UNIVERSITA' DI PISA

Facoltà di Scienza Matematiche, Fisiche e Naturali

Corso di Laurea in Tecnologie Informatiche

Tesi di laurea

Titolo:

STUDIO E IMPLEMENTAZIONE DI METODOLOGIE PROCEDURALI PER LA RAPPRESENTAZIONE TRIDIMENSIONALE DI EDIFICI URBANI AD ALTA COMPLESSITÀ

Autore

Antonio Francesco Bazzoni

Matricola: 232810

Relatori

Prof**. Massimo Bergamasco** - Ing. **Marcello Carrozzino** - Ing. **Franco Tecchia**

Controrelatore: Prof. **Giuseppe Attardi**

ANNO ACCADEMICO 2004-2005

Ringraziamenti

Dio solo sa quanto sia stato difficile,
ma alla fine ho rispettato tutte le scadenze!
Ringrazio
Marcello per avermi seguito e assecondato nello svolgimento della tesi;
ilVeraldo per essermi stato vicino in tutti questi anni, sia come amico che come tutore;
Jordans, mio coinquilino, fotografo quasi-semi-professionale e mago della rete domestica;
infine, ringrazio tutti gli Amici per avermi accompagnato nella mia vita studentesca.

Dedico questa tesi
ai miei Genitori, ai miei Nonni
e a tutti quelli che, come me, hanno la grafica nel cuore.

Indice

Indice delle figure

Prefazione

Ho scelto di iscrivermi alla facoltà di Informatica in quanto ho avuto modo di conoscere questa scienza fin dalle scuole superiori.
Sono sempre stato affascinato dalla parte algoritmica. Trovare l'implementazione più efficiente per la risoluzione di un dato problema è una buona sfida mentale, l'espressione dell'ingegno dell'autore. Il fine è creare ciò che si desidera nel minor tempo possibile e con le minori risorse; un "gioco" per la mente che regala soddisfazione.
Altra mia passione è la grafica, un campo che in questi ultimi anni è stato fortemente influenzato dall'informatica (chi non è rimasto affascinato dai primi videogiochi, per non parlare degli ultimi!).
La modellazione procedurale è stata perciò la scelta ideale, la congiunzione di queste mie due passioni.
La modellazione di edifici urbani è indubbiamente un campo di grande utilità, sia per la Computer Grafica che per la Realtà Virtuale, ma è solo un punto nello spazio immenso delle tecniche procedurali. In confronto alla totalità, in questo settore ci si sente tuttora dei pionieri in quanto le tecnologie utilizzate per la raccolta dei dati sono molto recenti e la ricerca è ancora agli inizi.
Sviluppare questa tesi è stata una bella esperienza, fortemente gratificante, poiché i risultati, in questo caso è proprio bene dirlo, si vedono. Avere davanti agli occhi una città generata

attraverso il proprio codice (AB-Block-Code) è una grossa soddisfazione, soprattutto se questa può essere equiparata a quelle generate dai "grandi" di questo settore.

Struttura della tesi

Ho cercato di scrivere questa tesi in maniera tale che possa essere letta da tutti, solo alcune sezioni sono più specifiche, interpretabili solo da chi è del settore.

Nel primo capitolo abbiamo una panoramica alla grafica e alle sue due forme fondamentali: la Computer Grafica e la Realtà Virtuale. Infine un'introduzione generale alle tecniche procedurali.

Nel secondo capitolo viene descritto il motivo per cui esistono le città virtuali, gli obiettivi che si possono raggiungere e lo stato dell'arte, più in specifico vengono analizzati alcuni casi studio.

Nel terzo capitolo troviamo un'introduzione al progetto (AB-Block), nella quale vengono elencati i motivi della sua creazione e gli obiettivi che si vogliono raggiungere. Inoltre vi sono degli approfondimenti sulle API utilizzate (QT e OpenGL).

Nel quarto capitolo vengono esaminate le strutture dati dell'AB-Block-Code, mentre nel quinto sono spiegati gli algoritmi più importanti. Nel sesto vengono illustrate le funzionalità offerte dall'interfaccia. Nel settimo sono presentati i risultati, sia qualitativi (resa delle geometrie), che quantitativi (valutazione delle performance).

Nell'ottavo le conclusioni e la descrizione di alcuni lavori futuri già in progetto.

Parole Chiave

Grafica 3D, modellazione procedurale, ambienti virtuali, città virtuali, edifici virtuali.

1 Introduzione

1.1 Computer Grafica e Realtà Virtuale

"Un'immagine vale più di mille parole", questo famoso detto esprime tutta la forza e la potenza della computer grafica. L'immagine ha sempre avuto un'enorme importanza nella storia dell'uomo, e l'utilizzo del computer l'ha saputa valorizzare ancor più, aumentandone sia la produttività che la qualità. La computer grafica è appunto quel settore dell'informatica che si occupa della creazione e dell'elaborazione, secondo una pipeline di operazioni e algoritmi, di immagini in due o tre dimensioni, fisse o in movimento. E' solitamente indicata con gli acronimi CG o CGI, quest'ultimo derivato da "Computer Generated Imagery". La CG è utilizzata in un'enorme varietà di applicazioni professionali e di consumo, quest'ultime sono videogiochi, ritocco fotografico, montaggio di filmati amatoriali, programmi educativi, presentazioni e web; mentre quelle professionali includono industria cinematografica (film di animazione digitale ed effetti speciali), tipografia (impaginazione di giornali e riviste), progettazione grafica (ad esempio CAD), visualizzazioni di dati tecnico-scientifici e sistemi informativi territoriali (SIT o GIS).

Le capacità della CG sono limitate solamente dalla potenza dell'hardware, che comunque cresce sempre più velocemente. Esiste un settore di ricerca in continua espansione sia dal punto di vista dell'hardware che da quello algoritmico, i cui obiettivi sono la velocità di elaborazione e il realismo o il non realismo (nel caso di NPR ovvero rendering non fotorealistico, o simulazioni di eventi non naturali), cioè, più correttamente, la capacità di un'immagine di essere *believable*, credibile.

fig. 1 Città realizzata in CG

Nel parlare comune solitamente si attribuisce lo stesso significato ai termini "computer grafica" e "realtà virtuale", ma, anche se strettamente legati, essi definiscono due materie diverse.

Per esprimere l'importanza della realtà virtuale bisognerebbe coniare un nuovo detto: "Un'esperienza vale più di mille immagini". In parole povere la differenza che c'è tra guardare un gioco a premi in televisione e parteciparvi.

Infatti, mentre nella computer grafica le immagini sono precalcolate, scorrelate dall'utente che le subisce passivamente, nella realtà virtuale il punto di vista è quello proprio dell'utente, non solo a livello visivo, ma si cerca di far interagire il più possibile i sensi dell'uomo, ovvero si crea un'esperienza *immersiva*. Questa è tanto migliore quanto più si riesce ad ingannare l'utente, cioè a fargli credere che ciò che sta vivendo sia vero e non virtuale.

Per ottenere un effetto sufficientemente fluido, è necessario visualizzare le immagini in tempo reale, cioè almeno a 25 fotogrammi al secondo. In caso contrario i movimenti nel mondo virtuale risultano essere imprecisi, a scatti, e possono dare anche un senso sgradevole di stordimento e nausea. Per questo motivo si da più importanza ad un basso tempo di rendering, a dispetto della qualità, che non è del tutto fotorealistica come con le tecniche della CG, le quali hanno bisogno di ore, se non giorni, per essere elaborate.

Nel campo della realtà virtuale abbiamo un numero sempre crescente di dispositivi di interfaccia, dovuto al fatto che l'obiettivo è far partecipare più sensi possibile all'esperienza. Nel settore visivo esistono le lenti polarizzate (utilizzate anche in CG) o ad otturatori, caschi per la visione stereoscopica, pareti retroproiettate (PowerWall) ecc.. Nel settore acustico esistono impianti surround e cuffie che riescono a dare spazialità al suono grazie a dei processori che lo rielaborano in base alla posizione della sorgente. Nel settore tattile abbiamo due tipi di interfacce, quelle che stimolano il tatto vero e proprio (riconoscere se una superficie è ruvida o liscia), e quelle aptiche, che danno il ritorno di forza (se spingo contro un muro non riesco a penetrarlo). Nel primo caso esistono dei guanti con speciali sensori, mentre nel secondo vengono utilizzati dei bracci meccanici.

Le interfacce nel settore del gusto e dell'olfatto sono ancora dei prototipi, ma la ricerca sta investendo anche su queste. Esiste poi il senso dell'equilibrio, cioè la capacità di capire se si è in posizione eretta o capovolti, o se il corpo sta subendo un'accelerazione in una certa

direzione. In questo caso esistono piattaforme semoventi e cuffie che tramite dei suoni agiscono sulla chiocciola all'interno dell'orecchio.

Insomma la computer grafica e la realtà virtuale differiscono considerevolmente, ma partono entrambe dalla costruzione di un mondo sintetico creato ad arte. "Creato ad arte" nel vero senso della parola, cioè realizzato da veri e propri artisti, i quali si occupano di modellare forme tridimensionali e disegnare le texture, cioè la "pelle" da applicare alle forme. Prendendo in considerazione la sola creazione (senza l'animazione), la difficoltà di questo lavoro è data dalla complessità delle forme da riprodurre, dal loro dettaglio e dalla vastità della scena.

fig. 2 Esempio di lenti stereoscopiche polarizzate

E' impensabile che nelle scene che oramai siamo abituati a vedere nei film ogni cosa sia fatta a mano da un'artista, sarebbe troppo costoso in termini di tempo e di denaro. E' necessario che alcuni aspetti della creazione siano automatizzati, cioè sintetizzati, in parte o interamente, dal computer tramite algoritmi specifici (procedure). Si parla perciò di tecniche procedurali.

1.2 Tecniche Procedurali

Le tecniche procedurali vennero introdotte nella metà degli anni 80 come strumento per la produzione delle texture degli oggetti. Si riconobbero in alcuni materiali dei *pattern*, ovvero regole cicliche. Una texture non era più data dalla scansione di un'immagine (foto o disegno), ma poteva essere definita tramite un algoritmo in grado di cogliere le proprietà intrinseche del materiale. Potevano così essere create migliaia di immagini diverse, sempre riconoscibili e classificabili come lo stesso materiale, ad esempio rocce, legno, acqua, sabbia e molte altre.

Nel corso della storia queste tecniche hanno fatto parte di una grossa area di ricerca tuttora in espansione, non solo per la produzione di immagini fisse ma anche di animazioni e simulazioni di fenomeni naturali come, ad esempio, il movimento del mare, del vapore e del fuoco.

L' estensione al campo della modellazione fu un passaggio "naturale". Applicare le stesse teorie alla creazione della geometria era una sfida allettante, e adesso possiamo dirlo, di sicuro successo.

E' facile constatare la presenza di pattern anche nelle forme (pensiamo ad esempio ai vari tipi di piante e alle diverse formazioni rocciose), purtroppo la progettazione degli algoritmi non è per niente elementare. La difficoltà della modellazione procedurale è direttamente proporzionale al dettaglio dell'oggetto.

Esistono tecniche basate su grammatiche (L-System), che permettono all'utente di sintetizzare con pochi parametri la complessità di un albero o di altri oggetti naturali attraverso l'utilizzo di un linguaggio formale che ne specifica le regole di crescita.

Per creare il "caos" proprio della natura vengono utilizzate delle funzioni stocastiche, alle quali possono essere affiancate funzioni rappresentanti leggi fisiche (ad esempio il tropismo di una pianta, ovvero l'attrazione gravitazionale applicata ai rami della pianta). Un punto di forza della modellazione procedurale è la parametricità, cioè la possibilità di scegliere il livello di realismo che si vuole ottenere.

Un'altra proprietà importante è l'astrazione. In un approccio procedurale possiamo specificare tutti i dettagli di una scena o di una sequenza astraendoli in una funzione che può essere richiamata quando serve. Si ottiene così un grosso risparmio di spazio su disco poiché la geometria non è specificata in maniera esplicita (insieme di vertici e facce), ma in

maniera implicita, cioè tramite un segmento di codice. Inoltre, nei sistemi interattivi si ha spesso bisogno di salvare più modelli dello stesso oggetto in diverse risoluzioni; con le tecniche procedurali invece basta passare in input alla funzione la risoluzione richiesta. Il tempo di creazione è spostato dal modellatore al computer. Con l'avvento dei processori grafici programmabili a basso costo le tecniche procedurali sono diventate vitali per la creazione di effetti di alta qualità nell'intrattenimento interattivo e nei videogames.
Attualmente molte applicazioni di rendering e animazione hanno interfacce per la produzione procedurale. Solitamente un buon algoritmo maschera la complessità dell'oggetto all'utente, il quale deve immettere solo pochi parametri. D'altra parte più parametri sono a disposizione, più è possibile rifinire i dettagli. E' compito di chi progetta l'algoritmo trovare un equilibrio tra i due aspetti.
Molte tecniche avanzate di modellazione sono principalmente procedurali. I modelli geometrici poligonali non sempre riescono a rappresentare in maniera ottimale la complessità delle forme. Esistono anche sistemi particellari e modelli di superfici implicite per la creazione di modelli organici come le blobby-molecules, le meta-balls e i soft-object.

Il campo della modellazione procedurale è veramente vasto, in questa tesi ci concentreremo sul settore delle tecniche procedurali applicate alla formazione di centri urbani.

2 Edifici Procedurali

2.1 Città virtuali

Come mai molti studiosi hanno investito tempo e denaro nel settore della creazione automatica di centri urbani? Ebbene esistono svariate aree di lavoro in cui è necessaria la visualizzazione di città. La città, l'insediamento umano, sia essa metropoli che zona rurale, è una componente onnipresente nella realtà che ci circonda. Sono pochissimi ormai i luoghi in cui l'uomo non sia arrivato. Gli insediamenti cambiano con l'ambiente e la cultura in cui sono immersi, si trasformano con il passare del tempo come fossero dei grandi organismi viventi. La nostra vita dipende anche dalla loro "salute". Ricreare città esistenti in un ambiente virtuale tridimensionale aiuta lo studio e l'osservazione di molti aspetti che magari non potrebbero essere notati avendo a disposizione solo cartine topografiche e delle tabelle di valori. Grazie ad una visualizzazione tridimensionale al computer è possibile creare simulazioni riguardanti diverse strutture cittadine, come impianto fognario, acquedotto, linee elettriche e telefoniche. Inoltre questi modelli sono di grande aiuto per quanto riguarda lo studio della crescita di alcuni quartieri, della sicurezza nelle strade o di possibili danni futuri causati da onde elettromagnetiche, terremoti, smottamenti e fenomeni atmosferici. Tutto questo può essere analizzato da più angolazioni diverse.

Oltre allo studio urbanistico, le città virtuali sono utilizzate per le simulazioni di guida di ogni mezzo, terrestre o aereo, civile o militare.
Queste compaiono anche nel cinema, nei videogiochi (in cui può essere importante creare anche gli interni degli edifici) e nelle ricostruzioni di città antiche.

fig. 3 Ricostruzione di San Andreas nel videogioco GTA

Infine, una invenzione di questi ultimi anni è il turismo virtuale, ovvero mappe tridimensionali ben dettagliate di località turistiche, che possono essere esplorate stando comodamente a casa (senza nulla togliere alla bellezza del viaggio "classico").

2.2 Costruire una città virtuale

La costruzione procedurale di una città non è un processo semplice, ma composto da almeno cinque fasi in cascata (ogni fase dipende dalla precedente). Esse sono:

1. *Urban zone* – vengono raccolte le informazioni riguardanti la geografia del terreno (altitudine, idrografia e vegetazione) e aspetti socio-statistici (densità

di popolazione, mappa stradale). Esse sono generalmente definite in un GIS (Geografic Information System).

