Physics-Based Chipmunk2D Game

Test Plan

Steven Palmer
⟨palmes4⟩
Emaad Fazal
⟨fazale⟩
Chao Ye
⟨yec6⟩

November 1, 2015

Contents

List of Tables

List of Figures

Revision History

Date	**Version**	**Notes**
October 25, 2015	1.0	Created document
October 31, 2015	1.1	Major additions to all sections

1 Overview

The purpose of this document is to provide a detailed plan for the testing of our game. The following brief outline gives and overview of what is covered in this document:

- A proof of concept test is described in §2.
- System testing is separated into game mechanics testing and game design testing. The set of tests that will be used in testing the system is described in §3.
- The set of tests that will be used to ensure that the software requirements specifications are met is described in §4.
- A timeline of the test plan is given in §5.

1.1 Test Case Format

The description of the tests that will be carried out are formatted in the following way throughout the document:

Test #:	**Test name**
Description:	A description of what is being tested
Type:	The type of test
Tester(s):	The people who will run the test (**manual only**)
Initial State:	The initial state of the system being tested (**unit test only**)
Input:	The input that will change the state of the system (**unit test only**)
Output:	The relevant output that is checked (**unit test only**)
Pass:	The pass criteria for the relevant output in the case of unit tests, or a description of the pass criteria for other tests

1.2 Automated Testing

Automated testing will be used for testing of the game mechanics system. Automated tests will include unit testing and coverage analysis.

1.2.1 Testing Tools

The software tools that will be used to carry out the automated testing are listed in Table 1

Table 1: List of testing tools

Tool	Description	Use
gUnit	Unit testing framework	Unit testing
COVTOOL	Test coverage analyzer	Analysis of unit test coverage

1.3 Manual Testing

Tests that are run manually will be carried out :

Game Developers Blah balh

Testing Group A group of δ individuals not involved in the development of the game. This group will used in two separate testing phases. Phase I

The user testing

1.4 List of Constants

Constants used in this document are listed in Table 2.

1.5 Terminology

Terminology used in this document are listed in Table 3.

Table 2: List of constants

Constant	Value	Description
α	5.0	Hero walk speed
β	3.0	Hero run speed factor: run speed = $\alpha \times \beta$
γ	100.0	Projectile speed
δ	10	Number of people in testing group
ϵ	99%	Coverage target
ζ	3.0	Enemy slow movement speed
η	10.0	Enemy medium movement speed
θ	30.0	Enemy fast movement speed

Table 3: List of terminology

Term	Definition
Floor	Horizontal obstacle that hero and enemies cannot pass through
Hero	User-controlled character
Platform	Horizontal obstacle that hero and enemies cannot pass through from above, but may pass through from below
Wall	Vertical obstacle that hero and enemies cannot pass through

2 Proof of Concept Testing

Before any serious development of the game begins, a proof of concept test will be carried out to show that the undertaking is feasible. The remainder of this section provides detail on what will be used as a proof of concept test.

2.1 Significant Risks

The successful completion of the project depends on overcoming the following significant risks:

1. In order to use the Chipmunk2D library it must first be successfully compiled. Since we intend for the game to be compatible with Windows 7, Mac OS X, and Ubuntu, there is a significant risk for the project to fail if compilation is not achieved on all three operating systems.

2. Chipmunk2D is a large library and its use is not straight forward. Successful implementation of the library features is crucial to the success of the project and the failure of this poses another significant risk.

2.2 Demonstration Plan

For a proof of concept test we will produce a working prototype that can be run on Windows 7, Mac OS X, and Ubuntu. The prototype will consist of a game demo that implements gravity and collision detection provided by the Chipmunk2D library. Rudimentary graphics will be used for the prototype since the scope is limited only to demonstrating that the identified risks can be overcome.

The prototype will consist of a small room in which a hero character and enemies exist. The room will be bounded by a floor below and walls on the left and right, all of which the hero and enemies cannot pass through. The room will contain platforms which the hero and enemies cannot pass through from above, but may pass through from below when jumping. A rough idea of the room is given in Figure 1.

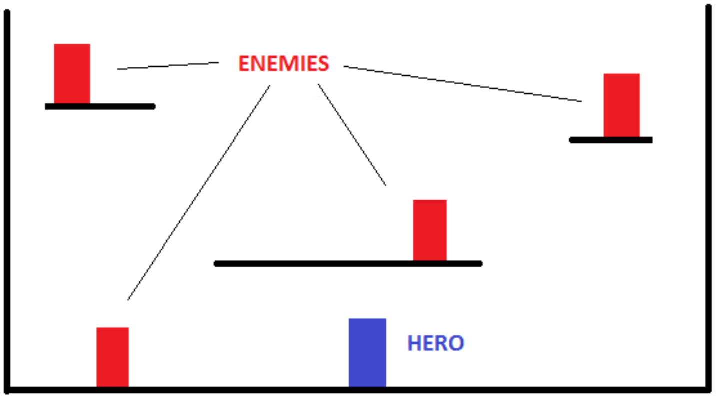

Figure 1: Proof of concept sketch

The hero character will be represented by a blue rectangle and will be controlled by the user in the following ways:

- The hero moves left and right using the ’a’ and ’d’ keys respectively
- The hero jumps by pressing the ’space’ key

- The hero shoots a projectile in the direction of the mouse cursor by left-clicking

Enemies will be represented by red rectangles and will not have any programmed AI (they will not move or attack). The hero will be able to attack enemies with a projectile, which will knock them back when they are hit. The hero and all enemies will be subject to gravity and will free-fall when there is no platform or boundary under them.

2.3 Proof of Concept Test

Test 2.3.1:	**Proof of Concept**
Description:	Tests whether significant risks to the completion of the project can be overcome
Type:	Proof of Concept (manual)
Tester(s):	Game developers
Pass:	Successful development of a small demonstration which makes use of the Chipmunk2D physics engine

3 System Testing

System testing is broken down into game mechanics testing and game design testing phases. The game mechanics testing phase take place as the game mechanics system is being developed. Once the game mechanics systems are in place, the .

3.1 Game Mechanics Testing

Automated unit testing will be used as the primary method for testing the game mechanics. The test cases that will be used are outlined in the remainder of this section.

3.1.1 Input Testing

The following tests will ensure that user inputs are handled properly.

Test 3.1.1.1: **Walk left, started from stationary**

Description: Tests if the hero walks left when the corresponding input is received when the hero is initially stationary

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity of zero

Input: Keyboard function called with simulated left key down stroke

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is $-\alpha$

Test 3.1.1.2: **Walk left, started from walking left**

Description: Tests if the hero walks left when the corresponding input is received when the hero is initially walking left

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity of $-\alpha$

Input: Keyboard function called with simulated left key down stroke

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is $-\alpha$

Test 3.1.1.3: **Walk left, started from running left**

Description: Tests if the hero walks left when the corresponding input is received when the hero is initially running left

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity of $-\beta \times \alpha$

Input: Keyboard function called with simulated left key down stroke

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is $-\alpha$

Test 3.1.1.4: **Run left, started from stationary**

Description: Tests if the hero runs left when the corresponding input is received when the hero is initially stationary

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity of zero

Input: Keyboard function called with simulated left key down stroke modified by the shift key

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is $-\beta \times \alpha$

Test 3.1.1.5:	**Run left, started from walking left**
Description:	Tests if the hero runs left when the corresponding input is received when the hero is initially walking left
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity of $-\alpha$
Input:	Keyboard function called with simulated left key down stroke modified by the shift key
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is $-\beta \times \alpha$

Test 3.1.1.6:	**Run left, started from running left**
Description:	Tests if the hero runs left when the corresponding input is received when the hero is initially running left
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity of $-\beta \times \alpha$
Input:	Keyboard function called with simulated left key down stroke modified by the shift key
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is $-\beta \times \alpha$

Test 3.1.1.7:	**Walk right, started from stationary**
Description:	Tests if the hero walks right when the corresponding input is received when the hero is initially stationary
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity of zero
Input:	Keyboard function called with simulated right key down stroke
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is α

Test 3.1.1.8:	**Walk right, started from walking right**
Description:	Tests if the hero walks right when the corresponding input is received when the hero is initially walking right
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity of α
Input:	Keyboard function called with simulated right key down stroke
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is α

Test 3.1.1.9: **Walk right, started from running right**

Description: Tests if the hero walks right when the corresponding input is received when the hero is initially running right

