TBA2 - Text Based Adventure Game 2

Haunted House Adventure

Final Report

Members:

Hualong Li
Graham Roese
Brandon Sedgwick
Keenan Taggart

I. Introduction

Our team has created a piece of interactive fiction, otherwise known as a text-based adventure game. A game of this type uses software to simulate environments, experiences, and characters which are affected by commands from the user. In most cases an overarching narrative is presented in which the user plays an active role.

We opted to focus our text-based adventure game around exploration and puzzle-solving. The theme used throughout the development was that of a Haunted House where a user is placed inside a home of several rooms and is required to explore each room thoroughly, interact with objects, and solve a challenge in-order to escape. All interactions are done through text commands that the user enters into the command line. This software utilizes text line parsing and interpretation of natural language, as well as data structures to logically and functionally represent the game world.

II. The User Experience

The game is mostly an event-driven experience for the user. For each input from the user, there is a confirmation of the input and a useful feedback event that will inform the user. The user will experience decision-making, character creation, and lore through the lens of the console game. All information and narrative is delivered via text to the player.

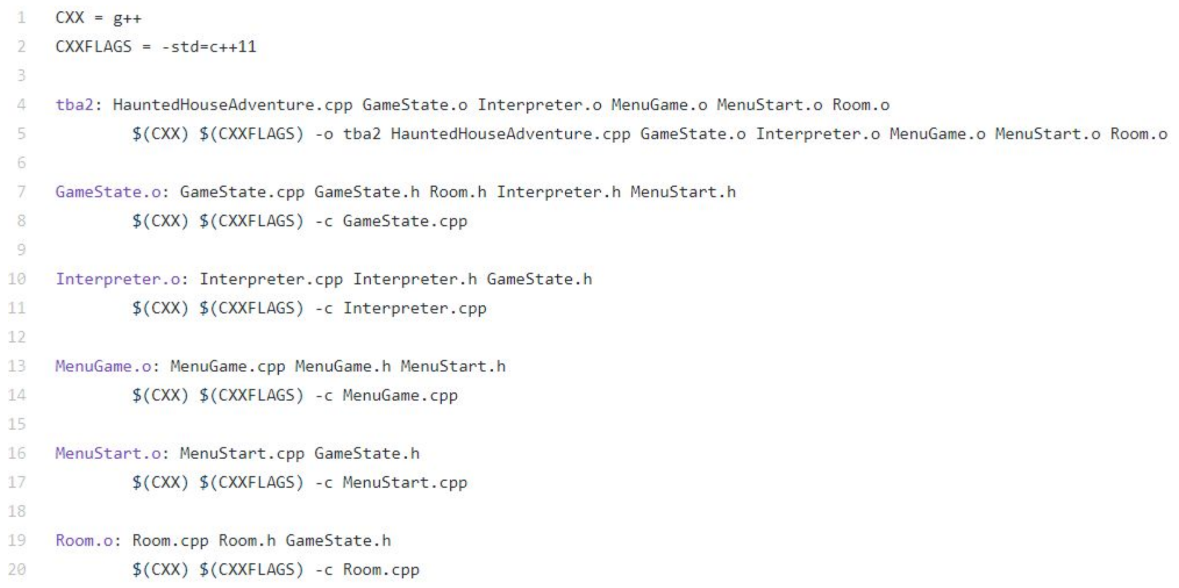
The user enters commands to the console in order to advance the game state. Commands can either be directly prompted from the terminal (such as making a choice of several finite options, e.g. “Would you like to save? (Y/N)”) or will follow a set list of commands that will be taught to the user during the beginning stages of the game.

The user has immediate access to gameplay help in the form of a Help menu. This Help menu provides information about game commands. The user also has immediate access to a menu where he or she can view information about items, health, game progress, and any other milestones that might help the user contextualize his or her progress.

The game also permits the user to save their progress at any time they choose, allowing them to close the game at their leisure and begin again from the point at which they left off. The save system preserves the player’s progress as well as changes made to the game world.

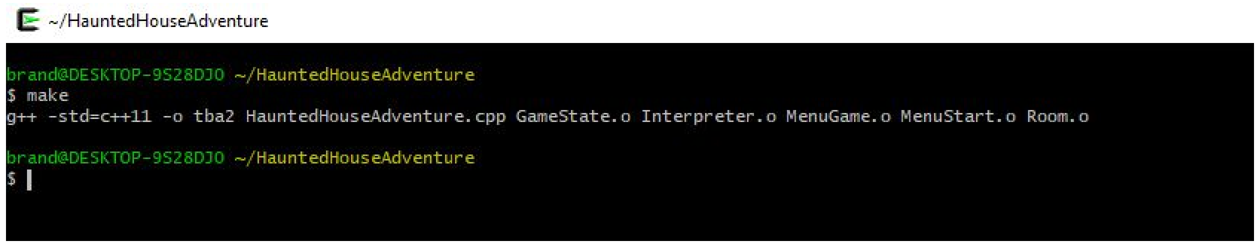
III. Setup and Usage

The executable for the game can be compiled utilizing the supplied makefile. Our team choose to write the software in C++, and the following makefile is capable of building the necessary executable to play the game:



```
CXX = g++
CXXFLAGS = -std=c++11

tba2: HauntedHouseAdventure.cpp GameState.o Interpreter.o MenuGame.o MenuStart.o Room.o
        $(CXX) $(CXXFLAGS) -o tba2 HauntedHouseAdventure.cpp GameState.o Interpreter.o MenuGame.o MenuStart.o Room.o

GameState.o: GameState.cpp GameState.h Room.h Interpreter.h MenuStart.h
        $(CXX) $(CXXFLAGS) -c GameState.cpp

Interpreter.o: Interpreter.cpp Interpreter.h GameState.h
        $(CXX) $(CXXFLAGS) -c Interpreter.cpp

MenuGame.o: MenuGame.cpp MenuGame.h MenuStart.h
        $(CXX) $(CXXFLAGS) -c MenuGame.cpp

MenuStart.o: MenuStart.cpp GameState.h
        $(CXX) $(CXXFLAGS) -c MenuStart.cpp

Room.o: Room.cpp Room.h GameState.h
        $(CXX) $(CXXFLAGS) -c Room.cpp
```

To compile, simply type ‘make’ within the code’s directory.



```
~/HauntedHouseAdventure

brand@DESKTOP-9S28DJ0 ~/HauntedHouseAdventure
$ make
g++ -std=c++11 -o tba2 HauntedHouseAdventure.cpp GameState.o Interpreter.o MenuGame.o MenuStart.o Room.o

brand@DESKTOP-9S28DJ0 ~/HauntedHouseAdventure
$
```

This creates an executable called ‘tba2’ which can be run by entering ‘./tba2’.

IV. Software & System Interplay

Because the application requirements restricted the use of software systems to only the Flip G++ compiler and the Standard Template Libraries, we decided to create our own toolset to accelerate development and make our code base more robust. To achieve this, we designed classes that would hold pertinent gameplay information such as Rooms, Inventory, and other gamestate variables such as whether or not a room is dark. We used standard data structures for these components, and then started creating code that would interact with our classes.

Interacting With Keyboard Input

The primary technique for achieving the goal of robust text parsing code was creating effective wrappers around STL data structures. This allowed for us to not only simplify our code, but also to account for the unpredictable nature of human-typed text input. For instance, we needed a solution that could take “eat a couple of bites out of a sandwich,” know that the user wants to eat something, look through the room for a sandwich, and the print the correct text. To achieve this, we created functions that would parse the text and check the text against the entire game environment. A sample encapsulated call to a text parser would look something like this:

- Get the text input, “eat a couple bites out of a sandwich”
- Look through the text for for any of our reserved verb keywords (“eat,” “look”, “savegame”, etc.)
- If we found reserved keyword, create a pair<Verb, string> with the correct verb.
- Now, locate vectors for the game environment. For a typical room, this would be the player’s inventory, the room features, the objects in the room, and any special items.
- Find if any objects are substrings of the input string. Try and see if we can match up a valid action between the verb and the found object if we found a substring.
- Perform the action.

Maintaining Robust Environment States

We tried to maintain code that followed real-world conventions. Simple add/erase options might work for simple operations, such as calling inventory.push_back(item) on the inventory vector, but this only accomplishes the goal trivially. In order to make code like this work, we would need to examine what would happen in a real situation. In a real situation, we would need to manage: preventing the item from being double-added (when the add function gets called twice somehow on the same object), checking that the object really exists in the room, adding the object to the inventory, and updating the player’s GameState to match what just happened. Thus, we are interacting with 4 classes in one action. Let’s take a look at the function at the core of taking an item, actionTake().

```
void GameState::actionTake(string strAction)
{
    // perform linear search for an item.
    // function returns 0 if item is not found.
    int index = itemExistsInRoom(strAction);

    // if we didnt' find the item, abort the attempt to search.
    if (index < 0)
    {
        cout << "That item does not exist in this room." << endl;
    }
    // else, we did find the item.
    else
    {
        // we returned an integer from itemExistsInRoom, so use that as the index.
        string strItem = this->currentRoom->vecInventory.at(index);
        // verify we don't have the item in our inventory
        // by calling itemExistsInIntenvory [sic]
        // if it returns -1, that means we don't have it in our item and we can add
it.
        if (itemExistsInIntenvory(strAction) == -1) {
            this->vecInventory.push_back(this->currentRoom->vecInventory.at(index));
        }
        // now, we can erase the object from the room.
        this->currentRoom->vecInventory.erase(this->currentRoom->vecInventory.begin()
+ index);
        string strOut = "";
        // print success message.
        strOut += "You took the " + strItem + " and added it to your inventory.";
        cout << strOut << endl;
    }
}
```

Even this code by itself doesn't solve the problem. In order to effectively use the keyboard input and distill only the relevant information, we needed to leverage the string library to parse out only the relevant information, making our searches meaningful and repeatable.

V. Auxiliary Tools, APIs, Libraries

Since the game world in our software would involve a variety of rooms and objects in those rooms, the team felt as though the game world and its functionality could be best represented in an object-oriented programming language and so selected C++ for the task. We could then use inheritance to quickly define the elements of the game world as well as their relationships to one another and the player. This also assured us a great deal of portability, meaning that our eventual deployment to the Linux platform would require little effort.

Our use of libraries was largely restricted to the Strings library, Input/Output libraries, and the Standard Template Library which all boast high portability. Given that much of the software's behavior is governed by receiving and presenting strings to the user, the Strings library was critical for our needs. It allowed input to be received and handled in a straightforward, error-resistant manner, and for output to be constructed and passed to auxiliary functions without complication.

The software also relies heavily on input/output libraries which allow for input/output streams, string streams, and file streams. Input/output streams were used directly for getting text from the user (through standard input) and pushing text to the user (through standard output). String streams allowed for user input to be broken up into words and fragments in order to detect verbs and direct objects in the user's command. File streams were necessary because of the vast amount of game text stored in auxiliary files which the software needed ways to access. Additionally, the save functionality needed ways to write data to the disk which was also covered by this library.

The Standard Template Library allowed us to use several convenient and efficient containers for our purposes. Each room in the text adventure needed to store lists of objects and features, so the vector container was a good standardized way to build these lists as well as search them when required with iterators. Further, we often needed to combine two disparate things together (such as an object in a room and an action being carried out on the object) and so the utility container Pair in the STL was a useful way to organize, express, and interpret binary relationships among incongruous game elements.

The map container in the STL was also critical for the functionality of the text interpreter. In order for the program to be tolerant of variations in expression of meaning it would need to have a way to notice and understand synonyms of basic game verbs. The map container allowed for a broad swathe of expressions to correlate to fundamental game actions that could be understood and carried out within the game. For example, the game allows for the user to "hit" different game elements but needs to allow for different ways to express this action. A map allows the software to associate words like "strike" or "punch" with "hit," such that unusual commands can be boiled down to their basic meaning in the game.

A substantial amount of development was carried out in Visual Studio in a Windows development environment. This choice meant the team could focus directly on building the software while smaller details like dependencies or build automation could be handled by our tools without much effort. This also meant that code could be shared, compiled, and executed seamlessly between different members of the team. Further, our project involves large amounts of text in discrete files which Visual Studio was able to filter/arrange in clean, logical ways.

Transitioning to Linux meant navigating away from Visual Studio and toward Make for build automation. This involved more work to create/maintain the rules and dependencies, but was achievable given the size of the project. Organization of the text files simply involved implementing a logical directory structure.

Version control for the project was all performed via Git. Its distributed nature allowed team members to work independently on features as delineated in the project plan and for easy integration of this work once it was completed. Use of Git also ensured high portability for our code which was a concern when considering our eventual transition to Linux.

Our collaborative tools were largely confined to GitHub -- a natural choice given our use of Git. This served as a centralized location to save our group's work as well as post changes which each group member would need to incorporate into their own local versions of the project. GitHub also promised reliability and accessibility as opposed to one of the group members running their own centralized server.

VI. Starting a Session and Playing the Game

About this Guide

This guide will take you through not only most of the rooms in the game, but also will provide some real use cases for each of the games library of actions, menus, features and objects.
Gameplay inputs are written in Orange text. So, when you see the text Use the hanger, simply type that phrase into the "Enter input:" prompt and move to the next line of text.
Remember also that when traveling, you can either type prompts like "Go North," "North, " "Go to the staircase," "Go staircase," or even just "staircase" to start traveling.
Each prompt in the guide should be accompanied with some info about how the game will react, or what your next step should be.
You'll see the following for each prompt:
Enter input: Text for you to type in à Explanation about what's going on.
When you see a room name in the document written in bold text, verify that you're in the correct room! If you're lost, consult the room guide at the end of this document.
We recommend playing the game with a terminal width of at least 80 characters.
Let's get started!

Starting the Game

If you haven't done so already, unzip the zip file and type make tba2 into your shell prompt to compile the game files necessary to start playing.
Start the game in the shell of your choice. On most shells, you'll just need to type ./tba2
Main Menu
Take a look at the menu. See that you have two main options: load a game or start a new game. Let's start a new game for this playthrough. We can load a game later.
Your answer: 1

Wine Cellar

This is where you start the game. You don’t know how you got here, but you know you need to get out.

Enter input: Help This will bring up the in-game context menu with all of the verbs we’ll use during the game.

Enter input: Look à You should have already seen this description, but typing look will bring it up again. We can see that there’s a room description, a features list, and an objects list. As a general rule, objects can be taken from the room and features must stay, but can be interacted with.

Enter input: Look at the bottle à Examine the bottle closer and see what the label says.

Enter input: Inventory à See that your inventory is empty. Let’s start taking items.

Enter input: Take the bottle right now à Take the bottle. We can use this long phrase to show that as long as you’re typing about taking the bottle, the application will respond accordingly.

Enter input: Inventory à Now, you can see the bottle in your inventory.

Enter input: Look à See that there’s a crawlspace. Let’s try and go through it.

Enter input: Go to the crawlspace àWhen you get to the other side, you should now be in the root cellar.

Root Cellar

Enter input: Go to the cellar door à Notice that the room is dark. You’ll need to make some light in here if you want to get out.

Enter input: Take the matches àGood, now you’re almost ready to light the room.

Enter input: Use the matches à Looks like that didn’t work. Didn’t you see a candle in the wine cellar room?

Enter input: Go crawlspace àGo back to the wine cellar.

Wine Cellar

Enter input: Take the candle àGood, now you should be able to make some light.

Enter input: Go crawlspace à Go back to the root cellar.

Root Cellar

Enter input: Hint à Get a hint. This might help if you want some help.

Enter input: Use matches (or, Use candle) à Now the room is lit. You can get out!

Enter input: Go cellar door à Let’s get out of here.

Kitchen

Enter input: Use the stove à See some whimsical text.

Enter input: Use the kitchen sink à More whimsical text.

Enter input: Go north à Let's check out that pantry.

Pantry

Enter input: Search the lockbox àGet the flashlight.

Enter input: Go south à Let's go back to the kitchen.

Kitchen

Enter input: Go west à Go to the dining room.

Dining Room

Enter input: Go rusty door à This would work normally, but we Need key 2.

Enter input: Take key 2 à This should help us with that door.

Enter input: Use the dining table à View some whimsical text.

Enter input: Go rusty door à We have the key now, so we can travel.

Hallway 4.

Enter input: Sit on the armchair à Might as well use "sit" while we can!

Enter input: Go south à We can head to the gallery now.

Gallery

Enter input: Take wedding photo àThis looks useful.

Enter input: Look at wedding photo à View a description of the picture.

Enter input: Drop wedding photo à We probably don't need this anymore.

Enter input: Look at the painting à View some whimsical text.

Enter input: Go south à head to hallway 5.

Enter input: Burn the cobwebs à Might as well burn these cobwebs.

Enter input: Go west à Go to Hallway 6.

Hallway 6

Enter input: Go attic hatch à We need to go up to the attic and start looking around.

Enter input: Look at the newspaper à We need a break. Let's check out the newspaper.

Enter input: Go south à Go south.

Music Room

Enter input: Play the guitar à You play the guitar, and find a cable. Add the cable to your inventory.

Enter input: Go north à Essentially fast traveling from music room to the bedroom.

Enter input: Go north à Essentially fast traveling from music room to the bedroom.

Enter input: Go west à Essentially fast traveling from music room to the bedroom.

Bedroom

Enter input: Drop the cable à Let's just do this for demo purposes.

Enter input: Use the tv à take a look at the error. We needed that cable!

Enter input: Take the cable à pick the cable back up.

Enter input: Use the cable à now you should be able to watch the video.

Enter input: East

Enter input: East

Enter input: North

Enter input: North

Enter input: North

Enter input: North

Enter input: North à We're headed to the foyer.

Foyer

Enter input: Use telephone à See the messages and start to make a plan.

Enter input: Go West à Take let's see what's in the cloak room.

Cloak Room

Enter input: Hint à Might as well get a hint while we're here.

Enter input: Search the hat rack à Search the hat rack and put a letter into your inventory.

Inventory

Enter input: Look at the letter à Get some information about the phone.

Enter input: Go east

Enter input: Go south

Enter input: Go South

Enter input: Go west.

Bathroom

Enter input: Take the hanger à We might need this.

Enter input: West à Let’s start traveling.

Enter input: West à Head to the billiard room.

Billiard room

Enter input: Play the billiard table à Remember the letter? Get the battery from the table by playing a game of billiards.

Enter input: North

Enter input: West à Another clue from the letter. Let’s check out the conservatory.

Conservatory

Enter input: Search the chest à Get the phone cord and add it to your inventory.

Enter input: East à Begin fast traveling back to the foyer, with a detour in the library.

Enter input: South

Enter input: East

Enter input: South à Go to the library.

Library

Enter input: Search the bookcase à See the message about a suspicious item.

Enter input: Hint

Enter input: Search with the hanger àGet the diamonds. Make sure you picked up the hanger from the bathroom if you need to (just type Inventory)

Enter input: Inventory à See that you now have some diamonds in your possession.

Enter input: North à Continue traveling to the foyer.

Enter input: East

Enter input: East

Enter input: North

Enter input: North à Now, let’s go to the foyer.

Foyer

Enter input: Use the telephone à Looks like this doesn't work. Let's read the message.

Enter input: Inventory à See battery and phone cord. Looks like we have what we need.

Enter input: Savegame à Let's make sure we can savegame here.

Enter input: Use the battery à Looks like that worked!

Enter input: Use the telephone à See the message changes. Looks like the phone isn't operational yet, though!

Enter input: Use the phone cord à Yes, that should do it.

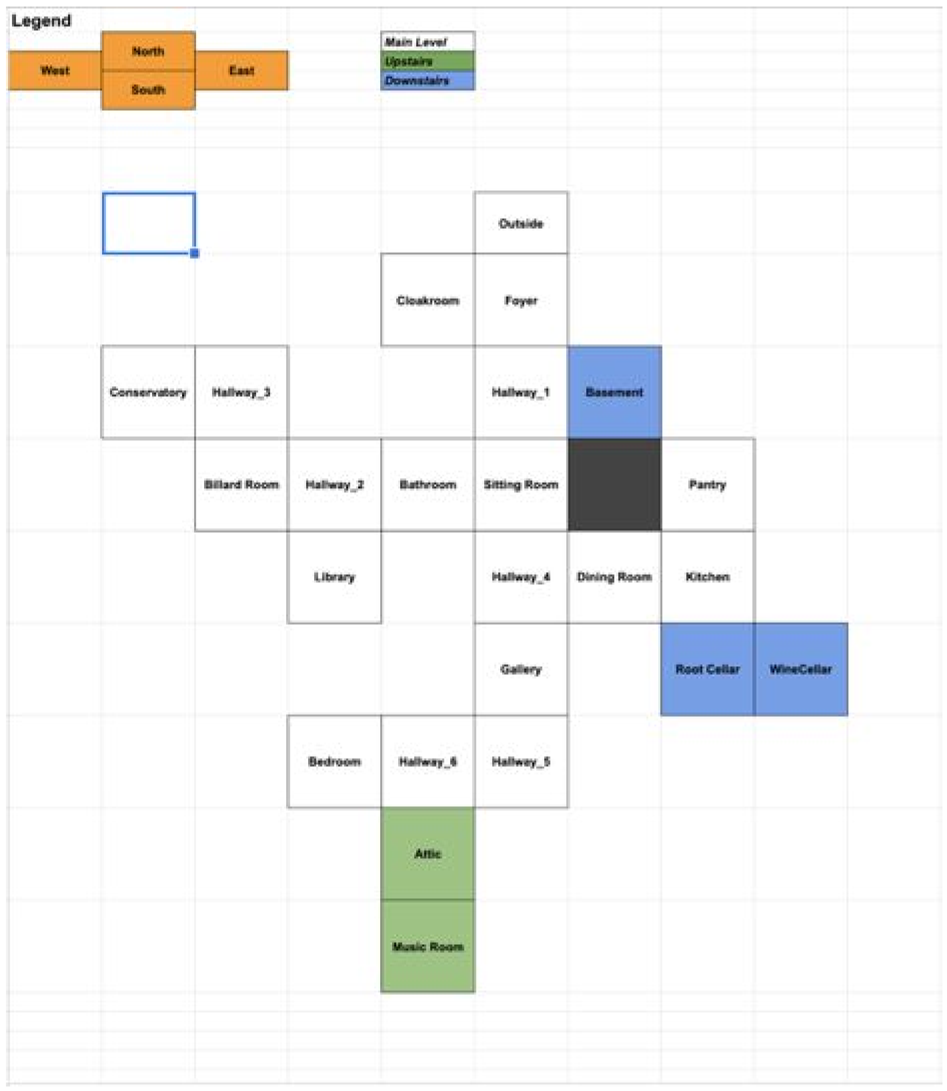
Enter input: Use the telephone à Now, we can make a call. The phone call will change based on whether or not you picked up the diamonds earlier. The application will then exit.

Testing Savegame/Loadgame

Start the application again with ./tba2

Either select 2 (I've been here before) and choose your game file, or select 1 (New game) and then type loadgame when given the opportunity. You can now verify that you're loaded a game, as you're standing in the foyer!

Room Guide



VII. Team member contributions

Graham participated in software development, documentation development, and gameplay development.

Brandon participated in software development, creation of utility files used within the game, documentation development, report creation and testing.

Keenan contributed to software development, reports/submissions/documentation, game writing, and testing.

Hualong developed game lore and provided architecture support.