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	Switch PoC 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. 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.

1.3 Acronyms, Abbreviations, and States

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

Abbreviation	**Definition**
GUI	Graphical User Interface
PoC	Proof of Concept

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 emulating 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

Table 4: **Table of States**

State	**Definition**
Developer State	The file system state corresponding to the latest source code revision from the Git repository
Fresh State	The file system state corresponding to a “fresh” Rogue Reborn installation
Gameplay State	Any application state that reflects the actual gameplay
High Score State	Any application state that reflects the top high scores screen
Menu State	Any application state that reflects the opening menu
Public Test State	The system state corresponding to an installation of Rogue Reborn that is shared by a subset of the public game testers

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, nonfunctional requirements, and proof of concept demonstration. Upon discussing the relevance of this project to the original *Rogue*, a variety of unit tests will be given followed by a sample usability survey to guage 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 module-level unit tests
- §7 Appendix for symbolic parameters and the aforementioned usability survey

2 Plan

2.1 Software Description

Initially, the plan for testing involved the usage of a pre-made testing system called Boost. Boost has industry renown and is very well documented. The drawback to using such a profound system is exactly its advantage - it is heavy, globally encompassing, and requires a lot of work to use properly. The Boost library is suitable for projects spanning years, with dedicated testing teams. This is not the present situation. With hardly over a month until the completion of the project, starting to use Boost would be most unwise.

Instead, an alternative solution has been proposed and implemented. Native test cases can be written in C++ to do exactly that which is required. The details of this implementation will be explained in the parts to follow.

2.2 Test Team

All members of the team will take part in the testing procedure. While Mikhail was given the title of project manager, and Ian C++ expert, Ori was assigned the role of testing expert. Testing will be monitored by Ori, but of course every member of the team will contribute to the testing facilities. It would be desirable for the team member who wrote class C to write the unit tests for this class. Due to the dependency-tree-like structure of the project's design, there will be cases where a unit test for one class encompasses a partial system test for another one. This can be extrapolated from the class inheritance diagram.

2.3 Automated Testing Approach

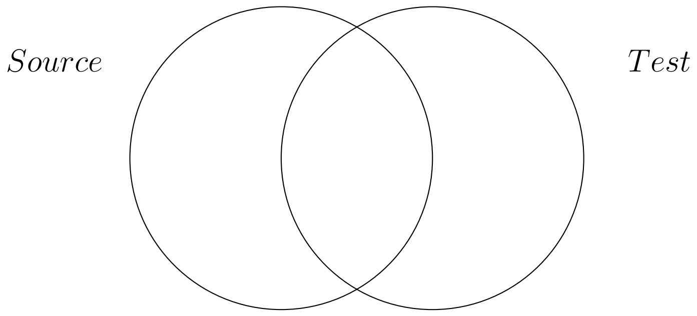
We have made a very large attempt at automating whatever we could for this project. In the real world, any task that *can* be automated, is automated. The steps we have taken are as follows:

- 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 git repository. Each run is documented and its history may be accessed.

- Write a special makefile that outputs 2 executables: the first being the actual project, and the second the project's tests. The details will be delved into in the following sub-section.
- The team's primary method of communication is Slack, a cross-platform, programmer-friendly chat interface. We hooked up the GitLab project repository to the Slack channel such that whenever a push is made or an issue addressed, a notification is sent. This method makes it far easier to communicate about project-related inquiries.

2.4 Testing Tools

The special makefile discussed previously utilizes a phenomenon of C++ to perform the necessary steps. First, it places *all* source files into a dedicated folder, distinguishing between program files and test files. This is an absolutely necessary step, as there is an important relationship between *source* and *test* classes. See the diagram below:



As the diagram above depicts, there are classes shared between both final programs. The vast majority of classes fall in the center, required by both the final project and its testing component. The files required by the test which are not required by the source are, obviously, testing-related files. These are the files that contain the test case implementations. At the time of writing, there is actually only one file required by source that is not required by the test, and that is the source program entry (i.e. the file that contains the main() method).

The entire procedure of file collection, compilation, and separate linking is handled by the makefile, and is triggered by the "make" command. Then, simply running Test.exe will fire off all of the pre-written tests.

There is 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. At the time of writing this document, it is not yet implemented, but note will be made when it does. It should be noted that if a build fails, the pipeline not only reports the failure, but also logs where the failure happened, down to the specific test case. This will hopefully make debugging a more pleasant experience later on.

As an extra safety measure, the Rogue++ team will also be utilizing a tool called Valgrind in the testing procedure. Valgrind is a tool that tests the amount of memory a C++ program utilizes, and detects memory allocation errors (such as memory leaks). This is an extremely useful and powerful tool. C++, unlike Java and other high level languages, does not have a built-in garbage collector. This is just one of the reasons why it is so much faster than the rest. A consequence of this, however, is that it is very easy to accidentally leave behind an object in memory, causing a memory leak in the program.

At the time of writing, the entire program occupies 1 MB of memory. This is not much, and even if it was all left behind in a leak, the system would not be too hindered. However, memory leaks represent only a consequence of a larger issue: incorrect code! Using Valgrind, we will be able to detect these kinds of errors, potentially pointing us in the direction of a crucial bugfix.

2.5 Testing Schedule

See Gantt Chart at the following url ... TODO

3 System Test Description

3.1 Tests for Functional Requirements

3.1.1 Basic Mechanics

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*.

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.

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.

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.

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

Functional Test # 6	
Type:	Type / ManAut / Color Box
Initial State:	InitState
Input:	Input
Output:	Output
Execution:	Execution

Functional Test # 7	
Type:	Type / ManAut / Color Box
Initial State:	InitState
Input:	Input
Output:	Output
Execution:	Execution

Functional Test # 8	
Type:	Type / ManAut / Color Box
Initial State:	InitState
Input:	Input
Output:	Output
Execution:	Execution

Functional Test # 9	
Type:	Type / ManAut / Color Box
Initial State:	InitState
Input:	Input
Output:	Output
Execution:	Execution

3.2 Tests for Non-Functional Requirements

3.2.1 Look and Feel Requirements

Non-Functional Test # 1	
Type:	Dynamic / Manual / Black Box
Initial State:	Public Test 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 a piece. 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

Non-Functional Test # 2	
Type:	Dynamic / Manual / Black Box
Initial State:	Public Test 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 they indicate that they no longer wish to play, their playing time will be recorded.

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 can be visible from the GUI.
Execution:	All strings in the Rogue Reborn source code will be concatenated with a newline delimiter and placed in a text file. A modern edition of Microsoft Word will be used to open this generated text file, and a developer can then manually correct all indicated errors that are potentially associated with a GUI output.

Non-Functional Test # 4	
Type:	Dynamic / Manual / Black Box
Initial State:	Public Test 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 key binding help screen. Next, the key bindings will be revealed and the users will continue to play 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

Non-Functional Test # 5	
Type:	Dynamic / Automatic / White Box
Initial State:	Public Test State
Input:	Users are instructed to play Rogue Reborn.
Output:	A log of occurrences 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 version will use a StopWatch implementation to measure the execution time of a computation, and if such a computation exceeds RESPONSE_SPEED milliseconds, the user action and timestamp will be recorded in a log file.

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:	A recursive `grep` command will be used to capture all lines in the Rogue Reborn source code that match REGEX_INTEGER (i.e., integer declarations). A group of Rogue++ developers can review these declarations together and alter them if deemed necessary to avoid integer overflow issues.

Non-Functional Test # 7	
Type:	Dynamic / Manual / Black Box
Initial State:	Public Test State
Input:	Playtesters are instructed to play Rogue Reborn for at least PLAYTEST_LONG_TIME hours.
Output:	A collection of crash occurrences along with descriptions that explain how the failure occurred.
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). If the application crashes during any time, the user must record the incident along with a description of the visible GUI state and the steps required to reproduce the failure. The Rogue++ team must address each crash by either resolving the issue or confidently declaring that the event is irreproducible.

Non-Functional Test # 8	
Type:	Dynamic / Manual / White Box
Initial State:	High Score State
Input:	A high score record file containing a large quantity of entries.
Output:	Screen denoting 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. One round of the game will be played, and when the high score screen is revealed, only the top HIGH_SCORE_CAPACITY scores should be displayed.

3.2.4 Operational and Environment Requirements

Non-Functional Test # 9	
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 are equipped with Intel x64 processors will be asked to download the latest Rogue Reborn distribution and attempt to run the executable. The user will then report if the game successfully runs on their machine.

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 licenses.
Execution:	The public distribution package will be visually inspected for extraneous files.

3.2.5 Maintainability Requirements

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:	A list of all bug reports and their corresponding resolution date (if closed).
Execution:	The Rogue Reborn GitLab repository will be queried for all issues concerning bugs (which are denoted by a “Bug” label). A developer can then manually verify that every closed bug fix request was resolved within a month of its creation.

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 and attempt to run the executable. The user will then report if the game successfully runs on their machine.

3.2.6 Security Requirements

Non-Functional Test # 13	
Type:	Dynamic / Manual / White Box
Initial State:	High Score State
Input:	A corrupted high score record file.
Output:	Screen denoting the top HIGH_SCORE_CAPACITY (valid) 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 executed to reach the High Score State, where invalid record file contents should be ignored and amended in the next write to the file.

3.2.7 Legal Requirements

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 (as referenced on the Rogue Reborn GitLab homepage) will be reviewed for legal requirements, and the public distribution package will be visually inspected to ensure that all license files are present.

3.2.8 Health and Safety Requirements

Non-Functional Test # 15	
Type:	Static / Manual / Black Box
Initial State:	Developer State
Input:	Two screenshots denoting the largest possible luminosity difference present between two 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. At this point, the average monochrome luminance across each frame will be calculated according to $$L = 0.299R + 0.587G + 0.114B$$ 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

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

Proof of Concept 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

Proof of Concept 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

Proof of Concept Test # 4	
Type:	Dynamic / Manual / Black Box
Initial State:	None
Input:	Repeated restarts of the game
Output:	Level should contain ROOMSPER_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

Proof of Concept 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

Proof of Concept 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

Proof of Concept 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

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?