Research

Text Based Adventure Games

Text based adventure games are some of the earliest computer games to ever exist. The first of which was called ADVENT – short for Colossal Cave Adventure - which was written in the FORTRAN programming language by Will Crowther. ADVENT ran on the PDP-10 mainframe at Bolt, Beranek and Newman where Crowther worked. These games were designed to work on teletype terminals, which are remote input/output interfaces for mainframe computers being used in time sharing configurations. These types of games were written as novelties by early programmers primarily as a diversion. Likely due to their novelty, these programs were quickly shared amongst the cohort using the mainframe as Easter Egg like diversions to play with from time to time.

Since it's initial creation, ADVENT has inspired an entire genre of computer games, and it could be argued that it inspired computer role playing games in general. Zork, one of the most popular offshoots of ADVENT, was hugely influential in helping to define the genre as a whole. As their popularity increased, so did their complexity, as games like MUD (multi-user dungeon) were created, allowing for some of the first multiplayer experiences in computer games.

In the modern day, the text based adventure genre persists in the form of interactive fiction. These games often have a graphical user interface that shows the current game scene in two-dimensional graphics, while keeping the core user interaction restricted to typing out commands. The newer versions of text based adventure games allow for a much greater diversity of inputs, employing considerably more sophisticated natural language parsers, detailed worlds, and deeper stories to advance the genre. Although not a particularly mainstream genre of gaming, text based adventures maintain a small but cult following of devoted fans.

Parsing Input

The text parsers used in early versions of adventure games were fairly rudimentary, while still affording a great deal of leeway in user input. This allowed these games to gain popularity not only with technically proficient programmers, but also laypeople who were enamored with the prospect of interacting with a computer using everyday language.

While parsers were originally quite rudimentary, relying on limiting the scope of valid inputs on the user side to provide game functionality, more modern parsers employ much more robust solutions. One such solution is recursive descent parsing, in which a series of increasingly more specific functions call one another depending on what form the input data takes. There are large general functions that allow for classifying entire sentences, and then each of the blocks of the sentence are then passed to simpler and simpler methods that further distill the meaning of a given token until it reaches one of the languages terminal grammar objects. That is, a grammar object that is atomic, or no longer able to be reduced to any smaller part. These atomic tokens, which are far simpler to understand are then parsed for meaning before being reassembled into a full message.

For the purposes of a simple text based adventure, the more rudimentary solution of strictly defining what is a grammatical sentence and then comparing input against it is most appropriate. Natural

language processing is a deep field, worthy of it's own study, and as such can become exceedingly complex for simple tasks.

Handling Data

JSON files are highly organized text files that follow a strict format in order to allow for data to be accessed by a variety of different programming languages, in a variety of contexts. Standing for JavaScript Object Notation, it was originally developed for passing data over networks, particularly for web browsers. Since it's inception, it has been expanded to be used in any scenario where passing data from one program to another is important; often over networks, but also between two programs executing on the same computer.

In order to use JSON with the java programming language, one should use the java libraries developed for the task of parsing JSON files. This requires specific java packages to be employed in the creation of a program, as well as following the particularities of the JSON format itself. Using these libraries one may read from a JSON file, or save data to one, as necessary. They are an extremely useful tool for saving data in a persistent state outside of volatile memory.

Sources

https://en.wikipedia.org/wiki/Colossal_Cave_Adventure
https://gamedev.stackexchange.com/questions/27004/how-should-i-parse-user-input-in-a-text-adventure-game
https://www.geeksforgeeks.org/recursive-descent-parser/
https://gamedev.stackexchange.com/questions/27004/how-should-i-parse-user-input-in-a-text-adventure-game

Design

Classes

Game

Contains the core game loop, which calls other class core methods in order to generate the gameplay loop. This loop consists of the following steps:

1. Print the current game scene to standard output
2. Prompt the user for input, save this input
3. Pass the user input to the parser for parsing
4. Pass the parsed commands to the Control class for execution
5. Update relevant game variables
6. Repeat

Parse

Processes raw user input into commands intelligible by the game. By retrieving lists of valid game objects, and valid game verbs, along with object and verb synonyms, it classifies each word of user input into it's relevant category. After the word has been classified, it places it into a fixed-length string array and returns it in a particular order. The string array shall always be of the form { *verb*,

object, context } where the verb and object are legal game words, defined in the data files. The context word is used as a fail safe. When the user enters a word that is not a legal verb or game object, the parser may save this as a context word, in order for more graceful error messages to be printed. In this way, general error messages (e.g. “I do not know how to do that”) become specific (e.g. “There is no banana here to take”). This gives the illusion of the game having a more robust parser than it in fact possesses.

Control

Functions as a traffic controller, accepting as input the verb-object-context triplet, and executing the appropriate methods to take actions as necessary. These actions are determined primarily by the verb in the triplet; since these are action words, they cause actions to occur. By matching legal game verbs to a method, these verbs direct the program to execute a particular method. The action methods located here accept as parameters the object and context words from the triplet, dictating how they act. In this way, the parser → control → method chain mimics the structure of a recursive descent parser. Although there is no recursion in intuiting a command, each stage of this chain further distills the overall command down to it’s composite parts, acting on them in turn.

Location

Defines all of the essential properties of a location object. Includes the names, as well as all of the game objects and characters that are present in that location, with which the player may interact. Additionally handles mapping the game locations and linking them together in a persistent map, held as a static list of location objects.

Character

Defines all of the essential properties of a character object. Includes the name, description, associated events, and dialogue for a character.

Item

Defines all of the essential properties of an item object. Includes the item’s name, as well as it’s description and associated events.

Inventory

Essentially a wrapper for a list of items. Allows for other classes to maintain inventories of items which are searchable, and easy to add to and remove from. Provides a unified method of handling which items are where.

Direction

Declares and defines the Direction enum. Creates game objects for the cardinal directions, as well as up and down, in order to facilitate mapping locations to said directions.

Events

Facilitates creating custom game functionality through highly custom methods. Each of these methods are associated to objects, locations, characters, or game actions through their names. Possesses a central method similar to control to direct execution of the correct method based on the event name. Events are triggered via other actions, according to current game context.

Data Files

Data files are highly organized into a specific format, inspired by JSON files. Each data field is headed by it's name, followed by a space and then a curly brace. Within the curly brace are property fields which are delimited by semicolons. The structure is as follows:

<field> { <property1>;
<property2>;
…
<propertyN>;
}

Each property may be either a single word, or several words separated by a delimiter. This delimiter is typically a space, but may also be a forward slash, or a comma, or some other well-defined delimiter. This allows for a property to have several values in it at any given time, so long as they are properly separated. The parser will only ever delimit properties using semi-colons, leaving value separation with a property up to the class that is using that data.

Each field relies on highly ordered properties in order to give a given property meaning within it's field. Below are the location-dependent meanings of each data type's properties:

Locations

Index	Meaning – Delimiter
0	Description – none
1	Return description – none
2	Characters – Spaces
3	Items – Spaces
4	Events – Spaces
5	Exits – Spaces
6	Exit Status – Spaces

Characters

Index	Meaning – Delimiter
0	Description – none
1	Return Description – none
2	Dialogue – Forward Slash
3	Inventory – Spaces
4	Events – Spaces

Items

Index	Meaning – Delimiter
0	Description – None
1	Events – Spaces

By following the above schema for game data, the program is able to assign meaning to otherwise meaningless strings. Note: a delimiter of “none” simply means that the property does not require any further splitting.