Duck Hunt Game Algorithm Document

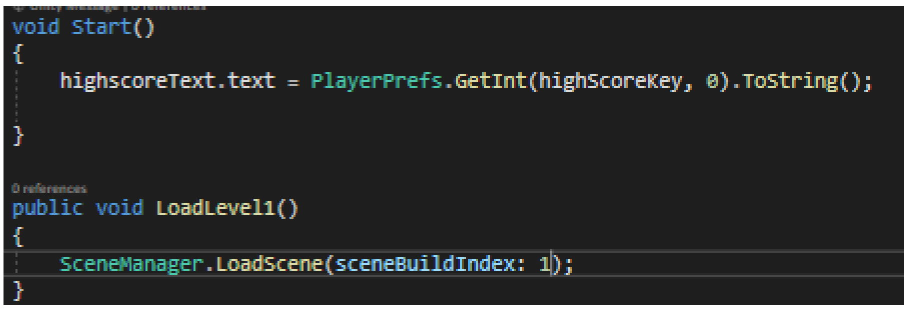
By Javon Nunes

Synopsis:

This document tries to illustrate the design patterns that were used, along with the reasons why and evaluate any other improvements that could have been made to the code.

Context:

1. Command / Chain of Command



```
void Start()
{
    highscoreText.text = PlayerPrefs.GetInt(highScoreKey, 0).ToString();

}

0 references
public void LoadLevel1()
{
    SceneManager.LoadScene(sceneBuildIndex: 1);
}
```

Fig 1.1 Main Menu code

The main menu uses the Command and Chain of Command behavioural pattern, because when a button is pressed, it will then send a command to Unity's SceneManager which tells it which level to load based on the index it is given. I also used Unity's in-built PlayerPrefs class to show in the main menu, the highest score achieved in the game.

2. Abstraction and Inheritance

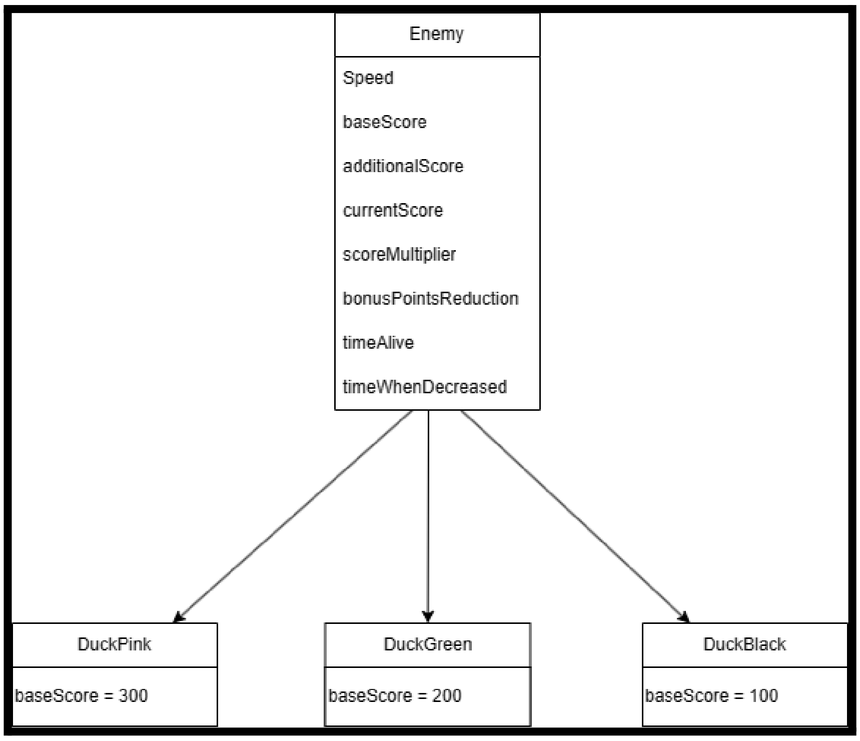


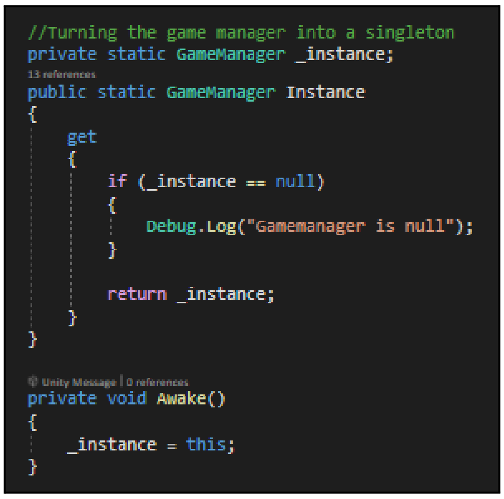
Figure 2.1 Shows the wireframe of the enemy class and its child classes

I used abstraction and inheritance to implement a class called Enemy that I can use to determine the different enemy types that spawn during the stages of the game. These enemies will each have a different base score, additional score and speed. I used this because it will help manage complexity between enemy classes, improve code reusability and help when expanding the game's enemy system. Using abstraction allows for a clear separation between what an object does and how it achieves it, promoting a modular and flexible design.

