LooterLand

Technical Design Document

Index

Summary

1.1. Technical Game Description

LooterLand is a game where a player moves around a level and picks up items while being chased by store employees. The objective of the game is to get as many points (money) as possible before the game timer runs out. The camera overviews the game and will always be rotated facing north of the map and above the Player so that the game is in a third-person view.

1.2. Feel of Gameplay

The feel of the game is supposed to arcade-like, having a low-poly graphics style that games during the ps1 era of games looked. The style of gameplay should be quick and and chaotic.

1.3. Game Systems Simplicity

The game mechanics and systems are made to feel simplistic, and not overly diverse so that the Player can focus on the main objective, and not be distracted or overwhelmed by other implemented systems.

Technical Goals

2.1. Inventory

The player will have an infinite inventory where items are converted into cash upon pickup. If the player is hit by a cop, they will drop generic grocery bag items that store a portion of the wealth the player had collected.

2.2. Highscore system

A total amount of cash accumulated will be calculated as the game is playing. Once the end state is reached, this value will be displayed to the Player so that they can compare this “run” with other runs they or other Players had played. This will be done by adding the value of the item to the total count during gameplay.

2.3. Responsive Input

Using input keys to perform actions and begin animations should feel quick and snappy. Smoothness can be sacrificed to give the game a videogame-like feel and allow the Player to perform actions instantly.

Technical Risks

3.1. Random Item Spawning

To make the game more fun, and to add replayability to the game, items will be randomly placed in random locations in the store. The “electronics section” (the top right corner of the map) has a higher chance of spawning rare and uncommon items.

3.2. Animations

To make the game feel alive, the Player will have animations for moving around the world, cops will also have their own animations to indicate states that the cop is in.

3.3. Hazards

Throughout the store, there are spills of various substances that apply different effects upon the Player and Cops. There are water puddles, spilled Wizard Speed cans, and sticky floor spots.

Code Style

4.1. Naming Rules

Functions, scripts, and scriptable objects should be named with PascalCase. Scriptable objects also should have SO immediately after the name of the object.

4.2. Variables

Public variables should be named with PascalCase. Private variables should have the prefix “_” followed with the name of the variable in camelCase.

4.3. Conditions

Conditionals will be formatted preferably as shown below:

```
if (condition) {}
```

Rather than

```
if (condition == true) {}
```

The use of guard clauses rather than large nested blocks within if/else statements will also be preferred use. Example of this is as follows:

```
if (!condition) return;
```

Rather than

```
if (condition) { // Large block of code // }
```

Prefer long conditionals to be on multiple lines:

```
if (condition1 ||
    condition2 ||
    condition3 ||
etc) { }
```

4.4. Classes and Structs

Prefer structs and Scriptable Objects to be used more for data storage and classes for functionality.

Prefer classes to always have a default, zero argument constructor (or constructor where all arguments have default values) in addition to any others such that all classes can be instantiated without passing arguments unless strictly necessary.

4.5. Scripts and ScriptableObjects

The title of Scripts/Behaviors should briefly describe what the Script does. Prefer for the title of the Script to not include “Script” or “Behavior” after the title of the name.

Example: Movement : MonoBehaviour

All ScriptableObjects scripts will end with SO also using PascalCase.

Example: PlayerDataSO:

Commit Policy

5.1. Commit Naming Conventions

Any and all commit messages should be descriptive as possible. Commit messages should begin with a prefix, followed by a description describing the purpose of the commit with details of each change made.

Prefix Examples:

- fix: Player movement script now stays at the last rotation angle.
- feat: Added slot selection to inventory system.
- chore: Forgot to save updated script for current scene file.
- docs: Added Documentation entry for CameraMovementBehavior.

5.2. Branching and Merging

Each contributor to the source code and project must work and commit into their own branch that branches off from main/dev. A single contributor should **never** make the decision to push into main, once changes have been reviewed to the team, push to dev. Once a contributor has

completed working-changes that they are confident in to their branch, they must show it to the team so that everyone can review it and agree to push it to dev. Once reviewed changes from all contributors are gathered in dev, all of the changes should be tested and conflicts/bugs should be resolved there in a separate branch. Once testing and bug fixing has been completed and the new version of the project has been reviewed, then the team should do a pull request to put the changes on the main branch.