2. *Road network* – se il GIS non contiene i dati sulla rete stradale, essa può essere ricostruita tramite due metodi: L-System e modellazione dichiarativa di linee. Con il primo sistema ad ogni passo vengono create delle nuove strade dette "successori ideali"; queste, prima di essere istanziate, devono superare alcuni test di controllo locali. Il secondo procedimento è sempre incrementale, nella fase iniziale l'utente immette le strade principali, l'espansione avviene tramite schemi urbani già definiti.
3. *Blocks* – cioè isolato, è un'area delimitata da strade. Il modello utilizzato per un isolato è il poligono, solitamente convesso, il che riduce notevolmente la complessità geometrica degli algoritmi.
4. *Lots* – è "la zona avente lo stesso indirizzo postale", ovvero le aree in cui risiederanno i singoli edifici. I metodi utilizzati solitamente sono basati sulla partizione geometrica o sull'istanziazione di pattern.
5. *Buildings* – definisce l'aspetto degli edifici. Tecniche utilizzate sono: Shape Grammar, Declarative Modeling, Split Grammar e Geometric Modeling.

In questa tesi verrà approfondita solo l'implementazione delle ultime due fasi: Lots e Building.

2.3 Classificazione degli edifici

Le costruzioni umane hanno diverse morfologie. Esse possono essere classificate secondo vari fattori, quali: periodo storico, architetto, tecnica di costruzione, utilizzo, collocazione e tipo di occupanti. La classificazione rende più facile l'individuazione di pattern utili alla caratterizzazione dei modelli procedurali. Per capire la varietà degli edifici esistenti ecco un piccolo elenco:

villa	tempio	baracca	castello	ufficio
magazzino	fattoria	stazione	ospedale	fabbrica
grattacielo	biblioteca	faro	motel	museo
prigione	palazzo	igloo	chiesa	stadio

2.4 Obiettivi e tecniche

Una città virtuale può essere costruita con tante tecniche diverse a seconda del risultato che si vuole ottenere. All'inizio della progettazione è bene fissare gli obiettivi. Occorre decidere innanzi tutto il grado di realismo da raggiungere, la città può essere ricostruita fedelmente o solo in maniera credibile (naturalmente la prima soluzione richiede sicuramente più controlli manuali).
Nel caso di riproduzioni fedeli le tecniche usate sono quelle della fotogrammetria e dell'analisi dell'immagine di foto aeree. Mentre nel secondo sono utilizzati strumenti automatici, come ad esempio L-System, i quali, a seconda della bontà delle regole, possono avvicinarsi molto alla versione reale.
Un altro fattore discriminante è la possibile traversabilità degli edifici, la quale richiede algoritmi per la costruzione degli interni. Solitamente questa proprietà è richiesta solo in applicazioni come i videogiochi.
Infine è bene scegliere il livello di dettaglio degli edifici, poiché questo fattore ha un grosso impatto nel caso la città venisse utilizzata in applicazioni interattive. Questa ultima condizione incide anche sul formato dei modelli, ricordiamo infatti che la modellazione procedurale permette la compressione dei dati e la multi risoluzione (LoD: Level of Detail).

2.5 Il DTM e la fotogrammetria

L'evoluzione delle metodologie di rappresentazione del territorio hanno introdotto di recente sul mercato una serie di prodotti che affiancano, supportano e talvolta sostituiscono le tradizionali cartografie, sia quelle su supporto cartaceo che quelle con struttura vettoriale.
Il modello digitale di una superficie è la sua rappresentazione numerica tridimensionale. Il modello è formato da un insieme di coordinate note in un sistema di riferimento arbitrario

in tre dimensioni. In realtà, per descrivere questo prodotto, vengono utilizzati diversi termini (DTM, DHM, DGM…), considerati spesso sinonimi, che si differenziano a seconda del tipo di dati che contengono.

Consideriamo solo il DTM (Digital Terrain Model). I dati racchiusi in esso non comprendono solo le quote, ma anche le caratteristiche naturali del terreno, come fiumi, vegetazione ecc..

La costruzione di un DTM è composta da quattro fasi: rilevamento, elaborazione, archiviazione e rappresentazione dei dati.

Il rilevamento dei dati ha lo scopo di creare informazioni discrete che sono alla base del DTM. Le coordinate dei punti del terreno possono essere determinate tramite una cartografia esistente, un rilievo topografico, un telerilevamento, ovvero da riprese effettuate dal satellite nello spazio, infine tramite fotogrammetria, ovvero riprese aeree. In base al modello di acquisizione il dato può essere strutturato in modi diversi. Tipico della fotogrammetria è quello a griglia, la quale può essere regolare o irregolare. