that of traditional MUDs. Users control graphical objects capable of basic animation and manipulate those objects through a command interface. They communicate with others through a chat feature. The environment itself is very general; it can pose challenges which users must solve cooperatively or provide a forum in which users can interact with each other.

1.2 Comparison with other networked applications

In this regard, Network Clubhouse is not unique from any of the graphical multi-user environments which currently are available in the network community or even the rising number of networked games such as "Doom" or "Netrek" that are popular among both children and adults. Since many networked applications which connect people together in communities already exist, it seems that a project such as Network Clubhouse would not be contributing to the field. However, the main difference between Network Clubhouse and other applications lies in the *focus* of those applications. Networked environments such as "Doom" typically focus on providing entertainment while minimally addressing education and constructive interactions. Consequently, these games may promote generally destructive behavior such as violence and adversarial communities because players compete against each other.

1.3 Goals

MUDs are usually primarily focused on solving goals posed by the environment, where

users are able but do not necessarily need to collaborate to solve those challenges. Network Clubhouse environments are more built around the user; environments are created to allow users to primarily explore interactions with each other toward some common goal. While general MUDs tend to stress interaction with the environment, Network Clubhouse MUDs are centered around creative writing, communication, and programmability.

Network Clubhouse an educational tool which is centered around collaborative design and modeling projects. Its goal is to enable children to develop constructive experiences together. Whereas traditional MUDs are generally concerned with achieving tasks that are inherent in the environment, Network Clubhouse is distinguished because it focuses on *construction*, where users *create* and *build* things together. Projects such as the "virtual fishtank," described below, will explore new educational phenomena which are based on extensive cooperation and group interaction.

Network Clubhouse is modelled after Logo, a general graphical programming environment which was developed primarily as a learning tool and has challenged researchers to rethink the role of construction and design in education.

1.4 Conclusion

A wide variety of graphical applications currently exist, and new ones are being introduced at a high rate as developers recognize the appeal of networked environments for both entertainment and interaction. However, Network Clubhouse uniquely explores the educational possibilities of collaborative environments and may revolutionize the manner in which children's education is integrated with new technologies.

Chapter 2

Previous Research and Motivation

2.1 Overview

The motivation behind Network Clubhouse relates to previous research at the Epistemology and Learning Group. The primarily aim of Network Clubhouse is to provide a multi-user version of an application called StarLogo developed by Mitchel Resnick. StarLogo, in turn, was an extension of Logo, a familiar children's educational tool.

2.2 The Epistemology and Learning Group

2.2.1 Overview

Before presenting the Network Clubhouse project, I will introduce some background information on the Epistemology and Learning Group at the MIT Media Lab, and particularly the relationship between the goals of our group and the Network Clubhouse project.

2.2.2 Goals

The Epistemology and Learning Group at the MIT Media Lab is engaged in the exploration of the role of computer tools in education. We explore how new technologies can be integrated with education goals of learning, constructing, collaborating. Our tools are presented in educational communities such as schools and museums.

2.2.3 Traditional technological paradigms

The Epistemology and Learning Group explores the role of construction and community in education. Traditionally, new network technologies are hailed primarily as ways of increasing information availability. Television allows lectures to be broadcast to arbitrarily large numbers of users, and VCRs enable students to record and share media information.

Similarly, computer network tools such as Netscape allow information to be communicated and distributed.

2.2.3.1 Information delivery

Computer network tools can be used to deliver information. Students might someday be exposed to an electronic classroom setting with multimedia tools which the instructor could use to more effectively communicate lecture topics. Electronic whiteboards, overhead monitors, and other audio and video devices might substantially improve the ability of the teacher to communicate ideas. Remote video conferencing is another example of the potential of this technology, allowing students to possibly participate in interactive lectures without leaving their homes.

2.2.3.2 Information accessibility

Computer network tools can also be used to increase information accessibility. Students could be assigned projects which required them to gather data from the web. Technologies such as Netscape would be available to search for relevant information.

- Example: Students might be presented with an art assignment and told to gather multimedia text and graphics information from various museums. In the past, this would be accomplished through library searches, whereas students can now easily use a Netscape browser to almost instantly retrieve relevant information.

As search engines and data organization improve, gathering and sharing information may becoming almost trivially easy compared to the effort current needed to gather information.

2.2.4 The Epistemology and Learning Group paradigm

While educators hail network technologies for their role in improving information availability, the Epistemology and Learning Group explores new ways of using the Internet as a medium for *construction* and building *community.*

2.2.4.1 Construction

Students learn through exposure to information in settings such as lectures, audio instruction tapes, and video programs. Another primary means is learning through *construction*, or *active* learning. Papert's theory on *constructionism* refers to two types of construction. The first asserts that knowledge is not simply transmitted from teacher to learner but is actively constructed in the mind of the learner. The second states that people learn with particular effectiveness when they are engaged in designing and constructing meaningful artifacts [Papert].

Children build tools and create projects as part of their basic learning process. Interaction with modelling clay such as "Play Doh" or basic construction kits such as LEGOs or "Tinker Toys" relate to Papert's dicsussion of learning through building and designing.

- Example: The MIT **6.270 LEGO Robotics Competition** illustrates how students can learn about mechanical and hardware design on a practical, useful level by constructing autonomous robots from LEGOs. Teams are provided with kits containing LEGOs, sensors, and microprocessor circuits which they use to build an autonomous robot to perform specific competition tasks. While classes may prepare students with theoretical knowledge about how to handle real-world principles and uncertainties, only through experiencing these in a hands-on setting can students understand them at a practical level. Many students praise 6.270 as a uniquely useful engineering class because of its emphasis on construction.

- Example: Ten-year-old children at the Open School simulated behavior of a clown fish in an ocean ecosystem [Kay, 140]. The clown fish interacts with a sea anemone to build immunity to its poisonous stings, and then take refuge from predators by hiding among the anemone's tentacles. While students could have learned about this behavior by reading books and watching videos, constructing the simulation allowed them to better relate to the challenges in the ocean environment and the benefits of symbiosis.

Information sharing through traditional non-interactive means such as lectures provides a foundation for learning, but students of all ages and fields can benefit from exposure to practical constructive experiences to complete their learning experience.

2.2.4.2 Community

Network tools also provide a means for building community. The success of MUDs (Multi-User Dungeons, or Multi-User Domains) as community-building environments illustrates the contribution of network tools to this goal. MUDs such as IRC, chatrooms, and other conferencing tools prove the capability of networked environments to connect people together in a shared space.

- Example: The first MUD was developed in 1979 to allow users to jointly play "Dungeons and Dragons." Eventually, users started emphasizing the actual goals of their environment less and began using it more as a communication tool. In 1989, James Apnes at Carnegie Mellon University created a project called *TinyMUD* where monsters and objects were largely removed from the environment [Bruckman 1, 96]. The MUD evolved into a forum where users could communicate virtually and extend the virtual world together. Subsequently, some MUDs became less goal-oriented and stressed interaction between people in a community.

Participation in communities is an essential part of child development, allowing comparison and collaboration on children's projects. Communities allow children to receive feedback on projects and build social skills.

2.2.4.3 Network Clubhouse and Community and Construction

Network Clubhouse follows the Epistemology and Learning Group's goal of using network tools to encourage community and construction. Children build objects in the world and are completely involved in generating the behavior of their creations. Children can also build environments in Network Clubhouse or manipulate existing ones in this constructive settings.

Children will interact with each other extensively in Network Clubhouse. This is not just a peripheral capability; it is the central basis behind their involvement in the world. Children need to cooperate to solve tasks such as designing ecosystems or addressing specific challenges presented by the environment. Not only will they be able to communicate with each other directly through the chat feature, but their objects will also interact and

respond with each other on a meaningful level. This sense of community building will encourage the kind of group and social skills essential in basic child development.

2.3 Logo

Logo is a programming environment for elementary school children created by Seymour Papert [Mozes, 6]. Users manipulate a single "turtle" object in 2-D graphical space. The turtle acts as a paintbrush by leaving a trail as it moves, allowing simple graphical sketches to be created. This is similar to using an "Etch-A-Sketch" to create pictures. Turtle manipulation is handled through a command-line interface where children can enter simple statements such as "*forward*," "*turn*," "*backward*," or "*penup*." Users can program *behaviors* into turtle objects by designing procedures which specify turtle actions.

2.3.1 Logo as an educational programming environment

Logo may resemble a basic drawing utility or application tool, but is far more comprehensive. It has challenged educators to rethink the educational process and especially the role of creation and construction in learning. Not only do children learn the fundamental programming concepts through using Logo, but the child's relation to his turtle object also involves basic educational concepts.

2.3.2 Active learning

The traditional learning process focuses on memorization and repetition, whether through mathematical tables or lectures. This is *passive* learning, and it arguably lacks some components which are important to learning. Students also learn through more *active* processes such as building sculptures in art class or preparing presentations on subjects. This is learning through *construction*, as described earlier.

- Example: In the college setting, it is obvious that building a compiler plays a fundamental role in learning about compilers. In MIT's **6.035 Compilers** class, students participate in extensive lectures on compiler theory, but the actual design and implementation of the compiler forms the basis of their learning experience. However, even in this exam-

ple, students are more or less handed the specifications and given little freedom for creativity.

Logo expands upon the formal teaching paradigm of passive learning by allowing children to interact with projects, becoming active participants in their own education. The user assumes the fundamental role as creator and designer.

2.3.3 Relationship to existing knowledge

Logo relates to the existing knowledge of children instead of dissociating teaching from previous experience. Traditional classes treat all students of a certain maturity to the same sets of exercises and homework without accounting for each individuals' past experience or behavior. Logo allows children to build upon concepts already understood by the child. "Even though educational topics, examples, and subject matter structure and sequence still need analysis are careful design, there is a broad consensus that they should begin with the knowledge states of the learner, and build from there" [Pea, 14].

- Example: Children are traditionally taught composition of shapes based on knowledge of algebra or geometry. Drawing these shapes, however, only requires understanding of position and motion [Mozes, 9].

2.3.4 Logo as a feedback tool

Logo supplies immediate feedback and allows for the most primitive level of debugging. Whereas debugging is traditionally viewed as a complicated, mature process, Logo introduces debugging to children at a level they can understand.

- Example: If the child is attempting to create a box but the turtle instead draws a triangle, he can immediately discover the problem by watching the turtle's steps and adjust the program to correct for it. The debugging process is intuitive because the child can "become" the turtle and trace the steps, equivalent to examining his own logic.

Enabling children to examine their own thought processes and providing immediate feedback to allow them to explore the validity of their design leads children to improve basic problem-solving skills.

2.4 StarLogo

StarLogo expands upon Logo to create an environment where multiple turtles can be simultaneously controlled and programmed. Whereas Logo allows control over only a single turtle, StarLogo can support hundreds or thousands of objects acting in parallel. In addition, StarLogo supports *patches*, which are active elements that comprise the environment. While an environment is usually considered a passive space in which objects interact, StarLogo patch environments can also assume a considerably active role. Patches can contain memory, react with turtles or other patches, and otherwise mutate over time.

- Example: An oil spill might be implemented through a patch environment. Each patch could "spread" oil to adjacent patches.

- Example: To simulate a forest fire, patches might represent grass and react to nearby fires by changing color and become burning grass. Likewise, grass patches adjacent to burning grass would become burning grass themselves. The burning would be a time-dependent phenomena where the grass would eventually stop burning, becoming dead grass, and possibly restore to normal grass after some further time.

- Example: In a simpler case, patches might remember how many turtles have traversed them and set some time-dependant brightness to reflect this.

The additional capabilities of StarLogo in terms of patches and multiple turtles allow new ideas to be conveyed. Some of these will be discussed below.

2.4.1 Emergent behavior

StarLogo can be used to simulate emergent behavior in nature. Many large systems behave as if governed by a central control, whereas in reality each component is simply responding to the others in a limited, predictable manner. Emergent behavior is the seemingly organized, higher-level behavior that arises from many individual interactions on a lower level. Here is a series of examples that illustrate this phenomenon.

- Example: Ant colonies find and retrieve food efficiency. To the outside observer, it may seem that ants have an organized means of scavenging food. Perhaps a central ant queen or worker directs the actions of the colony toward a food source once one is located.

In reality, ant colonies forage for food by searching randomly and leaving a time-dependent pheromone trail along the return path if they are successful. Other ants are in turn attracted to existing pheromone trails and follow the path of any they find. Eventually, as increasing numbers of ants locate the food source, a strong pheromone trail is developed and the majority of the ant colony will follow this trail until the food supply is exhausted.

