Distributed Tools for Distributed Thought: Networked StarLogo

by

Bruce R. Krysiak

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 1995

Author [illegible]
Department of Electrical Engineering and Computer Science
May 30, 1995

Certified by [illegible]
Mitchel Resnick
Thesis Supervisor

Accepted by [illegible]
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

Distributed Tools for Distributed Thought: Networked StarLogo

by

Bruce R. Krysiak

Submitted to the
Department of Electrical Engineering and Computer Science

May 30, 1995

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

ABSTRACT

StarLogo, an extension of the Logo programming environment developed over the last several years at MIT's Media Lab, serves as a microworld construction kit for decentralized systems. Until now, however, the StarLogo interface has been focused on single user interactions only. I have extended this interface to allow multiple users to interact and explore in a common, "shared" space. Users may add and control turtles in this space, one in which other users may have their own, independently operating turtles. In this way, multiple users are able to interact within a single, common environment, opening up the world of StarLogo to a new set of capabilities for shared learning.

This paper talks about the experiences of the author in designing and implementing the Distributed StarLogo system in Macintosh Common Lisp, including design and implementation issues and choices, project results, lessons learned, and future directions that can be explored beyond the current system.

Thesis Supervisor: Mitchel Resnick

Title: Head: Assistant Professor, Media Arts and Sciences

Acknowledgements

Like all those who have toiled before me, and like all those who are still to come, I owe a tremendous debt of gratitude to the many people around me who helped make this thesis possible. I would now like to thank:

Mitchel Resnick, my thesis advisor, for taking me under his wing and allowing me to explore many types of distributed thought through the use and modification of his creation, StarLogo, and for taking the time and effort to support and guide me on this great undertaking;

Andy Begel and Brian Silverman for helping me to make sense of the strange and wonderful beast that is Macintosh StarLogo;

Rick Boravoy and David Cavallo for helping me get started in the Media Lab and for first introducing me to Mitchel;

and my parents, for helping to get me where I am and for helping me go where I'm going.

1 Introduction

StarLogo, an extension of the Logo programming environment developed over the last several years at MIT's Media Lab, serves as a microworld construction kit for decentralized systems (Resnick 1992). Until now, however, the StarLogo interface has been focused on single user interactions only. I have extended this interface to allow multiple users to interact and explore in a common, “shared” space. Users may add and control turtles in this space, one in which other users may have their own, independently operating turtles. In this way, multiple users are able to interact within a single, common environment, opening up the world of StarLogo to a new set of capabilities for shared learning.

One such capability is the ability of several users to collaborate simultaneously in the design and construction of a shared artifact. With this functionality, I hope to allow users to experience a more synergistic style of learning than would be possible with several users, even in the same room, working on their own, distinct StarLogo worlds. This idea also resonates with the core of the StarLogo distributed mindset, where multiple agents may interact with the same StarLogo environment independently, effectively bringing a distributed *context* to development and learning with StarLogo’s distributed systems *content*.

1.1 What StarLogo Is

StarLogo is essentially an extension of Seymour Papert's introductory programming language, Logo (Papert 1980), for use in helping people explore and understand complex, decentralized systems. Three key points serve to distinguish it from its predecessor. First, instead of one turtle, users can control thousands of turtles simultaneously. This allows and encourages people to think about and use StarLogo in very parallelized, distributed fashions. Second, instead of a passive, unresponsive environment, StarLogo turtles live in a world of patches, each of which may be controlled in much the same way as the turtles themselves. By conferring an almost co-equal status to the environment, StarLogo tries to encourage thinking about interactions not just between turtles and turtles, but also between turtles and the environment. Finally, StarLogo turtles have better senses than ordinary Logo turtles. While Logo turtles are meant for drawing, StarLogo turtles come with a built-in ability to interact, lending themselves more toward the behavioral constructions for which they were designed (Resnick 1992). More detailed information is contained within the documentation distributed with the StarLogo package (Resnick 1995).

1.2 What StarLogo Is Good For

The driving force behind the StarLogo programming language is a desire to help people move beyond what Resnick calls the "Centralized Mindset," which is people's tendency to think about systems using a type of "centralized control" paradigm. This is a "top-down" approach to systems thinking. By contrast, StarLogo encourages a more "bottom-up" approach, one in which high level behaviors emerge from the interactions of simple, low-level rules. StarLogo has been used to help high school students create and explore many diverse kinds of decentralized systems, ranging from termites gathering wood chips to the formation of traffic jams (Resnick 1992).

1.3 Focus of the Thesis

My thesis focuses on the design and preliminary implementation of a networked version of StarLogo. It seems that this project will have some interesting implications involving collaborative learning, as well as the potential to add to what Sherry Turkle calls the "holding power" of StarLogo (Turkle 1984). One need only glance at a university computer center overnight to see the legions of devotees to MUDs, MOOs and other shared environments, as well as those enabling networked competition, such as Netrek and Xtank. By enabling users to interact in the StarLogo environment in these fashions, I hope to leverage the human fascination with shared experience to involve people in the systems learning power of StarLogo at a deeper level than they would experience otherwise. The distributed functionality will also give users a richer environment within which to learn about such decentralized systems, reinforcing the system's learning effectiveness.

2 Background

This section describes several of the motivating factors behind the Distributed StarLogo system, as well as listing several examples of possible domains and projects that the system could be applied to.

2.1 Motivations for Distributed StarLogo

The design and implementation of Distributed StarLogo was chiefly aimed towards exploring further various aspects of constructionism and decentralization beyond that which was already present in the original StarLogo package. In addition, it also attempted to explore how to encourage similar extensions by other future researchers. The mechanisms by which I tried to accomplish these goals follows.