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity of $\beta \times \alpha$

Input: Keyboard function called with simulated right key down stroke

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is α

Test 3.1.1.10: **Run right, started from stationary**

Description: Tests if the hero runs right when the corresponding input is received when the hero is initially stationary

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity of zero

Input: Keyboard function called with simulated right key down stroke modified by the shift key

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is $\beta \times \alpha$

Test 3.1.1.11:	**Run right, started from walking right**
Description:	Tests if the hero runs right when the corresponding input is received when the hero is initially walking right
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity of α
Input:	Keyboard function called with simulated right key down stroke modified by the shift key
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is $\beta \times \alpha$

Test 3.1.1.12:	**Run right, started from running right**
Description:	Tests if the hero runs right when the corresponding input is received when the hero is initially running right
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity of $\beta \times \alpha$
Input:	Keyboard function called with simulated right key down stroke modified by the shift key
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is $\beta \times \alpha$

Test 3.1.1.13: **Stop walking left**

Description: Tests if hero stops walking left when corresponding input is stopped

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity of $-\alpha$

Input: Keyboard function called with simulated left key up stroke

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is zero

Test 3.1.1.14: **Stop running left**

Description: Tests if hero stops running left when corresponding input is stopped

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity of $-\beta \times \alpha$

Input: Keyboard function called with simulated left key up stroke

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is zero

Test 3.1.1.15:	**Stop walking right**
Description:	Tests if hero stops walking right when corresponding input is stopped
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity of α
Input:	Keyboard function called with simulated right key up stroke
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is zero

Test 3.1.1.16:	**Stop running right**
Description:	Tests if hero stops running right when corresponding input is stopped
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity of $\beta \times \alpha$
Input:	Keyboard function called with simulated right key up stroke
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is zero

Test 3.1.1.17: **Jump from static object**

Description: Tests if hero jumps off a static object when corresponding input is received

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having y-velocity of zero and a bottom edge in contact with a static object

Input: Keyboard function called with simulated jump key down stroke

Output: Hero object y-velocity (side-effect)

Pass: Hero object y-velocity is greater than zero

Test 3.1.1.18: **Jump from midair not allowed**

Description: Tests if hero is unable to jump while in midair when corresponding input is received

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having y-velocity of zero and a bottom edge not in contact with a static object

Input: Keyboard function called with simulated jump key down stroke

Output: Hero object y-velocity (side-effect)

Pass: Hero object y-velocity is zero (unchanged)

Test 3.1.1.19: **Activate pistol weapon**

Description: Tests if hero weapon is changes to pistol when corresponding input is received

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having y-velocity of zero and a bottom edge not in contact with a static object

Input: Keyboard function called with simulated jump key down stroke

Output: Hero object y-velocity (side-effect)

Pass: Hero object y-velocity is zero (unchanged)

3.1.2 Static Object Collision Testing

The following tests will ensure that the collision detection system is working as intended with respect to dynamic objects colliding with static objects.

Test 3.1.2.1: **Wall obstructs hero walking left**

Description: Tests whether the hero is stopped by a wall object while walking left

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity $-\alpha$ situated directly to the right of a wall

Input: The chipmunk cpSpaceStep function is called

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is 0

Test 3.1.2.2: **Wall obstructs hero running left**

Description: Tests whether the hero is stopped by a wall object while running left

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity $-\beta \times \alpha$ situated directly to the right of a wall

Input: The chipmunk cpSpaceStep function is called

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is 0

Test 3.1.2.3: **Wall obstructs hero walking right**

Description: Tests whether the hero is stopped by a wall object while walking right

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity α situated directly to the left of a wall

Input: The chipmunk cpSpaceStep function is called

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is 0

Test 3.1.2.4:	**Wall obstructs hero running right**
Description:	Tests whether the hero is stopped by a wall object while running right
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity $\beta \times \alpha$ situated directly to the left of a wall
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is 0

Test 3.1.2.5:	**Floor supports hero from below**
Description:	Tests whether the hero is supported by a floor object
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity $\beta \times \alpha$ situated directly to the left of a wall
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is 0

Test 3.1.2.6: **Floor supports hero from below**

Description: Tests whether the hero is supported by a floor object

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity $\beta \times \alpha$ situated directly to the left of a wall

