POLITECNICO
DI TORINO

Algoritmi e tecnologie per la generazione parametrica e procedurale di ambienti urbani interattivi 3D per la riabilitazione cognitiva spaziale

Tesista: Francesco Paolo Maggiolino

Relatore: Andrea Giuseppe Bottino

Indice

Introduzione

Scopo dello strumento

Lo scopo di questa tesi è la creazione di uno strumento per la generazione procedurale e parametrica di ambienti cittadini in realtà virtuale navigabili.
Questo strumento farà parte di un'applicazione di test e riabilitazione della memoria spaziale di pazienti con diverse situazioni di deficit cognitivo.
Questo è un primo approccio nato dalle richieste degli specialisti psicologi in un contesto di pluriennale collaborazione del dipartimento di psicologia dell'università di Torino con SynArea sugli aspetti di utilizzo della realtà virtuale nel campo della riabilitazione cognitiva spaziale.
All'interno dell'applicazione, il paziente dovrà completare un certo numero di esercizi di orientamento spaziale.
Un esercizio consiste nell orientarsi all'interno di una città nella quale dovrà ricercare e memorizzare la posizione di alcuni oggetti per poi recuperarli.
Vedendo la strada percorsa dal paziente un terapeuta può stabilire il suo grado di capacità cognitiva, e l'esercizio stesso attiva aree del cervello relative alla memoria spaziale.

Il lavoro di questa tesi si prefigge di sviluppare un'applicazione solamente per la generazione procedurale dei suddetti ambienti cittadini, sulla quale in un futuro potranno avere luogo diversi esercizi.
Gli psicologi del dipartimento di psicologia dell'università di torino ha già eseguito dei test di navigazione in realtà virtuale di alcuni soggetti sani per verificare l'utilità dell'esercizio; nell'applicazione precedente la città è stata fornita da un modellatore che ha seguito le indicazioni degli psicologi.

L'applicazione deve essere in grado di offrire ai ricercatori, libertà sufficiente per generare indipendentemente ambienti cittadini con diverse caratteristiche dal punto di vista funzionale e visivo, da poter somministrare ai pazienti.

Vantaggi della generazione procedurale

Un'applicazione terapeutica in realtà virtuale può essere pensata come l'unione di due moduli principali: Il codice, contenente la logica, e gli asset, contenenti i modelli 3D, le immagini e il resto delle risorse che contribuiscono al risultato visibile finale.
Il codice è scritto da sviluppatori, mentre la creazione degli asset viene affidata a dei modellatori 3D.
Tradizionalmente nel caso di una applicazione che richiede la navigazione di una città, al modellatore viene richiesto di creare i palazzi, le strade, e gli altri elementi visivi, che poi mette assieme manualmente all'interno di un editor per creare un ambiente cittadino.
Un ambiente cittadino statico però, nella maggior parte dei casi, non è sufficiente e pone grandi limitazioni.

- Scarsa varietà degli ambienti: i pazienti hanno bisogno di cure diverse a seconda dell'insulto mentale ricevuto, la difficoltà di esplorazione della città deve essere dunque personalizzata attorno alle capacità mentali del paziente, un ambiente cittadino statico può risultare di difficoltà inadeguata per il paziente e offre al terapeuta poca flessibilità.
- La modellazione di una mappa statica diversa per ogni esigenza comporta dei costi insostenibili, sia per il tempo impiegato dal modellatore, sia per il tempo impiegato dagli psicologi esperti in materia a spiegare nel dettaglio come deve essere fatta la città.
- Oltre ai costi di produzione, la creazione di mappe statiche richiede potenzialmente mesi di lavoro, compromettendo la praticità dell'applicazione.

è indispensabile che il terapeuta che assegna questi esercizi di esplorazione abbia uno strumento veloce e flessibile per generare città su misura di ogni singolo paziente.
Ciò viene realizzato dallo strumento di generazione procedurale e parametrica di ambienti cittadini descritto in questa tesi.
La struttura dell'applicazione è simile a quella tradizionale ma con un'importante differenza: gli asset contengono dei componenti visivi base per la creazione dell'ambiente, e la

costruzione finale della città viene affidata al codice che è in grado, a partire dagli asset, di comporre una città:

- Proceduralmente: ovvero il compito viene affidato al codice che grazie a delle procedure costruisce il risultato atteso.
- Parametricamente: le procedure che costruiscono gli ambienti cittadini ricevono dei parametri che decidono il tipo di città, offrendo una notevole flessibilità di creazione.
- In tempo reale: tutto ciò avviene al momento dell'apertura dell'applicazione, senza bisogno di aspettare.

Questo approccio riduce a zero i tempi di modellazione della città; inoltre applica i principi della letteratura scientifica riguardante la percezione della città dell'uomo in modo che anche i terapeuti che non abbiano conoscenze approfondite possano usare l'applicazione in modo efficace.

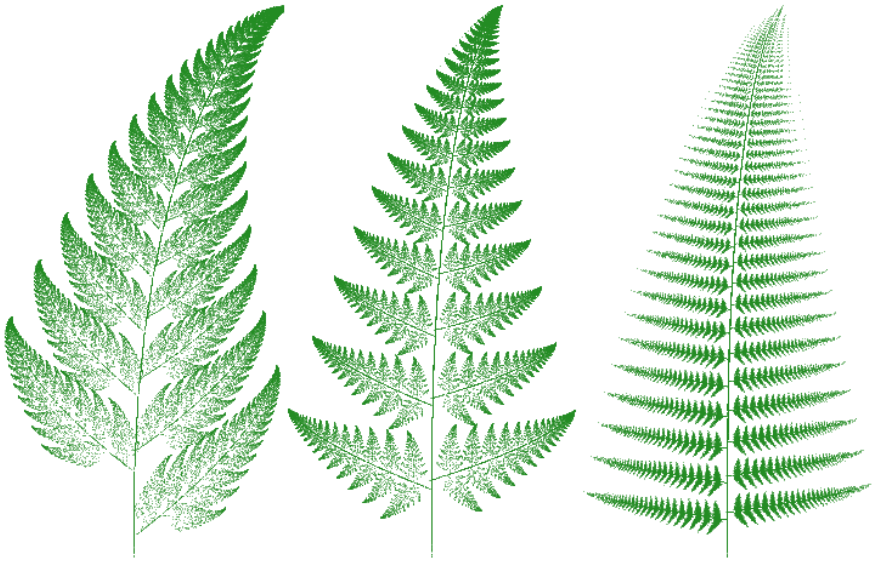
Stato dell'arte

Di seguito esploreremo una serie di tecniche e soluzioni usate per la generazione procedurale di ambienti 3D, e più nello specifico di città.
Parleremo di come con il giusto set di regole si possono generare strutture infinitamente grandi, partendo dai frattali, proseguendo a come gli L-systems forniscono una grammatica per l'espressione di tali regole, andremo a vedere come gli L-systems possono essere adattati alla generazione di reti stradali introducendo regole locali, e a come l'aggiunta di regole globali permette la generazione di reti verosimili, con una specifica struttura topologica.

Frattali per la generazione procedurale

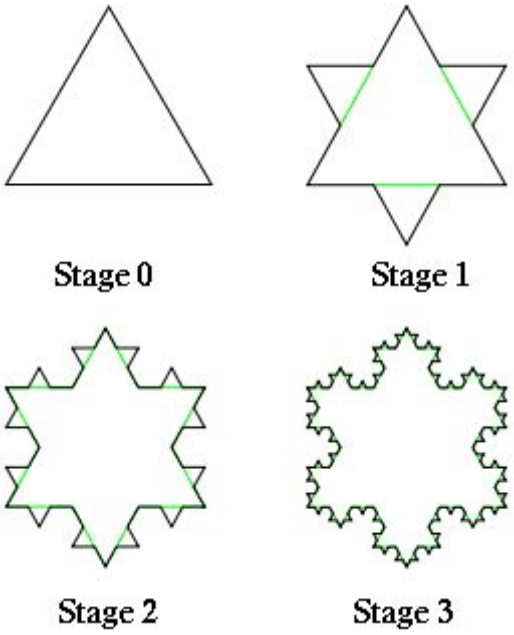
La parola frattale deriva dal latino fractus, ovvero rotto, il concetto è proprio quello di rappresentare figure con un alto grado di auto similarità, essi posseggono un livello di dettaglio infinito, ovvero più ci avviciniamo al frattale e più dettagli esso mostra.

Il livello di ricorsione di un frattale è in teoria infinito, ma essi possono essere rappresentati con semplici regole ricorsive, ciò li rende ottimi candidati per essere generati proceduralmente da un algoritmo.



Diversi tipi di felci frattali

I metodi geometrici tradizionali non rappresentano efficientemente elementi naturali, molti dei quali esibiscono pattern irregolari e auto simili, ovvero il loro aspetto rimane costante a diversi livelli di ingrandimento. come ad esempio cavolfiori, fiocchi di neve, catene montuose. queste forme possono essere descritte usando la matematica dei frattali.



koch snowflake

Mandelbrot fu il primo a usare computer per la visualizzazione di frattali complessi

Seppur molto potenti, I frattali si limitano a strutture auto simili, e per rappresentare strutture naturali nella computer grafica si tendono ad usare L-Systems.
Essi sono in grado di rappresentare strutture frattali, seppur avendo un grado di flessibilità molto maggiore

Lindemayer Systems

L'L-system, abbreviazione di sistema di lindenmayer è un sistema di riscrittura in genere ricorsivo, che fa uso di una specifica grammatica basata su stringhe.

Come i frattali, essi propongono un sistema grazie al quale con pochi parametri e regole in input si possono generare strutture potenzialmente infinite, sono dunque un scelta da considerare per la creazione procedurale di un ambiente cittadino.

Un L-System è formato da 3 elementi:

1. Un alfabeto
2. Una serie di regole di produzione
3. Un assioma.

L'alfabeto è una serie di caratteri, ogni carattere rappresenta un'istruzione di disegno particolare, le regole di produzione sono delle assegnazioni nelle quali da una lettera dell'alfabeto ne vengono prodotte altre, l'assioma è la stringa di partenza.

Seguendo le istruzioni dell'assioma si ottiene graficamente la prima generazione, applicando le regole di produzione sulla prima generazione di ottiene la seconda generazione e così via.

Ad esempio: consideriamo:

- Un alfabeto {A, B} nel quale A disegna una linea diagonale verso sinistra a partire dalla rotazione corrente, B disegna una linea diagonale destra, sempre coerente con la rotazione del nodo in cui si trova
- Delle regole del tipo: "A=AB"e "B=AB"
- Un assioma "AB"

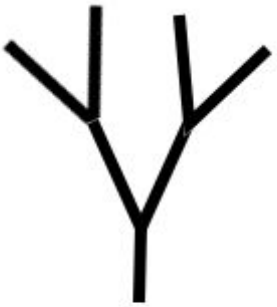
lo scopo è di, generazione per generazione, applicare le regole di produzione alla generazione precedente.

Nella prima generazione abbiamo l'assioma stesso: AB, che disegna due linee diagonali

AB

prima generazione: AB, due linee diagonali

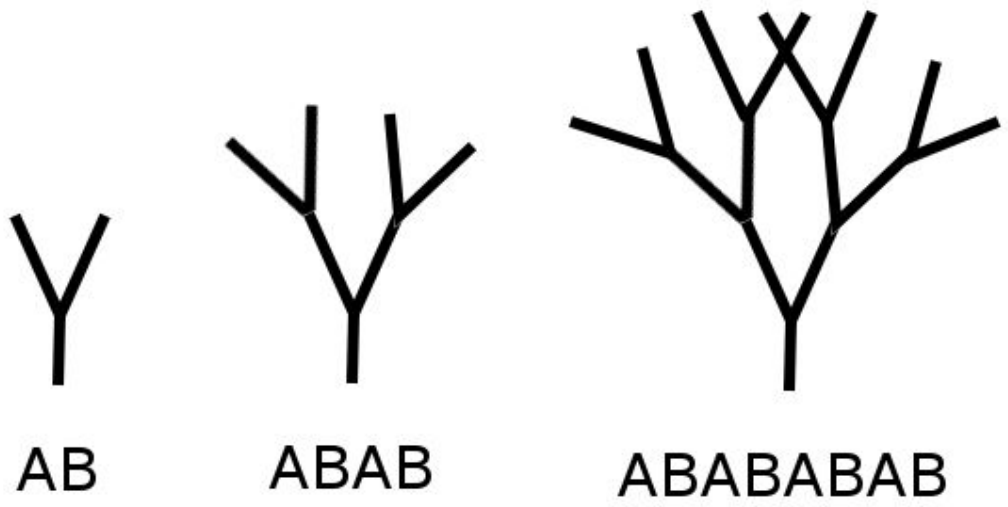
Nella seconda generazione, si applicano le regole di produzione allo stato corrente, dunque A diventa AB, e B diventa AB, ottenendo ABAB, che costruisce sopra alla prima generazione altre quattro linee diagonali



ABAB

Seconda generazione: ABAB

ABAB diventa il nuovo assioma al quale devono essere applicate le regole di produzione per arrivare alla terza generazione, applicate le regole si ottiene ABABABAB



Risultato di tre generazioni dell'L-system

L-systems e turtle graphics

Nell'esempio precedente ogni lettera dell'alfabeto rappresenta uno specifico disegno, traccia una linea, gli L-Systems possono fare uso del turtle graphics.

Turtle graphics è un tipo di grafica vettoriale che utilizza un cursore relativo, la tartaruga, con memoria di tre attributi: posizione, direzione e penna, essa a sua volta ha proprietà come colore, spessore, flag di acceso/spento.

Al cursore vengono impartiti comandi che vengono interpretati relativamente al valore degli attributi correnti, esempi di comandi possono essere:

- Vai avanti di 30 centimetri,
- Ruota di 90 gradi
- Accendi/spegni la penna.

Con semplici comandi di questo tipo si possono rappresentare figure complesse.

Nell'esempio seguente, la stella a nove punte è stata disegnata alternando i comandi:
vai avanti di 1, ruota di -100 gradi, vai avanti di 1, ruota di 140 gradi.

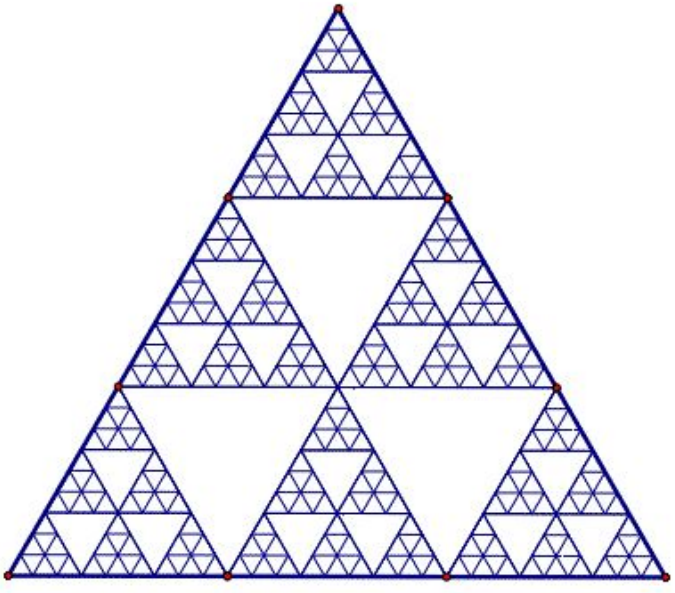
risultato di semplici istruzioni impartite al cursore tartaruga

Quando usati insieme a turtle graphics, gli L-Systems sono in grado di rappresentare strutture complesse con pochi semplici regole molto leggibili.

Usando L-Systems con turtle graphics si possono generare strutture frattali.
considerando:

- un alfabeto A,B,+,- nel quale A e B sono comandi di muoversi in avanti, + impartisce il comando di ruotare in senso antiorario di 60 gradi, - è una rotazione in senso orario di 60 gradi.
- delle regole A=B-A-B, B=A+B+A
- un assioma A

Si è in grado di rappresentare il frattale triangolo di Sierpinski.



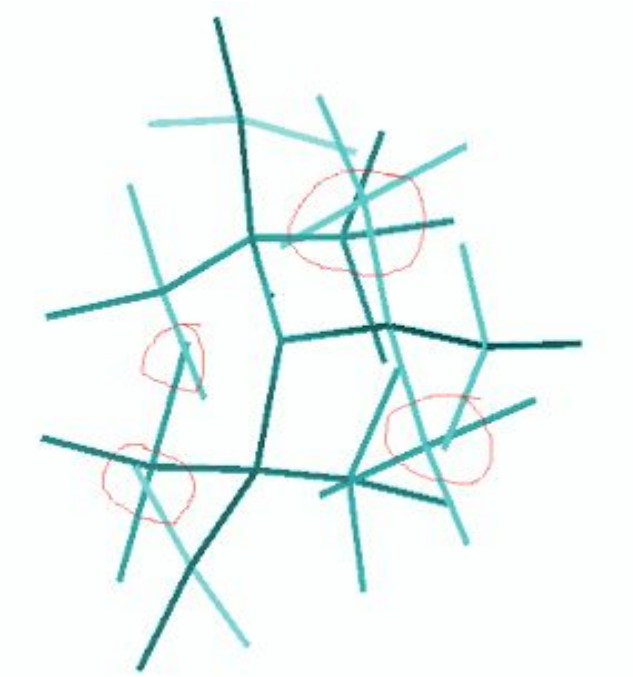
Triangolo di Sierpinski rappresentato con L-System e turtle graphics

L'L-system è stato inventato originariamente per la descrizione formale dei modelli di crescita delle piante e altri organismi naturali.
Oggi viene usato soprattutto per la generazione procedurale di piante e altri elementi naturali nella computer grafica.