Adding Content to Context

While it is clear that the *content* of StarLogo is a clear step towards the ultimate goal of encouraging decentralized thinking in the world, the design and context of StarLogo has still been entrenched in the centralized mindset. *One* user may use *one* computer to explore (usually) *one* system at a time. While one could argue that in reality there are *many* users with *many* computers working on *many* worlds, it seems like there is a difference between this "decentralized" model and actual decentralized phenomena in the real world.

One of the basic tenets of StarLogo is that complex phenomena arise from *interactions* among many very simple agents, and that is precisely what is missing from StarLogo, when viewed "up" one level. There is no way for *people* to interact with one another in the standard version of StarLogo. Of course, several people could be working at the same machine on a given simulation, but what if your collaborator is across the country? Or around the world? With normal StarLogo you would be stuck discussing your work via email, perhaps, or sending files back and forth on a regular basis. This thesis is concerned with helping to increase such human interactions and collaboration on StarLogo projects by making those interactions as effortless as possible, supporting and thereby encouraging their occurrence.

Introducing Shared Experiences

Distributed StarLogo also imbues explorations of its microworld with the power of the *shared experience*. While it may sometimes be valuable for a user to explore the StarLogo world without the disruptive activities of others, in many situations the addition of a social aspect to the task can increase the users' excitement and enjoyment of the experience. Having others able to share the process of learning and exploration gives those experiences greater impact and lasting power, as well as serving as a vital component in creating a StarLogo community, below.

Creating a StarLogo Community

With the distributed version of StarLogo, I also hope to reap some of the benefits gained by creating a community of users devoted to a common goal, namely exploring and learning about complex, decentralized systems. As Amy Bruckman relates in her research on community and learning, the presence of a community of learners can be a key motivator for learning. People can share ideas and projects with the community, which will then tend to increase the desire of others to contribute their own work to that community. In addition, a community can give emotional and technical support that is hard or impossible to come by in an individual learning context (Bruckman 1994). By allowing users to participate in system building projects together, I hope to eventually encourage a StarLogo community to form that will develop and maintain strong interpersonal bonds.

Encouraging Further Development

Distributed StarLogo will also fuel the further development of the StarLogo language itself in several directions. First, it will provide a new context for developers to think about the user and language interfaces. It also provides a springboard from which to explore new decentralized aspects of StarLogo development, as well as serving to generate new ideas for further language development. Finally, the distributed version can shed light on assumptions and limitations previously hidden under the single-user interaction model. By exposing these constraints, future development can focus on formalizing or eliminating these limitations, depending on whether they are seen to be beneficial or detrimental to the overall schema.

2.2 Potential Distributed StarLogo Projects

This is a short listing of a few examples of tasks that are possible with the new, distributed version of StarLogo. Future users are sure to dream up and explore many others, but I just wanted to try and give a feel for the kinds of things the new system will be good for, as well as to provide a more visual representation of where the different versions of Logo and StarLogo fit into a larger picture (below).

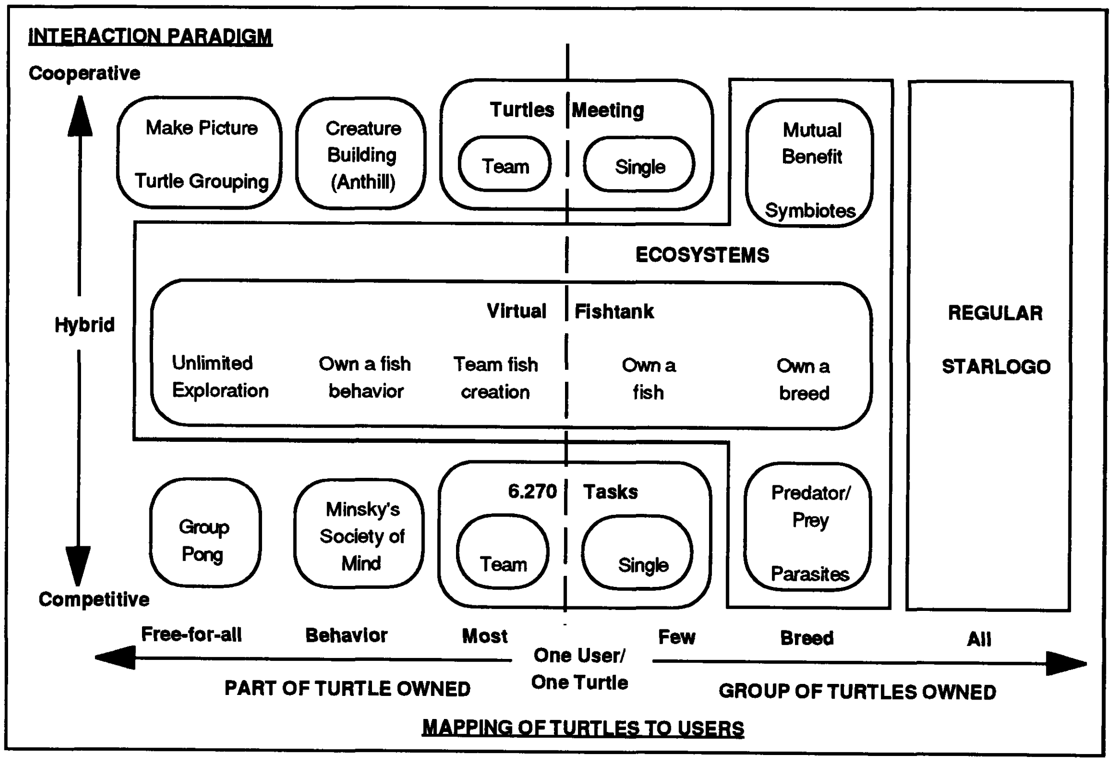

Figure 1. StarLogo user space.

In the distributed context, users of StarLogo can interact in a variety of ways. A useful way to think of these interactions is within a two dimensional space, with axes for the interaction paradigm of the users and the mapping of users to turtles (see figure 1). Users can either interact competitively, one user or group trying to beat another, or cooperatively, where a set of users must work together to accomplish a task or to solve a problem. A hybrid situation might exist when the application allows either cooperation or competition, but does not specify either, allowing the users the freedom to choose between them.

