MEDICAL GAMING SOLUTIONS

LEVELING UP HEALTHCARE THROUGH GAMING

Software Testing Plan

3-22-4

Version 1.0

Project Sponsor: Dr. Ashish Amresh

Faculty Member: Tayyaba Shaheen

Team Members:

Rain Bigsby, Veronica Cardenas,

Ethan Ikhifa, Lenin Valdivia

This document outlines the activities that ensure the necessary functional and non-functional characteristics are achieved within our project. It describes the type of testing that is going to take place to check the quality and effectiveness of our software.

Table of Contents

1.0 INTRODUCTION

COVID-19 vaccination could play a critical role in not only improving the symptoms, but preventing hospitalizations and deaths related to such a disease as well. As of May 3, 2023, about 8.3 million U.S. children aged 12-17 and 17.4 million children aged 5-11 had yet to receive their first COVID-19 vaccine dose [2]. Now, even with the apparent need to promote vaccine uptake among adolescents, there are currently no intervention studies underway aimed at improving these vaccination rates. We are designing a content agnostic game development framework that can be used to rapidly make games for different domains, such as COVID-19 and HPV. To increase the likelihood of improving vaccination rates, we are designing within the context of clinics where teens and their parents frequent and are waiting for their appointments. The games must therefore be fast-paced, fun, and at the same time provide a range of decision-making choices to fully engage the adolescents in addressing the outcomes. Our framework will focus on making games consisting of repetitive, quick “bursts” of gameplay that promotes learning by doing while minimizing the cost of failure and frustration for the player.

This document will examine Team Medical Gaming Solutions software testing regime. Software testing is the process of ensuring that one’s software product is achieving its intended purpose. This is examined through verification of essential components and the various types of software testing such as code review, functional testing, and unit testing.

To effectively test our game development framework, we will be utilizing common video game testing for quality assurance. These techniques include Unit, Integration, and Usability testing. Unit testing will be conducted by creating small test cases that can execute specific behaviors of our code. These tests can be run on our scripts, GameObjects, and other components of our framework. Integration testing is fundamentally the main objective of our framework. This is completed by creating the two use cases utilizing the components of our framework. Finally, usability testing will be conducted with beginner game developers who are familiar with the technologies used in our framework.

Our testing plan can be summarized as conducting Unit, Integration, Usability testing, and Functional testing. Such a testing plan is appropriate for our burst game framework, as the nature of framework development not only depends on individual components working on their own, but it is crucial that said components can work in tandem; and that is why our most critical testing will be Integration and Usability testing, as that is the main function of our framework - developers should be able to use our framework to integrate their own requirements specified to their medical subject.

2.0 UNIT TESTING

2.1 Introduction to Unit Testing

Unit testing is an important part of software development, making sure that individual components or units of a system work properly. Its main purpose is to verify the behavior of each unit/component against expected functionality, finding defects in the development. By using thorough unit testing strategies, we enhance the overall quality, maintainability, and reliability of our framework. One thing of note is that since our product is a framework for burst video games, our end users are other game developers with varying levels of experience in game development; in other words, our end users could be a researchers with little to no experience in game development or a well-versed unity game developer.

Our Process and Tools:

Our unit testing approach will focus on testing each independently developed component within our game framework in an environment that closely replicates our real-world use cases. we will use the Unity Test Runner, an integrated tool within the Unity Editor, to execute our unit tests within the Unity environment. This integration enables developers to run tests easily as part of their regular development process, ensuring any issues are identified and addressed promptly.

Testing Units:

Our unit testing efforts will primarily focus on testing the following key components of our game framework.

1. **Main Menu Component and GUI Customization**: We will validate the functionality of the navigation option and user input handling.

2. **Level Manager:** We will ensure proper spawning of players, determination of boundaries, management of checkpoints, and handling of respawning as well as switching between scenes.

3. **Environment Components (Platforms, Obstacles, etc.):** We will test the behavior of platform movement, obstacle interaction, and jump pad functionality.

4. **Knowledge Drops:** We will validate the collection and recording of player actions and interactions with they interact with a knowledge drop object within this component.

5. **Character Components:** We will test player and non-playable character movement, collision handling, and AI scripts for auto behaviors like spawning.