To the outside observer, it may appear that some sort of centralize control was "guiding" the ants to the food sources. In reality, however, the simple behavior of each ant in the decentralized system causes the emergent behavior to arise.

- Example: Flocks of birds align themselves in highly-organized V-shaped formations. An observer might believe that birds arrange by aligning themselves in accordance to a leader in the forefront. Perhaps each is assigned a position relative to the leader, similar to the method by which armies and battalions are organized. Surprisingly, birds "naturally" arrange themselves into regular patterns when each bird follows a simple set of commands to respond to its environment based on the proximity of birds in its immediate vicinity.

Whereas observers may believe birds arrange according to predetermined ranks and positions, the actual formation of flocks is a behavior that emerges from each individual bird obeying the same set of basic instructions.

- Example: Colonies of termites form very complex, regular mounds which would seem to indicate the management of some centralized control mechanism.

Once again, the highly organized mounds arise out of simple behavior by each termite.

- Example: Examples of emergent behavior also appear in society and economics. Macroeconomic theory is the study of the results of collective behavior from individual companies and systems. Investors recognize that highly regular, reasonably predictable stock trends appear in the overall stock environment through the collective actions of millions of buyers.

2.4.1.1 The importance of studying emergent behavior

Emergent systems are interesting simply from the perspective of understanding natural processes. Studying natural phenomena and the behavior patterns of animals poses a fascinating puzzle for biologists.

Exploring decentralized organizations can also aid in understanding the effectiveness of our own human-designed control systems and the structure of our societies. Sometimes centralized authority may be either unnecessary or even less efficient in systems. The role of decentralized systems in our society may be understressed and underutilized.

- Example: One group of students was asked to explore traffic jams [Resnick3, 40-41]. Each car was programmed to obey a simple set of instructions to accelerate unless another car was immediately ahead, and slow down if a radar trap was detected. This arguably is an accurate enough depiction of actual driving behavior on highways. Traffic jams formed in the vicinity of radar traps as cars slowed down and bottlenecked highway flow. This is an example of a *central* object, the radar trap, determining *behavior* (traffic jams) in the *system* (highway). However, after the radar traps were removed, traffic jam situations still arose regularly. The completely decentralized behavior of each car in the highway in terms of interaction with other cars continued to cause patterns of overall behavior to emerge in the overall system. This somewhat unexpected conclusion is the basis behind the phenomenon of emergent behavior.

2.4.1.2 StarLogo and Emergent Behavior

Arguably, each individual termite, ant, or bird can be thought of as obeying a simple set of "code" i nature. StarLogo enables simulation of emergent behavior through the creation of hundreds or thousands of objects with similar or identical instructions. StarLogo has successfully depicted termite mound-formation patterns, ant foraging behavior, traffic jam phenomena, forest fires, and countless other examples of decentralized behavior.

2.4.2 Mathematical phenomena

StarLogo can also solve probabilistic computation through its ability to simulate different "tries" with many turtle objects. A wide variety of mathematical and geometrical concepts can also be explored.

- Example: In traditional Logo, a turtle might draw a circle by successively moving forward and rotating at predetermined rates. Using Starlogo, one might imagine starting a number of turtle objects in one position, each facing outward at a random direction, and causing them to move forward a certain number of steps. In this case, the turtle objects would form the circle by "becoming" the circle themselves. Other types of geometric puzzles can be explored using related methods. [Resnick3, 36]

2.5 Network Clubhouse

Network Clubhouse expands on StarLogo by allowing multiple users to cooperate in the same shared virtual space. While traditional StarLogo simulations arrange interactions between many objects with similar characteristics, Network Clubhouse can additionally enable each user to define a unique object with appropriate behaviors.

StarLogo facilitated convenient creation of many identical turtle objects. However, it might be impractical or tedious to, for example, ask a user to create many different varieties of fishes to simulate an ocean ecosystem. The types of objects in this environment might not only significantly differ from each other, but each might additionally obey a more complicated set of rules than simple termites or the other types of objects described previously.

2.5.1 Some example applications

2.5.1.1 Overview

The Network Clubhouse environment is fairly general and can potentially support numerous types of applications and projects. This section will describe some of the potential uses of Network Clubhouse. Note that though these can be implemented using Network Clubhouse, a more important issue concerns to what extent each matches with the goals and motivations of the Network Clubhouse.

2.5.1.2 Emergent behavior

StarLogo studies in emergent behavior can be replicated using Network Clubhouse. One potential application of Network Clubhouse might be to simulate a "virtual fishtank." In this type of complex ecosystem, each participant would be motivated by a specific but different set of tasks and goals. A predator such as a shark might be programmed to attack smaller creatures. Fish might be programmed to jointly form schools (with similar commands as those used to program flocks of birds, for example) and avoid predators. Educa-

tors could then challenge children to observe the types of emergent behavior that would arise in this sort of environment.

2.5.1.3 More traditional MUDs

Since Network Clubhouse is a general tool for allowing users to coexist in a shared environment, it is possible to also encourage the types of interactions that are typical in traditional MUDs. For example, children can log into a dungeon or castle-like environment which contains logic puzzles or tasks which require group communication. Since one of the goals of Network Clubhouse is to encourage teamwork and basic cooperation, any sort of environment which is centered around this sort of behavior would be appropriate. Note that this would be different from MUDs such as fantasy roleplaying environments where children would primarily interact with computer-controlled agents in the dungeon, or adventure-type games where children would individually solve logic puzzles.

2.5.1.4 General interaction

In a more broad sense, encouraging communication is something which will be beneficial for children. As described earlier, one of two main goals of the Epistemology and Learning Group involves building community. If "meaningful" interaction can be encouraged in Network Clubhouse, then this sort of development will be a natural outcome.

2.5.1.5 Role Playing

I should mention role playing as a simple extension of the previous topic. Role playing arguably encourages social development and creativity. Since Network Clubhouse allows persistence of characters between logins, long-term, role-playing environments could be supported. The appropriateness of role playing relates to a broader issues of whether children's turtle objects should represent themselves or be merely objects to control. This issue will be discussed in some further detail below.

2.5.1.6 Creativity

Network Clubhouse can support a diversity of environments and scenarios. Children might be encouraged to build their own Network Clubhouse environments. Currently, our focus has been on some level of adult supervision and preparation of the Network Clubhouse world; however, in the future, it is not difficult to imagine providing environment-building tools for enable children to author their own environments. These would allow the children to exercise their imagination and presents a challenge for them to create models which are attractive enough that other children would want to explore.

Projects which are publicly exposed and receive continual feedback often are more useful and interesting than those that do not. This is a natural social outcome, possibly from both to the pressure of having others scrutinize the work and the additional motivation to create an impressive project in comparison to others. A community such as Network Clubhouse which allows children to view and react to each other's creations will undoubtedly encourage more thorough, well-developed, and meaningful projects.

As noted above, Network Clubhouse environments are fundamentally different from traditional MUDs. The types of environments created by children would be focused on user interaction and construction instead of merely solving challenges posed by the environment. Children would create environments which would allow other children to build and model behaviors, not design a complete environment which children could merely interact with.

- Example: A child might create a Network Clubhouse MUD of a schoolhouse model. Other users would be able to serve the roles of students and teachers, and users would work together to create a learning environment. Children might create tools in the environment such as chalkboards or homework. On the other hand, a traditional schoolhouse MUD might instead involve computer-controlled teachers and principals. Users might be challenged to earn the best grade through interaction with the teachers and other students. These two models differ fundamentally. In the first, children interact with each other primarily and explore relationships and methods to evolve a learning community.

They build into the environment and create behaviors for their objects. In the second, students react toward an individual goal in relationship with computer-controlled objects and other users which are designed to encourage or challenge that goal in specific, generally well-defined ways. They would each control a single object and their success and failure would be governed by the success of their representative in that environment.

Chapter 3

Application design issues

3.1 Overview

The most central issues in Network Clubhouse involve its high level design. The actual implementation challenges are standard and not unique, or at least not as interesting from a research standpoint. However, since we are exploring a new type of application to encourage a new type of learning, the specific design decisions behind Network Clubhouse should be addressed. Network Clubhouse contains numerous high level design decisions and philosophies; some of these follow naturally from its nature as a networked graphical MUD, while others arise specifically because of Network Clubhouse's focus as an educational tool.

3.2 Appearance

3.2.1 Overview

The Network Clubhouse client consists of a graphics window, command line, chat line, and function definition window. The graphics window displays the current environment, the command line allows users to send commands and receive debugging information, and the chat feature enables users to communicate with each other.

3.2.2 Graphics verses text environments

There are many advantages of using a graphical environment verses a text-based one. Both visual and textual media are used in our society; for example, books convey certain types of information better than movies and both thus exist in our society. Network Clubhouse will clearly benefit from using a graphical environment, however. Some of the reasons will be discussed below.

3.2.2.1 Visual appeal

Graphics environments are more visually appealing. Users are presented with a semi-realistic image of their objects and can interact in a space that is visually represented instead of described by text.

3.2.2.2 Information

It is said that "a picture is worth a thousand words." Graphical environments more easily depict complex environments. It is much easier to draw a picture of a chess board than to explain the position of each piece, and the picture is easier to relate to than the text.

3.2.2.3 Promoting creativity

Children generally can be more creative in graphics environments. While most young children have not yet mastered grammatical skills well enough to be extremely creative with text, children actively imagine in pictures and drawings, and a graphical environment entertains this.

3.2.3 Graphics view

Currently, the graphics window is a top-down view of the gridworld, although in the future some notion of depth and a more 3D-like view may be implemented. The traditional StarLogo model consists of a 2-D top-down view. The choice between a 2-D or 3-D representation is an open issue; a design decision needs to involve addressing the purpose of Network Clubhouse and the role of graphics in the application.

3.2.3.1 3-D view

3-D views, whether as "cockpit" views as in games such as Doom and Descent or more traditional views as represented in "My Make-Believe Castle," allow some semblance of height to be incorporated in the display. This also enables an additional degree of freedom in manipulation.

3-D views are also more generally more visually appealing then their 2-D counterparts. Since Network Clubhouse is aimed toward children, it may be more important that the application seem visually appealing than if Network Clubhouse were a utility for adults.

3.2.3.2 2-D view

A 2-D view may be more desirable in an application which is aiming to use graphics to communicate an idea instead of using graphics for the sake of being visually appealing. In StarLogo, the 2-D model most clearly allowed the behavior of the system to be communicated for each simulation. It was not necessary to make visually-appealing turtle objects or a fancy-looking environment; the simple representation of the environment allowed the graphics to be used as a tool for presenting ideas.

3.2.3.3 Conclusion

It is unclear whether we should adopt a simplistic 2-D model to allow for the graphics to serve as a visualization tool, or use a more visually-appealing 3-D model with fancier graphics to create a more interesting-looking tool. The resolution of this issue will depend largely on what we view as the primarily goal of Network Clubhouse. If we were simply creating a model graphical MUD, then sophisticated graphics and animation would most closely achieve that end. If we were creating an application to study emergent behavior and mathematical concepts, then a simpler 2-D graphics model would be desirable.

3.3 User control

3.3.0.1 Overview

Many issues related to the nature and extent of user control need to be considered. This section will be devoted to the relationship between the users and their turtles and the methods by which users create and possess turtles.

3.3.0.2 Persistence

Most MUDs support a notion of persistence, where users' objects are maintained between sessions. This allows the objects to grow and evolve as they interact with the world. Persistence is a central attribute in MUD games where the goal is usually to improve one's character in certain ways. Network Clubhouse follows this philosophy by allowing users to attach a persistent username to their turtle objects. When a user logs in initially, a new turtle is created and assigned default starting attributes. The user can then customize the turtle by defining shape, color, behaviors, and other attributes. He can also create private, higher-level functions which are available only for his turtle. When the user logs out, this information is maintained by the server and retrieved upon the next login so that the exact state of the turtle is returned to the user. Thus, his previously-defined higher level functions will be recorded as well as information on his turtle and environment.

Naturally, users should be given the option of self-destructing their own turtles and restarting. This is basically equivalent to the user logging in under a different username to assume control of a new turtle.

3.3.0.3 Activity between sessions

The user can specify commands for the turtle to be processed even after he logs out; for example, a turtle can be made to endlessly walk in circles even when the user is not active.

One issue concerns whether the user can elect to remove his turtle from the world when he is not present. Otherwise, the presence of hundreds of idle turtles may clutter the environment and distract users from distinguishing and interacting with active users. A common room serving somewhat as a locker might be incorporated where users could store their turtles when they are inactive.