Along the other dimension, the mapping of turtles to users may vary from any number of users to any number of turtles. All turtles may "belong" to one user, a user may own an entire breed of turtles, or he may own a few of a single breed of turtles. One can envision a dividing point between users owning turtles exclusively and users sharing turtles with others at the "one user/one turtle" point. The original Logo is a special case of this situation, where there is only one turtle *and* only one user (see below). Crossing to the shared turtle space, a user may own *most* of a turtle (sharing with few other users), *part* of a turtle (sharing with many or all other users but owning a "piece" or behavior of a turtle), or *none* of a turtle (all users can affect all turtles indiscriminately - a turtle "free-for-all").

A third dimension that can be seen to be occupied by the Logo family of programming environments is that of the total number of turtles present in their microworlds. While Logo is “stuck” in the one-turtle case, the StarLogo branch of the tree extended the number of turtles available to the user in the same way Distributed StarLogo extends the number of users that can connect to a given exploration.

Going back to figure 1, if one user owns all turtles, we have the standard non-networked StarLogo system. When each user controls a "breed" of turtles, one might imagine different types of ecosystems developing: users may develop predator-prey or parasite relationships (competitive case), or they may work together with other breeds to achieve a situation of mutual benefit or symbiosis (cooperative case). The special case of the Virtual Fishtank, an idea based upon an interactive museum exhibit proposed by David Greschler and Mitchel Resnick (Greschler and Resnick 1995) cuts across several user-to-turtle mappings on the hybrid axis. Users may program in various behaviors into the virtual fish, then release their creations into a shared tank to see how they interact with fish created by other users. Fish may work together (schooling), may ignore one another, or may compete (say, eat one another), but the exact direction the Fishtank takes is up to the users. When dividing the fish in the StarLogo Fishtank, each user could own a breed of fish, to be programmed en masse and interact with other breeds, they could own a few fish, programmed individually to interact with other fish, they may work on a fish in a team, or they could each program a behavior for a breed or for all of the fish. The extreme case is when all users can control all aspects of all fish; this I have labeled on the diagram as the "unlimited exploration" case.

Other special cases might include situations based on Loren Carpenter's group demonstrations of self-organization (Kelley 1994). In these demonstrations, each person in a group has a colored paddle, and by turning it between the green and the red sides, they can change their "input" to the overall task at hand. The input may change a single pixel on a main display, or it may increase the likelihood of moving a controller in different directions. Group tasks included forming various pictures (as when people create one huge picture at a football game by holding up colored cards), playing Pong (akin to simple computer tennis), and flying an airplane (where different segments of the room “voted” on flap and rudder controls). In Distributed StarLogo, for example, one could let all users control all turtles, and have the users as a group perform a given task (such as creating pictures or grouping together turtles of different colors) or compete with one another (e.g. Pong). When each user controls a single behavior of the turtles, users may decide to work together to build a shared world, such as the StarLogo anthill (Resnick 1992), or their behaviors may compete for control of the turtles (reminiscent of Marvin Minsky's Society of Mind model (Minsky 1986)).

A different twist on the distributed turtle grouping task might involve each user or team of users owning some subset or all of a color of turtles, and all users with like-colored turtles then must try and bring them all together. This is slightly different from the other grouping example in that users are less likely to unwittingly oppose each other, plus it is a good example of how a single task can be reinterpreted from a different perspective, providing fertile ground for the development of other new ideas.

Each January, MIT students may take a class known as "6.270", in which the object is to build and program an autonomous Lego robot to compete one-on-one versus other students' creations in some competitive task (such as gathering all the balls in a defined area) at the end of the month. With each user or team of users controlling a given number of turtles, one could imagine a 6.270-like contest being run within Distributed StarLogo. Instead of building a physical robot, however, students could program their turtles' behaviors and then let them compete with one another.

3 Design

The personal and social consequences of any medium (extension of ourselves) result from the new scale that is introduced into our affairs by each extension of ourselves, or any new technology.

- Marshall McLuhan (1964)

The process of Distributed StarLogo's design can be broken up into three distinct phases: goal identification, where the high level goals of the system were identified; issue identification, where the various problems and likely choice domains of the system were identified, and the actual decision phase, in which the various options for the final system design were explored and the consequences of each choice was weighted accordingly.

3.1 System Goals

In creating the design for the Distributed StarLogo system, I was influenced by several goals that helped to guide my thought about how the final product should operate, and perhaps more accurately, to “feel.”

Generality

First, the system should be as general as possible. I decided that since StarLogo is primarily concerned with decentralization, the operating metric for this criteria should be the level of decentralization present in the design. Much like Dee Hock in his quest to decentralize the structure of the Visa International banking services corporation (Dougherty 1981, and Hock 1994), this goal led me to try to “push power to the edges” as much as possible. This led me to consider using a “client / client” computation model, and to shift the control and responsibility in the final “client / server” model as much toward the clients as possible.

Flexibility

Second, the system should be as flexible as possible. To satisfy this goal, I had to consider many possible applications of the finished system as previously described, and to think about what types of functionality would be needed to be able to create such applications.

This led to design choices that allowed the system to be able to cover as many implementations within its conceptual framework as possible.

Non-intrusiveness

Next, the system should be non-intrusive. That is, the fact that the user is sharing the system with a group of other people should not adversely affect the ease with which projects are created. The heuristic I used here was to try and do as much "add-on" work as I could when enabling the system's distributed functionality, while trying not to change the underlying interface to the StarLogo programming language itself, as it was already very well-designed and well thought out.

