

SUPSI

Utilizzo di Octree per la generazione di mondi procedurali in Unity

Studente/i

Diego Vagni

Diego Vagni

Relatore

Masiar Babazadeh di Lugano

Correlatore

-

Committente

-

Corso di laurea

Ingegneria informatica

Modulo

C10325

Anno

2021

Data

10 giugno 2021

STUDENTSUPSI

Indice

1 Introduzione	3
1.1 Mondi nei videogiochi	4
1.2 Motivazione	6
1.3 Domanda di ricerca	7
2 Ricerca	9
2.1 Renderizzazione	9
2.2 Chunk e Region	10
2.3 Perlin Noise e octaves	12
2.4 Multi noise biome source	13
3 Implementazione	15
3.1 Architettura	15
3.1.1 Cos'è l'Octree	18
3.1.2 Octree vs Array	20
3.2 Sviluppo	21
3.2.1 Util	22
3.2.2 Octree	22
3.2.3 Cubedata & cubeprototype	23
3.2.4 Chunk, Region & world	26
3.2.4.1 Chunk	26
3.2.4.2 Region	26
3.2.4.3 World	28
3.2.5 Terrain generation & Biomi	31
3.2.6 RegionManager & saving	32
3.2.7 Render	34
4 Benchmark	37
4.1 Octree vs Octree	37

5 Conclusioni	39
5.1 Future work	40
5.1.1 Più dettagli	40
5.1.2 Performance	40
5.1.3 MeshBuilder	41
5.1.4 Noise	41

Elenco delle figure

1.1	Esempio di staccionata su Minecraft, posizionabile solo al centro del blocco, occupandolo completamente.	5
1.2	Scavando un buco nel terreno si può vedere la fondamentale struttura cubica del gioco 7 Days to Die.	6
1.3	Ogni parte della ringhiera occupa un blocco intero sebbene a livello fisico ne occupi solo una parte.	7
2.1	Il Texture atlas utilizzato all'interno del progetto.	10
2.2	Suddivisione di Chunk e Region nel progetto. In nero è rappresentato il bordo del mondo, in verde il bordo della Region e in grigio il bordo dei singoli Chunk	11
2.3	Rappresentazione visiva del Multi-noise biome source	14
3.1	Diagramma delle classi dell'intero progetto.	16
3.2	Questa è la pipeline del sistema	17
3.3	Rappresentazione dell'Octree.	18
3.4	Rappresentazione di un Quadtree.	19
3.5	Diagramma classi Octree.	22
3.6	Diagramma delle classi di CubeData e CubePrototype	24
3.7	Cubo rappresentante l'erba. Ogni faccia ha una texture diversa.	26
3.8	Diagramma delle Region e dei Chunk	27
3.9	Diagramma delle classi del mondo.	29
3.10	Il player è il centro del visible-radius, mentre i Chunk non ancora renderizzati ma in preparazione fra essi sono nel loaded radius	30
3.11	Diagramma delle classi della generazione del terreno e i biomi	31
3.12	Diagramma delle classi della generazione del terreno e i biomi.	33
3.13	Il texture atlas utilizzato all'interno del progetto, unico file di texture.	34
4.1	Grafico dei risultati su 1000 inserimenti e 1000 rimozioni.	38

Abstract

Nel mercato videoludico moderno i mondi di gioco sono sempre più ampi e dettagliati. In particolare la generazione procedurale di essi dà la possibilità al giocatore di esplorare mondi così grandi da poter essere considerati quasi infiniti, con come uniche limitazioni la memoria utilizzata e la precisione delle variabili utilizzate al suo interno. Questo progetto di diploma è atto a esplorare l'utilizzo di Octree, una struttura ad albero dove ogni nodo ha esattamente 8 figli, per ottimizzare la generazione procedurale di mondi nel motore di sviluppo in real time Unity. Dopo un primo studio dell'algoritmo e delle implementazioni già disponibili in Unity, si è inizialmente lavorato allo sviluppo della struttura dati dell'Octree per definire un mondo procedurale simile a quello di *Minecraft*, portando attenzione sia alla modularità dell'implementazione che alla separazione rispetto alle componenti fondamentali di Unity. In seguito si è sviluppato il motore per la generazione procedurale di mondi partendo da un seed, usando diverse tecniche di generazione, come per esempio il Perlin Noise, una funzione di generazione di numeri pseudo-random utilizzata per la generazione di un heightmap e la generazione di diversi biomi. Inoltre sono state inserite diverse ottimizzazioni tra cui l'utilizzo del multithreading per parallelizzare i lavori più pesanti computazionalmente e la generazione di mesh combinate con l'utilizzo di una Texture Atlas. Il risultato è un motore in grado di generare dei mondi con biomi diversi (Tundra, Deserto, Oceano, etc.) potenzialmente infiniti partendo da un seed con un forte decoupling dalle strutture fondamentali di Unity per poterlo scorporare ed utilizzare stand-alone. Il lavoro presenta anche un benchmark contro un'implementazione attualmente disponibile di Octree su Unity.

In the modern videogame market, the game worlds are increasingly large and detailed. In particular, the procedural generation gives the player the possibility to explore worlds so large that they can be considered almost infinite, with the only limitations being the memory used and the precision of the variables used within it. This diploma project is designed to explore the use of Octree, a tree structure where each node has precisely eight children, to optimize the procedural generation of worlds using the Unity engine as graphic engine. After an initial algorithm study, I initially worked on developing the Octree data structure to define a procedural world like *Minecraft*, paying attention to both the modularity of the implementation and the separation from the core components of Unity. Later the engine for the procedural generation of worlds has been developed using different generation techniques,

such as Perlin Noise, a pseudo-random number generation function used to generate a heightmap and the generation of different biomes. There are also some optimizations such as the use of multithreading to parallelize the heaviest computationally jobs and the generation of meshes combined with a Texture Atlas. The result is an engine capable of generating potentially infinite worlds with different biomes (Tundra, Desert, Ocean, etc.), starting from a seed with a strong decoupling from the fundamental structures of Unity to render possible to use it stand-alone. The work also presents a benchmark against the one of the available implementations of Octree in Unity.

Capitolo 1

Introduzione

Oggi i videogiochi sono sempre più grandi e dettagliati sotto ogni loro aspetto. Non solo c'è una grande cura nella narrativa e nello studio di nuove meccaniche, ma anche i mondi di gioco si fanno sempre più grandi e dettagliati. Negli ultimi anni abbiamo potuto vedere come i mondi di gioco sono diventati così ampi da permettere un gameplay anche a "mondo aperto", un genere che permette al giocatore di esplorare il mondo interagendo con esso senza esser necessariamente legato a una struttura videoludica a livelli. Questo non è stato sempre così, infatti i primi videogiochi nati, quasi in contemporanea ai primi computer, erano strutturati in modo molto diverso. All'inizio del decennio 1950 gli ingegneri informatici hanno iniziato a implementare semplici giochi all'interno di enormi mainframe a fini dimostrativi: computer come *Bertie The Brain*[5] erano utilizzati per mostrare al pubblico l'intelligenza dei computer, in questo caso offrendo la possibilità di giocare a Tic-Tac-Toe. Questi giochi, nati a fine dimostrativo, rimanevano comunque chiusi nelle università, salvo i periodi di porte aperte dove venivano mostrati al pubblico. Nel 1960 degli studenti del Massachusetts Institute of Technology (MIT) hanno creato *SpaceWar!*. Il gioco permetteva di simulare una piccola battaglia fra navicelle, sfruttando il piccolo monitor del Mainframe PDP-1 sul quale girava[15]. In *SpaceWar!* due navicelle si scontrano sparandosi evitando la gravità di una stella piazzata al centro dello schermo. In seguito alla nascita degli arcade nel 1971, il periodo che va dal 1975 al 1981 li vede spopolare, con giochi come *Space Invaders* (Taito, 1978), *Pac man* (Namco, 1980) e *Donkey Kong* (Nintendo, 1981). In questi anni all'interno dei videogiochi sono nate importanti meccaniche come il numero di vite, il punteggio e le leaderboard[1], importanti ai fini di gameplay e che sono andate a impattare la percezione e la fruibilità del videogioco, offrendo una struttura a livelli più definita. Nel 1976 nasce *Colossal Cave Adventure*, un'avventura testuale sviluppata da William Crowther e Don Woods, considerata da Ars Technica come, concettualmente, il primo gioco con mondo aperto della storia. Era nato un nuovo modo di giocare, nel quale il giocatore aveva la possibilità di immergersi in mondi creati ad arte per le storie di cui facevano sfondo[11]. Nel 1984 esce *Elite* (Acornsoft, 1984), un simulatore spaziale, primo vero videogioco a mondo

aperto come lo consideriamo oggi [11]. Anche molte saghe videoludiche sono nate in quegli anni come *Final Fantasy* (Square Enix 1986) e *The Legend of Zelda* (Nintendo, 1986). *The Legend of Zelda* è un chiaro esempio di come, con l'avanzare del tempo, i mondi di gioco si ampliano, e diventano più dettagliati: il giocatore deve esplorare un intero mondo per poter finire il gioco, superando labirinti e incontri con i boss. All'inizio del 2000 i giochi hanno ormai raggiunto gran parte della maturità odierna. Giochi come *Spyro the Dragon* (Insomniac Games, 1999) mostravano una complessità infinitamente superiore a quella di *Bertie the Brain*. Con l'avvento di internet agli inizi degli anni 2000 i videogiochi hanno potuto cominciare ad avvalersi di "patch", miglioramenti che sono fatti dopo il rilascio del gioco e DLC (Downloadable content), contenuto extra acquistabile e scaricabile da internet che amplia i giochi, aggiungendo aree, meccaniche e miglioramenti al gameplay. Uno dei più importanti aspetti che internet ha portato ai videogiochi è stato incrementarne la socialità, dando la possibilità ai giocatori di interagire fra di loro all'interno dei mondi di gioco. È così che è nato il genere MMO (Massively Multiplayer Online) dove decine di migliaia di giocatori possono interagire. Il più grande esempio di questo genere è *World of Warcraft* (Blizzard, 2004), in cui molti giocatori possono condividere un mondo fantasy e cambiarlo grazie alle loro azioni.

1.1 Mondi nei videogiochi

Negli ultimi 20 anni la tecnologia ha permesso al mondo videoludico di crescere enormemente, e insieme ad esso i mondi all'interno dei videogiochi. In particolar modo nei giochi a mondo aperto possiamo vedere diversi esempi di come la tecnologia sia stata utilizzata in maniera diversa. Il primo utilizzo di questa meccanica è *Minecraft* (Mojang, 2010), che ha venduto più di 200.000.000 di copie[9]. *Minecraft*, con il suo stile sandbox, ha creato un mondo in cui il giocatore poteva non solo esplorare e sopravvivere, ma anche costruire. Il mondo di gioco è generato automaticamente, questo permette al giocatore di vedere un mondo diverso ad ogni avvio di partita, controllabile comunque tramite un seed. Questo tipo di generazione procedurale ha permesso, tramite diverse ottimizzazioni, di ottenere mondi di dimensioni importanti. Questo porta però ad avere un minor controllo sui dettagli del mondo, infatti la generazione procedurale di un mondo è di difficile affinatura. Un altro svantaggio dello stile di mondo come implementato su *Minecraft*, è che è basato su Array cubiche[10]. Questo ha portato a scegliere una rappresentazione del mondo "a cubi", che limita nel posizionamento di alcuni blocchi, come mostrato nella Figura 1.1.

Un gioco che ha sfruttato la generazione procedurale in maniera diversa da *Minecraft* è *No Man's Sky* (Hello Games, 2016), che offre una galassia di 2^{64} pianeti[4] con un approccio completamente procedurale. I vari pianeti sono generati tramite terrain, una tecnica di generazione in grado, tramite HeightMap, di generare mesh per il terreno organiche e realistiche. Questo approccio risolve i problemi legati alle Array che ha un gioco come *Minecraft*, ma rende la costruzione meno organica e libera. Ad esempio gli animali presenti in *No Man's*



Figura 1.1: Esempio di stacionata su Minecraft, posizionabile solo al centro del blocco, occupandolo completamente.

Sky sono composti da parti facilmente riconoscibili, che fanno notare la loro natura procedurale. Un altro difetto di *No Man's Sky* è che nonostante vi sia un elevato numero di modelli e algoritmi di generazione, in un universo di gioco così vasto molti pianeti finiscono per essere simili, e poco memorabili. Questo avviene perché con un approccio procedurale lavorare sui dettagli, soprattutto su scale così ampie, è particolarmente difficile. Contrapposti a queste tipologie di gioco, ci sono videogiochi dove il mondo è più uno sfondo a una storia, come *GTA V* (Rockstar games, 2013). In questo caso, il mondo di gioco è molto meno vasto di quelli possibili generati proceduralmente. Questi mondi, infatti, vengono costruiti a mano dagli sviluppatori, andando però a sistemare nella scena ogni singolo dettaglio, spendendoci così molte ore di lavoro. Il vantaggio sul prodotto finale è che, anche se in scala minore, si ha un controllo assoluto sul mondo di gioco in ogni più piccolo dettaglio. Diversi giochi sono riusciti a ricreare città esattamente come nella realtà, come per esempio *Watchdogs* (Ubisoft, 2014) con Chicago. Grazie a questo maggior controllo su tutte le aree del mondo, risultano anche di più facile design altri aspetti del videogioco, come la curva di difficoltà o la possibilità di fare backtracking. Un esempio di buon design dei mondi è *Dark Souls* (Fromsoftware, 2011), dove il mondo di gioco è stato studiato nel minimo dettaglio sia per guidare il giocatore in maniera intuitiva verso la fase di gioco successiva, sia dando allo stesso diversi percorsi per tornare ad aree particolarmente importanti del gioco in fretta, da qualunque parte essi possano essere. Grazie a questa varietà, al giorno d'oggi nei giochi vengono sempre più spinti grandi mondi virtuali ben dettagliati. Giochi come i già citati *Minecraft* e *No Man's Sky* han mostrato al pubblico quanto in grande ormai le nostre simulazioni possono spingersi, mentre giochi come *GTA V* hanno mostrato il livello di dettaglio che si può raggiungere. Per questo alcuni giochi come *7 Days to Die* (Fun Pimp, 2013) o *Scrap Mechanic* (Axelot games 2016) hanno scelto un approccio più ibrido, generando un terrain e sfruttando un Array per poter far le costruzioni o per la gestione del terrain. La Figura 1.2 un esempio di design strutturato a cubi. Questo approccio permette una costruzione abbastanza organica e coe-



Figura 1.2: Scavando un buco nel terreno si può vedere la fondamentale struttura cubica del gioco 7 Days to Die.

rente del mondo di gioco, includendo però anche gli svantaggi degli altri metodi, in quanto, per esempio, rimane difficile il piazzare diversi oggetti all'interno dello stesso blocco, come mostrato nella Figura 1.3 in 7 Days to Die. Essendo inoltre un mondo generato, il dettaglio nel terrain non è sempre perfetto come dovrebbe, invece per quanto riguarda gli edifici si può notare un livello di dettaglio notevole. Questo perché gli edifici non sono generati in modo procedurale ma già disponibili in forma di modelli disegnati in fase di sviluppo e poi ripetuti nella generazione del mondo. Con l'avvento della tecnica molti videogiochi hanno studiato tecniche e algoritmi di generazione, creando un campo di studio molto interessante e ancora pieno di domande aperte.

