CS467-400
Thomas Buteau
Victor Encarnacion
Robert Francis Scanlon
6-7-18

Reticulum Final Report

Introduction

The Reticulum team created an Interactive Fiction game similar to Zork and Colossal Cave Adventure. The setting of the game is an alien space station and the objective is to 'escape' from the station where you have been held captive by an alien race. The only way to escape is to make it to the Escape Pod (one of the 'Rooms' implemented in the game). However, just making it to the Escape Pod room is not enough. The player must bring the necessary items to the Escape Pod to ensure their successful journey back home. The player must also figure out how to release the Escape Pod from the station or they will not be able to leave.

The user has to navigate through a total of 16 different locations on the space station. They will be expected to analyze the unique features of the rooms as they search for 8 objects that will be used to aid in their escape. The game will be played on the command line and the user will navigate the space station by entering commands such as "go north", "look at map", and "take key".

Our team spent a long time on the project plan and we were fortunate to have completed most of the project with very few surprises. We decided to make minor changes on how we implemented certain aspects of the game, but the overall design did not need to be altered.

Setup & Usage

Starting The Game:

- Copy spaceship_escape folder to flip
- $ cd spaceship_escape
- $ ./run (A minimum terminal window size of 80 X 40 is required)
- Enter "New game" at prompt

Instructions:

The first screen the player sees is menu as shown in Figure 1.1. If the player chooses "New game", the game will begin and the player will have to find their way off of the spaceship. "Load game" allows the player to load a previously saved game. "Walkthrough" displays a list of steps that will lead to one of the game endings. "Exit" simply quits the game.

Every time a player enters a new room for the first time, they will be presented a variety of details about the room. Figure 1.2 shows a typical screen. At the top of the screen is the name of the room. Below the name is a list of adjacent rooms that the player can travel to. Then

the long-form description of the room is shown. Finally, there is a prompt for the player to enter their text-based command.

- ➢ Tip: type “help” to display a list of available commands
- ➢ Tip: type “walkthrough” to display a cheat sheet.

Figure 1.1

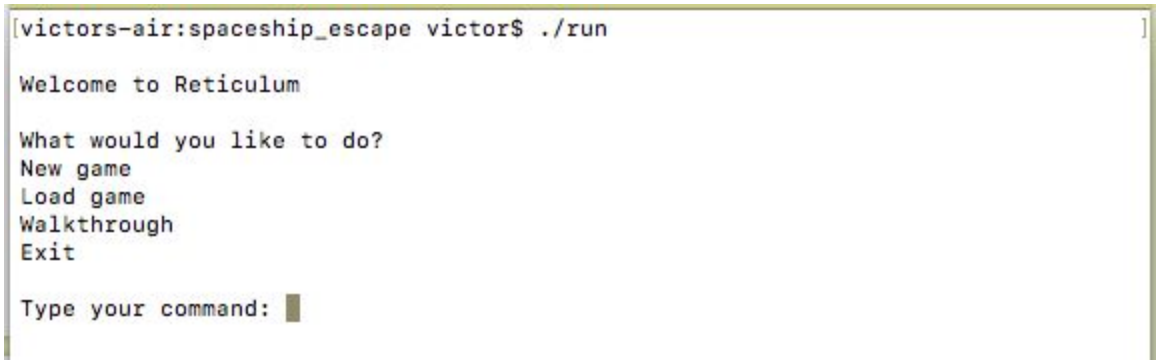
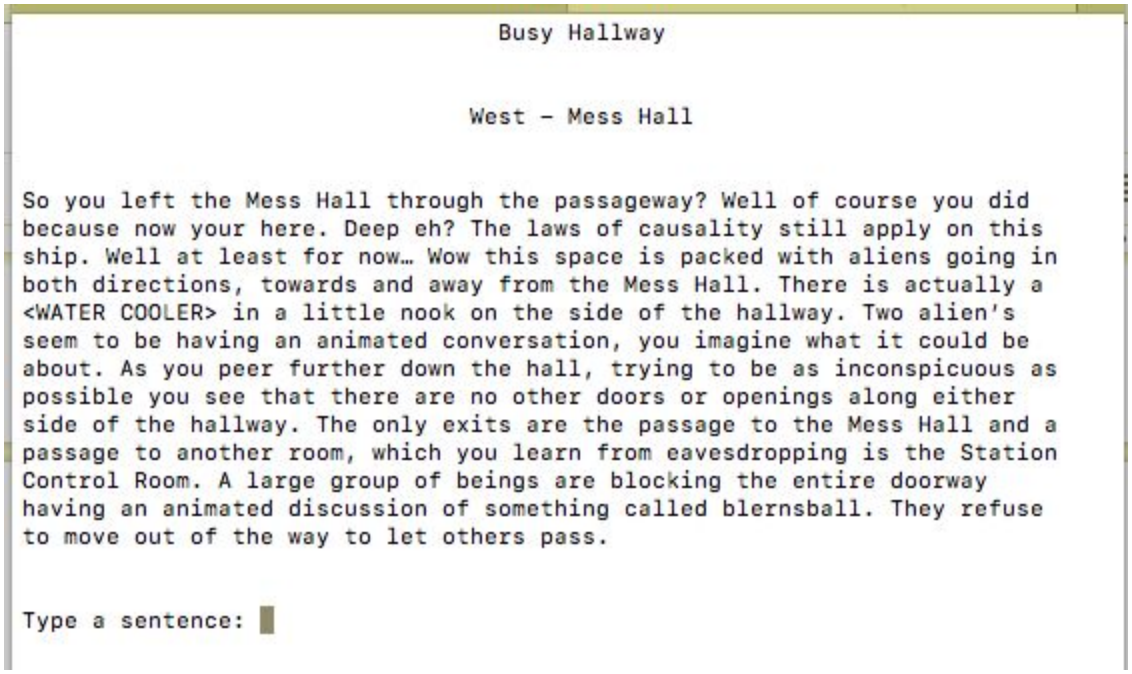


