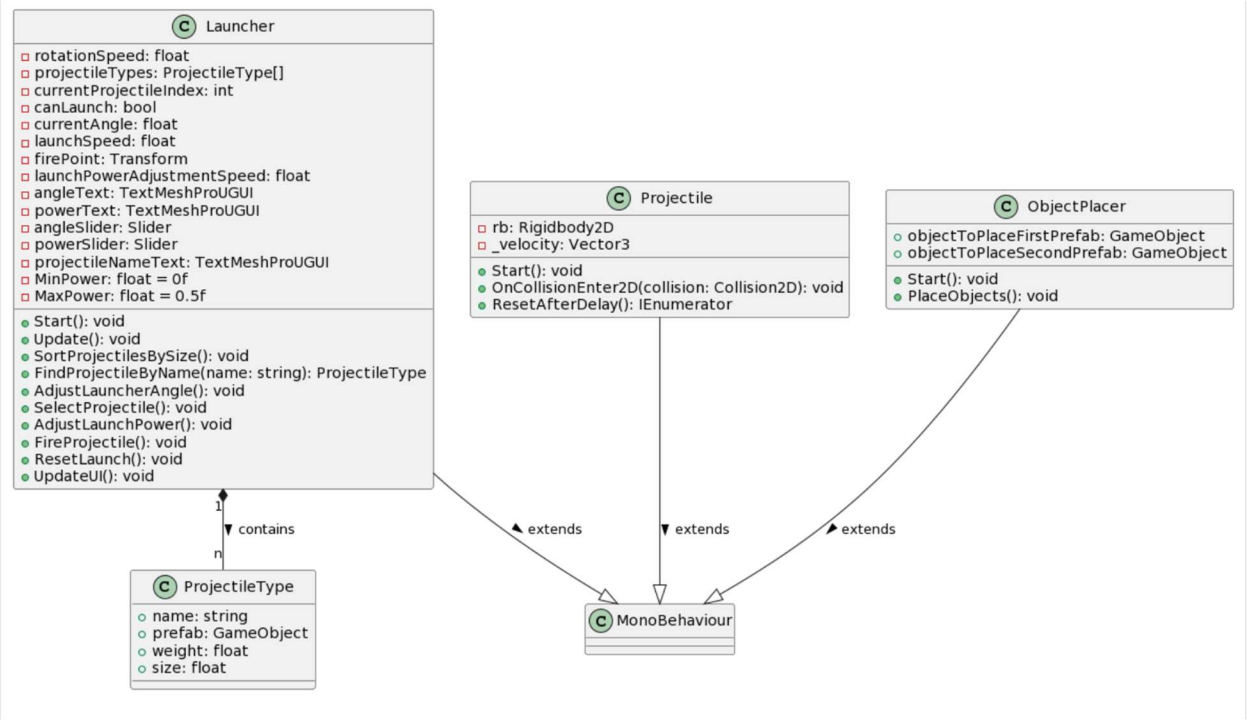
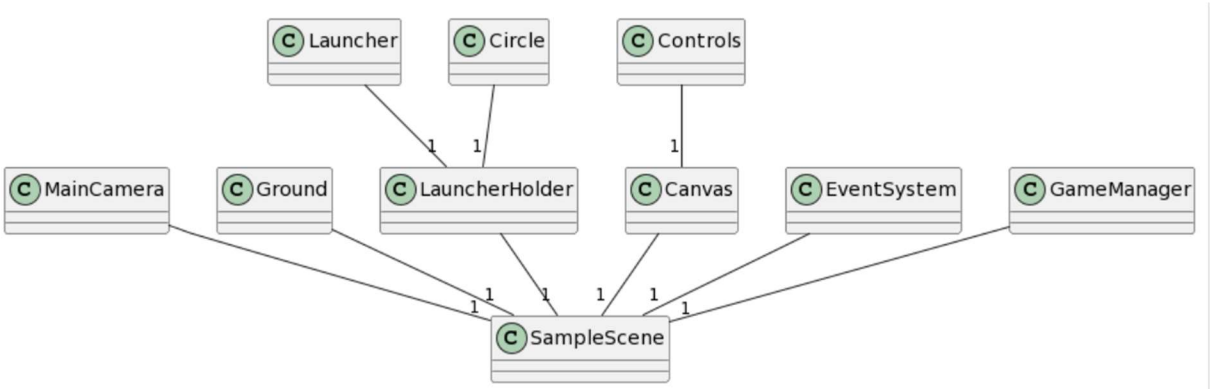
Code Structure



GameObject Structure



Compl

Sorting Algorithm for Projectiles:

```
    // Bubble Sort algorithm to sort projectiles by size
1 reference
void SortProjectilesBySize()
{
    int n = projectileTypes.Length;
    for (int i = 0; i < n - 1; i++) // Outer loop
    {
        for (int j = 0; j < n - i - 1; j++) // Inner loop
        {
            if (projectileTypes[j].size > projectileTypes[j + 1].size)
            {
                // Swap the elements
                ProjectileType temp = projectileTypes[j];
                projectileTypes[j] = projectileTypes[j + 1];
                projectileTypes[j + 1] = temp;
            }
        }
    }
}
```

Implements a Bubble Sort algorithm to order the projectileTypes array by the size of the projectiles. It iteratively compares adjacent elements (projectiles), swapping them if they are in the wrong order (if the preceding projectile is larger than the following one), thus ensuring the array is sorted by size. This method is critical for gameplay elements that rely on presenting the player with sorted options, affecting decision-making processes.

```
if (objectToPlaceSecondPrefab != null)
{
    // Ensure the second object is placed to the left of the first object by 2 to 5 units
    float randomXSecondOffset = Random.Range(2f, 5f);
    float secondObjectX = 0f;
    if (firstObjectInstance != null)
    {
        float firstObjectX = firstObjectInstance.transform.position.x;
        secondObjectX = Mathf.Clamp(firstObjectX - randomXSecondOffset, 0f, 8.5f); // Clamp to ensure it remains within bounds
    }
    else
    {
        // If the first object isn't instantiated, just place the second object randomly within the specified range
        secondObjectX = Random.Range(0f, 8.5f - randomXSecondOffset); // Adjust range to account for offset
    }

    // Instantiate and place the second object at y = 5 with the calculated x position
    secondObjectInstance = Instantiate(objectToPlaceSecondPrefab, new Vector3(secondObjectX, 5f, 0f), Quaternion.identity);

    // Randomly adjust the second object's scale in the y direction between 0.5 and 4
    float randomYScale = Random.Range(0.5f, 4f);
    Vector3 currentScale = secondObjectInstance.transform.localScale;
    secondObjectInstance.transform.localScale = new Vector3(currentScale.x, randomYScale, currentScale.z);
}
```

The code involves procedural logic to place a second game object within the game scene relative to the first object's position. This logic ensures variability in the game's level design and challenges.

Checking for Second Prefab's Existence: The process begins with a check to ensure the `objectToPlaceSecondPrefab` is defined (not `null`). This step is crucial to avoid attempting to instantiate an undefined object, which would lead to errors.

Determining X Position for Second Object: A random offset between 2 and 5 units is generated. This offset will determine how far to the left the second object should be placed relative to the first object. The presence of the first object influences this placement:

- If the first object exists, the script calculates the second object's X position based on the first object's X position minus the random offset. The use of `Mathf.Clamp` ensures this X position stays within the playable area's bounds (0 to 8.5 units), preventing the object from being placed out of reach or outside the game scene.

- If the first object is not present, indicating that the second object might be the first to be placed in the scene, it assigns a random X position within an adjusted range to maintain spatial integrity and gameplay balance.

Instantiating the Second Object: The second object is then instantiated at the determined X position, with a fixed Y position (`5f`), ensuring it appears above the ground or base level of the game scene. This specific Y value might be part of the level design, placing objects at different heights for visual or gameplay variety.

Adjusting Object Scale: Finally, the script randomly adjusts the second object's scale along the Y axis (height) between 0.5 and 4 units. This random scaling introduces additional variety to the obstacles or items within the game, affecting how players interact with them. For instance, a taller object might require different strategies to navigate compared to a shorter one.

```
1 reference
void FireProjectile()
{
    if (canLaunch)
    {
        GameObject selectedPrefab = projectileTypes[currentProjectileIndex].prefab;
        GameObject projectileInstance = Instantiate(selectedPrefab, firePoint.position, firePoint.rotation);
        Vector3 initialVelocity = transform.right * launchSpeed * projectileTypes[currentProjectileIndex].weight
        projectileInstance.GetComponent<Transform>().localScale *= projectileTypes[currentProjectileIndex].size;

        Projectile projectileController = projectileInstance.GetComponent<Projectile>();
        if (projectileController != null)
        {
            projectileController._velocity = initialVelocity; // Assuming Launch() is a method handling velocity
        }

        canLaunch = false; // Prevent further launches
        Invoke("ResetLaunch", 2); // Reset launch capability after 2 seconds
    }
}
```

Conditional Launch Check: The method begins with a check (if (canLaunch)) to determine if a projectile can be launched at this time. This typically prevents the player from launching another projectile if one is already in motion or if a certain game condition hasn't been met.

