Table of Contents

Acknowledgments

First of all, I would like to thank those that have helped me with this project: Melvin Cherno, James Cohoon, O.A Gianniny, Randy Pausch, Pavel Curtis at Xerox PARC, the MIT Media Lab, the LPC-Centre and Jason Curnow, and Seth Rich. Thank you all; I could never have done this without you.

To the other members of our Computer Science department who have helped, pushed, and inspired me, I thank you: John Knight, Tim Sigmon, and William Wulf.

To my close friends, some of whom have suffered through their senior theses with me, thanks: Aaron Cass, Dennis Cosgrove, Todd Hodes, Scott Witherell, Michael Long, Rich Moraski, and Kayte Stasny.

To the person I could never thank enough for her patience and love: Sandra Joan Kwitnieski. Thanks for putting up with me, man.

And finally, to my parents, thanks for supporting me, even when you weren't sure what I was doing. You're amazing.

Abstract

Virtual reality is one of the fastest growing fields in modern computer science. However, it is still limited by hardware. Another type of virtual environment, this one *text-based*, is gaining in popularity. In such an environment, all interaction is done via text, including descriptions and user commands. Hundreds of text-based multi-user domains, or MUDs, are currently up and running on the Internet. Most of these systems are purely recreational, however, research is being conducted on non-game applications of MUDs.

Most existing MUDs have been designed in an ad hoc manner, without a great deal of planning before coding began. MUD designers need a generalized global set of MUD concepts and features that they can examine before coding begins. This project creates a set of "building blocks" -- some low-level, some high-level -- for designers of text-based virtual realities. These blocks are determined by studying three well-known types of Internet MUDs: DikuMUD, LPMud, and LambdaMOO. The specific implementations of the blocks on the three subject MUDs is discussed and analyzed in order to determine which methods and concepts seem most effective.

Building Blocks of Text-Based Virtual Environments

G. Scott Evans

Undergraduate Thesis
Computer Science Department
University of Virginia

1. *Introduction*

1.1 MUDs and Virtual Reality

Virtual reality is a rapidly growing field in modern computer science. The concept of immersing a user in a virtual environment where he or she has powers that are bounded not by the real world, but by a computer-defined one, is intriguing.

However, graphical virtual realities are greatly limited by modern hardware. They depend on highly detailed, realistic graphics drawn in real-time. Such graphics are extremely difficult to create, especially at a reasonable cost. Furthermore, creating the "worlds" themselves is a fairly involved task; simply adding a desk to a room may take hours of drawing with a *CAD program*. Finally, adding functionality to objects is difficult as well. Even though the desk is visible in the room, the user may not be able to actually do anything with it. These problems become increasingly difficult if the idea of *multiple users* is introduced. Therefore, most of today's virtual realities support only single users, which is less desirable than the ability to support many users.

CAD: computer-aided design. CAD programs allow the user to draw objects in three dimensions.

multiple users - more than one user inside the environment simultaneously. In graphical virtual realities, this greatly increases complexity and running time.

Many of the computational problems associated with graphical virtual realities do not apply to a **text-based virtual environment**. In this type of environment, a user receives descriptions through text instead of graphics. Furthermore, all commands (such as commands for movement or communication) are given via text commands rather than by motion of the body.

As an example, consider the problem of adding a desk to a virtual room. In a text-based environment, adding a desk is simply a matter of writing a few sentences to describe it. Adding functionality is slightly more difficult, but with a well-designed system it can be done in a fairly short amount of time. Most importantly, it is much easier to support multiple users. One of the main limitations of graphical virtual realities is that their performance is limited by the number of lines or polygons that hardware can draw in a given amount of time. Since handling multiple users can mean that users see each others' movements, the amount of computation increases exponentially as users are added. However, processing a fairly small amount of text requires much less CPU power and can be done easily with modern workstations. All that is needed is a facility that enables users to connect to a remote *server*. The server handles the connections and processes user commands, responding appropriately. This connection can be made with a modem or, more commonly, through the *Internet* (see Figure 1). The term "virtual environment" seems more appropriate than "virtual reality" since the goal is not to fool the user's senses into believing that they are in another reality; the goal is to provide a symbolic environment that the user can interact with and affect.

server - a computer that accepts connections from other computers and runs the virtual environment software. Also see *client*.

Internet - a worldwide network that provides services to universities, research institutions, and corporations. Most electronic mail travels via the Internet.

These are the basic concepts behind a Multi-User Domain, or **MUD**. As Pavel Curtis puts it, "A MUD is a ... multi-participant, user-extensible virtual reality whose user interface is entirely textual" [Cur92]. In a MUD, users can interact with each other and affect their surroundings, all in real-time, using text commands like "get the cup" or "say 'hello' to Bob". Curtis gives this accurate description of a MUD:

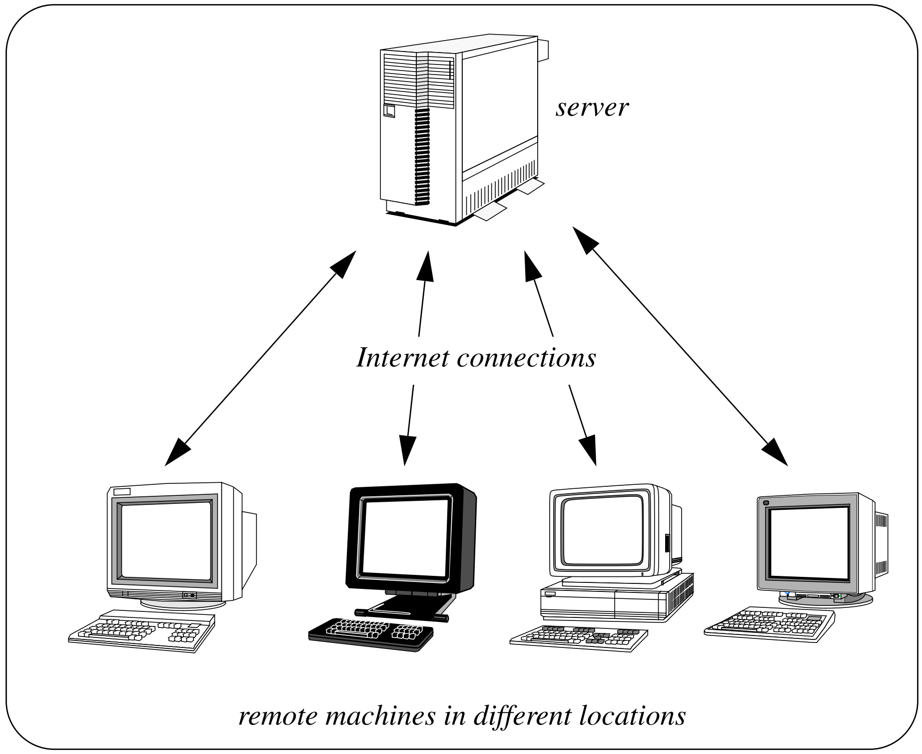

Figure 1. Multiple remote users connected to a single server

> A MUD is a software program that accepts 'connections' from multiple users across some kind of network (e.g. telephone lines or the Internet) and provides to each user access to a shared database of 'rooms', 'exits', and other objects. Each user browses and manipulates this database from 'inside' one of those rooms, seeing only those objects that are in the same room and moving from room to room mostly via the exits that connect them [Cur92].

Figure 2 shows an excerpt from a MUD session. The MUD involved is MediaMOO, running at M.I.T.'s Media Lab (see section 4., "MUDs Chosen for Study", on page 11) [Eva93c]:

```
>> look

The E&L Garden
The E&L Garden is a happy jumble of little and big
 computers, papers, coffee cups, and stray pieces
 of LEGO.
```

```
Obvious exits: ‘hallway’ to E&L Hallway, and ‘sts’ to
 STS Centre Lounge
You see a newspaper, a Sun SPARCstation IPC, Projects
 chalkboard, MediaMOO Map here.
Ron, Wilson, Inui, and Michele are here.

Michele says, “Hi, Scott.”

>> say Howdy!
You say, “Howdy!”

Inui says, “I’m new here.”

>> ‘Inui Nice to meet you. Do you do MUD research?
Scott [to Inui]: Nice to meet you. Do you do MUD
 research?
Inui says, “I’m a sociologist who’s interested in
 VR.”

Wilson [to Scott]: I actually got rejected from
 LambdaMOO today when I tried to connect.
Wilson says, “It had 100 people connected (that is
 the current limit).”

>> emote looks surprised. “Wow.”
Scott looks surprised. “Wow.”

>> say Physically, where is everybody?
You say, “Physically, where is everybody?”

Ron is in Albuquerque, NM.
Michele is in Cambridge, Massachusetts.
```

Figure 2. Excerpt from a typical MUD session

In this example, there were four characters in the room. The four users were physically in different parts of the country, but were interacting in real-time. It is this multi-user aspect that makes MUDs truly unique.

2. Background

2.1 History of MUDs

Some of the first popular text-based environments were textural adventure games like Adventure and Zork, which were widely used by personal computer owners. The first known public multi-user game was created in 1979 at Essex University by Roy Trubshaw and Richard Bartle [Bar90]. The multi-user feature of the game attracted many players, and soon others began writing their own MUDs.

2.2 Current MUD Usage

2.2.1. Recreational

Currently, most MUDs in existence are purely recreational; many are modeled after the "Dungeons and Dragons" game or other role-playing games that model physical attributes with numerical measures in a fantasy setting. Other MUDs provide a facility for socializing or creating new objects and areas for the enjoyment of the community [Cur93].

Most of these games are connected to the Internet, which allows worldwide access. Some are connected to pay services like CompuServe, allowing users without Internet access to play by modem. In both settings, they are extremely popular; there are hundreds of MUDs currently active on the Internet, and "MUD addiction" is a fairly common problem among players [Bru92].

2.2.2. Professional

Recently, the idea of using MUDs for non-gaming applications was conceived. At places like Xerox's Palo Alto Research Center and M.I.T.'s media lab, researchers are working to see how MUDs can be applied in the workplace as well as examining the social implications that MUDs may have. The following subsections briefly describe some current MUD research.

2.2.2.1. The Astro-VR System

Xerox PARC and the NASA/JPL Infrared Processing and Analysis Center are currently building a MUD system to be used by the international astronomy community. The system will provide services including self-containing electronic mail and news systems, shared links to online astronomical images, facilities for creating and giving presentations (including text and graphics), and public access to standard astronomy programs [Cur93].

The purpose of the Astro-VR system is to enhance communication between astronomers around the world. This may be done through casual conversation or formal, prepared presentations and demonstrations. The system is not intended to be completely text-based, since astronomers "rely heavily on photographs to convey ideas and discoveries" [Cur93]. However, Astro-VR will be based on a MUD and will make great use of its textual interface. For more on this subject, see section 5.10.2., "Beyond Text", on page 31.

2.2.2.2. The Jupiter System

The Jupiter project is another current project at Xerox PARC involving MUDs. However, the Jupiter MUD is being designed to run only within PARC. This gives the designers much more knowledge of the network and allows them to experiment with more a complicated system:

> To a large extent, Astro-VR represents the level of interface functionality that we can easily export to a large number of users, given the current state of the Internet. In an environment where we have much more control over the network and the workstations in use by the participants, we can explore the user of more advanced technologies. Then, as the Internet and users' workstations improve, we can use the lessons learned in our more controlled environment to improve the general lot [Cur93b].

Jupiter will be able to handle some real-time audio and video to allow presentations and conversations to be more natural [Cur93]. Again, this brings more than just text into the environment. The idea is to use the text-based environment as a useful interface to these powerful tools and as a powerful tool itself (see section 5.10.2., “Beyond Text”, on page 31).

2.2.2.3. MediaMOO

MediaMOO is a MUD running out of M.I.T.’s media lab. Its purpose is to provide a forum for MUD researchers to discuss research topics and experiment with ideas. Formal presentations are also occasionally given. This MUD is also interesting because it has provided “virtual links” to other MUDs to allow communication between users logged in to different MUDs (see section 5.10.1., “Networking”, on page 30).

2.3 Literature Review

Unfortunately, there is not much literature to be found on MUDs since virtual reality, especially text-based virtual reality, is a very young research topic. A small number of papers have been published on the social aspects of MUDs, but these are of limited use when discussing the design elements of MUDs. Very few publications discuss the purely technical aspects of text-based virtual environments. Pavel Curtis, a researcher at PARC, has published two papers on MUDs. One discusses social issues [Cur92], while the other talks about the Astro-VR and Jupiter projects [Cur93]. Curtis’ second paper is quite useful when considering design elements, for it gives a glimpse at some of the possible future uses of MUDs.

3. Problem Description and Project Goals

3.1 Previous Design Techniques

As previously stated, there are hundreds of MUDs currently running on the Internet. Most of these systems, however, are variations on a much smaller set of themes. Basically, a small number of popular MUDs have been written from scratch and their source code has been made public. Others then take this code and modify it. Because of this, MUDs are generally referred to by their “type”, that is, the original MUD that they are based on.

The problem with most existing MUDs is that they were usually designed “on-the-fly”, especially when an author is adding features to a MUD that was written by another programmer. A programmer has some ideas for the features of the MUD, but probably doesn’t consider how all of the elements of the MUD will fit together, or even if they will fit at all. This design technique may work in the short term, but causes problems in the long term for a few reasons, including:

- **The source code for the MUD may become difficult to read and maintain.** Most MUDs have tens of thousands of lines of code. To keep this code coherent as a whole, it is important to think about it before writing. This becomes especially important if the code is distributed for others to use.
- **Parts of the MUD itself may be difficult to use.** For instance, an author may realize that an important feature is missing from the design at a point where it is difficult to add the feature cleanly. This can result in features that don’t fit in with the overall feel of the MUD, which greatly reduces usability.
- **Important features may not be present at all.** When an author writes a MUD without carefully considering its requirements, some features may not be implemented at all. This generally causes other users of the code to modify it greatly to provide the needed functionality. This is obviously bad; it means more work for the users of the code, and worse yet, does not allow standards to exist within a MUD type.

