



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali  
Corso di Laurea in Informatica

Tesi di Laurea

PROGETTAZIONE E SVILUPPO DI UN  
AMBIENTE VIRTUALE IN UNITY

PLANNING AND DEVELOPMENT OF A  
VIRTUAL ENVIRONMENT IN UNITY

GIULIO FRINGUELLI

Relatore: *Francesco Tiezzi*

Anno Accademico 2024-2025



---

## INDICE

---

<b>ELENCO DELLE FIGURE</b>	<b>3</b>
<b>1 INTRODUZIONE</b>	<b>7</b>
<b>2 UNITY</b>	<b>9</b>
<b>2.1 Interfaccia . . . . .</b>	<b>9</b>
<b>2.1.1 Hierarchy . . . . .</b>	<b>10</b>
<b>2.1.2 Scene . . . . .</b>	<b>10</b>
<b>2.1.3 Game . . . . .</b>	<b>10</b>
<b>2.1.4 Inspector . . . . .</b>	<b>10</b>
<b>2.1.5 Project . . . . .</b>	<b>10</b>
<b>2.1.6 Console . . . . .</b>	<b>11</b>
<b>2.2 Lo spazio tridimensionale . . . . .</b>	<b>11</b>
<b>2.3 Namespace, classi, metodi e proprietà . . . . .</b>	<b>12</b>
<b>2.3.1 Object . . . . .</b>	<b>12</b>
<b>2.3.2 GameObject . . . . .</b>	<b>12</b>
<b>2.3.3 Component . . . . .</b>	<b>14</b>
<b>2.3.4 Input . . . . .</b>	<b>17</b>
<b>2.3.5 List&lt;T&gt; . . . . .</b>	<b>18</b>
<b>2.3.6 Physics . . . . .</b>	<b>18</b>
<b>2.3.7 Time . . . . .</b>	<b>19</b>
<b>2.3.8 Random . . . . .</b>	<b>21</b>
<b>2.3.9 Debug . . . . .</b>	<b>21</b>
<b>2.3.10 Vector3 . . . . .</b>	<b>22</b>
<b>2.4 Attributi . . . . .</b>	<b>23</b>
<b>2.5 Coroutine . . . . .</b>	<b>23</b>
<b>2.6 Material . . . . .</b>	<b>24</b>
<b>2.7 Prefabs . . . . .</b>	<b>25</b>
<b>3 SVILUPPO DELL'AMBIENTE VIRTUALE</b>	<b>27</b>
<b>3.1 Programmazione del giocatore . . . . .</b>	<b>27</b>
<b>3.1.1 Sviluppo dello script per la gestione del giocatore .</b>	<b>28</b>
<b>3.2 Partizionamento della mappa in chunk . . . . .</b>	<b>32</b>
<b>3.2.1 Generazione dei chunk . . . . .</b>	<b>33</b>
<b>3.2.2 Attivazione e disattivazione dei chunk . . . . .</b>	<b>44</b>
<b>3.3 Popolamento dei chunk tramite raycasting . . . . .</b>	<b>47</b>
<b>4 MOTIVAZIONI TECNICHE</b>	<b>59</b>
<b>4.1 Scelta dei modelli tridimensionali . . . . .</b>	<b>59</b>

4.2 Coroutine . . . . .	60
4.3 Sistema di generazione, attivazione e disattivazione . . . . .	61
4.4 Per quale motivo List e non Vector . . . . .	62
5 CONCLUSIONI . . . . .	63
5.1 Possibili sviluppi futuri . . . . .	63
BIBLIOGRAFIA . . . . .	65
RINGRAZIAMENTI . . . . .	67

---

## ELENCO DELLE FIGURE

---

Figura 2.1	Interfaccia di Unity. . . . .	9
Figura 2.2	Perpendicolarità tra gli assi. . . . .	11
Figura 2.3	Componente Transform. . . . .	15
Figura 2.4	Oggetto di gioco Capsule dotato di omonimo Collider in verde. . . . .	15
Figura 2.5	Il vettore trattato. . . . .	22
Figura 2.6	Composizione di più prefab. . . . .	26
Figura 3.1	Illustrazione del campo visivo della telecamera associata al giocatore. . . . .	28
Figura 3.2	Rappresentazione grafica della suddivisione descritta.	34
Figura 3.3	Debugging della generazione. . . . .	38
Figura 3.4	Il nuovo chunk. . . . .	41
Figura 3.5	Disposizione a L. . . . .	44
Figura 3.6	Modelli 3D rimasti attivi nonostante la disattivazione del chunk. . . . .	50
Figura 3.7	I nomi degli oggetti di gioco. . . . .	52
Figura 3.8	Prendendo come esempio il prefab Rock è possibile notare come si possano definire i limiti minimi e massimi con cui scegliere dimensione, rotazione e inclinazione.	54
Figura 3.9	Prefab di un sasso collocato sopra un albero, a sinistra del giocatore. . . . .	55
Figura 4.1	Low Poly e High Poly. . . . .	60
Figura 4.2	Visualizzazione dei cinque chunk attivi contemporaneamente, corrispondente al massimo carico di rendering previsto dal sistema. . . . .	62



*"We are the architects of our own reality."*  
— Jorge Luis Borges



# 1

---

## INTRODUZIONE

---

Lo sviluppo di ambienti virtuali estesi e dinamici rappresenta una delle sfide principali nella programmazione di videogiochi; tra i tanti ostacoli riscontrabili l'efficiente gestione delle risorse computazionali è probabilmente l'aspetto più rilevante per le prestazioni complessive.

In questa tesi verrà realizzato un mondo virtuale rappresentante un vasto bosco e verrà posta particolare attenzione sui metodi di gestione e ottimizzazione delle risorse, al fine di contenere il più possibile i costi computazionali.

Una delle tecniche che farà al caso nostro sarà proprio quella della *generazione procedurale*.

Come primo passo, verrà sviluppato un sistema di generazione per porzioni territoriali di forma quadratica; all'interno di ogni sezione saranno presenti delle aree lungo il perimetro, sulle quali, se il giocatore vi si trova, verrà generata la porzione di terreno adiacente.

Dopodiché verrà implementato un meccanismo di attivazione e disattivazione delle sezioni generate, il quale consisterà nel mantenere attiva una porzione quando confinante con quella in cui risiederà il giocatore, altrimenti verrà disattivata.

Infine avrà luogo la popolazione di queste, collocando sulla loro superficie, alcuni modelli tridimensionali raffiguranti elementi distintivi di un bosco; la procedura verrà realizzata attraverso la tecnica del raycasting, che comporta la proiezione di un numero finito di raggi - definito a priori in base alla categoria del modello - da una sorgente lungo una direzione stabilita.

Nel contesto informatico la generazione procedurale è un sistema impiegato per creare dati in modo algoritmico anziché manuale, in genere attraverso una combinazione di contenuti predefiniti e algoritmi con causalità controllata dal calcolatore. Nella computer grafica è comunemente usata per creare texture, modelli 3D o interi scenari; nei videogiochi permette di generare automaticamente grandi quantità di contenuti con

vantaggi che includono file più leggeri, ambienti più vasti e un gameplay meno prevedibile. [13] Questa affonda le proprie radici tra gli anni '60 e '70 e trova applicazioni videoludiche già nei primi anni '80, con titoli come *Maze Craze: A Game of Cops 'n Robbers* (1980) per Atari 2600. In questo gioco il giocatore veste i panni di un ladro intento a sfuggire ai poliziotti all'interno di un labirinto generato casualmente in tempo reale, aspetto che offriva esperienze di gioco sempre diverse. Nonostante le limitate risorse hardware della console, gli sviluppatori riuscirono a implementare queste tecniche, dimostrando la versatilità e l'efficacia della generazione procedurale anche su piattaforme con capacità ridotte. [14]

L'obiettivo di questa tesi è quindi quello di sperimentare con un algoritmo la generazione procedurale del terreno di un videogioco nel contesto del framework Unity.

La tesi è strutturata come segue: nel **Capitolo 2** verrà presentato il motore grafico Unity e le funzionalità impiegate, nel **Capitolo 3** sarà descritto dettagliatamente il processo di sviluppo dell'intero progetto, nel **Capitolo 4** giustificheremo le scelte progettuali del precedente, mettendo in risalto anche le limitazioni tecniche, infine nel **Capitolo 5**, sono riportate le conclusioni, alcune possibili prospettive di sviluppo futuro ed eventuali soluzioni ai limiti citati.

Il codice sorgente del progetto, comprensivo degli script C# e delle risorse necessarie alla generazione procedurale del terreno, è disponibile nel seguente repository GitHub: <https://github.com/giuliofringuelli-cpu/Progetto-Tesi>

# 2

---

## UNITY

---

Unity è uno dei motori di sviluppo più diffusi e versatili per la creazione di applicazioni interattive e videogiochi; supporta sia la grafica 2D che quella 3D, quest'ultima impiegata nel progetto, e si basa su un sistema a componenti che consente di costruire scene e oggetti. La versione adoperata è la 2022.3.59f1.

### 2.1 INTERFACCIA

All'avvio di Unity si possono notare sei aree principali: **Hierarchy**, **Scene**, **Game**, **Inspector**, **Project** e **Console**.

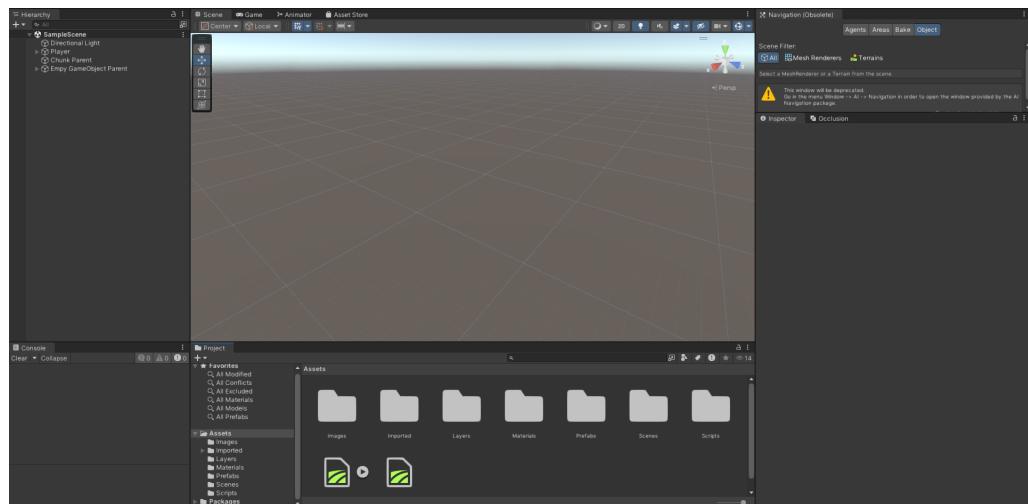


Figura 2.1: Interfaccia di Unity.

### 2.1.1 *Hierarchy*

È la gerarchia degli oggetti interni alla scena, i quali sono elencati nella rispettiva finestra e prendono il nome di **GameObject**: entità di base a cui è possibile associare uno o più **componenti**; quest'ultimi sono un insieme di dati responsabili dell'aspetto e del comportamento di un oggetto di gioco. Gli oggetti di gioco possono essere messi in relazione tra di loro formando una gerarchia: un oggetto **figlio** subisce molte delle trasformazioni effettuate sul **genitore** come per esempio disattivazione e cancellazione. [1]

Alla prima apertura di un progetto Unity nella gerarchia sono già presenti gli oggetti di gioco **Camera** e **Directional Light** che rappresentano rispettivamente il punto di vista dell'occhio che inquadra il gioco e il tipo di luce presente.

### 2.1.2 *Scene*

La finestra consente di manipolare direttamente gli oggetti di gioco presenti nella scena e visionare in tempo reale le modifiche apportate via Inspector.

### 2.1.3 *Game*

Il progetto mandato in esecuzione viene visualizzato in questa finestra, è pertanto una schermata di solo output e raffigura il risultato del lavoro svolto fino a quel momento.

### 2.1.4 *Inspector*

Questo pannello offre la visualizzazione e l'eventuale modifica di tutte le informazioni riguardanti il GameObject selezionato; è qui che si vedono tutti i componenti e le risorse che costituiscono un singolo oggetto di gioco.

### 2.1.5 *Project*

Contiene e organizza tutti i file utilizzati nello sviluppo del progetto; nel nostro caso sono state create sei cartelle in modo da mantenere separate le varie risorse in base alla loro tipologia ottenendo ordine e pulizia:

- **Scripts:** contiene i file di codice sorgente, i quali definiscono il comportamento degli oggetti e la logica del gioco;
- **Prefabs:** raccoglie i modelli tridimensionali e gli oggetti preconfigurati riutilizzabili all'interno della scena;
- **Materials:** include i materiali impiegati per definire l'aspetto superficiale degli oggetti;
- **Images:** comprende le texture e le immagini di supporto;
- **Imported:** contiene due sottocartelle dalle quali sono stati importati i modelli 3D utilizzati nel progetto;
- **Scenes:** ospita la scena principale del progetto.

#### 2.1.6 Console

Quest'ultimo pannello è essenziale per il debug poiché permette la visualizzazione dei messaggi di errore, compilazione ed esecuzione.

## 2.2 LO SPAZIO TRIDIMENSIONALE

In Unity 3D la posizione di ogni oggetto è descritta da una terna di coordinate ( $x, y, z$ ) che ne individua il punto esatto nello spazio virtuale. Tali coordinate si riferiscono a un sistema di riferimento cartesiano tridimensionale costituito da tre assi tra loro perpendicolari: ciò significa che, ponendosi idealmente sulla vetta di uno di essi potremmo osservare un angolo retto ( $90^\circ$ ) fra gli altri due.

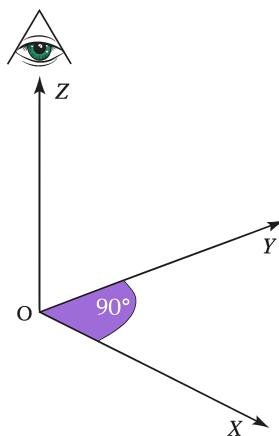


Figura 2.2: Perpendicolarità tra gli assi.

### 2.3 NAMESPACE, CLASSI, METODI E PROPRIETÀ

In primis è essenziale sapere che **UnityEngine** è il namespace che al suo interno possiede la collezione di classi e metodi basilari per sviluppare in Unity; per utilizzarla basta digitare la parola chiave `using` affiancata dal nome di esso. [1]

Trattiamo adesso esclusivamente le classi, i metodi e le proprietà utilizzate nel progetto.

#### 2.3.1 *Object*

È la superclasse principale e fornisce le funzionalità per la creazione, distruzione e gestione degli oggetti.

##### *Object.Instantiate*

Il metodo prende un oggetto di riferimento e crea una nuova istanza indipendente; è presente in diverse firme, ognuna delle quali contiene differenti parametri (overloading). La versione adottata nel progetto consente di specificare: l'oggetto originale di cui effettuare la clonazione, la posizione in cui apparirà, la rotazione e il componente Transform del GameObject di cui sarà figlio; la sua firma è riportata di seguito. [10]

---

```
public static Object Instantiate(Object original, Vector3
position, Quaternion rotation, Transform parent);
```

---

#### 2.3.2 *GameObject*

È una sottoclasse diretta della classe Object, pertanto ne eredita i metodi; viene utilizzata principalmente come contenitore per i componenti, il numero di riferimenti a questi è variabile in base alle esigenze. [1]

---

```
public sealed class GameObject : Object
```

---

La parola chiave `sealed` specifica che la classe non può essere ereditata da altre.

### *GameObject.GetComponent*

Il metodo recupera il riferimento a un componente del tipo specificato, fornendo il tipo come parametro di tipo al metodo generico; la sua firma è la seguente. [6]

---

```
public T GetComponent();
```

---

Nel seguente esempio si ha una classe C, la quale è un componente poiché eredita da MonoBehaviour, che a sua volta eredita da Behaviour e infine da Component; supponiamo che questa sia stata attaccata al GameObject G, il quale, da quel momento, ne possiede un riferimento. Nel file è presente la variabile privata controller di tipo CharacterController, all'interno del metodo Start() viene cercato su G un componente di tale tipo e, se trovato, viene effettuato l'assegnamento alla variabile controller.

---

```
public class C : MonoBehaviour
{
    private CharacterController controller;

    void Start()
    {
        controller = GetComponent<CharacterController>();
    }
}
```

---

### *GameObject.SetActive*

Il metodo attiva o disattiva il GameObject su cui viene effettuata la chiamata, l'attivazione o disattivazione dipendono dal valore del parametro fornito: se *true*, l'oggetto di gioco viene attivato, viceversa disattivato. La disattivazione disabilita qualsiasi componente del GameObject. [7]

---

```
public void SetActive(bool value);
```

---

### *GameObject.activeSelf*

Ritorna lo stato attivo locale del GameObject: *true* se attivo, *false* se inattivo; è una proprietà di sola lettura. Se un oggetto genitore viene disattivato anche tutti i suoi figli nella gerarchia risultano indirettamente inattivi;

tuttavia, se su di essi si richiama `GameObject.activeSelf`, il valore restituito sarà `true` poiché i figli non sono stati disattivati direttamente. Questo meccanismo può essere utile per determinare se la disattivazione di un oggetto è dovuta a quella del genitore o meno. [5]

---

```
public bool activeSelf;
```

---

### *Terrain*

È un tipo predefinito di `GameObject` che permette di creare paesaggi 3D gestendo altitudini e texture facendo impiego di strumenti per la modellazione.

Nel progetto è stato utilizzato come base per il posizionamento dei modelli tridimensionali impiegati; inoltre tramite il pannello `Terrain Settings` è possibile modificarne le dimensioni a proprio piacimento. [1]

### *Capsule*

Tipo specifico di oggetto di gioco a forma di capsula; nel nostro progetto è stato utilizzato per rappresentare il giocatore.

#### 2.3.3 *Component*

Come già accennato questi determinano funzionalità, aspetto e comportamento dei `GameObject`; ne esistono molteplici categorie, tuttavia in questa sottosezione ci limitiamo a esaminare quelli impiegati nel progetto.

---

```
public class Component : Object
```

---

### *Transform*

Rappresenta le trasformazioni 3D dell'oggetto nello spazio, più nel dettaglio ne riproduce scala, rotazione e posizione. Oltre a essere sottoclasse di `Component`, è importante osservare che risulta essere l'unico componente presente su ogni oggetto di gioco ed è impossibile rimuoverlo.

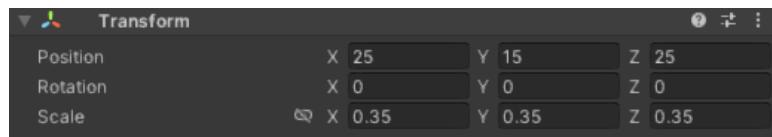


Figura 2.3: Componente Transform.

### *Collider*

È un componente che definisce una figura geometrica tridimensionale associata a un oggetto di gioco, impiegato ai fini di rilevare collisioni fisiche con altri oggetti. La forma del Collider non deve necessariamente seguire quella dell'oggetto: è sufficiente un collisore che approssimi quest'ultima e che al contempo garantisca un corretto sistema di rilevamento delle collisioni e non pesi eccessivamente nel suo calcolo fisico. Unity mette a disposizione tipologie di Collider prefabbricate che possono essere successivamente modellate per ottenere il risultato desiderato. Nei casi in cui sia necessaria estrema precisione si può ricorrere a un Mesh Collider, il quale riproduce fedelmente la geometria dell'oggetto, ma richiede un maggior costo computazionale.

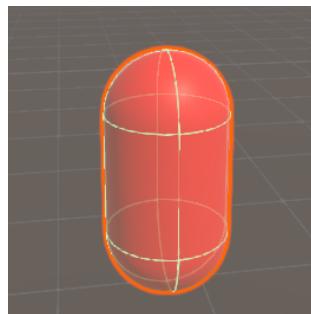


Figura 2.4: Oggetto di gioco Capsule dotato di omonimo Collider in verde.

---

```
public abstract class Collider : Component
```

---

La parola chiave *abstract* indica che non può essere istanziata direttamente, ma serve solo come classe base per i collisori specifici (BoxCollider, SphereCollider, CapsuleCollider, MeshCollider, ecc.), infatti ogni sottoclassa implementa le funzionalità concrete. [1]

### *Script*

È un file contenente codice sorgente, la cui estensione è .cs, rappresenta una classe scritta in C# che, come già detto nell'esempio precedente, è sottoclasse di MonoBehaviour.

Ogni script viene associato a un GameObject; le reazioni a determinati eventi o input dell'oggetto sono definite dai metodi della classe; nel momento in cui uno script viene creato si possono trovare al suo interno i due metodi seguenti:

- **Start()**: viene eseguito una sola volta all'inizio dell'esecuzione ed è utile per inizializzare variabili che dipendono da altri oggetti;
- **Update()**: viene eseguito continuamente durante l'esecuzione, impiegato per gestire input o logiche che devono aggiornarsi continuamente.

La frequenza con cui Update() viene richiamato dipende dal **frame rate**, cioè dal numero di fotogrammi elaborati dal motore grafico al secondo; tale valore dipende dall'hardware del computer, dalle impostazioni grafiche e dalla complessità della scena. [1]

### *MeshRenderer e MeshFilter*

Il MeshRenderer è un componente che si occupa della rappresentazione grafica di un oggetto tridimensionale nella scena; la sua funzione principale è quella di renderizzare la **mesh** associata, ovvero la struttura geometrica - composta da vertici, spigoli e facce - che definisce la forma dell'oggetto. Esso lavora in stretta collaborazione con il componente MeshFilter, il quale memorizza i dati della mesh, e con il Material, come vedremo più avanti, ne determina l'aspetto visivo. [1]

### *Behaviour e MonoBehaviour*

La classe Behaviour è una sottoclasse di Component e rappresenta la base per tutti i componenti che possono essere abilitati o disabilitati all'interno della scena. Tra le tante proprietà, essa ne fornisce una basilare che prende il nome di **enabled**, la quale permette di attivare o disattivare il comportamento di uno specifico componente senza rimuoverlo dal GameObject. [1]

La classe MonoBehaviour eredita da Behaviour e costituisce la classe madre di tutti gli script creati dall'utente in Unity, motivo per cui solo

le classi che derivano da MonoBehaviour possono essere aggiunte come componenti a un GameObject. [9]

### *CharacterController*

È una classe che consente di implementare il movimento e le collisioni di un oggetto di gioco; lo spostamento ha luogo solo quando il metodo CharacterController.Move della classe omonima viene richiamato. Il movimento del GameObject avviene nella direzione e verso del parametro passato come argomento, un Vector3. La gravità e la reazione a forze esterne che agiscono sul corpo dotato di CharacterController devono essere programmate. [1]

---

```
public CollisionFlags Move(Vector3 v);
```

---

Inoltre nel progetto è stata usata la variabile isGrounded della classe che stiamo trattando, questa indica se il CharacterController del GameObject a cui è attaccato, tocca il suolo. La variabile subisce un aggiornamento ogni volta che viene effettuata una chiamata a CharacterController.Move e restituisce *true* se il controller è entrato in collisione con il terreno sottostante durante il movimento. [2]

---

```
public bool isGrounded;
```

---

Per concludere si riporta la firma della classe.

---

```
public sealed class CharacterController : Collider
```

---

Questo pone fine alla trattazione delle classi fondamentali, procediamo dunque nell'esaminare quelle di supporto.

#### 2.3.4 *Input*

Unity gestisce l'input tramite la classe in questione; nel progetto è stato largamente utilizzato il metodo Input.GetAxis il quale restituisce un valore numerico compreso tra -1.0 e 1.0 che rappresenta l'intensità e la direzione del movimento lungo un determinato asse virtuale ("Horizontal" o "Vertical"). [8]

La seguente chiamata fa riferimento all'asse orizzontale, la variabile x assume valori prossimi a 1.0 quando l'utente preme il tasto →, e valori prossimi a -1.0 quando preme ←; se non viene premuto alcun tasto, allora x = 0.0. È fondamentale notare che le transizioni tra questi valori non sono

immediate: Unity interpola i risultati in maniera graduale, operazione capace di rendere i movimenti più fluidi e naturali.

Il comportamento è analogo per l'asse verticale, gestito dai tasti ↑/↓.

---

```
x = Input.GetAxis("Horizontal");
```

---

### 2.3.5 *List<T>*

Classe che rappresenta un elenco di elementi di un tipo generico; fornisce proprietà e metodi per gestire e manipolare gli elementi contenuti in essa. Una proprietà utile è `Count`, la quale permette di ricavare il numero di elementi presenti nella lista al momento del richiamo. [1]

Di seguito vengono mostrati alcuni metodi della classe.

<i>Metodo</i>	<i>Didascalia</i>
Add	Aggiunge un elemento alla fine della lista.
Clear	Rimuove tutti gli elementi dalla lista.
Insert	Inserisce un elemento nell'indice specificato.
Remove	Rimuove l'elemento passato come parametro.
Sort	Ordina in modo alfabetico o numerico gli elementi della lista.

### 2.3.6 *Physics*

È una classe che mette a disposizione un insieme di metodi e proprietà per impiegare la fisica.

#### *Physics.Raycast*

Il metodo proietta un raggio invisibile da un punto nello spazio e verifica se questo entra in collisione con uno o più oggetti della scena; nel progetto è stato utilizzato per posizionare i modelli 3D. Esistono diverse versioni del metodo; quella adottata nel progetto accetta come:

- **primo parametro:** un `Vector3` che rappresenta il punto di origine del raggio nello spazio tridimensionale, possiamo pertanto dire che questo è un vettore posizione;
- **secondo parametro:** un `Vector3` che indica direzione e verso di proiezione del raggio;

- **terzo parametro:** una variabile dichiarata con il modificatore **out** di tipo RaycastHit che viene riempita con i dettagli dell'impatto, nel caso questo avvenga, altrimenti rimane con i valori di default;
- **quarto parametro:** un valore numerico a virgola mobile che specifica la distanza massima percorribile dal raggio, denominabile come la sua estensione.

La struttura RaycastHit restituisce diverse informazioni utili sull'impatto, tra le seguenti possiamo considerare:

- `hit.point`: la posizione della collisione nello spazio;
- `hit.normal`: la normale alla superficie colpita, ovvero un vettore perpendicolare a questa nel punto di collisione;
- `hit.collider`: il collisore del GameObject sul quale è avvenuta la collisione;
- `hit.distance`: la distanza dall'origine al punto di impatto.

---

```
public static bool Raycast(Vector3 origin, Vector3 direction, out
    RaycastHit hitInfo, float distance);
```

---

### *Physics.gravity*

È un vettore tridimensionale rappresentato come variabile statica della classe Physics (ovvero rimane costante indipendentemente da dove viene richiamata), il cui valore di default è  $(0, -9.81, 0)$ , corrispondente all'accelerazione gravitazionale. [11]

---

```
public static Vector3 gravity;
```

---

### 2.3.7 *Time*

La classe Time fornisce importanti proprietà che consentono di lavorare con valori relativi al tempo. Come nella realtà, anche in una simulazione, il *tempo*, grandezza fisica fondamentale, è un elemento costante, sempre presente, che regola il ritmo con cui gli eventi vengono elaborati e aggiornati dal motore di gioco.

### *Time.deltaTime*

È una variabile di tipo statico. È un fatto sperimentalmente appurato che il frame rate dipende principalmente dalla potenza del computer su cui si lancia il progetto e, come detto in precedenza, la frequenza con cui `Update()` viene richiamato è esattamente pari al numero di fotogrammi al secondo poiché il metodo viene eseguito a ogni fotogramma del gioco; per questo motivo su elaboratori altamente prestanti il numero di chiamate al secondo può essere molto più alto rispetto a calcolatori meno efficienti e ciò si traduce in incoerenza comportamentale del programma: un oggetto di gioco in movimento possiede velocità diverse nei due computer, questo potrebbe avvenire anche all'interno di uno stesso in differenti istanti di tempo in quanto il numero di risorse disponibili non rimane sempre costante. [1]

Per risolvere tale problema occorre effettuare il prodotto del codice incaricato dello spostamento con `Time.deltaTime` all'interno di `Update()`, questo perché `Time.deltaTime` rappresenta il reciproco del frame rate.

Vediamo di chiarire il senso di quanto detto tramite un piccolo esempio.

---

```
void Update()
{
    0.transform.Translate(Vector3.forward * 5f);
}
```

---

Il corpo di `Update()` sposta in avanti l'oggetto di gioco 0 di 5 unità di spazio e, dal momento che questo viene chiamato una volta per ogni frame, 0 effettua uno spostamento di 5 unità a ogni frame. Vogliamo ora calcolare il valore dello spostamento totale in un secondo: sia  $f$  il frame rate espresso in fotogrammi al secondo (FPS),  $S$  lo spostamento effettuato a ogni chiamata dell'`Update` e  $S_t$  quello totale, allora vale che:

$$S_t = S \times f$$

Ad esempio:

$$f = 30 \frac{\text{frame}}{\text{s}} \Rightarrow S_t = 5 \frac{\text{unità}}{\text{frame}} \times 30 \frac{\text{frame}}{\text{s}} = 150 \frac{\text{unità}}{\text{s}}$$

$$f = 60 \frac{\text{frame}}{\text{s}} \Rightarrow S_t = 5 \frac{\text{unità}}{\text{frame}} \times 60 \frac{\text{frame}}{\text{s}} = 300 \frac{\text{unità}}{\text{s}}$$

Questo dimostra che, a parità di tempo, lo spostamento risulta maggiore nel secondo caso, sostenendo che la velocità dell'oggetto aumenta all'aumentare del frame rate.

Per rendere il movimento indipendente dal numero di frame al secondo, consideriamo ora il seguente codice:

---

```
void Update()
{
    0.transform.Translate(Vector3.forward * 5f * Time.deltaTime);
}
```

---

Sapendo che `Time.deltaTime` =  $\frac{1}{f}$ , si ha:

$$S_t = S \times f \times \frac{1}{f} = S = 5 \text{ unità/s}$$

Pertanto moltiplicando per `Time.deltaTime`, la velocità effettiva del movimento risulta costante e indipendente dal frame rate del sistema su cui viene eseguito il programma, e quindi dalla potenza di calcolo.

### 2.3.8 Random

È una classe statica che fornisce metodi e proprietà per generare numeri casuali, da non confondere con `System.Random` di C#.

#### *Random.range*

È un metodo presente in overloading, di seguito si vedono le versioni impiegate nel progetto. [12]

---

```
public static float Range(float minInclusive, float maxInclusive);
```

---

Genera un numero casuale in virgola mobile compreso tra due estremi, anch'essi tali; l'intervallo è inclusivo.

---

```
public static int Range(int minInclusive, int maxExclusive);
```

---

In questo caso il valore di ritorno è intero esattamente come i parametri, l'intervallo di generazione è esclusivo per quanto riguarda l'estremo maggiore.

### 2.3.9 Debug

Classe statica che contiene metodi utili al programmatore per il debugging. [3]

*Debug.Log*

Mostra un messaggio informativo nella console. [4]

---

```
public static void Log(object message);
```

---

Il metodo stampa il contenuto della variabile `message`, la quale è una stringa o un oggetto da convertire in essa per la visualizzazione.

2.3.10 *Vector3*

Dal punto di vista geometrico un vettore è un segmento caratterizzato da:

- **verso**: senso di percorrenza sul segmento;
- **direzione**: retta sulla quale giace il segmento;
- **modulo o intensità**: lunghezza del segmento.

In particolare in uno spazio a tre dimensioni, dato un punto di partenza  $P_1(x_1, y_1, z_1)$  e un punto di arrivo  $P_2(x_2, y_2, z_2)$ , il vettore che li collega è definito come la loro differenza:

$$\vec{v} = P_2 - P_1 = (x_2 - x_1, y_2 - y_1, z_2 - z_1) = (V_x, V_y, V_z)$$

dove  $(V_x, V_y, V_z)$  rappresentano rispettivamente le componenti del vettore lungo gli assi  $x$ ,  $y$  e  $z$ .

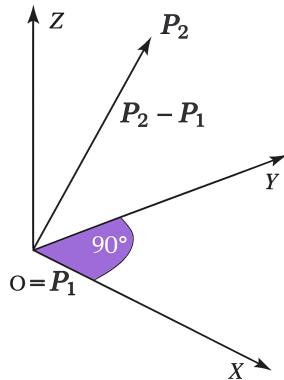


Figura 2.5: Il vettore trattato.

In Unity la classe `Vector3` consente di definire vettori a tre dimensioni come nell'esempio sopra; questa può risultare utile per descrivere la posizione di un punto nello spazio tridimensionale ed esprimerne la velocità.

Le tre coordinate sono di tipo `float`, il che significa che i movimenti e la scala degli oggetti possono essere microscopici. Relativamente al componente Transform, il campo Position - visibile all'interno della sua interfaccia - è proprio un `Vector3`. [1]

L'istruzione seguente rappresenta un vettore che punta dall'origine degli assi a un punto che si trova lungo l'asse X distante 5 unità; l'intensità del vettore è 5.

---

```
Vector3 v = new Vector3(5, 0, 0);
```

---

*Vector3.magnitude*

Proprietà che restituisce il modulo del vettore su cui viene richiamata, calcolato come:

$$|\vec{v}| = \sqrt{V_x^2 + V_y^2 + V_z^2}$$

---

```
public float magnitude;
```

---

## 2.4 ATTRIBUTI

Sono indicatori dichiarativi applicabili a classi, metodi e variabili; nel progetto sono stati impiegati i seguenti attributi. [1]

- `[SerializeField]`: permette di rendere visibili e modificabili nell'Inspector variabili `private` o `protected`, senza doverle dichiarare pubbliche;
- `[Header("Titolo")]`: aggiunge un'intestazione descrittiva nel pannello Inspector, utile per raggruppare le variabili di uno script sulla base della loro funzionalità migliorando organizzazione e leggibilità nella finestra.

## 2.5 COROUTINE

È un tipo speciale di funzione che può sospendere la propria esecuzione per un determinato tempo specificabile dal programmatore. [1] Per dichiararla si deve utilizzare il valore di ritorno `IEnumerator` ed è possibile avviare e fermare l'esecuzione rispettivamente mediante il richiamo di

`StartCoroutine()` e `StopCoroutine()`, mentre per sospornerne l'esecuzione di un certo numero di secondi `s` è necessario scrivere l'istruzione `yield return new WaitForSeconds(s);`.

---

```
public class C : MonoBehaviour
{
    int s;
    void Start()
    {
        s = 5;
        StartCoroutine(F());
    }

    IEnumerator F()
    {
        while (true)
        {
            Debug.Log("Hello World!");
            yield return new WaitForSeconds(s);
        }
    }
}
```

---

La coroutine `F()` viene chiamata una sola volta in `Start()` e ogni cinque secondi viene eseguita e stampa il messaggio `Hello World!`.

Nel progetto è stata impiegata per sostituire il metodo `Update()`.

## 2.6 MATERIAL

Determina l'aspetto visivo delle superfici assegnando loro colore, texture e intensità di riflessione. Nel progetto è stato esclusivamente utilizzato per attribuire il colore rosso alla capsula che rappresenta il personaggio e per colorare di verde il terreno; ciononostante si è ritenuto utile informare il lettore su cosa rappresentino le texture e l'intensità di riflessione. Le prime sono immagini bidimensionali applicate sulla superficie di un oggetto conferendo a questo maggiore realismo e dettaglio, il loro uso è spesso affiancato a quello delle normal map (immagini RGB che simulano la complessità del rilievo nelle superfici), le altre costituiscono l'indice di riflettanza della superficie selezionata. [1]

Sia chiaro che `Material` è una classe, spesso erroneamente considerata come un vero e proprio componente, ma questa nella realtà dei fatti non

risulta essere sottoclasse di Component o classi derivate da quest'ultima. Un materiale infatti non viene attaccato direttamente a un GameObject, bensì assegnato a un componente che gestisce la resa grafica, che come visto è il MeshRenderer.

## 2.7 PREFABS

Sono GameObject prefabbricati e salvati all'interno delle cartelle del progetto; possono essere inseriti in scena sia da codice che tramite trascinamento dalle cartelle in cui risiedono nell'Editor.

La loro utilità emerge quando uno stesso modello deve comparire più volte nella stessa scena; in questo caso si parla di *istanze* del prefab e possono essere considerate come cloni dell'originale, ma comunque modificabili singolarmente. [1]

A titolo di esempio la classe C riportata di seguito genera cinque istanze del prefab in posizioni casuali calcolate dal metodo RandomPosition.

---

```
public class C : MonoBehaviour
{
    private GameObject G;
    private GameObject prefab;

    void Start()
    {
        Vector3 positionSpawn;
        for (int i = 0; i < 5; i++)
        {
            positionSpawn = RandomPosition();
            G = GameObject.Instantiate(prefab, positionSpawn,
                transform.rotation);
        }
    }

    Vector3 RandomPosition()
    {
        Vector3 position = new Vector3(
            Random.Range(0, 50),
            0,
            Random.Range(0, 50)
        );
        return position;
    }
}
```

}

---

Un prefab può essere una composizione di più GameObject.

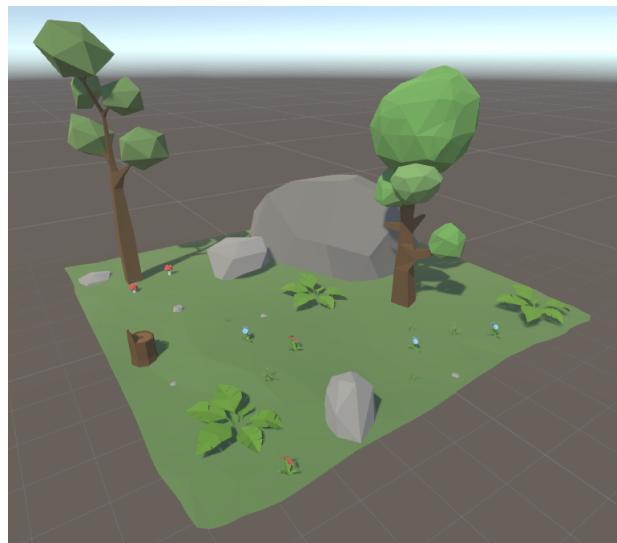


Figura 2.6: Composizione di più prefab.

# 3

---

## SVILUPPO DELL'AMBIENTE VIRTUALE

---

Il capitolo descrive le fasi di sviluppo dell'ambiente virtuale, realizzato attraverso una combinazione di operazioni svolte direttamente nell'interfaccia del motore grafico e attività di programmazione in C#. I risultati ottenuti vengono illustrati nel dettaglio, includendo anche gli script prodotti affiancati da una loro analisi.

### 3.1 PROGRAMMAZIONE DEL GIOCATORE

Come primo passo, nel motore grafico sono stati importati i pacchetti *Simple Nature Pack* e *Simple Low Poly Nature* permettendo l'impiego di un'ampia collezione di modelli tridimensionali.

Successivamente è stato creato un *GameObject* vuoto, ovvero privo di componenti a eccezione del *Transform*, e denominato *Chunk Parent*, la cui funzione è quella di contenere tutti i *chunk* dei quali rappresenta il nodo genitore nella gerarchia della scena; questa scelta è stata dettata dalla necessità di possedere un'organizzazione ordinata così da facilitarne la gestione.

In seguito è stato generato l'oggetto di gioco *Terrain* impostando le sue dimensioni a  $50 \times 50 \times 100$  unità attraverso l'*Inspector*; dopodiché è stato inserito un oggetto di tipo *Capsule* destinato a rappresentare il giocatore, al quale verrà successivamente associato lo script *PlayerController* per la gestione dei movimenti. Per distinguere correttamente la capsula nella mappa - il cui terreno sarà di colore verde - le è stato assegnato un materiale di cromia rossa; le sue dimensioni sono state impostate a  $0.35 \times 0.35 \times 0.35$  unità.

Le dimensioni del terreno e della capsula sono state impostate confrontandole con quelle predefinite dei modelli 3D all'interno della finestra *Scene*. Poiché i modelli risultavano molto più piccoli e il loro numero era elevato, per mantenere la proporzionalità, si è preferito ridimensiona-

re il terreno e la capsula, anziché intervenire singolarmente su ciascun modello, così da ottimizzare i tempi di lavoro.

La seguente operazione ha visto l'aggiunta di una telecamera che seguisse i movimenti del giocatore, manovra che è stata possibile rendendo l'oggetto di gioco Camera figlio di Player; infine questa è stata posizionata in modo tale da visualizzare in terza persona il giocatore con un angolo di inclinazione pari a 20° circa rispetto al piano orizzontale.

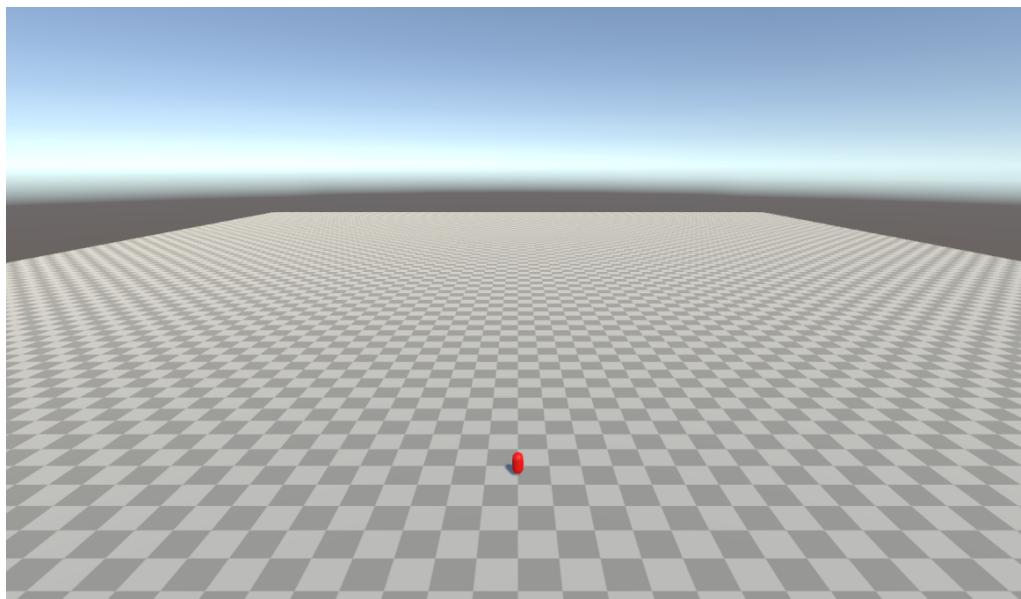


Figura 3.1: Illustrazione del campo visivo della telecamera associata al giocatore.

### 3.1.1 Sviluppo dello script per la gestione del giocatore

Giunti a questo punto ha avuto origine lo sviluppo di `PlayerController.cs`, che come prima operazione è stato associato all'oggetto di gioco Player; di seguito sono riportati il codice e una sua analisi.

---

```
public class PlayerController : MonoBehaviour
{
    [SerializeField] float speed;

    private CharacterController controller;
    private Vector3 directionMovement;
    private Vector3 velocity;

    private float moveX;
```

```

private float moveZ;

void Start()
{
    controller = GetComponent<CharacterController>();
}

void Update()
{
    moveX = Input.GetAxis("Horizontal");
    moveZ = Input.GetAxis("Vertical");

    directionMovement = transform.right * moveX +
        transform.forward * moveZ;

    if (directionMovement.magnitude > 1f)
    {
        directionMovement.Normalize();
    }

    velocity.x = directionMovement.x * speed;
    velocity.z = directionMovement.z * speed;

    if (controller.isGrounded)
    {
        velocity.y = 0.0f;
    }
    else
    {
        velocity.y += Physics.gravity.y * Time.deltaTime;
    }

    controller.Move(velocity * Time.deltaTime);
}
}

```

---

Il ruolo della variabile intera `speed` verrà approfondito subito dopo aver descritto le variabili che seguono:

- `controller`: riferimento al componente `CharacterController` di `Player`;

- `directionMovement`: vettore che indica le direzioni in cui deve essere effettuato lo spostamento;
- `velocity`: vettore velocità che definisce la rapidità del movimento del giocatore.

Tornando alla variabile sopra citata, questa non rappresenta direttamente la velocità ma un coefficiente scalare che, moltiplicato per `directionMovement`, consente il recupero di `velocity`; il suo valore può essere assegnato direttamente dall'Inspector di Unity dal momento che è presente l'attributo [SerializeField].

Nel motore grafico è stato poi selezionato l'oggetto di gioco `Player` e aggiunto ad esso il componente `CharacterController`, il quale di default possiede già un proprio `Collider` a forma di capsula che può essere adattato secondo le proprie esigenze.

---

```
moveX = Input.GetAxis("Horizontal");
moveZ = Input.GetAxis("Vertical");
```

---

Le prime due righe si occupano di registrare l'input da tastiera mediante il metodo `Input.GetAxis`.

---

```
directionMovement = transform.right * moveX + transform.forward *
moveZ;
```

---

L'istruzione sopra calcola la direzione del movimento in funzione dell'orientamento del personaggio; in particolare `transform.right` rappresenta il vettore unitario orientato verso destra, mentre `transform.forward` è il vettore unitario orientato in avanti; moltiplicandoli rispettivamente per `moveX` e `moveZ` e sommando i risultati, si ottiene il vettore `directionMovement`, che descrive tutte le possibili direzioni di movimento sul piano orizzontale. Ad esempio, supponendo:

---

```
moveX = -1.0f;
moveZ = 0.0f;
```

---

si ottiene che:

---

```
directionMovement = transform.right * (-1.0) + transform.forward
* (0.0);
```

---

equivalente a:

---

```
directionMovement = -transform.right;
```

---

In questo caso `directionMovement` punta esattamente nella direzione opposta a `transform.right`, cioè verso sinistra rispetto al sistema di riferimento di Unity.

---

```
velocity = directionMovement * speed;
```

---

L'istruzione permette di ricavare il vettore velocità del giocatore effettuando un banale prodotto tra `directionMovement` e `speed`. A tal riguardo eseguendo in un primo momento il codice è stato riscontrato un problema legato alla gestione della velocità nei movimenti diagonali: premendo simultaneamente due tasti direzionali non opposti (ad esempio avanti e destra), il modulo della velocità risultante era maggiore rispetto a quella lungo un singolo asse. Il comportamento era dovuto al fatto che `Input.GetAxis("Horizontal")` e `Input.GetAxis("Vertical")` - come già detto - restituiscono valori compresi tra  $-1.0$  e  $1.0$ ; conseguentemente in relazione all'esempio, se l'utente premeva:

- $\uparrow$ :

---

```
directionMovement = (0, 0, 1);
```

---

- $\rightarrow$ :

---

```
directionMovement = (1, 0, 0);
```

---

In entrambi i casi il modulo del vettore era unitario, ma se premeva contemporaneamente:

- $\uparrow$  e  $\rightarrow$ :

---

```
directionMovement = (1, 0, 1);
```

---

Il suo modulo corrispondeva a:

$$\sqrt{1^2 + 0^2 + 1^2} = \sqrt{2} \approx 1.41$$

Segue che veniva persa l'unitarietà del modulo della velocità traducendosi nell'aumento citato; onde evitare tale anomalia `directionMovement` viene normalizzato solo se il suo modulo è strettamente maggiore di uno.

---

Listing 1: Normalizzazione del vettore risultante.

---

```
if (directionMovement.magnitude > 1f)
```

---

```
{
    directionMovement.Normalize();
}
```

---

Viene poi gestita la gravità su Player: se questo si trova a terra `velocity.y` viene azzerata, altrimenti viene applicata servendosi delle classi `Physics` e `Time`:

---

```
velocity.y += Physics.gravity.y * Time.deltaTime;
```

---

Infine `velocity` viene passato come argomento del metodo `CharacterController.Move` richiamato su `controller`.

---

```
controller.Move(velocity * Time.deltaTime);
```

---

È utile osservare che in una prima versione di `PlayerController.cs` non era presente la gravità; l'introduzione di quest'ultima ha reso necessario aggiornare separatamente le componenti del vettore `velocity` altrimenti non veniva applicata correttamente e il personaggio non cadeva quando sospeso in aria. Il problema nasceva dal fatto che in origine veniva effettuata la riassegnazione completa della velocità:

---

```
velocity = directionMovement * speed;
```

---

questa però, causava l'azzeramento costante della sua componente verticale poiché in ogni istante di tempo `directionMovement.y = 0` e quindi era come se non venisse mai eseguita l'istruzione:

---

```
velocity.y += Physics.gravity.y * Time.deltaTime;
```

---

L'alterazione è stata risolta aggiornando separatamente le componenti di `velocity`.

---

```
velocity.x = directionMovement.x * speed;
velocity.z = directionMovement.z * speed;
```

---

### 3.2 PARTIZIONAMENTO DELLA MAPPA IN CHUNK

In primo luogo è stato creato il prefab del chunk. Per far ciò è stata generata un'immagine a tinta unita di colore verde chiaro utilizzando il software *Paint.NET* e successivamente importata nella cartella `Images`

del progetto; l'immagine è stata poi applicata a Terrain tramite il pannello *Paint Texture* dell'Inspector. Infine Terrain è stato rinominato in Chunk nella Hierarchy e per conclusione, trascinato nella cartella Prefabs ottenendo di fatto il prefab.

### 3.2.1 Generazione dei chunk

Nella sottosezione è iniziata la fase di stesura dello script denominato `ChunkHandler` volta a implementare il sistema di generazione dei chunk.

**Listing 2:** Dichiarazione delle variabili in `ChunkHandler.cs`

---

```
[SerializeField] GameObject player;
[SerializeField] GameObject chunkParent;
[SerializeField] GameObject chunkPrefab;
[SerializeField] float checkInterval;

private List<GameObject> chunkMap = new();
```

---

Il ruolo delle variabili è il seguente:

- `player`: riferimento al giocatore, indispensabile per poter accedere ai suoi componenti;
- `chunkParent`: riferimento al contenitore gerarchico in Unity creato nella sezione precedente;
- `chunkPrefab`: riferimento al prefab usato per generare nuovi chunk dinamicamente;
- `checkInterval`: variabile `float` che indica la frequenza temporale con cui viene controllata la posizione del giocatore;
- `chunkMap`: lista che tiene traccia dei chunk generati.

Successivamente è stato sviluppato il metodo `CheckPlayerPosition()`, il quale determinerà i chunk che dovranno essere generati, attivati e disattivati sulla base della posizione del giocatore; questo è in realtà una coroutine che viene richiamata come descritto di seguito.

---

```
void Start()
{
    StartCoroutine(CheckPlayerPosition());
}
```

---

All'interno della coroutine è stata inserita l'istruzione:

---

```
yield return new WaitForSeconds(checkInterval);
```

---

Questa istruzione fa sì che la coroutine venga messa in pausa per un intervallo di tempo pari al valore della variabile checkInterval.

All'interno del metodo viene creata e inizializzata la variabile di tipo `float border` che rappresenta lo spessore del bordo del chunk sensibile alla generazione di nuovi chunk adiacenti. In altre parole `border` definisce una fascia perimetrale, entro la quale, se il giocatore vi entra, viene attivata la creazione di un nuovo chunk; la posizione di questa fascia determina in quale direzione il nuovo chunk verrà generato.

Ogni chunk è suddiviso in tre tipi di aree:

- **verde**: quando il giocatore si trova all'interno non viene generato nessun nuovo chunk rispetto a quelli già presenti;
- **arancioni**: sono le fasce che circondano la superficie verde sui quattro lati; se il giocatore entra in una di queste viene generato il chunk adiacente alla fascia corrispondente;
- **blu**: si trovano nei quattro angoli del chunk, se il giocatore si posiziona in una di queste vengono attivati i due chunk adiacenti, uno lungo l'asse X e uno lungo l'asse Z; hanno forma quadratica e il loro lato è determinato dalla variabile `border`.

Il comportamento descritto è stato implementato mediante una serie di verifiche logiche che confrontano le coordinate del giocatore con quelle del chunk.

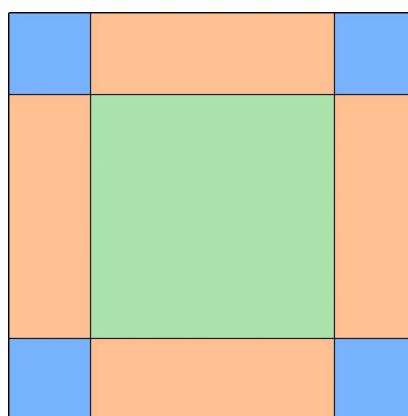


Figura 3.2: Rappresentazione grafica della suddivisione descritta.

Spostiamoci per un attimo nel metodo `Start()` in cui al suo interno viene istanziato il chunk iniziale, posizionato nell'origine del sistema di coordinate Unity (`Vector3.zero`), privo di rotazione (`Quaternion.identity`) e figlio di `chunkParent`; il chunk viene quindi aggiunto alla lista `chunkMap`.

---

```
void Start()
{
    GameObject firstGeneratedChunk = Instantiate(chunkPrefab,
        Vector3.zero, Quaternion.identity, chunkParent.transform);
    chunkMap.Add(firstGeneratedChunk);

    StartCoroutine(CheckPlayerPosition());
}
```

---

A questo punto è ripreso lo sviluppo del metodo `ChunkHandler.CheckPlayerPosition` e implementato il meccanismo di generazione dei chunk, il cui codice è riportato sotto e accompagnato da una sua analisi.

---

```
IEnumerator CheckPlayerPosition()
{
    float border = 10f;

    while (true)
    {
        foreach (GameObject c in chunkMap)
        {
            if (player.transform.position.x >=
                c.GetComponent<Terrain>().GetPosition().x &&
                player.transform.position.x <=
                c.GetComponent<Terrain>().GetPosition().x + distance &&
                player.transform.position.z >=
                c.GetComponent<Terrain>().GetPosition().z &&
                player.transform.position.z <=
                c.GetComponent<Terrain>().GetPosition().z +
                c.GetComponent<Terrain>().terrainData.size.z)
            {
                Debug.Log("Generazione di un nuovo chunk alla sinistra
dell'attuale...");
            }
            if (player.transform.position.x >=
                c.GetComponent<Terrain>().GetPosition().x +
                c.GetComponent<Terrain>().terrainData.size.x - distance
```

---

```
&& player.transform.position.x <=
c.GetComponent<Terrain>().GetPosition().x +
c.GetComponent<Terrain>().terrainData.size.x &&
player.transform.position.z >=
c.GetComponent<Terrain>().GetPosition().z &&
player.transform.position.z <=
c.GetComponent<Terrain>().GetPosition().z +
c.GetComponent<Terrain>().terrainData.size.z)
{
    Debug.Log("Generazione di un nuovo chunk alla destra
dell'attuale...");
}
if (player.transform.position.z >=
    c.GetComponent<Terrain>().GetPosition().z &&
    player.transform.position.z <=
    c.GetComponent<Terrain>().GetPosition().z + distance &&
    player.transform.position.x >=
    c.GetComponent<Terrain>().GetPosition().x &&
    player.transform.position.x <=
    c.GetComponent<Terrain>().GetPosition().x +
    c.GetComponent<Terrain>().terrainData.size.x)
{
    Debug.Log("Generazione di un nuovo chunk in basso
rispetto all'attuale...");
}
if (player.transform.position.z >=
    c.GetComponent<Terrain>().GetPosition().z +
    c.GetComponent<Terrain>().terrainData.size.z - distance
    && player.transform.position.z <=
    c.GetComponent<Terrain>().GetPosition().z +
    c.GetComponent<Terrain>().terrainData.size.z &&
    player.transform.position.x >=
    c.GetComponent<Terrain>().GetPosition().x &&
    player.transform.position.x <=
    c.GetComponent<Terrain>().GetPosition().x +
    c.GetComponent<Terrain>().terrainData.size.x)
{
    Debug.Log("Generazione di un nuovo chunk in alto
rispetto all'attuale...");
}
yield return new WaitForSeconds(checkInterval);
```

```

    }
}

```

---

All'interno di un loop `while(true)` eseguito a intervalli regolari definiti da `checkInterval`, il metodo scansiona i chunk presenti in `chunkMap` e controlla la posizione del giocatore rispetto ai bordi del chunk iniziale. Se il giocatore entra in una delle aree perimetrali, viene mostrato un messaggio di debug che simula la creazione di un nuovo chunk.

Di seguito viene analizzato esclusivamente il *primo costrutto if* dal momento che questi risultano pressoché identici, differenziandosi solo per la disposizione di componenti e variabili; l'idea di fondo resta comunque la stessa.

Lo studio è suddiviso in segmenti per semplificare la comprensione; si ricorda che questi sono legati tramite congiunzione logica.

---

```

player.transform.position.x >=
    c.GetComponent<Terrain>().GetPosition().x

```

---

Controlla che la coordinata `x` del giocatore sia maggiore o uguale alla coordinata `x` della posizione del chunk; in pratica verifica se il giocatore si trova a destra o sul bordo sinistro del chunk.

---

```

player.transform.position.x <=
    c.GetComponent<Terrain>().GetPosition().x + border

```

---

Verifica che la `x` del giocatore sia minore o uguale al limite calcolato a partire dalla `x` della posizione iniziale del chunk sommata al valore di `border`. Questa porzione pertanto controlla se il giocatore è all'interno della fascia perimetrale in direzione destra rispetto all'origine del chunk.

---

```

player.transform.position.z >=
    c.GetComponent<Terrain>().GetPosition().z

```

---

Si accerta che la coordinata `z` del giocatore sia maggiore o uguale a quella di partenza del chunk; quindi viene verificato se il giocatore è sopra o oltre il bordo inferiore del chunk.

---

```

player.transform.position.z <=
    c.GetComponent<Terrain>().GetPosition().z +
    c.GetComponent<Terrain>().terrainData.size.z

```

---

Analizza che la coordinata `z` del giocatore sia minore o uguale a quella iniziale del chunk sommata alla sua dimensione lungo l'asse `z`; in altre

parole esamina che il giocatore non abbia oltrepassato il bordo superiore del chunk.

È importante notare che tali condizioni non sono implementate come una catena di `if-else`: ciò significa che più logiche possono risultare soddisfatte in contemporanea. In questi casi, ad esempio quando il giocatore si trova in un angolo del chunk, vengono generati due chunk adiacenti.

All'interno dell'Editor di Unity a supporto del debugging osservabile dal codice, sono stati collocati dei parallelepipedi rappresentanti i confini delle varie zone di attivazione. Nella figura seguente è possibile osservare un esempio di generazione alternata dei chunk posti a sinistra e in basso rispetto a quello iniziale poiché Player si trova all'interno dell'area blu in basso a sinistra; nella console vengono stampati i messaggi di quale chunk dovrebbe essere generato.

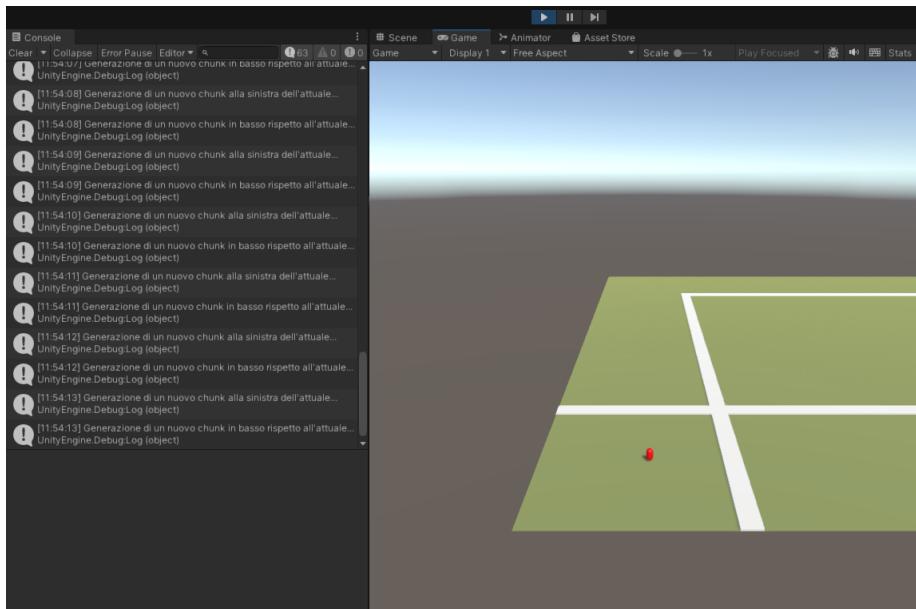


Figura 3.3: Debugging della generazione.

Terminata la fase di verifica si è proceduto con la concreta generazione dei chunk. Come prima operazione è stata introdotta la classe `ChunkDetails`, progettata per monitorare lo stato di generazione dei chunk adiacenti a quello corrente mediante l'impiego di quattro variabili booleane, una per ciascun lato; inoltre sono stati implementati i relativi metodi `getter` e `setter`. Infine lo script è stato associato come componente al prefab `Chunk`.

Listing 3: ChunkDetails.cs

---

```
public class ChunkDetails : MonoBehaviour
{
    private bool leftChunkWasGenerated;
    private bool rightChunkWasGenerated;
    private bool bottomChunkWasGenerated;
    private bool topChunkWasGenerated;

    public bool GetLeftChunkWasGenerated()
    {
        return leftChunkWasGenerated;
    }

    public void SetLeftChunkWasGenerated(bool leftChunkWasGenerated)
    {
        this.leftChunkWasGenerated = leftChunkWasGenerated;
    }

    public bool GetRightChunkWasGenerated()
    {
        return rightChunkWasGenerated;
    }

    public void SetRightChunkWasGenerated(bool
        rightChunkWasGenerated)
    {
        this.rightChunkWasGenerated = rightChunkWasGenerated;
    }

    public bool GetBottomChunkWasGenerated()
    {
        return bottomChunkWasGenerated;
    }

    public void SetBottomChunkWasGenerated(bool
        bottomChunkWasGenerated)
    {
        this.bottomChunkWasGenerated = bottomChunkWasGenerated;
    }

    public bool GetTopChunkWasGenerated()
    {
```

```

        return topChunkWasGenerated;
    }

    public void SetTopChunkWasGenerated(bool topChunkWasGenerated)
    {
        this.topChunkWasGenerated = topChunkWasGenerated;
    }
}

```

---

Tornando al metodo `ChunkHandler.CheckPlayerPosition` è opportuno precisare che nella relazione per le stesse motivazioni precedenti, viene riportata la descrizione dettagliata del funzionamento per un solo caso: quello riguardante la generazione del chunk a sinistra rispetto all'originale; detto ciò è stata aggiunta la seguente porzione di codice:

```

if (!c.GetComponent<ChunkDetails>().GetLeftChunkWasGenerated())
{
    Vector3 spawnPosition = new Vector3(c.transform.position.x -
        c.GetComponent<Terrain>().terrainData.size.x, 0f,
        c.transform.position.z);

    GameObject newChunk = Instantiate(chunkPrefab, spawnPosition,
        Quaternion.identity, chunkParent.transform);

    chunkMap.Add(newChunk);

    newChunk.GetComponent<ChunkDetails>().SetLeftChunkWasGenerated(true);
    c.GetComponent<ChunkDetails>().SetRightChunkWasGenerated(true);

    Debug.Log("Generazione di un nuovo chunk alla sinistra
        dell'attuale...");

}

```

---

Innanzitutto tramite `ChunkDetails.GetLeftChunkWasGenerated` viene verificato se il chunk alla sinistra dell'originale non sia già stato generato; solo in caso negativo si procede con l'istanziazione. La posizione del nuovo chunk viene calcolata sottraendo alla coordinata x del chunk corrente la sua larghezza (`terrainData.size.x`), mantenendo costante la coordinata z e fissando y a 0. L'istruzione successiva consente la creazione di un nuovo chunk mediante `Object.Instantiate` e passando come argomenti `chunkPrefab`, la posizione appena calcolata, la rotazione neutra e il padre `chunkParent`.

Per quanto detto in precedenza il chunk appena istanziato viene aggiunto alla lista `chunkMap` e infine vengono aggiornati i rispettivi flag di adiacenza.

Eseguito il progetto si è rilevato un messaggio di errore sulla console che derivava dall’uso improprio del `foreach` con la lista `chunkMap`; infatti un `foreach` utilizzato su una collezione non consente modifiche a quest’ultima e nel nostro caso effettuavamo proprio la chiamata al metodo `Add()` sulla lista variandone di fatto il contenuto. Al fine di fronteggiare il problema, all’interno del `foreach` si è costruito una copia della lista originale con l’istruzione seguente cosicché le iterazioni venissero effettuate su di essa e gli aggiornamenti sull’originale.

---

```
foreach (GameObject c in new List<GameObject>(chunkMap))
```

---

Rieseguendo il codice è stato ottenuto il seguente risultato:

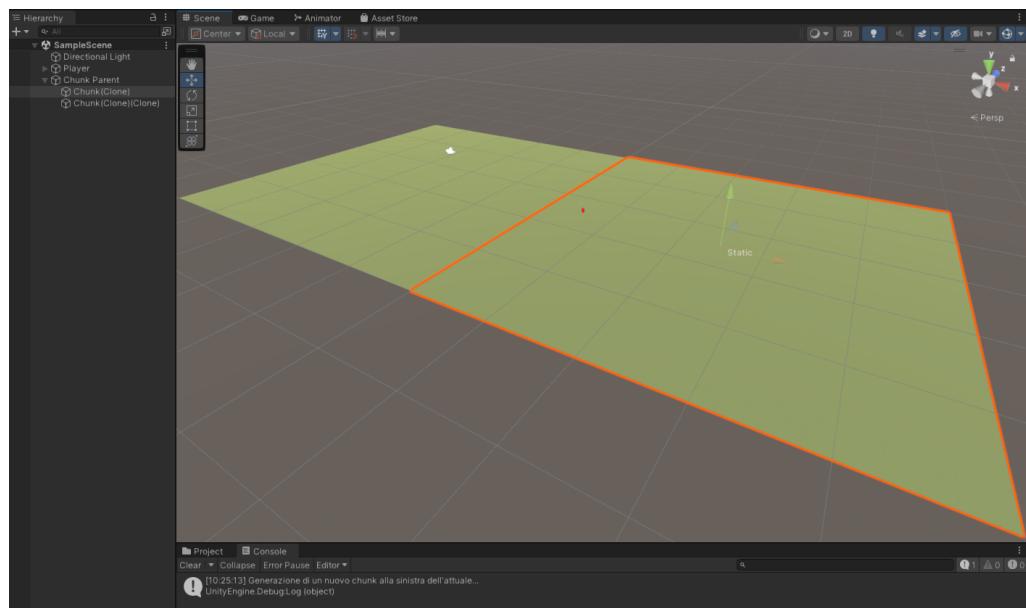


Figura 3.4: Il nuovo chunk.

Come si può notare dalla figura soprastante è stato generato un nuovo chunk; muovendo il giocatore al suo interno si è verificato che la suddivisione in aree funzionasse correttamente prestando attenzione a ulteriori istanziazioni e messaggi della console. In un primo momento il comportamento è risultato coerente con le aspettative, ma continuando l’ispezione è emerso un bug legato a specifiche disposizioni dei chunk: se quello appena generato risultava essere adiacente non solo al chunk che

ne aveva causato la creazione, ma anche ad altri già presenti nella scena, ne venivano erroneamente generati di nuovi sovrapposti a quelli esistenti; tale anomalia era dovuta al fatto che durante la generazione di un nuovo chunk venivano aggiornate solamente le variabili di adiacenza direttamente coinvolte senza verificare se la nuova istanza confinasse anche con altre porzioni di terreno già presenti nella scena. Non si trattava quindi di un errore nel codice quanto più un'insufficienza di controlli adeguati, infatti le istruzioni del seguente frammento di codice aggiornavano correttamente i flag di adiacenza: essendo stato generato un nuovo chunk alla destra di quello originario, veniva attivato il flag destro dell'ultimo aggiunto e simmetricamente il sinistro di quello preesistente, tuttavia questo aggiornamento non era sufficiente a determinare se il nuovo chunk confinasse o meno anche con altri già presenti.

---

```
newChunk.GetComponent<ChunkDetails>().SetRightChunkWasGenerated(true);
c.GetComponent<ChunkDetails>().SetLeftChunkWasGenerated(true);
```

---

Per risolvere questo problema è stato introdotto il metodo `ChunkHandler.FindAdjacentChunks` incaricato di individuare tutti i chunk confinanti con l'ultimo istanziato sfruttando il confronto tramite le loro posizioni e aggiornando i rispettivi flag.  
Ad esempio per verificare la presenza di un chunk alla sinistra di quello originario viene controllato che:

- la coordinata `z` della posizione del nuovo chunk coincida con quella del prefab già esistente:

---

```
c.transform.position.z == newChunk.transform.position.z;
```

---

- la coordinata `x` della posizione del prefab preesistente corrisponda a quella del nuovo chunk a cui viene sottratta la lunghezza del lato del chunk:

---

```
c.transform.position.x == newChunk.transform.position.x -
50;
```

---

Se entrambe le condizioni risultano essere soddisfatte, significa che il chunk preesistente si trova alla sinistra di quello appena generato e vengono quindi aggiornati i corretti flag: il chunk introdotto riconosce di avere un prefab alla sua sinistra mentre l'altro valida la presenza di un nuovo alla sua destra.

A questo punto è stato aggiornato `ChunkHandler.CheckPlayerPosition` inserendo la chiamata al nuovo metodo e rimuovendo le precedenti istruzioni relative ai flag ormai ridondanti.

Listing 4: `ChunkHandler.FindAdjacentChunks`

---

```

public void FindAdjacentChunks(GameObject newChunk)
{
    foreach (GameObject c in chunkMap)
    {
        if (c.transform.position.z == newChunk.transform.position.z
            && c.transform.position.x ==
            newChunk.transform.position.x - 50)
        {
            newChunk.GetComponent<ChunkDetails>().SetLeftChunkWasGenerated(true);
            c.GetComponent<ChunkDetails>().SetRightChunkWasGenerated(true);
        }
        if (c.transform.position.z == newChunk.transform.position.z
            && c.transform.position.x ==
            newChunk.transform.position.x + 50)
        {
            newChunk.GetComponent<ChunkDetails>().SetRightChunkWasGenerated(true);
            c.GetComponent<ChunkDetails>().SetLeftChunkWasGenerated(true);
        }
        if (c.transform.position.x == newChunk.transform.position.x
            && c.transform.position.z ==
            newChunk.transform.position.z - 50)
        {
            newChunk.GetComponent<ChunkDetails>().SetBottomChunkWasGenerated(true);
            c.GetComponent<ChunkDetails>().SetTopChunkWasGenerated(true);
        }
        if (c.transform.position.x == newChunk.transform.position.x
            && c.transform.position.z ==
            newChunk.transform.position.z + 50)
        {
            newChunk.GetComponent<ChunkDetails>().SetTopChunkWasGenerated(true);
            c.GetComponent<ChunkDetails>().SetBottomChunkWasGenerated(true);
        }
    }
}

```

---

Procediamo nell'analizzare se il metodo sia effettivamente in grado di risolvere il bug descritto in precedenza; a titolo di esempio si consideri la

configurazione in cui tre chunk siano disposti a formare una L (evidenziati in arancione nella figura sottostante), con il giocatore inizialmente posizionato nel chunk in basso a destra e successivamente in movimento verso l'alto.

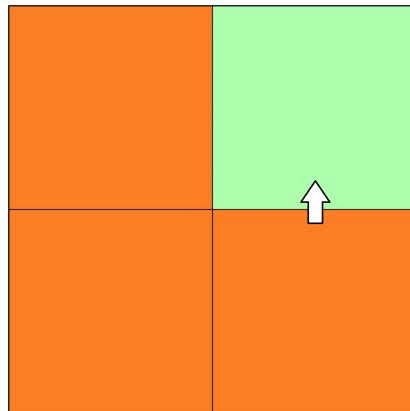


Figura 3.5: Disposizione a L.

Tale spostamento comporta la generazione di un nuovo chunk superiore, evento che a sua volta innesca la chiamata al metodo in questione. Durante la scansione questo individua correttamente due chunk confinanti sui due dei quattro lati di quello appena generato, inferiore e sinistro; conseguentemente aggiorna i relativi flag confermando la correttezza del funzionamento.

*Listing 5: "Confine inferiore"*

---

```
newChunk.GetComponent<ChunkDetails>().SetBottomChunkWasGenerated(true);
c.GetComponent<ChunkDetails>().SetTopChunkWasGenerated(true);
```

---

*Listing 6: "Confine sinistro"*

---

```
newChunk.GetComponent<ChunkDetails>().SetLeftChunkWasGenerated(true);
c.GetComponent<ChunkDetails>().SetRightChunkWasGenerated(true);
```

---

### 3.2.2 Attivazione e disattivazione dei chunk

Nella sottosezione è stato realizzato il sistema di attivazione/disattivazione dei chunk che ha avuto luogo mediante lo sviluppo del metodo `ChunkHandler.UpdateChunkVisibility`.

```
public void UpdateChunkVisibility()
{
    Game0bject actualChunk = null;

    foreach (Game0bject c in chunkMap)
    {
        actualChunk = FindActualChunk(player, c);
        if (actualChunk != null)
        {
            break;
        }
    }

    List<Vector3> activePositions = new() {
        actualChunk.transform.position };

    if
        (actualChunk.GetComponent<ChunkDetails>().GetLeftChunkWasGenerated())
    {
        activePositions.Add(new
            Vector3(actualChunk.transform.position.x - 50, 0f,
            actualChunk.transform.position.z));
    }
    if
        (actualChunk.GetComponent<ChunkDetails>().GetRightChunkWasGenerated())
    {
        activePositions.Add(new
            Vector3(actualChunk.transform.position.x + 50, 0f,
            actualChunk.transform.position.z));
    }
    if
        (actualChunk.GetComponent<ChunkDetails>().GetBottomChunkWasGenerated())
    {
        activePositions.Add(new
            Vector3(actualChunk.transform.position.x, 0f,
            actualChunk.transform.position.z - 50));
    }
    if
        (actualChunk.GetComponent<ChunkDetails>().GetTopChunkWasGenerated())
    {
        activePositions.Add(new
            Vector3(actualChunk.transform.position.x, 0f,
```

```

        actualChunk.transform.position.z + 50));
    }

    foreach (GameObject chunk in chunkMap)
    {
        bool shouldBeActive = false;

        foreach (Vector3 activePos in activePositions)
        {
            if (chunk.transform.position.x == activePos.x &&
                chunk.transform.position.z == activePos.z)
            {
                shouldBeActive = true;
                break;
            }
        }

        if (chunk.activeSelf != shouldBeActive)
        {
            chunk.SetActive(shouldBeActive);
        }
    }
}

```

---

Come prima operazione viene individuato il chunk su cui si trova il giocatore tramite una chiamata al metodo `ChunkHandler.FindActualChunk`, il cui valore di ritorno viene salvato nella variabile `actualChunk` di tipo `GameObject`. Successivamente viene definita una lista di tipo `Vector3`, chiamata `activePositions` che ha il compito di memorizzare le posizioni del chunk corrente e di tutti i suoi adiacenti già generati; la lista viene inizializzata con la posizione di `actualChunk`. Per stabilire quali chunk debbano essere aggiunti viene verificato per ciascun lato, se il chunk confinante con `actualChunk` è stato generato; in caso positivo la posizione del corrispondente chunk adiacente viene aggiunta alla lista `activePositions`.

Il ciclo `foreach` scorre tutti i chunk presenti nella lista `chunkMap`, per ciascuno viene inizialmente impostata la variabile booleana `shouldBeActive` a *false* che indica se il chunk deve rimanere attivo o meno. Successivamente tramite un secondo ciclo `foreach` interno al primo, viene confrontata la posizione del chunk corrente con tutte quelle contenute nella lista

`activePositions` e, se viene trovata una corrispondenza, il chunk deve essere attivato poiché risulta generato e confinante con `actualChunk`: in tal caso la variabile `shouldBeActive` viene impostata a *true* e il ciclo interno viene interrotto con `break`. A questo punto viene verificato se lo stato attuale del chunk (`chunk.activeSelf`) coincide con quello desiderato (`shouldBeActive`) e, se i due valori sono diversi viene aggiornato lo stato del chunk tramite `chunk.SetActive(shouldBeActive)`:

- *true* → il chunk viene reso visibile/attivo;
- *false* → il chunk viene disattivato/nascosto.

Si riporta `ChunkHandler.FindActualChunk`:

---

```
public GameObject FindActualChunk(GameObject player, GameObject
    chunk)
{
    if (player.transform.position.x >=
        chunk.GetComponent<Terrain>().GetPosition().x &&
        player.transform.position.x <
        chunk.GetComponent<Terrain>().GetPosition().x +
        chunk.GetComponent<Terrain>().terrainData.size.x &&
        player.transform.position.z >=
        chunk.GetComponent<Terrain>().GetPosition().z &&
        player.transform.position.z <
        chunk.GetComponent<Terrain>().GetPosition().z +
        chunk.GetComponent<Terrain>().terrainData.size.z)
    {
        return chunk;
    }
    return null;
}
```

---

### 3.3 POPOLAMENTO DEI CHUNK TRAMITE RAYCASTING

Arrivati a questo punto è giunto il momento di popolare i chunk generando su di essi - tramite la tecnica del raycasting - determinati prefab dei pacchetti citati all'inizio del capitolo. Prima di tutto è stato creato il metodo

`ChunkHandler.CalculateSpawnPosition`, il cui compito è quello di determinare una posizione di spawn valida per i modelli tridimensionali. Più

precisamente il metodo calcola casualmente delle coordinate x e z all'interno dei limiti del Terrain associato al chunk, sfruttando la funzione Random.Range.

---

```
float x = UnityEngine.Random.Range(chunk.GetComponent<Terrain>()
    .GetPosition().x,
    chunk.GetComponent<Terrain>().GetPosition().x +
    chunk.GetComponent<Terrain>().terrainData.size.x);
float z = UnityEngine.Random.Range(chunk.GetComponent<Terrain>()
    .GetPosition().z,
    chunk.GetComponent<Terrain>().GetPosition().z +
    chunk.GetComponent<Terrain>().terrainData.size.z);
```

---

Immediatamente dopo viene dichiarata la variabile height e inizializzata a un valore superiore rispetto all'altezza massima del terreno.

---

```
float height = chunk.GetComponent<Terrain>().terrainData.size.y +
    50;
```

---

In associazione con l'istruzione:

---

```
Vector3 rayOrigin = new(x, height, z);
```

---

consente di collocare il punto di origine del raycast in una posizione sopraelevata, cosicché il raggio proiettato verso il basso con l'istruzione seguente possa intercettare la superficie del Terrain.

Il successivo blocco di codice scritto effettua un raycast verso il basso e, nel momento in cui viene rilevata la collisione restituisce la coordinata del punto di impatto (hit.point), impiegata poi come posizione di spawn sul terreno.

---

```
Physics.Raycast(rayOrigin, Vector3.down, out RaycastHit hit,
    height);
return hit.point;
```

---

Si riporta per intero il metodo descritto.

---

```
Vector3 CalculateSpawnPosition(GameObject chunk)
{
    float x = UnityEngine.Random.Range(chunk.GetComponent<Terrain>()
        .GetPosition().x,
        chunk.GetComponent<Terrain>().GetPosition().x +
        chunk.GetComponent<Terrain>().terrainData.size.x);
    float z = UnityEngine.Random.Range(chunk.GetComponent<Terrain>()
```

```

    .GetPosition().z,
    chunk.GetComponent<Terrain>().GetPosition().z +
        chunk.GetComponent<Terrain>().terrainData.size.z);
    float height = chunk.GetComponent<Terrain>().terrainData.size.y
        + 50;

    Vector3 rayOrigin = new(x, height, z);
    Physics.Raycast(rayOrigin, Vector3.down, out RaycastHit hit,
        height);
    return hit.point;
}

```

---

In seguito è stata introdotta la classe `GameObjectPrefab` inizialmente definita:

```

public class GameObjectPrefab : MonoBehaviour
{
    public GameObject prefab;
}

```

---

Questa funge da contenitore di configurazioni per la generazione di prefab, al suo interno è presente un riferimento a questo da assegnare tramite l'editor grafico di Unity; la classe verrà arricchita più avanti.

Dopodiché in `ChunkHandler.cs` è stato sviluppato il metodo `SpawnEnvironmentObjectsOnChunk()` incaricato di popolare ciascun chunk con oggetti ambientali; viene richiamato ogni volta che una nuova porzione di terreno viene istanziata. Al suo interno viene innanzitutto fissato il numero di istanze da generare per ciascun elemento della collezione `objectsToPool`, successivamente un ciclo for calcola per ogni iterazione la posizione di spawn mediante il metodo `CalculateSpawnPosition()`, istanzia il prefab corrispondente e gli assegna la posizione calcolata inizializzando la rotazione al valore neutro `Quaternion.identity`.

```

void SpawnEnvironmentObjectsOnChunk(GameObject chunk)
{
    int amountToSpawn = 30;
    foreach (GameObjectPrefab element in objectsToPool)
    {
        for (int i = 0; i < amountToSpawn; i++)
        {
            Vector3 spawnPos = CalculateSpawnPosition(chunk);
            GameObject obj = Instantiate(element.prefab, spawnPos,
                Quaternion.identity);
        }
    }
}

```

```

    }
}
}

```

---

In Unity è stato poi creato un oggetto vuoto denominato Tree, al quale è stato associato lo script GameObjectPrefab; così da assegnare via Inspector il prefab corrispondente precedentemente spostato nella cartella Prefabs da quelle importate. Dopodiché si è intervenuti sull'oggetto di gioco Player che contiene il componente ChunkHandler.cs, più precisamente nel campo objectsToPool è stato trascinato il GameObject Tree in modo da popolare la lista con i prefab degli alberi.

Eseguendo il progetto tuttavia è emerso che gli oggetti ambientali venivano posizionati correttamente ma restavano visibili anche dopo la disattivazione del chunk: il problema derivava dal fatto che non erano stati impostati come figli del GameObject rappresentante il chunk, pertanto non ne ereditavano lo stato attivo/disattivo. Per risolvere il problema è stato sufficiente specificare come quarto parametro del metodo Object.Instantiate l'oggetto di gioco padre del modello tridimensionale, ossia il chunk che lo contiene passato come parametro al metodo.

---

```
GameObject obj = Instantiate(element.prefab, spawnPos,
    Quaternion.identity, chunk.transform);
```

---

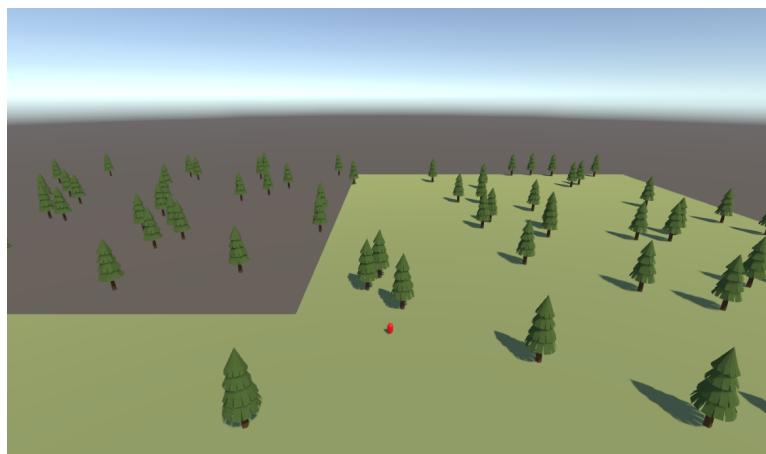


Figura 3.6: Modelli 3D rimasti attivi nonostante la disattivazione del chunk.

Visto che l'intenzione era quella di aggiungere nuovi oggetti tridimensionali allo scopo di ottenere maggiore biodiversità, una volta appurato che il progetto fosse privo di errori, è stato modificato il metodo

SpawnEnvironmentObjectsOnChunk() inserendo una porzione di codice incaricata nella determinazione della tipologia tipo del prefab e in conseguenza di associare un certo valore casuale alla variabile intera amountToSpawn all'interno di un determinato range che varia sulla base della categoria del prefab.

---

```
int amountToSpawn;
foreach (GameObjectPrefab element in objectsToPool)
{
    if (element.prefab.name.Contains("Flower"))
        amountToSpawn = UnityEngine.Random.Range(50, 101);
    else if (element.prefab.name.Contains("Rock"))
    {
        amountToSpawn = UnityEngine.Random.Range(10, 16);
    }
    else if (element.prefab.name.Contains("Grass"))
    {
        amountToSpawn = UnityEngine.Random.Range(100, 201);
    }
    else
    {
        amountToSpawn = UnityEngine.Random.Range(75, 131);
    }
    //...
}
```

---

Come si può notare dal codice il tipo di prefab viene riconosciuto attraverso il nome assegnato in Unity all'oggetto di gioco all'interno della Hierarchy. A tal fine in una fase preliminare sono stati creati nel motore grafico diversi oggetti vuoti, ciascuno con una parola chiave identificativa; su ognuno di essi sono state poi replicate le stesse operazioni illustrate in precedenza per l'oggetto Tree. In questa nuova versione il numero di modelli tridimensionali spawnati dipende proprio dal loro "tipo" in quanto per ognuno di essi è stato assegnato un valore diverso alla variabile amountToSpawn.

In conclusione al fine di ottimizzare l'ordine e la leggibilità della gerarchia in Unity, è stato creato un ulteriore oggetto di gioco vuoto con il compito di fungere da genitore di tutti i nuovi oggetti di gioco.



Figura 3.7: I nomi degli oggetti di gioco.

Siamo poi tornati nuovamente sulla classe `GameObjectPrefab` che è stata aggiornata con l'aggiunta di due intestazioni (Header), utili a organizzare in modo più chiaro l'interfaccia dell'Inspector di Unity; attraverso i campi associati a tali intestazioni è possibile specificare i valori minimi e massimi della scala e della rotazione del prefab. Successivamente sono stati generati anche i getter di tali variabili.

---

```
public class GameObjectPrefab : MonoBehaviour
{
    public GameObject prefab;

    [Header("Random Scale Range")]
    [SerializeField] private Vector3 minScale = Vector3.one;
    [SerializeField] private Vector3 maxScale = Vector3.one;

    [Header("Random Rotation Range")]
    [SerializeField] private Vector3 minRotation = Vector3.zero;
    [SerializeField] private Vector3 maxRotation = Vector3.zero;

    public Vector3 GetMinScale()
    {
        return minScale;
    }

    public Vector3 GetMaxScale()
    {
        return maxScale;
    }

    public Vector3 GetMinRotation()
    {
        return minRotation;
    }

    public Vector3 GetMaxRotation()
```

```
{  
    return maxRotation;  
}  
}
```

---

Di seguito viene quindi riportata l'aggiornata versione di `ChunkHandler.SpawnEnvironmentObjectsOnChunk`.

```
void SpawnEnvironmentObjectsOnChunk(GameObject chunk)  
{  
    int amountToSpawn;  
    Vector3 spawnPos;  
    foreach (GameObjectPrefab element in objectsToPool)  
    {  
        if (element.prefab.name.Contains("Flower")) {  
            amountToSpawn = UnityEngine.Random.Range(50, 101);  
        }  
        else if (element.prefab.name.Contains("Rock"))  
        {  
            amountToSpawn = UnityEngine.Random.Range(10, 16);  
        }  
        else if (element.prefab.name.Contains("Grass"))  
        {  
            amountToSpawn = UnityEngine.Random.Range(100, 201);  
        }  
        else  
        {  
            amountToSpawn = UnityEngine.Random.Range(75, 131);  
        }  
  
        for (int i = 0; i < amountToSpawn; i++)  
        {  
            spawnPos = CalculateSpawnPosition(chunk);  
  
            Vector3 randomEuler = new Vector3(  
                UnityEngine.Random.Range(element.GetMinRotation().x,  
                    element.GetMaxRotation().x),  
                UnityEngine.Random.Range(element.GetMinRotation().y,  
                    element.GetMaxRotation().y),  
                UnityEngine.Random.Range(element.GetMinRotation().z,  
                    element.GetMaxRotation().z)  
            );  
            GameObject obj = Instantiate(element.prefab, spawnPos,
```

```

        Quaternion.identity, chunk.transform);
Vector3 randomScale = new Vector3(
    UnityEngine.Random.Range(element.GetMinScale().x,
        element.GetMaxScale().x),
    UnityEngine.Random.Range(element.GetMinScale().y,
        element.GetMaxScale().y),
    UnityEngine.Random.Range(element.GetMinScale().z,
        element.GetMaxScale().z)
);
obj.transform.localScale = randomScale;
obj.SetActive(true);
}
}
}

```

Nella parte finale del metodo è stata aggiunta una breve porzione di codice che consente di specificare valori minimi e massimi della rotazione, inclinazione e scala dei prefab utilizzando la classe Game0bjectPrefab.

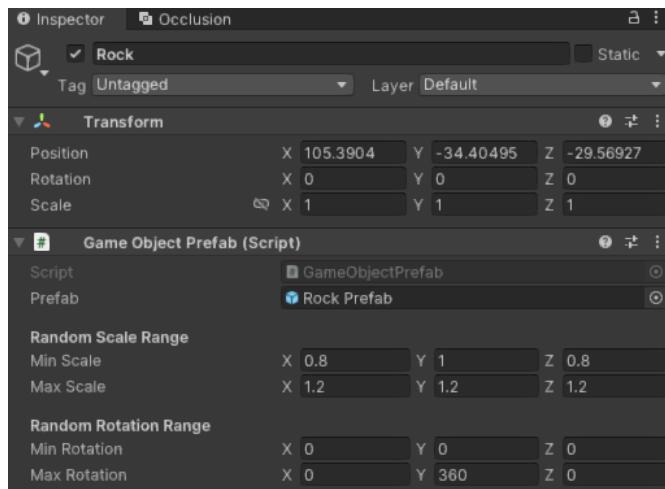


Figura 3.8: Prendendo come esempio il prefab Rock è possibile notare come si possano definire i limiti minimi e massimi con cui scegliere dimensione, rotazione e inclinazione.

Eseguendo ulteriormente il progetto è stato riscontrato un nuovo bug che riguardava il posizionamento dei prefab generati sul terreno. Originariamente il raycast utilizzato per calcolare la posizione di spawn non verificava se la collisione avvenisse effettivamente con il collider del Terrain. Di conseguenza in rare occasioni, i prefab potevano sovrapporsi

tra loro perché il raycast rilevava la collisione con altri prefab anziché col terreno del chunk come visibile nell'immagine sottostante.



Figura 3.9: Prefab di un sasso collocato sopra un albero, a sinistra del giocatore.

La soluzione ha consistito nell'effettuare un controllo all'interno di `CalculateSpawnPosition()`:

---

```
if (hit.collider is TerrainCollider)
{
    return hit.point;
}
return new Vector3(0f, 0f, -9999f);
```

---

In questo modo il metodo restituisce solo posizioni valide sul terreno, ignorando eventuali collisioni con altri oggetti; nella funzione `SpawnEnvironmentObjectsOnChunk()` viene verificato che il valore restituito sia valido (`spawnPos.y > -9998f`) prima di istanziare il prefab, se non dovesse essere tale allora non viene istanziato e ne vengono calcolati di nuovi fino a quando questo non risulta corretto. La versione definitiva di `SpawnEnvironmentObjectsOnChunk()`:

---

```
void SpawnEnvironmentObjectsOnChunk(GameObject chunk)
{
    int amountToSpawn;
    Vector3 spawnPos;
    foreach (GameObjectPrefab element in objectsToPool)
    {
        if (element.prefab.name.Contains("Flower"))
        {
            amountToSpawn = UnityEngine.Random.Range(50, 101);
        }
        else if (element.prefab.name.Contains("Rock"))
        {
            amountToSpawn = UnityEngine.Random.Range(10, 16);
        }
        else if (element.prefab.name.Contains("Grass"))
        {
```

---

```
        amountToSpawn = UnityEngine.Random.Range(100, 201);
    }
    else
    {
        amountToSpawn = UnityEngine.Random.Range(75, 131);
    }
    for (int i = 0; i < amountToSpawn; i++)
    {
        spawnPos = CalculateSpawnPosition(chunk);
        if (spawnPos.y > -9998f)
        {
            Vector3 randomEuler = new Vector3(
                UnityEngine.Random.Range(element.GetMinRotation().x,
                    element.GetMaxRotation().x),
                UnityEngine.Random.Range(element.GetMinRotation().y,
                    element.GetMaxRotation().y),
                UnityEngine.Random.Range(element.GetMinRotation().z,
                    element.GetMaxRotation().z)
            );
            GameObject obj = Instantiate(element.prefab, spawnPos,
                Quaternion.identity, chunk.transform);
            Vector3 randomScale = new Vector3(
                UnityEngine.Random.Range(element.GetMinScale().x,
                    element.GetMaxScale().x),
                UnityEngine.Random.Range(element.GetMinScale().y,
                    element.GetMaxScale().y),
                UnityEngine.Random.Range(element.GetMinScale().z,
                    element.GetMaxScale().z)
            );
            obj.transform.localScale = randomScale;
            obj.SetActive(true);
        }
    else
    {
        while (spawnPos.y == -9998f)
        {
            spawnPos = CalculateSpawnPosition(chunk);
        }
    }
}
}
```

---

Per ammorbidente l'atmosfera è stata introdotta della nebbia tramite le impostazioni di Unity (Window → Rendering → Lighting) scegliendo il colore grigio con una leggera sfumatura gialla che contribuisce a ricreare un ambiente più caldo.



# 4

---

## MOTIVAZIONI TECNICHE

---

In questo capitolo vengono illustrate le motivazioni sottostanti l'utilizzo di precise soluzioni portando anche degli esempi.

### 4.1 SCELTA DEI MODELLI TRIDIMENSIONALI

In questa sezione vogliamo chiarire perché abbiamo utilizzato determinati modelli tridimensionali piuttosto che altri.

#### *Rendering*

È il processo di calcolo svolto dal processore grafico che genera immagini in tempo reale trasformando modelli 3D, texture e luci in pixel visibili sullo schermo.

Nel caso in cui nella scena di Unity fossero presenti pochi oggetti con ridotto numero di dettagli, la renderizzazione richiederebbe tempi più brevi impattando positivamente sul frame rate, il quale sarà elevato; ma se il numero di oggetti dovesse essere alto e soprattutto dovessero avere dettagli elevati, la GPU si sforzerebbe maggiormente e potrebbero apparire dei rallentamenti (lag) tra un frame e il successivo, attesa dovuta ai calcoli necessari al fine di mostrarli con la corretta illuminazione, colore, posizione e altri fattori, sotto forma di immagine.

Pertanto è possibile dedurre che è opportuno cercare di limitare il numero di poligoni all'interno dei modelli tridimensionali impiegati nei videogiochi in quanto il rendering avviene in tempo reale, ovvero durante l'esecuzione dello stesso; cosa diversa invece avviene per i film di animazione - come per esempio quelli prodotti da Pixar o DreamWorks (Toy Story, Shrek, Madagascar, Cars e molti altri) - nei quali il rendering di ogni frame avviene isolatamente e poi montato assieme agli altri al fine di ottenere un filmato. In questo caso si parla di grafica pre-renderizzata, la quale non soffre di limitazioni relative alla qualità grafica perché

prima renderizzata e successivamente impressa su un video (infatti la renderizzazione avviene una sola volta per ogni frame). [1]

### *Low Poly*

I modelli tridimensionali impiegati nel progetto possiedono un ridotto numero di poligoni risultando in un aspetto stilizzato, squadrato e poco dettagliato; questi vengono identificati con l'appellativo "Low Poly" e sono il contrario degli "High Poly". Lo stile è nato per motivi di performance nei videogiochi a causa dei limiti suddetti, ma oggi è anche una scelta artistica intenzionale; tuttavia l'impiego dello stile è stato motivato da esigenze prestazionali.



Figura 4.1: Low Poly e High Poly.

## 4.2 COROUTINE

La coroutine `CheckPlayerPosition()` ricordiamo essere stata utilizzata nella classe `ChunkHandler` al posto di `Update()`; il suo scopo è stato quello di ottenere benefici prestazionali in quanto il codice al suo interno veniva eseguito un numero di volte al secondo pari al valore della variabile `checkInterval`. Se lo stesso codice fosse stato inserito in `Update()`, sarebbe stato eseguito a ogni frame (circa 60 volte al secondo), anziché una volta al secondo come avviene nel progetto, che come possiamo dedurre riduce sensibilmente il consumo di risorse.

*Perché definire coroutine proprio CheckPlayerPosition()?*

Il motivo è piuttosto semplice, al suo interno vengono svolte operazioni che devono essere costantemente ripetute e mai sospese, caratteristiche che l'avrebbero reso adatto per essere collocato nell'Update(); inoltre è il metodo maggiormente oneroso in termini di prestazioni in quanto è anche quello che ne richiama di più.

#### 4.3 SISTEMA DI GENERAZIONE, ATTIVAZIONE E DISATTIVAZIONE

Esteticamente il continuo generare, attivare e disattivare i chunk può risultare poco gradevole in quanto è totalmente visibile, ma ciò è stata una scelta volontaria proprio per visualizzare la logica del progetto.

Per quanto riguarda la generazione è opportuno considerare che questa ha seguito un modello geometrico piuttosto intuitivo; ricordiamo che le nuove porzioni di territorio vengono create unicamente ai lati di quelle preesistenti quando il giocatore raggiunge il rispettivo bordo del chunk corrente, senza prendere in considerazione i vertici. Questa scelta sebbene non sia l'unica possibile, è motivata da ragioni geometriche intuitive che permettono all'utente di dedurne facilmente il suo andamento.

Il sistema adibito per i chunk garantisce una gestione efficiente delle risorse; in particolare quelli disattivati, assieme ai loro componenti, non vengono più renderizzati e lo stesso vale per tutti i loro oggetti figli. Tuttavia è importante precisare che, nonostante l'ottimizzazione in termini di rendering e calcolo, i dati relativi ai chunk disattivati rimangono comunque caricati in memoria RAM; conseguentemente pur migliorando le prestazioni in tempo reale, la scena tende a incrementare progressivamente il consumo di memoria man mano che ne vengono generati di nuovi.

Occorre pure osservare che il sistema è stato progettato in modo da mantenere al massimo cinque chunk attivi contemporaneamente; inoltre per ciascuno di essi è stato definito un numero massimo di oggetti ambientali generabili attraverso il limite superiore della variabile amountToSpawn. Questo garantisce che dal punto di vista computazionale, escludendo la memoria, il carico di lavoro rimanga entro un limite superiore ben definito.

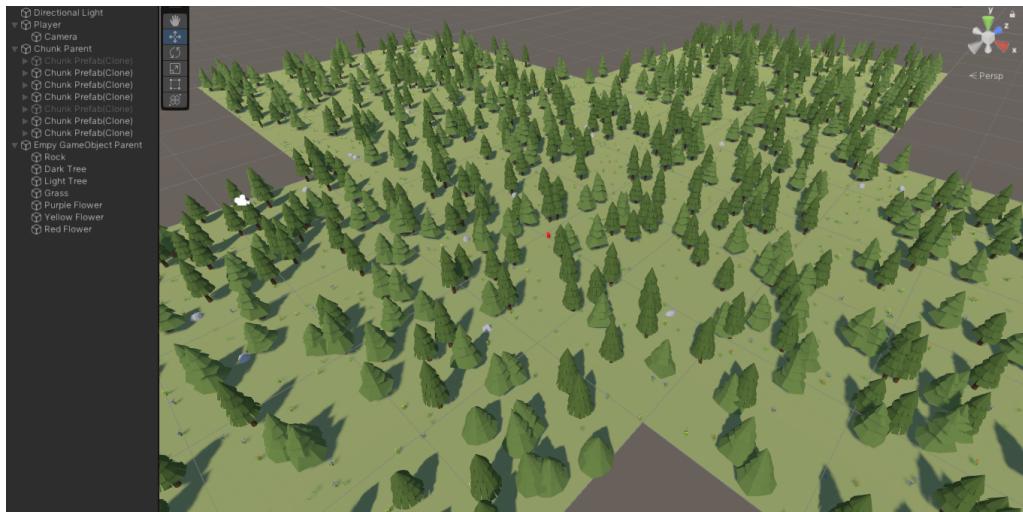


Figura 4.2: Visualizzazione dei cinque chunk attivi contemporaneamente, corrispondente al massimo carico di rendering previsto dal sistema.

Quando il progetto viene avviato il numero di fotogrammi al secondo si attesta intorno ai 350; muovendosi nella scena e generando nuovi chunk il valore tende a diminuire stabilizzandosi intorno ai 200. È interessante notare che indipendentemente dal numero di chunk generati - salvo casi estremi - il valore minimo rimane pressoché costante quando il numero di quelli attivi è massimo dimostrando l'efficacia del meccanismo di attivazione e disattivazione; solo nel caso in cui la RAM disponibile si esaurisca completamente, con conseguente utilizzo della memoria virtuale, si potrebbero osservare dei cali di questo.

#### 4.4 PER QUALE MOTIVO LIST E NON VECTOR

Infine si ricorda che all'interno di `ChunkHandler` è presente la variabile `chunkMap` di tipo `List<GameObject>`; occorre osservare che è stata scelta una lista al posto di un vettore perché necessitavamo di poter aggiungere e rimuovere elementi in fase di esecuzione, nonostante la minor efficienza di quest'ultima. [1].

# 5

---

## CONCLUSIONI

---

Nel complesso il progetto ha consentito di approfondire in modo concreto il tema della generazione procedurale osservando l'applicazione di concetti teorici all'interno del motore grafico.

### 5.1 POSSIBILI SVILUPPI FUTURI

Grazie alla natura intrinseca del progetto e alla presenza di un metodo per ogni operazione, questo risulta facilmente aggiornabile ed estendibile. Possiamo prevedere l'introduzione di un sistema che consenta all'utente di selezionare un numero minimo e massimo di chunk da generare al raggiungimento delle aree di attivazione. Un valore più elevato migliorebbe la resa visiva e la continuità dell'ambiente, ma senza delle tecniche che provvedano a gestire efficacemente le risorse, impatterebbe negativamente sulle prestazioni. La tecnica del pooling di oggetti potrebbe fare al caso nostro; generalmente è impiegata per creare un insieme finito di oggetti prima dell'avvio del gioco, i quali vengono attivati e disattivati nel momento del bisogno senza dover ricorrere a una continua creazione e distruzione, migliorando notevolmente l'efficienza, in quanto queste ultime sono operazioni più onerose. [1] Ipoteticamente nel progetto potremmo definire un gruppo finito di modelli tridimensionali destinati alla popolazione dei chunk; quando viene disabilitato uno di quest'ultimi, i modelli contenuti in esso vengono a loro volta disattivati e, se richiesto, riattivati in altre porzioni di territorio.

Tale processo migliorerebbe anche il problema trattato nel capitolo precedente relativo alla saturazione della memoria volatile, poiché avremmo anche un limite superiore agli oggetti totali da salvare, tuttavia non lo mitigherebbe del tutto visto che il numero di chunk generabili è potenzialmente infinito. Per risolverlo completamente potremmo inserire un limite massimo molto elevato al numero totale di porzioni territoriali generabili

oppure salvare i dati di creazione sul disco rigido e caricarli in RAM quando necessario o addirittura combinare assieme le due meccaniche.

Dal motore grafico l'utente può specificare il numero minimo e massimo di elementi per ogni categoria (alberi, rocce, cespugli, ecc.); un perfezionamento potrebbe essere quello di introdurre un menu per svolgere tale operazione e un meccanismo in grado di impedire un eccesso di oggetti tridimensionali così da limitare anche i corrispondenti collider, che potrebbero compromettere la fluidità del gameplay.

Infine un'altra possibilità di estensione potrebbe essere quella di includere l'aggiunta di nuovi modelli tridimensionali, la simulazione di condizioni atmosferiche variabili, la presenza di animali e la creazione di differenti ecosistemi.

---

## BIBLIOGRAFIA

---

- [1] William Giacinti. *UNYC# Volume 1*. L'Autore, 2020.
- [2] Unity Technologies. CharacterController.isGrounded scripting reference, 2025. Disponibile su: <https://docs.unity3d.com/ScriptReference/CharacterController-isGrounded.html>. Ultimo accesso: 10 novembre 2025.
- [3] Unity Technologies. Debug, 2025. Disponibile su: <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Debug.html>. Ultimo accesso: 10 novembre 2025.
- [4] Unity Technologies. Debug.Log scripting reference, 2025. Disponibile su: <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Debug.Log.html>. Ultimo accesso: 10 novembre 2025.
- [5] Unity Technologies. GameObject.activeSelf scripting reference, 2025. Disponibile su: <https://docs.unity3d.com/ScriptReference/GameObject-activeSelf.html>. Ultimo accesso: 10 novembre 2025.
- [6] Unity Technologies. GameObject.GetComponent scripting reference, 2025. Disponibile su: <https://docs.unity3d.com/ScriptReference/GameObject.GetComponent.html>. Ultimo accesso: 10 novembre 2025.
- [7] Unity Technologies. GameObject.SetActive scripting reference, 2025. Disponibile su: <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/GameObject.SetActive.html>. Ultimo accesso: 10 novembre 2025.
- [8] Unity Technologies. Input.GetAxis scripting reference, 2025. Disponibile su: <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Input.GetAxis.html>. Ultimo accesso: 10 novembre 2025.
- [9] Unity Technologies. Monobehaviour, 2025. Disponibile su: <https://docs.unity3d.com/6000.2/Documentation/>

[ScriptReference/MonoBehaviour.html](#). Ultimo accesso: 10 novembre 2025.

- [10] Unity Technologies. Object.Instantiate scripting reference, 2025. Disponibile su: <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Object.Instantiate.html>. Ultimo accesso: 10 novembre 2025.
- [11] Unity Technologies. Physics.gravity scripting reference, 2025. Disponibile su: <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Physics-gravity.html>. Ultimo accesso: 10 novembre 2025.
- [12] Unity Technologies. Random.Range scripting reference, 2025. Disponibile su: <https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Random.Range.html>. Ultimo accesso: 10 novembre 2025.
- [13] Wikipedia contributors. Generazione procedurale, 2025. Disponibile su: [https://en.wikipedia.org/wiki/Procedural\\_generation](https://en.wikipedia.org/wiki/Procedural_generation). Ultimo accesso: 22 settembre 2025.
- [14] Wikipedia contributors. Maze craze: A game of cops 'n robbers, 2025. Disponibile su: [https://en.wikipedia.org/wiki/Maze\\_Craze](https://en.wikipedia.org/wiki/Maze_Craze). Ultimo accesso: 22 settembre 2025.

---

## RINGRAZIAMENTI

---

*Ringrazio innanzitutto il Professor Francesco Tiezzi per la guida e per aver accolto l'idea di questa tesi.*

*Un ringraziamento speciale va ai miei genitori per avermi sempre sostenuto negli studi e permesso di intraprendere questo percorso.*

*Un grazie a mia zia, è anche merito suo se oggi sono arrivato a scrivere questa tesi.*

*Un grazie a tutto il resto della mia famiglia per il sostegno donatomi.*

*Un grazie agli amici più stretti, che mi hanno insegnato a non arrendersi anche quando la determinazione sembrava venire meno.*

*Infine, ringrazio Clara per aver creduto in me e per essermi stata vicino sia nei momenti di felicità che di difficoltà.*