



THE PIXEL WIZARD

Test Plan

Abstract

This document outlines the testing plan that will be used to test the game the pixel wizard

Keith Nolan

G00351932

Introduction

This document contains the testing strategy that will be performed to test the game **the pixel wizard**. The document is being written in conjunction as part of the overall testing strategy for the company game development international.

About

The game in question is a 2d platformer in which the player takes on the role of a wizard navigating through a pixelated 2d world. The overall objective is to overcome enemies and bosses in each level to beat the game. Each level will have an increasing level of difficulty thus creating a greater challenge with each level passed and giving them a sense of achievement for the player when they reach the end or winning conditions.

The game is built around being user friendly which enables the user to get started quickly and figure out how to play the game easily.

To progress in this game the player can shoot projectiles in the form of fire balls from his staff killing the enemies that confront him on his path. If a fire ball (projectile) encounters an enemy, it will either kill them or take a percentage of health from their life force. This depends on the enemies starting health and the degree of damage that a projectile deals upon hitting a foe.

There are three levels which need to be completed. If the player gets passed all three of these levels by overcoming the obstacles faced, they will have beaten the game in which case they will be presented with a screen informing them they have completed the game.

Enemies can also shoot projectiles at the player. This creates a greater level of difficulty for the player as they must evade oncoming projectiles, using the various controls outlined in the design documents control mechanisms. The **controls** of the game are as follows:

<i>Action</i>	<i>PC</i>	<i>Mobile</i>
Move Forward	Right arrow key/D	Arrow on screen
Move Backwards	Left arrow key/A	Arrow on screen
Jump	Up arrow key/W	Arrow on screen
Crouch	C	Arrow on screen (hold)
Attack	Left mouse click/R	Dedicated button
Pause/Resume	Spacebar	Button in top right of screen

Health

If the player is dealt damage from an enemy attack, they can replenish their health by collecting health pickups. These pickups will replenish some lost health to the player's health. The player then has a greater chance of surviving longer in the game. When the player walks into or collects the health it will then be added to their health UI. The health UI is displayed on the top left-hand side of the screen.

Each time the player has been hit by an enemy projectile their health will decrease by one, each time they collect a health pickup their health will increase by one. The player will be able keep playing the game if they have health. If the players health has decreased completely, they will die and must start the game again.

Enemy Health

The same method is applied for the enemy health. Generic enemies do not have a health bar displayed on the screen as it is likely that after being hit by one or two projectiles from the player they will die. The bosses on the other hand have their health displayed on the screen. This is a useful tool for the player as they know how many more times the boss must be hit by a projectile before that boss is dead and they can advance in the game.

Pause game

While in the main game the player can pause the game, which stops the game play. Which can be achieved by pressing the space bar. Upon pausing the game, the player is faced with various options that they can select these are outlined below.

Save game

If the player chooses to save the game, this will save their progress in the game. This progress can then be loaded from the main menu. This will allow the player to start from the point at which they saved the game.

Settings

If the player chooses the settings option, they can adjust or change the sounds and music in the game.

Restart

If this option is chosen the player will restart the current level they are in, all progress will be lost, and the player starts again

Exit game

If the exit game is chosen, the game will quit, and the player will leave the game.

Continue

Like any pause setting the game can be un-paused and the player will continue from the point and time at which they paused the game. This can be achieved by pressing the space bar again.

Main menu

When the game is first loaded the player is presented with a main menu that has three options. These options are listed below:

Play/load/save/delete game

If this button is selected it will bring the player into the game. What it loads exactly will depend on if the player has saved the game or not. If the player has no saved game data, it will bring the player to the first level will be loaded. A display will also appear outlining to the player the controls used to play the game. Once the player is happy with the controls they can continue and start playing the game.

If the player has saved game data and chooses the load button, the saved data will be loaded, and they will be taken to the point at which they saved the game.

If the delete save game button is chosen it will delete the players saved game data and they will have to start the game again.

Settings






If the settings button is chosen, the player will be taken to a menu where they can adjust or turn off the sound settings for the game.





Exit game

If this option is chosen, the application will quit.

Assets

The assets used in this game refer to how the game looks, what artwork it has. This ranges from the player character, the enemies, the projectiles, the scene itself. The assets used are listed below these have been taken from the design document.

Asset Name	Asset
Background	
Player Character	
Enemy Character	
Player/Enemy Projectile	
Ground Asset	

Platform One Asset	
Platform Two Asset	
Health Pickup	
Rock Asset	
Menu Logo	

Objectives and Tasks

Objectives

The objective of this test plan is to test the features of this game as stated in the design document and ensure they (the features) do what the game states before the release of the game for commercial use.

I have been tasked with testing this game as part of a white box testing effort. I will be testing this with full knowledge of the internals of the game as part of a service level agreement by game development international.

We the testers will be responsible for testing every aspect of the game from the features of the main menu to the actual game and game play itself.

Tasks

Testing

Myself and my team will test all the features of this application to ensure they all do their intended tasks, and everything works properly and within the scope of the application. This is to make sure that there are no major errors in this application upon it's releases to the public.

Problem reporting

Any problems with the application will be documented in the test case document. At the end of each day game development international will be informed of my findings and errors (if any) on each topic covered for their own records so, they can act accordingly about how to fix these errors if necessary or make a note of passing tests.

Post testing

We have also been tasked with post testing the application as well.

After the game has been released, I will continue testing it, reading any possible bugs reported by players and testing to see if those bugs reported are accurate. I will then report any findings to game development international. Where they can take the steps to develop new patches or updates to fix these reported bugs if they deem it necessary.

Scope

General

This section describes what is being tested which is as follows:

Opening/closing the application

- Test to make sure that the application opens when the user clicks on the application
- Test that the user is brought to the main menu upon opening the game
- Test to ensure that if the user opts to exits the game that the application quits as it should.

Menus

Main menu

- Test that the play button on the main menu brings the player to the game
- Test the save/load and delete game data features to ensure they do what they are specified to do.
- Test the volume and setting controls to ensure they do the specific task.
- Ensure the correct artwork is displayed on the main menu
- Ensure buttons are responsive and work correctly

Pause menu

- Ensure the space bar pauses and un pauses the game
- Ensure that if the game is saved, that it saves the data
- Test the volume and setting controls in settings to ensure they do the specific task.
- Test that the exit button exits the game
- Test that the gameplay stops when the pause button is pressed and resumes when pressed again.

Game play

- Test the enemy AI to ensure they behave within the constraints of the game.
- Test that the assigned buttons do the correct task
- Test the health UI associated with player and enemy boss so that they are responsive to damage
- Ensure that projectiles kill enemies and enemy projectiles deal damage to player
- Test that health pickups give player health and disappear after they have been collected
- Test the canvas and box colliders to ensure the player and or enemies cannot fall off the screen or run off the edge of game canvas.
- Testing death conditions
- Win conditions

- I will be testing to see if the game play is smooth and without glitches to bring a positive experience to the user. Ex the background transitions smoothly and the end of each cycle.
- Ensure that players health cannot go above the specified amount

Assets

- Test that assets appear correctly in the game
- Test that assets are responsive
- Test that animations look correct and transitions happen at the appropriate time.
- Test the user interface from a customer's point of view to ensure that it is easy to use and is understandable.

Underlying code

- Ensure that all functions and classes operate as intended
- Ensure that all code works properly when integrated together
- Ensure no errors are present in the code
- Ensure front end is responsive with the backend and vice versa

User Experience

- Ensure that the game and game play is user friendly and easy to comprehend for every user

Tactics

This is how I will accomplish the items that you have listed in the “Scope” section.

The testing process will be made up of several activities.

Unit testing

This testing strategy will focus on and test the classes and functions in this application. The plan is to test all the independent paths in the code and uncover the errors in that component. This will verify that data flows in and out of the component correctly. That the data is being processed correctly inside the component and maintains integrity within local data structures. It will also verify error handling and test the components in isolation.

The unit testing will be done with a testing suite that is relevant to the language the code is written in. From there I will write tests based on various functions and classes and ensure that they pass these tests. I will keep record of my completed tests in a test case document to make note of any tests that pass, as well as any tests that do not pass.

This will also include testing to make sure the application runs on all the required platforms and operating systems.

Integration Testing

In this testing effort different parts of the application will be tested working together. This is done to expose faults when different components are integrated together. A class or feature might work perfectly on its own, but when integrated with other classes, features etc they may not work as intended. These tests will be done to check that this type of error does not occur.

Playtesting

This series of tests is part of quality control (QC) will test how well the game plays and will help us work out the flaws in the game if there are any. It will help ensure that gameplay is a positive, smooth, and responsive experience for the user.

This is a follow up test to the unit testing and ensures that everything functions properly on the front end of the application and ensure that the game is fun for the player. These tests will include testing the menus, in game play, pause menu, enemies AI, winning conditions, losing conditions from a player’s point of view. Ensuring that each button and mouse movements do the correct task.

Acceptance testing

This will be a follow up to the previous three tests. This will test the users experience when playing the game to ensure that they feel like the game is user friendly and all features are well laid out, well documented and easy to comprehend. User stories will provide feedback on how they feel the game is and perhaps what they think is missing.

Testing strategy

This section describes the overall approach to testing. Which will be implemented as follows:

1. Unit testing

Unit tests will be carried out on a testing suite that is applicable to the code and environment used to create this application. Let us assume that this application is written in C# and was developed using the unity framework.

In this circumstance I will use the unity test framework. Which allows me to run unit tests on my C# code. It enables me to test the code in both the edit and play mode. Testing all the classes, functions, and the code within to ensure that they are working correctly and doing what they were designed to do.

Writing the tests will involve setting up test classes that test the relevant functional classes. This will involve testing the various methods within those classes, entering in the required values to run those tests specifying an expected result and comparing that to the actual result. If all goes well those tests will pass. This will be the testing method for testing the subsequent classes and functions as well.

An example of unit testing in this circumstance would be testing the players ability to walk, shoot, jump and run. These are core features of the game and at least on of these statements is likely to be ran quite often

Below is an example of a test case scenario where the players movement is being tested. The testing class includes the unity test attribute to ensure that tests are ran. These tests are checking to see if user input causes the player to move along the x and z axis.

```
using UnityEngine;

public class Player : MonoBehaviour
{
    public float Speed;

    private void Update()
    {
        var h = Input.GetAxisRaw("Horizontal");
        var v = Input.GetAxisRaw("Vertical");

        var x = h * Speed * Time.deltaTime;
        var z = v * Speed * Time.deltaTime;

        var movement = new Vector3(x, 0, z);

        transform.position += movement;
    }
}
```

```

{
    public class PlayerTests
    {
        [UnityTest]
        public IEnumerator Moves_Alone_X_Axis_For_Horizontal_Input()
        {
            var player = new GameObject().AddComponent<Player>();
            player.Speed = 1;

            yield return null;

            Assert.AreEqual(1, player.transform.position.x, 0.1f);
        }

        [UnityTest]
        public IEnumerator Moves_Alone_Z_Axis_For_Vertical_Input()
        {
            var player = new GameObject().AddComponent<Player>();
            player.Speed = 1;

            yield return null;

            Assert.AreEqual(1, player.transform.position.z, 0.1f);
        }
    }
}

```

Tests like these will be used to test each class and function to ensure that the code is running efficiently and properly. These include inputs, outputs, and overall logic of the game.

2. System and integration testing

As stated above this type of testing is done to test features from different parts of the program working together. This will expose defects in things like the interfaces and with interactions between different components.

System testing on the other hand is testing all the features working together. Integration testing will help to ease this process instead of taking an approach like “the big bang” where everything is just tested together straight away.

The methodology that myself and my team will be taking will be the “bottom up” approach. This is where we test the lower levels of the application first and gradually move up to the higher levels of the application. This approach is taken opposed to the “Top-down” approach which is in every way the opposite.

Implementation of this strategy

After the unit tests have been completed and results noted in the test case. We will begin to test the integration of the various classes and functions (working together as a unit). The same approach to the one used in the unit testing will be used. We will **assert** that the functions and classes work properly when implemented together.

A simple example of this would be the enemy's scripts. As there will be multiple enemies in this game they will all have base behaviours. So, they will have a base class that handles their A.I. For example, the damage function. All the enemies must have to ability to take damage when a successful attack from the player is implemented.

Below is an example of a base enemy class and the damage function within that class:

```
public class EnemyScript : MonoBehaviour
{
    //Health variable
    public int health;
    //Player
    [HideInInspector]
    public Transform player;
    //Speed variable
    public float speed;
    //variable to handle time between enemy attacks
    public float timeBetweenAttacks;
    //Variable to handle amount of damage dealt
    public int damage;
    //odds of enemy dropping a pickup
    public int pickupOdds;
    //Reference to pickups game object
    public GameObject[] pickups;
    //Variable to handle stop distance from enemy to player
    public float stopDistance;
    //Variable to handle time of attacks
    private float attackTime;
    //Variable to handle speed of attacks
    public float attackSpeed;

    public virtual void Start()
    {
        //assigning variable player to the object with the tag "Player"
        player = GameObject.FindGameObjectWithTag("Player").transform;
    } //Start

    private void Update()
    {
        //if player is not = null (dead)
        if (player != null)
        {
            //if the enemy distance is to far away from the player
            if (Vector2.Distance(transform.position, player.position) > stopDistance)
            {
                //move towards player
                transform.position = Vector2.MoveTowards(transform.position, player.position, speed * Time.deltaTime);
            }
            else
            {
                if (Time.time >= attackTime)
                {
                    StartCoroutine(Attack());
                    attackTime = Time.time + timeBetweenAttacks;
                }
            }
        }
        //if/else
    } //Update

    public void Damage(int damageAmount)
    {
        //subtracting damage amount from health variable
        health -= damageAmount;
        //if the health is less than or equal to 0 destory the enemy object
        if (health <= 0)
        {
            //101 because the last number is excluded
            int randomNum = Random.Range(0, 101);
            //if the random number is less than the odds of dropping a pickup
            if (randomNum < pickupOdds)
            {
                GameObject randomPickup = pickups[Random.Range(0, pickups.Length)];
                //create a random pickup at the enemys final position
                Instantiate(randomPickup, transform.position, transform.rotation);
            }
            //if the enemys health is less than or equal to 0 give the player a score of 10
            score.scoreValue += 10;
            //destory the enemy object
            Destroy(gameObject);
        } //if
    } //Damage
}
```

Below here is an example of an enemy deriving from that class, but with some separate features that make them unique.

```
> 3rd Year stuff > John_Stuck > Assets > Scripts > Enemy Scripts > MeleeEnemy.cs
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class MeleeEnemy : EnemyScript //Derived from enemy script
6  { //Enemy will chase player until it has reached stop distance
7      public float stopDistance;
8      //Variable to handle attack time
9      private float attackTime;
10     //Variable to handle attack speed
11     public float attackSpeed;
12
13     private void Update()
14     { //if player is not = null(dead)
15         if(player!=null){
16             //if the enemy distance is to far away from the player
17             if(Vector2.Distance(transform.position, player.position)>stopDistance){
18                 //move towards players position
19                 transform.position = Vector2.MoveTowards(transform.position, player.position, speed * Time.deltaTime);
20             }else{
21                 //if the time is greater than the attack time
22                 if(Time.time >= attackTime){
23                     //call attack method from enemyScript
24                     StartCoroutine(Attack());
25                     //Setting up attack time variable assigning it to Time.time + timeBetween attacks
26                     attackTime = Time.time + timeBetweenAttacks;
27                 }
28             }//if/else
29         }//if
30     }//update
31     //get player component call the damage function and deal x amount of damage
32     IEnumerator Attack(){
33         player.GetComponent<Player>().Damage(damage);
34         //position of melee enemy before he leaps towards the player
35         Vector2 originalPosition = transform.position;
36         //leap towards the players current position
37         Vector2 targetPosition = player.position;
38         //how much of animation has been done.
39         float percent = 0;
40         while(percent <=1){
41             //acts as a counter, every frame it will add a little bit to the percent variable
42             percent += Time.deltaTime * attackSpeed;
```

An integration test would include testing the default enemy script with the unique behaviour that this enemy has to make sure that enemies get or inherit the behaviours of both classes so it inherits abilities like the function to take damage but with its own behaviour such as the way in which it attacks the player.

Similar approaches will be taken when integrating the whole project and when it comes to integrating the entire system. When the entire system is integrated and working to a good level this phase of the testing will be finished. Should major errors occur when the system has been integrated these will have to be fixed by developers. Minute errors will not be as detrimental.

3. Play testing

This series of tests will be Play testing will be done to test how the game plays and ensuring that is up to a high standard.

Questions we will be asking ourselves here include:

- how fun the game is to play?
- does it get boring quickly?
- Will the user get what they paid for with his game?
- Is it something they will come back and play?
- Are transitions from scene to scene level to level smooth?
- Are enemy behaviours and player behaviours convincing and up to standard?
- Do the save and load game features work?

- Is the game simple to comprehend or will the player have to spend a lot of time “learning” how to play?
- Do objects stay where they are supposed to?
- Is background movement convincing?
- Are player and enemy movements and animations realistic?
- Are transitions between those animations convincing?

Asking ourselves questions like these will ensure a high standard is met. This will be done by a small team of chosen individuals from our group. Our findings will be noted.

Acceptance testing

This will be the final testing process, in which we will outsource some users who have never played the game to play it and report what they think in the form of user stories. A user story is information about what a user thinks about the product being tested. They give their positives negatives and any features that they think should be included.

This will give us an idea of what customers think about the game before it is released. It gives us an advantage to get ahead before the release to include these vital points. There are certain things that users expect from a product and not having these included could be detrimental to its success

Test cases

The tests and their results will then be added to the test cases which will include what was tested, what are the inputs, outputs, expected results, actual results and whether the test passes or fails.

Failing tests will be noted with a level of severity from 1-3. Should the error be a major showstopper such as when the player walks or shoots, they get an error or other entities such as enemies give errors when they move or shoot. This would be considered a major error as the player would be totally unable to play the game. Error with this type of severity will be sent back to game development international immediately as they cannot release a game with these types of errors.

On the other hand, if the game contains minute errors such as the game takes a few seconds more to load than expected, these errors will be noted but will not require immediate action.

