2003

Coolest MUD ever

Kevin Radloff
Colby College

Follow this and additional works at: http://digitalcommons.colby.edu/honorstheses

Part of the Computer Sciences Commons

Colby College theses are protected by copyright. They may be viewed or downloaded from this site for the purposes of research and scholarship. Reproduction or distribution for commercial purposes is prohibited without written permission of the author.

Recommended Citation
http://digitalcommons.colby.edu/honorstheses/203

This Honors Thesis (Open Access) is brought to you for free and open access by the Student Research at Digital Commons @ Colby. It has been accepted for inclusion in Honors Theses by an authorized administrator of Digital Commons @ Colby. For more information, please contact enhodes@colby.edu.
The Coolest MUD Ever

Kevin Radloff
Honors Independent Study
Spring 2003

Advisor: Clare Bates Congdon
Department of Computer Science
Colby College
Waterville, ME 04901
Acknowledgements

Thanks to everyone who was there for me during the course of my project. A big thank you to Kyle, Kate, Mari, Sara and everyone on IRC for being my friends, and providing me with endless entertainment and distractions.

Thanks also to Clare, Randy, Marc, and John for their entertainment, motivation, and advice.

Another big thank you to my dad, mom, sister, and stepmom for getting me here and paying for it.

And finally, thank you to Ryan "oGMo" Pavlik and "t'Sade" for initially getting me on the idea of a realistic MUD.
Contents

1 Introduction ........................................ 1

2 Background ......................................... 3
   2.1 MUD Vocabulary .................................. 3
   2.2 A History of MUDs ................................ 4

3 Tools and Methodology .............................. 9
   3.1 Implementation Language: Ruby ................ 9
   3.2 World Design Methodology ..................... 10
   3.3 Code Design ..................................... 11
      3.3.1 The Core and Event Queue ............... 11
      3.3.2 The World Code ............................ 15
      3.3.3 World Storage and OLX .................. 17
      3.3.4 Input Handling ............................ 19
      3.3.5 Environmental Realism ................... 21
   3.4 The Dark Side of Ruby and Our Implementation .. 29

4 Designing a World in CME ......................... 31
   4.1 The Joys of Increased Flexibility ............ 31
   4.2 Caveats of Having More Attributes .......... 33
   4.3 Implementing New Special Objects .......... 33
      4.3.1 Simple Machines: a Light Switch and Fixture . 34
      4.3.2 Complex Machines: an Elevator .......... 35

5 Conclusions and Future Work ..................... 37
   5.1 The End Result ................................ 37
   5.2 The Future .................................... 37

A Source Listings .................................... 41
Chapter 1

Introduction

MUDs, or Multi-User Dungeons, are text-based online multiplayer roleplaying games. Players type commands that allow them to interact with other players in a virtual world, and also interact with the world itself, for example, picking up objects or casting spells.

The goal of The Coolest MUD Ever (CME) was to build the basis of an event-driven and completely extensible MUD engine with unheard of levels of environmental realism. Whereas other MUDs are organized as a set of distinct nodes in which gameplay occurs exclusively of each other node, my MUD will blur the lines between nodes. While players still navigate between set points, actions are not necessarily exclusive to a single node. Depending on environmental factors such as lighting, landscape and weather, players may be able to see and hear what is going on in the surrounding area.

Additionally, in CME world designers are able to easily add new elements to their creations via an embedded scripting language. Agents, minigames, and new commands are all possible without even delving into the base source code. To enable this, Ruby, an easy-to-use object-oriented scripting language with some very nice features, was selected as an implementation language.

To evaluate CME, a test world was built that models the Seeley G. Mudd Science
Building, which houses the Computer Science department at Colby College. The complexities of the environment showed various weaknesses and bugs in the initial implementation of CME, leading to further development of more advanced techniques for producing environmental realism.

After working through the various problems that cropped up, it was shown that environmental realism is in fact quite possible and feasible within a MUD. CME is clearly far from perfect, but it does create a huge domain of new possibilities for the kinds of virtual worlds that can be modelled within a MUD.
Chapter 2

Background

2.1 MUD Vocabulary

When talking in detail about Multi-User Dungeons (MUDs), there's a set of concepts that should be understood:

**room/node** A room (or node in CME terminology) is the smallest unit of space within a MUD. All movement is done between nodes, and most interaction is typically done between players together in a node. Despite the nomenclature of “room,” multiple rooms in a MUD may be used to model a single logical room within the world if the room is large or complex.

**exit** An exit is a keyword that links one node to another. Usually these are preset and include “north,” “south,” “east,” “west,” “up,” and “down.”

**zone** A zone is a logical grouping of nodes that belong to a certain area. This could be a castle, town, forest, or anything of that sort. Data for different zones is often stored in separate files so they can be distributed by their authors to be included in different MUDs if the author is so inclined.

**mobile** A mobile is a generic name for anything that moves around within the world, including players, monsters, and non-playing characters.
item  An item is exactly what it sounds like and can include weapons, armor, potions, food, clothing, backpacks, and other things that mobiles can pick up and use. Items might also include statues, tables, or other large things that players may be able to interact with, but not actually carry.

The start of a MUDing session places the player in a node. The player must navigate between nodes using the exit keywords repeatedly in order to move around. While doing this, the player can pick up or drop items in nodes, or interact with other mobiles by talking or fighting.

2.2 A History of MUDs

Any history of MUDs begins with the popular single-player text-based adventure game called simply “Adventure.” It was written in 1975 by Will Crowther on a DEC PDP-10 and later extended in 1976 by Don Woods. It played somewhat like a single-player Dungeons and Dragons game, with the game itself functioning as the dungeon master (the person in a D&D game who creates and runs the world). In D&D, a group of one or more players assumes the role of adventurers within a fantasy world that the dungeon master has made up. The players navigate through the world on some sort of quest with the dungeon master setting up obstacles and encounters for the players. Each player has various statistics for the character that she is controlling, which are used in combination with dice rolling to calculate the success of various activities including hitting monsters in combat or casting spells. Adventure places the player in a virtual text environment where the game prints a short description of what is around the player, who has to navigate using brief English commands (see Figure 2.1). Various items are lying around within the world that the player must pick up and take along to use in solving puzzles elsewhere in the world. What exactly each item is for is up to the player to figure out.

The next step was a game inspired by Adventure, called “Zork.” Zork was written
2.2. A HISTORY OF MUDS

You are standing at the end of a road before a small brick building. Around you is a forest. A small stream flows out of the building and down a gully.
> go downstream
You are in a valley in the forest beside a stream tumbling along a rocky bed.
> enter forest
You are in open forest, with a deep valley to one side.
>
Figure 2.1: The start of a sample session of Adventure

in 1977 by Dave Lebling, Marc Blank, Tim Anderson, and Bruce Daniels on a PDP-10. It later gained much more popularity after a version was written for the company Infocom that ran on TRS-80 and Apple II personal computers. Zork played very much like Adventure but added a few new features, including more advanced command parsing. Adventure had only accepted commands of two words, but Zork had a more flexible and modular architecture that enabled the authors to create a much more sophisticated command parsing structure.

The first MUD, called simply “MUD,” was created by Roy Trubshaw and Richard Bartle at Essex University in 1978. It was originally written in MACRO-10 for a DECsystem-10, but was later converted to BCPL. The gameplay was modelled after Adventure and Zork, although it focused more on combat. The name was inspired by a variant of Zork called “DUNGEN.” MUD was of course a huge step beyond these because it allowed multiple players to play the game at the same time (putting the “Multi-User” in “Multi-User Dungeon”). The goal of MUD was to ascend to the level of a “Wizard,” that is, a player who had the ability to add on to the world. This was achieved by a system of scoring points to gain power.

Inspired by his time playing the original MUD, Alan Cox created AberMUD in
1987. The name came from where it was written, the University of Wales in Aberystwyth. AberMUD was another combat-oriented MUD. More often than not, especially versus earlier MUDs, wizard status was granted and not earned. Thus, only those chosen, not just anyone, could add on to the world and make new zones.

One of the servers that greatly helped to popularize MUDs was TinyMUD. Originally started as a weekend project by Jim Aspnes in 1989, it grew into something much larger (making its name rather ironic). The gameplay of TinyMUD was centered more around social interaction rather than combat as in previous MUDs, as normal players had the ability to build new rooms and zones. This made the goal to build exciting things, instead of running around killing monsters in areas someone else built. An important facet of TinyMUD was that it ran on a variety of UNIX systems, which greatly eased its widespread adoption.

A major step in the world of MUDs came when in 1989, after much playing of AberMUD and TinyMUD, Lars Penjö wrote LPMud as an effort to create an extensible MUD. LPMuds come in two pieces. The first is the MUD “driver,” which is a core in C that supports a C-like language called LPC. The second piece is the library or “mudlib” which is written in LPC and contains the actual code implementing the MUD. The use of LPC to implement the MUD makes the development extremely flexible. The world can be extended a great deal while the development paradigm remains at a relatively high level. There is even an online editor that wizards can use to write new functionality. Typically, though, LPMuds were also focused around combat like AberMUDs and quite unlike TinyMUDs.

Continuing the tradition of combat-oriented MUDs was DikuMUD, written in 1990 by Michael Seifert, Hans-Heurik Staerfeldt, Sebastian Hammer, Tom Madsen, and Katja Nyboe. DikuMUD subsequently spawned a whole family tree of offspring, including CircleMUD, Merc, Rom, and Phoenix. The goal of DikuMUD was to create a system like AberMUD while fostering more interaction between players. Like with AberMUD, only those chosen as wizards could build new zones.
2.2. A HISTORY OF MUDS

In the realm of socially-oriented MUDs, the next step was the concept of a MOO, or MUD Object Oriented. The first MOO was written in 1996 by Pavel Curtis. In a fashion quite similar to LPMud, the core was relatively bare of functionality but featured support for an object-oriented scripting language that the primary functionality of the MOO was written in. Players could then code arbitrary behavior for any kind of object they wanted by scripting it. Using the inheritance features in the embedded language, players could take advantage of other objects that had been written and extend them to create new ones. Like TinyMUDs, the goal of playing in a MOO is to create an interesting and exciting environment, as opposed to the mindless killing present in many MUDs.

Beyond fun and games, MUDs have in fact been used with success in academic and business arenas. Pavel Curtis and David A. Nichols describes the development and execution of two different MUD systems for use with presentations and collaboration at Xerox PARC [8].

The first system, Astro-VR, allows astronomers to meet online and communicate in real-time. Astro-VR uses a modified version of the popular LambdaMOO server. Aside from providing the standard ability to engage in real-time chat, there is also advanced functionality providing the ability to do real-time presentations with both text and images. Special "conference rooms" have the facilities to prepare a sequence of text paragraphs to go along with graphics that are displayed on viewers' screens.

The second system, Jupiter, is a step beyond this. Jupiter uses a heavily modified MOO codebase to enable audio and video chat along with graphical capabilities for objects within the MUD. The idea of the system is to encourage more casual interaction versus the formal meeting places of the Astro-VR system. Allowing audio chat is the first step toward this, as talking is much more natural than typing. Each room in the MUD has an audio channel associated with it, and all people in the room communicate on that channel. Moving to another room switches a user to that room's channel. Further features for enabling casual interaction include various ways to also
aid in chance encounters. Letting the users know about what is happening around them is a big help here. Using sounds to alert users of other people approaching and mixing in the audio from adjacent rooms at a lower volume level are two ways of granting this awareness.

The games and MUD servers mentioned above are but a brief survey of important milestones in MUD evolution. All along the way the codebases mentioned have given birth to many offspring, as programmers came along and decided to build upon the work of others and then in turn left their modifications for others still. I missed most of this history; my primary experience is with various modifications of DikuMUD derivatives (namely CircleMUD, ROM, and Merc), and my work reflects that.
Chapter 3

Tools and Methodology

3.1 Implementation Language: Ruby

A MUD is a rather clear-cut example of a situation where an object-oriented paradigm fits perfectly. Everything in a MUD is an object; the gameplay is really all about the manipulation of objects. A MUD is made up of mobiles, nodes, and items, perfectly suited to being encapsulated by objects within the implementation language. Thus, the primary criterion for the implementation language of CME was that it be object-oriented.

Examining the available object-oriented programming languages revealed one clearly superior option, Ruby. Ruby is a scripting language quite similar in syntax to the popular language Perl, though it differs in other important aspects. Everything in Ruby is an object, including primitives like numeric constants. The powerful object-orientedness combined with the familiarity given by my years of experience with Perl made Ruby a clear standout.

Ruby also has many other useful features. It includes a standard library with all of the expected common data structures and objects for file and network I/O, serialization, and other basic functions. Additionally, Ruby has a powerful framework for passing blocks of code around (like lambdas, in Lisp) to later execute them in their
original context. Multiple inheritance is not directly supported in Ruby, although it does have an equally functional alternative: methods may be declared in a module instead of a class declaration, and any number of modules may be included in and have their methods inherited by a class. These modules may not be directly instantiated as a class can, but they are very useful for grouping common functionality that will be used by multiple different classes.

An additional boon to using an interpreted scripting language is that it is a simple matter to load new code or reload existing code to implement new features or fix a bug. This both speeds up testing and gets rid of the annoyance factor that exists in MUDs written in compiled languages, where changes require the whole MUD to be rebooted, which is very disruptive to the players.

3.2 World Design Methodology

As discussed previously, a typical MUD is constructed of many different rooms. All navigation throughout the virtual world is accomplished by moving between these rooms (which may model an actual room, or just a discrete space outside). Rooms are connected by exits, which are typically a set of predefined keywords that include “up,” “down,” and the four cardinal directions. Any given room may connect to others with some, all, or none of its exits. These exits are also one way, so that a room with an exit “east” to another room does not necessarily imply an exit “west” connecting to the first room from the second.

With some very limited exceptions, all gameplay interaction takes place within a given room. Players have very little to no knowledge of what is happening around them. For example, the “say” command is used for normal conversation; however, if a player types “say hello”, then only players in the same room will see “Kevin says, ‘hello’.” There is no allowance to realistically model environments that require multiple rooms. One example of this is a hallway, with multiple doorways off of it.
To have more than two doors off of the hallway (apart from the ends), it is necessary to construct it from more than one room. The downside of this approach is that each section of the hallway is subsequently isolated from every other part. Whereas realistically someone speaking in one part of a straight hallway would be seen and heard by a second person in another part of the hallway, this is not the case in the virtual hallway that has been split up into multiple rooms.

Ideally, it would be highly desirable for events like a person talking to be perceived outside of the immediate location in which they take place. To enable this capability, CME implements a new paradigm where the world designer has the ability to define attributes to allow for much higher levels of realism. With the world more fully described, the MUD is able to adequately model environments such that the movement of light and sound is handled more realistically. In the previous example then, it would be simple to make a hallway in CME consisting of multiple rooms (or nodes, as they are called in CME) where it is possible to see and hear everything that is going on in the hallway. Also, the hallway could have a curve in the middle, blocking visual perception of events on either end, but still allowing players to hear audio.

3.3 Code Design

3.3.1 The Core and Event Queue

At the core of any MUD is a loop that periodically (often every 1/10th of a second, as in our case) checks for input from the connected clients and sends queued output. All I/O operations are done through a Client object that is associated with a player. Any input received is run through the command parser by the Client object. Typically, the code associated with a command is immediately executed. Any other code that is scheduled for delayed execution or is executed periodically is done so as part of the main game loop. For example, one common method of executing combat is to
iterate through all fighting mobiles every 2 seconds (20 spins of the loop) and do the necessary calculations and output for a round of combat at that time. Of course, this method produces a rather artificial environment where every mobile takes its swings in combat at the exact same time.

What CME does to allow for more natural and graduated execution of in-game events is to have an event queue which executes pending events on every iteration of the main loop. The programmer specifies the offset (as multiples of the loop resolution—so 1/10th of a second) and a block of code to be executed, and the code block is efficiently stored in a delta queue to await its time of execution.

A delta queue is a specialized priority queue (see Figure 3.1). Elements are ordered with a delta that denotes the time offset from the previous element indicating when they should be removed from the queue. The first element has a delta relative to the current time. Giving each element a delta instead of an actual time makes it more efficient because only the front element need be checked on each iteration. Thus, every time around the loop, if the front element has a delta of 0, it is removed. This action is repeated until an element with a delta of greater than 0 is at the front, since subsequent elements could also be scheduled to execute at the same time. Regardless of how many elements were removed during this process, the front element then has its delta decremented.

This creates a whole new paradigm for the design of code. In the case of the command parser, commands are still parsed as they are received from the client, but then the command parser schedules execution of the actual Command object associated with the command with an offset of 0. Within the main loop, input is received, commands are parsed, the event queue dispatches events, and then output is sent (see Figure 3.2). Thus, commands received during a given iteration of the loop will be scheduled for dispatch by the event queue and then subsequently executed as the event queue runs later in the same iteration. Resultant output will also be sent immediately thereafter. In this way, command execution is interleaved seamlessly
Delta Queue

Figure 3.1: A delta queue
with other events.

What the event queue means for periodic code execution is that instead of having global timers for executing a certain function across a set of objects within the MUD, the objects themselves can be responsible for locally executing equivalent functionality on their own schedules. That is, instead of iterating across all mobiles on the MUD every 50 iterations of the main loop to do a five (5) second upkeep, mobiles can schedule at their creation the code to be executed in five seconds. The scheduled code can then in turn schedule for itself to be executed in another five seconds. Thus, we have equivalent functionality, but each object is responsible for itself, and on its own schedule. This is a more real-time, realistic, and efficient method; objects only need to keep scheduling the upkeep if they need it, and there's no overhead of iterating through large numbers of objects that don't need it.

Figure 3.2: The order of execution in the main game loop
3.3. CODE DESIGN

Figure 3.3: The standard method versus passage paradigm

3.3.2 The World Code

Zones, Nodes, Passages, Items, and Mobiles

The inherent object-encapsulation already present in viewing our MUD world, which consists—at the most basic level—of only zones, nodes, mobiles, and items, dictates an object hierarchy for us. The obvious design then is to have a base class, Entity, and then have four child classes—Zone, Node, Mobile, and Item—which form the basis of everything within the MUD.

What’s not so clear is that while this would take care of everything in a classical MUD—that is, the world consists only of nodes and the mobiles and items within them—it isn’t quite flexible enough for everything I had in mind. The way that nodes are connected in typical MUDs is that there’s a set of preexisting exit names (usually north, south, east, and west) each of which may be set as an exit to another node. There is nothing special about these exits: they’re just a link to another node. However, the requirements in CME for advanced functionality within exits makes having a separate Passage object make a lot of sense (see Figure 3.3).
Using modules to generalize functionality

Having thus defined a base for objects within the world, Ruby makes it possible to mix and match functionality into them using modules (see Figure 3.4). The real power of separating nodes and passages comes from the fact that at the same time that the concept of each is removed from the other, it is possible to retain the functionality in both by having them both include the Environment module, which handles the environmental realism.

There are many other cases where Ruby’s multiple inheritance via modules makes things simpler. Multiple disparate classes within the world are also containers; mobiles have an inventory, nodes can have items in them, and there are likely to be container items like backpacks and bags in the world.
3.3. CODE DESIGN

3.3.3 World Storage and OLC

OLC stands for “Online Creation.” It is the standard paradigm for creating MUD world data. Typically, it consists of either a command-line or menu-based system in the MUD itself that is used to create and manipulate every object within the world. The OLC in CME is based around a single command, “olc.” It allows world designers to create new instances of objects, change existing instances, list all objects of a type, or delete an object. It operates on zones, nodes, items, mobiles, and passages.

The basic concept of how the world is stored is that zones are the top-level structure. The world of CME exists as a set of zones. Each zone contains its own set of nodes, items, and mobiles. Each node also contains a set of passages. Given this, objects in the MUD use a hierarchical id system as a reference:

```
zone#
zone#. (node#|item#|mobile#)
zone#.node#.passageKey
```

Zone, node, item, and mobile ids are numbers, although they are stored internally as strings. Passages are referenced by their keyword. The zone is the id of the zone containing the node, item, or mobile. In the case of a passage, it is prepended by the id of the zone and node that contain it. Node, item, and mobile ids are separated within a zone, so that a given zone may contain a node, item, and mobile all with the same id. This identification system exists for two reasons. The first is for the aid of world designers so that there is actually a way to easily refer to any object within the world while using OLC. The second reason is so that within the code, two loosely coupled objects can refer to each other. In other words, an object can store the logical id of a second object when they are not directly related and it is not feasible to store a pointer (which could change), and then call a function that will resolve the id to a pointer when it is needed.

All of these objects are also stored within a hierarchical fashion (see Figure 3.5).
A global hashtable contains references to all zones. Within each zone are three more hashtables which contain references to nodes, items, and mobiles contained within the zone. Within each node is yet another hashtable which contains its passages. In the case of nodes and passages, these are the actual nodes and passages which players walk around in within the world. However, the items and mobiles referenced and stored in this way are just templates; they are entities which have been set up (with descriptions and other necessary information) and are ready to be placed in the world, but do not actually necessarily exist anywhere in the world. To have template mobiles and items that exist in a zone load in a node at MUD startup, world designers use another variation of the OLC command.

This method of having a list of mobiles, items, and nodes for each zone comes from DikuMUD, although DikuMUD itself uses a fixed-size array that limits the number of objects in each zone. The id system is also a creation of my own; since DikuMUD has a fixed-size array of 100 for each type of object, objects are identified by an integer.
3.3. CODE DESIGN

that is the zone number (a non-negative integer) appended with the array index (00 to 99). Mobile 54 in zone 12 would be referred to as 1254.

3.3.4 Input Handling

The InputHandler class provides a framework for generalized input handling in CME. All input is passed to the InputHandler that is specified in the Client object corresponding to a given user. During normal gameplay, this points to a singleton instance of the CommandParser class. This class implements a command parsing system that handles the execution of all gameplay commands. At any time during play, the reference to the command parser may be replaced by another child of InputHandler—causing the Client object to pass input from the player to the new object instead—to accomplish specialized input functionality that cannot be handled by normal command-line input. Applications of this include online editing or prompts for nonstandard input.

Command Parser

The command parser is very loosely coupled to commands. That is, the command parser has no information about any commands that exist. All information on commands is defined in each command's class. Commands—which are children of the Command class—register themselves with the CommandParser class and are subsequently stored in its hash of commands. This method of implementing commands is nice because adding commands means touching only a single file (the one that contains the command itself). The downside is this approach does not provide any sense of order in terms of some commands having priority over others, although there is not currently incomplete command name matching so it does not matter. If the command parser accepted abbreviations for commands (for example, "l" instead of "look"), then there would have to be some sort of mechanism to denote that "l" and "lo" should match "look" and not "logout" or another command that the user would
be much less likely to be intending to type.

Input is matched against the full name of commands using the standard hash lookup methods to find the instance of the command object. Once retrieved, three criteria are checked before execution; each command has a minimum level, a minimum position (standing, sitting, etc.), and a flag indicating whether it can be done during combat. If these tests are passed, then the mobile executing the command and the arguments are passed to the command object’s run method.

Command Types

There are actually three types of commands: basic, tool, and special. What specifically happens when the command parser runs a command depends on the type of the command.

The majority of commands fall under the basic category. These commands are the normal type that simply execute according to what is defined in the command object, with no external prerequisites aside from the initial criteria mentioned above.

The second type of command is the tool type. This category is designed for commands that also have predetermined functionality, but require the mobile executing them to have a “tool” item. The tool does not do anything special except for the fact that it must have an empty method definition by the name of cmd_ plus the name of the command. The inventory (and surroundings, if so defined) of the mobile is automatically searched for the tool, and if one is not found then the failure is reported to the mobile.

The last type is the special command type, which is used for implementing commands within items. Any command which an item is to implement must actually exist as a global special command. This is superior to the commonly used method of allowing (and searching for) anonymous commands implemented by objects within the world. That is, allowing any random object sitting around in a node within the MUD to define a special command that is not defined anywhere else. If the command
3.3. CODE DESIGN

The real convenience in having object-oriented and pluggable input handling shows up when we move to an example outside of the normal command parsing. The Editor class is a fully self-contained online editor for players. There is no need for there to be any state tracking elsewhere; if a command requires multi-line input from a player, then all it needs to do is construct an Editor object and pass the player object and a block of code that is to receive the text input after the user has completed editing. The supplied code block should contain whatever code will handle what needs to be done with the text from the editor. The Editor object handles switching of the player’s input stream to itself. It can then just wait for the player to input lines of text, which it appends to the editing buffer. When the user is done, it switches the player back to the CommandParser and then calls the supplied code block, passing the input buffer as an argument.

3.3.5 Environmental Realism

Environmental realism can be accomplished in CME because of the extra parameters that world designers are able to set for nodes and passages.
Environments

The Environment module is included by Nodes and Passages to enable the environmental realism functionality. This is based mostly around three attributes: audio, video, and distance. The values of audio and video define how much light and sound is filtered by the environment (on a percentage scale), and the value of distance signifies the length of a space and hence additional filtration. The concept of distance is currently used only for passages and not for nodes. The ideas behind the values of audio and video differ slightly between the two of them, because of the limited information about the world that is encoded within Nodes and Passages, and differences in how light and sound are perceived.

Environmental Messages

Whenever an event occurs that is to be broadcasted according to the environment, an EnvironmentalMessage object must be created and passed to the Node object in which the event occurs. The message is composed of two parts. These two parts are the text of the audio component of the message and the text of the visual component of the message. Additionally, there is a flag that denotes what type of an event this message is conveying, and amplitudes for both the audio and video components.

The type of the event determines how the audio and video text is used and which one is preferred in the case of a mobile being able to perceive both portions of the event. In a normal event—say, someone sitting down—the video portion would be "sits down." The audio for this might be "rustling," to denote the sound of a person's clothing as they made this motion. If a mobile can both see and hear the person sitting down, then the video component is preferred over the audio, as the audio becomes somewhat redundant if the mobile is aware that someone did indeed sit down. Reporting only the video part completely ignores that there may have been a sound associated with the event, but in most cases the sound is implied by the user's conception of the event which is conveyed by the video itself. On the other hand, if
an event is of the speech type, then the audio is the actual spoken words, and so is preferred over the video when both are perceived. The video portion is ignored and automatically filled in as the speaker's lips moving.

There are also some subtleties with regard to what the audio and video amplitudes represent. This ties in to what was mentioned before about the limited information still known about the world even though there is a lot more known that in a typical MUD.

Audio is non-directional, so it can be heard as long as there's not too many obstacles between the source and the listener. This way, the values in a Node or Passage object for the audio filtration mesh perfectly with the amplitude given in the event message to create a quasi-realistic mechanism. Although the method currently used is not very scientific, the sense of "this is how loud an event was" and "this is the dampening effect of the environment" fit exactly with what is happening in reality. On the other hand, the video amplitude is not quite as straightforward or realistic. Line of sight is taken into account when event messages pass through a Passage object, but line of sight on a more local scale cannot be done; there is no way to model part of a node being blocked by an obstacle except to replace said node with two, which is not necessarily a solution. The video amplitude then must stand for how obvious or subtle the event is visually. The video filtration for an environment can only model how well an environment blocks vision in a general sense; it can only model something like "the doorway makes it possible to see this well into a room versus the space being open."

Message Passing

To show an example of how messages are passed around, imagine a set of nodes arranged as shown in Figure 3.6. Nodes 3 and 4 are part of a hallway. Nodes 1 and 2 make up a room off of the hallway. Neither nodes 1 and 2 nor nodes 3 and 4 have any obstructions between them. Between nodes 2 and 3 is an open door, while a window
in the wall makes it possible to see between nodes 1 and 4.

Now imagine an event occurring in node 4. Let's say that Kyle sits down. The message contains two components. The first is the audio part, which contains the sound Kyle makes when he sits down, which is "rustling." The second component is the visual part of this event, which in the case of Kyle sitting down is "sits down." As said before, both of these components also have a number signifying the initial amplitude.

When the event happens, the event object is handed to the node 4 object, which proceeds to attempt to pass it to all the connections it has. These happen to be its two passages. The two passages attempt to pass it on, with each of them connecting
3.3. CODE DESIGN

to a single node. The process then repeats until the event has traversed as far as it can. As this traversal is happening, there are actually two things going on, because of the separate audio and video components of the event (see Figures 3.7 and 3.8). Each time the event is passed, the amplitude of the audio and video are degraded depending on the parameters of the environment they are passing through, whether it is a Node object or a Passage object. If both of the amplitudes reach zero, then the event is not passed. If the event is passed to an environment which does not have line of sight to the source, then the video immediately goes to zero. Also, what tends to happen—since usually a passage going from one node to another has a corresponding passage going in the other direction—is that the event will end up on its way back to a node that it has already visited. It may not, however, visit a node a second time, except under special circumstances.

What makes the algorithm more complex is the fact that visiting a node a second time can be quite necessary. In our example, there are two paths from node 4 to node 1; sound can travel through nodes 2 and 3 to 1, and video can travel directly through the window. If the event cannot revisit node 1, then only either the video or audio will arrive there; whichever path happens to be followed first will "win out." The person watching Kyle through the window would only see him sit down or hear him sit down, but not both. What is done then is that the source and amplitude of both the highest amplitude audio and the highest amplitude video are recorded within each node. If and only if a secondary path of the event message can beat either of these amplitudes is it allowed to revisit a node, where it can replace either or both of the recorded maximums.

After the event has traversed as far as possible, the message text is formatted for each mobile in each node the event has successfully visited. As stated above, where a player can detect both the video and audio components, the video is favored in the standard case. In the example, players in the nodes with line of sight to Kyle in node 4 will both see and hear him, while players in node 2 will just hear rustling. The
Figure 3.7: The audio traversal
Figure 3.8: The video traversal
Figure 3.9: The end result; note the favoring of video versus audio
extra factor at this point, however, is lighting. The light levels at the origin of the event and the light level in the node that the mobile is further affect whether it can perceive the event. Besides that the light level at the origin could be too low for the mobile to see the event, if the light around the mobile is too high compared to the light at the origin then the mobile will also not be able to see it. On the other hand, if the light at the origin is exceptionally high compared to the light around the mobile, then the mobile won’t see the event. These signify that the mobile’s eyes would be adjusted to a certain light level, so that the mobile would be unable to see in light too much brighter or dimmer than that in its immediate environment.

3.4 The Dark Side of Ruby and Our Implementation

As exciting as all of this is, there is a downside to using an interpreted language like Ruby. Ruby does indeed give programmers a lot of power to extend CME and makes it quite easy at the same time, but it is necessary to be careful with this.

The problem is the nature of an interpreted language. Ruby will happily read a new file that a programmer inserts into the source file and redefine parts of any class the author felt like redefining, regardless of where it is. When Ruby encounters a class definition, it doesn’t care much if, where, and who made a previous declaration of that class or where the new definition is; it will forget about the old code and insert the new.

This means that it can be dangerous to write code if the author is sloppy. Classes in the utility library could get overwritten, or more importantly the main CME classes could get corrupted. Thus, it is imperative that all programmers on the MUD are trusted users. It is far too easy for someone to hijack the system. Sadly, this also means that a situation like an LPMud or a MOO where any user might have the ability to make new functionality could never happen.
Chapter 4

Designing a World in CME

Of course, the codebase of a MUD is just the half. Without a world to go along with it, even the Coolest MUD Ever would not be very exciting at all.

4.1 The Joys of Increased Flexibility

Modelling an area in CME feels much more real than in other test-based MUDs. Instead of thinking strictly in a grid pattern, the designer is free to mold things more appropriately to the environment. Thinking so specifically about bits like the angle of an exit, or how much sound and light pass between two areas makes the world that much more real in the mind of the designer.

For a test world, I chose to model the Seeley G. Mudd Science Building, the building containing the Computer Science Department at Colby College. This seemed like a good choice because the primary goal of testing was to see how the environmental realism works, and I know in particular the 4th floor of Mudd very well; modelling something extremely familiar to me in real life made it more scientific as I could attempt to make things exactly as I know they are in real life and then see if it works out correctly in the MUD. Additionally, it was quite amusing to be able to say I was building a Mudd MUD.
The 4th floor of Mudd has some interesting features that were critical in testing the environmental realism functionality of CME. Each floor of Mudd is centered around an L-shaped hallway (see Figure 4.1). Each end of the hallway has doors leading to stairwells, with an elevator also on one side. On the inside of the L on the 4th floor are two labs, the so-called Mudd Lab and the Linux Lab. Both of the labs have reinforced glass windows looking out into the hallway. The Mudd Lab is located at the corner of the hallway, with its windows going the entire length of the walls, giving anyone in the hall a complete view of the lab, and vice versa. The Linux Lab is located next door, with a window that gives a view of the inner half of the lab. The Linux Lab is actually the example given in the previous section.

The windows in both of these labs give rise to the situation where there are two paths between two nodes, with audio and video effectively favoring alternate paths. The original message-passing algorithm would let a given event visit a node only once.
This led to the situation where someone standing inside one of the two labs could only see or only hear someone in the hallway, where both should have been possible. Many long hours of agonizing over the situation led to the realization of how the single-visit functionality was completely messing things up.

4.2 Caveats of Having More Attributes

On the downside, having to put in a lot more attributes for every piece of the world can get somewhat annoying after a while. There is roughly twice the information as on a normal MUD to input for each node. Additionally, there is a comparable number of things to set for each passage, whereas typically there is no effort involved in connecting rooms. In the end, this results in having to put in quite a bit more information, since typically passages outnumber nodes by a good margin. The simple act of putting in lots of numbers can get quite prone to errors after a while.

Is it worth the tradeoff to have to work harder to get a more realistic world? I think so. The extra attributes default to settings where events will stay local to a node, so effectively leaving things as-is will default to the behavior of a normal MUD. However, it is wonderful to be constructing the world and at the same time be able to see it take shape \textit{realistically} as you interact with the environment.

4.3 Implementing New Special Objects

Faced with various man-made gizmos in Mudd (such as lights, blackboards, elevators, etc.), I decided to take on a couple of these as tests of the ease of expanding CME without having to delve into the base gameplay code. Not surprisingly, during the course of implementing these I ran into features that in fact should have been present in the base gameplay code but was not. Foremost among these was a generic method of defining and setting attributes for custom classes necessary to implement
the aforemented gizmos. Without this, custom attributes in new classes wouldn't be possible without changing the OLC and world loading and saving code, which of course makes no sense. Hence, a system where specialized classes could define arbitrary attributes and helper code to get and set them was added. The OLC system uses this helper code to be able to get and set the custom attributes without requiring any changes.

### 4.3.1 Simple Machines: a Light Switch and Fixture

My first attempt at custom objects was creating a light switch and light fixtures to put in the hallway. At the time I had just added support for variable light levels, so it seemed to make sense to add something to take advantage of it. Another level of realism is added, since there are indeed two light switches at each end of the Mudd hallways that control the lights for the two arms of the hallway.

A light switch is incredibly simple. There are two states: on and off. Flicking the switch sends either the message “on” or the message “off” to the light fixtures that are wired to the switch. The fixtures are equally as simple, with the same two states. In the off state, the fixtures put out no light. In the on state, the fixtures put out some preset amount of light.

In terms of the MUD, this gives us two objects, LightSwitch and Light, which are both child classes of Item. LightSwitch contains a cmd_switch method to implement the “switch” command. This command takes an argument of “off” or “on,” and sends the appropriate message to its associated Light objects. The state for both objects is switched, and the lights go either on or off.

Then comes the tricky part. Due to the way that objects are stored and then later created within the MUD, there is a “chicken and egg” problem between the LightSwitch object and its associated Light objects; that is, we don’t know whether the Light objects or the LightSwitch is going to be created first. This means that the LightSwitch object can’t rely on finding a reference to the Light object and
vice versa. Since they need to be connected somehow, it is necessary to bring in a third object. Thus, a global Hash object is created outside of the class code (so it will be initialized at startup, before the world code is loaded), which all Light and LightSwitch objects can reference. Both classes have a custom attribute that acts as a "channel" to communicate on. Whatever object is created first uses this attribute to look for an Array referenced in the Hash. If one is not found then it is created. Subsequent LightSwitch objects populate the array with references to themselves, so that the light switch can iterate through the array and easily message them all when it is switched on or off.

4.3.2 Complex Machines: an Elevator

My appetite whetted by the success with the lights, I decided to try something a bit more complex: an elevator. Of course, Mudd would not have been complete without its incredibly slow and exceptionally tiny elevator that also has far too many red, alarm-type buttons on the inside.

An elevator functions by interaction between three components: the elevator itself, the doors on each floor, and the buttons on the inside and outside of the elevator. In CME, this translates to two classes. The first is the Elevator class, a subclass of Node, which models the elevator itself and handles the opening and closing of doors. The second is the ElevatorButton class, a subclass of Item, which handles both inside and outside button operation. The elevator object has an attribute that contains values for the floors it services, which node and exit are associated with each floor, and a value (from $-\infty$ to $\infty$) that signifies where along a vector the floor exists (the shaft, if you will). The buttons are set to be for a certain floor. If a button is for the outside, it also has a direction. Both objects also have a name which is used as an index into a hashtable so the buttons can message the elevator, in the same fashion as in the light switch and light implementation.

When a button is pressed, it does indeed send a message to the elevator along
with which floor and which direction the call is for. The elevator subsequently queues the call, after using the information about floors that it has to find out to which node it is going to and which direction it needs to go. In order to be able to function like a real elevator and service calls that are along the way, the elevator functions by moving only one unit along the shaft every half second. The easy way would be to merely schedule an event an appropriate time in the future to signify travelling to the next floor, but this method would be hard to use if we want to service intermediate calls. By scheduling events every half second to move one unit and keeping track of the destination instead, the elevator can function in realtime. If the elevator is servicing a call, new calls will be queued and then checked to see if they are on the way. If a more recent call is on the way, then it will be switched to the destination and the old destination will be put onto the front of the call queue.

Upon arriving at the destination, the elevator uses the passage id associated with the floor to open the door at the same time that it opens its own door and updates its own passage to point to the outside node. By having the elevator do these operations itself, the need for an additional class to provide some sort of elevator door functionality is avoided.

The result of all this is a quite realistic elevator. It is fully customizable and can be placed by the world designer anywhere within the world as many times as the designer wants without touching any code.
Chapter 5

Conclusions and Future Work

5.1 The End Result

In the end, the results are rather pleasing. It has been shown that environmental realism is in fact possible within a MUD. There are certainly weak points in the current implementation that make it awkward or impossible to model some kinds of environments, but it is definitely a step in the right direction.

At the same time, CME has been shown to be easily extensible. Both relatively simple and rather complex extensions have been implemented with complete success, without having to do anything special within the main source code. The placement of these special objects is also especially easy and convenient, requiring nothing more than the standard world-building commands.

5.2 The Future

The major thing missing from CME at the moment is just a lot of the standard MUD functionality. Gameplay features like combat, spells, different races and classes are all missing. The current set of commands extends just barely beyond the realm of the bare essentials for navigation.
Essentially, all of these things boil down to creating flexibly designed attribute (in several respects of the word) systems. First of all, one of the current issues with the general design is the mess of attributes that are in place in the core gameplay classes. As previously discussed, I created an expandable attribute system that allows custom classes to define their own attributes without having to change any other code to use them. What would likely be a good idea and a cleaner solution would be to expand and extend this system to be easier to use for core attributes and then change the main classes to use it.

In a similar vein but a somewhat different issue is the question of how races and character attributes (strength, intelligence, charisma, etc.) should function. Tied in to this is a skill and spell system, most of which typically use character attributes to calculate success and/or power. I want all three of these to be customizable by a MUD creator, so a bit of design magic will be required.

Other minor bits are a system for communications channels and more robust logging. MUDs normally have a set of global “channels” for different purposes that users can turn on or off as they desire. The idea of having a predesigned system for these is so that it is easy to define new ones if the need arises. Typically, there is also a command that allows users to customize the colors of the different channels to their liking. Likewise, there needs to be a logging system that prints messages to all wizards, depending on what log level they have set for themselves.

The big question that remains is how the environmental realism will affect gameplay as it continues to take shape. Some of the hopes are that things like hiding and stalking enemies will take on a whole new shape, but how far-reaching the effects are remains to be seen.
Bibliography

[1] A history of 'Adventure'. Available WWW:

[2] Early MUD History. Available WWW:

[3] Internet history, design, advanced usage, security issues, and more. Available


[6] MUD Information and History. Available WWW:


[8] Pavel Curtis and David A. Nichols. MUDs grow up: Social virtual reality in

Appendix A

Source Listings
module ANSI

RESET  = "\[0m"
BOLD   = "\[1m"
ITALIC = "\[3m"
UNDER  = "\[4m"
INVERSE = "\[7m"
STRIKE = "\[9m"

NBOLD = "\[22m"
NITALIC = "\[23m"
NUNDER = "\[24m"
NINVERSE = "\[27m"
NSTRIKE = "\[29m"

BLACK = "\[30m"
RED = "\[31m"
GREEN = "\[32m"
YELLOW = "\[33m"
BLUE = "\[34m"
MAGENTA = "\[35m"
CYAN = "\[36m"
WHITE = "\[37m"
DEFAULT = "\[39m"

BBLACK = "\[30;1m"
BRED = "\[31;1m"
BGREEN = "\[32;1m"
BYELLOW = "\[33;1m"
BBLUE = "\[34;1m"
BMAGENTA = "\[35;1m"
BCYAN = "\[36;1m"
BWHITE = "\[37;1m"

BBGBLACK = "\[30m"
BBRED = "\[31m"
BBGREEN = "\[32m"
BBYELLOW = "\[33m"
BBBLUE = "\[34m"
BBMAGENTA = "\[35m"
BBCYAN = "\[36m"
BBWHITE = "\[37m"

TRANS = {
  "x" => RESET,
  "r" => RED,
  "y" => YELLOW,
  "g" => GREEN,
  "c" => CYAN,
  "b" => BLUE,
  "m" => MAGENTA,
  "w" => WHITE,
  "k" => BLACK,
  "t" => BRED,
  "y" => BYELLOW,
  "q" => BGREEN,
  "c" => BCYAN,
  "b" => BBLUE,
  "m" => BMAGENTA,
  "w" => BWHITE,
  "d" => BBBLACK,
  "o" => BBRED,
  "t" => BBGREEN,
  "j" => BBYELLOW,
  "u" => BBBLUE,
  "y" => BGMAGENTA,
  "p" => BBWHITE,
}

end

class String
  def ansi
    last = nil
    saved = nil
    self.gsub!(/\((.11l
      r = ''
      if ANSI::TRANS[$1]
        r = ANSI::TRANS[$1]
        last = r
      elsif $1 == '<'
        if last
          saved = last
        end
      elsif $1 == '>'
        if saved
          r = saved
        end
      else
        r = $1
      end
    }
    self
  end
end
require 'commandparser.rb'
require 'mobile.rb'

module Command
  class Base
    attr :minlevel
    attr :position
    attr :infight
    @@_instance = nil
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::DEAD
      @infight = true
    end
    def run(mobile, cmdline)
      end
    def Base.Instance
      @@_instance
    end
    end
  end
end

require 'command.rb'

module Command
  class Time < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::DEAD
      @infight = true
    end
    def run(mobile, cmdline)
      time = WorldTime.now
      mobile.textOut("It is hour \$(time.hour)(x on the day of \$(time.weekDay), day \$(time.day)(x of the month of \$(time.monthName), in the year \$(time.year))x\n\n".wrap)
    end
    def run(mobile, cmdline)
      end
    def Base.Instance
      @@_instance
    end
    end
  end
end
CommandParser.register('time', Command::Time.Instance)
require 'command.rb'

module Command
  class Desc < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::DEAD
      @infight = false
    end

    def run(mob, cmdline)
      if mob.is_a?(Player)
        if cmdline == ""
          mob.textOut("Your current description is:

            (Player)
          ")
        elsif cmdline.downcase == 'edit'
          Editor.new(mob)
        else
          mob.textOut("No changes.
          ")
        end
      else
        mob.textOut("Do what now?\n"
      end
    end

    @@_instance = Desc.new
  end
end

CommandParser.register('desc', Command::Desc::Instance)

require 'command.rb'

module Command
  class Inventory < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::DEAD
      @infight = true
    end

    def run(mob, cmdline)
      str = "You are carrying:\n"
      if mob.objects.size == 0
        str << " (nothing)\n"
      else
        mob.each_item { |i| str << " (#{i.sdesc})\n"
      end
    end

    @@_instance = Inventory.new
  end
end

CommandParser.register('inv', Command::Inventory::Instance)
CommandParser.register('inv', Command::Inventory::Instance)
require 'command.rb'

module Command
  class Quit < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::DEAD
      @infight = false
    end

    def run(mob, cmdline)
      if mob.is_a?(Player)
        mob.deactivate
        mob.client.close
      end
      @_instance = Quit.new
    end
  end

  CommandParser.register('quit', Command::Quit::Instance)

end

require "tool.rb"

module Command
  class Special < Tool
    def initialize(toolCommand, needCarry = false)
      super(toolCommand, needCarry)
    end

    def run(mobile, cmdline)
      name, args = cmdline.split(" ", 2)
      spec = Util::parseFindSpec(name)
      if name == "" || spec || spec.all || spec.count > 0
        mobile.textOut("You can't find it.
"
      return
      end
      if !needCarry
        if !(obj = mobile.node.findItem(spec))
          obj = mobile.node.findMob(spec)
        end
        if obj
          obj = mobile.findItem(spec)
        end
        if obj && obj.respond_to?(@toolCommand)
          obj.send(@toolCommand, mobile, args)
        else
          mobile.textOut("You wouldn't know what to do.
"
        end
      end
    end
  end

end

require "tool.rb"

module Command
  class Special < Tool
    def initialize(toolCommand, needCarry = false)
      super(toolCommand, needCarry)
    end

    def run(mobile, cmdline)
      name, args = cmdline.split(" ", 2)
      spec = Util::parseFindSpec(name)
      if name == "" || spec || spec.all || spec.count > 0
        mobile.textOut("You can't find it.
"
      return
      end
      if !needCarry
        if !(obj = mobile.node.findItem(spec))
          obj = mobile.node.findMob(spec)
        end
        if obj
          obj = mobile.findItem(spec)
        end
        if obj && obj.respond_to?(@toolCommand)
          obj.send(@toolCommand, mobile, args)
        else
          mobile.textOut("You wouldn't know what to do.
"
        end
      end
    end
  end

end
require 'command.rb'

module Command
  class Tool < Base
    #attr :toolCommand
    attr :needCarry
    def initialize(toolCommand, needCarry = false)
      @toolCommand = toolCommand
      @needCarry = needCarry
    end
    # We just find a tool and execute the (hopefully overridden) method
    def run(mobile, cmdline)
      obj = findTool(mobile)
      if obj
        doCommand(mobile, cmdline)
      else
        mobile.textOut("With what?\n")
      end
    end
    # What the command actually does goes here
    def doCommand(mobile, cmdline)
      end
    end
    private :doCommand, :findTool
  end
end

require 'command.rb'

module Command
  class Close < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::STANDING
      @infight = true
    end
    def run(mobile, cmdline)
      if (passage = mobile.node.findExit(cmdline)) && passage.visible
        if passage.door
          passage.door.open = false
          mobile.textOut('You close the #<passage.name>.
')
        else
          mobile.textOut('You can't do that.\n')
        end
      else
        mobile.textOut('It's already closed.\n')
      end
      @instance = Close.new
    end
  CommandParser.register('close', Command::Close.Instance)
end
module Command
  class Drop < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::RESTING
      @infight = true
    end

    def run(mobile, cmdline)
      spec = Util.parseFindSpec(cmdline)
      if target = mobile.findItem(spec, mobile)
        mobile.drop(target)
      else
        mobile.textOut("You can't find it.\n")
      end
      @@_instance = Drop.new
    end
  end
end

CommandParser.register('drop', Command::Drop::Instance)

require 'tool.rb'

module Command
  class Erase < Tool
    def initialize
      super("erase", true)
      @minlevel = 0
      @position = Mobile::Pos::STANDING
      @infight = false
    end

    def doCommand(mobile, cmdline)
      spec = Util.parseFindSpec(cmdline)
      if target = mobile.findItem(spec, mobile)
        mobile.erase_off(mobile, target)
      else
        mobile.textOut("You can't erase anything on that.\n")
      end
      @@_instance = Erase.new
    end
end

CommandParser.register('erase', Command::Erase::Instance)
require "command.rb"
require "util.rb"

module Command
  class Get < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::RESTING
      @infight = true
    end

    def run(mobile, cmdline)
      Logger.log(mobile.to_s + " : Look", LogLevel::INFO)
      if cmdline == ""
        mobile.textOut("Get what?\n")
      else
        target = cmdline.split(' ').first
        source = nil
        if target == "!`" && spec = Util.parseSpec(target[0])
          target = mobile.node.findItem(spec, mobile)
        else
          if target == "!`" && spec = Util.parseSpec(target[1])
            source = mobile.node.findItem(spec, mobile)
          else
            spec = Util.parseSpec(target[2])
            source = source.is_a?(Container)
            target = source.findItem(spec, mobile)
            else
              mobile.textOut("That's not a container.\n")
            end
          end
        else
          mobile.textOut("You can't find it.\n")
        end
      end
    end
  end
end

CommandParser.register("get", Command::Get::Instance)

module Command
  class Go < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::STANDING
      @infight = false
    end

    def run(mobile, cmdline)
      Logger.log(mobile.to_s + " : Go", LogLevel::INFO)
      if cmdline == ""
        mobile.textOut("Go where?\n")
      else
        passage = mobile.node.findExit(cmdline)
        if passage == nil
          mobile.textOut("Alas, you cannot go that way.\n")
        else
          if passage.door && !passage.door.open
            mobile.textOut("It's closed.\n")
          else
            mobile.textOut("You can't enter that.\n")
          end
        end
      end
    end
  end
end

CommandParser.register("go", Command::Go::Instance)

CommandParser.register("enter", Command::Go::Instance)
require 'command.rb'
module Command
  class Lie < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::SLEEPING
      @infight = false
    end
    def run(mobile, cmdline)
      em = EnvMessage.new(EnvMessage::NORMAL, EnvMessage::ACTION, mobile)
      if mobile.position == Mobile::Pos::LAYING
        mobile.textOut('You lie down.\n')
        em.audioIntensity = EnvMessage::Audio::WALKING
        em.audio = 'rustling'
        em.video = 'lies down'
      elsif mobile.position == Mobile::Pos::SLEEPING
        mobile.textOut('You wake up.\n')
        mobile.textOut(mobile.node.look(mobile))
        em.audioIntensity = 0
        em.videoIntensity = EnvMessage::Video::SUBTLE
        em.video = 'wakes up'
      else
        mobile.textOut('You’re already lying down.\n')
      end
      mobile.position = Mobile::Pos::LAYING
      mobile.node.doMessage(em)
    end
  end
  @__instance = Lie.new
CommandParser.register('lie', Command::Lie::Instance)
end

require 'command.rb'
module Command
  class Look < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::RESTING
      @infight = true
    end
    def run(mobile, cmdline)
      Logger.log('#{mobile.name}: Look', LogLevel::INFO)
      if cmdline == ''
        mobile.textOut(mobile.node.look(mobile))
      else
        $OR FIND EXIT OR ...
        args = cmdline.split(' ')
        if args.size == 1 & spec = Util.parseFindSpec(args[0])
          Logger.log(spec)
          if spec.all || spec.count > 0
            mobile.textOut('You can’t find it.\n')
          else
            if target = mobile.node.findMob(spec, mobile)
              target = mobile.node.findItem(spec, mobile)
            else
              target = mobile.findItem(spec, mobile)
            end
            if target
              mobile.textOut(target.look(mobile))
            else
              mobile.textOut('You can’t find it.\n')
            end
          end
        else
          mobile.textOut('You can’t find it.\n')
        end
      end
      mobile.textOut('You can’t find it.\n')
    end
  end
  @__instance = Look.new
CommandParser.register('look', Command::Look::Instance)
end
require 'command.rb'

module Command
  class Open < Base
    def initialize
      @minlevel = 0
      @inflight = true
    end

    def run(mobile, cmdline)
      if (passage = mobile.node.findExit(cmdline)) && passage.visible
        if passage.door
          if passage.door.open
            mobile.textOut("It's already open.\n")
          else
            passage.door.open = true
            mobile.textOut("You open the #{passage.name}(Passage::SOU\n")
          end
        else
          mobile.textOut("You can't find it.\n")
        end
        @@_instance = Open.new
      end
    end
  end

  CommandParser.register(\"open\", Command::Open.Instance)
end

require 'special.rb'

module Command
  class Push < Special
    def initialize
      super(\"push\", false)
    end

    def run(mobile, cmdline)
      if (passage = mobile.node.findExit(cmdline)) && passage.visible
        if passage.door
          if passage.door.open
            mobile.textOut("It's already open.\n")
          else
            passage.door.open = true
            mobile.textOut("You open the #{passage.name}(Passage::SOU\n")
          end
        else
          mobile.textOut("You can't do that.\n")
      else
        mobile.textOut("You can't find it.\n")
      end
      @@_instance = Push.new
    end
  end

  CommandParser.register(\"push\", Command::Push.Instance)
end

Sunday May 18, 2003
require 'command.rb'
require 'util.rb'

module Command
  class Put < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::RESTING
      @infight = true
    end

    def run(mobile, cmdline)
      if cmdline == ''
        mobile.textOut("Put what where?\n")
        return
      end

      if cmdline.size == 1
        mobile.textOut("Put what where?\n")
        return
      end

      if cmdline.size == 2
        target = mobile.node.findItem(cmdline[0])
        source = mobile.node.findItem(cmdline[1])
      end

      if target.nil?
        mobile.textOut("That's not a container.\n")
        return
      end

      if target.instance
        if target.instance != source.instance
          mobile.put(target.instance, source.instance)
        end
      end

      mobile.position = Mobile::Pos::RESTING
      mobile.node.doMessage(mobile)
    end
  end
  #_instance = Put.new
  CommandParser.register("put", Command::Put.Instance)
end

module Command
  class Rest < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::SLEEPING
      @infight = false
    end

    def run(mobile, cmdline)
      em = EnvMessage.new(EnvMessage::NORMAL, EnvMessage::ACTION, mobile)
      em.audioIntensity = EnvMessage::Audio::WALKING
      em.videoIntensity = EnvMessage::Audio::WALKING
      em.audio = "rustling"
      if mobile.position > Mobile::Pos::SITTING
        mobile.textOut("You sit down and start resting.\n")
        em.video = "sits down and starts resting"
        mobile.put(mobile, em)
      elsif mobile.position == Mobile::Pos::SLEEPING
        mobile.textOut("You wake up and start resting.\n")
        mobile.put(mobile, em)
      else
        mobile.textOut("You're already resting.\n")
      end
    end
  end
  #_instance = Rest.new
  CommandParser.register("rest", Command::Rest.Instance)
require 'command.rb'

module Command
  class Say < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::STANDING
      @infight = true
    end

    def run(mobile, cmdline)
      if cmdline == ""
        mobile.textOut("Say what?\r\n")
      else
        mobile.textOut("You say, '%{cmdline}'\r\n")
      end
    end
  end
end

require 'command.rb'

module Command
  class Sit < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::SLEEPING
      @infight = false
    end

    def run(mobile, cmdline)
      em = EnvMessage.new(EnvMessage::NORMAL, EnvMessage::ACTION, mobile)
      em.audioIntensity = EnvMessage::Audio::WALKING
      em.videoIntensity = EnvMessage::Video::WALKING
      em.audio = "rustling"
      if mobile.position > Mobile::Pos::SITTING
        mobile.textOut("You sit down.\r\n")
        mobile.textOut(mobile.node.look(mobile))
        em.audio = "sits down"
      elsif mobile.position == Mobile::Pos::SLEEPING
        mobile.textOut("You wake up and sit up.\r\n")
        mobile.textOut(mobile.node.look(mobile))
        em.audio = "wakes up and sits up"
      else
        mobile.textOut("You're already sitting.\r\n")
        return
      end
      mobile.position = Mobile::Pos::SITTING
      mobile.node.doMessage(em)
    end
  end
end

CommandParser.register('say', Command::Say.Instance)

CommandParser.register('sit', Command::Sit.Instance)
require 'command.rb'

module Command
  class Sleep < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::LAYING
      @infight = false
    end

    def run(mobile, cmdline)
      em = EnvMessage.new(EnvMessage::NORMAL, EnvMessage::ACTION, mobile)
      if mobile.position > Mobile::Pos::LAYING
        mobile.textOut('You lie down and go to sleep.

      else
        mobile.position = Mobile::Pos::SLEEPING
        mobile.node.doMessage(em)
      end
    end

    def run(mobile, cmdline)
      em = EnvMessage.new(EnvMessage::NORMAL, EnvMessage::ACTION, mobile)
      if mobile.position == Mobile::Pos::STANDING
        mobile.textOut('You stand up.

      else
        mobile.position = Mobile::Pos::STANDING
        mobile.node.doMessage(em)
      end
    end

    CommandParser.register('sleep', Command::Sleep.Instance)
  end
end

require 'command.rb'

module Command
  class Stand < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::LAYING
      @infight = false
    end

    def run(mobile, cmdline)
      em = EnvMessage.new(EnvMessage::NORMAL, EnvMessage::ACTION, mobile)
      if mobile.position == Mobile::Pos::STANDING
        mobile.textOut('You stand up.

      else
        mobile.position = Mobile::Pos::STANDING
        mobile.node.doMessage(em)
      end
    end

    CommandParser.register('stand', Command::Stand.Instance)
require 'special.rb'

module Command
  class Switch < Special
    def initialize
      super('switch', false)
      @minlevel = 0
      @position = Mobile::Pos::STANDING
      @infight = false
    end
    @@_instance = Switch.new
  end
end

CommandParser.register('switch', Command::Switch::Instance)

module Command
  class Twist < Special
    def initialize
      super('twist', false)
      @minlevel = 0
      @position = Mobile::Pos::RESTING
      @infight = false
    end
    @@_instance = Twist.new
  end
end