3.2 Building Blocks

One of the problems with current MUD design, then, is the lack of a written set of global needs that designers can use before they begin coding. This set -- the "building blocks" of MUDs -- is exactly what this project creates. These building blocks range from high-level concepts to specific low-level details. They are, essentially, the important features and design elements of MUDs in general.This set of building blocks will aid MUD designers in three important software engineering phases -- requirements analysis, design, and coding [Jal91].

3.3 Methodology

This paper presents some of the most important features of MUDs and describes various aspects of these features in detail. When necessary, there is a discussion of various existing implementations of specific building blocks on specific types of MUDs (see section 4., "MUDs Chosen for Study", on page 11), and the effectiveness of each implementation. This enables recommendations to be made regarding which implementation might be better and why.

It is important to emphasize that most of the data in this paper cannot simply be stated as fact, since there are no metrics by which such data is proven. The data is a result of:

- observations of a set of existing MUD types
- discussions with MUD users, researchers, and implementors
- analysis of these observations and conversations

Basically, this paper focuses on three distinct types of MUDs (see section 4., "MUDs Chosen for Study", on page 11), discusses their common characteristics and their differences, and analyzes them. Analyses are given with regard to usability, flexibility, and software engineering practice.

4. *MUDs Chosen for Study*

This paper focuses on three popular types of MUDs: DikuMUDs, LPMUDs, and LambdaMOOs. A recent issue of *The Totally Unofficial List of Internet Muds*, which lists most of the Internet's public MUDs, listed nearly 300 active MUDs on the Internet [Goe93]. The three MUDs studied cover about 60% of those listed. Obviously there are types of MUDs that are not covered by this study, but these three cover a fair amount of the spectrum and share many similarities with other types. The following subsections provide details on each type of MUD chosen for study.

4.1 DikuMUD

The original DikuMUD was developed in C by students at the University of Denmark. It is a combat-oriented gaming MUD with rules based on the Advanced Dungeons and Dragons game, and is one of the most popular types of MUD on the Internet; according to *The Totally Unofficial List of Internet Muds*, there are currently 70 active Dikus on the Internet [Goe93]. Many of these active DikuMUDs have been changed greatly from the original, so Dikus will often differ from site to site. However, most of the original DikuMUD's underlying design constraints and decisions remain for its derivatives.

DikuMUD was chosen for study because it is widely used even though it is fairly young. Furthermore, it is interesting to see what variations have been made to DikuMUD by other MUD administrators. These variations are often good clues as to what design elements MUDs should encompass.

4.2 LPMUD

LPMUD, according to the *Unofficial List*, is the most popular MUD on the Internet, with 95 currently active MUDs [Goe93]. It was written by Lars Penjö in 1990. From a player's point of view, it is usually very similar to a DikuMUD, but the underlying concepts are vastly different. LPMUDs contain their own programming language, LPC, which stands for Lars Penjö-C. Basically, the language is a watered-down version of C with object-oriented features added [Pen91].

LPMUD was chosen for study due to its extreme popularity and interesting design features.

4.3 LambdaMOO

LambdaMOO was written by Pavel Curtis at Xerox PARC in 1990 [Cur92]. Several MUDs are currently running the LambdaMOO server, including M.I.T.'s MediaMOO. LambdaMOO also has its own object-oriented programming language, MOO.

LambdaMOO was chosen for study for a few reasons. It is one of the newest MUDs available on the Internet, and the MUD research community supports it. Furthermore, LambdaMOO is one of the few MUDs that was not written to be a role-playing game. It is, however, *extremely* active; in early 1992 LambdaMOO averaged over 28 players logged on at a time for a good part of the day [Cur92]. Recently, the average has been closer to 60 or 70 users a day. At PARC, LambdaMOO now runs on a dedicated Sun Sparc workstation and its *core database* (see section 5.2, "Persistent Objects", on page 13) is growing rapidly; currently it occupies about 70 megabytes [Cur93c].

core database - a file or set of files that contains the definitions for all objects created online (see section 5.8, "Extensibility", on page 19).

5. Data: the Building Blocks

This section discusses common concepts and design features found in MUDs. When necessary, discussions will include how these concepts are approached for each type of MUD being studied. These concepts comprise the "building blocks" of MUDs, and are given in approximate order of increasing complexity.

5.1 Locations with Defined Scopes

The idea of "location" is one of the simplest but most important concepts incorporated into MUDs. In most MUDs, this translates to "rooms". A defined *scope* is necessary for each room. A user can sense events and objects within this scope, such as seeing and using objects within the room or hearing a conversation. However, the user is not aware of events and objects in other rooms in the environment, except for certain events which occur in a larger scope (see section 5.4, "Real-Time Communication", on page 16).

Without scopes a player in one room could "see" everything in every room -- the source of actions that a user will "sense" is limited to the scope that user occupies. So, as in the real world, a MUD room should actually define a scope.

The Jupiter Project at Xerox PARC implements scopes so that users are able to sense actions in nearby rooms, though they will be less noticeable than actions that occur in the actual scope the user occupies. Since the Jupiter Project will involve networked audio, this means that a user in one room will be able to hear a conversation in an adjoining hallway at a lower volume than the conversation in the room the user currently occupies [Cur93]. This is an interesting idea, and the Jupiter Project should provide details on its effectiveness.

5.2 Persistent Objects

In most MUDs, if a player is holding an object and exits the environment, the object will still be in that player's possession when he or she returns. This is an example of *persistence*.

Another instance of persistence that is even more important involves the environment itself. If the machine that is running the server crashes unexpectedly, will the environment be preserved? That is, will rooms still exist, as well as the objects in them? Will the set of defined users remain the same, or will all users have to re-create their characters?

instance - An instance of an object is a "clone" of that object. It has the same properties as the original object, but they may have different values. For example, Camaros are all instances of a "car", although each individual Camaro may be a different color.

In DikuMUD, the world is defined by an external world file, so the world will still exist after a reboot. However, the locations of objects are also defined in this file, so many objects may reappear even though their original *instances* were moved. Actually, the environment in DikuMUD "resets" after a certain amount of time so that objects are recreated in this manner [Dik90]. This is because, in a game context, a player with an interesting item will probably be very unlikely to let go of it, making the game less interesting to other players. However, in a non-game context, seemingly random duplicate appearances of objects can only confuse users.

The ***core*** holds definitions for all existing objects and commands that have been programmed in the MUD's programming language, or created from within the environment. For more on online creation, see section 5.8, "Extensibility", on page 19.

LambdaMOO and LPMUD do not reset their environments. In LambdaMOO, the current state of the environment, known as the *core* (or *LambdaCore*) [Cur92b] is kept in RAM. It is occasionally "checkpointed", that is, transferred to disk or tape for permanent storage. Thus, the environment is fairly persistent. However, a server crash will cause the loss of any data that is newer than the most recent checkpoint.

LPMUD does not keep the entire environment in memory; it puts the code for all objects on disk. This means that all objects are persistent from the time they are created. While this may be a little slower, it is certainly safer than the LambdaMOO method. Complete persistence, such as LPMUD's, is a desirable aspect of any MUD. For safekeeping, backup facilities can be provided to maintain a copy of the complete database elsewhere, as future versions of LambdaMOO will [Cur93c].