Prefab Selection: A specific prefab, which is a template for creating objects in Unity, is selected from an array of potential projectile types (projectileTypes[currentProjectileIndex].prefab). The current selection is determined by the currentProjectileIndex, which changes based on player input or game logic.

Projectile Instantiation: Using Unity's Instantiate method, a new projectile object is created in the game world at the designated launch position (firePoint.position) and orientation (firePoint.rotation). This is where the projectile physically appears and starts its motion in the scene.

Initial Velocity Calculation: The code then computes an initial velocity vector (initialVelocity) for the projectile. It combines the launcher's facing direction (transform.right), the predefined launch speed (launchSpeed), and the projectile's weight (projectileTypes[currentProjectileIndex].weight). The weight factor allows different projectiles to behave distinctively when fired, simulating realistic physics where heavier objects may move differently from lighter ones.

Scaling the Projectile: The size of the instantiated projectile is adjusted by scaling its transform according to the size value specified for the current projectile type (projectileTypes[currentProjectileIndex].size). This affects how the projectile looks and interacts within the game world.

Assigning Motion to the Projectile: After the projectile is instantiated and its properties are set, the method looks for a Projectile script attached to the new object. Assuming such a script is found, it assigns the initial velocity to a variable within that script (projectileController._velocity). This typically controls how the projectile will move after being launched.

Launch Control: The canLaunch flag is set to false, which acts as a control mechanism to prevent further launches until certain conditions are met or a set time has passed.

Reset Mechanism: Finally, a delayed call (Invoke("ResetLaunch", 2)) is set up to re-enable the launcher after 2 seconds, using the ResetLaunch method. This gives a pause between launches, simulating reloading or cooldown periods commonly seen in games to enhance gameplay balance and realism.

More Complex Code Examples

Arrays

Used In: Launcher script to hold the ProjectileType objects.

Explanation: The array projectileTypes is used to store different types of projectiles which the player can select and launch. It represents a collection of user-defined objects, each representing a potential projectile in the game.

User Defined Objects:

Used In: ProjectileType class within the Launcher script.

Explanation: The ProjectileType class is a user-defined object that holds the properties of each projectile type, such as name, prefab, weight, and size. Instances of this class are used to store data records that the launcher needs to function.

Objects as Data Records:

Used In: ProjectileType as part of the Launcher script.

Explanation: The ProjectileType instances serve as data records, each holding information about a projectile. This encapsulates related data in a structured way, akin to a record in a database.

Simple Selection:

Used In: Checking the canLaunch boolean within the FireProjectile method.

Explanation: The simple selection is performed using an if statement to check the condition before launching a projectile. It controls the flow of the game based on whether the launcher is ready to fire again.

Complex Selection:

Used In: SelectProjectile method within the Launcher script to select the current projectile.

Explanation: The complex selection is used when changing the currentProjectileIndex based on user input, employing modular arithmetic to cycle through the array of projectile types.

Loops:

Used In: The Update method within the Launcher script.

Explanation: The Update method itself is called once per frame, creating a game loop that constantly checks for player input and updates the game state accordingly.

Nested Loops:

Used In: SortProjectilesBySize method within the Launcher script.

Explanation: The method employs nested loops to perform a bubble sort on the array of projectile types, ordering them by size. The outer loop runs through all elements, and the inner loop performs the pairwise comparisons and swaps.

User Defined Methods with Parameters:

Used In: FindProjectileByName method in the Launcher script.

Explanation: This method accepts a string parameter (the name of the projectile) and searches the projectileTypes array for a match, demonstrating a method with input parameters.

User Defined Methods with Appropriate Return Values:

Used In: FindProjectileByName method in the Launcher script.

Explanation: It returns a ProjectileType object if a matching name is found or null otherwise, showcasing a method with a clearly defined return type based on the outcome of the search.

Use of Additional Libraries:

Used In: Throughout all scripts, with imports such as UnityEngine and TMPro.

Explanation: The scripts utilize Unity's libraries for game development functionalities like object instantiation, physics calculations, UI updates, and more. TMPro is used for advanced text rendering in the UI.

Searching:

Used In: The ResetAfterDelay coroutine within the Projectile script.

Explanation: The SceneManager.LoadScene method involves searching for and loading a scene by name, which may imply searching through Unity's build settings to find the corresponding scene file.