• Example: In our Virtual Fishtank, a user might create a school of fish which continues to

swim in the ecosystem. The fish would be programmed with a set of behaviors which guided them to interact with the environment. There is no reason why the fish should disappear when the user logs out; instead, they should remain as a permanent part of the ecosystem so they are available to interact with other users. The owner might desire to inactivate his fish after his session, perhaps if their behavior is not yet completely debugged. In this case, he could transfer the fish to a common room and retrieve them upon his next login.

3.3.0.4 Controlling other users' turtles

For obvious reasons, users probably should not be allowed to arbitrarily control turtles created by other users. As characters are often viewed by users as their "property," even among adult communities, some respect of ownership should be followed. Users can inadvertently affect each other's turtles adversely or even engage in destructive behavior (e.g. self-destructing another user's turtle) if this notion of privacy is not enforced.

Protection can be implemented by associating a password for each username. Complicated encryption and decrypting methods probably do not need to be implemented since users will be primarily children, who are unlikely to possess any real software or networking experience.

Although users should not be given access to each other's turtles, they might be allowed to share a common set of public turtle objects in the environment. However, to prevent potentially abusive or destructive behavior, these objects might be restricted in the types of operations they could perform.

- Example: In our virtual fishtank, users should collectively be allowed to alter certain objects, patches, or characteristics in the environment. This might include the flow rate of water in the ocean or growth rate of algae in the environment. However, users should not be allowed to enable to, for example, make the algae eat users' objects, or make the water poisonous.

3.3.0.5 One or more turtles

In traditional MUDs, users generally control only one object in the environment. This rationale could either be followed by Network Clubhouse, or Network Clubhouse could more closely model StarLogo in allowing users to control multiple turtles.

Advantages to a single turtle

> Not only does restricting a user to a single object simplify the amount of information the user needs to control at once, but it also encourages or forces the user to interact with others in order to accomplish tasks instead of simply controlling his own set of creatures. In addition, since there are multiple "rooms" or environments within the server, it would be difficult to control or represent objects simultaneously if they are in different rooms. Finally, it is more intuitive for users to control a single object. Since users and turtles are persistent and therefore can grow and evolve over long periods of time, the user might identify himself with one object to follow that object's evolution.

Disadvantages to a single turtle

> Restricting users to controlling only one turtle poses some limitations on the potential usefulness of Network Clubhouse. Some StarLogo applications and mathematical phenomena involve hundreds of objects, and it would be impractical to require that a hundred users to be present to control that many objects. If a single user were allowed to control many objects and simultaneously program them with a simple behavior, this would complement the StarLogo paradigm for creating simulations.

- Example: In the Virtual Fishtank, since many different types of creatures would need to be represented, each species would probably be designed by a different user. However, multiple numbers of the same species with identical behavior should certainly fall under the same user. In this way, schools of fish, schools of sharks, and groups of starfish could

each be controlled by one user.

Network Clubhouse simulations will generally allow for multiple turtle objects since the paradigm is not to have a turtle "represent" the user, in which case a one-to-one mapping is appropriate, but to allow turtles to represent objects other than the user.

3.4 Sharing functions

3.4.1 Overview

There is generally no function sharing between users. Each user's higher-level functions are only accessible by himself. If users wish to share functions, they must communicate each others' function definitions through chat or other means. There is currently no built-in mechanism in the normal case to allow users to share functions in the general case. The reasons will be described below.

3.4.2 Simplicity

As potentially hundreds of children use Network Clubhouse and define higher level functions, it would be unreasonable to expect users to be able to remember their own functions when a library of hundreds is displayed on their function definitions window. Users will quickly lose track of which functions they defined and which were defined by the other users.

3.4.3 Frequency of use

Users generally design higher-level functions as shortcuts for performing a series of commands. Therefore, users really only need to have access to shortcuts that they have defined and are likely to use instead of hundreds of different shortcuts which they would be unfamiliar with.

3.4.4 Programming experience

Since one of the goals of Network Clubhouse is to introduce children to computing and programming languages, it follows naturally that students should be encouraged to implement their own functions in order to gain the most programming experience. This promotes the users to understand how their functions work instead of only understanding the effects of functions borrowed from other users.

3.4.5 Different connotations

Function names may mean different things to different users. The command "jump" may convey a wide range of behavior depending on the nature of the user or the nature of the object. Therefore, it is more suitable to require users to each write their own "jump" command if they desire that behavior.

- Example: In our Virtual Fishtank, a "jump" command for a frog may involve finding an adjacent lily pad and directing the frog toward that with some horizontal and vertical trajectory. A "jump" command for a flying fish would involve a completely different behavior, motivation, and destination, since flying fish require tremendous energy to leave the water and usually do so only to catch food.

3.4.6 Conclusion

We will allow children or supervisors to submit certain functions into a standard library which can be accessed by other users. This might be useful if children are simulating an ecosystem where each object needs to react in a standard and predictable fashion to certain inputs. Having each user follow the same functions will guarantee a certain level of uniformity. The creators of the environments can provide some standard functions for children to explore and build on. Arguably, designing functions from scratch may be an imposing challenge for some children especially in light of some of the more complicated behavior that is needed in projects such as Virtual Fishtank. Allowing children to explore and modify existing code written by others would be a useful learning experience.

3.5 Interface

3.5.1 Overview

It is possible to design application interfaces in many different ways. Here are a couple that are used by Network Clubhouse.

3.5.2 Direct manipulation

Some simplistic graphical environments restrict user control of objects by use of the mouse or simple buttons. While this type of interface is generally easy to understand and use, it is insufficient for the type of flexibility and control which is needed for an application such as Network Clubhouse. Children need freedom in Network Clubhouse which is not available through a button and mouse interface. Therefore, although some buttons and mouse control are provided, most of the manipulation to Network Clubhouse will be accomplished through other means.

3.5.3 Command line

A text command line option affords the most flexible type of control for users. Adventure MUDs often support simple parsing and interpretation of text sentences. One possible objection to text interfaces is that the control is not as direct and immediate; for example, the precision of control over a turtle will depend on the user's typing speed and fluency. However, since much of the interaction between users and turtles should consist of defining behavior and then watching that behavior evolve, the frequency of commands should be low enough to make a command line interface sufficient.

- Example: Games such a DOOM interactively respond to single keyboard strokes to control character movement and other manipulations. The type of control that is needed in DOOM only requires simple interaction through the keyboard. It would be highly inappropriate to supply a command line interface which would process complete grammatical sentences such as "*forward 10*", "*rotate right 20*", etc., because the attainable frequency of command input would be intolerably slow.

As with all standard programming interfaces, the command line allows users to enter arbitrarily long strings of text and receive feedback in the same window. Another window is available to conveniently display user-defined functions. Users can scroll through both the command line interface and the function window to retrieve the history of their commands.

3.5.4 Rules

Another mix between direct manipulation and command line interfaces is through the implementation of *rules*. User can define rules which would be associated with objects. A user can then select a turtle, open it, and place a rule object inside to endow the turtle with some set of behaviors.

- Example: Users might create a banana object which contains some repeating Logo code to make objects "slip" occasionally as they move. A user might then insert a banana into his turtle object to make it exhibit this behavior. Or

3.6 The Network Clubhouse Language

3.6.1 Overview

One of the larger research issues concerns the type of language environment which children should use to control their turtles. Language designers work to produce languages that are powerful but easy to understand. The balance between these issues and their relationship to the intended audience is critical in determining the success of languages.

- Example: Assembly language is a powerful, general tool, but it is primarily used by expert programmers who are able to overcome the lack of readability and elegance in assembly programming. C is a more traditional language which incorporates readability with power, but many programmers are unhappy about the need for memory allocation and pointer manipulation which arguably should be absent from higher-level languages.

Since the audience of Network Clubhouse will be young children, the language needs to lean more toward ease of use than power. Languages for children already exist and

Logo has especially proven effective. The Network Clubhouse programming model closely follows Logo.

3.6.2 Designing a suitable language

There are innumerable issues conerning how to make a language usable, and one could easily devote an entire thesis study to this single issue. Fortunately, since Logo is already available as a foundation to the Network Clubhouse language, most of the discussion below will address why certain decisions were made in Logo or how built additional modifications have been supplied on top of the original language.

3.6.2.1 Parameter passing

Variables are passed into functions using the *pass by value* methodology, where a copy of the argument is given to the function. The original variable cannot be changed by operations on the local variable. This is consistent with variable passing in most traditional languages. Certain primitives use call by reference to permute their arguments. For example, *setvar*, or =, performs a permutation operation on its first argument. However, all user-defined functions operate on the call by value principle.

3.6.2.2 Global variables

Any variable defined within any context becomes a global variable. This includes variables defined within function calls. The "let" statement allows local scoping, while most variables defined by children will probably be global in nature to each object. There are several justifications behind this design.

Global variables are easier to maintain and remember.

> Global variables avoid the usual complications with scoping and keeping track of which variables are defined in what context. It is probably sufficiently powerful to allow children to assume that each variable they introduce maintains its value and is accessible for the remainder of the session.

Global variables allow modification of variables outside the function.

Since function calls can only report one value, programming languages often resort to extraordinary means to allow a function to modify more than one value. Pointers are the standard means of causing a function to modify more than one variable in C. It is unreasonable to expect that children would be able to understand pointer manipulation and call by reference. It is much easier to permit global variables to be defined within functions to allow those functions to pass back certain state other than their simple unary output.

3.6.3 Specification

The Network Clubhouse language closely resembles Logo. A few characteristics of the language are discussed here.

3.6.3.1 Variable definitions

Variables are introduced via a *make* command. Variable names can either be created and assigned through the syntax *make <variable> <value>*, or introduced and later assigned a value. Variable names are identified by a *:* in as the first character.

>make :x 10

>make :foo

>setfoo 10

3.6.3.2 Function definitions

Function definitions are defined via the *to* command, followed by the name of the function, the number of arguments, and the body.

```
>to foo :a :b :c
 fd :a
 rotate :b
```

```
fd :c
output :c
end
```

A function with an *output*, or *return*, statement is considered a "reporter", meaning that it should yield a return value along all paths.

3.6.3.3 Function calls

Function calls are invoked by indicating the function name followed by the number of expected arguments.

```
>foo :a :b :c
```

3.7 Multi-User Environments

3.7.1 Overview

Collaborative environments allow users to cooperate in building projects and solving problems. They have assumed an important role in our society throughout the development of civilization, and cooperation is inherent in many daily activities such as meetings, project development, business administration, and more mundane activities. Collaborative environments also are finding a growing role in our work environment since the introduction of computers. Video conferencing is an example of a feature which has surfaced recently, and other applications for business, recreation, and education are now available through computer networking.

As described before, Network Clubhouse draws some key benefits from being a collaborative environment, and its multiuser support is the main feature which distinguishes it from predecessors such as StarLogo and Logo.

3.7.2 Issues

Collaborative environments contain several key issues that need to be addressed before they can be useful. The more major ones include time, location, and consistency

[Mozes, p. 12]. This section presents an overview of these issues and their relation to the Network Clubhouse environment.

3.7.2.1 Time

Collaborative work can be performed either synchronously or asynchronously, depending on the degree of real-time communication that is involved in the cooperation. Synchronous collaboration involves situations where users are active at the same time.

This is the more commonly-used sense of collaboration involving users in the same meeting room or environment.

- Example: Some projects are dependent upon synchronous communication. For example, construction projects primarily involve synchronous work because workers need to actively help each other in order to accomplish certain goals. Placing a beam on a building requires one worker to operate a forklift, another to guide the operator, and several to secure the beam as it is being placed.

Asynchronous collaboration allows progress to be accomplished jointly but without real-time communication. This allows a greater flexibility because users are not constrained to the same time schedule, but is sometimes less effective if frequent communication is required.

- Example: For example, a group of people might be working on the same software project, where communication is infrequent and involves such medium as email. Each person would be responsible for an independent, modular piece of code; or, each group of people might implement a stage of the software life cycle such as writing requirements, developing code, or performing testing. Although the final project is the result of the collective work of each individual, the interaction during the design and implementation may have been minimal.

3.7.2.1.1 Network Clubhouse and Time

Network Clubhouse allows both types of collaboration, although it will mostly be used in a synchronous fashion. Users will log in simultaneously and communicate real-time through the use of the chat feature. They will work together in parallel to solve tasks, communicating and providing feedback during the process. Asynchronous collaboration is

also enabled, however. Users can log in and work on sections of the projects while others are inactive. The notion of persistence--that users' turtle objects remain in the environment even when they are not actively logged on--facilitates asynchronous work.

- Example: Several users might each be involved in a different aspects of a problem until they are all ready for integration. In our virtual fishtank, each user could conceivably complete his own fish or school of fish independently. After all the objects have been developed, the users can then log in simultaneously to view the results of their work and engage in debugging in a synchronous manner.

3.7.2.2 Location

Location simply refers to the placement of objects in the environment. As an environment might have many rooms, location will presumably affect the ability of users to communicate with each other.

3.7.2.2.1 Network Clubhouse and Location

Location has two implications in Network Clubhouse. First, the physical location of the users will affect their ability to communicate. If users sitting in the same classroom, they can use verbal and gestural communication to handle collaboration. Second, the location of turtle objects in the virtual world will affect their ability to communicate. Turtles in different rooms should not be able to communicate through the default of broadcasting messages to all users in "hearing range." Location will thus provide a means of separation and modularization in Network Clubhouse, allowing several groups of users to work on different projects in separate rooms without the disturbance of overlapping communication.

3.7.2.3 Consistency

Consistency is a technical issue which arises in networked applications. Users should ideally see an identical representation of their environment as all other users, and these in turn should reflect the server's model of the environment. Consistency refers to the how

accurately the screens of participants actually match. Graphical MUDs generally focus on creating a duplication of the surroundings for each user, enabling screens to reflect the same environment.

3.7.2.3.1 Network Clubhouse and consistency

Maintaining consistency is a difficult but fundamental issue to address in networked applications. Network Clubhouse does not guarantee consistency but incorporates measures which should provide a reasonable amount of accuracy. Networking issues such as dropped packets, corrupted packets, out of order delivery, or delayed data transmission are tolerated as an inherent source of inaccuracy. Potentially different processing speeds on computers and differing processing loads will impact the rate at which screen updates are maintained and thus will also affect the consistency between screens. However, since updates occur relatively frequently in comparison to the speed in which objects in the environment move, minor discrepancies or inaccuracies in the data should not lead to intolerable problems.

Chapter 4

Language issues

4.1 Overview

Network Clubhouse was primarily implemented in JAVA. This section will describe the design decisions behind using JAVA.

4.2 JAVA

4.2.0.1 Overview

JAVA is currently at the forefront of programming excitement. JAVA programming guides can be found in almost every software company office. The network community may view JAVA as a new revolutionary tool for communicating information, but the programming community has the burden of exploring the feasibility and limitations of JAVA. The Network Clubhouse project attempts to address some of these issues. This section will explain why JAVA was chosen as the platform for the client.

4.2.0.2 The JAVA paradigm

Netscape is seen primarily as a means of communicate information to users. Using web tools, users can make documents widely available for others to access. Netscape and Mosaic provide a means for users to use technology to improve information accessibility.

JAVA is hailed as an addition to this capability. It allows designers to incorporate animation, thereby allowing pages to convey additional information through dynamic, interactive art.

- Example: One simple HTML page teaches juggling using an animation with two hands and a few balls. Instead of simply presenting static images of successive positions of balls, the document can illustrate the motion of the balls and hands to convey more useful information.

4.2.0.3 Network Clubhouse and JAVA

While JAVA is primarily approached by the technology community as a tool for conveying information, Network Clubhouse focuses on a different paradigm where JAVA is seen as an interface for building applications. JAVA presents basic graphics functionality which can allow users to create widely accessible applications. Since JAVA is theoretically platform-independent, users can create public, JAVA-based applications which would be accessible from any Netscape browser. This eliminates the need to port software across platforms or require users to purchase application software.

Network Clubhouse is the first project from the Epistemology and Learning Group to explore the usefulness of building JAVA-based applications. Network Clubhouse addresses the exciting possibility of creating publicly available environments which users from anywhere in the world could access.

4.2.0.4 JAVA and research

JAVA is also seen as a research area. Since JAVA is fairly new, it undoubtedly contains bugs and other problems which might only surface when it is used to implement a large, ambitious project such as Network Clubhouse. Working in the JAVA environment is a means for the Epistemology and Learning Group to explore the potential of this new language and hopefully provide meaningful input to JAVA designers.

4.3 Cocoa

4.3.0.1 Overview

Cocoa is an interpreted language written on top of Java. It was developed primarily by Brian Silverman with the help of Andrew Begel. Although JAVA is powerful tool for creating web documents, it is also only understandable by expert programmers and software

engineers. Cocoa was designed to allow children to use a simpler model of Java to create their own dynamic web pages.

4.3.0.2 Network Clubhouse and Cocoa

The client was implemented partially in Cocoa. Cocoa suffers limitations from speed and performance, but is also a simpler language in which to understand, debug, and design programs. Because of unacceptable performance problems, much of the Network Clubhouse Cocoa code is being reimplemented in the underlying JAVA.

Chapter 5

Technical Design Background

5.1 Overview

This section will describe some technical background necessary for understanding the underlying design and implementation of Network Clubhouse code.

5.2 Network strategy

5.2.1 Overview

Network communication was accomplished through TCP and UDP connections. TCP is a guaranteed package delivery protocol which ensures that packets which are sent over the internet eventually reach their destination. UDP is a mechanism which allows for packet loss and packet dropping to achieve faster performance.

5.2.2 Unreliable and Reliable protocols

We recognized that some of the information passing between the client and server needed to be guaranteed, while other data could be transmitted unreliably. Important data included log in requests, commands, and chat requests. There all needed to be guaranteed for fairly apparent reasons:

5.2.2.1 Login

Login should be reliably communicated. The server needed to reliably handle log in requests where loss or corruption of information at either end would create an inconsistent state. The client could be handed erroneous information about the environment or an inaccurate version of his function definitions. Such errors could not be otherwise detected and handled; a reliable data transmission protocol needs to be used. In any case, since login

does not need to be a relatively fast operation, the benefits of guaranteed transmission could be afforded while sacrificing a tolerable performance drop.

5.2.2.2 Commands

Commands should be reliably communicated. At first it might seem that the unreliable UDP mechanism would be sufficient for handling commands. Any erroneous command would either be ignored by the server or produce undesired results. In either case, the effects would be noticed and correctable by the user. This reasoning fails on several accounts.

First, unexpected behavior may have a tendency to frustrate and confuse the user and also make the application seem unreliable.

Second, certain types of unexpected behavior may be irreparable. (e.g. if a forward request is somehow corrupted into a logout request).

Finally, subtle errors or corruption in data may remain unnoticed to the user for an extended period of time (e.g. if a variable value is corrupted but not immediately used).

A TCP protocol is the more desirable approach because commands are relatively infrequent and this application needs to focus more on correctness than performance with regard to command processing.

5.2.2.3 Chat

Chat commands should be reliably communicated. The correct implementation choice for the chat communication was less obvious. Packet loss and corruption is not as important an issue because receivers can easily ask for a resend, or users will usually notice if their messages are not apparently received by their intended audience. This phenomena occurs and is handled in both email and MIT's Zephyr service, where unreliable means are

used to transmit real-time messages across the network to specified users. However, while UDP would provide the necessary functionality, a TCP implementation was chosen instead because--once again--dialogue is predicted to be rather infrequent, and there is no need to achieve a critical speed for processing chat requests. Even a delay of a several seconds is tolerable.

5.2.2.4 Moving to new rooms

Changing environment information should be reliably communicated. When the user enters another room or domain, the complete specification of that environment (e.g. objects in the room and other features) is transmitted. Since this information is only conveyed once (i.e. upon entry), the specification of the environment needs to be transmitted correctly to prevent the user from possibly being subjected to an inaccurate state for the duration of his stay in that room.

5.2.2.5 Other

Other categories of information either required a faster communication mechanism than TCP or could afford an unreliable transmission mechanism. The most obvious candidates are user and board updates. Board updates need to be communicated relatively frequently to reflect changes in turtle position or environmental changes. Because Network Clubhouse should send at least three or four updates per second in its final form, a mechanism which will allow the least amount of effort in terms of package and delivery needs to be used. Although packets may be lost or corrupted in the process, the frequency of board updates ensures that errors would be largely unnoticed so long as they occur relatively uncommon.

5.2.3 TCP

TCP communication operates in two stages, a *connection* stage and a *communication* stage. A server first opens a socket bound to a specified port. The server and client must

then establish a connection through that socket, creating a new socket. Further communication is then handled by this socket, and the original socket remains open to accept new connections if desired.

5.2.4 UDP

UDP communication is *connectionless*, where the server listens on a specified socket for incoming messages without first establishing a dedicated connection with senders. Whenever a server receives a packet, information on the source of that packet is also included, allowing the server to readily send data back to the client. The server can receive and send data from an arbitrary number of hosts through a single socket.

5.2.5 Byte order

One frequent issue in network communication involves the byte order of packet information. When communicating 32-bit integers, for example, it is possible for the receiving end to either interpret the first 16 bytes as the high or low order bytes of the integer. Therefore, some sort of mechanism needs to be designed to ensure that information is transmitted and interpreted in a consistent manner. One obvious way is to split integers into high and low order bytes and send those separately. The receiving end will then expect the data to arrive in a predictable manner. Dividing an integer into high and low bytes is a fairly standard process, where the high order bytes are "*i << 16*" and the low order bytes are "*i && 0xFFFF*".

We have so far avoided this problem by only transferring unsigned characters, which are single bytes. However, as data becomes more involved and complex, the need for integer communication will become necessary.

5.2.5.1 Error handling

Several types of errors need to be anticipated. First, packets can be dropped and thus never received on the server side. Second, packets can become corrupted and deliver inac-

curate information. Finally, other applications can potentially interfere by sending bogus information to the server.

Most of these problems can be handled simply by establishing an agreement between the client and server as to the type of information that is expected. The client heads each packet with an identifier byte signifying that it the information is indeed client information, or the server will ignore or possibly log the unidentified message.

To ensure that packets are timely, a nonce is incremented and sent with each update. Outdated packets; that is, packets which arrive out of order, can be easily detected and ignored. Note that the same nonce is not sent to each client which is logged on; when the client logs in, a server process is forked which is dedicated to sending updates to that client, and the nonce is set to zero.

5.2.5.2 Network breakdown

For a couple of months, we experienced some unknown phenomena that was preventing our network communication from operating. To explain briefly, we initially were unable to receive UDP messages, and then TCP messages.

5.2.5.2.1 UDP

UDP communication functioned from my media lab computer or Athena dialup workstation to any other media lab or Athena workstation. Furthermore, my workstations could correctly receive information from the client PC or from the Macintoshes in lab. However, the server could not successfully send information to the client PC, despite checking that the port and IP address of the return packet were correctly designed. This problem persisted for many weeks, although we unsuccessfully experimented with different capture and send mechanisms. However, it somehow magically solved itself a couple of months later when the client PC suddenly began to receive the data. No changes had been made to the code, so we credit this as a purely external phenomena. However, during the course of

the problem, it became beneficial to design small UDP client and server programs which could send and receive simple messages. Specifically, I isolated the standard networking libraries I had designed into a program which took a port number and machine and opened either a UDP server at that port or a UDP client to the specified machine and port.

5.2.5.2.2 TCP

TCP communication suffered a similar problem where the server program was unable to receive information from the client. Recall that in TCP communication, a connection between the client and server needs to be established before information can be sent. Although the connection phase could be established between the client and server, no messages could be successfully received from the server. Messages were received and processed correctly on the client end, but the other end was not operating correctly. Once again, this problem seemed to solve itself after several weeks without intervention from us.

5.2.5.3 Conclusion

Due to my limited experience with networking, it is impossible for me to determine or even theorize as to the nature of the problems that were preventing proper communication between our client and server. It should be noted that these problems did not occur simultaneously; the UDP problem surfaced and disappeared, and then the TCP problem emerged and resolved itself. I have no advice on how to handle this problem should it occur again in the future and I regret not having the knowledge or resources to explore it further.

5.3 Multiple processes

5.3.1 Overview

The need for a multiple process server was critical. Most network communication reads lock the current process until a read is completed, necessitating either a separate process to continue server functionality or frequent timeouts. Since many clients could be logged into a single server, it follows naturally to have a server spawn a dedicated process to handle each client.

There are several methods to spawn and handle multiple process. I shall discuss each below.

5.3.2 Forks

Forking is the easiest means of spawning a process. A fork request creates an identical copy of the existing process, where the new process is given a duplicate but separate copy of the calling process's address space. The processes are virtually indistinguishable except that a separate process id number is assigned to the forked process.

