Hacker’s Labyrinth

Design Document

By: Jasper Charlinski
Sebastien Van Den Bremt
Ethan Posner
Cassius Galdames

Date due: Oct 30th, 2022

Instructor: Dr. David Wessels

Table of Contents

1. Introduction

This document is intended to give a detailed description of the game, "Hacker's Labyrinth." The game involves progressing through a series of computer science related puzzles, all within the context of having been taken captive by a deranged individual. This document describes the underlying systems and subsystems which will be used to facilitate the game, as well as how the team plans to implement them. Specific aspects of the game, such as the puzzles and game environment, are described in the requirements document with far greater detail. It should be consulted whenever sparse detail is given.

1.1 Context

Created by computer science students for CSCI 265 at Vancouver Island University (VIU). All aspects of this project must be completed before it is required to be demoed on December 7th 2022. It must be capable of running on a computer science lab machine at VIU.

1.2 Goals and Non-Goals

With this product, we plan to:

1. Help computer science students to explore and experience different topics of the field.
2. Present them with an interesting programming project with several interconnected components that they can inspect and learn from.
3. Provide a fun, light-hearted experience that can be replicated through subsequent playthroughs.

We do not plan to:

1. Give a thorough education of any topic.
2. Be overly complex and inaccessible.

1.3 Milestones

1. Working main-menu:

   The user should be able to type in commands and receive input back. When the user attempts to start the game, dummy output can be given such as "feature not yet implemented."

2. Basic minimally working level-manager and in-game menu with a dummy puzzle:

   The user should be able to start a dummy puzzle from the main menu. They should then be able to pause the level and be shown a simple in-game pause menu by pressing the 'M' key.

3. Addition of notebook and timer functionality to level manager:

   The user should be timed when they enter a puzzle. Once the timer runs out, a simple message should be displayed to notify the user that they have failed. The timer should continue counting down when they enter the in-game menu or any other screens. It should only stop upon successful or unsuccessful completion of a level.

   The user should also be able to enter a notebook menu from within the in-game menu. Within this screen, they should be able to input text and retrieve it later through the same interface. The timer should continue counting down while the user is in the notebook.

4. Implementation of main puzzles:

   The main puzzles of the game, as described in the requirements document, should be fully implemented and playable. At this point, any level should be accessible at any time (for testing purposes).

5. Addition of logic between levels:

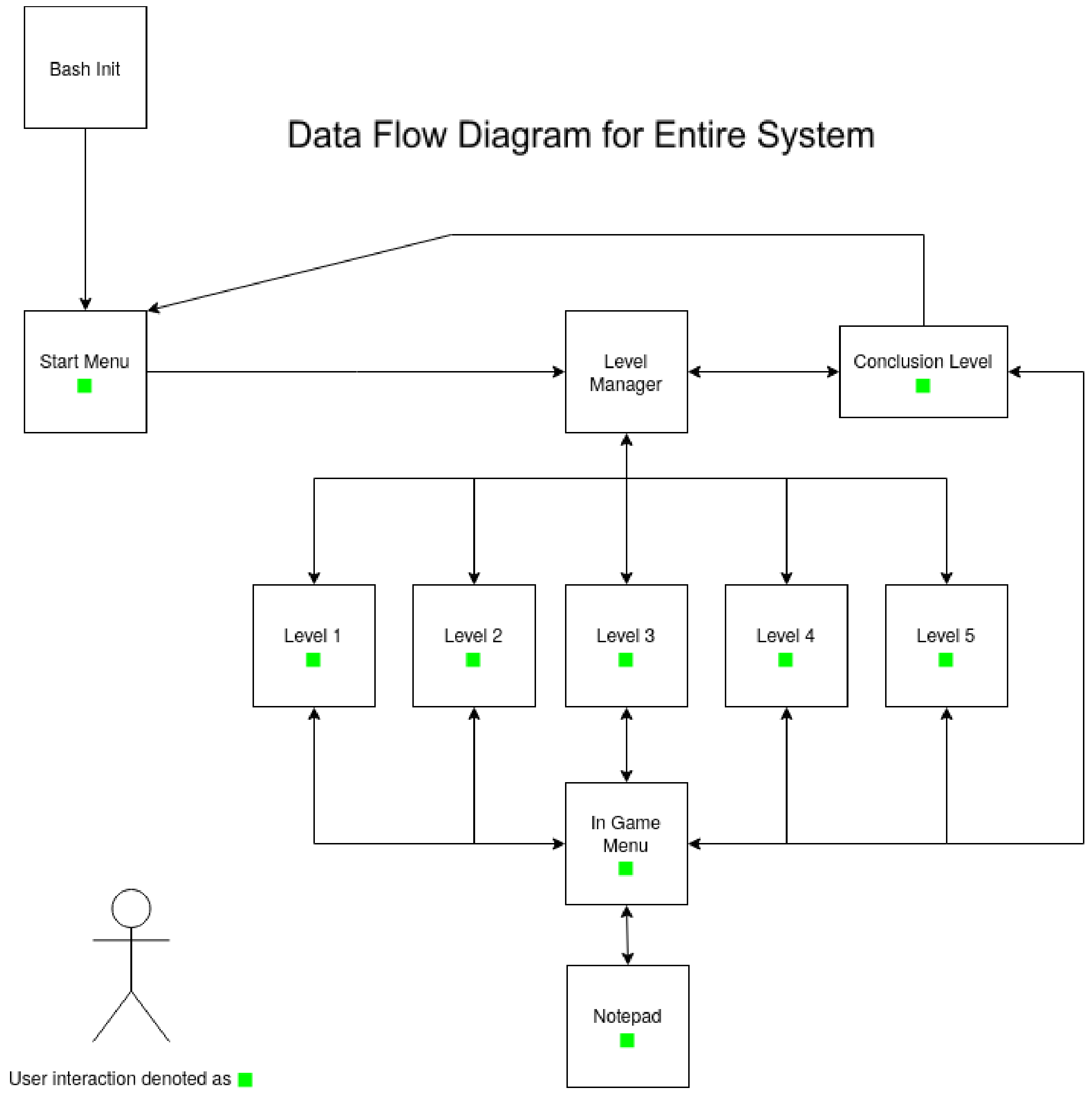
   Certain rules should be used to control aspects of how the levels are played, such as when specific ones are played and how much time is given. This will make each subsequent playthrough of the game different from the last. For example, when the player completes a level and chooses either a left or right door, their choices will need to affect subsequent levels.

6. Completion of final puzzle:

   The final puzzle should be implemented and playable. Conditions of the final puzzle should change depending on what level-logic was used and how previous levels were completed.

7. All parts of game polished:

   The game in its final, presentable state. This will include, for example, ASCII art and animations upon completion of levels or game-over. At this point in time, all parts of the game should line up with the specifications from the requirements document.



2. Logical Design

2.1 Main Shell Environment

The game will be run by a main shell script that will call to a INIT script to resize the screen, create the notepad files, initialize the variables, set a randomizer for levels and run the game. This script will be in the main directory for Hacker’s Labyrinth and all the other executables / images / files will be in subdirectories.

Main.sh (Level manager)

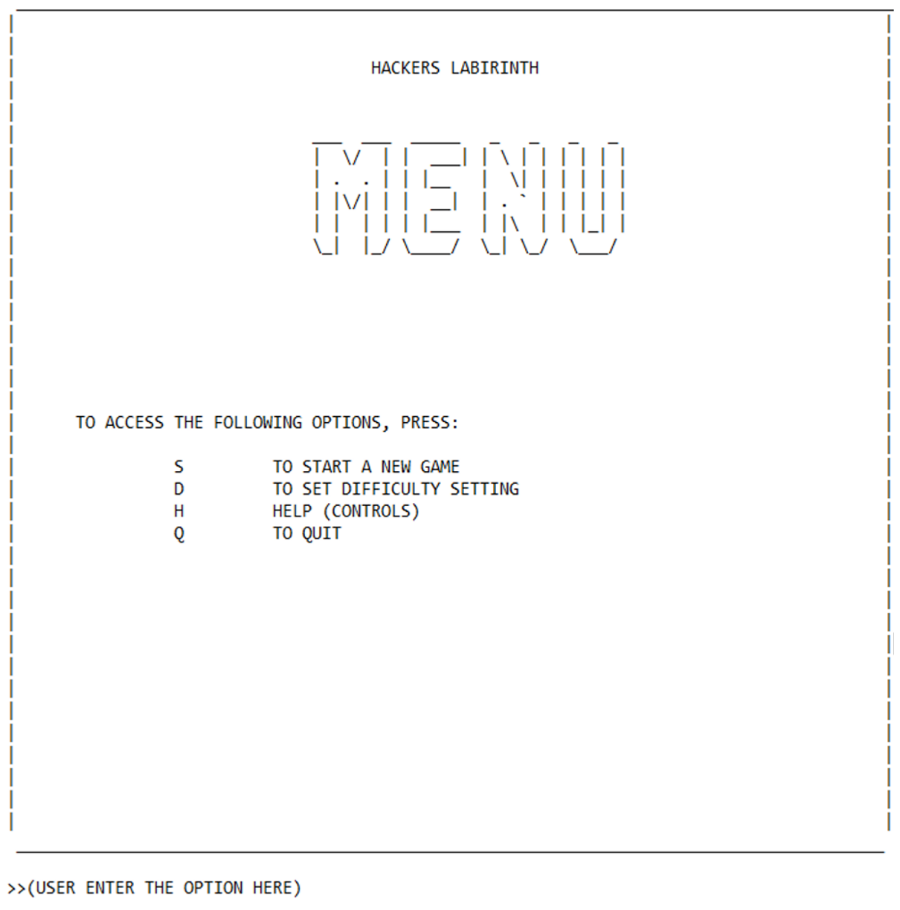
The main script will be a continuous loop that will run all other components of the game. It will start with setting up the environment with the Init.sh then by calling the main menu. The main script will be the method used for traversing directories and opening other executables or puzzles with use of the level manager. It will also be used to store into or retrieve data stored in local variables that can be accessed from lower level executables.

Init.sh

The initialization script will set up an optimal environment for the game to run in. It will also create all the local variables, names and paths needed for the game to run. It will start by resizing the terminal screen size. It will read the name of the user and store that as userName, start a timer variable and initialize a number-of-attempts-left variable. It will also create a text file that will be used for a notepad and fill it accordingly. All of these elements will be conveyed and used again from the Main.sh.

mainMenu.sh

After the environment is established the Main.sh will call the mainMenu.sh. This will display an ASCII styled menu that will have a case,esac statement to recognize user inputs and trigger functions accordingly. The mainMenu will allow the user to set the difficulty which would change the difficulty variable, view the control commands which would print a new screen with commands, quit the game which would call to exit, or start the game. Starting the game closes the menu, calls to the level manager, and commences the first puzzle.



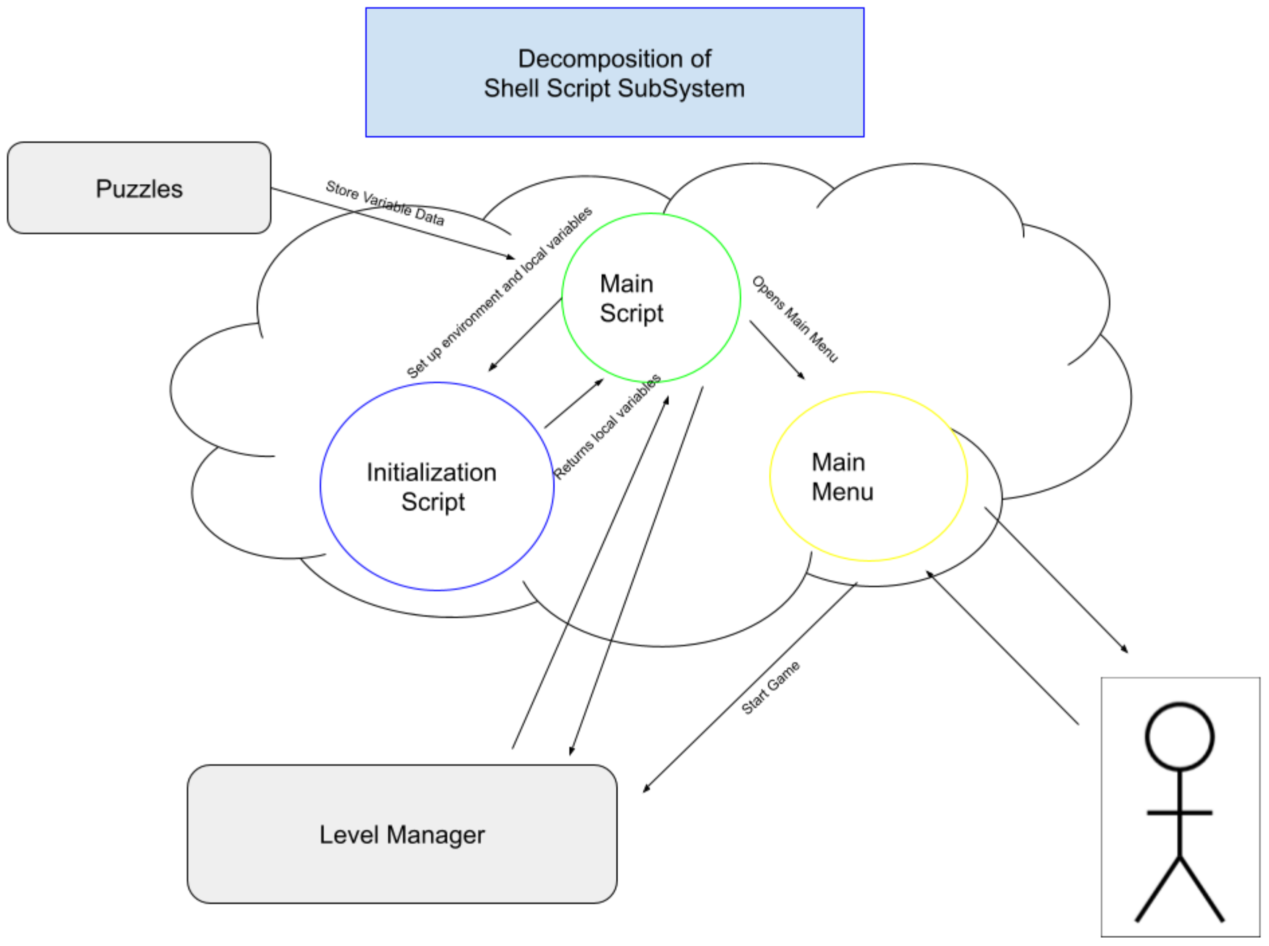
```
                    HACKERS LABIRINTH

                 ___  ___ _____ _   _ _   _
                 |  \/  ||  ___| \ | | | | |
                 | .  . || |__ |  \| | | | |
                 | |\/| ||  __|| . ` | | | |
                 | |  | || |___| |\  | |_| |
                 \_|  |_/\____/\_| \_/\___/


   TO ACCESS THE FOLLOWING OPTIONS, PRESS:

          S         TO START A NEW GAME
          D         TO SET DIFFICULTY SETTING
          H         HELP (CONTROLS)
          Q         TO QUIT

>>(USER ENTER THE OPTION HERE)
```



2.2 In Game Menu

The in-game menu will be written in its own .cpp and .h files and will be called to by each level when the user enters the command **:M**

See Requirements Document 2.1.2 for more detail on the menu layout.

2.3 Notepad

The notepad will be accessed from the ingame menu, see Requirement Document 2.3.3 for more detail.

When the user selects the notepad option, a Vim (like) window will take over the screen where the user can type into. The players notepad will be stored in a .txt file that will be appended to each time the user makes a change. When the user exits the Notepad they will return to the in-game menu.

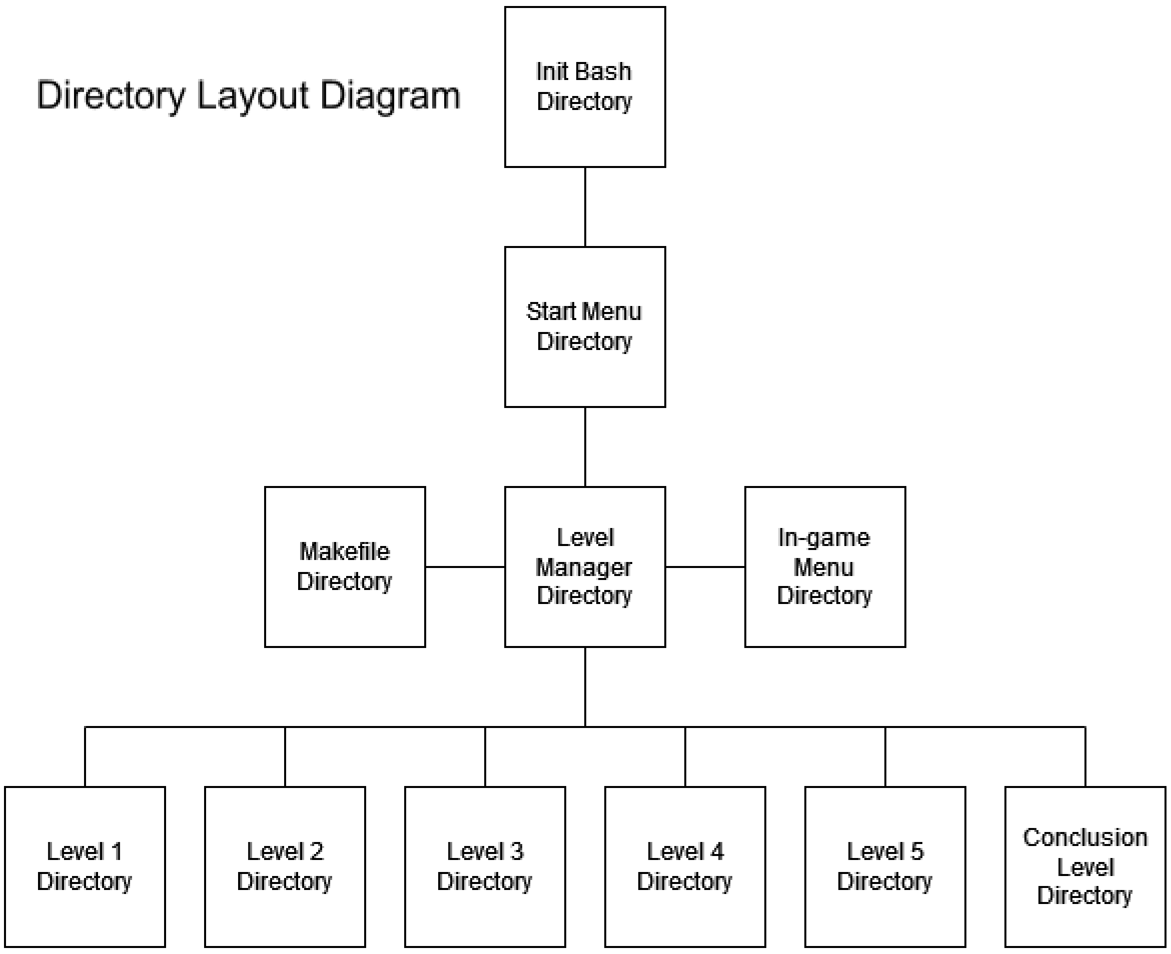
3. Physical Design

3.1 Languages and Tools

Running the game from a shell script will permit us to call to various game puzzles that can be programmed in different languages as the production team have conflicting familiarities and preferences on languages. The environment variables will be made in bash and used or appended to from the puzzle files. These will be either written in C++ or Python, as both languages can log functions from the terminal it will be running in.

3.2 File Directory Structure

The structure for the project's file directories will be a top-down system where the main executable script will have access to the environment initialization script. Its subdirectory will contain the Start menu and any animations, if the scope allows their implementation. The next directory level will contain the level manager script, the makefile for compilation and the in-game-menu. The final level of subdirectories will contain each puzzle’s directory. These will contain their puzzle.cpp, puzzle.h, puzzle.o and any other file they might be immediately dependent on. This directory architecture will allow ease of environment variable storage and use from the elements underneath it, will allow a smooth use of the makefile from its sub directories, and will allow a constant access to the in-game-menu from the puzzles.



3.3 Data Storage

3.3.1 Persistent:

The game's data will be appended to a logfile that will be stored for later access. The logfile will include userName, difficulty, entered results and user behavior.

3.3.2 Non-Persistent:

Most of the data will be stored in temporary environment variables that will be appended from the different puzzle programs. They will be reachable throughout the game duration but will disappear once the game is closed.

These will be the environment variable names, with their utilities:

time= A time tracking variable, when it reaches 0 the game is over.

userID= The name of the user.

diff= Difficulty setting.

powerUp= The value of the randomized power up allocated at the end of the puzzle.

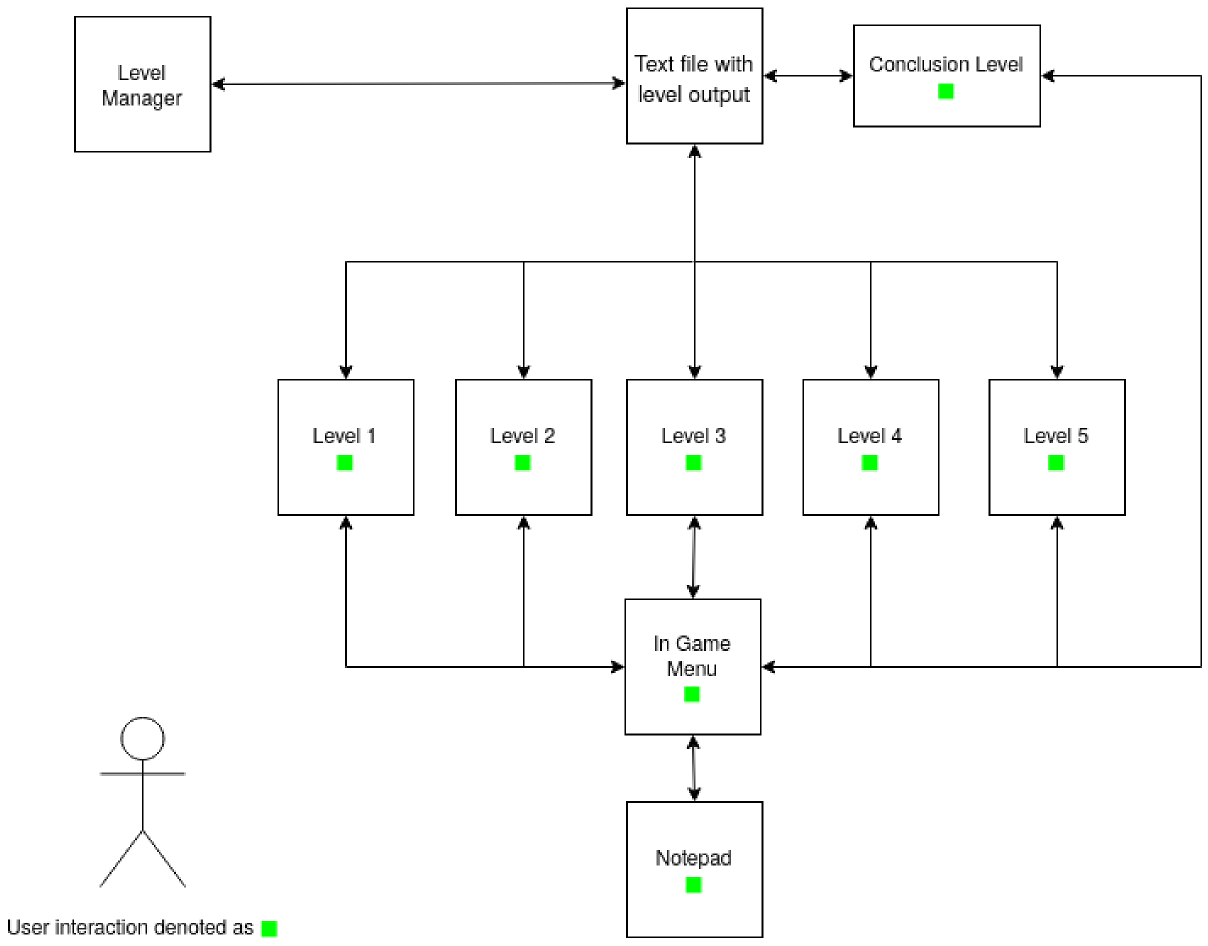
currLvl= The current number of puzzles finished.

maxLvl= Preset by the diff, if easy = 2 if medium = 3 if hard = 4. If the currLvl == maxLvl then go to the final puzzle.

3.4 Level Layout

All the puzzle layouts have been described in detail within the requirements document. Please refer to this for the details.

Data Flow Diagram of Level System



4. Testability

Since we are making a game with components that are minimally complex, there are few things to test. Each one will be tested by directing some input at an executable, and ensuring that some specific output is given or that it completes the expected task. Below is shown which specific systems should be targets for our tests.

1. Main Menu

   The menu can be tested by ensuring that each selectable item works and does what it is expected to do.

   Examples of tests for the in-game menu:

   - If "start game" is selected, a dummy game executable should be run. The in-game menu should no longer be shown.
   - If "end game" is selected, the program should end and all non-persistent game data should be cleared.
   - If "difficulty" is selected, and a difficulty value is entered, this should be stored globally so that every other component has access to it.

2. Level manager

   Since a large portion of the game's functionality will be handled by the level manager, it should be tested heavily. Its main functionality to be tested is how it runs games and passes input to them based on certain conditions, such as how previous levels were played and completed.

   Examples of tests for level manager:

   - If a level is successfully completed, the user should be prompted to open one of two doors. The test will enter input for its choice and expect a power up to be given.
   - Three tries are given to complete a puzzle. Ensure that if these are used up, the "game over" executable is run and the game resets.
   - If power ups are present, they should be active when a game starts. For example, if an "increased timer" power up was obtained, it should be active in the next level.
   - The conditions by which previous levels were completed should have an effect on the final level. This should be passed in as input when running the final level.
   - If the final level is completed, the "end game" executable should be run.

3. Each puzzle

   Every puzzle will be different, so it should be tested based on the specifications described in the requirements document. There are also generic tests that can be applied to any puzzle. For example, the timer running out or the in-game menu being accessed should be simulated to make sure the correct thing happens.

   Examples of tests for puzzles:

   - Ensure that if the "M" key is pressed from within the puzzle, the in-game menu is shown.
   - Ensure that if a timer is set for 180 seconds, and 181 system seconds pass, the puzzle ends.
   - If valid input is given to solve the puzzle, the puzzle should end as being successfully completed.

4. In-game menu

   To test the in-game menu, we'll first start its executable in the context of a dummy game, then run various inputs on it to make sure it performs correctly.

   Examples of tests:

   - Let the timer be for 120 seconds, and test what occurs after 121 system seconds pass. Since this is more than the allowed time, the game should now fail. Ensure the game-over screen is shown.

5. Notepad

   Examples of tests:

   - If input is typed into the notepad, it should be stored in a text document. When the notepad is run next, the same text should be shown.
   - If text is deleted, this should also be reflected in the text document.