Input: The chipmunk cpSpaceStep function is called

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is 0

Test 3.1.2.7: **Wall obstructs enemy moving left, low speed**

Description: Tests whether an enemy is stopped by a wall object while walking left

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity $-\alpha$ situated directly to the right of a wall

Input: The chipmunk cpSpaceStep function is called

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is 0

Test 3.1.2.8:	**Wall obstructs enemy moving left, medium speed**
Description:	Tests whether an enemy is stopped by a wall object while walking left
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity $-\alpha$ situated directly to the right of a wall
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is 0

Test 3.1.2.9:	**Wall obstructs enemy moving left, high speed**
Description:	Tests whether an enemy is stopped by a wall object while walking left
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity $-\alpha$ situated directly to the right of a wall
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is 0

Test 3.1.2.10: **Wall obstructs enemy moving right, low speed**

Description: Tests whether an enemy is stopped by a wall object while running left

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with enemy object having x-velocity ζ situated directly to the left of a wall object

Input: The chipmunk cpSpaceStep function is called

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is 0

Test 3.1.2.11: **Wall obstructs enemy moving right, medium speed**

Description: Tests whether an enemy is stopped by a wall object while walking right

Type: Unit Test (dynamic, automated)

Initial State: Custom in-game state with hero object having x-velocity η situated directly to the left of a wall object

Input: The chipmunk cpSpaceStep function is called

Output: Hero object x-velocity (side-effect)

Pass: Hero object x-velocity is 0

Test 3.1.2.12:	**Wall obstructs enemy moving right, high speed**
Description:	Tests whether an enemy is stopped by a wall object while moving right at high speed
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity θ situated directly to the left of a wall object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity (side-effect)
Pass:	Hero object x-velocity is 0

3.1.3 Dynamic Object Collision Testing

The following tests will ensure that the collisions between dynamic objects work as intended.

Test 3.1.3.1:	**Hero projectile collides with enemy**
Description:	Tests whether a projectile launched by the hero collides with enemies
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero projectile object with an x-velocity $-\gamma$ located directly to the right of an enemy object
Input:	The chipmunk cpSpaceStep function is called
Output:	State of the space
Pass:	Hero projectile object is removed from the space

Test 3.1.3.2:	**Enemy projectile collides with hero**
Description:	Tests whether a projectile launched by an enemy collides with the hero
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with an enemy projectile object with an x-velocity $-\gamma$ located directly to the right of the hero object
Input:	The chipmunk cpSpaceStep function is called
Output:	State of the space
Pass:	Enemy projectile object is removed from the space

3.1.4 Artificial Intelligence Testing

3.1.5 Save/Load Testing

The following tests will ensure that the game's saving and loading functions work properly.

Test 3.1.5.1:	**Load file from menu**
Description:	Tests whether a saved game can be successfully loaded from the main menu
Type:	Unit Test (dynamic, automated)
Initial State:	Main menu state
Input:	Load game function is called with file name
Output:	Game state
Pass:	Game state is equal to a predefined state that corresponds exactly to the file

Test 3.1.5.2: **Load file from menu, file does not exist**

Description: Tests that exception is thrown if a non-existent file is attempted to be loaded

Type: Unit Test (dynamic, automated)

Initial State: Main menu state

Input: Load game function is called with a non-existent file name

Output: File does not exist exception

Pass: File does not exist exception is thrown and handled

Test 3.1.5.3: **Load file from menu**

Description: Tests whether a saved game can be successfully loaded from the main menu

Type: Unit Test (dynamic, automated)

Initial State: Main menu state

Input: Load game function is called with file name

Output: Game state

Pass: Game state is equal to a predefined state that corresponds exactly to the file

Test 3.1.5.4:	**Load file from in-game**
Description:	Tests whether a saved game can be successfully loaded while in-game
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state different from that described by the saved game file
Input:	Load game function is called with file name
Output:	Game state
Pass:	Game state is equal to a predefined state that corresponds exactly to the file

3.1.6 Unit Test Coverage

Once the game mechanics and all of the corresponding unit tests have been implemented, . This will uncover portions of code that were not tested and allow for the design of additional

Test 3.1.6.1:	**Game mechanics coverage**
Description:	Tests that the game mechanics unit testing adequately covers the game mechanics code
Type:	Structural (dynamic, automated)
Initial State:	Custom in-game state different from that described by the saved game file
Input:	Load game function is called with file name
Output:	Game state
Pass:	Coverage of game mechanics code is greater than ϵ

3.2 Game Design Testing

Once the game mechanics systems have been implemented and shown to be working correctly through the testing described in §3.1, the game itself can be built on top. The design of the game can be broken down into game world design, story/objectives design, graphics, and sound. Automated testing for this phase would be time-consuming and difficult to implement. Therefore, all of the game design testing will consist of manual tests.

3.2.1 Game World Testing

Test 3.2.1.1:	**All areas reachable**
Description:	Tests that all areas of the game world that are intended to be reachable by the hero are in fact reachable by the hero
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	No areas are unreachable based on thorough . Alpha testers and beta testers will be asked to note down any

Test 3.2.1.2:	**No "points of no return"**
Description:	Tests that there are no areas of the game world that will cause the hero to become stuck (e.g. inescapable pits)
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	Hero does not become stuck on playthrough testing . Alpha testers and beta testers are asked to note down any

3.2.2 Story/Objectives Testing

3.2.3 Graphics Testing

3.2.4 Audio Testing

The following tests will be carried out to ensure that game audio is properly implemented.

Test 3.2.4.1:	**Hero movement sounds**
Description:	Tests if hero movement sounds are properly implemented
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	Appropriate sounds play when hero walks, runs, jumps, etc.

Test 3.2.4.2:	**Enemy movement sounds**
Description:	Tests if enemy movement sounds are properly implemented
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	Appropriate sounds play when enemies move

Test 3.2.4.3:	**Weapon fire sound**
Description:	Tests if hero weapon fire sound is properly implemented
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	Appropriate sounds play when hero fires weapon

Test 3.2.4.4:	**Enemy attack sounds**
Description:	Tests if enemy attack sounds are properly implemented
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	Appropriate sounds play when enemies launch attacks

Test 3.2.4.5:	**Collision sounds**
Description:	Tests if hero weapon sounds are properly implemented
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	Appropriate sounds play when hero fires weapon

3.3 General Testing

Throughout both of the system testing phases new code will be reviewed after each commit in an attempt to detect any visible errors.

Test 3.3.1:	**Code review**
Description:	Code is read through line by line while checking for errors
Type:	Structural (static, manual)
Tester(s):	Development team
Pass:	No errors found/all errors fixed

4 Requirements Testing

Testing will

4.1 Functional Requirements Testing

The functional requirements given in the software requirements specification document should all be implemented in the final version of the game. Since these requirements contribute to the , explicit testing of the requirements should be

Test 4.1.1:	**Functional requirements are met**
Description:	Game is compared with software requirements specification
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	All functional requirements are met and tested under system testing

4.2 Non-Functional Requirements Testing

The following tests will be carried out to ensure adherence to the non-functional requirements given in the software requirements specification.

Test 4.2.1:	**User experience, Phase I**
Description:	Game is compared with software requirements specification
Type:	Structural (static, manual)
Tester(s):	Testing group
Pass:	

Test 4.2.2:	**User experience, Phase II**
Description:	Game is compared with software requirements specification
Type:	Structural (static, manual)
Tester(s):	Testing group
Pass:	

Test 4.2.3:	**Spelling and grammar check**
Description:	The game uses proper English and is free of any spelling or grammatical errors
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	No spelling or grammatical errors are detected or all detected errors are corrected

5 Timeline

The testing

6 Appendix A: Testing Survey

The following survey will be filled out by members of the alpha and beta testing groups.

User Experience Survey

The following survey should be filled out after playing the game for at least 30 minutes.

Time played:

Please provide a ranking between 0 and 10 in each of the following categories. Please include notes on what you did and did not like, and what could be done to improve the game.

Entertainment: 0 1 2 3 4 5 6 7 8 9 10
[0 = most boring, 10 = most fun]

Difficulty: 0 1 2 3 4 5 6 7 8 9 10
[0 = easiest, 10 = most difficult]

Controls: 0 1 2 3 4 5 6 7 8 9 10
[0 = non-intuitive, 10 = intuitive]

Notes: