Physics-Based Chipmunk2D Game

Test Plan

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

April 26, 2016

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
November 1, 2015	1.2	Final version for rev 0

1 Overview

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

- A proof of concept test is described in §2.
- 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 and will utilize both unit testing and coverage analysis. Stubs and drivers will not be used due to the heavy use of side-effects in game code. To ensure that unit testing can be run while the game mechanics are in development the test cases have been grouped into sections that can be fully implemented once certain subsets of the game mechanics systems are completed.

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

Manual tests will be used for all game testing not classified as mechanics (the time that would be required to develop automated tests for these test cases would be prohibitive). The game development team will carry out these tests as the game is developed.

1.3.1 User Experience Testing

Manual testing will also be used to assess the user experience of the game. User experience testing will be completed by a testing group consisting of δ individuals who were not involved in the development of the game. The testing group will be given a copy of the game and asked to complete a survey (see Appendix A) after having played the game to provide feedback. The testing group will be asked to complete the survey twice: first in February (phase I) to provide initial feedback, and again in March (phase II) to provide feedback to assess improvements.

1.4 List of Constants

Constants used in this document are listed in Table 2. [Nice! —DS]

Table 2: List of constants

Constant	Value	Description
α	5.0	Hero walk speed
β	3.0	Hero run speed factor: run speed = $\alpha \times \beta$
λ	10.0	Hero jump speed
γ	200.0	Projectile speed
ζ	5.0	Enemy slow movement speed
η	20.0	Enemy medium movement speed
θ	50.0	Enemy fast movement speed
ϕ	10.0	Slow free fall speed
χ	50.0	Medium free fall speed
ψ	100.0	Fast free fall speed
ω	1.5	Knockback factor
ξ	5.0	Pistol damage
π	10.0	Shotgun damage
ρ	25.0	Rifle damage
Ξ	2.0	Enemy weak attack damage
Π	5.0	Enemy moderate attack damage
Σ	10.0	Enemy strong attack damage
ϵ	99%	Coverage target
σ	30	Frame rate target
δ	10	Number of people in testing group
Θ	6	Phase I testing entertainment target
Ψ	2	Phase I testing challenge range
Ω	8	Phase I testing controls target
Φ	10%	Phase II testing improvement target

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 describes the proof of concept test in detail.

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.

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

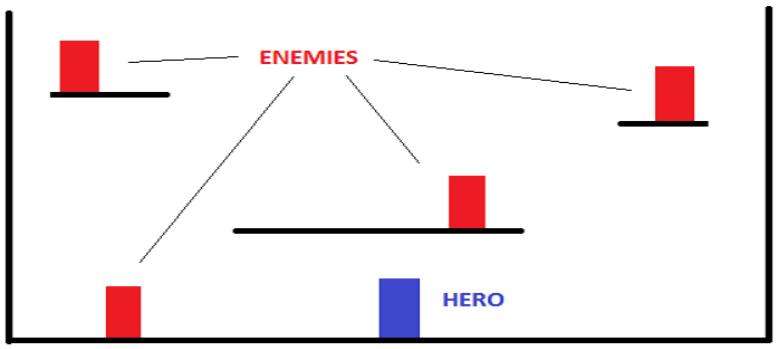

Figure 1: Proof of concept sketch

- 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

The proof of concept is given in test case format to adhere with the presentation of the other tests in this document.

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 and runs on Windows 7, Mac OS X, and Ubuntu

[Your demonstration plan and proof of concept test could be merged into one section as they are describing the same thing. The PoC Test subsection is fairly sparse on its own —DS]

3 System Testing

The testing of the system is broken down into game mechanics testing and game design testing phases. The game mechanics will be developed and tested first. Once the game mechanics systems are in place, the game design development and testing phase will begin.

