Wormwood - A Text-Adventure Game System

Allan Lavell
Alex Moriarty
Jacob Godin
Nick Pinto

April 8, 2009

Introduction

It's 1987: graphics haven't even been invented yet. You spend your nights sneaking into the local university computer lab to play text-adventure games on the latest hardware.

Now, it's 2009: You can play text adventures on your brand-new quad-core gaming PC. This is where Wormwood steps in.

To say that our work on Wormwood has had any one specific "goal" would be a bit of an over-simplification. Wormwood's overall goal is to make it easy for anyone to create their own text adventures, without putting any limits on how they use the engine. In the context of our work on Wormwood over the time period of the project, the main goal has been really "to get the basics working". This goal has largely been achieved, and in a way that should make it possible and maybe even easy to build upon in the future.

Project Management

Allan is the original creator of Wormwood. It was his idea to create the engine, and he took on the role of project leader. Jacob is the most familiar with Git, which is very useful source-control software, and he set up the repository that we use for wormwood. Allan and Jacob then acted as gatekeepers for the repository, and Nick and Alex sent their code to be checked and pushed to the repo if it was okay to include.

Structure of the Source Code

The code is divided into many separate classes, which are organized into packages. Here is a rundown of the packages:

Packages

- cmd: Contains all of the game commands. This directory is read by the parser to check to see if a particular command exists.
- core: Contains the core components of the Wormwood, including the Parser and Output handler.
- iface: Contains Interfaces to guide the creation of specific types of classes. Currently only contains the Command interface, which all commands must implement.
- obj: Contains basic building block classes of the system, such as Room, Player, Item, and so forth.
- game: Contains classes for building a game using Wormwood. Includes debugging features.

- doc: Contains documentation for the source code.

Descriptions of Classes

Game

The Game class has two important methods: prompt() and executeCommand(). prompt() gets the player's raw input from the terminal, and passes it over the Parser. The Parser returns the raw input as a Command object. executeCommand() takes this command object, and executes it by calling its exec method.

Parser

The Parser takes a raw String entered at the prompt. The general form for a command in Wormwood is *cmd_name arguments*. For example, the user might type "move e". The Parser splits this input up into the command and the arguments. It then uses Java's generic class constructors to dynamically create the Command class, based on whether or not it can find the corresponding cmd_name.java file in the cmd package. This simplifies things a lot, because it means that new commands can simply be added to the cmd directory without modifying the Parser to contain a list of available commands. It returns a generic Java Object containing the command.

Output

Output encapsulates printing to the screen. At this point, Output.println() simply calls System.out.println(), but it allows for greater flexibility in the future. If we were to decide to make a more complicated output system that did things like make sure words didn't get cut off when they reach the edge of the screen, but instead were pushed down to the next line, we would only have to change the code in Output.

Command

Command isn't exactly a class - it's an Interface. All commands written for Wormwood should implement the Command interface. It enforces a few basic methods that all commands must have to work properly: exec, construct, and toString. The details of the implementation are left up to the class writer, but this helps give the programmer an idea of what is required.

Entity

The base class for most of the objects in the game, Entity defines a few key methods and instance variables. Each entity has a name, a description (for when the entity is examined), a room description (what gets printed when the room is described to the player, i.e. "There's a chest in the corner of the room"), and an ArrayList of identifiers, which are names that can be used to identify the entity to commands. For example, an NPC (which extends the entity class) might have the room description "You see a man standing at a table in front of you." and the identifier "man". Then, when the player types "examine man", wormwood will reply with the man's description (e.g, "He's got a mean face, and blue jeans.").

Player

The Player class is an important class - it represents the player in the gameworld. It knows where the player currently is in the gameworld, and also attributes of the player. It contains the player's inventory, as well as their name, age, and gender. Basically the Player class keeps all the information that will be with the character wherever he/she goes. Think of it this way: the player will always be in a certain Room, so which room the player is in will always be known by the player class. The player will always have the same name, so that will also always be known by the Player class.

The Player class gets passed to the exec method of Commands by default, because it contains so much pertinent information.

Room

The basic building block in a gameworld created using Wormwood is a Room. Rooms inherit from the Entity class, mainly for the name and description variables and methods. Rooms are linked by Exits contained in the Grid (both of which are discussed elsewhere in this document). Rooms can contain items and NPC's, whose descriptions are given to the player along with the general description of the room.

Exit

Exits link rooms, and are member objects of the Grid (discussed below). Exits extend Entity, because they have names and descriptions that are displayed to the player. They also have two significant data members: passable and locked. If an Exit is passable, that means the player can use the Exit to go to wherever it points. If it is not passable, that means that the player can see and examine the Exit, but cannot go through. An example of an object that is not passable is a closed door. Using the open command, the player can make the Exit passable, if it isn't locked. If it is locked, it has to be unlocked before it can become passable.

Grid

The Grid is an attempt to link the whole gameworld together. It consists of a two-element Array of points where Rooms can be stored, and an ArrayList of Exits that link those Rooms together. The ultimate goal of the Grid is to make it easy to visualize where Rooms are in relation to each other in the gameworld. In the future, a GUI gamebuilder program could make use of the Grid to display to the game maker how everything interconnects.

Although not currently implemented, the idea is that there will be many Grids in a game, each containing their own "level" or "area". The programmer will then be able to link these Grids wherever and however they want to. Grids are a level of organization above the basic building block of the Room. They should help in cases like the following: the programmer is modeling a house with wormwood. They want to have a main floor, a basement, and a top floor. The main floor will have its own Grid, and be connected to the top floor, which is also on its own Grid. The basement Grid will be connected to the main floor. This allows the programmer to break things up into manageable sections.

Item

An Item is something that can be picked up by the player. It is then a part of the player's inventory, and can be used, or dropped, at will. "Using" an item merely invokes whatever command is associated with that particular item. For example, if you had an item pencil, then you would make a command "Write" that is associated with the pencil. Then, whenever the player "Uses" the pencil, "Write" is invoked. Some items have a limited number of uses.

Currently commands mixed with items is not completely fleshed out. Because the commands that are associated with items right now are simply normal commands in the cmd package, they can be used independtly of the items they are associated with, because the Parser automatically checks for everything in the cmd package. This must be changed. A solution would be to have an itemcmd package. This is most likely what will be implemented.

NPC

NPC's (Non-Player Characters) are right now basically just a direct extension of the Entity class. They don't have any special features implemented for them yet, but ultimately they will be the base class for characters in Wormwood. The player will be able to interact with them, potentially doing such things as speaking to them, trading/giving/receiving items with/from them, attacking them, being attacked by them, etc.

Create

Currently, the way to create a game is through the Create class. In the class' init() method, Grids, Entities, Rooms, etc., can all be defined by creating Java objects from their classes. This leads to a lot of manual labour, and will most definitely change. We are only now getting to the point where we have enough of the framework done to really define how games will be created using said framework. Some possibilities include: a GUI gamebuilder that automagically handles the creation of game entities in a neatly presented way; or writing a wormwood scripting language that simplifies the syntax for creating things in wormwood.

It is unfortunate that we haven't had more time to flesh this part of the system out, but as it stands, Wormwood works and it's manageable to create a game. We decided to make sure we had enough of the framework done so that we could actually *make* games before we prettied everything up. Optimization on an unstable base is not a good strategy.

Conclusion

Wormwood is a functional text-adventure game-creation system, and it has lots of room to grow. It already includes many core features, as well as nifty bits that fit together nicely. In its current state, a simple game can be written and played. The game-building tools are far from perfected, but that will come with time. The project has reached its goal of setting up a strong base, and the goal of easy-to-use game creation is in sight.