CommandParser.register('twist', Command::Twist::Instance)
require 'command.rb'

module Command
  class Wake < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::DEAD
      @infight = false
    end

    def run(mobile, cmdline)
      if mobile.position != Mobile::Pos::STANDING
        em = EnvMessage.new(EnvMessage::NORMAL, EnvMessage::ACTION, mobile)
        em.audioIntensity = EnvMessage::Audio::WALKING
        em.videoIntensity = EnvMessage::Video::WALKING
        em.audio = 'rustling'
        if mobile.position > Mobile::Pos::SLEEPING
          mobile.textOut('You stand up.
        em.video = 'stands up'
        else
          mobile.textOut('You wake up and stand up.
        mobile.video = 'wakes up and stands up'
      end
      mobile.position = Mobile::Pos::STANDING
      mobile.node.doMessage(em)
      mobile.textOut('You're already awake."
      end
      end
    end

    def initialize
      super('write', true)
      @minlevel = 0
      @position = Mobile::Pos::STANDING
      @infight = false
    end

    def doCommand(mobile, cmdline)
      if cmdline == ' '
        mobile.textOut('Write what on what?\n"
      else
        name, text = cmdline.split(' ', 2)
        if text == nil || text == ''
          mobile.textOut('Write what?\n"
        else
          spec = Util.parseFindSpec(name)
          if obj = mobile.node.findItem(spec, mobile) ||
            obj = mobile.findItem(spec, mobile)
            if obj.respond_to?('write_on')
              obj.write_on(mobile, text)
            else
              mobile.textOut('You can't write on that."
            end
          else
            mobile.textOut('Write on what?\n"
          end
        end
      end
    end
  end
end

CommandParser.register('wake', Command::Wake::Instance)
require 'command.rb'

module Command
  class Yell < Base
    def initialize
      @minlevel = 0
      @position = Mobile::Pos::STANDING
      @infight = true
    end

    def run(mobile, cmdline)
      if cmdline == ''
        mobile.textOut("Yell what?

      else
        mobile.textOut("You yell. '#{cmdline}'

      end

      @@_instance = Yell.new
    end
  end
end

module Command
  class Eval < Base
    def initialize
      @minlevel = 100
      @position = Mobile::Pos::DEAD
      @infight = true
    end

    def run(mobile, cmdline)
      begin
        mobile.textOut("#{cmdline}\n")
      rescue IndexError, SyntaxError, NameError, TypeError, ArgumentError => error
        mobile.textOut("#{error}\n")
      end
      end

    end
  end
end

@@_instance = Eval.new

CommandParser.register("eval", Command::Eval::Instance)
require 'command.rb'
module Command
  class Goto < Base
    def initialize
      @inlevel = 100
      @position = Mobile::Pos::DEAD
      @infight = true
    end

    def run(mob, cmdline)
      if node = Util.parseID('n', cmdline)
        mob.node.leave(mob)
        node.enter(mob)
        print(node)
        mob.textOut(mob.node.look(mob))
      else
        mob.textOut('Can't find specified node. \n')
      end
    end

    @@instance = Goto.new
  end

  CommandParser.register("goto", Command::Goto::Instance)
end

require 'ommand.rb'
require 'util.rb'
module Command
  class OLC < Base
    def initialize
      @inlevel = 100
      @position = Mobile::Pos::STANDING
      @infight = false
    end

    def run(mob, cmdline)
      if mob.is_a?(Player)
        return
      end

      if cmdline == ""
        mob.textOut('Do what? (new list view edit load write) \n')
      else
        subcmd, args = cmdline.split(' ', 2)
        case subcmd.downcase
        when 'new'
          olcNew(mob, args)
        when 'list'
          olcList(mob, args)
        when 'view'
          olcView(mob, args)
        when 'edit'
          olcEdit(mob, args)
        when 'load'
          olcLoad(mob, args)
        when 'delete'
          olcDelete(mob, args)
        when 'write'
          olcWrite(mob, args)
        else
          mob.textOut('Can't find specified node. \n')
        end
      end
    end

  private
  def olcNew(mob, args)
    if !args
      mob.textOut('Create what? (zone node mob item passage) \n')
    end
  end

  def olcNewZone(mob, args)
    subsubcmd, subargs = args.split(' ', 2)
    case subsubcmd.downcase
    when 'z'
      olcNewZone(mob, subargs)
    when 'zone'
      olcNewZone(mob, subargs)
    when 'n'
      olcNewNode(mob, subargs)
    when 'node'
      olcNewNode(mob, subargs)
    when 'm'
      olcNewMob(mob, subargs)
    when 'mob'
      olcNewMob(mob, subargs)
    when 'i'
      olcNewItem(mob, subargs)
    when 'item'
      olcNewItem(mob, subargs)
    when 'p'
      olcNewPass(mob, subargs)
    when 'passage'
      olcNewPass(mob, subargs)
    else
      mob.textOut('Can't find specified node. \n')
    end
  end
end
id, name = args.split(" ", 2)
if $zones[id]
  mob.textOut("Zone already exists!\n")
elsif id & name
  if id =~ /[0-9]/ && id.to_i > 0
    $zones[id] = Zone.new(id)
    $zones[id].name = name
    mob.textOut("Created zone (R$(args))(x.\n")
  else
    mob.textOut("Invalid zone id.\n")
  end
else
  mob.textOut("Create zone with what id and name?\n")
end
end
def olcNewNode(mob, args)
id, clas = args.split(· , 2)
if Util.parse!D( 'n', id)
  mob.textOut('Node already exists! 
")
else
  if id =~ /[0-9]/ && id.to_c
    node = clas.to_c.new(52)
  else
    node = Item.new(52)
  end
  node.parent = id
  $zones[$id].addItem(node, $2)
  mob.textOut("Added Item (R$)(x).\n")
end
else
  mob.textOut("Invalid node id.\n")
end
end
def olcNewMob(mob, args)
if Util.parse!D('m', args)
  mob.textOut("Mobile already exists!\n")
else
  id = [0-9].[0-9]'
  if $zones[$id]
    mobile = Mobile.new($2)
    mobile.parent = $1
    $zones[$id].addMoblPalable, $2)
    mob.textOut("Added mobile (RI(args)(x. 
")
  else
    mob.textOut("Zone (R$)(x) does not exist.\n")
  end
else
  mob.textOut("Invalid mobile id.\n")
end
end
def olcNewItem(mob, args)
id, clas = args.split(" ", 2)
if Util.parse!D('j', id)
  mob.textOut("Item already exists!\n")
else
  if id =~ /[0-9]/ && clas && clas.to_c
    item = clas.to_c.new($2)
    item.parent = $1
    $zones[$id].addItem(item, $2)
    mob.textOut("Added item (R$(id))(x.\n")
  else
    mob.textOut("Invalid mobile id.\n")
end
end
def olcNewPass(mob, args)
  key, node, clas = args.split(" ", 3)
  if key && node && key =~ /[A-Za-z]/
    if mob.node.findExit(key)
      mob.textOut("Exit already exists.\n")
    else
      key.downcase!
      node = Node.new(node, key)
      pass = Passage.new(mob.node, node, key)
      mob.node.addExit(pass)
      mob.textOut("Added exit (RI(key))(x) to (R$(node))(x.\n")
    else
      mob.textOut("Invalid exit id.\n")
    end
  end
end
def olcList(mob, args)
if !args
  mob.textOut("List what? (zone node mob item)\n")
  return
else
  subsubcmd, subargs = args.split(" ", 2)
  if subargs && $zones[subargs]
    mob.textOut("Zone not found.\n")
    return
  elsif !subargs
    zone = mob.zone
  else
    zone = subargs
  end
  cage subsubcmd.downcase
  when "zone" then olcListZone(mob)
  when "zone" then olcListZone(mob)
  when "node" then olcListNode(mob, zone)
  when "mob" then olcListMob(mob, zone)
  when "item" then olcListItem(mob, zone)
  else
    mob.textOut("List what? (node mob item)\n")
  end
end
def olcListZone(mob)
str = "Zones:\n" keys = $zones.keys.sort { |a,b| a.rjust(5) <=> b.rjust(5) }
keys.each { |key|
  zone = $zones[key]
  str << "(R$(zone.id.rjust(5))(x: (#(zone.name))(x.\n")
  mob.textOut(str)
end
def olcListNode(mob, zone)
str = "Zone (R$(zone)(x nodes:\n"
keys = $zones[zone].nodes.keys.sort { |a,b| a.rjust(5) <=> b.rjust(5) }
keys.each { |key|
  n = $zones[zone].nodes[key]
  str << "[#{(n.id).rjust(5)}]:#{(n.name)}
  end
  mob.textOut(str)
end

def olcListMob(mob, zone)
  str = "Zone [#{zone}]: mobs:
  keys = $zones[zone].mobs.sort { |a,b| a.rjust(5) <=> b.rjust(5) }
keys.each { |key|
  m = $zones[zone].mobs[key]
  str << "[#{(m.id).rjust(5)}]:#{(m.name)}
  end
  mob.textOut(str)
end

def olcListItem(mob, zone)
  str = "Zone [#{zone}]: items:
  keys = $zones[zone].items.keys.sort { |a,b| a.rjust(5) <=> b.rjust(5) }
keys.each { |key|
  i = $zones[zone].items[key]
  str << "[#{(i.id).rjust(5)}]:#{(i.sdesc)}
  end
  mob.textOut(str)
end

def olcView(mob, args)
  if !args
    mob.textOut("View what? (zone node mob item)\n"
    return
  end
  subsubcmd, subargs = args.split(" ", 2)
  case subsubcmd.downcase
  when "z" then olcViewZone(mob, subargs)
  when "lone" then olcViewZone(mob, subargs)
  when "n" then olcViewNode(mob, subargs)
  when "node" then olcViewNode(mob, subargs)
  when "p" then olcViewPassage(mob, subargs)
  when "passage" then olcViewPassage(mob, subargs)
  when "m" then olcViewMob(mob, subargs)
  when "mob" then olcViewMob(mob, subargs)
  when "i" then olcViewItem(mob, subargs)
  when "item" then olcViewItem(mob, subargs)
  else
    mob.textOut("View what? (zone node mob item)\n"
    end
end

def olcViewZone(mob, subargs)
  if args && $zones[zone]
    zone = $zones[zone]
  else
    zone = $zones[mob.zone]
  end
  mob.textOut("Zone #{(zone.to_s(true))}\n"
end

def olcViewNode(mob, subargs)
  node = nil
  if args
    node = Util.parseIDC'n', args
  end
  else
    node = mob.node
  end
  if node
    mob.textOut("Node #{(node.to_s(true))}\n"
end

def olcViewPassage(mob, args)
  if !args
    mob.textOut("View which passage?\n"
    end
  if pass = mob.node.findExit(Args)
    mob.textOut("#{(pass.to_s(true))}\n"
  else
    mob.textOut("Passage not found.\n"
end

def olcViewMob(mob, args)
  if args && mobile = Util.parseIDC'm', args
    mobile = Util.parseIDC'm', args
  else
    spec = Util.parseItemSpec(args)
    if !spec || spec.all || spec.count > 0
      spec = mob.node.findMob(spec)
      mob.textOut("#{(mobile.to_s(true))}\n"
    else
      mob.textOut("The specified mob was not found.\n"
    end
  else
    mob.textOut("The specified item was not found.\n"
end

def olcViewItem(mob, args)
  if args && item = Util.parseIDC'i', args
    item = Util.parseIDC'i', args
  else
    spec = Util.parseItemSpec(args)
    if !spec || spec.all || spec.count > 0
      spec = mob.node.findItem(spec)
      item = spec
      mob.textOut("#{(item.to_s(true))}\n"
    else
      mob.textOut("The specified item was not found.\n"
    end
  else
    mob.textOut("The specified mob was not found.\n"
end

def olcEdit(mob, args)
  if !args
    mob.textOut("Edit what? (zone node mob item)\n"
    return
  end
  subsubcmd, subargs = args.split(" ", 2)
  if subargs
    mob.textOut("Edit which object?\n"
    return
  end
  case subsubcmd.downcase
  when "z" then olcEditZone(mob, subargs)
  when "lone" then olcEditZone(mob, subargs)
  when "n" then olcEditNode(mob, subargs)
  when "node" then olcEditNode(mob, subargs)
  when "p" then olcEditPassage(mob, subargs)
  when "passage" then olcEditPassage(mob, subargs)
  when "m" then olcEditMob(mob, subargs)
  when "mob" then olcEditMob(mob, subargs)
```ruby
when 'l' then olcEditItem(mob, subargs)
when 'item' then olcEditItem(mob, subargs)
else
  mob.textOut("Create what\r\n")
end

def olcEditItem(mob, args)
  subcmd, subargs = args.split(" ", 2)
  if subargs
    subargs = ''
  end
  case subcmd
  when "name"
    zones[mob.zone].name = subargs
    mob.textOut("Set name to \#(subargs)\r\n")
  else
    mob.textOut("Edit which property?\r\n")
  end
end

def olcEditZone(mob, args)
  subcmd, subargs = args.split(" ", 2)
  if subargs
    subargs = ''
  end
  case subcmd
  when "name"
    zones[mob.zone].name = subargs
    mob.textOut("Set name to \#(subargs)\r\n")
  else
    mob.textOut("Edit which property?\r\n")
  end
end

def olcEditNode(mob, args)
  node = mob.node
  subcmd, subargs = args.split(" ", 2)
  if subargs
    subargs = ''
  else
    subargs.chomp!
  end
  case subcmd
  when "name"
    node.name = subargs
    mob.textOut("Set \#(node.parent).\#(node.id)\r\n")
  when "sdesc"
    node.sdesc = subargs
    mob.textOut("Set \#(node.parent).\#(node.id)\r\n")
  when "ldesc"
    Editor.new(mob) { |text|
      if text
        node.ldesc = text
        mob.textOut("Set \#(node.parent).\#(node.id)\r\n")
      else
        mob.textOut("No changes.\r\n")
      end
    } when "outside"
  if subargs.downcase == /^\[true\]$/
    node.outside = true
    mob.textOut("Set \#(node.parent).\#(node.id)\r\n")
  elif subargs.downcase == /^\[false\]$/$
    node.outside = false
    mob.textOut("Set \#(node.parent).\#(node.id)\r\n")
  end
  when "ambient"
    if subargs.to_i >= 0
      node.ambient = subargs.to_i
      mob.textOut("Set \#(node.parent).\#(node.id)\r\n")
    else
      mob.textOut("Set ambient light level to \#(node.ambient)\r\n")
    end
  when "audio"
    if subargs.to_i >= 0
      node.audio = subargs.to_i
      mob.textOut("Set \#(node.parent).\#(node.id)\r\n")
    else
      mob.textOut("Set audio to \#(node.audio)\r\n")
    end
  end
end

def olcEditPassage(mob, args)
  key, subcmd, subargs = args.split(" ", 3)
  key.downcase!
  if pass = mob.node.findExit(key)
    if subargs
      subargs = ''
    end
    case subcmd.downcase
    when "name"
      pass.name = subargs
      mob.textOut("Set \#(node.parent).\#(node.id)\r\n")
    when "outside"
      pass.visible = false
      mob.textOut("Set passage \#(pass.visible)\r\n")
    when "ambient"
      if subargs.to_i >= 0
        pass.ambient = subargs.to_i
        mob.textOut("Set passage \#(pass.ambient)\r\n")
      else
        mob.textOut("Set ambient level to \#(node.ambient)\r\n")
      end
    when "audio"
      if subargs.to_i >= 0
        pass.audio = subargs.to_i
        mob.textOut("Set passage \#(pass.audio)\r\n")
      else
        mob.textOut("Set audio to \#(node.audio)\r\n")
      end
  end
end
```

def olcEditMob(mob, args)
    id, subcmd, subargs = args.split(' ')
    target = nil
    spec = Util.parseFindSpec(id.downcase)
    if !spec.all && spec.count > 0
        target = mob.node.findMob(spec)
    end
    if !target
        target = Util.parseID('m', id)
    end
    if target && subcmd
        if !subargs
            subargs = ""
        else
            subargs.chomp!
        end
        case subcmd
        when "name"
            target.name = subargs
            mob.textOut("Set name to '{subargs}'.")
            newkeys = subargs.downcase.split(' ')
            newkeys.delete_if{|k| ![CharUtil.silents, 'a', 'an', 'the', 'of'] == k}
            newkeys = ['nil'] if newkeys.size < 1
            target.keywords = newkeys
            mob.textOut("Set keywords to '{newkeys.join(' ')}'").
        when "rdesc"
            target.rdesc = subargs
            mob.textOut("Set current rdesc to '{subargs}'").
        when "tdesc"
            Editor.new(mob)
            if text
                target.tdesc = text
                mob.textOut("Set tdesc to '{text}'").
            else
                mob.textOut("No changes.").
            end
        when "level"
            level = subargs.to_i
            if level > 0 && level <= 100
                target.level = level
                mob.textOut("Set level to '{level}'").
            else
                mob.textOut("Invalid level '{subargs}'").
            end
        else
            mob.textOut("Edit which property?").
        end
    else
        mob.textOut("Passage not found.").
    end
end
else
  mob.textOut("Edit which mob?\n")
end
end

def olcEditItem(mob, args)
  id, subcmd, subargs = args.split(" ")
  if !target
    target = Util.parseID('id', args)
  end
  if target && subcmd
    subargs = subargs.split(' ")
    subargs.chomp!
  end
  case subcmd.downcase
    when 'm' then olcLoadMob(mob, subargs)
    when 'mob' then olcLoadMob(mob, subargs)
    when 't' then olcLoadItem(mob, subargs)
    when 'item' then olcLoadItem(mob, subargs)
    else
      mob.textOut("Load what? (mob item)\n")
  end
end

def olcDelete(mob, args)
  if !args
    mob.textOut("Delete what? (zone node mob item passage)\n")
    return
  end
  subcmd, subargs = args.split(" ", 2)
  if !subargs
    mob.textOut("Load which object?\n")
    return
  end
  case subcmd.downcase
    when "zone" then olcDeleteZone(mob, subargs)
    when "node" then olcDeleteNode(mob, subargs)
    when "item" then olcDeleteItem(mob, subargs)
    when "p" then olcDeletePassage(mob, subargs)
    else
      mob.textOut("The specified item was not found.\n")
      return
    end
end

def olcDeleteZone(mob, args)
end

def olcDeleteNode(mob, args)
end

def olcDeleteItem(mob, args)
end

def olcDeletePassage(mob, args)
end

def olcLoadMob(mob, args)
  if target = Util.parseID('id', args)
    mob.node.enter(m = target.clone)
    mob.onload
    mob.textOut("Loaded (R#(target.name)(x)\n")
    mob.textOut("The specified mobile was not found.\n")
    return
  end
  mob.textOut("The specified mobile was not found.\n")
end

def olcLoadItem(mob, args)
  if target = Util.parseID('id', args)
    target = Util.parseID('id', args)
    target = target.clone
    mob.item = target
    mob.node.enter(m = target.clone)
    mob.onload
    mob.textOut("Loaded (R#(target.name)(x)\n")
    mob.textOut("The specified item was not found.\n")
    return
  end
  mob.textOut("The specified mobile was not found.\n")
end

def olcDeleteMob(mob, args)
end

def olcDeletePassage(mob, args)
end

def olcDeleteItem(mob, args)
end

def olcDeleteNode(mob, args)
end

def olcDeleteMob(mob, args)
end

Sunday May 18, 2003

Printed by Kevin Radoff

/base/cmds/wizard/olc.rb

Page 11/12
if args && target = Util.parseID('m', args)
  $zones[target.parent].mobs.delete(target.id)
target.onRemove
  mob.textOut('Mobile [R#(args)](x deleted).\n')
else
  mob.textOut('Mobile not found.\n')
end
end

def olcDeleteItem(mob, args)
  if args && target = Util.parseID('i', args)
    $zones[target.parent].items.delete(target.id)
target.onRemove
  mob.textOut('Item [R#(args)](x deleted).\n')
else
  mob.textOut('Item not found.\n')
end
end

def olcDeletePassage(mob, args)
  if args && mob.node.findExit(args)
    mob.node.removeExit(args)
  mob.textOut('Passage at [R#(args)](x) deleted.\n')
else
  mob.textOut('Passage not found.\n')
end
end

def olcWrite(mob, args)
  if args
    filename = "#{config['world']}/#{args}zone.data"
    Layout::Writer.save(filename, $zones[args])
    mob.textOut('Successfully wrote zone [R#(args)](x to "#{filename}".\n')
  else
    mob.textOut('Zone [R#(args)](x) not found.\n')
  end
end

require 'command.rb'

module Command
  class Reload < Base
    def initialize
      @minlevel = 100
      @position = Mobile::Pos::DEAD
      @infight = false
    end

    def run(mob, cmdline)
      if cmdline == '•'
        mob.textOut('Reload what?\n')
      else
        begin
          load cmdline
        rescue LoadError, NameError, SyntaxError => error
          mob.textOut('Error loading "#{cmdline}".\n')
        end
      end
    end