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 a hero object having x-velocity of zero
Input:	Keyboard function called with simulated left key down stroke
Output:	Hero object x-velocity
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 a hero object having x-velocity of $-\alpha$
Input:	Keyboard function called with simulated left key down stroke
Output:	Hero object x-velocity
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 a hero object having x-velocity of $-\beta \times \alpha$
Input:	Keyboard function called with simulated left key down stroke
Output:	Hero object x-velocity
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 a 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
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
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
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
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
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
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
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
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
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
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
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
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
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 space bar key down stroke
Output:	Hero object y-velocity
Pass:	Hero object y-velocity is λ

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 space bar key down stroke
Output:	Hero object y-velocity
Pass:	Hero object y-velocity is zero (unchanged)

Test 3.1.1.19:	**Activate pistol weapon**
Description:	Tests if hero weapon is changed to pistol when corresponding input is received
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero object
Input:	Keyboard function called with simulated '1' key down stroke
Output:	Hero object (enum) weapon
Pass:	Hero object weapon is PISTOL

Test 3.1.1.20:	**Activate shotgun weapon**
Description:	Tests if hero weapon is changed to shotgun when corresponding input is received
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero object
Input:	Keyboard function called with simulated '2' key down stroke
Output:	Hero object (enum) weapon
Pass:	Hero object weapon is SHOTGUN

Test 3.1.1.21:	**Activate rifle weapon**
Description:	Tests if hero weapon is changed to rifle when corresponding input is received
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero object
Input:	Keyboard function called with simulated '3' key down stroke
Output:	Hero object (enum) weapon
Pass:	Hero object weapon is RIFLE

Test 3.1.1.22:	**Fire weapon left**
Description:	Tests if hero weapon is fired to the left when corresponding input is received
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero object
Input:	Mouse function called with simulated left button down click with cursor located directly to the left of the hero object center
Output:	State of the space
Pass:	Hero projectile object has been added to the space with velocity $(-\gamma, 0)$

Test 3.1.1.23:	**Fire weapon up-left**
Description:	Tests if hero weapon is fired to the upper left when corresponding input is received
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero object
Input:	Mouse function called with simulated left button down click with cursor located at a 45° angle (from the horizontal) to the upper left of the hero object center
Output:	State of the space
Pass:	Hero projectile object has been added to the space with velocity $(-\sqrt{\frac{\gamma}{2}}, \sqrt{\frac{\gamma}{2}})$

Test 3.1.1.24:	**Fire weapon up**
Description:	Tests if hero weapon is fired upwards when corresponding input is received
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero object
Input:	Mouse function called with simulated left button down click with cursor located directly above the hero object center
Output:	State of the space
Pass:	Hero projectile object has been added to the space to the left of the hero object with velocity $(0, \gamma)$

Test 3.1.1.25:	**Fire weapon up-right**
Description:	Tests if hero weapon is fired to the upper right when corresponding input is received
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero object
Input:	Mouse function called with simulated left button down click with cursor located at a 45° angle (from the horizontal) to the upper right of the hero object center
Output:	State of the space
Pass:	Hero projectile object has been added to the space with velocity $(\sqrt{\frac{\gamma}{2}}, \sqrt{\frac{\gamma}{2}})$

Test 3.1.1.26:	**Fire weapon right**
Description:	Tests if hero weapon is fired to the right when corresponding input is received
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero object
Input:	Mouse function called with simulated left button down click with cursor located directly to the right of the hero object center
Output:	State of the space
Pass:	Hero projectile object has been added to the space with velocity (γ, 0)

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 object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity
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 object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity
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 object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity
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 object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity
Pass:	Hero object x-velocity is 0

Test 3.1.2.5:	**Floor supports stationary hero**
Description:	Tests whether the hero is supported by a floor object
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with stationary hero object situated directly on top of a floor object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object y-velocity
Pass:	Hero object y-velocity is 0

Test 3.1.2.6:	**Floor stops hero in free fall, low speed**
Description:	Tests whether the hero in low speed free fall is stopped by a floor object
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object with y-velocity $-\phi$ situated directly on top of a floor object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object y-velocity
Pass:	Hero object y-velocity is 0

Test 3.1.2.7:	**Floor stops hero in free fall, medium speed**
Description:	Tests whether the hero in medium speed free fall is stopped by a floor object
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object with y-velocity $-\chi$ situated directly on top of a floor object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object y-velocity
Pass:	Hero object y-velocity is 0

Test 3.1.2.8:	**Floor stops hero in free fall, high speed**
Description:	Tests whether the hero in high speed free fall is stopped by a floor object
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object with y-velocity $-\psi$ situated directly on top of a floor object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object y-velocity
Pass:	Hero object y-velocity is 0

Test 3.1.2.9:	**Ceiling obstructs hero jump**
Description:	Tests whether the hero is obstructed by a ceiling in mid jump
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having y-velocity λ situated directly under a ceiling object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object y-velocity
Pass:	Hero object y-velocity is less than or equal to 0

Test 3.1.2.10:	**Wall obstructs enemy moving left, low speed**
Description:	Tests whether an enemy object is stopped by a wall object while moving left at low speed
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity $-\zeta$ situated directly to the right of a wall object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity
Pass:	Hero object x-velocity is 0

Test 3.1.2.11:	**Wall obstructs enemy moving left, medium speed**
Description:	Tests whether an enemy is stopped by a wall object while moving left at medium speed
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with hero object having x-velocity $-\eta$ situated directly to the right of a wall object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity
Pass:	Hero object x-velocity is 0

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

Test 3.1.2.13:	**Wall obstructs enemy moving right, low speed**
Description:	Tests whether an enemy is stopped by a wall object while moving right at low speed
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
Pass:	Hero object x-velocity is 0

Test 3.1.2.14:	**Wall obstructs enemy moving right, medium speed**
Description:	Tests whether an enemy is stopped by a wall object while moving right at medium speed
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
Pass:	Hero object x-velocity is 0

Test 3.1.2.15:	**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
Pass:	Hero object x-velocity is 0

Test 3.1.2.16:	**Floor supports stationary enemy**
Description:	Tests whether an enemy is supported by a floor object
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a stationary enemy object situated directly on top of a floor object
Input:	The chipmunk cpSpaceStep function is called
Output:	Enemy object y-velocity
Pass:	Enemy object y-velocity is 0

Test 3.1.2.17:	**Floor stops enemy in free fall, low speed**
Description:	Tests whether an enemy in low speed free fall is stopped by a floor object
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with an enemy object with y-velocity $-\phi$ situated directly on top of a floor object
Input:	The chipmunk cpSpaceStep function is called
Output:	Enemy object y-velocity
Pass:	Enemy object y-velocity is 0

Test 3.1.2.18:	**Floor stops enemy in free fall, medium speed**
Description:	Tests whether an enemy in medium speed free fall is stopped by a floor object
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with an enemy object with y-velocity $-\chi$ situated directly on top of a floor object
Input:	The chipmunk cpSpaceStep function is called
Output:	Enemy object y-velocity
Pass:	Enemy object y-velocity is 0

Test 3.1.2.19:	**Floor stops enemy in free fall, high speed**
Description:	Tests whether an enemy in high speed free fall is stopped by a floor object
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with an enemy object with y-velocity $-\psi$ situated directly on top of a floor object
Input:	The chipmunk cpSpaceStep function is called
Output:	Enemy object y-velocity
Pass:	Enemy object y-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 is knocked back when colliding with enemy from left**
Description:	Tests whether a the hero is knocked back to the left when an enemy is collided with from the left
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero object with an x-velocity α located directly to the left of a stationary enemy object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity
Pass:	Hero object x-velocity is $-\omega \times \alpha$

Test 3.1.3.2:	**Hero is knocked back when colliding with enemy from right**
Description:	Tests whether a the hero is knocked back to the right when an enemy is collided with from the right
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero object with an x-velocity $-\alpha$ located directly to the right of a stationary enemy object
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero object x-velocity
Pass:	Hero object x-velocity is $\omega \times \alpha$

Test 3.1.3.3:	**Hero projectile hits enemy**
Description:	Tests whether a projectile launched by the hero hits an enemy
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 a stationary enemy object
Input:	The chipmunk cpSpaceStep function is called
Output:	State of the space
Pass:	Hero projectile object is removed from the space

Test 3.1.3.4:	**Enemy projectile hits 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 a stationary 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 Hit Points Testing

The following tests will ensure the proper functioning of the hit point system.

Test 3.1.4.1:	**Enemy health reduced when hit by pistol**
Description:	Tests whether enemy health is properly reduced when hit by a pistol bullet
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero pistol projectile object with an x-velocity $-\gamma$ located directly to the right of an enemy object with 100 hit points
Input:	The chipmunk cpSpaceStep function is called
Output:	Enemy health
Pass:	Enemy health is equal to $100 - \xi$

Test 3.1.4.2:	**Enemy health reduced when hit by shotgun**
Description:	Tests whether enemy health is properly reduced when hit by a shotgun bullet
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero shotgun projectile object with an x-velocity $-\gamma$ located directly to the right of an enemy object with 100 hit points
Input:	The chipmunk cpSpaceStep function is called
Output:	Enemy health
Pass:	Enemy health is equal to $100 - \pi$

Test 3.1.4.3:	**Enemy health reduced when hit by rifle**
Description:	Tests whether enemy health is properly reduced when hit by a rifle bullet
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero rifle projectile object with an x-velocity $-\gamma$ located directly to the right of an enemy object with 100 hit points
Input:	The chipmunk cpSpaceStep function is called
Output:	Enemy health
Pass:	Enemy health is equal to $100 - \rho$

Test 3.1.4.4:	**Enemy is killed when hit points reach zero**
Description:	Tests whether an enemy is killed when hit points reach zero
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a hero pistol projectile object with an x-velocity $-\gamma$ located directly to the right of a an enemy object with 1 hit point
Input:	The chipmunk cpSpaceStep function is called
Output:	State of the space
Pass:	Enemy object is removed from the space

Test 3.1.4.5:	**Hero health reduced when hit by a weak enemy attack**
Description:	Tests whether hero health is properly reduced when hit by a weak enemy attack
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a weak enemy projectile object with an x-velocity $-\gamma$ located directly to the right of a hero object with 100 hit points
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero health
Pass:	Hero health is equal to $100 - \Xi$

Test 3.1.4.6:	**Hero health reduced when hit by a moderate enemy attack**
Description:	Tests whether hero health is properly reduced when hit by a moderate enemy attack
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a weak enemy projectile object with an x-velocity $-\gamma$ located directly to the right of a hero object with 100 hit points
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero health
Pass:	Hero health is equal to $100 - \Pi$

Test 3.1.4.7:	**Hero health reduced when hit by a strong enemy attack**
Description:	Tests whether hero health is properly reduced when hit by a strong enemy attack
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a weak enemy projectile object with an x-velocity $-\gamma$ located directly to the right of a hero object with 100 hit points
Input:	The chipmunk cpSpaceStep function is called
Output:	Hero health
Pass:	Hero health is equal to $100 - \Sigma$

Test 3.1.4.8:	**Hero is killed when hit points reach zero**
Description:	Tests whether hero is killed when hit points reach zero (game over)
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state with a weak enemy projectile object with an x-velocity $-\gamma$ located directly to the right of a hero object with 1 hit point
Input:	The chipmunk cpSpaceStep function is called
Output:	Game over flag
Pass:	Game over flag is set

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:	Exception
Pass:	File does not exist exception is thrown

Test 3.1.5.3:	**Load file from menu, file type invalid**
Description:	Tests that exception is thrown if an invalid file is attempted to be loaded
Type:	Unit Test (dynamic, automated)
Initial State:	Main menu state
Input:	Load game function is called with an invalid file
Output:	Exception
Pass:	File invalid exception is thrown

Test 3.1.5.4:	**Save game**
Description:	Tests whether the game state can be successfully saved
Type:	Unit Test (dynamic, automated)
Initial State:	Custom in-game state
Input:	Save game function is called with file name
Output:	Game save file
Pass:	Game save file is equal to a pre-existing file that corresponds exactly to the initial game state

3.1.6 Unit Test Coverage

Once the game mechanics and all of the corresponding unit tests have been implemented, a coverage test will be run in parallel with the unit testing. This will uncover portions of code that were not tested and allow for the design of additional test cases if necessary.

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)
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 the design of the game world, enemy artificial intelligence, 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

The following tests will be carried out to ensure that the game world is designed correctly.

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 a thorough playthrough testing of the game

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:	There are no inescapable areas detected on a thorough playthrough testing of the game

3.2.2 Artificial Intelligence Testing

The following tests will be carried out to ensure that the enemy artificial intelligence systems are properly implemented.

Test 3.2.2.1:	**Artificial intelligence, enemy movement general**
Description:	Tests that enemy movement routines work as intended when the hero is not in proximity
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	All enemy types move as they were designed when the hero is not near

Test 3.2.2.2:	**Artificial intelligence, enemy movement in proximity**
Description:	Tests that enemy movement routines work as intended when the hero is in proximity
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	All enemy types move as they were designed when the hero is in proximity

Test 3.2.2.3:	**Artificial intelligence, enemy attack**
Description:	Tests that enemy attack routines work as intended when the hero is in proximity
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	All enemy types attack in the way they were designed when the hero is in proximity

3.2.3 Graphics Testing

The following tests will be carried out to ensure that the game graphics and animations are properly implemented.

Test 3.2.3.1:	**Textures**
Description:	Tests if textures are properly implemented
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	In-game textures appear correct by inspection

Test 3.2.3.2:	**Hero animations**
Description:	Tests if hero movement/attack animations are properly implemented
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	Hero animations appear correct by inspection and synch with inputs

Test 3.2.3.3:	**Enemy animations**
Description:	Tests if enemy movement/attack animations are properly implemented
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	Enemy animations appear correct by inspection

3.2.4 Audio Testing

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

Test 3.2.4.1:	**Background music**
Description:	Tests if background music is properly implemented
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	Background music plays while in game

Test 3.2.4.2:	**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 3.2.4.3:	**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 3.2.4.4:	**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 3.2.4.5:	**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 3.2.4.6:	**Collision sounds**
Description:	Tests if hero weapon sounds are properly implemented
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	Appropriate sounds play when collisions take place

4 Requirements Testing

Once the system implementation is complete, testing will be performed to ensure that all requirements contained in the software requirements specification (SRS) document have been addressed.

4.1 Functional Requirements Testing

The functional requirements given in the SRS document should all be implemented in the final version of the game. Since all functional requirements are objective and should be a part of the overall system, this will be readily verifiable by using the SRS document as a checklist.

Test 4.1.1:	**Functional requirements are met**
Description:	Game is compared with SRS document to ensure functional requirements are met
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	All functional requirements are met

[You should specify which of your test cases cover which of your requirements and then check to ensure all of your requirements are being covered. If they aren't, then you should create more test cases. Currently this section isn't really saying anything new. —DS]

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:	**Operating system support**
Description:	The game runs on Windows 7, Mac OS X, and Ubuntu
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	Game can be compiled and system tests all pass on each platform

Test 4.2.2:	**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)

Test 4.2.3:	**Hardware requirements**
Description:	Tests for the minimum hardware requirements required to maintain an average frame rate of at least σ frames per second
Type:	Functional (dynamic, manual)
Tester(s):	Development team
Pass:	A pass criterion is still being devised as of revision 0 [Good. Don't forget to update it! —DS]

4.2.1 User Experience Testing

The following non-functional requirements will be tested via a user experience survey administered to a testing group (see §1.3.1).

Test 4.2.1.1:	**Entertainment**
Description:	Tests that the game is entertaining
Type:	Functional (dynamic, manual)
Tester(s):	Testing group
Pass:	Phase I average survey score of at least Θ; Phase II average survey score improves on Phase I score by Φ

Test 4.2.1.2:	**Challenge**
Description:	Tests that the game is adequately challenging (not too easy or too hard)
Type:	Functional (dynamic, manual)
Tester(s):	Testing group
Pass:	Phase I average survey score within Ψ of 5; Phase II average survey score improves on Phase I score by Φ

Test 4.2.1.3:	**Controls**
Description:	Tests that the game controls are intuitive
Type:	Functional (dynamic, manual)
Tester(s):	Testing group
Pass:	Phase I average survey score of at least Ω; Phase II average survey score improves on Phase I score by Φ

5 Timeline

This document is structured to roughly match the anticipated chronology of testing. A proposed testing timeline is given in Table 3.

Table 3: Testing timeline

Completion Date	Responsible Party	Task
11/15/2015	Development team	Completion of the proof of concept demo
11/30/2015	Chao Ye	Implementation of input unit test cases
12/10/2015	Emaad Fazal	Implementation of static collision unit test cases
12/10/2015	Steven Palmer	Implementation of dynamic collision unit test cases
12/20/2015	Chao Ye	Implementation of hit points unit test cases
12/20/2015	Steven Palmer	Implementation of save/load unit test cases
12/30/2015	Emaad Fazal	Implementation of coverage testing
01/20/2016	Development team	Completion of game design testing
01/20/2016	Development team	Completion of requirements testing (excluding user experience related tests)
02/20/2016	Testing group	Completion of user experience survey, phase I
03/20/2016	Testing group	Completion of user experience survey, phase II

[Do you have any tests for invalid inputs (other than saving/loading)? —DS] [As it reads now, it seems that the hero character will have all weapons available to them at all times. Is this intended? If not, you need more test cases. —DS]

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: