Bruno Gomes, Blue Gravity Studios Technical Task Report

PlayerController system

The PlayerController script takes care of basic player movement and collisions. It uses a simple Rigidbody propulsion system to move the player and BoxCollider2D for collision with environment and interaction detection. It includes functions to disable player control without disabling the component entirely, for example, for when given UI elements are enabled.

Player animations are handled by a separate script PlayerAnimsController. The imported character uses the AnimatorController system with provided animation clips so all the script has to do is check for movement and tell the Animator to set parameters.

Notes: Flipping the character was not provided as an animation clip as these usually are done in code. I'm personally unsatisfied with the solution used (flipping transform in the Y axis) as it tends to cause problems in the long run but to the extent of my knowledge it was easier to do so as the character imported is composed of separately animated sprites in various objects. Flipping sprites individually would've been time consuming. If there is a more effective solution I am unaware.

CameraController system

Simple camera controller script to make the camera follow the player. Originally intended to have the camera lock in position on collision with level borders in style with reference games, but scrapped the idea for time. A collision check of the player with a border built into the tilemap likely would've achieved this effect.

Notes: camera offsets from player by 2 units up due to built in animations from imported character, which likely was meant for a more sidescroller type game so automatically bumps the character up.

Interaction system

The interaction system consists of a main Interaction script and a case-by-case script for each interactable object. It works with UnityEvents by having both the main script and the unique script attached to the interactable object where the Interact script handles detection with a trigger collider and invokes events which can be defined in the Inspector as the various methods from the unique script. It can handle player entering and exiting the trigger area as well as what happens when the interact button (hardcoded to 'E') is pressed while the player is within range.

Note: Originally was going to have the interactions be handled by the PlayerController script, checking to see if there are any interactable objects inside a trigger around the player and **which is closest**, but quickly realized this was redundant for the scale of the task. Would be a better solution however on a larger scale where interaction prompts can overlap and we might want to avoid messy solutions to preventing multiple interactions.

Tilemap

The scenery is a static image overlayed by several tilemaps which compose the solid objects, interactables and a level border separately. The solid objects tilemap simply uses a base tilemap collider that automatically draws collision meshes to the painted objects, the interactables tilemap does the same but with trigger colliders of invisible objects painted over the objects intended to be interacted with and the border tilemap is a supporting tilemap to the static background to prevent the player from walking over walls and out of bounds by applying invisible collision tiles.

Note: The tilemap assets I chose to import only included a full image of the background, originally I planned to use the sprite editor to slice up the image and use get the various walls and ground tiles and build a custom room. However the image had odd sizings and trying to neatly slice the image caused tearing in the textures. Upon experimenting with using the full image as a static sprite in the background it lined up perfectly with cell 1 size tilemap objects placed over it, so I just rolled with it and used the supporting border tilemap to add wall collision.

Additionally the interactables tilemap was drawn and thought out before I had decided to change how I wanted to approach interactables as a system. In hindsight this is redundant as it likely can only handle the interactions of a single object or an overall level based interaction. Placing interactable level elements individually would be more beneficial in the long run.

UI, UI Controller and Inventory Controller

The UI Controller is responsible for managing the various UI panels and windows, ensuring that no 2 menus can be open at once, and that each button leads to its respective UI menu. The Inventory Controller stores various information about the player such as current currency owned, items owned and outfits owned and listens for when UIController opens an inventory to load the items that must show there. The UI itself is composed of various screens/panels that are all assigned to variables on respective scripts that call them or directly in the UIController script. It uses layering of empty panels or objects on a few areas in order to include LayoutGroups neatly without disrupting the sizing of elements too much.

Note: The inventory controller loading method for items is sluggish and rushed, it shouldn't need to constantly destroy and load items into the inventory panel everytime it is opened. Same thing goes for the Outfit Selection panel. It's a functioning solution but not an ideal one. The inventory is also limited by the space available in the UI element created for it, only up to 9 slots. In hindsight this was easily rectifiable with a Scroll Rect component to allow more item space. The UI composition and the main Controller system for it is a simplified version of the UI system I've come to refine over various projects. Managing UI is one of my major difficulties but this system makes it simple although a bit cumbersome to understand and reference where necessary.

Outfit Identifier and Selector

The Outfit identifier is a small script consisting of a method and an integer value attached to a button prefab in order to create the buttons that have to send a reference of which outfit to load when pressed. The Selector is called from the Identifier to load the referenced outfit.

Note: This solution was done a bit rushed to avoid working through UnityEvent assigning. It's an unnecessary work around that I did because I was running out of time and wasn't able to make UnityEvents work. The whole system is also inconsistent, the solution to load the outfits isn't ideal and probably should've included reloading the scripts to ensure the game wouldn't get stuck in a scenario where it doesn't recognize the new player object.

Shop Controllers

A pair of 2 controller scripts that show item and outfit information on a sidebar when they're clicked in the shop and processes purchases and sales from the shop and adds them to the appropriate data in InventoryController.

Note: While I think the scripts are well designed, the whole system of the shop itself is poorly thought out and rushed as well. The shop isn't dynamic as the buttons for the available items are hardcoded with their values and events. A more dynamic solution with buttons generated based on a stock of items (itemsAvailable and outfitsAvailable variables) was a bit too complex for me to make work.

ScriptableObject Items

The items and outfits are set as scriptable objects to contain the data of the items and easily move them around inventories. Both scripts contain basic data such as name, description, price and a preview image / icon, that get displayed on the shop screen, only the outfits include prefabs to load using OutfitSelector and only the items contain a sell value as the outfits weren't considered to be sellable.

Note: Scriptable Objects are the main way I know how to code items in inventories. They can sometimes be a little limiting, but I think in a task of this magnitude they worked perfectly. I struggled a bit with whether to include item and outfit IDs but ended up only including them with items to make searching for items in the player inventory that can be sold easier.

Final Thoughts

The project started out smoothly but about halfway through a costly mistake on my part with Source Control forced me to start almost completely over with a new repository. It was an honest and most importantly **avoidable** mistake, which is why I decided against requesting extra time to do the assignment. I believe managing source control is a responsibility I'm being evaluated for with this task and a mistake in it is one that should have fair weight in my result. That mistake coupled with my fairly slow pace of development caused rushing the final parts and a lesser overall project than what I wished to deliver. I was only able to playtest the build once before delivering as I'm already pretty late on the deadline. Everything seemed functional but also very unstable so it's somewhat likely it will break during more thorough testing, but I hope the code itself proves of quality at the very least.