Ease of Use

Finally, the system should be easy to use. A system that no one understands or that requires a degree in rocket science to operate is useless in the educational context StarLogo is geared towards. This also ties in with the previous goal, as the original system sports a very nice user interface that I attempted to preserve as much of as possible. It might be most appropriate to say that, while extending the system, I attempted to do as little harm to it as possible.

3.2 Requirements

In creating a design such as this, there are many issues and choices that will inevitably arise. In my case, I was led into some design issues through careful examination and thought about how the system would be viewed by its users, and I simply fell into other issues due to the existing StarLogo implementation and the inherent conflict of trying to implement a very *decentralized* system within a very *centralized* computing environment. A discussion of some of these issues follows.

Turtles

StarLogo is comprised of turtles, so it was natural for me to think about how the turtles should work in the distributed version. Users may want to each have their own turtles that are completely within their control, and interact only between different turtles. Others may want to share turtles among several users, or to share all turtles equally. But what does it mean exactly to "share"? Of what are the turtles composed of? After some thought, it seems as though, in the StarLogo model, each turtle breaks down into two simpler pieces: its *state* and its *action*.

State

Each turtle's *state* essentially consists of all of its associated built-in variables, namely position, heading, color, pen state, visibility, plus all of its user-defined turtle variables. For simplicity, I just considered the simplest case of sharing, which is whether a variable is private (only the owner can change it), or public (anyone can change it). What is needed for some amount of flexibility in the turtles' state division is a mechanism to specify for each variable whether it is public or private. That way, users can specify whether anyone can change anything about a given turtle, nothing about a given turtle, or something in between.

Action

A turtle's *action* is whatever it is doing at a given time. Given the parallel model of StarLogo, a turtle could be performing multiple actions at the same time (running demons and responding to user commands, for example). In some sense, processes that are running simultaneously are *sharing* the turtle. While it would be nice to somehow specify relative amounts of processing time allocated for each demon and user command, that is not a feature of the current underlying implementation of StarLogo. Again moving back to a simpler case, for a given turtle there should at least be some method for specifying which other users can affect a given turtle's action, and in the simplest case, whether all users can affect a given turtle's action, or if only the turtle's owner is allowed to affect it.

Patches

The same questions arise for the patches that do for the turtles - how can they be shared, and what aspects of that sharing should or need to be specified? Patches are simplified in the facts that there are a fixed number of them and they are all immobile, at least in the standard version of StarLogo. This also complicates them, though, since any sharing specification would need to be defined over all 10,000 patches for every exploration, besides the fact that they *are* the environment over which the turtles roam and through which they interact. For generality, though, it seems like it might be desirable to have a method whereby the sharing attributes of patch states and actions could somehow be specified.

User Code

One aspect of the StarLogo model that is transparent in the single-user version but becomes immediately visible in the distributed case is the StarLogo code itself. Much like the turtles and patches, there needs to be some mechanism for determining which parts of the code are affectable and usable by which users, and which are not. Perhaps a user wants his code run by himself only, or perhaps also his close friends, but no one else, and only he can modify it. Or maybe he adheres to a more progressive philosophy, and he wants his code to be both modifiable and executable by all users. Maybe he doesn't even want anyone else to *see* his code. These are all issues that have been resolved most generally in the UNIX operating system, where file access is controlled by read, write, and execute privileges over a file's owner, user-definable groups associated with the file, and all other users (Bach 1988). Even greater control is afforded by the Andrew File Sharing system, where access can be controlled on a per-user basis (Zayas and Everhart 1988). Perhaps this functionality is overkill for an essentially collaborative system, but it is still an issue that needs to be taken into consideration.

Patch Environment

Remembering the lesson of levels emphasized in Resnick's original work on StarLogo, it is instructive to realize that the patches themselves exist in a topological environment of their own. In the standard version of StarLogo, the patches are arranged in a 100 x 100 grid with wraparound (a turtle that goes off one edge appears in the corresponding position on the other edge), a formation formally known as a *torus*. As it is possible in a distributed implementation of StarLogo to computationally map regions of the patch environment to logically distinct computers (different computers can perform the computations for different parts of the patch environment), the question of exactly how this mapping gets done becomes important. On first glance, it seems like the best overall performance will be afforded by the most direct mapping possible from the StarLogo world to the physical world, but the issue clearly needs further thought.

Computation

The complement to the patch environment issue is that of where the computation actually takes place in Distributed StarLogo. Should the computational distribution be static or dynamic? Equally allocated among the number of connected machines, or allocated in proportion to their available memory and computation resources, or according to some other scheme? How should new connections affect the environment's topology? Perhaps the most important question relating to this issue, though, is how many complicated mechanisms should be built into the system to support these ideas before their cost begins to outweigh their advantages?

Display Control

Another issue that arises, especially when coupled with the reality of limited network bandwidth, is how much control over the final display should a given local machine have? And how should that display be set up and supported? A local user may want to set his screen to display all of his turtles in red and all others in blue. Or he may want to do some other type of display remapping; maybe he only wants to look at a small portion of the world, or maybe he wants to see everything. In order to display the results of a local computation on a global set of screens (or to at least make that information available for display), all the relevant data must be sent by all computational hosts to all display hosts across the network. This includes data for each patch and turtle and their associated states, in addition to information about the overall patch topology and all of its changes, assuming we want to allow the users to be able to see a global picture of the entire environment. How much information can the available bandwidth handle? What is the best trade-off between speed and the ability to control display parameters locally?

Communications

In a tool such as Distributed StarLogo that is intended to foster communications and collaboration among users, it becomes necessary to create both a communications policy and a communications infrastructure, each of which plays off of the other. What types of user communication are most effective for Distributed StarLogo's purpose? What kinds of communication constructs can best foster those types of communications? How can these constructs be kept as non-intrusive as possible?

A second aspect to the communications conundrum is how to most effectively facilitate the creation of a "community" of StarLogo users, much like has been done with various MUD and MOO environments (e.g. Dibbell 1993). New users should be able to easily get help with the system, and experienced users should not grow frustrated with a lack of available resources appropriate for them, a delicate balance to say the least.

