

The Cow Ate My Brain
or
A Novice's Guide to MOO Programming, Part I
by
Loyd Blankenship ("Mentor")
Last Revised: September 3, 1993

This document is copyright 1993 by Illuminati Online (io.com). All rights reserved. You may distribute this electronic document as you wish, provided no fee is charged for doing so, and the text (including this disclaimer) is left untouched.

Introduction

When I first logged onto a MOO (about 2 months ago from the initial date of writing), I was both intrigued and appalled. Intrigued -- because of the infinite possibilities of a user-extendable, multi-user virtual reality. And appalled -- because of the scarcity of documentation and help for new users.

I came to MOOs as an experienced programmer. I already knew what Object Oriented Programming (OOP) was, and could program in half-a-dozen different languages. And I **still** had a hard time getting off the ground creating interesting places and objects, not to mention writing new verbs. I shudder at the thought of encountering a MOO and not knowing anything about programming and OOP . . .

This file is my attempt, while it's still fresh in my mind, to help teach a beginner how to become a useful MOO programmer. Let me point out right now that I am **not** a MOO programming expert. I've only written a few dozen good verbs at this point. But I feel like I'm now over the hump of the learning curve -- I understand how things work, and I know where to look to find answers to my questions. I want to help get you to this point.

As you go through this document, make frequent use of the ``help'` command on the MOO itself. For instance, if you don't completely understand the syntax of `@create`, try ``help @create'`. It isn't the best help system in the world, but combined with this file, you might get through.

Part I (this file) talks about MOO basics -- what they are, how they work. It covers building rooms (including exits), creating objects, and such basic topics as verbs, properties, object oriented programming and the like. Part II (which I hope to write in the next few weeks) will cover the basics of verb programming.

I'll continue to revise this file over time. If you have any comments, good or bad, please send them to me at one of the following (in order of preference): `mentor@fnordbox.io.com`

CI\$: [73407,515]

Genie: SJGAMES

I can also be found hanging out as "Mentor" on the Metaverse MOO (`metaverse.io.com 7777`) almost every day, Jay's House MOO once a week or so, and LambdaMOO on a sporadic basis. And if you're really desperate, you can send snail mail to me c/o Steve Jackson Games, PO Box 18957, Austin, TX, 78760.

What is a MOO?

If you're reading this, you've probably stumbled on to at least one MOO. At the high-concept end of things, a MOO is a multi-user, text-based virtual reality. At the low-concept end of things, think of it as a multi-player

□

Infocom-style game that lets you talk to other players and build new locations and objects.

The MOO is a program running on a Unix-type computer system. The original MOO is LambdaMOO, and was developed by Pavel Curtis at the Xerox Palo Alto Research Center from an initial body of code provided by Stephen White at the University of Waterloo. It has been up and running since October 1990.

The main program working the MOO is called the "server." The server handles all the nitty-gritty details such as input/output between the Internet and the MOO, parsing (figuring out) the lines you enter, and so on. The source code for LambdaMOO's server is freely available via FTP at `parcftp.xerox.com`. It has been ported to a number of systems, from BSD/I to

Linux to Amiga Unix. This isn't intended to be a comprehensive guide to setting up and running a MOO (mainly because I don't know much about it!), so we're going to assume that the reader has logged into someone else's MOO.

The other big duty of the server is to compile and run programs (called "verbs") written in a language called MOO. MOO is a cross between C++ and LISP. The programming language is thoroughly documented, but the docco is *not* really aimed toward the novice user.

If the server is the heart of a MOO, a giant database called the "core" is the brain. Initially, the core is fairly empty -- it defines basic objects (more on objects later) such as \$thing (the generic thing) and \$prog (the generic programmer, which is a player-character with the power to add more things). As players build new rooms and create new objects, the core gets bigger. The initial LambdaCore is also available via FTP from parcftp.xerox.com.

Enough technical babble -- from this point on, I'm assuming that you've connected to a MOO and have your own character.

What is Object Oriented Programming (OOP)?

Before you can truly understand MOO programming, you have to be clear on the concept of OOP, especially the idea of objects and inheritance.

Object Numbers

Everything on the MOO is an object. Players are objects, rooms are objects, exits and entrances are objects, editors are objects, and objects are, well, objects. Each object has an "object number". For instance, the generic thing might be #5. When you're in the MOO, you can substitute an object number for an object's name at any point. If I'm object #433, for instance, you could type "give ball to Mentor" or you could type "give ball to #433". If the ball is #731, you could type "give #731 to #433". The MOO doesn't care.

Object numbers are different from MOO to MOO -- just because I'm object #433 on Metaverse doesn't mean I'm object #433 on Opal, or on Lambda.

There are only a few basic object types defined in the MOO core. Everything else starts out as a copy -- a "child" -- of these basic objects. For example, there is a basic class called "\$thing" (a generic thing). If I wanted to create a Generic Ball, I might make it a child of \$thing (by typing `@create \$thing named "Generic Ball"' -- more on this later). Then I could create children of the Generic Ball, such as a basketball (by typing `@create "Generic Ball" named "basketball"), or a baseball, or a football, or even a Frisbee! (Think about it -- you do many of the same things with a Frisbee you do with a ball: throw, catch, drop, get, etc.).

□

Examining Objects

There are several different ways you can look at an object. The first (and most obvious) is `look'. If you type `look ball', you'll see the description of the ball.

There are several different ways you can look at an object. The first (and most obvious) is `look'. If you type `look ball', you'll see the description of the ball.

Next up, we have `@examine ball'. This shows you the object description, verbs and properties.

Then there's `@show ball'. This tells you the object name and number, the name and number of its parent object, the name and number of the owner, and a list of all visible verbs and properties (see below).

Finally there's `@dump ball'. This gives you a listing of all the things `@show' does, plus shows you the programs for all visible verbs *on that object*. This differs from the `@list' command, which we'll discuss below.

(If you try to do the `@show' command and it doesn't work, you need to change your parent object to the Generic Programmer. This won't allow you to write programs unless the wizards have set your programming bit -- see Quota, below -- but it will give you access to nifty things like `@show'. Just type `@chparent me to \$prog' and that will do it. If it doesn't, bug a wizard . . .)

You can get a list of all the messages on an object (see below) with the command `@messages <object>'.

What is a Verb?

A verb is, simply, a program that is attached to an object. For example, \$thing (our generic thing mentioned above) starts out with a few predefined verbs -- put, take, get, and drop. So you could type "get \$thing" and you'd have the generic thing (type `inventory' (abbreviatable to 'i') to see a list of things you're carrying).

When you're programming verbs, you'll almost always refer to them in the form <object #>:<verbname>. So in the case of \$thing (object #5), the `put' verb would be `#5:put'. If we had a ball that we could throw, the verb would be called `ball:throw'.

There are two ways to look at the MOO program that is attached to a verb. The first is using the @dump command, described above. The second is using the `@list <object#>:verb' command. The command @list starts by checking the object for the verb; if it isn't there, it checks the object's parent. If it isn't there, it checks the parent object's parent. And so on, until it reaches the top. So if you type "@dump ball", you aren't going to see the program for the `put' verb -- it's defined on the generic thing, not the ball. But if you type `@list ball:put', the server will tell you something like "There is no `put' defined on ball, but it's defined on one of ball's ancestors." Then it will show you the `put' program. This is our first example of how *inheritance* works. More later . . .

What is a Property?

A property is a value that is stored on an object. There are certain properties that are built into everything -- the description, for example. Every time you type `look <object>', the server runs the verb `look_self' on the object in question, and shows you the description of the object you looked at. We might also have a property on our Generic Ball called color, that you could set when you create children of the Generic Ball. If we created a football (@create Generic Ball named "football"), we could then set

```
□
the color using the `@property' command (@property football.color is
"brown"). Notice that when you're talking about properties, you use the form
<object #>.property (as opposed to <object #>:verb). It's easy to forget
which one wants a period and which one wants a colon. Use `@show object' to
get a list of all the properties on an object.
```

A property does not have to be a text string. They can also be lists or numbers. For example, a clubhouse might have a property called "members" as follows:

```
clubhouse.members = ("Bob", "Carol", "Ted", "Alice")
Or maybe it's just a number -- clubhouse.num_members = 4.
```

Messages

Messages are special kinds of properties; they are always text descriptions, and they end with `_msg'. For example, `move_msg' might be a message displayed when an object is moved. One of the first ways you'll want to customize your rooms and objects is to edit the messages on them. More on that after we've dealt with creating objects and building rooms.

You can get a list of all the messages on an object with the command `@messages <object>'.

Inheritance

We'll come back to creating new verbs and new properties later. Now we're to the most important part of OOP -- inheritance. Simply put, any verb or property on the parent is automatically available on all its children. Any child may have its own version of the verb instead, but if a verb is not specifically defined on a child, the child uses the verb as it is defined on the parent. Let's look at some examples.

