SE 3XA3: Test Plan
Rogue Reborn

Group #6, Team Rogue++

Ian Prins	prinsij
Mikhail Andrenkov	andrem5
Or Almog	almogo

Due Monday, October 31st, 2016

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	Version	Notes
10/21/16	0.0	Initial Setup
10/24/16	0.1	Added Unit Testing and Usability Survey
10/24/16	0.2	Added Most of Section 2
10/24/16	0.3	Added Section 1
10/26/16	0.4	Added PoC tests
10/26/16	0.4.1	Added Test Template
10/30/16	0.5	Added Non-Functional Req. Tests
10/30/16	0.5.1	Added Bibliography
10/31/16	0.6	Added Names to Test Template

1 General Information

1.1 Purpose

The purpose of this document is to explore the verification process that will be applied to the Rogue Reborn project. Interested stakeholders are welcome to view and critique this paper to gain confidence in the success of the final product. After reviewing the document, the reader should understand the strategy, focus, and motivation behind the efforts of the Rogue++ testing team.

1.2 Scope

This report will encompass all technical aspects of the testing environment and implementation plan, as well as other elements in the domain of team coordination and project deadlines. The document will also strive to be comprehensive by providing context behind critical decisions, motivating the inclusion of particular features by referring to the existing *Rogue* implementation, and offering a large variety of tests for various purposes and hierarchical units. Aside from the implementation, the report will also discuss a relevant component from the requirements elicitation process (and its relevance to the testing effort).

1.3 Acronyms, Abbreviations, and States

Table 2: **Table of Abbreviations and Acronyms**

Abbreviation	**Definition**
GUI	Graphical User Interface
IM	Instant Messenger
PoC	Proof of Concept
VPS	Virtual Private Server

Table 3: **Table of Definitions**

Term	**Definition**
Boost	C++ utility library that includes a comprehensive unit testing framework
Frame	An instantaneous "snapshot" of the GUI screen
Libtcod	Graphics library that specializes in delivering a roguelike experience
Monochrome Luminance	The brightness of a given colour (with respect to the average sensitivity of the human eye)
Permadeath	Feature of roguelike games whereby a character death will end the game
Roguelike	Genre of video games characterized by ASCII graphics, procedurally-generated levels, and permadeath
Slack	An online communication platform specializing in team and project coordination

Table 4: **Table of States**

State	**Definition**
Developer State	The file system state corresponding to the latest source code revision from the GitLab repository
Fresh State	The file system state corresponding to a "fresh" Rogue Reborn installation
Gameplay State	Any application state that reflects the actual gameplay
Generic State	The file system state corresponding to a functional (working) installation of Rogue Reborn
High Score State	Any application state that reflects the top high scores screen
Menu State	Any application state that reflects the opening menu
Seasoned State	The file system state corresponding to an installation of Rogue Reborn that already contains several high score records

1.4 Overview of Document

The early sections of the report will describe the testing environment and the logistic components of the Rogue Reborn testing effort, including the schedule and work allocation. Next, a suite of tests will be discussed with respect to the functional requirements, non-functional requirements, and the PoC demonstration. Upon discussing the relevance of this project to the original *Rogue*, a variety of unit testing strategies will be given followed by a sample usability survey to gauge the interest and opinion of the Rogue Reborn game. A breakdown of the sections is listed below:

§1 Brief overview of the report contents

§2 Project logistics and the software testing environment

§3 Description of system-level integration tests (based on requirements)

§4 Explanation of test plans that were inspired by the PoC demonstration

§5 Comparison of the existing *Rogue* to the current project in the context of testing

§6 Outline of the approach to be implemented for unit testing

§7 Appendix for symbolic parameters and the usability survey

2 Plan

2.1 Software Description

Initially, a large component of the testing implementation involved the usage of *Boost*. In general, Boost is regarded as an industry standard C++ utility library and comes packaged with a great deal of documentation (**?**). However, this is a double-edged sword — Boost is heavy, globally encompassing, and requires plentiful effort to properly setup. The Boost library is suitable for projects spanning years with dedicated testing and QA teams. Unfortunately, this is not the present condition of the Rogue Reborn project, and with the project nearing completion over the next month, the Rogue++ team agreed that it would be unwise to start using Boost.