Disadvantage

> Forking is a straightforward method of creating new independent processes. However, it does not by itself satisfy the requirements of the server because some sort of interprocess communication is necessary.

5.3.3 Shared memory

One way to maintain contact between processes is through the use of shared memory. Creating shared memory segments is not a difficult process, and shared memory is an easy way to enable processes to communicate with each other.

Disadvantage

> At first glance, sharing memory using C functions such as *shmget* and *shmat* is restricted as a privileged process that can only be performed by superuser functions. Therefore, it was impossible to easily allocate memory for sharing between processes.

5.3.4 Pipes

Pipes are another standard means of interprocess communication. Pipes open a read/write stream between processes. Processes can then perform commands similar to file operations through the pipe, such as reading and writing data.

Disadvantage

> Although pipes are certainly useful in many applications, they may become cumbersome as the server begins to handle many clients. Each time an update is performed, the server must notify each client process of the new change by writing to each pipe. Such an action may become intolerably slow as the number of clients increases. It would be useful if some method of simultaneously updating each client were available.

5.3.5 Threads

Fortunately, threads provide an easy means of solving the interprocess communication issue. Threads are lightweight, allowing the server to create many processes without significant overhead. They inherit the address space of the original process, allowing memory to be easily shared between the thread and calling process. Furthermore, C provides various routines for thread management, allowing calling processes to block on threads, cancel threads, or send other sorts of signals.

Threads and calling programs mutually share global variables. In addition, the called thread routine can be passed an argument from the original process. Passing an address allows sharing of the local variables referenced by that address.

- Example: An easy way to simultaneously notify all client update processes that the board has changed is to pass the address of the board as an argument to the update process. Whenever the board is changed in the original server process, the board will immediately change in the client processes.

Thread cancellation provides a convenient method for calling processes to manage threads. For example, if a client logs out, either the threads dedicated to that client needs to be directly notified of the event and terminate themselves, or the server can force a cancel on the threads. While it is more elegant to handle the matter in the former way, sometimes this is not possible (e.g. if the thread is blocked on waiting for input from the user) and the cancellation method is used instead.

5.3.6 Sproc

It is worth mentioning an analogous mechanism to threads that exist for process spawning on SGI's. There is no thread feature in C on SGI's; however, the command *sproc* is provided as a similar alternative. Like threads, sproc creates a new process which shares the same virtual address space as the original process. Unlike threads, processes created by sproc do not allow cancellation and blocking. This section is included in case the server is ported to SGIs in the future.

5.3.7 Threads and Timeouts

Threads can provide a way to mimic the *alarm* feature. Alarms are a convenient means of implementing timeouts for blocking processes such as network reads. Since network reads wait indefinitely until a message is received, alarms need to be set to allow the process to eventually give up and continue if that feature is desired. When an alarm command is activated, a process begins which will signal the calling process if it is not cancelled before a specified amount of time has elapsed. The calling process needs only to set a mechanism for handling the signal, usually involving some variant of goto whenever an alarm signal is detected. Thus, the process can jump away from a command which is waiting indefinitely.

Unfortunately, the standard alarm command on DEC stations suffers from an internal DECthreads problem which is inherent in that architecture. This problem only appears

when an alarm is called within a spawned thread, and it is unsolvable since it is a bug in the architecture itself. Fortunately, thread programming can be used to mimic alarms. When a function is requested to be timed by an alarm, the new alarm command spawns a thread to allow the execution of that function. After a specified time, the alarm process will cancel the thread if it is still active. This provides an equivalent, although less elegant, means of handling timeouts.

5.4 Stacks

5.4.1 Overview

The interpreter is implemented as a stack machine, analogous to traditional machine code interpreter design. Each context is given its own stack frame, and variables values are stored on the stack. Instructions are popped off an instruction stack and pushed onto a data stack, then popped from the data stack when they are needed by a function or primitive.

The use of an instruction and data stack easily allows for scoping and presents a standard, well-known method for machine code interpretation.

5.4.2 Pointers

Various pointers are used to access elements in the stack. These will be discussed below.

5.4.2.1 Instruction pointer

The instruction pointer points to the current instruction on the instruction stack. It is used to determine which instruction is currently being processed. The instruction pointer continues to increment as new commands are inserted onto the interpreter.

5.4.2.2 Stack pointer

The stack pointer points to the current entry in the stack. When data is pushed and popped from the stack, the stack pointer keeps track of the top of the stack.

5.4.2.3 Frame pointer

The frame pointer points to the base of the stack frame. Variables addresses are determined as offsets from the frame pointer.

5.4.3 Frames

Frames are the basic components for local scoping. Each stack frame is a portion of the stack which represents a context. The frames are comprised on several elements.

5.4.3.1 Instruction pointer

Each frame contains an old instruction pointer. This is needed because function calls cause the instruction pointer to jump to a new location (namely, the instruction list of the function). When the function returns and the frame is popped, the old instruction pointer allows the interpreter to proceed with the next instruction.

5.4.3.2 Frame pointer

The frame pointer points to the beginning of the last frame. When the frame is popped, the interpreter references this address to determine the base of the previous frame.

5.4.3.3 Variables

The variables come next. The first variable to appear is variable 0, followed by 1, etc.

5.4.3.4 Data stack

The data stack contains the pushed and popped values of arguments and can grow arbitrarily large. Arguments are pushed onto the stack when they are read, and popped from the stack when they are needed.

5.4.4 Variables and Functions

5.4.4.1 Local Variables

Variables are referenced as an offset from the stack pointer. Variable values appear after the entries where instruction, frame, and stack pointers are stored. Variable numbers are assigned starting from 0, so the variable number simply translates to the offset from the base. A stack entry will also hold the number of variables defined in that stack frame (i.e. the highest legal variable number for that frame). The stack implementation allows various types of scoping to be easily accommodated. Here is a list of options and implementations:

5.4.4.1.1 Lexicographic Scoping

Lexicographic scoping means that all variables defined in a context and strictly local to that context. If a variable is defined in a different frame than the current one, it cannot be legally accessed unless it is a global variable.

The implementation is straightforward. A variable number is mapped to its address and the value retrieved from there. If the variable number is greater than the legal number of variables as indicated in the stack frame information, the variable is considered a global and a lookup is performed in the global stack. The variable is guaranteed to be a legal value because the compiler should check and signal an error if the user attempts to use an undefined variable name.

5.4.4.1.2 Dynamic Scoping

Dynamic scoping means that all variables defined in the current frame or any frame above it (all the way until the first frame, where globals are stored) are legally accessible.

The implementation is somewhat similar to that of lexicographical scoping, except that the interpreter recursively checks through each frame starting from the current one until it finds a legal binding for the variable. If the current frame does not contain the variable, the interpreter looks up the location of the previous stack frame in the stack frame information and recursively checks there for the variable.

5.4.4.2 Globals

Global variables are given a separate, indefinitely expandable stack space. Most languages, such as C, only allow variable definitions at the beginning of functions. Our implementation remove this restriction by allowing variables to be defined dynamically, on demand. This needs to be possible because users will want to define variables as they are needed instead of somehow knowing which variables they will use for a session beforehand. Whenever the interpreter receives a new variable definition, it first checks if this is present in its stack of globals. If not, the interpreter increases the stack to accommodate the new variable and initial value.

5.4.4.3 Functions

Functions can be regarded as a special type of global variable. Like globals, they are given a separate, indefinitely expandable stack space. Each function variable points to a set of instructions which define the function. This instruction stack includes the number of arguments the function is expecting and the body of the function.

5.5 Multiprocessing

5.5.1 Overview

The interpreter is a single process which needs to emulate multiprocessing by handling commands for turtles in parallel. It simulates multiprocessing by looping through a list of active turtles and processing one instruction set from each.

It would have been possible to implement the server as a threaded process, where thread scheduling would automatically implement the multiprocessing. However, this was undesirable for a number of reasons.

Disadvantage

The most obvious problem is that there would be no way to ensure that

users would be given equivalent amounts of processing time. It would be entirely possible for one turtle to move twice for a single move from the other, or for one thread to not be scheduled for a relatively long period of time. If mechanisms were implemented to guarantee a fair sharing of processing time, the implementation would begin to closely resemble the serial process which the single process could implement anyway.

5.5.2 Requirements

The interpreter should be able to yield control to the next turtle at any point and resume processing of any command at any stage. This means that the complete state of the interpreter needs to be saved whenever the process switching occurs in order to not lose any critical information between process switches. This is accomplished by saving such data as the instruction pointer and stack pointer whenever a process yields. When the interpreter resumes on that process in the next cycle, it can recall those values and continue from where it stopped.

5.5.3 Implementation

The processor maintains a list of turtles, their instruction stacks, and their data stacks. It loops through the list and checks if each process has new commands to interpret, and interprets a subset of those commands before moving to the next turtle. After processing each turtle, the interpreter then waits for a specified amount of time before checking for new commands.

5.5.3.1 Reasons for waiting

The interpreter could simply loop through each turtle process and parse commands until none remained, and then continue to loop awaiting new commands. However, this would violate the goal that a certain reasonable speed should be simulated. If the user asks for a turtle to move forward 100 steps, the server should move the turtle at a reasonably

slow pace so its transitional behavior could be noticed by the user. Arguably, if the server were heavily loaded and running at a slow speed, then the length of time to wait between loops would decrease toward nothing.

5.5.3.2 Determining rate of processing

To claim that the server processes a subset of the user commands in each iteration is somewhat vague because the number of commands to be processed has not yet been specified. Some possible proposals are discussed below:

5.5.3.2.1 Process one command for each turtle

This is the simplest method to implement. The interpreter simply reads the next complete instruction, whether it is a function call or math operation or define or any of the other commands, and processes that. It then moves on to the next turtle. This is an attractive solution because it is relatively easy to implement; whenever the turtle processes a command, enters and leaves a frame, or enters a new frame upon entering a frame (i.e. calls a function within a function), then one command has been processed and the interpreter can move on.

Disadvantage

> Unfortunately, this design fails in giving equally perceived time to processes. If a user defines ten variables in a list and then calls a *forward* command, it is unreasonable to expect the interpreter to process only one variable definition per iteration instead of processing the entire set of variables and the forward command. Some more advanced notion of time sharing needs to be designed.

5.5.3.2.2 Achieve a certain speed of turtle change

The most intuitive way to handle commands is to achieve a constant speed for each client. Recall that the server waits in idle for a specified period of time between iterations

in order to create a perception of speed to the user. Therefore, in this implementation, a turtle should be able to make one "move", whether it is a *rotate*, *forward*, or other movement command, in one iteration. A variable definition or mathematical operation should not "count" as a move; the processor should interpret an arbitrary number of these operations in one step. When a new command is requested, the interpreter will process through the code until it encounters a command which changes the visible state of the turtle. It will then complete processing this command and move to the next process.

- Example: If a user performs a long series of mathematical operations and then calls *forward* on the result, then this should count as one command even through the user might have specified many mathematical operations.

Chapter 6

Technical Implementation Issues

6.1 Overview

The server is the main process of the application which manages all the application data and handles interaction with users. It can be divided into several parts: the interpreter, login manager, chat manager, board manager, and client manager. This section describes the implementation of each pieces of the Network Clubhouse server in technical detail.

6.2 Login manager

6.2.1 Overview

The login manager handles all login requests from the client. It also keeps track of all threads spawned for handling each client and destroys those when the client logs out.

6.2.2 Implementation

The Network Clubhouse server listens for login requests on a predetermined port. To avoid misinterpreting bogus data, the server checks incoming packets for a login signature code as the first byte. The remainder of the packet specifies the username of the client. The server determines whether the user is a new or old user, possibly asking for a password to authenticate the request, and sends back information either of the existing turtle or a newly created one. Since all turtle information is stable and maintained by the server even after logout, the information transmitted may include user-defined functions from previous sessions.

The server packages this information along with other data specific to the state of the environment and the characteristic of other turtles. In addition, the server sends a UDP

port number where it expects to open a connection to send unreliable data during the session. When the client acknowledges the UDP port, the login is said to be finished.

6.3 Update manager

6.3.1 Overview

The board update manager is responsible for periodically reading the state of the environment and creating a packet which each client process then sends to the individual client. There are a few methods for handling updates, and some of the possible strategies will be discussed below.

6.3.2 What to send

The first major issue involving the update manager concerns *what* information to send. Various methods for handling board updates present themselves immediately. One is to send a complete specification of the state of the environment in each update. Another is to send information only on parts of the board which have changed since the previous update. A third is to send information on how the information has changed. The advantages and disadvantages of these approaches are discussed here:

6.3.2.1 Complete specification

Complete specification means sending each information on each part of the environment and each user of the environment in order. The complete state is communicated in each packet.