Our generic \$thing has some basic thing-like verbs on it: put, take, get, drop, etc. When we create our Generic Ball, we'll be adding some new verbs: throw, catch and bounce. But we don't have to worry about writing our own put, take, get and drop for the Generic Ball -- it *inherited* them from \$thing. When we make a child of the Generic Ball (say, a basketball), we might add the verbs dribble and shoot. When we did a `@dump basketball', it would only show these two verbs. But we could still catch it and throw it, because it inherited the catch and throw verbs from the Generic Ball. And we could still get and drop the basketball, because it inherited those verbs

from the Generic Ball, who in turn inherited them from \$thing.

It works the same way with properties. If we made the description of Generic Ball say "This is an ugly green ball", then when we typed `look football', we'd see "This is an ugly green ball." You would have to change the description of the football to something more appropriate, thereby writing over the initial value of the inherited property.

Now that that's out of the way, let's start building.

Quota

When you get your character on a MOO, you start with a certain amount of quota. Each unit of quota allows you to create one object. This number is typically six to eight. You'll need to check with each MOO to find out their policy for giving extra quota.

Initially, you probably won't be allowed to write programs -- just to build new rooms and create new objects. It's usually not hard to convince a wizard to turn on your programmer bit, though.

If you run out of quota, you're stuck. You can always recycle something you created, though. This gives you back the quota. More on recycling later .

□

. .

Building a Room

The first thing everyone wants to do when they get online is to build a house. This can be anything from an apartment to a Pez dispenser to a dumpster to a cloud castle. My current house is a cardboard box with an opium den in the basement and a programming lab attached (plus various secret passages leading to all sorts of strange things . . . :-)

Start by typing `help @dig'. I'll wait. That does a very good job of explaining how to create your initial room. You won't have any exits or entrances at first -- there's no place to exit or enter *to*.

When you create this room, it isn't `attached' to the rest of the MOO geography. The only way to get to it is to teleport. You teleport around by typing `@move me to <object # of the room you want>'. Different MOOs have different policies about linking new creations into the existing world. This takes a wizard to do.

Now that you've got a home, type `@sethome'. This makes your new room your home -- your character will live here when you're offline, and all you have to do is type `home' from anywhere in the MOO, and you'll be teleported back here.

Next you need to give your room a description. Type `@describe here as "This is my new home! This message is displayed whenever anyone enters here! I better make it good!'

Notice that you don't need a close-quotation mark, just an open one. The `@describe' program assumes that everything after the first " is description.

Adding Exits

Now, suppose you want to add a second room to your fledgling mansion. Do `help @dig' again. Follow the directions. Let's say you wanted to put an exit in the south wall leading from your Living Room (the first room you created) to a porch. You'd type `@dig s,south|n,north named "The Porch"'.

Let's break that down. The first part says to dig an exit that can be accessed by either "south" or "s" going into a new room named "The Porch". By putting the `|n,north' part in, you tell the program to make *another* exit going from The Porch back into the Living Room, called with "north" or "n". If you'd just typed `@dig s,south named "The Porch"', you could walk out on the porch, but you couldn't get back.

You can do all sorts of interesting combinations. For instance, if you had described your room as having a rope hanging from the ceiling, you might do a `@dig climb,u,up|climb,descend,d,down named "The Roof"'. Now anyone in your Living Room could type `climb', and they'd be taken to the roof. Once there, they could type `descend' or `down' or `d' or `climb' and come back down.

One thing you'll notice quickly is that every exit is an object. So if you've got a Living Room with a sliding glass door going out to The Porch, you've really got 4 objects: The Living Room, The Porch, the exit The Living Room --> The Porch, and the exit The Porch --> The Living Room. Where'd all

my quota go?!?

Interior Decorating With Messages

Now we're back to messages. (Remember messages? If not, go back up and reread the section on Properties. I'll wait.)

The easiest way to customize your rooms is to edit the messages on the

□
room. You've already done this a little bit when you used '@describe'. But there are a lot more messages on rooms (and especially on exits) that you'll want to play with. You can get a list of all the messages on an object by typing '@messages <object#>'.

Any property that ends in '_msg' is displayed by '@messages' and can be edited as follows: Let's say we've got a property on our football called 'punt_msg'. This is displayed whenever anyone executes the verb football:punt (they'd do this by typing 'punt football', probably). The syntax is '@<property> <object> is "message"'. When you're using this method to edit a message, you *don't* have to include the '_msg' extension. For example, we'd type:

```
@punt football is "You kick the football."
```

Another, more concrete example concerns exits. Exits are chock full of messages such as:

```
leave_msg: displayed to a player passing through  
the exit.
```

```
oleave_msg: displayed to everyone else in the room  
when someone uses the exit.
```

There are many more. Do a '@messages <object # of the exit>' to see them all. To set the leave_msg you'd type '@leave <object#> is "You walk through the glass doors into the sunlight.'. For messages that other people in the room see, the player's name is automatically prefaced. So the oleave_msg might be '@oleave <object#> is "walks through the glass doorway to the porch.'.