3.3 Choices and Compromises

As I worked my way through the design and implementation of Distributed StarLogo, I found myself filled with ideas, but often hampered by their scope and size relative to the intended scope of the thesis. Often, what I wanted to do would best be done by throwing away the entire system and starting over, if not from scratch, from something very close to it. As this was not possible, many of the implementation decisions were heavily influenced by the existing StarLogo implementation and the ease with which I could retrofit the desired functionality to it. My true goal was not to complete the "ultimate Distributed StarLogo" package, but to simply take another step in the right direction, as much to show what is possible and useful in terms of the distributed model as to create the actual system in its own right. I certainly hope that I came reasonably close to achieving that goal.

Computation and Patch Environment

The first implementation decision I faced was how to structure the computation in the new StarLogo environment. As I discussed it with my advisor and other people involved with the original StarLogo package, three classes of possibilities quickly presented themselves.

The Distributed World

In this proposal, the world would be organized much like a patchwork quilt. Each local machine would join the "StarLogo world," perhaps by consulting a central server to point to other machines that would be it's "neighbors" one each of the four sides of the local screen. Anything on the local screen would be the responsibility of the local machine to keep track of, and when a turtle moves off an edge of the screen, both it and the code it is executing is sent to the appropriate neighbor for further execution, the local machine simply "forgets" about it.

Advantages

The great advantage of this model is that it carries the "distributed mindset" to an extreme. The computer only concerns itself with what is nearby and doesn't care what is happening outside of its own local area.

Disadvantages

Unfortunately, while it would appear simple to cause turtles who would have “wrapped” around the screen to be sent to another server, the code that would have to control that is deep in the bowels of StarLogo’s assembly language segment, which controls all turtle and patch process executions. This code would have to be heavily reworked and retuned for acceptable performance levels. In addition, migrating code from machine to machine is no small task, either. There are many other “fuzzy” issues involved, too. Who controls non-local turtles? How can you tell what your turtles are doing once they leave your screen? How do you tell them to stop what they’re doing, or tell them to do something else? Or are they completely autonomous? If so, do they ever die? How? This option, which on a first look appeared attractive, ended up being completely unworkable for the purpose of this thesis.

The Complete World

The complete world model is basically a simple client-server model. Clients can connect to a central server, send and receive code and messages to other users, and receive picture display information. All code is run on the server, and the clients simply serve as relatively dumb terminal interfaces to the remote host.

Advantages

This model seemed most achievable for the purposes of this thesis, as it did not seem to have quite as many possibly insurmountable obstacles impeding its creation. Most work could be done at a high level, freeing the implementation to "black-box" the assembly-language portion of the code, greatly speeding the development process. Also, the original interface code was written in a fairly modular way, which lent to its easy reuse and modification for this project, preventing the new system both from needing to be completely rewritten and from it ending up as a giant hack that was completely incomprehensible, unusable, non-extendible, and unmaintainable.

Disadvantages

This model, while being on the leading edge of decentralization in software as far as actual commercial production systems are concerned, still falls far short of the original lofty goal of a completely decentralized environment for StarLogo. The server would need to be very fast in order to properly perform the necessary computations, and the network bandwidth would hit a serious bottleneck at the main server with the large amount of screen data needing to be sent from it to each client.

Hybrid Worlds

The idea of the hybrid world is, as the name implies, a combination of the distributed and complete models. The most distinguishing of these models is one in which each client controls all turtles it creates, as well as all patch variables in its local 100 x 100 grid. The turtles could then be off of the local grid while the local machine is still keeping track of them and running their code. Whenever there is any non-local patch or any turtle interactions, a message is sent to a central "dispatch server" that either routes the request to the proper local machine, or sends that machine's address back to the originating client for distributed processing. Each local machine would be individually responsible for responding to all such requests, just like in the completely distributed case.

Advantages

This model seems to embody a "best-of-both-worlds" approach to the distributed design. Most of the computation and control is kept on the local machines, while centralization is avoided as much as possible..

Disadvantages

Again, however, the practical necessity of performance that forced the original StarLogo design towards assembly language has hindered the efforts of extending it without a disproportionately large amount of effort. This type of extension would be possible, but again, the resulting code would end up as a "kludge" (computerspeak for an incoherent mess that just happens to do what it's supposed to) that would not confer the benefit of a basic platform to build on to future system designers.

The Centralized Mindset Wins Again

While it blatantly flies in the face of the spirit of the original StarLogo package in many ways, the centralized server model seemed to be the right way to go for this project. By avoiding development at the assembly code level, I have tried to avoid passing along the spectre of low-level code to future developers as much as I could. Also, one of the main goals of the original StarLogo package was to help people explore and think about decentralized systems as well as shifting mental paradigms from the "one" to the "many," and I think that this apparently centralized choice continues StarLogo's tradition in these senses. From the user's perspective, this is probably a *better* choice than the others, since it does not limit his ability to view or change the system in some of the ways the other systems might have. He will simply see many people working together on the same StarLogo canvas, and generally won't realize what a kind of "deal with the devil" had to be made in order to realize that situation.

Interlude

Once the computation model was settled on, many of the other issues came into much clearer focus. In addition to the domain-specific issues they raise, all these design choices can be evaluated with respect to the space of possibilities defined by how a given parameter specification is allowed to be changed in the final system. On one axis, we examine whether the specification will be built in to the system, or whether it will be user-definable. On the other axis we have the possible rate of change of the parameter: is it a very static specification, one that is made once and then remains forever (or at least until the power goes out), or can it be changed dynamically, on the fly? Figure 2, below, gives an idea of how I tried to visualize this space.

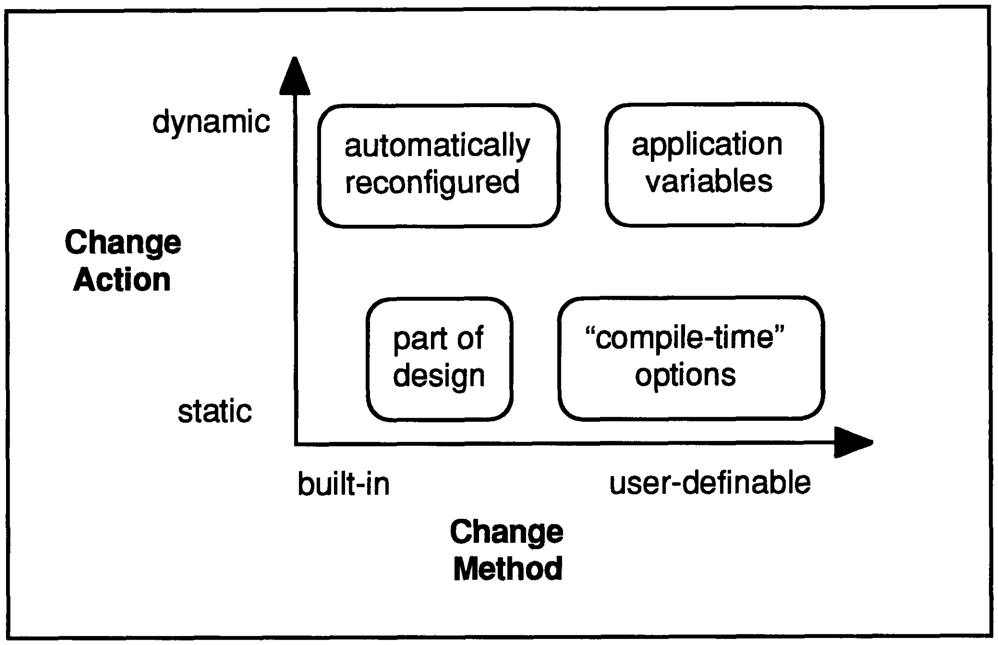

Figure 2. State-space for StarLogo functional specifications.

For example, if a system's designer makes the decision as to how a given functional specification will work and builds it into the system, that parameter can be seen as residing in the lower left quadrant of the graph. If the designer builds in enough intelligence so that the server can automatically reconfigure itself for changing conditions (as in the issue of computational division), that would tend towards the upper left quadrant. If the user can only define a given specification once, or if it is very cumbersome to change (akin to "compile-time options" in some other languages such as C), those functional specifications would be in the lower right of the graph, and finally, those parameters that the user can both define himself and change in the course of a program's execution are in the "dynamic, user-definable" upper right quadrant of the space. Regular StarLogo's variable declarations would then fall somewhere between the top and bottom on the right-hand side of the graph, as they can be changed within the scope of the running StarLogo programming framework, but not within the scope of a given user's individual, running StarLogo program itself without forcing a recompilation.

The design goals of generality and flexibility both can be viewed as forces pushing the design choices up and to the right of the diagram, but their inherent complexity involved in their implementation kept pulling them back toward their original static, system-defined state at the origin. Like in any complex system, I did not try and find the "ideal" value for each of these parameters, instead, I attempted to find the best "middle-ground" within which, given the constraints present, the system's development could most effectively proceed.

Patch Environment

Since the Distributed StarLogo system operates in a client / server fashion on top of the original StarLogo, the possible issue of patch environment topology becomes a non-issue: The environment simply remains a 100 x 100 toroidal grid. This is another good example of issues that arose in the design of the distributed system that were completely transparent (and "obvious") when viewed from the single-user mindset but became non-trivial when reexamined in the distributed context.

Turtles

Looking at the original StarLogo's implementation, I realized that although it would be reasonably simple to do some preprocessing of code sent to the compiler for turtle and patch execution to facilitate state sharing specifications, it would be difficult if not impossible to do the same thing for turtle and patch actions. Hence, I made the decisions to focus exclusively on StarLogo's state variables and their sharing specifications, and to allow the users much less latitude in the action specifications.

The way turtle state sharing specification works in the final system is that the user, in the variable declarations at the start of his code, indicates which turtle variables are to be private by using the original "turtlesown [foo bar]" syntax, and then specifies public turtle variables with a new declaration, "turtlespublic [bat baz]". Any thusly declared variable is then open to all users, and anyone can modify the value in any way they choose at any time. Private variables are modifiable only by the turtle's owner, or by other turtles that belong to him.

By using the phrase "turtles that belong to him," I am implicitly declaring that turtles, as well as variables, are somehow owned by StarLogo users. This is a basic form of action sharing specification, done automatically by the system at runtime. Any turtles created by a user with the normal "crt n" command are private turtles, belonging exclusively to him. Only public turtle variables are able to be changed by patches and other users and their turtles. On the other hand, users may create public turtles with the new "crtpub n" command. These turtles' actions and their state are able to be affected by any user at any time through any means whatsoever, be it an action or a state command.

Patches

In thinking about how sharing should work in relation to patches, I found myself wondering what exactly would it mean for a patch to be "private?" It is, after all, a part of all the turtles' environment and it didn't seem to make much sense that a turtle wouldn't be able to interact with its local or even non-local environment at any given time. I found myself asking if that was really in the spirit of the original StarLogo, and decided that the answer is really *no*. As most turtle interactions happen by using the environment as an intermediary, this could effectively block turtles from communicating with one another, a central StarLogo concept. This though process highlights what has turned out to be another lesson of this thesis - what is left *out* of a model is just as important, maybe even more so, than what is kept *in*. In the real world, anything you leave lying around is fair game for anyone to wander along and fiddle with; so it is in Distributed StarLogo.

Shared Code

The design decision I made regarding shared code in Distributed StarLogo was that users should enter their own code into their local machines, only after which is all of their code uploaded to the server, compiled, and made available to users for execution. In this case, once the code is uploaded, it is “out of the user’s hands” in some sense, as it then resides on the server, which makes the decision of how then to deal with code sharing all the more important.

The conclusion I finally came to basically involved sidestepping the issue entirely. Yes, it might be nice to be able to allow others to modify code that belongs to you, but there is an inherent increase in the language complexity that comes along with such a feature which seemed to be too high a price to pay in this case. My final design allows for users to download the entire code library at any time, and to run any of the users’ code on their own private turtles or on any of the public turtles. When a user leaves the system, so does the body of code he brought with him. This prevents the server from becoming cluttered with old code that no one is currently using, but still allows other users to archive others’ useful code and to reload it should those people decide to leave the system.

This design also shifts the burden for enabling collaboration between users towards the person-to-person communication end of the spectrum, rather than allowing cooperation through the impersonal means of modifying each other’s code. This is surely of some benefit in a world where parents complain of their children spending too much time on the computer and not enough time interacting with their peers, but in some of those same eyes it may also seem to exasperate the problem when it is viewed as a lack of *face-to-face* interaction. My only reply is that StarLogo is not meant to be all things to all people, and there are always going to be some tradeoffs made in a design such as this. If this package does not meet their needs well enough to be useful, they can feel free not to take advantage of its use.

Display Control

While many ideas were kicked around in relation to a Mosaic-style control of the end user's local display for StarLogo explorations, they all assumed that a high bandwidth was possible between the server and each of its clients, or that enough compression could be achieved to send the necessary amounts of data. Unfortunately, the network interface that was chosen for this project, the TCP/IP (Transmission Control Protocol / Internet Protocol) protocol suite, did not turn out to be up for the task. Even sending trivially small amounts of data across a connection of a few feet took over one-third of a second. This pretty much eliminated most of these ideas for contention, as it took heavy compression and code optimization to simply send color change information across the development room at that rate. That frame rate is reasonable, but it will still take some major improvements before the smooth quality of the original version is equaled. Unfortunately, this was one of the less desirable tradeoffs that had to be made in the completion of the project, as in a visually-oriented system such as StarLogo, the display speed and smoothness are important factors in users' enjoyment and comprehension of the overall package and in the learning gained from it.

Communications

The final design issue to be resolved related to the portion of the communications interface for the system visible to the end users. This was probably the most important decision to be made, especially due to the fact that it would take most of the burden for enabling Distributed StarLogo users to collaborate and to share thoughts and ideas about Distributed StarLogo, in conjunction with the system's built-in code handling facilities.

In today's chiefly text-oriented computer communications medium, there are essentially two models on which to model a communications structure: the MUD or MOO model, and the Usenet News model. Their key features are summarized below.

MUD / MOO Model

In this model, users connect to a central server, much like in the Distributed StarLogo system. There they can type in commands that immediately appear on all or some of the other users' screens, as well as being able to create new locations and actions (or "verbs" as they are called in MOO parlance). Users are located in virtual "rooms," an apt name since that is exactly how MOO interactions tend to feel: very informal and chatty, much like a group of people gathered together in a room talking with one another.

Usenet News Model

Usenet News, on the other hand, is modeled on a more asynchronous method of communicating with other users: the bulletin board. Users may "post" messages to newsgroups, which then appear in a listing along with all the other currently active messages on the group. Other users then select which messages they want to read from that list. This model doesn't quite map onto the StarLogo model as well as the MOO metaphor does, but it has some valuable features nonetheless, such as greater message permanence and built-in archiving features.

Bessie, Come Home!

Although it might be valuable to be able to store some older messages for community access in Distributed StarLogo, I felt that the great majority of user interactions would be of a more ethereal, transient nature, and thus would not necessitate such a feature to be built directly into Distributed StarLogo. Other, external media, such as the real Usenet News system and the World Wide Web, are much more suited for such types of interactions.

In addition, the MOO model of interactions simply fit more closely with the underlying constraints of Distributed StarLogo. MOOs are built on a synchronous model of communications, as is Distributed StarLogo, while bulletin boards are more of an asynchronous concept. Coupled with the fact that the unique characteristics of newsgroups, message permanence and archiving, were already available elsewhere, I felt that it would be an acceptable design decision to allow users the freedom to choose their own, external discussion forum in cases where asynchronous communication or message permanence were required. I also hypothesize that as the community of Distributed StarLogo users grows, the availability of information accessible in regards to its use outside the scope of the package will increase dramatically. Already, there is a mailing list for the users of the original system, which distributed users can easily tap into, and I feel that this trend will only continue.

4 Results

For me, a long and arduous software development task has ended, but for others now and to come, that is not the case. Here I hope to extend some of my insights and experiences to others in the hope that they may be able to glean some sort of useful knowledge from them.

4.1 Lessons Learned

I learned many valuable lessons, both technical and personal, throughout the course of this project, way too many to list here. Here are a few of the perhaps more relevant ones:

Challenge Assumptions

One of the valuable lessons I found in developing this system was danger of making any early assumptions. When I was on my way toward making some such incorrect assumption, I was fortunate enough to have very helpful and knowledgeable colleagues who were able to steer me back on course before I went astray. One bad assumption I did make, however, was in the case of the assumption that I should use TCP/IP as the underlying protocol suite for network transactions simply because it is the de facto standard. As it turned out, that choice was one of the most limiting factors in the final system's ultimate performance.

The Value of Documentation

If there was one lesson to be learned from my experience with the implementation of Distributed StarLogo, it was how important proper programming style and documentation is for people who have to deal with maintaining and extending a given set of code at a later date. If it were not for the valuable help I received from StarLogo's original architects and implementers, Distributed StarLogo would have taken a much longer time and more hard work than it already did. As such, I have attempted to make my code as accessible and as straightforward as possible, and I hope that I will have made the way a little clearer for those who will follow me.

Perspective Matters

Several of the design issues that arose in the development of Distributed StarLogo would never have been issues or even really considered in a single-user context. The idea of the code as an object needing to be shared and managed and the fact that patches exist in their own environmental topology would probably not have been as extensively considered when the system was only focused on a single user. In addition, some of the design decisions that appeared to be suboptimal when evaluated with respect to one metric turned out to probably be the *best* choice when seen in another light, underscoring the need to look at things from as many points of view as possible before their full implications can be fully comprehended.

The Importance of Omission

A final lesson that I noticed time and time again throughout my development of Distributed StarLogo and in creating this thesis was that the things you leave out of a system can be just as important, if not more so, than those things you decide to keep in. Some of the most important design decisions were omissions rather that inclusions: the lack of privacy for the patch variables, the impermanence of user communications and the lack of a comprehensive built-in code sharing facility are the three that become immediately apparent. Shifting to another level, it would not have been possible to develop the system to the point it is today without many omissions of production code "niceties" and other features that would have increased the scope of this project beyond a reasonable level, another example of this principle in action.

5 Conclusions

The Distributed StarLogo package is just another small step along the road to an elimination of the dominance of the "Centralized Mindset" in people's systems learning experiences. There are still many other aspects left open for future exploration and research. Here I touch on just a few of these opportunities and leave the reader

5.1 New Features

When using any new product, who doesn't get the urge to say "Well, that's nice, but if it only did *this*..."? I have already talked about most of these features, but they are listed here in the approximate order of importance I would give to their further development.

Faster Video

As mentioned earlier, one of the most limiting factors in this project was the slow networking speed achieved with TCP/IP in MCL (Macintosh Common Lisp). I was barely able to achieve 3 frames per second over a span of a few feet. Admittedly, the need to loop over all 10,000+ patches each time step to detect color changes was one limiting factor, but the overall data flow rate was not significantly altered by sending the same amount of predetermined data. Options for correcting this situation include rewriting the MCL TCP class wrapper, or migrating the communications function to another platform altogether, such as UDP (Unreliable Datagram Protocol), but there is some research to be done to determine which of these routes would be the most effective, especially for a given limited amount of available effort.

Client Display Control

With faster video networking will come the ability to send more display data across the network with each frame. A useful change might be to send both turtle and patch states across as the display information, which the client can then choose to display according to which turtles are his, or which internal turtle and patch variables he is interested in, or according to some other algorithm altogether. This is much like the philosophy behind the World Wide Web: send the underlying structure to be displayed, then allow the client to decide for himself how that structure should appear on the screen, a strategy that resonates well with StarLogo's natural bent toward decentralized power and control.

Faster / Incremental Compiler

Another problem with the Distributed StarLogo prototype is that whenever new user code or demons arrive at the server, the entire simulation must shut down, load the code, and recompile. It would be very useful if the compiler were "smarter" in acquiring and loading new code; the exploration could continue uninterrupted while these changes were being made. In addition, the network lag combined with the inherent care with which the compiler takes to receive and load code make these uploads take significantly longer in the distributed version than with standard StarLogo, a situation that could be corrected either algorithmically or though the acquisition of faster hardware.

5.2 Summary

The original StarLogo system was based on two separate and very powerful ideas: that helping people learn about distributed systems is an important aspiration, and that people learn most effectively by constructing personally meaningful artifacts within the domain being learned, what Seymour Papert called *Piagetian learning* (Papert 1980).

The motivation behind Distributed StarLogo was a combination of these two ideas: Papert asserts that such Piagetian learning takes place most effectively in environments and cultures that are rich in the building blocks needed for such knowledge to be assimilated. By helping add a distributed context to StarLogo, I have tried to extend the environment a small step in that direction, but there is still a long way to go. Opportunities for further investigation and development will surely abound in the near future for researchers in all fields interested in the overreaching paradigm shift towards decentralization and self-organization that StarLogo is emblematic of, especially if people's initial reactions to Distributed StarLogo are any indication. As Hock has noted about such decentralized constructs (Dougherty 1981), "It proves the old saying that nothing is as powerful as an idea whose time has come."

References

1. Bach, M. (1986). *Design of the UNIX OS.* New York: Prentice-Hall.

2. Bruckman, A. (1994). *Programming for Fun: MUDs as a Context for Collaborative Learning.* FTP file from: ftp://media.mit.edu/pub/asb/papers/necc94.txt.

3. Dibbell, J. *A Rape in Cyberspace.* From Village Voice, December 21, 1993.

4. Dougherty, J. (1981) *Visa International: The Management of Change.* Boston: HBS Case Services. Harvard Business School Case Study #9-482-022.

5. Greschler, D. and Resnick, M. (1995). *The Virtual FishTank.* Proposal to the National Science Foundation and the Informal Science Education Program.

6. Hock, D. (1994). Personal communication.

7. Kelley, K. (1994). *Out of Control.* New York: Addison-Wesley.

8. McLuhan, M. (1964). *Understanding Media: The Extensions of Man.* London: Routledge.

9. Minsky, M. (1986). *Society of Mind.* New York: Simon and Schuster.

10. Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas.* New York: Basic Books, Inc.

11. Resnick, M. (1992). *Beyond the Centralized Mindset: Explorations in Massively Parallel Microworlds.* Epistemology and Learning Group, MIT Media Laboratory, Cambridge, MA.

12. Resnick, M. (1995). *Getting Started With StarLogo.* Epistemology and Learning Group, MIT Media Laboratory, Cambridge, MA.

13. Turkle, S. (1984). *The Second Self: Computers and the Human Spirit.* New York: Basic Books.

14. Zayas, E. and Everhart, M. (1988). *Design and Specification of the Cellular Andrew Environment.* Information Technology Center, Carnegie-Mellon University.