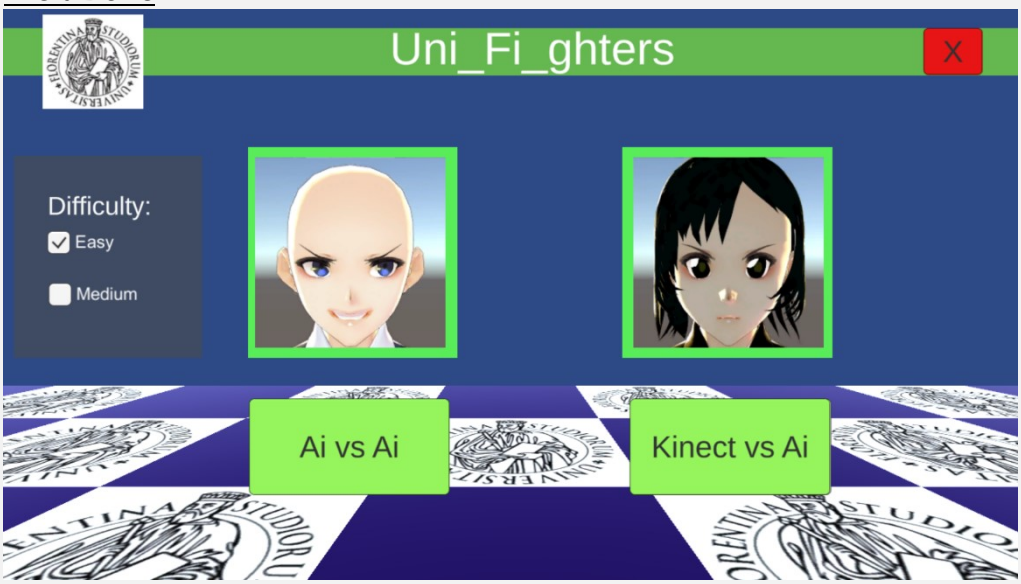
UNIVERSITÀ
DEGLI STUDI
FIRENZE

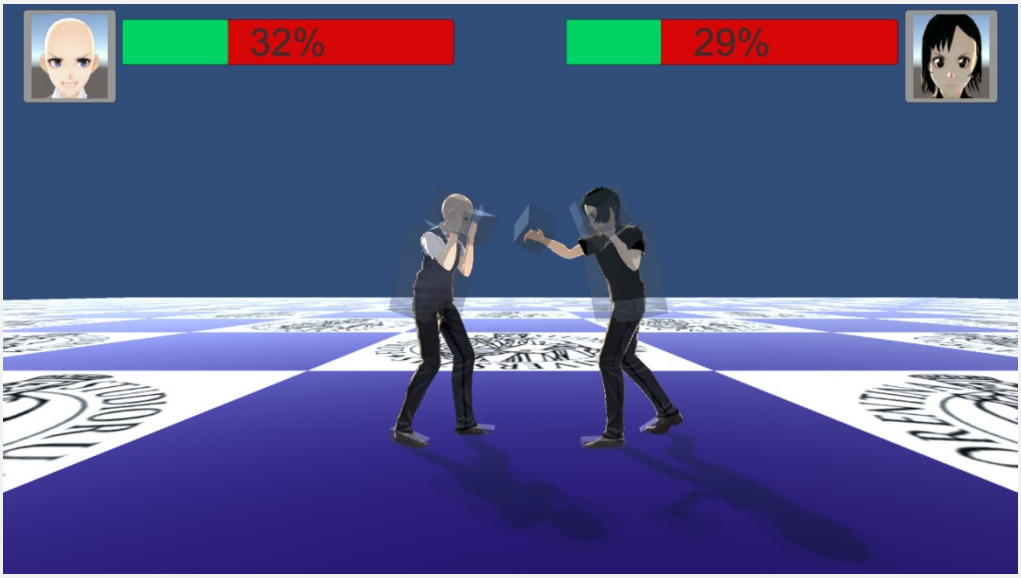
Progetto Computer Graphics
Uni_Fi_ghters

Plesoiu Silviu Robert

5424944

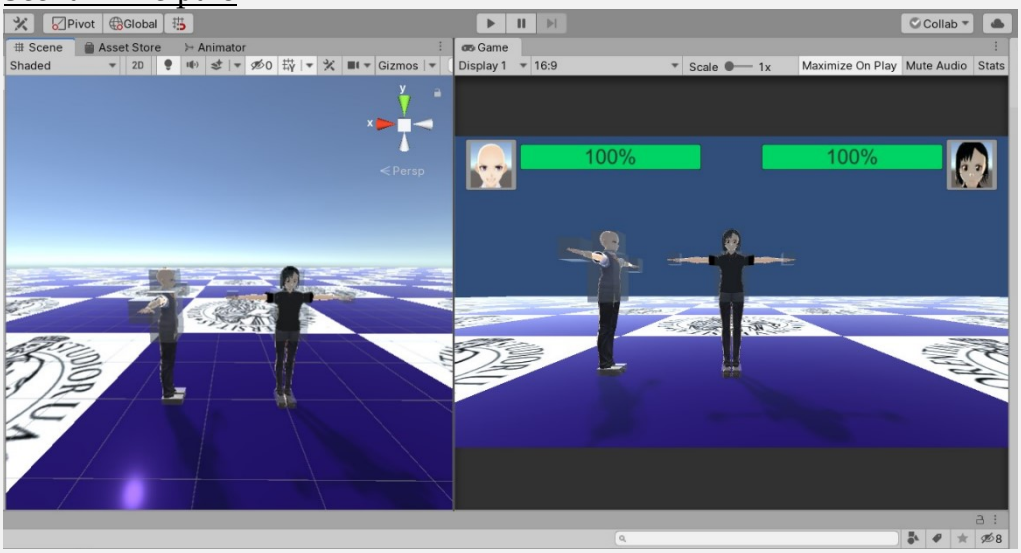
Prefazione



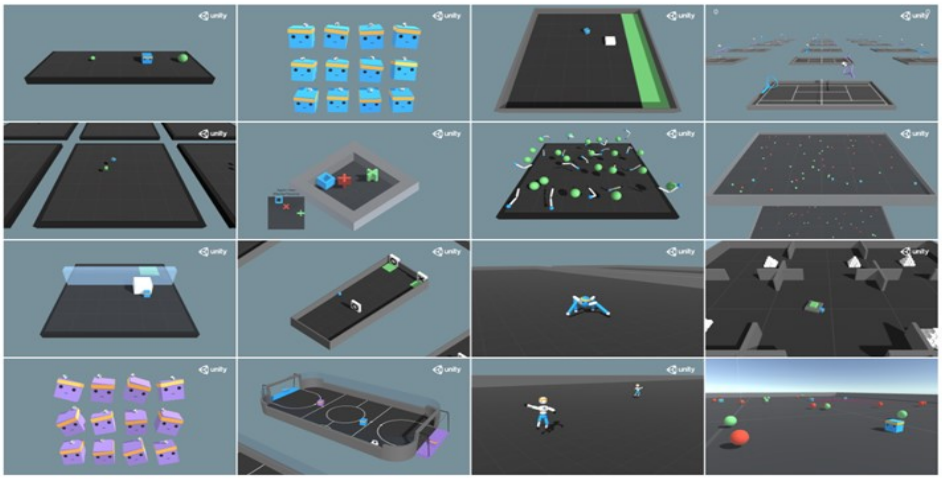


Il progetto si pone l'obiettivo di implementare un piccolo gioco che prevede l'utilizzo del Kinect dove il giocatore dovra' affrontare, diciamo in "combattimento" un avversario guidato da una intelligenza artificiale. Il progetto si pone l'obiettivo di essere il piu' minimalista possibile in ambito Ui, in quanto l'obiettivo e' quello di mostrare delle tecniche base che possano essere prese come spunto per progetti futuri.
Demo Youtube del progetto, scontro IA vs IA:
https://www.youtube.com/watch?v=zcIz8NV7MWc

Scena Principale



Il progetto e' stato sviluppato su Unity utilizzando come linguaggio di programmazione C#. Nella scena principale del progetto abbiamo degli elementi UI che sono le barre della vita dei giocatori, un pavimento che ricorda una Scacchiera (dove i tasselli sono il logo della nostra universita' e dei tatami blu) e gli avatar dei giocatori.
L'avatar a Sinistra e' controllato da una Intelligenza artificiale mentre l'avatar a destra e' controllato da un giocatore attraverso il Kinect.
L'intelligenza artificiale e' stata sviluppata grazie ad un plug-in di Unity chiamato "ml-agents". https://github.com/Unity-Technologies/ml-agents

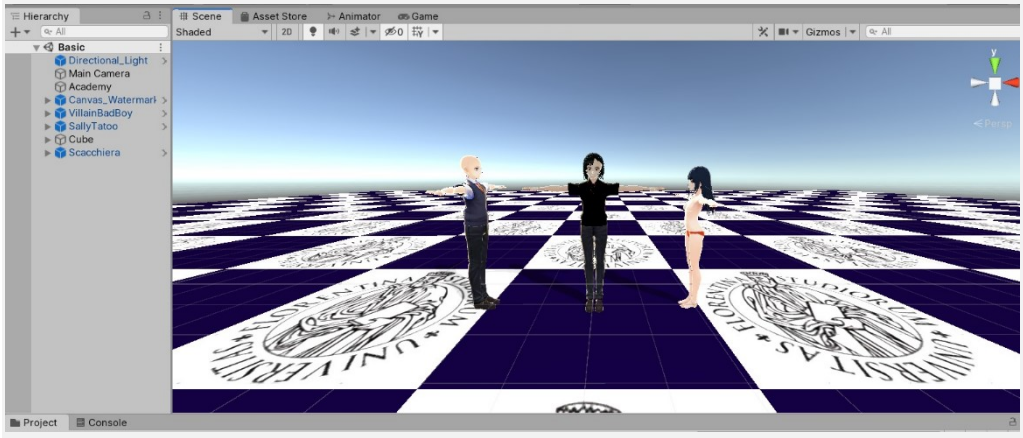


SetUp e Tutorial Training

ML-Agents fornisce un modo per addestrare degli agenti a compiere una determinata Task. Nel nostro caso l'agente dovra' essere addestrato ad effettuare un "combattimento".
Ma prima di entrare nel vivo del progetto facciamo un passo indietro mostrando un esempio piu' semplice per entrare nelle meccaniche del progetto.

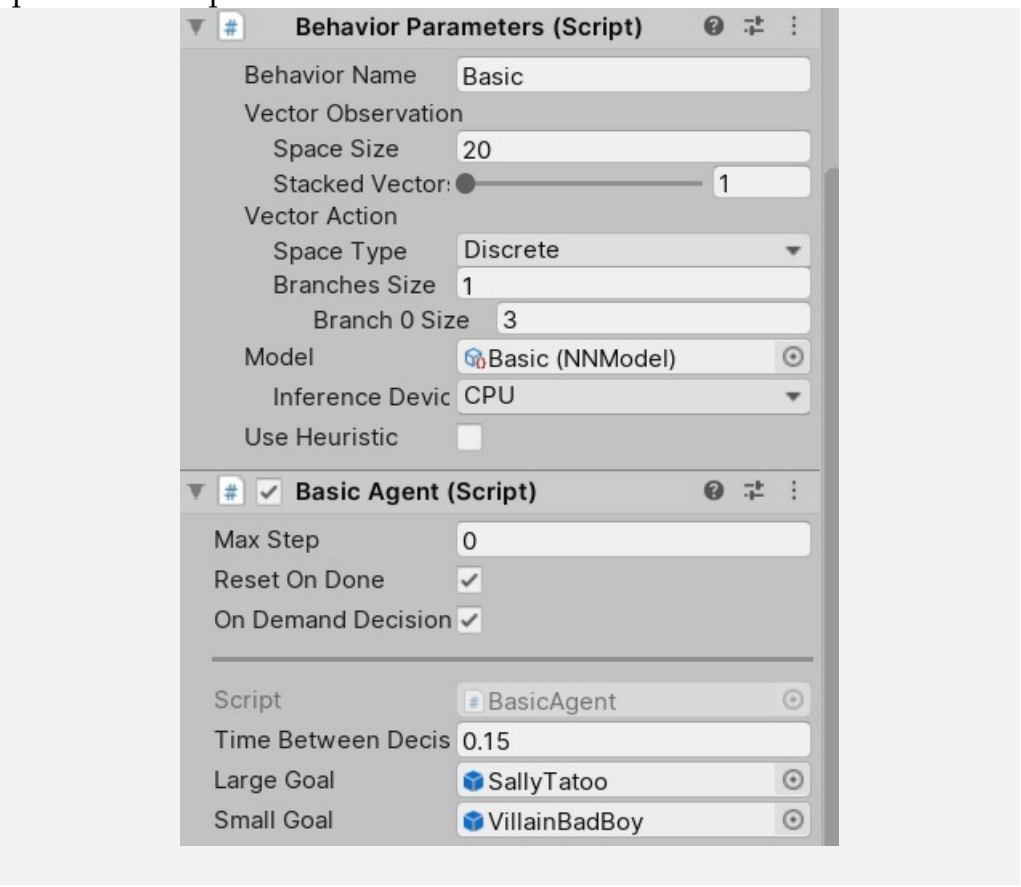
Per il setUp di Unity Ml-Agents guardare i pdf tutorial al set_up.
Ma prima di fare il setUp continuare con la lettura per avere una visione migliore del quadro generale.

Ml-Agent Basic



Basic e' un scena all'interno del progetto il cui scopo e' quello di fare da tutorial per poter utilizzare e fare delle modifiche a nostro piacimento con il tool Ml-Agents.
Nella scena abbiamo 3 modelli 3D: l'agente, un avatar maschile ed uno femminile.
L'agente ha tre azioni a disposizione: muoversi a destra, stare fermo o muoversi a sinistra.
Task: L'agente deve imparare a muoversi esclusivamente verso l'avatar femminile.
Vediamo come si fa a dire all'Agente quali azioni esso puo' eseguire e su cosa si basa l'apprendimento.

Se clikkate nell'oggeto Cube della scena in Unity, nell'inspector vedrete questi due scripts



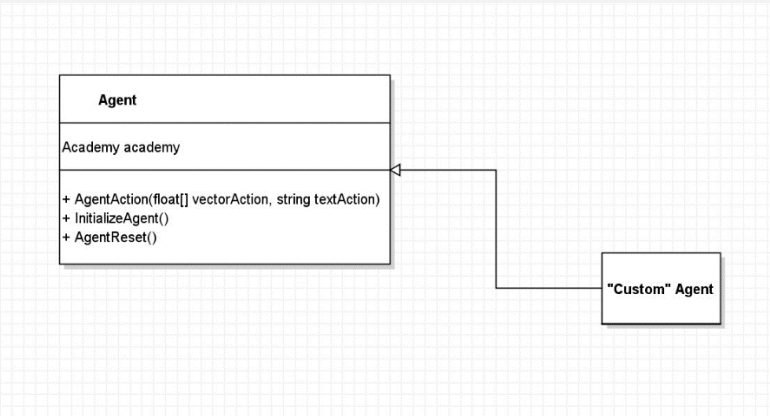
Lo script Behavior Parameters e' uno script che non andiamo ad implementare noi, ma semplicemente ci viene fornito (si puo' modificare ma non conviene per task semplici..) mentre lo Script Basic Agent lo andiamo ad implementare noi da zero.
Come funziona? Dall'interfaccia di Unity vanno settati alcuni parametri come si puo' vedere. A noi interessa il campo Space Type del primo script (cercate di guardare l'immagine mentre leggete o vi perdete ahaha), come vedete e' settato su Discrete. Se ci clikkate vi da un'altra opzione che e' "Continuos", quest'altra si basa su calcoli tensoriali e serve se l'agente deve diciamo "guardare" l'ambiente che lo circonda, ma richiede un certo costo computazionale.
Sotto "Space Type" leggiamo la label "Branches Size". "Branches Size" indica la tipologia di azioni che l'agente puo' eseguire, una variabile insomma, mentre "Branch0 Size" indica la cardinalita' del dominio di tale variabile.

D'ora in poi diro' "Branch n Size" per generalizzare.
Per esempio nel nostro caso abbiamo settato "Branches Size" a 1 e "Branch n Size" a 3. "Branches Size" settato ad uno equivale a dire che l'agente puo' svolgere una sola azione, che nel nostro caso e' quella di spostarsi. Mentre "Branch n Size" settato a 3, indica la cardinalita' del dominio di tale azione. Il dominio dell'azione "spostarsi" e' fatto cosi', DominioSpostarsi = {stai fermo, vai a sinistra, vai a destra }.
Se oltre a spostarsi mi sarebbe piaciuto che l'agente eseguisse un'altra azione avrei dovuto impostare "Branch Size" a 2 (fatto cio' sotto "Branch n Size" comparira' una nuova label "Branch n1 Size").
Facciamo finta voglia che l'agente emetta dei suoni mentre si muove, t.c.: Emettere Suoni = {fai silenzio, fai rumore buffo, canta, fai uno schiamazzo di pericolo }, avrei dovuto impostare "Branch n1 Size" = 4.
Quindi attraverso lo script Behavior Parameters setto il numero di azioni (che possiamo pensare come variabili) e poi per ciascuna azione imposto la cardinalita' del dominio di tale Azione.

Abbiamo parlato di Domini delle azioni, e abbiamo settato la dimensione di tale Domini attraverso Unity, ma come faccio effettivamente a mappare tali domini nelle istanze da noi desiderate???

Questo va fatto a livello di codice!

Creare una classe che estende lo script fornito da Ml-Agent "Agent" ed implementare tramite override i metodi mostrati nell'Uml




```
public override void AgentAction(float[] vectorAction,string textAction)
    {
        var movement = (int)vectorAction[0];
        var direction=0;
        switch (movement)
        {
            case 1:
                direction = -1;
                break;
            case 2:
                direction = 1;
                break;
        }

        m_Position += direction;
        if (m_Position < m_MinPosition) { m_Position = m_MinPosition; }
        if (m_Position > m_MaxPosition) { m_Position = m_MaxPosition; }

        transform.position = new Vector3(m_Position -10f, 0f, 0f);
        AddReward(-0.01f);

        if (m_Position == m_SmallGoalPosition)
        {
            Done();
            AddReward(0.1f);
        }

        if (m_Position == m_LargeGoalPosition)
        {
            Done();
            AddReward(1f);
        }
}
```

Nel metodo AgentAction(`float[] vectorAction`) diciamo all'Agente cosa deve fare.
La dimensione di vectorAction equivale al numero settato su "Branches Size".
Nel caso specifico riportato nel codice di prima come potete vedere dal codice andiamo a fare uno Switch case su il valore contenuto nel primo elemento del vettore.
I valori che questi puo' assumere dipendono dal numero che noi abbiamo settato in "Branch n Size" (vi ricordate? n=0) in un range fatto cosi' [0, Branch n Size -1] appartenente all'insieme dei numeri Naturali (potete discutere se 0 appartenga o no a N e infodermi la vostra filosofia Matematica. Ma era per dire che siamo nel caso discreto).
Il nostro dominio dell'azione "spostarsi" e' fatto nel modo seguente

DominioSpostarsi = { stai fermo, vai a sinistra, vai a destra }

con il costrutto Switch case andiamo effettivamente a mappare tali valori in una corrispondenza biunivoca:
0 = stai fermo,
1 = vai a sinistra,
2 = vai destra.
A livello di codice ho una variabile chiamata "direction" che vado a sommare alla coordinata "X" dell'agente.
Per andare a sinistra direction vale -1, per andare a destra la pongo uguale a 1, e 0 per stare fermi.

```
m_Position += direction;
transform.position = new Vector3(m_Position -10f, 0f, 0f);
```

Facciamo un RECAP veloce.
Se voglio creare un agente customizzato, gli va "dato" tramite l'inspector di Unity lo script "Behavior Parameters" e creare uno script nuovo che estenda la classe Agent.
Per semplicita' lavoriamo nel caso discreto. Impostiamo il campo Branch Size con il numero di azioni concesse all'Agente.
Per esempio nel progetto setto Branch Size a due perche' l'agente deve imparare a muoversi mantenendo una giusta distanza dall'avversario come prima azione, e imparare a portare i giusti colpi come seconda azione. Una volta fissato il numero di azioni va fissata la cardinalita' del dominio di ciascuna azione (Le azioni sono variabili).

Come in Basic l'agente che combatte potra' stare fermo, fare un passo avanti e un passo indietro. Quindi cardinalita' del dominio della prima azione =3. Per le mosse di combattimento l'agente potra' non so difendersi, tirare un pugno, tirare un calcio laterale, tirare un calcio circolare. Quindi cardinalita' del dominio dell'azione combattere uguale a 4. PAM!!
Ora che abbiamo definito numero di azioni e cardinalita' del dominio di ogni singola azione, a questo punto dobbiamo scrivere il codice.
Andiamo nella classe che estende la classe Agente, e implementiamo facendo un override il metodo AgentAction(`float[] vectorAction`).
AZIONE 1
vectorAction[0]==0? Stai fermo
vectorAction[0]==1? Fai un passo avanti
vectorAction[0]==2? Fai un passo in dietro.

AZIONE 2
vectorAction[1]==0? Difendi
vectorAction[1]==1? Tira un pugno
vectorAction[2]==2? Tira un calcio laterale
vectorAction[3]==3? Tira un calcio circolare

(ora magari nel codice 0= pugno e difendi =3 non so spero si capisca l'idea che c'e' dietro.).

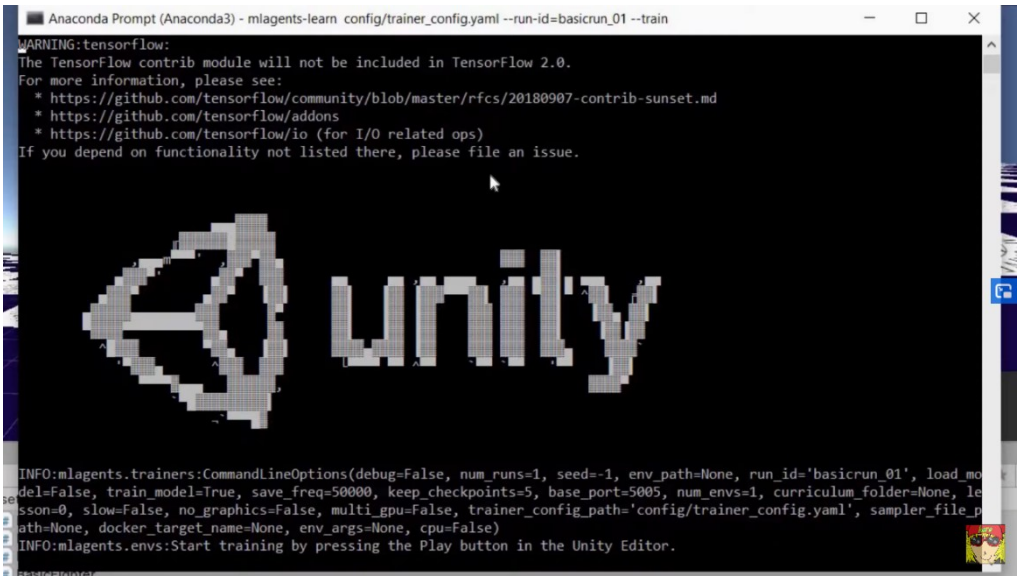
Un altro dubbio che non abbiamo risolto e' chi ci fornisce l'input "`float[] vectorAction`"?. A questo ci pensano
"Two deep reinforcement learning algorithms, Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC)"
(I link portano ad una spiegazione dei singoli algoritmi)

All'inizio i valori contenuti nel vettore di float vengono generati in maniera casuale. L'agente e' assistito all'apprendimento attraverso dei Rewards come si puo' vedere dal codice.
ML agents suggerische che i Rewards dovrebbero stare in un range del tipo [-1, 1].
In Basic si da un Reward di 1 quando l'agente collide con l'avatar Femminile e un Reward di 0.1 quando collide con l'avatar maschile.
Quando l'agente porta a termine una Task (fallimento o riuscita) va invocato il metodo Done( ) per dire all'algoritmo che la Task e' stata

eseguita e di provare a dare nuovi valori al vettore vectorAction in base ai Rewards accumulati.

Per fare il training si necessita di un oggetto di tipo "Academy" nella scena. Basta creare un oggetto empy nella scena e affibiargli lo script Academy.
Ora che abbiamo capito, non tanto come funziona l'algoritmo.. ma come creare un nostro Agente personalizzato possiamo iniziare la fase di Training.
Se si e' eseguito il mio tutorial su come fare il setUp di Ml-Agents procedere nel seguente modo.
Aprire il promp di Anaconda ed eseguire i seguenti comandi.

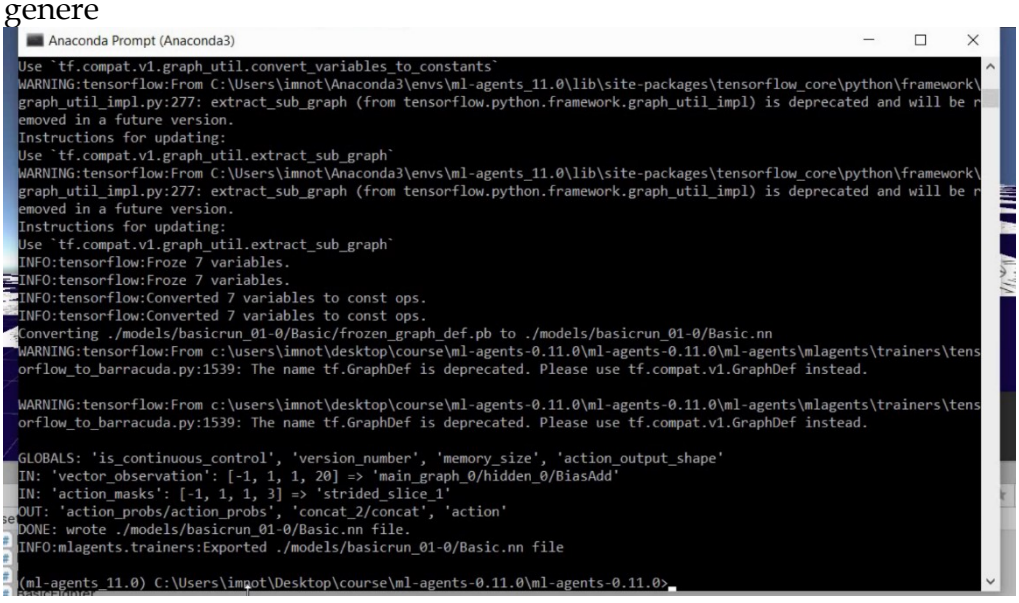
```
conda activate ml –agents_0.11.0  **commando che attiva l'enviroment

cd C:\Users\imnot\Desktop\course\ml-agents-0.11.0\ml-agents-0.11.0  **cambiate cartella dove avete messo l'enviroment creato durante il setUp.

mlagents-learn config/trainer_config.yaml --run-id=basicrun_01 –train

**comando per fare partire il trainig
```



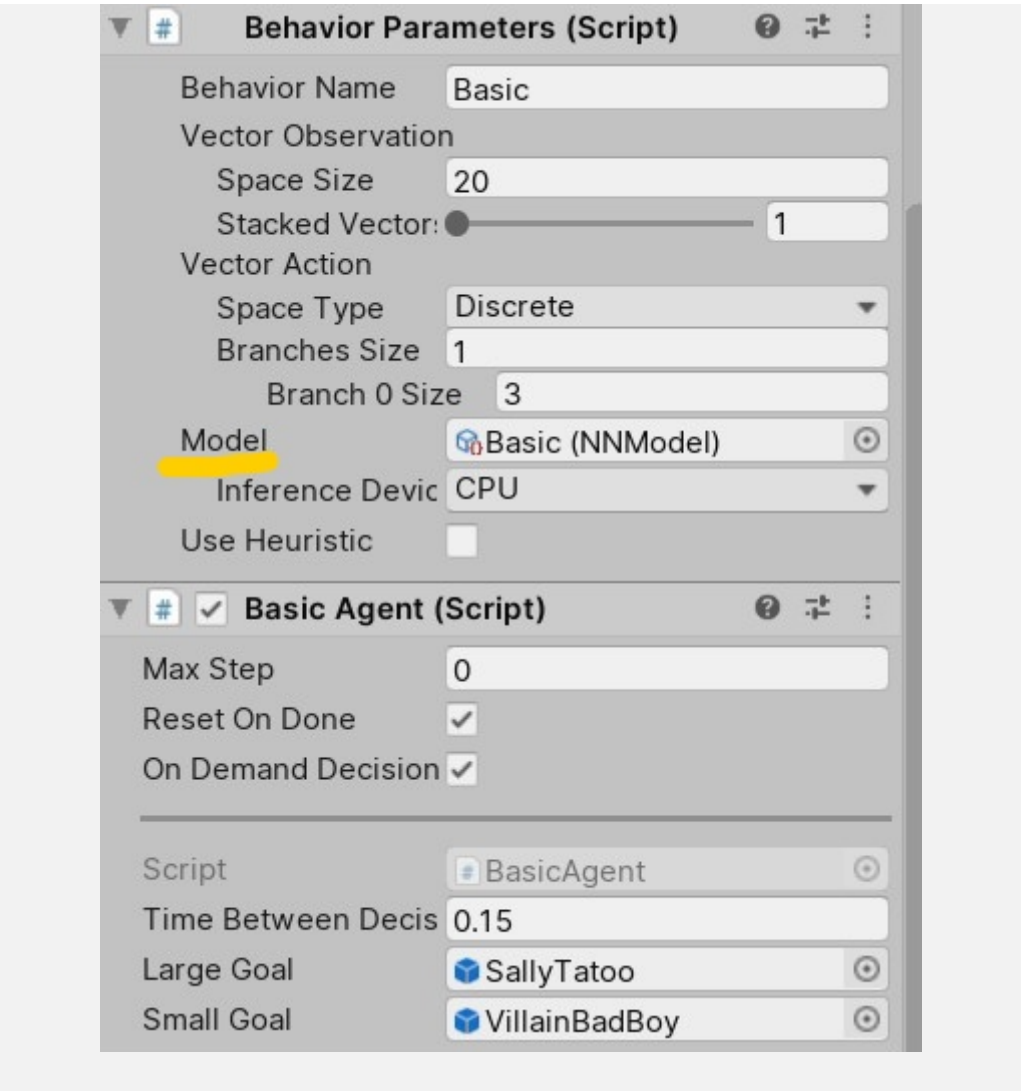
Se tutto va a buon fine dovreste vedere il logo di Unity e ci dovrebbe essere scritto di premere il pulsante "Play" nell'editor di Unity.

https://www.youtube.com/watch?v=sUZTs7sgHbE&t=7s

un video che mostra l'operazione.
A fine operazione su Conda prompt dovrebbe apparire una cosa del genere



Essenzialmente viene creato un file modello. Questo file e' molto Importante in quanto detta la Policy dell'agente durante il "gioco".

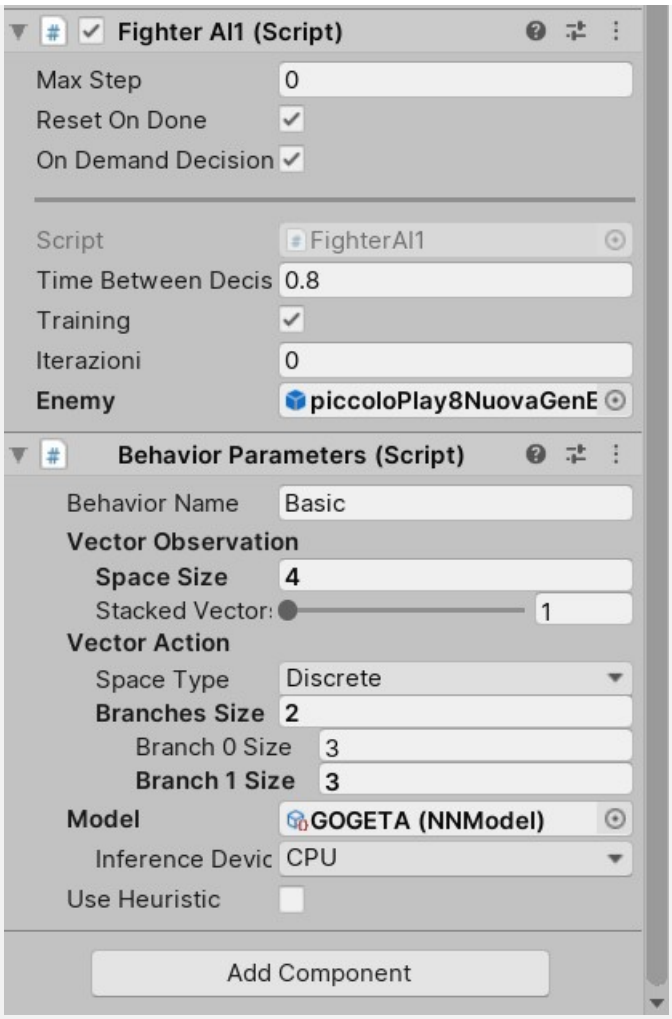


Importare all'interno di Unity il file creato prima e con un Drag&Dropp settarlo nel campo modello.
Cosi' facendo una volta premuto "Play" su Unity l'agente seguira' la policy del modello che abbiamo creato.

