ALMA MATER STUDIORUM–UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

ESTENSIONE DEL GAME ENGINE UNITY PER LA GENERAZIONE PROCEDURALE DI AMBIENTI PER VIDEOGIOCHI

Relatore:
Prof.
MORENO MARZOLLA

Presentata da:
STEFANO BETTINELLI

Sessione I

Anno Accademico 2014-2015

Indice

Introduzione

In questo elaborato viene descritto il lavoro svolto per sviluppare un generatore procedurale di ambienti per videogiochi, più propriamente definiti con il temine *dungeon*. Letteralmente la parola dungeon viene tradotta dall'inglese come prigione o prigione sotterranea, l'accezione che viene utilizzata però nel mondo dei videogiochi è spesso differente, con essi si intende infatti specificare aree di gioco formate da stanze interconnesse con contenuti di vario tipo dipendenti dal gioco in questione.

Questa tesi è strutturata in cinque capitoli: nel primo capitolo viene esposto lo stato dell'arte relativo alla generazione procedurale in cui saranno descritte alcune delle tecniche più utilizzate; nel secondo capitolo verrà mostrato lo strumento sviluppato, partendo da una descrizione generale fino ad analizzare in dettaglio il funzionamento dell'algoritmo di generazione; nel terzo capitolo viene esposta la struttura del codice sviluppato; nel quarto capitolo viene descritta l'interfaccia grafica utilizza per la generazione procedurale e infine nel quinto ed ultimo capitolo vengono descritti gli strumenti utilizzati per lo sviluppo (ambienti di lavoro, linguaggi, ecc.) esponendo inoltre le conclusioni ed i possibili sviluppi futuri del progetto.

Il generatore è stato sviluppato come estensione integrabile all'interno di Unity. Unity è un game engine, cioè uno strumento specifico per lo sviluppo di videogiochi. Permette di creare e

manipolare oggetti nello spazio tridimensionale e bidimensionale e consente di sviluppare la logica di gioco attraverso la scrittura di programmi in linguaggio C#, JavaScript oppure Boo. Uno degli aspetti più interessanti di Unity è la sua flessibilità nell'integrazione di strumenti di terze parti, attraverso un sistema d'importazione di *assests*, che è possibile definire come un insieme di risorse (quali codici, modelli poligonali, elementi audio ecc...) pronti per l'utilizzo che consentono di velocizzare in certi casi lo sviluppo di un video gioco. Ad esempio esistono degli assets integrabili all'interno di un progetto Unity e che forniscono un sistema di controllo già pronto del personaggio giocante, in questo modo gli sviluppatori possono concentrarsi su altri aspetti di gioco. Unity fornisce un framework di API piuttosto semplice e ben documentato e consente di estendere la sua interfaccia grafica con nuove funzionalità. Sfruttando questi due aspetti (l'importazione di assets e la buona documentazione) è stato sviluppato un generatore di dungeon procedurale integrabile all'interno di un qualsisi progetto Unity che attiva una nuova voce di menu con la quale è possibile aprire un pannello contenente una semplice interfaccia grafica per gestire il generatore procedurale.

Il generatore si basa su modello a griglia che mappa il contenuto del dungeon in una matrice di interi bidimensionale, che viene utilizzata sia come struttura dati di supporto e sia per ragioni di ottimizzazione dell'algoritmo di generazione. L'algoritmo è composto di tre fasi: la prima fase crea le stanze e le dispone nello spazio, la seconda fase crea un grafo astratto di connessione che collega le stanze e nella terza ed ultima fase crea i corridoi di connessione; al termine della generazione si ottiene un oggetto allocato nello spazio che funziona come contenitore di tutti gli elementi di cui è composto un dungeon (stanze, celle dei pavimenti, mura ecc.).

Il generatore è stato creato con l'obiettivo di essere uno strumento da utilizzare in fase di sviluppo di un gioco, infatti il flusso di lavoro ideale in cui si colloca è quello in cui viene utilizzato dai level designer (cioè delle persone esperte nella creazione di livelli di gioco) per generare in modo veloce livelli, sui quali poi è possibile intervenire manualmente per popolarli e modificarli come meglio essi credono.

L'intero progetto è stato sviluppato con licenza GPLv3 ed è possibile scaricarlo interamente dal seguente link:

`https://github.com/stefanobettinelli/UnityPDG`

Ritengo personalmente che uno degli aspetti più importanti che permette una crescita incisiva nel settore videoludico sia la facilità con cui gli sviluppatori hanno accesso agli strumenti di creazione di contenuti e in questo senso ho ritenuto che rilasciare lo strumento con una licenza open source fosse la scelta migliore, dandogli l'opportunità di essere migliorato anche da persone esterne.

Capitolo 1

In questo capitolo sarà fornita una definizione delle tecniche e delle metodologie che fanno parte del *Procedural Content Generation*, spesso abbreviato come PCG, spiegando in quale settore è collocato e il perché del suo utilizzo.

1.1 Generazione procedurale di contenuti

Il PCG (dall'inglese Procedural Content Generation) in letteratura viene spesso definito come *un metodo algoritmico di creazione di contenuti,* con *limitata interazione da parte degli utenti esterni* [2]. Detto più semplicemente con PCG ci si riferisce a software in grado di creare contenuti in modo indipendente oppure con un limitato supporto dell'intervento dell'utente umano.

Cosa s'intende però con contenuti? In realtà le tecniche di PCG sono state utilizzate in molti campi, come la musica e la scrittura e spesso sono state argomento di discussione in ambito scientifico, sono infatti molteplici le pubblicazioni di articoli a riguardo. Nel corso degli anni comunque si è vista una tendenza sempre maggiore dell'impiego di queste tecniche nell'industria dei videogiochi. Il PCG è, in effetti, nato e si è sviluppato grazie al bisogno di

automatizzare la creazione di contenuti all'interno dei video giochi, il primo videogioco che ne ha fatto uso è stato Rogue nel 1980 [9].
Con il PCG è possibile generare livelli, mappe, regole di gioco, textures, storie, oggetti, missioni, armi, veicoli, personaggi ecc.
La generazione procedurale di mappe e/o livelli può essere, in prima istanza, categorizzata secondo due tipi di generatori: quelli che producono ambienti esterni come ad esempio paesaggi e ambienti urbani e quelli che generano ambienti chiusi come ad esempio caverne e sistemi di stanze tra loro collegate (noti anche come dungeons).
Le tecniche di generazione procedurale sono strettamente legate al tipo di risultato che si vuole ottenere, cioè al gioco per cui i contenuti saranno destinati. Quindi bisogna tener presente dei limiti imposti del gioco, al fine di creare del materiale che sia effettivamente fruibile ed evitare di creare contenuti del tutto casuali che produrrebbero un'esperienza video ludica di bassa qualità.
Per chiarire meglio il concetto di generazione procedurale, può essere utile stilare una breve lista di cosa può essere considerato PCG e cosa invece non è PCG. Nel primo elenco vediamo ciò che può essere considerato un *procedural content generator*:

- Uno strumento software in grado di creare livelli labirintici con una limitata interazione del level designer.
- Un sistema che crea in modo dinamico, con metodi evolutivi, elementi di gioco quali ad esempio armi.
- Un engine che sia in grado di creare in modo automatico della vegetazione (SpeedTree [10]).
- Un gioco che automaticamente bilancia la sua difficoltà in base al comportamento dei giocatori, proponendo un livello di sfida adeguato per gli utenti.

Vediamo invece cosa non è possibile considerare come un sistema che utilizza tecniche di PCG

- Un editor di mappe che necessita l'input di un utente per la loro creazione.

- Tecniche di intelligenza artificiale applicate a personaggi non giocanti: esse non creano nuovi contenuti ma sfruttano metodi di IA per dar vita ad oggetti e/o personaggi già presenti nel mondo di gioco
- Un generatore totalmente casuale di contenuti che non tiene conto dei vincoli del gioco a cui è destinato.

L'impiego delle tecniche PCG nel mondo dei videogiochi ha avuto successo solo su pochi titoli inizialmente, tra cui ricordiamo Diablo (Blizzard 1996), che è in grado di generare nuovi dungeon ad ogni sessione di gioco, Civilization (MicroProse 1991) in cui il PCG è utilizzato per la generazione delle mappe, oppure Borderland dove queste tecniche si applicano per generare le armi.
Da un punto di vista accademico il PCG, pur essendo stato studiato ragionevolmente a fondo, non trova una convergenza delle tecniche verso un modello riconosciuto come standard, ed inoltre il materiale di riferimento è spesso frammentato e ripetuto. Queste mancanze sono principalmente dovute al fatto che le tecniche di PCG non possono essere strettamente correlate tra loro essendo intrinsecamente legate al prodotto finale che si vuole ottenere.
Spesso si fa anche molta confusione nel differenziare il contenuto prodotto con tecniche di PCG, dal contenuto prodotto attraverso l'intervento umano. Ad esempio: in giochi come Sim City o Minecraft, che sfruttano una componente basata su PCG per generare il mondo iniziale, si permette ai giocatori di modellare a loro piacimento il contenuto generato, ma questo prodotto modificato non ha alcuna correlazione con il contenuto procedurale inizialmente generato in modo algoritmico.
Un altro concetto che viene spesso nominato in letteratura o a cui si fa riferimento erroneamente nel web è il concetto di *casualità* nei confronti della generazione procedurale. Si pensa che i generatori procedurali altro non siano che aggregatori di asset (elementi di gioco) che vengono dispiegati in modo casuale in un certo spazio. Questo non è vero: tutti i giochi che producono contenuti in modo procedurale lo fanno in modo che tali contenuti siano fruibili, cioè sfruttano le proprietà del gioco e basandosi sui limiti imposti dal game play, cioè su come effettivamente sarà giocato il gioco. Comunque è possibile dare un'interpretazione del fattore della casualità in questo contesto come elemento stocastico sul tipo dei contenuti prodotti, che variano in modo pseudo-casuale pur restando all'interno dei *confini* di gioco. Il fattore

stocastico a differenza di quello totalmente casuale permette agli algoritmi di sfruttare i parametri forniti in input: si può ad esempio pensare di avere un valore di input minimo e massimo del danno di un'arma e all'interno del quale è possibile generare valori casuali secondo una determinata distribuzione. L'aspetto stocastico dei generatori è fondamentale anche per un altro motivo: permette di avere contenuti sempre diversi che offrono teoricamente un valore di rigiocabilità infinito.

Il PCG, recentemente, sta riscuotendo sempre più apprezzamento, grazie ai numerosi vantaggi che apporta, come ad esempio la velocità di creazione dei contenuti e la riduzione notevole dei costi di sviluppo. Il team dei level designer può essere infatti ridotto proprio grazie a questo tipo di strumenti automatici che possono essere utilizzati ripetutamente finché non producono risultati soddisfacenti. Come effetto collaterale quindi il PCG ha favorito la formazione di team di sviluppo numericamente sempre più ridotti e la maggior parte di essi è composta da sviluppatori indipendenti che attutiscono i costi delle fasi di level design proprio sfruttando queste tecniche.

Il PCG rimane comunque agli occhi di molti una tecnica di costruzione dei contenuti un po' oscura proprio per la sua natura automatizzata e non direttamente controllabile con l'intervento umano.

I contenuti prodotto con software PCG possono in qualche modo essere valutati, ma questo non è compito semplice. Valutare la qualità di un dungeon spesso si riduce a fattori soggettivi, purché sia chiaramente fruibile, altrimenti non ha alcun senso parlare di qualità. In letteratura si possono trovare alcuni tentativi [13] [14] ideati per adottare metriche di giudizio della qualità dei prodotti dei generatori PCG. In genere però quando si sviluppa un generatore procedurale bisogna, come sempre, tenere conto del tipo di gioco a cui è destinato e su tali presupposti trovare delle metriche adeguate di valutazione.

Prima di terminare questa sezione è doveroso specificare che la scelta di Unity come engine in cui integrare il generatore è stata dettata soprattutto dalla sua popolarità e della reperibilità della sua documentazione. I concetti su cui si basa l'intero progetto possono essere tranquillamente adattati per qualunque altro game engine.

1.2 Generazione procedurale di dungeons

Tipicamente un dungeon consiste di una serie di stanze interconnesse, che definisco la struttura topologica, inoltre un dungeon è anche caratterizzato dal suo contenuto: al suo interno è possibile trovare mostri da affrontare e tesori da raccogliere ecc.
Quando si utilizza un metodo di PCG per generare dei dungeon possiamo più propriamente parlare di *procedural dungeon generation* PDG, ma per semplicità usare termine PCG durante tutta la trattazione. Un generatore di dungeon procedurale si occupa principalmente di creare la topologia di un dungeon, ma spesso anche del contenuto interno ed è caratterizzato da tre elementi:

1. Dal modello di rappresentazione astratta: cioè dal metodo utilizzato per descrivere le caratteristiche di un dungeon.
2. Dal metodo per costruire/implementare il modello del punto 1.
3. Infine dal metodo per costruire in modo concreto la geometria codificata nel modello astratto.

Questi tre punti guideranno la spiegazione delle varie fasi attraversate dall'algoritmo di generazione che è stato implementato.
Esistono numerose tecniche di generazione di dungeon che possono essere classificate in due macro categorie: quelle basate su metodi costruttivi che forniscono una singola istanza di output oppure quelle basate su metodi di ricerca che tendono a iterare sugli output prodotti al fine di raggiungere un certo livello di threashold di qualche parametro desiderato [15]. I metodi costruttivi hanno il vantaggio di essere molto più efficienti da un punto di vista temporale e quindi sono più adatti alla generazione online, mentre quelli evolutivi tendono a produrre i migliori risultati dopo un certo numero di iterazioni.
Spesso in letteratura vengono utilizzati i due termini *online* e *offline*: il termine online viene utilizzato per i generatori in grado di funzionare in fase di runtime di un videogioco, mentre, in in contrapposizione, il temine offline riguarda i metodi di generazione utilizzati in fase di sviluppo.

Un'importante proprietà che accomuna tutti i generatori è il livello di controllo con cui è possibile, pilotare i risultati prodotti. Un buon livello di controllo assicura che i livelli prodotti siano coerenti con gli input immessi pur mantenendo la caratteristica più importante: la diversità degli output ad ogni generazione.
È anche possibile combinare diversi metodi e tecniche di generazione, ottenendo sistemi sempre più complessi e sofisticati in grado di essere comandati in modo sempre più preciso dai parametri di input al fine di ottenere specifici tipi di output. In questo senso quindi il PCG si sta evolvendo come uno strumento ad alto livello a supporto per il design dei livelli, intesi sia come ambienti esterni che interni [4].
Nelle sezioni successive sarà fornita una breve descrizione di alcuni dei più noti metodi costruttivi di generazione, tra cui vi sono: il metodo basato sul *partizionamento dello spazio*, il metodo basato ad *agente*, che costruisce passo dopo passo la geometrica del dungeon, il metodo basato su *automi cellulari* ed infine il metodo basato su grammatiche generative per la definizione di aspetti di alto livello di design inerenti al sistema di gioco che si vuole creare.

1.2.1 Metodo basato sul partizionamento dello spazio

Con questa tecnica si parte da una determinata porzione di spazio (2D o 3D) che viene ricorsivamente partizionata. Il metodo di partizionamento più utilizzato è quello binario che consiste nel dividere in due parti ricorsivamente le porzioni che si ottengono.
Le sezioni ottenute possono essere organizzate secondo un modello astratto di albero binario, in cui le foglie rappresentano una porzione di spazio adibita al contenimento di una singola stanza, il collegamento tra le stanze avviene grazie alle connessioni nell'albero binario: le foglie di uno stesso genitore sono collegate direttamente formando un nodo di livello superiore, quest'ultimo viene connesso con un nodo fratello attraverso il nodo padre, cioè la radice di un sottoalbero; il processo di collegamento continua fino a quando non vengono connessi tra loro i due figli della radice.
Questo metodo presenta l'importante vantaggio di garantire la non sovrapposizione delle stanze e per contro vincola la creazione dei dungeon su un dato spazio di partenza prefissato, quindi da questo punto di vista fornisce poco controllo ai level designer.

In figura 1 è possibile vedere un processo di divisione dello spazio e di connessione delle stanze: nella colonna di sinistra si parte da un'area rettangolare 'A' che viene divisa in due e a loro volta in modo ricorsivo le aree ottenute vengono divise, il processo di partizionamento continua fino a che non si ottiene il numero sezioni desiderato. Quindi si passa alla seconda fase raffigurata nella colonna di destra, dove si creano le stanze e le varie connessioni.

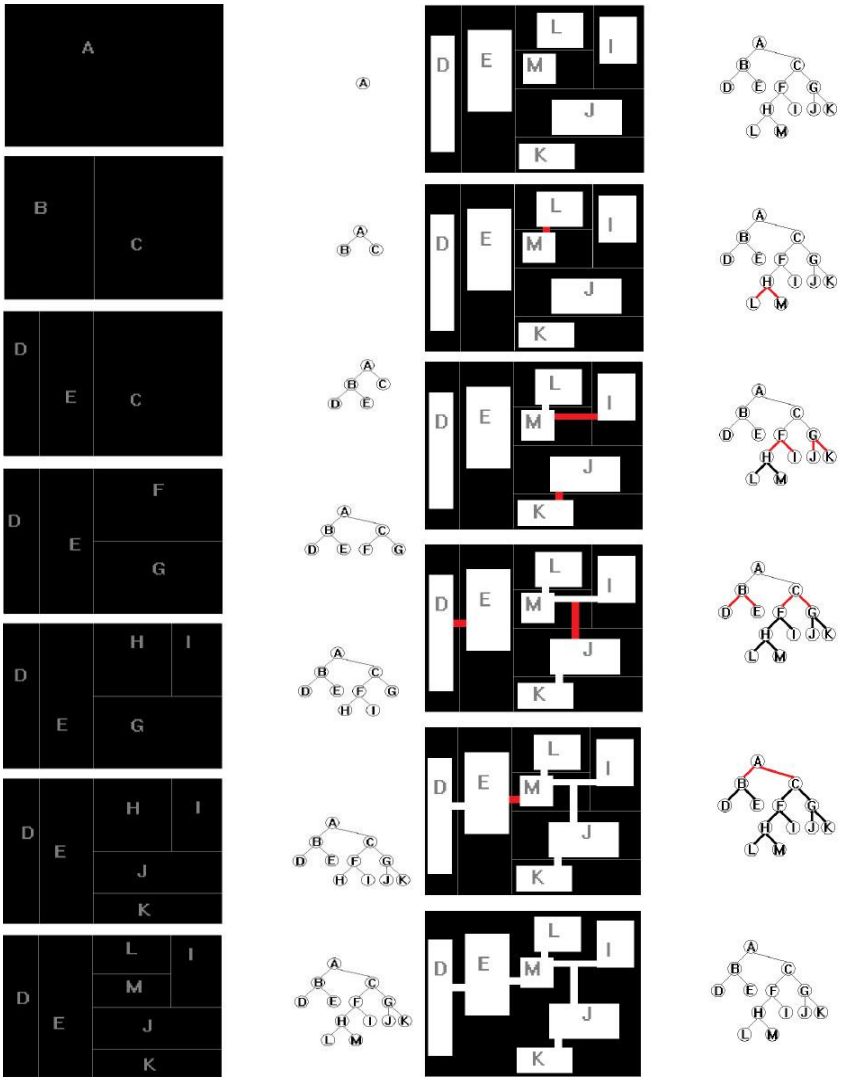

Figura 1. Nella colonna di sinistra è mostrato il processo di divisione dello spazio mentre in quella a destra si può osservare il processo di creazione delle stanze e di connessione (fonte immagine [1]).

1.2.2 Metodi basati su agenti

Generalmente con le tecniche procedurali di questo tipo si utilizza un singolo agente incaricato di creare tutta la struttura del dungeon, i risultati che si ottengono con questo tipo di approccio sono meno schematici rispetto a quelli che si ottengono con il partizionamento dello spazio, ma in realtà questo dipende da come viene implementato il comportamento dell'agente: un agente con un comportamento fortemente casuale produrrà dungeon più caotici contrariamente ad un agente che agisce con schemi più controllati. Non vi è inoltre alcuna garanzia che con questo metodo non si possa incorrere in sovrapposizioni di stanze. Ad esempio un agente, posizionato inizialmente in un punto casuale dello spazio, può, nel suo ciclo esecutivo, operare nel modo seguente: provare a costruire una stanza, controllare se la stanza creata provoca eventuali sovrapposizioni, se sì, allora non costruisce la stanza ma si sposta nello spazio creando corridoi fino a che non decide di riprovare a costruire di nuovo la stanza. Il ciclo continua fino a che non sono state create tutte le stanze volute, oppure, in caso di fallimento, se l'agente raggiunge una posizione in cui non si possono più costruire né stanze e né corridoi evitando sovrapposizioni.

Il vantaggio di questo tipo di approccio è la minore rigidità nei confronti delle dimensioni delle stanze, diversamente dal metodo di partizionamento dello spazio, come svantaggio si ha, per la natura stessa del metodo, che non è possibile stabilire a priori se la generazione fallisca o meno.

Il fattore principale, che determina se questo tipo approccio riesca a generare buoni risultati, è il grado di controllo che si ha del modo con cui opera l'agente.

Figura 2. In questa figura è possibile vedere l'agente in azione: parte da un punto casuale della mappa e crea la prima stanza (nella terza immagine), poi costruisce un corridoio diretto prima verso est e poi verso sud (quarta immagine). Continua a operare fino alla nona figura in cui si trova incastrato dopo aver costruito una stanza avendo raggiunto il confine, oltre il quale non gli è concesso costruire e ogni sua possibile scelta di costruzione produrrebbe una sovrapposizione (fonte [1]).

1.2.3 Metodi basati su automi cellulari

Nell'articolo del 2010 da parte di Johnson [5], viene descritto un metodo basato sugli automi cellulari per la generazione di dungeon ambientati all'interno di strutture cavernicole. L'obiettivo di Johnson era di creare livelli infiniti per un gioco di esplorazione di strutture scavate nella roccia. La topologia, essendo infinita, non poteva essere contenuta in nessun supporto di memorizzazione e quindi i dungeon venivano generati a tempo di esecuzione.
Il metodo si è basato sull'utilizzo di alcuni parametri di input:

- Una percentuale di rocce, cioè di zone inaccessibili.
- Un determinato numero di iterazioni dell'automa cellulare.
- Un valore di threshold di celle adiacent per la definizione di una roccia pari a 5.

Ogni dungeon generato è una griglia di 50x50 celle, in cui ogni singola cella può avere due stati: vuoto o roccia.

Inizialmente la griglia è vuota e il processo di generazione funziona nel modo seguente:

- La griglia viene riempita di rocce seguendo una distribuzione uniforme di probabilità.
- Vengono eseguiti un certo numero di passi dell'automa cellulare e viene applicata una singola regola di trasformazione di stato delle celle: una cella viene trasformata in roccia se e solo se almeno 5 celle vicine sono rocce altrimenti viene trasformata in uno spazio vuoto.

L'aspetto positivo di questo metodo è adatto a funzionare in modalità online, lo svantaggio invece è che non si riesce a far combaciare gli ingressi e le uscite delle stanze generate. Quindi, per ovviare a questo problema, la generazione delle stanze viene fatta in modo sequenziale: dopo aver generato una stanza vengono generate le stanze adiacenti e se i corridoi tra le stanze non si connettono correttamente, vengono creati in modo forzato fino a che non si stabilisce la giusta connessione.

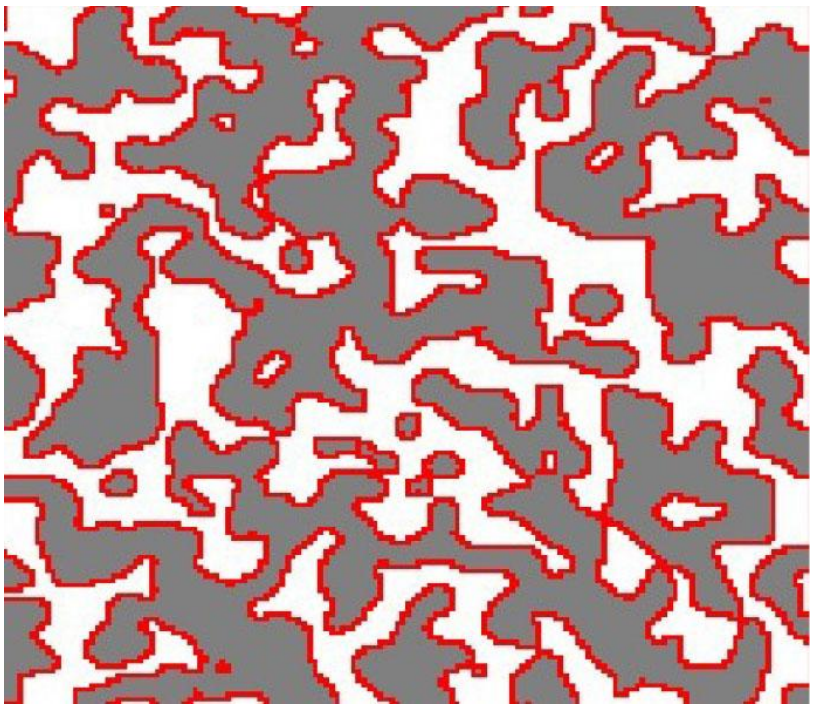

Figura 3. Un possibile risultato del metodo di generazione basato su automi cellulari. Le aree percorribili sono quelle grigie (fonte [1]).

L'aspetto negativo di questo metodo è dovuto alla poca controllabilità attraverso i parametri di input, ogni singolo parametro ha impatto su un numero troppo elevato di fattori della generazione.

1.2.4 Metodi basati su grammatiche su grafi

Inizialmente le grammatiche generazionali erano state ideate per descrivere in modo formale un insieme di frasi linguistiche, attraverso regole ricorsive di trasformazione con singole parole, usate come termini di un linguaggio. Da queste si sono poi sviluppate le grammatiche su grafi in cui ogni nodo corrisponde a un simbolo terminale.
Queste tecniche sono state in prima battuta utilizzate nel lavoro di Adams [6] per produrre dungeon destinati a giochi con visuale in prima persona, al lavoro di Adams hanno avuto seguito alcuni progetti sempre basati su grammatiche su grafi, tra cui quello di Dormans [7] e soprattutto quello di van der Linden [8] in cui viene mostrato come sia possibile mettere a disposizione dei designers uno strumento di alto livello che consente di definire la struttura di

un dungeon andando in prima istanza a definire elementi del game play che si intende produrre. La figura 4 mostra come un dungeon, definito come grafo connesso di azioni che il giocatore può intraprendere, può essere tradotto in forma concreta dal grafo iniziale; producendo una serie di stanze interconnesse in cui in ognuna sarà possibile compiere le azioni definite dai nodi del grafo.

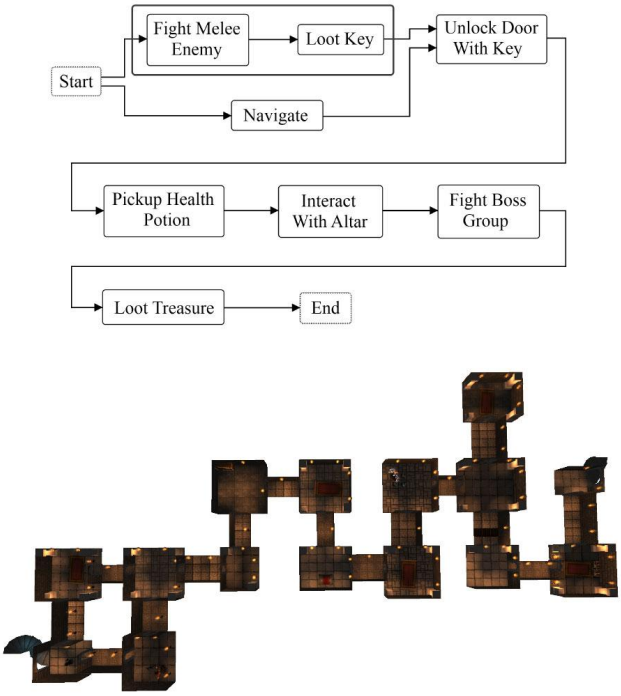

Figura 4. Dalla figura vediamo due fasi della creazione: nella prima, quella sopra, è rappresentata la grammatica di gioco mappata su un grafo, nell'immagine sottostante vi è il dungeon traddotto dal grafo e realizzato in 3D (fonte [1]).

1.2.5 Metodi per valutare i generatori procedurali

Dato un metodo per la generazione procedurale può essere utile valutarne la qualità in qualche modo. Esistono molte ragioni per cui si può essere interessante valutare un generatore procedurale:

1. Ad esempio si vuole poter valutare le prestazioni dell'algoritmo di generazione oppure valutare sotto determinati criteri gli output che produce.

2. Per certificare che alcune qualità di generazione dichiarate vengano effettivamente prodotte.
3. Per condurre una comparazione con altri generatori. La comunità di sviluppatori di generatori è sempre più in espansione ed è quindi utile capire quali sono i progressi che si ottengono in relazione ad altri progetti.

Il concetto più importante da tenere in mente quando si cerca di valutare un generatore è il seguente: bisogna assicurarsi che il metodo di valutazione che si utilizza ponga in rilievo l'aspetto peculiare del generatore per il quale si vuole condurre gli esperimenti.
Spesso si sente parlare anche del grado di espressività del generatore [16] facendo riferimento allo spazio di potenziali dungeon (o più in generale di ambienti) che il generatore è in grado di produrre, osservando i prodotti in modo da capire se esiste una tendenza particolare emergente dalle simulazioni. Questo tipo di esperimenti viene effettuato decidendo per prima cosa le metriche di valutazione e la loro disposizione sugli assi e solo successivamente ha senso produrre un insieme di generazioni campione da valutare. Di solito, è buona norma scegliere metriche che non abbiano correlazione con i parametri di input, in modo da capire se esistono tendenze nuove che non si riesce a stabile a priori durante lo sviluppo; in caso contrario, scegliendo invece delle metriche strettamente correlate con gli input, si ottengono solo risultati confermativi e cioè ovvi.

Capitolo 2

Dopo aver passato in rassegna alcuni metodi di PCG nel primo capitolo, spieghiamo ora in dettaglio la struttura del generatore procedurale di dungeon sviluppato. Saranno utilizzate alcune figure per esporre meglio i concetti ed inoltre saranno riportati dei pezzi di pseudo codice con un formalismo semplificato simile al linguaggio Python, quindi senza parentesi e senza punti e virgola, in modo da semplificarne la lettura.
Le sezioni di cui è composto questo capitolo sono le seguenti:

- La sezione riguardante il funzionamento generale del generatore.
- La sezione in cui si illustra come il generatore è stato integrato in Unity.
- La sezione in cui viene spiegato l'algoritmo di generazione: questa sarà la parte più corposa in cui vengono esposte le varie fasi dell'algoritmo ed analizzate nello specifico, sia dal punto di vista delle scelte implementative che da un punto di vista del costo computazionale.

L'obiettivo dello strumento sviluppato è di fornire supporto per la generazione procedurale di dungeon in modalità offline, integrando il generatore nel flusso di lavoro dei level disigners. I risultati di output ottenuti diventano la base strutturale su cui poter creare un dungeon, popolandoli in una fase successiva con gli elementi tipici di cui può essere composto.

Prima di entrare nei dettagli più specifici delle scelte architetturali e implementative, trovo sia importante specificare in quale tipo contesto questo software viene collocato e il perché di alcune decisioni.

Come accennato nell'introduzione il generatore sviluppato, si colloca all'interno del game engine Unity, la scelta di Unity come framework di base è dettata dal fatto che è l'engine più utilizzato dagli sviluppatori indipendenti, nonché sempre più apprezzato ed utilizzato anche dall'industria videoludica commerciale. L'idea è quella fornire un primo passo verso un'integrazione sempre più costante di strumenti di questo tipo all'interno dei game engine in modo da favorire lo sviluppo di contenuti generati in modo procedurale, ottenendo tutti i benefici del caso, quali ad esempio: l'accorciamento dei tempi di sviluppo e la creazione di contenuti che non sono mai ripetitivi e che idealmente aumentano la longevità di un gioco in modo infinito.

Il generatore sviluppato non si colloca in modo preciso in nessuno dei metodi visti nel primo capitolo, ma cerca comunque di risolvere alcuni degli aspetti critici che si presentano ogni volta che si vuole produrre in modo automatico la struttura di base di un dungeon. In particolare, adotta un algoritmo che permette di disporre le stanze in modo che non occorrano mai sovrapposizioni tra stanze e fornisce un certo grado di controllo per quanto riguarda lo spazio di separazione tra stanze, per mezzo di un parametro di input detto *minShiftValue*. Inoltre le connessioni tra le stanze vengono realizzate costruendo un grafo planare detto *Relative Neighbourhood Graph* (RNG) [11] che contiene come sottoalbero il minimo albero di copertura euclideo e inoltre fornisce un buon grado di connessione dei nodi, mantenendo il grafo connesso. In figura 5 vediamo un esempio di questo tipo di grafo.

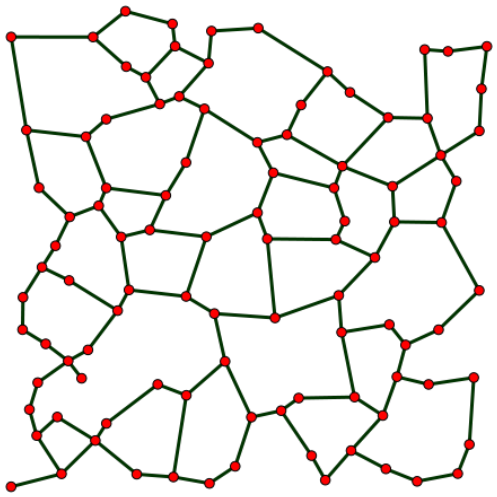

Figura 5. Esempio di grafo RNG (fonte Wikipedia).

2.1 Funzionamento del generatore

Dal punto di vista dell'utente finale, il generatore si presenta con una semplice interfaccia grafica disposta all'interno di Unity in cui è possibile inserire i dati di generazioni più importanti come ad esempio la dimensione delle stanze e il loro numero. Vi sono inoltre anche due pulsanti, uno per la creazione del dungeon e uno per la sua distruzione. Quando viene premuto il pulsante di creazione viene avviata la generazione e quindi attivato l'algoritmo a tre fasi, una terminato nel pannello della scena è possibile vedere il risultato della produzione.

2.2 Integrazione del generatore in Unity

Prima di vedere come il progetto è stato integrato in Unity è importante accennare al modo generale con cui si struttura lo sviluppo di un gioco, in quanto è la base su cui si fonda il generatore sviluppato.
Unity adotta un'architettura orientata ai componenti [12], dove ogni elemento di gioco (detto anche game object) è composto da componenti che ne definiscono il comportamento e le proprietà. Alcuni componenti di base vengono sempre inclusi nei game object che si creano, mentre altri vengono aggiunti dallo sviluppatore a secondo delle necessità. I tipi di componenti esistenti sono molteplici, si va dai componenti che definiscono proprietà fisiche a componenti per la gestione delle animazioni ecc. L'aspetto più interessante però è che i componenti

possono essere creati da zero: implementandoli come classi in uno dei tre linguaggi supportati. Così facendo è possibile fornire ai singoli game object dei comportamenti specifici, portandoli in certo senso in vita. Il discorso invece è diverso per quanto riguarda lo sviluppo di codice destinato a creare un'estensione dell'interfaccia di Unity, in questo caso non si utilizzano i game object i e componenti, ma è sufficiente che il codice sviluppato venga inserito in una cartella specifica detta *Editor* e che si utilizzino delle librerie specifiche e dei decoratori per i metodi implementati. Quindi sfruttando questa flessibilità, messa a disposizione dalle API, è stata costruita la piccola interfaccia grafica, attivabile attraverso il menu Windows ➔ Dungeon Generator Editor, che una volta aperta sarà possibile disporre all'interno dell'area di lavoro come un qualsiasi altro pannello.

Il codice sviluppato è diviso in due cartelle principali, la cartella *Scripts* designata a contenere i file sorgenti necessari per la generazione procedurale e la cartella *Editor*, in cui saranno contenuti i sorgenti necessari per estendere l'interfaccia di Unity.

Vediamo finalmente come l'algoritmo di generazione è stato integrato all'interno di Unity, in modo da poter funzionare secondo la filosofia di game object e componenti su cui si basa l'engine. Il punto di connessione tra il *DungeonEditor* (l'interfaccia) e l'algoritmo di generazione è rappresentato dal *prefab DungeonGenerator*. I prefab sono semplici game object, cioè entità disposte nello spazio, ma che a differenza dei semplici game object, che esistono solo in fase di runtime, sono salvati in memoria secondaria e quindi sono riutilizzabili in qualsiasi momento, anche per progetti diversi. Il ruolo reale del prefab DungeonGenerator però non è quello di poter essere istanziato, esso in realtà svolge la funzione di contenitore di componenti che nel complesso implementano l'algoritmo di generazione.

Oltre al *DungeonGenerator*, gli altri prefab utilizzati dall'algoritmo sono i seguenti:

- *DungeonCell*: contiene come componente la classe C# DundegonCell.cs e definisce semplici proprietà spaziali. Viene utilizzato come oggetto tridimensionale allocabile nello spazio ed ha una forma quadrata di dimensione unitaria. Questo prefab rappresenta in sostanza la singola mattonella del pavimento delle stanze e dei corridoi che andranno a formare i dungeon.
- *WallUnit*: è molto simile al DungeonCell e al suo interno contiene il componente C# WallUnit.cs che definisce le sue proprietà spaziali ed è orientato in ortogonale rispetto

al piano orizzontale. Rappresenta una singola unità di muro delle stanze e dei corriodi dei dungeon.

- *DungeonRoom*: questo prefab rappresenta una singola stanza nello spazio 3D e contiene il componente DungeonRoom.cs in cui sono definite le sue proprietà e i metodi per la manipolazione spaziale di una singola stanza, inoltre contiene il riferimento anche ai due prefab DungeonCell e WallUnit.
- *Dungeon*: infine quest'ultimo prefab contiene il componente Dungeon.cs in cui sono presenti i riferimenti agli oggetti DungeonRoom, DungeonCell e WallUnit ed ha la funzione di contenitore dell'intero dungeon.

La struttura nel complesso è quella di oggetti che incapsulano altri oggetti più piccoli che globalmente vengono tutti contenuti all'interno del prefab Dungeon.
Riassumendo, al fine di implementare il generatore come estensione del game engine sono state utilizzate le API fornite direttamente dal framework di Unity per realizzare la GUI, mentre per implementare l'algoritmo di generazione, sono stati utilizzati i concetti base su cui si fonda Unity: l'architettura basata sui componenti e l'utilizzo dei prefab per contenere le classi implementate e gli elementi di base, quali piastrelle e pareti unitarie, che compongono nel complesso i dungeon.

2.3 Algoritmo di generazione

L'aspetto più importante del progetto è l'algoritmo di generazione composto da tre fasi principali:

1. Nella prima fase l'algoritmo si occupa di disporre nello spazio 3D le *n* stanze che si vogliono creare.
2. Nella seconda fase viene creato il grafo di connessione di tipo RNG in cui i nodi sono identificati dai centri delle stanze e gli archi rappresentano le connessioni tra le stanze che si intende stabilire.
3. Nella terza ed ultima fase l'algoritmo utilizza le connessioni create nel punto 2 per formare i corridoi di collegamento tra le stanze.

Nelle sezioni successive saranno descritte in dettaglio queste fasi, motivando le scelte implementative e fornendo i loro costi computazionali al caso pessimo.
Ogni fase dell'algoritmo si basa sui tre concetti fondamentali con cui si modella un metodo procedurale (introdotti nel primo capitolo):

1. Il modello di rappresentazione astratta.
2. Il metodo di costruzione del modello astratto.
3. Il metodo di costruzione della geometria del modello astratto.

La differenza tra il secondo e il terzo punto consiste nel fatto che nel secondo punto si implementa il codice necessario per la mappatura del modello astratto su delle strutture dati, mentre con il terzo punto si utilizzano queste strutture dati per costruire sullo spazio 3D la geometria di un dungeon.
Vediamo ora come questi tre concetti corrispondono alla realizzazione delle tre fasi di cui consta l'algoritmo. Saranno presentate in ordine la prima fase di creazione e disposizione delle stanze, poi la seconda fase di connessione attraverso un grafo RNG e infine l'ultima fase di realizzazione dei corridoi.

2.4 Creazione e disposizione delle stanze nello spazio

Questa fase si occupa di creare e di disporre le stanze all'interno dello spazio tridimensionale in modo che alla fine di questo processo non esistano sovrapposizioni. Tra i parametri di input vi sono le dimensioni delle stanze quali: la massima altezza e larghezza, la minima altezza e larghezza; all'interno di questi intervalli, per ognuna delle stanze, vengono scelte in modo casuale i valori. Oltre a questi dati sulle dimensioni, l'algoritmo riceve anche un altro parametro fondamentale, di cui si è brevemente parlato in precedenza, questo valore detto *minShiftValue* è un valore intero e viene utilizzato dall'algoritmo (come tra poco sarà mostrato) per spostare le stanze nel momento in cui si verificano delle sovrapposizioni.
L'idea generale del funzionamento dell'algoritmo è la seguente: le stanze vengono processate sequenzialmente dalla procedura. Ognuna di esse viene inizialmente posizionata nell'origine

dello spazio, cioè nel punto di coordinate *<x=0,y=0,z=0>* (in Unity le ordinate sono identificate dalla lettera *z* mentre invece la terza dimensione è la *y*). L'algoritmo poi cerca di collocare ogni stanza, effettuando degli spostamenti dal punto iniziale verso l'alto oppure verso destra; la scelta della direzione avviene in modo casuale. Una volta effettuato lo spostamento, l'algoritmo verifica che non si siano formate delle sovrapposizioni, in caso contrario continua a spostare la stanza processata verso l'alto o verso destra.
Il motivo per cui le stanze sono inizialmente poste tutte nell'origine e poi da lì spostate è dovuto al fatto che in questo modo si riesce ad evitare che gli spazi che si creano tra le stanze dopo gli spostamenti rimangano vuoti. Tali spazi, infatti, potrebbero potenzialmente essere occupati dalle altre stanze, ma questo dipende principalmente anche dal valore minimo di shift che viene utilizzato: se è molto elevato gli spazi vuoti saranno comunque presenti e rimarranno inutilizzati, mentre più è basso (tendente ad 1) e più è probabile che si ottengano come risultato dei dungeon compatti con pochi spazi vuoti tra una stanza e l'altra. Il valore *minShiftValue* è, in effetti, un parametro di input molto importante che i level designer possono sfruttare, insieme agli altri dati riguardanti le dimensioni delle stanze, per regolare la densità di un dungeon, in termini di numero di stanze su una determinata area.

2.4.1 Modello di rappresentazione astratta

Per implementare l'algoritmo è stata utilizzata una matrice a due dimensioni, detta *tileMatrix*, in cui si mappano i quadranti unitari di Unity in tre possibili valori 0, 1, 2 con il seguente significato:

- Con 0 si indica che l'unità di spazio è vuota.
- Con 1 si indica che l'unità di spazio è occupata da una porzione unitaria del pavimento di una stanza.
- Con 2 si indica che l'unità di spazio è occupata da una porzione unitaria di un corridoio di connessione.

Anche se lo spazio in cui sarà creato il dungeon è tridimensionale è sufficiente una matrice a due dimensioni per il solo scopo di tenere traccia di cosa è stato creato e posizionato nello

spazio. Per cui la mappatura tra le coordinate vere (a tre componenti) viene tradotta a due componenti utilizzando la prima (l'asse x) e la terza (l'asse z) per indicizzare la matrice.
La matrice quindi permette, come sarà mostrato a breve, all'algoritmo di verificare se la posizione scelta per piazzare la stanza è già occupata da un'altra stanza, i corridoi in questa fase non sono ancora presenti, essi verranno inseriti nell'ultima fase seguendo le connessioni del grafo RNG.
Un altro possibile metodo per rilevare le sovrapposizioni sarebbe potuto essere attraverso l'utilizzo del sistema di collisione dei game object, che è già implementato da Unity stesso. L'utilizzo di questo sistema però avrebbe attivato il motore della fisica comportando quindi uno sforzo computazionale inutile, inoltre il motore fisico viene attivato solo in fase di run-time mentre lo strumento che è stato realizzato viene utilizzato per generare dungeon non a run-time, bensì a tempo di editing, cioè nel momento in cui si sta realizzando un dungeon e non mentre lo si sta effettivamente giocando. Quindi la scelta di usare una matrice di mappatura è stata più efficiente per lo scopo prefissato, poiché comporta la semplice manipolazione di numeri interi con un costo computazione al caso pessimo quadratico nelle dimensioni della matrice, come poi sarà mostrato.
Vediamo ora un esempio: la matrice seguente, contenente 3 stanze viene mappata nel dunegon di figura 6.

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1
0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1
0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1
1,1,1,1,1,1,1,1,2,2,1,1,1,1,1,1,2,2,2,2,2,2,2,2,0,0,0
1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0
1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0
1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0
1,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0
```

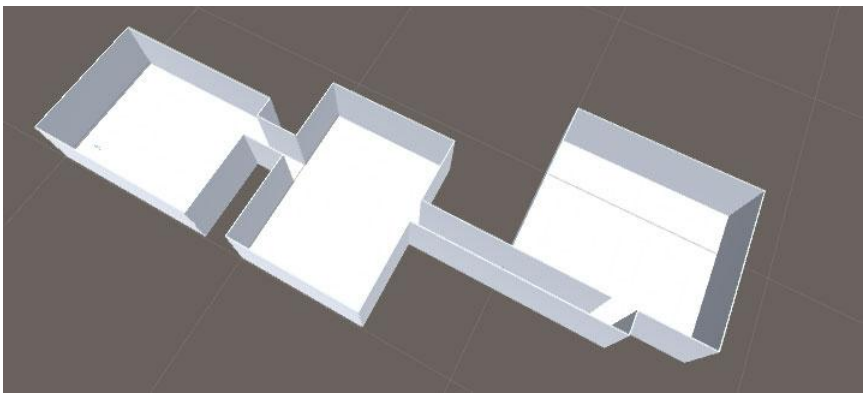

Figura 6. Un piccolo dungeon di tre stanze

2.4.2 Metodo di costruzione del modello astratto

Con il metodo di costruzione del modello astratto in questa prima fase ci si riferisce all'insieme di operazioni computazionali e di strutture dati che permettono di realizzare la mappatura delle stanze, disposte nello spazio, sulla matrice a due dimensioni (timeMatrix).
Lo pseudo codice di questa fase, che realizza l'algoritmo di disposizione delle stanze è il seguente:

```
1. for (int i = 0; i < roomNum; i++)
2.       Rooms[i] = createRoom(minW, maxW, minH, max)
3.            while(tileMatrix.checkOverLap(Rooms[i]))
4.                dir = Random.Range(0, 2)
5.                 if (dir == 0)
6.                     Rooms[i].moveRoom(minShitValue,"UP")
7.                 else if( dir == 1 )
8.                     Rooms[i].moveRoom(minShitValue,"RIGHT")
```

L'algoritmo consta di un for esterno che itera per un numero di volte pari a quello delle stanze che si vogliono creare. Dopo aver creato una stanza, attraverso la funzione `createRoom`, con il ciclo while interno esegue un loop che effettua lo spostamento della i-esima stanza in alto oppure a destra, fintanto che non provoca più sovrapposizioni. L'esistenza di una

sovrapposizione viene direttamente verificata utilizzando il metodo `checkOverLap` della matrice `tileMatrix`.

Come si può notare, fino ad ora, le stanze non sono state create, infatti, in questo modo l'algoritmo si limita solo ad usare i dati contenuti nell'array `Rooms` senza muoverle nello spazio, evitando quindi fare chiamate al sottosistema di *rendering* per la visualizzazione e la manipolare nello spazio delle stanze; ci si limita semplicemente ad utilizzare oggetti molto semplici come la `tileMatrix` e il suo contenuto.

2.4.3 Metodo di costruzione e disposizione delle stanze del dungeon

La creazione nello spazio delle stanze viene effettuata dopo il ciclo for del metodo di costruzione del modello astratto, iterando sull'array `Rooms` per prendere i dati delle stanze ed istanziarle nello spazio usando le API di Unity

Per questioni di ottimizzazioni invece di utilizzare *HxW* (con H altezza e W larghezza) singoli game object per le mattonelle di una stanza aumentando il carico di lavoro imposto alla scheda video, il pavimento delle stanze viene realizzato come singolo oggetto delle dimensioni specifiche di ogni stanza. Le celle però, anche se invisibili, saranno istanziate e continueranno ad esistere in memoria come oggetti e saranno utilizzate per memorizzare i riferimenti alle unità di muro possedute.

2.4.4 Costo computazionale

Il costo computazionale di questa fase è sostanzialmente dovuto al metodo di costruzione del modello astratto, che opera direttamente sulla *tileMatrix* (il metodo di costruzione e di disposizione delle stanze nello spazio ha un costo trascurabile pari ad $O(n)$).

Il caso pessimo in cui bisogna collocarsi è quello in cui le stanze vengono posizionate una dopo l'altra in una singola direzione, questa situazione anche se molto improbabile, può verificarsi quando l'algoritmo posiziona le stanze sequenzialmente in una singola direzione. Ad esempio può scegliere di fare crescere il dungeon verso l'alto andando a posizionare una stanza sopra l'altra oppure verso destra. Nella figura 7 si è cercato di riprodurre manualmente il primo caso in cui le stanze vengono sovrapposte una sopra l'altra.

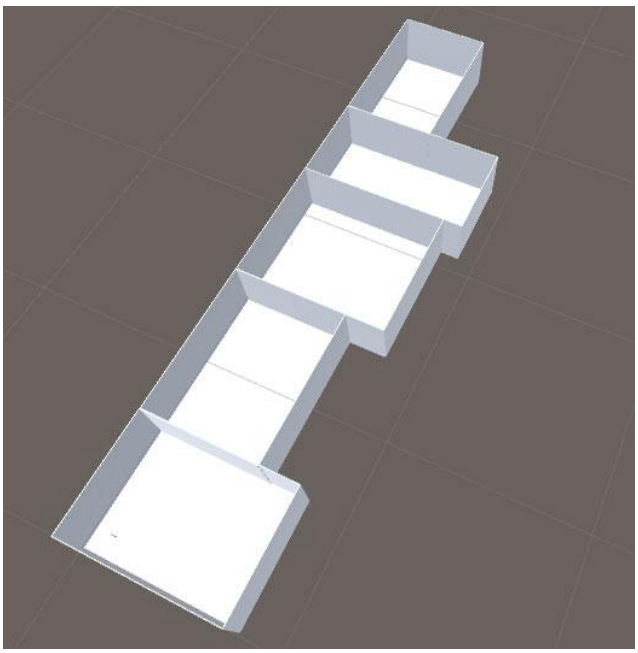

Figura 7. Improbabile disposizione delle stanze dovuta a un comportamento al caso pessimo dell'algoritmo.

Il costo del ciclo while dello pseudo-codice visto nel metodo di costruzione del modello astratto è dovuto al metodo `checkOverLap`, utilizzato nella guardia: considerando tutte le stanze della stessa dimensione, in modo da semplificare il calcolo del costo, questo metodo controlla che creando una stanza non occorrano sovrapposizioni e quindi il suo costo è pari alla dimensione della stanze, moltiplicato per il numero di stanze precedentemente piazzate. Tenendo conto anche del for esterno e scrivendo il costo in modo formale, esso è pari a: $O(\ (w * h)\ *\ \sum_{i=1}^{n} i\ )$ dove w e h sono rispettivamente la larghezza e l'altezza delle stanze ed n è il numero di stanze da creare, l'intero costo può essere riscritto anche come $O((w * h)\ * n^2)$. Generalmente il numero di stanze è molto maggiore del prodotto $(w * h)$, per cui si può affermare che il costo di questa fase è pari ad $O(n^2)$.

Si noti che al termine di questa prima fase non esiste alcuna connessione tra le stanze e che quindi sono ancora isolate l'una dall'altra.

2.5 Connessione delle stanze

In questa sezione viene descritta la seconda fase dell'algoritmo, che ha il compito di connettere le stanze dei dungeon.

2.5.1 Modello di rappresentazione astratta

Una delle scelte più importanti nella progettazione del generatore procedurale è stata quella di decidere il tipo di connessione che si voleva stabilire tra le stanze, quindi con il modello di rappresentazione astratta si stabilisce la struttura che permette di collegare le stanze; astraendo dal fatto che sono stanze, ma considerandole invece come singoli punti sul piano euclideo. La scelta di come strutturare i collegamenti è stata guidata dai seguenti vincoli:

- Tutte le stanze devono poter essere visitate e una volta terminata la generazione non si devono verificare casi in cui esistono stanze non raggiungibili.
- Deve essere possibile attraversare tutte le stanze percorrendo la minima quantità di spazio di interconnessione: partendo dal centro di una qualunque stanza, deve esiste un insieme di collegamenti il cui costo totale deve essere minimo (cioè deve esserci un albero di copertura minimo).
- Devono essere presenti connessioni ridondanti tra le stanze, che possibilmente formano cicli, questo rende più interessanti i percorsi e consente l'esplorazione del dungeon con percorsi multipli

Date queste premesse la struttura dati che è più appropriata per modellare questo tipo di connessione è chiaramente un grafo non orientato, nel caso particolare dell'output desiderato si vuole che il grafo non presenti sovrapposizioni degli archi, perché questo significherebbe che nel modello concreto poi i corridoi sarebbero sovrapposti; come vedremo più avanti però, nel metodo di costruzione, non sarà possibile evitare del tutto le sovrapposizioni, ma almeno per quanto riguarda il modello astratto questo è uno dei requisiti. La scelta naturale ricade quindi sui grafi planari. In particolare, come introdotto all'inizio del capitolo 2, per connettere le stanze viene costruito un grafo di tipo RNG [11]. I grafi RNG fanno parte della famiglia di grafi planari e connettono un insieme di nodi sul piano euclideo, seguendo la seguente regola:

si crea un arco tra la coppia di nodi *i* e *j* se e solo se non esiste un altro nodo diverso da *i* e da *j* tale che la sua distanza sia da *i* che da *j* è minore della distanza tra *i* e *j*, è possibile vedere un esempio di questa grafo in figura 5 a pagina 25. Il tipo di connessioni che si creano è adeguato per il generatore, infatti, idealmente si desidera che le connessioni tra le stanze crescano tra di esse in modo graduale, non esistono archi molto lunghi che saltano da un estremo all'altro del grafo, infine i grafi RNG, per come vengono costruiti garantiscono la presenza del albero di copertura euclideo come sottoalbero, per cui si può affermare che soddisfano i requisiti cercati.
Bisogna notare però che la scelta progettuale di decidere il modo con cui collegare le varie stanze è semplicemente una delle possibili scelte ed è importante sottolineare che non esiste un modo migliore dell'altro per formare le connessioni nei dungeon.

2.5.2 Metodo di costruzione del modello astratto

L'algoritmo di connessione (implementato all'interno della classe Dungeon.cs) viene eseguito successivamente alla creazione e alla disposizione delle stanze con assenza di sovrapposizioni.
L'algoritmo considera le stanze come punti singoli, le cui coordinate corrispondono ai centri delle stanze, cioè nell'incrocio delle due diagonali. La lunghezza degli archi viene calcolata come semplice distanza euclidea nello spazio bidimensionale.
L'algoritmo procede nel seguente modo: per ogni coppia di nodi <*i,j*> calcola la loro distanza *d(i,j)* e le distanze *d(k,i)*, *d(k,j)* con un terzo nodo *k*, diverso da *i* e *j*. Itera quindi *k* su tutti i nodi rimanenti e se non esiste alcun nodo *k* tale che abbia entrambe le sue distanze *d(k,i)* e *d(k,j)* minori della distanza *d(i,j)* allora si crea effettivamente l'arco tra la coppia <*i,j*>.

Lo pseudo codice dell'algoritmo per creare il grafo RNG è il seguente:

```
1. for(i=0;i<roomNum;i++)
2.       for(j=i+1;j<roomNum;j++)
3.             skip=false
4.             for(k=0;k<roomNum;k++)
5.                   if(k==i || k==j)
6.                         continue
7.                   if (Max(dist(i,k),dist(j,k)) < dist(i,j))
8.                         skip=true
9.                         break
10.            if(!skip)
11.                  createEdge(i,j)
```

2.5.3 Metodo di costruzione del grafo RNG

Il metodo di costruzione di questa seconda fase non è molto rilevante in termini delle strutture che vengono allocate nello spazio, in quanto non ce ne sono. Il grafo RNG non viene concretamente creato ma serve solo come base di riferimento su cui si appoggerà l'ultima fase destinata a creare i corridoi; quindi il metodo di costruzione del grafo RNG si limita a creare un reticolo di connessione al di sopra delle stanze già presenti nella scena. Questo avviene sfruttando la funzione `MonoBehaviour.OnDrawGizmos()` di Unity, che fa parte di quell'insieme di funzioni che il motore di rendering di Unity esegue all'interno del suo ciclo principale e viene utilizzato per disegnare sullo schermo oggetti che non faranno parte del gioco, ma servono prevalentemente in fase di sviluppo.
In figura 8 è possibile vedere il risultato dell'algoritmo che crea il grafo, applicato ai risultati della prima fase. Nella figura il grafo viene rappresentato da delle linee di connessione rosse che collegano i centri delle stanze. Questo grafo non fa effettivamente parte del dungeon, infatti, è possibile disattivare o attivare la sua visualizzazione attraverso l'interfaccia del generatore.

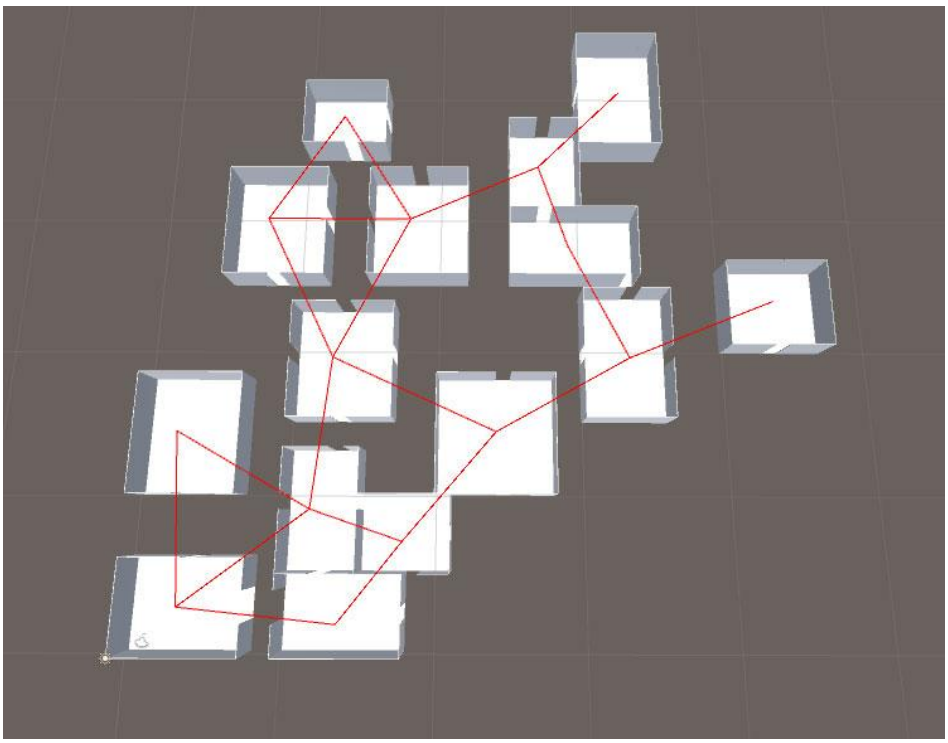

Figura 8. Un dungeon formato da 15 stanze, con larghezza e altezza che variano dalle 5 alle 10 unità. Il valore del parametro minShiftValue è pari a 5 unità, il seme che ha generato questa disposizione è 6810754.

Dalla figura 8 si può notare che alcune unità di mura perimetrali delle stanze sono mancanti in effetti questo è corretto in quanto questa figura è stata ricavata nascondendo attraverso l'interfaccia grafica del generatore i corridoi, per cui il risultato è del tutto normale.

2.5.4 Costo computazionale

Il costo computazionale di questa fase è dovuto al metodo di costruzione del modello astratto ed è pari a $O(\sum_{i=1}^{n-1}(n-i)*n)$, infatti il *for* più interno viene eseguito per n volte moltiplicato per il numero di volte per cui viene eseguito il for subito sopra, cioè $n-i$, dove i varia con il contributo del for più esterno. Dato che la sommatoria presente nel costo rappresenta la somma dei primi $n-1$ numeri, si ottiene un costo quadratico, per cui è possibile approssimare il costo di questa seconda fase con $O(n^3)$. Il contributo in termini di costo della prima fase risulta ora trascurabile.

2.6 Creazione dei corridoi di connessione

Questa è la terza e ultima fase dell'algoritmo, in cui si utilizza il grafo RNG formato nella fase precedente per creati i corridoi nello spazio tridimensionale. Questa è stata fase più complessa, non tanto da un punto di vista algoritmico, ma per la copertura di tutti i casi che si possono presentare creando i corridoi.

Parlando in termini di correttezza, l'obiettivo è stato quello di avere le stesse connessioni del grafo senza incappare in possibili problemi quali ad esempio: l'isolamento delle stanze, possibile grazie a incroci di corridoi che formano vicoli ciechi, oppure stanze adiacenti che dovrebbero essere connesse, ma che non presentano alcun modo di essere raggiunte l'una dall'altra. Gli incroci che si possono formare in questa fase non sono derivanti dallo schema di connessione RNG, infatti, il grafo RNG assicura che i suoi archi non si sovrappongono mai per come viene costruito; ma la scelta implementativa con cui vengono concretamente creati i creati i corridoi può portare a delle situazioni in cui si verifica questo tipo di problema.

Vediamo allora la descrizione di questa fase secondo i soliti tre punti: nella rappresentazione del modello astratto sarà descritta l'idea generale di come si connettono due stanze, con il metodo di costruzione del modello astratto viene invece descritto da un punto di vista algoritmico come si realizzano le connessioni ed infine nel modello di costruzione della geometria dei corridoi è descritto come vengono creati i corridoi; quest'ultimo punto è quello meno importante in quanto utilizza i soliti strumenti, quali i prefab e le API di Unity per l'allocazione di game object nello spazio.

2.6.1 Modello di rappresentazione astratta

Ogni collegamento, identificato da un arco del grafo RNG e reso concreto come corridoio, viene generalmente composto da due segmenti tra loro intersecati. La figura 9 chiarisce meglio questo concetto: l'arco che collega le due stanze viene tradotto nei due segmenti che compongono il corridoio. Nel caso in questione si può osservare che la connessione che parte dalla prima stanza, quella in basso a sinistra e che arriva alla seconda avviene prima con un segmento orizzontale e poi con uno verticale, questo percorso però non è l'unico possibile: esso, infatti, viene deciso in modo casuale in questa terza fase dell'algoritmo. Un possibile

percorso alternativo è rappresentato dai due segmenti verdi, che formano un corridoio che cresce inizialmente in verticale e poi in orizzontale, fino a raggiungere il perimetro della seconda stanza.

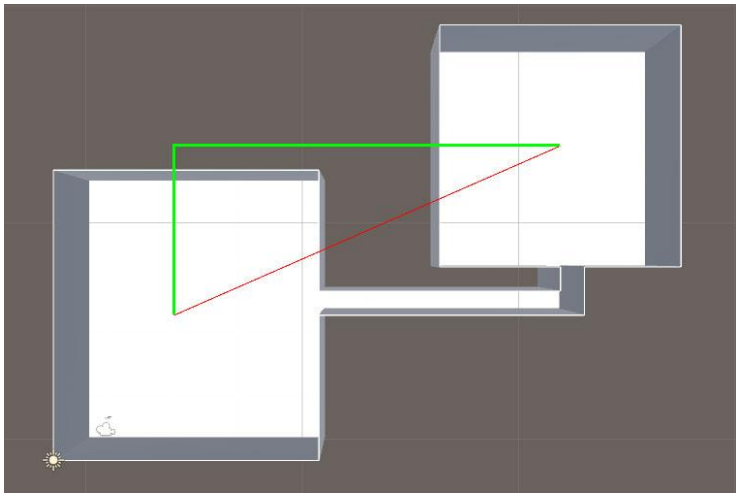

Figura 9. Due stanze collegate correttamente con un corridoio a 2 segmenti.

Il caso visto in figura 9, è uno dei casi base, ma non è il più semplice. Quello più semplice occorre quando le due stanze sono allineate, come ad esempio si può osservare in figura 10, dove la vicinanza tra le due stanze fa si che non ci sia bisogno del secondo segmento di connessione.

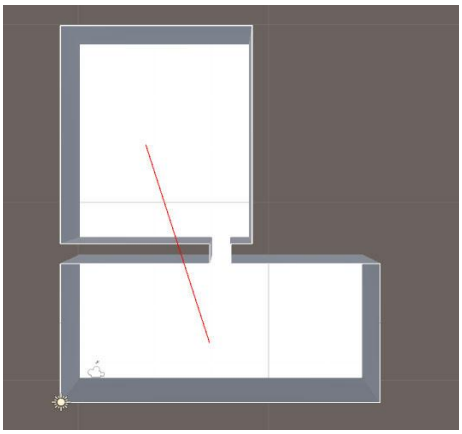

Figura 10. Due stanze collegate correttamente direttamente da un unico segmento di corridoio.

Esiste un caso ancora più semplice dei due appena visti e si presenta quando le stanze sono adiacenti e una porzione del loro perimetro è in comune. Un risultato di questo tipo è osservabile dalla figura 11.

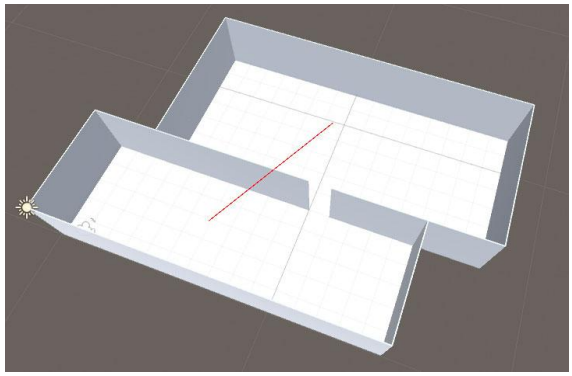

Figura 11. Due stanze adiacenti collegate sulla porzione di perimetro condiviso.

Questo tipo di risultato è stato ottenuto impostando il valore di *minShiftValue* a 1, in questo modo nella fase di piazzamento delle stanze, ad ogni passo le stanze vengono spostate di una sola unità. Impostando ad 1 questo valore si ottengono sempre agglomerati di stanze compatte, in cui i collegamenti sono perlopiù perimetrali. In figura 12 si possono osservare 10 stanze di un dungeon create e collegate con *minShiftValue* uguale ad 1.

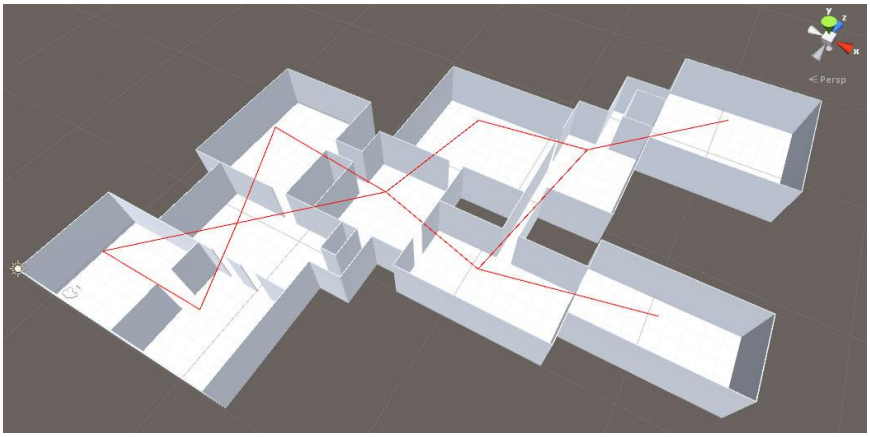

Figura 12. Dungeon formato con minShiftValue uguale a 1.

Oltre a questi casi vi sono quelli più complessi che si presentano nel momento in cui i corridoi si incrociano; vedremo nella sezione successiva come vengono gestiti da un punto vista algoritmico, ai fini però del modello astratto è utile avere un'idea di quando si presentano. Il concetto fondamentale però su come gli archi di connessione vengono tradotti rimane sempre il solito e cioè attraverso due semplici segmenti tra loro congiunti. In figura 13 si può vedere un caso particolare in cui tre corridoi si incrociano in modo corretto, è possibile notare che non si formano vicoli ciechi, che andrebbero ad isolare le stanze.

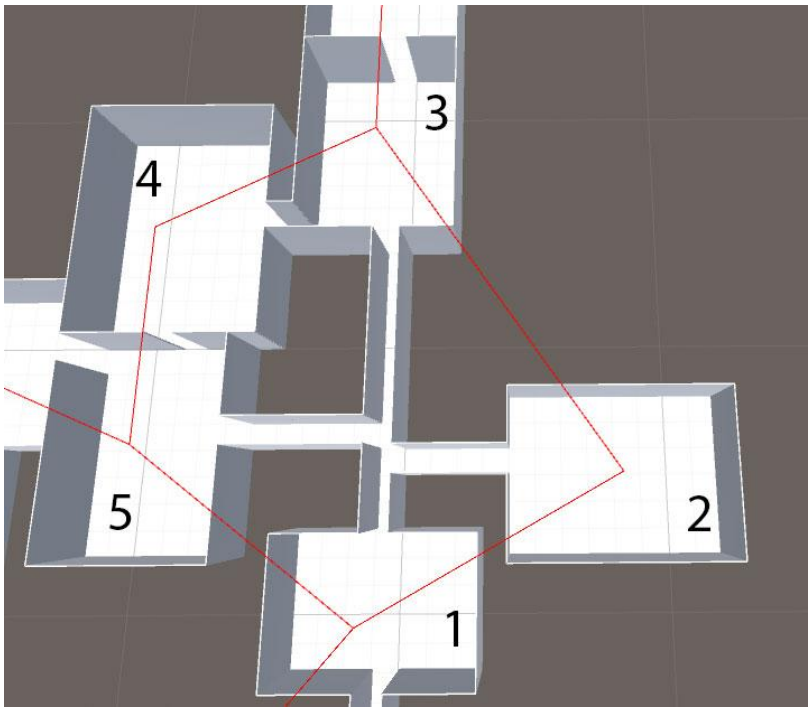

Figura 13. Porzione di un dungeon che presenta un interessante collegamento tra 5 stanze con la formazione di un incrocio di corridoi.

Nella figura sopra i collegamenti che si formano tra le seguenti coppie di stanze <1,2>, <2,3> e <1,5> producono come risultato la formazione di un punto di incrocio dei corridoi dando la sensazione che sia un'unica struttura coesa, mentre in realtà in totale sono stati creati 6 segmenti, 2 per ogni coppia di stanze. Questo tipo di situazione occorre molto spesso quando il numero di stanze comincia a essere consistente e si ottengono dungeon più interessanti da essere esplorati, contrariamente ad una struttura più regolare e semplice.

2.6.2 Metodo di costruzione del modello astratto

Descriviamo ora da un punto di vista algoritmico il processo con cui vengono realizzati i corridoi, saranno analizzati i singoli casi particolare che si possono manifestare. Al fine di avere una visione globale delle operazioni svolte è possibile riassumere i passi di creazione di un corridoio nel seguente modo:

1. Inizialmente viene effettuata la chiamata alla funzione principale che crea il corridoio tra due stanze.
2. In seguito si eseguono le chiamate alle funzioni designate a creare i due segmenti, scegliendo in modo casuale se creare prima quello verticale e poi quello orizzontale oppure viceversa.
3. Infine si verifica la raggiungibilità delle stanze: ricordando che ogni segmento è composto da semplici celle unitarie, a cui è possibile affiancare delle unità di muro, la funzione che costruisce ogni singola cella di un corridoio si occupa anche di rilevare eventuali sovrapposizioni con altre stanze, oppure con corridoi già esistenti e quindi di distruggere alcune unità di muro per liberare i passaggi.

Per ogni coppia di stanze collegate da un arco viene invocata la seguente funzione:

```
createCorridor(Rooms[i], Rooms[j])
```

Questa funzione da inizio alla creazione del corridoio tra la stanza i-esima e j-esima, con i seguenti passi:

- Nel corpo della `createCorridor` si invocano le funzioni che creano i due segmenti di un corridoio decidendo casuale l'ordine di creazione dei segmenti; vediamo lo pseudo-codice:

```
1. dx = roomA.Center.x - roomB.Center.x
2. dz = roomA.Center.z - roomB.Center.z
3. if (dz < 0) nextDirZ = "north"
4. else nextDirZ = "south"
5. if (dx < 0) nextDirX = "east"
6. else nextDirX = "west"
7. dir = Random(0,1)
8. switch (dir)
9.       case 0:
10.         lastPosH=createHCorridor(roomA.Center,dx,nextDirZ)
11.         lastPosV=createVCorridor(lastPosH,dz,"")
12.         break
13.      case 1:
14.         lastPosV=createVCorridor(roomA.Center,dz,nextDirX)
15.         lastPosH=createHCorridor(lastPosV,dx,"")
16.         break
```

Le prime sei istruzioni servono per impostare la prossima direzione di crescita, cioè si decide come direzionare il secondo segmento che compone il corridoio, quindi a tal fine si calcolano i due valori dx e dz, che identificano rispettivamente la distanza sulle ascisse e sulle ordinate tra i centri delle stanze A e B. Se dz è minore di 0 allora si imposta la direzione di crescita del secondo segmento verso l'alto, cioè "north", questo significa che la stanza B si trova al di sopra della stanza A in termini di ordinate, altrimenti la direzione successiva di crescita è "south". Un discorso analogo è valido per le ascisse, cioè per decidere se il prossimo segmento deve crescere verso est o ovest.

Poi in modo casuale si imposta la con la variabile `dir` la direzione di crescita del primo corridoio, se è pari 0 allora è orizzontale e poi verticale, viceversa se è 1. Come si può vedere le funzioni `createHCorridor` e `createVCorridor` ricevono come terzo parametro una stringa che stabilisce la direzione di crescita del secondo segmento, inoltre si può notare che nella prima chiamata questo valore viene impostato, mentre nella seconda viene passata una semplice stringa vuota in quanto ormai il corridoio è stato formato nella sua interezza e non ci sono più direzioni di crescita ulteriori.

Per chiarire meglio questo processo nella figura 14 viene mostrato un caso di due stanze collegate, la figura è arricchita con le informazioni riguardanti i valori che assumono le variabili dello psuedo-codice appena descritto (i due segmenti sono anche evidenziati con delle linee verdi)

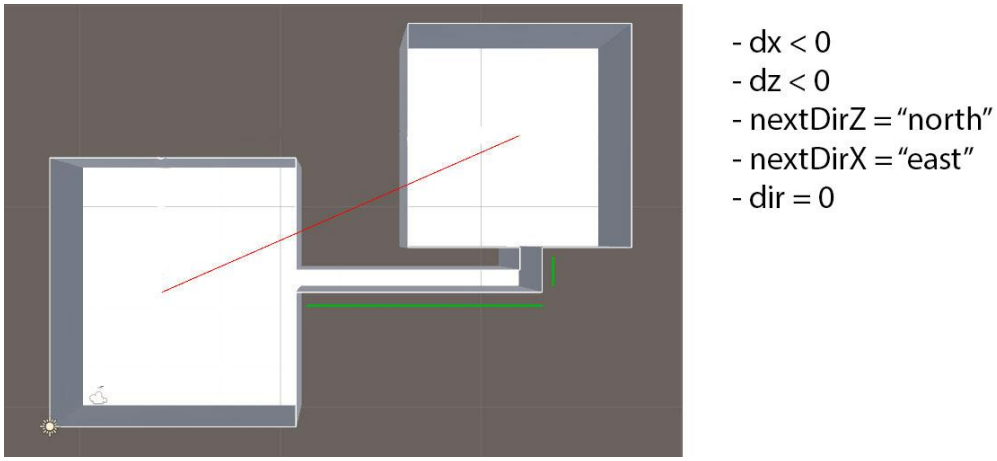

Figura 14. Sono mostrati i valori assunti da alcune variabili utilizzate nell'algoritmo nella fase di collegamento.

Vediamo ora le operazioni che vengono effettuate dalle due funzioni `createHCorridor` e `createVCorridor`; per semplicità verrà descritta soltanto la prima.

La funzione `createHCorridor`, come già visto, prende come parametri di input il centro di una stanza come coppia di coordinate <x,z>, un valore intero che identifica la lunghezza del segmento e come ultimo parametro una stringa che rappresenta la direzione di un eventuale secondo segmento; l'utilizzo di quest'ultimo parametro sarà chiarito a breve.

Pur traducendo la funzione in pseudo-codice, si sarebbe ottenuto un listato molto lungo e quindi sua descrizione sarà divisa in varie parti.

La prima parte è piuttosto semplice: si itera sulla lunghezza `lenght` (l'intero in valore assoluto del secondo parametro) in modo da creare fisicamente ogni singolo elemento unitario che compone un corridoio (composto da una mattonella di base e due unità di muro laterale).

```
1. for (i=0;i<Abs(lenght);i++)
2.       if (i == (Abs(lenght) - 1))
3.             lastTile = true
4.       if (lenght < 0)
5.             startP=new Coord2D(startP.x+1,startP.z)
6.             aCell=createCorTile(startP,"east",nextSdir,lastTile)
7.             direction=1
8.       else if (lenght>0)
9.             startP=new Coord2D(startP.x-1,startP.z)
10.            aCell=createCorTile(startP,"west", nextSdir,lastTile)
11.            direction=-1
12.      if (i<(Abs(lenght)-1) && aCell!=null)
13.            CreateCorridorWalls(aCell, 0, aCell.transform)
```

Tralasciando momentaneamente la semantica della variabile `direction` è possibile osservare che all'interno del ciclo si decide in base al segno della lunghezza se la direzione di crescita è verso est oppure verso ovest. Quando viene raggiunta l'ultima iterazione si imposta il valore della variabile booleana `lastTile` che viene passata come parametro alla funzione `createCorTile` ed utilizzata per gestire la formazione delle giunzioni tra i due segmenti di un corridoio. Con le ultime due istruzioni si verifica se la cella di base del corridoio è stata creata e se l'indice d'iterazione è minore del penultimo passo, in tal caso si creano le unità di mura laterali. Quando invece si effettua l'ultima iterazione, l'operazione che crea i muri non viene eseguita ed è delegata alla seconda parte di questa funzione, che si occupa di effettuare questo tipo di rifinitura nelle giunzioni tra segmenti e contempla tutti vari casi possibili. Per descriverla è più utile utilizzare delle figure, anziché lo pseudo-codice, mostrando quali casi si possono presentare. Possiamo distinguere subito 2 macro casi, che loro si dividono in altri due ulteriori casi, vediamoli singolarmente:

1. Se la prima direzione di crescita è verso est e quindi la variabile `direction` ha valore pari ad 1 allora si hanno i due sotto casi possibili:
    a. Se il secondo segmento cresce verso nord allora l'ultima cella del primo segmento dovrà avere un muro di confine rivolto a sud e uno rivolto verso est; è possibile osservare questo configurazione in figura 15, caso a.
    b. Se invece una volta creato il primo segmento si decide che la crescita del secondo segmento è verso sud allora l'ultima cella del primo segmento dovrà avere un muro nord e uno rivolto verso east; è possibile osservare questo caso in figura 15, caso b.
2. Il secondo caso occorre quando la prima direzione di crescita è verso ovest, impostando questa volta la variabile `direction` al valore -1, ottenendo anche qui due casi:
    a. Se il secondo segmento è fatto crescere verso nord allora l'ultima cella del primo segmento dovrà avere un muro rivolto verso ovest e uno rivolto verso sud; è possibile osservare questo caso in figura 16, caso a.
    b. Se invece il secondo segmento cresce verso il basso, quindi verso sud l'ultima cella del primo segmento dovrà avere un muro rivolto ad ovest e uno rivolto a nord; è possibile osservare questo caso in figura 16, caso b.

Nell figure sottostanti le zone di giunzione dei segmenti sono state evidenziate da un rettangolo verde e con un colore giallo invece sono evidenziate le unità di muro impostate per la giunzione.

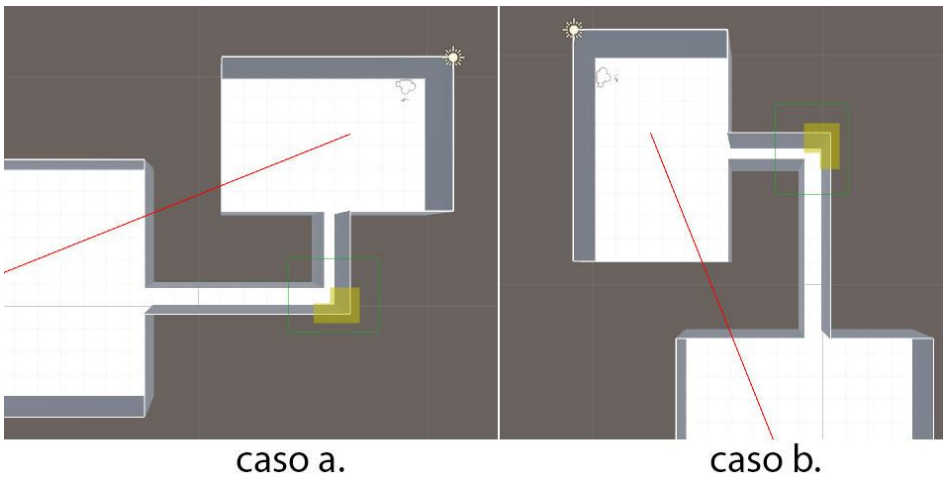

Figura 15. La direzione del primo segmento orizzontale è verso est e quella del secondo segmento nel caso a. è verso nord, nel caso b. è verso sud.

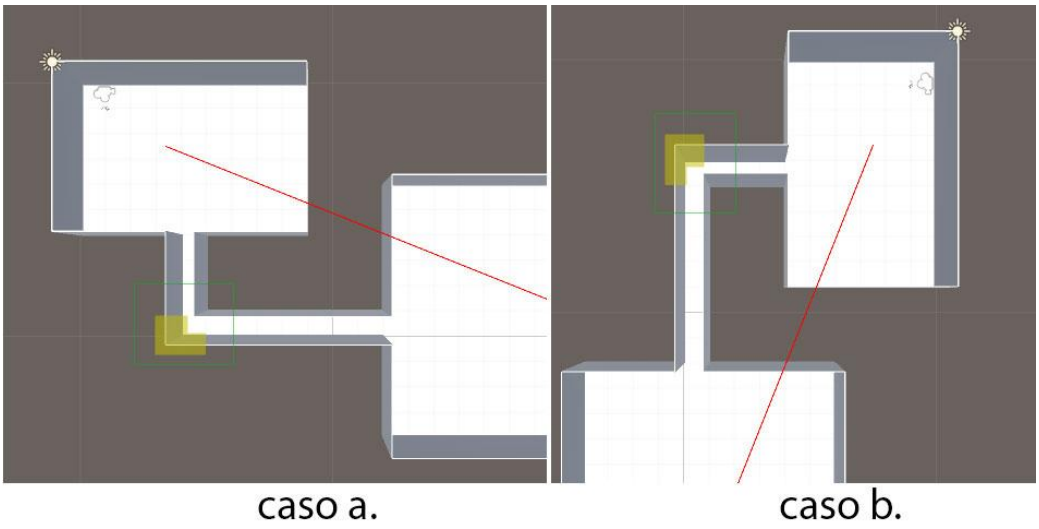

Figura 16. La direzione del primo segmento è orizzontale ed è verso ovest, mentre quella secondo segmento nel caso a. è verso nord e nel caso b. è verso sud.

Oltre a questi quattro casi quindi ne esistono altri quattro e contemplano le altre possibili configurazioni che si presentano quando il primo segmento di crescita è in verticale invece che in direzione orizzontale.

La descrizione ora continua esaminando come viene effettivamente creata una singola porzione di corridoio, in particolare vedremo come è implementata la funzione:

```
corridorCell createCorTile([Coord2D],[string],[string],[bool])
```

In input si aspetta di ricevere: una coordinata spaziale in due dimensioni, due stringhe e un valore booleano. Il primo parametro, la coordinata spaziale, indica il punto in cui si esegue la creazione, la prima stringa definisce la direzione di crescita del segmento di cui fa parte l'unità di corridoio che sta per essere creata, la seconda stringa invece si riferisce all'eventuale direzione di crescita del secondo segmento (se esiste), l'ultimo parametro indica se si sta creando l'ultima unità del segmeno. La funzione restituisce l'oggetto di tipo cella creato.
Le operazioni interne a questa funzione sono principalmente una serie di controlli "if" che esaminano le configurazioni possibili in cui si può trovare il processo di creazione e che decidono il modo corretto di procedere della computazione. Per esporre meglio le operazioni si utilizza di nuovo di un pezzo di pseudo-codice preso direttamente come traduzione dal codice reale scritto in C#:

```
1. if (tileMatrix[c.z, c.x] == 0)
2.       tileMatrix[c.z, c.x] = 2
3.       aCell = CreateCorridorCell(c)
4.       activeDungeonCells.Add(c, aCell)
5.       return aCell
```

La prima operazione che si effettuata è un controllo sulla posizione `[c.z, c.x]`, se questa è pari a 0 ed è quindi libera si imposta il valore 2 nella matrice, indicando che in quella posizione si sta per creando un pezzo di corridoio. Il riferimento alla cella appena creata viene inoltre salvato in dizionario, utilizzando come chiave la sua posizione (vedremo poi perché). Alla fine poi il riferimento alla cella viene restituito al chiamante.
In seguito abbiamo il seguente frammento di pseudo-codice, che viene eseguito solo se nel passo precedente non si è riusciti a creare la cella (e cioè in posizione `[c.z, c.x]` c'era un valore diverso da zero):

```
1. if (tileMatrix[c.z, c.x] == 1 && !lastTile)
2.       destroyWall(c, sDir, true)
3. else if (tileMatrix[c.z, c.x] == 1 && lastTile)
4.       destroyWall(c, nextSdir, false)
5.       destroyWall(c, oppositeDir(sDir), false)
6. if (tileMatrix[c.z, c.x] == 2)
7.       if (lastTile)
8.             destroyWall(c, nextSdir, false)
9.             destroyWall(c, oppositeDir(sDir), false)
10.      else
11.            destroyWall(c, sDir, true)
12.      return null;
```

Nel primo if si controlla se si è all'interno di una stanza, in altre parole se nella coordinata `[c.z, c.x]` di `tileMatrix` sia memorizzato un valore pari ad uno ed inoltre si verifica che non sia l'ultima porzione del segmento. Se la condizione congiunta è vera, allora si esegue l'operazione:

```
destroyWall(c, sDir, true)
```

Questa funzione riceve come parametri una coordinata `c` in due dimensioni, una stringa che indica una direzione polare e un valore booleano. La funzione esegue la distruzione di un'unità di muro presente nella cella di coordinate `c`, orientata nella direzione `sDir`, il terzo valore booleano indica che devono essere distrutti due unità di muro, ovvero: se la direzione passata è est oppure ovest allora saranno distrutti i muri rivolti ad est e ad ovest, mentre se la direzione è nord o sud allora verranno distrutti i muri rivolti verso sud e nord. In questo modo si consente a un corridoio in uscita da una stanza di crearsi un varco verso la direzione percorsa, distruggendo eventuali muri nel suo percorso e poi nella stanza di destinazione distruggerà il muro perimetrale d'ingresso. Così facendo si creano i varchi di uscita da una stanza di origine e d'ingresso verso una stanza destinazione.
L'algoritmo contempla sia il caso della figura 10, in cui il corridoio viene creato sui perimetro adiacente condivido da due stanze e anche il caso più classico della figura 14. Infatti, facendo riferimento alla figura 14 è possibile osservare che la creazione del primo segmento orizzontale

comporta anche l'abbattimento del muro per il primo varco in uscita, quando poi si raggiunge il confine della prima stanza e la condizione seguente è vera:

```
tileMatrix[c.z, c.x] == 1 && !lastTile
```

Se invece la condizione precedente non si verifica, si prosegue con quella successiva nel ramo else subito sotto:

```
tileMatrix[c.z, c.x] == 1 && lastTile
```

Si vuole cioè capire se sia stato raggiunto l'ultimo elemento del segmento, pur rimanendo all'interno di un'area occupata da una stanza, che potrebbe essere sia quella di partenza oppure quella di destinazione, ma potrebbe benissimo anche essere una stanza di transizione che si trova nel percorso tra il punto di origine e di destinazione. In questo caso quindi si eseguono le due istruzioni di distruzione:

```
destroyWall(c, nextSdir, false)
destroyWall(c, oppositeDir(sDir), false)
```

Con la prima operazione di distruzione si demolisce un solo muro (infatti viene passato false come terzo parametro) nella direzione `nextSdir` del successivo segmento. Con la seconda istruzione invece si demolisce il muro dell'ultima cella nella direzione opposta di crescita del primo segmento. Queste due istruzioni servono sostanzialmente a coprire un caso molto particolare: cioè quando il corridoio che si sta creando deve attraversare una stanza intermedia tra l'origine e la destinazione, a tal proposito è utile osservare la figura 17. Dalla figura si vede che dalla stanza numero 1 esiste un collegamento verso la stanza 3, rappresentato dal percorso evidenziato in verde. L'algoritmo decide di passare attraverso la stanza numero 2, quindi vengono eseguite proprio le due istruzioni appena descritte, infatti, nell'ultima cella del primo segmento (quello verticale) il percorso si trova nella stanza 2 e sono vere sia la condizione di trovare 1 nella tileMatrix che di essere nell'ultima cella del segmento. Per cui vengono distrutte le unità muro corrispondenti alla direzione opposta di marcia del primo segmento (cioè quella verso sud) demolendo il muro orizzontale evidenziato in rosso e distruggendo anche il muro

evidenziato in rosso verticale che è orientato nella direzione del secondo segmento, cioè verso est, equivalente al valore che assume la variabile nextSdir.

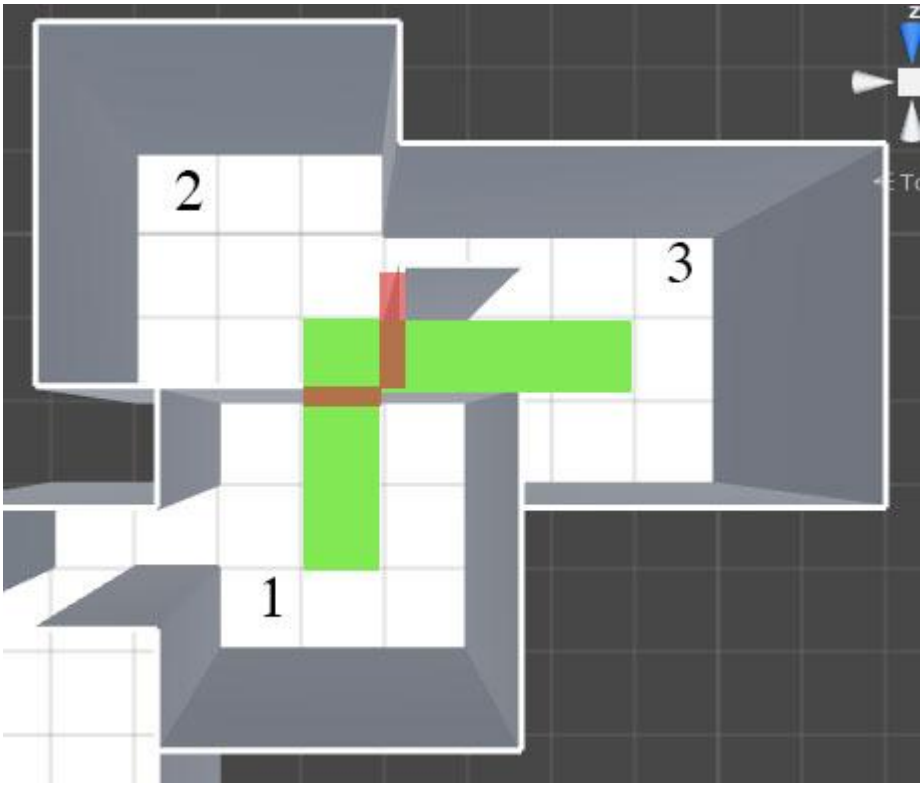

Figura 17. Caso particolare in cui il percorso, evidenziato in verde, attraversa una stanza intermedia tra l'origine (la stanza 1) e la destinazione (la stanza 2), i muri che vengono distrutti sono evidenziati invece in rosso.

Con l'ultimo if, sulla riga 6, si controlla se il contenuto della matrice nella coordinata `c` è pari a 2, si vuole capire se si sta tendando di costruire la porzione di corridoio sovrapponendola ad una già esistente e bisogna quindi distingue due ulteriori casi: se si sta considerando l'ultima cella del segmento oppure se è una cella intermedia. Vediamo prima quest'ultimo caso che è più semplice, in questa situazione viene è eseguita l'istruzione.

```
destroyWall(c, sDir, true)
```

per cui quando si verifica una sovrapposizione di corridoi vengono demoliti gli eventuali muri di ostruzione in entrambe le direzioni di crescita del segmento corrente, infatti viene passato `true` come terzo parametro alla funzione `destroyWall`, indicandole che vanno distrutte le coppie di muri nella direzione `sDir` e quindi nord e sud se la direzione è nord o sud e est e ovest se la direzione è est o ovest.

Se invece si sta costruendo l'ultima cella del segmento e c'è una sovrapposizione con un altro corridoio, allora bisogna liberare solo le ostruzioni nella direzione di marcia del segmento successivo (se esiste) e quelle nella direzione opposta del primo segmento. Un esempio di sovrapposizione di corridoi è mostrato in figura 18, nella quale abbiamo 3 stanze in cui la numero 1 e la numero 2 vogliono connettersi alla numero 3. Se ad esempio l'algoritmo di connessione decide ci collegare prima la stanza 1 e la stanza 3, creando un segmento verticale diretto verso sud e poi uno orizzontale diretto verso est per raggiungere la destinazione, quando poi si crea il corridoio che collega la stanza 2 alla stanza 3 viene rilevata una sovrapposizione nella terza cella del segmento orizzontale. Quindi vengono abbattuti tutti gli muri formati nella costruzione del corridoio precedente (east e ovest); nel caso in questione esiste solo un muro rivolto verso ovest che faceva parte della giunzione dei due segmenti di collegamento tra la stanza 1 e 3.

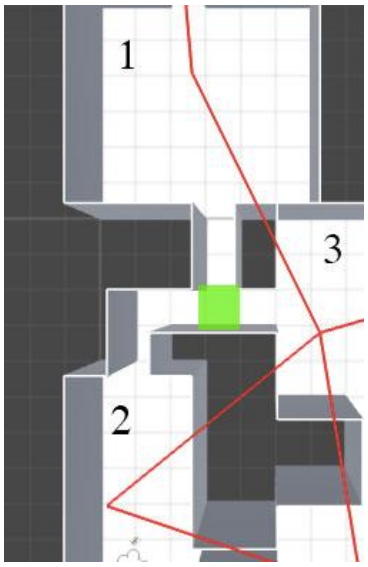

Figura 18. Caso di sovrapposizione tra due corridoi.

2.6.3 Metodo di costruzione della geometria dei corridoi

I corridoi sono costituiti da un insieme di singole unità: una cella di base e da eventuali unità di muro. Sono utilizzati i prefeb DungeonCell per la base e WallUnit per i muri e sono instanziati nello spazio invocando la funzione:

```
CreateCorridorCell(c)
```

Che riceve come parametro la coordinata spaziale in due dimensioni del punto in cui andrà creata la singola porzione di corridoio (in figura 19 è possibile vedere un esempio di un pezzo di corridoio composto da 4 celle).

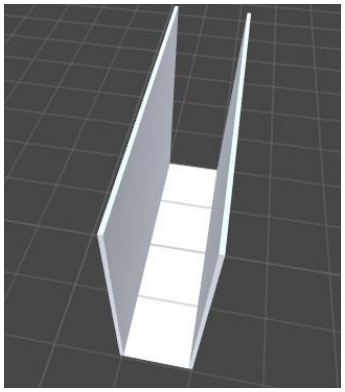

Figura 19. Una porzione di corridoio.

Come si è visto una parte molto rilevante del lavoro svolto in questa è dovuto alla distruzione dei di ostruzione presenti nei percorsi tra le stanze utilizzando la funzione `destroyWall`. La distruzione è resa possibile dalle API di Unity che consentono di deallocare gli oggetti presenti nello spazio: durante la costruzione dei corridoi le celle vengono memorizzate in un dizionario usando come chiave la coordinata spaziale, per cui nel momento in cui l'algoritmo rileva una situazione in cui c'è bisogno di questo tipo di deallocazione non fa altro che accedere al

dizionario con la coordinata in questione ottenendo alla cella, che a sua volta contiene anche il riferimento ai che si vuole distruggere.

2.6.4 Complessità computazionale

Come sempre bisogna cercare di porsi nel caso pessimo, che in questo caso occorre nell'ipotesi in cui ogni corridoio creato sia formato sempre da due segmenti e affinché questo possa succedere, la prima fase dell'algoritmo deve disporre le stanze in modo da formare una sequenza orientata come una retta a 45 gradi di inclinazione. Quindi l'algoritmo di connessione trovandosi un tale configurazione connetterà le stanze in modo sequenziale una dietro l'altra e la complessità risulterà essere la sommatoria nel numero di archi, che sarà pari ad $n - 1$, della somma della complessità computazione dovuta alla creazione dei due segmenti:

$$O\left(\sum_{i=1}^{n-1}(HorizontalDistance_i + VerticalDistance_i)\right)$$

Questa complessità è stata riportata solo ai fini della completezza, ma è molto difficile che ci si riesca a porre in questo caso e anche se si dovesse verificare, la complessità dell'intero algoritmo di generazione, asintoticamente, risulta sempre essere dovuta all'algoritmo di creazione del grafo RNG, pari ad $O(n^3)$.

Capitolo 3

In questo capitolo sarà fornita una vista ad alto livello della struttura del software sviluppato, specificando la funzione delle singole classi sviluppate.

3.1 Struttura del software

Da un punto di vista architetturale emerge che la struttura del codice sviluppato segue uno schema di tipo Model–View–Controller [17] (figura 20), infatti, è possibile dividere il generatore nei tre concetti principali sui cui si basa il pattern MVC.
La parte centrale del software, cioè quella in cui è stato implementato l'algoritmo di generazione, identificata dal *model*, è indipendente dall'interfaccia grafica e incapsula le strutture fondamentali utilizzate per la generazione come ad esempio la matrice `tileMatrix` ed è in grado di comunicare con la *view* fornendo i risultati di generazione.
La view in realtà non è stata implementata nel progetto poiché viene direttamente fornita da Unity, si utilizza infatti il pannello della scena che consente di visualizzare i risultati prodotti nello spazio tridimensionale.
Il controller è la parte che riceve in input i parametri e avvia il processo di generazione, che naturalmente corrisponde all'interfaccia grafica.

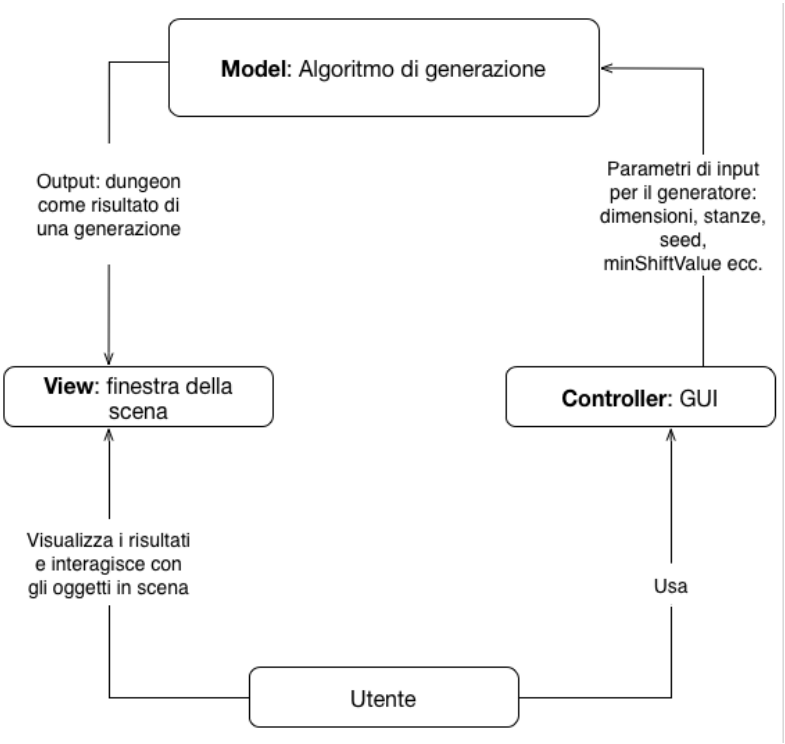

Figura 20. Pattern MVC del generatore.

Vediamo ora quali sono le classi che compongono il controller e il model. Per implementare il controller, cioè la GUI, è stata creata una singola classe C# `DungeonEditor.cs` nella quale è contenuta tutta l'implementazione dell'interfaccia grafica (la cui descrizione si trova nel capitolo 5).

Il model invece è composto da un totale di 9 classi, delle quali saranno descritte le più importanti, mettendo in luce le loro caratteristiche e funzionalità. Saranno prima descritte le classi più semplici e via via quelle più complesse fino ad arrivare a descrivere per ultima la classe più importante, che è `Dungeon.cs`, contenete l'algoritmo di generazione a tre fasi.

- `WallUnit.cs`: è una classe molto semplice ed uno dei componenti del prefab omonimo WallUnit (contenente il modello poligonale delle unità di muro), possiede anche il riferimento alla cella a cui appartiene.

- `DungeonCell.cs`: questa classe è un componente del prefab omonimo DungeonCell (contenente il modello poligonale delle unità di pavimento). Al suo interno vi è anche il riferimento a tutte le mura che può possedere nelle quattro direzioni polari.
- `Direction.cs`: è una classe statica e non viene associata come componente a nessun game object, ma serve come supporto alla rotazione delle unità di muro in modo che siano rivolte verso la direzione desiderata. Contiene la definizione delle direzioni polari come `enum`:

```
public enum Direction {
    North,
    East,
    South,
    West
}
```

che servono per indicizzare un array di rotazioni:

```
private static Quaternion[] rotations = {
        Quaternion.identity,
        Quaternion.Euler(0f, 90f, 0f),
        Quaternion.Euler(0f, 180f, 0f),
        Quaternion.Euler(0f, 270f, 0f)
    };
```

- `IntVector2.cs`: questa è una classe di supporto, infatti anche se il prodotto finale del generatore è presente nello spazio tridimensionale, gran parte dell'algoritmo di generazione si basa su operazioni in due dimensioni per cui questa classe è stata creata per fornire supporto da questo punto di vista. In particolare, Unity definisce la classe Vector3 adatta per il 3D, ma è poco adeguata per operazioni più semplici in due dimensioni.

- `DungeonRoom.cs`: questa classe modella il concetto di stanza ed è uno dei componenti del prefab omonimo DungeonRoom, che possiede anche le classi `DungeonCell.cs` e `WallUnit.cs` come componenti: una stanza infatti è un contenitore di unità di muro e di celle del pavimento. Contiene inoltre una lista di elementi di tipo `DungeonCell` che tiene traccia delle celle attive che compongono la stanza e il riferimento a un oggetto della classe `RoomData.cs` in cui saranno salvate tutte le proprietà di una stanza (come vedremo nel punto successivo). In questa classe si trova anche una funzione per generare i valori dimensionali in modo casuale all'interno degli intervalli di input ricevuti dalla dall'interfaccia.
- `Dungeon.cs`: questa classe è praticamente il cuore di tutto il sistema, utilizza quasi tutte le altre classi appena descritte e al suo interno vi è l'implementazione dell'algoritmo di generazione a tre fasi che viene avviato invocando il metodo `Generate`. Possiede i riferimenti a tutte le classi che globalmente compongono un dungeon, quali: `WallUnit`, `DungeonRoom` e `DungeonCell`. Per quest'ultima classe oltre ad avere un riferimento al tipo, ha anche una collezione di `DungeonCell` che viene popolata invocando il metodo di creazione di un singola stanza, che come già spiegato, restituisce la lista di celle del pavimento della stanza creata. La collezione di celle attive viene utilizzata dalla terza fase dell'algoritmo, quando si ha la necessità di rimuovere delle unità di muro i cui riferimenti sono contenuti nelle celle. Guardando i nomi dei metodi della figura 21, si può facilmente capire il loro scopo, ad esempio per la creazione dei segmenti orizzontali si utilizza il metodo `createHorizontalCorridor`, alcuni metodi però sono meno diretti, ad esempio `OnDrawGizmos`, questo viene eseguito dal sottosistema ad ogni aggiornamento di rendering ed ha il compito di visualizzare tutti quegli oggetti di tipo Gizmos, che sono in sostanza delle linee di overlay che non fanno parte del prodotto finale, ma sono utilizzate prevalmentemente in fase di sviluppo. Nel caso del generatore le linee di overlay sono quelle rosse che rappresentano il grafo di connessione dei centri delle stanze. In questa classe vi è anche l'implementazione dell'intera struttura `TileMatrix`.

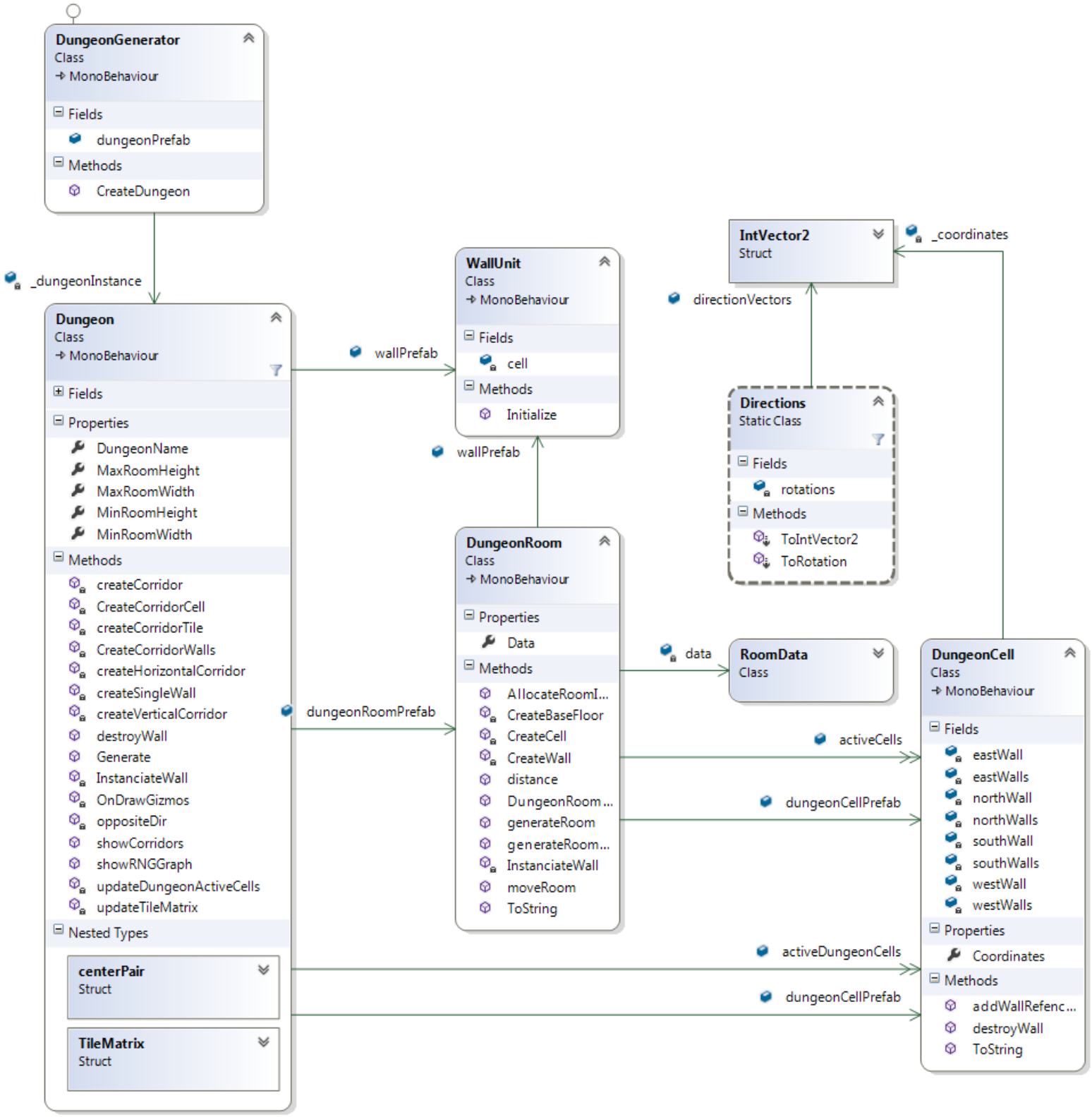

Figura 21. Diagramma delle classi del generatore.

Capitolo 4

4.1 Interfaccia grafica del generatore

L'interfaccia grafica del generatore è stata creata utilizzando le API messe a disposizione dal framework di Unity. Unity consente di creare nuove opzioni, come ad esempio nuove voci del menu principale tramite l'utilizzo dell'attributo `MenuItem` come decoratore per le funzioni statiche, che implementano le funzionalità dell'interfaccia grafica.
All'interno del file sorgente che implementa l'interfaccia sono state inserite le seguenti linee di codice (in C#) che consentono l'attivazione nel menu principale della voce "`Dungeon Generator Editor`":

```
1. [MenuItem("Window/Dungeon Generator Editor")]
2.     static void  ShowWindow ()
3.     {
4.             EditorWindow.GetWindow(typeof(DungeonEditor));
5.       }
```

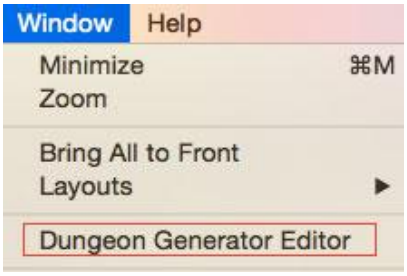

Figura 22. Accesso all'interfaccia dal menu Window.

La figura 22 mostra come è possibile avviare l'interfaccia.

Unity inoltre prevede che tutto il codice destinato ad implementare le nuove funzionalità di estensione dell'editor grafico devono trovarsi all'interno di una cartella specifica detta "Editor". In particolare in quest'ultima troviamo il file sorgente `DungeonEditor.cs` sviluppato in C#, in cui vi è tutto il codice necessario per realizzare l'estensione (figura 23).

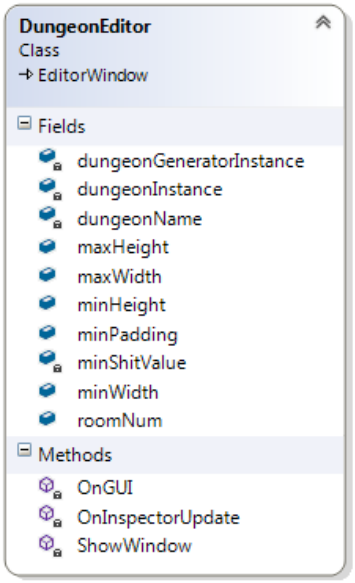

Figura 23. Diagramma della classe `DungeonEditor` per l'estensione dell'interfaccia di Unity.

Questa classe oltre a risiedere all'interno della cartella nominata `Editor` deve anche estendere la classe `EditorWindow`.

All'interno del metodo `OnGUI` si trova l'implementazione che gestisce i parametri di input e l'invocazione dell'algoritmo di generazione, memorizzato nella variabile `dungeonGeneratorInstance` (istanza della classe `DungeonGenerator`).
L'interfaccia del generatore si presenta con l'aspetto grafico della figura 24: sostanzialmente è un pannello che può essere spostato e posizionato all'interno di Unity e come si può notare presenta una corrispondenza diretta con i campi visibili nella figura 23:

1. Il nome del dungeon.
2. Il dungeon generator: l'oggetto prefab che contiene l'algoritmo di generazione.
3. Le dimensioni minime e massime delle stanze.
4. Il numero di stanze.
5. Il minimo valore di "shift", detto anche *minShiftValue*.
6. Una casella spuntabile che consente di inserire un intero corrispondente al seme di generazione, questo parametro è molto importante, permette di impostare il seme della classe `Random` di Unity, utilizzata durante tutto il processo di generazione. Se questo campo viene attivato (attraverso la spunta sulla casella) è possibile impostare il valore del seme con un intero qualunque, in modo da riprodurre lo stesso identico dungeon ad ogni iterazione (purché si mantengano intatti gli altri valori). In questo si possono ripresentare le stesse generazione con un singolo seme, inoltre questo parametro è stato di fondamentale importanza in fase di debugging, infatti riproducendo lo stesso risultato, venivano riprodotti anche gli stessi errori e questo ha semplificato la ricerca e la correzione degli errori. Se invece non viene inserita la spunta il seme stesso cambia in continuazione, ma nella finestra di debug di Unity, una volta terminata una generazione viene stampato il valore del seme utilizzato; nel caso in cui lo si voglia riutilizzare in futuro.
7. Subito dopo vi sono altre due caselle spuntabili, la prima permette di nascondere o visualizzare il reticolo del grafo RNG, mentre la seconda permette di nascondere o visualizzare i corridoi.

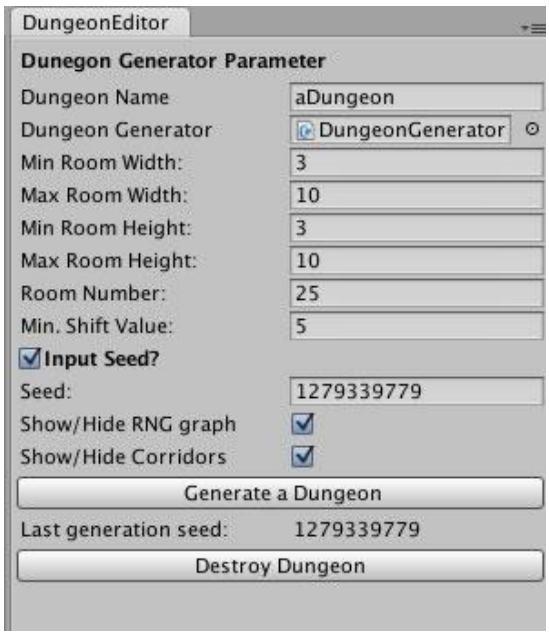

Figura 24. Interfaccia grafica del generatore

4.2 Installazione del generatore in Unity

Il generatore è stato sviluppato sotto licenza GPLv3 ed è possibile trovare l'intero progetto su github nel link seguente:

`https://github.com/stefanobettinelli/UnityPDG`

da cui è possibile scaricare un pacchetto in formato zip in locale, decomprimerlo ed ottenere una cartella di nome `UnityPDG-master`. Una volta avviato Unity, per avere l'estensione attiva nella propria area di lavoro, è sufficiente importare nella cartella degli asset del progetto corrente la cartella `UnityPDG-master`, così facendo nel menu `Window` comparirà la possibilità per aprire il pannello contenente l'interfaccia e quindi cominciare a generare dungeon.
Una volta generato un dunegon esso viene contenuto nella sua interezza, comprese stanze e corridoi, in un unico game object che prenderà il nome inserito nell'interfaccia (figura 25).

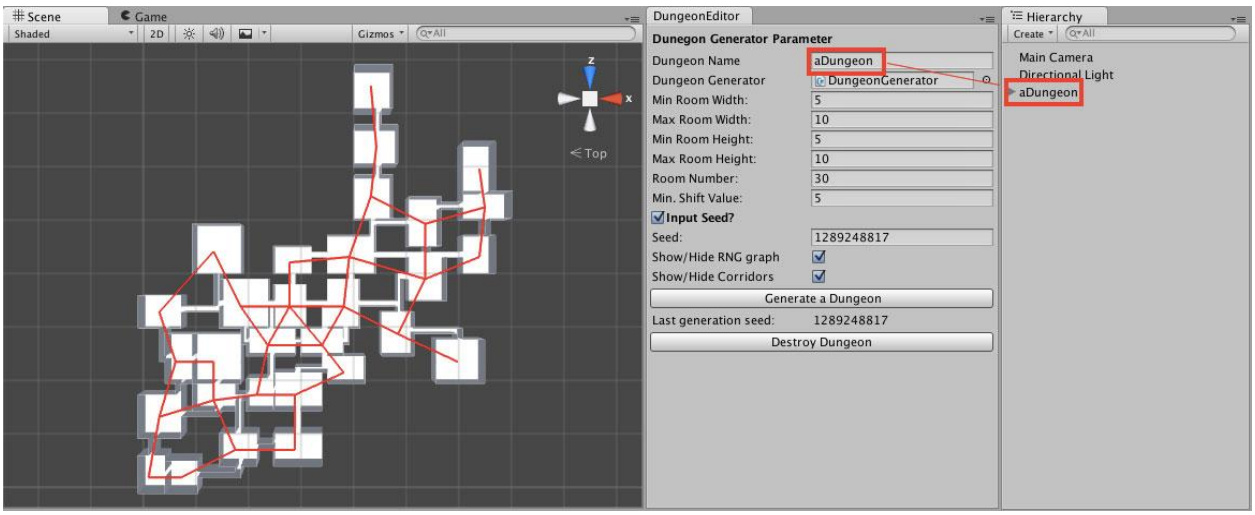

Figura 25. Il dungeon di nome "aDungeon" creato sarà contenuto in un game object omonimo.

In questo modo la struttura del dungeon generata è indipendente rispetto al gioco specifico che si sta sviluppando e i nuovi game object che saranno creati potranno sia essere figli del game object del dungeon oppure no, in base alle esigenze. Comunque la cosa fondamentale da comprendere è che una volta generato un dungeon, questo si comporta come un qualunque game object che si possa creare e/o importate con Unity.

Per verificare che le strutture generate consentissero l'esplorazione è stato utilizzato uno degli asset prefabbricati di Unity che permette di controllare un personaggio con una visuale in terza persona, in questo modo si può constatare oltre alla raggiungibilità delle stanze anche il corretto funzionamento delle collisioni tra la struttura e il personaggio, senza i quali si manifesterebbero comportamenti anomali, come ad esempio la caduta infinta nel vuoto del personaggio. Perciò quando si crea un dungeon e si vuole fare una prova di esplorazione basterà trascinare con il mouse il prefab di nome `ThirdPersonControllerPCG` nel pannello Hierarchy (figura 26).

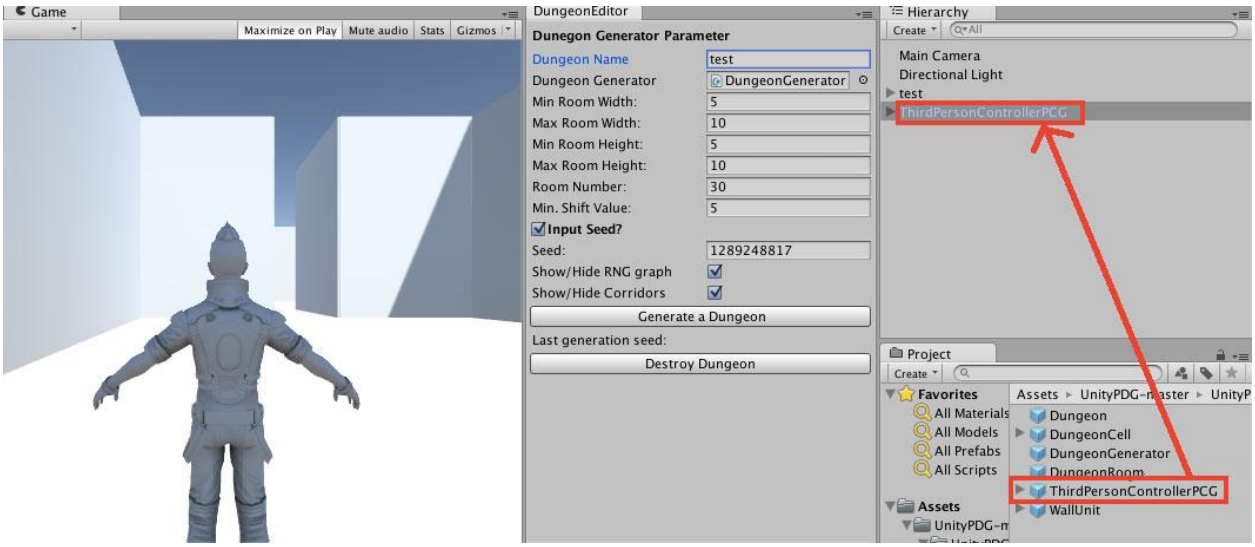

Figura 26. Per eseguire un test di esplorazione basta trascinare il prefab del personaggio con visuale in terza persona nel pannello Hierarchy.

Premendo il pulsante play nella barra degli strumenti viene avviato il gioco ed è possibile controllare il personaggio ed esplorare la struttura creata.

Nella figura 27 vengono mostrate alcune possibili generazioni con 50 stanze e con gli stessi valori di input delle dimensioni (minimo 3 massimo 10 per altezze e larghezze) e del seme (1278641459), ma variando in modo incrementale il valore di `minShiftValue` (2, 4, 6, 8, 10, 12). L'idea è di mostrare come questo valore influisca sulla compattezza del dungeon: a valori bassi le stanze sono più vicine mentre con valori alti lo spazio e la lunghezza dei corridoi aumenta.

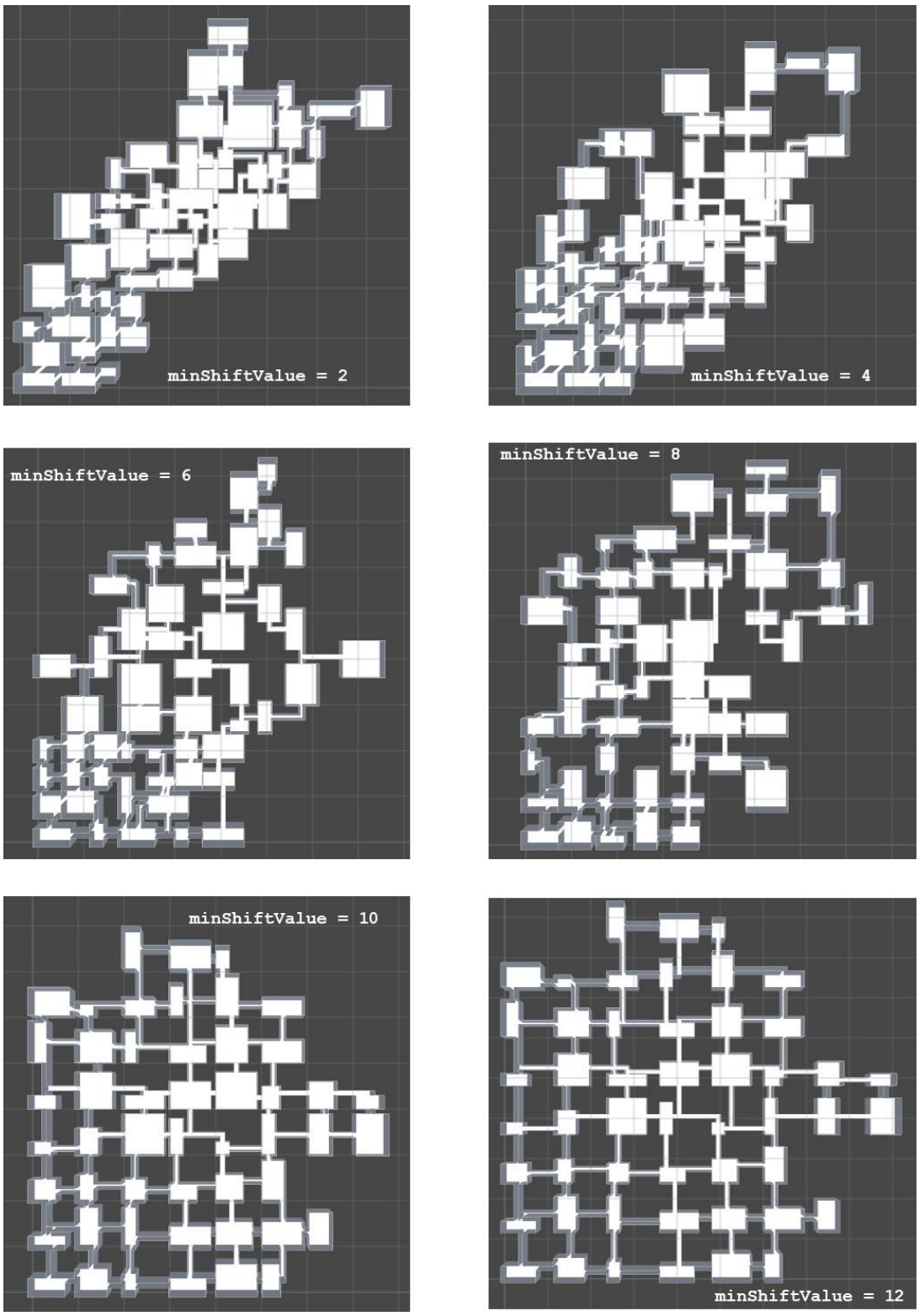

Figura 25. Un gruppo di 6 generazioni in cui varia solo il valore di minShiftValue.

Capitolo 5

5.1 Strumenti usati

Il progetto è stato sviluppato principalmente su un computer desktop con processore i5 2500k, scheda grafica gtx 970, 8GB di RAM e con sistema operativo Windows 7. Molti test sono stati condotti anche su un computer portatile con processore i5, scheda video integrata e 4GB di RAM. Anche se non è stato condotto uno studio specifico per misurare i tempi di generazione, è possibile affermare che le prestazioni sul computer fisso sono risultate chiaramente superiori rispetto a quelle del computer portatile, ma la differenza è stata minima. Ad esempio generando un dungeon con le stesse configurazioni di input, con 100 stanze, il computer desktop è risultato più veloce di 3 centesimi di secondi rispetto al portatile, terminando la computazione in 1.03 secondi contro gli 1.06 secondi del portatile.
Per lo sviluppo del codice (oltre a Unity chiaramente) è stato utilizzato Visualstudio 2013 come ambiente di sviluppo, mentre come strumento di versionamento è stato utilizzato GIT con il supporto di github come repository remoto.

5.2 Conclusione e sviluppi futuri

Lo studio e lo sviluppo condotto hanno dato la possibilità di dimostrare che strumenti quali i generatori produrali di contenuti possono essere facilmente integrabili all'intero di un game engine qualunque, purché questo fornisca, come nel caso di Unity, una struttura sottostante flessibile ed espandibile.
Nello specifico il generatore sviluppato riesce a produrre livelli in modo ragionevolmente veloce ed è molto facile da utilizzare. Può quindi potenzialmente entrare a far parte del flusso di lavoro di un level designer, consentendogli di ridurre i tempi di sviluppo e fornendogli strutture generate in modo automatico, pronte per essere modificate e migliorate a seconda del bisogno.
Le scelte implementative dell'algoritmo di generazione risultano essere corrette, i dungeon infatti sono esplorabili nella loro interezza, le connessioni tra le stanze per mezzo del grafo RNG producono reticoli di collegamenti che una volta tradotti nei corridoi diventano interessanti da un punto di vista esplorativo, che è uno degli aspetti più importanti del lavoro svolto.
Il progetto comunque offre molti spunti per essere espanso e sviluppato ulteriormente:

- Uno degli aspetti che non è stato trattato è la valutazione di aspetti qualitativi dei dungeon, stabilendo a priori determinate misurazioni da rilevare le simulazione potrebbero mostrare tendenze interessanti ed inaspettate.
- Potrebbe essere interessante modificare l'algoritmo nella seconda fase, quella di creazione del grafo, ideando metodi alternativi per collegare le stanze e poi confrontarle con quelle ottenute dal grafo RNG. Lo stesso discorso vale anche per la prima fase, ideando un nuovo metodo per la disposizione delle stanze nello spazio.
- L'implementazione e l'integrazione di alcune nuove funzionalità all'interno dell'interfaccia potrebbero essere molto desiderabili, come ad esempio un metodo per importare texture da applicare ai pavimenti e/o alle unità di muro. Oppure funzionalità automatizzate per popolare le strutture prodotte con elementi tipici dei dungeon.

- Infine modificare il generatore in modo da essere in grado di funzionare anche in modalità online: mentre il gioco è in esecuzione vengono dinamicamente generate ed aggiunte strutture di stanze interconnesse a quelle già esistenti.

Bibliografia

1. Shaker N., Liapis A., Togelius J., Lopes R., Bidarra R. PCG Book: Chapter 3 Constructive generation methods for dungeons and levels (DRAFT)

2. Togelius J., Kastbjerg E., Schedl D., Yannakakis G.N.: What is procedural content generation?: Mario on the borderline. In: Proceedings of the 2nd Workshop on Procedural Content Generation in Games (2011)

3. Baghdadi W., Shams Eddin F., Al-Omari R., Alhalawani Z., Shaker M., Shaker N.: A Procedural Method for Automatic Generation of Spelunky Levels

4. Doran J., Parberry I.,Dept. of Comput. Sci. & Eng., Univ. of North Texas, Denton, TX, USA: Controlled Procedural Terrain Generation Using Software Agents

5. Johnson L., Yannakakis G.N., Togelius J.: Cellular Automata for Real-time Generation of Infinite Cave Levels. In: Proceedings of the ACM Foundations of Digital Games. ACM Press (2010)

6. Adams D.: Automatic generation of dungeons for computer games (2002). B.Sc. thesis, University of Sheffield, UK

7. Dormans J.: Adventures in level design: generating missions and spaces for action adventure games. In: PCG'10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games, pp. 1:1–1:8. ACM (2010).

8. Linden R.v.d., Lopes R., Bidarra R.: Designing procedurally generated levels. In: Proceedings of IDPv2 2013 - Workshop on Artificial Intelligence in the Game Design Process, co-located with the Ninth AAAI Conference on Artificial Intelligence in Interactive Digital Entertainment, pp. 41–47. AAAI, AAAI Press, AAAI Press, Palo Alto, CA (2013). ISBN 978-1-57735-635-6

9. Rogue (Video Game): http://en.wikipedia.org/wiki/Rogue_(video_game)

10. Speed Tree: http://en.wikipedia.org/wiki/SpeedTree

11. Toussaint G.: The Relative Neighborhood Graph of a Finite Planar Set

12. Now You're Thinking With Components: http://gamedevelopment.tutsplus.com/articles/unity-now-youre-thinking-with-components--gamedev-12492

13. Smith G., Whitehead J.: Analyzing the Expressive Range of a Level Generator (2010)

14. Liapis A., Yannakakis G. N., Togelius J.: Towards a Generic Method of Evaluating Game Levels (2013)
15. Togelius J., Yannakakis G. N., Stanley K. O., Browne C.: Search-Based Procedural Content Generation: A Taxonomy and Survey (2010)
16. Smith,G., Whitehead,J. :Analyzing the expressive range of a level generator. In:Proceedings of the 2010 Workshop on Procedural Content Generation in Games, p. 4. ACM (2010)
17. Model–view–controller: http://en.wikipedia.org/wiki/Model-view-controller