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

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 Symbols

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

Abbreviation	Definition
PoC	Proof of Concept

Table 3: **Table of Definitions**

Term	Definition
Boost	C++ utility library that includes a comprehensive unit testing framework
Libtcod	Graphics library that specializes in emulating a roguelike experience
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

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:
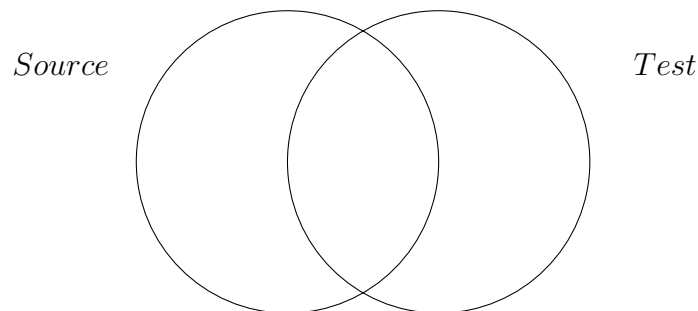
As the diagram above depicts, there are classes shared between both final programs. The vast majority of classes fall in the center, required by both the final project and its testing component. The files required by the test which are not required by the source are, obviously, testing-related files. 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.

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 Area of Testing1

Functional Test # 1	
Type:	Dynamic / Manual / Black Box
Initial State:	Menu screen
Input:	Alphanumeric keyboard interrupts
Output:	Valid characters that are pressed will appear on the screen beside the character name prompt
Execution:	Typing the name of the character on the keyboard

3.1.2 Area of Testing2

...

3.2 Tests for Non-Functional Requirements

3.2.1 Area of Testing1

3.2.2 Area of Testing2

...

Non-Functional Test # 1	
Type:	Dynamic / Manual / Black Box
Initial State:	Gameplay screen
Input:	Game model
Output:	Graphical display
Execution:	Visual inspection

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++

1. test-id2

   Type: Dynamic/Manual/Automatic

   Initial State: None

   Input: 30-60 seconds of gameplay

   Output: Breakdown of program memory usage.

   How test will be performed: Use valgrind memcheck utility to verify that program does not leak memory or utilize uninitialized memory.

7

4.2   Basic Movement

1. test-id1

   Type: Dynamic/Manual

   Initial State: Normal playstate

   Input: Movement commands

   Output: Player should move about the level, without clipping through walls or failing to walk through empty space

   How test will be performed: A developer will manually walk through the level

4.3   Score File

1. test-id1

   Type: Dynamic/Manual

   Initial State: Main menu

   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.

   How test will be performed: A developer will manually perform the above input, and verify the output.

4.4   Line of Sight System

1. test-id1

   Type: Dynamic/Manual

   Initial State: Normal playstate

   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 only their adjacent squares 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.

How test will be performed: 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.

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, connectness, 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.

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 4: **Symbolic Parameter Table**

Parameter	Value
FINAL_LEVEL	26
WIDTH_RESOLUTION	1280
HEIGHT_RESOLUTION	400
VIEW_DISTANCE	2
START_LEVEL	1
MINIMUM_ENTERTAINMENT_TIME	20
MINIMUM_RESPONSE_SPEED	30
HIGH_SCORE_CAPACITY	15
LUMINOSITY_DELTA	0.5

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?