1. Introduction

This project simulates a golf course setting where an NPC collects golf balls while managing stamina and optimizing its actions based on predefined game styles. Players can choose between **Chain Style** and **Sequence Style**, which influence how the NPC navigates and prioritizes tasks. The project utilizes Unity's **NavMesh** system, **Behaviour Tree** patterns, and custom logic to create an interactive decision-making process.

---

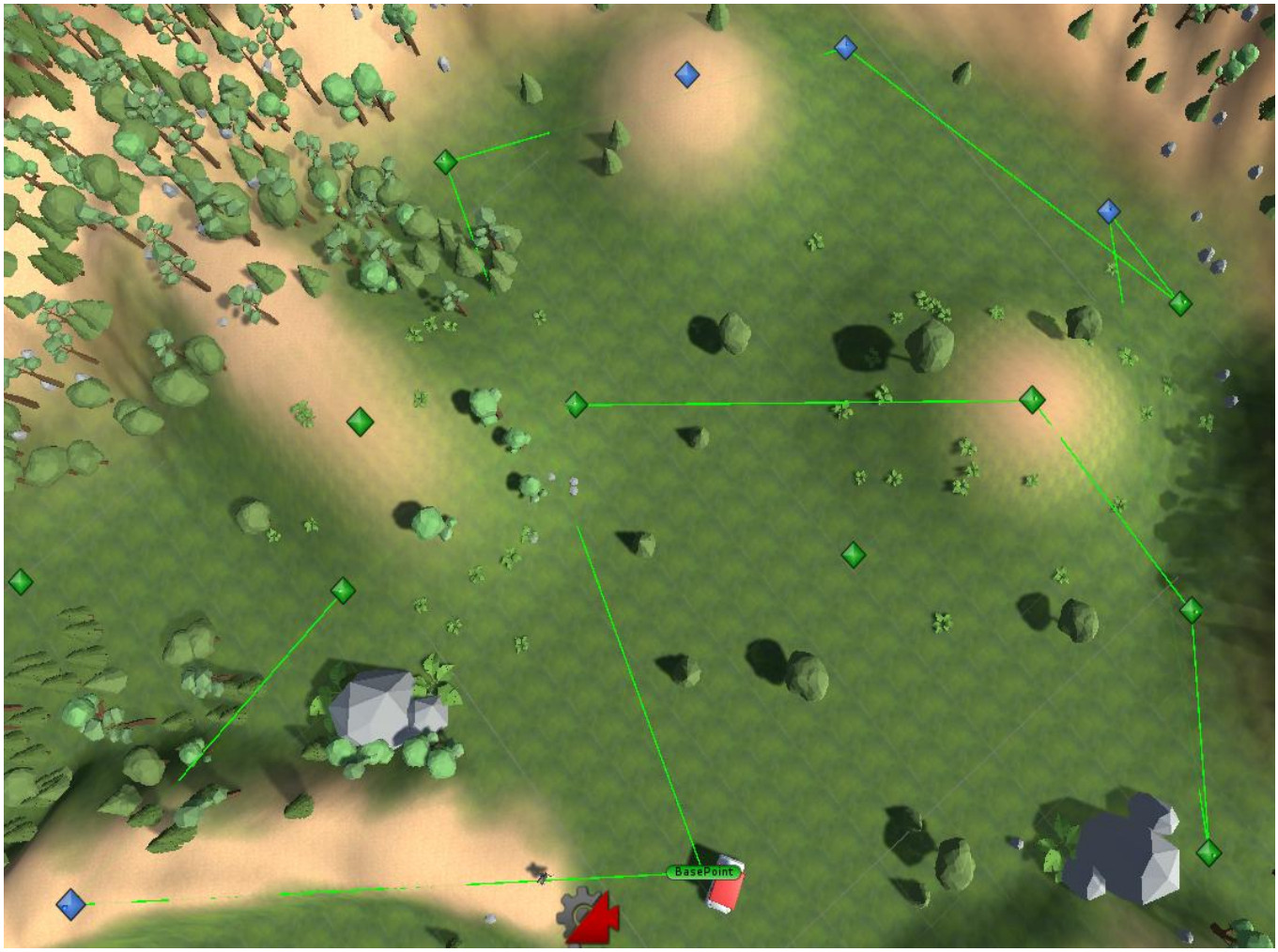
2. Game Mechanics

Objective

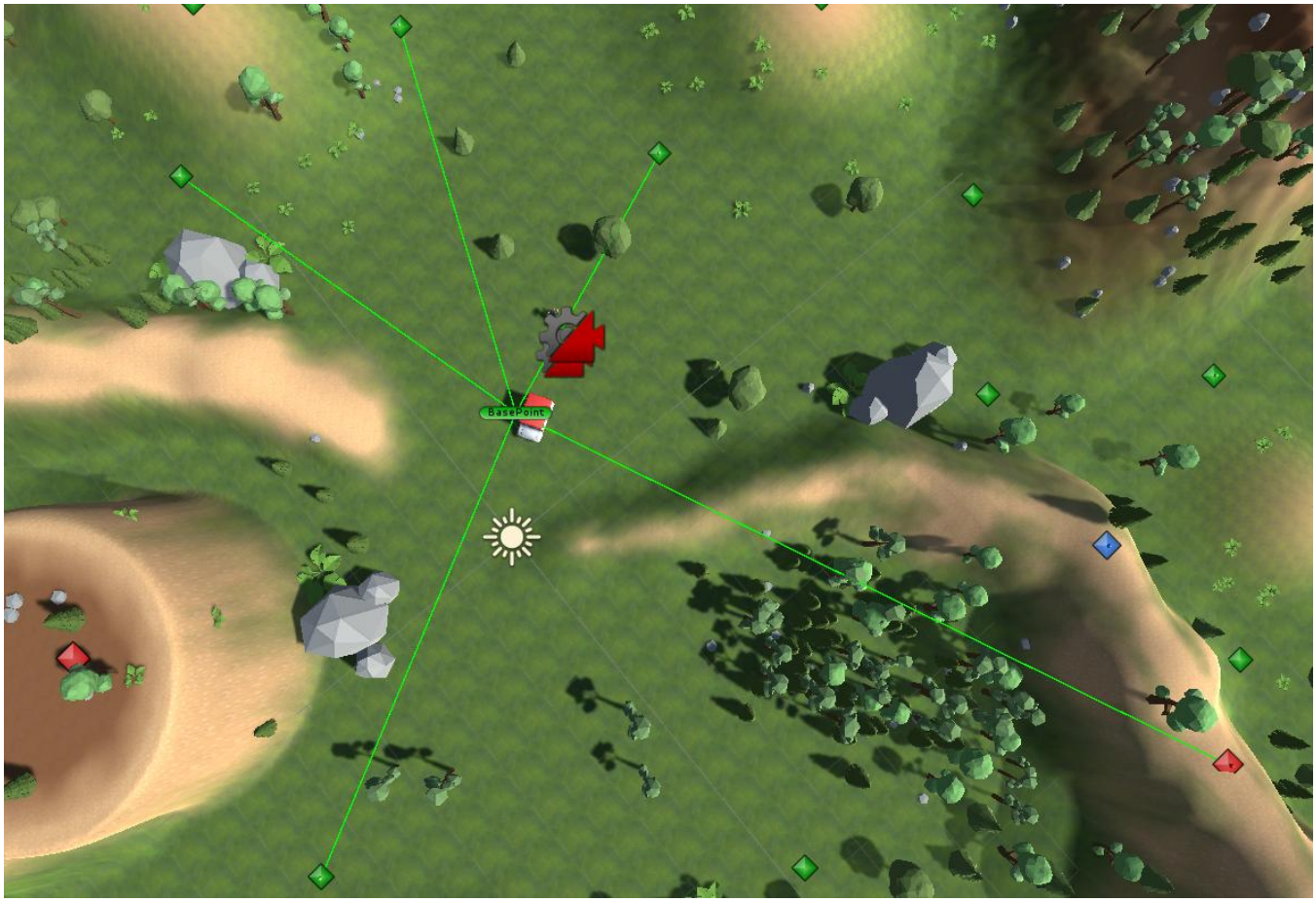
The NPC collects golf balls scattered across a map, either in one continuous trip or by returning periodically to a designated drop-off point (cart). The goal is to maximize rewards before running out of stamina.

Game Styles

1. **Chain Style**:
    - The NPC collects as many golf balls as possible in one trip.
    - Returns to the cart only after all possible balls are collected.



2. **Sequence Style**:
    - The NPC collects one golf ball at a time.
    - Returns to the cart after each collection.



Core Gameplay Flow

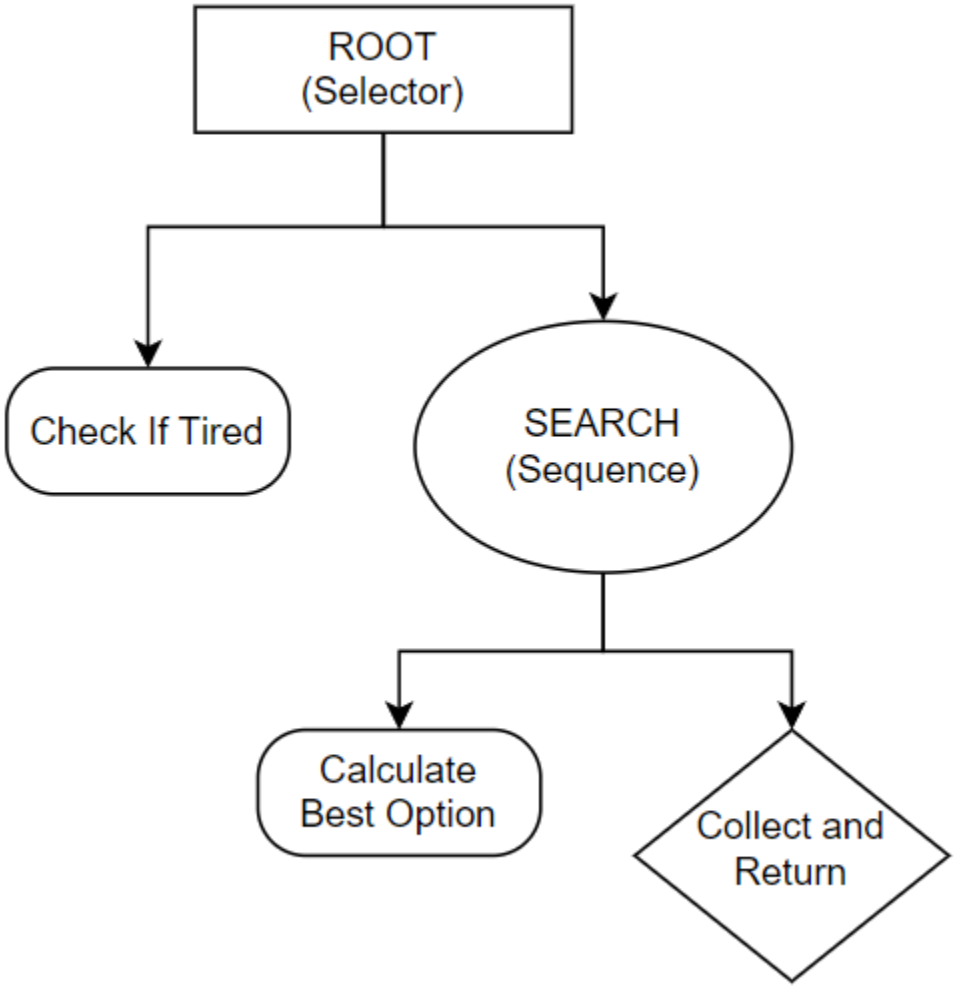
- The player selects a game style via radio buttons in the UI at start.
- The NPC navigates the map based on the selected style.
- The game ends when either:
    - All golf balls are collected.
    - The NPC runs out of stamina.

Player Interaction

- **UI**: A basic menu which also includes radio buttons for game style selection.

3. Decision-Making Algorithm

The decision-making logic is implemented using a **Behaviour Tree** pattern, allowing for modular and extensible decision flows.



Behaviour Tree Structure

1. **Root Node**:
    - Selector combining two sequences: one for checking stamina, another for performing actions.

2. **Nodes**:
    - **CheckIfTired**: Determines if stamina is too low to proceed.
    - **CalculateBestOption**: Selects the optimal ball or chain of balls to collect.
    - **Collect**: Executes the movement sequence for the chosen target(s).