5.3 Transportation and Navigation

If there are many defined scopes in an environment, it becomes important for users to have a facility for movement between scopes. To increase the usability of a MUD, these methods should be consistent and efficient, otherwise it may become frustrating for the user to go from place to place. Traditional MUD methods for movement come from early text-based adventure games -- “north”, “south”, “west”, “east”, “up”, and “down”. These are usually shortened to ‘n’, ‘s’, ‘w’, ‘e’, ‘u’, and ‘d’. Many MUDs, including LambdaMOO, allow builders to name (or *alias*) exits, so a user can type something like `elevator` rather than be concerned with the compass direction of the elevator. This is useful. In both instances, users should have a method of determining the exits of a room.

One of the main advantages of a virtual environment is that non-standard methods of movement can be implemented. For instance, it is unnecessary to enforce a “grid-like” map; that is, all moves do not have to be of the same length -- the user can move through a long hallway in one command, just as they can move through a small room. Making every move travel an equal distance serves no useful purpose.

Other useful transportation methods exist, and unless they go against the ideology of the MUD, should be provided since they can greatly increase transportation efficiency. The first method is commonly called *teleporting*. Teleporting allows a user to move directly from one location to another without actually moving through the entire path to get there. If teleporting is allowed, there is no requirement at all that locations be assessable to each other by walking -- an environment could simply consist of a group of rooms, none of which connect, and users could simply teleport from room to room. This may be less intuitive than an environment with hallways and doors, but in many cases it may make transportation faster and easier, since users no longer need to remember compass directions or exit names. However, they do need to remember room names.

A second useful movement feature is written into LambdaMOO. The `@join` command allows a user to simply type a command like `@join jonathan`, which will move the user into the same room as the character named Jonathan. This method of movement will increase efficiency in applicable situations. The user does not need to determine where Jonathan is and then walk or even teleport there; one command will complete the whole process.

At least one DikuMUD, the Sloth DikuMUD at the University of Texas, provides a command called `hunt`. This command will take a user to another user, but through all the steps required to get there by walking. This is useful in a game context where teleporting may be too powerful. However, in a professional context it is most likely an annoyance since it is time-consuming and does not add any convenience.

5.4 Real-Time Communication

Since one of the main motivations behind MUDs is the ability to bring multiple users into the virtual environment, users clearly should be able to communicate. Probably the most natural form of communication is simply talking to other people in the same room, and due to the intuitive nature of this type of communication, it is useful to provide it. However, a virtual environment allows for more creative and effective communication capabilities as well. For instance, it is certainly useful to be able to communicate with groups of people in other scopes.

Communication can be generalized using scopes. For instance, a virtual environment can be set up so that a user can communicate within the room he or she occupies, within a defined area, or within the entire environment. Also, users may communicate with a certain set of users, regardless of their scope. For instance, some MUDs provide a `shout` command, which allows a user to send a message to every active user. Certain DikuMUDs provide a `grouptalk` command, which allows a user to communicate with a defined group of people.

Other MUDs provide “radios” with different “frequencies” to allow a flexible facility for group communication. (For a more lengthy discussion on the actual implementation of communication mechanisms, see section 5.9, “More on Online Programming Languages and Design Concerns”, on page 25.)

Another popular type of MUD communication that has proved useful is the `emote` command, which allows a user to express feelings and actions in a non-verbal manner. For instance, if a user typed `emote smiles`, the occupants of the room would see `John smiles` (see the excerpt from a MUD session in Figure 2). `Emote` is a very popular command; it is difficult to log onto a MUD without seeing many users communicating with it. The popularity of this command indicates that it is useful in MUD communication, and therefore should be included in a MUD’s set of commands.

5.5 Other Forms of Communication

The above communication is all real-time. It is also useful for users to be able to communicate in “non-real” time. This can be in the form of electronic mail or a “bulletin board” system, where users “post” messages on the board for others to read, similar to Usenet newsgroups on the Internet. LambdaMOO provides both mechanisms through the same system. Users can send mail to each other or to “collections”, which are received by a group of people [Rog92]. These collections have themes, ranging from MOO programming questions to a discussion list for users who live on the east cost of the United States. This mailing system seems effective; it is fairly simple, and gets a great deal of usage on LambdaMOO. There are currently over 60 active collections on LambdaMOO, and users exchange e-mail often. The only hinderance to the use of the mailer is the text editor (see section 5.9.4., “Text Editors”, on page 29).

5.6 Ownership and Levels of Control

If there exists an environment containing objects, it seems natural that users should be able “pick up” these objects and “carry” them from place to place. Many MUDs, especially gaming MUDs, introduce the concepts of “weight” and "size" to determine whether or not users can carry a certain object (see section 5.7, "Properties of Objects", on page 18). However, this may not be necessary in many environments; in fact, it may be very useful for a user to be able to hold an unbounded number of objects.

Some MUDs implement an "ownership" system in which the "owner" of an object may use or carry it, while other users may not.This implies the concept of **permissions**, similar to that of the UNIX file system. In UNIX, there are three defined user sets: the owner, a defined group, and the world. The owner can give three types of permission to each group: read, write, and execute permission. By implementing a system similar to this on MUDs, a great deal of flexibility is obtained.

This concept of **levels of control** can be useful when applied to commands. That is, there may be certain commands that can be executed by only a select group of users. For instance, in most MUDs only high-level officers (sometimes called "wizards") can create or delete character accounts, or shut down the system. Certainly, every MUD should have some type of permissions system. It is impractical and unsafe for all users have access to the entire command set.

5.7 Properties of Objects

All MUDs provide a method of defining and changing the *properties* of objects. This is necessary in order for objects to be interesting, since otherwise objects are completely static. Curtis gives the following example:

> ... an object representing a rock might have properties indicating its weight, chemical composition, and/or pointiness, depending upon the uses to which the rock was to be put in the virtual reality. [Cur92]

type - most programming languages define several different *types*, and require that variables be declared to hold only values of one type. Examples of types include integers, boolean values, and strings of characters.

Therefore, a generalized property system should be available. LambdaMOO's properties have no *type* constraints; they can hold any legal MOO value. This provides for ease of programming. LPC, like C, requires that all variables (the

LPC equivalent of a property) be defined to be of a certain type. Properties on LambdaMOO are also bound by the same permissions systems that verbs and objects are bound by [Cur92]. This is a useful design idea; there are many cases where a property should not be readable by the entire world. For instance, an object might have a password. One simple method to protect the password would be to make it unreadable to the world.

It is just as important to be able to define new properties of objects online. In DikuMUD, new properties of objects can not be defined online, which greatly restricts the extensibility of the MUD (see section 5.8, “Extensibility”, on page 19).

5.8 Extensibility

Extensibility is certainly one of the most difficult MUD concepts to handle gracefully and effectively. It is also the most important; the environment, the objects it contains, and the commands available all must be created somewhere. There are two basic approaches to this problem: online and offline creation.

5.8.1. Offline Creation

5.8.1.1. Objects

Offline creation does not allow users to define new objects from within the environment. This means that whoever wishes to define a new object must have access to the database files that contain object information for the MUD. DikuMUDs employ online creation. To add an object definition to a DikuMUD, it is necessary to change the database files and then restart the MUD, since the database file is only read on startup. When the MUD is restarted, active users are logged off of the system and must log back in. This also means that the new objects or rooms cannot be tested until the system is restarted, which is clumsy, since entering a new object may require multiple reboots to fix an error. In DikuMUD, it is quite possible to enter data incorrectly, since the database files are set up in a very cryptic form. The following is a DikuMUD definition of a simple room[Dik90b]:

```
#100
The Lego temple~
 You stand in a tiny, red temple; built entirely from Lego bricks.
It is, sadly, not a very interesting place, and perhaps you should
leave through the portal which leads south to a sunny garden.
~
1 12 0
D2
You see the grand portal of the Lego church. Beyond is an inviting
garden.
~
portal grand~
1 2 107
E
portal~
The portal is high and arched, built out of lego bricks of the
finest quality.
~
E
brick~
The bricks are all in bright different colours.
~
S
```

Figure 3. DikuMUD database entry for a room

This database format is difficult to read and easy to make mistakes in. From a software engineering point of view, this type of file format is a poor choice for the purpose; it will be difficult to debug, and extremely difficult to modify in the future [Jal91, pp. 234-235].

Several DikuMUD administrators have modified their systems extensively to support online creation. Online creation was also one of the motivations for the creation of LPMUD [Pen91b]. The fact that experienced administrators are willing to go to the effort of implementing online creation (even to the extent of creating an entire new MUD) strongly indicates that online creation is preferable to offline creation.

5.8.1.2. Commands

hard-coded - any aspect of a program that cannot be changed without changing its source code is said to be hard-coded (similar to being hard-wired).

In DikuMUD, the code for all user commands is *hard-coded* into the server program in To add any new functionality to objects or the environment, the server must actually be recompiled and, of course, restarted. For obvious reasons, this is a poor way of implementing command definition. Also, with any type of offline creation, it is extremely difficult for users that are not at the site of the MUD to add features and objects to the environment. Of course, *some* functions must be implemented in the server code (seesection 5.9, “More on Online Programming Languages and Design Concerns”, on page 25 and section 6.2, “Interpretations”, on page 33), but providing offline command definition as the method of providing new commands is an extremely poor design choice.

5.8.2. Online Creation

Both LPMUD and LambdaMOO support online creation. In these systems, nearly all of the environment is created online. A small set of low-level functions is implemented in the *driver* (or the server), which in both cases is written in C. The driver accepts connections from other machines and runs a very small set of commands that other commands are all derived from [Wal93]. These other commands are actually written online in the MUD’s programming language.

Although the specifics are somewhat different, LambdaMOO and LPMUD have similar approaches to the creation of objects and commands. In both environments, commands are implemented as **verbs** that are defined as properties of objects. When the user enters a command, the server calls the associated code for the verb. It decides which code to call by running through a set of objects and finding which object has the given verb defined for it. If the verb is defined on more than one object, the code will be called for the first object defined the verb that the server finds. If the rest of the command (the text after the verb itself) matches the text the verb’s code is defined to expect, that verb’s code is executed. If not, the server continues to look for a verb to execute [Cur92b, pp.

10-14]. For instance, if a user enters the command `wear jacket` in a room containing a computer, a shirt, and a jacket, the server might first check the computer to see if it has the verb `wear` defined. Since the computer will not have the verb defined, the server will check the shirt next. Although a shirt probably has the verb `wear` defined for it, the direct object jacket does not match, so the server then checks the jacket. This check will pass, and the jacket's `wear` code will be executed.

The code for these verbs is written in the MUD's programming language. The following shows the code for the `say` command in LambdaMOO and LPMUDs, respectively. In LambdaMOO, the code appears in the definition of a room object. In LPMUD, the `communicate` verb is defined for the player object.

```
@args #3:"say" any any any
@chmod #3:say rxd
@program #3:say
player:tell("You say, \"", argstr, "\"");
this:announce(player:titlec(), " says, \"", argstr, "\"");
return;

"cute add-ons to things people say";
if (!argstr)
      player:tell("Say what?");
elseif (argstr[length(argstr)] == "?")
      player:tell("You ask, \"", argstr, "?\"");
      this:announce(player:titlec(), " asks, \?"", argstr,
"\"");
endif
```

Figure 4. "Say" code for LambdaMOO [Cur91]

```
communicate(str)
{
      string verb;

      verb = query_verb();
      if (!str)
            str = "";
      if (verb[0] == "'"[0])
            str = extract(verb, 1) + " " + str;
      if (!confirm_flag)
```

```
            write("Ok.\n");
        }
        say(wrap(cap_name + " says: " + str + "\n", 65, 4));
        if (confirm_flag)
            write(wrap("You say: "+str+"\n", 65, 4));

        return 1;
}
```

Figure 5. "Say" code for LPMUD [JCur92]

The code for `communicate` calls `say()`, which is an **efun**, or external function. This is a command that is actually coded into the LP driver. LPC has about ninety efuns [Pen91]. LambdaMOO has about sixty similar functions, called built-in functions [Cur92b]. The `tell()` function, which is called by `say`, is a built-in function.

Note the syntactic similarities between LPC and the C programming language. The author of LPMUD designed LPC this way so that experienced programmers would feel more comfortable with the language [Pen91b]. LPC differs from C in that it has some object-oriented constructs, namely `inherit()`. When an object's code begins with an `inherit(x)` statement, the object inherits all the variables and functions of the object `x` [Fas93]. However, other variables and functions can be defined for the new object, including redefinitions of the parent object's variables and functions. This provides an excellent method for creating an object that has all the functionality of an existing object with a few changes.

LambdaMOO's language, MOO, is not designed to be similar to any popular programming language. It was "designed to be easy to learn for most non-programmers" [Cur92b]. It is also object-oriented; MOO stands for "MUD, Object Oriented" [Cur92b]. In MOO, objects are created with "parent" objects from which they inherit verbs and properties. This is slightly different than the mechanism provided by LPC, which states the objects' parents in the actual code, but the result is effectively the same: the child can now make use of the parent's implementation of an operation as well as provide its own definition for that operation. [Cur92b].

This is the heart of these MUDs' extensibility: an online programming language. Even the few DikuMUDs that have been modified to support online creation do not have a programming language, which severely restricts the type of building that can be done online. In these DikuMUDs the user can only create copies of predefined objects and change the values of their properties. This means that new functionality still must be added into the server code. So, for instance, one could not build a radio from scratch unless a radio already existed in the C code for the database.

In LambdaMOO or LPMUD, however, building a radio online from scratch would be fairly straightforward. Again, in both LambdaMOO and LPMUD, basically all existing objects and their related verbs have been created in the MUD environment.

Building is made easier by the inclusion of object-oriented features. Without such features, similar objects could not share (inherit) functionality; all functionality would have to be written from scratch. The object-oriented paradigm also makes it easy to create multiple objects with the same functionality; they are simply **instances** of a given **class** of object.

In both LambdaMOO and LPMUD, the creation of an object involves creating a generic, nondescript object and adding properties and verbs to it, either by inheriting or coding them. This is a concept that both programmers and non-programmers can understand.

The conclusion is that the flexibility and adaptability of a MUD is greatly enhanced if it is extensible from within. An internal programming language seems to be the best way to do this, for it allows fully flexible creation of objects and commands that operate on them.

5.9 More on Online Programming Languages and Design Concerns

Given that a MUD should have an online programming language to allow generalized extensibility, the next important matter is the actual implementation of objects and commands (verbs) -- the actual design of the MUD.

5.9.1. Efficiency

Generally, the main loop of a MUD involves three steps:

- Check for and handle new connections
- Loop through all active connections, checking for commands
- Process commands

Steps two and three of this list can be implemented in different ways; the important requirement is that a user's commands be run in the order the user gives them This ensures that the user does not get unexpected results if one command depends on another running first. A secondary concern might be that all user commands execute in the actual order they are given, that is, that if user A gives a command before user B, user A's command will run first. Since input from all connections is not received concurrently, this guarantee can not be made. However, it is possible to make such a guarantee within the amount of time it takes to loop through the set of connected users, since commands may be placed on a queue as they are received. LambdaMOO does not implement this system; instead, users who have averaged the least amount of CPU usage recently have their commands run first [Cur93b].

Since there are generally many users logged in at once (usually anywhere between two and eighty or ninety), it becomes important to handle step three of the above list -- the processing of pending commands -- as efficiently as possible. This ensures that users will have a reasonable response time to commands.

An ***interpreted language*** is not translated into machine language before being run. Programs written in interpreted languages are generally slower, but can be modified and executed without any intermediate steps.

5.9.2. Efficiency and MUD Code

Generally, online MUD programming languages are *interpreted*. Lars Pensjö, author of LPMUD, says in the LPC manual: “As it is important that objects be loadable 'on the fly' in a game, I choose to make [LPC] an interpreted language" [Pen91]. Functions written in an online interpreted language will run somewhat slower than functions compiled on the server machine, since as the MUD program is interpreted, it will call functions that are written in the server. The server essentially defines a *virtual machine* that the MUD programming language runs on top of.

virtual machine - a program that provides a set of commands that can be called by applications that run "on top" of it. This is analogous to the set of commands provided by a computer for programs that run "on top" of it.

This creates a dichotomy for the designer: should functionality be hard-coded into the server for greater speed, or should it be written in the online programming language for flexibility?

LambdaMOO's server provides enough built-in functions to allow other functions to be written online, that is, no built-in functions are provided that could be written easily using other built-ins. LPMUD, on the other hand, does provide some extra functionality in the server. To clarify, here is an example:

LambdaMOO and LPMUD both provide message-sending functionality in their servers. However, the LPMUD server provides more specific mechanisms in the server itself as efuns, while in LambdaMOO the specific mechanisms are implemented in LambdaCore. The LambdaMOO server simply provides the `notify()` function, which sends a string to a player if the player is connected [Cur92b]. LPMUD provides four efuns: `write`, which has the same function as LambdaMOO's `notify`; `say`, which sends a string to all users in the same room; `tell_room`, which sends a message to all objects in a room, including non-player objects that have a routine to handle such an event, and `shout`, which sends a string to all players [Pen91]. All of these commands can be implemented online using only an equivalent of the `notify` function, since both environments also provide efuns (or built-in commands, see section 5.8.2., "Online Creation", on page 21) to get a list of all current users as well as users in the room. Since the LPMUD efun commands are part of the compiled driver code, they will run faster than commands written in the online language, which are interpreted.

To compare the relative speeds of efuns and LPC code, the following code was used [JCur93]:

```
/* test shout efun vs. LPC code              */

test1(string str)

{
      shout(str + "\n");
      return 1;
}

test2(string str)
{
      object *ob;
      int i;

      ob = users();
      for (i = 0; i < sizeof(ob); i++) {
            tell_object(ob[i], str + "\n");
      }
      return 1;
}
```

Figure 6. Code used to test effeciency of efuns vs. LPC code

This code implements two simple tests, `test1()` and `test2()`. Each implements a version of the `shout` command. The `shout` command was chosen since it generally requires looping through a fairly large number of users. Such a loop should be a good testing ground for performance. The first function calls the `shout()` efun, while the second provides the same functionality with code written in LPC. The server's `shout()` code essentially does the same routine as `test2()`, but in actual compiled C code. Tests were timed with the `gauge` command, which returns the number of *CPU ticks* required for a given operation to complete. First, each of the above operations was `gauge`d. Next, a loop was added so that every call to a test function was the same as calling it 20 times. Finally, the call that actually produced the messages in each routine was

CPU ticks - a measurement of how much computing time an operation takes.

duplicated so two message-sends would take place without the overhead of a loop. Results were as follows:

	one call	two calls	20 calls
run 1	1007	1019	1291
run 2	1014	1019	1291
run 3	1014	1019	1291
Average	**1012**	**1019**	**1291**

Table 1. Time for efun version of shout to run

	one call	two calls	20 calls
run 1	1434	1329	2558
run 2	1341	1348	2186
run 3	1392	1253	2558
Average	**1389**	**1310**	**2434**

Table 2. Time for LPC version of shout to run

Clearly the efun version was faster, although for a single call the difference was not great. However, when a large number of calls in a row were involved, the LPC code was much slower. This could be because the efun spent most of its time waiting to queue up and run, but after beginning execution required almost no time at all. Certainly larger, more complicated routines would show a much greater performance differential, since the LPC would still spend a great deal of time interpreting commands.

This means that it may be worth implementing some common commands in the server itself for performance. If a certain server routine requires replacing, it can always be overridden with commands written in the online language.

5.9.3. Server Requirements

Given that a MUD can be extended from within using a programming language, it becomes important to be sure that there exists a full set of commands in the server that a programmer may call to create new functions. The contents of the set depend entirely on the features of the MUD and its database, but it is very important that programmers are given functions that allow access to every aspect of the environment. There are some common low-level building blocks that MUDs should provide facilities for in the server. Communication and the existence and persistence of objects are issues that must be addressed at the server level. That is, the only way a “say” command can be written in the online programming language is by using a server command that sends a message to a user. Such a function is one of the most basic necessary server commands, since without it users could receive no feedback. The same goes for issues of persistence; if the server does not provide a method of storing database information, these methods can not be built online.

With a robust set of low-level constructs, the entire MUD may be built online from scratch. LambdaMOO in its barest form has only three commands available to users, two of which are intended to work with clients. The command that is interesting to a player is `.program`, which associates a verb with a given object [Cur92b]. This means that every object class and every verb is implemented in MOO code rather than in the server. When getting the LambdaMOO code to set up a new site, a new system administrator also receives a copy of LambdaCore that implements all of the basic features of LambdaMOO such as rooms, player objects, and so on. This shows that the LambdaMOO server is truly robust, and that LambdaMOO is completely flexible and extensible.

5.9.4. Text Editors

In order to enter and edit code, electronic mail, or any other potentially long piece of text, some type of online text editor is necessary. The LambdaMOO text editor is actually a room with some fancy verbs. LPMUD uses an editor based on `ed`, a basic UNIX text editor. Both of these text editors are fairly clumsy to use since they are **line-based**, which means that a user must be explicitly editing one line at a time. Since MUDs run through a basic telnet connection, this appears to be the best they can do; a **screen-based** editor (such as vi or emacs) would require a more complicated connection. However, the LPC-Centre LPMUD has implemented an interesting and useful service that can greatly increase text editing efficiency.

Instead of connecting to the usual port of LPCC with `telnet` (see {APPENDIX on connecting to MUDs}), a user can use `ftp`, the Internet's file transfer protocol, to connect to an LPCC ftp server. Users can then download and upload text files to and from the MUD. This allows users to edit text on their own machines with any editor they like. Ftp is especially compatible with the LPMUD model since all objects are defined by **files** containing code. The files reside in directories that are set up in a structure similar to the UNIX filesystem. Since ftp is meant to work with the UNIX filesystem, the two models mesh together easily. Such an interface is an improvement over online editors since the user can work more comfortably and productively.

5.10 Future Concerns

5.10.1. Networking

While most of today's MUDs run independently of each other, this is changing. LambdaMOO programmers have implemented MUD networking with a system called MOOnet (similar systems have been implemented for a few other MUD types). MOOnet allows LambdaMOO users to communicate from MUD to MUD in real-time or by email. Other services are also provided. For instance, the `@netwho` command allows a user to see if another user is currently connected to any of the networked MOOs [Wil93b]:

```
@netwho blackbriar
Player name      Connected  Idle time  Location
-----------      ---------  ---------  --------
Blackbriar       an hour    11 seconds OpalMOO
Blackbriar       49 minutes a minute   LambdaMOO
```

Figure 7. Sample output for the @netwho command [Eva93]

Other available commands include `@netpage`, which allows a user to talk to a user on another MOO, and `@netsend`, which allows a user to send MOO-mail between MOOs [Wil93b].

Currently the MOOnet system only works between LambdaMOO systems, but programmers plan to extend it to enable similar communications with users on other types of MUDs. This would certainly be easier if some type of standard was established for MUD networking. It is possible that the package transmission protocols implemented by LambdaMOO programmers to communicate with other types of MUDs may be the beginning of such a standard.

Another potentially useful networking idea being developed for LambdaMOO is MOOftp, which will allow the transfer of object definitions between MOOs [Eva93b]. This is slightly different from the LPCC's ftp implementation (see section 5.9.4., "Text Editors", on page 29), since its purpose is to transfer objects between MUDs rather than transferring code into a MUD from a non-MUD environment (such as the UNIX shell).

5.10.2. Beyond Text

The Jupiter Project at Xerox PARC may be a great leap in MUD technology due to its use of networked audio and video. The rationale behind the inclusion of such facilities is that although text is universally available, it has many drawbacks in the workplace as a mode of communication. Most importantly, typing is much slower than talking on the phone or in person, and therefore will not be a popular alternative to communicating with speech [Cur93]. However, the MUD may provide a useful and friendly interface to powerful networked features. The following is an example given by Pavel Curtis:

> I virtually walk into your virtual office on the MOO. Automatically, my speaker and microphone are patched into the same networked audio channel as yours, so that you can hear everything I say and vice-versa. Also, a window appears on my screen showing the picture from your video camera (and that of anyone else in that

> virtual room) and vice-versa. I 'pick up' the virtual whiteboard in your office and up pops another window on both our screens, this time one that supports simple drawing operations; when I draw on my window, or point to pieces of the drawing, you see those things on your window (again, and vice-versa) [Cur92c].

client - a program that communicates with another program, called the *server*, and carries out certain actions depending on the commands it receives.

Any type of extended functionality such as the above requires a specialized *client* program instead of a bare telnet connection, since the data being passed back and forth will need to include commands that the user should not see. This client-server model allows for compatibility with any system by abstracting user interface commands into a high-level, non-platform-specific command set. A simple example would be a command sent by the server to open a new window on the screen. The actual command might just be some simple text describing the high-level characteristics of the window, such as size and screen position. However, all machines connected to the server will have their own client programs that do the low-level calls to actually create the window on the user's machine. This means that even though one user might be running a workstation with X Windows while another is using a Macintosh, the server will send the same command to each machine, and the client will decide how to actually do the work. This makes the server code fairly easy to write since all calls are at such a high level [Cur92c]. It also allows complicated operations without wasting a tremendous amount of Internet (or any other network) *bandwidth*.

bandwidth - the amount of data a network can transit per unit time. Therefore a large amount of data is said to use a great deal of bandwidth.

5.11 Applying the Building Blocks

This chapter provides a generalized set of "building blocks" for MUD design. These building blocks are concepts and features that are found in most MUDs, notably the three popular MUDs studied in this paper, DikuMUD, LambdaMOO, and LPMUD. While most of the building blocks are present in more than one of the studied MUD types, many of them are approached in different ways. By examining these different methods, their relative strengths and weaknesses can be compared. This makes it possible to see which approach, or combination of approaches, are better and why. These building blocks, when combined, lay the foundation for an extendable, powerful MUD.

Given this set of building blocks, how should it be used? Basically, a designer can use this list to conduct a requirements analysis for a MUD system. While some blocks may not apply, most should be applicable to all MUDs. By making early decisions on the implementations of the building blocks, the designer avoids many of the problems previously encountered in MUD designs.

6. Conclusions

6.1 Summary

There are currently hundreds of MUDs running on the Internet, nearly all of which are used for recreational purposes. There is a small group of MUDs that are used for non-recreational purposes such as MUD research. Most MUDs are derivatives of one of the systems in a smaller set of MUDs, referred to in this paper as "MUD types".

These MUD types and their derivatives have generally been designed in an ad hoc manner. When their designs are closely analyzed, as in this paper, a number of shortcomings appear. These shortcomings may have been avoided had the designer conducted a more thorough requirements analysis for the system.

This paper, then, provides MUD designers with something they previously had to work without. By examing the building blocks and associated discussions, designers can avoid re-inventing the wheel as well as avoiding bad decisions that have been made already. Essentially, this paper helps us to learn from our mistakes.

6.2 Interpretations

The most important low-level aspect of a MUD is its facility for extension of the environment. Without online creation though an online programming language, a MUD is far less flexible, and therefore potentially much less powerful. A MUD that provides little functionality but has a robust set of server commands (see below) is far more useful than a MUD that implements a great deal of unalterable functionality; the former can have an unbounded amount of functionality added, while the latter can *only* have whatever functionality is built into it. It may be useful to code some routines into the server for pure speed, however, in most situations it may be more desirable to have access to the command's code from within the environment.

Both LambdaMOO and LPMUD implement object-oriented online programming languages. The object-oriented nature of these languages is useful in the MUD model, as it allows slightly (or greatly) modified versions of existing objects to be created without replicating code. DikuMUD, on the other side of the spectrum, does not provide any facility for online extension, which severely restricts its flexibility.

An online programming language seems to be the most natural way to facilitate extensibility, since it allows for complete flexibility within a context that is familiar to many users. LambdaMOO's and LPMUD's programming languages have similar constructs and characteristics; however, LPC is based on C, which is one of the most popular programming languages in computer science and other fields. This means that most programmers will feel instantly comfortable and familiar with the syntax and constructs of the language. Granted, not every user will know C before coding in LPC, but in contrast, all new LambdaMOO users have probably never seen a language quite like MOO. However, MOO was specifically designed to be easy to learn for non-programmers; this aspect alone is certainly useful in some situations. The choice of online programming language, then, should depend somewhat on the intended users of the MUD.

What services should be provided for users by programmers? This depends greatly on the purpose of the system; however, most of the general mechanisms presented in this paper will probably be necessary, and many of the slightly more complicated features mentioned can boost users' productivity.

Finally, it is very important to realize that MUDs are in a very early stage of development. Designing a flexible and extensible MUD is extremely important if the designer would like the MUD to survive longer than a few years.

6.3 Recommendations for Future Work

The scope of this project did not include the acutal building of a MUD using the building blocks. To determine the effectiveness of the data, the next step is to actually design and test a MUD using these building blocks. Useful metrics will be necessary to determine if the resulting MUD is better than existing MUDs. If the new MUD is indeed better, the results of this project can be considered correct. Either way, the project is a success if it causes MUD developers to carefully consider design concerns.

Research involving MUDs is just beginning in a small number of research institutions. It appears that the future of MUDs depends on the findings of these groups. Currently, the problem is determining which applications are well-suited to MUDs, and which are not. If Xerox PARC is successful with the Astro-VR project and the Jupiter project, many doors may open for MUDs and they may quickly grow in popularity. As this happens, the design of MUDs will become more important. Hopefully, this document will provide a good starting place for designers of the next generation of MUDs, whatever their purpose.

Bibliography

[Bar90] Bartle, Richard, “Interactive Multi-User Computer Games”, MUSE Ltd. Research Report, December 1990.

[Bru92] Bruckman, Amy, “Identity Workshop: Emergent Social and Psychological Phenomena in Text-Based Virtual Reality”, MIT Media Laboratory, April 1992.

[Cur91] Curtis, Pavel, code for `say` verb, taken from the definition of the `$room` object on OpalMOO, March 1993.

[Cur92] Curtis, Pavel, “Mudding: Social Phenomena in Text-Based Virtual Realities”, proceedings of the 1992 Conference on Directions and Implications of Advanced Computing, Berkeley, May 1992. Also available as Xeroc PARC technical report CSL-92-4.

[Cur92b] Curtis, Pavel, “LambdaMOO Programmer’s Manual”, October 1992, available using anonymous FTP at `parcftp.xerox.com` in `/pub/MOO`.

[Cur92c] Curtis, Pavel, electronic mail, “Re: Articles on MUDs?”, October 21, 1992.

[Cur93] Curtis, Pavel and David A. Nichols, “MUDs Grow Up: Social Virtual Reality in the Real World”, Xerox PARC, Janurary 19, 1993, available using anonymous FTP at `parcftp.xerox.com` in `/pub/MOO/papers`.

[Cur93b] Curtis, Pavel, electronic mail, “Main loop”, March 17, 1993.

[Cur93c] Curtis, Pavel, electronic mail, “Disk-based DB consistency issues”, March 17, 1993.

[Dik90] DikuMUD documentation file `database.doc`, author unknown.

[Dik90b] DikuMUD documentation file `dbsup.doc`, author unknown.

[Eva93] Evans, Scott, log taken from MediaMOO session, March 28, 1993.

[Eva93b] Evans, Scott, conversation with Seth Rich on OpalMOO, March 28, 1993.

[Eva93c] Evans, Scott, log from MediaMOO session, March 30, 1993.

[Fas93] Fastenrath, Bernhard et al, “LPMud, a Programmable Multi-User Game”, revised version of original text by Lars Pensjö, pre-release, Feburary 7, 1993.

[Goe93] Goehring, Scott, “The Totally Unofficial List of Internet Muds”, Volume 5, Issue 6, March 12, 1993, available using anonymous FTP from `caisr2.caisr.cwru.edu` in `/pub/mud`.

[Jal91] Jalote, Pankaj, *An Integrated Approach to Software Engineering*, Springer-Verlag, 1991.

[JCur92] Curnow, Jason et al, `communicate()` function, taken from `player.c` on the LPC-Centre LPMUD, March 1993.

[JCur93] Curnow, Jason, `test_object.c`, modified by Scott Evans, March 1993.

[Pen91] Pensjö, Lars, “LPmud, a Programmable Multi-User Game”, 1991, available using anonymous FTP at `ftp.cd.chalmers.se` in `/pub/lpmud`.

[Pen91b] Pensjö, Lars, “Idea Behind this Game”, text available in the file `/lib/concept` on the LPC-Centre LPMUD.

[Rog92] Roger (character name on LambdaMOO), help file for `@send` command.

[Smi93] Smith, Jennifer and Felix Ortony, “Frequently Asked Questions: Basic Information about MUDs and MUDding”, March 1993, posted monthly to Usenet news and available using anonymous FTP from `ftp.math.okstate.edu` in `/pub/muds/misc/mud-faq`.

[Wal93] Waldrop, James, “Frequently Asked Questions for rec.games.mud.lp”, March 11, 1993, available by email from jlw@shekel.cs.columbia.edu.

[Wil93] Wilson, Andrew, “MOO-net Documentation”, February 12, 1993, available in the Virtual Library at OpalMOO or LambdaMOO.

[Wil93b] Wilson, Andrew, help file for the Network Features Object on MediaMOO, March 1993.

Appendix 1. Glossary

The terms listed in this glossary are found in *italics* throughout the text.

bandwidth the amount of data a network can transmit per unit time. A large amount of data is said to use a great deal of bandwidth.

CAD computer-aided design. CAD programs allow the user to draw objects in three dimensions.

client a program that communicates with another program, called the *server*, and carries out certain actions depending on the commands it receives.

core the definitions of all existing objects and commands that have been programmed in the MUD's programming language or created from within the environment.

core database a file or set of files that contain the core.

CPU ticks a measurement of how much computing time an operation takes

hard-coded any aspect of a program that cannot be changed without changing its source code is said to be hard-coded (similar to the term "hard-wired").

instance an instance of an object is a "clone" of that object. It has the same properties as the original object, but they may have different values. For example, Camaros are all instances of a "car", although each individual Camaro may be a different color.

Internet a worldwide network that provides services to universities, research institutions, and corporations. Most electronic mail travels via the Internet.

interpreted language is not translated into machine language before being run. Programs written in interpreted languages are generally slower than ones that are not, but they can be modified and executed without any intermediate steps.

multiple users more than one user inside an environment simultaneously. In graphical virtual realities, this greatly increases complexity and running time.

server a computer that accepts connections from other computers and runs the virtual environment software

type most programming languages define several different types and require that variables be declared to hold only values of one type. Examples of types include integers, boolean values, and strings of characters.

virtual machine a program that provides a set of commands that can be called by applications that run “on top” of it. This is analogous to the set of commands provided by a computer for programs to run “on top” of it.

Appendix 2. Internet Addresses of Selected MUDs

To connect to a given MUD on a UNIX system, simply enter the command:

```
telnet <Internet Address and port>
```

at the prompt. If the connection is successfully made, the MUD system should provide helpful prompts from then on.

Name	Internet Address and port number
LambdaMOO	lambda.parc.xerox.com 8888
OpalMOO	cc.brynmawr.edu 7777
LPC-Centre LPMud	victoria.city.ac.uk 6999
Genesis LPMud	milou.cs.chalmers.se 2000
Sloth DikuMUD	tokio.cs.utexas.edu 2002
WileyMUD II	wiley.cs.wmich.edu 3000

These addresses are taken from the *Totally Unofficial List of Internet Muds* [Goe93].