CS 467

Spring 2017

Auriga: Final Report

Team Members

Gregory Fernandez

Jason Goldfine-Middleton

David Moon

Introduction

Throughout the term, our Capstone Project team has been working on a command-line based adventure game called Auriga in which the user plays as a robot, Auriga-7B, who powers on mysteriously and “wakes up” in a robotics facility and must explore it to find out what happened to itself. Throughout the adventure, Auriga-7B meets and interacts with other robots as well as some of the facility’s organic employees. The game allows the player to move through spaces, gathering clues and objects needed to interact with the game world, traverse through it and learn a deeper truth behind the Auriga facility and its robotic inhabitants.

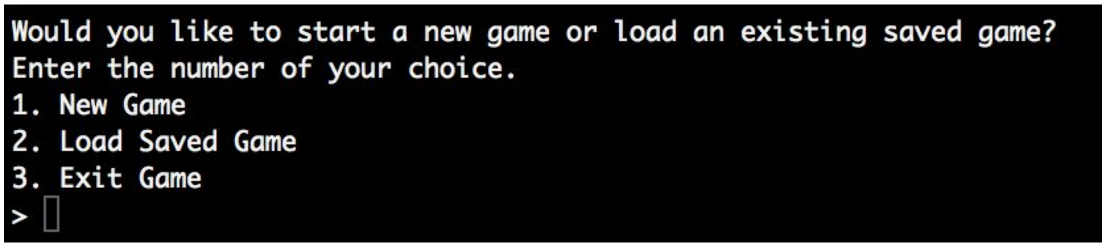
The game is in the style of classic text-based adventure games like “Adventure” and “Zork”. Like those games, Auriga features a list of commands that can be input by the user to interact with the game world, its objects and characters. As the player moves from room to room, the interface describes their location, and any items or characters there. The engine and parser facilitate interaction between the player, the items (both in the world and in the player’s inventory) and the characters in the various rooms in ways that are engaging.

Auriga was coded entirely in Python and is the culmination of a whole term’s worth of hard work by team members David Moon, Jason Goldfine-Middleton and Gregory Fernandez. The engine’s main components were developed in their entirety by David Moon. Jason Goldine-Middleton worked on the thorough and sophisticated parser. Gregory Fernandez handled tests for the engine and took on some of the project management for the group. We used doxygen to generate documentation for our project, and to aid in inter-team communication.

Setup and Usage

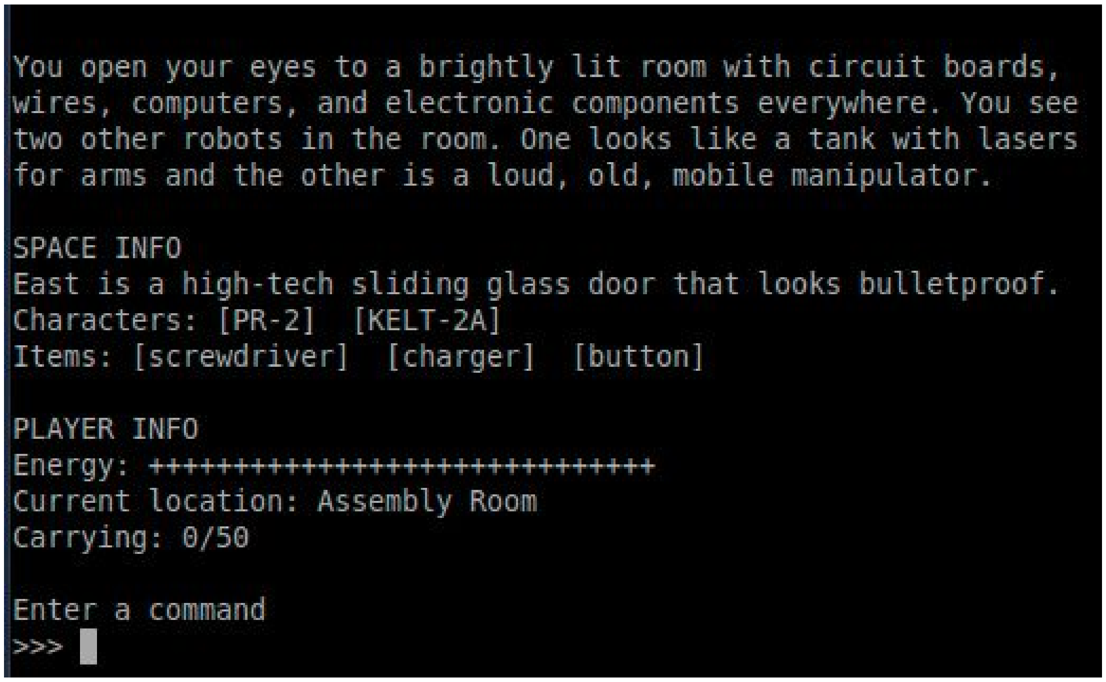
As a Python project, Auriga does not require any type of installation, setup or compilation, just that Python ≥3.3 be used to run the load_auriga.py script in order to run the game. Auriga must be run on a UNIX-based operating system, and it was tested on the OSU linux server, flip.

Upon running the game, the player is met with a menu asking for input on whether to start a new game or load from a saved game file. If the player selects to start a new game, they are presented with an introduction to the story, given a quick explanation of how to interact with the game world via parser commands, and briefly shown the help menu that explains how the game works (the help menu is accessible anytime by typing "help" at the game's command prompt).



```
Would you like to start a new game or load an existing saved game?
Enter the number of your choice.
1. New Game
2. Load Saved Game
3. Exit Game
> 
```

Immediately after choosing to start a new game (and going through the introduction) or loading, the player is launched into the game. In the case of a new game, the player starts in the Assembly Room and is greeted by the following description of the room and its contents, along with Auriga's minimalist UI:



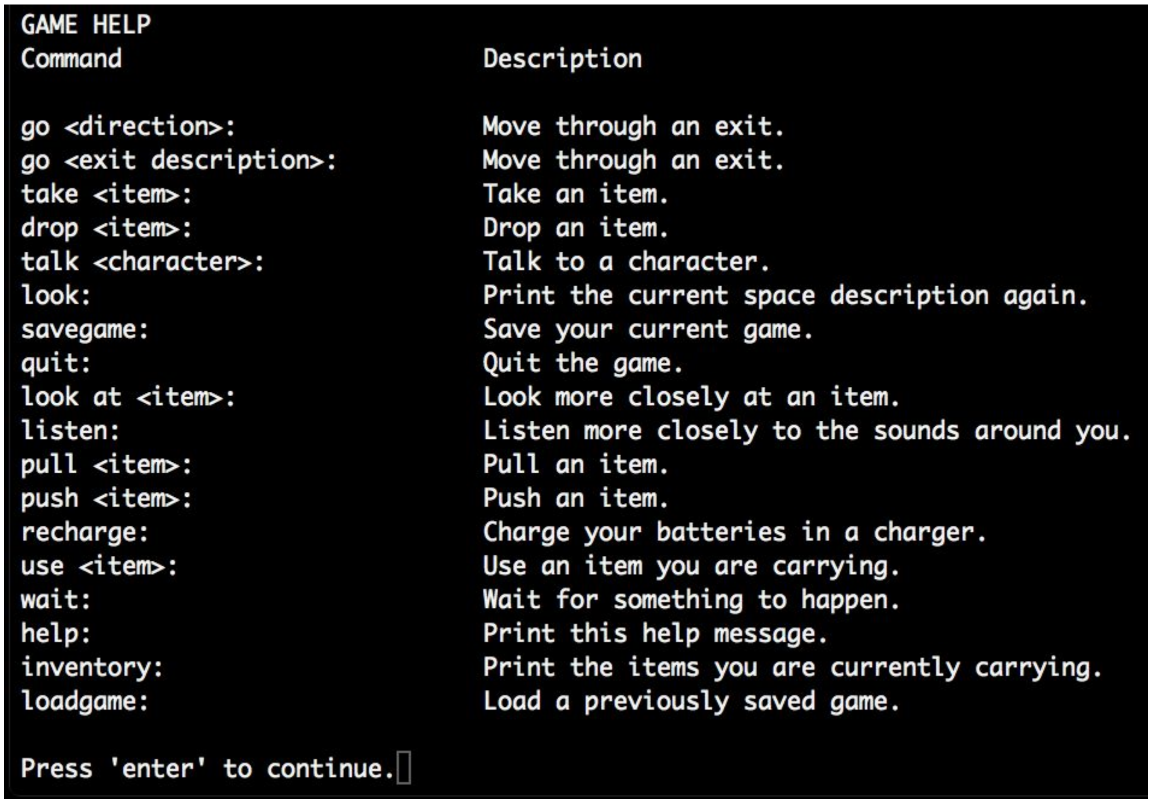
```
You open your eyes to a brightly lit room with circuit boards,
wires, computers, and electronic components everywhere. You see
two other robots in the room. One looks like a tank with lasers
for arms and the other is a loud, old, mobile manipulator.

SPACE INFO
East is a high-tech sliding glass door that looks bulletproof.
Characters: [PR-2]  [KELT-2A]
Items: [screwdriver]  [charger]  [button]

PLAYER INFO
Energy: ++++++++++++++++++++++++++++++
Current location: Assembly Room
Carrying: 0/50

Enter a command
>>>
```