    @@_instance = Reload.new
  end

  CommandParser.register('reload', Command::Reload::Instance)
end
require "inputhandler.rb"

class CommandParser < InputHandler
  @attr :commands
  private :initialize
  def initialize
    @commands = Hash.new
  end

  def register(name, command)
    # add some sort of error handling?
    @commands[name] = command
  end

  def CommandParser.register(name, command)
    @instance.register(name,
                        command)
  end

  def process(mob, text)
    return if text == ''
    cmd, args = text.split('""', 2)
    cmd.downcase!
    if args == nil
      args = ''
    end
    Logger.log('cmd: "" + cmd + '""', LogLevel::INFO)
    seq.queue(0) { 
      if mob.alias[cmd]
        cmd, newargs = mob.alias[cmd].split('""', 2)
        if !newargs
          newargs = ''
        end
        newargs.gsub!(/#/ , args)
        args = newargs
      end
      if @commands[cmd] && mob.level >= @commands[cmd].minlevel
        if mob.position >= @commands[cmd].position
          @commands[cmd].run(mob, args)
        else
          case mob.position
          when Mobile::Pos::DEAD
            mob.textOut("In your dreams, or what?\n")
          when Mobile::Pos::SLEEPING
            mob.textOut("You'd better stand up first.\n")
          when Mobile::Pos::LAYING
            mob.textOut("You'd better stand up first.\n")
          when Mobile::Pos::RESTING
            mob.textOut("You'd better stand up first.\n")
          end
        end
      else
        mob.textOut('Command not found.\n')
      end
    }
  end

  def findItem(spec, mob = nil)
    ret = nil
    if spec.all || spec.count > 0
      ret = Array.new
    end
    each_item { |i|
      if i.match?(spec.name)
        ret << i
      end
    }
  end

  def remove(object, loc = "in")
    if @objects[loc]
      return false
    else
      # um, some kind of check? :)
      @objects[loc].unshift(object)
      return true
    end
  end

  def insert(object, loc = "in")
    if @objects[loc]
      return false
    else
      # attempt to insert an object. Returns false if this isn't allowed.
      @objects[loc] = object
    end
  end

deal each_item
  @stowLocations, each { |loc|
    @objects[loc] = Array.new

  }
def setLocation(loc)
    @stowLocations.push(loc)
    @objects[loc] = Array.new
  end

  def addLocation(loc)
    @stowLocations.push(loc)
    @objects[loc] = Array.new
  end

  def removeLocation(loc)
    @stowLocations.delete(loc)
    @objects[loc] = nil
  end

  def register(name, command)
    # add some sort of error handling?
    @commands[name] = command
  end

  def CommandParser.register(name, command)
    @instance.register(name,
                        command)
  end

end

module Container
  attr :stowLocations
  attr :objects
  attr :capacity
  def initContainer(stowLocations = ["in"], capacity = 0)
    @stowLocations = stowLocations
    @objects = Hash.new
    @stowLocations.each { |loc|
      @objects[loc] = Array.new
    }
    @capacity = capacity
  end

  def setLocation(loc)
    @stowLocations.push(loc)
    @objects[loc] = Array.new
  end

  def addLocation(loc)
    @stowLocations.push(loc)
    @objects[loc] = Array.new
  end

  def removeLocation(loc)
    @stowLocations.delete(loc)
    @objects[loc] = nil
  end

  # define new to be the same as loc
def setLocation(loc, new)
    if @objects[loc]
      @objects[new] = @objects[loc]
    else
      # return an error code, or something?
    end
  end

  # Attempt to insert an object. Returns false if this isn't allowed.
def insert(object, loc = "in")
    if @objects[loc]
      return false
    else
      # um, some kind of check? :)
      @objects[loc].unshift(object)
      return true
    end
  end

  def remove(object, loc = "in")
    if @objects[loc]
      return false
    else
      # attempt to remove an object. Returns false if this isn't allowed.
      @objects[loc].delete(object)
      return true
    end
  end

  def findItem(spec, mob = nil)
    ret = nil
    if spec.all || spec.count > 0
      ret = Array.new
    end
    each_item { |i|
      if i.match?(spec.name)
        ret << i
      end
    }
  end

end
if !mob || mob.seeLight?(i.light) > 0 ||
  mob.see?(mob.node.light)
if spec.all
  ret.push(i)
elsif spec.offset > 1
  spec.offset = spec.offset - 1
elsif spec.count > 0
  spec.count = spec.count - 1
  ret.push(i)
else
  return i
end
}

require "inputhandler.rb"
class Creator < InputHandler
  PASSWD = 0
  CONF_PASSWD = 1
  def initialize(client)
    @state = PASSWD
    @passwd = nil
    @client = client
    @client.put("Hello #{@client.player.name}! Welcome to the character creation process.
    Password: ")
    @client.putAC(Telnet::WILL, Telnet::TELOPT_ECHO)
  end
  def process(player, text)
    case @state
    when PASSWD
      if text =~ /^a-zA-Z0-9$/
        @client.put("Passwords must be alphanumeric. Sorry.
        Password: ")
      else
        @passwd = text.clone
        @client.put("Confirm password: ")
        @state = CONF_PASSWD
      end
    when CONF_PASSWD
      Logger.log("#{@text}" (#{@passwd}))
      if text == @passwd
        @client.putAC(Telnet::WONT, Telnet::TELOPT_ECHO)
        Player.updatePasswd(@client.player, text)
        Player.write2(@client.player)
        @client.finishCreation
      else
        @client.put("Password mismatch. Try again.
        Password: ")
      end
      @state = PASSWD
    end
  end
end

creator.rb

Page 1/1

Page 2/2

container.rb

Page 2/2
require "inputhandler.rb"

class Editor < InputHandler
  def initialize(player, &finish)
    @buffer = Array.new
    @player = player
    @finish = finish
    @inputHandler = self
    @dead = true
    @client.noPrompt = true
    @client.put("Editor commands: /save to save, /cancel to cancel\n"
    )
    @player.client.put(‘\’ ‘)
  end

  def process(mob, text)
    case text
    when "\view"
      @player.client.put(@buffer.join(‘\r\n’))
      @player.client.put(" \"
    when "\save"
      self.complete(true)
    when "\cancel"
      self.complete(false)
    else
      @buffer.push(text.chomp)
      @player.client.put("\’ ‘")
    end
  end

  def complete(save)
    @dead = false
    @client.noPrompt = false
    @player.inputHandler = CommandParser::Instance
    if (save)
      @finish.call(@buffer.join("\r\n"))
    else
      @finish.call(nil)
    end
  end

end

class Entity
  attr :id
  attr :parent, true
  attr :nosave, true

  def initialize(id)
    @id = id
    @parent = ""
    @nosave = false
  end

  def look(mobile)
    ""
  end
end
module Environment
  attr :density, true
  attr :video, true
  attr :audio, true
  attr :distance, true
  def initEnvironment(density, video, audio)
    @density = density
    @video = true
    @audio = true
    @distance = 0
  end
  
  def doMessage(message)
    @print("Starting message: \\
      @message.audio\n      @message.video\n      \n")
    passMessage(message, message.audioIntensity, message.videoIntensity, nil)
    message.visited.each_pair { |env, visit| 
      @int. vint, sourceEnv = intensity
      env.each.mob { |m| 
        if m != message.source && m.deaf && m.position > Mobile::Pos::SLEEPING
          #if v = message.formatVideo(m, vInt, sourceEnv)
          #  m.textOut(v)
          #elsif a = message.formatAudio(m, aInt, sourceEnv)
          #  m.textOut(a)
          #end
          if text = message.format(m, visit)
            m.textOut(text)
          end
        
      }
    } 
    
    # pass visual/audio message
    def passMessage(message, aInt, vInt, sourceEnv)
      @print("Visit: #{@self.name}\n")
      message.visit(@self, aInt, vInt, sourceEnv)
      newAudio = degradeAudio(aInt)
      newVideo = degradeVideo(vInt)
      if newAudio <= 0 && newVideo <= 0
        return
      end
      each.exit { |e|
        if !message.void?(e)
          @print("Go to: #{@e.name}\n")
          if e.hasLost?(sourceEnv)
            @print("Kill Loser\n")
            newVideo = 0
          else
            newVideo = degradeVideo(vInt)
          end
        
        if message.visited?(e)
          @print("Try to revisit: #{@e.name} ... \")
          if message.canRevisit?(e, newAudio, newVideo)
            @print("Yup\n")
            e.passMessage(message, newAudio, newVideo, self)
          else
            @print("Denied\n")
          end
        
        else
          e.passMessage(message, newAudio, newVideo, self)
        end
      }
    end
  end
end

# degrade audio
def degradeAudio(audioIntensity)
  if @distance <= 1
    d = 100
  else
    d = 100 - @distance * 3
  end
  return audioIntensity * @audio * d / 10000
end

# degrade video
def degradeVideo(videoIntensity)
  if @distance <= 1
    d = 100
  else
    d = 100 - @distance
  end
  return videoIntensity * @video * d / 10000
end
class EnvVisit
MAXVISITS = 5
attr :topAudio
attr :topVideo
attr :count

def initialize
  @topAudio = nil
  @topVideo = nil
  @count = 0
end

def visit(aInt, vInt, source)
  @count += 1
  if @topAudio
    if aInt > @topAudio[0]
      @topAudio = [aInt, vInt, source]
    else
      @topAudio = [aInt, vInt, source]
    end
  end
  if @topVideo
    if vInt > @topVideo[1]
      @topVideo = [aInt, vInt, source]
    else
      @topVideo = [aInt, vInt, source]
    end
  end
end

def canRevisit?(aInt, vInt)
  if @count < MAXVISITS
    if @topAudio
      if aInt > @topAudio[0]
        return true
      end
    end
    if @topVideo
      if vInt > @topVideo[1]
        return true
      end
    end
    return false
  else
    return true
  end
end

class EnvMessage
# Sensitivity to perceive this...
LOW = 0
NORMAL = 1
HIGH = 2

# Type of message
TALKING = 0
MOVEMENT = 1
ACTION = 2

module Audio
WHISPER = 25
WALKING = 75
end

module Video
SUBTLE = 50
STANDING = 75
WALKING = 100
RUNNING = 125
COMBAT = 150
end

attr :visited
attr :avoid
attr :audio, true
attr :video, true
attr :audioIntensity, true
attr :videoIntensity, true
attr :level
attr :origin
attr :source

def initialize(level, type, source, audioIntensity, videoIntensity = 0, audio = '**', video = '**)
  @visited = Hash.new
  @avoid = Array.new
  @distance = 0
  @level = level
  @type = type
  @origin = source
  @source = source
  @audioIntensity = audioIntensity
  @videoIntensity = videoIntensity
  @audio = audio
  @video = video
end

def visit(env, aInt, vInt, sourceEnv)
  if !@visited[env]
    @visited[env] = EnvVisit.new
  end
  @visited[env].visit(aInt, vInt, sourceEnv)
end

def visited?(env)
  @visited.member?(env)
end

def canRevisit?(env, aInt, vInt)
  if @visited[(env)
    return @visited[(env).canRevisit?(aInt, vInt)
  else
    return true
  end
end

def avoid(env)
  @avoid.push(env)
end

def avoid?(env)
  @avoid.include?(env)
end

def format(target, visit)
  sourceEnv = nil
  if visit.topVideo
    if canSee = ((target.see?(visit.topVideo[1]) > 0) || target.seeLight?(@source.light)) > 0)
      target.seeSurroundings?(origin, light) ||
      target.seeLight?(@source.light) > 0))
    end
  end
end
sourceEnv = visit.topVideo[2]
end
else
  canSee = false
end
if visit.topAudio
  if canHear = target.see?(visit.topAudio[0]) > 0)
    if !sourceEnv
      sourceEnv = visit.topAudio[2]
    end
  else
    canHear = false
end
if canSee & canHear
  if sourceEnv
    case sourceEnv.type
      when Passage::Type::DIR
        where = "In"
      else
        where = "Through"
    end
    case type
      when TALKING
        if canSee & canHear
          "#{source.name} name[#{audio}].
        else
          if canSee
            "#{source.name[#{video}].
          end
        else
          "#{source.name} name[#{audio}].
        end
      when MOVEMENT
        if canSee
          "#{source.name[#{audio}].
        else
          "You hear #{audio}.
        end
      when ACTION
        if canSee
          "#{source.name[#{audio}].
        else
          "You hear #{audio}.
        end
      end
    end
  end
end
end
require "entity.rb"
require "property.rb"
require "onevent.rb"

class Item < Entity
  include Property
  include Onevent

  attr :sdesc, true
  attr :rdesc, true
  attr :ldesc, true
  attr :keywords, true
  attr :light, true
  attr :carry, true
  attr :zone, true

  def initialize(id)
    super(id)
    initProperty()
    @sdesc = ""
    @rdesc = ""
    @ldesc = ""
    @keywords = Array.new
    @light = 0
    @carry = true
    @zone = nil
    @node = nil
  end

  def match?(key)
    @keywords.include?(key)
  end

  def node=(n)
    @node = n
  end

  def node(node)
    if @zone && @node
      return $zones[@zone][@node]
    else
      return nil
    end
  end

  def look(mobile)
    return @ldesc + "\r\n"
  end

  def to_s(color = false)
    if color
      str = "Item (#{@parent}.#{id}(#{self.type})\r\nsdesc: (#{@sdesc})\r\nrdesc: (#{@rdesc})\r\nldesc: (#{@ldesc})\r\nkeywords: (#{@keywords.join(\" \")})\r\ncommands: (\" \")\r\nend
      mech = Array.new
      self.methods.each { |method|
        if method =~ /cmd_/m
          mech.push(method)
        end
      }
      str << mech.join(" ")
    else
      str = "Item (#{@parent}.#{id}(#{self.type})\r\nsdesc: (#{@sdesc})\r\nrdesc: (#{@rdesc})\r\nldesc: (#{@ldesc})\r\nkeywords: (#{@keywords.join(\" \")})\r\ncommands: (\" \")\r\nend
      mech = Array.new
      self.methods.each { |method|
        if method =~ /cmd_/m
          mech.push(method)
        end
      }
      str << mech.join(" ")
    end
  end
  def initialize(id)
    super(id)
    initProperty()
    @sdesc = ""
    @rdesc = ""
    @ldesc = ""
    @keywords = Array.new
    @light = 0
    @carry = true
    @zone = nil
    @node = nil
  end

  def process(text)
    .end
  end
end

require 'inputhandler.rb'
require 'item.rb'
str = "\r\n(prop_to_s(color).chomp)"
end

module Layout
  class Base
    attr :sid
    def initialize(zone, node)
      @sid = zone
      @zone = $zones[zone]
      @node = $zones[zone][node]
    end
    def load
      end
    end
  end
end
require 'node.rb'
require 'item.rb'

module Layout
  class Item < Base
    attr_accessor :item, :max
    
    def initialize(zone, node)
      @item = 0
      @max = 0
    end
    
    def load
      if @zone.items[@item]
        @node = @zone.items[@item].clone
        @zone = @node.parent
        @node = @node.id
        @item = @node.id
        @zone = @node.parent
      end
    end
  end
end

module Loader
  class Loader
    @current_zone = nil
    
    def load(filename)
      @current_zone = nil
      file.open(filename, 'r') do |file|
        file.each do |line|
          line.chomp
          if line.chomp == 'n'
            command << line
          else
            if @current_zone
              dispatch_load(command)
            else if command.first =~ /zone:/ && @current_zone = dispatch_load(command)
              to_i >= 0
            else
              Logger.log("Error in #\{filename\}: No Zone.", LogLevel::ERROR)
              return
            else
              command.clear
            end
          end
        end
      end
    end
    
    def dispatch_load(command)
      case command.first
      when /
        return load_zone(command)
      when /node:/
        return load_node(command)
      when /passage:
        return load_passage(command)
      when /item:
        return load_item(command)
      when /mob:
        return load_mob(command)
      end
    end
    
    def get_sub_cmd(command)
      subcmd = Array.new
      print("\{\#\{command.first\}\}\")
      while command.first =~ /%\$/
        subcmd << command.shift
        if command.first
          command.shift
        end
        print("\{\#\{subcmd.join\}\}\")
        return subcmd
      end
    end
    
    def load_properties(obj, props)
      props.each do |p|
        key, val = p.chomp.split('=', 2)
        obj.setProperty(key, val)
      end
    end
  end
end

