Mislead

An IF Engine & CSCI1101 Project

Peter Smith
Zack MacKay
Izik Arruda
Eric Nguyen
Dylan Gidney

-

Objective

Mislead is a java-based code set for the genre of game known as text adventures or interactive fiction. In a text adventure, the player interacts with the game world by typing simple commands which are interpreted by a parser into game actions. The game then outputs text describing what the player can see and hear. Through exploring and examining the world, the player learns sufficient information to understand the game's model and solve its puzzles.

At its most basic, traditional interactive fiction allows the player to move between multiple rooms and pick up objects. It also allows them to use those objects to solve puzzles by using them on the environment. The interface of a text parser usually interprets commands in the simple imperative form, such as CLOSE DOOR. In the oldest of games, most actions that did not lead to solving a puzzle were not understood, returning an error message: "You can't pick that up", "There's no need to eat that", that sort of thing. Most basic puzzles will require implementing commands that contain both a primary and a secondary object. LOCK DOOR WITH KEY, for example.

The player must be capable of listing the objects they are carrying using a command called "inventory" to keep track of progress. This command allows the player to quickly take stock of what they've picked up, and thus their options for solving a problem in front of them.

More sophisticated game concepts require the modeling of time, usually by turn count: counting the actions the player has taken. For example, a lit fuse might take a while to burn, or a large monster might occasionally move from room to room.

Design

Mislead is separated into two parts: One section, called the "Parser", is for communicating with the player through the interface, accepting input, and translating it into actions. The other section, the "Model", models the state and behaviour of the game world, tracking objects, rooms, and the position of the player.

The Parser

The Game Class

The game object, once instantiated, creates a game frame that contains UI elements for the player: A text box that contains the output of all game text, and an input box below it that allows the player to input commands and hit "enter", which sends the output to a parser object.

It uses multiple text display fields which are all sorted with a BorderLayout object. An instance of JscrollPane stores previous input and output. To one side, a smaller text pane displays the open exits,

and items visible in the room. The input field is a Jtextfield object, which, whenever “enter” is pressed, sends the string contained in it to the parser.

The game class also contains three static fields – one for a player object, one for a parser object, and one for a time object. Information about these three objects is needed throughout the program, so putting references to them in static fields makes them easy to find and access when we need to.

The interface frame also contains a compass that displays available exits from the room the player is standing in, which makes it easier for the player to orient themselves. It changes the state of the compass by polling the state of each of the six exits from the room the player is standing in. (See the “Room” and “Exit” class.)

The Parser Class

Commands from the player are routed to the Parser class, which interprets them using its parse(String in) function. It assumes the command is space character delimited and separates the string into single-word tokens using a StringTokenizer, and then shuffles those tokens into a stack. As the StringTokenizer pushes the words into the stack in order, we get them out of the stack backwards.

This means that a command such as OPEN DOOR WITH KEY gets read as KEY WITH OPEN DOOR. The parser class looks at each word in sequence, figures out what sort of word it is, and performs a behaviour based on what it sees.

The parser contains lists of words: verbs and prepositions are hard-coded, whereas the list of nouns is automatically generated, with each thing created in the game model adding its name to the list using the constructor.

On seeing a noun, the Parser assumes it indicates the object of the action. It then tries to find the thing, looking in the player’s inventory for items the player is carrying, and in the room’s contents for things the player can see. If it can’t find the object, the action fails, telling the player “I can’t see that here.”

Upon seeing a preposition, such as “on”, the Parser then realizes the object it saw earlier is actually the secondary object of the action, and thus moves the object into the secondary object.

Upon seeing a verb, the Parser takes the object and the secondary object (if it has one) and asks the Understand class to turn it into an action.

The Understand Class

The Understand class is a large switch block that takes the objects given to it by the Parser and a string containing a verb and translates it to an action. Its three understand methods correspond to zero, one, or two nouns. In the one or two noun case, it calls the relevant action method of the primary object of the command. In the no-noun case, it calls a special action: Either travelling in a direction (such as NORTH, SOUTH, and so on: Those are verbs in an IF game!) or calling a special action, such as “i” for taking inventory.

The Io Class

The Io class (for “In / Out”) is a utility class whose static method, out(String str) forwards a string to the game object to be printed. It exists to make it easy to change the output from going to the console to going to a game object without having to change the code in more than one place, and because “Io.out(...)” is easy to remember and fast to type.

The Model

The Thing Class

Every object in the game world that the player can interact with inherits functionality from the Thing class. Its task is to contain the default failure behaviour for all actions and functionality, which are then overwritten by individual subclasses.

Each thing has a method for each verb in the game. When the player tries eating a lit torch, the action that gets called is torch.eat(). Even though eating a torch is ridiculous, we’ve promised the Understand class that calling thing.eat() makes sense no matter what thing you give it, so all things need to have some response to such a call. (For the most part, it’s a failure message such as “You can’t eat that.”)

Individual things inherit from the thing class and overwrite their methods with more specific ones that change the game state; for example, to put a hamburger in the game, one would write a Hamburger class that extends from Thing, and have it replace the default eat() method with one that contains the description and effects of eating a hamburger.

All things have a name, which the player and game refer to it by, and a description, which is printed when the player examines the thing closely.

Not every single thing mentioned in the game is modeled as a thing: For example, we don’t instantiate the soil beneath your feet or the sound of rain, even though we might mention them in the description – unless either of those is used in a puzzle.

The Carryable Class

Carryables are a special sort of thing that can be picked up and carried with the player, thus the name: Items the player can get. The get() and drop() functions of a carryable move the item from the room into the player’s inventory, or vice-versa, if appropriate.

The Room Class

Every location in the game that the player can visit is an instance of the Room class. All rooms have a name and a description. The description being how the room is described to them when they enter, or

look. On visits after the first, the full description is replaced with a shorter one, since it's likely that the player already has some idea what's in there, and is only interested in the most important parts of it.

Every room contains a Thinglist called its contents: All the things the player can interact with in the room.

Each room has six fields for exits, which describe its connections to other rooms. Exit fields are usually null, but if the player can travel in the north direction from a room, there will be an exit to the north.

The Thinglist Class

The Thinglist is an array-list of Thing objects, such as the items visible in a room, or in the player's inventory. The find function hunts through a Thinglist to find an object with the given name, returning it if the object is found in the list and is visible. Item names are assumed to be either unique, or never visible from the same location.

The find method searches through the Thinglist for the first object with the given name, and returns it. The has method is similar, except it returns true on a successful find instead of the object: "Does this list have that item?"

The Exit Class

An exit has two fields called its sides, which contain references to the two rooms it connects; an exit is a connection between two rooms. An exit can be closed – if so, traveling through it fails, printing a given reason, such as "the door is shut" or "you cannot fly".

When the player travels through an exit, the travel() function checks which of the two sides they're on, and moves them to the other.

The Player Class

The player occupies a given room, stored in a player object. They also have a list of items they carry, called the inventory. When the player types a direction such as "north" or "south", the north method of the player object checks the player's location, then checks that location's north exit, then tries to travel through it.

The Game class contains a static reference to the current player object, which can be used to find what room the game is currently viewing.

The Time Class

The Time class contains methods for scheduling events to occur at some later point in time. A Time object has a time field, an integer that's incremented by every successful action, as well as the player simply typing "wait".

Each thing object has a tick() function. This typically does nothing, but if an item has an event it wants to schedule, that event is described in the tick method – such as the rain object filling a place with water, or the water object drowning the player if they spend too much time submerged.

The Time class contains a linked list called the schedule, a list of which thing's tick methods are going to activate, and on which turns. Order is maintained whenever a new item is added to it.

Whenever a method in the game wants to have some event happen at a later period in time, it calls its schedule(int t) method, which inserts a node for that item in the schedule to happen in t turns.

All successful actions increment a turn counter in the Time class. After every such increment, it checks the linked list to see if it's time for the first event to trigger. If it is, it removes the event, activates it, then checks again. It is possible for multiple events to be scheduled to happen on the same turn, so if there are multiple events with the same turn value, the schedule just activates each of them in turn.

Examples

The Parser and the Model - Moving the Player About

When the player travels through an exit, they type a command such as NORTH. Since this command only contains one verb, the Parser instantly routes it to Understand, like so:

In Parser

```
if(isVerb(nextWord)){Understand.understand(nextWord,object,secondary);
                valid=false;}
```

Understand then notes that it did not get any nouns, so it must be a direct command, which it interprets with a switch block:

In Understand

```
static void understand(String verb){
		switch(verb){
		case "north":
			Game.player().north();
			Game.getTime().increment();
			break;
```

The Understand utility of the parser reroutes this to the player.North() function, which checks the north exit of the current room and calls its travel function, if such an exit exists.

In Player

```
	public void north(){
		Exit destination = location.getNorth();
		if (destination!=null){destination.travel();}
		else{Io.out("At least from here, there's no way further north.");}
	}
```

In Exit

```
	//Moves the player from one side of the exit to the other side of the exit.
	public void travel(){
		if (open){
		if (Game.player().getLocation()==side1){side2.enter();}
		else if (Game.player().getLocation()==side2){side1.enter();}
		// if they are not on either side, that is not how exits work.
		}
		else{Io.out(closedReason);}
```

In Room

```
	public void enter(){
		Game.player().setLocation(this);
		describe(!visited);
		visited=true;
	}
```

Complicated Behaviour – The Torch

The included demo game, zDemo, contains a torch that can be lit using fire and extinguished using water. It serves as a good example of how to implement special Thing behaviour using Mislead.

The torch is an instance of class zTorch which inherits most of its behaviour from the base Thing class, but overwrites it in specific ways, mostly in places where being on fire matters.

For example, if you try to eat it, instead of the basic Thing message:

In Thing

```
	public void eat(Thing o){
		Io.out("Didn't anyone tell you not to eat strange objects?");
	}
```

You get:

In zTorch

```
	public void eat(){
		if(lit){Io.out("Fire-eaters do their tricks with low-temperature flames.
This isn't one of those.");}
		else{Io.out("You try a taste of the tar, and -- nope, not mustard.");}
	}
```

Note that the torch contains a field unique to its sub-class called “lit”, true if the torch is currently burning. To light the torch, we simply use it on an existing source of flame: There are two possible in the demo game, instances of zFirepit and zFirebox. We can use the torch to light the firebox, too, since it isn’t burning at the beginning of the game.

In zTorch

```
	public void use(Thing o){
		if (o instanceof zFirepit){
		light();
		}
		else if(o instanceof zFirebox){
			if (lit){
				((zFirebox) o).light();
			}
			else if(((zFirebox) o).isLit()){
				light();
			}
			else{

				Io.out("You could probably light this, if your torch was
burning.");
			}
		}
		else{
			Io.out("You rub a bit of the tar on the "+o.getName()+", to no
effect.");
		}
	}
```

Note the use of “instanceof” and typecasting – When the Understand class calls torch.use(o), it is only promising to torch that its argument is a thing. To better ascertain what exactly it is, we use an if statement containing “instanceof” to ensure that it is of a certain class.

Once we are certain of what the given object the player is using the torch on is, we can specify what subclass we're certain it belongs to using ((zFirebox) o) and safely access the methods of that subclass.

Of course, getting the torch to the firebox without it being extinguished is another matter.

Output

Here is an example output, wherein the player attempts to cross the basin with the lit torch, and fails. You have to swim across, you see.

A few days ago, a friend of yours doing research in the area complained to you- A logging company, ZexCo, doing some clear-cutting in the area stumbled upon an archaeological find- just from what you heard of the stonework alone, it's World Heritage Site material, and rumor is that it goes deeper still.

But it seems like they've been pouring a lot of effort into hushing it up. If the place ends up protected, they'll lose prime logging land.

So you're trespassing. If you can bring evidence back - serious handheld evidence - of the value of this place, UNESCO will come booming in and ZexCo will get booted out.

Well, I mean, you hope.

You clutch your camera and forge onwards through the brush...

Entryway

You had to buy a few people a few drinks to find this. Just off of a logging trail to the west, you can see a trapezoidal tunnel of black, carved stone stretching out to the east. The torrential rain roars a chorus as it hits along the canopy above and the ferns all around you.

i

camera

look tunnel

The engravings along the entrance to the east are purely geometrical - no people depicted, no animals, no objects, just lines and hexagons. Unusual for the area.

photograph tunnel

You line up and snap a good photo of the

tunnel.

east

Antechamber

A bath of warmth hits you as you descend into the earth, into a circular stone chamber bathed in the light of a roaring flame. A firepit fills the center of the room, set on a circular patch of dirt that the black stone tiles encircle. Torches line alcoves along the outer rim of the room - you consider taking one. The exit rises to the west.

A stairwell on the far side leads down.

look firepit

A tremendous smokeless flame burns in the firepit. The base of it is filled with a bluish tar with a mustardy scent. It was burning even before you got here - did someone light it?

get torch

Took torch

i

camera

torch

look torch

A length of wood coated in a strange bluish tar that smells of mustard, like that which fills the firepit.

use torch on firepit

The tar catches flame quickly.

look torch

A length of wood coated in a strange bluish tar that smells of mustard, like that which fills the firepit.The bluish tar on the end of this torch burns with an unabating, smokeless flame.

down

Descent

Down the stairwell from the Antechamber is a long, gently sloping tunnel eastward.

Water laps at the east edge of the passage.

east

Basin Side

A short stone ledge that overlooks a huge reservoir at the heart of the tunnel systems, a huge cylindrical basin wrought from dark

stone.Stairs rise to the west. The basin is to the east.

The water rises gently over the ledge, filling the basin.

east

Basin Center

The center of a vast stone basin that extends below and above you.

Water spans between two ledges to the west and east

You swim on the surface of the water.

The water puts out the flame of the torch with a hiss of steam - still no smoke.

east

Basin Side

A short stone ledge that overlooks a huge reservoir at the heart of the tunnel systems, a huge cylindrical basin wrought from dark stone.A short hall runs to the east. The basin is to the west.The remains of black pillars, pitted, pumice-like, stand here - three of them.

Even the largest fragments of the pillars that once stood here float and shift easily on the standing water.

Of four great pillars that stood here, one remains. Its squared black surface is aerated and pitted, its base almost worn away.

look torch

A length of wood coated in a strange bluish tar that smells of mustard, like that which fills the firepit.

Conclusion

Mislead successfully interprets player commands, given as plain text, into game actions. Though it is finicky about what commands it accepts, it is roughly on par with the engine used in the earliest of interactive fiction games, such as the Colossal Caves Adventure, or Zork I.

Its implementation of object behaviour, and its scheduling of events, are sufficient to create a simple game containing timed events and win and lose conditions. Such an example game, zDemo, is included.

Appendix I – Commands

Mislead understands these verbs:

Look, Get, Drop, Open, Close, Drink, Eat, Push, Pull, Hit, Use, Read, Go, Enter, Lock, Unlock, Use, Put and Photograph.

It also understands these special in-game commands:

North, South, East, West, Up, Down – Travels in given direction, if possible.

i – lists items in inventory.

wait – passes a turn without doing anything.

Interface commands are prefaced with a slash ('/'). They are as follows:

/quit, /exit – Closes the game.

/small, /normal, /large – Adjusts font size.

/style – Switches from a grey color scheme to a black color scheme, and vice versa.

Appendix II – zDemo

In zDemo, you play an intrepid explorer descending into a strange ancient ruin that a logging operation near your hometown has discovered, and has tried to hush up. If you can proceed into the ruin and bring back an impressive treasure – or a photograph of it – you can use that to get the ruin declared a world heritage site, forcing the clear-cutting to end.

I mean, you hope.

The goal of zDemo is thus to explore as far as possible into the ruin and reach the treasure at the end, then return from the trail you arrived by.

The ruin is an ancient reservoir. A machine can be operated, once powered, to open the roof and let rain pour in, which will fill the basin with water, allowing you to travel further.

A lit torch will be useful in the game, but figuring out how to keep it lit requires some effort on your part.

We hope you enjoy zDemo.