Advantages

> This requires the least amount of processing from the server end, since the server needs only to send the environment without distinguishing how its elements are changing.

Disadvantages

The main disadvantage of complete specification is packet size. If this was not an issue, board specification would possibly be the ideal mechanism in any MUD application. However, packet sizes can become unwieldy as the environment becomes larger; for example, in our sample gridworld, a 20x20 board already requires some 400 bytes of information just to specify the color of each grid. This would become even more intolerably large if the grid cells contained other features such as shape and size. Additionally, an unbounded number of turtles can exist in the environment simultaneously, and each turtle needs to be completely characterized in terms of position, color, direction, and potentially many other features.

6.3.2.2 Changes only

Changes only implies sending solely what information has changed after the last update, whether this be parts of the environment or turtles in the environment. Therefore, the packet size will vary depending on how many elements are changing.

Advantages

This avoids many of the problems in complete specification since relatively small amounts of state are likely to change for any given update. Sending changes only can give packet sizes approaching zero at best when no information is sent.

Disadvantages

The main disadvantages of sending changes only is the possibility of producing an inaccurate state. If a grid is changed but its new state is not properly communicated, its incorrect representation will remain until it is changed again. If a turtle moves or changes direction but this is not updated, then it will remain inaccurate until it moves again. This is a less

severe problem since turtle state will tend to change more frequently than that of the environment, so discrepancies will remain for shorter periods of time. An additional disadvantage is that packet size can become even larger than those of complete specification if enough elements in the environment are changing rapidly. In complete specification, the position or name of each element that is communicated does not need to be specified; the information is simply passed in order and received in order.

- Example: A 20x20 grid can be communicated as 400 successive pieces of data without individually specifying the x,y of each grid cell, as long as the client knows what order the information is arriving in. This mechanism is not analogous for the changes only method. Each element must be identified when it is sent; that is, if only five grid cells change, then their x,y positions must be communicated in addition to the changes in order to inform the client as to what is being updated. Therefore, if even more than 1/3 of the grid cells change in a given packet, this implementation will require a larger packet than if the entire board was simply communicated in order.

6.3.2.3 Information only

Information only means sending the actual user commands to each client and having the client process those commands and arrive at same result as the server and other processes. This allows the "brains" of the application to be moved primarily to the client and turns the server into a simple mechanism for relaying user commands and other minimal information to the clients.

Advantages

The advantage of taking the burden of computation off the server is obvious in applications such as Network Clubhouse where the number of clients loading the server is theoretically unbounded. Obviously, server performance will begin to degrade as it is forced to manage and communicate with increasing numbers of clients. Moving the processing to the clients allows the server's responsibility to decrease.

Disadvantages

This method is discussed for the sake of completeness, although it is apparent that it is not suitable for this application both from a philosophical and design standpoint. This process is only effective if the commands are deterministic; that is, there is no opportunity for a command to be processed in more than one way. For example, any introduction of randomness into the system will necessarily produce inaccurate views of the state by different clients. Furthermore, it suffers the same disadvantages of the changes only method, in that any miscommunication will produce a significant inaccuracy which, in this case, can never be resolved.

From a philosophical standpoint, decentralizing control to the clients would be a poor design decision. Information is more easily handled and maintained when it is centralized in one location. This allows the clients to become simple dumb display devices with minimal functionality.

6.3.2.4 Conclusion

A secondary, somewhat cursory issue should be addressed to make this discussion complete. Advantages and disadvantages were presented primarily in light of packet size and consistency of state. Another issue involves the actual time needed by the client in each case to process the information. If the processing power on the client side is low, particularly in terms of packet reading and parsing and graphical display, then those issues must be considered in the discussion of implementation. However, since our experience has shown us that this is not a significant factor, the only real drawback of the complete specification method becomes the enormous packet size, which potentially allows for more errors and higher processing time. Therefore, a combination of the complete specification and change only method was implemented. The majority of the communication

between the client and server will consist of changes in the state of the environment. In addition, a complete specification will be regularly sent to ensure that the information remains relatively accurate. We hope this will allow us to maintain relatively high performance and tolerably high accuracy. This theory still needs to be tested more extensively, especially in the evolution from the Gridworld environment to the Network Clubhouse environment.

6.3.2.5 Specific implementation

The implementation of each method will be discussed below.

6.3.2.5.1 Complete specification

Complete specification is a straightforward process so long as the client and server agree on a standard order in which the information is to be received. Currently, only board updates are sent through the UDP port, but the server needs to distinguish whether the update is a complete specification or changes only update. This is handled by putting an identifier as the first byte of the packet.

6.3.2.5.1.1 Environment

In our Gridworld, the grids in the environment are sent row-wise to reflect the way information is traditionally stored in double arrays. That is, the first row of grid colors is sent left to right, followed by the second, where 0,0 represents the top left corner of the grid. Since colors range are non-negative integers from 0-139, each grid cell can be characterized by a single unsigned byte (uchar).

6.3.2.5.1.2 Turtles

Turtles are identified by username, and each turtle's complete specification is sent in the packet. This includes the name, x, y, color, dir in that order. Multiple turtles per user may be allowed in the future and would appear with the same identifier.

6.3.2.5.1.3 Other information

Other objects will also be added into the environment as Gridworld evolves. These will probably be treated similarly to turtles in terms of specification.

6.3.2.5.2 Update only

Most of the information will remain the same for update only packets. The only distinction is that grid changes need to be prefaced by the x,y position of the grid cell.

6.3.2.5.3 Threading

The server initially spawns one process which determines board updates and packages them into a structure. When a client logs in, the server spawns another process which periodically reads from this structure, creates a packet from it, and sends that to the client via UDP transmission.

The reasoning follows that the updates should be sent at a constant rate independent of actions by the user, until the user logs out, verses maybe only sending updates when requested by the user or whenever the user sends a command.

6.3.3 When to send

A second major issue concerning the board manager involves *when* and under what circumstances update information is sent.

6.3.3.1 Send only when needed

Arguably, the update manager could be implemented to only create a new packet whenever the state of the environment changes. The client processes can then be informed that a change has occurred and send the new information to the clients to reflect the new state. This may reduce network traffic because packets would only be sent when necessary. However, this design is undesirable for at least a few reasons.

Complexity

First, this adds another level of complexity to the system. The client processes would be forced to detect changes and send packets at a nonconstant

rate. The server would need to be able to detect changes in the state (one way to do this would be to set a dirty bit in the data whenever some turtle process or patch changed). Although this method could be implemented without too much trouble, it begs the question of whether such a complicated design is necessary.

Limits on processing speed

The network communication and processing speed of the client have shown that the client can only realistically handle around 5 updates a second. If information happens to arrive at a higher frequency, the data would either be processed at less than real-time speed, or the packets would be ignored. Information will probably frequently changes at rate greater than 5 frames per second.

- Example: If a hundred turtles were moving at the same time, or the patch environment was mutating at a nontrivial rate, then the state would change at a much faster rate than updates could handle.

Need for large numbers of updates

Even if the client could process updates in a negligible amount of time, it would still be questionable whether this idea would be advantageous. Five updates a second is certainly fast enough to produce smooth animation, so a much higher resolution is probably not necessary.

6.3.3.2 Send on demand

It is also possible for the server to only send packets when requested by the client. This would allow the server to obtain some idea as to the speed that the client is running. Also, the server would only need to update its packet as fast as clients requested updates instead of at a constant, arbitrary rate. This procedure is desirable in many different situations,

such as if the client is regularly sending information back to the server (e.g. if the client is continually sending the x,y position of the mouse for real-time processing, the server could send an update every time a new x,y position packet was received).

This protocol could be easily adapted by Network Clubhouse. The client could receive a packet, process it and update the display, and send back a simple acknowledgment informing the server that it is ready for more. However, we chose not to implement this method for a few reasons.

Simplicity

The first argument is based on simplicity. It is much easier for the server to simply blindly send packets at a constant rate instead of waiting for an acknowledgment.

Speed

A certain amount of processing time would be needed for the additional sending and receiving of acknowledgment packets. Although probably not significant, this is an unnecessary overhead.

Robustness

The advantage behind UDP implementation is that acknowledgments and delivery are not guaranteed. Therefore, it makes more sense from a design standpoint to have the server process send UDP packets at a constant rate regardless of whether an acknowledgment is received. Otherwise, some error-handling mechanisms would need to be incorporated for resends and timeouts if a packet were lost and either the server was left waiting for a nonpending acknowledgment or the client was waiting for a nonpending update. If implemented, the error-handling and resend mechanisms would begin to resemble a TCP implementation, which was avoided as a design in

the first place because guaranteed packet delivery was deemed unnecessary.

6.3.3.3 Conclusion

It should be fairly clear that a non-regular update design is not the correct methodology for this type of networked application. The second option of sending only on demand was also outruled because of the need for added complexity. The philosophy behind the board update policy was that it should be as simple as possible to allow the server to concentrate on other tasks without a huge overhead in sending updates. The impact of lost packets, out of order packets, or packets sent too quickly or too slowly is minimal; as long as the packets are sent at a reasonable rate, the client's display will be accurate to a tolerable degree.

6.3.4 Client processes

Given that the main manager is updating its representation of the state at a constant rate, each client process should read this packet at a constant rate to send to the client. For obvious synchronization reasons, the client packet should be sending the packets at roughly the same speed that the server is updating them.

As an added feature, the client process can check to make sure that it does not send the same packet twice. This can be easily accomplished by adding a nonce to the server packet which is updated whenever the packet is changed. Instead of waiting for a fixed period of time, the client process could instead wait for the nonce to change and send the packet. Note that this is different from the nonce that the client sends the user. When a user logs in, it expects the first packet update to be a zero and increment by one successively. Therefore, the client packet must keep a separate and unique nonce which it will send to its particular user and increment each time the packet is sent. Another way to interpret this

is to say that the client process holds a certain offset from the nonce maintained by the server packet so that the client receives whatever nonce it is expecting to.

In any case, as stated before, the client will read the packet, add its own nonce at the head of the packet, and send the resulting output to the client process.

6.4 Chat Manager

6.4.1 Overview

The chat window provides a mechanism for users to broadcast and receive conversational messages with other users. The success of forums such as IRC and the MIT-wide zephyr service prove the effectiveness and usefulness of this feature. Naturally, the need to communicate among users is critical in this application both for problem solving and general social interaction.

Chat commands are communicated through the same TCP socket as normal commands. The client marks the chat message with an identifying byte in the header of the packet to distinguish the message from normal commands. The server receives these commands and broadcasts them to all visible users without interpretation of the content.

6.4.2 Features

The chat server does no processing on the actual content of the message; it only acts to relay them to users. Some features can be easily implemented in the future to allow for additional functionality. These include scanning for inappropriate language and specifying group and individual messages.

6.4.2.1 Scanning for inappropriate language

It should be fairly straightforward to scan a message for inappropriate or vulgar language. How this would be handled is more of an educational issue; the information could be logged or the sending user could be notified and denied access in the future. The

amount of scanning also would need to be addressed; the scanner should not be confused by valid words which happen to contain inappropriate language as substrings, but this would leave it blind if users simply append a nonsense character before or after the vulgarity. One possible solution would be to simply log "suspicious" words for an administrator to later peruse.

6.4.2.2 Individual and group messages

Most chat forums allow users to send messages to individuals or groups of users instead of broadcasting to all active ones. This feature can be incorporated easily by allowing the user to specify a destination name at the head of the message. Users can then form groups for communication; in this case, any group name which is not equivalent to an existing username would be allowed, and messages addressed to the group would be sent to all active users in that group. Finally, users should be allowed the capability of ignoring message from undesired users. This can be easily implemented, although system messages, such as from the administrator of the application, should not be overridden.

6.4.3 Implementation

There are many possible designs for a chat server. Two of the more promising methods are discussed here.

6.4.3.1 One sending process

It is feasible to design the chat server to simply listen for requests at a single port and blindly echo those to users who are currently logged in. The server could maintain a list of all active users and their network addresses and simply iterate through the list and send the message to each user on that list in order. This can be implemented by spawning one process for each user to listen for incoming messages, and maintaining one process to handle sending the message to all currently logged users.

6.4.3.2 Multiple sending processes

A chat server could contain multiple sending processes. This design would be similar to the previous save that the server would spawn one process for each client to send messages to that client instead of keeping a central list of all clients. Whereas the other design places the burden in the centralized control of the main chat process, this appears more elegant because each process becomes responsible for communicating with its single user. If a connection crashes or experiences difficulty, the effects are local to that process and the server can continue to operate without being adversely affected. This design also follows more consistently with the overall strategy of spawning individual processes to handle various needs associated with each client.

The chat server spawns a dedicated process for each client that logs in. This client process then listens for chat commands and notifies the central server upon reception. Another process listens for new messages in the chat server and broadcasts those to its client. The main chat process waits for clients to notify it of new incoming messages. It then processes those and transfer them to an "outgoing message" bin.

A synchronization difficulty arises because it is possible for the outgoing message from the chat server to be changed before each chat process can send the message to its client.

- Example: Imagine that a message arrives from a client and is transferred as the incoming message by the main chat server. The server then copies this to its outgoing message bin and expects the individual chat processes to notice the update and send it to the clients. However, if a chat process is slow in executing or delayed, or two incoming messages arrive near simultaneously, it is possible for the message to be lost entirely, or at least lost to some of the clients.

Fortunately, there are several implementations to address this issue.

6.4.3.2.1 Ignore the problem

It is arguable that the issue is not critical enough to warrant implementing a complicated handling mechanism. The clients might simply assume that messages would not

arrive at such a high rate that a significant number would be mishandled. Instead of addressing this issue at the server end, the clients could depend on the users to request resends whenever lost messages caused confusion.

Advantages

This method is simple and elegant. There are many cases where communication is lost between parties, whether through lost post office mail, wrong numbers, or misdirected email.

- Example: MIT's zephyr service is a utility which allows users to send real-time messages to each other's screens through a central server. Zephyr does not guarantee delivery, however, and it is the user's responsibility to resend or request resends if messages are expected but not received. The popularity of Zephyr proves that it is effective despite this uncertainty.

Disadvantages

Obviously, this method should only be considered if no other feasible solution can be determined. In cases where a solution would force a slow or overly complicated system, a less robust system might be opted. However, it is clear in this case that several simple solutions can alleviate this problem.

6.4.3.2.2 Limit throughput

If the server forces a considerable amount of time after each incoming message arrives before allowing another to replace it, the server can be reasonably certain that the chat processes have each individually updated their state and sent their messages to the clients. This is the simplest method. However, it makes no guarantees that the clients will always receive the messages; it simply makes it more likely that they will.

6.4.3.2.3 Force notification

The server can, upon receiving a message, force each process to acknowledge that they have received and processed the message. In this implementation, the server would need to know the number of active clients and hold the message until all chat processes had acknowledged the message. This method is somewhat cumbersome to implement and the entire server would be disrupted if any client process unexpectedly died or hung for a long period of time.

6.4.3.2.4 Implement a queue

The server can maintain a small queue size. When messages arrive, the server copies them into its queue. Clients will notice the presence of a non-empty queue and copy the contents to the user. In this design, messages remain in the server for a relatively long period of time, affording the clients ample time to notice and process them.

6.4.3.2.5 Conclusion

A queue is a reasonable mechanism to ensure that chat processes are able to respond to incoming messages before they are erased. The queue grows as messages arrive and erases messages from the head at a constant rate, allowing clients to search the entire queue for messages that they have not yet processed. To accomplish this, each message in the queue will be identified by a nonce, and chat processes will handle all messages that are greater than the last nonce they have each seen.

6.4.4 Debugging notes

The chat server was implemented as a stand-alone, independent tool. The functionality is rather general and can be incorporated into any application which required a chat feature. The server simply listens for input and echoes it back to active clients. For debugging purposes, a basic interface has been built which compiles with the chat code. This allows users to independently run a chat server and log in from simple clients to send and receive text.

6.5 Compiler

6.5.1 Overview

The compiler translate ascii text into byte codes that the interpreter can understand. A suitable compiler must obey several principles. These will be discussed below.

6.5.1.1 Independence

The compiler must be relatively independent of the interpreter. This means that it should not need to rely on direct communication or information from the interpreter in order to function successfully.

6.5.1.2 Determinism

The compiler should be deterministic; that is, there should be only one means to parse and translate each incoming command. The compiler should not need to "guess" at the correct interpretation of the sentence or attempt multiple parsing paths before reaching the correct one. It is tolerable for certain commands to have more than one possible interpretation, but only if the compiler can arbitrarily but consistently parse the sentence in only one correct way.

The Network Clubhouse compiler is more of a "hack" than an elegant implementation of a traditional language compiler. The compiler performs the usual tokenization, but parsing does not follow a well-defined grammar. Nor does the compiler perform the traditional code optimization techniques. Because the client interface to Network Clubhouse is through an interactive command line, the compile design needs to meet different specifications than a language compiler which parses a large amount of text and generates an executable.

6.5.1.3 Speed

Our compiler needs to be designed for optimal speed. The total response time of the system to the user consists of the compilation time plus the execution time. Therefore, the compiler must be optimized for reasonably fast performance.

6.5.2 Implementation overview

Our compiler operates in three stages.

6.5.2.1 First stage: function reading

In the first stage, the compiler reads all user-defined functions and primitives to determine how many arguments each function requires and whether each function outputs in the course of its execution. This allows the compiler to determine whether proper grammatical rules are obeyed for function calls.

6.5.2.2 Second stage: tokenization and translation

The second stage simultaneously tokenizes words and translates them to proper notation.

6.5.2.2.1 Tokenization

Each word must be tagged with an identifier which informs the interpeter of the type of the argument. An argument can be an integer, variable, pointer, function name, etc.

6.5.2.2.2 Translation

The main challenge of the compiler is to translate commands from infix and postfix to prefix notation. Since the interpreter is stack-based, arguments must appear before the function call to allow arguments to be conveniently popped from the stack and read by the function.

6.5.2.3 Third stage: code generation

The final stage is the actual conversion of the output stream from characters to byte codes. This is accomplished through a table lookup associating words with numbers.

6.5.3 Detailed explanation of implementation

The compiler reads through the input string and writes to an output string. A sentence either compiles successfully, in which the compiler sends the resulting byte stream to the interpreter, or the compiler signals a compile-time error and invokes mechanisms to alert the user.

6.5.3.1 First step: function reading

The compiler begins by reading through the user functions and classifying them by number of arguments and return type. If the function contains an "output" (equivalent to a "return" command in C) within its body, the compiler classifies it as a reporter. After the compiler has completely scanned through the user-defined functions and primitive definitions, it will have a complete list of legal function names. As a further distinction, the compiler scans through a list of infix primitives (mostly mathematical operations) and classifies these as infix operators. The ability of the compiler to handle infix operation is certainly necessary if mathematical constructions in the language are to be intuitive.

6.5.3.1.1 User-defined functions

For user-defined functions, the compiler compiles each function and communicates the byte codes to the interpreter. The interpreter then stores the code of the function in its memory to use when the function is later called.

6.5.3.2 Second step: tokenization and translation

The compiler begins to reading a word off of the input stream. The word can either be a function call, list begin, or argument to an infix operation. Formally:

S -> ***AS*** || ***e***

A -> ***F*** ||
    (S) ||
    A op A ||
    v ||
    n

A = argument nonterminal

F = function nonterminal (F -> function1 name || function2 name || ...)

S = start nonterminal

e = empty terminal

v = variable terminal (v -> variable1 name || variable2 name || ...)

n = number terminal (n -> 0 || 1 || 2 .... || 9)

op = infix operator terminal (op -> + || - || > || ...)

6.5.3.2.1 Functions

If the argument is a function accepting n arguments, the compiler processes the next n arguments as function arguments and appends those to the output stream, finally adding the identifier for the function along with the tag *FUNCTION*. This is equivalent to the formal expression:

$$F \rightarrow function\_name\ A^n$$

If the function is expecting a return value for the argument but none is provided, then the compiler will signal an error. While some functions, such as *repeat*, allow arguments which do not return values, other functions, such as mathematical operations, require arguments to eventually parse to values which then are then used by the operator. Note that anything which validly reduces to one value can therefore be considered as an argument (e.g. function calls, lists, infix operations which are reporters).

6.5.3.2.2 Lists

If the function is a list, the compiler recursively processes the arguments in that list, and finally notes whether the result of the list outputs a value. If so, the list can be validly used as an argument for a function which expects a value. Note that our formal definition

allows sequences of commands to appear in lists and nested lists, as is appropriate in traditional list syntax.

6.5.3.2.3 Infix operators

If the argument is neither a list or a function, the compiler processes the next word, assuming it will be an infix operator. If this is the case, the compiler then proceeds to process the function as usual while remembering that it has already received one argument.

If the next word is not an infix operator, the function compiler signals an error.

6.5.3.2.4 Variables

Variable arguments can either be a value or pointer. Formally:

V -> RVARIABLE || LVARIABLE

Variable values are used for call-by-value operators, which our language primarily operates on. Since most variables are global, call-by-value allows non-local variables to be mutated inside function calls. In this case, the variable is tokenized with an *rvalue* tag.

A relatively small set of special primitives use call-by-reference. The most obvious is *setvar*, or =. In this case, the variable is tokenized with an *LVALUE* tag and treated as a pointer.

6.5.3.2.5 Numbers

Numbers are simply tagged with a *NUMBER* tag and passed to the output string.

6.5.3.2.6 New definitions

The compiler needs to be capable of dynamically handling new definitions of functions and variables. The following section explains how new definitions are processed.

Variables

> When the compiler reads a new variable definition, it adds the variable name to its list of known variables. It also assigns a variable number identifier (i.e. an increasing number starting from 0 for the first variable) as a

mapping hint. When the variable is referenced in the future, the compiler uses this mapping to convert the variable name into a number which can be used by the interpreter.

Functions

Function definitions are treated similarly to variable definitions except that additional information needs to be maintained and storage must be permanent. Although variables only last through the current session, functions need to be maintained across sessions and therefore should be stored on stable storage such as a separate file for each user.

When the compiler reads a user function definition, it adds the function to its set of known user functions, assigns a function identifier to it (i.e. an increasing number starting with 0 for the first function), and determines whether the function is a reporter (i.e. if it contains a *return*, or *output*, statement within its body). This information is stored in the user's file to be used for processing further references to that function in function calls.

6.5.3.3 Third step: conversion

After the translation to postfix notation is completed and compile-time errors are checked, the final stage of translating directly from characters to byte codes becomes straightforward. The output string will be in the form of "identifier value identifier value..."

- Example: For the primitive *setxy* expecting two arguments, the input might be *setxy 10 10*, which would translate to *NUMBER 10 NUMBER 10 PRIMITIVE setxy.*

The compiler simply scans through the output string and matches each token word to its byte code and each token value to its byte code.

6.5.3.3.1 Primitives

If the token is a primitive, the compiler scans through the byte codes of primitives and performs a stringmatch between the name and list of primitive names it is recognizes. Since the primitive is guaranteed to match somewhere (or the compiler would have beforehand signalled an "unknown variable name" error during compile time), this should be a straightforward process.

6.5.3.3.2 Functions

Similarly, if the token is a function, the compiler scans through the byte codes of functions and performs a stringmatch. The natural question arises as to where this mapping between functions and numbers is stored. To answer, recall that each user possesses a datafile containing a list of user-defined functions. The compiler scans through this list of function definitions and assigns numbers to each function on an increasing scale starting with 0. It then passes the function number and definition to the interpreter. Whenever a function name appears in an input string, the compiler converts the name to a function number based on this heuristic. The only requirement is that function numbers do not change during the course of the session, which should not happen if new function definitions during the session are handled correctly.

6.5.3.3.3 Variables

A somewhat more difficult problem presents itself in terms of variable translation. Recall that the compiler operates with string names for variables, and the interpreter operates with number identifiers for variables. For example, the variable *var1* may correlate with variable number 0 in the interpreter, corresponding to the first variable in the stack frame.

Therefore, the compiler needs to somehow assign and translate variable names into numbers for the interpreter, analogous to function name translation. If a variable is global, this is not difficult. The compiler simply uses an identical algorithm to function determina-

tion, storing a mapping between variable names and increasing numbers and appending global variables to that list whenever they are defined in the session.

However, handling local variables in functions is a somewhat more difficult process, especially as functions can be nested. An inherent issue in local variables is that *arg1* will refer to one variable in one scope and another in another scope. Since variables are lexicographically scoped, the compiler needs to additionally handle this translation. That is, if *arg1* does not appear in the current scope, the compiler should perform a lookup in the previous scope, all the way until it searches the global scope and either find the variable there or signals an error. There are some possible ways of accomplishing this translation:

6.5.3.3.3.1 Restrictions on the language

The easiest way to solve this problem is to restrict the type of variable names that can be used in the language. Users could be instructed to name their variables in order; either the first variable should be named '1' or 'A', and the second '2' or 'B', etc. This would alleviate much of the burden of translation from the compiler, which could directly translate 'A' into 0, 'B' into 1, etc., and assume that these refer to legal variables in the interpreter.