2.a **Bridge Structure**

Using abstraction also meant I had to use a **Bridge Structural design**, as each variation of the Enemy needed to have a different speed, colour, base score and additional score. This allowed me to create independent enemy types, while still keeping its core principles and functionality.

3. **Singleton**

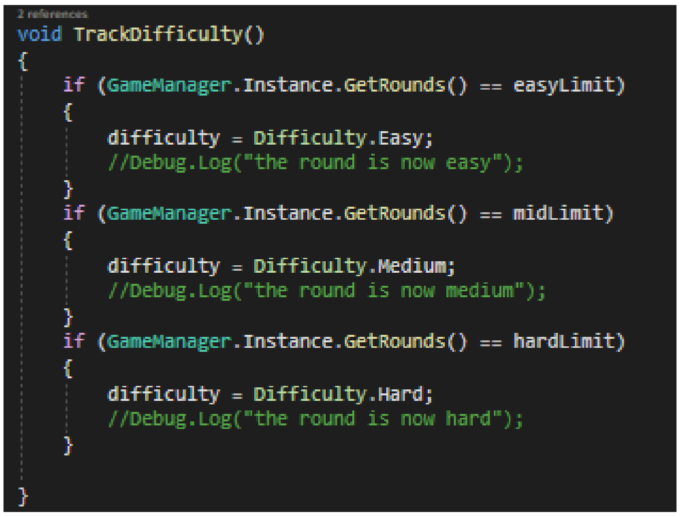


```
//Turning the game manager into a singleton
private static GameManager _instance;
13 references
public static GameManager Instance
{
    get
    {
        if (_instance == null)
        {
            Debug.Log("Gamemanager is null");
        }

        return _instance;
    }
}

Unity Message | 0 references
private void Awake()
{
    _instance = this;
}
```

Fig 3.1 Example code of the game manager

Given the time constraints, I opted to use a singleton for both the GameManager, and the SpawnManager and making a singleton provides a global point of access to the object. This was done to ensure that these managers have only one instance running, and this helped as I would not need to constantly reference the managers class in other game objects. It also provides a singular point of control and is persistent .

4. **FiniteStateMachine/Switches**



```
2 references
void TrackDifficulty()
{
    if (GameManager.Instance.GetRounds() == easyLimit)
    {
        difficulty = Difficulty.Easy;
        //Debug.Log("the round is now easy");
    }
    if (GameManager.Instance.GetRounds() == midLimit)
    {
        difficulty = Difficulty.Medium;
        //Debug.Log("the round is now medium");
    }
    if (GameManager.Instance.GetRounds() == hardLimit)
    {
        difficulty = Difficulty.Hard;
        //Debug.Log("the round is now hard");
    }

}
```

4.1 Above shows the functions that keeps track of the games states

I used Finite State Machines and Switches to help control the difficulty within the game manager. This helped with building readable code by reducing how many ‘if statements’ were written, and if I wanted to build onto the states further by adding more complex behaviours I could easily do so.

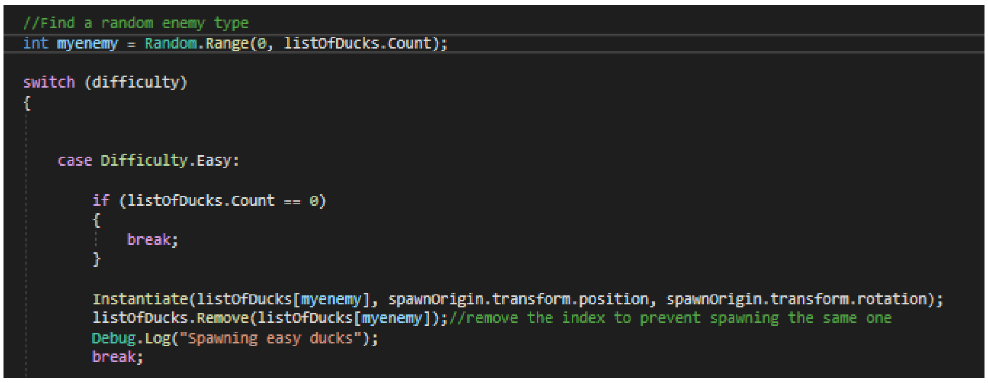
5. Arrays

It made sense to use an array instead of a list because the targets that an enemy could potentially fly to remained static. I also opted for an array instead of another data structure, such as a dictionary, because an array will *always* have fast look-up times, with the time complexity being O(1). Dictionaries have an O(1) look-up on *average*, but in the *worst* case it would be O(n) due to hash collisions.

Arrays are indexed sequentially, and are contiguous in nature, whereas dictionaries don’t have either of those qualities. This means that the CPU can take advantage of optimization techniques such as cache lines and prefetching.

However, when deleting enemy types from a list in my SpawnBehaviour, I was unable to use a normal array as it is a static data structure, and therefore its size cannot be modified.

6. Lists

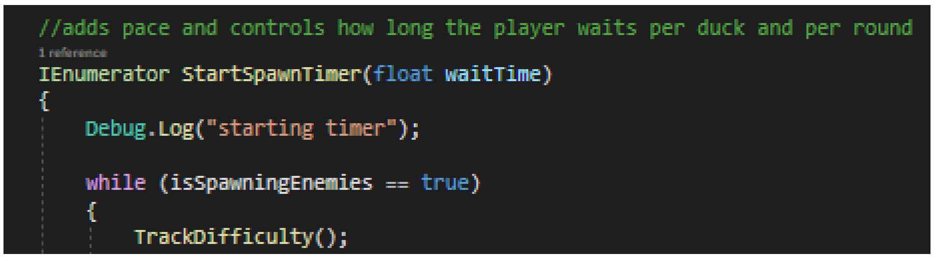


```
//Find a random enemy type
int myenemy = Random.Range(0, listOfDucks.Count);

switch (difficulty)
{


    case Difficulty.Easy:

        if (listOfDucks.Count == 0)
        {
            break;
        }

        Instantiate(listOfDucks[myenemy], spawnOrigin.transform.position, spawnOrigin.transform.rotation);
        listOfDucks.Remove(listOfDucks[myenemy]);//remove the index to prevent spawning the same one
        Debug.Log("Spawning easy ducks");
        break;
```

Fig 6.1 Above shows the deletion of the index depending on the case.

To ensure the game didn't feel too predictable, I made sure to spawn a random enemy type based on the assigned index, then delete the reference at that index when it was spawned so it could not be used again. When all 10 enemies are killed, a new list would be created based on the ramping difficulty and then we repeat the process in the next round. As lists are dynamic, it was suitable to use it for counting the amount of enemies to spawn and the size of collection changes.

7. Coroutines

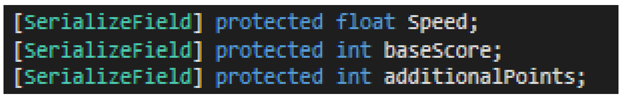


```
//adds pace and controls how long the player waits per duck and per round
1 reference
IEnumerator StartSpawnTimer(float waitTime)
{
    Debug.Log("starting timer");

    while (isSpawningEnemies == true)
    {
        TrackDifficulty();
```

Fig 7.1 Above shows the function name used for spawning enemies

For this project I wanted to keep to the practice of reducing the use of the 'Update()' function, so instead I used Unity's Coroutine function to help control the pacing of how fast each duck spawns, when and allow more control with the wait time between rounds. Corountines allow for more control of asynchronous tasks without blocking the main thread of code.

8. Protected variables



```
[SerializeField] protected float Speed;
[SerializeField] protected int baseScore;
[SerializeField] protected int additionalPoints;
```

Fig 8.1 Example of some variables in the 'Enemy' Class

Protected variables were used in the 'Enemy' base class as I only wanted them to be accessed within subclasses. This allowed me to control what objects had access to certain variables while still maintaining the integrity of the code.

9. Evaluation

With extra time I could explore testing and using more advanced systems in unity such as:

9.a Scriptable Objects

Considering the size of the game, it could have been built entirely using scriptable objects, as they allow storing data without being attached to a specific game object. This is very efficient when creating projects which require a modular and reusable approach for game elements.

9.b Event System

Instead of directly calling the function from the Managers in the game, I could've also used Unity's Event system. This is good for sending messages to different game objects without having to directly reference them. This minimises the need for game objects being dependent on one another and makes it easier to modify some systems.

9.c Interfaces/Managers

As the project was small there was no need for me to use interfaces. If the project was to be built upon further, it would be wise to use them for objects such as the Player and Enemy. Classes that use an interface must follow a contract, and this helps ensure consistent behaviour across different game objects.

9.d Unity's ECS

Another architecture that could be used if the project was to be built upon further would be Unity ECS(Entity Component System). This would be good for large scale AI tasks and could help optimise memory usage.

10. Behavioral Design Improvements

10.a StateDesign

Similar to the Finite state machine, however the finite state machine starts to become a problem the more conditions you add to the switches, in the end the code becomes unmanageable. Alternatively, you could create a state ‘Class’ that holds a reference to the state that the object is in.