

Progettazione Videogame

Introduzione	2
Unity3D.....	2
Aspetti generali.....	2
Creazione dell'ambientazione	3
<i>Modellazione del terreno.....</i>	<i>3</i>
<i>Alberi, erba e altri dettagli.....</i>	<i>4</i>
<i>Windzone</i>	<i>6</i>
<i>Luci.....</i>	<i>6</i>
Creazione del Gameplay e del Player	7
<i>Box Collider</i>	<i>7</i>
<i>Footstep</i>	<i>8</i>
<i>Shooter e ChangeGun</i>	<i>9</i>
<i>Silenziatore</i>	<i>11</i>
Creazione dei nemici	12
<i>Ragdoll.....</i>	<i>12</i>
<i>Animation System.....</i>	<i>13</i>
<i>Navigation System.....</i>	<i>18</i>
<i>Damage System</i>	<i>19</i>

Introduzione

Il presente elaborato ha la finalità di analizzare e descrivere lo sviluppo di un'applicazione che rientra, in particolare, nella categoria 'Gaming'. Il processo di sviluppo è dettagliato seguendo l'implementazione di un gioco del genere *sparatutto in prima persona (FPS)*.

Il gioco in questione propone al giocatore la caccia degli animali, tramite arma da fuoco, all'interno di uno scenario 'Open World' in 3D.

Unity3D

Unity3D è l'ambiente di sviluppo utilizzato per questo progetto. Si tratta di un tool di sviluppo completamente integrato che fornisce numerose funzionalità out-of-the-box per lo sviluppo di videogiochi o altri applicativi interattivi 3D. E' possibile sviluppare ambienti e scenari eterogenei configurando luci, audio, effetti speciali, funzionalità legate alla fisica ed animazioni per poi essere testate e compilate per svariate piattaforme, quali Mac, PC, Linux, Web, iOS, Android, Wii, PS3 e Xbox.

Aspetti generali

Ogni elemento dell'applicazione in Unity è un **Game Object**, ossia un singolo oggetto gestibile individualmente attraverso una serie di **Componenti** configurabili che ne delineano le proprietà.

Ogni Game Object contiene almeno un componente chiamato **Transform** che tiene traccia della posizione, rotazione e della scalatura dell'oggetto all'interno dello scenario 3D. Sono inoltre disponibili svariati componenti che si occupano di gestire l'illuminazione dell'oggetto, le sue proprietà fisiche, le collisioni, le videocamere, gli effetti speciali, ecc. Ogni componente è configurabile tramite un insieme di **Variabili**.

Altro elemento fondamentale alla base dello sviluppo in Unity è l'**Assets**, ossia un file che rappresenta un qualsiasi oggetto che può essere utilizzato nel gioco. Esso può essere ad esempio un file audio, un modello 3D, un'immagine di texture o un'animazione.

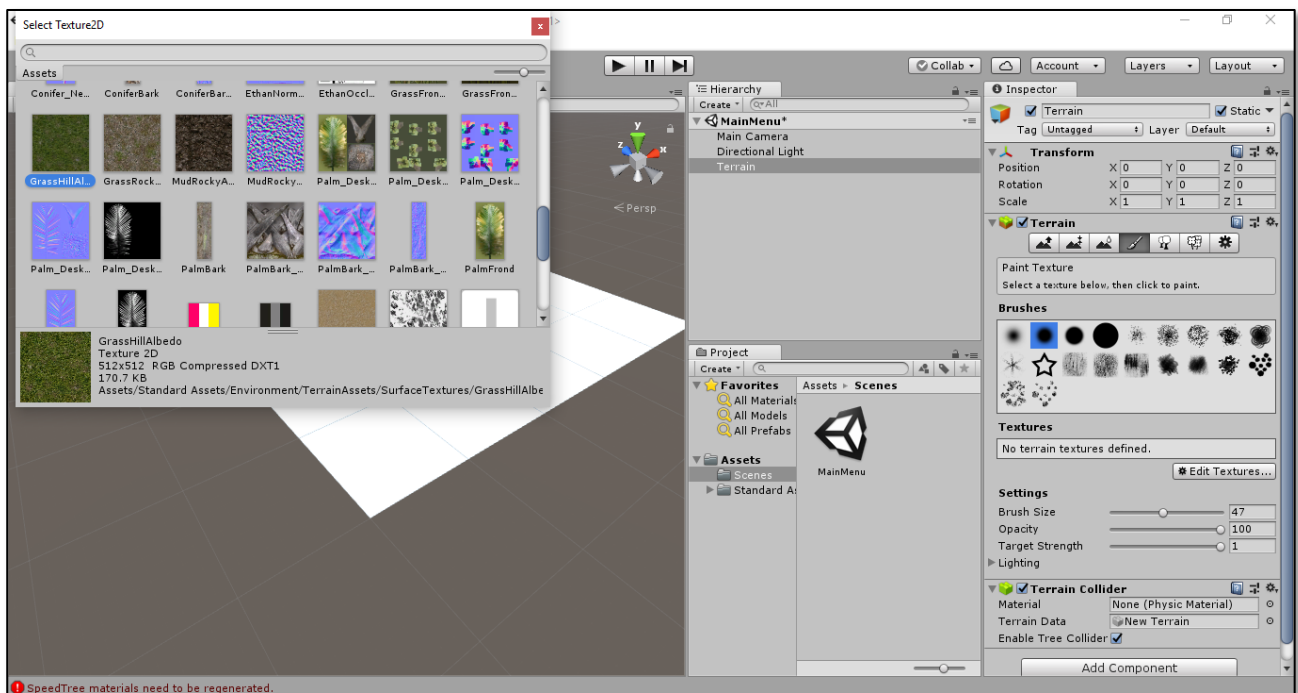
Unity inoltre fornisce la possibilità di estendere le sue funzionalità base con l'importazione di appositi **Script**, pezzi di codice che assolvono ad una determinata funzione, scritti generalmente in linguaggio Javascript, C# o Boo e che vengono trattati come dei componenti.

Per quanto riguarda la modellazione 3D si fa uso della rappresentazione tramite **Mesh**, ossia un insieme di vertici e poligoni che definiscono la forma dell'oggetto 3D. L'ambiente Unity non fornisce un proprio tool per la creazione di Mesh ma fornisce un'ottima integrazione con i più popolari software di modellazione quali Maya, 3DsMax e Blender.

Creazione dell'ambientazione

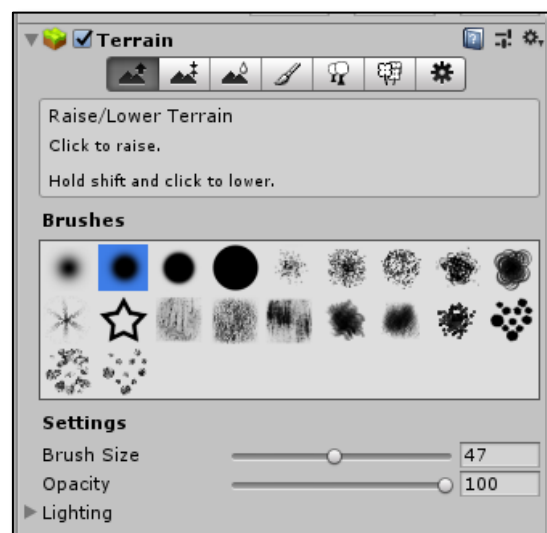
Il primo passo fondamentale è rappresentato dall'importazione del package di *Standard Assets* messo a disposizione dall'ambiente Unity nel quale sono contenuti la maggior parte di texture, modelli 3D e controller utilizzati all'interno del gioco.

È necessario poi creare una **Scena** che verrà popolata con tutti gli oggetti del gioco. Tramite il pannello *Hierarchy* viene creato un *Game Object* di tipo **Terrain**. E' possibile aggiungere un'immagine di texture alla superficie creata per dare colorazione e dettagli all'ambiente. Questo è possibile farlo tramite il pulsante edit texture all'interno dell'*Inspector*.



Modellazione del terreno

Viene generata una morfologia del territorio tramite gli strumenti contenuti nel componente **Terrain**.

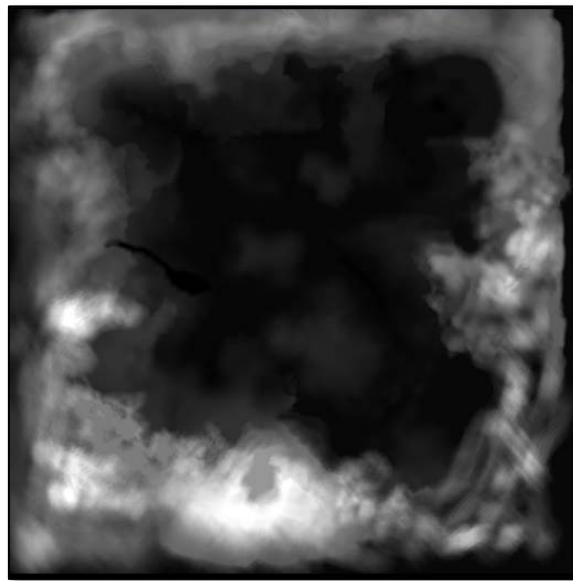




Tramite i primi tre pulsanti è possibile modellare manualmente le altezze sul terreno. In pratica è possibile disegnare le altezze tramite il cursore del mouse, una volta selezionato uno dei pulsanti.

Dietro questo meccanismo semplice ed intuitivo è nascosta una modellazione tramite **Heighmap**. L'altezza di ogni punto sul terreno è rappresentata come un valore in una matrice rettangolare. Questo array può essere rappresentato utilizzando un'immagine in scala di grigi nota come Heighmap.

Per modellare il terreno direttamente tramite heighmap è sufficiente importare la mappa delle altezze in formato *RAW* all'interno del pannello Terrain. Di seguito la heightmap utilizzata, dove il colore nero corrisponde all'altezza 0, e i grigi più chiari rappresentano altezze maggiori:



Alberi, erba e altri dettagli

La superficie creata può essere riempita con degli alberi, ed è fatto più o meno allo stesso modo in cui sono possono essere disegnate le altezze.



Selezionando il pulsante mostrato in figura, e scegliendo il modello tramite 'Edit Trees' è possibile disegnare delle toppe di alberi sulla mappa. La dimensione delle toppe e la distanza tra gli alberi possono essere configurate nella sezione impostazioni del componente Terrain.

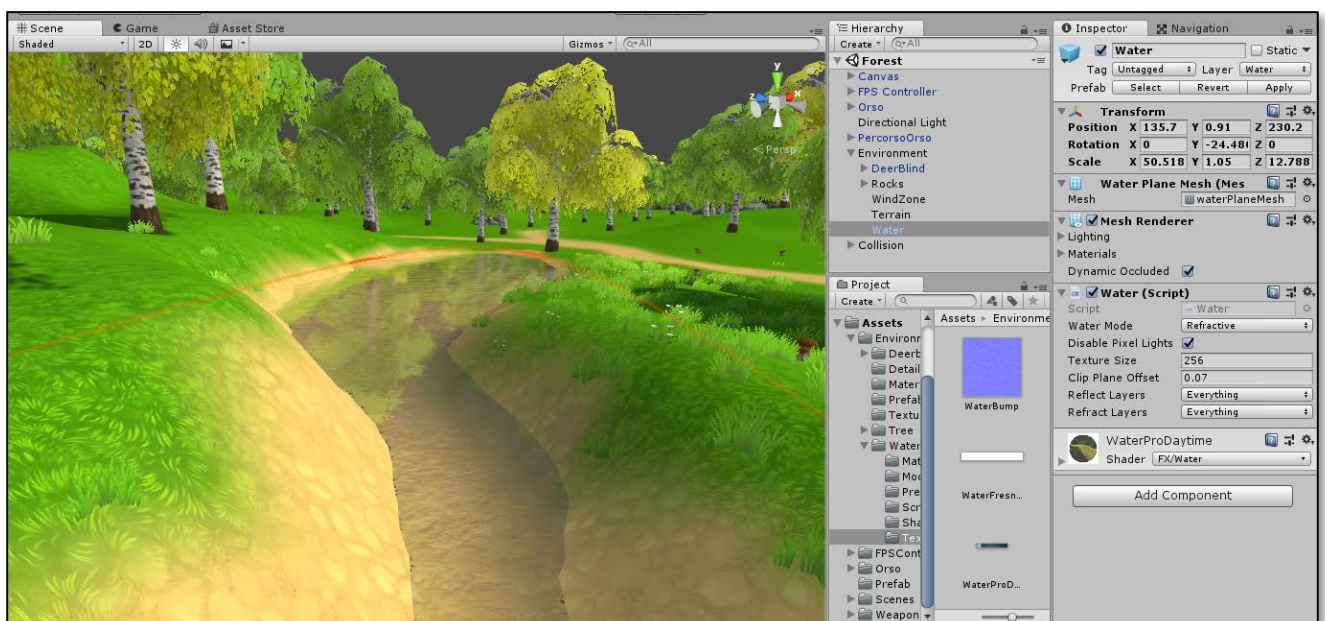


Inoltre, tramite lo stesso meccanismo (painting) è possibile aggiungere erba, fiori, rocce o altri dettagli .



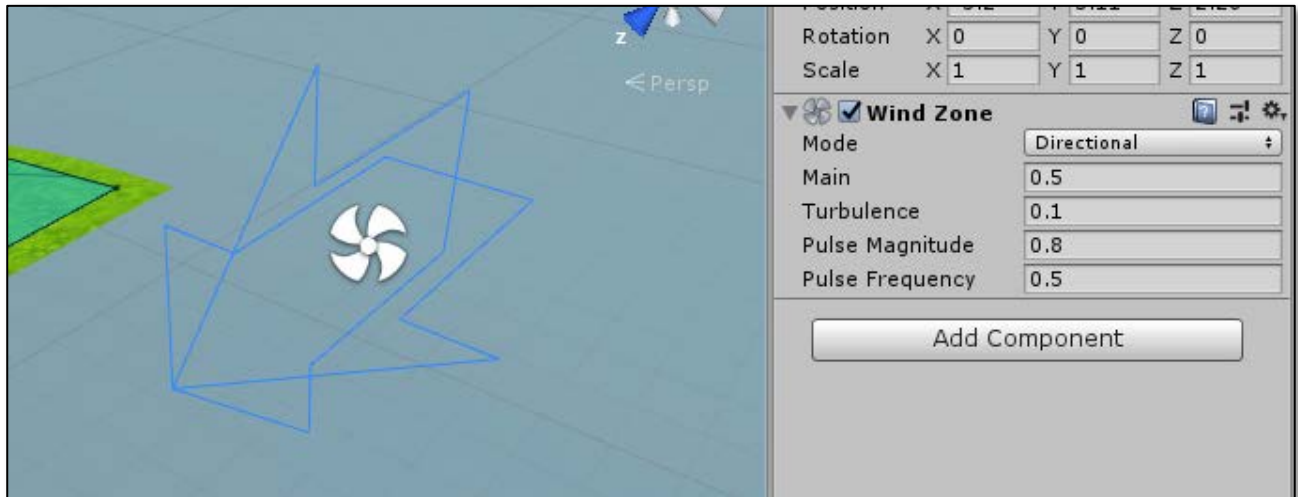
Tramite questo tasto è si accede al pannello di configurazione dei 'Dettagli' dove è possibile scegliere texture, dimensioni della toppa ed inoltre l'azione del vento.

Successivamente vengono create le zone d'acqua. All'interno del package importato 'Standard Assets' sono inclusi molti Prefab utili alla simulazione degli specchi d'acqua.



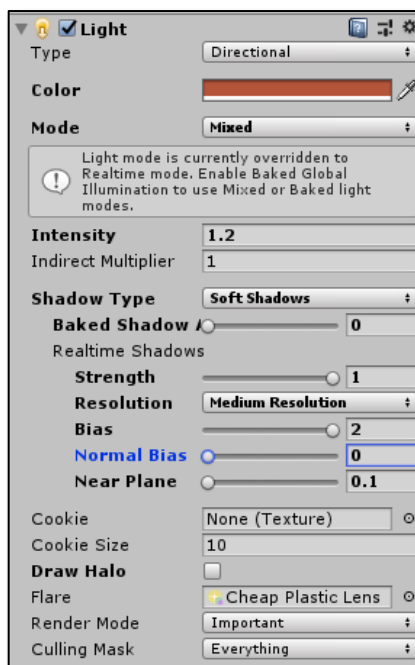
Windzone

E' possibile simulare l'azione del vento sul terreno creando uno o più oggetti con il componente **Windzone** attivato. Alberi e erba all'interno del raggio di azione si piegheranno con un animazione realistica.



Luci

Le luci sono una parte essenziale della scena. Esse possono essere aggiunte dal menu GameObject->Light. E' possibile scegliere il formato della luce, manipolare la posizione e direzione e cambiarne il colore.

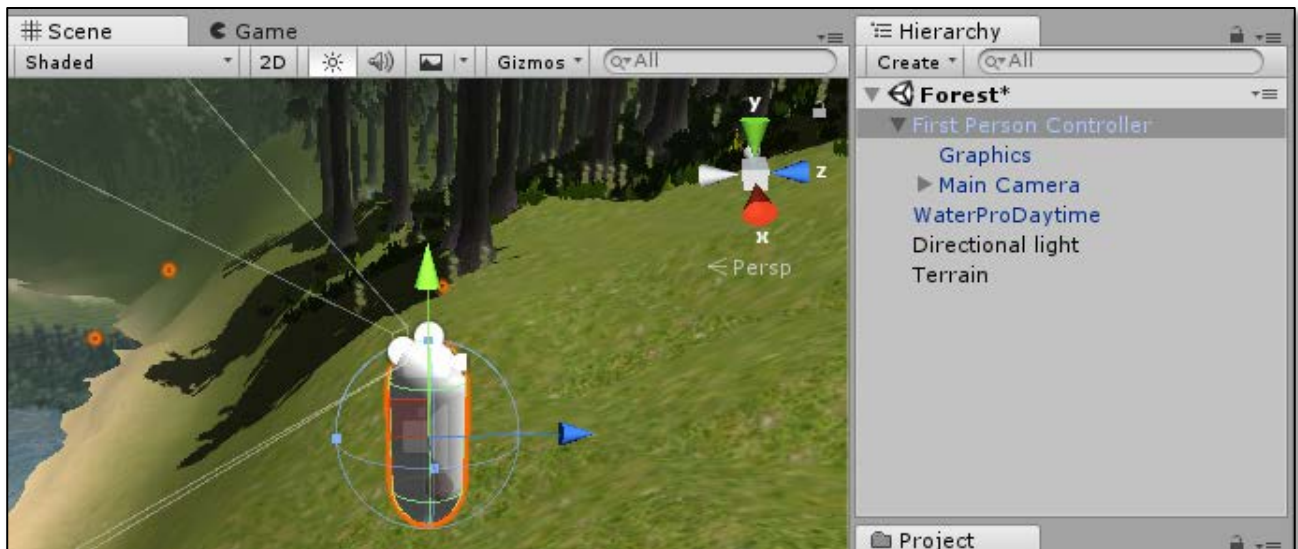


Automaticamente, impostando un punto luce vengono create le ombre per tutti gli oggetti della scena. Inoltre impostando una texture nel campo **Flare** viene simulato l'effetto delle luce che si rifrangono all'interno dell'obiettivo della camera nel momento in cui viene centrato il punto luce.

Creazione del Gameplay e del Player

Tra gli assets standard importati in precedenza troviamo, tra le altre cose, il **Character Controller**.

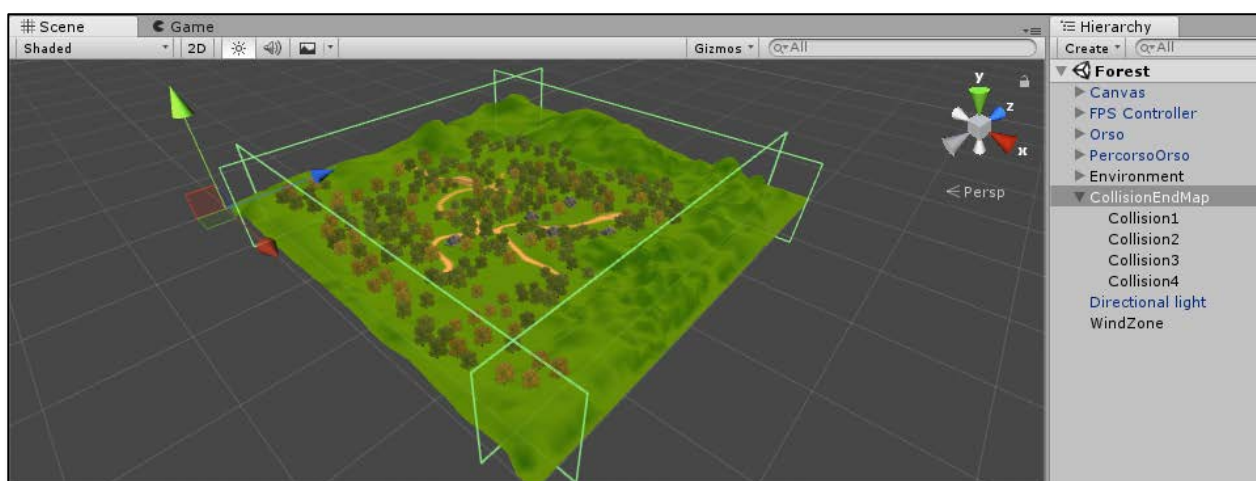
Il Controller è l'oggetto utilizzato per il controllo del giocatore e può essere configurato in *prima o terza persona*. Esso è composto da una parte *Graphics* che contiene il modello 3D fisico del personaggio e la parte *Main Camera* che consiste nell'oggetto che fa funzionare la visuale del giocatore.



Al suo interno sono già precaricati gli script che gestiscono il movimento (spostamento, corsa, accovacciamento e salto) e lo spostamento della visuale tramite mouse.

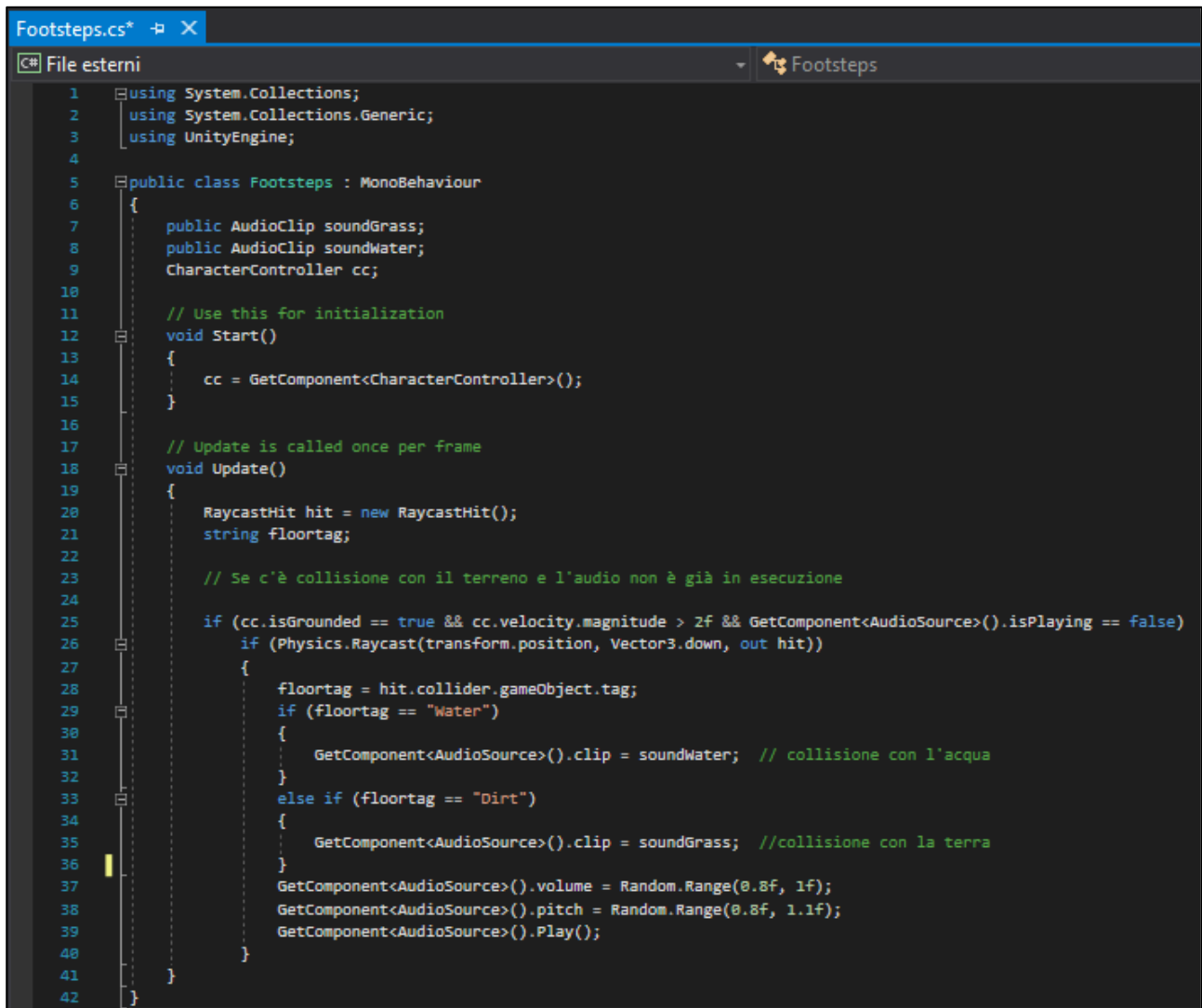
Box Collider

Per evitare che il player esca fuori dalla mappa e quindi cada nel vuoto sono stati inseriti quattro collider box in prossimità dei quattro bordi della mappa. Un **Box Collider** è un collider primitivo a forma di cubo (in questo caso parallelepipedo) che permette di intercettare le collisioni con gli altri oggetti che ne predispongono.



Footstep

Personalizziamo il nostro controller aggiungendo il suono dei passi quando cammina sul terreno. Per fare ciò è sufficiente aggiungere il componente 'Audio Source' all'oggetto Controller, all'interno del quale viene caricato automaticamente dallo script, all'occorrenza, il file audio che riproduce il suono del passo. Successivamente è necessario scrivere un piccolo script ed aggiungerlo al medesimo oggetto tramite il componente 'Script'.



```
1 using System.Collections;
2 using System.Collections.Generic;
3 using UnityEngine;
4
5 public class Footsteps : MonoBehaviour
6 {
7     public AudioClip soundGrass;
8     public AudioClip soundWater;
9     CharacterController cc;
10
11     // Use this for initialization
12     void Start()
13     {
14         cc = GetComponent<CharacterController>();
15     }
16
17     // Update is called once per frame
18     void Update()
19     {
20         RaycastHit hit = new RaycastHit();
21         string floortag;
22
23         // Se c'è collisione con il terreno e l'audio non è già in esecuzione
24
25         if (cc.isGrounded == true && cc.velocity.magnitude > 2f && GetComponent<AudioSource>().isPlaying == false)
26             if (Physics.Raycast(transform.position, Vector3.down, out hit))
27             {
28                 floortag = hit.collider.gameObject.tag;
29                 if (floortag == "Water")
30                 {
31                     GetComponent<AudioSource>().clip = soundWater; // collisione con l'acqua
32                 }
33                 else if (floortag == "Dirt")
34                 {
35                     GetComponent<AudioSource>().clip = soundGrass; // collisione con la terra
36                 }
37                 GetComponent<AudioSource>().volume = Random.Range(0.8f, 1f);
38                 GetComponent<AudioSource>().pitch = Random.Range(0.8f, 1.1f);
39                 GetComponent<AudioSource>().Play();
40             }
41     }
42 }
```

Come possiamo notare lo script è adattativo: tramite l'oggetto RayCast si detecta il tag dell'oggetto con il quale il controller sta avendo una collisione. Se il tag corrisponde a "Water" allora al componente AudioSource è assegnato il file del suono per l'acqua, altrimenti, quando viene detectato il tag "Dirt" si assegna il suono per la terra. Successivamente l'audio è eseguito con parametri di volume e pinch randomici per dare una dinamica più realistica al suono del passo.

Shooter e ChangeGun

Finora è stata descritta tutta la parte del Gameplay che riguarda il movimento. La fase successiva consiste nel fornire al player un'interfaccia FPS (First Person Shooter), sia grafica, e quindi che mostri l'arma impugnata in primo piano (con tutte le animazioni annesse) e il numero di munizioni, sia logico-funzionale, che quindi permetta di sparare, mirare e colpire un bersaglio targettizzato.

Di seguito è riportato uno screenshot dell'interfaccia del giocatore.

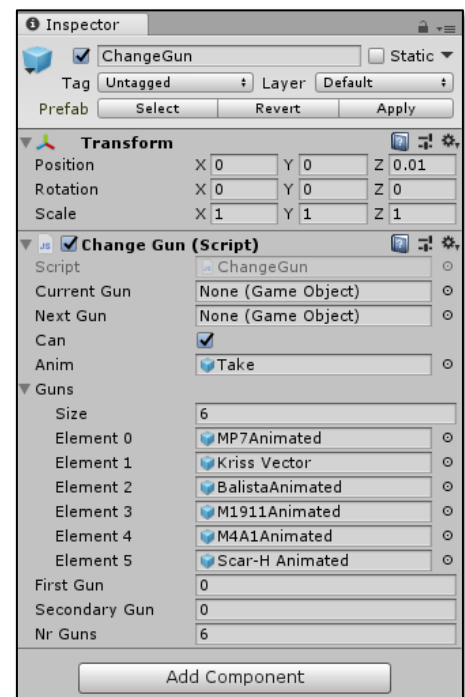


Il modello 3D del player non è stato caricato dato che è stata scelta una visuale in prima persona. Il body del controller è infatti un oggetto vuoto. Sono stati importati degli assets relativi a 6 tipi di arma da fuoco, tra i quali il giocatore può scegliere e cambiare a piacimento durante il gioco. All'avvio del gioco uno script permette di spawnare una dell'armi impostata come default.

Tramite l'inspector viene riempito un vettore di armi istanziato all'interno dello script. All'avvio del gioco (funzione Start) viene attivata l'arma di default, quella all'indice 0 nel vettore di armi. Per ogni frame (funzione Update) viene verificata la pressione dei tasti numerici tra [1,6]. Se è stato premuto uno di questi tasti, nella variabile 'NextGun' viene caricata l'arma all'indice (i-1) e poi viene chiamata la funzione ChangeGun. Questa esegue le animazioni per l'arma uscente e quella entrante (tramite gli indici CurrentGun e NextGun salvati) ed attiva la nuova arma corrente.

Di seguito è riportato lo script che permette di cambiare l'arma del player a runtime e i parametri di ingresso configurati tramite l'Inspector.

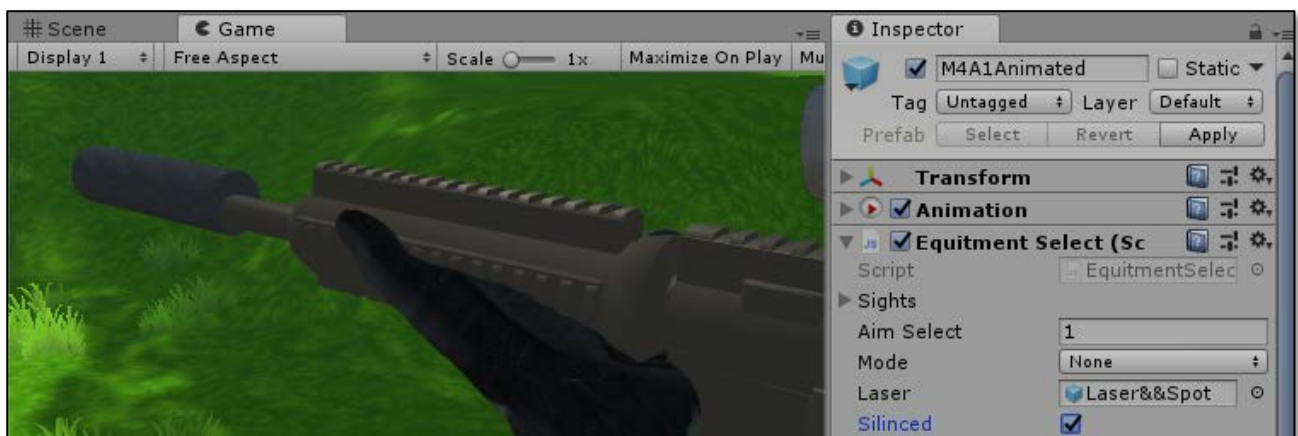
```
ChangeGun.js X Footsteps.cs
1  var CurrentGun:GameObject;
2  var NextGun:GameObject;
3  var Can:boolean=true;
4  var Anim:GameObject;
5  var Guns:GameObject[];
6  var FirstGun:int = 0;
7  var SecondaryGun:int = 0;
8
9  var NrGuns:int = 6;
10
11  function Start() {
12      Guns[FirstGun].active = true;
13      CurrentGun = Guns[FirstGun];
14      NextGun = Guns[SecondaryGun];
15  }
16
17  function Update() {
18      if (Can) {
19          for (var i:int = 1; i <= NrGuns; i++) {
20              if (Input.GetKeyDown("'" + i)) {
21                  SecondaryGun = i - 1;
22                  NextGun = Guns[SecondaryGun];
23                  ChangeGun();
24              }}}
25  }
26
27
28  function ChangeGun() {
29      Can = false;
30      Anim.GetComponent(<Animation>()).Play("TakeIn");
31      CurrentGun.GetComponent(<Animation>()).Play("TakeOut");
32      CurrentGun.GetComponent("Gun").CanFire = false;
33      yield WaitForSeconds(0.5);
34      NextGun.GetComponent("Gun").CanFire = false;
35      Anim.GetComponent(<Animation>()).Play("TakeOut");
36      CurrentGun.active = false;
37      NextGun.active = true;
38      NextGun.GetComponent(<Animation>()).Play("TakeIn");
39      yield WaitForSeconds(0.5);
40
41      var Save = CurrentGun;
42      CurrentGun = NextGun;
43      NextGun = Save;
44      CurrentGun.GetComponent("Gun").CanFire = true;
45      Can = true;
46  }
47
```



Altri due script che non sono stati riportati qui per non appesantire la trattazione gestiscono le funzionalità 'caricare l'arma', 'mirare', 'sparare' ed ovviamente le animazione e i suoni annessi.

Silenziatore

Grazie allo script `EquitmentSelect` invece è possibile inserire e disinserire all'occorrenza il *silenziatore* dell'arma che si sta usando premendo il pulsante 'P'.



Di seguito lo script che implementa questa funzionalità:

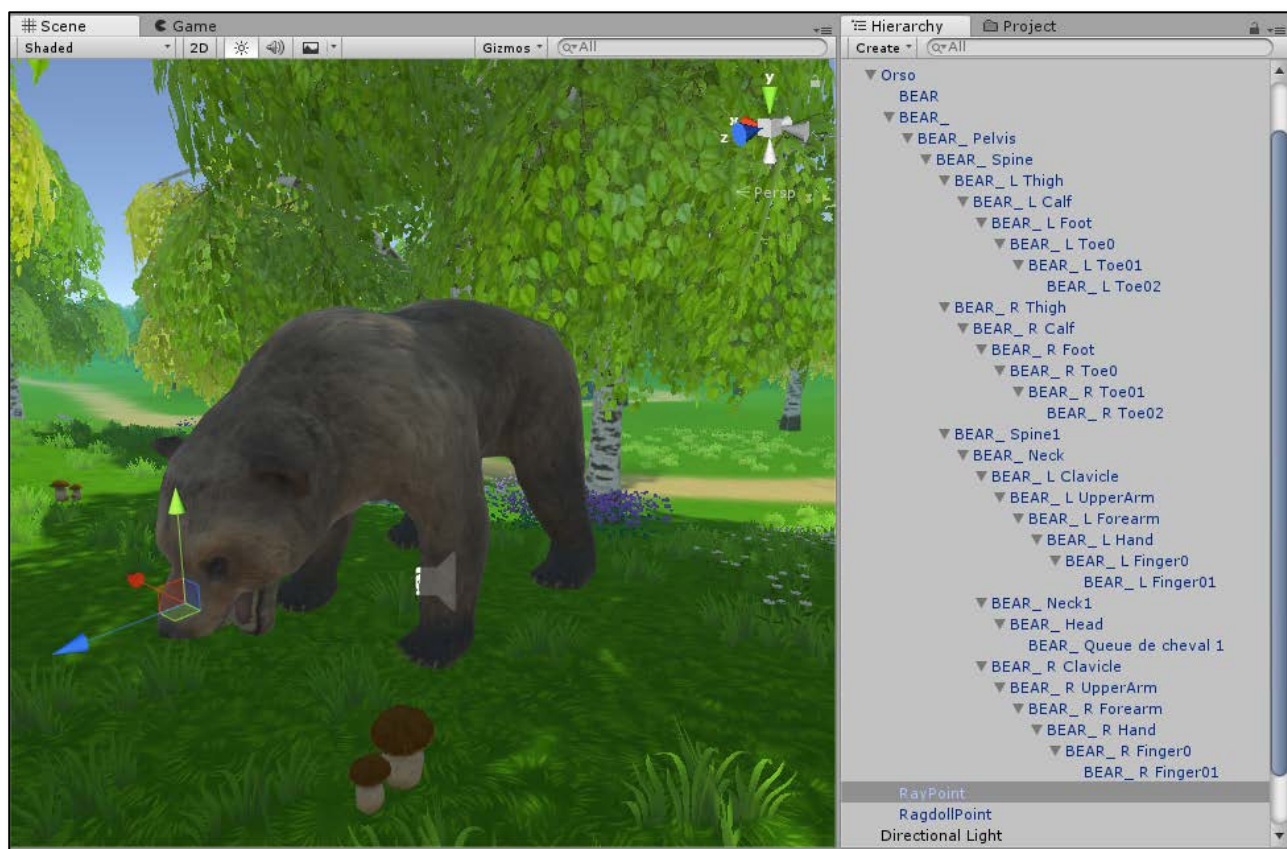
```
EquipmentSelect.js
16
17 var Silenced:boolean=false;
18
19 function Start () {
20     BeforeGetComponent(Gun).GunOptions.RandomShoot+1;
21     Sights[AimSelect].active=true;
22 }
23
24
25 function Update() {
26
27     GetComponent(Gun).GunOptions.Silenced = Silenced;
28
29     if (Input.GetKeyDown(KeyCode.P)) {
30         if (Silenced) {
31             Silenced = false;
32             GetComponent(Gun).GunOptions.Silenced = Silenced;
33         }
34         else {
35             Silenced = true;
36             GetComponent(Gun).GunOptions.Silenced = Silenced;
37         }
38     }
39 }
```

Creazione dei nemici

La seguente sezione si propone di descrivere:

- Creazione del nemico, individuato in un orso;
- Implementazione della logica di *Detection* del Player da parte degli animali;
- Implementazione del *Damage System* del nemico.

Una volta scaricato ed importato un package asset contenente il modello 3D di un **orso**, corredato con tutte le texture, i materiali, le animazioni e i suoni è necessario lavorare per mettere insieme questi componenti e renderli funzionali.



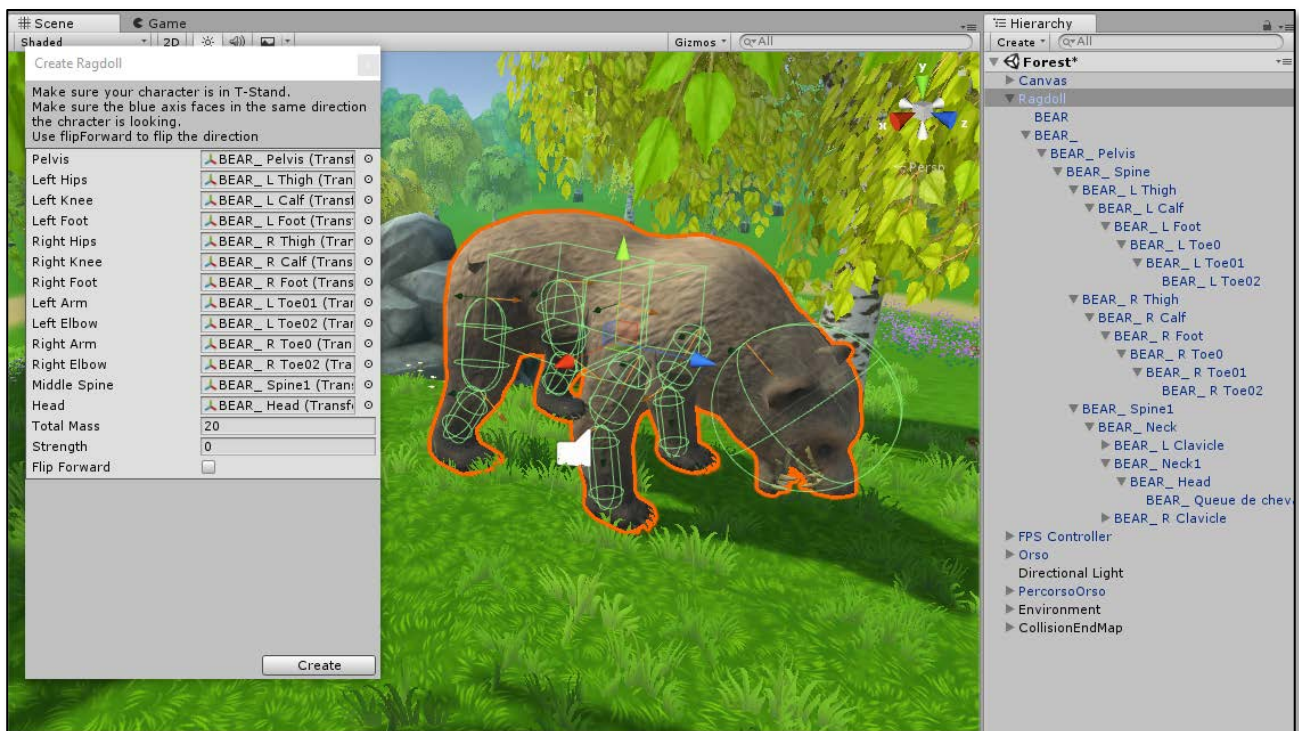
L'oggetto Orso conterrà un sottoggetto che rappresenta lo scheletro (BEAR_) e un altro sottoggetto che contiene la mesh vera e propria e che accorpa modello e texture (BEAR).

Inoltre è stato creato un oggetto vuoto chiamato RayPoint, utile successivamente per configurare il punto di attacco (posizionato in prossimità della bocca dell'orso).

Ragdoll

Altro sottoggetto da aggiungere al modello è una **Ragdoll**. Questa rappresenta lo scheletro del modello 3D, composto da ossa e giunti tra ossa. E' utile ad esempio per evitare di sviluppare una animazione che simuli la morte dell'animale. Infatti quando attivata, la ragdoll fa cadere il corpo al suono come se fosse sostenuto da alcuna forza.

Per creare una Ragdoll è possibile utilizzare il 'Ragdoll Wizard' fornito dall'ambiente.



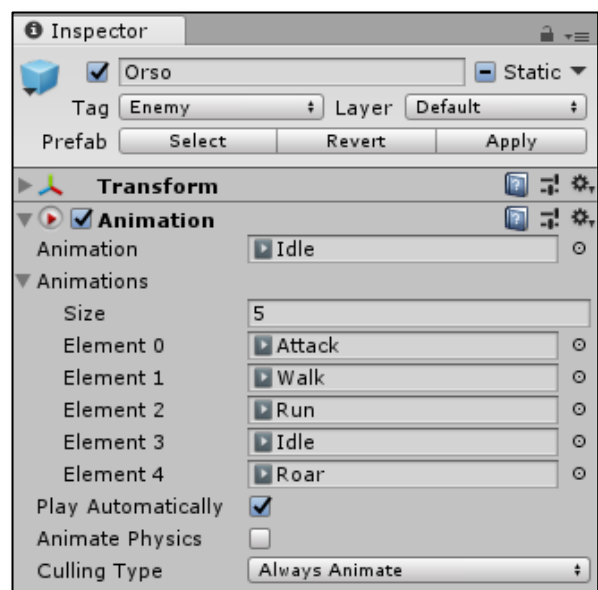
Facendo click destro, troviamo tra gli oggetti 3D l'oggetto Ragdoll. Si aprirà una finestra dove è possibile configurare tutti gli elementi che fanno parte dello scheletro importato in precedenza.

Successivamente nello script che controlla tutte le interazione dell'orso, alla sua morte, l'oggetto 'Orso' verrà sostituito con la Ragdoll.

Animation System

Per gestire le animazione dell'orso non si è fatto ricorso all'**Animator Controller**, il sistema di animazione integrato, ma è stato gestito il tutto tramite uno script che attiva le *clip di animazione* all'occorrenza.

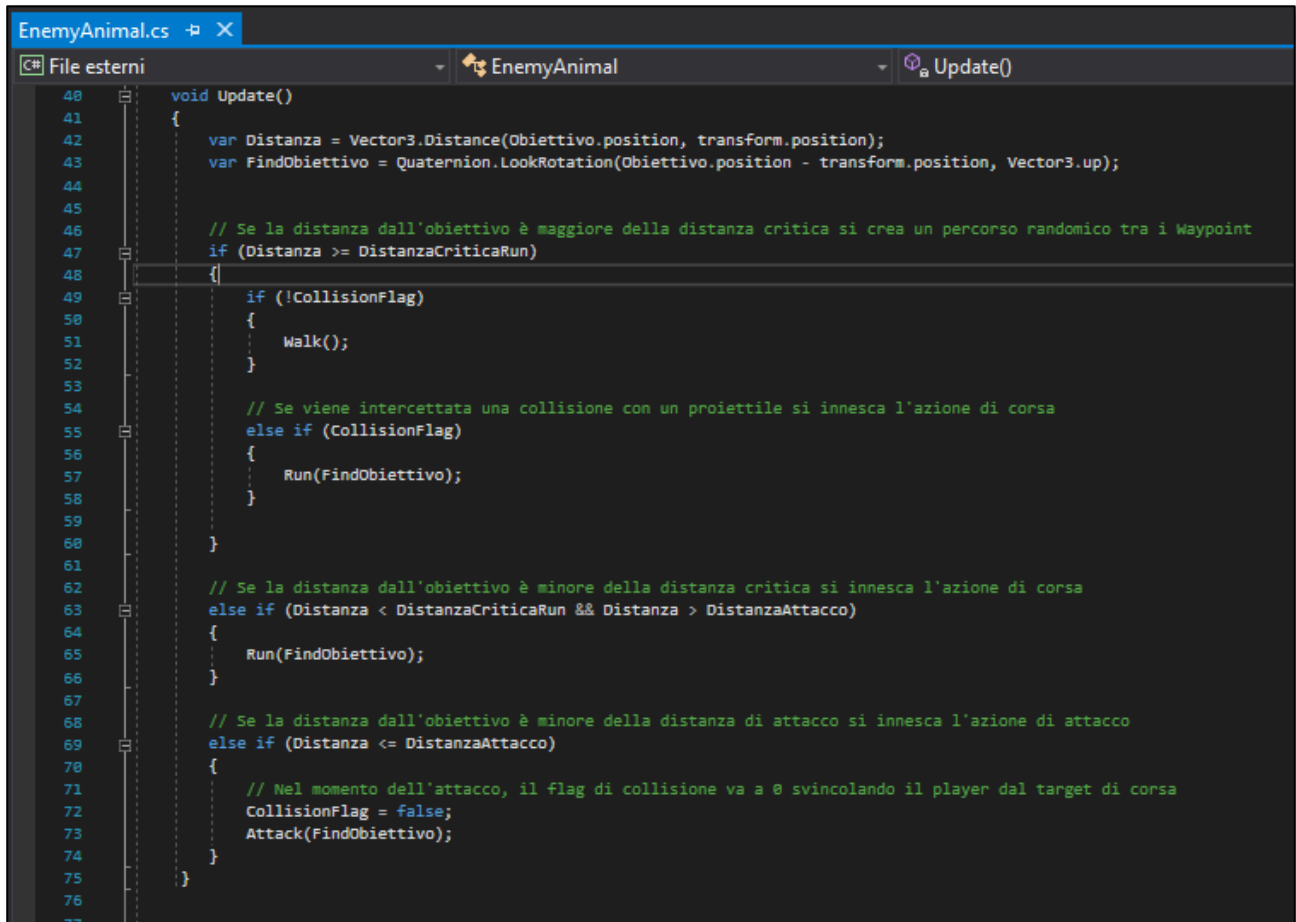
All'interno dell'oggetto Orso è necessario comunque includere il componente Animation per contenere tutte le clip utili.



Il flusso di controllo dell'orso, eseguito per ogni frame nell'ambito della funzione **Update**, è il seguente:

- Viene misurata la distanza dell'orso rispetto al player.
- Se la distanza è maggiore di una distanza critica prestabilita allora l'orso è in modalità **Walk**.
- Se la distanza è minore della distanza critica viene innescata la modalità **Run**.
- Se la distanza è minore di una distanza di attacco prestabilita allora si innesca la modalità **Attack**.
- Se viene intercettata una collisione con un proiettile da parte dell'orso si innesca la modalità **Run**.

Di seguito è riportata la funzione Update all'interno dello script 'EnemyAnimal':



```
40 void Update()
41 {
42     var Distanza = Vector3.Distance(Obiettivo.position, transform.position);
43     var FindObiettivo = Quaternion.LookRotation(Obiettivo.position - transform.position, Vector3.up);
44
45     // Se la distanza dall'obiettivo è maggiore della distanza critica si crea un percorso randomico tra i Waypoint
46     if (Distanza >= DistanzaCriticaRun)
47     {
48         if (!CollisionFlag)
49         {
50             Walk();
51         }
52
53         // Se viene intercettata una collisione con un proiettile si innesca l'azione di corsa
54         else if (CollisionFlag)
55         {
56             Run(FindObiettivo);
57         }
58     }
59
60     // Se la distanza dall'obiettivo è minore della distanza critica si innesca l'azione di corsa
61     else if (Distanza < DistanzaCriticaRun && Distanza > DistanzaAttacco)
62     {
63         Run(FindObiettivo);
64     }
65
66     // Se la distanza dall'obiettivo è minore della distanza di attacco si innesca l'azione di attacco
67     else if (Distanza <= DistanzaAttacco)
68     {
69         // Nel momento dell'attacco, il flag di collisione va a 0 svincolando il player dal target di corsa
70         CollisionFlag = false;
71         Attack(FindObiettivo);
72     }
73 }
74
75
76
77
```

Walk

Per simulare un comportamento 'intelligente' da parte dell'orso, quando si trova nello stato *walk* esso si sposta in maniera pseudo-casuale sulla mappa. Questo è possibile perché è stato creato un reticolo abbastanza fitto di **Waypoint** sulla mappa, ovvero di oggetti vuoti che rappresentano punti da raggiungere e a runtime viene creato un percorso scegliendo tra questi casualmente.

I Waypoint sono inseriti all'interno di un vettore; il punto da raggiungere viene inizializzato con un elemento di questo vettore. Ad ogni chiamata della funzione Walk viene fatto il seguente check:

- Se la distanza dal waypoint da raggiungere è maggiore di 1 viene ruotata la direzione dell'orso e viene settata la **destinazione** verso il Waypoint. Inoltre vengono settate **velocità** e **animazione**.
- Altrimenti, se la distanza è minore di 1 vuol dire che il waypoint è stato raggiunto. Viene scelto un Waypoint casualmente all'interno del vettore escludendo l'ultimo raggiunto.

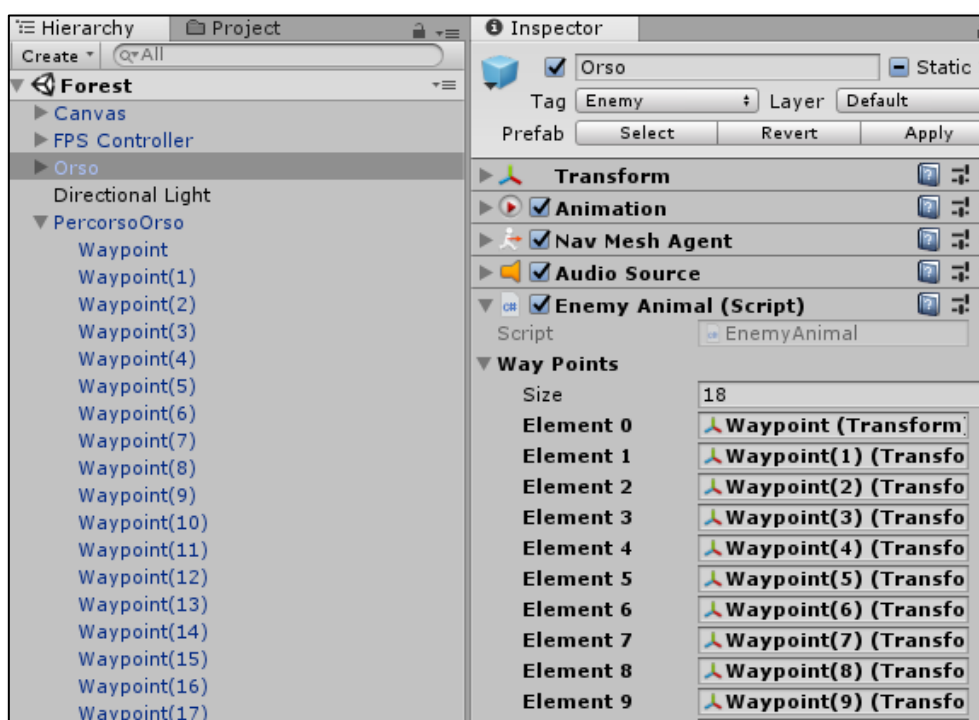
```

EnemyAnimal.cs* X
File esterni EnemyAnimal Run(Quaternion FindObiettivo)

76
77 public Transform[] WayPoints;
78
79 void Walk()
80 {
81     // Distanza dell'orso dal Waypoint successivo
82     var IrWp = Vector3.Distance(WayPoints[NextWP].position, transform.position);
83
84     // Se la distanza è maggiore di 1 ruota l'angolazione e setta la destinazione al WayPoint successivo
85     if (IrWp >= DistanzaWP)
86     {
87         var Angolo = Quaternion.LookRotation(WayPoints[NextWP].position - transform.position, Vector3.up);
88         transform.rotation = Quaternion.Slerp(transform.rotation, Angolo, Time.deltaTime * VelocitaGiro);
89         GetComponent<NavMeshAgent>().destination = WayPoints[NextWP].position;
90         GetComponent<NavMeshAgent>().speed = VelocitaWalk;
91         GetComponent<Animation>().Play(NomeAnimazioneWalk, PlayMode.StopAll);
92     }
93
94     // Altrimenti, a Waypoint raggiunto, si sceglie un altro WayPoint da raggiungere
95     else
96     {
97         // Prelievo di un Waypoint random diverso da quello attuale
98         int RandomWayPoint = Random.Range(0, WayPoints.Length - 1);
99         while (RandomWayPoint == NextWP)
100         {
101             RandomWayPoint = Random.Range(0, WayPoints.Length - 1);
102         }
103         NextWP = RandomWayPoint;
104     }
105 }
106
107

```

Di seguito è mostrato come viene riempito il vettore di Waypoint tramite l'Inspector:



Quando vengono create delle variabili nell'ambito di un componente *script*, all'interno del inspector è possibile configurare il loro valore; in questo caso viene fatto con degli altri oggetti del progetto.

Run

Quando viene chiamata la funzione run, in precedente, nello scope della funzione Update, è stata calcolata la posizione del Player, verso la quale l'orso deve correre. Questa viene passata come parametro di ingresso alla funzione; viene ruotata la **direzione**, settata la **destinazione** e vengono regolate **velocità** e **animazione** da eseguire.

```
void Run(Quaternion FindObiettivo)
{
    // Ruota l'angolazione in direzione del player e setta la destinazione in quel verso
    transform.rotation = Quaternion.Slerp(transform.rotation, FindObiettivo, Time.deltaTime * VelocitaGiro);
    GetComponent<NavMeshAgent>().destination = Obiettivo.position;
    GetComponent<NavMeshAgent>().speed = VelocitaRun;
    GetComponent<Animation>().Play(NomeAnimazioneRun, PlayMode.StopAll);
}
```

La funzione Run può essere chiamata o quando il Player entra in un **raggio** critico dall'orso prestabilito, oppure quando viene rilevata una collisione con un proiettile.

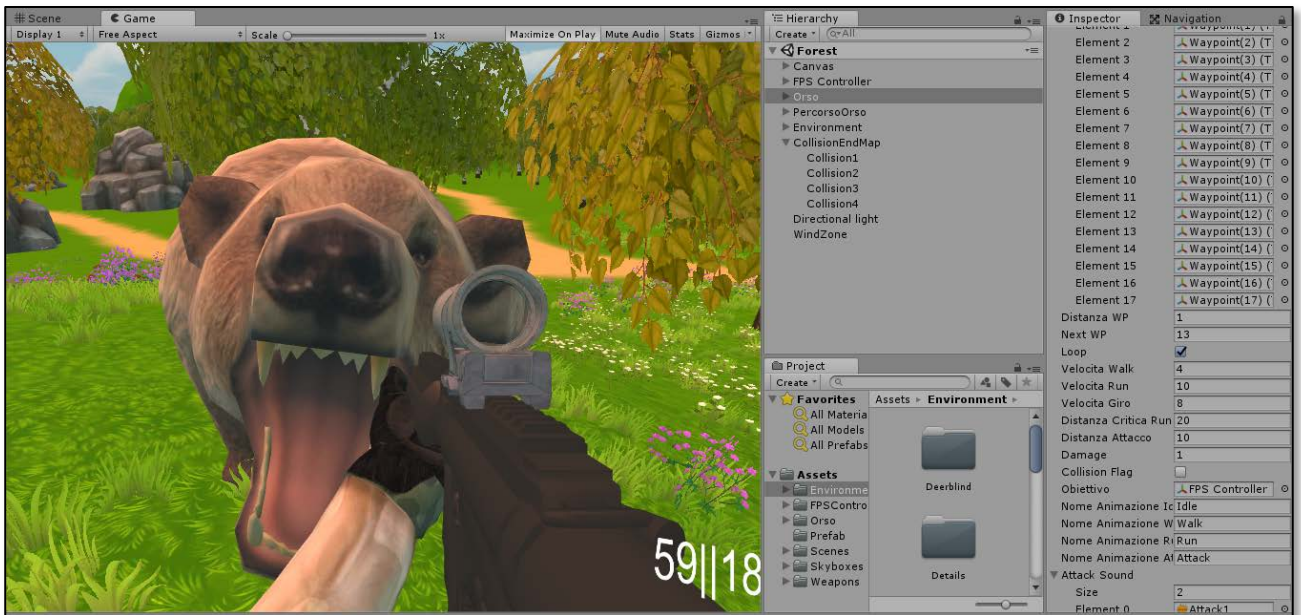
Attack

Dato che questa funzione viene chiamata quando l'orso raggiunge il player, la velocità del movimento viene settata a 0 e per ogni frame viene adattata l'angolazione dell'orso rispetto al player. Viene inoltre eseguita l'animazione dell'attacco e riprodotti i suoni relativi.

```
EnemyAnimal.cs
File esterni
EnemyAnimal
Attack(Quaternion FindObiettivo)

125 void Attack(Quaternion FindObiettivo)
126 {
127     Debug.Log("Attack State");
128
129     // Ruota l'angolazione in direzione del player e setta la velocità a zero
130     transform.rotation = Quaternion.Slerp(transform.rotation, FindObiettivo, Time.deltaTime * VelocitaGiro);
131     GetComponent<NavMeshAgent>().speed = 0;
132
133     // Esegue l'animazione dell'attacco se non già in esecuzione
134     if (!GetComponent<Animation>().IsPlaying(NomeAnimazioneAttacco))
135     {
136         GetComponent<Animation>().Play(NomeAnimazioneAttacco, PlayMode.StopAll);
137
138         // Esegue l'audio dell'attacco se non già in esecuzione
139         if (GetComponent<AudioSource>().isPlaying == false)
140         {
141             int index = Random.Range(0, AttackSound.Length);
142             GetComponent<AudioSource>().clip = AttackSound[index];
143             GetComponent<AudioSource>().volume = Random.Range(0.8f, 1f);
144             GetComponent<AudioSource>().pitch = Random.Range(0.8f, 1.1f);
145             GetComponent<AudioSource>().Play();
146         }
147     }
148 }
149
150
```

Uno screen dello stato di Attack dell'orso.



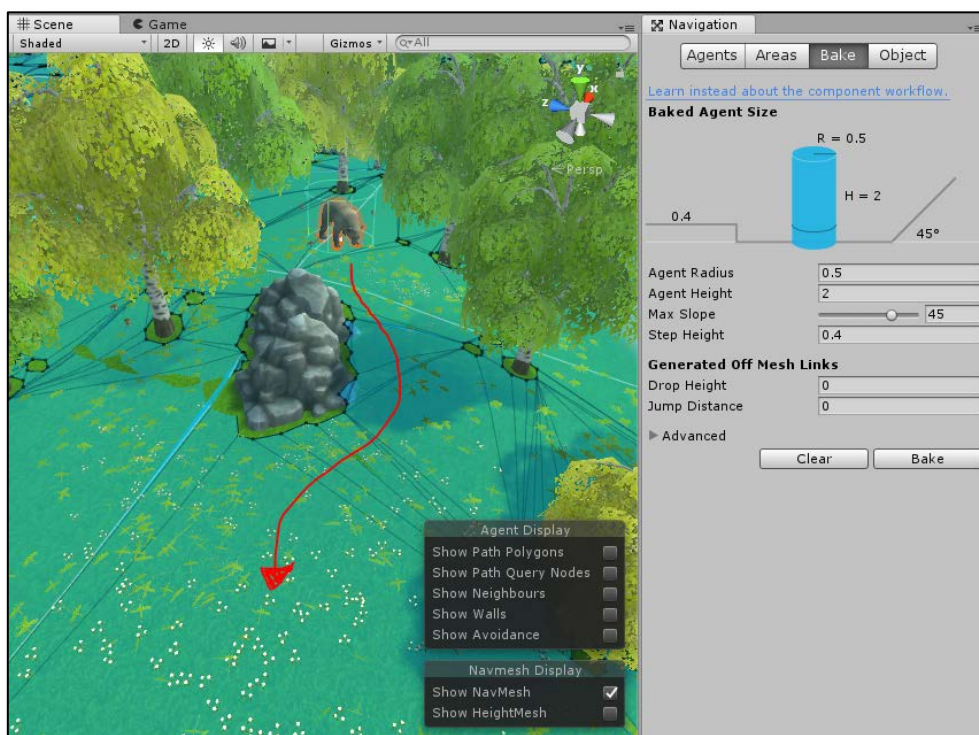
Navigation System

I target da raggiungere, come abbiamo visto in precedenza, sono definiti a run time a seconda della funzione chiamata (Waypoint/Player).

Per configurare invece i percorsi veri e propri da seguire tra i target si fa uso del componente '**Nav Mesh Agent**'. Esso permette di configurare destinazione e velocità del movimento.

```
GetComponent<NavMeshAgent>().destination = Obiettivo.position;  
GetComponent<NavMeshAgent>().speed = VelocitaRun;
```

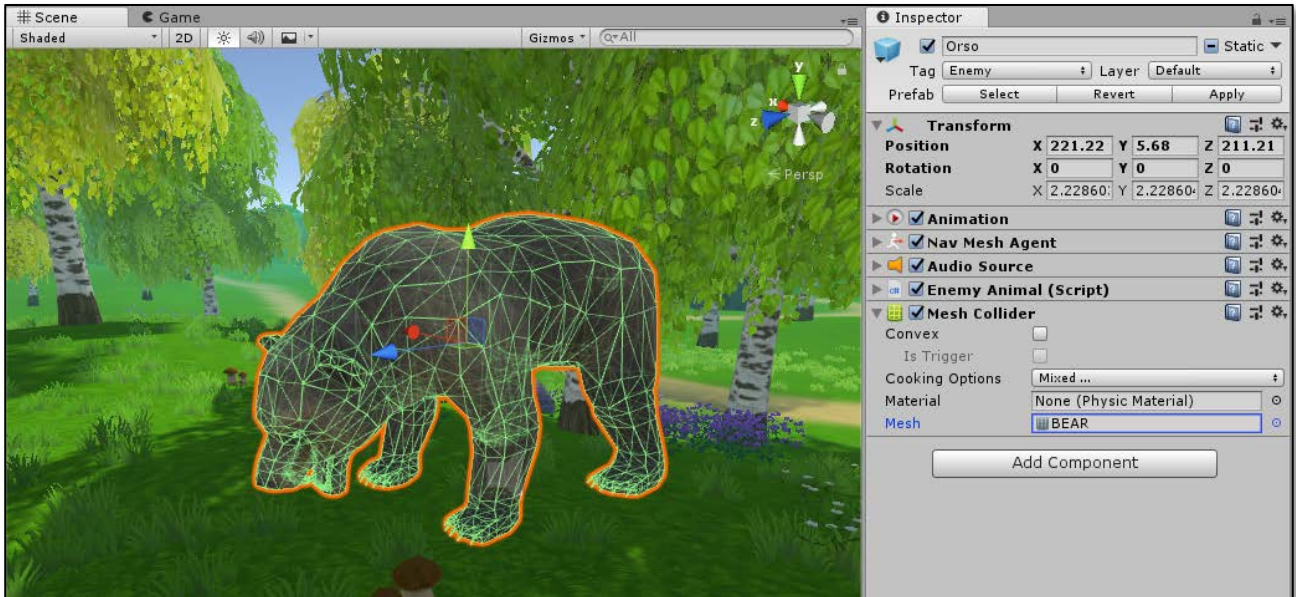
Interfacciandosi con la **NavMesh**, risulta inoltre indispensabile per gestire *automaticamente* la detection degli ostacoli durante i movimenti, e dunque evitarli.



Aperto il pannello Navigation, dopo aver configurato tutti i parametri dimensionali, è possibile, tramite il pulsante **Bake** creare la NavMesh, una struttura poligonale che descrive le *superfici calpestabili* del gioco e consente di trovare il percorso percorribile tra due posizioni

Collider

Per definire la fisica dell'orso è necessario aggiungere il componente **Collider** all'interno dell'oggetto in questione. Il Collider permette di definire la forma dell'oggetto ai fini delle collisioni fisiche. Per simulare collisioni più accurate è stato utilizzato un *Mesh Collider*.



Il **Mesh Collider** permette di costruire un Collider basandosi sulla forma della Mesh Asset; esso è molto più accurato di un collider primitivo, ad esempio a forma di cubo, ma richiede chiaramente un *overhead computazionale*.

Questo componente è indispensabile per rilevare le interazione dell'orso con il proiettile sparato e quindi per programmare il sistema di danno.

Damage System

Una volta definiti la dinamica dei percorsi dell'orso e la sua fisica è necessario introdurre una sistema che gestisca la logica del danno all'interno del gioco. Quando il player spara un colpo e riesce a centrare l'orso la salute dell'orso deve diminuire, fino ad un punto limite, dopo del quale l'orso muore.

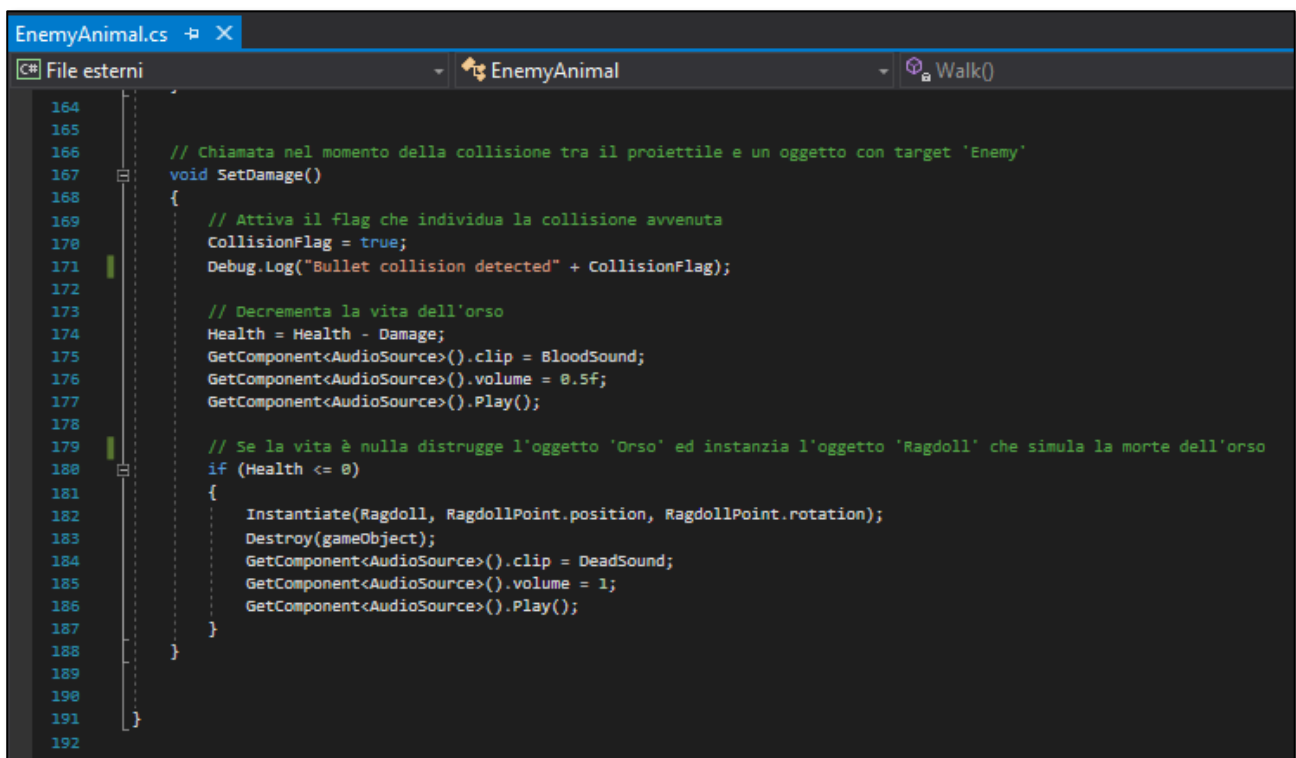
Essenzialmente la logica è basilare. Si deve rilevare la collisione tra il **proiettile** sparato e l'**orso**; questi due oggetti sono contrassegnati con due *target diversi*, 'bullet' e 'enemy'.

All'interno dell'arma attivata è eseguito il seguente script che verifica la collisione:

```
function Update () {
    var fwd = transform.TransformDirection(Vector3.forward);
    if ( Physics.Raycast( transform.position,transform.forward, hit)){
        var hitRotation = Quaternion.FromToRotation(Vector3.forward, hit.normal);

        if ( hit.collider.gameObject.tag == "Enemy" ){
            hit.collider.gameObject.SendMessage("SetDamage",Damage);
            Hitted.Doo=true;
            var CreateBlood=Instantiate(Particles.Blood, hit.point, hitRotation);
            CreateBlood.transform.parent = hit.collider.gameObject.transform;
        }
    }
}
```

Quando i due oggetti con i due target diversi entrano in collisione viene chiamata la funzione **SetDamage** dell'orso che gestisce il danno. Inoltre viene simulata una macchia di sangue tramite l'oggetto Particle.



```
164
165
166 // Chiamata nel momento della collisione tra il proiettile e un oggetto con target 'Enemy'
167 void SetDamage()
168 {
169     // Attiva il flag che individua la collisione avvenuta
170     CollisionFlag = true;
171     Debug.Log("Bullet collision detected" + CollisionFlag);
172
173     // Decrementa la vita dell'orso
174     Health = Health - Damage;
175     GetComponent<AudioSource>().clip = BloodSound;
176     GetComponent<AudioSource>().volume = 0.5f;
177     GetComponent<AudioSource>().Play();
178
179     // Se la vita è nulla distrugge l'oggetto 'Orso' ed istanzia l'oggetto 'Ragdoll' che simula la morte dell'orso
180     if (Health <= 0)
181     {
182         Instantiate(Ragdoll, RagdollPoint.position, RagdollPoint.rotation);
183         Destroy(gameObject);
184         GetComponent<AudioSource>().clip = DeadSound;
185         GetComponent<AudioSource>().volume = 1;
186         GetComponent<AudioSource>().Play();
187     }
188 }
189
190
191
192
```

La funzione **SetDamage** in primo luogo attiva il flag che individua la collisione avvenuta. Questo flag viene controllato dalla funzione update e innesca l'azione di corsa verso il target Player.

Inoltre decrementa la vita dell'orso di un unità, e riproduce il suono del sangue. Se la vita arriva a zero distrugge l'oggetto che rappresentava l'orso ed istanzia un oggetto **Ragdoll** che simula la sua morte.

