Joseph Schlegel

Dr. Joules Beeston

CS 383

February 8, 2024

Player Upgrades Champion Document

# Introduction

This document will describe one feature of a game titled *Midnight Slice Madness*. The game will be a horror/thriller game with the core gameplay featuring the player driving from point A to point B, while avoiding several hazards such as enemies and the environment. But the player won’t be helpless against this dangerous world!

My feature for this game is the player upgrade mechanisms. This will include multiple aspects including different types of upgrades, different levels of upgrades, and an upgrade selection menu. This task has multiple different functions that will require both scripting in C# and working with Unity’s GUI . The upgrade feature will be integral to the player’s experience, serving as one of the main motivations of the player. Furthermore, the upgrade screen will serve as the transition period between each delivery and must handle game scene management.

The complexity of the code will depend greatly on the interface between the upgrade system and the player. While this feature will be entirely my own, cooperating and communicating with fellow software engineers on this project is an utmost priority, as I am not building the majority of the player characters movements and interactions with the game environment.

# Case Use Diagram with Scenario

A diagram of a system

Description automatically generated

**Name**: Player Upgrade System

**Summary**: This process will have a very simple use case. The player will be given the option to spend some of their currency to upgrade the player character in a variety of ways. After they have made sufficient selections, they will then play the next level, which extends to the other use cases of the rest of the game like core gameplay and loading the level. The player will also be able to view their balance and previous upgrade selections.

**Actors**: Player

**Preconditions**: End of the previous round, and game is set to continue.

**Basic Sequence:**

**Step 1:** The upgrade menu is displayed.

**Step 2:** The player views their balance.

**Step 3:** The player views their previous upgrades.

**Step 4:** The player selects their upgrades.

**Step 5:** The player continues to the next level/round.

**Exceptions:**

(from step 4):

**Step 1:** The player selects an upgrade.

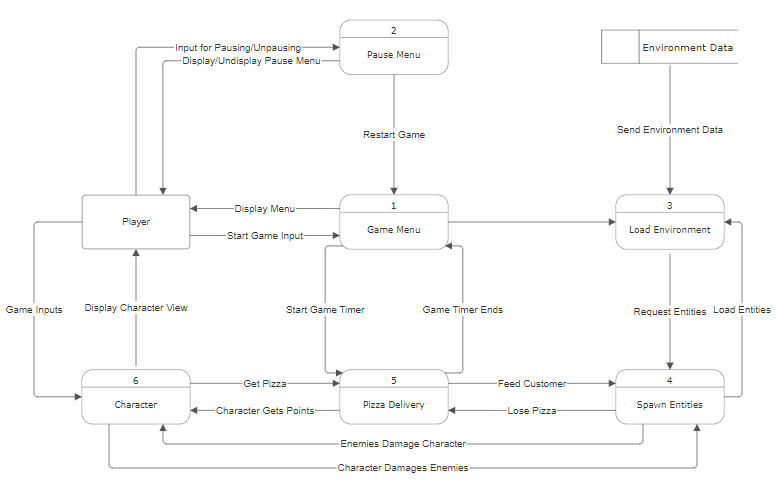
**Step 2:** The balance of the player is reduced appropriately.

**Step** **3**: The player character is modified to reflect their upgraded abilities.

**Step 4:** Return to upgrade menu.

**Post Conditions:** The player is upgraded and ready for the next delivery.

# Data Flow Diagrams

These diagrams will depict the game in its entirety and show a focused data flow diagram for the upgrade menus under the section Game Menu.

A diagram of a game

Description automatically generated

# Acceptance Tests

The functionality of this feature at it’s very base is to transition the player from one round to the next, serving as the review of the players previous actions and allowing the player to make decisions planning for their future. There are some definite elements and some indefinite elements to this application. For example, the player must always exit the upgrade menu eventually, either by potentially quitting the game, or advancing to the next round.

To test this feature, it’s important that the player can move from the upgrade menu, but also that when they do select an upgrade, that it is applied. For this reason, I will describe two acceptance tests.

**Acceptance Test 1:** Player leaving the menu.

This feature will allow the player to end their time choosing upgrades, and push them into the next round, or potentially quit the game. Part of the functionality is difficult to test, including mouse clicks on the screen. However, each button on the screen relates to precisely one function call that then can take control of and manipulate the game environment.

The simple way to test this is to call the function otherwise called by the button click and validate the result. Validation can be achieved by ensuring that there has been a change in the game scene, or that a new game scene has been loaded. Simple Debug.Log() calls can also facilitate this process.

**Acceptance Test 2:** Player Selecting An Upgrade

Once again, the feature requires a mouse-click from the user. However, using the same logic of function calling above, now it’s possible to execute the actual upgrade of the player, which does two things. The upgrade lowers the balance of the player and provides them with an upgrade via equipment or pure statistical increase, such as movement speed.

To validate that the functions to upgrade the player are working properly, it’s only necessary to check that the player’s upgrades were actually applied. So, in this case, while the player is being upgraded, the expected player state is calculated and stored. In other words, the value for each of the players statistics and upgrades is saved during the upgrade process, then once everything is complete, validation is possible by cross-referencing the expected player state and the actual player state.

**Example Output for Upgrade Checker**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Upgrade | Power | Current Level | Expected Level | Final Level | Success? |
| Move Speed | 1.05 | 2 | 2.10 | 2.05 | 0 |
| Light Pow | 1.02 | 0.5 | 0.51 | 0.51 | 1 |
| Battering Ram | N/A | 0 | 1 | 1 | 1 |

This output shows how one might check if variable upgrades such as those to movement speed and Boolean upgrades such as whether a player has purchased an ability have been properly applied.

# Timeline

|  |  |  |
| --- | --- | --- |
| Work Item(P – program) | Duration(Weeks) | Predecessor Task(s) |
| 1. Upgrades Design | 1 |  |
| 2. (P) Game Scene Handling | 2 |  |
| 3. (P) Abstract upgrade classes implementation | 2 | 1 |
| 4. (P) Player stat modification | 2 | 3 |
| 5. (P) User interface | 1 | 2, 4 |
| 6. Testing | 2 | 5 |
| 7. Finalization/Ship | 1 | 6 |