Figure 1.2



Cheat Sheet:

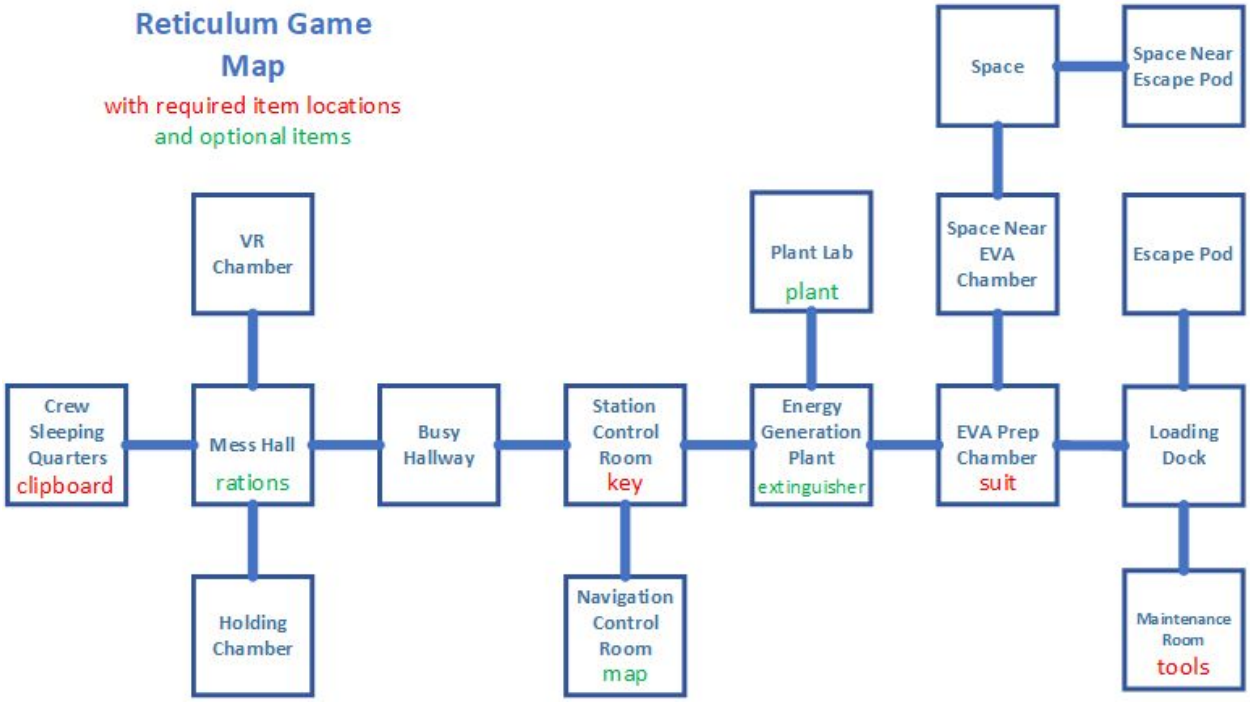
The game requires collecting a variety of items and paying attention to in-game prompts to complete the goal of escaping the spaceship. Below is a walkthrough that allows the player to finish the game. Use the map in Figure 1.3 to help keep track of your location.

1. Collect rations

    a. Go North to Mess Hall
    b. Take rations
2. Take clipboard
    a. Go West to Crew Sleeping Quarters
    b. Take clipboard
3. Use clipboard
    a. Go East to Mess Hall
    b. Go East to Busy Hallway
    c. Use clipboard
4. Take key
    a. Go East to Station Control Room
    b. Take key
5. Take map
    a. Go South to Navigation Control Room
    b. Take map
6. Take suit
    a. Go North to Station Control Room
    b. Go East to Energy Generation Plant
    c. Go East to EVA Prep Chamber
    d. Take suit
7. Take tools
    a. Go East to Loading Dock
    b. Go South to Maintenance Room
    c. Take tools
8. Use tools
    a. Go North to Loading Dock
    b. Go West to EVA Prep Chamber
    c. Go North to Space Near EVA Chamber
    d. Go North to Space
    e. Go East Space Near Escape Pod
    f. Use tools
9. Use key and ship
    a. Go West to Space
    b. Go South to Space Near EVA Chamber
    c. Go South to EVA Prep Chamber
    d. Go East to Loading Dock
    e. Go North to Escape Pod
    f. Use key
    g. Use ship to complete the game

➢ Tip: try finishing the game with different items in your inventory to see different endings!

Figure 1.3



Additional Tools, APIs, Libraries, etc.

Game Engine:

For this project, the general flow of the program is that a Game object is created and this object will act as the main control. The Game object has several members which keep track of the game state including the player's inventory, the coordinates of the player's current location, a list of end-game flags which determine when the end game will trigger along with what ending the player will receive, a copy of the currently occupied room, and finally a dictionary of the rooms in the world map. In this world map dictionary, each key is a 2d grid location coordinate and the value paired with said key is a dictionary of all the mutable values for that room, such as its inventory, and some non-mutable values, such as its name. The first time a player reaches a room that room's location in the map is populated with data from the corresponding room's JSON file. This way we have the ease of manipulating JSON files instead of raw code to modify the rooms while also maintaining replayability because we are not modifying the JSONs during gameplay.

The Game object also stores a Room object for the room the player is currently occupying. This Room object reads in from its file each time the player travels or a game is loaded. When a player travels to a room, the visited flag for that room in the map array is triggered. If the room had not previously been visited, the inventory of the Room object will be copied into the inventory at that location in the map array. The long description from the Room object will also be written to the screen. If the visited flag had previously been set, then the short

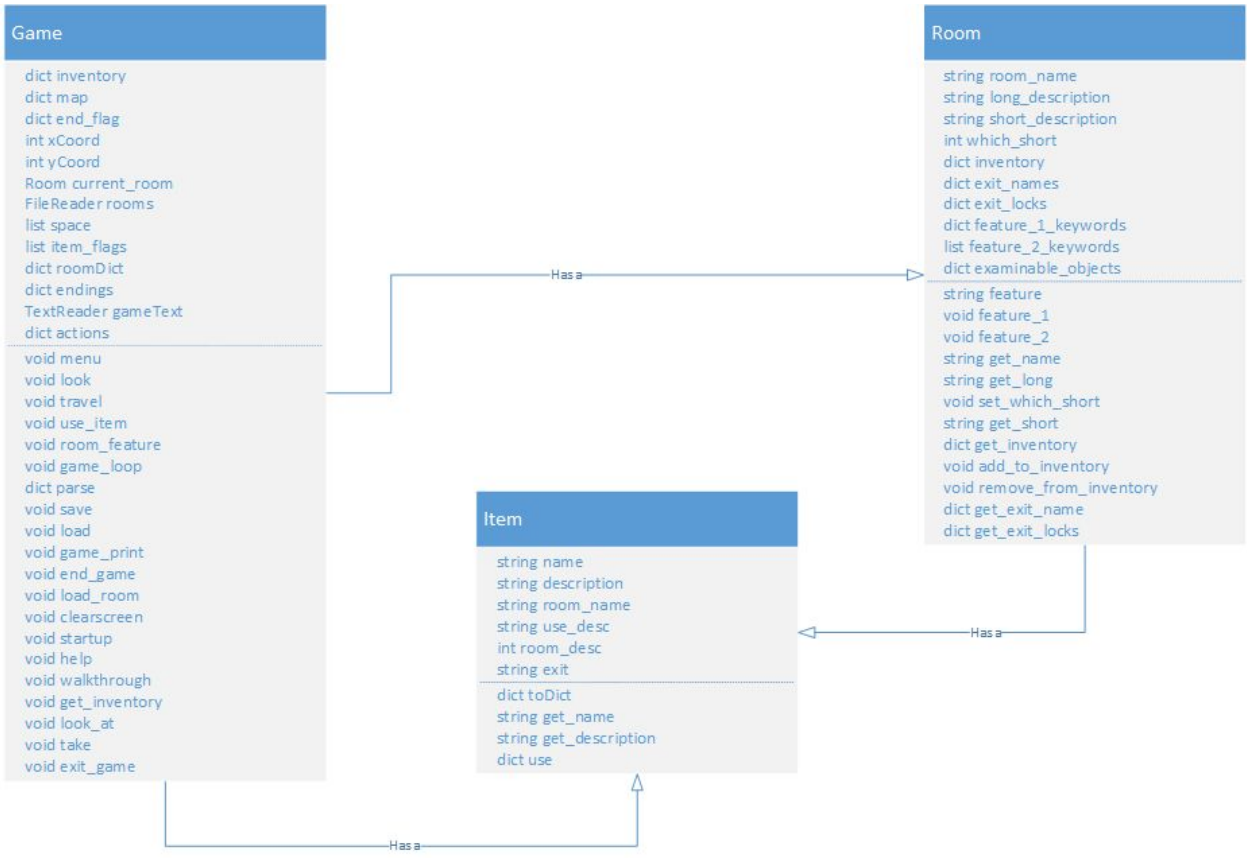
description determined from that room's map data will be used and the inventory in the map array will not be changed. This design choice was made to preserve the default configuration of each room between playthroughs. It also makes it possible for player progress to be saved. Additionally, this meets the design requirement of loading each room from a file.

The Game class has methods associated with the various player actions (look, use, take, move, help, etc.). There are also methods for saving and loading a game, parsing user input, and a menu function. In addition to these methods that the user will interact with, there are methods to load a new room, clear the screen, print formatted data to the screen, start up the game, and end the game. The most complex method, parse, will be discussed in greater detail later in the report.

Each Room object contains its name, a dictionary of different descriptions to be used after certain events transpire, an inventory, a dictionary of the name of the connected rooms, a dictionary of flags to denote that the player cannot currently use a particular exit, and a dictionary of features for the room.

Each Item object contains its name, its description, the name of the room it is to be used in, the description of its use, the number of the room description to be displayed after the its use, as well as the room exit that is unlocked by its use.

Below is a UML diagram of the classes used in this project.



File I/O Engine:

Apart from the program logic, for example, the text parsing engine and program code to support the general flow of gameplay, most data is loaded during game initialization from files residing outside of the program code. As per the program requirements, each ‘room’ is defined in a unique data file. A room’s data file contains all of the text information which is presented to the player. For example, the short and long form room descriptions, as well as any optional messages, are found in each room’s data file.