Our focus on testing these key components is so that we can ensure that they are playing their role in the functionality and user experience of our game framework. By making sure that the correctness of these units, we can create a solid foundation for building and expanding our framework while minimizing the risk of introducing bugs or errors. Also, thorough unit testing of these components adds to the overall stability and reliability of our software, ultimately increasing the overall user experience of end-users.

2.2 Main Menu Component and GUI Customization

The main menu and GUI components are essential in delivering ease of use to the participants playing any video game. These two components are what allow the player to interact with the game by either starting the game, interacting with the game settings, learning the current stats about the character using the HUD, or interacting with knowledge drops. Without these components, it will be difficult, if not impossible, for a player to understand how the game functions or even play the game.

From a developers perspective, these two components must be completely customizable to fit their own medical setting. Developers must be able to move around images, buttons, and interfaces to their liking. These objects must also be fully customizable within the Unity inspector, meaning that the sprites used for the objects, size, rotation, and more are able to be switched out without affecting the way the game mechanics operate or the other components in general. There must also be an option to remove any GUI component, if the developer decides it is not necessary for their situation. For example, a developer can remove the time clock on the GUI without it affecting the health bar.

For unit testing purposes, these two components will be assessed by inserting different sprites or moving around the GUI parent objects. The test will be conducted by ensuring that no other components are affected by the dynamic GUI and main menu objects. The easiest way to determine if the GUI component works is to remove one of the HUD objects and check if it causes errors to the script attached.

2.3 Level Manager

The level manager oversees multiple subcomponents that are essential for game progression. It handles character spawning, determining boundaries, checkpoints, and loading the next scene after level failure or hitting a checkpoint. This component is essential to have when employed within the context of a fast-paced

game played within a short time period. Some of these subjects are visible to the player, which can influence their decision making process making this component fundamental in terms of our clients requirements.

The level manager is a tool for developers to utilize. It is mostly fixed code, meaning that the only customization that is available is done within Unity's inspector. Developers should be able to add the next scene they would like to spawn into, customize the respawn delay, add a fall detector, and add checkpoints as well. Respawning a player is also an optional choice for a developer.

Unit testing the level manager will be executed by adding different GameObjects that correlate to the player, fall detector, and scenes within any game. Although these subcomponents are all individual, they rely on each other to progress to the next scene that the developer chooses to load. That is why they all fall under the level manager. The unit test will pass if the debug console says that the player collided with either the checkpoint or the fall detector. If this message is displayed in the console, a new scene should load.

2.4 Environment Components

The environment component will allow developers to choose different objects to fill in their own levels. These objects include obstacles, jump pads, and platforms for the player to navigate through a level. These objects should be fully customizable and scalable, sprites can be added to fit the theme of the game and they should also be able to change the shape and size of the object. Unity offers pre-built scalability options within its inspector. Developers will be able to drag and drop the environment components and then edit it within the inspector.

This component can easily be unit tested by using the debug console, similar to the level manager. Whenever the player collides with the objects, it should display the action that occurred corresponding to the object. For example, when a player collides with the jump pad, the console should display the tag of the character and that the player jumped successfully. The player should then jump into the air after colliding with the jump pad. This will ensure that the environment components function correctly.

2.5 Knowledge Drops

To enforce the importance of education related to a medical subject, the knowledge drop component is essential to include within our framework. One of our clients requirements is to have a "specific behavior change outcome", which in our situation is to increase vaccination rates. These knowledge drops are implemented by including interactive text about the medical subject, which the player can collect and increase their score by interacting with them. This serves as an incentive for interacting with the knowledge drop.

To implement knowledge drops within these medical-based video games, a developer can drag the knowledge drop prefab into their Unity hierarchy and edit the text within the inspector. They can also change how fast or slow they would like the text to appear if it is a dialogue knowledge drop. The scripts for the knowledge drops are fixed, but they can be added to the knowledge drop canvases depending on what the developers needs are.

To unit test this component, we will test if the display function is correctly called when the condition occurs within the script. This condition could be when a player interacts with a knowledge drop orb or when a player fails or passes a level. The unit test could be simply implemented within the knowledge drop script and displayed within the console while the game is running.

2.6 Character Components

The character component is crucial for our burst game framework. Without a character component, there is no game for the user to play. Developers will be able to customize the character movement by similar checking and unchecking boxes within the Unity inspector. Like the other components, simplicity is of the utmost importance when implementing a component into an environment. This is also applied to the character component where the developer has to drag and drop a character into their level. Users can control the character by jumping, walking, and avoiding enemies.

To unit test this component, the debug console will also be utilized and the results will be displayed once the game is played and the character is being controlled. For a unit test to pass, the character must respond correctly according to the user input, with the correct velocity imputed from the developer in the Unity inspector. For example, if the user input is to move right or left, the character must do the same.

3.0 INTEGRATION TESTING

3.1 Introduction

In this section, we are delving into how each component within our framework interacts with one another when attempting to put a game together. The goal of such integration testing, in our case, is to examine the success of an implementation of our framework. In particular, we are testing for a successful output when various components are put together. Now, to understand the rationale for utilizing such integration testing, one must understand the nature of developing any kind of framework: how the tools and components developed within the framework should be able to be seamlessly integrated together by any future developer. And given such a nature for framework development, it is only integral to test for individual components being able to work together.

3.2 Integration Points

Indeed, there are a handful amount of components within our framework to consider for integration testing. However, we will primarily focus on certain interactions between certain components, such as the level manager with the character component, as we anticipate that these interactions will occur the most when future developers utilize our framework to develop their medical literacy games. And so, our proposed effort to test for seamless integration between our components is as follows.

3.2.1 Integration Point 1: Intended Gameplay Loop is Realized

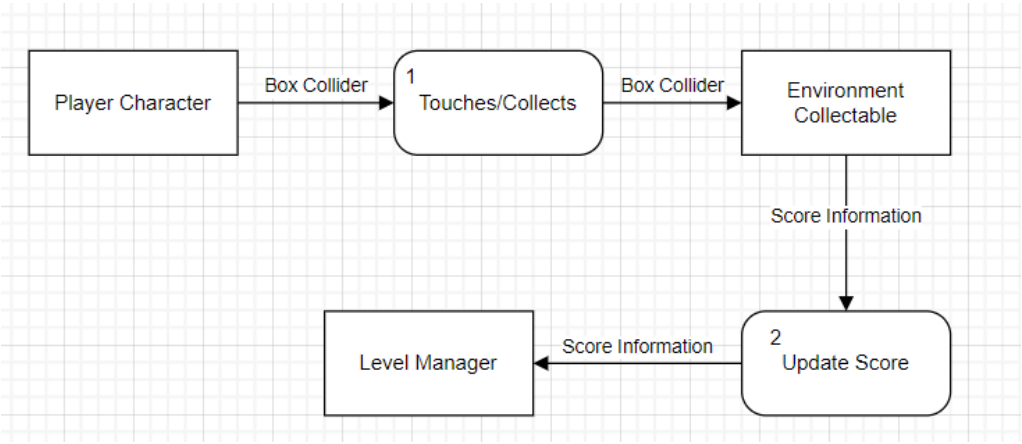
Modules involved: Level Manager, the Character component, and the Environment component.

Harness: There will be at least two mockup game scenes, each with a Character prefab representing the player, an empty game object with the level manager script attached to it, and various collectable prefabs from the Environment component.

Correct Integration: When the player touches a collectable, the collectable should disappear, and the level manager script should promptly update the score. Now, should the score reach the score goal set within the level manager script, the game should promptly end. Moreover, when the player touches the empty game object with the level manager script attached to it, the game scene should be switched to the next intended game scene set within the level manager script.

Contract: The Level Manager should only interact with the character prefab object with the “Player” tag attached to it.

3.2.A - Integration Point 1: In-Game Score Data Flow



3.2.2 Integration Point 2: Obstacles Kill the Player

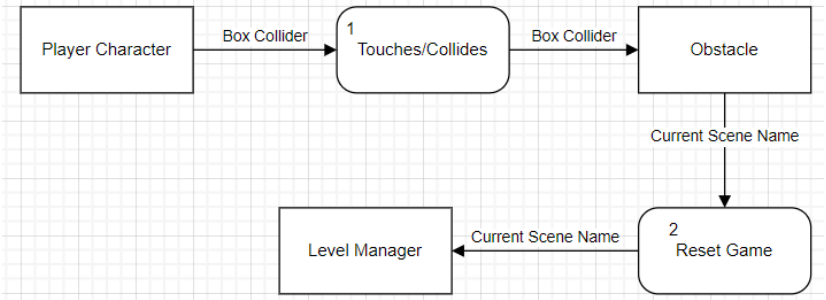
Modules involved: The Environment module, and the Character component.