Instead, an alternative solution has been proposed: native test cases can be written in C++ to perform exactly the required tasks and nothing extra. The details of this implementation will be explained in the following sections.

2.2 Test Team

All members of the Rogue++ team will take part in the testing procedure. While Mikhail and Ian were assigned the roles of Project Manager and C++ Expert respectively, Ori was given the title of Testing Expert. Testing will be primarily monitored and maintained by Ori although every team member will contribute to the testing facilities. The logic behind this rationale is that it would be desirable for the team member who wrote class C to write the unit tests for the same class C. Due to the dependency structure of the project's design, there will be cases where a unit test for one class will encompass a partial system test for another class. These instances can be extrapolated from the class inheritance diagram.

2.3 Automated Testing Approach

There has been considerable effort expended towards automating project infrastructure components. In the real world, any task that *can* be automated, should be automated. The steps that have been performed to reduce manual labour are as follows:

- Set up a GitLab pipeline for the project. The pipeline is programmed to run a series of commands on an external VPS whenever a push is made to the GitLab repository. Every run is logged and its history may be accessed at any time.
- Write a special makefile that produces 2 executables:

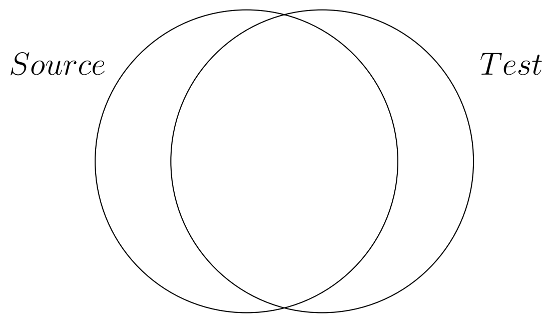
  1. The Rogue Reborn game executable
  2. The project test suite.

  The details of this process will be described in the following sub-section.
- The team's primary method of communication is Slack: a cross-platform and programmer-friendly IM. The Rogue++ team hooked the GitLab project repository to the team's Slack channel such that whenever the repository detects activity, a notification is sent to the channel. This method greatly improves the team's awareness about each other's contributions and also facilitates communication about project-related inquiries.

2.4 Testing Tools

The special makefile discussed above utilizes a phenomenon of C++ to perform the necessary steps. First, it places *all* source files into a dedicated folder to distinguish them between program files and test files; this is mandatory since there is an important relationship between the *source* and *test* classes. Consider the diagram below:

Figure 1: **Source and Test Relationship**



As the diagram depicts, there are classes that are shared between both final programs. In fact, the vast majority of classes fall in the center and are required by both the game executable as well as the testing component. The files that are necessary for the tests but not for the source are, obviously, testing-related files that contain the test case implementations. At the time of writing, there is only one file required by the source code that is not required by the test code: the source program entry (i.e. the C++ file that contains `main()`).

The entire procedure of file collection, compilation, and separate linking is handled by the makefile, and is triggered by the `make` command. From there, simply running `Test.exe` will trigger all of the pre-written tests.

There is also a plan to implement a Python script on the GitLab pipeline that will cause the build to fail if any of the tests do not pass. It should be noted that, if a build fails, the pipeline not only reports the failure, but also logs the location of the failure down to the specific test case. This will hopefully expedite the debugging process and lead to more responsible development further into the project timeline.

As an extra safety measure, the Rogue++ team will also be utilizing a tool called *Valgrind* in the testing procedure. Valgrind is a powerful analysis tool that tests the amount of memory a C++ program utilizes and detects memory allocation errors such as memory leaks (**?**). C++, unlike Java and other high level languages, does not include a built-in garbage collector (otherwise there would be nothing left!) to give programmers total control over their application lifetime. Consequently, it is a common mistake to accidentally leave unreferenced objects in memory and cause a memory leak in the program.

At the time of writing, the Rogue Reborn application occupies approximately 1 MB of RAM during peak execution. Although this is a minute quantity, memory leaks are representative of a larger issue: incorrect code! By using Valgrind, the Rogue++ team will be able to detect the presence of these errors and indicate the direction of the next crucial bug fix.