Esempi di modelli 3D creati a partire da L-System

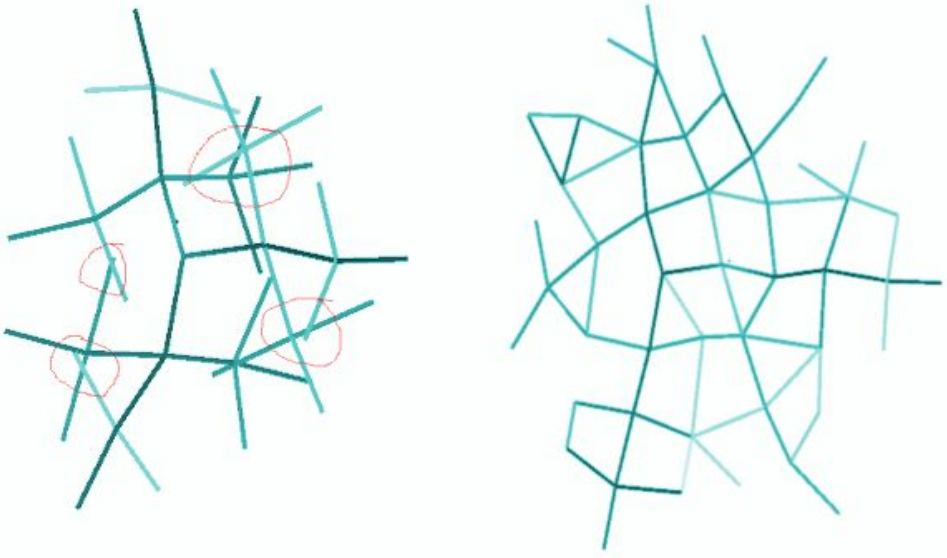
Self sensitive L-Systems per la generazione di reti stradali

Mentre la struttura di una pianta disegnata su un piano bidimensionale può sembrare una rete stradale, mancano delle regole che permettono al sistema di svilupparsi in maniera consapevole di se stesso, ovvero in maniera *self sensitive*, per evitare che le strade vengano prodotte una sopra l'altra e rompano le regole base di urbanistica.



Rete non self-sensitive

Questo approccio viene descritto e usato nella pubblicazione di Parish e Muller “Procedural Modeling of Cities” in cui, per ottenere questa proprietà addizionale, vengono introdotti “local constraints” che redirigono la creazione di un segmento di strada in modo da creare una rete stradale coerente.



A sinistra un normale L-system, presenta diversi errori stradali, a destra un self-sensitive L-system, genera una rete stradale coerente

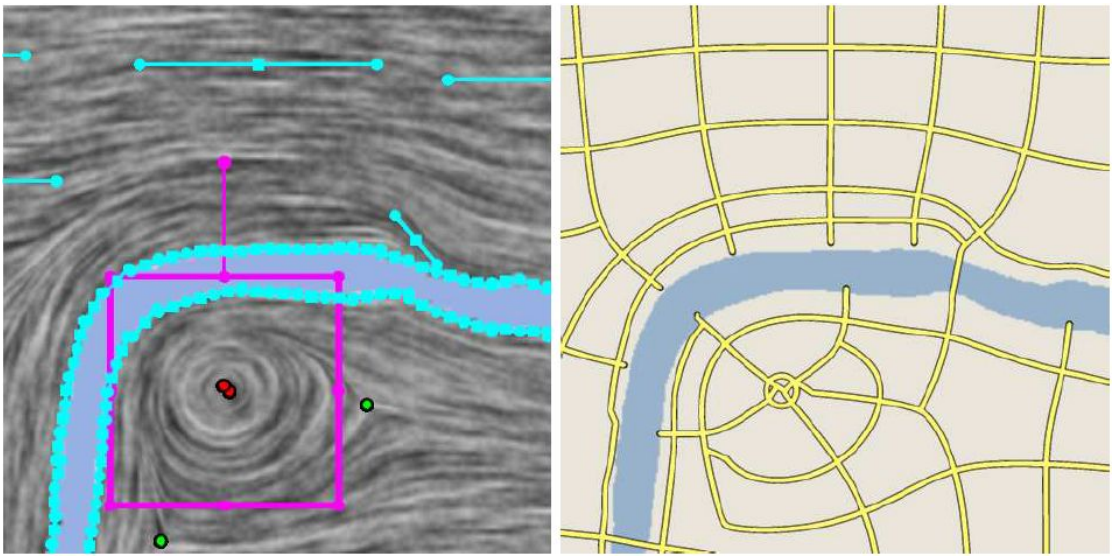
In effetti, la conoscenza del nostro grafo di strade di se stesso, porta diversi benefici oltre a quello precedentemente citato.

Questa self-sensitivity del sistema, sfruttata in un certo modo, porta alla definizione di regole locali di creazione, che, parametrizzate in maniera corretta possono introdurre della varietà desiderata.

Regole Globali per città verosimili

Creare una qualsiasi rete stradale coerente non basta come risultato finale.
come posso comunicare al mio modello il desiderio di avere una città che segua una struttura a cardo e decumano come New York? Oppure una mappa che segua un modello di cerchi concentrici come Parigi? o ancora una che segua il pattern di un incrocio a Y.
Questa richiesta infatti richiede l'applicazione di regole globali, che vengano applicate per dare una forma o uno scopo alla rete stradale.
L'utilizzo di regole globali in contemporanea alle regole locali dà la libertà di creare un vasto spazio di mappe, adatte a diversi tipi di sperimentazione.

il metodo usato in *Interactive Procedural Street Modelling* per questo scopo usa dei tensori. Essi generano dei campi vettoriali che guidano le strade. La creazione di un campo vettoriale però non è un modo semplice attraverso il quale un utente può definire la forma della città, dunque in questa applicazione verrà usato un approccio più semplice basato sulla direzione di strade principali.



Tensori per la creazione di reti stradali

Divisione in Lotti

In Parish e Muller "Procedural Modeling of Cities" dopo la generazione della rete stradale viene effettuata una successiva divisione degli isolati in lotti, ovvero gli spazi nei quali possono essere successivamente generati gli edifici.

divisione degli isolati in lotti

Dal punto di vista visivo, la divisione degli isolati in lotti permette di diversificare l'altezza, l'architettura e più in generale l'aspetto degli edifici appartenenti allo stesso isolato, rendendo la città più realistica, dando possibilmente ad ogni lotto una configurazione unica.

Sebbene ciò può sembrare un vantaggio, nel contesto in cui questa applicazione deve essere sviluppata fa perdere il controllo della difficoltà del percorso cittadino che si vuole creare,

poichè può introdurre involontariamente aree facilmente distinguibili l'una dall'altra, introducendo dunque landmark in maniera casuale.

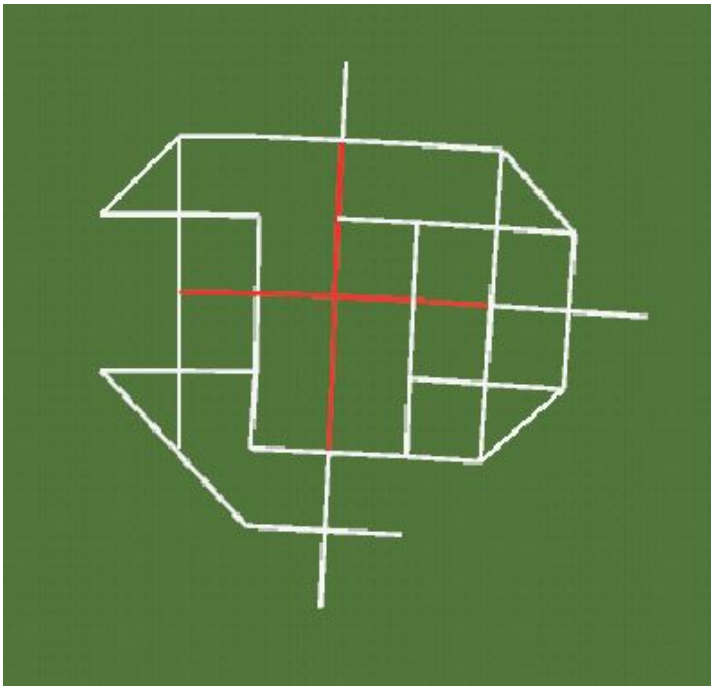
Per questo motivo nella seguente applicazione non viene effettuata nessuna divisione in lotti, ogni isolato costituirà un unica unità.

Generazione Città

La generazione della città e la sua seguente esplorazione è divisa in 3 passi.

Il primo passo è la generazione logica della rete stradale.

In questo passo vengono popolate le strutture dati che caratterizzano solo la rete stradale, principalmente rappresentata con un grafo.



Rete stradale

Il secondo passo consiste nella disposizione logica dei landmark.

Che vengono disposti in base ai parametri ricevuti in input, e mostrati con delle icone, per dare un'idea funzionale dell'ambiente cittadino.

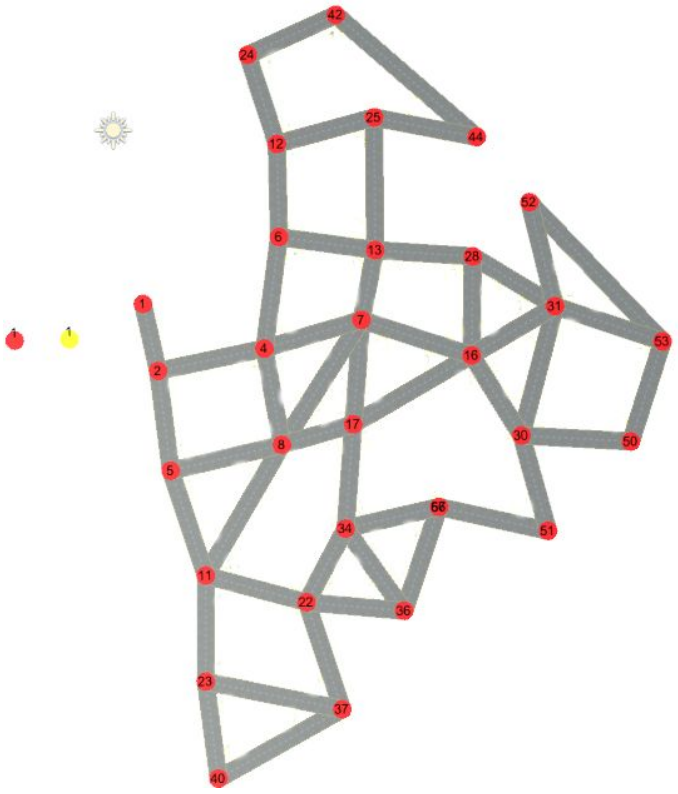
Rete stradale con icone indicanti i landmark

Il terzo passo è la generazione grafica della città, dalle strade e di tutti gli elementi visivi a partire dalle strutture dati popolate nei passi precedenti, a questo punto la città è pronta ad essere esplorata.

Generazione della rete stradale

Rappresentazione della rete stradale

La rete stradale viene rappresentata come un grafo non orientato nel quale i nodi del grafo rappresentano gli incroci, gli archi del grafo rappresentano i segmenti di strada.
Questo tipo di rappresentazione, seppur complessa da gestire matematicamente, ha un limite: le strade nella città possono solo essere formate geometricamente da spezzate, non è possibile la generazione di strade che presentano curve di qualunque tipo.



Grafo stradale

la rete stradale è rappresentata attraverso la classe Network, che contiene una lista dei nodi del grafo, e la classe Node, che rappresenta i nodi del grafo, e contiene una lista di tutti nodi vicini.

La classe Network, oltre a contenere informazioni di diverso tipo specifiche della rete stradale in questione, contiene la lista dei nodi, questo dato, può sembrare ridondante, ma assicura un facile accesso sequenziale a tutti i nodi del grafo.

La rappresentazione, seppur limitata, offre un ampio grado di libertà per quanto riguarda gli aspetti della rete stradale, che sono ritenuti importanti per l'esplorazione e la memorizzazione di ambienti cittadini: difficoltà topologica, visibilità dei landmarks, densità dei landmarks, grandezza della città.

Generazione di nuovi nodi

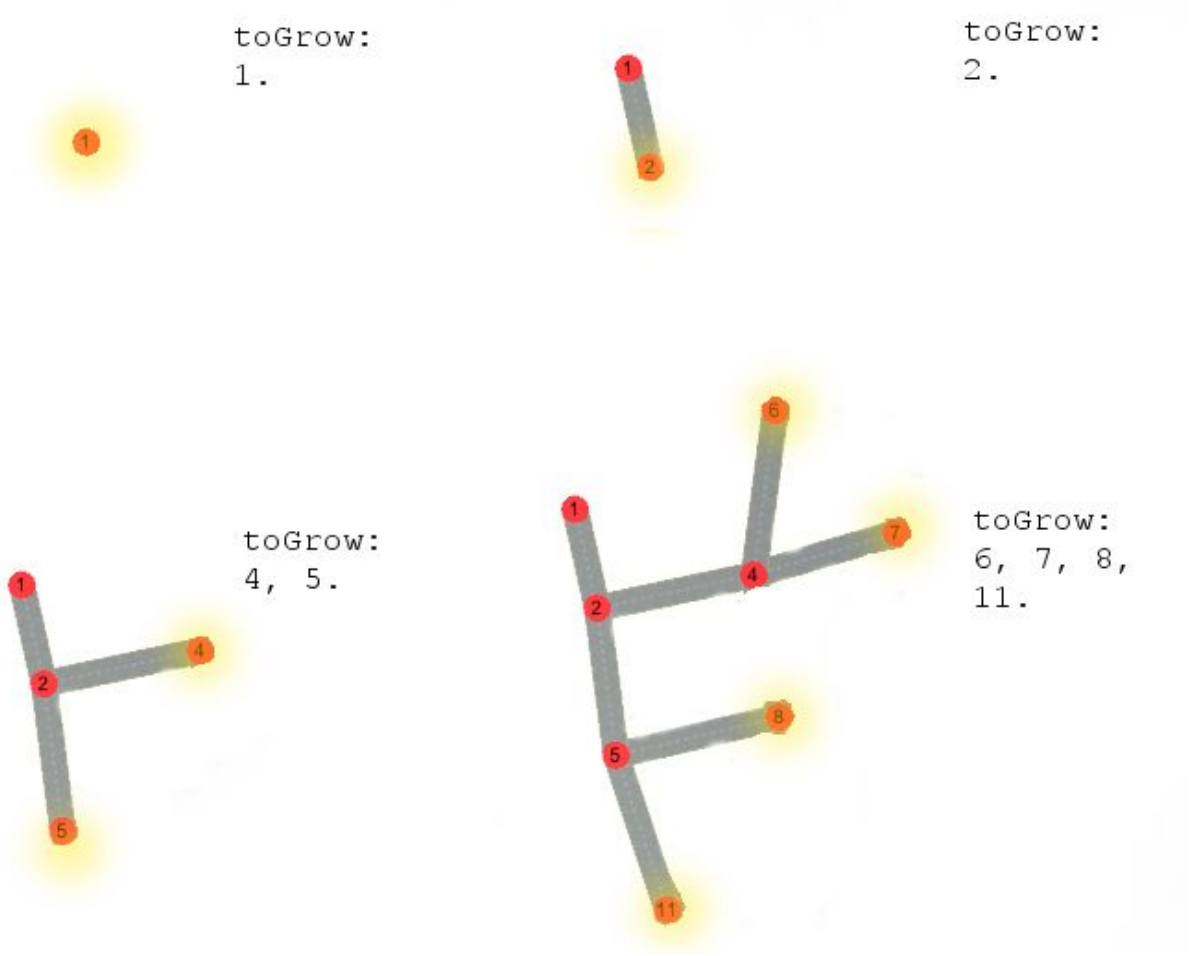
Dati i parametri, la rete stradale viene generata a partire da un singolo nodo padre, che costituisce il punto di origine della rete.
dal nodo padre parte lo sviluppo dei nodi successivi, che formano archi (segmenti di strada) partendo dal padre; a loro volta, da questi nodi vengono generati altri nodi con i rispettivi archi e così via.

Per la creazione degli ambienti cittadini, ho deciso di usare la proprietà di autogenerazione propria degli L-Systems, inserire delle regole aggiuntive che durante la creazione di una generazione siano in grado di redirezionare, o in alcuni casi impedire la crescita di alcuni archi, e sostituire la rappresentazione in stringhe con un grafo, indispensabile per le operazioni successive.