Behaviour Tree Script Overview

Base Class for Trees:
The abstract Tree class initializes the Behaviour Tree and ensures evaluation only occurs when the game is active.

```
protected void Update()
{
    if (!_controller.IsPlaying) return;
    _root?.Evaluate();
}
```

Root Setup for NPC Behaviour:
The NPC-specific behaviour tree combines logical nodes into a sequence and selector hierarchy.

```
protected override Node SetupTree()
{
    var checkIfTired = new CheckIfTired(_controller);
    var calculateBestOption = new CalculateBestOption(_controller);
    var collect = new Collect(_controller);

    Sequence search = new Sequence(new List<Node> { calculateBestOption, collect });
    Selector root = new Selector(new List<Node> { checkIfTired, search });

    return root;
}
```

Algorithm Details

Path Cost Calculation

```
private float CalculatePathCost(Vector3 from, Vector3 to)
{
    var path = new NavMeshPath();
    if (NavMesh.CalculatePath(from, to, NavMesh.AllAreas, path)
        && path.status == NavMeshPathStatus.PathComplete)
    {
        var pathLength = 0f;
        for (var i = 1; i < path.corners.Length; i++)
            pathLength += Vector3.Distance(path.corners[i - 1], path.corners[i]);

        return pathLength * _controller.GetStaminaCostPerUnit();
    }

    return float.MaxValue;
}
```

- The NPC calculates the stamina cost of moving between positions using Unity's NavMeshPath.

Reward Evaluation

The algorithm determines the best collectable based on the following formula:

```
var score = collectable.RewardPoint - totalCost;
```

Where totalCost includes:

- Stamina cost to the ball.
- Stamina cost to return to the cart (if in **Sequence Style**).

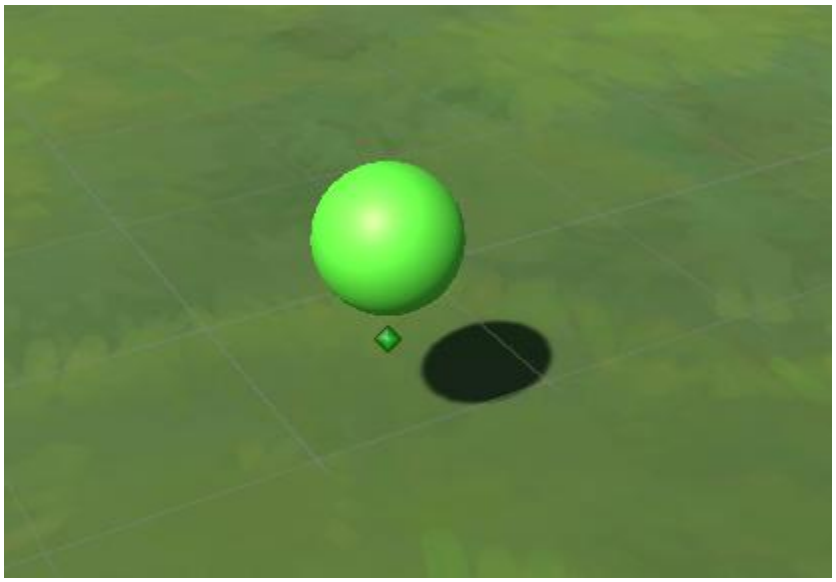
Key Algorithms in Action

1. **Chain vs. Sequence**: The GenerateChain method dynamically adapts the NPC's actions based on the chosen game style.

```
switch (gameStyle)
{
    case GameStyle.Sequence:
        chain.Add(cartTransform);
        currentStamina -= CalculatePathCost(bestNext.transform.position, cartTransform.position);
        currentPosition = cartTransform.position;
        break;
    case GameStyle.Chaining:
        currentPosition = bestNext.transform.position;
        break;
}
```

4. Collectables

The Collectables system represents golf balls scattered across the course. These objects provide points when collected by the NPC and use animations to enhance visibility and engagement.