2.5 Testing Schedule

The Gantt Chart can be accessed at this location.

3 System Test Description

3.1 Tests for Functional Requirements

3.1.1 Basic Mechanics

New game start - Functional Test # 1	
Type:	Dynamic / Manual / Black Box
Initial State:	Nothing running.
Input:	A new game is started.
Output:	The program is started.
Execution:	Either double-clicking the .exe or via terminal: *./RogueReborn.exe*.

Save game - Functional Test # 2	
Type:	Dynamic / Manual / Black Box
Initial State:	Game screen
Input:	Save command is given or save key is pressed.
Output:	A message saying that the game has been saved is shown to the user in the status box.
Execution:	A user will have to play the game and trigger the input sequence. This process can be verified to work by the following test.

Load game - Functional Test # 3	
Type:	Dynamic / Manual / Black Box
Initial State:	Game screen
Input:	Load command is given or save key is pressed.
Output:	A message saying that the game has been loaded is shown to the user in the status box. The data model (level, player, monsters, etc.) is also updated to reflect the state changes.
Execution:	A user will have to play the game and trigger the input sequence to load, and verify that it is in fact the same state that was previously saved.

New game starting statistics - Functional Test # 4	
Type:	Dynamic / Automatic / Black Box
Initial State:	Nothing running.
Input:	A new game is started.
Output:	The player has the default starting gear and statistics.
Execution:	This feature can be tested by analyzing a save file. In the file is listed everything about the player, meaning the information can be attained from there.

Help command - Functional Test # 5	
Type:	Dynamic / Manual / Black Box
Initial State:	Game screen
Input:	The ”help” command is given or the ”help” key is pressed.
Output:	The user is shown a screen with a list of possible actions and other information
Execution:	Players will be given the game with no instructions or guide. The usefulness and accessibility of the help screen will be judged by their performance after having seen the help screen.

3.1.2 Interaction

Detailer player information - Functional Test # 6	
Type:	Dynamic / Manual / Black Box
Initial State:	Game screen.
Input:	None.
Output:	Details about the player (such as level, health, known status effects, current depth, etc.) are displayed at the bottom of the screen, in the area known as the ”Info bar”.
Execution:	At random points during the playtest, players will be asked to answer basic questions about their player. To answer these questions, the player will have to refer to the info bar.

Environment inspection - Functional Test # 7	
Type:	Dynamic / Manual / Black Box
Initial State:	Game screen.
Input:	The "look" key or command, and then an environment aspect character.
Output:	After the input is supplied, a brief description of the environment aspect is supplied. This can be limited to a word or two (i.e. "This is an Emu").
Execution:	Players will be told about the "look" key before starting, and will have to employ it to get to know their surroundings.

Pass turn - Functional Test # 8	
Type:	Dynamic / Manual / Black Box
Initial State:	Game screen.
Input:	The player wishes to skip his turn. This is usually the case if an enemy is about to move perpendicularly to the player's pre-determined projectile path, which will place the enemy in the direction of the player's projectile.
Output:	All entities but the player act, performing whatever action their AI has instructed them to perform.
Execution:	Players will be asked to skip their turn several times once an enemy is spotted.

Trap activation - Functional Test # 9	
Type:	Dynamic / Manual / Black Box
Initial State:	Game screen.
Input:	A dungeon level that can generate traps (This only occurs in the deeper levels).
Output:	A message and effect describing what the trap has done.
Execution:	Players will be asked to report any trap they come across and the effect it has bestowed upon them.

3.1.3 The Dungeon

Staircase guarantee - Functional Test # 10	
Type:	Dynamic / Automatic / Black Box
Initial State:	Nothing running.
Input:	A randomly generated dungeon (preferably many).
Output:	An assertion that all contain a downwards staircase.
Execution:	The algorithm for this is rather straight-forward; it is a simple BFS or DFS touring every passable block in the dungeon.