Disadvantage

> Experienced users would not be terribly challenged in obeying this protocol. However, it would be somewhat disappointing if Network Clubhouse resorted to this method since it places an unreasonable burden on the user and violates general language methodology.

6.5.3.3.3.2 Direct translation from variable name to ascii

One possible way to handle this problem is to convert each variable name to its ascii equivalent. For example, if all variable names were one character long, the compiler could translate the character to its ascii representation and pass this code to the interpreter. The

interpreter would need to somehow use that value to find the appropriate address of the variable in its stack frame.

Disadvantage

This is also a somewhat cumbersome and undesirable process. We would like the compiler to assign meaningful numbers to the variables (e.g. variable 0 for the first variable, variable 1 for the next) so that the interpreter could easily translate those into addresses in its stack frame.

6.5.3.3.3.3 Communication with the interpreter

Since scopes change with the addition of each new stack frame, one possible way to resolve the translation issue is to somehow include the mapping at the beginning of each stack frame. A table of variable name to number pairs can be included in the stack.

- Example: If two local variables, A and B, exist within the new stack frame, the first three stack entries in the frame could be "<ascii value of 'A'> 0 <ascii value of 'B'> 1". Of course, the 0 and 1 can be eliminated if the compiler can assume that the order of appearance of ascii values corresponds with the order of appearance of variable numbers. The compiler could then somehow access these entries from the interpreter to convert the variable into its appropriate number this way; or the compiler, on receiving the ascii variable name, could use this table to translate the name into the appropriate number.

Disadvantage

This process requires entirely too much communication between the compiler and interpreter, which should be relatively separate entities. Recall that one of the guidelines of compiler design is that the compiler and interpreter should be separate entities which require minimal intercommunication.

Conceptually, the interpreter should not need to keep track of the ascii variable names in order to function correctly and perform translations. Similarly, the compiler should not need to inquire the stack frame of the

interpreter in order to operate correctly. Although this means was considered as a possible alternative, it was weighed more as a "hack" than an elegant design solution.

A more serious problem is that although the compiler is currently implemented on the server side, it should be moved to the client in the final version of Network Clubhouse. In this case, the compiler and server would not be able to reasonable communicate with each other in a manner which is suggested here.

6.5.3.3.3.4 Global variables

One attractive way to solve this issue is to restrict the appearance of local variables to only function arguments. In this situation, any variable definition that appears at any time should be considered a global variable and appended to the stack of global variables. Translation can then follow easily. To handle function arguments, the compiler arbitrarily assigns variable numbers of each variable argument that appears in the function parameters. It can then translate any variable names it sees within the function body based on those assignments while it is processing the function definition.

This follows closely with the Logo model that all variable definitions become global, whether the variable is defined within the global context or within a local frame. This method suffices for now; however, as *let* assignments and dynamic scoping become implemented, a binding tree or other means of translation will be explored.

6.5.4 Additional notes

6.5.4.1 Errors

Compile-time errors should be communicated to the user. Various standard methods can be used to allow reporting of compiler errors, and these are followed in our implementation. An error message includes the following:

- Word and line number where the error occurred

- Nature of the error
- Suggestions on how to rectify the error.

• Example: "Compiler error, line 3, at word "print". Expected LIST, read PRIMITIVE."

6.5.4.1.1 Implementation

Because the compiler resides on the client machine, it is fairly easy to communicate compiler messages to the client and request that they be displayed on the command line screen.

6.5.4.2 Debugging

It is convenient to have a direct interface to the compiler for debugging purposes (rather than needing to log in through the client to test commands). I have provided a short program that interacts strictly with the compiler by asking for text from the user, reading the results of the compiler, and printing out the resulting byte codes. This program uses a simple command line and compiles independently with the compiler code.

6.6 Interpreter

6.6.1 Overview

The interpreter receives machine code instructions from the compiler and processes those into turtle commands. It is also responsible for managing the functions, variables, and memory for each individual turtle.

6.6.2 Implementation

The interpreter reads values from the instruction stack and performs operations on them, using the data stack as "scratch space" to store and retrieve information that it encounters as it is processing the instructions.

The compiler translates the user's commands into machine code, converting each data or word to a pair which is comprised of the data value and type (token). Examples of the type include *NUMBER*, *VARIABLE*, and *FUNCTION*. The type is the key which allows

the interpreter to determine how the data value is to be interpreted. Therefore, the interpreter reads off pairs of data points and expects each data point to be prefaced by its type.

A detailed explanation of each allowed type and its implementation will be discussed here.

6.6.2.1 Numbers

When a number type is read, the interpreter simply pushes that value onto the data stack.

6.6.2.2 Rvariables

Rvariables, or pass-by-value variables, cause the interpreter to perform a lookup of the variable value and pop the result onto the data stack. Recall that a lookup of the variable value simply consists of reading whatever stack entry is stored in the offset of the variable number from the frame pointer.

6.6.2.3 Lvariables

Lvariables, or pass-by-reference variables, or pointers, cause the interpreter to pass the actual address of the variable onto the data stack.

6.6.2.4 Function

The function type indicates that a function call is to be performed. The interpreter determines the function address from the function identifier and proceeds to create a new stack frame for processing the function. The instruction pointer is set to the address of the function's instruction stack, and the number of arguments for that function is determined. These arguments are then popped from the data stack and copied into their appropriate addresses in the new stack frame. One subtlety is that the arguments need to be popped off the original frame first before the new frame is created, then pushed onto the data stack of

the new frame. This implementation issue arises because a new stack frames is always immediately appended to the end of the previous stack frame.

6.6.2.5 Function definitions

A function definition request causes the interpreter to create storage for a new function specification. The interpreter allocates an instruction stack for the function, scans through the function and transfers its contents to this instruction stack, and performs other book-keeping operations such as determining the number of arguments of the function and deciding whether the function is a report (which is true only if a *RETURN* command is present within the function).

6.6.2.6 Lists

Lists allow commands to be grouped for use with operations such as *REPEAT*, and are also used to implement order of operations for mathematical operations. When a list begin is read, the contents of the list are pushed onto the stack to serve as a single argument. Of course, lists can be nested, so the server needs to be careful enough to push the contents of the entire list including any any nested ones.

6.6.2.7 Primitives

Primitives are predefined functions which are implemented in the underlying language of the server. Primitives form the basis of the higher level language, and all functionality is defined on top of the primitives layer. Primitives cause the interpreter to call a C function which carries out the desired behavior.

6.6.3 Examples of primitives

A section is devoted to discussing a subset of the various primitives that are available in our language. Because some of these primitives non-trivial, an explanation of their design and implementation may be useful as a guideline to future construction.

6.6.3.1 FUNCTION_END / RETURN

A *function_end* command informs the interpreter that it has reached the end of a function. The function should then restore the original instruction pointer and pop the stack frame. Similarly, a *return* command also restores the original instruction pointer and stack frame, but additionally pushes the value of the *return* variable onto the data stack so it can be immediately accessed as an argument.

6.6.3.2 SETVAR

The *setvar* command is currently the only command which performs direct manipulation on a variable's address (pointer or *LVALUE*) instead of its value (*RVALUE*). The *setvar* command expects the address of the first variable (i.e. the variable's numerical id) and its new value. It then replaces whatever is in the address of the the variable with the requested value.

To make *setvar* more convenient to use, *setvar <variable name> <value>* is instead syntactically equivalent to *set<variable name> <value>*.

>make :foo

>setfoo 10

When a variable is defined, the *set<variable name>* primitive is automatically defined as well.

6.6.3.3 REPEAT

Repeat expects a numerical argument and a list and processes the contents of the list for the number of times specified by its first argument. Most of the time, the interpreter should process one *repeat* iteration and then yield to the next process.

- Example: If a turtle was requested to move 100 steps using a *repeat* command, the repeat should process only one move step in each iteration, thus taking 100 cycles to complete the command instead of only one.

This behavior is accomplished using a simple *goto* in the *repeat* procedure. When a *repeat* command is encountered, the iteration number is pushed onto the data stack, and then the list of commands to repeat. The process then interprets the list (or a subset of the list, if there are, for example, multiple move commands within the list) and decrements the argument by one on completion. If the iteration number is greater than zero, the interpreter performs a jump to the beginning of the repeat command and yields to the next operation.

When dynamic scoping is fully implemented, another means of handling *repeat* will be to treat *repeat* as a real user-defined command with one argument, so that it will create a new stack frame with the iteration number as its single argument. The instruction pointer will point to the beginning of the set of commands in the list and the interpreter will proceed with interpreting commands there. When the interpreter reaches the end of the list, it will pop the stack frame as it normally would after the end of the function. However, the old frame pointer will have been previously replaced by the beginning of the current frame so that the interpreter really performs a simple *goto* to the beginning of the command again. This will repeat until the argument is zero, after which the original old frame pointer will be restored.

I believe this is roughly the method that is currently used in StarLogo to implement repeat. It is seen more as a hack than an elegant design solution.

6.6.3.4 PRINT

Print allows the user to request the interpreter to display a value. Print expects one argument and sends the desired output along the same lines at the TCP chat line.

Chapter 7

Conclusion

Network Clubhouse part of an ambitious educational effort by the Epistemology and Learning Group at the Media Lab to expose children to networked constructive environments. It is *graphical*, benefiting from the natural advantages of a visual environment. It is *programmable,* allowing students to learn through producing behaviors in the objects. It is *constructive*, focusing on the educational goal of learning through creating and building projects. It is *networked,* allowing students to cooperate and interact as part of their learning experience.

Most importantly, it is *unfinished*, and much work needs to be done before Network Clubhouse becomes a reality. Efforts to implement Network Clubhouse will continue this summer, and next year another developer will assume responsibility of the Network Clubhouse task. Though much of this document was devoted to describing the implementation challenges and design decisions, the focus of Network Clubhouse revolves around its potential use and capabilities. How will children use and respond to Network Clubhouse? Now successful will Network Clubhouse be in bringing communities of children together to cooperate on ambitious learning projects? How will children react to the programming language and features of Network Clubhouse? These questions address the exciting future of Network Clubhouse and its potential contribution on the educational community.

References

Drafts and internal references (available in the Epistemology and Learning Group)

[1] [Resnick 1] Resnick, M. New Paradigms for Computing, New Paradigms for Thinking.

[2] [Resnick 2] Resnick, M. Building on the Net: New Opportunities for Construction and Community.

[3] [Resnick 3] Resnick, M. Distributed Constructionism.
NOTE: This paper will appear in the *Proceeds of the International Conference on the Learning Science*, Northwestern University, Evanston, IL 1996.

[4] [Resnick 4] Resnick, M. Building on the Net: New Opportunities for Construction and Community.

References

[5] [Resnick 5] Resnick, M. Turtles, Termites, and Traffic Jams. MIT Press. Cambridge, MA. 1994.

[6] [Resnick 6] Resnick, M. Behavior Construction Kits. *Communications of the ACM*, July 1993, Vol 36, No. 7.

[7] [Repenning] Repenning, A. Programming Substrates to Create Interactive Learning Environments. *Interactive Learning Environments*, 1994, Volume 4, Issue 1.

[8] [Kay] Kay, Alan C. Computers, Networks and Education. *Scientific America*, September 1991.

[9] [Bruckman 1] Bruckman, A and Resnick, M. The MediaMOO Project: Constructionism and Professional Community. *Convergence*, 1995, Volume 1, Number 1.

[10] [Papert] Papert, S. The Children's Machine. Basic Books, 1994.

Other Thesis Work

[11] [Bruckman 2] Bruckman, A. MOOSE Crossing: Creating a Learning Culture. MIT Massachusetts Institute of Technology, November 1994.

[12] [Sargent] Sargent, J. The Programmable LEGO Brick: Ubiquitous Computing for Kids. Massachusetts Institute of Technology, February 1995.

[13] [Mozes] Mozes, A. The Ninja Neighborhood: A Collaborative Learning Environment for Children. Massachusetts Institute of Technology, May 1993.

Programming References

[14] [Robbins] Robbins, K and Robbins, S. Practical UNIX Programming. Prentice Hall, NJ. 1996.

[15] [Lynch] Lynch, D. and Rose, M. Internet System Handbook. Addison Wesley Publishing Company, CT. 1993.