1.2 Motivazione

Questa poliedricità dei videogiochi di toccare contemporaneamente così tanti campi mi ha sempre affascinato e questo mi ha portato a scegliere una tesi nell'ambito videoludico. Infatti fin dai primi progetti un po' più seri che ho affrontato, il tema della generazione dei mondi è stato preponderante. Il generare mondi in maniera procedurale è sempre stato il passo successivo e una sfida che ho voluto affrontare. Nel panorama videoludico odierno esistono molti tipi di mondi come abbiamo visto, ma ognuno di essi ha delle grandi limitazioni o nella fase di creazione (come *GTA V* e il suo costo esorbitante, limitante per piccole case) o nella gestione dei dati una volta che il mondo è generato. Mi sono per questo avvicinato al tema per studiare meglio le criticità dei vari sistemi, approfondendo lo studio personale all'interno di questo campo. Nella mia tesi ho optato per lo studio di un mondo con generazione procedurale infinita, stile *Minecraft* perché, nonostante la maggior difficoltà in alcuni settaggi, è potenzialmente anche utilizzabile come base per un mondo più piccolo. Avendo il codice sorgente e sapendo come il mondo è salvato, infatti, la costruzione in un secondo momento di un tool che permetta di editare con più precisione un mondo non è particolarmente

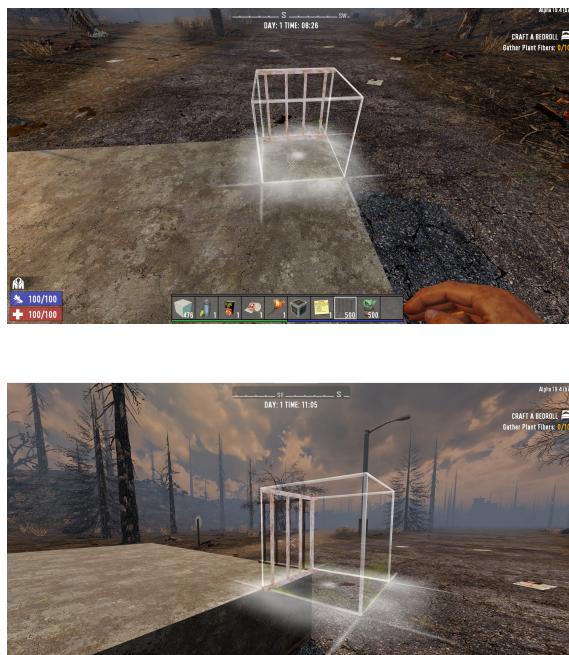


Figura 1.3: Ogni parte della ringhiera occupa un blocco intero sebbene a livello fisico ne occupi solo una parte.

dispendioso. In questo progetto ho voluto fare particolarmente attenzione alla modularità. Grazie a questo tipo di approccio, Unity è facilmente slegabile, con poche classi di puro rendering grafico da riscrivere in caso di utilizzo di un altro engine. L'altro vantaggio di questo approccio è che gran parte degli oggetti all'interno della scena sono facilmente modificabili. Questo permette una maggiore scalabilità del prodotto e facilita la modifica del mondo sia a livello grafico che di rappresentazione.

Un lavoro di questo tipo mi permette di seguire una mia vocazione: in futuro mi piacerebbe riuscire ad aprire una startup che produce videogiochi. Poder partire con un tool che mi permette di velocizzare la creazione del mondo è un vantaggio nel mercato. Vorrei riuscire, in futuro, ad utilizzare questo codice all'interno dei miei progetti, continuando a migliorarlo. Questi fatti mi ha portato a pormi una prima domanda: è possibile creare una generazione procedurale potenzialmente infinita di mondi totalmente slegata da Unity?

1.3 Domanda di ricerca

- È possibile utilizzare un Octree per gestire la generazione procedurale di un mondo?
- È possibile disaccoppiare totalmente la generazione di un mondo procedurale rispetto a Unity?
- È possibile trovare un modo per aumentare la precisione in cui gli oggetti vengono messi all'interno dei cubi?

- È possibile avere una gestione dei Chunk in tre dimensioni?

L'essermi chiesto se la creazione procedurale di un mondo fosse totalmente slegabile da Unity, non è stata l'unica domanda che mi sono posto. Nel panorama odierno, l'esempio più solido di un mondo generato proceduralmente è *Minecraft*, che offre al giocatore un ambiente sandbox dove può esplorare, sopravvivere e costruire ciò che vuole. Scritto in Java, esso permette di generare mondi immensi, in quanto il suo limite è la precisione delle variabili utilizzate per gestire le coordinate del mondo, ovvero dei float.[14] In questo momento *Minecraft*, però, a mio avviso, ha due grandi limitazioni: la prima è che il mondo si espande solo sull'asse X e Z, mentre l'asse Y viene emulato dalla dimensione del Chunk, mentre la seconda è che non si possono piazzare più oggetti all'interno dello stesso blocco. Queste limitazioni, però, non sono presenti solo all'interno di *Minecraft* ma anche alla maggior parte dei giochi che utilizzano le stesse tecniche di generazione dei mondi. Dato che, essendo lo state of the art, *Minecraft* è stato uno degli esempi che ho analizzato più a fondo per questo progetto, mi sono chiesto per quale motivo sono state inserite queste limitazioni, e come poterle affrontare all'interno del mio progetto, anche in vista del mio lavoro futuro. Le mie ricerche mi hanno, inoltre, portato a chiedermi se è possibile utilizzare un Octree per generare mondi potenzialmente infiniti, sfruttandolo per gestire in maniera più accurata sia la memoria che il mondo di gioco. In questa tesi voglio mostrare come ho risolto questi problemi, utilizzando come base del mondo un Octree e offrendo un benchmark con un'implementazione dello stesso fatta in Unity. Ho pensato di organizzare la mia tesi nella seguente maniera:

Nel secondo capitolo presenterò le ricerche fatte, esponendo le mie considerazioni riguardo alle scoperte fatte.

Nel terzo capitolo parlerò in maniera più approfondita dell'architettura che ho pensato, andando a spiegare nel dettaglio la maggior parte delle scelte implementative che ho fatto.

Il quarto capitolo, invece, presenta il benchmark con l'Octree che ho trovato sviluppato in Unity, andandolo ad analizzare e mostrando dati che mettono a confronto le due diverse implementazioni con i loro pro e i loro contro.

Nel quinto e ultimo capitolo traggo le mie conclusioni riguardanti al progetto, andando a valutare oggettivamente il lavoro fatto e parlando dei passi successivi da compiere nell'implementazione.

Capitolo 2

Ricerca

In questo capitolo sono racchiuse tutte le ricerche fatte per permettere l'implementazione del progetto. Si analizza più nel dettaglio le ricerche riguardanti il metodo di renderizzazione scelto per visualizzare i dati generati dal mondo. In seguito sono state fatte ricerche riguardanti la suddivisione dei dati all'interno dei giochi odierni, per valutare una buona suddivisione delle classi e come gestirle in maniera intelligente, scegliendo di suddividere il mondo in Chunk e Regioni. Molte ricerche sono anche state fatte sulle funzioni di rumore, come il Perlin Noise e il Simplex noise, funzioni che generano numeri pseudo-casuali utilizzate all'interno di applicazioni simili per lo stesso motivo. Infine l'ultima sezione del capitolo si concentra sulla spiegazione del Multi-Noise Biome Source, una metodologia di lavoro che permette, grazie all'utilizzo di funzioni di noise, di generare diversi biomi all'interno del mondo, andando anche a valutare come poterli gestire in fase di generazione per avere un risultato organico e realistico.

2.1 Renderizzazione

Per renderizzare il mondo sono ricorso alla Voxellizzazione del mio Octree. La voxellizzazione è la rappresentazione di diversi punti nello spazio seguendo gli assi X Y Z[3], e vista la natura cubica della foglia di un Octree, ho optato per questa forma come base del mondo. Per come ho implementato l'Octree, però, non sono legato necessariamente a questa forma, in quanto le foglie dell'Octree contengono comunque oggetti. Basta implementare un'estensione di quest'ultimi per cambiare il comportamento. Un altro approccio infatti sarebbe quello di sfruttare un HeightMap e salvarla, adeguatamente elaborata, all'interno delle singole foglie. Questo permetterebbe di costruire un vero e proprio terrain di facile modifica. La renderizzazione del mondo è un'aspetto cruciale nella fase di implementazione e fra le più dispendiose dell'intero sistema. Infatti per ogni foglia dell'albero, ogni cubo, è necessario andare a costruire tutte e sei le facce per renderizzarlo. Un'ottimizzazione, però, che si può fare è quella di renderizzare esclusivamente le facce esposte, ovvero quelle che accanto

hanno una foglia vuota o che contiene un cubo trasparente (come il vetro, ad esempio). per far ciò, però, è necessario controllare tutti e sei i cubi adiacenti alle facce di quello che si sta analizzando. Nel caso si lavori con le Array questo lavoro può venir fatto all'interno di tre cicli for innestati l'uno dentro l'altro, mentre con un Octree si può fare la stessa cosa utilizzando un solo ciclo for, abbassando la complessità. Questo però rende molto difficile la logica per il controllo dei cubi adiacenti. Infatti, essendo che hanno dimensione variabile, il controllo diventa molto complicato perché in campo entrano variabili come la dimensione della foglia. Una volta controllati tutti i cubi all'interno di una zona, le mesh risultanti vengono unite in un'unica mesh. Questo permette di risparmiare diverse chiamate alla GPU, ma aggiunge la limitazione che è possibile avere un solo material e quindi una sola texture.

Per questa ragione ho scelto di implementare un Texture atlas, ovvero una texture unica contenente tutte le altre texture riguardanti i cubi e il mondo di gioco, visibile nella Figura 2.1.¹



Figura 2.1: Il Texture atlas utilizzato all'interno del progetto.

2.2 Chunk e Region

Per gestire al meglio il mondo generato, è stata effettuata una ricerca sullo stato dell'arte nei giochi moderni. Per gestire un open world, infatti, spesso si utilizza dividerlo in quadranti. Il terrain o il cubo, di solito, sono l'elemento più piccolo del mondo, ovvero la singola unità che il giocatore può modificare. L'approccio sviluppato in questo lavoro può essere visualizzato nella Figura 2.2.

Molti cubi sono contenuti all'interno di un Chunk, un'area più vasta e di conseguenza di più facile gestione. Infatti il Chunk è un ottimo modo per poter gestire una grande zona modi-

¹recuperato da https://aminoapps.com/c/minecraft/page/blog/mcpe-0-15-0-blocks-and-items-textures/avu0_ueE54zR58KZZDkLxn3Bxbj7v8, sito di modding che rilascia in open source le texture

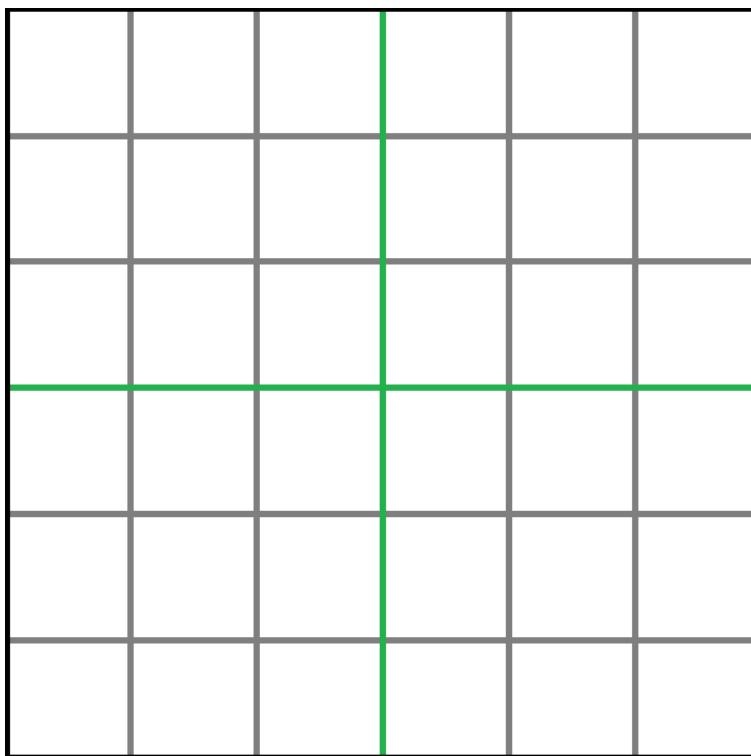


Figura 2.2: Suddivisione di Chunk e Region nel progetto. In nero è rappresentato il bordo del mondo, in verde il bordo della Region e in grigio il bordo dei singoli Chunk

ficabile comodamente. Non solo le funzioni del Chunk possono automaticamente essere propagate ad ogni cubo al suo interno, ma certe operazioni (come l'eliminazione di tutto il Chunk) non hanno bisogno di essere propagate. Questo permette di poter avere anche diverse decine di migliaia di cubi nella scena senza dover necessariamente iterare su tutti ad ogni frame. I Chunk però non possono essere di dimensione troppo elevata, in quanto la renderizzazione dipende da essi. Inoltre il numero di cubi all'interno di un Chunk cresce esponenzialmente rispetto alla sua dimensione, rendendo molto più lento il codice. Infine ci sono le Region, che non sono altro che una collezione di Chunk. Non solo hanno tutti i vantaggi che hanno i Chunk con i cubi, ma permettono di gestire la memoria in maniera più precisa, potendo essere lazy initialized e precaricando, anche da file, solo i Chunk necessari.

Con un mondo generato proceduralmente la dimensione e il tempo di ricerca all'interno di un file è fondamentale. Senza le Region (salvate ognuna in un proprio file), infatti, il mondo necessiterebbe di essere salvato in un file unico, rendendo quasi impossibile la ricerca di un Chunk in caso di mondi molto grandi. Non è realmente pensabile di salvare singolarmente i Chunk, magari uno per file, in quanto la loro ridotta dimensione creerebbe problemi in quanto ogni file salvato rappresenterebbe solo una piccolissima parte del mondo, creando troppo overhead.

2.3 Perlin Noise e octaves

Il Perlin Noise è un tipo di rumore descritto per la prima volta nel 1983 dall'informatico statunitense Ken Perlin[13]. La sua implementazione usualmente è fatta in due o tre dimensioni, ma è un algoritmo facilmente utilizzabile in N dimensioni. Il Perlin Noise è implementato definendo una griglia di vettori randomici, eseguendo alcune funzioni algebriche e infine interpelando i risultati per dare come output sempre un valore compreso fra 0 e 1. Lo pseudocodice del Perlin Noise è mostrato negli Algoritmi 2.1 qui di seguito:

Algorithm 2.1: Perlin noise

```

1 Define grid [NxN]
2 foreach (grid_intersection)
3     define Vector_N(random(-1,1))
4
5 Given a point find the  $n^2$  nearest vector\_\_N,
6 foreach(Vector_n)
7     calculate_offset(Point, Vector_N)
8
9 values []
10
11 for (i < numberOffsets)
12     values[i] = vector_n Dot offset
13
14 return interpolation(values)

```

Un altro vantaggio del Perlin Noise è il fatto che dato un input, restituisce sempre lo stesso risultato. Ciò rende particolarmente facile il seeding, in quanto basta traslare l'input di un valore randomico per cambiare abbastanza i valori da generare un mondo completamente diverso. La complessità del Perlin Noise è $O(2^N)$ dove N è il numero di dimensioni. In computer grafica di solito il Perlin Noise viene usato per generare texture procedurali, in quanto il suo comportamento è pseudo-random. Infatti, nonostante sia differente per ogni valore in una maniera che può parere randomica, due valori calcolati con input simili daranno comunque un output simile. Il Perlin Noise utilizzato all'interno del mio progetto è in due dimensioni, e si basa sulla coordinata X e Z del cubo che sto generando (una maggiore varietà nei cubi è data dai biom).

Quando si calcola il valore del Perlin Noise, se si vuole avere più naturalezza, si possono aggiungere delle "ottave". Questo procedimento è fatto modificando le coordinate prima di utilizzare nuovamente il Perlin Noise. Andando a combinare le coordinate con delle funzioni periodiche possiamo sfruttare proprietà come ampiezza e frequenza per simulare un maggior dettaglio. All'interno di un ciclo for, infatti, si calcola il valore del Perlin Noise diverse volte, ogni volta radoppiando la frequenza e dimezzando l'ampiezza delle funzioni di input. Ogni giro del ciclo è un'ottava. Un maggior numero di ottave porta a un risultato più levigato, dando così un'impressione di maggior naturalezza, andando a ridurre la spigolosità del terreno e dei picchi. Una volta che il valore è stato calcolato, può essere passato all'interno del

programma per essere valutato e utilizzato come input per tirar fuori dati come l'altezza del terreno in quella coordinata, la temperatura, e il bioma di appartenenza del blocco. Grazie a questa tecnica si possono generare proceduralmente diversi piani, con valli e montagne che si estendono quasi all'infinito, in quanto i limiti della funzione sono date dai limiti del tipo di variabili utilizzate. Lo svantaggio del Perlin Noise, inoltre, è che sia i valori di input che il valore di output della funzione necessitano diversi calcoli per poter dare risultati buoni, e sono di difficile tuning.

2.4 Multi noise biome source

Per la generazione dei biomi mi sono appoggiato allo stesso sistema che utilizza Minecraft[10]. Minecraft all'interno del suo motore per scegliere il bioma utilizza un approccio denominato "Multi Noise Biome Source"[8]. Il suo funzionamento è il seguente. Prima di tutto si sceglie il numero di dimensioni che vogliamo che il nostro spazio dei biomi abbia. Per uno spazio a 3 dimensioni sono necessarie 3 variabili. Esse sono slegate dai nomi che hanno, ma è particolarmente comodo vederle nel modo seguente.

- Altezza
- Temperatura
- Umidità

Ognuno di questi valori è calcolato grazie alla funzione di Perlin Noise, ma con diverse funzioni per elaborare l'output del "rumore". Una volta calcolati questi valori, essi vengono riportati ad un valore compreso fra -1 e 1, dopodichè si utilizzano i valori di altezza, temperatura e umidità per scegliere quale bioma deve essere generato in quella posizione, come mostrato nella Figura 2.3. Lo pseudocodice di riferimento è mostrato nell'Algoritmo 2.2.

Algorithm 2.2: Biome generation algorithm

```

1  for(x in Chunk){
2      for(z in Chunk){
3          Temperature = perlin(func(x,z));
4          Height = perlin(anotherfunc(y,z))
5          Humidity = perlin(anotherotherfunc(x,z))
6
7          biome = interpolation(Temperature , Height , Humidity)
8      }
9  }
```

Il risultato di ciò è che, se dichiarati nella maniera giusta, all'interno del mondo i biomi si generano in modo coerente. Come vediamo nella Figura 2.3, infatti, basta aggiungere il bioma "Spiaggia" correttamente all'interno dei bin contenenti i biomi, per essere sicuri che prima di passare da un Oceano a un qualunque altro bioma ci sia una spiaggia. Questo approccio è

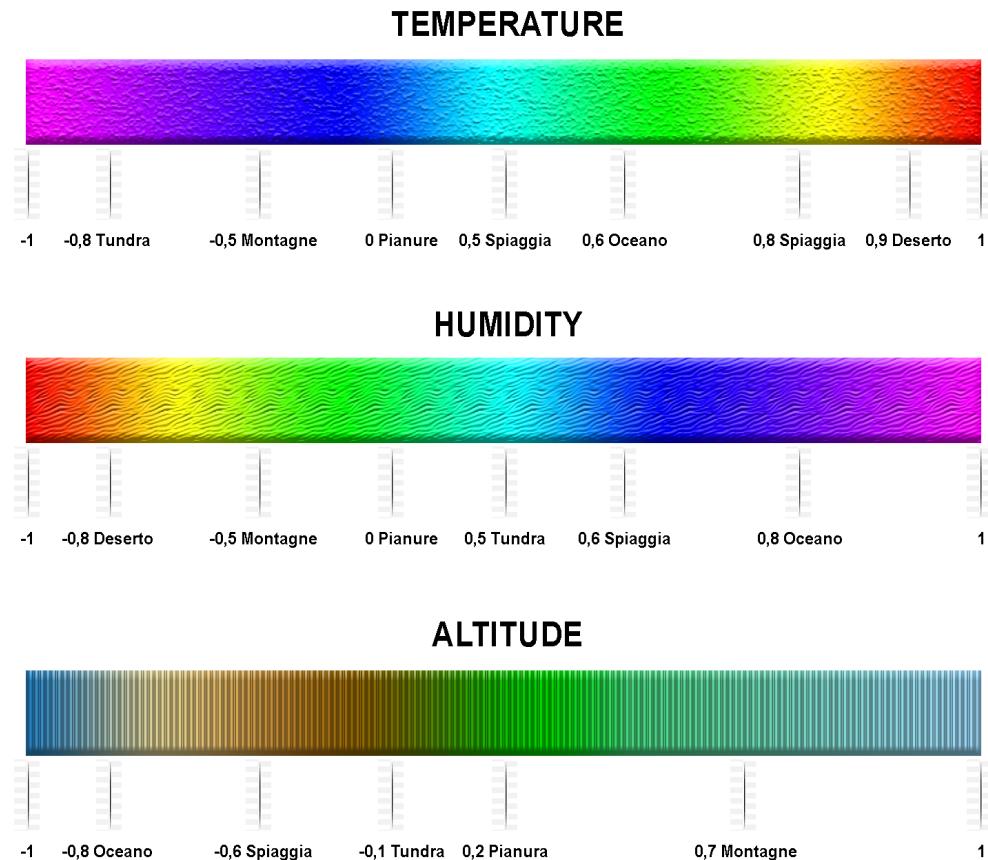


Figura 2.3: Rappresentazione visiva del Multi-noise biome source

molto comodo per creare tanti biomi diversi, anche facilmente configurabili dall'esterno [8]. Lo svantaggio, però, è che il tuning risulta poco intuitivo.

Capitolo 3

Implementazione

All'interno di questo capitolo sono analizzate tutte le scelte implementative fatte durante il lavoro qui presentato. In una prima parte viene descritta in maniera più esaustiva l'architettura del progetto, insieme alla pipeline che il sistema segue dall'inizio alla fine. Nella seconda parte del capitolo invece vengono discussi tutti i dettagli implementativi delle varie classi, portando particolare attenzione alle scelte che hanno permesso un disaccoppiamento totale da Unity e a quelle che hanno permesso una modularità del progetto e la sua ottimizzazione.

3.1 Architettura

Il progetto è scritto in linguaggio C#, e come motore grafico utilizza lo Unity Engine. È stato fatto uno sforzo in più, però, per mantenere Unity il più slegato possibile dal progetto, puntando anche alla portabilità del prodotto finale. Infatti, in caso di cambio Engine, è necessario implementare nuovamente poche classi di puro rendering grafico. Le responsabilità di queste classi è solo quella di incapsulare gli oggetti corrispondenti all'interno del sistema e visualizzarli. Nella Figura 3.1 queste classi sono connesse alle altre tramite linea tratteggiata, per sottolineare i punti di separazione fra Unity e il progetto. Un altro occhio di riguardo è stato dato alla modularità del progetto, cercando di individuare le parti di codice più importanti e astraeendole, utilizzando spesso il strategy pattern e la dependency injection per aiutarmi in questo compito. La struttura dati di base è l'Octree, che non ha alcuna connessione con Unity. L'ho costruito anche con l'utilizzo dei generics per aumentar ulteriormente l'usabilità. Ho sfruttato l'Octree per rappresentare i Chunk nel mondo, al posto di un'Array cubica. Infatti ogni Chunk è un Octree di dimensioni modificabili dall'editor di Unity, dove ogni foglia è un cubo, contenente un oggetto T generico concretizzato in una classe chiamata CubeData. L'oggetto CubeData, facilmente estendibile tramite ereditarietà, contiene i dati del cubo, come il tipo di cubo, la posizione e il bioma. Inoltre se si volesse renderizzare un terrain sarebbe all'interno di questa classe salvati una parte dei

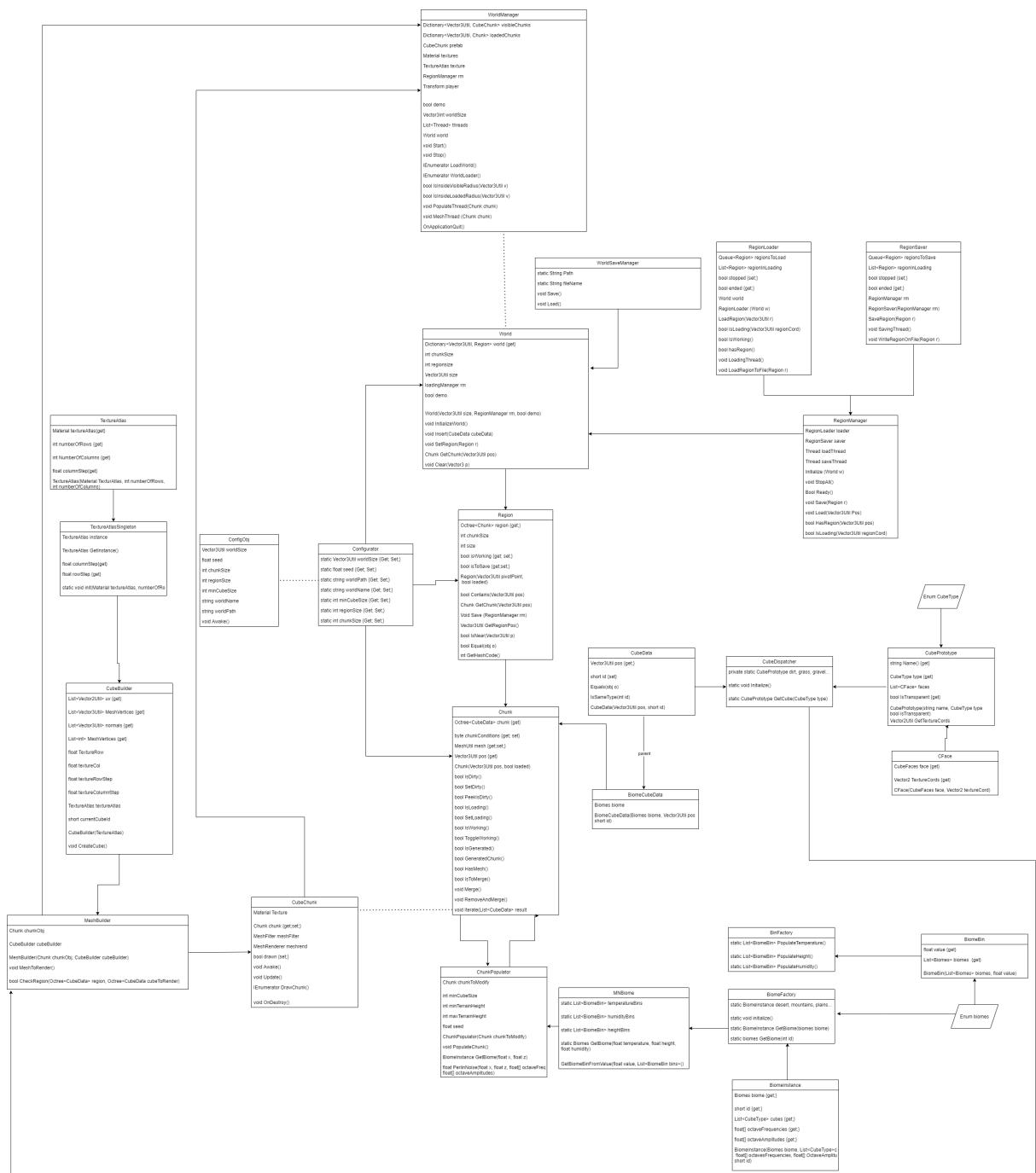


Figura 3.1: Diagramma delle classi dell'intero progetto.

vertici che lo rappresentano. Nel caso all'interno dell'Octree ci siano 8 cubi dello stesso tipo e con lo stesso genitore, esso è in grado di eseguire il merge, togliendo 7 oggetti dalla memoria, senza perdere alcun dato. Questo procedimento avviene ovviamente anche al contrario. Se da un cubo di lato 2 voglio rimuovere un cubo di lato 1, l'Octree prima suddivide il cubo da 2 in 8 cubi più piccoli, e poi rimuove il cubo target. Si sono raccolti i Chunk in

Region, che hanno un funzionamento molto simile ai Chunk, ma non abbastanza da impiegare un composite pattern. Anche all'interno delle Region ho sfruttato un Octree per salvare i Chunk. In questo momento all'interno delle Region l'utilizzo di un Octree è praticamente identico a quello di un Array perché istanzio tutti i Chunk della Region quando la genero. Se venisse sfruttata la lazy initialization, andando a caricare i Chunk solo quando sono strettamente necessari, ci sarebbe un piccolo guadagno di memoria rispetto ad un Array, con un conseguente calo della complessità in quanto si ciclerebbe su meno elementi quando si va a lavorare sull'intera Region. Tutte le Region sono gestite dalla classe del mondo, che ha la responsabilità di caricarle da file, o generarle se necessario, e di salvarle su file una volta che il player è troppo distante. Come prima cosa, nella scena di Unity, viene caricato il mondo, generandolo e passandoli i parametri necessari tramite l'editor di Unity. Questo passaggio avviene tramite una classe statica di configurazione che ha un rispettivo in Unity per poterci accedere tramite Editor. Solo infine, sempre grazie al multithreading, viene renderizzato ogni cubo nella scena. Per fare ciò si itera sull'Octree, ritornando tutte le foglie che contengono un figlio (null rappresenta l'aria). Per ogni cubo, quindi, viene controllato il cubo adiacente ad ogni faccia per decidere se renderizzarla o meno. Qui vengono creati i vertici della mesh e associate le coordinate di texturizzazione in base al cubo che deve essere renderizzato e alla sua posizione nel texture atlas. Infine vengono combinate tutte le mesh del Chunk che sto renderizzando e mostrate a schermo. Questa pipeline è riassunta nella Figura 3.2. Grazie a questo tipo di approccio e a un'organizzazione delle classi con

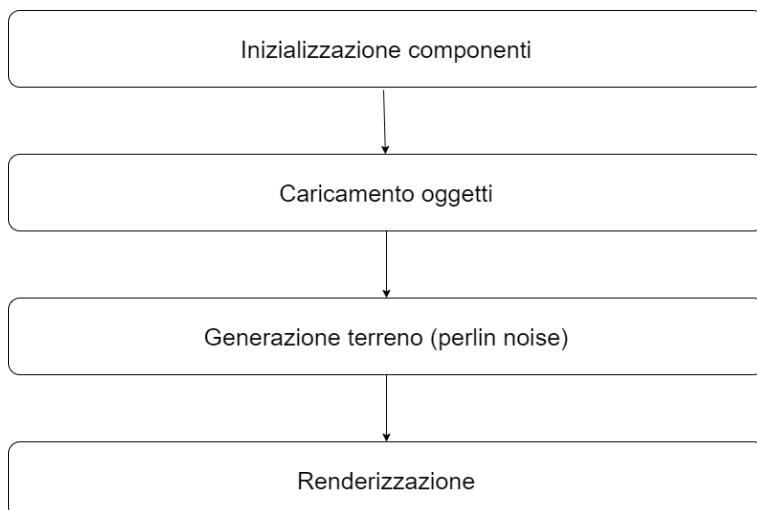


Figura 3.2: Questa è la pipeline del sistema

responsabilità ben definite si è in grado di controllare ogni step della generazione del mondo nel dettaglio.

3.1.1 Cos'è l'Octree

L'Octree è una struttura ad albero in cui ogni nodo ha esattamente otto figli[7]. Questo permette di astrarlo, vedendolo come un cubo dove ogni figlio è un cubo con un'ottava del volume del genitore. La Figura 3.3¹ mostra l'Octree in maniera visuale. L'Octree è nato nel 1980 quando Donald Meagher, dell'i-

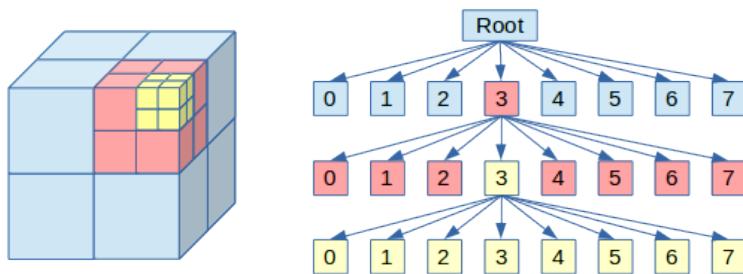


Figura 3.3: Rappresentazione dell'Octree.

stituto politecnico di Rensselaer l'ha descritto nel report "Octree Encoding: a New Technique for the Representation, Manipulation and Display of Arbitrary 3D Objects by Computer"[7]. L'Octree, come struttura dati, porta diversi vantaggi. Infatti rappresentando gli oggetti all'interno di un Octree si ha la possibilità di accedere ad essi tramite la rappresentazione vettoriale della loro posizione. Questo permette anche di localizzare facilmente alcuni oggetti di scena e i loro immediati vicini, senza necessariamente esplorare l'albero per la massima profondità. Ciò permette all'Octree di essere usato in diverse aree nei videogiochi, come per la renderizzazione dei livelli di dettaglio (LOD)[6], o la ricerca del nearest neighbour. Anche per quanto riguarda le collisioni gli Octree possono tornare molto utili in quanto permettono di controllare eventuali collisioni fra oggetti effettivamente vicini senza dover controllare la loro posizione. In un caso come questo, infatti, basta controllare fra di loro gli oggetti all'interno della regione dell'Octree presa in considerazione. L'Octree, però, ha anche diversi svantaggi. Il primo svantaggio è la complessità rispetto ad altre strutture dati. Per inserire o cercare un nodo, infatti, non abbiamo una complessità di $O(N)$ come una lista o come un Array, ma abbiamo una complessità per la scrittura di $O(\log_2 N)$. Un altro svantaggio dell'Octree è che spostare oggetti da un nodo all'altro o rimuoverli e tenere l'albero in uno stato corretto è un'operazione molto difficile, visto che bisogna considerare i casi in cui l'oggetto si sovrappone ai confini della foglia. Nel caso peggiore, inoltre, due foglie adiacenti possono aver come unico parente in comune la root dell'albero, rendendo l'operazione computazionalmente più pesante dato che bisogna passare tramite la root e poi discendere nuovamente verso la foglia target. Per questa ragione nella maggior parte delle applicazioni conviene ricreare l'intero albero da zero ad ogni frame, perdendo performance. La memoria invece è vantaggiosa rispetto a un'Array che avrebbe $dimensione^3 * refWeight$ memoria

¹ Immagine modificata da quella presa nel sito <https://geidav.wordpress.com/2014/07/18/advanced-Octrees-1-preliminaries-insertion-strategies-and-max-tree-depth/>

usata. L'Octree usa $K \log_2(N) * refWeight$. K in questo caso è il numero di nodi dell'Octree e N è la dimensione dell'Octree. Per spiegare in maniera più chiara l'Octree, si prenda per esempio il Quadtree. Il Quadtree (rappresentato nella figura 3.4²) non è altro che una versione in 2D dell'Octree. Il funzionamento è identico in ogni dettaglio a parte dell'aggiunta dell'asse Y. Come ogni albero il Quadtree deve implementare le 4 funzioni di un albero:

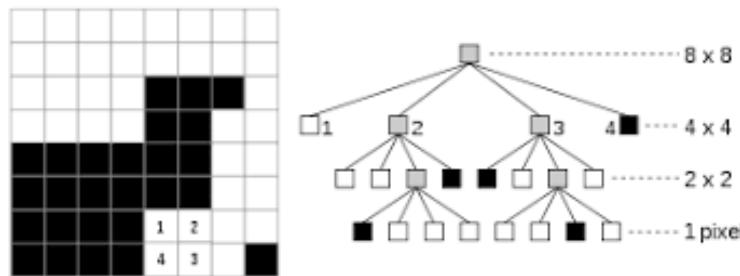


Figura 3.4: Rappresentazione di un Quadtree.

- Inserimento
- Rimozione
- Ricerca
- Lettura

Per l'inserzione in un Quadtree l'algoritmo è molto semplice.

- Si parte dalla root
- Se il nodo inserito all'interno del Quadtree è esterno allo stesso o si è in una foglia non suddivisibile e con al suo interno già un figlio si esce tornando un errore (se non si mette un limite a quanto si può suddividere il Quadtree, i due oggetti finirebbero probabilmente in due foglie diverse, ad eccezione del caso in cui i due oggetti non abbiano la stessa posizione.)
- Si sceglie il nodo figlio in cui l'oggetto deve essere inserito
- Se il nodo figlio è vuoto si mette all'interno l'oggetto inserito. si ferma la funzione.
- Se il nodo figlio ha un oggetto al suo interno ed è suddivisibile si suddivide. si inseriscono a questo punto al suo interno sia l'oggetto da inserire originariamente che l'oggetto che era nel nodo appena suddiviso.

²immagine presa da <https://en.wikipedia.org/wiki/Quadtree>

- controllo il nodo genitore del nodo appena inserito. Se tutti i figli sono oggetti dello stesso tipo, inserisco un'unica istanza di quell'oggetto nel nodo genitore e elimino tutti i nodi figli (merge)

La ricerca invece avviene in questo modo:

- Si parte dalla root
- Se il punto non è all'interno dell'area rappresentata dal Quadtree si stoppa la ricerca e si ritorna un errore.
- Se il nodo corrente è suddiviso si individua in quale nodo è il punto cercato.
- Se il nodo corrente non è suddiviso e non ha oggetti al suo interno, si ritorna null, altrimenti si ritorna l'oggetto all'interno del nodo.

La rimozione invece funziona in questo modo:

- Si parte dalla root
- Se l'oggetto da rimuovere non è all'interno del Quadtree stoppo la rimozione e ritorno False
- Se è all'interno del Quadtree lo rimuovo.
- Si controllano tutti i figli del nodo genitore di quello a cui ho appena rimosso l'oggetto.
- Se il nodo genitore ha tutti figli senza oggetti, li elimino dalla memoria (merge). Il merging viene fatto ricorsivamente fino alla root se necessario.
- Rimosso l'oggetto e esegioto il merge, si ritorna True.

Invece per quanto riguarda l'iterazione ho optato per accedere ricorsivamente, in ordine, in ogni figlio dell'albero, con un approccio Preorder.

Essendo un albero, l'Octree/Quadtree ha una struttura fortemente ricorsiva, in quanto una foglia non è altro che un altro Octree/Quadtree.

3.1.2 Octree vs Array

L'Octree ha diversi vantaggi rispetto all'Array. Prima di tutto a livello di memoria. A parità di dimensioni, infatti, un'Array cubico occupa sempre $N^3 * refWeight$ byte, mentre un Octree occupa $K * \log_2(N) * refWeight$ all'interno della memoria, come si è mostrato in precedenza. Questo permette di salvare una grande quantità di memoria soprattutto in casi in cui l'Octree riesca ad avere pochi nodi. Mentre in un Array la velocità di accesso ad un elemento in una data posizione è costante - O(1) - in un Octree essa dipende dalla profondità dello

stesso, con una complessità di $O(\log_2(N))$ dove N è la profondità del nodo. Nel mio progetto questo valore, comunque, è pensato per non crescere eccessivamente, in quanto più di 8 livelli di profondità implicherebbe una dimensione del Chunk, o una dimensione minima del cubo, che porterebbe ad avere un eccessivo numero massimo di figli, un problema che avrebbe anche l'Array. Infatti, essendo dinamico, a parità di dimensioni il mio Octree, nello scenario peggiore, non avrà mai un numero di elementi maggiore rispetto al corrispondente in un Array. Mediamente un Chunk contiene meno della metà dei cubi che dovrebbero essere salvati in un Array delle medesime dimensioni. Questo valore però fluttua enormemente a dipendenza del quantitativo di cubi diversi all'interno del sistema, senza però mai andare oltre al caso peggiore. Riuscire ad avere un numero così ridotto di elementi, e tenendo la complessità di accesso a un elemento di massimo $O(K^*(\log_2(8)))$, si ha comunque un aumento delle performance rispetto a un Array, oltre che il risparmio di memoria. Questo fatto aiuta anche in situazioni quali la ricerca, la quale in un Array ha una complessità lineare. Uno svantaggio grosso dell'Octree, però, è la difficoltà di implementazione, essendo molto più complicato da gestire rispetto a un Array. Il fatto che l'Octree è un albero porta a un altro vantaggio, in quanto tutte le sue funzionalità sono possibili da descrivere in maniera ricorsiva.

3.2 Sviluppo

Alcuni componenti richiedono una piccola inizializzazione. Questo perché si è voluto sfruttare il flyweight pattern il più possibile. Un esempio di questa applicazione è la classe CubePrototype, che rappresenta in maniera più completa un cubo. Infatti è questa classe che contiene tutti i dettagli che rappresentano il cubo, come il nome, la posizione all'interno del texture atlas e se è trasparente. Questo permette di salvare grandi quantitativi di memoria, in quanto ogni cubo viene definito staticamente. L'inizializzazione serve esclusivamente a creare le diverse istanze di CubePrototype da salvare all'interno delle variabili statiche. Ho utilizzato questo approccio anche con le informazioni riguardanti i biom. Dopo l'inizializzazione, il mondo controlla se deve caricare da file le informazioni sui Chunk vicino al player o se le deve generare. Nel primo caso i dati vengono parsati da file e i direttamente inseriti nelle Region e di conseguenza nei Chunk. Nel secondo caso, invece, I Chunk vengono generati. Per fare questo ho sfruttato il multithreading come per la renderizzazione, in quanto ogni Chunk si genera indipendentemente rispetto agli altri. Per la generazione sfrutto il Perlin Noise, anche se è possibile utilizzare anche un Simplex Noise[12] cambiando una singola funzione. Si è deciso di non implementarlo per non inserire un check sul tipo di noise da utilizzare, in quanto è una sezione di codice che deve eseguire spesso ed è una delle parti che si è dovuto ottimizzare di più. Una buona via per risolvere il problema sarebbe la dependency injection, passando dall'esterno un oggetto che implementa esclusivamente una funzione di noise. Grazie al valore restituito dalla funzione di noise si può a calcolare

tutti i dati necessari, come l'altezza del terreno a quelle coordinate, il bioma e il tipo di cubo da inserire all'interno del Chunk.

3.2.1 Util

Parte dell'implementazione sono classi di utilità che sono state create per tenere disaccoppiato Unity il più possibile. Prima di tutto sono state implementate delle versioni semplificate (con le funzioni necessarie al progetto) del Vector3, in una classe chiamata Vector3Util. Si è aggiunto a questa classe un cast implicito al Vector3 di Unity, in maniera tale che esso venga fatto in automatico negli unici momenti che effettivamente serve passarlo all'API di Unity. Questa classe viene utilizzata molto anche dal mondo come chiave all'interno di un dizionario, per cui ho implementato una funzione di hash in maniera tale da provare a ridurre il numero di collisioni all'interno del dizionario, che in C# è implementato come una HashTable[2]. È stata fatta la stessa cosa per il Vector2, in maniera tale da non dover utilizzare il Vector2 di Unity per salvare all'interno del mio progetto le coordinate UV (coordinate che fanno da ponte tra lo spazio bidimensionale delle texture e quello tridimensionale delle mesh). Stessa ragione per cui esiste anche una classe per le Mesh, in maniera che fosse possibile calcolarle su un thread a parte, in quanto le API di Unity non sono richiamabili da un thread che non sia quello principale. Infine è stata creata anche una classe Math, al cui interno è stato messo il codice necessario a convertire le coordinate del mondo a coordinate delle Region e a coordinate dei Chunk. Queste funzioni potevano essere messe come statiche all'interno delle Region o dei Chunk, ma si è preferito tenerle in un posto centralizzato (sempre tenendole statiche per avere un accesso veloce all'interno del codice).

3.2.2 Octree

Octree
<pre> Vector3Util PivotPoint {get} float size {get} T children {get} Octree<T>[] divided {get} Octree(Vector3Util pivotPoint, float size) void Iterate(List<Octree<T>> result) bool CheckRes(float targetRes) int GetNumberOfChildrens() bool Contains(Vector3Util pos) Octree<T> GetRegion(Vector3Util pos, float res) bool Insert(T obj, float res) bool Remove(T toRemove) void Merge() void Subdivide() </pre>

Figura 3.5: Diagramma classi Octree.

Per l'Octree si è preferito considerare come punto d'ancoraggio quello che si trova in basso a sinistra nella faccia posteriore del cubo, utilizzando un Vector3util chiamato pivotPoint,

con il supporto di un float a rappresentare la lunghezza del lato del cubo. Un altro campo della classe sono un'Array, quando non utilizzata settata a null per salvare ulteriore memoria, che rappresenta eventuali foglie figlie. Ho optato per usare un'Array al posto di una lista per avere un accesso più diretto e una più facile gestione dei figli. Con una lista avrei sicuramente potuto risparmiare memoria, facendo una lazy initialization dei nodi vuoti, ma questo complicava la logica di molto e aggiungeva molti check nelle funzioni più usate dell'Octree. Infine ho un campo T, che estende l'interfaccia IPositionable, che rappresenta il singolo figlio dell'Octree. Anche qui si sarebbe potuto usare una lista per poter salvare più figli all'interno della stessa foglia, ma si è pensato che fosse più pulito, dato anche l'uso dei generici, che nel caso servisse basterebbe passare all'Octree una lista come tipo di figlio direttamente in fase di costruzione. In questa maniera non si è persa l'elasticità che mi potrebbe servire in futuro, ma si è limitato i numeri di calcoli e di cicli che il codice deve fare, dandone la responsabilità a chi lo utilizzerà. In fase di costruzione l'Octree ha bisogno solamente del pivotPoint e della Size. Oltre ai getter per i vari parametri e i figli, l'Octree implementa molte funzioni di utilità, private e non. La funzione più importante è Iterate, che ricorsivamente controlla ogni foglia e aggiunge esclusivamente le foglie che hanno un figlio al loro interno ad una lista passata dall'esterno. Questo permette, dopo un passaggio su tutto l'Octree, di linearizzare l'albero e poter ciclare su di esso con un unico ciclo for, o con un singolo foreach. Invece di passare alle funzioni di insert e remove un oggetto (l'oggetto figlio T) e un vettore, passo solo l'oggetto T. Questo è il motivo per cui implementa l'interfaccia IPositionable, una semplice interfaccia che definisce la funzione *GetPosition*, e quindi il passaggio del vettore viene fatto in maniera implicita, in quanto interna all'oggetto T. Se serve in futuro si potrà facilmente encapsulare l'oggetto List all'interno di una classe che implementa anche questa funzione per aver la possibilità di avere più figli all'interno della stessa foglia. La funzione Subdivide, invece, serve a suddividere l'Octree e generare l'Array con all'interno i nodi figli. Il codice della funzione non è difficile, ma ognuno degli 8 nodi figli deve essere dichiarato separatamente, e poi assegnato all'Array che li contiene. L'ultima funzione all'interno dell'Octree è la funzione merge, che definisce a null l'Array con i nodi figli, liberando così tutto lo spazio allocato per tenere le referenze ad essi, salvando $8 * \text{ByteIndirizzamento}$ byte di memoria

3.2.3 Cubedata & cubeprototype

L'elemento più piccolo del sistema è composto principalmente da due classi: CubeData e CubePrototype. CubeData è la classe base all'interno dell'Octree. Essa contiene una referenza a un Vector3Util che rappresenta la posizione nello spazio del cubo. L'altro campo della classe è l'Id, uno short che rappresenta il tipo di cubo che è (ad esempio, dirt, gravel, ...), visibile nell'Algoritmo 3.1. Si è scelto di rappresentare questo Id come uno short perché difficilmente saranno necessari 32767 tipi di cubo diversi, inoltre risparmia memoria rispetto

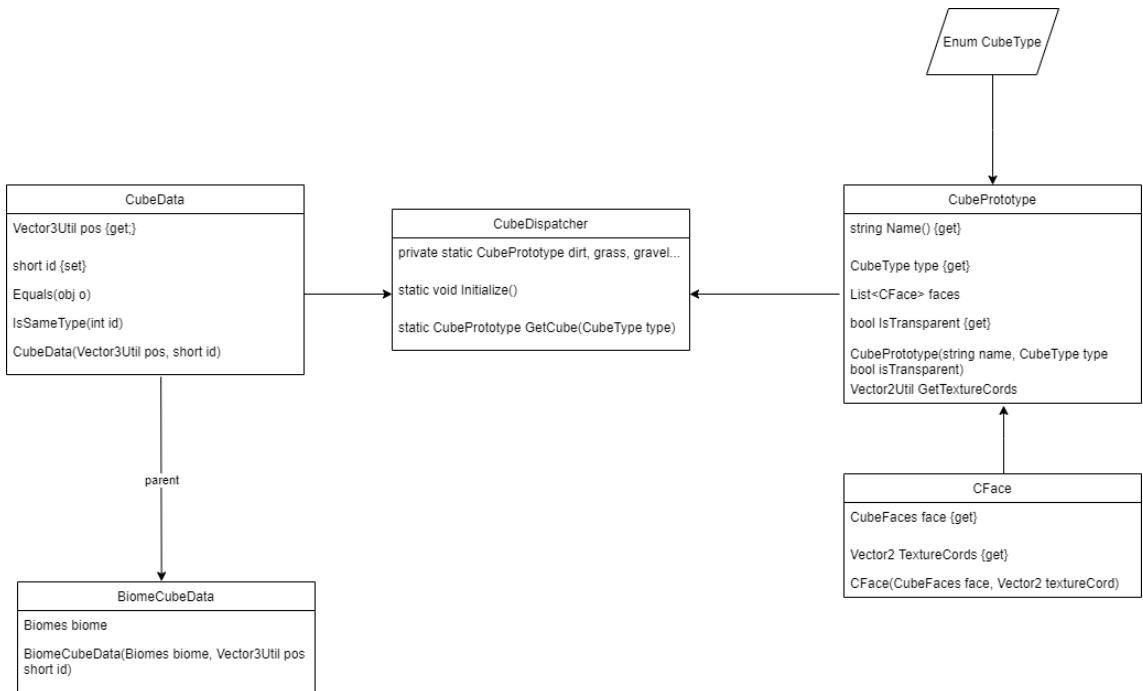


Figura 3.6: Diagramma delle classi di CubeData e CubePrototype

a un intero o un long. Questo Id è anche l'indice di un enumerativo chiamato `CubeType`, per semplicità di scrittura.

La classe `CubeData`, implementa l'interfaccia `IPositionable`, necessaria per funzionare all'interno dell'Octree. Inoltre è estesa dalla classe `BiomeCubeData`, che contiene un enumerativo, rappresentante del bioma in cui il cubo è contenuto. Ho scelto di salvare questa informazione all'interno del cubo perché potenzialmente è un'informazione che può venir chiamata diverse volte all'interno di uno stesso frame, quindi ho preferito sacrificare un po' di memoria in favore di un minor numero di calcoli nella CPU.

Algorithm 3.1: Enum dei tipi di cubo.

```

1 public enum CubeType {
2     ERROR = 0,
3     COMPLEX = 1,
4     DIRT = 2,
5     GRAVEL = 3,
6     SAND = 4,
7     GRASS = 5,
8     STONE = 6,
9     REDSAND = 7,
10    WATER = 8
11 }

```

La classe `CubePrototype`, invece, rappresenta la tipologia di cubo. Oltre a contenere il nome del tipo di cubo, contiene un Enumerativo che lo rappresenta. Si è optato per fare una classe singola contenente tutti i dati come parametri, piuttosto che creare una gerarchia per evitare

problemi di class explosion in un progetto che già ne soffre. Oltre a ciò ogni CubePrototype contiene un booleano che rappresenta se il cubo è trasparente (i.e., acqua) oppure no. Infine contiene una lista contenente l'istanza della classe CFace. questa classe di supporto, rappresenta le facce del cubo e contiene le coordinate UV necessarie per renderizzare la sezione giusta del Texture atlas. Le CFace possono essere di 9 tipi diversi, una per direzione (Bottom, Top, ...) e una che mi rappresenta tutte le facce non presenti nella lista. Le CFace sono state fondamentali perché in questa maniera, con una semplice lista, posso cambiare la texture ad ogni singola faccia del cubo indipendentemente. In questa maniera posso creare cubi come l'erba, dove la faccia superiore deve renderizzare verde, quelle laterali devono passar da verde a marrone e quella sotto marrone, come nella Figura 3.7. Allo stesso tempo, però, se un cubo ha tutte le facce uguali, o solo una diversa, basta passargli un numero limitato di oggetti. Oltre alla comodità questo riduce anche il peso in memoria, in quanto non necessariamente tutti i cubi hanno un oggetto per rappresentare ogni faccia. La classe CubePrototype viene gestita dalla classe CubeDispatcher. Questa classe contiene una variabile statica per ogni tipo di cubo presente, come visto nello snippet di codice 3.2.

Algorithm 3.2: CubeDispatcher variable list

```

1  public class CubeDispatcher
2  {
3      private static CubePrototype dirt, grass, gravel, redsand, stone, sand, water;
4
5      public static void Initialize() {
6          dirt = new CubePrototype("Dirt", CubeType.DIRT);
7          dirt.AddFace(new CFace(CubeFaces.All, new Vector2(14, 18)));
8
9          ...
10     }
11
12     public static CubePrototype GetCube(CubeType type) {
13         switch (type) {
14             case CubeType.DIRT:
15                 return dirt;
16
17             ...
18         }
19     }

```

Si è optato per renderle statiche per poter sfruttare il flyweight pattern, e condividere più informazioni possibili tra i cubi. Questo ha permesso di salvare molta memoria all'interno del programma, oltre che a ridurre le chiamate al garbage collector, rispetto al generarle e cancellarle ad ogni necessità. L'unico drawback di questa implementazione è che necessita un'inizializzazione alla partenza del programma, per poter popolare una singola volta le variabili con i CubePrototype corrispondenti. Una volta inizializzato, il CubeDispatcher ha una sola funzione al suo interno. Dato il tipo di cubo (l'id di cubeData) ritorna il CubePrototype corrispondente.

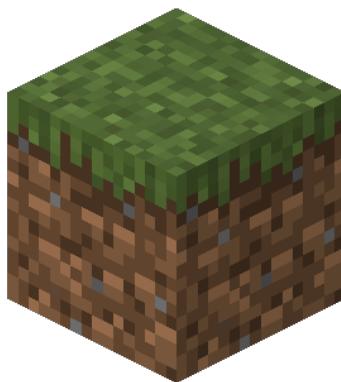


Figura 3.7: Cubo rappresentante l'erba. Ogni faccia ha una texture diversa.

3.2.4 Chunk, Region & world

3.2.4.1 Chunk

I Chunk sono oggetti creati per rappresentare piccole aree di spazio e raggruppare più CubeData sotto uno stesso cappello. Il Chunk ha una posizione, che corrisponde al pivotpoint dell'Octree al suo interno, per consistenza. All'interno della classe Chunk esiste una variabile chiamata ChunkConditions. Questa variabile è un byte, che, con l'aiuto di una maschera, è utilizzato per salvare fino a 8 variabili booleane. È stato scelto questo approccio perché riesce a ridurre drasticamente la memoria utilizzata per il salvataggio dei booleani. L'ultimo campo della classe Chunk invece è di tipo MeshUtil, ed equivale a null. Questo perché se viene creato un Chunk molto probabilmente si dovrà renderizzarlo a breve e aggiungere una referenza ad aggetto non impatta quanto impatterebbe farlo ai cubi. Questo campo mi permette di iniziare a pre-generare la mesh ancora prima che Unity la richieda effettivamente. Visto che è una funzione pesante la generazione della mesh, poterla generare mentre il player ancora si sta muovendo nel mondo permette di guadagnare frame in fase di renderizzazione, in quanto si tratta solo di convertirla a mesh di Unity e applicarla. Vi è una funzione che controlla se una data foglia dell'Octree può essere unita tramite la funzione merge. Per farlo si va a individuare il suo Parent e si controlla se ha dei nodi figli o meno. Se ha nodi figli, si controlla per ognuno che sia effettivamente una foglia, e se contiene o meno un oggetto. Se tutte le foglie figlie hanno lo stesso tipo di cubo o sono null, allora si ritorna True. Alla prima differenza fra i contenuti delle foglie figlie si ritorna False, interrompendo ulteriori controlli.

3.2.4.2 Region

La classe Region, è una collezione di Chunk della medesima zona. Anche la Region ha al suo interno un Octree, in quanto in caso di Lazy initiazation si risparmia memoria. Nel caso dell'implemtazione sviluppata, però, all'interno del costruttore della Region si va già

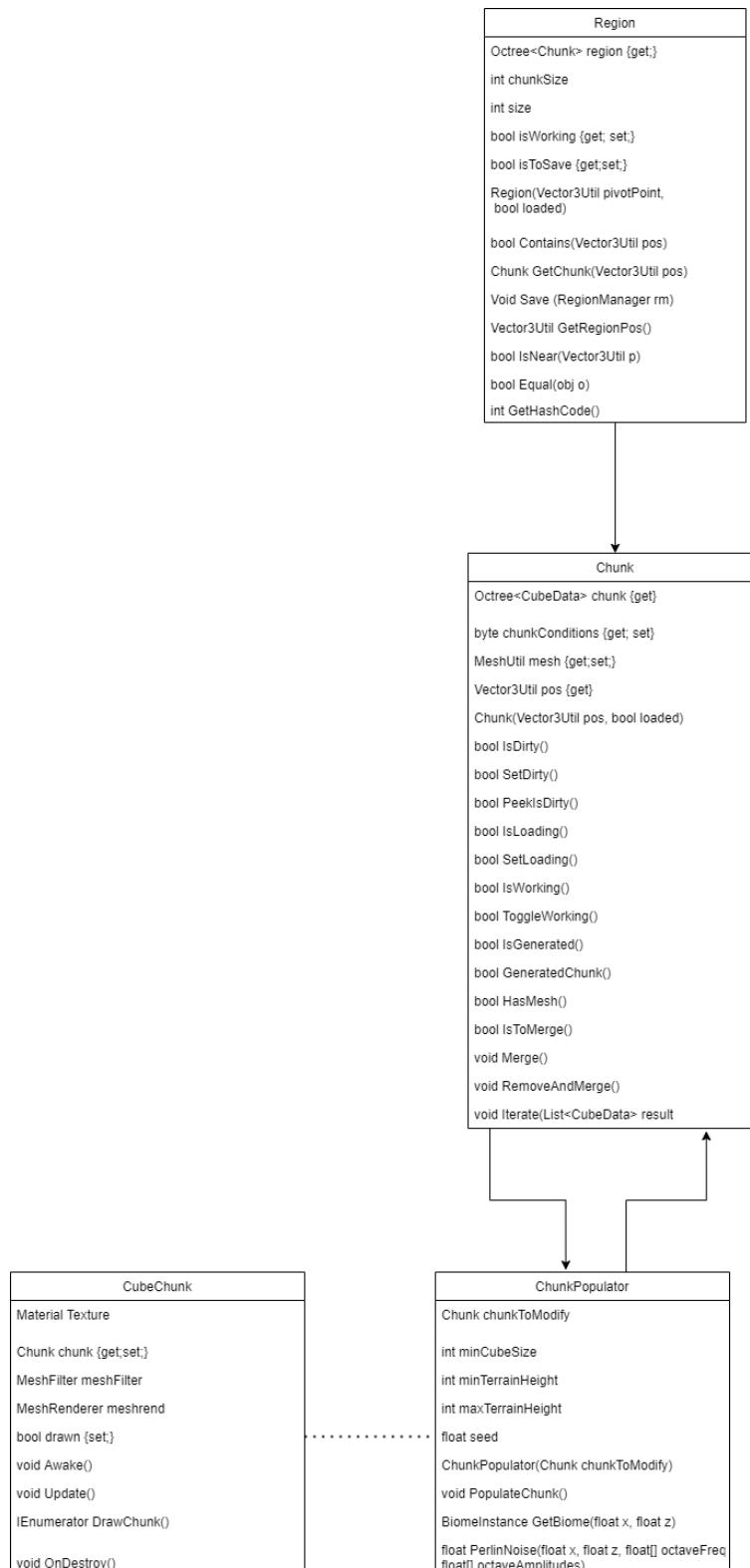


Figura 3.8: Diagramma delle Region e dei Chunk

a istanziare tutti i Chunk che la compongono. La generazione o il caricamento del Chunk vero e proprio avviene solo su richiesta, ma l'oggetto di per sé è già pronto in precedenza, va solo popolato. Gli altri campi della classe sono la dimensione del Chunk e la dimensione della Region. Anche questi parametri, come la dimensione del Chunk all'interno della classe Chunk, sono statici in inizializzazione, ma messi all'interno delle Region per semplificare il multithreading. Le Region in memoria, infatti, non sono mai più di 8, quindi questa perdita di memoria è trascurabile. Infine si hanno due booleani, uno che indica se la Region sta lavorando, caricando o salvando Chunk su file, mentre l'altro, settato a True solo quando un Chunk della Region viene modificato, serve a dirmi se devo salvare la Region quando sarà da scaricare su file. Infatti se un player passa attraverso la Region senza modificare alcun Chunk, sarebbe inutile riscrivere tutta la Region su file. In fase di generazione, ovviamente, l'inserimento dei cubi per rappresentare il terrain generator è considerato una modifica alla Region, e quindi almeno la prima volta che ci passa il player essa viene salvata su file. La Region, inoltre, contiene funzioni che aiutano a individuare se il giocatore è vicino alla Region in questione o se questa contiene una determinata posizione. Anche i Chunk possono essere facilmente recuperati da questa classe, nel caso bisognasse lavorarci in maniera più precisa.

3.2.4.3 World

Infine si ha la classe World, che contiene un dizionario con chiavi dei Vector3Util e come oggetti le Region corrispondenti. Questo approccio è molto comodo perché l'accesso a dizionario, implementato come un hashtable[2] all'interno di C#, è O(1). Questa classe gestisce solo questo, caricando e scaricando su file in caso di necessità le Region che gli vengono richieste. Prima di creare tutte le Region, si controlla se esistono già salvate su file. Solo in un secondo momento viene controllato su file se esiste. Se esiste viene popolata con i dati letti da file, altrimenti viene segnata come caricata, e il resto del programma gestisce la generazione de terreno e la costruzione della mesh. Inoltre si è dotato il mondo di una modalità "Demo", che non salva nulla su file generando di volta in volta i Chunk. Questa modalità è quella che in uso, per esempio, nel menu di gioco, quando serve un mondo rapido da mostrare senza effettivamente salvarlo. Per renderizzare il mondo e gestirlo dipendentemente del frame, si è creato un wrapper in Unity, il WorldManager. Questa classe si prende cura di istanziare il mondo di gioco e di richiedere le Region in maniera accurata. Per fare questo ha però bisogno di un po' di funzioni ausiliarie. Le due funzioni ausiliarie più importanti sono PopulateThread e MeshThread, tutte e due che ricevono in input un Chunk. Il funzionamento di ambedue è triviale, in quanto creano un thread per popolare il Chunk e creare la mesh del Chunk rispettivamente, e lo fanno partire. In questa maniera sia la generazione del terreno e la creazione della mesh avvengono su thread personali che fanno andare il codice in parallelo, dandomi la possibilità di tenere diversi Chunk in diverse fasi della generazione e caricarli prima che siano necessari. Con i Chunk grandi 16x16x16 il tempo



Figura 3.9: Diagramma delle classi del mondo.

medio di generazione è 100ms come il tempo per generare la mesh. Per questa ragione si sono create due aree di caricamento, il `visibleRadius` e il `loadedRadius`. Tutti i `Chunk` all'interno del `visibleRadius` vengono effettivamente istanziati come `CubeChunk` e visualizzati, se pronti. I `Chunk` all'interno del `loadedRadius`, invece, sono nei vari step della pipeline. Se si stanno caricando da file aspettano di finire e poi generano le mesh, altrimenti prima generano il proprio terrain e solo successivamente la mesh. Idealmente durante il tempo che il player si muove abbastanza nel mondo per richiedere un nuovo `Chunk`, in memoria essi sono già pronti da visualizzare. Tutte le `Region` al di fuori del `loadedRadius` vengono salvate su file e scaricate dalla memoria. Per sapere se un `Chunk` è all'interno di un'area, che sia la `visible` o la `loaded`, ho costruito delle funzioni apposite. Queste funzioni, semplicemente, data la posizione del player controllano se il `Chunk` è all'interno di un determinato raggio. Questo processo è visualizzato nella figura 3.10.

Infine il `WorldManager` ha due coroutine. La prima Coroutine è quella che carica il mondo per la prima volta. Controlla se c'è da caricare un file o generare un nuovo seed, e ini-

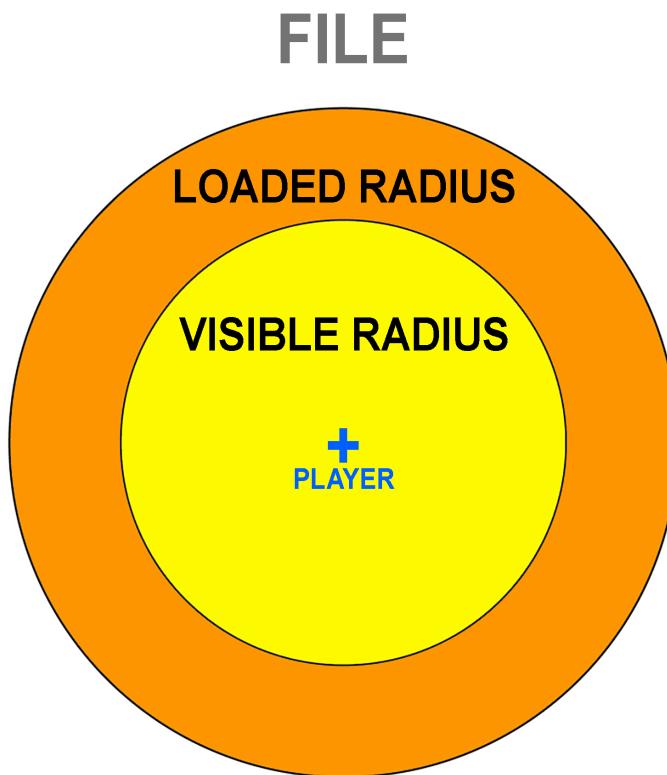


Figura 3.10: Il player è il centro del visible-radius, mentre i Chunk non ancora renderizzati ma in preparazione fra essi sono nel loaded radius

zializza gli ultimi componenti, che necessitano dati direttamente dal mondo. Finito questo passaggio l'esecuzione passa alla funzione principale, che tiene il mondo caricato. Questa Coroutine, principale bottleneck dell'applicazione, è il cuore del sistema. Prima di tutto si salva la posizione corrente del player per quel ciclo. Dopodichè scarica dalla memoria tutte le Region troppo lontane dal player, ciclando su tutte le Region in memoria e controllandole una ad una. fa la medesima cosa per i CubeChunk renderizzati che ormai son troppo lontani dal player. Dopodichè cicla su un'area grande quanto il loadedRadius spostandosi di volta in volta della dimensione di un Chunk. Come se fosse un'Array cubica in questo punto c'è un controllo di ogni Chunk della loaded area. Queste due aree sono rappresentate da due dizionari, per evitare di dover richiamare i Chunk ad ogni frame dal mondo. Solo come primo check si deve controllare se il Chunk è già all'interno del dizionario che sto utilizzando, altrimenti si recupera dal mondo. Se il Chunk non è stato generato, non sta già lavorando e non si sta caricando da file, viene passato al ChunkThread per farlo inserire all'interno di un Thread per generare il terrain, altrimenti si controlla se non sta lavorando, se è stato generato, non si sta caricando e non ha la mesh, viene passato al MeshThread per generare la mesh. Se invece il Chunk è all'interno del dizionario in cui metto i Chunk del visibleRadius, si elimina e viene cancellata l'istanza del CubeChunk dalla scena. Se invece il Chunk è

all'interno del visibleRadius ma non del rispettivo dizionario, si instanzia un CubeChunk con il rispettivo Chunk. L'ultima funzione del WorldManager è la funzione Stop, che si prende cura di fermare tutti i thread dell'applicazione quando essa viene chiusa, o quando si torna al menu principale, dando così il tempo ad eventuali Chunk di salvarsi su file e tutto di concludere senza errori.

3.2.5 Terrain generation & Biom

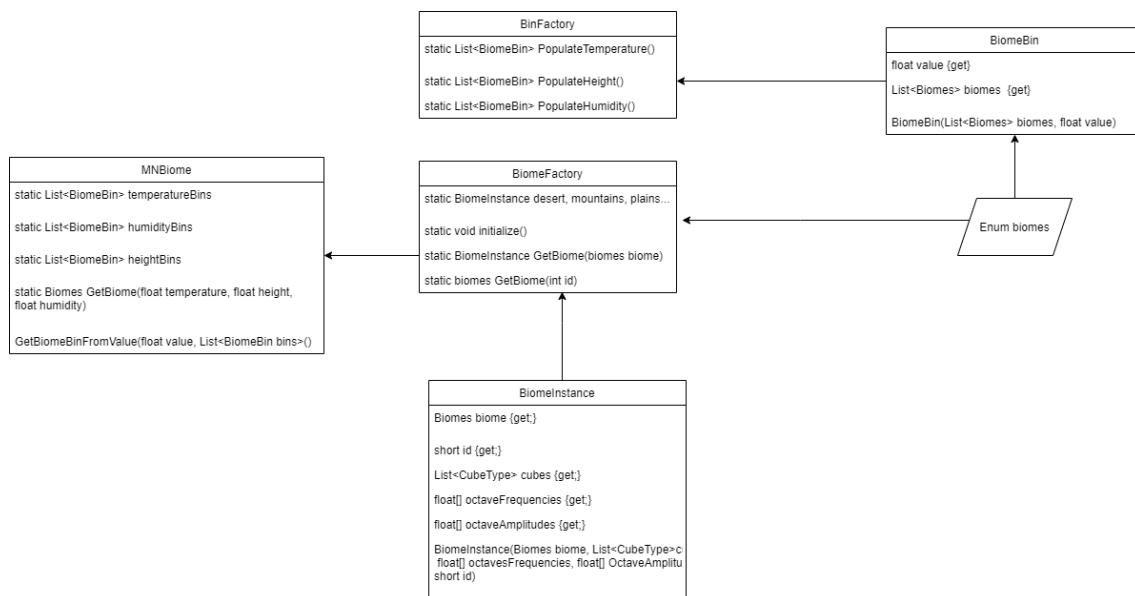


Figura 3.11: Diagramma delle classi della generazione del terreno e i biom

Questo modulo è particolarmente corposo, in quanto al suo interno ci sono diverse classi che servono alla generazioni dei biom. Il punto focale di questa parte è la classe Chunk-
Populator. Questa classe, che esegue su un thread a sè stante per ogni Chunk che deve
essere popolato, contiene la funzione PopulateChunk. Il flow della funzione è relativamente
semplice, in quanto cicla con due cicli for su tutto il Chunk, muovendosi di volta in volta
di una dimensione standard per i cubi. Così facendo, sostanzialmente, si muove sull'asse
delle X e sull'asse delle Z. Innanzitutto si utilizzano le coordinate X,Z per calcolare in quale
bioma ci si trova. Come secondo step si passano, insieme al seed, le coordinate X Z at-
tuali al Chunk, e tramite un'espressione matematica. si calcola l'altezza massima dei cubi
in quella zona. In questa maniera si riesce a generare un terreno proceduralmente senza
dover utilizzare i random. In questo passaggio si sono inserite le ottave del Perlin Noise, che
permettono di dare ad ogni bioma la propria impronta stilistica e la propria differenziazione.
Finito di popolare il Chunk, si fa un update a Chunk che ha finito di lavorare ed è stato ge-
nerato. Con questi dati è facile scegliere il blocco giusto da piazzare grazie al sistema che
è stato creato per i biom. Questo processo è spiegato meglio nello snippet di pseudcodice

 Algorithm 3.3: ChunkPopulator

```

1  for (x in Chunk){
2    for (z in Chunk){
3      BiomeInstance cubeBiome = GetBiome(x, z);
4      float value = PerlinNoise(x + seed, z + seed, biome.Frequencies(), biome.Amplitudes());
5      int y = getHeight(x,z);
6      short id = CubeDispatcher.GetCube(cubeBiome.GetCube());
7      for (Chunkheight to y inside Chunk){
8        ChunkToModify.Insert(generatedCube);
9      }
10    }
11  }

```

La classe BiomeInstance è la classe che si utilizza per rappresentare il bioma. Oltre a un enumerativo per il tipo di bioma, ha anche un Id, il bioma al suo interno ha salvato i dati per poter fare le ottave, e una lista di cubi utilizzabili all'interno di quel bioma. Questa classe si comporta essattamente come il CubePrototype, sfruttando il flyweight pattern per condividere le informazioni fra diversi cubi senza appesantire la memoria. Un'altra classe importante è il BiomeBin, che aggredisce insieme un bioma e un valore che può andare da -1 e 1. Queste classi rappresentano all'interno del codice il Multi Noise Biome Source. Ogni bin rappresenta un bioma sulle scale del Multi Noise Biome Source, in maniera tale che tramite un algoritmo posso scegliere il bioma corretto. Questi bin infatti vengono inseriti in tre liste, una per le altezze, una per le temperature e una per le umidità. Dati due valori X Z, ne calcolo il Perlin Noise. A questo punto si generano temperatura altezza e umidità inserendo il valore appena calcolato all'interno di diverse sinusoidi. Questo aiuta a far rimanere i valori fra -1 e 1 e a tenere una certa regolarità. La scelta vera e propria viene fatta all'interno della classe MNBiome, in una funzione statica chiamata GetBiome(). All'interno di questa funzione, che per input riceve i tre valori di altezza temperatura e umidità, cicla su ogni lista con ogni valore, andando a recuperare il bioma che si avvicina di più. Una volta che si ottengono i tre bin, si controlla se c'è un bioma che compare con una frequenza maggiore rispetto alle altre, e si procede alla scelta.

3.2.6 RegionManager & saving

Visto il grande numero di dati all'interno del mondo, ci si è domandati come tenerli salvati su file. Si è optato per creare file delle dimensioni delle Region, riducendo di molto il numero di file da salvare e avendo comunque un piccolo spazio di ricerca per ogni file, così che non intacchi troppo le performance. Per far interagire le Region sui file, si sono create due classi: il RegionLoader e il RegionSaver. Queste due classi girano ognuna su un thread dedicato. Il RegionSaver, ha due principali funzioni. La prima è SavingThread, composta da un semplice ciclo che continua ad eseguire fino al termine del programma che costantemente controlla se in c'è una Region da salvare. Nel caso ci sia la passa alla seconda funzione di questa classe, WriteRegionOnFile. Questa seconda funzione serve a scrivere

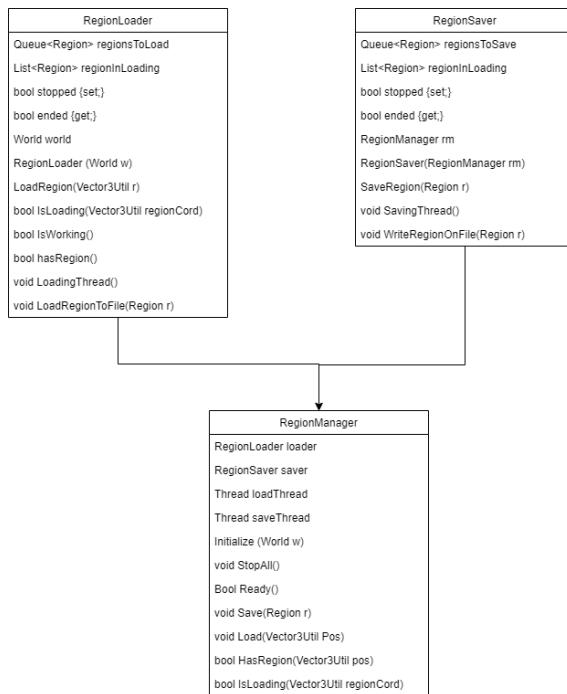


Figura 3.12: Diagramma delle classi della generazione del terreno e i biomì.

la Region su file. Per ogni Region che deve esser salvata viene creato un file chiamato con la posizione della Region. Questo permette di evitare di salvar la posizione dentro al file, facendo risparmiare tempo in quanto accede direttamente tramite nome file, senza dover ciclare sui file all'interno della cartella per controllare quale esiste. La prima funzione che si salva è il numero di Chunk all'interno della Region. Dopo di ciò un viene fatto un ciclo su ogni Chunk, e per esso viene salvata la posizione, un booleano che rappresenta se è stato già generato o meno e il numero di cubi che esso contiene. Solo infine esiste un terzo ciclo che itera su ogni CubeData all'interno del Chunk per salvarne la posizione, la dimensione, presa dall'Octree, il bioma e il tipo di cubo. Il RegionLoader, invece, funziona nella stessa maniera, leggendo nello stesso ordine i dati da file e caricando le Region. L'unico dettaglio che differenzia queste due classi è nella funzione di incodamento delle richieste. Infatti quando si chiede una Region al RegionLoader, come prima cosa la istanza, definendola come in caricamento, e la restituisce subito al mondo, mettendola contemporaneamente nella coda. Quando il thread cerca di caricarla da file, controlla per prima cosa se il file esiste. Se il file non esiste setto la Region come caricata e il mondo si prende cura di generare i Chunk e renderizzarli come visto in precedenza. Il RegionLoader e il RegionSaver sono gestiti dal RegionManager, una classe a cappello che serve principalmente a gestire il RegionSaver e RegionLoader. Non solo fa partire i thread di salvataggio e caricamento, e delega le funzioni di salvataggio e caricamento al RegionLoader e al RegionSaver, ma permette anche di fermare questi thread o di controllare se il RegionLoader sta caricando una particolare regione. Per fare ciò controlla semplicemente che la regione che si sta caricando sia all'interno della

coda di caricamento o se sta lavorando ed è in caricamento. Non basta controllare questa seconda cosa perché in alcuni corner cases si creano problemi di concurrency.

3.2.7 Render

Per renderizzare il tutto, si hanno classi che aiutano nel processo. Una delle classi più importanti è la classe `TextureAtlas`. Un texture atlas non è altro che una collezione di texture messe nella stessa immagine. Questo permette di poter renderizzare tutti i cubi di un Chunk all'interno della stessa mesh, in quanto una mesh può avere un solo material e quindi una sola texture. La Figura 3.13 mostra il texture atlas utilizzato.



Figura 3.13: Il texture atlas utilizzato all'interno del progetto, unico file di texture.

Avendo tutte le texture sullo stesso file, il modo per dare diverse texture a diversi cubi è quello di calcolare gli UV in maniera corretta. Questa classe è molto semplice, serve per definire la texture che si utilizza, oltre che calcolare, date le dimensioni, il numero di righe e di colonne del texture atlas. Si calcolano anche due valori, salvati in un `Vector2`, che sono la larghezza e l'altezza di ogni singola texture. In questa maniera, gli indici all'interno delle `CFace` permettono di calcolare gli UV corretti per il mappamento delle Texture. Per quanto riguarda la creazione della mesh del cubo si è lavorato su una classe chiamata `CubeBuilder`. Al suo interno questa classe accetta un `CubeData` con `CFace` e una lista delle facce da renderizzare. Per ogni faccia si va a istanziare dinamicamente i vertici del quadrato che la compone, tenendo in considerazione la dimensione e la posizione. Il `cubeBuilder`, al suo interno ha una funzione creata ad Hoc per ogni faccia del cubo singolarmente, in maniera da poter aver un controllo totale sulle facce renderizzate. I dati del cubo appena creato vengono salvati all'interno di diverse liste (vertici, triangoli, normali, uv) pronte per essere recuperate una volta che tutti i cubi del Chunk sono stati processati. Il flow del processamento dei cubi avviene all'interno della classe `MeshThread`. Questa classe, lanciata dal `WorldManager`, ha una funzione che accetta un `Chunk`. Inizialmente itera su di esso grazie alla funzione

dell'Octree per iterare e per ogni cubo fa diversi controlli. Infatti per ogni faccia del cubo, bisogna valutare se esiste un cubo adiacente, e se è trasparente. Se la faccia che si sta controllando effettivamente da sull'aria (valore null) oppure su un cubo trasparente (come il vetro) la si deve aggiungere ad una lista. Una volta controllate tutte le facce del cubo è possibile passare la lista al CubeBuilder. Quando il programma ha finito di ciclare su tutti i cubi del Chunk, viene preparata la MeshUtil e assegnata al Chunk target. A quel punto in caso di necessità il Chunk ha i dati per renderizzarsi. Tutte le Mesh di un Chunk vengono combinate all'interno di una mesh unica per comodità e performance. Le mesh infine vengono mandate attraverso i processi di batch utilizzati da Unity. Quando Unity effettua un render raccoglie i dati all'interno di pacchetti (batch) per passargli alla GPU per renderizzarli. Le Batch però vengono richiamate per ogni mesh separatamente e questa operazione è molto dispendiosa, per cui è bene limitarla.

Capitolo 4

Benchmark

In questo capitolo è descritto il benchmark che è stato fatto fra l'implementazione dell'Octree di questo progetto e un'implementazione dello stesso fatta in Unity. Sia il metodo di valutazione che i risultati delle ricerche sull'argomento sono descritte qui in maniera più dettagliata.

4.1 Octree vs Octree

Nella scena di Unity l'Octree ha diversi utilizzi e ci sono diversi tipi di implementazioni che si basano tutte sull'Octree implementato da Unity nel loro github Unity-technologies¹. L'implementazione di Octree da parte di Unity ha due modalità, la modalità Boundaries, che permette un più facile utilizzo di features di Unity come le collisioni, e la modalità Point-based, molto simile all'implementazione mostrata in questa tesi. Oltre a fare merge dei nodi, questa versione dell'Octree è in grado di allargarsi e stringersi, andando a rappresentare un'area spaziale più o meno grande, a seconda delle necessità.

Per stringere l'Octree, dalla root viene controllato ricorsivamente se c'è solo una regione che al suo interno ha degli elementi. Nel caso esista, quella diventa la nuova root. Per allargarlo invece viene calcolato dove sarebbero i 7 nodi che condividerebbero il genitore con l'attuale root e assegnano una root comune a tutti. Anche questo passaggio avviene ricorsivamente, in quanto anche se si sta mettendo un oggetto molto lontano, ogni volta che si allarga l'Octree raddoppia le sue dimensioni, con una crescita esponenziale. Questo Octree però è difficilmente paragonabile all'implementazione presentata in questo lavoro di diploma, in quanto utilizza al suo interno diverse funzioni di Unity, anche per calcolare i Bound, funzioni che son computazionalmente pesanti. Secondo l'autore la maggior parte del tempo la si perde attraversando tutti i nodi. Una soluzione che suggerisce è quella di trovare un modo per linearizzare i nodi e cercarli all'interno di una singola Array. Per questo benchmark è stata testata l'implementazione presentata di Octree con esso, mettendo a

¹<https://github.com/Unity-Technologies/UnityOctree>

confronto 1000 inserimenti e 1000 rimozioni. Ambedue gli Octree partono inizialmente vuoti, avendo come unico nodo la root. Visto la differenza nella tecnica e le ottimizzazioni del caso, l'implementazione di questo lavoro di tesi risulta essere più veloce di quella di Unity.

La Figura 4.1 mostra i risultati del benchmark effettuato tra le due implementazioni. Per 1000 inserimenti e 1000 rimozioni, l'implementazione effettuata per questa tesi impiega rispettivamente 4ms e 9ms. L'implementazione di Unity impiega rispettivamente 4ms e 30ms. (Figura 4.1)

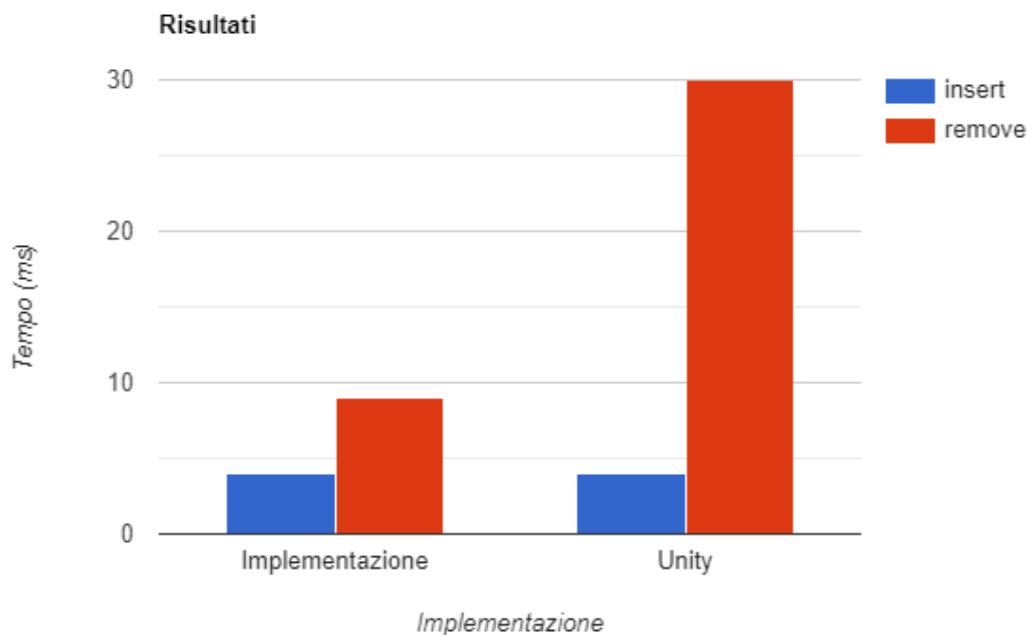


Figura 4.1: Grafico dei risultati su 1000 inserimenti e 1000 rimozioni.

Questo perché l'azione di shrinking risulta essere molto pesante. Per diminuire i boundary dell'Octree, controlla se la root ha un solo figlio popolato. In quel caso si prende cura di trasformare quell'unico figlio popolato in una nuova root e cancella il resto. Questo fa partire il Garbage Collector diverse volte, rallentando l'applicazione. Questa operazione ha il vantaggio di salvare molta memoria. Con un sistema del genere è possibile usare un unico Octree per tutto il mondo, andando però a perdere molto tempo in fase di accesso in quanto il numero di figli dell'Octree può diventare molto grande, specialmente nel caso si utilizzi per la generazione procedurale di un mondo potenzialmente infinito, facendogli perdere gran parte dei vantaggi dell'utilizzo di un Octree.

Capitolo 5

Conclusioni

Il progetto è costituito da circa 2700 righe di codice, suddivise in una trentina di classi. Il consumo medio di RAM del progetto mentre viene eseguito è di 250mb, mentre il tempo di inizializzazione è di circa 10 secondi, considerando Chunk di dimensione 16x16x16, e un loadingRadius di 3 Chunk. Le Region salvate su file, invece, hanno un peso che può arrivare fino a circa 12mb. Nonostante le tante righe di codice sono molto soddisfatto del lavoro svolto. Personalmente sento di aver raggiunto gli obbiettivi prefissati in quanto il progetto è un'ottima base per quello che voglio sviluppare in futuro. Sono riuscito a mantenere il progetto facilmente estendibile e scalabile grazie all'approfondito uso di design pattern e generics. Pure a livello di performance sono soddisfatto, vedendo che il progetto riesce ad andare a più di 30fps, quindi considerabile in realtime. A livello visivo il risultato è soddisfacente, con una distribuzione dei biomi particolarmente buona. Inoltre le ricerche fatte sono state interessanti da ogni punto di vista e apprezzo il fatto che continuando a lavorare su questo progetto posso approfondirle ulteriormente.

- È possibile utilizzare un Octree per gestire la generazione procedurale di un mondo?
- È possibile disaccoppiare totalmente la generazione di un mondo procedurale rispetto a Unity?
- È possibile trovare un modo per aumentare la precisione in cui gli oggetti vengono messi all'interno dei cubi?
- È possibile avere una gestione dei Chunk in tre dimensioni?

In conclusione è possibile l'utilizzo di un Octree per gestire la generazione procedurale di un mondo, anche avendo un fortissimo disaccoppiamento rispetto al motore grafico. Avere più oggetti all'interno di un cubo è di difficile risoluzione, ma la possibilità, grazie all'Octree, di avere cubi di dimensione variabile aiuta nel ridurre la percezione del problema. La gestione dei Chunk in tre dimensioni è di difficile risoluzione in quanto è un problema di complessità $O(N^3)$, quindi necessita di molte ottimizzazioni per poter eseguire in maniera veloce.

La memoria utilizzata dal programma, infatti, non è eccessiva e ciò permette di costruirci sopra un gioco senza essere troppo limitati lato performance. Per quanto riguarda invece la complessità totale, è $O(N^3)$, limitata sia dalla costruzione delle Mesh che dal ciclare su tre dimensioni per la gestione dei Chunk. Quest'ultima parte, in particolare, è il più grande bottleneck presente nell'applicazione, andando ad intaccare il framerate in caso di loadedRadius moderatamente grande. Per riuscire a ottenere un framerate accettabile (30fps) ho dovuto tenere il loadedRadius relativamente basso, andando così a intaccare pure il visibleRadius e non avendo un eccessiva render distance, e avendo comunque dei piccoli picchi di lag. Inoltre alcune funzioni sono state particolarmente ostiche da programmare dal punto di vista logico proprio perché si è lavorato in tre dimensioni, dovendo andare a definire per ogni asse e ogni direzione le operazioni da eseguire (vedi MeshBuilder) e ciò ha reso sia lo sviluppo che il debugging molto più complicato di quanto mi aspettassi.

5.1 Future work

Nonostante sia soddisfatto dei progressi nel progetto svolto e abbia raggiunto i miei obiettivi per questa tesi, ci sono ancora molte cose da fare per migliorare il mondo e aumentare ulteriormente il framerate.

5.1.1 Più dettagli

Come migliorie al mondo vorrei aggiungere fiumi, alberi, grotte e dettagli di questo genere. Ho evitato di farlo in questo progetto perché voglio portare particolare attenzione alla generazione durante questo passaggio, mettendoci uno sforzo paragonabile a quello di questa tesi. Anche i biomi vorrei fossero più variegati, utilizzando anche la coordinata Y nella loro scelta, dando così la possibilità di creare biomi sotterranei o avere biomi in cielo (ad esempio grotte particolari o un bioma con isole volanti).

5.1.2 Performance

Per quanto riguarda le performance, invece, ci sono ancora diversi punti dove si può ottimizzare. La scelta dei Chunk da caricare in base alla posizione del player, per esempio, è possibilmente parallelizzabile. Questo, però, aggiungerebbe altri thread al sistema, il che rischia di creare un grande overhead in caso di sovraccaricamento del computer. Una soluzione al problema sarebbe quello di avere una thread pool ben studiata. Anche questo lavoro, però, richiede parecchie ore di studio e valutazioni riguardanti ogni aspetto dello sviluppo.

5.1.3 MeshBuilder

Per tutto il periodo della tesi mi sono scontrato con un problema nella renderizzazione delle mesh. La natura dinamica della dimensione dei cubi, infatti, rende la renderizzazione un po' più complicata a livello di logica. Questo perché il controllo riguardante i cubi vicini è più difficile da fare. La soluzione più semplice sarebbe di accedere a ogni foglia dell'Octree tramite tre cicli for innestati uno dentro l'altro e controllando di volta in volta sezioni della dimensione minima di un cubo. Questo approccio, però, non mi piace particolarmente perché se si utilizzano tre cicli for innestati, converrebbe usare una Array al posto dell'Octree e perderei tutto il vantaggio dell'approccio utilizzato. Una soluzione più intelligente l'ho individuata solo di recente e non ho avuto il tempo di realizzarla. Il mio Octree necessita di una funzione che data una direzione ritorna tutti i cubi che giacciono sulla faccia richiesta. Purtroppo, però, è un procedimento che difficilmente si può generalizzare, e di conseguenza va fatto implementando singolarmente le procedure da fare per ogni singola direzione.

5.1.4 Noise

All'interno del mio progetto sarebbe possibile utilizzare anche altri tipi di noise, ma al momento c'è solo il Perlin Noise. Mi piacerebbe trovare un modo elegante per sfruttare lo Strategic pattern e la dependency injection per poter cambiare il noise da utilizzare. Sempre riguardo al noise utilizzabile, sarebbe molto utile creare un file di configurazione anche per i bin dei biom, in maniera tale da poterli modificare con tool esterni o, comunque, senza aver bisogno di aprire un IDE.

Bibliografia

- [1] Matt Barton e Bill Loguidice. «A history of gaming platforms: Atari 2600 Video Computer System/VCS». In: *Gamasutra: The Art & Business of Making Games*. Retrieved August (2008).
- [2] *C# dictionary is an hashtable.* <https://docs.microsoft.com/it-it/dotnet/api/system.collections.generic.dictionary-2?view=net-5.0>. Accessed: 09-06-2021.
- [3] James D Foley et al. «Spatial-partitioning representations; Surface detail». In: *Computer Graphics: Principles and Practice* (1990).
- [4] Chris Higgins. «No Man's Sky would take 5 billion years to explore». In: *Wired. com*. Aug 18 (2014), p. 2014.
- [5] Josef Kates. «Bertie the Brain». In: *Toronto: Canadian National Exhibition* (1950).
- [6] David Luebke et al. *Level of detail for 3D graphics*. Morgan Kaufmann, 2003.
- [7] Donald Meagher. «Geometric modeling using octree encoding». In: *Computer graphics and image processing* 19.2 (1982), pp. 129–147.
- [8] *Minecraft noise.* https://minecraft.fandom.com/wiki/Noise_generator. Accessed: 03-06-2021.
- [9] *Minecraft sales.* <https://www.statista.com/statistics/680124/minecraft-unit-sales-worldwide/>. Accessed: 03-06-2021.
- [10] *Minecraft world.* <https://minecraft.fandom.com/wiki/World>. Accessed: 03-06-2021.
- [11] Richard Moss. «Roam free: A history of open-world gaming». In: *Ars Technica* (2017).
- [12] K Perlin. «Noise hardware. In real-time shading'». In: *SIGGRAPH Course Notes* (2001).
- [13] Ken Perlin. «Improving noise». In: *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*. 2002, pp. 681–682.
- [14] Tyler Quiring. «From voxel vistas: Place-making in Minecraft». In: *Journal for virtual worlds research* 8.1 (2015).

- [15] Alexander Smith. *They Create Worlds: The Story of the People and Companies That Shaped the Video Game Industry, Vol. I: 1971-1982*. CRC Press, 2019, pp. 309–310.