Key Components

a. Collectable Base Class

The Collectable_Base abstract class defines the basic structure for collectables, including:

- **Reward Points**: Points assigned to each collectable.
- Abstract methods:
    - **Setup**: Initializes reward points.
    - **Collect**: Handles actions when collected.

```
1. public abstract class Collectable_Base : MonoBehaviour
2. {
3.     public int RewardPoint => _rewardPoint;
4.     protected int _rewardPoint;
5.
6.     public abstract void Setup(int rewardPoint);
7.     public abstract int Collect();
8. }
```

b. Golf Ball Collectable

The Collectable_GolfBall class implements:

- **Setup**: Assigns reward points dynamically.
- **Collect**: Disables the collectable after interaction and awards points.
- **Floating Animation**: Uses a sinusoidal motion for visibility.

```
1. public class Collectable_GolfBall : Collectable_Base
2. {
3.     public override void Setup(int rewardPoint) => _rewardPoint = rewardPoint;
4.
5.     public override int Collect()
6.     {
7.         Debug.Log($"{name} Collected!");
8.         return RewardPoint;
9.     }
10.
11.     IEnumerator AnimateCollectable()
12.     {
13.         while (true)
14.         {
15.             model.position += Vector3.up * Mathf.Sin(Time.time) * 0.5f;
16.             yield return null;
17.         }
18.     }
19. }
20.
```

---

4. Managers

The Managers layer centralizes game logic, including state management, collectable spawning, scene control, scoring, UI updates, and camera handling. This design ensures modularity and simplifies coordination between game systems.

Key Components

a. GameManager

Handles core game state transitions and play style updates:

- **GameState**: Tracks states like Menu, Playing, and Tired.
- **GameStyle**: Toggles between Chaining and Sequence modes.

Example:

```
1. public void ChangeState(GameState newState)
2. {
3.     if (newState == State) return;
4.     OnBeforeStateChanged?.Invoke(newState);
5.     State = newState;
6.     OnAfterStateChanged?.Invoke(newState);
7.     Debug.Log($"New state: {newState}");
8. }
```

b. CollectableManager

Spawns and manages collectables on the terrain:

- Adjusts spawn points to terrain height.
- Spawns collectables based on available points and configurations.

Features:

- **Dynamic Spawning**: Balances object count with spawn locations.
- **CollectableData**: Stores prefab references, spawn counts, and reward points.

Example:

```
 1. private void SpawnCollectables()
 2. {
 3.     foreach (var collectable in collectables)
 4.     {
 5.         var spawnPoints = new List<Transform>(collectable.PossibleSpawnPoints);
 6.         for (int i = 0; i < collectable.SpawnCount; i++)
 7.         {
 8.             if (spawnPoints.Count == 0) break;
 9.             var index = Random.Range(0, spawnPoints.Count);
10.             var instance = Instantiate(collectable.Prefab, spawnPoints[index].position,
Quaternion.identity);
11.             instance.Setup(collectable.RewardPoint);
12.             spawnPoints.RemoveAt(index);
13.         }
14.     }
15. }
```

c. SceneManager

Handles scene reloading:

- **ReloadScene**: Restarts the current scene.

d. ScoreManager

Manages and updates player scores:

- **OnScoreChanged**: Adjusts score and updates the UI.
- **Score Display**: Uses UIManager for consistent visuals.

e. UIManager

Controls UI states and displays:

- **Canvas Management**: Activates/deactivates UI elements based on GameState.
- **Stamina Bar**: Dynamically reflects the NPC's stamina.

Example:

```
1. private void OnGameStateChanged(GameState state)
2. {
3.     mainMenuCanvas.SetActive(state == GameState.Menu);
4.     gamePanelCanvas.SetActive(state == GameState.Playing);
5. }
```

f. CameraManager

Switches between camera views based on the game state:

- **FollowCam**: Focuses on NPC during gameplay.
- **CloseLookupCam**: Focuses on close lookup on NPC during menu state.

5. NPC

This codebase manages the behavior, movement, animation, and stamina system of a Non-Playable Character (NPC) within a game. Each component handles specific functionality, defining how the NPC interacts with the game world.

a. NPCController

Primary Role: The central controller for the NPC, coordinating other components.

b. AnimationController

Primary Role: Manages NPC animations.

c. MovementController

Primary Role: Handles NPC movement and path planning.

d. StaminaController

Primary Role: Calculates and manages the NPC's stamina consumption.

Conclusion

Modularity of design with custom managers allows this project to scale well and perform effectively. Moreover, the integration of the Behavior Tree mechanism for decision-making by NPCs is included. For further information, refer to my post on Medium.