CCPS 406 Project: Text Adventure

Ilkka Kokkarinen

Chang School of Continuing Education
Toronto Metropolitan University

This assignment is dedicated to Tim Hartnell (1951 – 1991), an unsung home computing pioneer whose book "Creating Adventure Games on Your Computer" inspired this instructor at the age of twelve to create his own text adventure games on ZX Spectrum Basic language, which is also why now, almost forty years later, my software engineering students get to hopefully experience the same thrill of coding an old school text adventure game.

Version of August 29, 2024

Interactive fiction

Written word has always been the universal interface of impersonal communication. Before the explosion of real-time computer graphics, old school **text adventure** games, later generalized into the interactive fiction genre, captured the hearts and minds of computer nerds. Text adventure games had their heyday in the eighties, heralded by classics such as The Hobbit and the Zork series, and inspired the genre of "choose your own adventure" books. The mighty one *mega*hertz power of the 8-bit home computers was sufficient to run such games, and the small memory space of 32 kilobytes or such inspired techniques to express the game and its data in a more compact fashion.

The simple yet infinitely extensible structure of text adventures also then inspired many budding programmers to implement their own adventure games. Commercially, the entire genre has long since passed on into the realm of retro-gaming, but there still exists a small but vibrant subculture of hobbyists who create and share their own games. The classic commercial games that are today old enough to have grandkids also contributed lingo to the colourful hacker jargon, such as the terms "grue" and "xyzzy".

Your instructor can still fondly remember writing his own text adventure game at the tender age of twelve, working on his little ZX Spectrum to encode the rooms and items as simple integer arrays, all the while simultaneously learning a whole bunch of English from the computer books found at the local library. Even if the material bits and bytes of that game have long since been recycled into other things on this mortal plane, its spirit carries on as the inspiration for this project.

The best way to get the idea of these games is to play them through an emulator. The site "Interactive Fiction" offers not only all the classics of Infocom, but the finest selection of hobbyist labours of love from our current millennium. Students looking for more information and inspiration about the history and conventions of this genre should check out the excellent articles at "The Digital Antiquarian" by Jimmy Maher. For game design in general, columns of Ernest Adams also offer many valuable insights, especially about what *not* to do when creating a game.

As an important side note of computer science history, text adventure games were a trailblazer with the revolutionary concept of using a **domain-specific mini-language** to describe the game world and its entities. Such high-level representation is compiled into bytecode representation and executed inside a virtual machine that acts as the game engine. Instead of having to port every game separately from scratch to different computers, the data files that define the game content can be played in every environment for which that virtual machine has been ported. Today we take the portability of all programs for granted in all our favourite high-level universal programming languages such as Java or Python whose compiled bytecode is executed inside such virtual machines! Such high-level representation of the game semantics also allows hobbyists to both "mod" existing games and create entire new games without programming skills needed to design and implement an entire domain language with its compiler and virtual machine.

Once some hobbyist put in the effort to port this virtual machine to JavaScript, in one swoop these classic text adventures of Infocom, Level 9 and lesser-known game designers of that era became playable inside your web browser. An even more general way to play these old computer games written for the antiquated 8-bit hardware is to use some emulator of the original computer hardware as a virtual machine to run the original machine code instructions directly. The latter technique even lets you execute *any* program that was written to the original machine, not just that particular text adventure running inside its own virtual machine.

Forget the turtles; reality consists of virtual machines all the way down! How sure are you fundamentally that the physical reality around you, or to be more precise, the model inside your head in your conscious experience of existence, is the lowest level of machinery that exists in the

reality stack? Which level of The Matrix does the current "you" actually live on? Think about the behaviour of your program in multiple semantic levels from the imaginary world depicted in the story down to the low-level details of your data structures. (If you manage to create a strange loop between these hierarchies, perhaps you could make a YouTube video about it.)

Gameplay

In its classic form, a text adventure game is leisurely **turn-based**. The gameworld is a **graph** structure whose **nodes** are called **rooms**, connected to each other with **edges** that correspond to compass directions. (The word "room" is used for a particular location even if the story takes place outdoors, or inside a cave.) The gameworld graph is conventionally displayed using boxes as nodes (see the example map of Zork II). At each turn, the game prints the description of what the player currently sees, and prompts the user to enter a command to describe what they want to do next.

This project **most definitely does not** expect you to create an entire virtual machine with a powerful natural language parser, along with a computationally universal mini-language used to describe the behaviour of the items and characters inside the gameworld. It is enough for your command parser to recognize simple commands of the form `"verb"` or `"verb object"`. The majority of actual games recognize a common core of important verbs such as `take`, `drop`, `inventory` and `use`, with occasional abbreviations to lessen the typing burden. The `object` of the command is some **item** in the game world, where each item has been given a unique name so that the parser can tell them apart.

The natural language parser in the classic Infocom games could handle complex sentences such as `"plant the pot plant into plant pot"` or `"tell the robot to move north"`. This is not that surprising once we learn that Infocom itself came out of the very same outfit that spearheaded the research in natural language processing in the seventies. (The influence of classic works of good old-fashioned artificial intelligence such as SHRDLU and ELIZA is pretty clear in the rear view mirror, not just in the parser itself but also in the mechanism that executes these user commands to update the state of the gameworld.)

The gameplay inside your project might look something like the snippet below. The commands entered by the player after the prompt are displayed in boldface for clarity. This little game mockup takes place in the standard high fantasy setting inspired by the works of J.R.R. Tolkien and your teenage game sessions of *Dungeons & Dragons*. However, your game may belong to any genre from science fiction through detective puzzles via survival horror into a mundane slice-of-life dramedy, or even be a postmodern mashup of multiple genres.

```
[Small room]
You are in a small room with a table. A sweet scent drifts in the air.
There is a book on the table. The south wall has a wooden door.
> inventory
You are carrying a small dagger and a potion.
Thorin enters from the south.
> get book
Sorry, I don't know the verb "get".
> take book
Taken.
> read book
The book is written in an old Elvish script that you can't understand.
Thorin sits down and sings about lapis lazuli.
```

```
> s
[Official room]
You are in a large round room with a high vaulted ceiling, a place
where important and far-reaching decisions are clearly made on a
regular basis. The stone walls are covered with elaborate tapestries
that depict the heroic deeds of kings of the ancient lore. To the
south is a large hallway. A wooden door leads north.
The goblin enters from the south.
> wield dagger
The dagger is now your weapon.
The goblin swings at you, but misses.
> hit goblin
You swing at the goblin, but miss.
The goblin hits you, causing 3 points of damage! Your health is 17/20.
> n
[Small room]
You see Thorin.
Thorin drops an amulet.
```

Each room conventionally has a **long** and **short description**, so that the long description is printed the first time that the player enters that room, or when the player explicitly requests to see it again with the command **`look`**. From the second time on, only the short name of the room is printed.

There is no requirement for your game to contain any elements of combat and mayhem, but the storyline can just as well consist of cozy puzzles. Your game can also be a mixture of cooperative and adversarial elements, so that the real "puzzle" is finding a way to herd the other characters to achieve some global goal that the player character could not possibly achieve alone.

Technical requirements

Implementation platform

The recommended programming languages in this project are Python and Java. Other reasonable programming languages are also allowed, subject to the prior approval of the instructor. However, the **design** of the program must be done in an **object-oriented fashion**, even if the actual **implementation** is then realized in a programming language without such mechanism.

The group must agree on some **official style guide** for their chosen programming language and the organization of code, including the comment style and conventions. The entire group must follow that style guide to the letter through the entire project source code. If several sufficiently official style guides exist out there, the group may freely choose any one of them.

The game must read the user commands from the **standard input**, and print everything to the **standard output**. This will also simplify your **system testing** by allowing you to pipe in the player commands from a text file. If your game contains elements of randomness, make sure to hardcode the **seed** value of your RNG during the development to make your tests repeatable.

Your group must implement the entire game from scratch, without using any existing utility code and text adventure functionality that is not already part of the standard library of your chosen

programming language. (The exception to this rule are the libraries needed to read and parse some standard data format of structured text, as below.)

Mandatory functionality

Before defining the functionality, let us be absolutely clear **what this game is not**.

- A "choose your adventure" game that consists of individual screens where the player must choose one of the explicitly offered moved to proceed to the next screen until the game is completed (for example, certain "dating sims") is **far too simple to be acceptable as a project for this course**. A text adventure must at least attempt to create an illusion of an open world and exploration, which this genre does not even try to fulfill.
- As fun and interesting as the roguelike genre can be, and would have surely also been an educational project on its own, **roguelike games are not text adventures**, and therefore are not acceptable as projects in this course. In professional software engineering, you are supposed to create the system according to the customer's requirements, not the system that you are itching to create for your own needs.

To clarify the first point: **at every step of the game, users must be able to enter any legal command as their next move. At no point in the game may there be a menu where the user makes a choice from the given set of alternatives to make their next move. Any game that violates this hard requirement will be rejected.**

You don't need to design and implement an entire gameworld with an entire epic storyline full of puzzles and dramatic twists in style of *Game of Thrones* that lead to an exciting final conclusion. However, to demonstrate that your game engine is genuinely **effective** in the sense that a full-sized text adventure game *could* in principle be implemented on it, your project must be at least large enough to fulfill all of the following complexity requirements:

- The command parser must recognize and handle **at least six different verbs**. The commands `N`, `E`, `W`, and `S` and such for moving between rooms do not count as verbs in this calculation. (As explained in the article "Parser games", the game design should tacitly make it evident to the player which verbs are meaningful in your game, instead of reducing the players to blind guesswork of typing in possible words from a dictionary.)
- The gameworld must contain **at least eight separate rooms**.
- The gameworld must contain **at least ten separate passive items** that the characters can pick up and use to make meaningful changes to the gameworld. For example, a key that opens a chest so that the player can take out items from the chest and put them in, or a blindfold whose wearing makes the player not see the room descriptions and what the other characters are doing there.
- To score more than 15/20 points for the final presentation, the game must have at least two **non-player characters** (NPC's) that move and act in the gameworld under the same rules as the player character. Statues that sit in one room and allow players to talk to them in menu based interaction do not count as NPC's.

Every character, both the player and the NPC's, gets to act once per turn, even if they are not presently in the same room as the player character. Their actions also affect the gameworld even if the player is not present in the same room to observe those actions. When it is the player's turn to act, the game prompts the user to enter the command that determines the next action that the player performs inside the game world. When it is some NPC's turn to act, some simple form of AI, most likely some kind of **state machine** whose states represent the current mood and the state of mind of that NPC, determines the action taken. Thanks to the famous Tamagotchi effect, characters

that are fundamentally nothing but almost ridiculously simple state machines may still be externally perceived as conscious personalities that have goals, beliefs, emotions and plans.

In all levels of life, magic tricks always turn out to be trivial once you get to watch them from the backstage, instead of sitting with the audience to see what they see, filtered by the fourth wall. More generally, the author has found over and over again in his life that seemingly complex things generally turn out simpler than originally assumed. The flip side of this coin is that seemingly simple things will generally turn out to be more complex than initially assumed, especially if you actually have to implement them mechanistically so that they will keep working even after you are no longer present to energetically wave your hands around. (See the classic post "The Door Problem" for a good illustration of this principle.)

Good software engineering

Since the "skin-contained ego" represented by the bunch of zeros and ones that we for the moment refer to as the "player character" is fundamentally no different from any of the other characters or other items inside the virtual gameworld (other than being the proverbial hero of his own story, and therefore necessarily the villain in somebody else's who is trapped inside the same zero-sum existence), there should be no reason whatsoever for the program code that executes some particular game action such as picking up an item would be implemented in separate functions for the player character and the NPC's. Keep the fundamental design principle of "Don't repeat yourself" firmly in your mind. Cast a cold eye on your source code regularly, and refactor mercilessly to cut out duplicate statements, functions and classes!

Another excellent general design principle, the zero-one-infinity rule, also turns out to be highly relevant in this context. Once your game engine allows the existence of two separate characters who are not joined at the hip to always act in lockstep, **absolutely nothing should change in the core implementation of this engine itself** when the third, the fourth and even the hundredth additional character joins the ensemble cast!

Even when the individual state machines that implement their behaviours and chosen actions are relatively simple, unpredictably chaotic and complex interactions and causal chains of drama can spontaneously erupt from the interplay of the actions of these characters inside the simulated gameworld. With the help of some controlled randomness, every game session can be a brand new and different experience, in spirit of the "roguelike" adventure games even when the gameplay itself is purely textual. The most famous example of unintentionally complex and therefore also more delightful emergent gameplay in text adventures is surely *The Hobbit*, as explained in the articles "The Hobbit" and "The Hobbit Redux".

Since this is not a creative writing course (although at some point, quality does indeed have a quantity of its own), you are allowed to ~~swipe~~ adapt settings, themes, ideas and puzzles from existing text adventures and other games, provided that you properly attribute these ideas to their original sources. You can look for inspiration and ideas in articles such as "Ten great adventure-game puzzles".

Separation of code and data

Following the principles of data-driven programming, the logic of the game engine will be implemented in your chosen programming language. You may not use existing text adventure engines or utility libraries for this purpose, since creating a design for the customer requirements from scratch and turning it into an executable implementation is the whole point of this course! However, you are not allowed to hardcode the entire gameworld into the code itself, but the data

that describes the structure of the gameworld must be read from one or more external data files. These data files must consist of raw text without any binary data, for reasons explained in the section "The Importance of Being Textual" of "The Art of Unix Programming".

The more of the description and functionality of the game world you are able to lift from code to data, the higher mark your overall design will get, but also your later development process will certainly become correspondingly easier. Even if you don't aim high, all the way up to the level of purity espoused in the article "Data-Driven Design", you should still try to move as much complexity from code to data as you can anyway, not just in this project but also in many others.

You may use some open standard data format such as YAML or JSON to encode the gameworld information as structured text. You may use external libraries to read in and parse that standard data format, if your language of choice does not already offer that functionality in its standard library. The precise structure and content of these text files is left for you to design, but this structure must be sufficiently clear and straightforward that anybody could in principle edit and extend it to create new rooms, items and other content into your game. The article "Zork on the PDP-10" may give you some ideas about how to do this.

Again, this project does not expect you to reach anywhere close to the complexity level of the Infocom parser and its game engine. These days we have more than enough memory to keep the data of the entire adventure game in memory. The article "ZIL and the Z-machine" is an enjoyable read about how these kinds of issues were hand-hacked back in the day, uphill both ways rain or shine without complaining! (That title also sounds like it could also be a good name for some dad rock band or a "nu-swing" orchestra, assuming that genre is now a thing. I am too old to know what kind of things your generation has come up with in your Tikkity Tok videos...)

Nice-to-have functionality

Any fool can create (and many do) an *ad hoc* design that fits the current needs like a glove, but cannot be extended and adapted in any systematic fashion to fit tomorrow's needs that do not jibe with the assumptions behind the original spaghetti design. However, by following proper design principles of SOLID and GRASP, your concepts and things will connect together in code in natural harmony and unity in the eternal flow of Tao, requiring no further effort from your part. Analogous to drinking a glass of water on a hot summer day, you don't need to be consciously aware of what exactly causes these benefits to realize, and you still get to enjoy those benefits just the same!

(Also, as the famous expression goes, a private who tries to act like a general will not only make a poor general, but a poor private. The chain of command exists for a good reason, to manage the chaotic fog of war that no prepared plan ever survives the initial contact with. In the immortal words of Mike Tyson, everyone has a plan until they get punched in the mouth.)

The following extra functionality is not required, but will improve your project mark. You should get all the mandatory parts working properly before spending too much time with these additional features. On the other hand, proper design from the get-go will happily provide you at least some of these additional features for essentially free, thus presenting a vivid testimony to the flexibility and extensibility of your design.

- Add some non-player characters to the story and the gameworld of your game so that these NPC's act and modify the gameworld in some meaningful fashion.
- When describing a room and the items and characters that it contains, pretty print everything as a coherent paragraph (as if the whole thing had been written by a human author) instead of printing the room description followed by the items and characters as a

list of individual words. Using proper articles and other such details of grammar gives a good professional impression.

- Printing `"You see Thorin, the goblin and Nathan Detroit."` as one message sounds much more natural compared to printing `"You see Thorin."` `"You see the goblin."` and `"You see Nathan Detroit."` as three separate messages, each message on its separate line.
- Perhaps your room description data text could allow special placeholders (akin to `%s` and other format codes in `printf`) that the game engine then dynamically fills in with names and such to describe some particular part of the current moment.
- Make sure that your paragraphs of output are neatly printed within the assumed line width so that the lines won't look all ragged and such, whether a placeholder in some sentence is filled with the value `"book"` or with the value `"revised edition of the collected works of Aloysius T. Snuffleupagus, with commentary by professor P. P. Weisenheimer, Esq."` It would be nice if your printing logic worked for arbitrary line lengths, formatting the output into neat paragraphs regardless of whether the lines are capped to 60, 80 or however many *n* characters.
- The game allows the player to save a snapshot of the game session, so that the session can later be restored to the exact same game state to play from the second time.
- Offer some kind of "God's eye" verbose mode that lets the player see everything that is going on in the gameworld, probably with some additional normally unseen state data. Combined with a cheat mode that allows you to directly access and mutate the underlying data structures, having this feature will also make debugging your code easier.
- Grant the player the ability to temporarily control other characters and see through their eyes, perhaps via some kind of mind control spell in a fantasy setting. This could almost follow as an afterthought from a proper design to begin with that makes no distinction between the player character and the NPC's.
- Special items that temporarily affect the character's abilities within the game, such as a hallucination potion, strength increasing potion, drunkenness potion, et cetera.
- Design your data structures to allow arbitrary containment of container items and characters inside each other. Can your data representation distinguish between a gem being inside the room, that gem being inside a bag inside the room, and that gem being inside the bag inside the chest in the stomach of a whale? Perhaps even rooms are special kinds of containers that can hold not just items but arbitrary characters inside them?
- Could your rooms contain other rooms? Could items such as some magic snow globe contain rooms inside them, making Keanu really go "whoa"? In fact, once your design allows some items to be **containers**, isn't a "room" merely a container that can hold other items and characters inside it, and allow special actions that make a character move from one room to another? Is a character just a special case of a container that can perform actions and move around? It's up to you to decide how these concepts relate to each other, and design a class hierarchy to reflect your decisions.
- Finally, does the gameplay that emerges from the chaotic combinations of actions of NPC's ever surprise your group? That is, does your game cross the extremely high threshold of the Lovelace test? (Just to be clear, we are not expecting, let alone requiring, anybody to achieve that feat. But if you do, make sure to document it somehow so that others can also see it.)

Project timeline

To ensure that your group encounters the issues of communication and work allocation of a real world software engineering project, this project must be created in a **group of three or four** students. Zero, one and two are simple numbers, but three is the magic number when complexity emerges not just in theoretical computer science, but in all walks of life. Exceptions to this rule are allowed only in exceptional circumstances. (At least in this project you get to choose the people you work with, a luxury that is rarely granted to our kind.) All members of the group will receive the same marks for each component of the project.

All project materials, both code and documents, **must be kept in a GitHub repository** shared by the members of the group. This repository must contain the full version history of every text document, source code file and data file involved in the project.

The timeline of the project stages during the semester is given in the table below. Certain documents are due that particular week. To keep you disciplined with this project, **any documents submitted after the deadline will automatically receive a zero mark**. Final presentation slots will be scheduled during the lectures so that each group gets to proudly show the other students what it has achieved. These presentations should use slides and other presentation aids to supplement the narration, as if reporting to the big cheese whose John Hancock ultimately appears in your pay checks. Focus your presentation on the important aspects of the project.

Week	Scheduled activity	Mark
1	Organize the group, agree on the roles and responsibilities. Set up the GitHub repository used by the project, and settle on the programming language and the coding conventions that your group will uniformly follow.	-
2	Within the group, brainstorm ideas for the general setting and storyline of the game. What are the characters and their motivations and goals in the story? What are the dramatic situations that they encounter? What kind of items exist in the game and what can the characters do with them?	-
3	Write the general storyline document that describes the gameworld and its theme, along with the goal of the player and summary of what is required to reach that goal through the obstacles in the game. This document should also include at least a first draft of the gameworld map, if only to nudge you to practice using the tools that you will later create UML diagrams with.	5
4	Design the structure of the textual data files used in this program, specifying how the program messages, rooms, items and characters of the game are represented inside these files. Clearly delineate the dynamic aspects of the game expressed in these data files from its static rules written into the source code. The more functionality and behaviour you can lift from code to data, the higher your mark from this part will be.	5

5	Create an architecture diagram for the entire system, separating the parts of the system and outlining their responsibilities and connections to each other.	5
6	Create a UML class diagram to document the classes used in the program. Start implementing these classes and the methods in them, along with their associated unit tests.	5
7	Create a UML state diagram to fully document the behaviour and possible states of one particular NPC in the game. Continue working on the implementation and unit tests.	5
8 & 9	Continue to code, test, debug, refactor, rinse and repeat.	-
10	Document the design patterns that you used in this project, along with the anti-patterns that you encountered and the refactoring that scrubbed your design clean of any such code smells. Complete the alpha version of codebase.	5
11	Demonstration of the project from the user's point of view, highlighting the storyline and the mechanics of the gameworld, as these would be observed by the player who has no access to the source code.	10
12	Demonstration of the project from the programmer's point of view, highlighting interesting architecture and implementation details in the source code and structure of the gameworld data files.	20
	Total project mark	**60**