Physics-Based Chipmunk2D Game

Test Plan

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

October 31, 2015

Contents

1 Overview . . . **1**
- 1.1 Constants . . . 1
- 1.2 Terminology . . . 1

2 Proof of Concept Testing . . . **2**
- 2.1 Significant Risks . . . 2
- 2.2 Demonstration Plan . . . 2
- 2.3 Proof of Concept Test . . . 3

3 System Testing . . . **4**
- 3.1 Game Mechanics Testing . . . 4
  - 3.1.1 Input Testing . . . 4
  - 3.1.2 Static Object Collision Testing . . . 12
  - 3.1.3 Dynamic Object Collision Testing . . . 14
  - 3.1.4 Collision Callback Testing . . . 15
- 3.2 Game Design Testing . . . 15
  - 3.2.1 Game World Testing . . . 16
  - 3.2.2 Story/Objective Testing . . . 16
  - 3.2.3 Graphics Testing . . . 16
  - 3.2.4 Sound Testing . . . 16

4 Requirements Testing . . . **16**

5 Timeline . . . **16**

Revision History

Date	Version	Notes
October 25, 2015	1.0	Created document

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 Constants

Constants used in this document are listed in Table 1.

Table 1: List of constants

Constant	Value	Description
α	5.0	Walk speed
β	3.0	Run speed factor: run speed $= \alpha \times \beta$

1.2 Terminology

Terminology used in this document are listed in Table 2.

Table 2: List of terminology

Term	Definition

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.

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

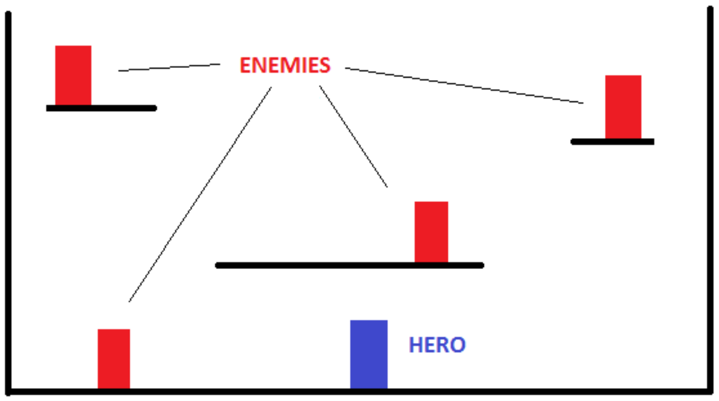

Figure 1: Proof of concept sketch

- 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

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

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 processed 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 game state with hero object having x-velocity of zero
Input:	Keyboard function called with simulated left key down stroke
Output:	Hero x-velocity (side-effect)
Pass:	Hero 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 game state with hero object having x-velocity of $-\alpha$
Input:	Keyboard function called with simulated left key down stroke
Output:	Hero x-velocity (side-effect)
Pass:	Hero 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 game state with hero object having x-velocity of $-\beta \times \alpha$
Input:	Keyboard function called with simulated left key down stroke
Output:	Hero x-velocity (side-effect)
Pass:	Hero 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 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 x-velocity (side-effect)
Pass:	Hero 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 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 x-velocity (side-effect)
Pass:	Hero 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 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 x-velocity (side-effect)
Pass:	Hero 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 game state with hero object having x-velocity of zero
Input:	Keyboard function called with simulated right key down stroke
Output:	Hero x-velocity (side-effect)
Pass:	Hero 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 game state with hero object having x-velocity of α
Input:	Keyboard function called with simulated right key down stroke
Output:	Hero x-velocity (side-effect)
Pass:	Hero 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 game state with hero object having x-velocity of $\beta \times \alpha$
Input:	Keyboard function called with simulated right key down stroke
Output:	Hero x-velocity (side-effect)
Pass:	Hero 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 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 x-velocity (side-effect)
Pass:	Hero 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 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 x-velocity (side-effect)
Pass:	Hero 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 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 x-velocity (side-effect)
Pass:	Hero 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 game state with hero object having x-velocity of $-\alpha$
Input:	Keyboard function called with simulated left key up stroke
Output:	Hero x-velocity (side-effect)
Pass:	Hero x-velocity is 0

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 game state with hero object having x-velocity of $-\beta \times \alpha$
Input:	Keyboard function called with simulated left key up stroke
Output:	Hero x-velocity (side-effect)
Pass:	Hero x-velocity is 0

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 game state with hero object having x-velocity of α
Input:	Keyboard function called with simulated right key up stroke
Output:	Hero x-velocity (side-effect)
Pass:	Hero x-velocity is 0

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 game state with hero object having x-velocity of $\beta \times \alpha$
Input:	Keyboard function called with simulated right key up stroke
Output:	Hero x-velocity (side-effect)
Pass:	Hero x-velocity is 0

Test 3.1.1.17:	**Jump from static object**
Description:	Tests if hero jumps when corresponding input is received
Type:	Unit Test (dynamic, automated)
Initial State:	Custom 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 y-velocity (side-effect)
Pass:	Hero y-velocity is greater than zero

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 while walking left
Type:	Unit Test (dynamic, automated)
Initial State:	Custom 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 x-velocity (side-effect)
Pass:	Hero x-velocity is 0

Test 3.1.2.2:	**Wall obstructs hero running left**
Description:	Tests whether the hero is stopped by a wall while running left
Type:	Unit Test (dynamic, automated)
Initial State:	Custom 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 x-velocity (side-effect)
Pass:	Hero x-velocity is 0

Test 3.1.2.3:	**Wall obstructs hero walking right**
Description:	Tests whether the hero is stopped by a wall while walking right
Type:	Unit Test (dynamic, automated)
Initial State:	Custom 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 x-velocity (side-effect)
Pass:	Hero x-velocity is 0

Test 3.1.2.4:	**Wall obstructs hero running right**
Description:	Tests whether the hero is stopped by a wall while running right
Type:	Unit Test (dynamic, automated)
Initial State:	Custom 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 x-velocity (side-effect)
Pass:	Hero x-velocity is 0

3.1.3 Dynamic Object Collision Testing

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

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 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 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 Collision Callback Testing

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/objective 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):	Game developers, alpha testers, beta testers
Pass:	No areas are unreachable based on thorough . Alpha testers and beta testers are asked to note down any

3.2.2 Story/Objective Testing

3.2.3 Graphics Testing

3.2.4 Sound Testing

4 Requirements Testing

5 Timeline