IronBelly Test Documentation by Hector Quintana

This document would explain the IronBelly Unity Test, my approach to solving it and documentation of the systems created.

The Test:

The test was to create a Unity project with a set of systems:

- Pool system
- Find nearest neighbor
- Random 3D movement within range

Prioritizing good execution of the systems and optimization, as well as setting up a Test Scene where the systems could be tested using UI inputs.

Pool System

The implementation of the Pool System is to create it as a generic Pool that can be extended, optimization was the key.
I created a generic Pool class, checking for a simple way to make it generic and optimize on its pool:

- The Pool<T> class is a generic class that can handle any type of object.
- The new() constraint to have the ability to create new T objects with a parameterless constructor.
- It uses a Queue<T> to store the pooled objects, using FIFO principle for optimization.
- The constructor initializes the pool with a specified number of objects.
- The ExpandPool method adds more objects to the pool.
- The GetObject method retrieves an object from the pool, expanding it if necessary.
- The ReturnObject method returns an object back to the pool.

After testing, I removed the new() constraint on the Pool system to prevent extra objects being created and considered it wasn't necessary for the test.
Instead adding a Func<T> to delegate methods for when creating new objects.

GameObjectPool system

The GameObjectPool system uses the generic Pool system to create a pool of GameObjects derived from a prefab.

- It allows setting the initial pool size and the prefab to pool via the inspector.
- In Start(), it initializes the pool and pre-instantiates the objects
- CreateObject to instantiate from the prefab, deactivate the object and return to the Pool.
- GetPooledObject retrieves an object from the pool and activates it.
- ReturnPooledObject deactivates the object and returns it to the pool.
- ExpandPool can be called to manually add more objects to the pool.

To get better testing results a SetRandomPos method was created to have the option to initialize the GameObject in a random position within a range.
But after continuing the test, knowing the 3D Movement System would control the initial positioning of the spawned objects, it was removed.

Add a List<GameObject> to save all active objects in the pool, as well as objectsSpawned integer to get a count of these active objects.
Create two Unity Action objects to activate when objects are pulled from the pool or returned to it.

Find Nearest Neighbor system

It needs to be a script attached to a GameObject, checking for all enabled GameObjects with the same script and getting the one nearest to him, connect them with a LineRenderer, this should happen dynamically.

- A static list allNeighbours is used to keep track of all instances of FindNearestNeighbour script attached to GameObjects in the scene. This ensures that newly instantiated objects are automatically considered.
- The Awake and OnEnabled methods add the instance to the list, OnDestroy and OnDisabled removes it when the GameObject is destroyed. This keeps the list updated with active GameObjects, avoiding looking for them when looking for the nearest neighbor.
- The FindNearest method iterates over the list of neighbors to find the closest one calculated using Vector3.Distance.
- A LineRenderer component is used to draw a line between the current GameObject and its nearest neighbor. The LateUpdate method continuously checks for the nearest neighbor and updates the line accordingly.

Created a material for the Line Renderer to help on optimization.
A Cube Test prefab was created with FindNearestNeighbour script attached to it, as well as the Line Renderer component and its material.
Another material was created with GPU instancing enabled to help in graphical optimization for the Cube Test instances.

3D Random Movement system

A “Random3DMovement“ script attached to the GameObject to control. Would generate a random position within a range base of 3 floats (x, y, z), and make the object move toward that position, changing constantly to another random position within the range.

- zoneX, zoneY, and zoneZ define the boundaries of the movement zone and are exposed to the editor for easy configuration.
- moveSpeed determines how fast the object moves.
- changeDirectionInterval specifies how often the object should change its direction.
- SetRandomTarget() calculates a new random target position within the defined zone.
- MoveTowardsTarget() moves the object towards the target position.
- UpdateTargetPosition() updates the target position at regular intervals defined by changeDirectionInterval.
- Using a Coroutine UpdateTargetPosition to check the time interval to set a new random position optimize the callbacks.

Added the script to Test Cube prefab.

After checking next steps on the Test, also added the SetInitialPosition to create a random position to initialize the GameObject every time it's enabled.

Setting Test Scene

The Test Scene would be where all systems would be implemented, with a UI system to input changes on the GameObjectPool system to spawn and despawn objects.

Four UI objects were added to the scene.

- TextMeshProUGUI text label to show the count of active GameObjects from the GameObjectPool
- A InputField to get the output of the user of the quantity it wants to spawn and despawn.
- Two Buttons to activate the Spawn and Despawn methods of SpawnDespawnUISystem.

UI Systems

SetUIActiveObjectsCount is in charge of getting the count of active objects of the GameObjectPool in the scene and showing it in the UI label.
SpawnDespawnUISystem is in charge of getting the input from the Input Field, and Spawn or Despawn GameObjects form the GameObjectPool, considering the output the user can give to prevent errors.