Connectedness & Reachability - Functional Test # 11	
Type:	Dynamic / Automatic / White Box
Initial State:	Nothing running.
Input:	A randomly generated dungeon (preferably many).
Output:	An assertion that the dungeon is connected and all tile are reachable from one-another.
Execution:	Again, another simple algorithm. A BFS or DFS can acquire a list of all passable tiles in the dungeon, which can be compared to the list provided by the source-code. If the two lists match, then the assertion is true.

Line of Sight - Functional Test # 12	
Type:	Dynamic / Manual / Black Box
Initial State:	Game screen.
Input:	Player is somewhere in the dungeon that is recognizable (i.e. not hidden), and player is not blind.
Output:	Visibility dependent on surroundings. If in a room, the player should be able to see the entire room. If in a corridor, the player should only be able to see in a 3x3 square centered on the player.
Execution:	Players will be asked to assess the visibility standards. This is a bug-prone feature, as many exceptions exist in the realm of ”What is the player on?”.

Amulet of Yendor - Functional Test # 13	
Type:	Dynamic / Automatic / White Box
Initial State:	Nothing running.
Input:	Levels generated with a depth of 26
Output:	A correct assertion that all levels generated contain the amulet somewhere on the level.
Execution:	It only takes a double-nested for-loop to make sure that somewhere in the level, on a passable tile, the amulet exists. Any since we already know that every passable tile is reachable, we know that the amulet is as well.

Searching & Finding - Functional Test # 14	
Type:	Dynamic / Manual / Black Box
Initial State:	Player in the dungeon beside a hidden door/passage.
Input:	The player activates the ”search” command to look around.
Output:	The door or passage is either revealed or stays hidden.
Execution:	Players will be told before the game begins to occasionally look out for hidden doors, as they are normally fairly hard to find. Once found, players will document the number of searches they needed to uncover the hidden door.

3.1.4 Equipment

Inventory tracking - Functional Test # 15	
Type:	Dynamic / Manual / Black Box
Initial State:	Game screen.
Input:	New players are instructed to play the game with no special requirements.
Output:	No player experiences a situation in which their inventory is mis-represented. All items collected by the player should be kept track of and indexed.
Execution:	Players will be asked to maintain, on a piece of paper, their inventory, and at the end of the game compare their copy to that of the game.

Identification & Naming - Functional Test # 16	
Type:	Dynamic / Manual / Black Box
Initial State:	Game screen.
Input:	Players are instructed to pronounce the names of all items they collect.
Output:	Players are **not able** to pronounce items they have yet to *identify*.
Execution:	To test the terribleness of the randomly-generated names, players will be asked to try and pronounce them. While some may succeed, the names will all be utterly nonsensical.

Armor & Deterioration - Functional Test # 17	
Type:	Dynamic / Manual / Black Box
Initial State:	Game screen.
Input:	Players are assured that no bad thing could happen to their armor.
Output:	Players should complain that their armor is somehow being damaged.
Execution:	Aquators and traps are able to destroy player armor. Approximately at level 6, players will start finding such setbacks, and report their results.

3.1.5 Combat

Monster AI - Functional Test # 18	
Type:	Dynamic / Automatic / White Box
Initial State:	Nothing running.
Input:	A target position to chase, given to all monsters in the dungeon.
Output:	(Most) monsters calculating ideal paths towards the target specified. Some monsters have a different expected behavior.
Execution:	An automatic script can easily be created to generate a level, plant some monsters in it, and simulate a player character somewhere on the map. Then, a traceback log of monster paths could be created and analyzed, by having the player simulation always skip its turn. This way enemies will have a non-moving target to path to.

Monster attack pattern - Functional Test # 19	
Type:	Dynamic / Automatic / Black Box
Initial State:	Nothing running.
Input:	No target for monsters to attack.
Output:	Monsters roaming around pointlessly, waiting for something to do.
Execution:	Like the previous test, a level could be generated and populated with enemies. Unlike the previous test case, however, no player will be supplied in this level. Monsters should aimlessly wander the halls of the dungeon and find no meaning or purpose.

3.2 Tests for Non-Functional Requirements

3.2.1 Look and Feel Requirements

Aesthetic Similarity Check - Non-Functional Test # 1	
Type:	Dynamic / Manual / Black Box
Initial State:	Generic State
Input:	Users are asked to rate the aesthetic similarity between *Rogue* and Rogue Reborn.
Output:	A numeric quantity between 0 and 10, where 0 indicates that the graphics are entirely disjoint and 10 indicates that the graphics are virtually indistinguishable.
Execution:	A random sample of users will be asked to play *Rogue* and the Rogue Reborn variant for PLAYTEST_SHORT_TIME minutes. Afterwards, they will be asked to judge the graphical similarity of the games based on the aforementioned scale.

3.2.2 Usability and Humanity Requirements

Interest Gauge Check - Non-Functional Test # 2	
Type:	Dynamic / Manual / Black Box
Initial State:	Generic State
Input:	New users are instructed to play Rogue Reborn.
Output:	The quantity of time the user willingly decides to play the game.
Execution:	A random sample of users who are unfamiliar with *Rogue* will be asked to play Rogue Reborn until they feel bored (or MAXIMUM_ENTERTAINMENT_TIME has expired). Once the user indicates that they are no longer interested in the game, their playing time will be recorded.

English Mechanics Check - Non-Functional Test # 3	
Type:	Static / Manual / White Box
Initial State:	Developer State
Input:	Rogue Reborn source code.
Output:	An approximation of the English spelling, punctuation, and grammar mistakes that are visible through the GUI.
Execution:	All strings in the Rogue Reborn source code will be concatenated with a newline delimiter and outputted to a text file. A modern edition of Microsoft Word from (?) will be used to open this generated text file, and a developer will manually correct all of the indicated errors that are potentially associated with a GUI output.

Key Comfort Check - Non-Functional Test # 4	
Type:	Dynamic / Manual / Black Box
Initial State:	Generic State
Input:	Users are asked to rate the intuitiveness of the Rogue Reborn key bindings.
Output:	A numeric quantity between 0 and 10, where 0 indicates that the key bindings are extremely confusing and 10 indicates that the key bindings are perfectly natural.
Execution:	A random sample of users who are inexperienced with the roguelike genre will be asked to play Rogue Reborn for SHORT_TIME minutes without viewing the in-game help screen. Next, the key bindings will be revealed, and the users will continue to play the game for an additional PLAYTEST_SHORT_TIME minutes. Afterwards, they will be asked to judge the quality of the key bindings based on the aforementioned scale

3.2.3 Performance Requirements

Response Delay Check - Non-Functional Test # 5	
Type:	Dynamic / Automatic / White Box
Initial State:	Generic State
Input:	Users are instructed to play Rogue Reborn.
Output:	A log of occurrences that indicate events where a computation that was initiated by a user input took an excessive quantity of time to execute.
Execution:	A random sample of experienced users will be asked to play a special version of Rogue Reborn for PLAYTEST_MEDIUM_RANGE minutes. This edition will utilize a StopWatch implementation to measure the execution time of a computation, and if the computation exceeds RESPONSE_SPEED milliseconds, the user action and the associated timestamp will be recorded in a log file.

Overflow Avoidance Check - Non-Functional Test # 6	
Type:	Static / Manual / White Box
Initial State:	Developer State
Input:	Rogue Reborn source code.
Output:	All declarations of integer-typed variables.
Execution:	All occurrences of lines that match REGEX_INTEGER (i.e., integer declarations) in the Rogue Reborn source code will be outputted to a file. A group of Rogue++ developers will then review these declarations together and alter them if deemed necessary to avoid integer overflow issues.

Crash Collection Check - Non-Functional Test # 7	
Type:	Dynamic / Manual / Black Box
Initial State:	Generic State
Input:	Playtesters are instructed to play Rogue Reborn for at least PLAYTEST_LONG_TIME hours.
Output:	A collection of crash occurrences along with a detailed description of the failure environment.
Execution:	All Rogue Reborn playtesters will be required to play the game for at least PLAYTEST_LONG_TIME hours in total (spanned over multiple sessions if desired). Every time the application crashes, the playtester must record the incident along with a description of the visible GUI state and the steps required to reproduce the failure. After this data has been collected, the Rogue++ team will address every crash occurrence by either resolving the issue or confidently declaring that the event is irreproducible.