Harness: There will be at least two mockup game scenes, each with a Character prefab representing the player, and various obstacle prefabs from the Environment component.

Correct Integration: When the player touches an obstacle prefab, the game should result in a game over, and the current level resets.

Contract: Each of the obstacle prefabs inserted into the scene should only interact with the character prefab object with the “Player” tag attached to it.

3.2.B - Integration Point 2: Game Over Data Flow



3.2.3 Integration Point 3: Platforms Prevent Characters From Falling

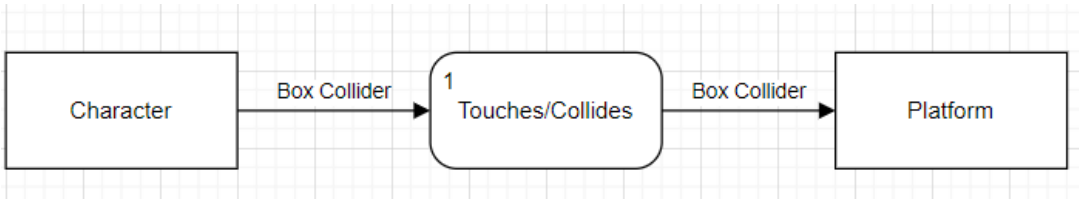
Modules involved: The Character component, and the Environment component.

Harness: There will be at least two mockup game scenes, each with a Character prefab representing the player, a Character prefab representing an NPC, and various platform prefabs from the Environment component.

Correct Integration: For whatever kind of platform prefab the developer inserts, be it the moving platform or a stationary one, it should act as a place in the game for any characters to be grounded.

Contract: Each of the platform prefabs should only interact with any character prefabs in the game scene.

3.2.C - Integration Point 3: Obstacle Collision Data Flow



3.2.4 Integration Point 4: Knowledge Drops Are Shown to the Player

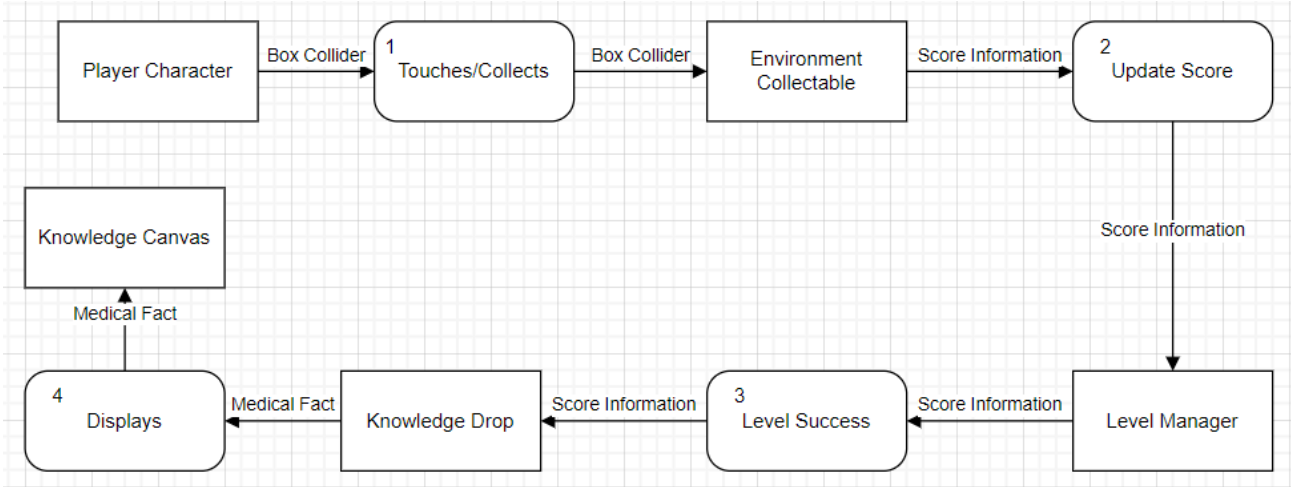
Modules involved: Level Manager, the Character component, the Environment component, and the Knowledge Drop component.

