SE 3XA3: Test Plan

Team 10, MacLunky
Albert Zhou, zhouj103
Abeer Al-Yasiri, alyasira
Niyatha Rangarajan, rangaran

April 12, 2021

Contents

List of Tables

List of Figures

Table 1: Revision History

Date	**Developer(s)**	**Change**
February 27, 2021	Albert, Abeer, Niyatha	Version 0 made
March 30, 2021	Abeer	Test Schedule and unit testing
April 1, 2021	Niyatha	Spider testing
April 10, 2021	Albert	Additional test cases

1 General Information

1.1 Purpose

The purpose of the document is to describes the testing plan, procedures, and tools the development team will use to validate and verify the system functions and behaviour. All of the test cases describe in the document have been set before the final implementation is completed or applying testing in the development process. This document will be used in the next phases of the project to verify that the development team is on the right track and that the system meets the requirements.

1.2 Scope

The scope of the test plan is to guide the development team in the testing phase of the project by providing an outline of the required test cases and type of testing the development team needs to adapt to verify the system functionality.

1.3 Acronyms, Abbreviations, and Symbols

Table 2: **Table of Abbreviations**

Abbreviation	**Definition**
FPS	Frames per second
SRS	Software requirements specication
PoC	Proof of concept

Table 3: **Table of Definitions**

Term	**Definition**

1.4 Overview of Document

The test plan document will give a complete description of the project's testing techniques, tools, and schedule of execution. Also, the document will outline specific tests cases with description of how to execute the testing and goal of each. The test cases will be primarily used to provide verification of the system following the requirements.

2 Plan

2.1 Software Description

MacLunky is a re-implementation of the open source Pylunky game that is a Python version of the original Spelunky game. The objective of the game is to reach the end point of the game by advancing through the game map without losing all life points. The game will feature enemy characters and traps that will present as challenges throughout the game. However, the player will have access to weapons and tools that if used properly will provide a defense mechanism for the player and means to ensure the player survives the game course. The player will start with heartStartAmount life points and weapons that can be used at any point in the game. If the player wins the game it will end with inputting username that is saved with the game; else the game will shut down and close the window.

2.2 Test Team

The test team for this project will be the development team of MacLunky. Each team member will be responsible for writing and executing tests suites for the modules. The work breakdown of the testing suites will be assigned in the next project milestone.

2.3 Automated Testing Approach

The project will not be using automated testing approach in the testing phase of the project. The testing team decided to use manual testing approach. The reason behind this decision is because the project involves the redevelopment of game the main focus of testing will be to provide a fully functional and enjoyable gaming experience. The testing will showcase the importance of validating user interactions with the system and presenting an easy to use implementation of the original game that is more challenging. Another reason for not using automated testing is that the development team don't have access to the appropriate tools required to executes tests in this approach therefore using manual testing approach increase the testing team confidence in executing test correctly and efficiently.

2.4 Testing Tools

The testing plan will not specify any testing tools that will directly used for the system testing of the project since the team has adapted a manual testing approach. The only tools the tests will be using are command-line executions and executable windows to generate the game and run the tests on. However, ~~the project will use coverage metric tool called Coverage.py that will measure the code coverage of the program and provide a coverage report~~ the team decided not to dive into automated testing tools for coverage metrics due to the time constraints on the project.

2.5 Testing Schedule

See Gantt Chart under the Gitlab repository of the project under the Project Schedule folder.
The following table will outline the testing results of the FR testing and NFR testing done for the POC and Rev0 demonstrations.

Test Number	Test Result	Test Date
3.1.1.1	Pass	Feb 2021
3.1.1.2	Pass	Apr 2021
3.1.1.3	Pass	Feb 2021

3.1.2.1	Pass	Feb 2021
3.1.2.2	Pass	Feb 2021
3.1.2.3	Pass	Feb 2021
3.1.2.4	Pass	Feb 2021
3.1.2.5	Pass	Feb 2021
3.1.2.6	Pass	Feb 2021
3.1.2.7	Pass	Feb 2021
3.1.2.8	Pass	Apr 2021
3.1.2.9	Pass	Apr 2021
3.1.2.10	Pass	Apr 2021
3.1.2.11	Pass	Apr 2021
3.1.3.1	Pass	Feb 2021
3.1.3.2	Pass	Feb 2021
3.1.3.3	Pass	Feb 2021
3.1.3.4	Pass	Feb 2021
3.1.3.5	Pass	Feb 2021
3.1.3.6	Pass	Feb 2021
3.1.3.7	Pass	Feb 2021
3.1.3.8	Pass	Feb 2021
3.1.3.9	Pass	Feb 2021
3.1.3.10	Pass	Feb 2021
3.1.3.11	Pass	Feb 2021
3.1.3.12	Pass	Feb 2021
3.1.3.13	Pass	Feb 2021
3.1.3.14	Pass	Feb 2021
3.1.3.15	Pass	Feb 2021
3.1.3.16	Pass	Feb 2021
3.1.3.17	Pass	Feb 2021
3.1.3.18	Pass	Apr 2021
3.1.3.19	Pass	Feb 2021
3.1.3.20	Pass	Feb 2021
3.1.3.21	Pass	Feb 2021
3.1.3.22	Pass	Feb 2021
3.1.3.23	Pass	Feb 2021
3.1.3.24	Pass	Feb 2021
3.1.3.25	Pass	March 2021
3.1.3.26	Pass	Feb 2021
3.1.3.27	Pass	Apr 2021

3.1.3.28	Pass	Feb 2021
3.1.3.29	Pass	Feb 2021
3.1.3.30	Pass	Feb 2021
3.1.3.31	Pass	Feb 2021
3.1.3.32	Pass	Feb 2021
3.1.3.33	Pass	Feb 2021
3.1.4.1	Pass	Feb 2021
3.1.4.2	Pass	Feb 2021
3.1.4.3	Pass	Feb 2021
3.1.4.4	Pass	Feb 2021
3.1.4.5	Pass	Feb 2021
3.1.4.6	Pass	Feb 2021
3.1.4.7	Pass	Feb 2021
3.1.4.8	Pass	Feb 2021
3.1.5.1	Pass	Feb 2021
3.1.5.2	Pass	Feb 2021
3.1.5.3	Pass	Feb 2021
3.1.5.4	Pass	Feb 2021
3.1.5.5	Pass	March 2021
3.1.5.6	Pass	March 2021
3.1.6.1	Pass	Feb 2021
3.1.6.2	Pass	Feb 2021
3.1.6.3	Pass	Feb 2021
3.1.7.1	Pass	Feb 2021
3.1.7.2	Pass	March 2021
3.1.7.3	Pass	March 2021
3.1.7.4	Pass	March 2021
3.1.7.5	Pass	March 2021
3.1.7.6	Pass	March 2021
3.1.7.7	Pass	March 2021
3.1.8.1	Pass	Feb 2021
3.1.8.2	Pass	March 2021
3.1.8.3	Pass	March 2021
3.1.8.4	Pass	March 2021
3.1.9.1	Pass	Feb 2021
3.1.9.2	Pass	March 2021
3.1.9.3	Pass	March 2021
3.1.9.4	Pass	March 2021

3.1.10.1	Pass	Feb 2021
3.1.10.2	Pass	Feb 2021
3.1.10.3	Pass	Feb 2021
3.1.11.1	Pass	Feb 2021
3.1.11.2	Pass	Feb 2021
3.1.11.3	Pass	Feb 2021
3.1.11.4	Pass	Feb 2021
3.1.11.5	Pass	Apr 2021
3.1.12.1	Pass	Apr 2021
3.1.12.2	Pass	Apr 2021
3.1.12.3	Pass	Apr 2021
3.1.12.4	Pass	Apr 2021
3.1.12.5	Pass	Apr 2021
3.1.12.6	Pass	Apr 2021
3.1.13.1	Pass	Feb 2021
3.1.13.2	Pass	Feb 2021
3.1.13.3	Pass	Feb 2021
3.1.13.4	Pass	Apr 2021
3.1.13.5	Pass	Feb 2021
3.1.13.6	Pass	Feb 2021
3.1.13.7	Pass	Apr 2021
3.1.13.8	Pass	Feb 2021
3.1.13.9	Pass	Feb 2021
3.1.13.10	Pass	Feb 2021
3.1.13.11	Pass	Feb 2021
3.1.13.12	Pass	Apr 2021
3.1.13.13	Pass	Apr 2021
3.1.13.14	Pass	Apr 2021
3.1.14.1	Pass	Feb 2021
3.1.14.2	Pass	Feb 2021
3.1.14.3	Pass	Feb 2021
3.1.15.1	Pass	Apr 2021
3.1.15.2	Pass	Apr 2021
3.1.15.3	Pass	Apr 2021
3.1.16.1	Pass	Feb 2021
3.1.16.2	Pass	Feb 2021
3.1.16.3	Pass	Feb 2021
3.1.16.4	Pass	Feb 2021

3.1.17.1	Pass	Feb 2021
3.1.17.2	Pass	Feb 2021
3.1.17.3	Pass	Feb 2021
3.2.1.1	Pass	Apr 2021
3.2.1.2	Pass	Apr 2021
3.2.1.3	Pass	Apr 2021
3.2.1.4	Pass	Apr 2021
3.2.1.5	Pass	Apr 2021
3.2.1.6	Pass	Apr 2021
3.2.2.1	Pass	Apr 2021
3.2.2.2	Pass	Apr 2021
3.2.3.1	Pass	Apr 2021
3.2.3.2	Pass	Apr 2021
3.2.3.3	Pass	Apr 2021
3.2.3.4	Pass	Apr 2021

3 System Test Description

3.1 Tests for Functional Requirements

3.1.1 System

Test 3.1.1.1:	System launch from source files
Requirements:	FR1
Description:	Tests if the game launches correctly from the source files.
Type:	Unit test (dynamic, manual)
Initial State:	The source files has been downloaded and unzipped.
Input:	The game is executed with Python3.
Output:	None.
Pass:	The game opens in a new window with a level and the player.

Test 3.1.1.2:	System launch from executable
Requirements:	FR2
Description:	Tests if the game launches correctly from an executable.
Type:	Unit test (dynamic, manual)
Initial State:	The executable has been downloaded and unzipped.
Input:	The executable is run.
Output:	None.
Pass:	The game opens in a new window with a level and the player.

Test 3.1.1.3:	System termination
Requirements:	FR2
Description:	Tests if the game terminates correctly.
Type:	Unit test (dynamic, manual)
Initial State:	The game is running.
Input:	keyEsc is pressed.
Output:	None.
Pass:	The game terminates and the window closes.

3.1.2 Display

Test 3.1.2.1:	Camera position
Requirements:	FR3, 4
Description:	Tests if the camera is positioned correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has coordinates $x1$ and $y1$.
Input:	cam() is called.
Output:	Camera coordinates $x2$ and $y2$.
Pass:	$x2 = x1 -$ camWidth/2 and $0 \leq x2 \leq$ mapWidth and $y2 = y1 -$ camHeight/2 and $0 \leq y2 \leq$ mapHeight.

Test 3.1.2.2:	Camera move
Requirements:	FR3, 4
Description:	Tests if the camera is moving correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has coordinates $x1$ and $y1$.
Input:	move() is called.
Output:	Camera coordinates $x2$ and $y2$.
Pass:	$x2 = x1 -$ camWidth/2 and $0 \leq x2 \leq$ mapWidth and $y2 = y1 -$ camHeight/2 and $0 \leq y2 \leq$ mapHeight.

Test 3.1.2.3:	Health display
Requirements:	FR5
Description:	Tests if the health is displayed correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has n hearts.
Input:	ui() is called.
Output:	None.
Pass:	n hearts are displayed horizontally in the top left corner.

Test 3.1.2.4:	Health update
Requirements:	FR5
Description:	Tests if the health is updated correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has n hearts and takes m damage.
Input:	damage() is called.
Output:	None.
Pass:	$n - m$ hearts are displayed horizontally in the top left corner.

Test 3.1.2.5:	Gold display
Requirements:	FR5
Description:	Tests if the gold is displayed correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has n gold.
Input:	ui() is called.
Output:	None.
Pass:	n amount of gold is displayed in the top left corner under hearts.

Test 3.1.2.6:	Gold update
Requirements:	FR5
Description:	Tests if the gold is updated correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has n gold and is near a gold bar.
Input:	use() is called.
Output:	None.
Pass:	$n +$ valGoldBar amount of gold is displayed in the top left corner under hearts.

Test 3.1.2.7:	Game over
Requirements:	FR74
Description:	Tests if the game over screen is displayed correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has 0 hearts.
Input:	ui() is called.
Output:	None.
Pass:	The game over screen is displayeds.

Test 3.1.2.8:	Bombs display
Requirements:	FR5
Description:	Tests if the number of bombs is displayed correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has n bombs.
Input:	ui() is called.
Output:	None.
Pass:	n number of bombs is displayed in the top left corner next to hearts.

Test 3.1.2.9:	Bomb update
Requirements:	FR5
Description:	Tests if the bombs is updated correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has n bombs and is near a bombPile object.
Input:	use() is called.
Output:	None.
Pass:	n+valBombPile bombs is displayed in the top left corner next to hearts.

Test 3.1.2.10:	Ropes display
Requirements:	FR5
Description:	Tests if the number of ropes is displayed correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has n ropes.
Input:	ui() is called.
Output:	None.
Pass:	n number of ropes is displayed in the top left corner next to bombs.

Test 3.1.2.11:	Rope update
Requirements:	FR5
Description:	Tests if the ropes is updated correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has n ropes and is near a ropePile object.
Input:	use() is called.
Output:	None.
Pass:	n+valRopePile bombs is displayed in the top left corner next to bombs.

3.1.3 Player

Test 3.1.3.1:	Player Jumps
Requirements:	FR7, 15
Description:	Tests if the player is able to jump when the corresponding control input is sent by the user.
Type:	Unit Test (dynamic, manual)
Initial State:	Player in a stationary game state with y-direction speed of zero recorded.
Input:	Keyboard function called with pressing keyJump down stroke.
Output:	Player y-direction speed at maximum of jump.
Pass:	Player y-direction speed is playerJumpSpeed

Test 3.1.3.2:	Player Moves Left
Requirements:	FR7, 14
Description:	Tests if the player is able to move in the left direction when the corresponding control input is sent by the user.
Type:	Unit Test (dynamic, manual)
Initial State:	Player in a stationary game state with player's x-position recorded and x-speed is zero
Input:	Keyboard function called with pressing keyLeft down stroke
Output :	Player's x-position.
Pass :	Player's x-position decreased by playerSpeed

Test 3.1.3.3:	Player Moves Right
Requirements:	FR7, 14
Description:	Tests if the player is able to move in the left direction when the corresponding control input is sent by the user
Type:	Unit Test (dynamic, manual)
Initial State:	Player in a stationary game state with player's x-position recorded and x-speed is zero
Input:	Keyboard function called with pressing keyRight down stroke
Output	Player's x-position.
Pass :	Player's x-position increased by playerSpeed

Test 3.1.3.4:	Player Can't Jump at an Occurring Jump Action.
Requirements:	FR15
Description:	Tests if the player is able to jump while being in a jumping position when the corresponding control input is sent by the user.
Type:	Unit Test (dynamic, manual)
Initial State:	Player in a moving game state of a jump with y-direction speed of playerJumpSpeed.
Input:	Keyboard function called with pressing keyJump down stroke.
Output:	Player y-position at maximum of jump.
Pass:	Player y-position at maximum of jump has increased by playerJumpHeight.

Test 3.1.3.5:	Player Collect
Requirements:	FR8
Description:	Tests the player's interaction with the treasure box by being able to collect the gold placed on the ground by the game system.
Type:	Unit Test (dynamic, manual)
Initial State:	Player is at the same x-position as the gold and the player's gold score is zero.
Input:	Keyboard function called with pressing keyInt down stroke.
Output :	Player gold score.
Pass:	Player gold score is valGoldBar.

Test 3.1.3.6:	Player Opens Box
Requirements:	FR8
Description:	Tests the player's interaction with the treasure box by being able to open the box placed on the ground by the game system.
Type:	Unit Test (dynamic, manual)
Initial State:	Player is at the same x-position as the gold and the game system has one treasure box object in the system's list of objects.
Input:	Keyboard function called with pressing keyInt down stroke.
Output :	List of the system objects on display in the game.
Pass:	List of the system objects on display in the game has no treasure box entity

Test 3.1.3.7:	Player Stomps Enemy
Requirements:	FR37
Description:	Tests if the player is able to inflict damage upon an enemy character with the stomping player-enemy interaction.
Type:	Unit Test (dynamic, manual)
Initial State:	The player is falling on the enemy object with speed of fallSpeed and distance less than fallDmgDist.
Input:	Keyboard function called with move or jump key being pressed.
Output:	Player location (x, y) and list of the system objects displayed in the game.
Pass:	Player's position is equal to the enemies position at stomping and the list of system objects have no record of the enemy.

Test 3.1.3.8:	Player Releases Rope
Requirements:	FR24
Description:	Tests if the player is able to release the rope when the corresponding control input is sent by the user
Type:	Unit Test (dynamic, manual)
Initial State:	The player is interacting with rope at a position as the rope object.
Input:	Keyboard function called with pressing keyJump down stroke.
Output:	Player's (x, y) position
Pass:	Player is no longer interacting with the rope but the player's (x, y) position is different than the rope.

Test 3.1.3.9:	Player Grabs Rope.
Requirements:	FR22.
Description:	Tests if the player is able to get on the rope when the corresponding control input is sent by the user
Type:	Unit Test (dynamic, manual)
Initial State:	The player is within range to interact with the rope object.
Input:	Keyboard function called with pressing keyUp down stroke.
Output:	Player's (x, y) position
Pass:	Player is interacting with the rope and the same position as the rope.

Test 3.1.3.10:	Player Climbs Up
Requirements:	FR23.
Description:	Tests if the player is able to climb up when the corresponding control input is sent by the user
Type:	Unit Test (dynamic, manual)
Initial State:	The player is interacting with rope at a position as the rope object with enough rope distance in the positive y-direction.
Input:	Keyboard function called with pressing keyUp down stroke.
Output:	Player's (x, y) position
Pass:	Player is still interacting with the rope but the player's y-position increased by playerClimbSpeed

Test 3.1.3.11:	Player Climbs Down
Requirements:	FR23.
Description:	Tests if the player is able to climb down when the corresponding control input is sent by the user
Type:	Unit Test (dynamic, manual)
Initial State:	The player is interacting with rope at a position as the rope object with enough rope distance in the negative y-direction.
Input:	Keyboard function called with pressing keyDown down stroke.
Output:	Player's (x, y) position
Pass:	Player is still interacting with the rope but the player's y-position decreased by playerClimbSpeed.

Test 3.1.3.12:	Player Throws Object
Requirements:	FR10, 18
Description:	Tests if the player is able to throw object when the corresponding control input is sent by the user
Type:	Unit Testing (dynamic, manual)
Initial State:	Player is stationary with throwable object type in their hands and empty space in fornt of the player.
Input:	Keyboard function called with pressing keyThrow down stroke.
Output:	Throwable object speed and player's hands state.
Pass:	Throwable object speed is throwSpeed and the player's hands are empty

Test 3.1.3.13:	Player Picks up Object
Requirements:	FR17.
Description:	Tests if the player is able to pick up object on the ground when the corresponding control input is sent by the user
Type:	Unit Test (dynamic, manual)
Initial State:	The player is stationary and at the same (x, y) as the interactive object with the player's hand state is empty.
Input:	Keyboard function called with pressing keyPick down stroke.
Output:	Player's hand state.
Pass:	Player's hand state is occupied.

Test 3.1.3.14:	Player Use Bomb
Requirements:	FR9
Description:	Tests if the player is able to use and place bomb in the game when the corresponding control input is sent by the user
Type:	Unit Test (dynamic, manual)
Initial State:	The player is stationary and at an empty space position.
Input:	Keyboard function called with pressing keyBomb down stroke.
Output:	New bomb object created and its position.
Pass:	New bomb position is the player's position when the user pressed the key.

Test 3.1.3.15:	Player Use Weapon
Requirements:	FR12
Description:	Tests if the player is able to use weapon in the game when the corresponding control input is sent by the user
Type:	Unit Test (dynamic, manual)
Initial State:	The player is stationary and at an empty space position.
Input:	Keyboard function called with pressing keyweap down stroke.
Output:	New weapon object created and its position.
Pass:	New bomb position is the player's position when the user pressed the key.

Test 3.1.3.16:	Player Use Weapon For Period of Time and Cant use anything else
Requirements:	FR27
Description:	Tests if the player is able to use weapon once every weaponDelay and can't use anything else at the same time in the game when the corresponding control input is sent by the user
Type:	Unit Test (dynamic, manual)
Initial State:	The player is stationary and at an empty space position.
Input:	Keyboard function called with pressing keyweap down stroke and pressing keyPick.
Output:	New weapon object created and its position and the player hands state.
Pass:	New bomb position is the player's position when the user pressed the key and the returns can't pick up message with player's hand used.

Test 3.1.3.17:	Player Hits and Damages Snake Enemy.
Requirements:	FR26, 25
Description:	Tests if the player is able to damage snake character upon hitting it with weapon.
Type:	Unit Test (dynamic, manual)
Initial State:	The player is stationary and at position where the snake is with weaponAttack.
Input:	Keyboard function called with pressing keyweap down stroke.
Output:	Snake health.
Pass:	Snake health is reduced by weaponDmg.

Test 3.1.3.18:	Player Hits and Damages Spider Enemy.
Requirements:	FR26, 25
Description:	Tests if the player is able to damage spider character upon hitting it with weapon.
Type:	Unit Test (dynamic, manual)
Initial State:	The player is stationary and at position where the spider is with weaponAttack.
Input:	Keyboard function called with pressing keyweap down stroke.
Output:	Spider health.
Pass:	Spider health is reduced by weaponDmg.

Test 3.1.3.19:	Player Use Rope
Requirements:	FR10
Description:	Tests if the player is able to use and place rope in the game when the corresponding control input is sent by the user
Type:	Unit Test (dynamic, manual)
Initial State:	The player is stationary and at an empty space position.
Input:	Keyboard function called with pressing keyRope down stroke.
Output:	New rope object created and its position.
Pass:	New rope position is the player's position when the user pressed the key.

Test 3.1.3.20:	Player Falls
Requirements:	FR28.
Description:	Tests if the player's response of no longer being on solid ground is falling.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player is stepping onto empty space with no block underneath.
Input:	Keyboard function that moves the player.
Output:	Player speed.
Pass:	Player speed is equal to fallSpeed.

Test 3.1.3.21:	Player Lands Back on the Ground.
Requirements:	FR28.
Description:	Tests if the player's response after being in the air is landing on the closes ground block underneath.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player is in the air with a ground block underneath that is of a distance less than fallDmgDist.
Input:	Keyboard function that moves the player.
Output:	Player position and speed.
Pass:	Player speed is zero and position is at the closet block underneath.

Test 3.1.3.22:	Player Can't move past blocks.
Requirements:	FR28.
Description:	Tests if the player is able to move on a block when the corresponding control input is sent by the user.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player is in one position away from a block to the left.
Input:	Keyboard function called with pressing keyLeft down stroke.
Output:	Player position and speed.
Pass:	Player speed is PlayerSpeed and position remains the same.

Test 3.1.3.23:	Player Can't move outside the map.
Requirements:	FR28.
Description:	Tests if the player is able to move on a block when the corresponding control input is sent by the user.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player is in one position away from the map boundary to the left.
Input:	Keyboard function called with pressing keyLeft down stroke.
Output:	Player position and speed.
Pass:	Player speed is PlayerSpeed and position remains the same.

Test 3.1.3.24:	Player Crouches.
Requirements:	FR11.
Description:	Tests if the player is able to be in a crouch position when the corresponding control input is sent by the user
Type:	Unit Testing (dynamic, manual)
Initial State:	The player is stationary.
Input:	Keyboard function called with pressing keyCrouch down stroke.
Output	Player state.
Pass:	Player state is crouching.

Test 3.1.3.25:	Player Health Reduced When Falling From a Peak
Requirements:	FR29, 30.
Description:	Tests if the player health system is reduced correctly upon falling from high position in the game.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player with a health of heartStartAmount is falling at a speed of fallSpeed and the fall distance is fallDmgDist.
Input:	Keyboard function that moves the player.
Output:	Player health.
Pass:	Player health reduced by fallDmg.

Test 3.1.3.26:	Player Health Reduced When Hit by Snake Enemy.
Requirements:	FR36.
Description:	Tests if the player health system is reduced correctly upon being hit by a snake character in the game.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player with a health of heartStartAmount is stationary at a position within attack area of the snake character that is moving at a speed of snakeSpeed.
Input:	None.
Output:	Player health.
Pass:	Player health reduced by snakeDmg.

Test 3.1.3.27:	Player Health Reduced When Hit by Spider Enemy.
bfRequirements:	FR36.
Description:	Tests if the player health system is reduced correctly upon being hit by a spider character in the game.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player with a health of heartStartAmount is stationary at a position within spiderSense attack area of the spider character that is stationary.
Input:	None.
Output:	Player health.
Pass:	Player health is reduced by spiderDmg.

Test 3.1.3.28:	Player Health Reduced When Hit by Arrow.
bfRequirements:	FR71.
Description:	Tests if the player health system is reduced correctly upon being hit by an arrow in the game.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player with a health of heartStartAmount is stationary at a position within arrowSense attack range of the arrow trap box and the arrow is moving arrowSpeed towards the player.
Input:	None.
Output:	Player health.
Pass:	Player health is reduced by arrowDmg.

Test 3.1.3.29:	Player Health Reduced When Overlap with Spike.
bfRequirements:	FR73.
Description:	Tests if the player health system is reduced correctly upon overlapping with a spike trap.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player with a health of heartStartAmount is stationary at the same position as the spike trap.
Input:	None.
Output:	Player health.
Pass:	Player health reduced by spikeDmg.

Test 3.1.3.30:	Player Health Reduced When hit by bomb explosion.
bfRequirements:	FR16.
Description:	Tests if the player health system is reduced correctly upon overlapping with a bomb explosion.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player with a health of heartStartAmount is stationary at the same position as the stationary bomb.
Input:	None.
Output:	Player health.
Pass:	Player health reduced by bombDmg.

Test 3.1.3.31:	Player Killed When Health Reaches Zero.
bfRequirements:	FR74.
Description:	Tests if the player is removed from the game upon having zero health.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player with a health of heartStartAmount is stationary at a position within an attack that will cause a damage of heartStartAmount.
Input:	None
Output:	List of player object.
Pass:	Empty list.

Test 3.1.3.32:	Player Dodges a Moving Arrow Attack.
bfRequirements:	FR7.
Description:	Tests if the player is able to dodge an arrow attack by jumping.
Type:	Unit Testing (dynamic, manual)
Initial State:	The player with a health of heartStartAmount is moving at playerSpeed with an arrow moving toward the player at arrowSpeed.
Input:	Keyboard function called with pressing keyJump down stroke.
Output:	Player health.
Pass:	Player health remains the same after passing the arrow attack.

Test 3.1.3.33:	Player Starts the game with heartStartAmount hearts, bombStartAmount bombs, and ropeStartAmount ropes..
bfRequirements:	FR13.
Description:	Tests if the player is player starts the game with the right amount of resources.
Type:	Unit Testing (dynamic, manual)
Initial State:	The game is being generated.
Input:	Game file initiated.
Output:	Player health, bombs, and ropes.
Pass:	Player has heartStartAmount hearts, bombStartAmount bombs, and ropeStartAmount ropes.

3.1.4 Bomb

Test 3.1.4.1:	Bomb placed
Requirements:	FR9
Description:	Tests if a bomb placed by the player has the correct coordinates.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has coordinates $x1$ and $y1$.
Input:	makeEnt() is called.
Output:	Bomb is placed with coordinates $x2$ and $y2$.
Pass:	$x2 = x1$ and $y2 = y1 + \text{playerHeight} - \text{bombHeight}$

Test 3.1.4.2:	Bomb time set
Requirements:	FR16
Description:	Tests if a bomb placed with the correct time.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a bomb has been placed.
Input:	makeEnt() is called.
Output:	The time remaining on the bomb
Pass:	The time remaining is bombTime seconds.

Test 3.1.4.3:	Bomb explosion time
Requirements:	FR16
Description:	Tests if a bomb explodes after the correct amount of time.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a bomb is placed.
Input:	tick() is called
Output:	The time remaining on the bomb
Pass:	The bomb explodes if it has 0 remaining seconds.

Test 3.1.4.4:	Bomb explosion size
Requirements:	FR16
Description:	Tests if the explosion of a bomb is the correct size.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a bomb has exploded.
Input:	explode is called().
Output:	None.
Pass:	The explosion has a diameter of bombSize blocks.

Test 3.1.4.5:	Bomb explosion destruction
Requirements:	FR16
Description:	Tests if a bomb's explosion deals the correct amount of damage to all entities and destroys all blocks in range
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a bomb placed on a block has exploded with a nearby player, snake, chest, and arrow trap.
Input:	None.
Output:	None.
Pass:	The block is destroyed and the player, snake, chest, and arrow trap take bombDmg damage.

Test 3.1.4.6:	Bomb pickup
Requirements:	FR17
Description:	Tests if a picked up bomb has the correct coordinates
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player with coordinates $x1$ and $y1$ on a block places a bomb.
Input:	pickup() is called
Output:	Bomb coordinates $x2$ and $y2$.
Pass:	$x2 = x1 + (\text{playerWidth} - \text{bombWidth})/2$ and $y2 = y1 + (\text{playerHeight} - \text{bombHeight})/2$

Test 3.1.4.7:	Bomb throw
Requirements:	FR18
Description:	Tests if a thrown bomb is moving at the correct speed.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a bomb has been thrown.
Input:	throw() is called.
Output:	Bomb speed.
Pass:	Bomb speed is throwSpeed horizontally.

Test 3.1.4.8:	Bomb throw explosion
Requirements:	FR19
Description:	Tests if a thrown bomb explodes upon contact with a block.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a bomb has been thrown towards a block.
Input:	move() is called.
Output:	None.
Pass:	Bomb explodes upon contact with the block.

3.1.5 Rope

Test 3.1.5.1:	Rope throw
Requirements:	FR10
Description:	Tests if a rope placed by the player has the correct coordinates.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has coordinates $x1$ and $y1$.
Input:	makeEnt() is called.
Output:	Rope is placed coordinates $x2$ and $y2$.
Pass:	$x2 = x1$ and $y2 = y1$

Test 3.1.5.2:	Rope throw speed
Requirements:	FR10
Description:	Tests if a thrown rope is moving at the correct speed.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a rope has been thrown.
Input:	throw() is called.
Output:	Rope speed.
Pass:	Rope speed is $-$throwSpeed vertically.

Test 3.1.5.3:	Rope max height
Requirements:	FR20
Description:	Tests if a thrown rope stops after reaching the maximum height at the correct coordinates.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a rope is moving with coordinates $x1$ and $y1$.
Input:	move() is called.
Output:	Rope coordinates $x2$ and $y2$.
Pass:	$x2 = x1/\text{blockSize} * \text{blockSize} + \text{blockSize}/2$ and $y2 = y1/\text{blockSize} * \text{blockSize} + \text{ropeLength} * \text{blockSize}$

Test 3.1.5.4:	Rope attachment in block
Requirements:	FR20
Description:	Tests if a thrown rope stops in a block at the correct coordinates.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a rope is moving with coordinates $x1$ and $y1$.
Input:	move() is called.
Output:	Rope coordinates $x2$ and $y2$.
Pass:	$x2 = x1/\text{blockSize} * \text{blockSize} + \text{blockSize}/2$ and $y2 = y1/\text{blockSize} * \text{blockSize}$

Test 3.1.5.5:	Rope extension
Requirements:	FR21
Description:	Tests if an attached rope extends down correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where an attached rope has coordinates $x1$ and $y1$.
Input:	makeRope() is called.
Output:	The Rope end with coordinates $x2$ and $y2$.
Pass:	$x2 = x1$ and $y2 = y1 + n *$ blockSize where $n =$ The number of empty blocks below the rope $\leq$ ropeLength

Test 3.1.5.6:	Rope crouch throw
Requirements:	FR11
Description:	Tests if a rope placed by the player while crouched has the correct coordinates.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has coordinates $x1$ and $y1$.
Input:	makeEnt() is called.
Output:	Rope is placed coordinates $x2$ and $y2$.
Pass:	$x2 = x1 +$ blockSize and $y2 = y1$

3.1.6 Weapon

Test 3.1.6.1:	Weapon swing
Requirements:	FR25
Description:	Tests if the weapon appears at the correct coordinates.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where player has coordinates $x1$ and $y1$.
Input:	Keyboard function called with pressing keyWeap down stroke.
Output:	Weapon is placed coordinates $x2$ and $y2$.
Pass:	$x2 = x1 +$ playerWidth and $y2 = y1 +$ playerHeight$/2$

Test 3.1.6.1:	Weapon damage
Requirements:	FR26
Description:	Tests if the weapon damages enemies correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state with a player and a nearby snake.
Input:	Keyboard function called with pressing keyWeap down stroke.
Output:	None.
Pass:	Snake loses weaponDmg health

Test 3.1.6.1:	Weapon delay
Requirements:	FR27
Description:	Tests the delay of the weapon.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state with a player.
Input:	Keyboard function called with pressing keyWeap down stroke.
Output:	None.
Pass:	The weapon can be used only once every weaponDelay seconds.

3.1.7 Gold

Test 3.1.7.1:	Gold bar pickup
Requirements:	FR32, 48
Description:	Tests if picking up a gold bar adds the correct amount to total gold.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player has 0 gold and is near a gold bar.
Input:	use() is called.
Output:	None.
Pass:	Player's gold increases by valGoldBar.

Test 3.1.7.2:	Ruby pickup
Requirements:	FR32, 45
Description:	Tests if picking up a ruby adds the correct amount to total gold.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player has 0 gold and is near a ruby.
Input:	use() is called.
Output:	None.
Pass:	Player's gold increases by valRuby.

Test 3.1.7.3:	Emerald pickup
Requirements:	FR32, 47
Description:	Tests if picking up an emerald adds the correct amount to total gold.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player has 0 gold and is near an emerald.
Input:	use() is called.
Output:	None.
Pass:	Player's gold increases by valEmerald.

Test 3.1.7.4:	Sapphire pickup
Requirements:	FR32, 46
Description:	Tests if picking up a sapphire adds the correct amount to total gold.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player has 0 gold and is near a sapphire.
Input:	use() is called.
Output:	None.
Pass:	Player's gold increases by valSapphire.

Test 3.1.7.5:	Diamond pickup
Requirements:	FR32, 44
Description:	Tests if picking up a diamond adds the correct amount to total gold.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player has 0 gold and is near a diamond.
Input:	use() is called.
Output:	None.
Pass:	Player's gold increases by valDiamond.

Test 3.1.7.6:	Bomb pile pickup
Requirements:	FR32, 88
Description:	Tests if picking up a bomb pile adds the correct number to bombs.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player has 0 bombs and is near a bomb pile.
Input:	use() is called.
Output:	None.
Pass:	Player's bombs increases by valBombPile.

Test 3.1.7.7:	Rope pile pickup
Requirements:	FR32, 89
Description:	Tests if picking up a rope pile adds the correct number to ropes.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player has 0 ropes and is near a rope pile.
Input:	use() is called.
Output:	None.
Pass:	Player's ropes increases by valRopePile.

3.1.8 Sprite

Test 3.1.8.1:	Idle sprite
Requirements:	FR34
Description:	Tests if the correct player sprite is used while idle.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player is idle.
Input:	changeState() is called.
Output:	None.
Pass:	The player has the idle sprite.

Test 3.1.8.2:	Falling sprite
Requirements:	FR34
Description:	Tests if the correct player sprite is used while in the air.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player is in the air.
Input:	changeState() is called.
Output:	None.
Pass:	The player has the falling sprite.

Test 3.1.8.3:	Climbing sprite
Requirements:	FR34
Description:	Tests if the correct player sprite is used while climbing.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player is climbing.
Input:	changeState() is called.
Output:	None.
Pass:	The player has the climbing sprite.

Test 3.1.8.4:	Crouching sprite
Requirements:	FR34
Description:	Tests if the correct player sprite is used while crouching.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player is crouching.
Input:	changeState() is called.
Output:	None.
Pass:	The player has the crouching sprite.

3.1.9 Chest

Test 3.1.9.1:	Chest placement
Requirements:	FR39
Description:	Tests if a chest has the correct coordinates based on the map.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a chest has coordinates $x1$ and $y1$.
Input:	makeEnt() is called.
Output:	None.
Pass:	$x1$ and $y1$ are correct based on the map.

Test 3.1.9.2:	Chest open
Requirements:	FR40, 41, 42, 43
Description:	Tests if a chest is opened correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a chest has coordinates $x1$ and $y1$.
Input:	use() is called.
Output:	Gold bar or ruby with coordinate $x2$ and $y2$.
Pass:	$x2 = x1$ and $y2 = y1$ and the chest is removed from the game.

Test 3.1.9.3:	Chest pickup
Requirements:	FR91
Description:	Tests if a picked up chest has the correct coordinates
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player with coordinates $x1$ and $y1$ on a block and is near a chest.
Input:	pickup() is called
Output:	Chest coordinates $x2$ and $y2$.
Pass:	The chest is in the player's hand.

Test 3.1.9.4:	Chest throw
Requirements:	FR91
Description:	Tests if a thrown chest is moving at the correct speed.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a chest has been thrown.
Input:	throw() is called.
Output:	Chest speed.
Pass:	Chest speed is throwSpeed horizontally.

3.1.10 Enemy map placement

Test 3.1.10.1:	Enemy entities like spider and snake are placed on the map by reading the coordinates from the map file.
Requirements:	FR79
Description:	If the enemy is created inside a solid block it has no functionality in the game
Type:	Unit test (dynamic, manual)
Initial State:	Map coordinates for enemies are selected so it coincides with inside a block. Player's health is H. The player approaches the block containing the enemy.
Input:	ReadMap and Mapcell generate the game map in the main function
Output:	The enemy entities are motionless and do not attack or reduce health of the player
Pass:	The enemy entities do not move whether or not the player is in range by a block and the player's health remains as H.

Test 3.1.10.2:	Player-enemy Defense - Whip
Requirements:	FR12
Description:	The enemy is damaged when a whip is latched onto the enemy
Type:	Unit test (dynamic, manual)
Initial State:	The player object approaches the enemy object such that the player is able to overlap the whip object onto the enemy
Input:	Keyboard function called with pressing lshift key.
Output:	The enemy loses health by weaponDmg from initial health H
Pass:	The enemy damaged health is the original health(H)-weaponDmg

Test 3.1.10.3:	Player-enemy Defense - Stomp
Requirements:	FR37
Description:	The enemy is killed when a player jumps onto a enemy entity
Type:	Dynamic and Unit testing
Initial State:	The player object approaches the enemy object such that the player jump onto the enemy
Input:	ent.damage() is called to reduce the enemy's health when the objects overlap
Output:	The enemy loses health by weaponDmg from initial health H
Pass:	The enemy damaged health is the original health(H)-stompDmg

3.1.11 Snake

Test 3.1.11.1:	Snake creation
Requirements:	FR50, FR52
Description:	The snake entities are placed all around the map at the start of the game
Type:	Unit test (dynamic, manual)
Initial State:	The player starts the game
Input:	Map object is created using the class MapObj and ReadMap.
Output:	The snake is seen moving in a straight line turning at solid blocks at different points of the map.
Pass:	The coordinates of the snake in the map file match the initial starting positions of the snakes that appear on the game map

Test 3.1.11.2:	Snake moves towards the right and alters direction upon a solid block (moves left).
Requirements:	FR51
Description:	Tests whether the snake switches to the opposite direction upon coming in contact with a solid block
Type:	Unit test (dynamic, manual)
Initial State:	Snake entities created by an in-game map file placing them at specific map locations.
Input:	No input, snake entity and solid blocks are created is generated by the game
Output:	snake changes initial right direction towards the left
Pass:	Snake oscillates right to left and left to right upon contact of a solid block with snakeSpeed.

Test 3.1.11.3:	Snake attacks a player on the way of its straightline path of movement.
Requirements:	FR54
Description:	Tests whether the player's health is reduced by snakeDmg when the snake entity overlaps with the player.
Type:	Unit test (dynamic, manual)
Initial State:	Snake entities created by an in-game map file placing them at specific map locations. Player has health of **H**
Input:	No input, snake entity is created is generated by the game
Output:	Player health reduces by snakeDmg
Pass:	Player health is H-snakeDmg

Test 3.1.11.4:	Snake attacks the player killing the player
Requirements:	FR54
Description:	Tests whether the player dies if the health of the player is less than or equal to the snakeDmg
Type:	Unit test (dynamic, manual)
Initial State:	Snake entities created by an in-game map file placing them at specific map locations. Player has health of **H**
Input:	No input, snake entity is created is generated by the game
Output:	Player health reduces to 0 resulting in game over
Pass:	Player health is 0

Test 3.1.11.5:	Player dodges snake by jumping over it
Requirements:	FR15
Description:	Tests whether the player is able to jump over the snake
Type:	Unit test (dynamic, manual)
Initial State:	Snake entities created by an in-game map file placing them at specific map locations. Player has health of **H**
Input:	The upward arrow keyboard input is pressed
Output:	The player jumps over the snake without reducing their own health
Pass:	Player health is H and the snake still exists and is in motion on the map.

3.1.12 Spider

Test 3.1.12.1:	Spider creation
Requirements:	FR55
Description:	The spider entities are placed all around the map at the start of the game
Type:	Unit test (dynamic, manual)
Initial State:	The player starts the game
Input:	Map object is created using the class MapObj and ReadMap.
Output:	The spider is seen dormant on certain map locations waiting for the player to come in range of 4 blocks from it.
Pass:	The coordinates of the spider in the map file match the initial starting positions of the spider that appear on the game map

Test 3.1.12.2:	Spider jumps on the player if the player is within a range.
Requirements:	FR56
Description:	Tests whether the spider and player's relative position fit the attack range of the spider which a block size.
Type:	Unit test (dynamic, manual)
Initial State:	Spider entities created by an in-game map file placing them at specific map locations. Player has health of **H**. Spider and player's x position are less than 4 * 32 units that is a block width.
Input:	No input, spider entity is created is generated by the game
Output:	Spider attacks the player and then cool offs and continues to attack the player if the player is still in range.
Pass:	The spider takes the same x and y coordinates to attack the player before it cools off for a specified cool off time

Test 3.1.12.3:	Spider attacks the player by jumping on them reducing their health
Requirements:	FR62
Description:	Test whether the player's health reduces for the time the player stays within the attack range of the spider.
Type:	Unit test (dynamic, manual)
Initial State:	Spider entities created by an in-game map file placing them at specific map locations. Player has health of **H**. Spider and player's x position are less than 4 * 32 units that is a 4 block width.
Input:	No input, spider entity is created is generated by the game
Output:	Player health reduces by spiderDmg continously till player moves away from the spider by 4 blocks.
Pass:	Player health is H-spiderDmg and keeps reducing by spiderDmg till player and spider x coordinates differ by 4 * 32 units.

Test 3.1.12.4:	The spider remains dormant during the cool off stage after attacking the player in range.
Requirements:	FR60
Description:	Test the relative positions of the spider and player have to be greater than 4 blocks to cause the spider to detach from the player.
Type:	Unit test (dynamic, manual)
Initial State:	Spider entities created by an in-game map file placing them at specific map locations.
Input:	No input, spider entity is created is generated by the game
Output:	Spider and player coordinates differ by > 4 *32 units.
Pass:	If the x coordinates of the player and spider are > 4 *32 the spider should remain stationery at 4 blocks away from the player. The player's health remains unchanged.

Test 3.1.12.5:	Spider attacks the player killing the player
Requirements:	FR62
Description:	Tests whether the player dies if the health of the player is less than or equal to the spiderDmg
Type:	Unit test (dynamic, manual)
Initial State:	Spider entities created by an in-game map file placing them at specific map locations. Player has health of **H**.
Input:	No input, spider entity is created is generated by the game
Output:	Player health reduces to 0 resulting in game over
Pass:	Player health is 0

Test 3.1.12.6:	Spider continuously attacks the player killing the player
Requirements:	FR62
Description:	Tests whether the player dies if the player continues to stay in range of the spider that is a less than or equal to 4 blocks away.
Type:	Unit test (dynamic, manual)
Initial State:	Spider entities created by an in-game map file placing them at specific map locations. Player has health of **H**.
Input:	No input, spider entity is created is generated by the game
Output:	Player health continously reduces by spiderDmg and finally reduces to 0 resulting in game over
Pass:	Player health is 0

3.1.13 Arrow Trap

Test 3.1.13.1:	Arrow Trap Box Generated With The Game Starting.
Requirements:	FR63
Description:	Tests the existence of the trap in the game is generated correctly.
Type:	Unit Test (dynamic, manual)
Initial State:	The game is starting by running the game file with trap box object specified at position (x,y) on the map.
Input:	Game file initiated.
Output:	List of the trap objects in the game.
Pass:	List of trap objects in the game include an arrow trap box object.

Test 3.1.13.2:	Arrow Trap Box is placed on a block.
Requirements:	FR64
Description:	Tests the existence of the trap in the game is generated correctly.
Type:	Unit Test (dynamic, manual)
Initial State:	The game is starting by running the game file with trap box object specified at position (x,y) on the map.
Input:	Game file initiated.
Output:	Position of the trap object and position of blocks.
Pass:	Position of the trap object is above a block position by 1.

Test 3.1.13.3:	Arrow Trap Box Detects Player Approach from Left.
Requirements:	FR65, 66, 67
Description:	Tests if the arrow trap box detects the player approach the box position correctly.
Type:	Unit Test (dynamic, manual)
Initial State:	The player is moving at playerSpeed toward the trap box position from the left and it is within the arrowSense range.
Input:	None.
Output:	List of the arrow objects in the game.
Pass:	List of arrow objects in the game includes arrowNum new arrows that was created by the trap.

Test 3.1.13.4:	Arrow Trap Box Detects Player Approach from Right.
Requirements:	FR65, 66, 67
Description:	Tests if the arrow trap box detects the player approach the box position correctly.
Type:	Unit Test (dynamic, manual)
Initial State:	The player is moving at playerSpeed toward the trap box position from the right and it is within the arrowSense range.
Input:	None.
Output:	List of the arrow objects in the game.
Pass:	List of arrow objects in the game includes arrowNum new arrows that was created by the trap.

Test 3.1.13.5:	Arrow Trap Box Can't Shoot Arrow When there is an arrow in action shot by the same box.
Requirements:	FR67
Description:	Tests if the arrow box generates only one arrow upon detecting a player from the left and then the player moves to the right.
Type:	Unit Test (dynamic, manual)
Initial State:	The player was detected by the arrow trap box within the arrowSense range from the left but at the same time the player changes position to the right and enters the area of arrowSense range again.
Input:	None.
Output:	List of arrow objects in the game.
Pass:	List of arrow objects in the game has only one arrow object created.

Test 3.1.13.6:	Arrow Trap Box Shoots Arrow Toward the Player from Left.
Requirements:	FR65, 66
Description:	Tests if the arrow is shot in the correct direction where the player was detected left of the trap box.
Type:	Unit Test (dynamic, manual)
Initial State:	The arrow trap box detected the player that is approaching from the left.
Input:	None.
Output:	Arrow speed and direction of movement.
Pass:	Arrow speed is arrowSpeed and it is moving in the left direction (negative x-direction).

Test 3.1.13.7:	Arrow Trap Box Shoots Arrow Toward the Player from Right.
Requirements:	FR65, 66
Description:	Tests if the arrow is shot in the correct direction where the player was detected right of the trap box.
Type:	Unit Test (dynamic, manual)
Initial State:	The arrow trap box detected the player that is approaching from the right.
Input:	None.
Output:	Arrow speed and direction of movement.
Pass:	Arrow speed is arrowSpeed and it is moving in the right direction (positive x-direction)

Test 3.1.13.8:	Arrow Hits the Static Player.
Requirements:	FR71
Description:	Tests if the arrow collision with a static player is successful that results in decreasing the player's health.
Type:	Unit Test (dynamic, manual)
Initial State:	The arrow is moving in the direction at which the player was detected from.
Input:	None.
Output:	Player health.
Pass:	Player health was reduced by arrowDmg.

Test 3.1.13.9:	Arrow Hit the Moving Player.
Requirements:	FR71
Description:	Tests if the arrow collision with a moving player is successful that results in decreasing the player's health.
Type:	Unit Test (dynamic, manual)
Initial State:	The arrow is moving in the direction at which the player was detected from.
Input:	None.
Output:	Player health.
Pass:	Player health was reduced by arrowDmg.

Test 3.1.13.10:	Arrow fly in a straight line unaffected by gravity.
Requirements:	FR69
Description:	Tests if the arrow stays at the same level throughout its movement from when it was shot by the trap.
Type:	Unit Test (dynamic, manual)
Initial State:	The arrow is moving in the direction at which the player was detected from.
Input:	None.
Output:	Arrow y-position.
Pass:	Arrow y-position remains constant.

Test 3.1.13.11:	Arrow fly stops when encountering an object in the game.
Requirements:	FR70
Description:	Tests if the arrow will stop shooting when colliding with an object in the game.
Type:	Unit Test (dynamic, manual)
Initial State:	The arrow is moving in the direction at which the player was detected from.
Input:	None.
Output:	Arrow speed.
Pass:	Arrow speed is zero.

Test 3.1.13.12:	Arrow self destructs when encountering an object in the game.
Requirements:	FR70
Description:	Tests if the arrow will cease to exist when colliding with an object in the game.
Type:	Unit Test (dynamic, manual)
Initial State:	The arrow is moving in the direction at which the player was detected from.
Input:	None.
Output:	List of arrow objects.
Pass:	Empty list of arrow objects.

Test 3.1.13.13:	Arrow pickup
Requirements:	FR90
Description:	Tests if a picked up arrow has the correct coordinates
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a player with coordinates $x1$ and $y1$ on a block and is near an arrow.
Input:	pickup() is called
Output:	Arrow coordinates $x2$ and $y2$.
Pass:	The arrow is in the player's hand.

Test 3.1.13.14:	Arrow throw
Requirements:	FR90
Description:	Tests if a thrown arrow is moving at the correct speed.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where an arrow has been thrown.
Input:	throw() is called.
Output:	Arrow speed.
Pass:	Arrow speed is throwSpeed horizontally.

3.1.14 Spike

Test 3.1.14.1:	Spike Trap Generated With The Game Starting.
Requirements:	FR63
Description:	Tests the existence of the trap in the game is generated correctly.
Type:	Unit Test (dynamic, manual).
Initial State:	The game is starting by running the game file with spike object specified at position (x, y) on the map.
Input:	Game file initiated.
Output:	List of the trap objects in the game.
Pass:	List of trap objects in the game include an spike trap object.

Test 3.1.14.2:	Spike Trap is placed on a block.
Requirements:	FR72
Description:	Tests the existence of the trap in the game is generated correctly.
Type:	Unit Test (dynamic, manual)
Initial State:	The game is starting by running the game file with trap box object specified at position (x,y) on the map.
Input:	Game file initiated.
Output:	Position of the trap object and position of blocks.
Pass:	Position of the trap object is above a block position by 1.

Test 3.1.14.3:	Spike Damages Player.
Requirements:	FR73
Description:	Tests if the spike damages the player when the player steps on the trap.
Type:	Unit Test (dynamic, manual).
Initial State:	The player has collided with the static spike trap and they are both at the same position (x, y).
Input:	None.
Output:	Player health.
Pass:	Player health was reduced by spikeDmg.

3.1.15 Game Ending

Test 3.1.15.1:	Exit door
Requirements:	FR75
Description:	Tests if the game ends when reaching the exit door.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where the player is near the exit door.
Input:	use() is called.
Output:	None.
Pass:	The player wins.

Test 3.1.15.2:	Score save
Requirements:	FR75, 76
Description:	Tests if the user name and score are saved.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where the player has won and entered their name.
Input:	User's name and score.
Output:	Score save file.
Pass:	The user name and score is save to the file.

Test 3.1.15.3:	Score display
Requirements:	FR77
Description:	Tests if the scores are displayed.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state with a non-empty score save file.
Input:	keyScoreboard is pressed.
Output:	None.
Pass:	The top scores are displayed.

3.1.16 Level

Test 3.1.16.1:	Level creation
Requirements:	FR78, 79
Description:	Tests if a level is created correctly based on the map.
Type:	Unit test (dynamic, manual)
Initial State:	The game has launched.
Input:	ReadMap() is called.
Output:	None.
Pass:	All blocks, doors, and entities are as specified in the map.

Test 3.1.16.2:	Player starting location
Requirements:	FR80
Description:	Tests if the player is placed correctly based on the map.
Type:	Unit test (dynamic, manual)
Initial State:	The game has launched.
Input:	ReadMap() is called.
Output:	None.
Pass:	The player starts at the entrance door.

Test 3.1.16.3:	Clear path
Requirements:	FR81
Description:	Tests if there is a clear path in the level.
Type:	Unit test (dynamic, manual)
Initial State:	The game has launched.
Input:	ReadMap() is called.
Output:	None.
Pass:	There is a sequence of empty blocks from the entrance door to the exit door.

Test 3.1.16.4:	Block types
Requirements:	FR82
Description:	Tests if all blocks are correct based on the map.
Type:	Unit test (dynamic, manual)
Initial State:	The game has launched.
Input:	ReadMap() is called.
Output:	None.
Pass:	All blocks are either solid or empty as specified in the map.

3.1.17 Sign

Test 3.1.17.1:	Sign placement
Requirements:	FR85
Description:	Tests if a sign has the correct coordinates based on the map.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state where a sign has coordinates $x1$ and $y1$.
Input:	makeEnt() is called.
Output:	None.
Pass:	$x1$ and $y1$ are correct based on the map.

Test 3.1.17.2:	Sign text
Requirements:	FR86
Description:	Tests if a sign has the correct text based on the map.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state with a sign.
Input:	makeEnt() is called.
Output:	Sign message.
Pass:	Sign message is correct based on the map.

Test 3.1.17.3:	Sign reading
Requirements:	FR87
Description:	Tests if a sign displays the message correctly.
Type:	Unit test (dynamic, manual)
Initial State:	Custom game state with a player in front of a sign.
Input:	None.
Output:	None.
Pass:	The sign message is displayed at the bottom of the screen.

3.2 Tests for Nonfunctional Requirements

3.2.1 Usability Testing

Test 3.2.1.1:	Game feeling
Requirements :	NFR1
Description:	Tests if the feeling of the game matches expectations.
Type:	Usability(dynamic, manual, functional)
Pass:	Users are given 10 minutes to play the game. Passes if 80% of players feel like they're playing Spelunky.

Test 3.2.1.2:	Immersion
Requirements :	NFR2
Description:	Tests if users feeling like they are in the game.
Type:	Usability(dynamic, manual, functional)
Pass:	Users are given 10 minutes to play the game. Passes if 80% of players feel like they feel like the player himself.

Test 3.2.1.3:	Control feeling
Requirements :	NFR3, NFR9
Description:	Tests if the controls create instant changes.
Type:	Usability(dynamic, manual, functional)
Pass:	All inputs made by the user immediately move the character.

Test 3.2.1.4:	Game understanding
Requirements :	NFR4, NFR5
Description:	Tests if the game provides users with a good understanding of it.
Type:	Usability(dynamic, manual, functional)
Pass:	Users are given 10 minutes to play the game. Passes if 80% of players can play without outside help.

Test 3.2.1.5:	Game progress
Requirements :	NFR6
Description:	Tests if the game provides users with a good understanding of their progress.
Type:	Usability(dynamic, manual, functional)
Pass:	Signs are located in major parts of the game detailing progress.

Test 3.2.1.6:	Game text
Requirements :	NFR7
Description:	Tests if the game has clear and legible text.
Type:	Usability(dynamic, manual, functional)
Pass:	All game text is size 10 Verdana and is legible to users.

3.2.2 Stress Testing

Test 3.2.2.1:	Large maps
Requirements :	NFR10
Description:	Tests the game with a 1000 by 1000 map.
Type:	Stress(dynamic, manual, functional)
Pass:	The game operate normally in at least 30 fps.

Test 3.2.2.2:	Large number of entities
Requirements :	NFR10
Description:	Tests the game with a 1000 entities.
Type:	Stress(dynamic, manual, functional)
Pass:	The game operate normally in at least 30 fps.

3.2.3 Performance Testing

Test 3.2.3.1:	Game response time
Requirements :	NFR8
Description:	Tests the game's response time.
Type:	Performance(dynamic, manual, functional)
Pass:	The game responds to all user inputs in at least 10ms.

Test 3.2.3.2:	Game frame rate
Requirements :	NFR10
Description:	Tests the game's framerate.
Type:	Performance(dynamic, manual, functional)
Pass:	The game operates in at least 30 fps.

Test 3.2.3.3:	Game storage
Requirements :	NFR11
Description:	Tests the game's storage size.
Type:	Performance(static, manual, functional)
Pass:	The game file size is at most 16GB.

Test 3.2.3.4:	Missing/incomplete file Exception handling
Requirements :	NFR25
Description:	Tests if the game handles errors relating to missing/incomplete files.
Type:	Performance(dynamic, manual, functional)
Pass:	The game terminates with an error message if a file is missing/incomplete.

3.3 Traceability Between Test Cases and Requirements

All the test cases in this document have a direct relation to the system requirement stated in the SRS document. Each test case is designed to prove at least one requirement. The traceability between the test cases and the requirements have been explicitly mentioned in the above test cases.

4 Tests for Proof of Concept

4.1 Display

The PoC tests covered the basic camera and user interface. The camera is fixated on the player and moves within the boundaries of the map to keep the player in focus. The player's health and gold is displayed and updated when taking damage and collecting treasure. (3.1.2)

4.2 Player

The PoC tests covered the movement, states, and actions of the player. The player can move in four directions and has idle, walking, falling, climbing, and crouching states. Each has their own specific movement restrictions. Idle has no movement. Walking has left and right. Falling has all 4 directions while in the air. Climbing has up and down only on ropes. And crouching is for holding down while idle. The player can jump, open chests, collect treasure, place bombs, throw ropes, pickup/throw chests, and use a whip. (3.1.3)

4.3 Bomb

The PoC tests covered creating, counting down, and exploding bombs. Bombs are created with a timer. The timer ticks down every second. Reaching 0 creates a explosion that damages everything nearby. (3.1.4)

4.4 Rope

The PoC tests covered creating, and extending ropes. Ropes can be thrown up and attach to a block or the background upon travelling the maximum distance. They can also be placed in front without throwing. (3.1.5)

4.5 Weapon

The PoC tests covered using the whip. It damages any enemies it touches while present, (3.1.6)

4.6 Snake

The PoC tests covered a player-snake interaction. This involved the player attack and snake attack. The player is able to defend themselves using a whip which can damage the health of a spider or stomp on a snake to completely kill it. Furthermore, if the player does not defend themselves then they can get health damage if the snake moves over them. (3.1.11)

4.7 Collect treasure

The PoC tests covered a the concept of a chest and collecting treasure coming from it. If the player comes in contact with treasure they can whose to open it. This results in a gold treasure that was in the box being placed where the treasure was originally. Next the player can choose whether or not they wish to collect the treasure. If they do then it increases their gold score. (3.1.7)

4.8 Arrow Trap

The PoC tests covered the arrow trap generation and the collision of the player with the shot arrow. It tested the arrow trap sensor to validate the behaviour of the trap follows the requirements. The PoC tests showed arrow trap was generated properly with a working sensor and ability to shoot arrows upon the player stepping into the attack area of the trap. (3.1.13)

4.9 Spike Trap

The PoC tests covered the spike trap generation and the ability to damage player upon a player stepping on a trap. The PoC showed that the system passes all the spike trap tests and it functions fully according to the functional requirements. (3.1.14)

4.10 Level

The PoC tests covered level generation. Levels were created from a map file. The location and types of all entities are specified and the layout of blocks in the level.(3.1.16)

4.11 Sign

The PoC tests covered displaying signs. Sign messages were shown at the bottom of the screen when the player was near one, (3.1.17)

5 Comparison to Existing Implementation

The existing functionalities only involve:

1. Player movement: The player can jump and move right and left using keyboard inputs. Damage is taken for falling too far. 3.1.3
2. Map entities: Entities such as treasure, signs, map solid blocks and block spaces are created as part of the game map as soon as the game is started. 3.1.9
3. Chests: Chests can be interacted with and creates gold bars or rubies that are collected automatically. 3.1.9
4. Display: The camera is focused on the player. Health and gold are displayed. 3.1.2
5. Sign: Sign messages are displayed at the bottom of the screen when the player is near them. 3.1.17

6 Unit Testing Plan

Unit Testing will be executed manually by running the game and following the test cases provided in section 3. Unit testing will be performed bottom-up. The individual parts of modules will be tested first and then the parts that require other modules. Since the modules are complete, there is no need for stubs or drivers. Unit testing coverage metrics include achieving 90% statement code coverage that will be done through the use of the code coverage tool Coverage.py. Not only that but also the team will manually ensure that the unit test cases will cover all the modules of the system and their methods. Coverage...

6.1 Unit testing of internal functions

Internal functions will be tested using the tests specified in section 3. For every module, its functions are exercised in at least one of functional tests. The goal of each test is to verify and validate that the internal system functions meet the functional requirements of the system. These tests will be from the developer perspective that are executed by the developer inputs and manually. Therefore, the developer will have the choice to create more tests as seen fit.

6.2 Unit testing of output files

The score save file should have ~~2 columns separated by commas. The first column are names up to nameLength characters and the second is the corresponding scores~~ one entry of the highest score the user has attempted in the life time of the game being used.

7 Appendix

7.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

Table 5: Symbolic Parameter Table

Symbolic Parameter	Description	Value
camWidth	The width of the camera, in pixels.	320
camHeight	The height of the camera, in pixels.	320
Symbolic Parameter	Description	Value
keyEsc	The key to quit the game.	esc key
keyLeft	The key to move left.	left arrow key
keyRight	The key to move right.	right arrow key
keyUp	The key to move/look up.	up arrow key
keyDown	The key to move/look down.	down arrow key
keyJump	The key to jump.	space key
keyInt	The key to interact.	tab key
keyBomb	The key to use a bomb.	b key
keyRope	The key to use a rope.	v key
keyweap	The key to use the weapon.	left shift key
keyThrow	The key to throw object in empty space	left shift key
keyPick	The key to pick up object.	down arrow key and left shift key
keyCrouch	The key to crouch.	down arrow key
playerSpeed	The speed of the player in pixels per second.	90
playerJumpHeight	The maximum height of the players jump in pixels.	72
playerJumpSpeed	The speed the player reaches when jumping in pixels per second.	3
playerClimbSpeed	The speed of the player in pixels per second when climbing.	90
playerWidth	The width of the player, in pixels	15

playerHeight	The height of the player, in pixels	22
weaponDmg	The amount of damage dealt by the weapon.	1
weaponDelay	The delay between weapon uses in seconds	0.5
weaponAttack	The attack range of using the weapon in pixels	16
fallDmg	The amount of damage the player takes when falling in blocks.	1
fallDmgDist	The minimum fall distance in blocks the player must fall to take fall damage	4
heartStartAmount	The number of hearts the player starts the level with.	4
bombStartAmount	The number of bombs the player starts the level with.	4
ropeStartAmount	The number of ropes the player starts the level with.	4
bombTime	The number of seconds a bomb takes to explode.	2
bombSize	The explosion size of a bomb in pixels.	48.
throwSpeed	The speed of a thrown bomb in pixels per second.	90
camWidth	The width of the camera, in pixels.	320
camHeight	The height of the camera, in pixels.	320
bombHeight	The width of a bomb, in pixels.	12
bombWidth	The height of a bomb, in pixels.	12
fallSpeed	The speed a bomb falls, in pixels per second.	72
bombDmg	The amount of damage a bomb deals.	10
Ropelength	The length of a rope in blocks.	4
stompDmg	The amount of damage the player deals to an enemy when jumping on them.	1
snakeSpeed	The speed of a snake in pixels per second.	45
snakeHearts	The number of hearts a snake has.	1

snakeDmg	The amount of damage dealt by a snake.	1
spiderSense	The number of blocks to the player for a spider to be active.	4
spideJumpHeight	The height a spider jumps in pixels.	72
spideJumpDist	The distance a spider jumps in pixels.	72
spiderJumpTime	The amount of time a spider is spent jumping in frames.	24
spiderJumpDelay	The delay between spider jumps in seconds.	2
spiderHearts	The number of hearts a spider has.	1
spiderDmg	The amount of damage dealt by a spider.	1
arrowSense	The number of blocks an arrow trap can detect movement.	4
arrowNum	The number of arrows shot by an arrow trap.	1
arrowSpeed	The speed of an arrow in pixels per second.	90
arrowDmg	he amount of damage dealt by an arrow.	2
spikeDmg	The amount of damage dealt by landing on a spike.	4
valDiamond	The gold value of a diamond.	5000
valRuby	The gold value of a ruby	1600
valSapphire	The gold value of a sapphire.	1200
valEmerald	The gold value of an emerald.	800
valGoldBar	The gold value of a gold bar.	500
scoreCalc	The score amount.	the player's gold

7.2 Usability Survey Questions?

1. Is it easy to switch in between controls with the given positions?
2. Do the controls have all the necessary player functions?

3. Do the system respond correctly to the user input controllers?
4. Is the color scheme of MacLunky provide clear display of the game?
5. Is the text size and font type of MacLunky easy to read and recognize?
6. Did the game shut down unexpectedly?
7. Did the game pause unexpectedly and then continued or just simply stayed frozen?
8. Is the game's level of challenge close to the compared game out there?
9. Are the symbols and graphics easy to understand? If not what were they?
10. Is the game too long to play or too short?
11. Are there any issues with the player character?
12. Are there any issues with starting the game?