Raptor-AI

Progetto di Fondamenti di Intelligenza Artificiale

Gennaio 2023

Repository Github: https://github.com/Hikki00/Raptor-AI.git

Membri:

- Consiglio Luigi, 0512110485
- D'Antuono Francesco Paolo, 0512109798
- Miron Roberto Andrei, 0512110581
- Scermino Simone, 0512110611
- Vitale Ciro, 0512110719

Indice

1 Introduzione

Raptor-AI è un videogioco che si basa sull'utilizzo del Reinforcement Learning per addestrare un agente, nel nostro caso un kart, a compiere scelte che gli permettano di completare un tracciato di gioco nel modo più efficiente possibile. Per addestrare l'agente, è stata utilizzata la libreria open-source *MLAgents* di *Unity* e il framework open-source *PyTorch*, che consentono la creazione, l'addestramento e valutazione dei modelli di Intelligenza Artificiale necessari per il controllo degli agenti all'interno dell'ambiente. Inoltre, è stato utilizzato *C#* per l'implementazione degli script dell'agente e *TensorFlow* per visualizzare e analizzare i risultati.

Il kart è stato sottoposto a diverse sessioni di training sui quattro tracciati previsti, durante le quali ha dovuto affrontare diverse sfide come curve e ostacoli. Successivamente, grazie al Reinforcement Learning, il kart ha acquisito la capacità di superare le difficoltà e di perfezionare la propria performance nel completamento del percorso.

I risultati ottenuti sono stati molto positivi, il kart ha ottenuto una progressione nei suoi tempi di gara e ha incrementato le sue performance durante il corso delle sessioni di allenamento. Nella prima sezione del documento, esploreremo le caratteristiche dell'ambiente in cui il kart si muove, in modo da comprendere come esso dovrà interagire e come potrà imparare ad adattarsi. La seconda parte del testo sarà dedicata alla descrizione del metodo di progettazione del sistema di apprendimento utilizzato per il kart. Verranno illustrate le tecniche adottate per rendere il veicolo in grado di navigare autonomamente i circuiti.

2 Obiettivi

L'obiettivo del progetto è quello di realizzare un kart autonomo che sia in grado di apprendere e migliorare la propria guida su quattro tracciati differenti, attraverso l'utilizzo del Reinforcement Learning, con l'aggiunta di ostacoli per aumentare la sfida. Inoltre, sarà possibile giocare contro gli agenti addestrati utilizzando il proprio kart.

2.1 PEAS

Un ambiente viene generalmente descritto tramite la formulazione PEAS, ovvero Performance, Environment, Actuators e Sensors.

- *Performance Measure.* La misura di prestazione adottata prevede la correttezza di esecuzione nel minor tempo possibile di un tracciato. Questo dipenderà soprattutto dai reward associati ad ogni oggetto di scena che l'agente toccherà nell'addestramento.
- *Environment.* L'ambiente è costituito da una scena, che contiene tutti i tracciati, sul quale è stato addestrato l'agente e che possono essere giocabili da una singola persona selezionando il tracciato e il numero di giocatori con i quali vuole giocare. Il numero di giocatori scelto corrisponderà al numero di agenti che competeranno contro il giocatore. Ogni tracciato sarà costituito da checkpoint, muri, linea di traguardo e eventualmente ostacoli che permetteranno all'agente di capire, tramite il reward associato ad essi, come si dovrà muoversi all'interno di esso.
- *Actuators.* Gli attuatori sono componenti essenziali che consentono all'agente di interagire con l'ambiente circostante e di effettuare varie azioni. Nel nostro caso specifico, abbiamo un kart i cui attuatori includono dispositivi di movimento come un sistema di propulsione, frenata e sterzata, che consentono al veicolo di accelerare, frenare e cambiare direzione.
- *Sensors.* I sensori attraverso i quali riceve gli input percettivi, sono degli oggetti di Unity denominati `Ray Perception Sensor 3D` che, ci consentono di ampliare il campo visivo del kart e di riconoscere gli oggetti di scena, così da poter evitare eventuali collisioni.

2.1.1 Caratteristiche dell'ambiente

L'ambiente operativo è:

- *Parzialmente Osservabile.* L'agente non ha la possibilità di vedere tutto il tracciato, ma solo ciò che i sensori rilevano.
- *Stocastico.* L'agente compierà azioni specifiche che possono portare a risultati e stati differenti e quindi lo stato successivo dell'ambiente risulta essere incerto.
- *Sequenziale.* Ogni decisione può influenzare tutte quello successive.
- *Semi-Dinamico.* L'ambiente non cambia, ma il comportamento dell'agente sull'ambiente cambia a seconda di come osserva l'ambiente e di ciò che è riuscito ad apprendere.
- *Discreto.* Le azioni del kart saranno azioni distinte e sono: accelerazione in avanti, sterzata a destra, sterzata a sinistra e frenata.
- *Multi-Agente.* L'ambiente permette di addestrare l'agente utilizzandone più di uno, inoltre anche quando si vuole giocare, lo si può fare contro più di un agente.

2.2 Analisi del problema

Il problema richiede di riuscire ad addestrare uno o più agenti, su differenti tracciati, aventi diverse difficoltà. Ci sono diversi algoritmi di apprendimento che abbiamo analizzato per addestrare un kart su tracciati diversi oltre al Reinforcement Learning:

- *Apprendimento Supervisionato.* Costituito da una grande quantità di dati di tracciati già percorsi da kart addestrati, etichettati con le azioni appropriate per ogni punto del tracciato, per prevedere le azioni appropriate per nuovi tracciati.
- *Apprendimento Non Supervisionato.* Analizza i dati dei sensori del kart e identifica pattern nelle azioni del kart che possono essere utilizzati per sviluppare una strategia di guida.
- *Apprendimento Evolutivo.* Genera e testa diverse strategie di guida per il kart, utilizzando la selezione naturale per individuare le strategie più efficaci.

Ci sono alcuni svantaggi comuni nell'utilizzo di algoritmi di apprendimento supervisionato, non supervisionato e evolutivo per addestrare un kart su tracciati diversi:

- *Apprendimento Supervisionato.* Richiede una grande quantità di dati etichettati, che possono essere costosi e difficili da raccogliere per tracciati non ancora percorsi dal kart. Inoltre, il modello addestrato può sovrastimare le sue prestazioni se i dati di addestramento non rappresentano adeguatamente la varietà dei tracciati che il kart deve affrontare.
- *Apprendimento Non Supervisionato.* Può essere difficile da interpretare i risultati ottenuti e trovare relazioni utili tra i dati dei sensori e le azioni del kart. Inoltre, gli algoritmi di apprendimento non supervisionato non forniscono direttive su come agire sui dati, quindi è necessario sviluppare ulteriori metodi per utilizzare i pattern identificati.
- *Apprendimento Evolutivo.* Richiede una grande quantità di tentativi ed errori per generare e testare diverse strategie di guida, il che può essere computazionalmente costoso e richiedere molto tempo. Inoltre, gli algoritmi di apprendimento evolutivo possono essere sensibili ai parametri di configurazione e possono essere difficili da ottimizzare.

Tuttavia, è importante notare che ognuno di questi algoritmi ha le sue specificità e potrebbe essere adatto per una determinata applicazione. La scelta dipende dalle specifiche del progetto, dai dati disponibili e dalle risorse a disposizione, in questo caso la scelta è ricaduta sul Reinforcement Learning perché permette al kart di imparare attraverso tentativi ed errori, adattando la propria strategia in base alle ricompense o punizioni ricevute. Inoltre, il Reinforcement Learning è in grado di gestire situazioni incerte e non lineari, come quelle che si possono incontrare in un tracciato di guida, rendendolo un'opzione ideale per l'addestramento del kart.

Il **Reinforcement Learning** è una tecnica di Machine Learning, in cui un agente interagisce con l'ambiente per ricevere feedback, in forma di ricompense (reward), durante l'esecuzione dell'agente. L'agente prende decisioni su come agire basandosi sull'esperienza passata e sulla ricompensa ottenuta, con l'obiettivo di massimizzare il suo guadagno a lungo termine. Nel Reinforcement Learning, l'agente impara attraverso la sperimentazione e l'esperienza, anziché tramite la supervisione o l'apprendimento per imitazione. Si tratta di un approccio utilizzato in diversi ambiti, come il gioco, la robotica e il Natural Language Processing.

Un esempio di Reinforcement Learning può essere il comportamento di un animale che cerca di ottenere il cibo. Supponiamo che l'animale sia in un labirinto e che ci siano diversi percorsi che può seguire per raggiungere il cibo. Inizialmente, l'animale potrebbe scegliere casualmente un percorso e seguirlo, ottenendo o meno il cibo alla fine. Ogni volta che l'animale ottiene il cibo, riceve una ricompensa positiva. Se invece non ottiene il cibo, riceve una ricompensa negativa. Con il tempo, l'animale impara a scegliere i percorsi che gli hanno permesso di ottenere il cibo con maggiore frequenza, evitando quelli che invece non hanno portato a una ricompensa. In questo modo, l'animale "impara" a scegliere il percorso più adeguato per ottenere il cibo, massimizzando il suo guadagno a lungo termine.

Il Reinforcement Learning è stato probabilmente la scelta migliore perché consente di addestrare il kart a prendere decisioni efficaci in situazioni complesse e incerte, utilizzando un processo di apprendimento per tentativi ed errori. Inoltre, il Reinforcement Learning è particolarmente adatto per i sistemi che operano in ambienti dinamici e in cui i dati di addestramento sono limitati o costosi da raccogliere.

3 Soluzione del problema

Abbiamo un personaggio e una lista di checkpoint, l'obiettivo è far muovere il personaggio attraverso i checkpoint in ordine, senza deviare dal percorso. Creiamo un agente, che gestirà l'Intelligenza Artificiale per l'allenamento e il gioco. Per sviluppare un agente che sarà in grado di navigare autonomamente senza errori lungo un percorso di qualsiasi difficoltà, creiamo uno script chiamato `MoveToCheckpointsAgent` e utilizziamo `Unity.MLAgents` invece di ereditare da `MonoBehaviour`, ereditando invece dalla classe `Agent`.

Il modo in cui l'agente impara è attraverso il Reinforcement Learning, che si basa su un loop: l'agente ottiene informazioni dall'ambiente mediante le *observation*, poi, prende una *decision* basata sui dati che possiede e, infine, fa un'*action*, se essa è giusta esso ottiene un *reward* positivo altrimenti negativo. Quindi, questo è un ciclo continuo, dove l'agente impara continuamente in base alle sue osservazioni e quali azioni portano alle ricompense più alte.

Per applicare un **algoritmo** di Reinforcement Learning, in un videogame di kart racing, per far completare il tracciato in maniera corretta, bisognerà seguire i seguenti passi:

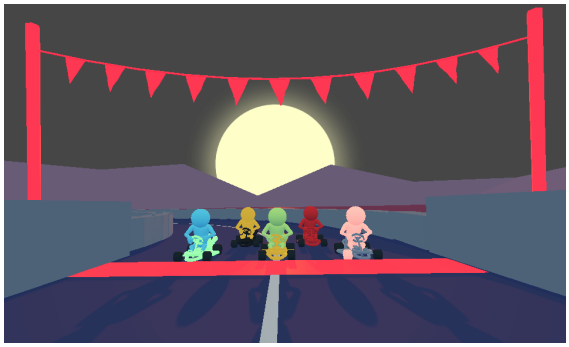
1. *Definire l'ambiente.* L'ambiente sarà costituito dal tracciato del gioco, dalle caratteristiche della macchina e dalle azioni che questa può compiere (accelerazione in avanti, accelerazione all'indietro, sterzata a destra, sterzata a sinistra e frenata);
2. *Scegliere l'algoritmo di Reinforcement Learning.* L'algoritmo si utilizzerà per far "imparare" alla macchina a completare il tracciato.
3. *Definire le ricompense.* Definire le ricompense che verranno assegnate alla macchina in base alle azioni che compie. Assegniamo una ricompensa positiva se la macchina riesce a completare il tracciato senza schiantarsi contro un muro oppure oltrepassare i corretti checkpoint, e assegniamo una ricompensa negativa se la macchina si scontra contro i muri.
4. *Eseguire il training.* Per addestrare l'algoritmo, si utilizza un processo di ripetizione in cui la macchina gioca diverse volte, ogni volta utilizzando quanto appreso dalle partite precedenti per migliorare le proprie decisioni e azioni successive. In questo modo, l'algoritmo "impara" a scegliere le azioni più adatte per completare il tracciato nel modo più efficiente possibile, massimizzando il guadagno a lungo termine.
5. *Testare il risultato.* Testare il risultato, facendo giocare la macchina senza alcun intervento umano. Se l'algoritmo di Reinforcement Learning ha imparato correttamente, il kart sarà in grado di completare il tracciato da solo, senza bisogno di interventi umani.

4 Componenti del progetto

In questa sezione andremo a mostrare tutte le componenti del progetto che abbiamo utilizzato per realizzare gli obiettivi prefissati.

4.1 Kart

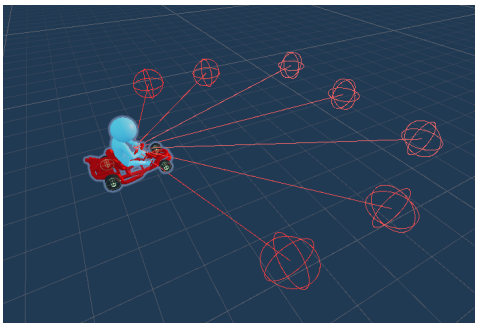
Il **kart** è l'agente del nostro sistema che dovrà apprendere come riuscire a completare il tracciato senza commettere errori e nel minor tempo possibile. Esso è costituito da un body e delle ruote e il suo comportamento è definito nello script `ArcadeKart` dove oltre ai comandi manuali sono settati tutti i parametri per poter effettuare tutte le azioni di: accelerazione, sterzata e frenata. Abbiamo implementato un meccanismo di controllo per distinguere tra le macchine presenti, identificando quella che agisce come agente e quella che non lo è, e per comunicare queste informazioni al motore grafico. Il kart, per permettere l'apprendimento, avrà dei sensori definiti con la componente di `Unity Ray Perception Sensor 3D` che permettono di riuscire a far vedere una parte più ampia dell'ambiente circostante, un collider, chiamato `KartBouncingCapsule`, che serve per riconoscere le collisioni nelle quali si troverà coinvolto, le animazioni corrispondenti alle azioni effettuate e lo script `KartClassicAgent` (estende la classe `Agent`, che fa parte di `Unity MLAgents`) che implementa l'algoritmo di Reinforcement Learning. Inoltre i `BehaviorParameters` definiscono i parametri che gli agenti utilizzeranno nell'addestramento. Gli algoritmi e gli script in generale verranno spiegati nella sezione *Algoritmo di Machine Learning*.



Il Kart è un `GameObject` il quale è un oggetto fondamentale di Unity che rappresenta qualsiasi elemento in una scena, come per esempio un personaggio, un pezzo di terreno, una telecamera o persino uno spazio vuoto. In breve, i `GameObject` possono essere definiti come la colonna vertebrale di Unity perché vengono utilizzate per creare, manipolare e organizzare tutti gli elementi di un gioco.

4.2 Sensori

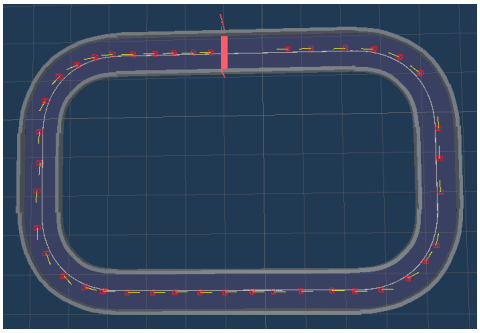
I sensori sono una componente di Unity, denominata `Ray Perception Sensor 3D`, che permette al kart di avere una visione dell'ambiente circostante. Rappresentano i sensors della formulazione PEAS dell'agente e dunque riusciranno a far osservare ad esso, in un certo range, l'ambiente vicino e fargli capire quindi dove si trovano gli oggetti di scena. Questi sensori utilizzano "raggi" (ovvero linee) per rilevare le collisioni con gli altri oggetti presenti nella scena. Il `Ray Perception Sensor 3D` in Unity è un componente che permette di simulare un sensore a raggi in un gioco. Questo componente consente di rilevare gli ostacoli e le collisioni nell'ambiente simulato, utilizzando raggi virtuali che "scansionano" l'ambiente intorno al giocatore. Noi, lo abbiamo utilizzato per rilevare collisioni con gli ostacoli lungo il percorso, come i muri veicoli, per determinare la distanza tra il kart e gli ostacoli, e per rilevare i checkpoint. In questo modo, il gioco può reagire alla collisione, ad esempio rallentando il kart e può anche aiutare il giocatore a evitare gli ostacoli in futuro. Il `Ray Perception Sensor 3D` è un'ottima scelta per i giochi di guida, poiché consente di creare un'esperienza di gioco realistica e reattiva.



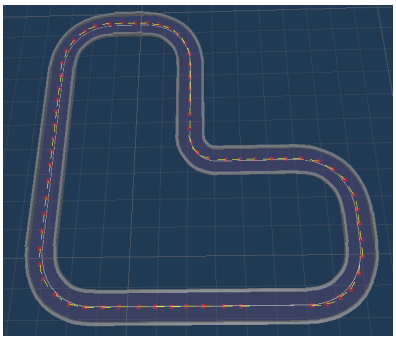
4.3 Tracciati

I tracciati rappresentano il nostro ambiente prinicipale dove le macchine competeranno per vincere. Essi sono composti da una linea di traguardo, chiamata `KartFinishLine` che permette di capire quando è che in un tracciato finisce il giro e dunque la gara, ovviamente il primo che arriva al traguardo vince la gara. Sono stati realizzati quattro tracciati ognuno aventi caratteristiche e difficoltà differenti, questi sono:

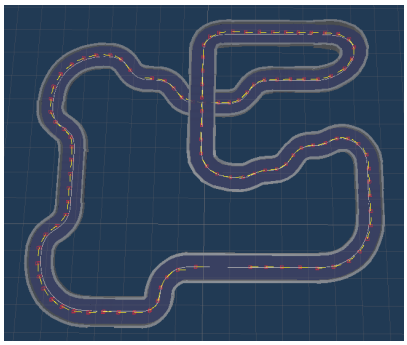
- *OvalTrack*. Rappresenta il tracciato più semplice che ci ha permesso di capire come inizialmente si dovesse addestrare il nostro agente.



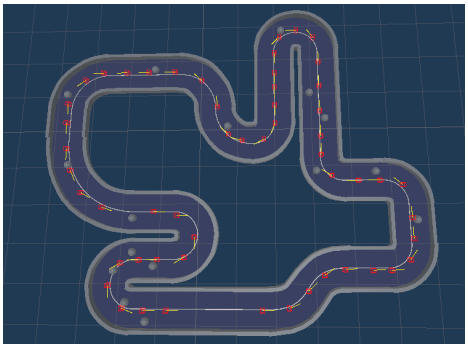
- *CountryTrack*. Rappresenta il tracciato più equilibrato poiché non è troppo lungo ma neanche troppo corto e aggiunge un minimo di difficoltà in più nell'apprendimento dell'agente.



- *MountainTrack*. Rappresenta il tracciato più lungo. Per addestrare le macchine abbiamo dovuto aumentare gli step e aggiungere dei reward più pesanti che permettessero di velocizzare il processo.

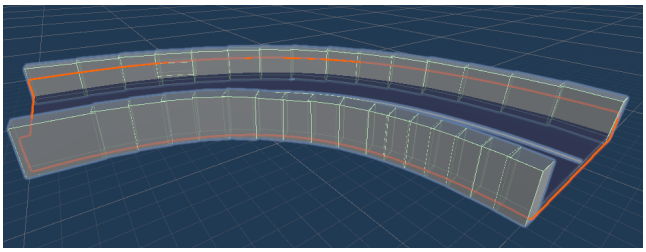


- *WindingTrack*. Rappresenta il tracciato più complesso, poiché pieno di curve ed ostacoli. Anche in questo caso abbiamo dovuto ricalcolare i reward per ogni azione compiuta del nostro agente poiché inizialmente, non riusciva ad apprendere in modo corretto.



4.4 Muri

I muri possono sembrare inutili, ma in realtà sono parte integrante dell'apprendimento poiché ci permettono di capire quando l'agente sta sbagliando e quindi ad esso sarà associato un reward negativo che permette all'agente poi di evitarlo nelle osservazioni successive. Inoltre, anche i muri hanno degli script associati che permettono di essere riconosciuti (come anche gli altri oggetti di scena). In particolare, abbiamo utilizzato la funzione `OnTriggerEnter` e `OnTriggerStay` per determinare quando effettivamente il collider del kart tocca con quello del muro e quindi assegnare poi il reward negativo.

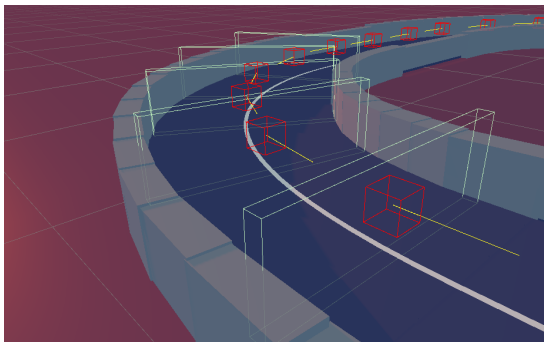


Ecco lo script per implementare il controllo dell'`OnTriggerEnter` che abbiamo citato sopra.

```
public class WallSingle : MonoBehaviour
{
    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Wheel" || other.tag == "Player")
        {
            Debug.Log("Wall Hit");
        }
    }
}
```

4.5 Checkpoint

I checkpoint sono anche essi parte integrante dell'apprendimento e svolgono una funzione uguale ed opposta a quella dei muri, poiché essi quando vengono toccati e dunque attraversati aggiungeranno un reward positivo e dunque in questo modo il kart capisce che passare nei checkpoint è l'azione che deve compiere. Come i muri, anche i checkpoint utilizzeranno `OnTriggerEnter` per controllare il contatto tra il collider del checkpoint e quello del kart.



In sintesi, implementare un sistema di checkpoint in un gioco di corse o in qualsiasi gioco in cui il giocatore o qualsiasi oggetto segua un percorso preimpostato, serve a identificare se il giocatore sta seguendo correttamente il percorso. Il sistema di checkpoint è stato creato per costringere il giocatore a seguire il percorso corretto, evitando che salti attraverso parti del percorso. Il sistema consiste nell'utilizzare checkpoint e verificare quando il giocatore ne attraversa uno, verificando che sia quello giusto e tracciando il prossimo checkpoint. Per tracciare quando il giocatore attraversa un checkpoint si utilizza un trigger collider semplice.

4.5.1 CheckpointSingle

```
public class CheckpointSingle : MonoBehaviour
{
    private TrackCheckpoints trackCheckpoints;
    private Vector3 vectorTransform;


//tiene traccia dei checkpoint superati da ogni macchina presente in gara
    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Player" || other.tag == "Agent")
        {
            trackCheckpoints.CarThroughCheckpoint(this, other.transform);
        }
    }

    //metodi di servizio (get, set)

    public void SetTrackCheckpoints(TrackCheckpoints trackCheckpoints)
    {
        this.trackCheckpoints = trackCheckpoints;
    }
```

Il metodo `OnTriggerEnter`, viene chiamato quando un collider in questo caso la nostra auto, entra in questo collider (cioè nel checkpoint). Il player ha un semplice componente `Player`, quindi possiamo semplicemente controllare se other.tag è uguale a player, e con questo identifichiamo che è il player. Lo script gestisce solo un singolo oggetto checkpoint. Ora abbiamo bisogno di un altro script che sarà responsabile dell'intera logica del tracciato e lo chiamiamo `TrackCheckpoints`.

4.5.2 TrackCheckpoints

Aggiungiamo su Unity un oggetto chiamato `AgentCheckpoints` ed esso conterrà i vari checkpoint e gli passiamo lo script che inseriremo in `TrackCheckpoints`. Nello script cicleremo attraverso questo container e passeremo attraverso i singoli checkpoint.

```
//inizializza il conteggio di checkpoints per ogni macchina in gioco
    private void Awake()
    {
        Transform checkpointsTransform = transform.Find("AgentCheckpoints");

        checkpointSingleList = new List<CheckpointSingle>();
        foreach (Transform checkpointSingleTransform in checkpointsTransform)
        {
            CheckpointSingle checkpointSingle = checkpointSingleTransform.GetComponent<CheckpointSingle>();

            checkpointSingle.SetTrackCheckpoints(this);
            checkpointSingle.SetVectorTranform(checkpointSingleTransform.transform.forward);
            checkpointSingleList.Add(checkpointSingle);
        }

        getLastCheckpoint();

        nextCheckpointSingleIndexList = new List<int>();
        foreach (Transform carTransform in carTransformList)
        {
            nextCheckpointSingleIndexList.Add(0);
        }
    }
```

Nel metodo `Awake`, cerchiamo il contenitore dei checkpoints e lo salviamo in `checkpointsTransform` adesso passiamo in rassegna tutti i figli con un `foreach`. Aggiungiamo il singolo checkpoint alla lista di checkpoints. In Unity, il componente `Transform` è una parte fondamentale del `GameObject`. Questo elemento è responsabile della posizione, della rotazione e della scala dell'oggetto 3D. Per tenere traccia dell'ordine dei checkpoints, facciamo un elenco per conservare tutti i nostri `CheckpointSingle`. Ogni volta che attraversiamo un checkpoint lo aggiungiamo alla lista. Dobbiamo tenere traccia del prossimo indice del checkpoint del giocatore, quindi creiamo un indice e quando startiamo esso parte da zero.

In sintesi, questo codice inizializza un conteggio di checkpoint per ogni macchina nel gioco, trovando tutti i componenti `CheckpointSingle` nella traccia e memorizzandoli in una lista. Inoltre, crea una lista per tenere traccia dell'indice del prossimo checkpoint per ogni macchina.

```
// controlla che il checkpoint attraversato dalla macchina sia quello giusto (devono essere attraversati in sequenza da 0 a n-1)
    // questo metodo viene utilizzato in particolare nel training, in modo da poter assegnare reward adeguati all'agente
    // checkpoint corretto = +1
    // checkpoint non corretto = -1
    public void CarThroughCheckpoint(CheckpointSingle checkpointSingle, Transform carTransform)
    {
        int nextCheckpointSingleIndex = nextCheckpointSingleIndexList[carTransformList.IndexOf(carTransform)];

        if (checkpointSingleList.IndexOf(checkpointSingle) == nextCheckpointSingleIndex)
        {
            nextCheckpointSingleIndexList[carTransformList.IndexOf(carTransform)] = (nextCheckpointSingleIndex + 1) % checkpointSingleList.Count;

            GameObject go = GameObject.Find(carTransform.name);
            KartClassicAgent other = (KartClassicAgent)go.GetComponentInParent(typeof(KartClassicAgent));
            if (other != null)
            {
                other.AddRewardOnCar(carTransform, 1f);
            }
        } else {
            GameObject go = GameObject.Find(carTransform.name);
            KartClassicAgent other = (KartClassicAgent)go.GetComponentInParent(typeof(KartClassicAgent));
            if (other != null)
            {
                other.AddRewardOnCar(carTransform, -1f);
            }
        }
    }
```

La funzione `CarThroughCheckpoint` controlla che il checkpoint attraversato dalla macchina sia quello giusto. Il metodo `CarThroughCheckpoint` prende in input due parametri: `checkpointSingle` che è il checkpoint attraversato e `carTransform` che è la macchina. Il metodo è utilizzato in particolare durante il training, in modo da poter assegnare reward adeguati all'agente. In primis vengono trovate l'indice del checkpoint successivo per l'agente corrente, utilizzando la lista `nextCheckpointSingleIndexList` e la lista `carTransformList`. Successivamente, si verifica se l'indice del checkpoint attualmente attraversato sia uguale all'indice del checkpoint successivo. In caso affermativo significa che il checkpoint è stato attraversato in ordine corretto, quindi viene aggiornato l'indice del prossimo checkpoint e viene assegnato un reward positivo (1) all'agente. In caso contrario, significa che il checkpoint non è stato attraversato in ordine corretto, quindi viene assegnato un reward negativo (-1) all'agente.

In sintesi, questo codice serve a assicurare che l'agente stia seguendo il percorso corretto e assegnare un reward adeguato in base alle azioni intraprese dall'agente, incentivandolo o punendolo a seconda del suo comportamento.

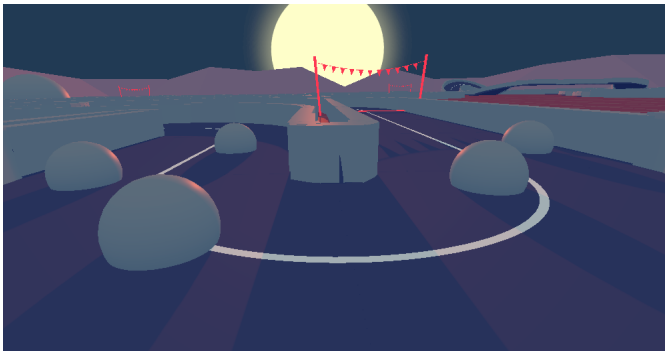
Vogliamo usare più macchine quindi memorizziamo gli indici dei checkpoint attraversati per ogni macchina in una lista. E per le dimensioni definiamo un elenco di transforms che saranno il `carTransformList`, mettiamo `[SerializeField]` così possiamo settarlo nell'editor.

I punti di controllo della pista vengono resettati utilizzando il metodo `ResetCheckpoint`. Questo metodo semplicemente resetta tutti i checkpoint per poter riuscire poi ad effettuare altri addestramenti ripartendo da zero e quindi nuove esecuzioni senza tener conto di quelle già effettuate.

```
public void ResetCheckpoint(Transform carTransform)
    {
        nextCheckpointSingleIndexList[carTransformList.IndexOf(carTransform)] = 0;
    }
```

4.6 Ostacoli

Gli ostacoli sono componenti di scena molto simili ai muri, poiché si comportano come essi solo che hanno una forma e una posizione differente. Ovviamente l'agente dovrà, in questo caso, non solo evitare i muri ma anche gli ostacoli che si troveranno sparsi nel tracciato. Anche in questo caso ci sarà lo stesso controllo effettuato per i muri.



5 Tecnologie e strumenti utilizzati

Introduciamo le tecnologie utilizzate per l'implementazione del progetto, iniziando dall'ambiente di modellazione della gioco, passando per le tecnologie di machine learning utilizzate, linguaggi di programmazione utilizzati e supporti ausiliari.

5.1 Unity 3D

Unity consente agli sviluppatori di creare giochi 3D e 2D, nonché realtà virtuale e aumentata, simulazioni, visualizzazioni e applicazioni interattive. Il motore di gioco di Unity è altamente personalizzabile e offre una vasta gamma di funzionalità, tra cui rendering, fisica, illuminazione, animazioni e audio.

Il sistema di asset management di Unity consente agli sviluppatori di gestire facilmente i loro asset, tra cui modelli 3D, texture, suoni e script, rendendo più semplice la creazione e la manutenzione dei progetti. Unity è una piattaforma di sviluppo di giochi e simulazioni interattive che permette agli sviluppatori di creare contenuti tridimensionali (e bidimensionali) utilizzando modelli predefiniti chiamati `GameObject`. Con Unity, gli sviluppatori possono definire la posizione, la rotazione e la scala di ciascun `GameObject` nello spazio virtuale, e possono anche personalizzare ulteriormente i GameObject utilizzando texture, materiali, script e componenti di fisica. In sintesi, Unity è un ambiente di sviluppo visivo con una vasta gamma di funzionalità, che consente di creare giochi e simulazioni interattive per diverse piattaforme.

5.2 Unity MLAgents

ML-Agents è un plugin per Unity che consente ai giochi e alle simulazioni di fungere da ambienti per la formazione di agenti intelligenti. Consente agli sviluppatori di addestrare modelli di apprendimento automatico in un mondo virtuale, utilizzando una varietà di algoritmi di apprendimento per rinforzo. Il plugin fornisce un insieme di strumenti e API per creare e controllare gli agenti, nonché per comunicare con i modelli di apprendimento automatico. Ciò consente la creazione di comportamenti AI sofisticati in un gioco o simulazione, come la capacità di navigare in ambienti complessi, risolvere enigmi o giocare in modo competitivo contro altri agenti.

`MLAgents` consente agli sviluppatori di creare agenti di apprendimento automatico all'interno dei loro giochi e simulazioni, utilizzando algoritmi di apprendimento come il Reinforcement Learning.

Con `MLAgents`, gli sviluppatori possono creare agenti che possono interagire con l'ambiente del gioco, apprendere dalle esperienze e migliorare le loro prestazioni nel tempo. Gli sviluppatori possono anche creare ambienti di addestramento personalizzati per i loro agenti e utilizzare il toolkit per effettuare il training degli agenti su un cluster di calcolo distribuito.

`ML-Agents` include anche metodi specifici per l'interazione tra l'agente e l'ambiente di gioco.

Ecco alcuni dei metodi più rilevanti:

- `OnActionReceived`. Questo metodo viene chiamato ogni volta che l'agente riceve un'azione dal sistema di apprendimento automatico. Gli sviluppatori possono utilizzare questo metodo per eseguire azioni specifiche all'interno del loro ambiente di gioco in base alle azioni ricevute dall'agente.
- `CollectObservations`. Questo metodo viene utilizzato per raccogliere le osservazioni dell'ambiente che l'agente utilizza per prendere decisioni. Gli sviluppatori possono utilizzare questo metodo per raccogliere informazioni come la posizione dell'agente, la velocità e la distanza dai checkpoints.
- `OnEpisodeBegin`. Questo metodo viene chiamato all'inizio di ogni episodio di gioco, gli sviluppatori possono utilizzare questo metodo per eseguire azioni specifiche, come il reset dell'ambiente o l'inizializzazione delle variabili.
- `AddReward`. Questo metodo viene utilizzato per assegnare una ricompensa all'agente in base al suo comportamento. Gli sviluppatori possono utilizzare questo metodo per premiare o penalizzare l'agente in base alle azioni che ha intrapreso.
- `EndEpisode`. Permette la conclusione dell'episodio e si occupa del reset dell'agente, richiamando a sua volta il metodo `OnEpisodeBegin`.

5.3 TensorFlow

TensorFlow è una libreria open-source per l'apprendimento automatico sviluppata dal team di Google Brain. Fornisce un insieme completo di strumenti per la costruzione e il deploy di modelli di apprendimento automatico, compreso il supporto per l'apprendimento profondo e le reti neurali. TensorFlow è progettato per essere flessibile ed efficiente, permettendo agli sviluppatori di creare e eseguire facilmente modelli di apprendimento automatico su una varietà di piattaforme, dai desktop e dai server ai dispositivi mobili e edge. Includi anche una grande collezione di modelli pre-costruiti e pre-addestrati che possono essere utilizzati come punto di partenza per sviluppare modelli personalizzati. Inoltre, TensorFlow fornisce una varietà di strumenti di visualizzazione per aiutare gli sviluppatori a capire meglio e a risolvere i problemi dei loro modelli.

5.4 C#

C# è un linguaggio di programmazione che viene comunemente utilizzato nello sviluppo di videogiochi e media interattivi. È uno dei principali linguaggi di programmazione supportati da Unity, un popolare motore di gioco e piattaforma di sviluppo. Inoltre è un linguaggio di programmazione orientato agli oggetti, il che significa che si basa sul concetto di "oggetti" che hanno proprietà e metodi che possono essere manipolati e interagiti. C# è anche un linguaggio fortemente tipizzato, il che significa che le variabili devono essere dichiarate con un tipo di dati specifico, come int o string.

In Unity, gli sviluppatori possono utilizzare C# per creare script che controllano il comportamento degli oggetti di gioco, come personaggi, veicoli e ambienti. Questi script possono essere allegati agli oggetti nell'editor di Unity e possono essere utilizzati per controllare cose come il movimento, la fisica, l'animazione e l'input dell'utente. Inoltre, C# può anche essere utilizzato per creare strumenti personalizzati e plugin per Unity, che possono aiutare a semplificare il processo di sviluppo e aggiungere nuove funzionalità al motore.

5.5 PyTorch

PyTorch è una libreria open-source di apprendimento automatico basata sulla libreria Torch. È principalmente sviluppata dal gruppo di ricerca sull'intelligenza artificiale di Facebook ed è ampiamente utilizzata per applicazioni come il computer vision e il natural language processing. PyTorch è progettato per essere facile da usare ed intuitivo, rendendolo una scelta popolare per la ricerca e lo sviluppo.

PyTorch include una libreria di differenziazione automatica incorporata, che rende facile eseguire i calcoli del gradiente per la retropropagazione, che è cruciale per l'addestramento dei modelli di apprendimento automatico.

6 Algoritmo di machine learning

L'algoritmo di machine learning che utilizziamo si chiama `KartClassicAgent` ed è uno script che estende la classe `Agent` di Unity `MLAgents` associato al kart che rappresenta il nostro agente. Andiamo a vedere nel dettaglio i metodi del nostro algoritmo che definiscono come l'agente osserva l'ambiente e associa dei reward alle azioni che ha compiuto.

6.1 Metodi più rilevanti

- `OnEpisodeBegin`. Inizializzazione della macchina, effettuata all'inizio di ogni episodio. Gli episodi vengono resettati dopo un certo numero di step che vengono settati su Unity e che ci permettono di dare all'agente un numero di azioni, entro le quali esso dovrà riuscire ad apprendere in modo corretto i tracciati.
- `OnActionReceived` Consente l'interpretazione dell'input (di valore discreto) che l'agente vuole dare alla macchina, passandolo allo script di movimento.

  primo vettore => sinistra/dritto/destra; secondo vettore => accelera/frena
- `AddRewardOnCar`. Segna il valore adeguato all'azione scelta dall'agente
- `CollectObservations`. Effettua la raccolta delle osservazioni che l'agente ha sull'ambiente. In questo caso vediamo quanto correttamente la macchina è rivolta verso il prossimo checkpoint
- `OnTriggerEnter` e `OnTriggerStay`. Entrambe utilizzate per il controllo delle collisioni tra la macchina e il muro che delimita il tracciato.
- `OnCollisionEnter` e `OnCollisionStay`. Entrambe utilizzate per il controllo delle collisioni tra la macchina e l'oggetto chiamato "Sphere", che rappresenta un ostacolo del tracciato WindingTrack.

6.1.1 OnEpisodeBegin

```
void Awake()
   {
       arcade = GetComponent<ArcadeKart>();
   }

//inizializzazione della macchina, effettuata all'inizio di ogni episodio

//(gli episodi ricominciano quando il numero di step disponibili raggiunge
il limite impostato nell'editor di Unity)
   public override void OnEpisodeBegin()
   {
       transform.position = spawnCarPos;
       transform.forward = spawnCarFor;

       trackCheckpoints.ResetCheckpoint(capsule);

       arcade.Rigidbody.velocity = default;

       m_Acceleration = false;
       m_Brake = false;
       m_Steering = 0f;
   }
```

Questo codice mostra la funzione `OnEpisodeBegin`, che viene eseguita all'inizio di ogni episodio in un gioco Unity. La funzione ha alcune operazioni che vengono eseguite per inizializzare la macchina per un nuovo episodio. In particolare abbiamo:

- la posizione e la direzione della macchina vengono impostate sui valori predefiniti (`spawnCarPos` e `spawnCarFor`) che rappresentano il punto di spawn iniziale
- la velocità del `rigidbody` arcade viene impostata sui valori di default
- le variabili `m_Acceleration`, `m_Brake` e `m_Steering` vengono impostate sui valori predefiniti (rispettivamente: `false`, `false` e `0`)

6.1.2 OnActionReceived

```
//consente l'interpretazione dell'input (di valore discreto) che
 l'agente vuole dare alla macchina,
   //passandolo allo script di movimento
   // primo vettore => sinistra/dritto/destra
   // secondo vettore => accelera/frena
   public override void OnActionReceived(ActionBuffers actions)
   {
       base.OnActionReceived(actions);
       InterpretDiscreteActions(actions);

       other.FixedUpdateForML(this.GenerateInput());
   }
```

Questo codice mostra la funzione `OnActionReceived(ActionBuffers actions)` che viene chiamata quando l'agente riceve un input (di valore discreto) per controllare la macchina, le azioni possibili sono frenata, accelerazione e sterzata.

La funzione ha due compiti principali:

- *Interpretazione dell'input.* La funzione chiama un altro metodo chiamato `InterpretDiscreteActions(actions)` per interpretare l'input ricevuto e utilizzarlo per controllare la macchina. Il primo vettore dell'input viene utilizzato per controllare la direzione della macchina (sinistra, dritto o destra) e il secondo vettore per controllare l'accelerazione o il freno.
- *Passaggio dell'input allo script di movimento.* La funzione chiama il metodo `FixedUpdateForML` su un oggetto chiamato `other` passando un input generato dalla funzione `GenerateInput`, in modo che lo script di movimento possa utilizzare l'input per aggiornare la posizione e la velocità della macchina.

In sintesi questo codice gestisce l'input ricevuto dall'agente e lo passa allo script di movimento della macchina per gestire la sua posizione e velocità.

6.1.3 AddRewardOnCar

```
//assegna il valore adeguato all'azione scelta dall'agente
    public void AddRewardOnCar(Transform carTransform, float quantity)
    {
        if (carTransform == capsule)
        {
            AddReward(quantity);
        }
    }
```

Questa funzione `AddRewardOnCar(Transform carTransform, float quantity)` è utilizzata per assegnare una ricompensa all'agente per un'azione specifica. La funzione accetta due parametri:

- `carTransform`. Rappresenta l'oggetto Transform del kart su cui si vuole assegnare la ricompensa.
- `quantity`. rappresenta il valore della ricompensa che si vuole assegnare all'agente.

La funzione utilizza un controllo condizionale if per verificare se il Transform della macchina passata come parametro corrisponde alla capsule della macchina che corrisponde al `KartBouncingCapsule`. Se i due `Transform` corrispondono, la funzione chiama un metodo di base `AddReward(quantity)` che è una funzione ereditata da Agent e serve per assegnare la ricompensa specificata al valore dell'agente. In caso contrario non assegna alcuna ricompensa. In sintesi questo codice assegna una ricompensa all'agente solo se la macchina passata come parametro è quella giusta, altrimenti non fa nulla.

6.1.4 CollectObservations

```
//effettua la raccolta delle osservazioni che l'agente ha sull'ambiente

//in questo caso vediamo quanto correttamente la macchina è rivolta verso il prossimo checkpoint
    public override void CollectObservations(VectorSensor sensor)
    {
        Vector3 checkpointForward = trackCheckpoints.GetNextCheckpoint(capsule);
        float directionDot = Vector3.Dot(capsule.forward, checkpointForward);
        sensor.AddObservation(directionDot);
    }
```

Questa funzione `CollectObservations(VectorSensor sensor)` è utilizzata per raccogliere le osservazioni dell'agente sull'ambiente circostante. La funzione accetta un parametro `VectorSensor sensor`, questo è un sensore vettoriale che viene utilizzato per raccogliere le osservazioni. Utilizziamo un oggetto chiamato `trackCheckpoints` per ottenere il prossimo checkpoint (chiamando il metodo `GetNextCheckpoint(capsule)`) e calcola il prodotto scalare tra la direzione in cui si trova la macchina (capsule.forward) e la direzione verso il prossimo checkpoint (`checkpointForward`). Il prodotto scalare è un numero compreso tra `-1` e `1` che indica quanto la macchina è rivolta verso il prossimo checkpoint.

Un valore di `1` indica che la macchina è perfettamente allineata con il checkpoint, un valore di `-1` indica che la macchina è perfettamente allineata nella direzione opposta, e un valore di 0 indica che la macchina è perpendicolare al checkpoint. Infine, la funzione utilizza il sensore per aggiungere l'osservazione del prodotto scalare al sensore. L'agente può utilizzare questa osservazione per decidere la prossima azione da intraprendere. In sintesi lo script raccoglie l'osservazione sulla posizione della macchina rispetto al prossimo checkpoint, utilizzando il prodotto scalare tra la direzione della macchina e la direzione del checkpoint e lo aggiunge al sensore.

6.1.5 OnTriggerEnter e OnTriggerStay

```
//controllo delle collisioni su una qualsiasi parte del muro che delinea il tracciato
    private void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Wall")
        {
            AddReward(-0.3f);
        }
    }

    private void OnTriggerStay(Collider other)
    {
        if (other.tag == "Wall")
        {
            AddReward(-0.005f);
        }

    }
```

Questo codice mostra due funzioni: `OnTriggerEnter(Collider other)` e `OnTriggerStay(Collider other)` entrambe utilizzate per il controllo delle collisioni tra la macchina e i muri identificati in Unity da un tag `Wall`. La funzione `OnTriggerEnter(Collider other)` viene chiamata quando la macchina entra in contatto con il muro e la funzione `OnTriggerStay(Collider other)` viene chiamata mentre la macchina rimane in contatto con il muro. Entrambe le funzioni utilizzano un controllo condizionale if per verificare se l'oggetto con cui la macchina collide ha il tag `Wall` e dunque è un muro o meno. Se la macchina collide con il muro assegna una penalità di `-0.3f` all'agente utilizzando il metodo `AddReward(-0.3f)`. La funzione `OnTriggerStay(Collider other)` assegna una penalità di `-0.005f` all'agente mentre la macchina è in contatto con il muro utilizzando il metodo `AddReward(-0.005f)`. In sintesi lo script assegna una penalità all'agente quando la macchina collide con il muro e mentre la macchina è in contatto con il muro.

6.1.6 OnCollisionEnter e OnCollisionStay

```
//controllo delle collisioni sulla sfera (ostacolo del tracciato 4)
    private void OnCollisionEnter(Collision other)
    {
        if (other.gameObject.name == "Sphere")
        {
            AddReward(-0.3f);
        }
    }

    private void OnCollisionStay(Collision other)
    {
        if (other.gameObject.name == "Sphere")
        {
            AddReward(-0.005f);
        }

    }
```

Questo codice mostra due funzioni: `OnCollisionEnter(Collision other)` e `OnCollisionStay(Collision other)` entrambe utilizzate per il controllo delle collisioni tra la macchina e un oggetto chiamato "Sphere", che rappresenta un ostacolo del tracciato *WindingTrack*. La funzione `OnCollisionEnter(Collision other)` viene chiamata quando la macchina entra in contatto con un oggetto "Sphere" e la funzione `OnCollisionStay(Collision other)` viene chiamata mentre la macchina rimane in contatto con l'oggetto "Sphere". Entrambe le funzioni utilizzano un controllo per verificare se l'oggetto, con cui la macchina collide, è un oggetto "Sphere" utilizzando il nome `gameObject.name` dell'oggetto. In questo caso la funzione `OnCollisionEnter(Collision other)` assegna una penalità di `-0.3f` all'agente utilizzando il metodo `AddReward(-0.3f)`. La funzione `OnCollisionStay(Collision other)` assegna una penalità di `-0.005f` all'agente mentre la macchina è in contatto con la sfera utilizzando il metodo `AddReward(-0.005f)`. In sintesi questo script assegna una penalità all'agente quando la macchina collide con la sfera e mentre la macchina è in contatto con la sfera.

6.2 Conclusioni

L'algoritmo di Reinforcement Learning che abbiamo utilizzato per addestrare il kart, è stato sviluppato attraverso un processo di analisi e sperimentazione. Inizialmente, abbiamo esaminato attentamente l'ambiente in cui il kart si sarebbe mosso e i modi in cui avrebbe potuto muoversi. Successivamente, abbiamo compreso come utilizzare i metodi forniti dal framework `MLAgents` per implementare un algoritmo di base. Attraverso il metodo empirico, abbiamo continuato a testare e migliorare il codice, eseguendo più volte l'algoritmo e ottenendo miglioramenti ad ogni iterazione. L'algoritmo ci ha fatto capire che ogni agente necessita di effettuare molte osservazioni e azioni prima di imparare a risolvere un tracciato in modo corretto. Con il passare del tempo, l'agente riusciva a trovare sempre più la soluzione ottimale per il problema in questione. In definitiva, l'utilizzo del Reinforcement Learning si è rivelato essere la scelta vincente per addestrare il kart a percorrere tracciati in modo corretto.