Score Overflow Check - Non-Functional Test # 8	
Type:	Dynamic / Dynamic / White Box
Initial State:	High Score State
Input:	A high score record file containing a large quantity of entries.
Output:	Rogue Reborn GUI displaying the top high scores.
Execution:	The Rogue Reborn developers will artificially fabricate a high score record file with at least HIGH_SCORE_CAPACITY + 2 records. The game will then be played until the high score screen is revealed; only the top HIGH_SCORE_CAPACITY scores should be displayed.

3.2.4 Operational and Environment Requirements

Processor Compatibility Check - Non-Functional Test # 9	
Type:	Dynamic / Manual / Black Box
Initial State:	Fresh State
Input:	Users are instructed to install and run Rogue Reborn on their personal machines.
Output:	An indication of whether or not the game is able to successfully execute.
Execution:	A random sample of users with computers that are equipped with Intel x64 processors will be asked to download the latest Rogue Reborn distribution, perform any necessary installation, and then run the executable file. The user will then report if the game was able to successfully run on their machine.

Streamline Distribution Check - Non-Functional Test # 10	
Type:	Static / Manual / Black Box
Initial State:	Developer State
Input:	Rogue Reborn distribution package.
Output:	An indication of whether or not the distribution contains any files aside from the primary executable and the associated development licenses.
Execution:	The public distribution package will be visually inspected for extraneous files.

3.2.5 Maintainability Requirements

Bug Productivity Check - Non-Functional Test # 11	
Type:	Static / Manual / Black Box
Initial State:	Developer State
Input:	All ITS issues labeled as bugs in the Rogue Reborn GitLab repository.
Output:	An indication of whether or not all bug reports were closed within a month of their conception.
Execution:	The Rogue Reborn GitLab repository will be queried for all issues concerning bugs (which are denoted by a "Bug" label). Next, a developer will manually verify that every closed bug fix request was resolved within a month of its creation.

Linux Compatibility Check - Non-Functional Test # 12	
Type:	Dynamic / Manual / Black Box
Initial State:	Fresh State
Input:	Users are instructed to run Rogue Reborn on their personal machine.
Output:	An indication of whether the game can successfully execute.
Execution:	A random sample of users with computers that use a modern 64-bit Linux operating system will be asked to download the latest Rogue Reborn distribution, perform any necessary installation, and then run the executable file. The user will then report if the game was able to successfully run on their machine.

3.2.6 Security Requirements

Illegal Records Check - Non-Functional Test # 13	
Type:	Dynamic / Manual / White Box
Initial State:	Seasoned State
Input:	A corrupted high score record file.
Output:	Rogue Reborn GUI displaying the top high scores.
Execution:	The Rogue++ team will illegally modify a high score record file by manually altering or adding values such that the expected format or value integrity is violated. These modifications should include negative high score values, missing text, and incorrect delimiter usage. The game will then be played until the high score screen is revealed; all invalid record file contents should be ignored and amended in the next write to the record file.

3.2.7 Legal Requirements

License Presence Check - Non-Functional Test # 14	
Type:	Static / Manual / Black Box
Initial State:	Developer State
Input:	Rogue Reborn distribution package.
Output:	An indication of whether or not the distribution is missing any mandatory license files.
Execution:	The original *Rogue* source code hosted by (?) will be reviewed for legal requirements, and the public distribution package will be visually inspected to ensure that all mandatory license files are present.

3.2.8 Health and Safety Requirements

Seizure Prevention Check - Non-Functional Test # 15	
Type:	Dynamic / Manual / Black Box
Initial State:	Developer State
Input:	Two screenshots denoting the largest possible luminosity difference present between consecutive frames.
Output:	The difference in luminosity between the two captured frames.
Execution:	After identifying the frame pair that is most likely to induce a seizure, the game will be played to reach the states that reflect each frame (this should be a brief process; no clever game model manipulation is required). At the occurrence of each desired frame, the game screen will be captured and saved. At this point, the average monochrome luminance across each frame will be calculated according to the formula $$L = 0.299R + 0.587G + 0.114B$$ where L is the luminance, R is the red RGB component, G is the green RGB component, and B is the blue RGB component (**?**). Finally, the absolute value of the luminance difference can then compared to LUMINOSITY_DELTA.

4 Tests for Proof of Concept

4.1 Static Testing

Compile Test - PoC Test # 1	
Type:	Static / Automatic / White Box
Initial State:	None
Input:	Program Source
Output:	Program Executable
Execution:	Verify that the program compiles with g++.

Memory Check - PoC Test # 2	
Type:	Dynamic / Manual / White Box
Initial State:	None
Input:	A brief but complete playthrough of the game.
Output:	Breakdown of program memory usage.
Execution:	A tester will briefly play the game, and a developer will use Valgrind's memcheck utility to verify that program does not leak memory or utilize uninitialized memory.

4.2 Rendering

Render Check - PoC Test # 3	
Type:	Dynamic / Manual / l Box
Initial State:	Black
Input:	Gameplay State
Output:	30-60 seconds of gameplay.
Execution:	The player character and any dungeon features should be shown at the correct location with the correct glyphs. Correct player statistics will be shown along the bottom. The dialog box will correctly display the log and any prompts.

A tester will manually play the game and verify the display is correct.

4.3 Dungeon Generation

Dungeon-Gen Check - PoC Test # 4	
Type:	Dynamic / Manual / Black Box
Initial State:	None
Input:	Repeated restarts of the game
Output:	Level should contain ROOMS_PER_LEVEL rooms, which should form a connected graph.
Execution:	A tester will manually start the game, briefly explore the level to verify correct generation, then repeat this process until confidence is achieved.

4.4 Basic Movement

Movement Check - PoC Test # 5	
Type:	Dynamic / Manual / Black Box
Initial State:	Gameplay State
Input:	Movement commands
Output:	Player should move about the level, without clipping through walls, failing to walk through empty space, or jump to an unconnected square.
Execution:	A tester will manually walk through the level, and visually verify correctness.

4.5 Score File

Scoring File Check - PoC Test # 6	
Type:	Dynamic / Manual / Black Box
Initial State:	Menu State
Input:	Enter name, then quit, restart game, enter name again, and quit.
Output:	1st name should appear in both the first and second score screens. The 2nd should appear in the second. Both should have correct values for level, cause of death/quit, and gold collected.
Execution:	A developer will manually perform the above input, and verify the output. Should be tested both with and without an initial score file.

4.6 Line of Sight System

LoS Check - PoC Test # 7	
Type:	Dynamic / Manual / Black Box
Initial State:	Gameplay State
Input:	Movement commands
Output:	Screen should display correct portions of level, with correct coloration schemes. This means that the player should be able to see the entirety of a room they are in or in the doorway of, and VIEW_DISTANCE squares away if they are in a corridor. Squares that the player has seen in the past but cannot see currently should be shown greyed out. Squares they have not seen should be black and featureless.
Execution:	A developer will manually walk through the level, verifying that the above LoS rules are preserved, especially in edge cases like the corners of rooms and doorways.

5 Comparison to Existing Implementation

The original *Rogue* is feature-full, and luckily, open source. This means that many, if not all of the features in Rogue++ can be tested in accordance with their similarity to the original game. Some examples are discussed below.

An attempt has been made to replicate nearly one-for-one the items, loot, and treasure obtainable in the original *Rogue*. Wands, staffs, rings, potions, ammunition, weapons, armor and more were all copied over with the same values in place. Regarding the items available for collection, players of the original game should feel right at home with the new Rogue++. Unlike some more modern games, the original *Rogue* does not specify how effective an attack is besides hit or miss, as does Rogue++. This means that a player experienced with the original game may expect certain behavior out of a weapon or item, and find a difference in its effectiveness, despite the near one-to-one transition. This could stem from a variety of sources, perhaps most likely of which is a piece of code that does something unexpected, in an unexpected place.

Another aspect of the game that was replicated as the source-code describes is dungeon generation. Of course, today we are using a more advanced data structure, with several capabilities that were not available for the C of 1980, but the idea behind the data structure is the same. The process followed for dungeon generation in 1980 was somewhat ill-conceived and convoluted. Despite this its discernible aspects were used as inspiration for the algorithm used in Rogue++. So while at the end of the day the two do not follow the same algorithm, the end result is close enough, and test cases are included to make sure that all properties of the old *Rogue* are satisfied in Rogue++.

Another way Rogue++ can be compared to the original *Rogue* is by its controls. This is something that can be algorithmically tested, and guaranteed to function exactly as intended. Every key in *Rogue* is mapped to a specific action, which can be replicated one-for-one in Rogue++. WIP

6 Unit Testing Plan

After examining the boost library's utilities for unit testing, we have decided we will not use a unit testing framework for testing the product. We concluded that adding a framework would not make the work significantly easier, while reducing our flexibility and adding installation difficulties. Since we are not using a framework, drivers will be written by hand. Stubs will be produced when necessary to simulate system components. Since there are no database or network connections, stubs should hopefully be kept to a minimum. However, functions may be required to construct objects in states suitable for easy testing, for example creating a level or player with certain known properties, rather than by random generation.

6.1 Unit testing of internal functions

Internal functions in the product will be unit tested. This will be reserved for more complex functions so as to not waste development time unnecessarily. As complete code coverage is not a goal, generic code coverage metrics will not be used. Instead, care will be taken that complex functions are covered by unit tests. The following are examples of internal functions that are initial candidates for unit testing. Other functions will be added as necessary:

- The dungeon generation functions. The work of generating the dungeon is complex, but it is also easy to automate verification of dungeon properties such as a correct number of rooms, connectedness, compliance with formulas for item generation, presence or absence of certain key features such as the stairs connecting levels or the Amulet of Yendor in the final level.
- The keyboard input functions. As libtcod provides a Key struct which models keyboard input, we can mock/automate these functions. They are fairly complex, and since they return a pointer to the next desired state (similar to a finite state machine) we can easily verify their behavior.
- The item activation functions. For example it could be verified that when the player drank a potion of healing their health increased (if it was not at its maximum), that a scroll of magic-mapping is reveals the level, or that a scroll of identification reveals the nature of an item.

- The item storage functions. Each item is mapped to a persistent hotkey in the player's inventory. Certain items can stack with copies, reducing the amount of inventory space they take up, and how they are displayed. These factors make the inventory fairly complex. It is however easily verifiable, and automated testing can examine edge cases that would be impractical to test manually.

6.2 Unit testing of output files

There is only one output file for the product, the high score file, which stores the scores in a csv format. The production and reading of this file can be unit-tested by verifying its contents after writing to it, and by providing a testing version of the file with known contents and verifying the function reads them correctly.

7 Appendix

This is where you can place additional information.

7.1 Symbolic Parameters

Table 5: **Symbolic Parameter Table**

Parameter	**Value**
ROOMS_PER_LEVEL	9
FINAL_LEVEL	26
HEIGHT_RESOLUTION	400
LUMINOSITY_DELTA	0.5
MINIMUM_ENTERTAINMENT_TIME	20
MINIMUM_RESPONSE_SPEED	30
HIGH_SCORE_CAPACITY	15
PLAYTEST_SHORT_TIME	5
PLAYTEST_MEDIUM_RANGE	10-20
PLAYTEST_LONG_TIME	3
REGEX_INTEGER	`(char|int|long).*(,|;)`
START_LEVEL	1
VIEW_DISTANCE	1
WIDTH_RESOLUTION	1280

7.2 Usability Survey Questions

1. Is there any game feature you were unable to figure out how to utilize?
2. How helpful was the help screen for you?
3. Was there anything going on in the game that the interface failed to make clear to you or deceived you about?
4. What common UI interactions did you find particularly lengthy?
5. What aspects of the interface did you find unintuitive?
6. How responsive was the interface?
7. How easy was it to see everything that was going on?
8. How effective are the graphics/symbols?
9. Would an alternative input device such as a mouse make interacting with the interface easier for you?
10. Is there any extra functionality you would like added to the interface?
11. How difficult was it to learn the game? How much experience do you have with Roguelikes?
12. How helpful was the original game manual?
13. How pleasing was the color scheme?
14. Was the font large enough for easy use?
15. Were you able to learn the hotkeys easily?