Item Purchase

The `BuyableItem` class inherits from `ItemSO` and implements the `IItemAction` interface, enabling items to be purchased. The `PerformAction` method verifies if the player has enough money to buy the item, deducts the cost of it from their inventory while adding to theirs and removes it from the shops inventory. This method ensures that purchase transactions are verified and processed correctly.

Item Equipping

The `EquippableItem` class also inherits from `ItemSO` and implements `IItemAction`, but this time it's for items that can be equipped. The `PerformAction` method equips the item to the player, updating animations and triggering relevant events through the `EquipmentEvents` class. The event system (`EquipmentEvents`) manages equip and unequip events, allowing other game components to respond appropriately to these actions.

Inventory Management

The `InventorySO` class is a `ScriptableObject` holding a list of items in the inventory. It provides methods for adding, removing items, and offers event-style callbacks to other systems regarding changes in state regarding inventory. This class offers an enhancement foundation that bases the management of the player's inventory, ensuring item additions or removals treated accordingly.

Player Movement and Interaction

The `PlayerInput` script implements the `IMovementInput` interface and manages player input for character and interaction movements. It detects whether the player presses specific keys to move the character or interact with the environment.

The `PlayerMovement` script takes inputs in the form of movement, applying 2D physics to alter the position of the player; the `PlayerRenderer` updates the direction of the player's sprite depending on the movement direction.

Player Animations

The `PlayerAnimations` script coordinates the movement and equipment animations of the player. This script runs coroutines to loop between animation frames while the player is moving. These coroutines are stopped off when the player stops moving. In addition, they update equipment sprites whenever an item has been equipped or unequipped.

NPC Interaction

The script `PlayerAIInteractions` allows the player to interact with NPCs in a game. It casts something called a ray from a specific player point, allowing the detection of nearby NPCs for context interactions like opening a shop to buy items.

Scene Management

The script `GameManager` handles scene loading. When the application starts up, it initializes the scene of the main game.

How to play

Move using "WASD" or Keyboard Arrows.

Open inventory pressing "I".

Interact with the NPC looking at him and pressing "E".

Left Mouse click on inventory item show its image and description.

Right Mouse click on inventory item show its actions.

Thought process during the interview

My mindset was on how best I could structure my code properly for Blue Gravity, where quality was going to be paramount. Making sure that the dependencies of the script stayed as slim as possible and following all the best practices concerning programming was a consistent effort. I strived to incorporate design patterns and principles of SOLID wherever applicable in design.

I began with the core functionality such as movement, animation, and interaction. This provided a firm base of operation while keeping it simple. Character inventory management followed, and I really struggled to do something effective and flexible that could easily be reused between the player and shop inventories. This really helped the development not only in process but also saved a lot of time.

On the ground of this laid platform, I went further to create equipable items for the character and implemented them well, with buying functions at the shop. This holistic sketch ensured both functionality and efficiency in the general development.

Assessment of my performance

Well, this code did not turn out to be that structured from my side as I expected; hence, I have fallen back on some of the older code and then picked up with others to better make it just in time for my deadline. Being a junior programmer, I still find it rather challenging to structure a project just by myself from scratch. That means it's not that optimized as it would have been if there was more time. Also, there are minor bugs in the project; part I didn't have time to address, which is outlined in the next section. Finally, I didn't have an opportunity to adjust the UI according to the style of the game.

Bugs and more

The biggest challenge I encountered was the lack of commits I hadn't done on GitHub due to internet connectivity issues, at the end, the last commit had most of the codes that were completed.

One major problem I had was obtaining the index slot for each item in the inventory. This created a bug where the player had multiple instances of the same type of item in their inventory. Due to the saving of the index of the item and not the slot, it caused an error where when the player dropped an equipped item, it would not remove said item from the character's attire properly. In the shop, purchasing one of the same kind while on the last inventory would remove items in the earlier slots rather than the intended slot.

Also, I didn't have time to implement the player's money UI (he starts with $100).

Also, I didn't have time to put in better interaction between inventories. For example, it's possible to open the player and shop inventory simultaneously. Also, the player can walk around with the inventory open, which is bad for usability.

Another feature that isn't finished is the proper selling system of the shop. That means that currently there is no way to sell items back to the shop once it's bought.

There are comments throughout the code, nearly every other line, demarcating sections using //begin pre-written code and //end pre-written code, marking where a predefined code section or sometimes just a placeholder is located and would need further implementation/integration.