Simple shop 2d documentation

Author: Bruno Locha Gomes Oliveira

This document serves as a guide on the creative process of the SimpleShop2D code and general structure. The used architecture uses object oriented programming patterns and concepts, like SOLID, clean code and DRY. My main design reference was Pokemon (GBC / GBA).

1- Core

First, using the features described as required, I created a simple state machine to control the game flow and prevent inappropriate behaviors from the Player interactions. Using this approach, the base structure gains flexibility and is easier to create new general states, like pause.

The GameController class was created, using the Singleton pattern, to manage general state machine transitions and serve as a common contact point to all entities that wish to create events that need special state handling. For the current game state information distribution, instead of using the game controller as a bridge, I created a component using the Observer pattern to handle the new information receive and distribution. This component registers itself on the state machine and passes the new information to the appropriate listeners using custom UnityEvents, and to handle these events, i created an generic interface to ensure that the target class has the appropriate tools.

At least, the core needed something to handle the player inventory information, such as current items, equipped ones, buy and sell operations, etc. For that purpose, the DataController class was created. Further this class should also execute save and load operations to preserve the current player data even if the application is closed. It's also desirable that this class was unique and preserved the information correctly, so I used a static behavior to make sure the object containing this doesn't get destroyed in a scene load operation.

2- Characters

The actors in this game were separated into two common kinds: NPC and Player. To handle the common characteristics, like collision ability and basic animation, an abstraction was used to create the Character class. Using this, it’s easier to create new types of characters, like the shop owner, with similar behavior. I also wanted to give the NPC a discrete movement system to give more life to this prototype, but I didn't get the time to work on this feature.

3- Dialog

As I have exposed in this document, dialog pop in and out it’s handled by the GameController, but the system used to trigger this behavior it’s an tricky one. First, the dialog structure was created using an ScriptableObject base. This allows not only abstraction, but also the game design and programmers to create new dialogs and interactions without much effort, giving the project mobility and speed. Next, the DialogableObject component handles the trigger, passing dialogs to the game controller as such their callbacks. This pattern allowed shop owner dialogs to use the same system as npc’s, but with a different result. Also, fun iterations like talk to rocks are also possible, making the dialog system not dependable to characters. The original idea was to create an question option in this system, but I also lacked the time.

4- items

The item management system it’s an simple hierarchy flow using Inventory -> InventoryItem -> Item. The item was also created using an scriptable object approach so their design can receive changes without code operations. The shop and the player inventory also uses this flow, making it possible to create similar components to manage their UI representation. Like the dialog system, the shop and inventory windows pop in and out it's also handled by the GameController class.

5- Improvements

- Sound system
- Pooling system to avoid instantiate and destroy procedures
- Question options on dialog system
- Pause state
- Npc movement system