Mar 13, 03 14:40 layout.item.rb
 Mar 17, 03 11:01 layout.loader.rb

Page 1/1
Page 1/4
def Loader.loadZone(command)
  zone, id, clas = command.shift.chomp.split(*', ', 3)
  #Logger.log("loadZone:#{id}, #{id.to_i}")
  $zones[id] = Zone.new(id)
  if !id || id.to_i >= 0
    NLogger.log('id}.f{id.to_i}
    $zones[id] = Zone.new(id)
  else
    return -1
  end
  command.shift
  $zones[id].name = getSubCmd(command).join.chomp
  return id
end

def Loader.loadNode(command)
  node, id, das = command.shift.chomp.split(*', ', 3)
  Logger.log('loadNode', id), (id.to_i);
  if id && id.to_i > 0
    node = das.to_c.new(id)
    else
      node = Node.new(id)
      node.parent = @$currZone
      $zones[@currZone].addNode(node, id)
  else
    return -1
  end
  command.shift
  subcmd = getSubCmd(command)
  node.name = subcmd.join.chomp
  subcmd = getSubCmd(command)
  node.desc = subcmd.join.chomp
  subcmd = getSubCmd(command)
  case subcmd.join.chomp
  when 'true'
    node.outside = true
  when 'false'
    node.outside = false
  end
  subcmd = getSubCmd(command)
  trans = subcmd.join.chomp
  subcmd = getSubCmd(command)
  distance, hangle, vangle = subcmd.join.chomp.split(*', ', 3).collect { |i| i.to_i }
  subcmd = getSubCmd(command)
  audio, video = subcmd.join.chomp.split(*':', 2).collect { |i| i.to_i }
  subcmd = getSubCmd(command)
  if doorstr = subcmd.join.chomp
    door = true
    doorloc = nil
    passages, sopen, daudio, dvideo = doorstr.split(*', ', 4)
    passages.split(*', ')
    if p =~ /
      pass = Passage.new($zones[@currZone][from], to, key, trans)
      $zones[@currZone][from].addExit(pass)
      pass.name = name
      pass.keywords = keywords
      pass.visible = visible
      pass.navigable = navigable
      pass.distance = distance
      pass.hangle = hangle
      pass.vangle = vangle
      if door = true
        if sopen == 'true'
          pass.door = true
        else
          pass.door = false
        end
      end
      pass.audio = audio
      pass.video = video
      if door = true
        pass.door = doorloc
      else
        pass.door = true
      if sopen == 'true'
  end
  end
  pass = Passage.new($zones[@currZone][from], to, key, trans)
  $zones[@currZone][from].addExit($zones[@currZone][to])
  if from || to || key
    $zones[@currZone][from].addExit($zones[@currZone][to])
    $zones[@currZone][from].addExit(pass)
    pass.name = name
    pass.keywords = keywords
    pass.visible = visible
    pass.navigable = navigable
    pass.distance = distance
    pass.hangle = hangle
    pass.vangle = vangle
    pass.audio = audio
    pass.video = video
    if door = true
      pass.door = doorloc
    else
      pass.door = true
  if sopen == 'true'
end

loadProperties(node, getSubCmd(command))
node.onLoad

Mar 17, 03 11:01 layout.loader.rb Page 2/4

Mar 17, 03 11:01 layout.loader.rb Page 3/4
```ruby
require 'node.rb'
require 'mobile.rb'

module Layout
  class Mobile < Base
    attr :mobile, true
    attr :max, true
    
    def initialize(zone, node)
      super(zone, node)
      @mobile = 0
      @max = 0
    end
    
    def load
      if @zone.mobs[@mobile]
        @node.enter = @zone.mobs[@mobile].clone
        m.onLoad
      end
    end
    
    def loadItem(command)
      item, id, clas = command.shift.chomp.split(":", 3)
      if id && id.to_i >= 0
        if clas && clas.to_c
          item = clas.to_c.new(id)
        else
          item = ::Item.new(id)
        end
        item.parent = @@currZone
        $zones[@@currZone].addItem(item, id)
      else
        return -1
      end
      command.shift
      subcmd = getSubCmd(command)
      item.rdesc = subcmd.join.chomp
      subcmd = getSubCmd(command)
      item.ldesc = subcmd.join.chomp
      subcmd = getSubCmd(command)
      item.keywords = subcmd.join.chomp.split(",")
      loadProperties(item, getSubCmd(command))
    end
    
    def loadMob(command)
      mob, id, clas = command.shift.chomp.split(":", 3)
      if id && id.to_i >= 0
        mob = ::Mobile.new(id)
        mob.parent = @@currZone
        $zones[@@currZone].addMob(mob, id)
      else
        return -1
      end
      command.shift
      subcmd = getSubCmd(command)
      mob.name = subcmd.join.chomp
      subcmd = getSubCmd(command)
      mob.data = subcmd.join.chomp
      subcmd = getSubCmd(command)
      mob.tdata = subcmd.join.chomp
      subcmd = getSubCmd(command)
      mob.tkeywords = subcmd.join.chomp
      loadProperties(mob, getSubCmd(command))
    end
  end
end
```
require "zone.rb"
require "node.rb"
require "item.rb"
require "passage.rb"

module Layout
  class Writer
    def save(filename, zone)
      File.open(filename, "w") do |file|
        saveZone(file, zone)
        zone.nodes.each_value { |node|
          if !node.nosave
            saveNode(file, node)
          end
        }
        zone.mobs.each_value { |mob|
          if !mob.nosave
            saveMob(file, mob)
          end
        }
        zone.items.each_value { |item|
          if !item.nosave
            saveItem(file, item)
          end
        }
      end
      Writer.saveSubCmd(file, "command")
    end

    def saveSubCmd(file, subcmd)
      file.puts("\n")
    end

    def saveZone(file, zone)
      saveSubCmd(file, "zone:
      each_value { |zone|
        saveNode(file, node)
      }
    end

    def saveNode(file, node)
      saveSubCmd(file, "node:
      each_value { |node|
        saveNode(file, node)
      }
    end

    def saveItem(file, item)
      if item.type != Item
        saveSubCmd(file, "item:
      else
        saveSubCmd(file, "item:"(item.id)
      end
    end

    def saveMob(file, mob)
      if mob.type == Mobile
        saveSubCmd(file, "mob:
      else
        saveSubCmd(file, "mob:"(mob.id)
      end
    end

    def savePassage(file, passage)
      saveSubCmd(file, "passage:
      each { |passage|
        if !passage.nosave
          savePassage(file, passage)
        end
      }
    end

    def saveProps(file, obj)
      text = ""
      each_property { |key, val|
        text << "#{key}:#{val.get_property_str(key)}\n"
      }
      saveSubCmd(file, text)
    end

    def saveDelimiter(file)
      saveSubCmd(file, "")
    end

    def saveSubCmd(file, str)
      file.puts(str)
    end
  end
end

def Writer.saveSubCmd(file, Subcmd)
  saveProps(file, node)
  saveDelimiter(file)
  node.each_exit { |passage|
    if !passage.nosave
      savePassage(file, passage)
    end
  }
end

def Writer.savePassage(file, passage)
  saveSubCmd(file, "passage:
  each { |passage|
    if !passage.nosave
      savePassage(file, passage)
    end
  }
end

saveSubCmd(file, "command")
saveProps(file, node)
saveDelimiter(file)
node.each_exit { |passage|
  if !passage.nosave
    savePassage(file, passage)
  end
}
end
require 'inputhandler.rb'

class LineEditor < InputHandler
  def initialize(player, message, acceptNull = true, &finish)
    @text = ''
    @player = player
    @message = message
    @acceptNull = acceptNull
    @finish = finish
    @player.inputHandler = self
    @player.deaf = true
    @player.client.noPrompt = true
    @player.client.put("##(message) ")
  end

  def process(mob, text)
    if text == ''
      if @acceptNull
        @text << text
        self.complete
      else
        @player.client.put("##(message) ")
      end
    else
      @text << text
      self.complete
    end
  end

  def complete
    @player.deaf = false
    @player.client.noPrompt = false
    @player.inputHandler = CommandParser.Instance
    @finish.call(@text)
  end
end
require 'entity.rb'
require 'periodic.rb'
require 'container.rb'
require 'property.rb'
require 'onevent.rb'

class Mobile < Entity
  include Periodic
  include Container
  include OnEvent
  include Property

  module Pos
    DEAD = 0
    SLEEPING = 1
    LAYING = 2
    RESTING = 3
    SITTING = 4
    STANDING = 5
  end

  attr :zone, true
  #attr :name, true
  #attr :rdes
  attr :keywords, true
  attr :ldecs, true
  attr :race, true
  attr :level, true
  attr :light, true
  attr :hearing, true
  attr :sight, true
  attr :lightSight, true
  attr :position, true
  attr :alias
  attr :active
  attr :dead, true
  #attr :busy
  #attr :busyCount
  attr :inputHandler, true

  def initialize(id)
    super(id)
    initPeriodic()
    initContainer("in", 1000)
    initProperty()
    @name = '(nil)'
    @rdesc = Array.new
    @ldecs = ''
    @keywords = []
    @zone = 0
    @node = 0
    @level = 0
    @light = 0
    @lightSight = 25
    @sight = 50
    @position = Pos::STANDING
    @alias = {
      'e' => 'go east',
      'east' => 'go east',
      'w' => 'go west',
      'west' => 'go west',
      'n' => 'go north',
      'north' => 'go north',
      's' => 'go south',
      'south' => 'go south',
      'u' => 'go up',
      'up' => 'go up',
    }
    @inputHandler = CommandParser::Instance
  end

  def rdesc
    text = ''
    case @position
    when Pos::SLEEPING then text = "#{@name} is sleeping here."
    when Pos::LAYING then text = "#{@name} is laying down here."
    when Pos::RESTING then text = "#{@name} is resting here."
    when Pos::SITTING then text = "#{@name} is sitting here."
    when Pos::STANDING then text = "#{@name} stands here."
    else
      text = @rdesc[@position]
    end
    text
  end

  def match?(name)
    @keywords.include?(name)
  end

  def process(text)
    @inputHandler.process(self, text)
  end

  def busy?
    return @busy
  end

  def busy(period)
    @busy = true
    @busyCount += 1
    @queue.put(period) { self.free }
  end

  def free
    @busyCount -= 1
    if @busyCount < 1
      @busy = false
    end
  end

  def activate
    @active = true
    self.heartbeat
  end

  def deactivate
    @active = false
    end
    def heartbeat
      end
    def textOut(text)
      nil
    end
  end
end

require 'entity.rb'
require 'periodic.rb'
require 'container.rb'
require 'property.rb'
require 'onevent.rb'
def node=n
    @node = n
end
end

def see?(intensity)
    if intensity < @sight
        return 1
    else
        return
    end
end

def seeLight?(intensity)
    if intensity < node.light / 4 || intensity > node.light + 500 || intensity < @lightSight
        return 1
    else
        return false
    end
end

def seeSurroundings?(intensity)
    if seeLight?(intensity) && intensity > @lightSight + 100
        true
    else
        false
    end
end

# This is for what you see if you look 'at' this mobile
def look(mob)
    text = "#{(@name)}\r\n#{(@id)}\r\n\n" + text
end

def to_s(color = false)
    text = "\n" if color
    text = "Mobile: (#(@parent))\r\n#{(@id)}\r\n#{(@sdesc)}\r\n#{(@name)} R\nRace: (#{(@race))\r\nLevel: (#{(@level))\r\nPosition: (#{(@position))\r\nPath: (#{(@rpath))\r\nKeywords: (#{(@keywords).join( ' ')}))\r\n#{(@ldesc)}\r\n\n" if @ldesc
    else
        text = "Mobile: (#(@parent))\r\n#{(@id)}\r\n#{(@sdesc)}\r\n#{(@name)} R\nRace: (#{(@race))\r\nLevel: (#{(@level))\r\nPosition: (#{(@position))\r\nPath: (#{(@rpath))\r\nKeywords: (#{(@keywords).join( ' ')}))\r\n#{(@ldesc)}\r\n\n"
end
def get(target, source = nil, loc = "in")
    str = "\n" if source
    if target.is_a?(Array)
        target.each ( |
            if i.carry
                node.remove(i)
                self.insert(i)
                i.zone = i.node = nil
                str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
            else
                if target.is_a?(Array)
                    target.each ( |
                        if i.carry
                            node.remove(i)
                            self.insert(i)
                            i.zone = i.node = nil
                            str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
                else
                    if target.carry
                        node.remove(target)
                        self.insert(target)
                        target.zone = target.node = nil
                        str << 'You get #{(target.sdesc)} from #{(source.sdesc)}\n" end
                    else
                        self.remove(target)
                        self.insert(target)
                        target.zone = target.node
                        str << 'You drop #{(target.sdesc)}\n"
                    end
                end
            end
        end
        textOut(str)
    end
else
    self.remove(target)
    self.insert(target)
    target.zone = target.node
    str << 'You drop #{(target.sdesc)}\n"
end
def put(target, source = nil, loc = "in")
    self remove(target)
    node.insert(target)
    target.zone = target.node
end

def drop(target)
    if target.is_a?(Array)
        str = "\n" target.each ( |
            if i.carry
                node.remove(i)
                self.insert(i)
                i.zone = i.node = @node
                i.node = @node
                str << 'You drop #{(i.sdesc)}\n"
end
        textOut(str)
else
    self.remove(target)
    node.insert(target)
    target.zone = target.node = @node
    textOut('You drop #{(target.sdesc)}\n"
end
end

def hear?(intensity)
    if intensity < @hearing
        return 1
    else
        return 0
    end
end

def @sdesc = @node = nil
str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
else
    if target.carry
        source.remove(target)
        self.insert(target)
        i.zone = i.node = nil
        str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
else
    if target.is_a?(Array)
        target.each ( |
            if i.carry
                node.remove(i)
                self.insert(i)
                i.zone = i.node = nil
                str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
                else
                    if target.carry
                        node.remove(target)
                        self.insert(target)
                        target.zone = target.node = nil
                        str << 'You get #{(target.sdesc)} from #{(source.sdesc)}\n" end
                    else
                        self.remove(target)
                        self.insert(target)
                        target.zone = target.node
                        str << 'You drop #{(target.sdesc)}\n"
                    end
                end
        end
        textOut(str)
    end
else
    self.remove(target)
    self.insert(target)
    target.zone = target.node
    str << 'You drop #{(target.sdesc)}\n"
end
end
def put(target, source = nil, loc = "in")
    self remove(target)
    node.insert(target)
    target.zone = target.node
end

def drop(target)
    if target.is_a?(Array)
        str = "\n" target.each ( |
            if i.carry
                node.remove(i)
                self.insert(i)
                i.zone = i.node = @node
                i.node = @node
                str << 'You drop #{(i.sdesc)}\n"
end
        textOut(str)
else
    self.remove(target)
    node.insert(target)
    target.zone = target.node = @node
    textOut('You drop #{(target.sdesc)}\n"
end
end

def hear?(intensity)
    if intensity < @hearing
        return 1
    else
        return 0
    end
end

def @sdesc = @node = nil
str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
else
    if target.carry
        source.remove(target)
        self.insert(target)
        i.zone = i.node = nil
        str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
else
    if target.is_a?(Array)
        target.each ( |
            if i.carry
                node.remove(i)
                self.insert(i)
                i.zone = i.node = nil
                str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
                else
                    if target.carry
                        node.remove(target)
                        self.insert(target)
                        target.zone = target.node = nil
                        str << 'You get #{(target.sdesc)} from #{(source.sdesc)}\n" end
                    else
                        self.remove(target)
                        self.insert(target)
                        target.zone = target.node
                        str << 'You drop #{(target.sdesc)}\n"
                    end
                end
        end
        textOut(str)
    end
else
    self.remove(target)
    self.insert(target)
    target.zone = target.node
    str << 'You drop #{(target.sdesc)}\n"
end
end

def hear?(intensity)
    if intensity < @hearing
        return 1
    else
        return 0
    end
end

def @sdesc = @node = nil
str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
else
    if target.carry
        source.remove(target)
        self.insert(target)
        i.zone = i.node = nil
        str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
else
    if target.is_a?(Array)
        target.each ( |
            if i.carry
                node.remove(i)
                self.insert(i)
                i.zone = i.node = nil
                str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
                else
                    if target.carry
                        node.remove(target)
                        self.insert(target)
                        target.zone = target.node = nil
                        str << 'You get #{(target.sdesc)} from #{(source.sdesc)}\n" end
                    else
                        self.remove(target)
                        self.insert(target)
                        target.zone = target.node
                        str << 'You drop #{(target.sdesc)}\n"
                    end
                end
        end
        textOut(str)
    end
else
    self.remove(target)
    self.insert(target)
    target.zone = target.node
    str << 'You drop #{(target.sdesc)}\n"
end
end

def hear?(intensity)
    if intensity < @hearing
        return 1
    else
        return 0
    end
end

def @sdesc = @node = nil
str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
else
    if target.carry
        source.remove(target)
        self.insert(target)
        i.zone = i.node = nil
        str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
else
    if target.is_a?(Array)
        target.each ( |
            if i.carry
                node.remove(i)
                self.insert(i)
                i.zone = i.node = nil
                str << 'You get #{(i.sdesc)} from #{(source.sdesc)}\n" end
                else
                    if target.carry
                        node.remove(target)
                        self.insert(target)
                        target.zone = target.node = nil
                        str << 'You get #{(target.sdesc)} from #{(source.sdesc)}\n" end
                    else
                        self.remove(target)
                        self.insert(target)
                        target.zone = target.node
                        str << 'You drop #{(target.sdesc)}\n"
                    end
                end
        end
        textOut(str)
    end
else
    self.remove(target)
    self.insert(target)
    target.zone = target.node
    str << 'You drop #{(target.sdesc)}\n"
end
end
require 'entity.rb'
require 'container.rb'
require 'environment.rb'
require 'property.rb'

class Node < Entity
  include Container
  include Environment
  include Property
  include OnEvent

  attr :name, true
  attr :desc, true
  attr :ldesc, true
  attr :outside, true
  attr :ambient, true
  attr :mobLayout
  attr :itemLayout
  #attr :exits
  attr :mobs

  def initialize(id)
    super(id)
    initContainer()
    initEnvironment(0, 100, 100)
    initProperty()
    @name = ''
    @desc = ''
    @ldesc = ''
    @outside = true
    @ambient = 0
    @mobLayout = Array.new
    @itemLayout = Array.new
    @exits = Hash.new
    @mobs = Array.new
  end

  def addExit(passage)
    @exits[passage.key] = passage
  end

  def removeExit(key)
    @exits[key].onRemove
    @exits.delete(key)
  end

  def reset
    @mobLayout.each { |m| m.load }
    @itemLayout.each { |l| l.load }
  end

  def enter(mob)
    mob.node = @id
    mob.zone = @parent
    @mobs.unshift(mob)
  end

  def leave(mob)
    mob.node = '0'
    mob.zone = '0'
    @mobs.delete(mob)
  end

  def hasLOS?(env)
    true
  end

  def light
    l = @ambient
    if @outside && WorldTime.now.isDay?
      l = Weather::Light::SUN
    end
    each_item { |i|
      l = i.light
    }
    each_mob { |m| l = m.light }
  end

  def each_exit
    @exits.each_value { |e|
      yield e
    }
  end

  def each_mob
    @mobs.each { |m|
      yield m
    }
  end

  def findExit(key)
    if @exits[key]
      # check that it's visible or something?
      return @exits[key]
    end
    nil
  end

  def findMob(spec, mob = nil)
    if spec.all || spec.count > 0
      return nil
    end
    each_mob { |m|
      if m.match?(spec.name)
        if !mob || m.mobLight?(m.light) > 0
          if spec.offset > 1
            spec.offset = spec.offset - 1
          end
          return m
        end
      end
    }
  end

  def findExits(key)
    @exits[key].onRemove
    @exits.delete(key)
  end

  def light
    l = @ambient
    if @outside && WorldTime.now.isDay?
      l = Weather::Light::SUN
    end
    each_item { |i|
      l = i.light
    }
    each_mob { |m| l = m.light }
  end

  def each_exit
    @exits.each_value { |e|
      yield e
    }
  end

  def each_mob
    @mobs.each { |m|
      yield m
    }
  end

  def findExit(key)
    if @exits[key]
      # check that it's visible or something?
      return @exits[key]
    end
    nil
  end

  def findMob(spec, mob = nil)
    if spec.all || spec.count > 0
      return nil
    end
    each_mob { |m|
      if m.match?(spec.name)
        if !mob || m.mobLight?(m.light) > 0
          if spec.offset > 1
            spec.offset = spec.offset - 1
          else
            return m
          end
        end
      end
    }
  end

  def look(mob)
    text = String.new(@name)
    if mob.level > 100 && mob.is_s?(@ldesc)
      text = '([^@parent].([^@id]).)+'
    end
    if mob.mobLight?(self.light) > 0 && mob.mobSeeSurroundings?(self.light)
      text = '([^@ldesc]).([^@idesc]).([^@ldesc]).([^@idesc])
    end
    else
      text = '([^@ldesc]).([^@idesc]).([^@ldesc]).([^@idesc]).([^@ldesc]).([^@idesc]).([^@ldesc]).([^@idesc]).([^@ldesc]).([^@idesc]).([^@ldesc]).([^@idesc]).([^@ldesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idesc]).([^@idec...
```ruby
module OneEvent
  def onLoad
  end
  def onRemove
  end
  def onPickup
  end
  def onDrop
  end
end
```
require 'entity.rb'
require 'environment.rb'
require 'transport.rb'
require 'property.rb'

class Passage
  include Environment
  include Transport
  include Property
  include OnEvent

  module Type
    KEY = 0
    DIR = 1
  end

  STR = ['KEY', 'DIR']

  Opposite = { 'east' => 'west', 'west' => 'east', 'north' => 'south', 'south' => 'north', 'up' => 'down', 'down' => 'up' }

  SOURCE = 0
  TARGET = 1
  REAL = 2

  attr :from
  attr :to
  attr :key
  attr :type
  attr :visible, true
  attr :navigable, true
  attr :keywords, true
  attr :transition, true
  attr :hangle, true
  attr :vangle, true
  attr :door

  class Door
    attr :name, true
    attr :passages
    attr :startOpen, true
    attr :audio, true
    attr :video, true

    def initialize(p)
      # @name = "door"
      # @passages = Array.new
      # @passages.push(p)
      @startOpen = false
      @open = true
      @audio = 0
      @video = 0
    end

    def initialize(from, to, key, transition = '')
      super("#{from.id}.#{to.id}.#{key}"")
      initEnvironment(0, 0, 0)
      initTransport(from, to)
      initProperty()
      if Opposite(key)
        @type = Type::DIR
      else
        @type = Type::KEY
      end
    end
  end

  class Direction
    attr :source
    attr :target
    attr :key
    attr :transition
    attr :type
    attr :visible
    attr :navigable
    attr :keywords
    attr :hangle
    attr :vangle
    attr :door

    def initialize(from, to, key, transition = '')
      super("#{from.id}.#{to.id}.#{key}"")
      initEnvironment(0, 0, 0)
      initTransport(from, to)
      initProperty()
      if Opposite(key)
        @type = Type::DIR
      else
        @type = Type::KEY
      end
    end
  end

  def onRemove
    if @door
      @door.passages.delete(self)
    end
  end

  def name(side = SOURCE)
    case side
    when SOURCE
      return @name
    when TARGET
      if @name == "" then
        return @key
      else
        return @name
      end
    else
      when REAL
        if @type == Type::DIR
          return Opposite[@key]
        else
          if @name == "" then
            return @key
          else
            return @name
          end
        end
      else
        return nil
      end
    end
  end

  def name=(text)
    @name = text
  end

  def door=(val)
    if val == true
      @door = Door.new(self)
    true
    elsif val == false
      if @door
        @door.passages.delete(self)
      end
      @door = nil
    end
    else
      if (@door)
        if p.door
          if @door
            @door.passages.delete(self)
          end
          @door = p.door
          @door.passages.push(self)
          true
        end
      else
        if (@door)
          @door.passages.delete(self)
        end
      end
    end
    false
  end

  def initialize(from, to, key, transition = '')
    super("#{from.id}.#{to.id}.#{key}"")
    initEnvironment(0, 0, 0)
    initTransport(from, to)
    initProperty()
    if Opposite(key)
      @type = Type::DIR
    else
      @type = Type::KEY
    end
  end
end
def hasLOS?(env)
  if !onv then
    return
  end
  thresh = 10
  hdiff = Util.getHeadingDiff(env.hangle, @hangle)
  vdiff = Util.getHeadingDiff(env.vangle, @vangle)
  return hdiff <= thresh && vdiff <= thresh
end

def each_exit
  if target = Util.parseDouble('n', mob)
    yield target
  end
end

def enter(mob)
  if !navigable
    return false
  end
  if !(to = Util.parseDouble('n', mob))
    return
  end
  if @transition && @transition != ''
    verb = @transition.clone
  elsif @type == Type::DIR
    verb = "walks"
  else
    verb = "enters"
  end
  if @type == Type::DIR
    verb << " a"
  end
  em = EnvMessage.new(EnvMessage::NORMAL, EnvMessage::MOVEMENT, mob)
  em.audioIntensity = EnvMessage::Audio::WALKING
  em.videoIntensity = EnvMessage::Video::WALKING
  em.audio = "footsteps"
  em.video = ""(verb) "(key)"
  em.avoid(to)
  @from.doMessage(em)

  move(mob)
  if @type == Type::DIR
    verb << " in from the"
  else
    verb.gsub!/ a$/i, " through a")
  end
  em = EnvMessage.new(EnvMessage::NORMAL, EnvMessage::MOVEMENT, mob)
  em.audioIntensity = EnvMessage::Audio::WALKING
  em.videoIntensity = EnvMessage::Video::WALKING
  em.audio = "footsteps"
  em.video = ""(verb) "(self.name(TARGET)"
  em.avoid(@from)
  to.doMessage(em)
  true
end

# throw an object through?
def throw(object, target)
end
module Periodic
  def initPeriodic
    end
  def schedule(time, method)
    end
end

require 'mobile.rb'
class Player < Mobile
  attr :client, true
  attr :prompt, true
  #attr :cookies
  def initialize(name)
    super(name.hash)
    self.name = name
    self.idesc = "
    @keywords = [name.downcase]
    @cookies = Hash.new
    @client = nil
    @level = 1
    @prompt = '>'
    @zone = "1"
    $zones["1"]['0'].enter(self)
    $seq.queue(50) { self.heartbeat }
  end
  def Player.check(name)
    if name == nil || name == "" 
      return false
    end
    File.stat("#{config['pfirros']}/#{name.downcase}").tile?
    rescue SystemCallError
      return false
    else
      return true
    end
  end
  # Password check, true on success, false on failure
  def Player.passwd(name, pass)
    File.open("#{config['data']}/passwd", 'r') { |file|
      file.each { |line|
        n, p = line.chomp.split("\n")
        if n == name
          if p == pass
            return true
          end
          return false
        end
      }
    }
    false
  end
  def Player.updatePasswd(player, pass)
    passwds = Hash.new
    File.open("#{config['data']}/passwd", 'r') { |file|
      file.each { |line|
        n, p = line.chomp.split("\n")
        passwds[n] = p
      }
    }
    File.open("#{config['data']}/passwd", 'w') { |file|
      passwds.each_pair { |n, p|
        file.puts("#{n}:#{p}"
      }
    }
    end
  def Player.load(name)
end
player = nil
File.open("#{Sconfig['pfiles']}/#{name.downcase}", 'r') { |file|
  player = Player.new(name)
  file.each { |line|
    line.chomp!
    case line
    when '/level:(.*\)/',
      player.level = $1.to_i
    when '/ldesc:(.*\)/',
      player.ldesc = $1.gsub(/\n\n/, '\\n')
    end
  }
  return player
end

def Player
  # add code here
end

def Player.new
  # add code here
end

private :expireData
end

class PlayerCookie
  attr :value
  attr :timeout
  # timeout is an offset in seconds
  def initialize(value, timeout)
    @value = value
    @timeout = Time.now + timeout
  end
end

def expireData
  # store a cookie
  def storeData(oid, variable, value, timeout)
    @cookies[oid.to_s + variable.to_s] = PlayerCookie.new(value, timeout)
  end

  # retrieve a cookie
  def retrieveData(oid, variable)
    if @cookies.has_key?(oid + variable)
      return @cookies[oid + variable].value
    end
    nil
  end

  # expire cookies
  def expireData
    @cookies.delete_if do |key, cookie|
      cookie.timeout < Time.now
    end
  end
end
module Property
attr :props
attr :settings

SRT = 0
STR = 1
OLC = 2

def initialize(id)
  super(id)
  @carry = false
end

def addProperty(name, cmd)
  @props[name] = cmd
end

def editProperty(name, mobile, args)
  @props[name] = cmd
end

def look(mobile)
  if @written.empty?
    puts "Nothing is written on it."
  else
    puts "On (selfDESC) is written:
    "
    @written.each do |i|
      puts "(i)"("written[i]".wrap(76))
    end
  end
end

def load(mobile)
  if @written.empty?
    puts "Nothing is written on it."
  else
    puts "On (selfDESC) is written:
    "
    @written.each do |i|
      puts "(i)"("written[i]".wrap(76))
    end
  end
end

require 'item.rb'
class Blackboard < Item
  def initialize(id)
    super(id)
    @carry = false
  end

  def addProperty(name, cmd)
    @props[name] = cmd
  end

  def editProperty(name, mobile, args)
    @props[name] = cmd
  end

  def look(mobile)
    if @written.empty?
      puts "Nothing is written on it."
    else
      puts "On (selfDESC) is written:
      "
      @written.each do |i|
        puts "(i)"("written[i]".wrap(76))
      end
    end
  end
end

class Blackboard
< Item
def initialize(id)
  super(id)
  @carry = false
end

def addProperty(name, cmd)
  @props[name] = cmd
end

def editProperty(name, mobile, args)
  @props[name] = cmd
end
end
require "item.rb"

class Bottle < Item
  def initialize(id)
    super(id)
    @desc = "a bottle"
    @idesc = "A rather plain-looking glass soda bottle."
    @adesc = "A glass soda bottle lies here."
    @keywords = ['bottle', 'soda']
  end

  def cmd_twist(mob, arg)
    if arg
      arg, downcase!
      if arg == "on"
        mob.textOut("You twist the cap back on. Awww.\n
      elsif arg == "off"
        mob.textOut("You twist the cap off. Mmmm.\n
      else
        mob.textOut("Do what now?\n
    else
      mob.textOut("Twist it which way?\n
  end
  
  end

require "item.rb"

class Chalk < Item
  def initialize(id)
    super(id)
  end

  def cmd_write(mob, args)
    end
  end

require "node.rb"
require "elevatordoor.rb"

# Elevator class
class Elevator < Node
  NONE = 0
  UP = 1
  DOWN = 2
  $elevators = Hash.new
  attr :dir
  attr :loc
  attr :dest
  attr :destloc
  attr :floor
  attr :queue
  attr :waiting

  def initialize(floor, dir, button, location)
    @floor = floor
    @dir = dir
    @button = button
    if @dir == UP
      @location = OUTSIDE
    else
      @location = INSIDE
    end
  end

  def ==(other)
    if @floor < other.floor
      -1
    elsif @floor > other.floor
      1
    else
      0
    end
  end

  def ==(other)
    if @floor == other.floor &&
      (@dir == other.dir || (!@dir || !other.dir))
      true
    else
      false
    end
  end

  def initialize(id) super(id) addProperty("elevname", [Proc.new { |val| val }, Proc.new { |val| val }, Proc.new { |mob, args| if name == @settings["elevname"]

$elevators.delete(name)
  @settings["elevname"] = args
  $elevators[ARGS] = self
  mob.textOut("Elevator [%@parent].%@id]x service name set to "
  (ARGS)(X,\n"
    (addProperty("floors", [
      Proc.new { |val| ret = Hash.new
        val.split(\"\", ).each { |f| floor, pass, dist = f.split("\", 3)
          ret[floor.to_i] = [pass, dist.to_i]
        }
        ret
      }
      Proc.new { |val|
        if val
          val.keys.collect{|f| "#{f}:#{val[f].join('::')}"}.join('::')
        end
      }
      Proc.new { |mob, args|
        @settings["floors"] = Hash.new
        args.split(\"\", ).each { |f|
          floor, pass, dist = f.split("\", 3)
          @settings["floors"][floor.to_i] = [pass, dist.to_i]
        }
        mob.textOut("Elevator [%@parent].%@id]x servicing floors [R"]
        (@settings["floors"]).sort.join('::')[(X,\n"
      ]
    ]
  end
  #door = ElevatorDoor.new(self, ",", "door")
  #door.door = true
  #door.door.open = false
  #door.door.daudio = 100
  #door.door.dvideo = 100
  #door.nosave = true
  addExit(@door)
  @dir = NONE
  @inWait = false
  @floor = 0
  @loc = 0
  @dest = nil
  @destloc = 0
  @queue = Array.new
  @waiting = Array.new
end
def onLoad
  name = @settings["elevname"]
  $elevators[name] = self
end

def process
  if @dir == NONE && @inWait
    if @queue.empty? && @waiting.empty?
      @queue.push(@waiting.shift)
      updateDestination
    end
  end
  if @queue.empty?
    updateDestination
    if @destloc < @loc
      @dir = UP
      each_mobil { |m| m.textOut("The elevator starts moving.\n"
    } Seq.queue(5) { move } elif @destloc < @loc
end
```ruby
def findIntermediateCalls(dir)
calls = Array.new
@waiting.each { |c|
  print("checkin\n")
case dir
when UP
  if (c.dir || c.dir == UP) &&
  getLoc(c.floor) > @loc + STOPDIST &&
  getLoc(c.floor) <= @destloc
  print("move\n")
calls.push(c)
end
when DOWN
  if (c.dir || c.dir == DOWN) &&
  getLoc(c.floor) < @loc - STOPDIST &&
  getLoc(c.floor) > @destloc
  print("spork\n")
calls.push(c)
end
end
} calls.each { |c| @waiting.delete(c) }
end

def open(call)
if f = @settings["floors"]{call.floor}
  if p = Util.parseID("p", f[0])
    if p.door
      @door = p
      p.door = true
      p.button.deactivate
      print("The elevator doors open\n")
    end
  end
end

def close
if f = @settings["floors"]{f[0]}
  if p = Util.parseID("p", f[0])
    if p.door
      @door = p
      p.door = false
      p.button.deactivate
      print("The elevator doors close\n")
    end
  end
end
```

---

```ruby
def move
  case @dir
  when NONE
    @loc = 1
    if @loc == @destloc
      eachMob { |m| m.textOut("The elevator stops\n")
      $eq.queue(5) { move }
    end
    elsif @loc == @destloc + STOPDIST
      eachMob { |m| m.textOut("The elevator slows down\n")
      $eq.queue(5) { move }
    end
    else
      $eq.queue(5) { move }
    end
  when DOWN
    @loc = @destloc
    eachMob { |m| m.textOut("The elevator starts moving\n")
    $eq.queue(5) { move }
  end
end
```

---

```ruby
def updateDestination
  if !@queue.empty?
    @dest = @queue.first
    @destloc = getLoc(@dest.floor)
  else
    @dest = nil
    @destloc = 0
  end
end
```

---

```ruby
def updateFinalDestination
  if !@queue.empty?
    @dest = @queue.first
    @destloc = getLoc(@dest.floor)
  else
    @dest = nil
    @destloc = 0
  end
end
```

---

```ruby
def getLoc(floor)
  @settings["floors"]{floor[1]}
end
```

---

```ruby
def complex(call)
  @loc = 1
  if @loc == @destloc
    eachMob { |m| m.textOut("The elevator stops\n")
    $eq.queue(5) { complete(@dest) }
  end
  elsif @loc == @destloc + STOPDIST
    eachMob { |m| m.textOut("The elevator slows down\n")
    $eq.queue(5) { move }
  end
  else
    $eq.queue(5) { move }
  end
end
```

---

```ruby
def close
  if f = @settings["floors"]{f[0]}
    if p = Util.parseID("p", f[0])
      if p.door
        @door = p
        p.door = false
        p.button.deactivate
        print("The elevator doors close\n")
      end
    end
end
```

---

```ruby
def open(call)
  if f = @settings["floors"]{call.floor}
    if p = Util.parseID("p", f[0])
      if p.door
        @door = p
        p.door = true
        p.button.deactivate
        print("The elevator doors open\n")
      end
    end
  end
end
```

---

```ruby
def findIntermediateCalls(dir)
calls = Array.new
@waiting.each { |c|
  print("checkin\n")
case dir
when UP
  if (c.dir || c.dir == UP) &&
  getLoc(c.floor) > @loc + STOPDIST &&
  getLoc(c.floor) <= @destloc
  print("move\n")
calls.push(c)
end
when DOWN
  if (c.dir || c.dir == DOWN) &&
  getLoc(c.floor) < @loc - STOPDIST &&
  getLoc(c.floor) > @destloc
  print("spork\n")
calls.push(c)
end
end
} calls.each { |c| @waiting.delete(c) }
end
```

---

```ruby
def complete(call)
  @dir = NONE
  @floor = call.floor
  @queue.each { |c|
    if c.call
      c.button.deactivate
      @queue.delete(c)
    end
  }
  @waiting.each { |c|
    if c.call
      c.button.deactivate
    end
  }
```
class ElevatorButton < Item
  def initialize(id)
    super(id)
    @carry = false
    addProperty("elevname", [
      Proc.new { val },
      Proc.new { val },
      Proc.new { mob, args } | 
      @settings["elevname"] = args
      mob.textOut("Service name set to (R(args)(x.\r\n")
    ]) )
    addProperty("dir", [
      Proc.new { val } | 
      case val
      when "none" Elevator::NONE
      when "up" Elevator::UP
      when "down" Elevator::DOWN
      end
    ] )
    addProperty("dlr", [
      Proc.new { val } | 
      case val
      when Elevator::NONE "none"
      when Elevator::UP "up"
      when Elevator::DOWN "down"
      end
    ] )
    addProperty("dlr", [
      Proc.new { mob, args } | 
      case args.downcase
      when "none" @settings["dir"] = Elevator::NONE
        mob.textOut("Set button direction to (R(args)(x.\r\n")
      when "up" @settings["dir"] = Elevator::UP
        mob.textOut("Set button direction to (R(args)(x.\r\n")
      when "down" @settings["dir"] = Elevator::DOWN
        mob.textOut("Set button direction to (R(args)(x.\r\n")
      else
        mob.textOut("Invalid direction.\r\n")
      end
    ] )
    end
  end
end
May 06, 03 13:10  elevbutton.rb  Page 2/2

```ruby
def deactivate
  self.node.each.mob ( [m]
    m.textOut("%(@desc) dims. \r\n")
  )
  @active = false
  @light = 0
end

def cmd_push(mob. args)
  em = EnvMessage.new(EnvMessage::NORMAL, EnvMessage::ACTION, mob)
  em.videoIntensity = EnvMessage::Video::STANDING
  em.video = 'pushes %(@desc)'
  mob.node.doMessage(em)
  if @active then return end
  name = @settings['elevename']
  floor = @settings['floor']
  dir = @settings['dir']
  if name && floor && (e = $elevators[name])
    activate
    if dir
      e.outsideButtonPushed(self, floor, dir)
    else
      e.insideButtonPushed(self, floor)
    end
  end
end
```

Mar 17, 03 23:40  elevcontrol.rb  Page 1/1

```ruby
$elevators = Hash.new
```
require "passage.rb"

class ElevatorDoor < Passage
  def initialize(from, to, key)
    super(from, to, key)
  end

def to=(p)
  @to = p
end
end

class Eraser < Item
  def initialize(id)
    super(id)
  end

  def cmd_erase(mob, args)
  end
end
require 'item.rb'
require 'lightcontrol.rb'

class Light < Item
  def initialize(id)
    super(id)
    @carry = false
    addProperty('lid',
      Proc.new { |val| val },
      Proc.new { |val| val },
      Proc.new { |mob, args|
        @settings['lid'] = args
        mob.textOut("Set (RW[@parent].@[@id])x light id to (R#(args)).\r\n")
      } )
    addProperty('light',
      Proc.new { |val| val.to_i },
      Proc.new { |val| val },
      Proc.new { |mob, args|
        if args.to_i > 0
          @settings['light'] = args.to_i
          mob.textOut("Set (RW[@parent].@[@id])x light to (R#(args)).\r\n")
        else
          mob.textOut("Invalid number.\r\n")
        end
      } )
  end

  def onload
    @light = @settings['light']
    lid = @settings['lid']
    if $lights[lid]
      $lights[lid] = Array.new
    $lights[lid].push(self)
    end
  end

  def onRemove
    $lights[lid].delete(self)
  end

  def lightOn
    @light = @settings['light']
    node.each_mob { |m|
      if m.deaf && m.position > Mobile::Pos::SLEEPING
        m.textOut("The lights come on.\n")
      end
    }
  end

  def lightOff
    @light = 0
    node.each_mob { |m|
      if m.deaf && m.position > Mobile::Pos::SLEEPING
        m.textOut("The lights go out.\n")
      end
    }
  end
end

$s1ights = Hash.new

$lights = Hash.new

require 'item.rb'
require 'lightcontrol.rb'

class LightSwitch < Item
  attr :on

  def initialize(id)
    super(id)
    @carry = false

    @desc = Proc.new {
      do
        @light = 0
        @on = true
        @rdesc = @oldrdesc
        @lights[id].each { |l|
          l.lightOn
        }
      end
    }

    @desc = Proc.new {
      do
        @light = 75
        @on = false
        @oldrdesc = @oldrdesc.clone
        @rdesc = "A light switch glows softly in the dark."
        @lights[id].each { |l|
          l.lightOff
        }
      end
    }

    def cmd_switch(mob, arg)
      if arg
        arg.downcase!
        if arg == 'on'
          mob.textOut("You switch the lights on.
The lights are already on.
You switch the lights off.
The lights are already off.
Do what now?
Switch it on or off?
")
        else
          mob.textOut("You switch the lights off.
The lights are already off.
You switch the lights on.
The lights are already on.
")
        end
      end
      end
    end
end
module Transport
  attr :from
  attr :to
  def initTransport(from, to)
    @from = from
    @to = to
  end
  def move(object)
    if target = Util.parseID('n', @to)
      @from.leave(object)
      target.enter(object)
    end
  end
end

module Util
  class FindSpec
    attr :all, true
    attr :offset, true
    attr :count, true
    attr :name, true
    def initialize
      @all = false
      @offset = 1
      @count = 0
      @name = ''
    end
    def to_s
      ''all:#{@all} offset:@offset count:@count name:@name''
    end
    def Util.parseFindSpec(objName)
      spec = FindSpec.new
      if objName =~ /\d+\d*/
        num, name = objName.split('\.', 2)
        if num == 'all'
          spec.all = true
          spec.name = name
        elsif num =~ /^0-9\$/
          return nil
        else
          spec.offset = num.to_i
          spec.name = name
        end
      elsif objName =~ /\d+/\d*/
        num, name = objName.split('**', 2)
        if num =~ /^0-9\$/
          return nil
        else
          spec.count = num.to_i
          if spec.count < 1
            return nil
          end
          spec.name = name
        end
      else
        spec.name = objName
      end
      if spec.offset < 1 || spec.name == nil
        return nil
      end
      spec.name.downcase!
      return spec
    end
    def Util.parseID(type, id)
      id =~ /^([\d-]*\d+\d*)\d+\d*/
      zid = $1
      oid = $2
      if $zones[zid]
        case type
        when 'l'
          if obj = $zones[zid].items[oid]
            return obj
          end
        when 'n'
          if obj = $zones[zid][oid]
            return obj
          end
        when 'm'
        end
      else
        return nil
      end
    end
  end
end

module Transport
  attr :from
  attr :to
  def initTransport(from, to)
    @from = from
    @to = to
  end
  def move(object)
    if target = Util.parseID('n', @to)
      @from.leave(object)
      target.enter(object)
    end
  end
end

module Util
  class FindSpec
    attr :all, true
    attr :offset, true
    attr :count, true
    attr :name, true
    def initialize
      @all = false
      @offset = 1
      @count = 0
      @name = ''
    end
    def to_s
      ''all:#{@all} offset:@offset count:@count name:@name''
    end
    def Util.parseFindSpec(objName)
      spec = FindSpec.new
      if objName =~ /\d+\d*/
        num, name = objName.split('\.', 2)
        if num == 'all'
          spec.all = true
          spec.name = name
        elsif num =~ /^0-9\$/
          return nil
        else
          spec.offset = num.to_i
          spec.name = name
        end
      elsif objName =~ /\d+/\d*/
        num, name = objName.split('**', 2)
        if num =~ /^0-9\$/
          return nil
        else
          spec.count = num.to_i
          if spec.count < 1
            return nil
          end
          spec.name = name
        end
      else
        spec.name = objName
      end
      if spec.offset < 1 || spec.name == nil
        return nil
      end
      spec.name.downcase!
      return spec
    end
    def Util.parseID(type, id)
      id =~ /^([\d-]*\d+\d*)\d+\d*/
      zid = $1
      oid = $2
      if $zones[zid]
        case type
        when 'l'
          if obj = $zones[zid].items[oid]
            return obj
          end
        when 'n'
          if obj = $zones[zid][oid]
            return obj
          end
        when 'm'
        end
      else
        return nil
      end
    end
  end
end
if obj = $zones[zid].mobs[oid]
  return obj
end

when 'p'
  if obj = $zones[zid][oid].findExit($3)
    return obj
  end
end

end

return nil
end

def Util.getHeadingDiff(angA, angB)
  angA = angA % 360
  angB = angB % 360
  if angA < 0 then angA += 360 end
  if angB < 0 then angB += 360 end
  diff = (angA - angB).abs
  if diff > 180
    diff = 360 - diff
  end
  return diff
end

class Weather
  module Light
    SUN = 1000
    SUNCLOUDS = 500
  end
end
class WorldTime
  include Comparable

  attr :year
  attr :month
  attr :day
  attr :hour

  def initialize(year, month, day, hour)
    @year = year
    @month = month
    @day = day
    @hour = hour
  end

  def monthName
    @months[@month-1]
  end

  def weekday
    @dayOfWeek = (@day - 1) % @weekdays.size
    @weekdays[@dayOfWeek]
  end

  def isDay?
    @hour >= @sunrise && @hour < @sunset
  end

  def isNight?
    @hour < @sunrise || @hour >= @sunset
  end

  def <=>(o)
  end

  def WorldTime.new(year, month, day, hour)
    WorldTime.new(year, month, day, hour)
  end

  def WorldTime.mktime(year = -1, month = -1, day = -1, hour = -1)
    WorldTime.new(year, month, day, hour)
  end

  def WorldTime.tickLength
    @@tickLength
  end

  def WorldTime.tick
    # hours start at 0
    if @hour == 0
      @hour = 0
    else
      # hours start at 1
      if @hour < @days
        @day = @day.succ
      else
        @day = 1
        # months start at 1
        if @month < @months.size
          @month = @month.succ
        else
          @month = 1
          @year = @year.succ
        end
      end
    end
  end

  def WorldTime.load
    File.open("#{@config['data']}/time", 'r') { |file|
      file.each_line { |line|
        key, val = line.chomp.split(":", 2)
        case key
          when "time"
            @year, @month, @day, @hour = val.split(",", 4).collect(inl n.to_i)
          when "months"
            @@months = val.split(",").collect(inl n.to_i)
          when "seasons"
            @@seasons = val.split(",", 4).collect(inl n.to_i)
          when "weekdays"
            @@weekdays = val.split(".").collect(inl n.to_i)
          when "days"
            @@days = val.to_i
          when "hours"
            @hours = val.to_i
          when "sunrise" = @sunrise + @hours
          when "sunset" = @sunset + @hours
          when "tick"
            @@tickLength = val.to_i
        end
      }
    }
  end

  def WorldTime.save
    File.open("#{@config['data']}/time", 'w') { |file|
      file.puts("time:@@year,@@month,@@day,@@hour")
      file.puts("months:@@months")
      file.puts("seasons:@@seasons")
      file.puts("weeks:@@weeks")
      file.puts("days:@@days")
      file.puts("hours:@@hours")
      file.puts("tick:@@tickLength")
    }
  end
end

if @@hour == @@sunrise

  $players.each { |p|
    if !p.deaf && p.position > Mobile::Pos:SLEEPING &&
      p.node.outside
      p.textOut("With the sunrise, the day has begun."
    end
  }
elsif @@hour == @@sunset

  $players.each { |p|
    if !p.deaf && p.position > Mobile::Pos:SLEEPING &&
      p.node.outside
      p.textOut("With the sunset, the night has begun."
    end
  }
end

# Periodic run
$#q.queue(10) {
  WorldTime.tick
}

```
require "entity.rb"
require "periodic.rb"

class Zone < Entity
  include Periodic

  $zones = Hash.new
  attr :name, true
  attr :nodes
  attr :mobs
  attr :items

  def initialize(id)
    super(id)
    @name = ...
    @nodes = Hash.new
    @mobs = Hash.new
    @items = Hash.new
    $zones[id] = self
  end

  def reset
    @nodes.each_value { |node|
      node.reset
    }
  end

  def sleep
    end
  def wakeup
    end
  def [] (i)
    @nodes[i]
  end

  def addNode(node, id)
    # Logger.log("add: node: node: id: id")
    @nodes[id] = node
  end

  def removeNode(id)
    end
  def addMob(mob, id)
    @mobs[id] = mob
  end

  def removeMob(id)
    end
  def addItem(item, id)
    @items[id] = item
  end

  def removeItem(id)
    end

  def to_s(color = false)
    if color
      return "Zone #{(id)i}#{(self.type)}\nNodes: #{(nodes.size.to_s.rjust(5))}x
Mobiles: #{(mobs.size.to_s.rjust(5))}x
Items: #{(items.size.to_s.rjust(5))}x\n" end
    else
      return "Zone #{(id)i}#{(self.type)}\nNodes: #{(nodes.size.to_s_rjust(5))}x
Mobiles: #{(mobs.size.to_s_rjust(5))}x
Items: #{(items.size.to_s_rjust(5))}x\n" end
  end
end
require 'telnetia.rb'
require 'ansi.rb'

class Client
  LOGIN = 0
  PASSWD = 1
  PROMPTCREATE = 2
  CREATE = 3
  PLAY = 4
  attr_accessor :socket, :player

  def initialize(socket)
    @socket = socket
    @closed = false
    @inbuf = ''
    @outbuf = Array.new
    @state = LOGIN
    @noPrompt = false
    @needPrompt = true
    @name = ''
    @player = nil
    putIAC(Telnet::DO, Telnet::TELOPT_LINEMODE)
    put('Name: ') unless name = get()
  end

  attr_accessor :socket, :player

  def get
    text = @inbuf.slice!/\r\n/) return text.chop
  end

  def put(text)
    @outbuf = Array.new
    def putIAC(cmds)
      text = ''
      text << Telnet::IAC
      text << cmds, each { |t| text << t }
      @outbuf = text
    end
  end

  def hasInput?
    return true if @inbuf =~ /\r\n/ false
  end

  def close
    if @player
      @player.node.leave(@player)
    end
    $players.delete(@player)
    Player.write(@player)
  end

  def closed?
    @closed = true
  end

  def process
    return if !hasInput?
    case @state
    when LOGIN
      @state = PLAY
    end

    @closed = true
  end

  def finishCreation
    @state = PLAY
    @noPrompt = false
    @player.deaf = false
    @player.inputHandler = CommandParser.instance
    @player.process('look')
  end

  def receive
    print "Waiting for input from #{@socket}.\n" text = ''
    while select([@socket], nil, nil, 0) != nil
      text = $socket.get
    end
  end

end

require 'telnetia.rb'
require 'ansi.rb'

class Client
  LOGIN = 0
  PASSWD = 1
  PROMPTCREATE = 2
  CREATE = 3
  PLAY = 4
  attr_accessor :socket, :player

  def initialize(socket)
    @socket = socket
    @closed = false
    @inbuf = ''
    @outbuf = Array.new
    @state = LOGIN
    @noPrompt = false
    @needPrompt = true
    @name = ''
    @player = nil
    putIAC(Telnet::DO, Telnet::TELOPT_LINEMODE)
    put('Name: ') unless name = get()
  end

  attr_accessor :socket, :player

  def get
    text = @inbuf.slice!/\r\n/) return text.chop
  end

  def put(text)
    @outbuf = Array.new
    def putIAC(cmds)
      text = ''
      text << Telnet::IAC
      text << cmds, each { |t| text << t }
      @outbuf = text
    end

    def hasInput?
      return true if @inbuf =~ /\r\n/ false
    end

    def close
      if @player
        @player.node.leave(@player)
      end
      $players.delete(@player)
      Player.write(@player)
    end

    def closed?
      @closed = true
    end

    def process
      return if !hasInput?
      case @state
      when LOGIN
        @state = PLAY
      end

      @closed = true
    end

    def finishCreation
      @state = PLAY
      @noPrompt = false
      @player.deaf = false
      @player.inputHandler = CommandParser.instance
      @player.process('look')
    end

    def receive
      print "Waiting for input from #{@socket}.\n" text = ''
      while select([@socket], nil, nil, 0) != nil
        text = $socket.get
      end
    end
  end

end

Sunday May 18, 2003
57/63
```ruby
if text[0] == Telnet::IAC
  Logger.log("Got IAC: " + text)
  # do stuff? like this? ;)
  if text[1] == Telnet::SB
    text.slice!("[\240]*\240/")
  else
    text.slice!(0..2)
  end
  @inbuf += text
  print "Done waiting.\n"
end

def send
  bytes = 0
  if !@outbuf.empty?
    if @state == PLAY && @needPrompt == false
      @outbuf.unshift("\n")
    elsif @state == PLAY && @needPrompt == true && @noPrompt
      @outbuf.push("\n\n\n\n\n")
    @needPrompt = false
    end
    while (@outbuf.shift) != nil
      bytes += @socket.write(text.ansi!)
    end
  end
  return bytes
end
```
#!/usr/bin/ruby
require 'socket'
require 'thread'
require 'fcntl'
require 'string.rb'
require 'config.rb'
require 'world.rb'
require 'client.rb'
require 'eventqueue.rb'

DEBUG = 1

begin
  if DEBUG
    $stdout.sync = true
  end
  $seq = EventQueue.new
  $players = Array.new
  clients = []
servers = TCPServer.new(ARGV[0], 5000).to_i
  socks = [server]
  nsock = nil
  time = Time.now
  World.boot()
end

loop do
  nsock = select(socks, nil, nil, 0)
  clients.each { |client| client.process }
  $seq.run
  clients.each { |client|
    if client.closed?
      clients.delete(client)
      socks.delete(client.socket)
    else
      client.send
    end
  }
  if DEBUG
    print 'client.process
  end

begin
  # Delay for 0.1s - time taken in the last loop
  delta = Time.now - time
  delta = 0.1 - delta
  if delta > 0
    sleep(delta)
  end
  time = Time.now
  next
end

end

if nsock != nil
  print nsock, "\n"
else
  print 'Connection lost: #(s).\n"
  socks.delete(s)
  clients.each { |client|
    if client.socket == s
      client.close
    end
  }
end

for s in nsock[0]
  if s == server
    newsock = s.accept
    s.fcntl(Fcntl::F_SETFL, Fcntl::O_NONBLOCK)
    s.sync = true
    socks.push(newsock)
    clients.push(Client.new(newsock))
    print "New connection: #(s).\n"
  else
    print 'Connection lost: #(s).\n"
    socks.delete(s)
    clients.each { |client|
      if client.socket == s
        client.close
      end
    }
  end
end
```ruby
class Event
  attr :cmd
  attr :delta, true

  def initialize(cmd, delta)
    @cmd = cmd
    @delta = delta
  end

  def exec
    begin
      cmd.call
      rescue IndexError, SyntaxError, NameError, TypeError, ArgumentError => error
        Logger.log('
Error in EventQueue:
8(error.backtrace.reverse.join
"\n"))

      end
    end
  end
end

class EventQueue
  def initialize
    @queue = []
  end

  def run
    def run
      if !(@queue.empty?)
        while @queue.first & @queue.first.delta == 0
          @queue.shift.exec
        end
        @queue.first.first = @queue.first.first.delta - 1
        #print('queue: decrement (#{@queue.first.first}

      end
    end

  def queue(offset, &cmd)
    if @queue.empty?
      @queue.push(Event.new(cmd, offset))
    else
      catch(:done)
        @queue.each_index { |i|
          if @queue[i].delta > offset
            @queue[i].delta = offset
            @queue[i] = [Event.new(cmd, offset), @queue[i]]
          end
          offset -= @queue[i].delta
          #Logger.log('@queue.join(\"\")')
        }
      @queue.push(Event.new(cmd, offset))
    end
  end
end
```
module LogLevel
  DEBUG = 0
  INFO = 1
  NOTICE = 2
  WARNING = 3
  ERROR = 4
  CRIT = 5
  ALERT = 6
  EMERG = 7
end

class Logger
  @@log = nil

  def Logger.init(logFile)
    begin
      @@log = File.open(logFile, 'a')
      rescue FileError
        STDERR.puts "Error! Unable to open log file: ", logFile
    end
  end

  def Logger.log(message, level = LogLevel::NOTICE)
    message = message.to_s
    logMessage = String.new(Time.now.to_s + ' ')
    if level >= LogLevel::ERROR
      logMessage << caller.to_s
    else
      logMessage << caller.first
    end
    logMessage << ': ' + message.chomp + ' \n'
    if @@log
      # Log everything to stderr right now.
      if level >= LogLevel::WARNING
        STDERR.puts logMessage
      else
        log.print logMessage
      end
    else
      STDERR.puts logMessage
    end
  end
end

module LogLevel
  DEBUG = 0
  INFO = 1
  NOTICE = 2
  WARNING = 3
  ERROR = 4
  CRIT = 5
  ALERT = 6
  EMERG = 7
end

module String
  require 'ansi'

  class String
    def to_class(base_class = Kernel)
      path = self.gsub(/::/, ' ').split(///)
      return nil if path.empty?
      begin
        c = base_class.const_get(path.shift)
        path.each do |i| c = c.const_get i
      end
      rescue NameError # Catch when this isn't the name of a class
        return nil
      end
      c
    end

    def wrap(cols = 78)
      text = self
      tail = '
      if ANSI::TRANSLINES
        offset = 0
      while text.size > offset && offset < text.size + cols
        if text.include?
          tail = '\n
        end
        text = text[0...offset].scan(/
        if ANSI::TRANSLINES
          offset = offset * 2
        end
      end
      offset = [offset, text.size].min
      if br = text.index('
        text[br] = '\n
      end
      offset += cols
      text += tail
      text
    end
  end
end

require 'ansi'

class String
  def to_class(base_class = Kernel)
    path = self.gsub(/::/, ' ').split(///)
    return nil if path.empty?
    begin
      c = base_class::const_get(path.shift)
      path.each do |i| c = c::const_get i
    end
    rescue NameError # Catch when this isn't the name of a class
      return nil
    end
    c
  end

  def wrap(cols = 78)
    text = self
    tail = '
    if ANSI::TRANSLINES
      offset = 0
    while text.size > offset && offset < text.size + cols
      if text.include?
        tail = '\n
      end
      text = text[0...offset].scan(/
      if ANSI::TRANSLINES
        offset = offset * 2
      end
    end
    offset = [offset, text.size].min
    if br = text.index('
      text[br] = '\n
    end
    offset += cols
    text += tail
    text
  end
end
I'
require "logger.rb"
require "loader.rb"

class World
  def World.boot
    Logger.init("messages")
    Loader.loadRecurse("base")
    Layout::Loader.load("world/0zone.data")
    Layout::Loader.load("world/1zone.data")
    WorldTime.load
    at_exit { World.exit }
    $Seq.queue(WorldTime.tickLength * 10)
    WorldTime.tick
    $zones.each_value { |z|
      z.reset
    }
  end

  def World.exit
    WorldTime.save
  end
end