FINE DEL MEGA TUTORIAL SUL TRAINING..

IA combattimento.

Ho implementato due modelli di combattimento.
Un modello in modalita' Easy ed uno Medium.
Easy:

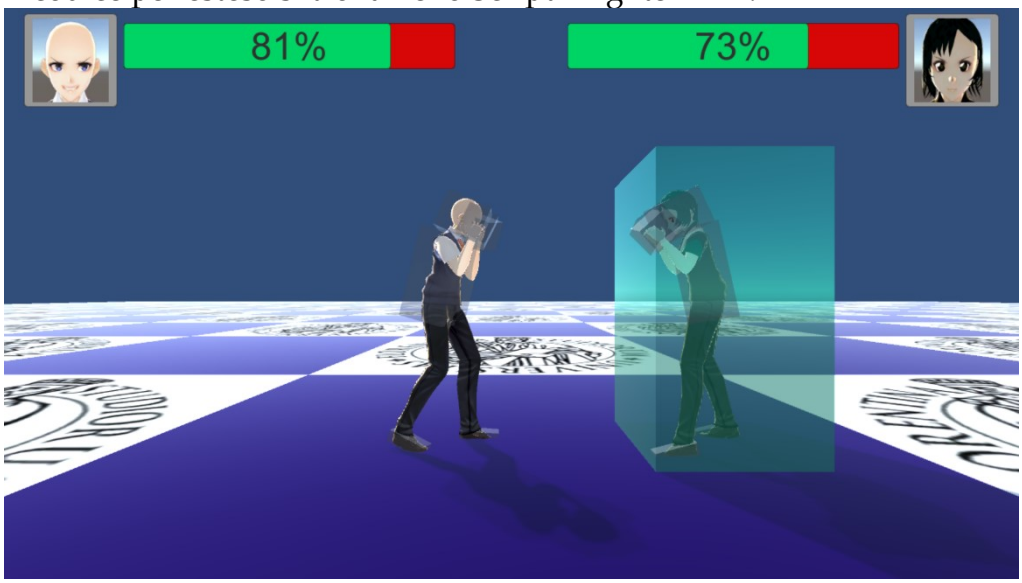


```
public override void AgentAction(float[] vectorAction, string
textAction)
    {
        if (GetComponent<CollisionDetection>().currentHp > 0) //
controllo per vedere se l'agente e' vivo
        {
            ActionMovement(vectorAction);

            if (Winning())
            {

                ActionFightOffensive(vectorAction,decider.Offense);


            }
            else
            {

                ActionFightDefensive(vectorAction,decider.Offense);
            }
        }
        else
        {
            AddReward(-10);
            if(training) GetComponent<CollisionDetection>().Reset();
            Done();
        }




        if
(enemy.gameObject.GetComponent<CollisionDetection>().currentHp <= 0)
        {
            AddReward(1);
            Done();
        }
        else
        {
            AddReward(0.5f);
        }
        iterazioni++;
        if (iterazioni == 20)
        {
            Done();
        }

    }
```

La strategia in modalita' Easy ha le seguenti due azioni.
Spostarsi = { passo in avanti, passo in dietro, stai fermo }
Combattere ={ Calcio Laterale, Pugno, Calcio Circolare }.
L'agente nella modalita' Easy tiene traccia dei suoi attuali Hp e quelli dell'avversario, la propria posizione e quella dell'avversario attraverso il metodo CollectObservation( ).

```
public override void CollectObservations()
    {
   //  i dati osservati e' meglio se vengono normalizzati
   AddVectorObs(GetComponent<CollisionDetection>().currentHp/150);
   //tiene d'occhio la vita normalizzata

   AddVectorObs(enemy.GetComponent<CollisionDetection>().currentHp/150);
   // tiene d'occhio la vita dell'avversario normalizzata
   AddVectorObs(transform.position.x/maxRadius); // tiene d'occhio la
                                                    propria posizione
   AddVectorObs(enemy.transform.position.x / maxRadius); // tiene
                                d'occhio la posizione dell'avversario


    }
```

In modalita' Easy a seconda se l'Agente stia vincendo o perdendo, esso cambia il dominio dell'azione combattere sostituendo "Calcio Laterale" con "Difesa". Nel gioco la Difesa e' rappresentata come un solido di colore Verde che wrappa l'avatar dell'agente/Giocatore, durante la Difesa non si subiscono danni.
Il codice per esteso si trova nello Script "FighterAI1".



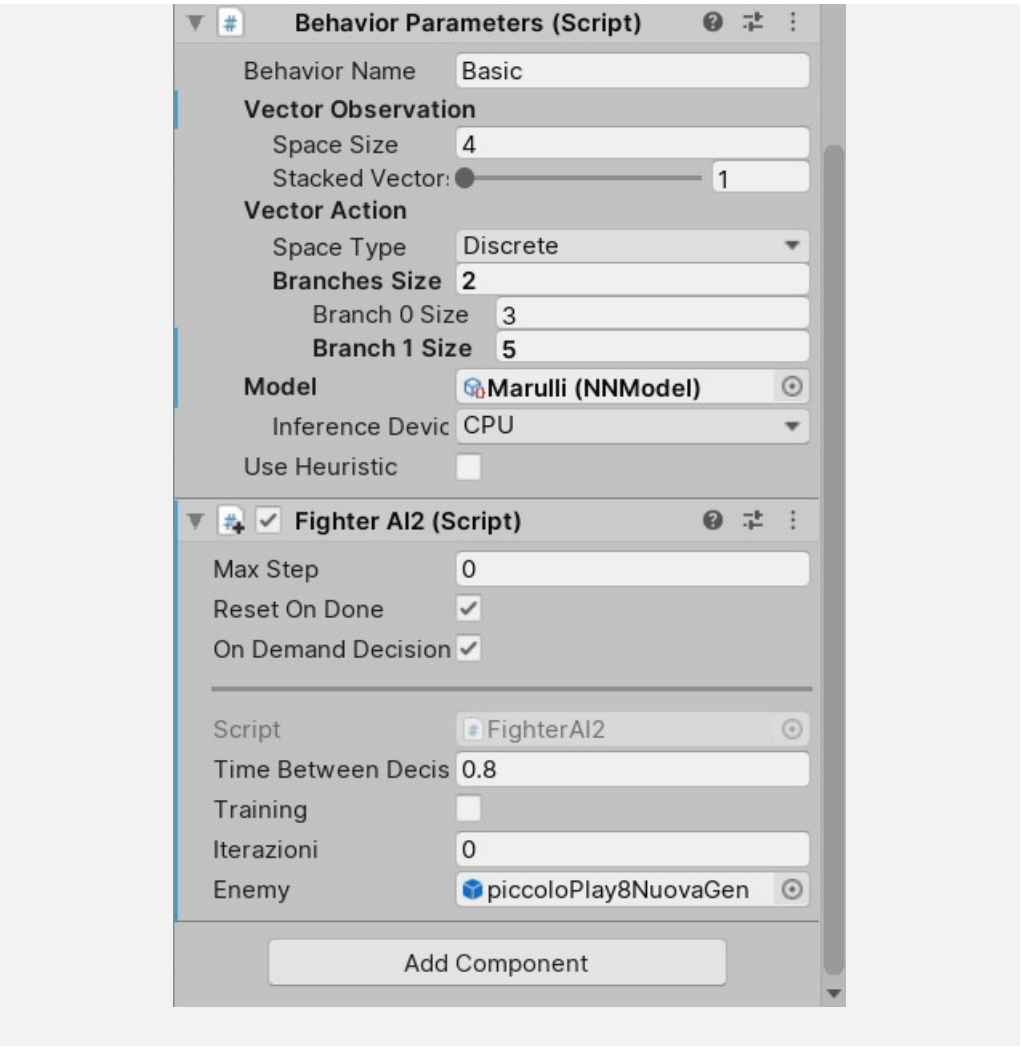
Avatar in posizione di Difesa*

Il dominio dell'azione Combattere e' mappato nel seguente modo:

```
private void
ActionFightOffensive(float[]vectorAction,Dictionary<int,string>offense)
    {

        var attack = (int)vectorAction[1];
        switch (attack)
        {
            case 0:
                    DeActivateGuard();
                    animator.Play(offense[0], 0, -1);

                    ChangePosition(stepKick);
                    AddReward(0.5f);

                break;


            case 1:
                DeActivateGuard();

                animator.Play(offense[1], 0, -1);
                ChangePosition(stepKick);
                AddReward(1f);

                break;
            case 2:
                DeActivateGuard();
                animator.Play(offense[2], 0, -1);
                ChangePosition(stepKick);
                AddReward(0.5f);
                break;
        }
    }
```

La variabile nel codice "offense" e' un dizionario <intero,stringa>.
La stringa restituita da offense e' scelta per invocare il metodo Play(string) di Unity per eseguire l'animazione del colpo da effettuare.

Modalita' Medium:



La differenza con la strategia citata prima e' che il Dominio dell'azione Combattere e' piu' grande, ha cardinalita' 5 ed ha piu' ampia gamma di scelta tra le azioni.
Quindi l'agente impara pattern piu' interessanti, come la schivata.
Lo Script della strategia si chiama FighterAI2

SISTEMA DI COLLISIONI IN BREVE


```
public class BoxHit : MonoBehaviour
{
    public float damage;
    public CollisionDetection collisionDetector;

    IEnumerator Trump()
    {
        yield return new WaitForSeconds(1);
    }

    private void OnCollisionEnter(Collision collision)
    {

        if (!collisionDetector.DefenseOn)
        {
            if (collision.gameObject.tag == "Fist" ||
                collision.gameObject.tag == "Kick")
            {


if(collision.gameObject.GetComponent<BoxHitter>().collisionDetection.
playerOne != collisionDetector.playerOne)

                {
                    collisionDetector.UpdateHealthBar(damage);
                }
            }
        }
        StartCoroutine(Trump());
        StopCoroutine(Trump());
  }
```

L'idea e' che ci sono dei solidi che wrappano testa, corpo, pugni e piedi degli Avatar. Avviene danno quando c'e' collisione tra un Solido etichettato "Fist" o "Kick" e un Solido che wrappa Corpo e testa.
Si utilizza il Pattern Strategy e si delega alla classe CollisionDetection di fare l'update della Barra Hp dei giocatori e far partire l'animazione del Danno subito all'Avatar.

```
public class CollisionDetection : MonoBehaviour
{
    public Animator animator;
    private float HpMax = 150;
    public float currentHp = 150;
    public Image currentHealth;
    public Text ratioHp;
    public GameObject guard;
    private bool defenseOn=false;
    private float input=-0.25f;
    public bool playerOne=true;
    public List<Transform> box = new List<Transform>();

    public bool DefenseOn { get => defenseOn; set => defenseOn = value;
}
    public float Input { get => input; set => input = value; }

    public void UpdateHealthBar(float damage)
    {

        animator.Play("DAMAGE", -1, 0f);
        HitTransition();
        currentHp -= damage;
        float ratio = currentHp / HpMax;

        if (currentHp <= 0)
        {
            currentHealth.rectTransform.localScale = new Vector3(0, 1,
                                                                 1);
            ratioHp.text = "DEAD";
            animator.Play("knockdown_A", -1, 0f);
        }
        else
        {
            currentHealth.rectTransform.localScale = new Vector3(ratio,
                                                                 1, 1);
            ratioHp.text = (ratio * 100).ToString("0") + "%";
        }
    }
    public void Reset()
    {
        currentHp = 150;
        UpdateHealthBar(0);
    }
```

```
    // Start is called before the first frame update
    void Start()
    {
        if (playerOne)
        {
            input *= -1;
        }
        animator = GetComponent<Animator>();
        foreach(Transform child in transform)
        {
            if (child.tag == "Fist"|| child.tag == "Kick")
            {
                box.Add(child);
            }
        }
    }
    public void HitTransition()
    {
        Vector3 destination = new Vector3(transform.position.x + Input,
                                                                    0, 0);
        transform.position =
              Vector3.MoveTowards(transform.position,destination, 0.5f);
    }
    public void DeactivateOffense()
    {
        foreach(Transform child in box)
        {
            //child.GetComponent<BoxCollider>().isTrigger = false;
            child.GetComponent<BoxCollider>().enabled = false;
        }
    }
    public void ActivateOffense()
    {
        foreach (Transform child in box)
        {
            //child.GetComponent<BoxCollider>().isTrigger = true;
            child.GetComponent<BoxCollider>().enabled = true;
        }
    }

    public void ActivateGuard()
    {
        DeactivateOffense();
        defenseOn = true;
        guard.GetComponent<Renderer>().enabled = true;

    }
    public void DeActivateGuard()
    {
        ActivateOffense();
        defenseOn = false;
        guard.GetComponent<Renderer>().enabled = false;
    }
}
```

KINECT

L'interazione con il Kinect avviene attraverso Nuitrack
https://download.3divi.com/Nuitrack/doc/Architecture_page.html
Dal Link:
" Nuitrack API is a native (C++) synchronous interface based on the asynchronous middleware layer that conceals all the interactions with 3D sensor and other devices, as well as with an operating system (Linux, Android). As a result, the user only applies Nuitrack API for writing cross-platform applications.

Nuitrack makes it possible to receive data from a 3D sensor to Android without root rights required. The interaction with the Android OS occurs through JNI.

The SDK contains a Nuitrack C# wrapper, a plugin for Unity3D and a plugin for Android devices."

Tutorial per l'installazione:

https://download.3divi.com/Nuitrack/doc/Installation_page.html

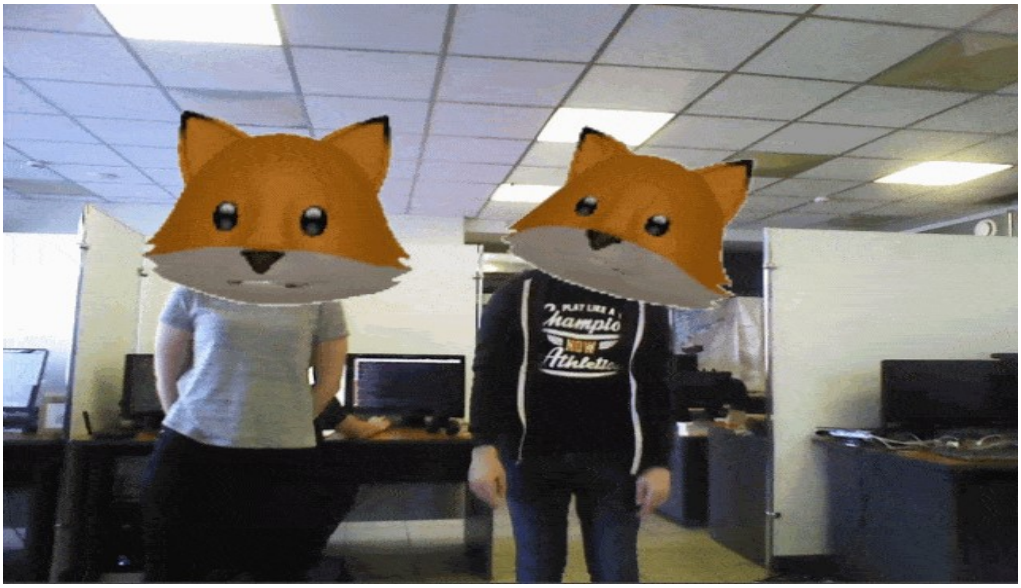
Il vantaggio di utilizzare Nuitrack e' che permette il crossPlatform su Android, Linux e Windows. Per di piu' si puo' utilizzare sia su Unity che su Unreal Engine 4. Il team di sviluppo e' molto attivo e sta sviluppando nuove API che utilizzano il machine Learning per fare il tracking delle dita di una mano semplicemente utilizzando il Kinect.

Un altro vantaggio e' che Nuitrack non funziona solamente con il kinect ma anche con altri sensori.

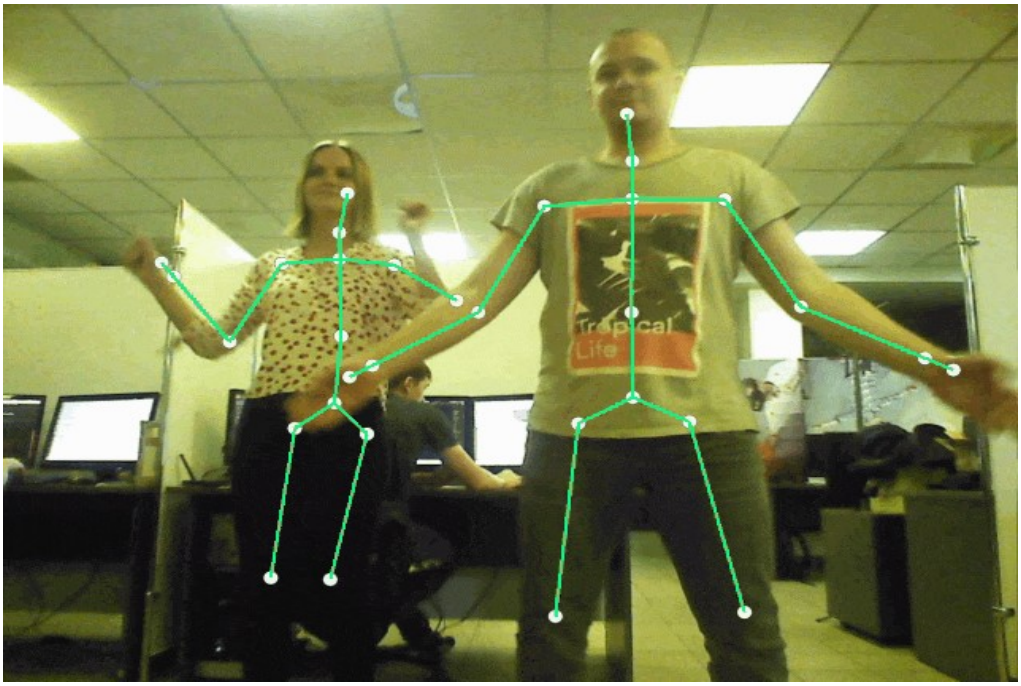
(mi sembra di stare facendo uno spot pubblicitario ma ci sono tanti vantaggi sul serio.)

Ci sono molti tutorial e la Community e' attiva
https://download.3divi.com/Nuitrack/doc/UnityTutorials_page.html

Traking Facciale con un Emoji stile Iphone X

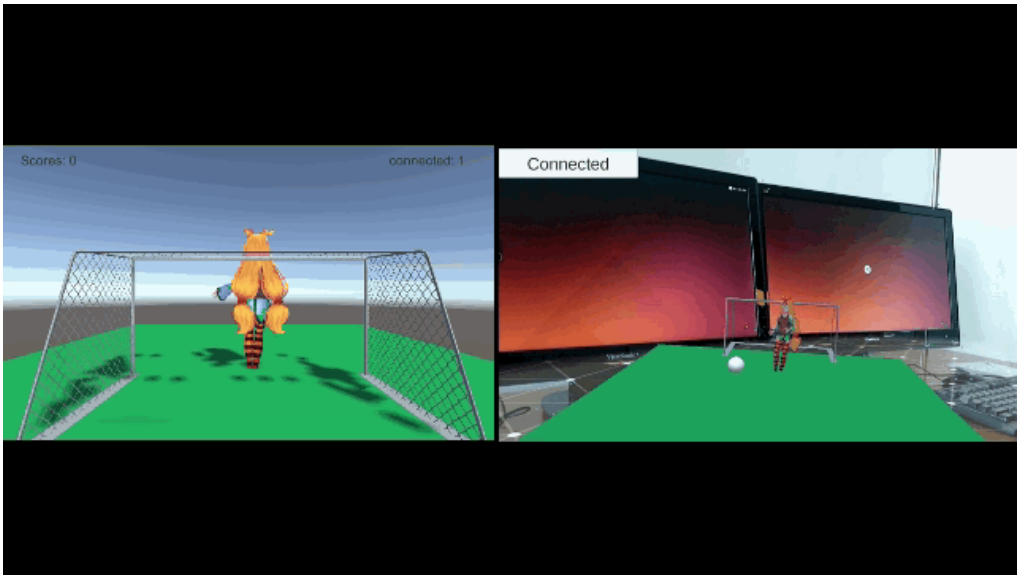


Displaying Skeletons on an RGB Image



User Segment Visualization

Creating an AR Football Game using Nuitrack and ARCore



Traking dell'Avatar per Combattere

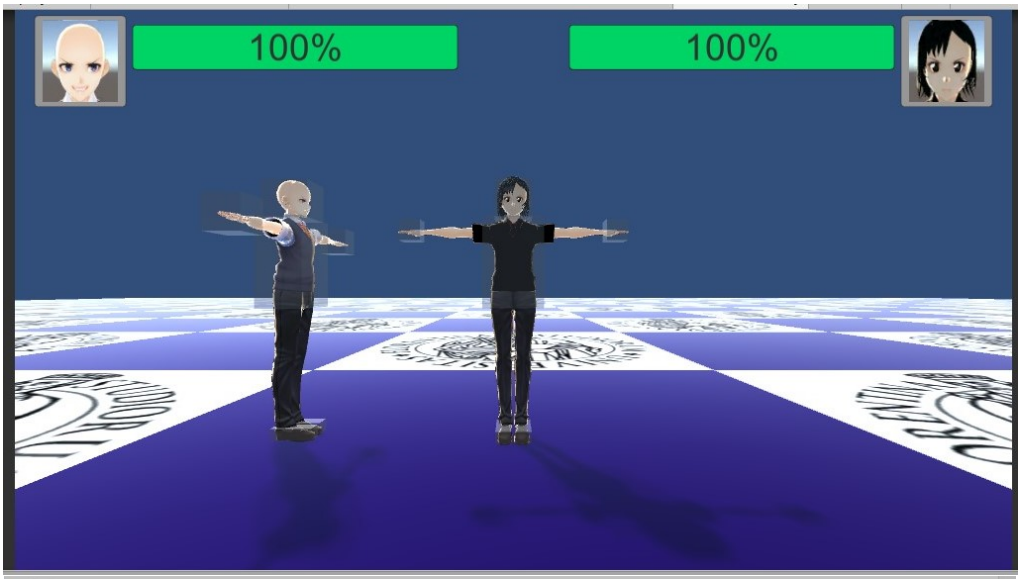
Mi sono basato su questo schema per mappare i vari Bones.

```
public class ThanosMovements : MonoBehaviour
{
    [Header("Rigged model")]
    [SerializeField]
    ////////////////////////////////
    ModelJoint[] modelJoints;

    /// <summary> Model bones </summary>
    Dictionary<nuitrack.JointType, ModelJoint> jointsRigged = new
Dictionary<nuitrack.JointType, ModelJoint>();

    void Start()
    {
        for (int i = 0; i < modelJoints.Length; i++)
        {
            modelJoints[i].baseRotOffset = modelJoints[i].bone.rotation;
            jointsRigged.Add(modelJoints[i].jointType, modelJoints[i]);
        }
    }
```

```
    void Update()
    {
        if (CurrentUserTracker.CurrentSkeleton != null)
ProcessSkeleton(CurrentUserTracker.CurrentSkeleton);
    }

    void ProcessSkeleton(nuitrack.Skeleton skeleton)
    {
        //Calculate the model position: take the Torso position and
invert movement along the Z axis
        Vector3 torsoPos = Quaternion.Euler(0f, 180f, 0f) * (0.001f *
skeleton.GetJoint(nuitrack.JointType.Torso).ToVector3());
        transform.position = torsoPos;

        foreach (var riggedJoint in jointsRigged)
        {
            //Get joint from the Nuitrack
            nuitrack.Joint joint = skeleton.GetJoint(riggedJoint.Key);

            ModelJoint modelJoint = riggedJoint.Value;

            //Calculate the model bone rotation: take the mirrored joint
orientation, add a basic rotation of the model bone, invert movement
along the Z axis
            Quaternion jointOrient =
Quaternion.Inverse(CalibrationInfo.SensorOrientation) *
(joint.ToQuaternionMirrored()) * modelJoint.baseRotOffset;
            modelJoint.bone.rotation = jointOrient;
        }
    }
}
```

Il codice in cui uso le API di Nuitrack*

Il suo scopo e' quello di Mappare un vettore di Bones in "`nuitrack.JointType`" e fare l'update delle rotazioni di tali Bones.



Thanos Movements (Script)

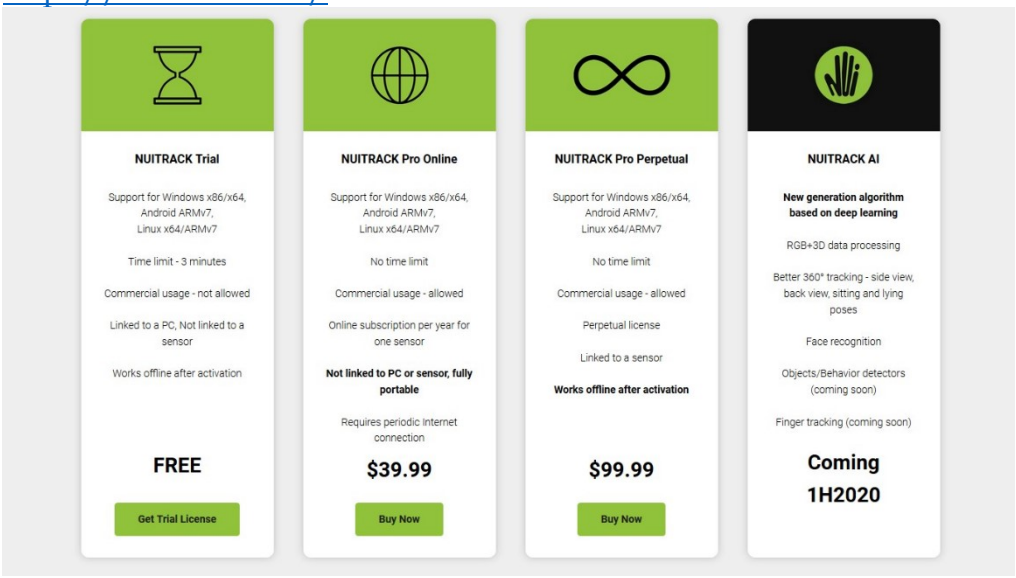
Script	ThanosMovements
Rigged model	
Model Joints	
Size	**16**
Element 0	
Bone	J_Bip_L_UpperArm (Transform)
Joint Type	Left Shoulder
Parent Joint Type	None
Element 1	
Bone	J_Bip_L_LowerArm (Transform)
Joint Type	Left Elbow
Parent Joint Type	None
Element 2	
Bone	J_Bip_R_UpperArm (Transform)
Joint Type	Right Shoulder
Parent Joint Type	None
Element 3	
Bone	J_Bip_R_LowerArm (Transform)
Joint Type	Right Elbow
Parent Joint Type	None
Element 4	
Bone	J_Bip_C_Head (Transform)
Joint Type	Head
Parent Joint Type	None
Element 5	
Bone	**J_Bip_C_Chest (Transform)**
Joint Type	**Torso**
Parent Joint Type	None
Element 6	
Bone	J_Bip_L_UpperLeg (Transform)
Joint Type	Left Hip
Parent Joint Type	None
Element 7	
Bone	J_Bip_L_LowerLeg (Transform)

piccoloPlay

Auto Generate Lighting On

Ci vuole un po' di olio di gomito perche' vanno settati tutti tramite inspector di Unity.. Pero' ne vale la pena.
Come si puo' notare dallo screen precedente i solidi che wrappano i pungi dell'Avatar controllato dal giocatore tramite Kinect sono piu' piccoli. Il giocatore per attivare la guardia deve incrociare i pugni facendo collidere i due solidi che wrappano i pugni (il modo piu' semplice che mi e' venuto in mente). A breve un video sull'utilizzo del kinect in combattimento.

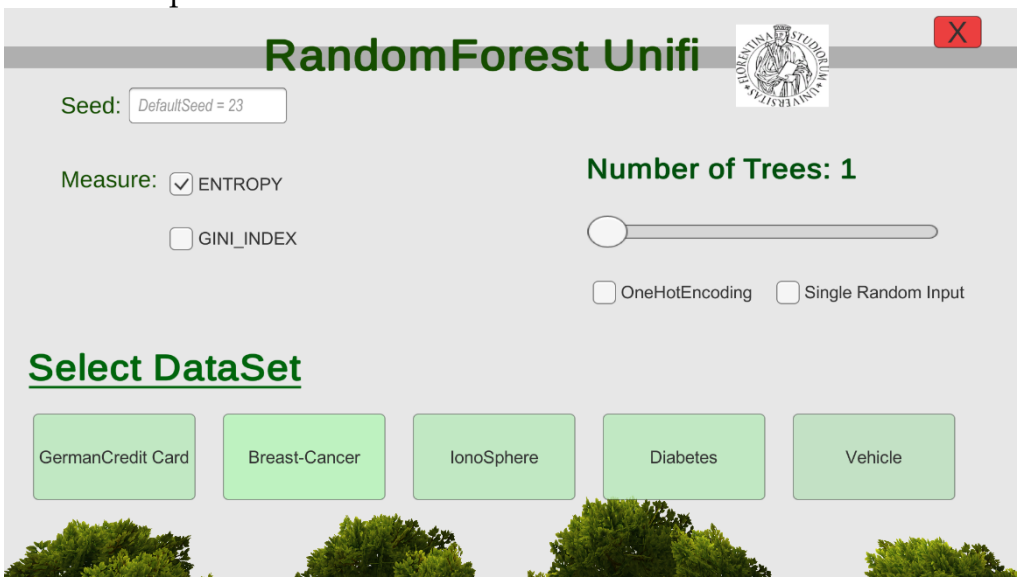
Concusioni.
Il codice non e' libero da Spaghetti Code.
Nuitrack ha il difetto essere su licenza in realta', qui presente il piano.
https://nuitrack.com/

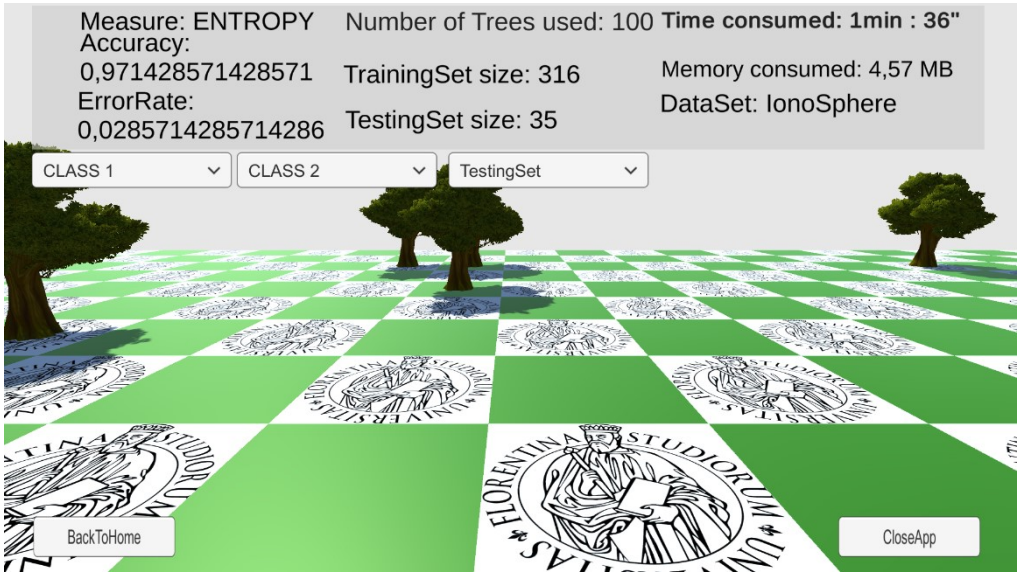


La versione FREE pero' ci permette di imparare ad utilizzare e implementare abbondantemente tutti i tutorial presenti facendoci una buona idea di quello che possiamo fare.

Ora a livello scolastisco non e' il massimo dover acquisire una licenza pero' paragonato a roba piu' professionale per fare motion Capure come Xsens la quale licenza arriva a costare sui 15'000$, 100$ se uno volesse creare qualcosa di commerciale senza dovere reinventare la ruota secondo e' un buon tradeOff (Io ho acquistato la licenza Pro a Natale pagando sui 50$).
Link XSens: https://www.xsens.com/products/mvn-animate

Spero il progetto vi sia piaciuto e possiate trarne spunto per implemtare i vostri “videogiochi” diciamo. Io sono piu’ interessato alla parte cinematrografica e penso che le tecnihce viste siano il futuro per poter realizzare contenuti di alta qualita’ con un low Budget.
Un po’ di sensi di colpa alla fine ad aver utilizzato tutto come un tool pero’ li ho alleviati implementando negli utlimi 2 mesi l’algoritmo di base con il quale funziona il Kinect: RANDOM FOREST.





https://github.com/piccoloplay/ProgettoIA
se siete interessati.