From this point, the player can begin interacting with the characters and items to explore the Auriga facility, and discover the truth about Auriga-7B and the other robots there. A look at the Game Help reveals the actions available to the player:



```
GAME HELP
Command                         Description

go <direction>:                 Move through an exit.
go <exit description>:          Move through an exit.
take <item>:                    Take an item.
drop <item>:                    Drop an item.
talk <character>:               Talk to a character.
look:                           Print the current space description again.
savegame:                       Save your current game.
quit:                           Quit the game.
look at <item>:                 Look more closely at an item.
listen:                         Listen more closely to the sounds around you.
pull <item>:                    Pull an item.
push <item>:                    Push an item.
recharge:                       Charge your batteries in a charger.
use <item>:                     Use an item you are carrying.
wait:                           Wait for something to happen.
help:                           Print this help message.
inventory:                      Print the items you are currently carrying.
loadgame:                       Load a previously saved game.

Press 'enter' to continue.
```

The parser allows for some variation of some of these commands via the use of aliases, and they don’t have to be used exactly as described above - “charge”, for example, will work just as well as “recharge”. The player could also type, “go through the steel door” instead of “go steel door”, or “go east”.

Application Design

Auriga was designed and implemented from its inception under an object-oriented design paradigm. The team, led by Engine Developer David Moon, initiated the project by coming up with the classes that would make up the core of the game, their attributes and behaviors, as well as the relationships and dynamics between them. The Auriga class, implemented in *load_auriga.py*, brings the rest of the engine’s components together and inherits from the abstract Game class. Given this, it makes sense to begin by covering the Game class’ role in helping drive the application.

The Game class handles the basic structure of gameplay. Its major components are a Player object, an event status variable, which keeps track of the player’s achievements as they complete the game. The event status is used to modify room descriptions, character dialogues and the state of hidden exits for some of the rooms. In addition, the Game class has functions that handle the saving and loading of game data during play. The save function creates a directory structure for JSON files that contain all of the game data. The load function is designed to load a game instance from the same directory structure. The directory structure is detailed below.

The game is initially loaded from the JSON files in the *init_auriga* directory. The basic structure of a game is as follows:

```
init_auriga/
  init_auriga/
    player/
      player_name.json
    game/
      game_name.json
    items/
```

```
        item_name_unique_id.json
        ...
    characters/
        character_name_unique_id.json
        ...
    spaces/
        space_name_unique_id.json
        ...
    exits/
        exit_id_unique_id.json
```

In this structure, there can be any number of items, characters, spaces, and exits (each with their own JSON file). The *init_auriga* directory is nested within another *init_auriga* directory with the intention to allow the addition of new levels or environments inside the top level *init_auriga* directory. For example, to expand the game we could add a *level_2* directory that would have the same structure as outline above. Once the player completed *level_1*, the *level_2* game directory could be loaded, and the game would continue. Saved games are saved in the exact same structure, but are nested under the *saved_games* directory, instead of the top level *init_auriga* directory.

It is *load_auriga.py*'s Auriga class that really implements the game class, however, through Python's inheritance features. Aside from the attributes and functions discussed above, Auriga's *main()* function is the driver of the game, beginning by presenting the player with the main menu on the game's startup and calling the *start()* function inherited from Game to run the game logic. It also contains the *use()* function which is relied upon heavily to trigger all kinds of item-based events in the game. Finally, it also owns the *push()* and *pull()* functions which control the rest of the player-item interactions available in the game.

Of all the other engine components, the most involved and interconnected with other components is the Player class. It interacts with spaces and exits via its *go_exit()* function, with items in spaces and its own "inventory" in the *take()*, *drop()*, *charge()* and *look_at()* functions and with characters via the *talk()* function. The Space class, likewise, interacts with as many of the game components, although not by behavior - most of the Space class' functions are simple getters and setters, and the rest are used to print information to the player - but via object composition.

The Exit, Character and Item classes are all fairly straightforward. Their behavior is limited to getters and setters and, in the case of Character and Item, a small number of attributes that all have unique values. The Exit and Item classes have attributes such as *locked*, *visible* and *unlock_item* (in the case of Exit) that are referenced by other classes to determine their level of visibility and interactivity at a given stage in the game.

The other major component of the game/application, which lives outside the engine resources, is the parser, developed by Jason Goldfine-Middleton and represented by the Parser class contained in the *parser/parser.py* file. The parser relies on a collection of JSON files containing all the canonical names and synonyms for the various spaces, exits, characters and items, as well as the directions and actions that were included into the game's design. Additionally, there are JSON files specifying a broad range of articles and prepositions for the parser to identify and deal with appropriately. The result is a system that can effectively parse fairly natural language within the constraints of the game.

Class Descriptions

Item:

The Item class represents an object that the player can interact with. Some items can be picked up, some items, when picked up, can change the state of the game, and some items can change the player's attributes. A few examples of items in the game are:

- **Charger:** Fills the player's energy to maximum capacity
- **Button:** Pushing a button can unlock exits
- **Lever:** Pulling a lever can cause the current environment to change
- **SSD:** Using the SSD on a broken robot in the game fixes the character, and the character helps the player in advancing the game

Character:

The Character class represents animate objects in the game. The player can talk to characters, and they often give the player vital clues as to what they need to do next to advance the game. A character's response to the player may change as the game state changes.

Space:

The Space class represents a room or a place that the player can occupy. Spaces must have at least one exit that the player can enter or leave through. Spaces in the game are linked by Exit objects. Spaces have descriptions that can change as the state of the game changes. A long, detailed description is printed the first time a player enters the space in that game state, and a brief description is printed subsequent times. A space can contain Character, Item, and Exit objects.

Exit:

The Exit class is used to link Space objects in the game. A player moves through an Exit to gain access to another Space. An exit may be visible or invisible to the player, and it may also be locked or unlocked to the player. A locked Exit may contain an "*unlock_item*", which is a specific Item in the game that allows the player to unlock that Exit. Without that Item, the player cannot pass through the exit. Other Exits may unlock via player actions. Exits contain both a description like, "steel door" or "large opening", and also a cardinal direction like, "north", "west", or "up".

Player:

The Player class represents the user who is interacting with the game. A Player has a capacity for carrying Items, and a finite amount of energy. A player has a boolean value that indicates whether or not they are "alive", and also a location attribute that corresponds to the Space object that the Player currently occupies.

The Player class contains the majority of action functions that a user can invoke while playing the game. A few examples of a Player action function are: *go_exit()*, *take()*, *talk()*, *drop()*, etc.

Game:

The Game class is the primary class to the Auriga game. It is the one class that ties all of the other classes together. There is an event_status field that corresponds to specific achievements the player must accomplish to advance the game. The *event_status* variable increments linearly, and cannot go backwards. A Game can contain any number of Space, Character, Exit, and Item objects, and one Player object. The specific commands that the Game supports either through Game action functions, or Player action functions are: go, take, drop, talk, look, savegame, quit, look at, listen, pull, push, charge, use, wait, help, inventory, and loadgame. A few examples of action functions that reside in the Game class, as opposed to the Player class, are: *save()*, *load()*, *help()*, etc.

Auriga:

The Auriga class is a child class of the Game class, and is meant to contain all of the Auriga specific details of a Game. Objects are distinguished by name in this class, which only apply to Games of this type. For that reason, many Game functions are overridden to allow for special use cases of *use()*, *push()*, and *pull()*. This enables us to specify items, characters, or spaces that only exist in the Auriga game, but may not exist an every Game.

Parser:

The Parser class represents a standalone module of the Auriga program, built for a very specific task: interpreting all user input as in-game commands. Being aware of the canonical names that identify each action, item, character, exit type, and other grammatical elements of natural English, the parser is able to distill down to its essence each string a user enters at the command prompt. In addition to interpreting the structure of the words entered by the user, the Parser served to identify various aliases for canonical names, allowing the user to be flexible with her directives.

generate_auriga.py:

The *generate_auriga.py* script is not a class, but it was used as a helper script to generate all of the JSON files for the Auriga game. We did this by implementing the load and save functionality, and then created all of the game objects in the *generate_auriga.py* script. We then called the *save()* function, which generated JSON files for every object contained in the game. This method was far easier than creating all of the initial game objects through JSON files.

Team Member Accomplishments

David Moon worked on the engine's design, each of its components, and took on the task of designing and implementing the data files, as well as the directory structure that houses them, used for save and load functionality. David also took the lead in designing the game itself, the theme behind it, and the various rooms and puzzles that players traverse.

Jason Goldfine-Middleton designed and implemented our highly sophisticated parser, determining the various aliases that the player would have as options for the literals that the parser translates for the engine. He worked closely with David Moon to ensure that the parser and engine spoke and understood the same language. With a simple and robust API, a wide range of user inputs are handled. Jason and David also collaborated in completing the entirety of the mid-point report for the group.

Gregory Fernandez worked on project management for the group, laying out tasks at the beginning of the term and listing them out on a shared Redmine server. He also created unit tests for the engine components and the runner for said tests and wrote the majority of the final report for the team.

Deviation from Initial Plan

While there were some trivial deviations from the initial project plan, the only significant change was the decision not to use the Python curses library for text formatting.

Conclusion

Auriga is the product of a whole term’s worth of collaborative effort between three OSU CS students in designing, implementing, testing and debugging a Python game that was built from scratch. From the outset, the assignment was taken as a base to push the programming skills acquired through the program into a realm that enabled sincere expressiveness and creativity. After meeting the program specifications as we had initially set out to do, we were able to tinker with the UI to add a bit more amusement. David and Greg came up with the idea of creating a soundtrack, complete with audio effects for special events, to accompany gameplay. David was able to implement this audio system during the final week of development and test it on his local Ubuntu box, but this solution was not portable to macOS or OSU’s engineering servers, leading us to create a parallel branch of the game with audio. This version is presented in the final demonstration video of the Auriga game for your enjoyment. The team has found that collaborating on Auriga was mentally expansive, challenging, and fun and we look forward to all player feedback.

You can view a video walkthrough of the game with audio at:
https://media.oregonstate.edu/media/t/0_thfu2xgy