**Unity Final Project**

**Game Overview:**

The game is a top-down 2d survival game, the goal is to survive waves of enemies for an allotted amount of time and by the end of that time the player will face off against a boss. If the player defeats the boss and survives the waves of enemies then he will move on to the next level.

To help the player survive he is given an assortment of weapons, powers, and potion pickups.

**Instructions:**

“wasd” or arrow keys for movement. Shooting happens automatically with weapons.

1234 or “qerf” for special abilities, for lighting ability use mouse to aim.

“T” to open inventory.

Press

**Player:**

The player game object has scripts that handle movement, shooting, powers, data, health.

One of the scripts attached to the player is a data script which holds all the info of the player’s scriptable object (all characters have scriptable objects that hold their health, name, speed)

Instead of passing it to every script that might need the SO we have a script that is the data script to hold that data, which makes things more organized if you know where to look.

The player has super power shooter script, this script allows the player to trigger 4 distinct powers. There is a UI element on screen that displays the 4 powers, when the player uses one of the we see the fill of the image that corresponds with that power change depending on the cooldown time.

This is done in the following way:

First when the player uses the power we trigger an event ***m\_PowerUseEvent?.Raise(i\_PowerIndex);***

**M\_PowerUseEvent** is a scriptable object that acts like a channel.

***Raise(i\_PowerIndex)*** *is a function within the* ***m\_PowerUseEvent*** *scriptable object that triggers the event:* ***OnPowerUsed?.Invoke(powerIndex);***

So in essence ***m\_PowerUseEvent?.Raise(i\_PowerIndex);*** triggers the event***.***

In another script that handles the power UI we subscribe to the event:

**m\_PowerUseEvent.OnPowerUsed += HandlePowerUse;**

**note:** we use this kind of event system for many things including the game state machine that controls sound, so we won’t go into it with as much detail as we did now. What we described now is the same principal as what we used in the other places so it would be redundant.

Its always some kind of channel system set up, with listeners on one end and triggers on the other end, while we have a scriptable object that holds the event and acts like a channel, that whichever script has access to the SO channel will have the ability to listen(subscribe) to the event and also to trigger events

so, when the player uses the power in another script and triggers the event the UI controller will be alerted and will trigger the function to handle the UI (**HandlePowerUse**).

During cooldown time the powers image will “load” visually, the way this is done is by “filling the image from 0 to 1 in a continuous way by combining a timer between frames

(**timer += Time.deltaTime;)** and using a co routine so that the while loop won’t fill the entire image in 1 frame, with each time within the fram that passes we fill the image depending on the amount of time that passed, this way the image looks to be filling continuously.

**Enemies:**

There are different kinds of enemies, including bosses which are also considered enemies, much like the player they have health, movement, shooting, data, handlers, they also use the same character SO which the player uses.

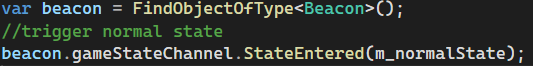
The difference between each enemy and bosses besides trivial health and speed, is movement scripts and attack scripts, regular enemies have a handle movement by “following the player” with every frame check, the velocity is constantly updated depending on the players location. We have a boss that has a spiral shooting ability, this ability works by shooting bullets in a circle with a slight delay between each one, and each bullet one instantiated is given a static starting velocity with a certain rotation, and this velocity remains throughout the bullet’s lifetime. We managed to get this effect by dividing 360 by the number of bullets and then looping from 0 to num of bullets this way we get every multiple withing the 360/num of bullets range, for example of we have 20 bullets so 360 / 20 is 8, so the first bullet has an angle of 0 and then 8 and then 16 until we reach 360, for each angle we instantiate a bullets and set its rotation to that angle by using **Quaternion.Euler(0, 0, angle) \* Vector3.up.** (this is a common and easy way to set the rotation of an object in unity instead of trying to do sin and cos calculations). That rotation will be the direction we want the bullet to shoot in so we set the bullets velocity to the above direction \* speed. Notice the player has a gun that does something identical only without the delay between bullets, this is his shotgunshooter, also the enemy has access to shotgun. This is one example of re using cool features to create other cool features easily that we did in this project.

This boss also has a unique movement, he is instantiated within a frame, and he will move in some direction, when he hits the frame wall, he will change directions and go opposite, this way he will go back and forth within the frame.

**Game state machine:**

We have a game state machine that handles in game state and pause state, this was copied from Luna and I haven’t don’t too much with it. But I took a similar concept and implemented our own state machine that handles music in the game. During the gameplay there are two states, one is when there are regular enemy waves attacking the player and another when a boss is spawned and attacks the player, we can call these, normal state and nightmare state, when the game begins, we enter normal state and normal music begins to play, and when the boss is spawned we enter nightmare state and nightmarish music begins to play.

This works in the following way:

When the game begins we trigger a function 

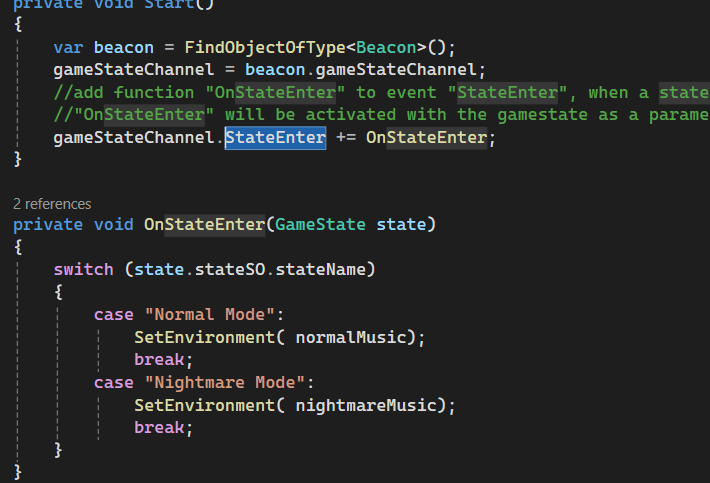
When the function is triggered it will invoke an event

***A computer screen shot of a black background

Description automatically generated***

The gameState parameter is the gameState that Luna defined, the only thing we really need for our purposes is the gameState name, so we added to the GameState SO a gameStateName field.

So, the StateEntered was triggered with the game state as a parameter which means that any function who is subscribed to that event will be triggered and will get the gameState as a parameter. We have a gameStateListener object in our scene that has a script with a function that is subscribed to the StateEnter event. And when this function gets triggered, it will check the name of the gameState which it received as a parameter and set the music depending on the gameState



**Scenes:**

we have a main menu, lvl 1/2/3, player death, end game scenes.

The main menu has two buttons, a start button and an instructions button, when you click start, we will load the level 1 scene, if we click instructions button then a instructions UI will pop up.

The player death scene loads up when the player dies, and it has a button that when clicked gives the player the option to play again when clicked also in the endgame scene that loads up after the player defeats the lvl 3 boss the player has the option to click a button to play again.

**Inventory**

The player has an option to open the inventory and use the collected items from the levels. The inventory system uses an InventoryManager script that acts as a singletone for transferring collected items to the next levels.

**Manager**

InstantiateManagers script is tied down to a gameObject prefab that is in every scene and its goal is to Instantiate the InventoryManager, the InventoryManager has a dictionary for the collected items, it has a add item function that checks if their is an item similar already, if yes, it checks if the number is higher the max. In the end it calls for an event that signals an item has been added.

likewise for removing an item it finds the item and decreases its count and checks if the number is zero, in the end sends an event item has been removed (or the count decreases), If the game is over and the player chooses to retry then resetInventory gets called and clears the dictionary.

**Item Scriptable Objects**

All the items are using Scriptable Objects that stores the its data, like its name, max stack, and enum of its type, etc.

**Inventory UI**

The ui element of the inventory system is managed by the InventoryUI script that is in each panel of each scene, its goal is to represent and access the inventory for the player, it displays each collected item with its icon and count, and when items are added or removed, event listeners in InventoryUI respond by refreshing the display showing the player the current inventory.

***Functions***

When an item is added to the inventory, InventoryUI uses HandleItemAdded to check if the item already has a UI slot, If yes, it updates the item quantity directly, if not, AddNewItemUI creates a new UI slot for it, complete with the item icon, quantity, and clickable functionality for item interaction, If an item is removed or used up, HandleItemRemoved is decreasing the quantity or removing the UI slot if the count reaches zero.

***On Item Clicked***

The OnItemClicked function give the players the ability to use the items in the inventory, when the player clicks on an apple which is a health item it will heal him, cherries will give speed boost banna will shrink the player, weapons will get equipped, the item count is checked and changed by UpdateItemUI, which displays the correct quantity or removes the item if the count is zero.

**Item Pickup**

The ItemPickup script is attached to all of the collectible items, its goal is to triggers the AddItem function in InventoryManager when the player collides with an item, there also is an animation for when the item is collected and then destroys the item on the map.

**Animation And Sprites**

There are animations for items, the player and enimeies, for the items there is an idle animation and a collected when the item collides with the player (except the weapons) the initial and constant state is idle.

**Player**

The player has two animations, idle and running, the running animation is triggered when the player moves otherwise the idle state is constant, in addition the player sprite will change depending on the player move direction and will shrink in size if used a shrink item (banana).

**Enemies**

All the enemies have a running animation and death animation (except boss 2 that has death animation and idle), the constant state is running, and the death is triggered when enemy dies.

**Explosion Effect**

Triggers when an enemy is hit in the EnemyHitByBulletHandler script for each enem by Instantiate, using a particle effect it looks like an explosion.

**Sprites**

Most of the sprites are from the asset store some are hand made like the lazer beam, the shock wave, clock, health bar and cannon, when the player or enemy is hit the is a specific flash color change.

**Maps**

The level maps are using a tilemap with sorting layers, there are total of 6 sorting layers, the default being the player one, the one below are ground, walls and shadows, and all the above are roof tops and accessories for the map.

***Level Design***

The levels have items and weapons scattered around the map to collect and use, ponds that heal the player and obstacles that can stop the player.