In the above example, if I walked from your Living Room to your Porch, I'd see "You walk through the glass doors into the sunlight." If you were sitting in the Living Room when I walked through the door, you'd see "Mentor walks through the glass doorway to the porch."

Interior Decorating With Details

If you had to create an object for every little thing in your rooms, you'd quickly run out of quota! While you might want to create a couch, and maybe a TV or Radio or Clock for the Living Room, you probably don't want to create a rug (unless it's covering a trap door) or a fireplace or the coasters on the coffee table. The '@detail' command lets you fake this.

The syntax is '@detail <thing[,thing...]> is "Detail Text"'. Let's flesh out the Living Room by way of illustration. We start by describing the Living Room (this assumes we're inside the room when we're typing this).

```
@describe here as "This is Mentor's living room. It is sparsely furnished,  
with just a couch, a few pillows and a rug. There is a painting hanging on  
the west wall, and a sliding glass door opens to the porch toward the south.
```

Pretty basic. Right now, if a player types 'look wall' or 'l wall' or 'l painting', the MOO will respond "You don't see anything special." Boring. Let's put a painting on the west wall.

```
@detail wall,painting is "There is a large Real Musgrave original hanging on  
the wall, showing a horde of delinquent Pocket Dragons mugging a teddy bear.
```

Now, when someone walks into the living room, they'll see the basic description. If they're curious enough to type 'l painting' or 'l wall', they'll see the detailed description of the painting.

□
You can give clues to other details within a detail. For instance:

```
@detail pillows,pillow is "These are groovy purple pillows that were  
undoubtedly stolen from Zarabeth's place. There is a button pinned to one of  
the pillows."
```

@detail button is "Excuse me, I'm from the BATF. Could you direct me to the Steve Jackson Games compound?"

Only people looking at the pillows will even know a button exists! If someone does a `@show' on the room, they'll discover it, but most people won't ever do that . . .

As you can see, you can add a lot of neat design elements to rooms without having to spend quota or program a verb!

Aliases

An alias is merely a shorthand way of referring to the object that the MOO will understand. For instance, Coffeeman has the alias Cman. For the football, I might type `@create Generic Ball named "Mentor's Magic Football:football,ball,mmf'.

Now `l mmf', `l ball', `l Mentor's Magic Football' and `l football' would all be interpreted the same way by the MOO.

You can add aliases to *any* object by using the `@rename' command. Its syntax is `@rename <object#> to "NewName:alias1,alias2,...". So if we wanted to give the Living Room the alias `lr', we could type `@rename The Living Room to "The Living Room:lr"'. Another option is to `@addalias alias1,alias2,... to <object>', which does all the same things, but won't let you accidentally rename the object.

In 99% of the cases, you can use an alias anyplace you can use the proper name. So if you have an object "Sushi Tray:tray", you'll get the same thing when you type `look tray' as you do when you type `look Sushi Tray'.

Creating Objects

We've already touched on the `@create' command earlier, when we made our Generic Ball and our football and basketball. Let's get specific.

The syntax for `@create' is `@create <parent object> named "Name:alias1, alias2, etc.'. So we might want to create a flying carpet as follows.

```
`@create $thing named "Mentor's Flying Carpet:mfc,carpet'.
```

Of course, it won't fly yet -- we'd have to add some verbs to make it fly -- but it would at least *look* cool, and that's half the battle. (In a reality based on text, what it says it is, it IS!)

Describing Objects

Once an object is created, you describe it in much the same way as you do a room. `@describe <object name or #> as "Description'. Now, whenever anyone looks at the object, they'll see the description.

Recycling

There will come a time, whether from lack of quota or from simple ennui, that you will want to get rid of one of your objects. To do this, you use the command `@recycle <object#>'. Be careful about this, though! Once you do it,

□
it's gone forever.

When you recycle an object, your quota is increased by one.

It's A Wrap

That's it for part one. This should be enough to get you started creating and building your own objects and rooms. There's no substitute for actually getting in there and doing it. If you make something you don't like, you can always recycle it!

As I said at the beginning, I'm always interested in feedback (good and bad). Please contact me at one of the addresses or MOOs listed at the beginning of this file. Look for Part II of this tutorial, which will cover programming basics, Real Soon Now.

Acknowledgements

Special thanks to all the people who've helped me learn one end of the cow from the other: Coffeeman, Zachary and Taylor on Multiverse and Zippy and Joe on Jay's House. Thanks to Pavel for taking time to read this and comment,

and yduJ, who I've never met, but who gives great tutorial.