Battle Tanks: Test Report

Compiled for version 1.3

Daniel G. Holmes 551240 and Jonathan D. Gerrand 349361

1 Introduction

This document seeks to provide its reader with an overview of the testing methodology and framework implemented in development of the <Battle Tanks> game. Several iterations of testing were performed throughout game development cycle and these are highlighted within the body of the report. Initially instructions are provided for replication of the testing environment, followed by an overview of the testing methodology applied. The testing groups are then defined and their intent is substantiated. A brief commentary then follows surrounding excluded test cases. This allows for extensions of the testing environment to be suggested.

2 Testing Environment

Table 1 provides the system setup specifications used during test development of the <Battle Tank> game.

Tab. 1: Setup specifications for the GoogleTest environment.

Operating System	IDE	gtest version	GCC version
Windows 7/8	Code::Blocks 12.11	1.6.0	4.7.1

The following subsections provide a description on the setup, use and extension of this testing environment.

2.1 Setup

The following steps can be used to setup gtest and to use it for testing-based development with the <Battle Tanks> source code:

1. Download gtest from the provided URL: https://code.google.com/p/googletest/downloads/list
2. Unzip the downloaded file and follow the instructions within the README file, this will provide details on compiling the provided binaries for your system.
3. Ensure that the relevant compiler and linker settings are configured within your IDE by adding *libgtest.a* and *libgtest_main.a* to the project build options.

2.2 Using and Extending the Test Environment

Following the successful completion of the above steps, the <Battle Tanks> source code can be run by opening the *ClassTests.cpp* found within game folder. To extend the testing framework, new tests conforming to the gtest syntax can be appended to the end of this file.

3 Testing Methodology

The <Battle Tanks> source code has been tested with a Test-Driven Development (TDD) strategy, implementing the unit-test based methodology with regression objectives. This arose from the nature of the source of the produced for the game which was highly modular and could be readily separated into Data, Logic and Presentation layers. The sub-classification of these class layers allowed tests to be grouped accordingly and provided for a logical progression of testing from the Data to the Logic layers. It is noted that due to time and framework limitations the presentation layer of the code was not tested. This is discussed later in section 5.

3.1 Test groups

3.1.1 Data Layer

Automatic unit tests were initially written for the Data layer of the source code. This layer comprised of all data-container classes and the base game entity units such as the Orientation, Tank, Missile, Mine, Barrier and Turret classes. These classes possess minimalistic logic and testing sought to ensure that the state of these classes remained invariant throughout the use of their constructors and member functions. Table 2 below provides a brief description of the tests performed upon each class within the Data layer.

Tab. 2: Summary of tests performed upon the Data classes of the <Battle Tank> game.

Class	Constructor Initialization	Movement and Rotation	Member Functions Maintain Invariance
Orientation	✓	✓	✓
Barrier	✓	·	✓
Missile	✓	✓	✓
Mine	✓	·	✓
Tank	✓	✓	✓
Turret	✓	✓	✓

3.1.2 Logic Layer

Within the <Battle Tanks> game, classes which where categorized within the logic layer were predominantly grouped as *Managers* and were assisted through the use of the *GeometryEngine* class. Automatic tests written for these classes sought to characterize the correct logical execution of their member functions and manipulation of the data that they acted upon. Due to a relationship similar to that of dependency injection, one which involved the managers being passed shared pointers to the data they acted upon instead of constructing this data themselves, unit tests could be performed successfully by constructing a entity whose state was known outside of the manager and then validating the resulting entity state once the manager had executed a management tick (a full iteration of its logical responsibilities). Table 3 below displays a summary of the functionality tested within the logic layer of the game.

Tab. 3: Summary of tests performed upon the Logic classes within the <Battle Tanks> game

Class	Functionality tested	Functionality validation detail.
MoveManager	Management tick	Unblocked entities update movement patterns. Blocked tanks do not move. Blocked missiles reflect correctly.
TrackingManager	Management tick	Tank and turret tracking coordinates update successfully, turrets set to fire when within range of tank entities.
DestructionManager	Management tick	All collided entities successfully deleted from the the game world.
CollidionManager	Management tick	Tank-barrier, tank-tank and missile-barrier collisions result in blocked state. Tank-missile, tank-mine and turret-missile collisions result in deletion state.
GameStateManager	Management tick	Tanks cannot lay or fire missiles before reload timer refreshes.
TurretManager	Management tick	Turrets are correctly rotated.
GeometryEngine	Vector calculations performed correctly.	Collisions occur correctly. Vector lengths calculated correctly. Positional relations correctly described.

4 Testing analysis

As of the current version <1.3> of the game, 79 tests from 13 test cases were written. All of these tests were passed by the source code. In evaluation of the tests performed, it is noted that a logical sequence of tests were written which first ensured the invariance of the data classes, and then tested the correct logical execution of logic-layer classes. Performing tests within this order ensured that logical processes acting on private data classes which failed to pass could be accurately diagnosed and corrected without ambiguity surrounding the integrity of the private data itself. As mentioned in subsection 3.1.2, the correct manipulation of private logic-owned data classes could also be effectively tested through a technique similar to that of dependency injection.

5 Further work

As stated within Section 3, a key area in which automatic testing has not been performed is that of the presentation layer. This testing largely surrounds the correct graphical display of the game world. Several predominant reasons exist for this and include the consequential testing of 3rd-party libraries (SFML) and the limitation of the unit test framework to perform visual or similarity-based testing. Noting this, strong recommendation is made for the implementation of such tests through the extension of the current testing environment.

6 Conclusion

Utilizing the gtest framework, 79 automatic unit tests have been written and passed. Testing ensures that all data and most logic based classes operate according to their defined functioning. Recommendation is made to extend the testing framework to incorporate the testing of the source code's presentation layer.