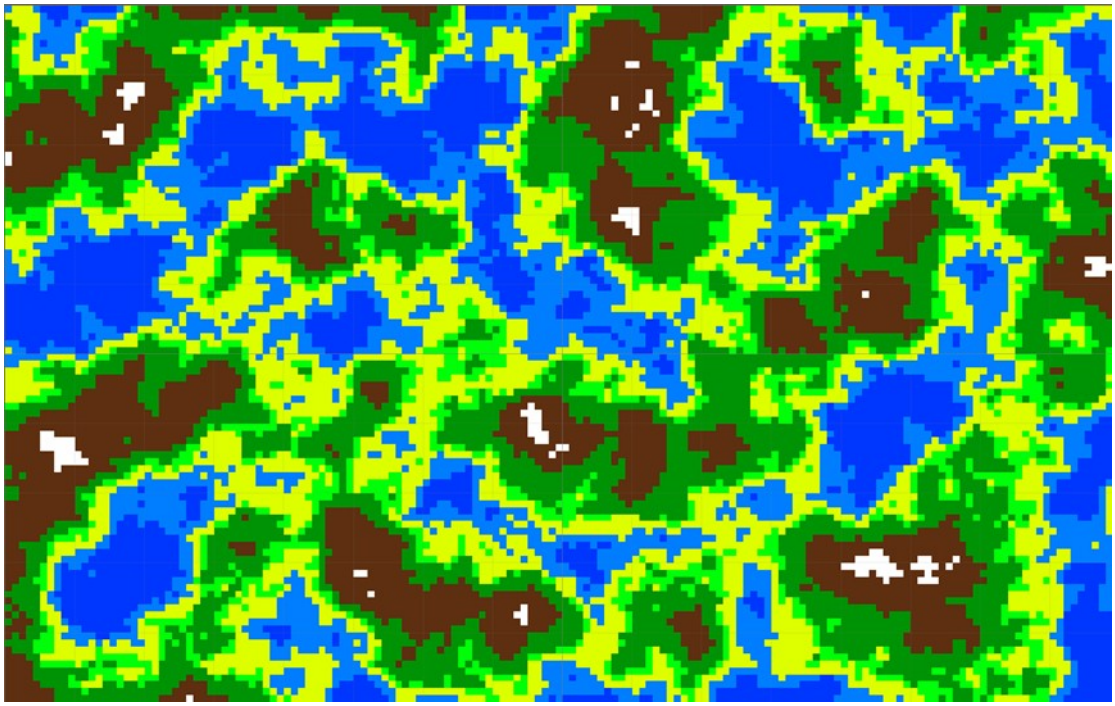
Lavoro per il concorso di Scienza e Gioventù

Mappinator



Ricerca sulle tecniche per la generazione procedurale di mappe

Gregorio Silvestri

Liceo Cantonale Lugano 1

Docente: prof. Amedeo Mazzoleni

Indice

1. Premessa e Abstract

L’idea per la realizzazione di questo lavoro mi è venuta ascoltando la conferenza di Helbert Wolverson al RogueLike Festival del 2020[1]. Durante il suo intervento Wolverson ha presentato diverse tecniche per la generazione procedurale di mappe per i videogiochi, un argomento che ho trovato molto interessante. Per questo motivo ho deciso di affrontarlo in prima persona, scegliendo cinque di queste tecniche ed implementandole nel motore di gioco Unity, in modo da comprenderne il funzionamento e i relativi vantaggi e svantaggi. Mi auguro che la lettura di questo lavoro sia interessante e dia la possibilità al lettore di approfondire questo tema, molto attuale ed importante dell’industria videoludica. Infatti, benché la generazione procedurale sia presente da quasi mezzo secolo, il repertorio di tecniche per ottenere risultati sempre più soddisfacenti è in costante aumento. Basti pensare che la tecnica più nuova, la *Wave Function Collapse*, è stata inventata solo nel 2019, e che ancora oggi moltissimi giochi utilizzano gli algoritmi che andrò ad implementare.
Ringrazio il professor Amedeo Mazzoleni e Daniele Mapletti per il loro prezioso aiuto durante lo svolgimento di questo lavoro ed auguro una piacevole lettura.

2. Introduzione

Con la realizzazione di questo lavoro ho intenzione di approfondire il campo della generazione procedurale di contenuti, in particolare delle mappe per i videogiochi. Con generazione procedurale, si intende la creazione di contenuti digitali automatizzata, dunque vengono creati dei dati in modo algoritmico piuttosto che manuale. Il vantaggio di questo approccio sta nella quantità di contenuti che è possibile generare, e le mappe per i videogiochi ne sono un ottimo esempio. Creando manualmente una o poche mappe il giocatore avrà la stessa esperienza ad ogni partita giocata, almeno per ciò che riguarda l'esplorazione del territorio in cui si muove. Utilizzando invece un algoritmo di generazione procedurale, il giocatore potrà ottenere una mappa diversa ad ogni partita, aumentando il fattore di rigiocabilità. Per questo motivo la generazione procedurale di mappe è stata impiegata nell'industria fin dai primi tempi: ne sono un esempio "Rogue" del 1980 o "Hack" del 1982[2]. L'uso di queste tecniche è continuato fino ai giorni nostri, e sono tutt'ora utilizzate per creare mondi pressoché infiniti, sia per quanto riguarda le dimensioni del mondo stesso, sia nel numero di mappe che è possibile generare. Due buoni esempi di ciò sono "No man's sky" del 2016 e "Minecraft" del 2011.
Il mio obbiettivo in questo lavoro è duplice: inizialmente quello di riuscire ad implementare cinque tecniche di generazione procedurale nel motore di gioco *Unity*, e in seguito quello di confrontare i risultati ottenuti dall'impiego di ogni tecnica per determinarne i relativi vantaggi e svantaggi.
Le cinque tecniche che ho scelto sono:

- *Simple Room Placement* e *Binary Space Partition* per la generazione di mappe in stile *dungeon.*
- *Drunkard's Walk* e *Cellular Automata* per la creazione di mappe simili a caverne.
- *Perlin Noise* per realizzare mappe di ampi spazi aperti in stile topografico.

Il lavoro sarà quindi strutturato in due parti. La prima parte sarà una spiegazione dell'idea alla base ed il modo in cui ho implementato ogni tecnica, evidenziando le difficoltà pratiche sorte durante la scrittura del codice. La seconda parte sarà un'analisi atta a mettere in luce le potenzialità ed i difetti di tutte le tecniche, per poi compararle tra loro e dare la mia opinione su quali siano le più adatte a seconda della situazione e degli obbiettivi che si desidera raggiungere con il loro impiego.

3. Materiali e metodologie

Lo scopo di questa sezione è quello di illustrare i materiali ed i metodi utilizzati per raggiungere i risultati ottenuti, in modo tale da rendere l'implementazione qui presentata ripetibile e verificabile, ottenendo gli stessi risultati alle stesse prestazioni. Inoltre questa sezione vuole costituire un'utile fonte di informazioni per il confronto tra le varie tecniche prese in analisi.

3.1 Materiali

Essendo questo un lavoro nell'ambito delle scienze informatiche, dividerò i materiali utilizzati in due categorie, software e hardware. L'hardware che ho impiegato durante la programmazione non è stato il Rasberry Pi 4 in versione 8Gb fornito dalla scuola, ma il mio portatile personale, provvisto di processore Intel i7-4702MQ a 3200Mhz, grafica integrata Radeon HD 8670A e 16Gb di memoria Ram DDR3 a 1600 Mhz. Il motivo di questa scelta è stata la maggiore portabilità del mio dispositivo personale e la sua maggiore potenza in comparazione al RasberryPi. Dal lato software ho utilizzato il sistema operativo Linux Mint 20.3 e il motore di gioco Unity Editor 2021.3.4.f1.

3.2 Metodologie

Tutte le tecniche sono state implementate in un progetto Unity 2D, ad eccezione di *Perlin Noise*, nel cui caso ho preferito iniziare il progetto in un ambiente 3D, essendo possibile espandere questa implementazione in futuro, utilizzando il programma per generare mondi in tre dimensioni.
Per spiegare il codice scritto presuppongo che il lettore lo stia leggendo in contemporanea a di questo commento. Al seguente link ho reso disponibile il codice sorgente dell'intero lavoro: https://github.com/gregghy/lam, le diverse tecniche separate nei vari *branch* della pagina *Github.*
Il motivo di questa scelta stilistica è stato il fatto che ritengo abbia più senso per il lettore poter analizzare liberamente tutto il codice nel suo insieme, piuttosto che doversi limitare al solo pezzo di codice inserito nel lavoro. Inoltre il codice disponibile su *Github* è scaricabile, ed apribile come progetto di *Unity* sulla propria macchina locale, in modo tale da poterlo testare e modificare liberamente.

3.2.1 Metodologia – Simple Room Placement

L'idea dietro a questa tecnica è piuttosto semplice. In un certo spazio vengono generate in modo randomico le posizioni per le stanze. Se la posizione corrente coincide (interamente o in parte) con la posizione di un'altra stanza già generata, il programma ne sceglierà una nuova, ripetendo il questo ciclo fino a quando non genererà l'ultima stanza. A questo punto utilizzerà un algoritmo di *pathfinding* per collegare tutte le stanze tramite dei corridoi.

Per prima cosa quindi, definiamo lo spazio che andrà riempito con muri, stanze e corridoi, la mappa vera e propria. Con un valore in altezza ed uno in larghezza, possiamo immaginare questo spazio come un piano cartesiano, in cui ogni punto ha un valore *x* in larghezza e un valore *y* in altezza.

Creiamo il vettore 2D *map*, che definisce ogni punto della nostra mappa. Assegniamo a *map* in ogni punto della mappa due possibili stati: 0 o 1. Tramite la funzione di Unity *OnDrawGizmos* coloriamo di nero i punti in cui *map* ha valore 1 (simboleggiando gli spazi pieni come i muri), e di bianco i punti in cui *map* ha valore 0 (dove abbiamo uno spazio vuoto, come stanze o corridoi).

Ritenendo che sia più facile lavorare per sottrazione, ovvero ritagliare gli spazi vuoti da uno spazio completamente pieno, ho definito inizialmente la funzione *FillMap,* che ha come scopo quello di assegnare a tutti i punti un valore di 1, in questo modo otteniamo di riempire completamente la mappa.

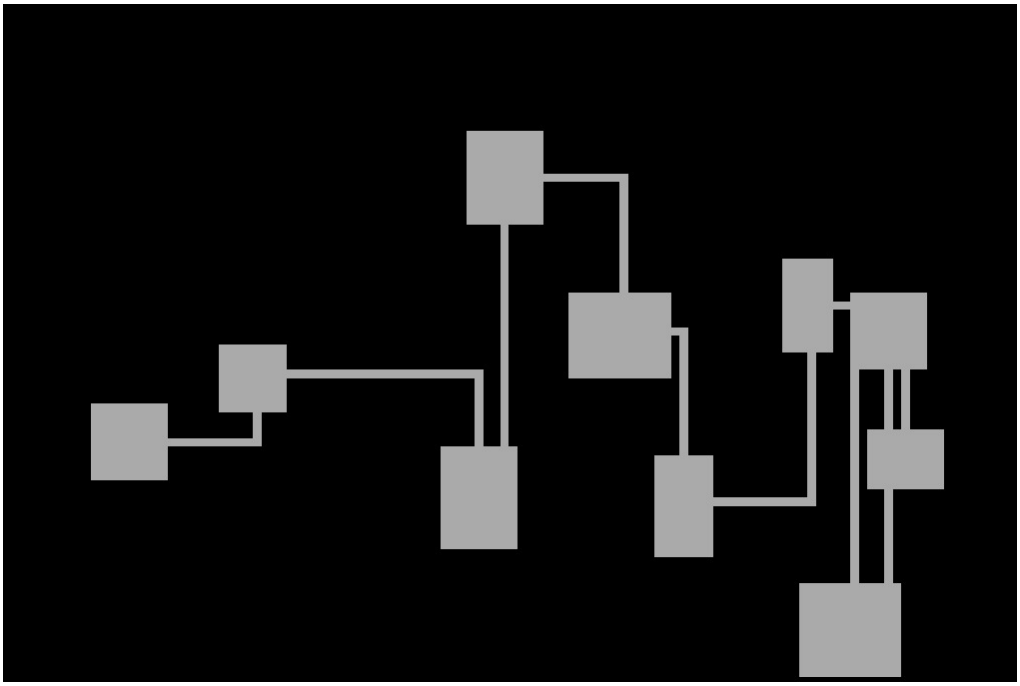
Ora dobbiamo ritagliare lo spazio delle stanze, per fare ciò ho creato prima la funzione *FindASpot,* il cui scopo è quello di stabilire la posizione dei quattro angoli di una stanza in modo randomico, e poi la funzione *GenerateRoom,* che utilizza le coordinate calcolate da *FindASpot* per creare la stanza.

Per rendere le stanze delle dimensioni desiderate basta inserire dei *range* all'interno dei quali vengono selezionati i valori degli angoli. Dopo aver fatto ciò la funzione controlla che la posizione creata collegando i quattro angoli non coincida con quella di una stanza generata precedentemente. Per fare questo basta limitarsi a controllare lo stato di *map* nei punti scelti: se lo stato di *map* è già pari a 0 significa che è già presenta una stanza in quella posizione.

Se la posizione scelta è corretta, tramite *GenerateRoom* viene cambiato lo stato di tutti i punti compresi tra i lati della stanza, rendendola definitiva, e viene inoltre marcato il centro della stanza tramite un nuovo vettore, *roomPos*. Questo ci servirà per la generazione dei corridoi tra le stanze.

La funzione viene iterata per il numero di stanze che si desidera creare, e una volta finito il ciclo si passa alla generazione dei percorsi, realizzati dalla funzione *CreatePath.* Questa funzione raccoglie in una lista chiamata *xpos*

tutti i valori sull'asse delle ascisse dei punti marcati da *roomPos* come centro di una stanza in ordine crescente, ed inserisce nella lista *ypos* i relativi valori in altezza. In questo modo i punti vengono ordinati facendo sì che il primo punto sia quello più vicino al confine sinistro della mappa, mentre l'ultimo il più lontano. L'ordine stabilito nel caso in cui due punti abbiano il medesimo valore *x,* è quello di inserire prima il punto con *l'y* minore. Per fare un esempio, se assumiamo che un punto P si trovi sulle coordinate [x, y], un punto L su [x1, y] e un punto M su [x, y1], con x1 > x e y1 > y, l'ordine della lista sarà [P, M, L]. Una volta ordinata la lista, la funzione unisce le stanze, percorrendo la distanza tra i punti presenti nella lista prima in larghezza e poi in altezza, in una forma ad angolo retto, cambiando lo stato di *map* dei punti percorsi in 0.
Questo algoritmo è piuttosto semplice, pur essendo ben adatto allo scopo. Infatti permette di generare delle mappe con una struttura lineare (percorribili da sinistra verso destra), ma ogni tanto crea anche situazioni più interessanti di una semplice fila ordinata, grazie alla posizione randomica delle stanze.
Una volta implementata l'idea di base ho aggiunto qualche dettaglio che migliorasse il risultato finale. Ho inserito delle dimensioni minime e massime delle stanze all'interno della mappa, dal momento che stanze sproporzionate possono rovinare l'esperienza di gioco; ho esteso il controllo dello spazio occupato da una stanza precedente per la creazione di una nuova, in modo tale che venga sempre lasciato un margine vuoto tra due stanze, che altrimenti si unirebbero risultando un unico spazio.
Di seguito è riportata l'immagine di una mappa ottenuta.

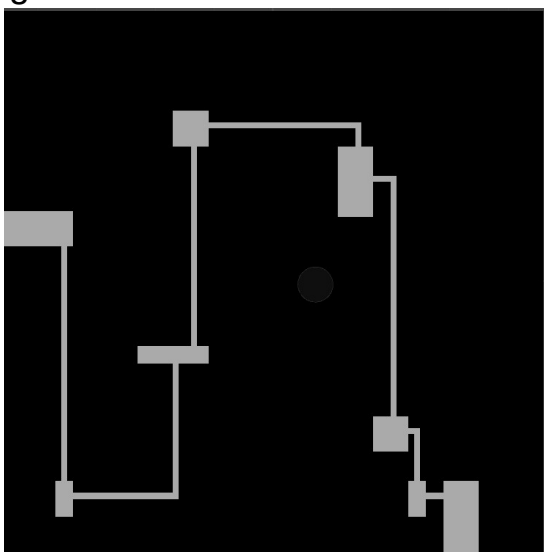


3.2.2 Metodologia – Binary Space Partition

Questa tecnica prevede di dividere a metà lo spazio della mappa un numero di volte sufficiente a raggiungere le dimensioni desiderate per una stanza. La direzione in cui dividere la stanza ad ogni iterazione è scelta randomicamente, e questo permette di ottenere mappe diverse ogni volta. Il risultato di questo procedimento dovrebbe essere quello di ottenere una struttura più ordinata rispetto alla tecnica precedente.

Per implementare questa tecnica ho utilizzato molto del codice scritto per *Simple Room Placement*, in quanto queste due tecniche servono a raggiungere lo stesso tipo di mappa. Dopo aver riempito tutto lo spazio della mappa con la funzione *FillMap* tale quale alla tecnica precedente, assegniamo a quattro variabili i valori degli angoli della mappa, corrispondenti ai punti [1, 1], [width, height], [width, 1] e [1, height]. Poi peschiamo un numero tra 0 e 1: se otteniamo 0 divideremo la stanza orizzontalmente, se invece viene scelto 1 si divide verticalmente. La divisione viene effettuata dimezzando il valore delle variabili che hanno assegnati i valori degli angoli. Una volta iterato questo processo il numero di volte necessarie ad ottenere le dimensioni della stanza desiderate, si passa alla prossima. Una volta create tutte le stanze possiamo procedere a collegarle; per farlo ho riutilizzato lo stesso algoritmo di *pathfinding* spiegato del punto 3.2.1.

Ho avuto qualche difficoltà in questo caso ad ottenere i risultati sperati, perché le stanze uscivano spesso molto sproporzionate, allungate in modo estremo in una direzione o nell'altra. Per mantenere la forma che desideravo, ho inserito un'ulteriore condizione nella decisione della direzione in cui dividere lo spazio. Nel caso in cui le proporzioni di una stanza risultino pari o superiori ad ½ in altezza o larghezza, la divisione successiva deve necessariamente essere svolta nell'altro senso. In questo modo sono stato in grado di ottenere delle stanze dalle proporzioni bilanciate, come quelle che si possono trovare nel seguente risultato finale.



3.2.3 Metodologia – Drunkard’s Walk

Drunkard’s Walk e *Cellular Automata* sono le tecniche che ho scelto per generare delle mappe che simulassero un ambiente chiuso ma naturale, come grotte e caverne. Essendo sempre un ambiente chiuso come le due precedenti tecniche, ricreiamo la mappa sulla falsa riga delle due tecniche precedente, con il vettore *map* per assegnare i due valori ai punti e la funzione *OnDrawGizmos* per colorarli.

Drunkard’s Walk prevede di posizionare al centro della mappa completamente piena (quindi il caso in cui tutte le caselle hanno il valore pari a 1, che otteniamo con la solita funzione *FillMap*), un punto con stato 0. Ad ogni iterazione il punto si sposta in modo randomico in una delle quattro direzioni (sopra, sotto, destra e sinistra), cambiando il valore della posizione in cui si è mosso in 0. Il punto può essere fermato in due casi: quando viene superato il numero di iterazioni indicato precedentemente, oppure quando raggiunge il bordo della mappa. Il modo più semplice per dare queste condizioni è inserire il codice del movimento in un ciclo *while*. Finché le condizioni per l’interruzione non vengono soddisfatte il punto continuerà a muoversi.

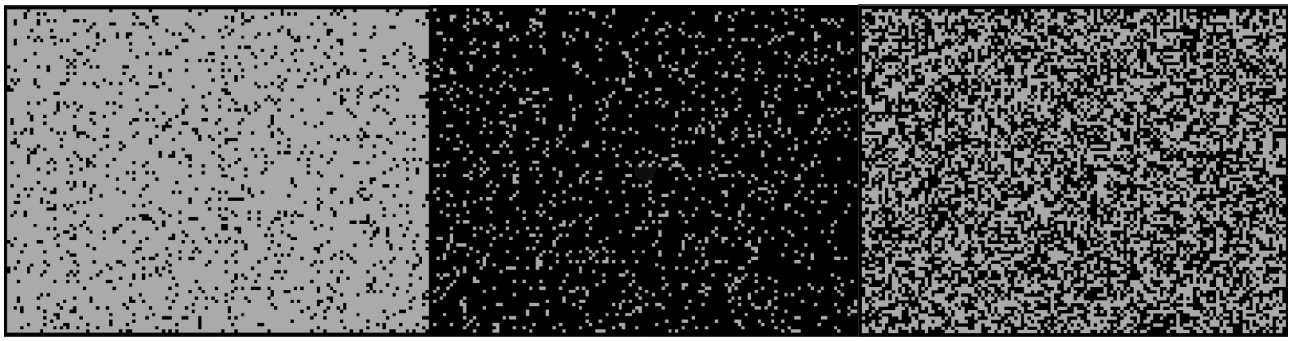
Il risultato che otteniamo è uno spazio vuoto e continuo, dalla forma squadrata e con più bracci, ma nel complesso la caverna avrà un aspetto totalmente casuale e dalle forme e dimensioni più disparate. Riporto qui sotto due mappe come esempi della differenza di dimensioni che si può ottenere.

3.2.4 Metodologia – Cellular Automata

Questa è una delle tecniche più interessanti, e il risultato che ci permette di ottenere è molto utile per la creazione di mappe, ma può sicuramente essere utilizzata anche per la generazione di altri contenuti. L'idea alla base è quella di sfruttare delle singole entità governate da regole semplici per ottenere dei comportamenti complessi nell'insieme; creiamo una mappa di gioco un po' come api e formiche costruiscono alveari e formicai. Per farlo utilizzeremo lo stesso concetto del celebre gioco *The Game of Life* ideato dal matematico John Horton Conway [3]. Questo gioco si basa sull'idea che avendo una griglia si possano riempire le caselle per rappresentare delle cellule. Queste vanno disegnate seguendo 4 semplici regole: ogni cellula viva con meno di due cellule vive vicine muore, ogni cellula viva con due o tre cellule vive vicine sopravvive fino alla prossima generazione, ogni cellula viva con tre o più cellule vive vicine muore, ogni cellula morta con esattamente tre cellule vive vicine diventa viva. Il risultato sarà quello di creare delle strutture composte dalle singole cellule, che nascono e muoiono in base alle regole del gioco.

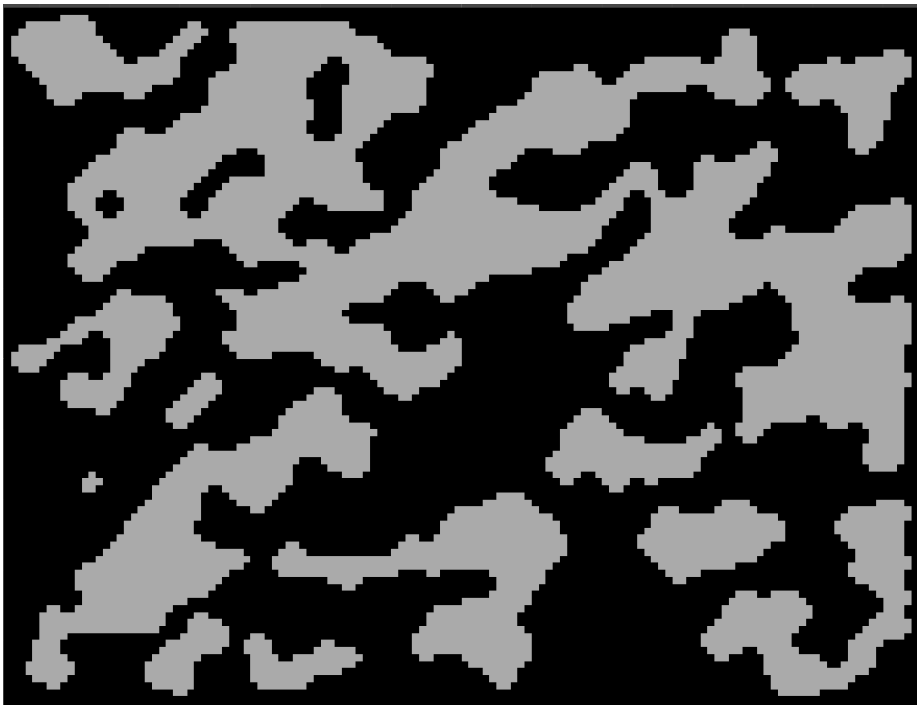
La nostra mappa sarà quindi creata in uno spazio composto da caselle che possono avere valore 0 e 1, come nelle implementazioni precedenti. Tuttavia invece di riempire tutta la mappa e in seguito ritagliare gli spazi vuoti come abbiamo fatto precedentemente, stavolta assegneremo ad ogni casella della mappa valore random di 0 o 1, tramite la funzione *RandomFillMap*. Il numero di caselle da riempire è determinato dalla variabile *RandomFillPercent.* Più è alto il valore di questa variabile, maggiore sarà il riempimento iniziale della mappa.

A seguito sono mostrati i risultati del riempimento effettuato prima con un valore di *RandomFillPercent* pari a 10, poi a 90 e infine al valore ideale, intorno a 50.



Per ottenere questo risultato, passiamo in rassegna ogni punto della mappa, i bordi verranno assegnati come 1, mentre il valore delle altre caselle verrà assegnato in modo *pseudo-random*, ovvero randomico dal punto di vista statistico, ma assegnato in modo deterministico e riproducibile. Possiamo ottenere questa caratteristica utilizzando un *seed,* ovvero un numero che può venire inserito o essere generato casualmente e che permette di ottenere una

mappa diversa da ogni altro *seed*, ma sempre identica quando viene utilizzato lo stesso. Questo ha il vantaggio di creare mappe che siano riproducibili, pur essendo in numero infinito.
A questo punto ogni cellula avrà un valore di 0 o 1, viva o morta nel gioco di Conway, muro o vuoto nella nostra mappa. Ora dobbiamo simulare le generazioni delle cellule, che determineranno quali sopravvivranno e quali moriranno secondo le regole del gioco. In questo caso le regole sono molto semplici e vengono definite nella funzione *SmoothMap:* se una cellula ha più di quattro vicini in vita muore, altrimenti vive. Sono giunto a questo bilanciamento delle regole prendendo spunto da altre implementazioni di *Cellular Automata* [4], ma è possibile regolarsi come si desidera facendo delle prove. Il numero di cellule vive nelle vicinanze di ogni singola cellula viene calcolato dalla funzione *GetSurroundingWallCount,* la quale analizza la griglia 9x9 intorno ad ogni cellula e restituisce il numero di celle in vita all'interno della griglia a *SmoothMap.* Eseguendo la funzione per un certo numero di iterazioni, le quali rappresentano le generazioni di cellule che nascono e muoiono, il risultato sarà quello di ottenere delle zone di vuoto nella mappa, dalla forma molto tondeggiante e naturale, solitamente divise in più caverne indipendenti tra loro.
Di seguito è riportata una mappa ottenuta come esempio.



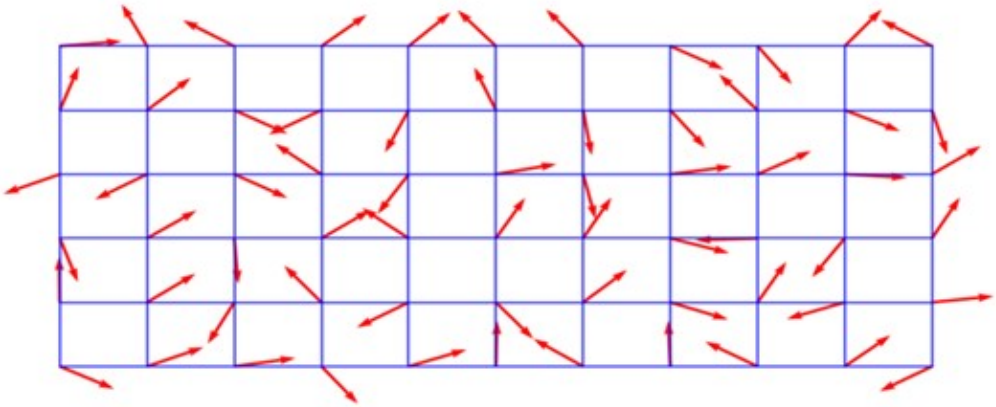
3.2.5 Metodologia – Perlin Noise

L'ultima tecnica è la più complessa e articolata, per organizzare al meglio il progetto ho infatti diviso il codice su più *files*, tutti trovabili nella cartella *Assets* del progetto.

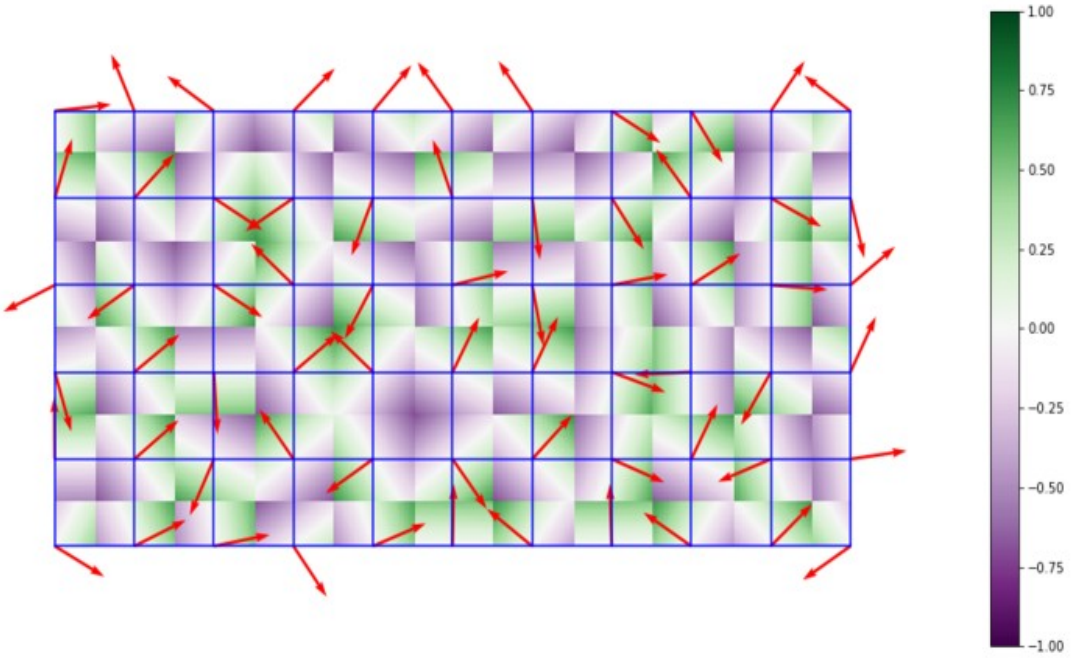
Questa tecnica utilizza l'algoritmo *Perlin Noise* (da cui appunto prende il nome), sviluppato dal *computer scientist* Kenneth Perlin [5]. L'algoritmo *Perlin Noise* non è utilizzato solo nella generazione di mappe, ma è popolare anche per la generazione di *textures* nel cinema e nei videogiochi. Il funzionamento dell'algoritmo si divide in tre fasi.

Nella prima viene creata una griglia in cui ad ogni angolo di una casella viene assegnato un vettore gradiente.

[https://en.wikipedia.org/wiki/Perlin_noise#/media/File:PerlinNoiseGradientGrid.png]

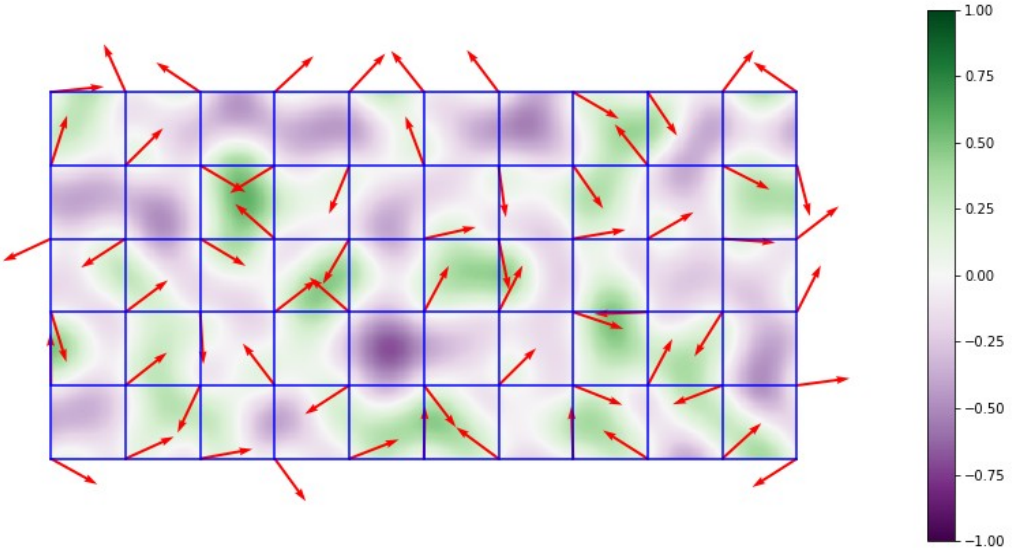


Poi si prende un punto all'interno della casella e viene calcolato il prodotto scalare tra il vettore gradiente e il vettore tra il punto scelto ed il rispettivo angolo.



[https://en.wikipedia.org/wiki/Perlin_noise#/media/File:PerlinNoiseDotProducts.png]

Viene poi fatta un'interpolazione dei punti calcolati, ottenendo come risultato finale la mappa caratteristica di Perlin Noise.



[https://en.wikipedia.org/wiki/Perlin_noise#/media/File:PerlinNoiseInterpolated.png]

L'algoritmo *Perlin Noise* è fortunatamente già implementato in *Unity*, all'interno della libreria *Mathf*, essendo questa una tecnica molto conosciuta ed utilizzata. La funzione *PerlinNoise* [6] ci permette infatti, dati due valori *x* e *y*, di ottenere un valore compreso tra zero e uno. Questi valori sono i dati che vanno a costruire la nostra mappa: colorando i vari punti secondo il proprio valore in un gradiente dal nero al bianco, dove i valori pari a 0 vengono colorati di nero e i valori pari a 1 vengono colorati di bianco, otteniamo una mappa come questa.

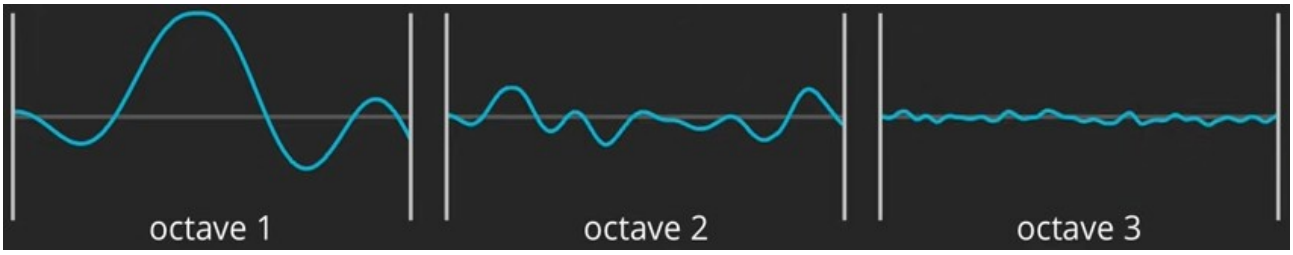
[https://en.wikipedia.org/wiki/Perlin_noise#/media/File:Perlin_noise_example.png]

Immaginando la crosta terrestre, possiamo dire che essa non è perfettamente liscia e piatta, ma consiste in un continuo alternarsi di numerosi solchi e cime. Utilizzando *PerlinNoise* otteniamo un valore associato ad ogni punto della mappa, possiamo quindi interpretare quel valore come l'altezza del terreno in quelle determinate coordinate, otteniamo come risultato che i punti con valore 0 siano i più bassi, mentre quelli con valore 1 i più alti. In base a questa interpretazione, possiamo utilizzare i punti più bassi come profondità marine, e salendo arrivare gradualmente fino alle cime dei monti.

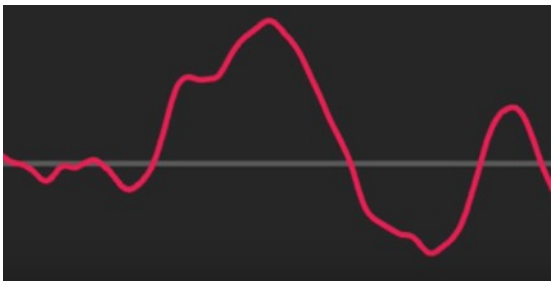
Come prima cosa nel file *Noise* ho creato la funzione che ci restituirà la *NoiseMap*. La funzione *GenerateNoiseMap* richiede diversi argomenti, i quali sono utilizzati per determinare alcune caratteristiche della nostra mappa. *MapWidth* e *MapHeight* come nelle tecniche precedenti rappresentano

rispettivamente la larghezza e l'altezza delle nostre mappe 2D. Il *seed* è un numero generato randomicamente, e serve a rendere le nostre mappe infinite nel numero, ma riproducibili in modo totalmente identico utilizzando uno stesso *seed*, come fatto per l'implementazione di *Cellular Automata*. Le *octaves* sono un trucchetto che ci permette di incrementare l'illusione della nostra mappa di sembrare il più naturale e realistica possibile. Se immaginiamo di tagliare lateralmente la nostra mappa possiamo ottenere un'onda che rappresenta il nostro terreno. Ogni onda ha un *amplitude* e una *frequency.* La prima è l'altezza dell'onda, i valori sull'asse delle *y*, mentre la seconda è la sua ampiezza, ovvero i valori sull'asse delle x. Per rendere la nostra mappa più realistica, possiamo sovrapporre diverse di queste onde, che chiamiamo *octaves*. Ognuna avrà altezza e ampiezza differenti. La *frequency* di ogni *octave* verrà assegnata elevando valore chiamato *lacunarity* al numero dell' *octave* presa in considerazione. In questo modo la prima *octavea* avrà la *frequency* più bassa, mentre l'ultima la *frequency* più alta.

Visto che ogni *octave* rappresenta un livello di dettaglio diverso nella forma del terreno, più *octaves* si aggiungono e meno influenza devono avere nel profilo finale del terreno. Questa decrescita dell'importanza la otteniamo diminuendo gradualmente l'*amplitude* delle *octaves*, risultato che otteniamo



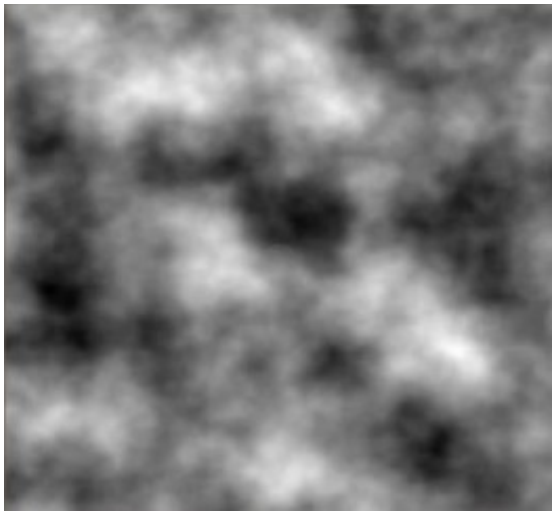
calcolando la *amplitude* in modo simile a come abbiamo fatto con la *frequency*, inserendo una nuova variabile, chiamata *persistance.* Sommando tutte le *octaves* insieme raggiungiamo una forma del terreno decisamente più credibile ed interessante.



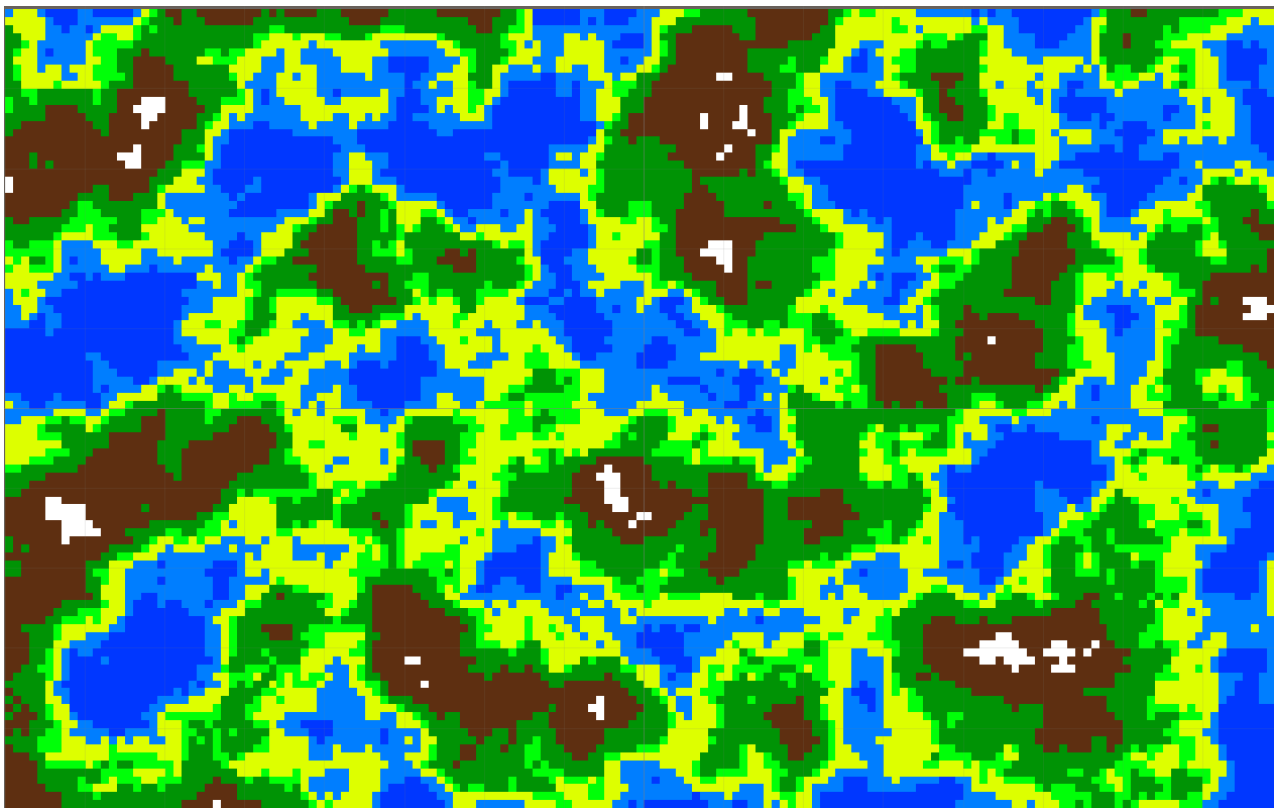
Fonte di tutte le immagini delle onde [7]

Il parametro *scale* ci permette di zoomare sulla nostra mappa, raggiungendo una dimensione della zona visualizzata tale da ottenere dei territori dalle dimensioni e proporzioni credibili. L'*offset* infine ci permette di muoverci all'interno della mappa, visualizzandone solo una porzione, in base agli spostamenti indicati in *x* e *y.* Questo ci permette di ottenere un numero di mappe pressoché infinito anche con un solo *seed*, nel caso desiderassimo

utilizzare delle mappe dalle dimensioni ristrette, dal momento che potremmo spostare ogni volta l'*offset*.
Come risultato finale della nostra funzione *GenerateNoiseMap,* otteniamo infine una mappa più sfumata e dettagliata di quella precedente, che in bianco e nero si presenta in questo modo. A sinistra troviamo la mappa ottenuta senza l'inserimento del codice relativo alle *octaves*, mentre a destra possiamo osservare la *noise map* finale.



A questo punto dobbiamo colorare questa immagine in modo tale da ottenere come risultato una mappa in stile topografico, evidenziando con diversi colori le diverse altezze del terreno. Ho fatto ciò nel file *MapGenerator,* la cui *array* pubblica *regions* ci permette di impostare direttamente dall'editor di *Unity* i parametri ed i colori secondo cui definire le varie zone della mappa. Io ho impostato i parametri di *regions* in modo tale che i punti con valori inferiori a 0,3 risultassero blu scuri, tra 0,3 e 0,4 azzurri, inferiori a 0,6 verde chiari, a 0,7 verde scuri, a 0,9 marroni e fino a 1 bianchi. La funzione *GenerateMap* di *Mapgenerator* controlla l'altezza di ogni punto e lo colora in base alle indicazioni di *regions.* Il file *MapDisplay* ci permette di scegliere se visualizzare la mappa a colori o la *noise map* in bianco e nero. Il file *TextureGenerator* infine crea la *texture* che viene applicata al piano presente nell'editor, il quale rappresenta la superficie del nostro mondo. La mappa che segue è un esempio dei risultati che otteniamo.

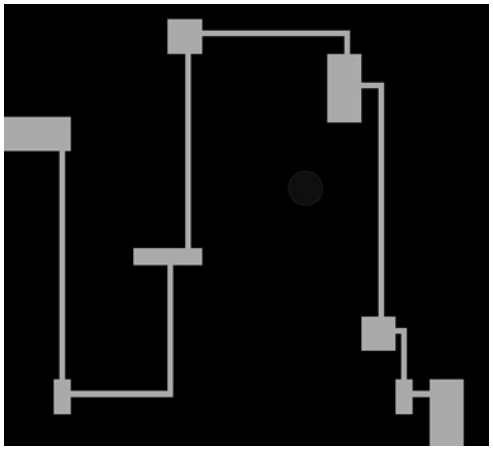


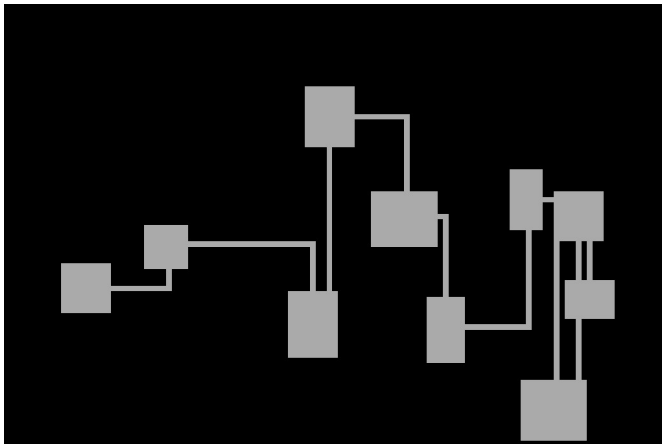
Possiamo notare che le forme del terreno non sono realistiche, pur avendo sempre delle forme dall'aspetto naturale. Questo perché i continenti della Terra non si sono formati in modo del tutto casuale, ma seguendo una logica di carattere geologico. Tuttavia il risultato ottenuto con l'utilizzo di *Perlin Noise* è più che soddisfacente, dal momento che la mappa nella maggior parte dei casi deve essere uno spazio di cui il giocatore non è pienamente consapevole, e che rimane invisibile a suoi occhi proprio perché è sufficientemente naturale da non disturbare la sua immersione nel mondo di gioco.

4. Risultati e discussione

4.1 Dungeons Maps

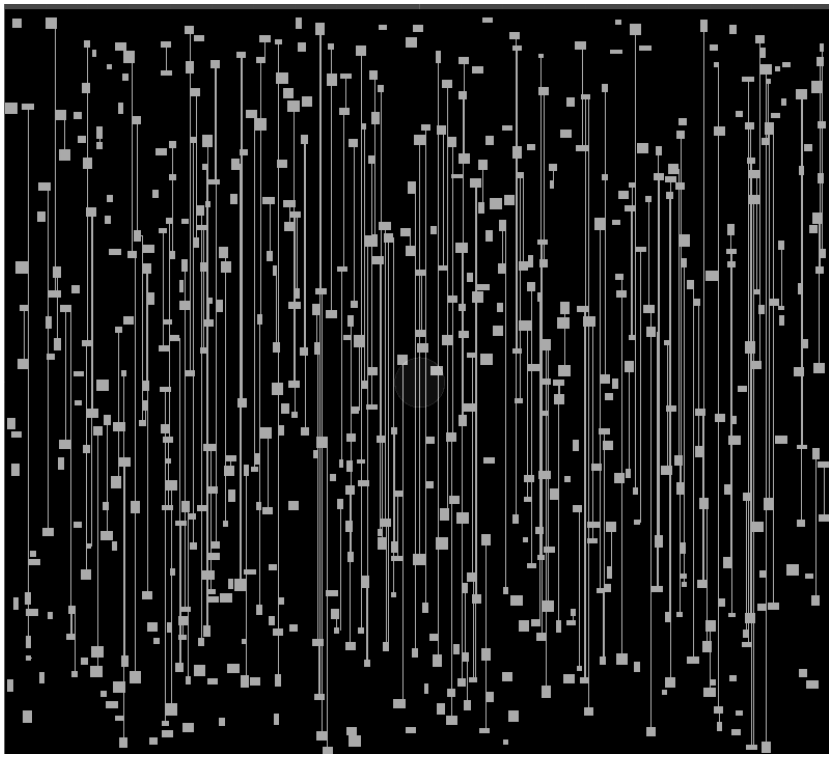
L'implementazione di entrambe le tecniche è andata a buon fine, permettendomi di raggiungere risultati abbastanza soddisfacenti in entrambi i casi, che ci permettono di raggiugere qualche importante conclusione.





La prima mappa, ottenuta con *BSP,* è il risultato di diverse prove del bilanciamento tra il numero delle stanze da creare e le iterazioni di divisione dello spazio. Tuttavia possiamo notare che questa configurazione ci porta ad ottenere spazi molto dilatati nella nostra mappa, che creano lunghi e noiosi corridoi tra una stanza e l'altra. Dal momento che normalmente in un videogioco sono le stanze ad essere i punti di interesse principali, mentre i corridoi sono solo spazi per poterle collegare, il risultato di *SRP* è molto più soddisfacente. Inoltre le dimensioni delle stanze in *BSP*, essendo determinate dalla divisione dello spazio della mappa, sono relative alla dimensione dello spazio totale, e per questo non è possibile aggiustare lo spazio tra le stanze semplicemente ingrandendole, come è invece possibile fare per *SRP.* Tentare di abbassare il numero di iterazioni in *BSP* per ottenere stanze più grandi non è ugualmente fattibile, dal momento che le stanze a questo punto troppo grandi inizierebbero a sovrapporsi tra loro. Per ovviare a questo problema si potrebbero Unire le due tecniche, facendo un controllo della posizione prima di creare la stanza come in *SRP*, ma a questo punto non si otterrebbe altro che un algoritmo molto più lento, perché per ogni posizione indesiderata si dovrebbe ricominciare a dividere la stanza daccapo, invece di prendere dei punti casuali come nella *SRP* vera e propria.
Per tutti questi motivi credo che in generale la tecnica migliore da utilizzare per creare mappe di *dungeons* tra *Simple Room Placement* e *Binary Space Partition* sia la prima, tranne esigenze particolari, in cui si desiderano *dungeon* molto lunghi e con le stanze notevolmente isolate tra di loro.

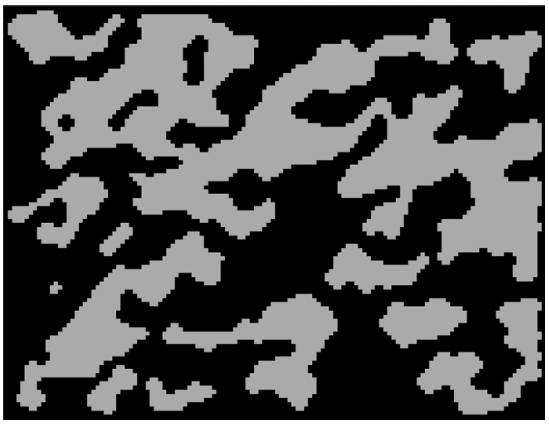
Personalmente lavorando alle due implementazioni ho avuto l'impressione che in *SRP* si abbia un maggior margine di manovra per quanto riguarda future espansioni del programma. Si può controllare in modo più puntuale la generazione delle stanze, in modo tale da ottenere una mappa più interessante per il giocatore, e adattare questa tecnica a vari tipi di gioco.
Possiamo fare un ulteriore riflessione sull'algoritmo di *pathfinding* che ho programmato. Se per mappe piccole come quelle degli esempi portati fino ad ora dona risultati soddisfacenti, in mappe molto più grandi non si può dire la stessa cosa.



Possiamo notare che le stanze non sono state collegate tra loro in modo soddisfacente, lasciando molte stanze completamente isolate e molte interruzioni nei sentieri, i quali si estendono per tutta l'altezza della mappa portando ad un zig-zag frenetico e sicuramente poco appassionante per il giocatore. Questo problema è causato dalla struttura stessa dell'algoritmo. Andando a cercare la stanza più vicina sull'asse delle ascisse indipendentemente dalla sua posizione in quello delle ordinate, trovandosi in una mappa così estesa vengono percorse grandi distanze in verticale tra una stanza e l'altra. Se si desidera quindi creare delle mappe molto estese, consiglio di codificare un algoritmo di *pathfinding* differente, che decida quali stanze collegare in modo più raffinato, magari basandosi sull'area che circonda la stanza di partenza, ricercando la prossima in ogni direzione.
Dal punto di vista delle prestazioni entrambe le tecniche raggiungono gli stessi risultati, creando anche mappe piuttosto grandi, con una *width* pari a 1200, *height* a 900 e 500 stanze all'interno, in un tempo inferiore al millisecondo.

4.2 Caves Maps

Al contrario della categoria precedente, i risultati tra le due tecniche implementate sono molto differenti tra loro.



Nel primo caso, la mappa creata con Drunkard''s Wark, gli spazi sono molto squadrati e spesso interrotti da sezioni di muri, un elemento che non è necessariamente negativo, dal momento che questi piccoli ripari possono costituire degli ostacoli, fornire protezione al giocatore o nasconderlo ai nemici e in generale rendono il gioco all'interno della mappa più interessante. Esteticamente, più che una caverna il risultato di *DW* sembra un *dungeon* costituito da qualche antica struttura parzialmente crollata.

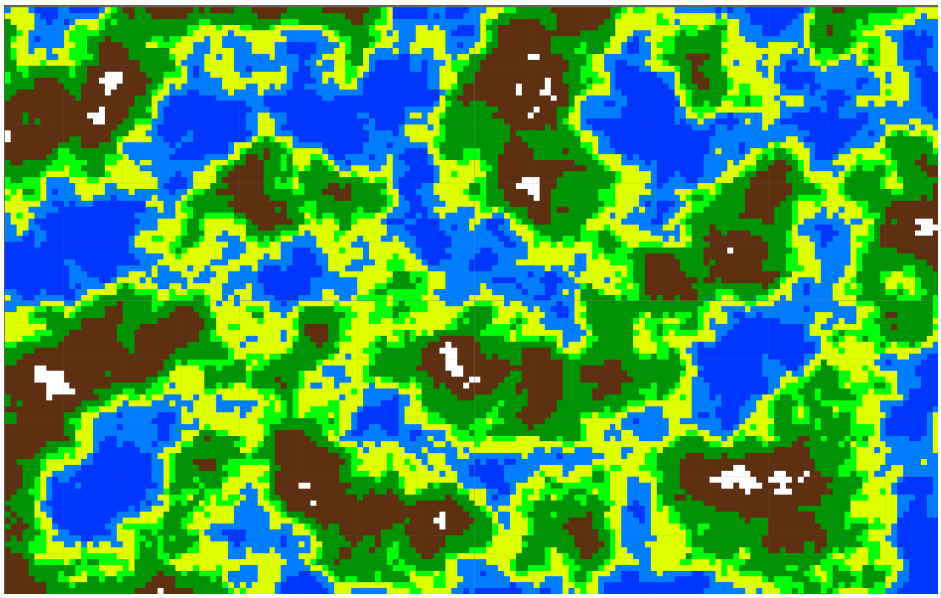
La seconda mappa invece, realizzata tramite *Cellular Automata*, ha un aspetto molto più rotondo e naturale, assomigliando molto di più ad una caverna di origine naturale, scavata nei secoli. Quasi sempre il risultato di *CA* sarà una mappa costituita da un grande corpo vuoto principale, affiancato da sacche più piccole isolate tra loro. Questa struttura suggerisce di utilizzare queste mappe in un gioco che preveda una meccanica di scavo, in modo tale che il giocatore possa penetrare nelle pareti e raggiungere le varie grotte più piccole disperse nella mappa. Al contrario in *DW* è garantito che la caverna sia un pezzo unico, e il giocatore possa sempre raggiungere ogni punto in qualsiasi momento.

Lo svantaggio di *CA* è che pur potendo creare un numero di mappe pressoché infinito, queste tendono ad assomigliarsi molto tra di loro, e alla lunga credo che possano risultare monotone al giocatore. Al contrario *DW* fornisce mappe molto diverse sia in forma sia in dimensioni, elemento che può determinare una rigiocabilità del gioco molto maggiore. *CA* ha però il vantaggio di permettere di controllare le dimensioni dello spazio a disposizione del giocatore tramite la variabile *RandomFillPercent.* Questo ci permette di creare mappe di dimensioni diverse in modo più controllato rispetto a *DW,* dove la dimensione della mappa è totalmente casuale e decisa in modo indipendente dal nostro volere.

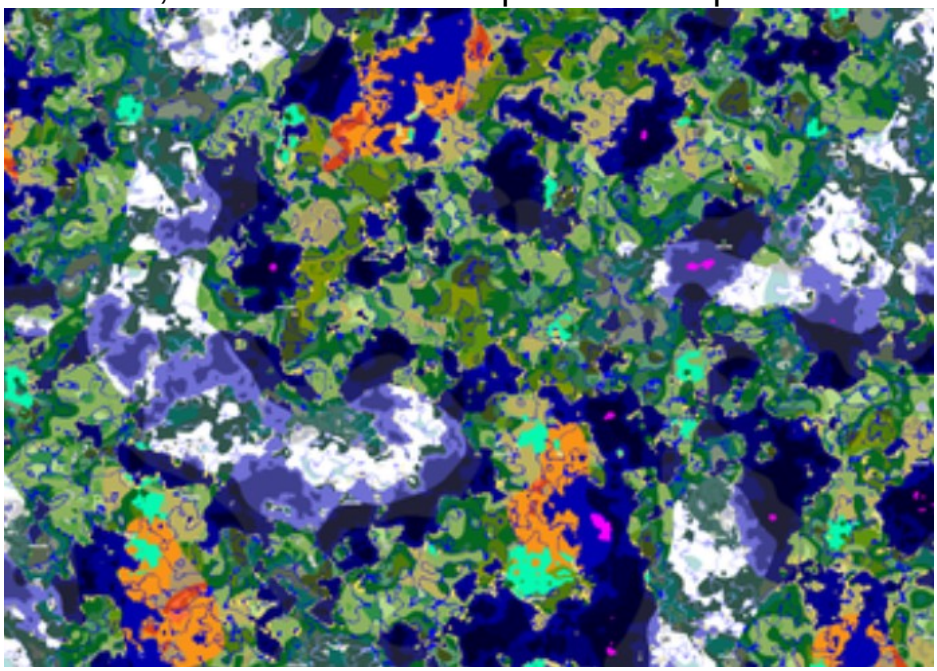
Le prestazioni delle mie due implementazioni sono molto simili quando le mappe che si desidera creare non sono troppo grandi, all'incirca come nelle mappe mostrate negli esempi soprastanti, dove *width* ha valore di 160 e *height* di 100; infatti entrambe realizzano la mappe in un tempo minore dei 2 millisecondi, però più le dimensioni della mappa crescono e più le prestazioni differiscono. Creando mappe con *width* pari a 1400 e *height* pari a 960, *CA* impiega tra i 140 e i 150 millisecondi, mentre *DW* ha una forbice molto più ampia in proporzione, ma rimane all'incirca tra i 4 e i 20 millisecondi nella maggior parte dei casi. La variabilità nel tempo di esecuzione di *DW* è dovuta al fatto che il programma decide in modo del tutto aleatorio la direzione in cui muoversi, e si ferma quando raggiunge uno dei quattro confini, dunque alle volte può capitare che percorra quasi tutta la mappa, mentre altre volte può fermarsi dopo poco tempo.
La differenza nelle prestazioni delle due tecniche è certamente importante, ma non a livello tale da rendere l'uso di *CA* invalidante, rischiando di rovinare l'intero gioco. Tanto più che la generazione della mappa di gioco avviene solitamente una sola volta, all'inizio della partita, e in tutte le sessioni di gioco successive della stessa partita la mappa viene semplicemente caricata. Per questo motivo non ritengo che il maggior tempo di generazione della mappa di *CA* sia un fattore decisivo nella scelta della tecnica da utilizzare.

4.3 Perlin Noise

L'ultima implementazione, *Perlin Noise*, permette di creare ampie mappe di spazi aperti, adatte per tipi di giochi completamente differenti dalle quattro mappe precedenti, come *open-world* e strategici manageriali e tattici.



Possiamo osservare che il risultato ottenuto è una mappa con diversi biomi definiti dalla loro altitudine, inoltre essendo *Perlin Noise* gradiente, si ha una certa costante gradualità nel passaggio tra 0 e 1, impendendo in questo modo la formazione di paesaggi come montagne a picco sul mare o pianure sconfinate. Complessivamente assomiglia molto alle mappe del popolare videogioco *Minecraft,* di cui ne inserisco qui sotto una per un confronto.



[https://www.reddit.com/r/Minecraft/comments/rcfogj/the_new_world_generation_looks_awesome_even_on_a/]

Come si può facilmente notare sono molto simili, questo perché *Minecraft* per generare i suoi mondi di gioco utilizza degli algoritmi di *gradient noise*, gruppo di cui fa parte anche quello di *Perlin Noise.* Tuttavia è possibile notare facilmente dalla mappa di *Minecraft* che questo gioco non utilizza solo l'altezza per suddividere il proprio mondo in biomi, ma tiene in considerazione sicuramente anche altri parametri, ottenendo delle mappe molto più complesse e meno ripetitive della semplice implementazione che ho sviluppato. Tuttavia questo ci mostra le potenzialità di questa tecnica, che può venire implementata in diverse generazioni di mondi, in modo tale da ottenere ottimi risultati. Un altro esempio di come può essere utilizzata è il gioco *Dwarf Fortress,* che affianca la generazione del mondo fisico creato da *Perlin Noise* ad algoritmi che generano proceduralmente civiltà e avvenimenti, creando in questo modo una storia sempre diversa per un mondo nuovo ad ogni partita, le cui tracce si possono leggere sulla mappa, sotto forma di città e strade.

[https://www.rockpapershotgun.com/heres-a-peek-at-dwarf-fortresss-upcoming-non-ascii-maps]

5. Conclusione

Sono molto felice di aver raggiunto tutti gli obbiettivi che mi ero prefissato all’inizio di questo lavoro. I risultati ottenuti sono stati molto soddisfacenti e mi hanno permesso di fare un ottimo confronto tra le varie tecniche, permettendomi di inquadrarle meglio secondo la loro utilità nelle diverse situazioni. Le implementazioni realizzate per costruire questo lavoro sono pubbliche, e possono essere ulteriormente ampliate, anche con alcuni spunti forniti in questo lavoro. In futuro potrebbe essere interessante estendere il confronto, aggiungendo ulteriori implementazioni di altre tecniche.

6. Introduzione seconda parte

Questa seconda parte del lavoro, svolta dopo la redazione del testo precedente, mira a sperimentare una nuova tecnica, *Wave Function Collapse.* Lo scopo è quello di provare ad implementare anche questa tecnica, per ottenere risultati analoghi a quelli precedenti utilizzando un solo algoritmo. Questo è possibile perché *WFC* si basa sull'idea di automatizzare il posizionamento di varie *tiles* in modo coerente tra loro. Le *tiles* sono un concetto già conosciuto e molto utilizzato nello sviluppo di videogiochi. Sono delle piastrelle di uguali dimensioni, su cui vengono disegnati gli elementi della mappa. Queste piastrelle vengono poi posizionate vicine le une alle altre in modo tale che le illustrazioni si uniscano formando un disegno più ampio.
Con *WFC* proverò ad automatizzare il processo di scelta nel posizionamento di queste piastrelle, rendendo possibile generare infinite mappe con qualsiasi set di *tiles*.
Le quattro implementazioni sono divise in due approcci differenti:

- *WFC*-single-tiles, con cui ho creato *WFC-simple caves* e *WFC-Perlin Noise*
- *WFC-3x3 tiles,* con cui ho creato *WFC-complex caves* e *WFC-dungeons*

Alla fine confronteremo i risultati ottenuti con *WFC* e le tecniche mostrate in precedenza, cercando di capire se questo nuovo approccio può essere in grado di sostituire la varietà di algoritmi sviluppati nel corso dei decenni passati.

7. Materiali e metodologie

I materiali utilizzati sono stati quelli indicati nella sezione 3.1, ad eccezione del linguaggio di programmazione Python, che ho utilizzato per scrivere il codice di *WFC-strade.*
La metodologia per *Wave Function Collapse* in generale è riassunta nel seguente pseudocodice:

```
initialize_grid:

input:
	n tiles in x -> numero
	n tiles in y -> numero
	possibili tiles -> lista di array 3x3

procedimento:
	accoppia tutte le posizioni possibili ed assegna loro tutti
	i pesi di ogni possibilità

	for x in nx:
		for y in ny:
			add x, y, weights

output:
	lista con {x, y e vettore con i pesi} per ogni casella


randomly_collapse_the_first:

input:
	lista con {x, y e vettore con i pesi} per ogni casella

procedimento:
	propaga sui lati -> (x == 0 || x == width-1 || y == 0 || y
	== height -1)

	prendi un elemento random e collassalo


output:
		lista con {x, y e vettore con i pesi} per ogni casella
(aggiornata con i nuovi pesi)

propagate:

	input:
		lista con {x, y e vettore con i pesi} per ogni casella


	procedimento:
```

```
            propaga sulle tiles vicine a quelle gia' collassate

      output:
            lista con {x, y e vettore con i pesi} per ogni casella
(aggiornata con i nuovi pesi)

collapse:

      input:
            lista con {x, y e vettore con i pesi} per ogni casella

      procedimento:
            prendi tutte le caselle con il minor numero di pesi
            scegli randomicamente una delle possibilita' rimaste
(in base al peso di ogni possibilit'a)

      output:
            lista con {x, y e vettore con i pesi} per ogni casella
(aggiornata con i nuovi pesi)

struttura:

      Start {
            initialize_grid()
            randomly_collapse_the_first()
            while all_tiles not collapsed:
                  propagate()
                  collapse()
      }
```

7.1 Metodologia – WFC- single-tiles

Ho affrontato l'implementazione di *WFC* in due modi differenti: prima utilizzando singole *tiles* il cui valore viene selezionato in base a quello dei vicini; mentre nel secondo approccio ho diviso le singole *tiles* in griglie 3x3.
Questo primo approccio è stato implementato in Unity come tutte le tecniche precedenti, e in questo modo ho creato due *proof of concept*. Il primo mira a mostrare come si possano creare delle caverne grazie a *WFC*, mentre il secondo punta a sostituire *Perlin Noise* nella generazione di spazi aperti.

7.1.1 Metodologia – WFC- simple caves

Ho deciso di iniziare con questa tecnica dal momento che bastavano due sole *tiles,* una bianca ed una nera, e per questo è stata più semplice da gestire.

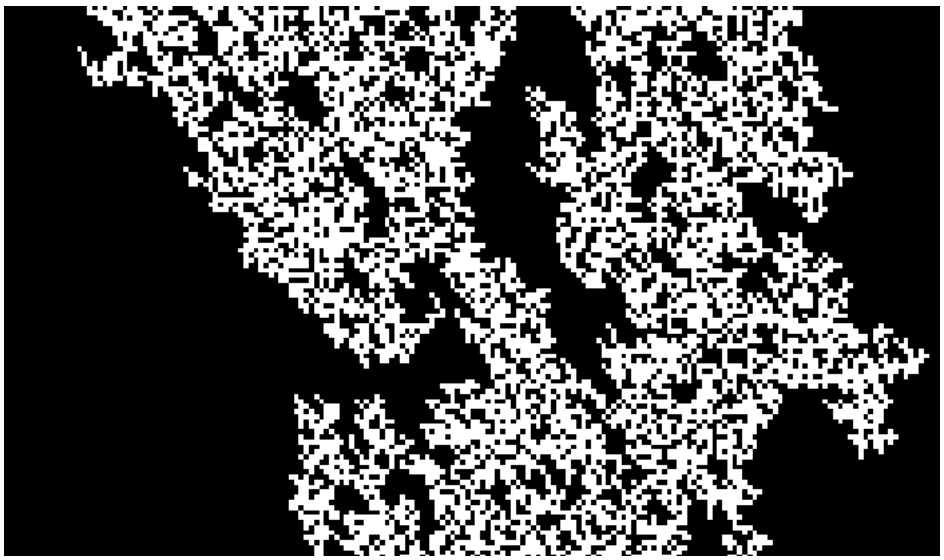
WFC si basa sull'idea che le *tiles* vadano piazzate secondo certe regole in base ai loro vicini. Queste regole modificano il peso (la probabilità) di una certa scelta. Infine viene estratta la *tile*, questo processo si chiama collasso, da cui prende il nome la tecnica.

Per creare delle caverne viene inizialmente effettuato un primo collasso, del tutto aleatorio, su una o più caselle. Da qui possiamo iniziare a propagare, ovvero cambiare i pesi delle caselle vicine a quelle già collassate.

Per esempio, in questa mia implementazione, ogni casella ha una variabile chiamata *weight* ed ogni casella ha tre stati possibili: 0 se non è ancora stata collassata, 1 se è stata collassata come bianca e 2 se viene collassata come nera*. Weight* è inizialmente uguale a 0 ed aumenta di uno per ogni *tile* pari ad 1 che la casella a vicina. Grazie al valore di *weight* sono in grado di avere diverse regole per differenti probabilità. Se per esempio *weight* è uguale a 4, significa che tutte le caselle intorno a quella presa in considerazione sono state collassate come bianche, pertanto la regola che ho messo in questo caso è che la *tile* diventi bianca 9 volte su 10.

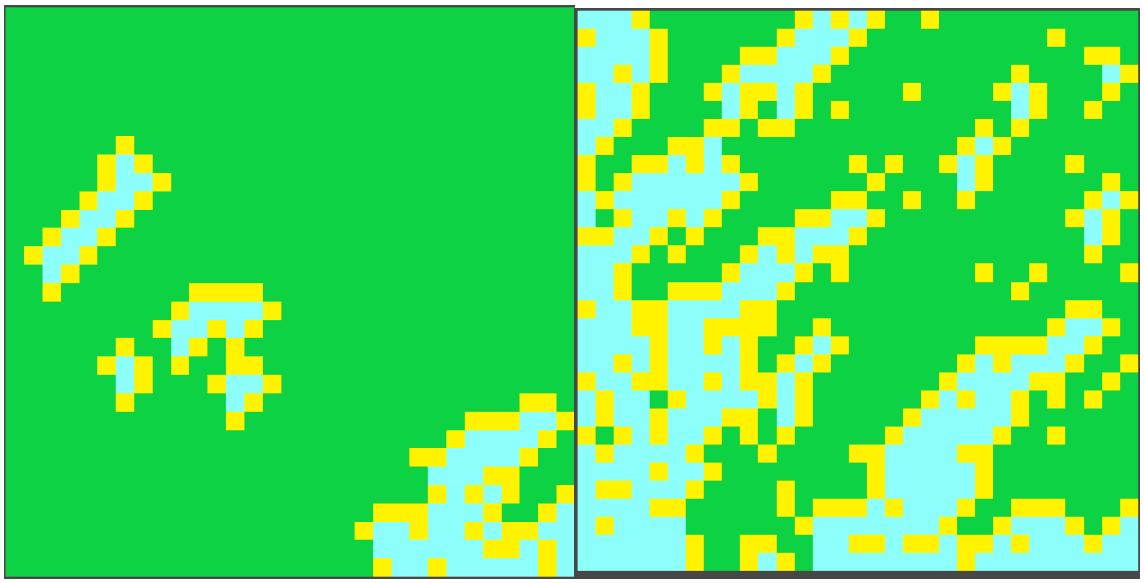
Questa sequenza viene iterata il numero specificato di volte, e infine vengono "dipinte" le *tiles* del giusto colore in base al valore specificato.

Per migliorare questa generazione, si può considerare di immagazzinare le *tiles* in una struttura a spirale. Questo perché immagazzinandole in modo lineare passando in sequenza riga per riga o colonna per colonna, la mappa viene allungata sulla diagonale, finendo per allungarsi in modo poco naturale ed estetico.



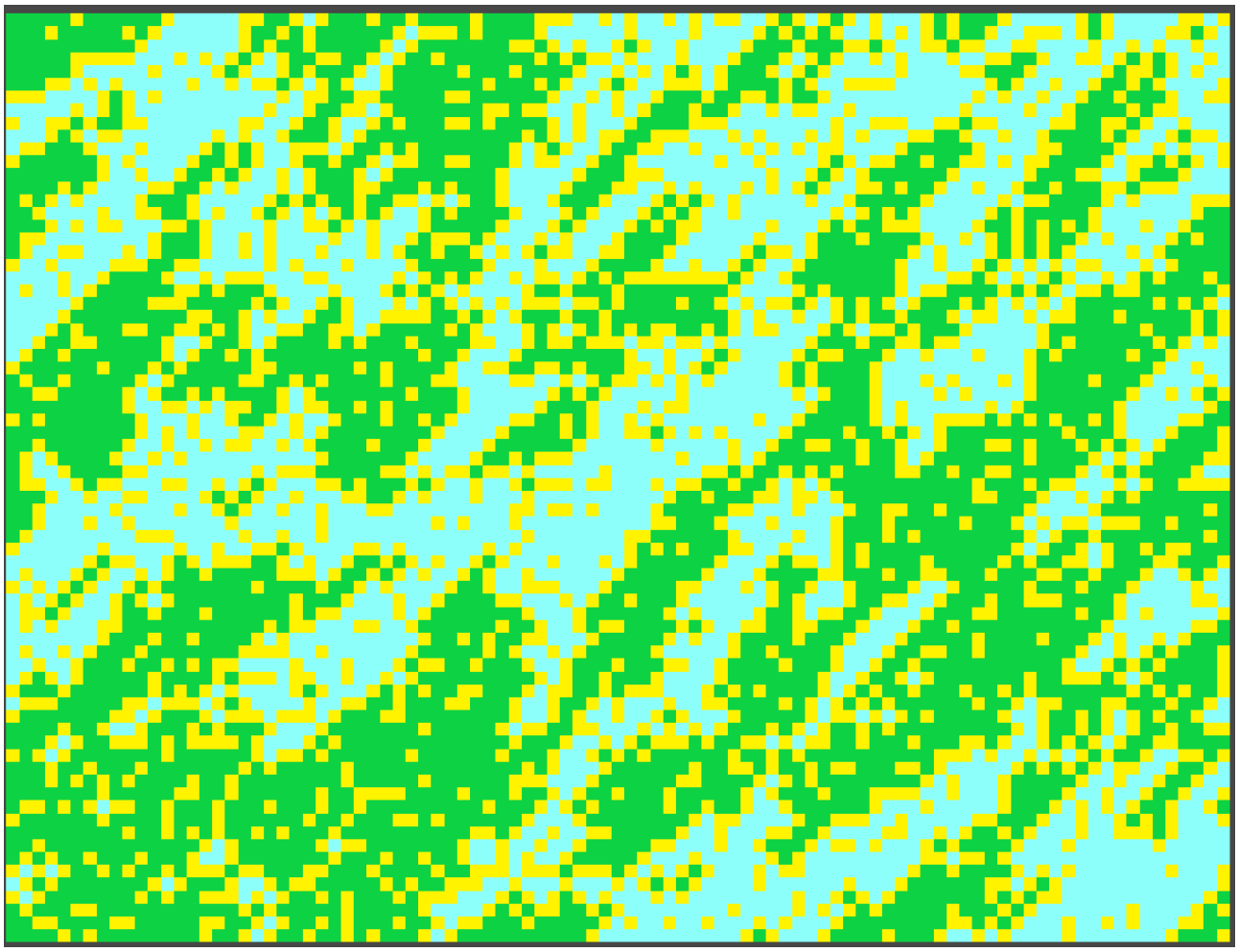
7.1.2 Metodologia – WFC-Perlin Noise

Questa implementazione è molto simile a quella che è stata spiegata precedentemente, dobbiamo semplicemente gestire un numero di *tiles* maggiore, dal momento che il nostro scopo è quello di ricreare una mappa simile a quella che abbiamo generato con *Perlin Noise*, quindi necessitiamo di più colori. Per questo semplice esempio ne userò tre, azzurro per il mare, giallo per le spiagge e verde per i territori dell'entroterra.
Partiamo sempre collassando uno o più punti in modo casuale, per poi poter propagare sulle caselle limitrofe. In questo caso d'uso vogliamo semplicemente evitare che le caselle verdi si trovino vicine a quelle azzurre. Per poter aggiustare i pesi in modo tale da ottenere questo risultato, propagando andremo a verificare quali *tiles* sono presenti vicino alla casella presa in considerazione. Se saranno presenti *tiles* verdi o azzurre, immagazzineremo queste informazioni nelle *bool is_grass* e *is_water*. A questo punto possiamo decidere come aggiustare le nostre probabilità. Iniziamo vietando il posizionamento di caselle verdi nel caso in cui *is_water* sia vera e viceversa. Poi possiamo decidere quanto spazio dare alle *tiles* gialle. Per esempio potremmo posizionarle solo nel caso in cui entrambe le booleane siano vere, ottenendo quindi una mappa prevalentemente verde o azzurra. Oppure per renderla più simile a *Perlin Noise* possiamo fare in modo che nel caso in cui sia vera solo una delle due variabili, le probabilità di ottenere una *tile* gialla invece di una verde o azzurra siano di 1 a 4, come nell'esempio qui a destra.



Si può facilmente notare che questa implementazione è tutt'altro che perfetta, si vedono spesso errori degli angoli in basso a sinistra sui confini tra *tiles* verdi e azzurre. Questo è per via della struttura dell'immagazzinazione delle

tiles, come spiegato. La problematicità viene ancora più evidenziata se allarghiamo la mappa che desideriamo ottenere.



Con dimensioni maggiori si può notare facilmente la tendenza di queste mappe a svilupparsi per strisce diagonali. Come detto prima questo problema è più causato dal tipo di strutture che ho creato durante l'implementazione che con la tecnica in sé. Infatti l'idea di *WFC* è quella di basarsi sui vicini, se la le *tiles* vengono immagazzinate in modo non ottimale come in questo caso, si incorre in questo tipo di problemi.
L'esempio precedente però ci mostra chiaramente le grandi potenzialità di questa tecnica rispetto a *Perlin Noise.* Con Perlin Noise vengono create per forza di cose dei bordi sfumati tra una regione e l'altra, mentre qui possiamo decidere noi che tipo di confini vogliamo ottenere.

7.2 Metodologia – WFC- 3x3 tiles

Questo secondo approccio a *WFC* è più complesso ed interessante. Dividendo ogni *tile* in 9 caselle più piccole sono stato in grado di disegnare forme come angoli, corridoi e pareti. Poi il posizionamento delle *tiles* viene deciso in base solo ai lati combacianti, e non necessariamente su tutta la *tiles*. Grazie a ciò sono stato in grado di generare mappe di caverne più raffinate e personalizzabili delle precedenti e anche mappe complesse come quelle di un *dungeon*. Per implementare questi algoritmi non ho utilizzato Unity, bensì Python, trovandomi più a mio agio con questo linguaggio di programmazione nel gestire la maggiore difficoltà di questa parte del lavoro.

7.2.1 Metodologia – WFC- complex caves

Inizialmente ho definito tutte le *tiles* possibili. Ho creato sei possibilità: completamente piene, completamente vuote e infine le quattro piene solo per uno dei lati alla volta. Ogni casella della mappa è definita in una lista contenente la sua posizione tramite le coordinate *x* e *y,* una lista con i pesi per tutte le *tiles* ed una variabile booleana chiamata *collapsed*, vera nel caso in cui la casella sia già stata collassata e falsa in caso contrario.
Dopo la creazione della mappa delle dimensioni specificate di *width* per *height*, tramite la funzione *initialize_grid*, con la funzione *first_collapse* viene collassata in modo aleatorio una delle caselle. Una volta fatto ciò vengono chiamate le funzioni *propagate* e *collapse,* iterandole fino al completamento della mappa. *Propagate* si occupa di aggiornare la lista dei pesi in base alle regole specificate. In questo caso vogliamo evitare che le *tiles* completamente vuote si trovino direttamente in contatto con quelle completamente piene, e che il loro passaggio sia mediato dalle *tiles* piene solo per un lato. Ad esempio se la casella presa in considerazione è completamente vuota, quella superiore dovrà essere o vuota anch'essa, oppure piena solo sul lato superiore. La lista dei pesi viene gestita in questo modo. La posizione del peso nella lista determina a quale *tile* sta facendo riferimento, mentre il peso stesso è pari a 1 se la *tile* deve poter essere scelta, mentre è uguale a 0 se la *tile* non deve venire piazzata. Riferendoci all'esempio di prima possiamo dire che il peso della *tile* completamente vuota e di quella piena solo sul lato superiore è 1, mentre il peso di tutte le altre è 0.
Una volta propagato sulle *tiles* vicine a quelle già collassate la funzione *collapse* si occupa di collassare tutte le *tiles* con meno pesi pari a uno, dunque su cui la scelta viene effettuata su meno opzioni. *Collapse* sceglie in modo casuale una *tile* tra le possibilità rimaste dopo la propagazione e la casella viene collassata sulla *tile* scelta.
Il risultato di questo programma è una lista contenente tutte le *tiles* della mappa in questa forma:
[[[1, 1, 1], [0, 0, 0], [0, 0, 0]], [[1, 1, 1], [0, 0, 0], [0, 0, 0]], [[1, 1, 1], [1, 1, 1], [1, 1, 1]], [[1, 1, 1], [1, 1, 1], [1, 1, 1]], [[1, 1, 1], [1, 1, 1], [1, 1, 1]], [[1, 1, 1], [1, 1, 1], [1, 1, 1]], [[1, 1, 1], [1, 1, 1], [1, 1, 1]], [[1, 0, 0], [1, 0, 0], [1, 0, 0]], [[0, 0, 1], [0, 0, 1], [0, 0, 1]]]

La prima *tile* è quella di posizione 0,0; la seconda 1,0, ecc. Arrivati a una *x* pari alla *width* viene aumentata di uno la *y*. Così via fino all'ultima *width*, *height*.
Sistemando le *tiles* della lista precedente secondo la loro posizione otteniamo questa mappa:

1 1 1 1 0 0 0 0 1
1 1 1 1 0 0 0 0 1
1 1 1 1 0 0 0 0 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 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 0 0

Questo semplice esempio di una griglia 3x3 ci mostra come siamo stati in grado di ottenere una mappa con due caverna distinte, una di maggiori dimensioni sotto e una più piccola sopra. Rispetto alla caverna precedentemente ottenuta con *WFC* (che somigliava più a *DW),* questa ultima è più simile a *CA*.

7.2.2 Metodologia – WFC- dungeons

Le *tiles* possibili in questo caso sono 8. Una completamente piena, una completamente vuota, una strada verticale, una orizzontale e quattro strade ad angolo. La potenzialità di *WFC* è proprio quella che ci basta cambiare le *tiles* e le loro regole per ottenere tutte le mappe che desideriamo, infatti il codice rimane esattamente identico dal punto di vista funzionale.
Le regole in questo caso sono che le strade e gli angoli devono essere sempre posizionati in modo da combaciare, senza lasciare corridoi ciechi. Ad esempio una strada orizzontale alla sua destra potrà avere solo una strada orizzontale, un angolo che a partire dal lato sinistro curvi sopra o sotto oppure una stanza.
Il risultato del programma sarà una lista proprio come nell'implementazione precedente:
[[[0, 0, 0], [0, 1, 1], [0, 1, 0]], [[0, 0, 0], [1, 1, 1], [0, 0, 0]], [[0, 0, 0], [1, 1, 1], [0, 0, 0]], [[0, 0, 0], [1, 1, 1], [0, 0, 0]], [[1, 1, 1], [1, 1, 1], [1, 1, 1]], [[0, 1, 0], [0, 1, 1], [0, 0, 0]], [[0, 0, 0], [1, 1, 1], [0, 0, 0]], [[1, 1, 1], [1, 1, 1], [1, 1, 1]], [[0, 1, 0], [0, 1, 0], [0, 1, 0]]]

Una volta disposte le caselle nello spazio il risultato è questa mappa:

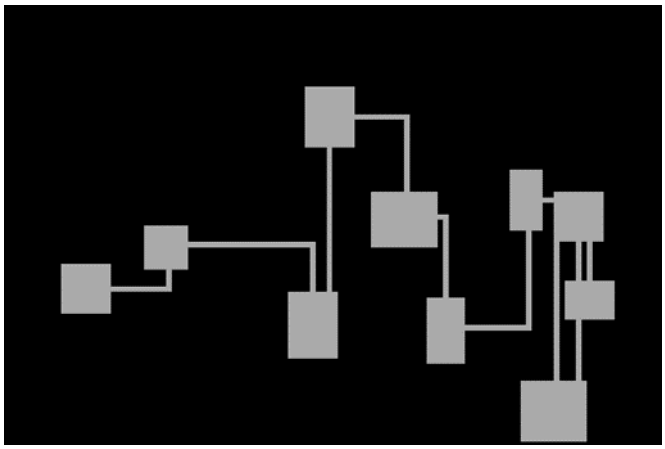
0 0 0 1 1 1 0 1 0
1 1 1 1 1 1 0 1 0
0 0 0 1 1 1 0 1 0
0 0 0 1 1 1 0 1 0
1 1 1 1 1 1 0 1 1
0 0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1
0 1 0 0 0 0 0 0 0

Come si può vedere abbiamo ottenuto una stanza nella parte superiore aperta verso l'alto collegata a due corridoi che escono sul lato sinistro. A destra e in basso abbiamo altri due corridoi aperti sui lati inferiore, superiore e destro.

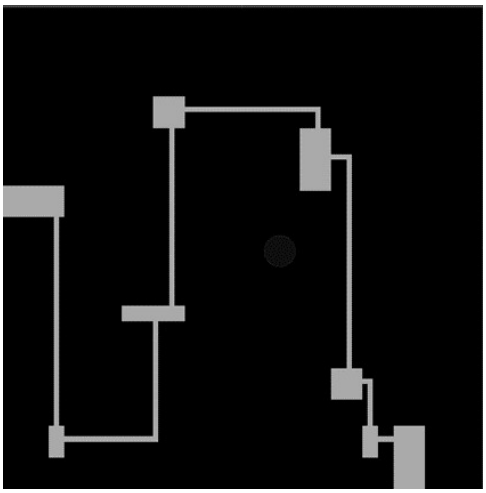
8. Risultati e discussione

Il risultato di tutte le implementazioni di *WFC* è stata una mappa più o meno simile a quella della tecnica originale. Alcune, come quella di *WFC-Perlin Noise* sono molto rozze e semplici rispetto all'originale, ma credo che la riproducibilità delle mappe possa comunque essere confermata.

Dungeons:



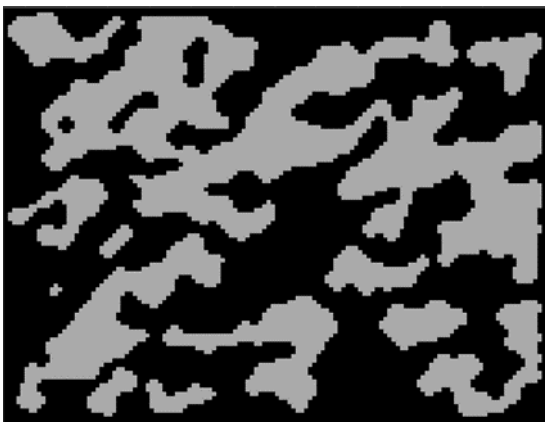
0 0 0 1 1 1 0 1 0
1 1 1 1 1 1 0 1 0
0 0 0 1 1 1 0 1 0
0 0 0 1 1 1 0 1 0
1 1 1 1 1 1 0 1 1
0 0 0 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 1 1
0 1 0 0 0 0 0 0 0

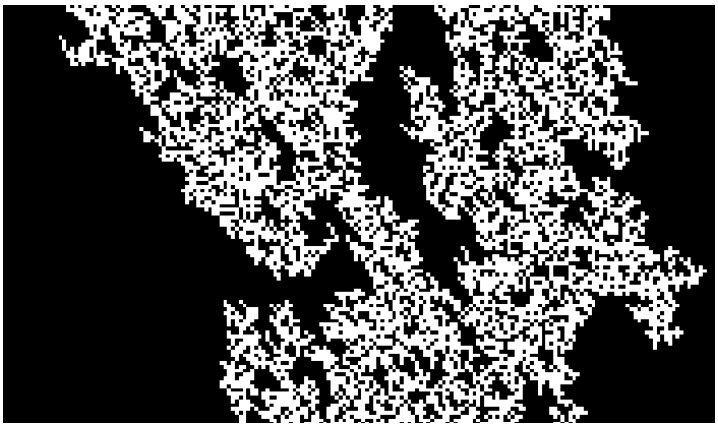


La mappa qui mostrata di *WFC* è di dimensioni ridotte rispetto all'originale, di conseguenza è presente una sola stanza. Questo è dovuto al fatto che al momento della redazione di questo testo non sono ancora riuscito a sviluppare un sistema con Python che possa trasformare automaticamente la lista in una mappa. Tuttavia possiamo comuqnue vedere facilmente che i criteri di giocabilità per una mappa *dungeon* sono rispettati. I corridoi puntano in direzioni diverse, pertanto verrano create numerose stanze in modo non lineare ed interessante. Inoltre su 81 *micro-tiles* 18 sono occupate da una stanza, quindi possiamo presumere che il rapporto tra lo spazio percorso dal giocatore ed il suo arrivo in una nuova stanza sia sicuramente migliore rispetto a *BSP.* Alla luce di queste considerazioni direi che *WFC* non ha solo replicato le tecniche precedenti, ma le ha addirittura superate,

ottenendo una mappa migliore, sicuramente nel caso di *BSP* e forse anche di *SRP*.

Caverne:

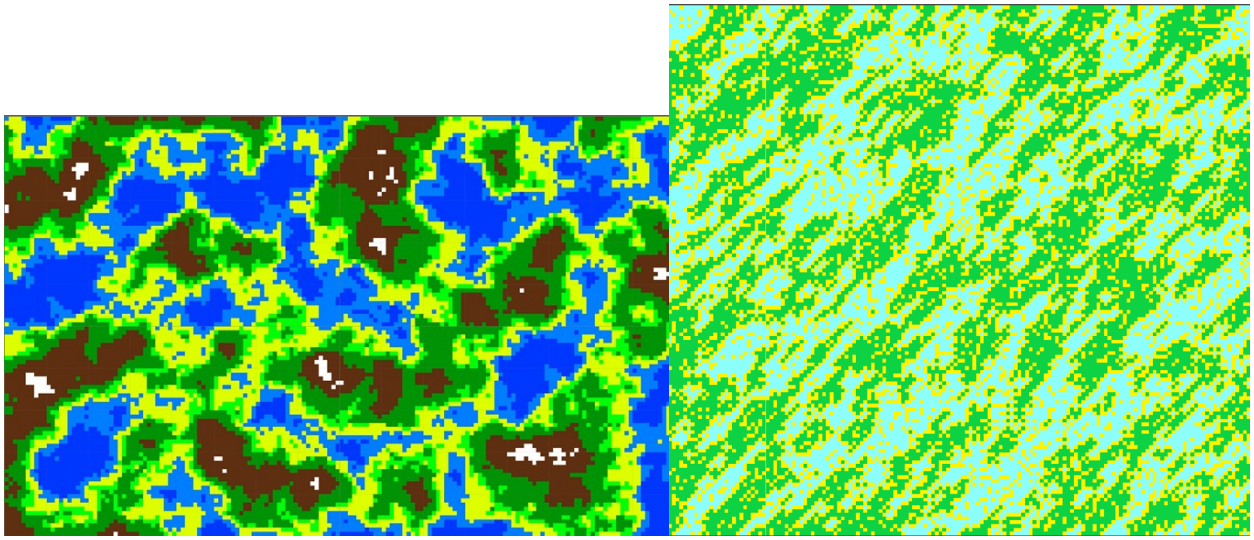




1 1 1 1 0 0 0 0 1
1 1 1 1 0 0 0 0 1
1 1 1 1 0 0 0 0 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0

I due differenti approcci a *WFC* hanno dato come risultato due mappe piuttosto diverse, Il primo, con le *tiles* a blocchetti unici, ha finito per generare una mappa più “gradulosa”, rendendola simile a quella di *DW.* Il secondo approccio invece ha creato delle caverne più grandi e compatte, divise tra loro, risultando quindi più simile a *CA.* Per il problema con Python spiegato prima la mappa 3x3 è molto più piccola delle altre. Questo ci rende difficile prevedere se con dimensioni maggiori il risultato finale sarebbe stato ancora più simile a *CA*, finendo per arrotondare le forme, o le varne sarebbero state più piccole e frammentarie come nella *WFC* a *tiles* singole.

Perlin Noise:



Questa implementazione di *WFC* è la meno riuscita e simile all'originale. Molti dei problemi che saltano subito all'occhio tuttavia sono dovuti a miei errori o lacune, come ad esempio la forma allungata e orientata in diagonale dei territori; o gli errori begli angoli in basso a sistri lungo i confini tra *tiles* azzurre e verdi. Perlin Noise ha una caratteristica particolare, cioè che a seconda di come viene declinata può consistere sia in un vantaggio che in uno svantaggio. Per creare una nuova zona con Perlin Noise basta assegnare un colore ad una certa fascia di altezza, mentre con *WFC* dobbiamo creare una nuova *tile* e tutte le regole necessarie a far sì che venga generate nel luogo giusto, rendendolo un processo molto più difficoltoso di quello di *Perlin Noise.* Tuttavia questo rende *WFC* molto più flessibile e versatile di *Perlin Noise*, che come discusso in precedenza, ci permette di creare delle zone in base ad un solo paramentro, l'altezza. Utilizzando *WFC* invece possiamo decidere noi in base a quali regole creare una nuova zona, potendo generare (con un'implementazione migliore) mappe più variegate e complesse di quelle ottenute con *Perlin Noise*.

9. Conclusione

Nel complesso penso che questa seconda parte del lavoro abbia ben dimostrato che è possibile utilizzare *Wave Function Collapse* per sostituire tutte le tecniche più vecchie con un solo algoritmo. Grazie alla sua modularità e versatilità *WFC* ci permette di generare qualsiasi tipo di mappa 2D per i videogiochi, cambiando solamente le *tiles* e le regole fornite all'algoritmo. Credo che questa tecnica potrebbe rappresentare un notevole passo avanti se utilizzata nell'industria, perché permetterebbe di sviluppare un solo codice per generare tutte le zone di gioco.

6. Indice delle abbreviazioni e glossario

BSP: Binary Space Partition

CA*:* Cellular Automata

Dungeon: strutture di fantasia, solitamente dimora di più sfide che il giocatore deve affrontare per raggiungere l'ultima stanza o trovare un tesoro.

DW*:* Drunkard's Walk

Open-world: gioco la cui mappa è aperta ed esplorabile liberamente dal giocatore.

Pathfinding: tipo algoritmo atto a cercare un percorso tra due o più punti nello spazio.

Randomico: casuale, non definito precedentemente.

Range: il ventaglio di tutti i numeri compresi tra due numeri dati. *Es:* range [0, 100] significa tutti i numeri compresi tra 0 e 100.

Roguelike: genere di videogioco caratterizzato da livelli generati proceduralmente e morte permanente del giocatore.

SRP: Simple Room Placement

Textures: immagine digitale rappresentante la superficie di un modello tridimensionale.

Tiles: piastrelle digitali solitamente di forma quadrata con varie textures che vengono disposte una affianco all'altra facendo combaciare le textures per formare un disegno più grande coerente.

WFC: Wave Function Collapse

7. Bibliografia e sitografia

Di seguito sono riportate tutte le fonti utilizzate per la realizzazione di questo lavoro.

[1] Herbert Wolverson, *Procedural Map Generetion Techniques*, 2020 virtual Roguelike Celebration Festival
https://www.youtube.com/watch?v=TlLIOgWYVpI&t=345s

[2] Wikipedia, *List of roguelikes,* September 28 2022
https://en.wikipedia.org/wiki/List_of_roguelikes

[3] Martin Gardner, *The fantastic combinations of John Conway's new solitaire game "life",* Scientific American October 1970 (pag. 120-123)
https://web.stanford.edu/class/sts145/Library/life.pdf

[4] Sebastian Lague, *Procedural Cave Generation,* Youtube
https://www.youtube.com/watch?v=v7yyZZjF1z4&t=3s

[5] Ken Perlin, *Making Noise,* GDCHardCore December 9 1999
https://web.archive.org/web/20071008162042/http://www.noisemachine.com/talk1/index.html

[6] Unity Documentation, *Mathf.PerlinNoise,* November 25 2022
https://docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html

[7] Sebastian Lague, *Procedural Landmass Generation,* Youtube
https://www.youtube.com/watch?v=wbpMiKiSKm8