Object Pooling:

Object pooling is implemented by creating an object pool class and declaring the object prefab and instantiating the number of objects needed. The duck object is stored and if needs to be deactivated and the game is set to inactive and stored within the ‘pools’ list.

```
public static void Unactive(GameObject duck)
{
    string key = duck.name.Replace("(Clone)", "");

    if (pools.ContainsKey(key))
    {
        pools[key].inactive.Push(duck);
        duck.transform.position = pools[key].Enemy.transform.position;
        duck.SetActive(false);
    }
    else
    {
        GameObject newDuck = new GameObject($"{key}_POOL");
        Pools newPool = new Pools(newDuck);

        duck.transform.SetParent(newDuck.transform);

        pools.Add(key, newPool);
        pools[key].inactive.Push(duck);
        duck.SetActive(false);
    }
}
```

When we need to instantiate the duck will be reactivated but setting it to active and setting its position, rotation, etc.

```
public static void Active(GameObject duck, Vector3 pos, Quaternion rot)
{
    GameObject obj;
    string key = duck.name.Replace("(Clone)", "");

    if (pools.ContainsKey(key))
    {
        if (pools[key].inactive.Count == 0)
        {
            Object.Instantiate(duck, pos, rot, pools[key].Enemy.transform);
        }
        else
        {
            obj = pools[key].inactive.Pop();
            obj.transform.position = pos;
            obj.transform.rotation = rot;
            obj.SetActive(true);
        }
    }
    else
    {
        GameObject newEnemy = new GameObject($"{key}_POOL");
        Object.Instantiate(duck, pos, rot, newEnemy.transform);
        Pools newPools = new Pools(newEnemy);
        pools.Add(key, newPools);
    }
}
```

This implementation should optimize the scene since it will uninstantiate the enemy objects once they are killed and reinstantiate the objects once they're needed again for the next level. This will save memory and optimize performance since instantiating and destroying every enemy object when they're killed can be costly. The unity profiler will display a big spike at the beginning and end of every level since it instantiates and destroys the game objects. Once we implement pooling there should only be a spike at the beginning of the game since it will be the initial duck instantiation and minor spikes if move objects will need to be renergate when the level rounds increase. The spikes between levels will now be minor since these objects are only being set from inactive to active game objects.

Command Design Pattern:

The command design pattern was attempted to be implemented for rebinding the controller to invert the aim vertical movement if the player misses two ducks in a row. This can be done by checking the whether the shots collide the game objects tagged as enemy and if two shots do not collide if enemy then the command design pattern will execute the invert command by

vertically inverting the mouse tracking. This feature is less of a optimization pattern but it'll be beneficial to the game by adding a challenging element to player's ensuring the don't miss their shots. I was unfortunately unable to implement this.

Game Management System:
A memory management can be implemented into the game to help keep track of areas in the heap that are unused. When game objects are instantiated like the bullets or ducks a new block of memory is requested, which the manager will choose an unused area and removes the allocated memory from the unused space. This will greatly reduce the memory and computation time and benefit the game since it is a level based game where the ducks become resilient and increase in quantity as the level increases.