Harness: There will be at least two mockup game scenes, each with a Character prefab representing the player, an empty game object with the level manager script attached to it, and various collectable prefabs from the Environment component, and a Canvas game object representing the knowledge drop to be shown.

Correct Integration: When the player reaches the score goal after touching the last needed collectable, the knowledge drop canvas object should pop up as an indicator of a success on the level. Moreover, when the player is killed by an obstacle, the knowledge drop canvas object should pop as an indicator of a failure on the level. Finally, should the player interact with knowledge orb prefab by pressing the intended button on it, a text box with supposed factual information should appear.

Contract: Apart from the character component, every other component involved should only be limited to interacting with the character prefab with the "Player" tag attached to it.

3.2.D - Integration Point 4: Game Success Knowledge Drop Data Flow



4.0 USABILITY TESTING

4.1 Introduction

The quality of the user interface, as well as the overall framework, will be determined by the impression of the end user's experience with the project. The goal of usability testing is for us to examine feedback from users based on their capabilities and knowledge relating to the project. We can then refactor or reassess any technical aspects of the framework based on collective agreement on the outcomes of the tests. Overall, this category of testing mainly focuses on the aspects of the user interface being easy to understand and use, and improving in areas where a positive impact on the user's experience is lacking or leaves them confused.

4.2 Objectives

Before we initiate testing with our users, we must determine what type of users we need for the tests to be successful. As the project is quite specific, being a video game development framework, our range and variety in users will not be too large. We mainly intend to have users with some basic software development knowledge and users with game development skills and knowledge of Unity. A range from beginner to intermediate programmers should give us an accurate assessment of how clear or confusing the framework is for new users, depending on their skill level. The components within the framework must be clearly defined and usable since our goal is to speed up the development process for creating games.

With our users in mind, we seek to determine a proper assessment based on specific objectives that will benefit the finalized version of our video game framework. The results will allow us to maintain the framework to be the best possible product we can develop in our time frame.

4.2.1 Understandable Components

The included base components should be made clear by name or description within the comments on what its expected behavior is. As this is the main aspect that will make the whole framework, users should have very minimal confusion when interacting with the given components. This assessment will be based around what the users reactions are to a variety of components, as well as what behavior or additions they would like to see from specific ones.

4.2.2 Customization Friendly

All the base components included in the framework will offer a wide variety of mechanics to test and use for creating a game. However, it would be insufficient to limit the user on what they can create, and what they can allow the components to do. Therefore, we must assess the customization of the framework by determining how easy it is for users to modify existing components, while also gaining an understanding of how much time or lines of code are needed to create a desirable outcome with the framework. Component script modification will depend on the experience level of the users. If they are not comfortable with editing/writing C# scripts or do not have much experience with the language, guidance from our team can be offered in order to gain a proper assessment from beginner users.

4.2.3 Interactable Level

The overall framework needs to allow the user to make a completely functional level that can be designed with the included platforms and necessary game mechanics. This section is meant to assess the overall usability of the components and their interactions with each other. The framework is designed in a modular structure in which the components can behave independently when set up correctly. Any bugs found in testing the full level or interactions between components that are not made clear should be reported during testing and will be assessed accordingly.

4.3 Testing Plan

The objectives listed in section 4.2 will need a formally developed plan to accurately assess the outcomes. The end users for this testing case may vary in skill level as well, so the objectives should be straightforward in terms of execution. To abide by the nature of the framework, the plan should also not be too specific on what the steps are to allow freedom in users customization preferences. The plan will give users enough directions to start with for making a functional level, then will follow more open-ended instructions so they create their level as they wish. Therefore, after being given access to the framework, users will be given this guide for testing:

- User character component to choose style of gameplay for players
    - Select character component
    - Choose options in the prefab inspector to adjust gameplay settings (moves vertical, horizontal, jump, etc.)
    - If any additional code is needed to properly create your player character, please write how much was needed and any other comments here:

- Use environment components to design base level (platforms and obstacles)
    - Drag in platform component(s) (standard or movable) into the Unity scene
    - Select the obstacle component and select a given shape on the prefab inspector
    - Once selected, drag component into scene
    - If any additional code is needed to properly create your environment, please write how much was needed and any other comments here:

- Test interactions between components
    - Click the play button at the top of the Unity editor to test interactions between your character and the environment
    - Click play to exit when finished testing
    - Add an additional component(s) (knowledge drop, audio manager, level manager, etc.) of your choice
    - Click play again to test interaction between newly added components
    - If any additional code is needed to properly implement interactions, please write how much was needed and any other comments here:

This guide will be received virtually by users for their convenience. Once users have finished their level, they will be asked to share their final thoughts/comments on the framework as a whole, where our team will take notes on their reports. These reports will also be discussed virtually on the end users desired platform.

4.4 Testing Outcomes

Our expectations for these tests are to find some reasonable reports from users that will require our maintenance during the final steps of development. We do not expect the testers to have a completely perfect experience with this version of the framework, but rather have them gain a general understanding of the project and what they can do with it. Our team has set some standards for what shall take priority based on the results of the tests, which includes fixing a bug/component that is reported by 50% or more of testers to ensure the majority of feedback is met with results in our time frame. Next, the team can vote on if any reports on components from less than 50% of users should be implemented, if we can realistically develop a functional change with our remaining time. Finally, we will plan to implement any minor changes from users feedback that are known to be realistically developed within a short time, regardless of percentage. The time frame for minor changes is expected to be less than an hour of development time, to ensure any and all members working on these changes can prioritize major changes. This set of rules will guide the rest of our development in this project to be as clean and usable as possible.

5.0 FUNCTIONAL TESTING

Another important thing to test for are the specific performance requirements that we initially acquired from our client in the early stages of this project. And so, the performance requirements with the utmost priority that we will test functionality for are as follows: compatibility, optimization, and maintainability.

5.1 Compatibility

Developers will most likely not find interest in a framework that is not compatible with some of the most famously used software. Therefore, we shall ensure that the compatibility of our project satisfies enough products to give developers an incentive to experiment with the framework. And so, to test for compatibility across multiple platforms, we will develop various use cases utilizing our framework, each use case being a different game type. Then, we will simply attempt to deploy each game on various platforms such as a tablet, smartphone, and a laptop.

5.2 Optimization

Our framework's performance needs to be at an optimal level to ensure users have a smooth experience when developing. The components we have created are made to be simple and easy on the system when running. Each component holds the sprite of a simple shape and color to abide by this. To test the performance of the framework in real time, we will implement a level with a large amount of components and objects taking place in a single scene and observe how it handles multiple objects being used at the same time.

5.3 Maintainability

Maintainability highlights how the framework can be updated, modified, and sustained over time. A maintainable system reduces the burden on developers and ensures the longevity of the framework. The test cases for maintainability will simply involve having other developers report back to us how satisfactory it was for them to develop their games utilizing our framework.

6.0 CONCLUSION

In conclusion, our Software Testing Plan is essentially an outline of a comprehensive approach to ensure our framework's functionality and reliability that we plan to take. Throughout this software testing plan, we have detailed our strategies for unit testing, integration testing, and usability testing, specifically designed for the needs of our project. Unit testing is important for verifying the behavior of components within our framework. By testing key components such as the Main Menu, Level Manager, Environment Elements, Knowledge Drops, and Character Components, we can establish a solid foundation for our framework's functionality. In short, you can use the NUnit framework and Unity Test Runner to write scripts that automatically test the functionality of your Unity components. NUnit provides a framework for writing and organizing tests using C#, while Unity Test Runner allows you to run these tests directly within the Unity Editor. Integration testing focuses on the interaction between the components of our framework. By testing integration points such as gameplay loop play-throughs, obstacle interactions, platform functionalities, and knowledge drop interactions, we make sure that our framework functions as intended when all components are integrated. Usability testing is crucial for evaluating the user interface and overall usability of our framework. This user-focused approach ensures that our framework meets the needs and expectations of our target users, leading to a highly reusable and user-friendly software product. Our Software Testing Plan is designed to ensure that our framework meets the highest quality, functionality, and usability standards. By testing our framework through unit testing, integration testing, and usability testing, we are confident that our software product will be error-free, functional, and highly usable, ultimately delivering an enjoyable experience for developers, clients, and end-users.