In addition to textual room descriptions, each rooms data file contains information about the items which can be found in said room. Information about how the placement of these items within a specific room may affect gameplay is also found. For example, a key placed in the Holding Chamber is used to get to the final escape pod. Features, specific to each room, for example, how the gameplay is altered by the player being in a room, is encoded in these files as well. For example, being in the VR chamber disorients the player. Finally, these ‘game room’ data files are used to define the topographic structure of the game, that is, how all the rooms are connected. By defining the exits which exist for each room in the room’s data file, the structure of the physical gamespace is defined.

By defining all this information in data files, the characteristics of a room are encapsulated in one location and are not scattered throughout other program code. From a development and maintenance perspective, this structure is beneficial as it allows for a clear separation of the program’s logic and data. Therefore, room parameters such as a room’s textual descriptions, default items, or unique features can be altered by simply editing a room’s data file *without* having to alter any program code, thereby greatly reducing the risk of inadvertently introducing bugs into the program code.

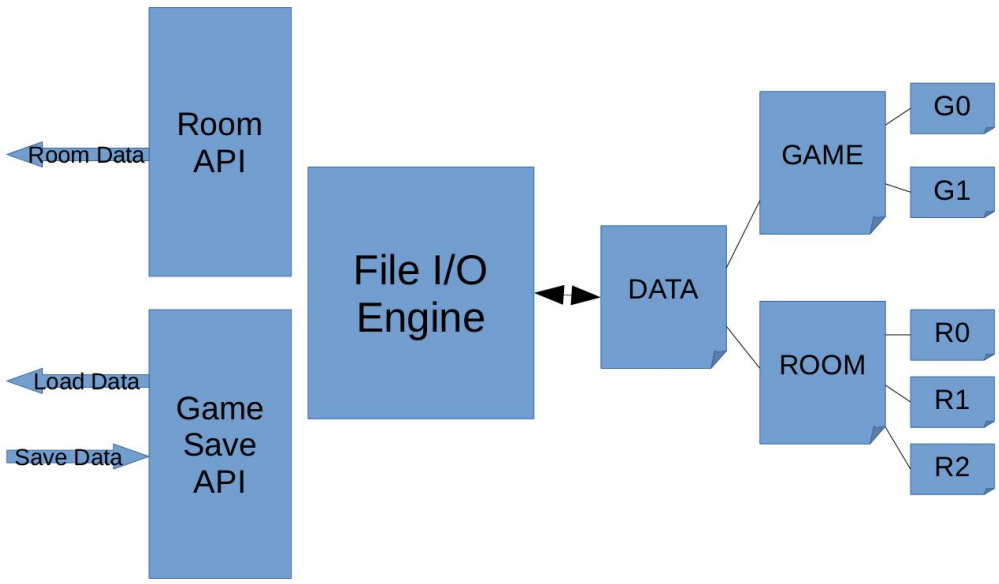
One of the most important features of the file I/O system of the program is the interface provided to the rest of the games code. While packed with necessary data, the room data files would be useless without the ability to be loaded, parsed, and provide the rest of the program with the necessary information. This task is provided by the file I/O engine and interface. As with the rest of the program, these features are implemented using the Python 3 programming language and its standard libraries. Low level file reading and writing is handled by Python’s default file read and write operations, which themselves interface with the C Standard Library to provide access to operating system features. Additional custom code is built to augment the default Python functions to make the task of reading the data files simpler. Finally, a defined API consisting of a set of functions, is provided by the file I/O engine to the rest of the game to allow for room data to be efficiently passed along to the other game modules.

Although there are almost endless ways the room data may be encoded within the data files, the game development team has decided to format all data files in JSON format. This format allows the data files to exists in a human readable format which aids in both debugging and manipulation of each room’s data parameters. Additionally, the use of JSON format allows for the use of Python 3’s versatile json module. This module allows for JSON encoded data to be loaded directly into Python data structures such as dictionaries and lists, simplifying and speeding up the development process immensely.

In addition to reading ‘room’ data files, the file I/O engine is also responsible for reading and writing ‘saved game state’ files. When a player chooses to save their current game, the

current state of the game is passed to the file I/O engine, which writes the necessary data to a file stored in a defined directory. Similarly, when a player wishes to load a previously saved game, the file I/O engine is called to read from an existing file and pass the data back to the game engine. As with the room data files, all saved game data is stored in JSON format.

Below is a high-level diagram of the File I/O Engine.



Text Parsing Engine:

The parser is a separate module that analyzes the user's input in order to perform some action. The parser tokenizes the user's command into a python list of individual words. The parser then removes common stop words ("and", "the", "a", etc.) which have no impact on the functionality. The parser then finds the action verb in the list. The verb is used as a key to find an associated function in a dictionary. The functions is called with the python dictionary containing a variety useful objects such as a word list and the game state.

The parser has some additional capabilities, as well. First, the parser is case insensitive. All input is automatically converted to lowercase before being tokenized. Second, the parser is tolerant of some spelling errors. Minor errors such as transposed letters, missing letters, or words that off by one letter are automatically corrected. The following verbs will be used in the game.

Verb	Function
• look	give long form explanation of current room
• look at <feature or object> • examine <feature or object>	describe feature or object

• go <direction> • go <room> • <direction> • <room> • travel <direction> • travel <room>	go to indicated room
• take <object> • pick up <object> • grab <object> • drop <object>	place object in inventory or room
• use <object> • try <object>	use object for an effect
• consume <object>	eat meal
• help	list available verbs
• inventory	list contents of inventory
• savegame	save state of game
• loadgame	load state of game
• exit	exits the game

Game Design:

The game is set on a space station staffed by non-human lifeforms. The player is a human who was shipped here erroneously due to a mix up in a shipping manifest and is trying to find a way to get back to Earth.

The game is more silly than serious (think Space Quest but not as well written). The current layout of the station is shown on the map below with the player starting in the Holding Chamber and needing to gather several items on their way to the Escape Pod to successfully leave the station and navigate back to Earth. There are multiple ways the game can end including a few interesting player deaths. For example, leaving the EVA Chamber and going out into space without putting on the space suit first is generally considered a bad idea.

Software Libraries:

- Python 3
- JSON Python library

Development Tools:

- Notepad++

- VIM text editor
- Flip (Testing environment)

Teammate Contributions

The development effort was divided among the three team member in the following way. Thomas Buteau was responsible for the development of the primary game engine which contains the core program logic and allows gameplay to take place. Robert Scanlon was responsible for the development of the file I/O engine and the data format of all required data files. This allows for the loading of ‘room’ files as well as the game ‘save’ and ‘load’ features. Victor Encarnacion was the lead developer of the command parsing engine, which allows for the player to interact with the game in an intuitive and efficient manner.

Deviations from Original Plan

During development there were some changes made to the design engine. These were largely due to initially being unfamiliar with Python and building the design based on experience with how C and C++ operate. Thankfully these changes were relatively small and mostly consisted of replacing structs with a series of dictionaries. In general, what we found was that our original plan was very solid and helped guide us during the entirety of the project. As development went on some new features were added as the functional realities of the design and design requirements became better understood. As an example, we figured that the best way to associate verb commands with the their functions was to create a dictionary whose values were the functions. Another example is that we took advantage of the already existing JSON data parser to improve the code structure and avoid having to hardcode some of the in-game instructions.

Conclusion

The Reticulum team designed and implemented an entertaining adventure game wholly from scratch utilizing Python 3 in order to grow and demonstrate our abilities as software developers. Although each developer was assigned a primary area of responsibility, each team member remained available to assist every other member of development in any area of the game. Specifically, the complexity present in the ‘natural language processing’ feature of the command parsing engine required work from all team members.

Now that we have concluded the project, the Reticulum team developed an engaging Adventurer-Style command-line game which will hopefully be enjoyed by all those with an opportunity to play. Each team member gained valuable skills, including a deeper understanding of the Python programming language, the software development process, the creative process

of designing a virtual word, and most importantly, the work and commitment required to work effectively with a team of other developers.

Code References

http://textblob.readthedocs.io/en/dev/quickstart.html
https://norvig.com/spell-correct.html