Per implementare questo meccanismo di autogenerazione, viene usata una coda di nodi che contiene i nodi periferici che devono ancora crescere.
All'inizio, il nodo padre viene creato e posizionato manualmente nello spazio.

esso viene aggiunto nella lista dei nodi della rete, e aggiunto alla coda di nodi dal quale ne nasceranno altri (toGrow).

la funzione generateNetwork è responsabile di prelevare dalla coda dei nodi da crescere un nodo e generare da esso altri nodi e archi la cui quantità e posizione dipende dai parametri iniziali di creazione mappa, una volta stabilita la posizione generateNetwork chiama la funzione grownode_color, che prendendo come argomento il nodo di partenza, la posizione del nodo che si vuole generare e il colore della strada(per debug), crea un nodo in quel punto e lo collega al nodo di partenza con un arco.



Crescita di una rete stradale usando una coda di nodi

Dentro generateNetwork la funzione grownode_color viene chiamata tre volte per formare un incrocio assieme al segmento di partenza.

In assenza di valori di offset casuali questo produrrebbe un incrocio perfetto, che ripetuto abbastanza volte produrrebbe una griglia regolare.

è qui che entra in gioco la parametricità che combinata con una funzione random permette di ottenere le mappe desiderate, e sempre diverse.
la variabile *angle_variation* nella funzione grownode_color, se maggiore di 0 assicura che gli angoli tra le strade non siano 90° , ma che a seconda del bisogno possono essere creati incroci irregolari.
la variabile *road_lenght_variation* se maggiore di 0 assicura che i segmenti di strada non siano tutti uguali, introducendo un fattore di difficoltà in più.

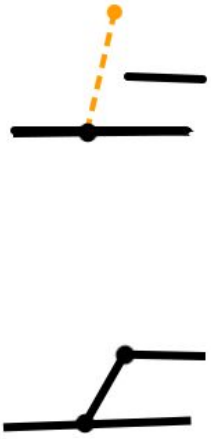
Regole locali:Algoritmo di Snapping

L'algoritmo di snapping determina le regole locali che rendono il nostro L'system self sensitive.
Esso è responsabile di redirezionare un nodo che sta per essere creato in modo da tenere la coerenza, viene chiamato algoritmo di snapping, Esso
L'algoritmo di snapping viene eseguito durante la creazione di ogni nodo.
Dato il nodo di partenza e quello aspirante di arrivo, controlla che il nodo di arrivo rispetti le regole locali della rete e, in caso negativo, redireziona il nodo oppure lo cancella.

L'algoritmo opera come segue: esso controlla che la posizione del nodo è coerente con la rete esistente, in caso positivo ritorna senza far niente, altrimenti riconduce la situazione ad una di queste tre violazioni:

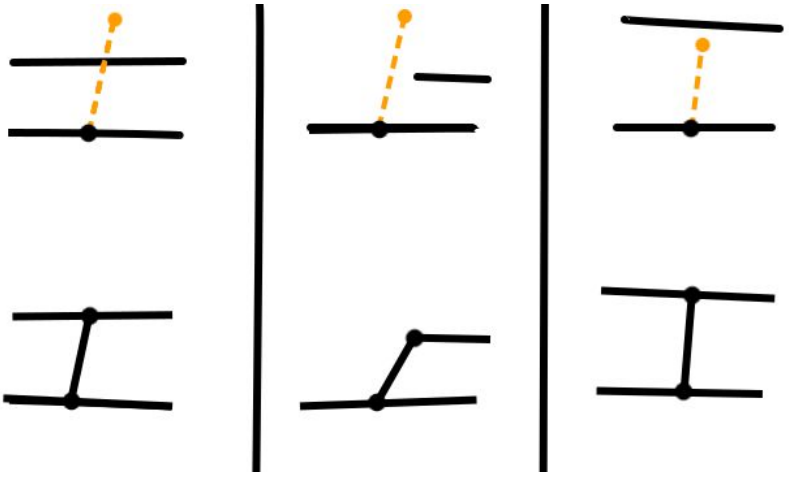
- Sovrapposizione: il nodo creato genererebbe un arco che si sovrappone ad un arco esistente: in questo caso l'arco esistente si divide in due segmenti, e il nuovo nodo diventa il nodo che separa i due segmenti, questo evita che le strade si calpestino a vicenda.

- Vicinanza di nodo: il nodo creato genererebbe un arco che si trova troppo vicino ad un nodo esistente, questa situazione a seconda della distanza può generare città con reti poco realistiche in cui si trovano strade che chiaramente potrebbero essere collegate ma non lo sono, l'algoritmo di snapping fa corrispondere il nuovo nodo con il nodo vicino.



- Vicinanza di segmento: il nodo generato si trova molto vicino ad un segmento esistente, anche questo caso genera strade separate di poco, che chiaramente dovrebbero essere unite, la soluzione è quella di far incrociare l'arco da generare con l'arco vicino per formare un incrocio a T.

In presenza di più di uno di questi eventi, viene considerato solo quello più vicino al nodo di partenza.



sinistra: sovrapposizione, centro: vicinanza di nodo, destra: vicinanza di segmento

l'output dell'algoritmo di snapping è un nodo, che si trova vicino alla posizione desiderata, ma segue le regole di una rete stradale tradizionale.

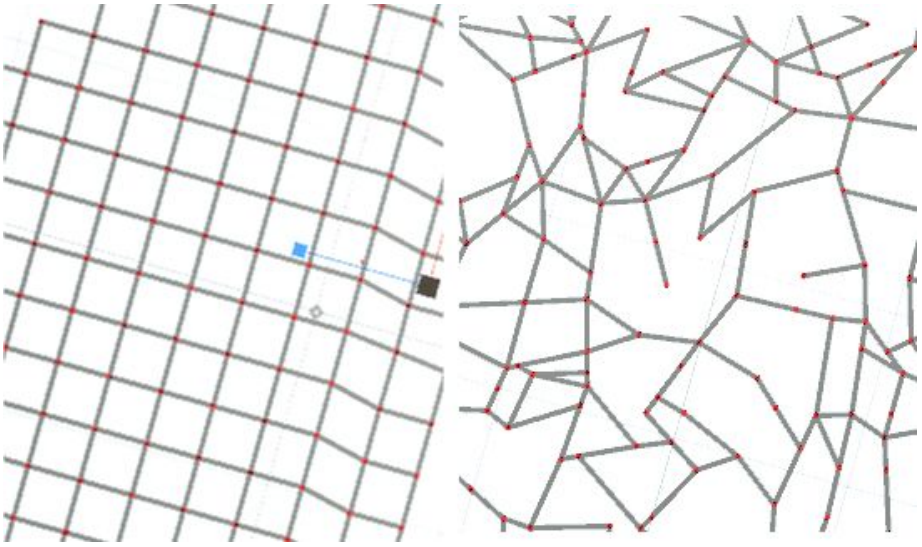
anche nel processo di snapping sono presenti delle variabili casuali parametrizzate che hanno come scopo la parametrizzazione e variabilità della città.
esse sono:

- *connectivity* che nel caso in cui debba avvenire uno snapping può negare l'operazione facendo in modo che nessun nuovo segmento di strada venga generato, abbassando questo valore si fa in modo che il grafo stradale sia meno connesso, il valore a 1

genera un grafo di tipo lattice, molto più facile da esplorare, un valore uguale a 0 genera un albero.

- *snapsize* è un valore tolleranza, indica la distanza sotto al quale vengono considerati gli eventi di vicinanza di nodo e vicinanza di segmento.
- *min_angle* è l'angolo sotto il quale due strade non possono esistere perché creerebbero una curva troppo spigolosa.

Tutti i parametri soprastanti sono raccolti nella voce *difficoltà topologica secondaria* nella GUI.



sinistra: strada facile topologicamente, destra: strada difficile topologicamente.

Nel caso pratico dell'immagine, si può notare come tutti i fattori che rendono facile la navigazione, sono presenti nella parte sinistra:

- incroci regolari a 90°, la città ha solo due direzioni principali, ogni strada può essere solo parallela o perpendicolare a qualsiasi altra strada.
- grafo di tipo lattice molto connesso, è facile passare da qualsiasi punto della città a qualsiasi altro punto.
- direzione delle strade sempre uguale, le strade hanno una forte identità.
- lunghezza dei segmenti di strada sempre uguale.

Nell'immagine a destra invece:

- gli incroci sono raramente regolari, sono presenti anche numerosi incroci a tre, gli angoli non sono mai di 90°.

- il grafo stradale è poco connesso, per passare da un punto all'altro della mappa si possono usare molti meno percorsi, questo forza il paziente a memorizzare numerosi percorsi e a creare una mappa mentale.
- le strade sono spezzate imprevedibili che non seguono mai una direzione coerente, e non portano da nessuna parte in particolare, esse hanno poca identità.
- la lunghezza dei segmenti di strada è fortemente variabile, ciò rende difficile per l'utente misurare le distanza percorse mentalmente.

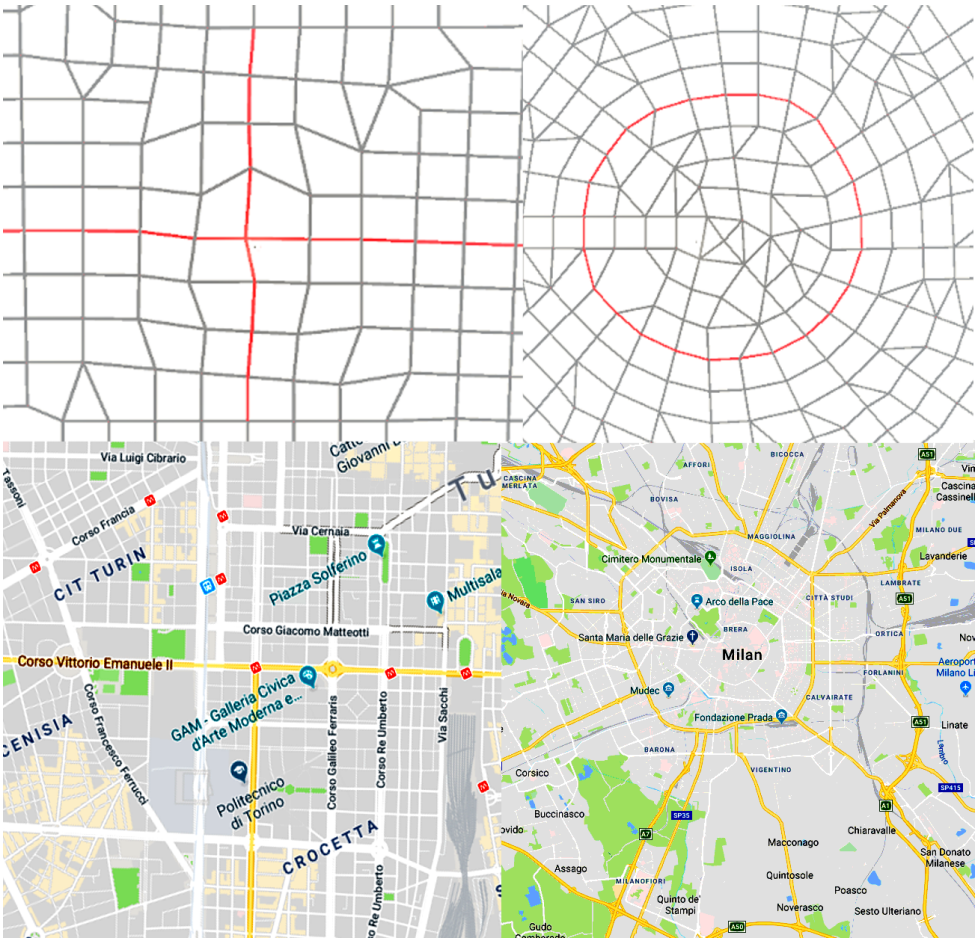
Regole globali e strade principali

Applicare regole locali permette la generazione di una rete stradale coerente, l'applicazione di regole globali invece è quello che permette di dare alla città una certa forma o una certa direzione di sviluppo.

In questo progetto le regole globali sono applicate ad ogni arco creato grazie ad un attributo di *Node* chiamato *direction,* esso indica la direzione che la strada segue per arrivare a quel nodo,

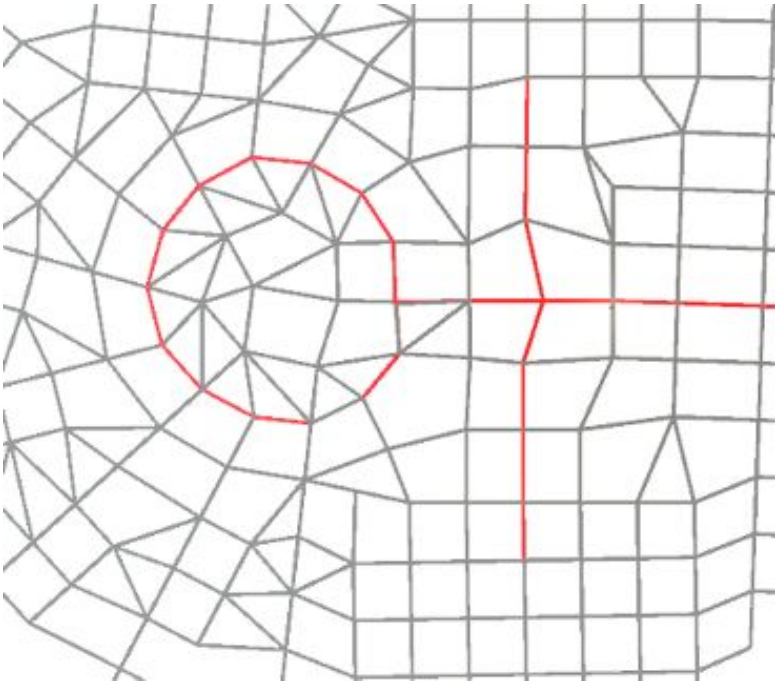
ovviamente è un valore valido solo se il nodo è nella coda di crescita*,* dunque ha solo un nodo vicino, che è quello che lo ha generato.

questo attributo di direzione è importante poiché, costruita una rete iniziale principale (un cardo e un decumano ad esempio) tutte le strade create dopo seguiranno quel pattern grazie al valore di direzione imposto dalla rete principale.



Sinistra: pianta romana, destra: pianta circolare (Torino e Milano)

Nella figura è colorata in rosso la rete principale, e in grigio quella secondaria.
è da notare che la rete principale non è solo un incrocio principale della città, ma è responsabile di dare la direzione di sviluppo (regola globale) dell'intera città.



Esempio di pianta mista

Questo sistema non pone alcun tipo di limitazione sul pattern della città, che può anche seguire forme diverse, nel caso di sopra la rete principale è composta da un cerchio e una retta, in questo caso le strade secondarie sono influenzate dalla parte di rete primaria più vicina in quel punto.

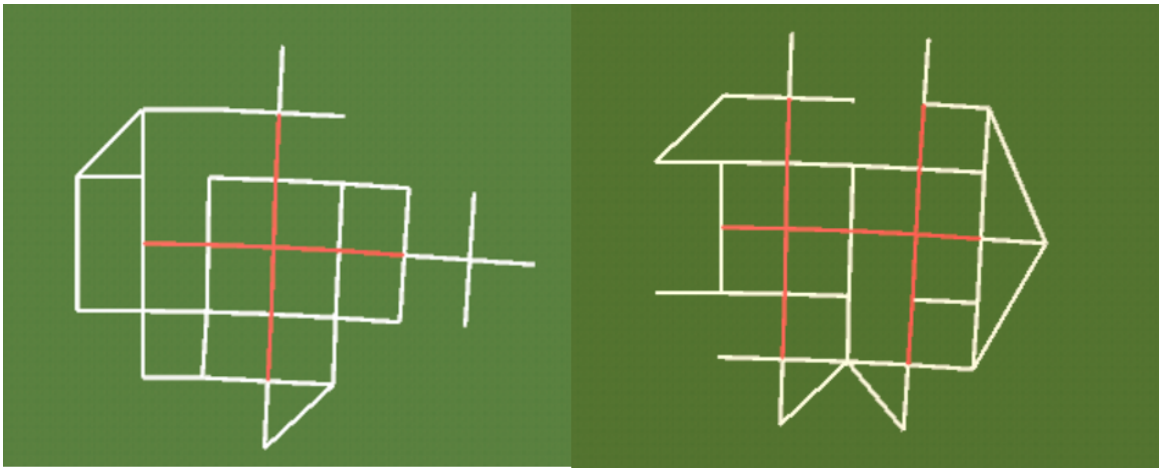
Le parametrizzazioni esposte fino ad ora sono sufficienti a descrivere la difficoltà topologica di una città.
per difficoltà topologica si intende la difficoltà di navigare un ambiente, data solo dalla morfologia delle strade.
Il libro *The image of the city* di Kevin Lynch descrive i parametri legati alla topologia che influenzano la difficoltà di navigazione di una città.

Tra questi, i più importanti per l'aspetto topologico sono:

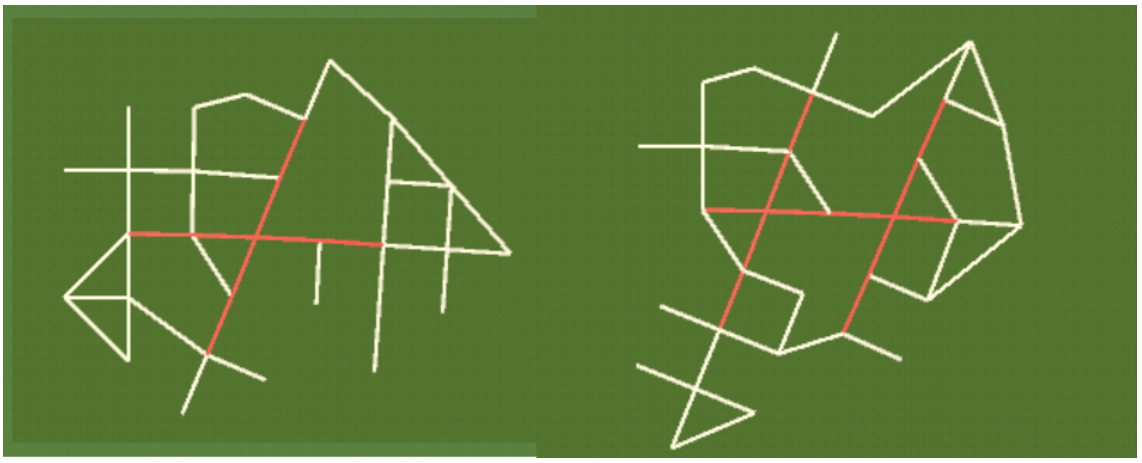
- Quanto la città è connessa, ovvero la quantità di percorsi esistenti tra un nodo e l'altro della città.

- Quanto ortogonali sono gli incroci, in presenza di incroci non perfetti, è meglio un incrocio a X imperfetto, rispetto ad un incrocio a Y geometricamente perfetto, questo perché l'incrocio a Y cambia nettamente la direzione e l'identità delle strade che si incrociano.
- Direzione delle strade: per la psicologia umana, una strada è un qualcosa con una direzione, o in alternativa qualcosa che porta a una particolare destinazione, una strada che curva lentamente cambiando direzione disorienta significamente l'utente, ed è più difficile di una strada che procede a zig-zag mantenendo la stessa direzione generale.
- Quanto regolare è il pattern di ripetizione delle strade, ad esempio gli isolati si ripetono ad una cadenza fissa.

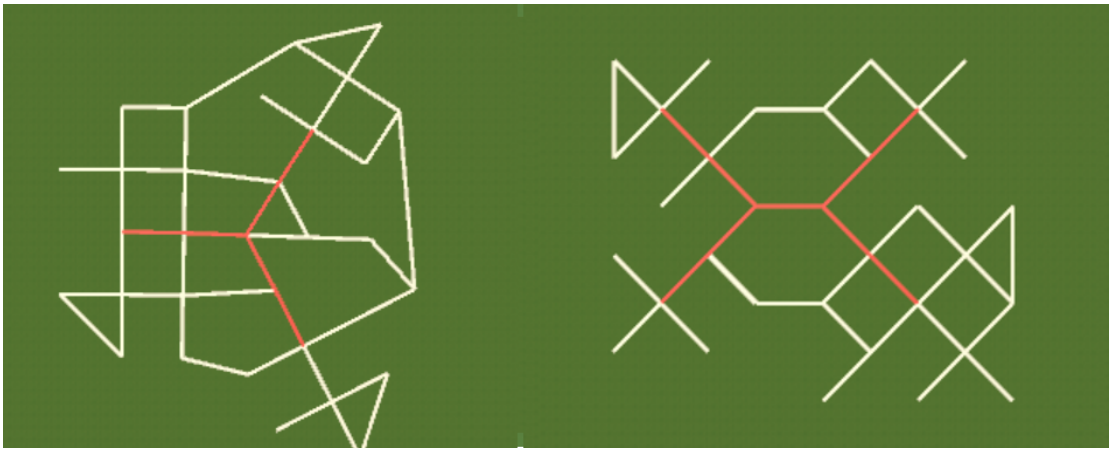
Dunque a Partire da questi assiomi l'applicazione utilizza diversi pacchetti di regole globali che rappresentano livelli di difficoltà topologica crescente.
Esse sono:



Croce, doppia croce



Croce storta, doppia croce storta



Ypsilon, doppia ypsilon

Essi sono generati nell'applicazione attraverso specifiche funzioni, ad esempio: PrimaryGenerationCross, PrimaryGenerationDoubleCross , eccetera…
Ogni funzione, genera una rete stradale principale, che imposta, in ogni punto del grafo, la direzione che le strade secondarie seguiranno, essa è registrata nella variabile del noto chiamata *direction.*

Generazione visiva della città

La generazione del grafo di strade è il più alto livello di astrazione della città, sono le fondamenta sulle quali tutti gli elementi visivi si posizionano e concretizzano.
è proprio grazie a questi elementi che riusciamo a passare da una mappa ad una città vera e propria.

Gli elementi visivi che compongono la città sono dunque:

- **Palazzi semplici**: che costituiscono il grosso della città, essi devono essere realistici e abbastanza vari, ma non devono avere particolarità che permettono all'utente di distinguerli l'uno dall'altro: essendo la priorità dell'applicazione il controllo sulla navigabilità e non sul realismo, la città potrà sembrare monotona, questa monotonia è il prezzo da pagare per avere la sicurezza che l'utente utilizzerà come punti di riferimento per orientarsi, solo gli elementi decisi dal terapeuta prima di cominciare l'esercizio.

- **Strade, marciapiedi e aiuole**: questi tre elementi costituiscono l'area della città che l'utente è in grado di percorrere, le strade possono essere primarie o secondarie, le strade primarie sono grandi il doppio di quelle primarie, e contengono elementi che le contraddistinguono, come ad esempio file di alberi, lampioni, cespugli, eccetera.

- **Landmarks:** I landmarks sono elementi della città che risultano facili da distinguere da tutto il resto. La loro funzione è quella di essere un punto di riferimento per l'esploratore, essi sono fondamentali per la memorizzazione di una città.

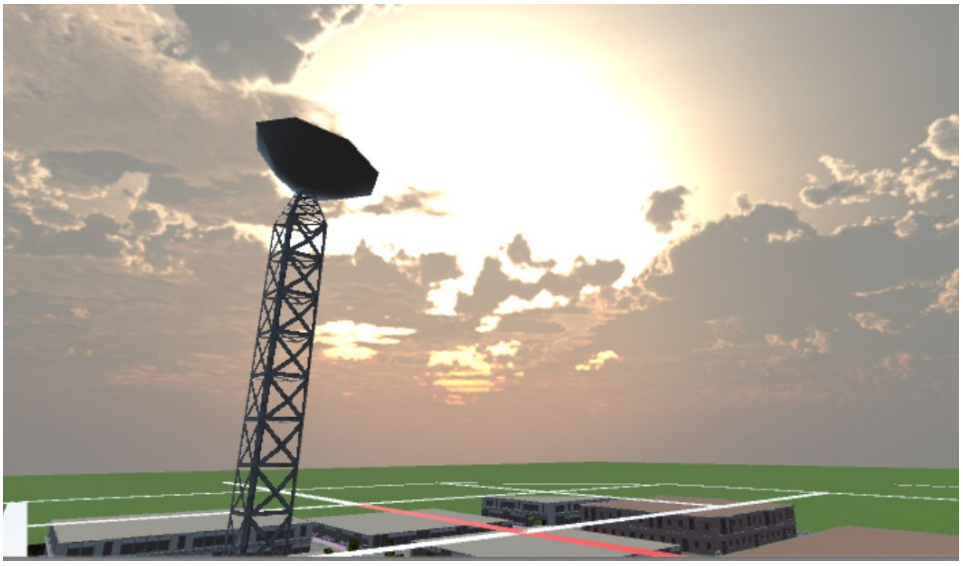
  Il palazzo più alto della città può essere considerato un landmark perchè ha una caratteristica che lo distingue dagli altri palazzi, si può dire lo stesso di un palazzo di dimensioni normali ma con un altro colore, o con una forma diversa.

  Edifici a parte, in una città in cui i palazzi sono visivamente monotoni, qualsiasi elemento visivo che si distingua, da uno straccio poggiato sul marciapiede, ad un albero, ad un'automobile parcheggiata, sono tutti elementi distinguibili che aiutano l'utente ad orientarsi

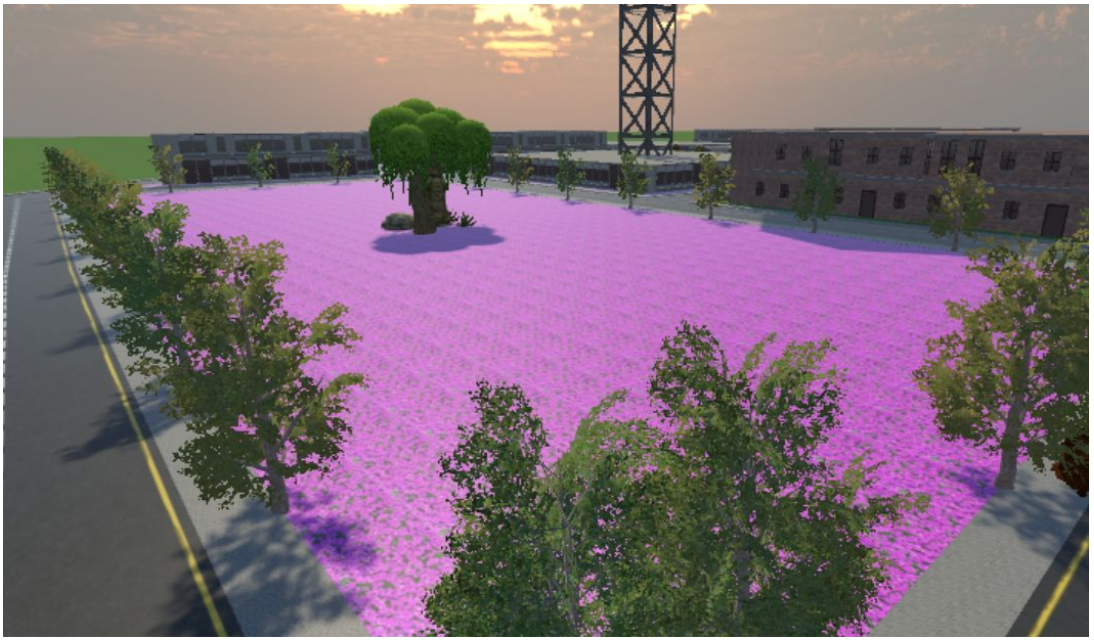
  - **Prossimali:** sono I landmark di dimensione per le quali non sono visibili da grandi distanze, essi sono: piazze, negozi, automobili, idranti e altri elementi piccoli presenti in una città
  - **Distali:** sono landmark che possono essere facilmente essere visti da una discreta distanza, principalmente si tratta di edifici e costruzioni alte.

    La distanza a cui possono essere visti è un parametro della generazione della città, strettamente correlato con la facilità di navigazione.

- **Quartieri**: I quartieri costituiscono un elemento molto importante per facilitare l'utente ad orientarsi, essi sono aree della città con delle caratteristiche comuni, le caratteristiche possono essere il tipo di architettura, il colore dei palazzi, la grandezza delle strade, odori particolari, suoni particolari, eccetera… essi dicono all'esploratore in maniera molto palese, grossolanamente in che area della città si trova.



- **Cielo:** il cielo può aiutare un esploratore a capire come è orientato se contiene informazioni di direzione, ad esempio un tramonto segna una direzione in tutta la città in due modi: proiettando ombre sull'asfalto tutte nella stessa direzione, e mostrando un gradiente arancione, blu, dalla direzione del sole a quella opposta.



- **Piazze** le piazze possono essere considerate dei landmark prossimali, in quanto sono elementi distinguibili della città, ma non si possono vedere da troppo lontano.

Riconoscimento di Isolati, Minimal Cycle Basis

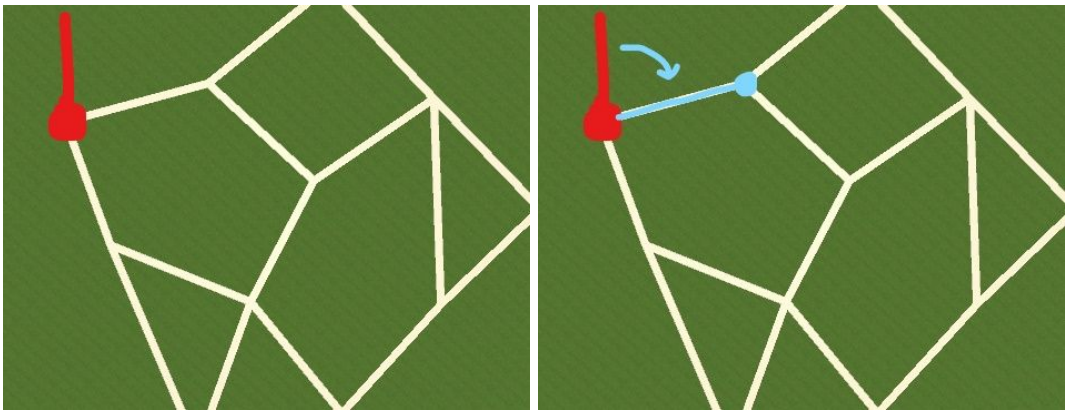
Un passo intermedio da fare alla struttura dati della rete, prima di generare tutti gli elementi grafici è il riconoscimento degli isolati, infatti senza riconoscere di essi non sarebbe neanche possibile la generazione delle strade, men che meno quella dei palazzi e delle piazze.

Un isolato è un concetto facile dal punto di vista della visione umana, che purtroppo risulta in una descrizione verbale e matematica complessa: Un isolato è una parte poligonale di città i cui lati sono segmenti di strada, e che non è attraversata internamente da strade.

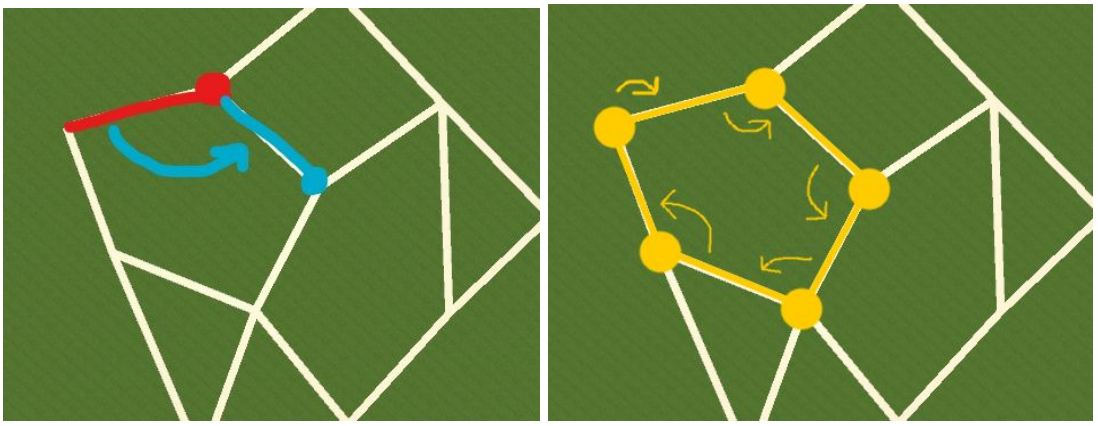
Gli isolati della città corrispondo ai cicli minimi del grafo. ovvero i cicli che non contengono cicli al loro interno.

L'algoritmo per trovarli si chiama Minimal Cycle Basis.
In questo caso, ad ogni nodo del grafo è associata un vettore posizione bidimensionale, ciò permette di semplificare di molto l'algoritmo.
Dunque dato un grafo i passi sono:

1. Prelevare il nodo più a sinistra (usando i vettori posizione)
2. Tracciare un arco verticale dall'alto verso il nodo prelevato
3. Selezionare il prossimo nodo procedendo in senso orario.
4. Selezionare il prossimo nodo procedendo in senso antiorario.
5. Ripetere il passo 4 finchè non si ritorna al nodo di partenza, i nodi così selezionati faranno parte dell'isolato, Rimuovere gli archi dell'isolato prima di ritornare al punto 1 per il rilevamento del prossimo isolato



A sinistra il passo 1 e 2, a destra il passo 3



A sinistra il passo 4 e a destra il passo 5

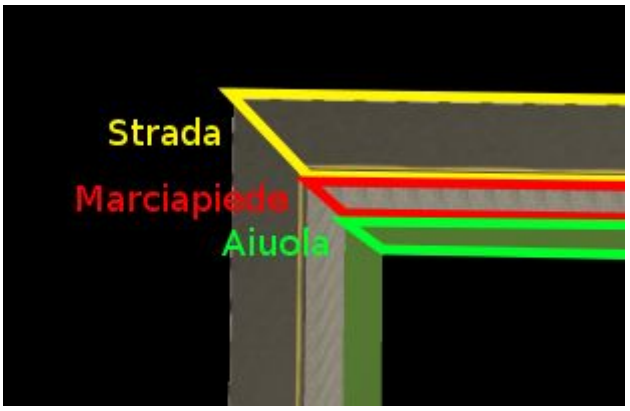
Dopo la generazione della rete stradale viene eseguito l'algoritmo di minimal cycle basis finchè non si esauriscono i nodi della rete stradale, Ogni Isolato viene dunque rappresentato da un vettore di nodi della rete, tutti gli isolati vengono raccolti in una lista per poi usarli per generare la città graficamente

Costruzione di strade marciapiedi e aiuole

Una volta estratti gli isolati, si possono cominciare a generare graficamente le strade, i marciapiedi e le aiuole.

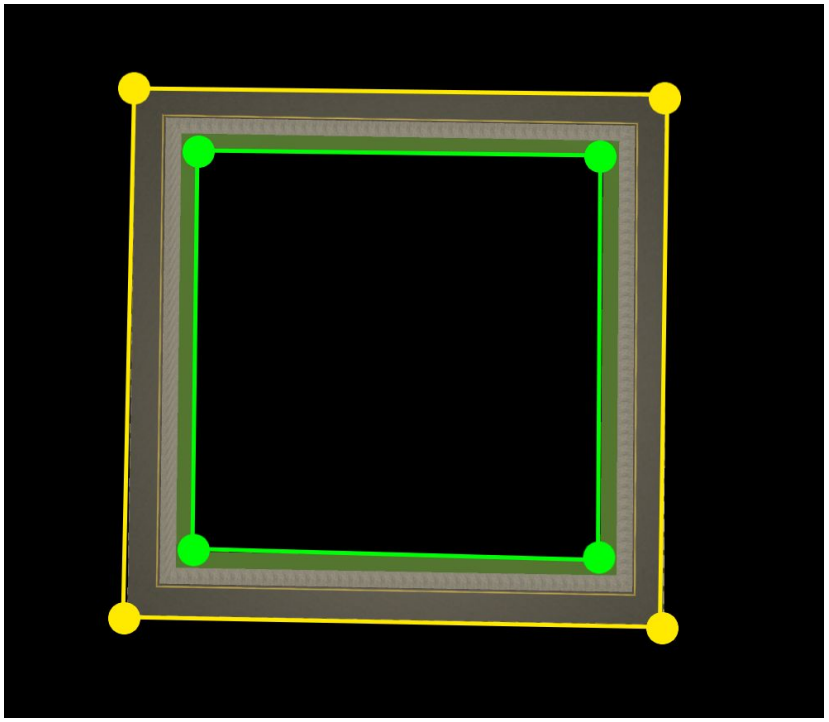
Una mesh viene generata per ogni lato di ogni segmento di strada, ottenendo dunque dei romboidi.

all'avanzare della generazione, romboidi condividono tra di loro molti vertici, questo permette una transizione grafica priva di artefatti dovuti ai lati dei romboidi.



Mesh romboidi unite tra di loro per formare la strada

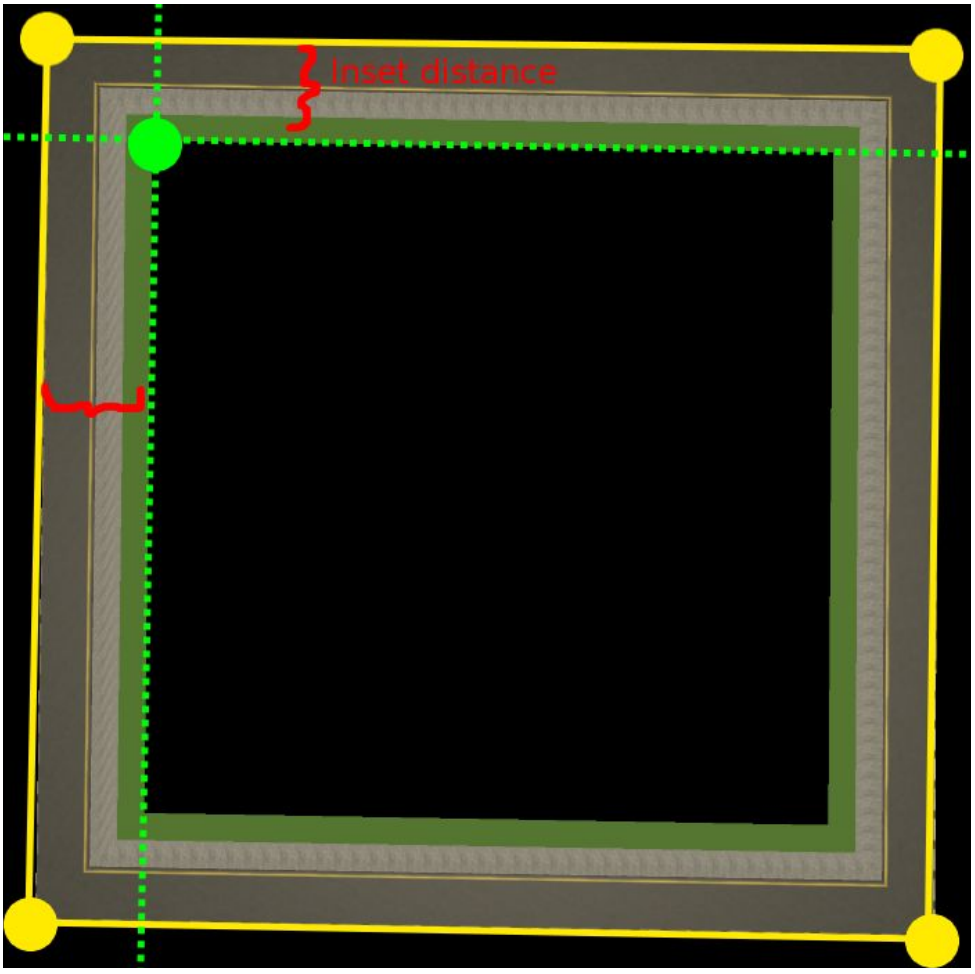
Per generare queste mesh romboidi però, serve un altro set di vertici 2D, all'interno di quelli che definiscono l'isolato, per formare così dei poligoni concentrici all'interno del poligono dell'isolato



le mesh stradali vengono disegnate grazie all'ausilio di poligoni concentrici

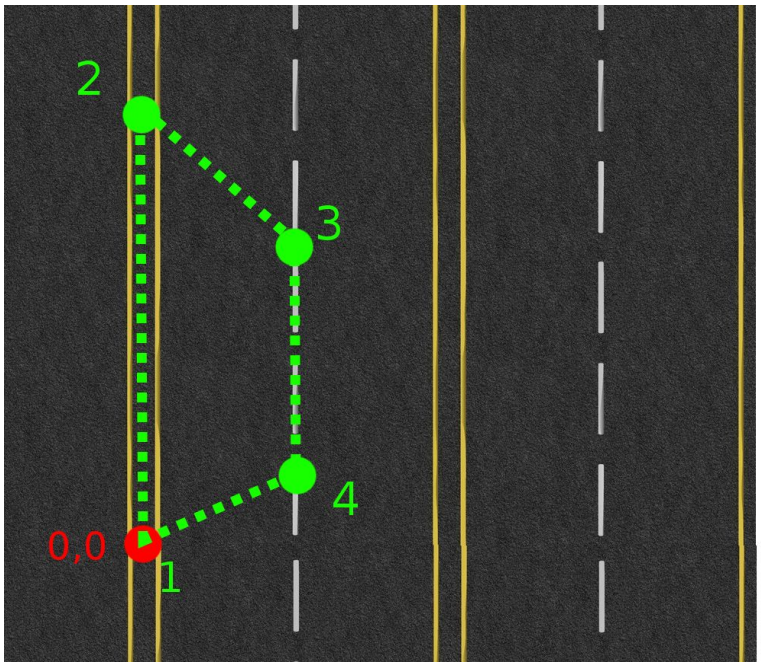
L'algoritmo di *polygon inset,* è quello che crea questi poligoni concentrici, esso opera generando un punto per volta, per generare il punto di inset di un altro punto esso considera due rette: una che parte dal punto precedente e arriva al punto corrente, l'altra che parte dal punto corrente e arriva al punto seguente, le rette sono spostate verso l'interno dell'isolato di un offset che determinerà la larghezza della strada.

Il punto in cui si incrociano queste rette è il punto di inset del punto corrente considerato, questi passi vengono eseguiti per tutti i punti dell'isolato per ottenere il poligono concentrico interno ad esso.



il punto di incontro delle due rette genera il punto di inset.

Una volta ottenuto il poligono concentrico, per ogni romboide generato dai due poligoni viene creata una mesh, assegnato un materiale, e infine alla mesh viene applicato un UV mapping dipendente dalla direzione, che viene ricavata dall'ordine dei poligoni.



UV-mapping dei romboidi di strada

Essendo che tutti i romboidi di strada vengono generati coerentemente con la direzione, alla fine, dopo che vengono generati tutti i romboidi, si ottiene un risultato visivamente coerente.

Costruzione di un palazzo

In questa applicazione, se si intende riempire un isolato, lo si può fare con una piazza, una costruzione landmark, o un semplice palazzo, questo capitolo si concentra sulla costruzione dei semplici palazzi a partire dalle informazioni di un isolato.
La costruzione di un palazzo, parte dunque con la definizione di un'area poligonale nell'ambiente 3D che corrisponde all'area del palazzo, essa è rappresentata da una lista di punti, che sono gli angoli del poligono.
Prima della costruzione del palazzo nell'oggetto dell'isolato interessato vengono calcolati i punti del poligono più interno con l'algoritmo di inset, essi sono immagazzinati in una lista di istanza dell'isolato.

Come spiegato in precedenza, gli asset dell'applicazione contengono solamente dei blocchi base con cui comporre gli edifici, non gli edifici stessi.

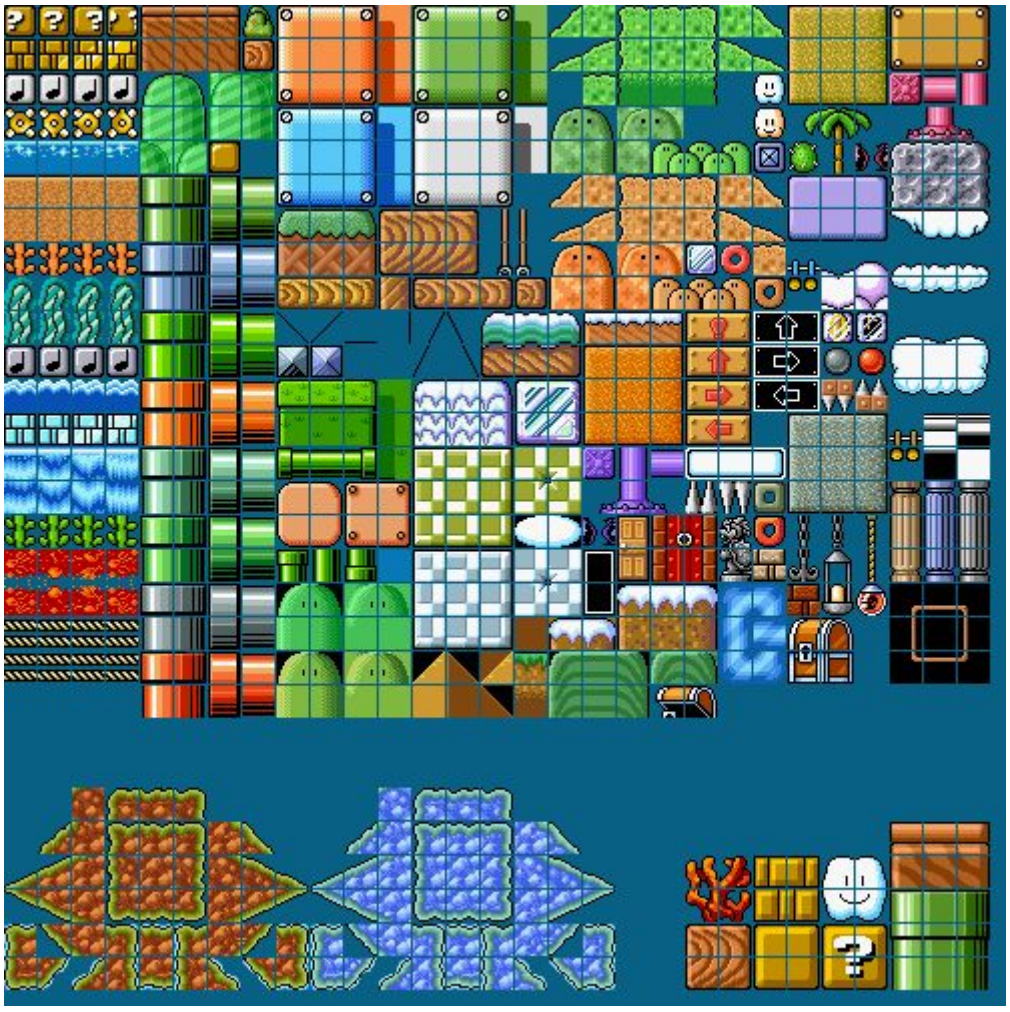
Questo approccio offre la possibilità di costruire edifici realistici ma monotoni in real time, senza l'aiuto di un artista 3D.

Le immagini seguenti mostrano una sequenza di blocchetti di base e l'edificio che risulta dall'unione dei blocchetti:

Tiles per la generazione di un palazzo.

Palazzo risultante dalla ripetizione di 4 tiles.

Questa tecnica di mettere insieme tanti blocchetti noti per creare diverse configurazioni viene chiamata tiling, ed è molto usata nei videogiochi 8-bit, ad esempio Super Mario Bros, nel quale una manciata di tiles (blocchetti) può creare una grandissima varietà di scenari.



Tiles in Super Mario Bros. 3

l'applicazione genera una facciata di un edificio con una combinazione di tiles.

Dati i vettori di (tiles, tiles per il primo piano, punti del poligono che delimita il palazzo, altezza, id e vettore di landmark prossimali) viene generato graficamente l'edificio,
Le tiles per il piano terra sono diverse dalle altre per permettere la creazioni di edifici con portoni, dunque più realistici, e per permettere il collocamento di facciate di negozi al primo piano.

Il poligono viene diviso in segmenti e per ogni segmento viene generata una facciata:
A sua volta la facciata viene costruita come una serie di file di tiles, fino a raggiungere il numero di piani desiderato.

ogni piano viene costruito disponendo le tiles una dopo l'altra per creare un piano di un palazzo.

Componenti di un palazzo

Unity e Prefabs

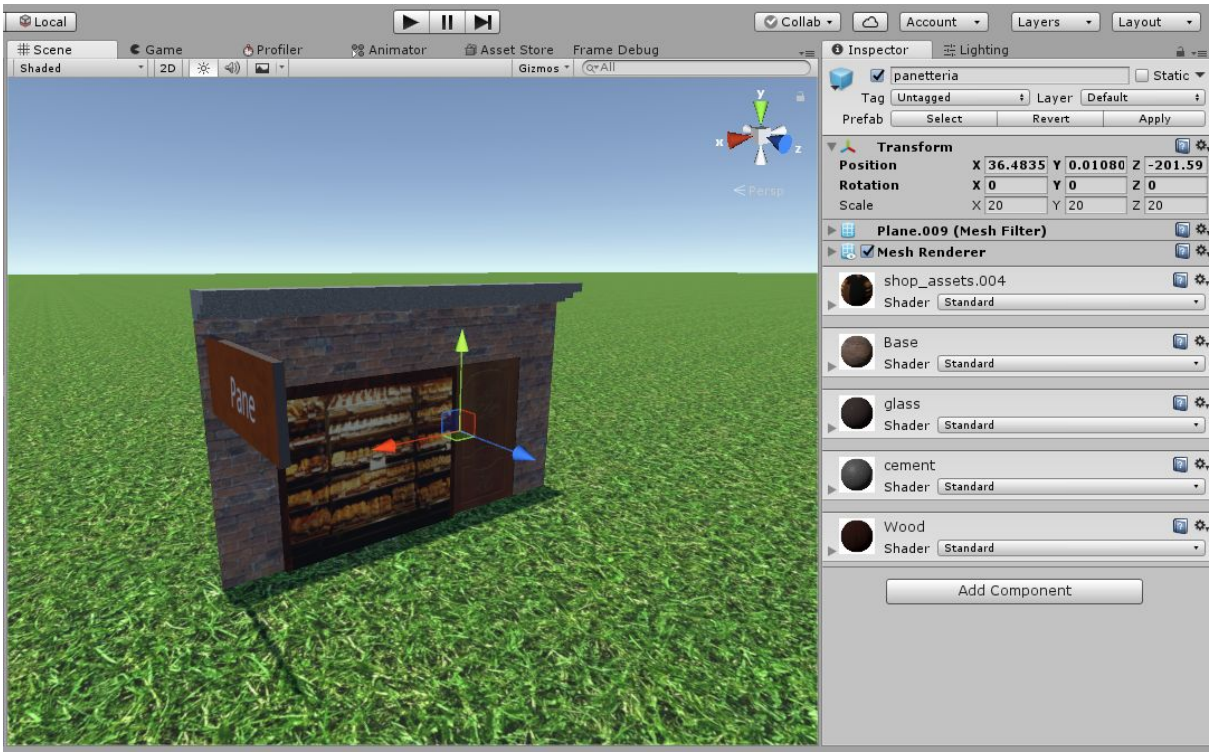
Il principale strumento usato per lo sviluppo dell'applicazione è Unity,

Unity è un motore di gioco multipiattaforma che permette lo sviluppo di giochi e altri contenuti grafici multimediali in 2D e 3D.

Esso offre un ambiente di sviluppo leggero e flessibile, tra le sue caratteristiche, quelle importanti per questa applicazione sono:

- Facilità di utilizzo, grazie ad una viewport che permette la visualizzazione delle scene non solo run time, ma anche durante lo sviluppo, e un editor drag and drop , che permette la gestione delle risorse, e la parametrizzazione degli script.

- Scripting con C# un linguaggio a oggetti diffuso, che non espone le difficoltà di gestione della memoria e del ciclo di vita delle variabili allo sviluppatore, ma che permette alta leggibilità e potenzialità architetturali grazie alla sua natura fortemente tipizzata.
- Un asset store molto ricco, in cui è possibile scaricare gratuitamente diversi tipi di asset, tra i quali modelli 3D, script, shaders, materiali, essenziali quando non si ha un budget per un team di modellatori 3D.
- La possibilità di salvare prefabs.



L'editor di Unity offre una comoda viewport

L'elemento base dello Unity editor è il GameObject, esso viene usato per la creazione di qualsiasi elemento presente nel gioco.

Ogni GameObject è composto da components, che ne determinano le caratteristiche e funzionalità.

Tutti i GameObject devono possedere almeno un componente di tipo Transform, che ne definisce la posizione, scala, e rotazione. altri components possono essere scripts in C#, luci, telecamere, materiali, meshes, componenti per l'interazione con il motore fisico, eccetera.

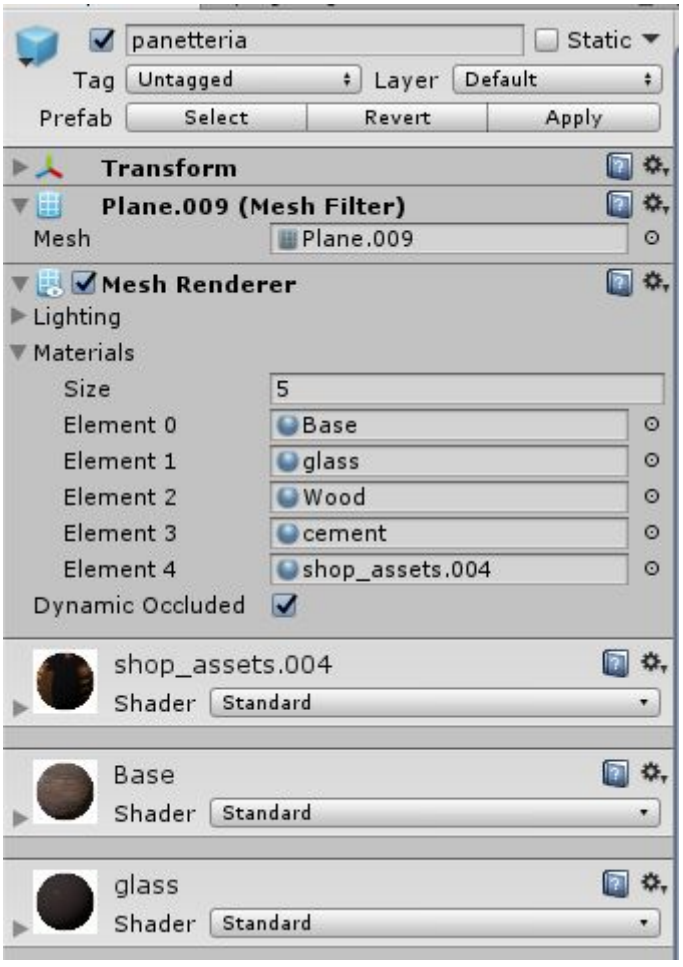
Una delle caratteristiche chiave di Unity, essenziale per lo sviluppo dell'applicazione è il suo sistema di prefabs.

Il prefabs system permette il salvataggio di GameObjects come risorse, che possono essere riutilizzate durante il gioco.

Qualunque modifica al prefab viene riflessa su tutte le sue istanze, permettendo la veloce esecuzione di grossi cambiamenti questa azione viene effettuata grazie alla presenza della funzione di apply nell'editor.

Ogni istanza di un prefab può essere comunque modificata separatamente a seconda dei bisogni, o di ritornare al prefab di partenza grazie al bottone di revert.

La caratteristica più importante dei prefabs è la possibilità di essere istanziati proceduralmente a runtime, questa è la caratteristica su cui si basa la generazione procedurale dell'intera applicazione.



Il prefab di una panetteria, notare i bottoni di revert e apply

Utilizzo dei prefabs e modularità

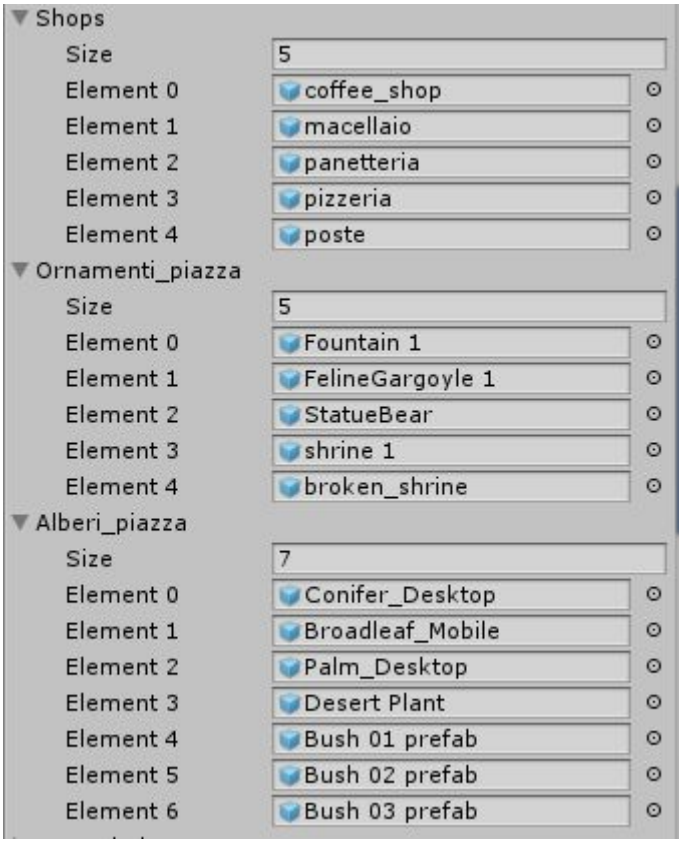
L'applicazione fa vasto uso del sistema dei prefab di unity per instanziare i GameObject runtime.
Gli elementi che vengono istanziati runtime sono:

- Landmarks distali
- Landmarks prossimali
- Decorazioni all'interno di piazze
- Tiles di un palazzo, compresi i negozi

Essi possono essere istanziati dal codice, specificando il tipo di prefab, la posizione, e la rotazione.

Gli assets di una applicazione Unity, compresi i prefabs possono essere inseriti in una cartella speciale chiamata "Resources", le risorse in quella cartella, possono essere usate facilmente da codice invocando la funzione Resources.Load.
I prefabs possono anche essere passati come variabile di istanza pubblica di una classe di uno script C#.
Questo tipo di dipendenza, combinata con la possibilità di istanziare tali prefabs runtime è quello che offre all'applicazione la sua estrema modularità.

Nel codice dell'applicazione non è specificato alcun prefab per l'istanziazione runtime delle risorse, essi vengono passati dall'editor in vettori di prefabs propriamente organizzati.

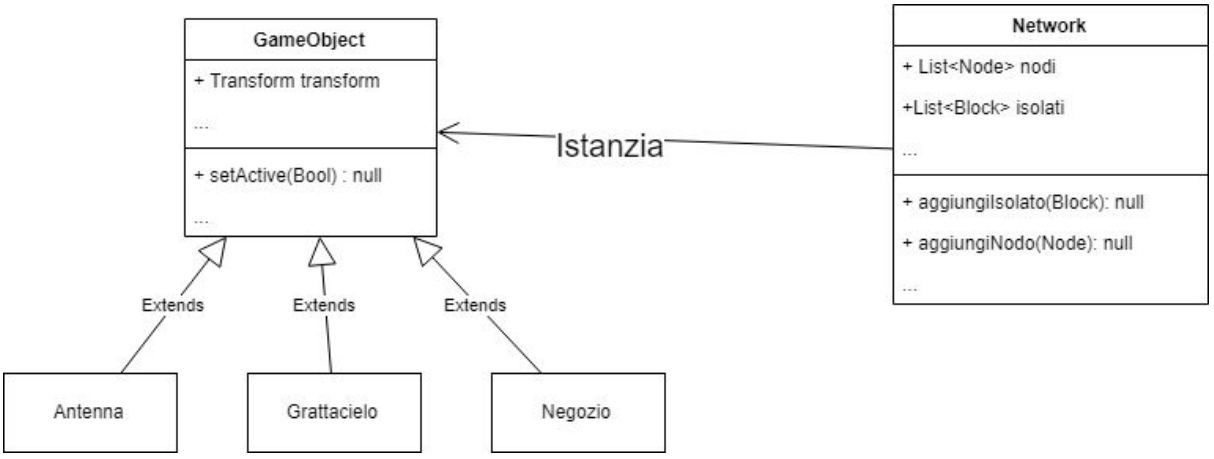


Prefabs di gameobjects passati agli script

Questo sistema segue il principio di software engineering del cosiddetto loose coupling.
In un sistema loosely coupled gli elementi architetturali che interagiscono non devono conoscere i dettagli implementativi l'uno dell'altro per interagire correttamente.

Solitamente, per creare interazioni loosely coupled di fa uso delle interfacce, che specificano le caratteristiche minime indispensabili che le classi devono avere per interagire tra di loro, in questo caso agli algoritmi di generazione procedurale descritti non importa il tipo specifico di elemento da istanziare nella scena, basta che sia un GameObect.
Allo stesso modo, I GameObjects che devono essere istanziati nella scena sono completamente indipendenti dall'algoritmo che li istanzia.
Questa estraneità degli elementi architetturali dell'applicazione permette un alta modularità del sistema, e riusabilità degli elementi.

Gli oggetti da istanziare nella scena possono essere manipolati da qualsiasi tipo di algoritmo, e gli algoritmi di generazione procedurale dell'applicazione possono essere usati per istanziare un vasto numero di GameObjects.



Relazioni tra gli elementi dell'architettura

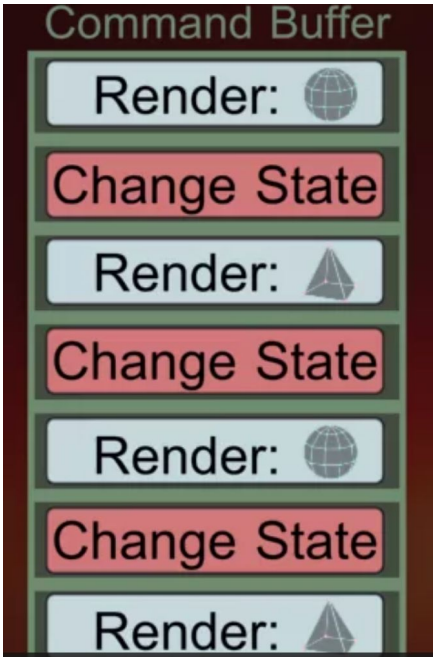
Un esempio specifico è la disposizione di landmark prossimali nella scena, l'algoritmo per la disposizione di tali elementi gestisce GameObject, questo permette a tale algoritmo di funzionare a prescindere dal tipo di GameObject gli viene passato.
Un vantaggio pratico è che tutti gli elementi grafici possono essere sostituiti nell'editor senza cambiare il modo in cui l'applicazione funziona.

Unity Batches e Draw calls, ottimizzazione run time:

Costruendo una città con le funzioni sopra citate purtroppo porta a gravi problemi di performance in quanto la scena avrà centinaia di oggetti diversi da renderizzare separatamente, ognuno dei quali conterrà meshed, materiali e shaders diversi.
Per il rendering di una scena, Unity sposta gli elementi visivi dalla RAM alla VRAM, e impartisce una serie di istruzioni che vengono eseguite dalla GPU, ogni oggetto nella scena corrisponde ad una draw call, ovvero un'istruzione che la CPU manda alla GPU per chiedere di disegnare un elemento a schermo.

Prima del disegno di un elemento però, la GPU deve cambiare il suo stato, lo stato contiene informazioni sul materiale dell'elemento da disegnare, ovvero gli shaders, le textures e i parametri del materiale.

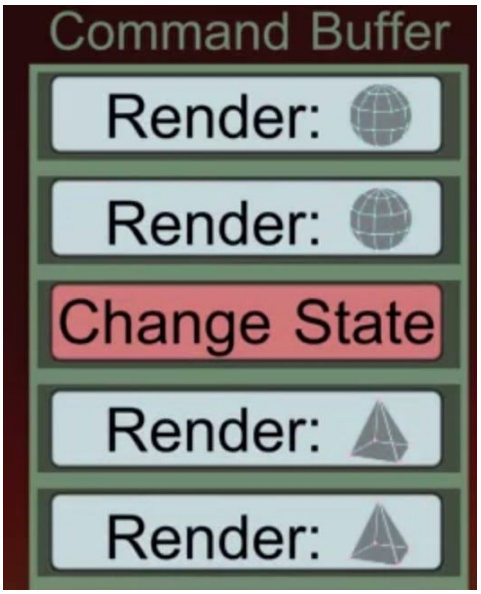
Questo cambio di stato è un'operazione molto dispendiosa da parte della GPU, il sovraccarico della CPU invece avviene quando troppe istruzioni di Draw calls vengono inviate.



Prima di ogni draw call la GPU cambia il suo stato

Unity prima dell'esecuzione di un applicazione esegue un dynamic batching, ovvero unisce geometricamente gli oggetti con lo stesso materiale per ridurre il numero di draw calls.

Per migliorare le performance però occorre ridurre significativamente il numero di cambi di stato, che avvengono quando si vuole disegnare a schermo un oggetto con un materiale diverso.



Un cambio di stato per molte draw calls

La soluzione è quella di unire, oltre alle geometrie, i materiali degli oggetti in scena, in questo modo con un cambio di stato è possibile disegnare oggetti che originariamente avevano materiali diversi.

Questo è effettuato dalla funzione di AdvancedMerge.
Essa, a partire da una serie di oggetti con materiali diversi, crea un unico oggetto che possiede un materiale che ingloba tutti i materiali degli oggetti originari.
Questa funzione viene chiamata allo startup dell'applicazione, e seppur richiedendo una breve attesa, essa migliora significativamente le performance a runtime.

Disposizione dei Landmark

Per disporre i landmark all'interno della città, bisogna assicurarsi che essi siano distribuiti in maniera uniforme, evitando che si concentrino in certe aree della città, lasciando vuote altre aree, i motivi sono principalmente 2:

1. Distribuendo i landmark in maniera uniforme gli ambienti cittadini che vengono prodotti hanno una varietà funzionale minore tra di loro, assicurando che la difficoltà tra città con parametri simili non cambi (minore varietà inter-cittadina)

2. La difficoltà interna di un ambiente cittadino è uniforme, dunque ogni angolo della città è tanto facile da navigare quanto ogni altro angolo.

Un terzo motivo potrebbe derivare dall'aggiunta di realismo che si otterrebbe, tuttavia il realismo è in ogni caso in secondo piano, rispetto alla funzionalità degli ambienti.

Per disporre dei landmark in maniera uniforme serve prima di tutto la lista di posti in cui possono essere disposti e la loro posizione.
Grazie agli step precedenti, la nostra struttura dati contiene la lista di isolati di cui è composto (`public List<Block> blocks`), ogni isolato contiene una variabile di istanza che ne definisce il centroide.

Dunque, dato un numero di landmark e una lista di possibili posizioni, il problema è quello di trovare le posizioni tra quelle disponibili che garantiscono uniformità di distribuzione, o in termini più concreti: che massimizzano la distanza minima tra due landmark disposti.

Si tratta quindi di un problema NP, non risolvibile in tempo proporzionale a una formula polinomiale del numero di elementi in ingresso, un exact method per trovare la soluzione impiegherebbe sicuramente troppo.

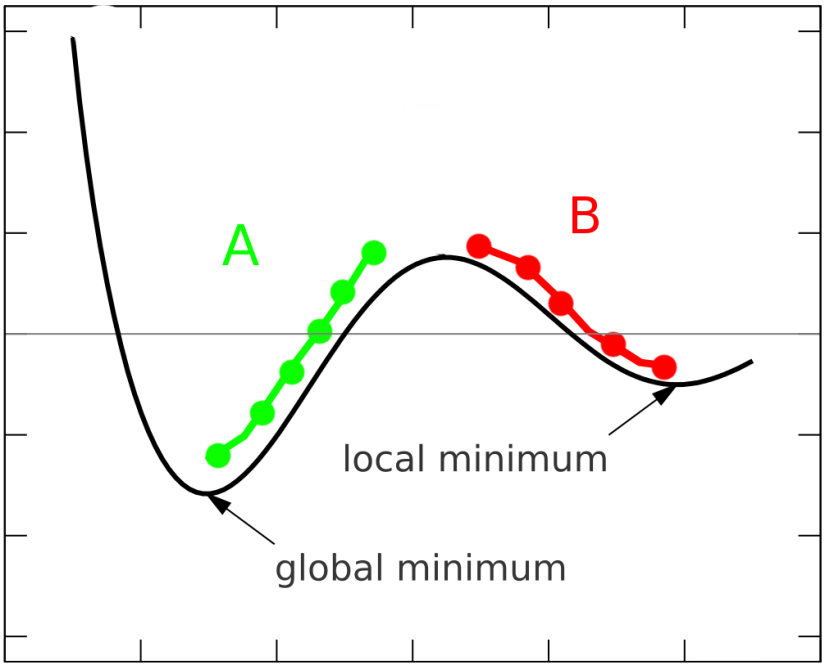
Euristiche e steepest descent

Ho deciso di risolvere il problema con un'euristica, la cui caratteristica è quella di ottenere un risultato sub-ottimo in un tempo accettabile piuttosto che mirare all'ottenimento di un risultato perfetto.
Per la creazione di un'euristica sono necessari:

1. Una soluzione di partenza
2. Un metodo di valutazione di una soluzione
3. Un modo di spostarsi da una soluzione ad un altra

Le filosofie principali che guidano un euristica sono: l'*explore* e l'*exploit.*
L'*exploit* è l'azione di spostarsi di poco nello spazio delle soluzioni verso soluzioni sempre migliori, con lo scopo di raggiungere il minimo locale più vicino, sfruttando la soluzione di partenza, la filosofia dell'*exploit* è rappresentata al meglio in un euristica di tipo *steepest descent*, chiamata anche *hill climbing,* nel quale a partire da una soluzione, mira a migliorarla in una successione di cicli di ottimizzazione, in cui in ogni ciclo la soluzione è migliore di quella del ciclo precedente senza mai accettare soluzioni peggiori, in nessuna condizione.

Questo è il punto debole di un'euristica di tipo steepest descent: una volta raggiunto un minimo locale essa non è più in grado di spostarsi nello spazio delle soluzioni alla ricerca dei un minimo locale più piccolo, o nei casi più ottimistici del minimo globale.



2 euristiche steepest descent con diversi punti di partenza.

Nell'immagine sopra vediamo una rappresentazione semplificata in due dimensioni nella quale nell'asse delle ascisse è rappresentato lo spazio di soluzioni, e nelle ordinata la bontà delle soluzioni.

Le linee colorate rappresentano due iterazioni diverse dell'euristica.

Si può notare come la stessa euristica steepest descent può ottenere risultati diversi partendo da due soluzioni di partenza diverse, A e B.

Per mitigare questo problema senza ricorrere ad un *exact method* ci si può spostare verso un approccio di *explore*.

La filosofia dell'*explore,*vuole esplorare lo spazio di possibilità, senza mai però cercare un ottimo in alcun modo, l'euristica che usa solamente questa filosofia viene chiamata *random walk.*

Ovviamente esplorare lo spazio di soluzioni senza mai spostarsi verso soluzioni migliori non porta alcun beneficio.

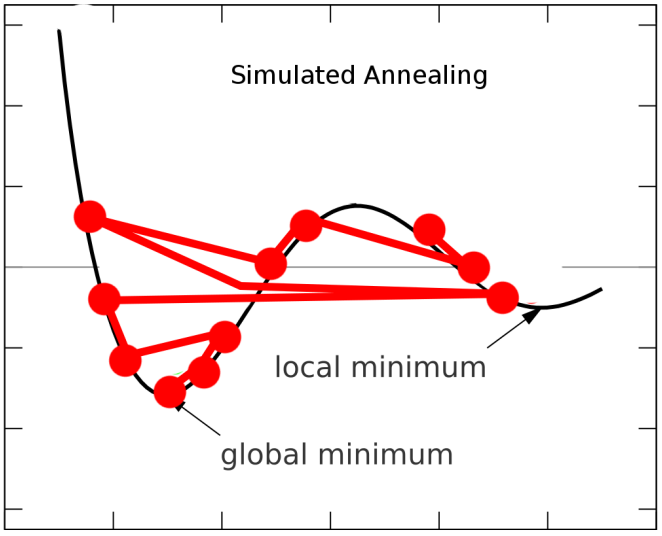
Simulated Annealing

Un'euristica di tipo *simulated annealing* cerca di usare un misto tra *explore* ed *exploit.*

Essa è caratterizzata da un parametro di temperatura *T*.

Più è alto il valore di temperatura e più si è tolleranti nello spostamento, accettando soluzioni anche molto peggiori di quella da cui si è partiti.Ogni iterazione viene chiamata *epoca.*

All'andare avanti delle epoche la temperatura si abbassa, e con essa la tolleranza negli spostamenti, finché si accettano solo soluzioni migliori, e l'euristica converge dunque ad uno *steepest descent.*



Prendendo soluzioni peggiori, il simulated annealing può uscire da minimi locali

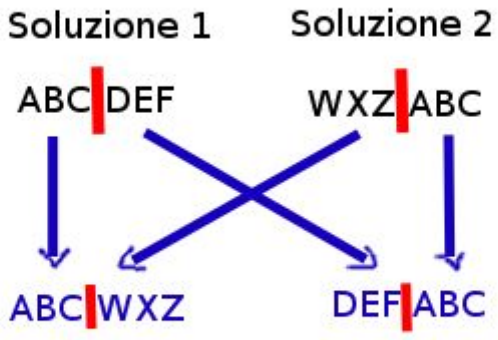
Algoritmo Genetico

Un algoritmo genetico invece sfrutta il concetto di *crossover* nel quale mettendo insieme parti di due soluzioni se ne può ottenere una migliore.

In un algoritmo genetico si parla di *popolazione* per fare riferimento ad un insieme di soluzioni iniziali piuttosto che una sola, le iterazioni dell'algoritmo vengono chiamate generazioni. dopo la creazione di ogni generazione:

1. La popolazione corrente crea una nuova generazione grazie all'operatore di *crossover.*

2. La nuova generazione viene mutata, ovvero dei cambiamenti casuali vengono effettuati a ogni soluzione
3. Le soluzioni così generate vengono valutate e le peggiori vengono scartate.



Operazione di Crossover tra due soluzioni

Gli algoritmi genetici possono essere molto efficienti quando una soluzione può essere divisa in parti, geni, e la bontà di una soluzione può migliorare mettendo insieme parti di soluzioni diverse.

Dato che lo spazio di soluzioni del problema in questione non contiene differenze troppo grandi tra i minimi locali, e non c'è nessun requisito stretto della bontà della soluzione ho deciso di usare diversi *steepest descent* che partono da soluzioni casuali diverse e di prendere la soluzione migliore, Inoltre lo steepest descent assicura il ritrovamento di un minimo locale nel minor tempo possibile, il ché si adatta bene ad un'applicazione real time.

Implementazione

La classe che contiene la logica dell'euristica è `LandmarksDistributor`.
La funzione da cui parte l'elaborazione ad alto livello è findsolution.
Che, dati:

1. un vettore contenente le possibili posizioni per i landmark
2. il numero di landmark da distribuire
3. il numero massimo di cicli di ottimizzazione

Essa ritorna la soluzione individuata dall euristica in un vettore di posizioni 2D.
Lo scopo è quello di ottenere una serie di posizioni ben distanti tra di loro.

Ricerca di una soluzione di partenza

La soluzione di partenza viene trovata in maniera del tutto casuale, data una serie di punti in cui i landmark potrebbero trovarsi e il numero di landmark da posizionare, seleziona casualmente i punti dal primo parametro.

Dopodichè partono i cicli di ottimizzazione, contenuti nella funzione improve_worst.
essa mira al miglioramento della soluzione ricevuta come parametro effettuando uno spostamento nello spazio di soluzioni.

è dunque importante discutere come una soluzione viene valutata e cosa viene considerato come spostamento.

Valutazione di una soluzione

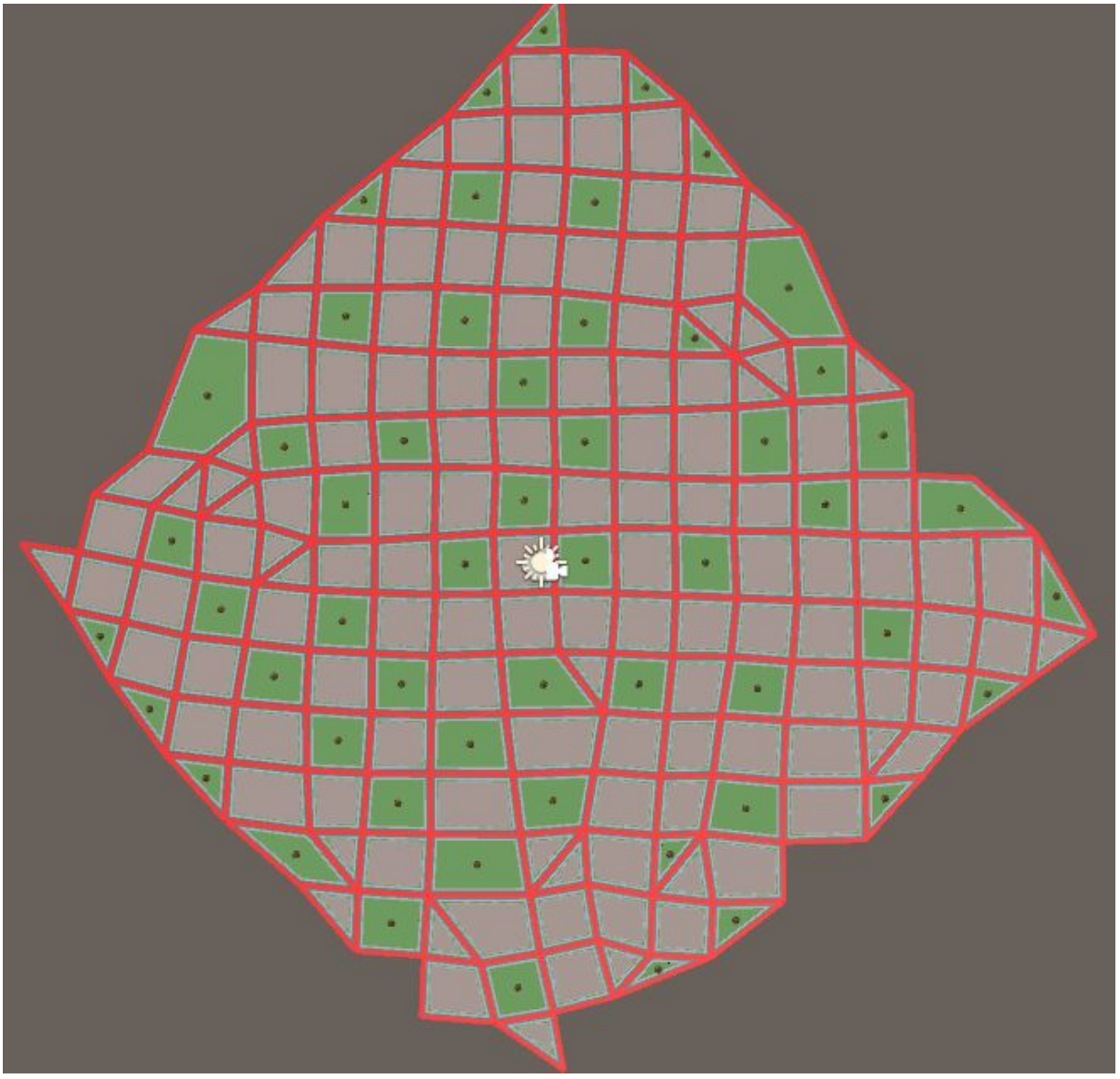
Una città ha landmark ben distribuiti quando la distanza tra i due landmark più vicini è grande.
questa distanza viene estratta dalla funzione worst, che legge la distanza di tutti i nodi presenti nella soluzione corrente a due a due, e ritorna la distanza maggiore.

Spostamento nello spazio di soluzioni

Una volta che si è deciso che la soluzione corrente non è sufficiente, è giunto il momento di cercare un'altra soluzione.

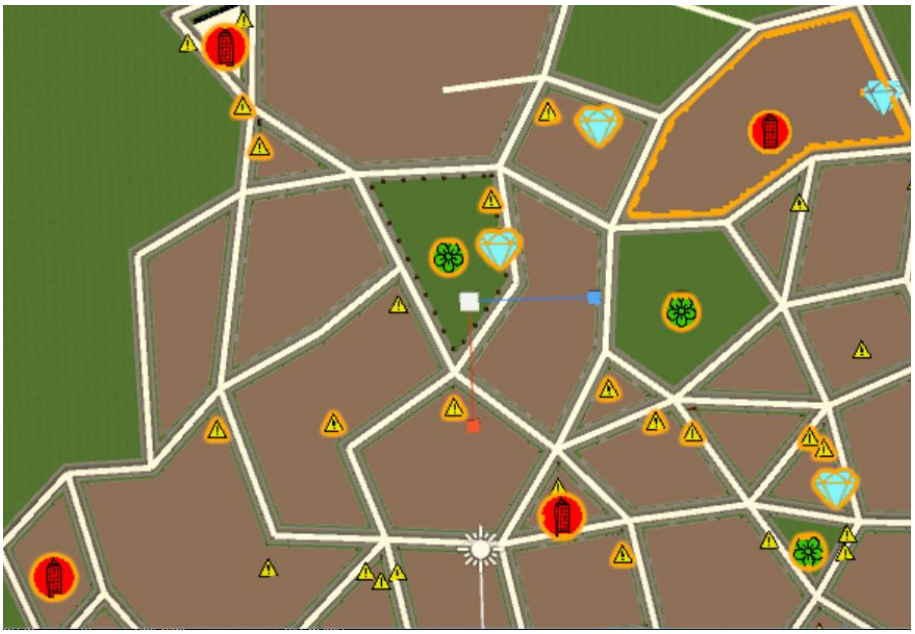
Nel nostro caso uno spostamento nello spazio di soluzioni è costituito dallo spostamento di un landmark da una posizione ad un'altra.
La funzione `improve_worst` effettua solo spostamenti che migliorano la soluzione.

Questa euristica viene eseguita più volte per ottenere diverse possibili soluzioni dalle quali si preleva la migliore.
Questo assicura un tempo di elaborazione contenuto, e una soluzione che non rappresenta necessariamente un minimo locale troppo alto.



Risultato di una serie di iterazioni di steepest descent per la distribuzione di piazze

Questo metodo viene ripetuto diverse volte durante l'applicazione per il posizionamento di diverse classi di landmark, parchi, palazzi, negozi e oggetti.



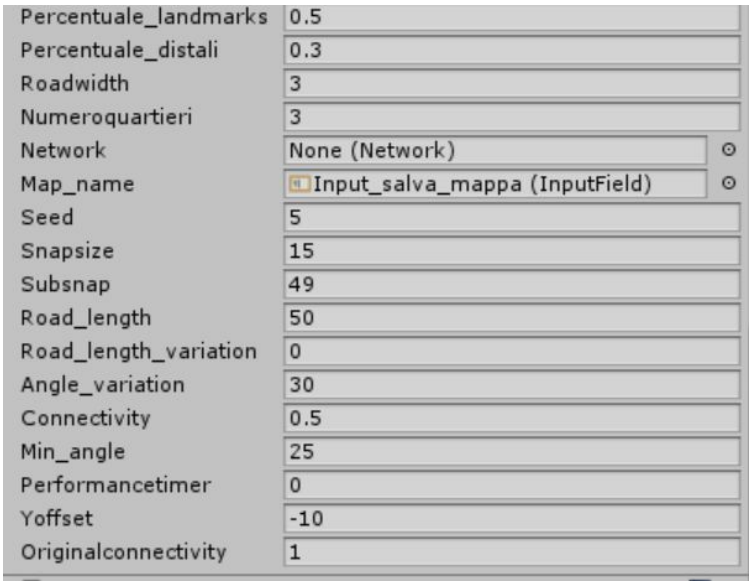
posizionamento uniforme di diversi tipi di landmark rappresentati da icone simboliche

Interfaccia Grafica

Un elemento molto importante dell'applicazione è l'interfaccia grafica, essa permette al terapeuta o allo psicologo ricercatore la creazione di un ambiente cittadino a partire da parametri da lui immessi.
Questo processo deve avvenire in modo più intuitivo possibile senza bisogno di istruzioni aggiuntive, e deve anche offrire un livello soddisfacente di flessibilità.

Requisiti

La creazione di un ambiente cittadino richiede l'immissione di un numero di parametri troppo alto per essere immesso in modo efficiente dal terapeuta.



Parametri richiesti per la generazione cittadina

Essi vanno dunque filtrati in qualche modo per presentare solo i parametri di interesse.
Essendo l'applicazione in fase iniziale, i possibili utenti non sono strettamente definiti con sicurezza, due figure principali sono però:

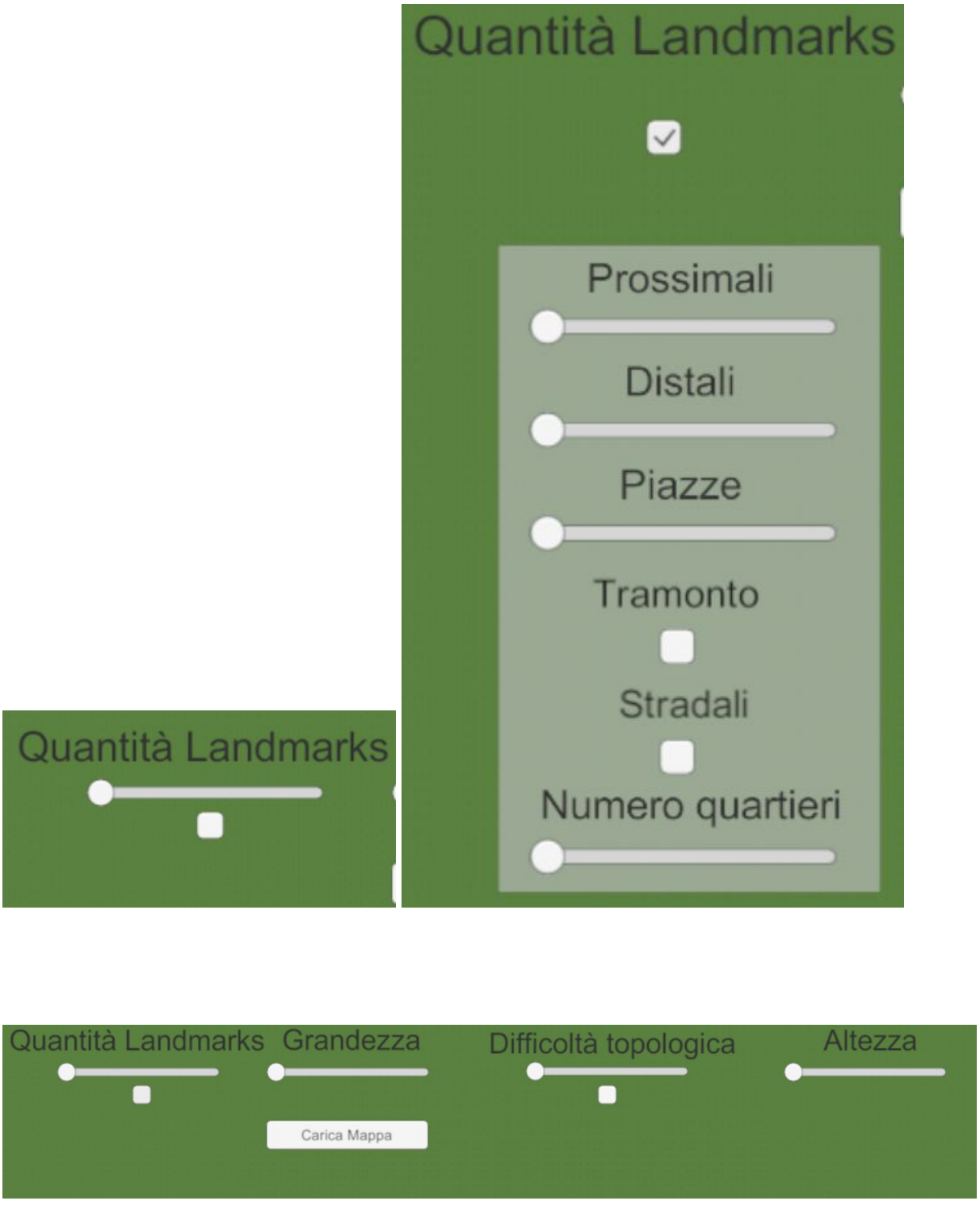
1. Terapeuta: comprende lo scopo dell'applicazione e le basi della psicologia che insegna le interazioni tra un uomo e un ambiente da esplorare, conosce il paziente a cui viene somministrato l'esercizio, è in grado di fornire semplici istruzioni per la creazione di un ambiente cittadino.

2. Psicologo-Ricercatore: ha una profonda conoscenza dell'interazione tra un uomo e un ambiente da esplorare, è in grado di fornire informazioni più dettagliate sulla morfologia dell'ambiente cittadino e vuole sfruttare al massimo la varietà che l'applicazione offre

Dunque vi è la presenza di due utenti diversi con conoscenze e bisogni diversi.
Un'interfaccia utente troppo dettagliata non permetterebbe al terapeuta di esprimere in modo semplice la città richiesta.
Un'interfaccia utente troppo filtrata d'altro canto limiterebbe lo psicologo e la sua abilità di sperimentare.

UI Espandibile

La soluzione impiegata unisce il meglio dei due mondi, essa è un'interfaccia utente a opzioni espandibili:
Il punto di partenza è la quantità massima di opzioni, che a seconda del tipo di utente possono essere collassate fino ad avere solo quattro opzioni.
gruppi di opzioni vengono collassate in opzioni più generiche, che nel codice vengono di nuovo espanse in quelle particolari.



UI in modalità terapeuta, richiede l'immissione di soli quattro parametri

UI in modalità psicologo, viene richiesta l'immissione di undici parametri

Esplorazione Utente

Esplorazione di un ambiente cittadino in terza persona

Questo strumento nel futuro farà parte di un progetto più grande con della logica di gioco per la diagnostica e riabilitazione.
Anche se nel futuro verrà introdotta della logica di gioco, è importante che le città generate possano essere esplorate, questo permette una valutazione della qualità delle mappe, ma permette di utilizzare lo strumento come un'applicazione completa, e cominciare a fare della ricerca nell'ambito di psicologia.

Ho deciso di offrire un esplorazione in terza persona, utilizzando un'automobile.
L'automobile in terza persona offre alla maggior parte della popolazione un sistema molto intuitivo e familiare per esplorare uno spazio tridimensionale.
Essa rende anche l'esplorazione piacevole, caratteristica importante quando a un paziente devono essere somministrate diverse ore di esercizi terapeutici.

La velocità, campo visivo e raggio di curvatura, sono stati concordati con il team di psicologi per assicurarsi che non intaccassero sull'efficacia terapeutica del prodotto.

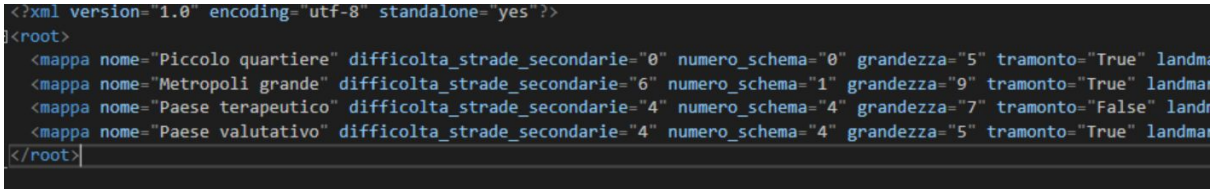
Salvataggio e caricamento mappe

La generazione semi casuale degli ambienti assicura che ad un paziente si possa somministrare esercizi di simile difficoltà, ma diversi.
Usando la stessa identica città, si corre il rischio che l'utente debba effettuare molto meno sforzo in quanto può memorizzarla.

Tuttavia un altro caso d'uso di questa applicazione è la valutazione uniforme delle capacità cognitive di diversi pazienti, per effettuare questo tipo di test è necessario che la mappa somministrata sia esattamente la stessa.

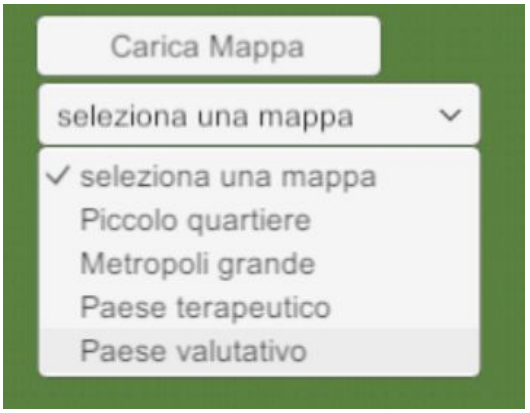
Questo caso d'uso fa scaturire dunque, la necessità di una funzionalità di salvataggio e caricamento mappe.

Essendo che una mappa viene generata a partire da una serie di parametri più un seme casuale, per il salvataggio e il caricamento delle mappe possono essere gestite proprio queste informazioni in un semplice file xml



File xml contenente i dati delle città

Dunque quando l'utente vuole salvare una mappa, preme l'apposito bottone sulla UI, dà un nome alla mappa e i suoi parametri vengono salvati su di un file xml.
Il caricamento di una mappa invece vede l'estrazione dei parametri dal file xml, e la creazione dell'ambiente 3D a partire dagli stessi.



UI per il caricamento delle mappe

Files di log

Logging è l'atto di registrare gli eventi, i risultati intermedi, o le comunicazioni che avvengono all'interno di un'applicazione per vari scopi, il modo più semplice di fare logging è quello di scrivere tutto in un log file.

Lo scopo del logging in questo caso è diagnostico e di debug nel caso si dovessero incontrare comportamenti inaspettati dell'applicazione.
Il file di log permette dunque di capire cos'è andato storto durante l'esecuzione, anche dopo che il bug è stato incontrato, senza il bisogno di debugging attivo in tempo di esecuzione.
Il file di log permette anche ad un utente non tecnico di poter fornire informazioni utili al programmatore per aiutarlo a debuggare l'applicazione.

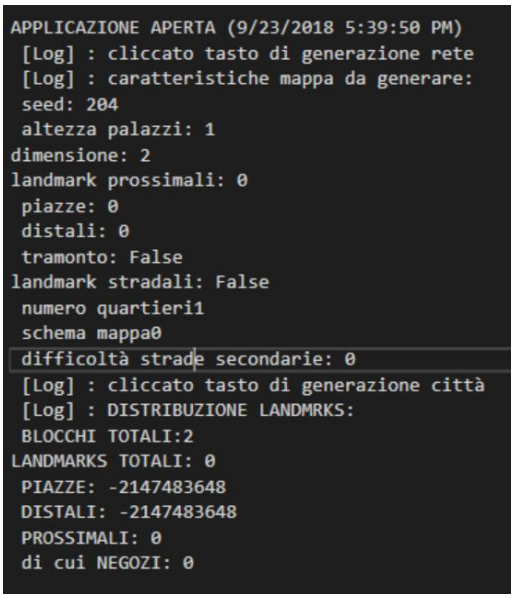
Unity permette di aggiungere funzioni di callback che vengono chiamate quando viene scritto qualcosa nel Log,
Per implementare il logging in un file di log è bastato aggiungere una callback a questo evento che scrive i messaggi di log in un file.

Si è deciso di creare un file di log ogni volta che viene fatta partire l'applicazione, il file di log avrà come nome la data e l'ora in cui l'applicazione è stata fatta partire.

Gli eventi principali che vengono registrati nel file di log sono:

- Interazione dell'utente con la UI
- I parametri della città

Dato che l'applicazione è single thread, queste informazioni bastano a riprodurre le circostanze che hanno portato un certo bug.



file di log contenente la lista di azioni che l'utente ha eseguito

Ambiente di esecuzione e performance

L'applicazione è stata sviluppata per essere eseguita su desktop almeno di fascia media presenti in un ambiente di ricerca od ospedaliero.
Seppur Unity è stato scelto principalmente per la facilità di utilizzo, esso permette l'esportazione dell'applicativo per diverse piattaforme, WIndows, MacOs, Web, il che offre ulteriore flessibilità.

Su una macchina IntelCore i7, 8 Giga di ram, scheda video NVidia Geforce GT 1050 l'applicazione riesce a rimanere costante sui 60 frames per secondo anche quando viene creata la città più grande e complessa che l'applicazione fornisce.
Questo garantisce un'ottimale esperienza di uso.

Conclusione

Risultati ottenuti

Questo progetto ha raggiunto gli obiettivi preposti.
Il prodotto è un'applicazione interattiva che permette la generazione procedurale e parametrica di ambienti cittadini, e ne permette l'esplorazione.
Di seguito, sono elencate alcune desiderabili proprietà del prodotto:

Parametrizzabile: insieme alla collaborazione dei professori di psicologia dell'università degli studi di Torino, è stato identificato l'insieme di parametri richiesto dai vari utenti dell'applicazione per garantire massima flessibilità, e libertà di sperimentazione, l'applicazione offre la possibilità di modificare questi parametri.

Usabile: il design della User Interface permette all'utente di scegliere la libertà con cui settare i parametri dell'applicazione a seconda del suo livello di conoscenza, grazie alle funzionalità di salvataggio e caricamento delle mappe, l'applicazione permette di riutilizzare il lavoro creativo fatto da altri utenti.
Inoltre, nel caso di richieste particolari, è anche possibile disegnare le strade primarie a mano, grazie alla modalità manuale.

Modulare ed espandibile: grazie all'uso di un loosely coupled design, è semplice rimpiazzare, aggiungere o eliminare elementi grafici dell'applicazione, o algoritmi di ottimizzazione utilizzati, ciò facilità qualsiasi lavoro venga svolto successivamente per ampliare l'applicazione.

Realistica: grazie al motore grafico di unity, e alla vastità di elementi presenti nello Unity asset store, il risultato ottenuto offre una discreto realismo, che non diminuisce mai la funzionalità terapeutica dell'applicazione.

Piacevole: L'esplorazione della città viene effettuata in terza persona con un'automobile, la cui velocità, campo visivo, e raggio di curvatura garantiscono un esplorazione piacevole, limitata solo dai requisiti funzionali dell'applicazione.

Multipiattaforma: l'applicazione, essendo stata sviluppata un Unity permette la creazione di eseguibili per diverse piattaforme, seppur la piattaforma richiesta è pc, in un futuro, può anche essere esportata per mobile, o per web.

Sviluppi futuri

Questo strumento, è solo una prima parte del prodotto finale.
Il prodotto finale è un'applicazione per la diagnostica e riabilitazione della memoria spaziale di pazienti con diverse situazioni di deficit cognitivo.
Nel campo della psicologia, ulteriori studi devono essere eseguiti per capire il metodo migliore di inventare e somministrare tali esercizi per l'ottenimento dei massimi risultati.
Questa applicazione offre sia uno strumento flessibile per ulteriore sperimentazione nell'ambito psicologico, sia le fondamenta di un prodotto finale a scopo diagnostico e riabilitativo.
Ulteriori sviluppi di questo strumento saranno un più vasto catalogo di asset visivi tra cui scegliere per la creazione della città, che possono essere divisi in stili, e modellati in house da artisti, in modo da ottenere componenti architetturali più realistici, Modelli 3D con diversi LOD possono essere usati per aumentare ulteriormente le performance.
Cosa più importante è la logica di gioco che deve essere costruita sopra lo strumento sviluppato.
Il tipo di logica di gioco dovrà essere sviluppato in stretta collaborazione con un team di psicologi ricercatori.
Principalmente la logica di gioco dovrà permettere all'utente, oltre che esplorare la città, anche raccogliere degli oggetti, poichè sicuramente un esercizio terapeutico consiste nell orientarsi all'interno di una città nella quale dovrà ricercare e memorizzare la posizione di alcuni oggetti per poi recuperarli.

Il comportamento del paziente durante l'esecuzione degli esercizi non può sempre essere osservato strettamente da un terapeuta in tempo reale, perciò nel futuro, l'applicazione potrebbe registrare e salvare sessioni svolte, integrandolo con la possibilità di avere account, per permettere il raccoglimento dei dati specifici ad un particolare paziente.

Il passo ancora successivo, e più ambizioso, consiste nel cercare di diagnosticare, e somministrare i giusti esercizi terapeutici senza il bisogno di un terapeuta, automatizzando completamente il processo.

Bibliografia

Y. Parish, P. Müller: Procedural Modeling of Cities

P. Prusinkiewicz: Graphical applications of L-systems

G. Rozenberg, A. Salomaa: The mathematical theory of L systems

G. Kelly, H.McCabe: A Survey of Procedural Techniques for City Generation

R. Tadei, F. Della Croce, A. Grosso: Fondamenti di Ottimizzazione

R. Tadei, F. Della Croce: Elementi di Ricerca Operativa

G. Kelly: An Interactive System for Procedural City Generation

R. Goldman, S.Schaefer, T. Ju: Turtle Geometry in Computer Graphics and Computer Aided Design

G. Chen, G Esch, P.Wonka, P. Müller, E. Zhang: Interactive Procedural Street Modelling

D. Eberly: Constructing a Cycle Basis for Planar Graph

docs.unity3d.com: Draw call batching

K. Lynch: The Image of the City

B. Mandelbrot: The fractal geometry of nature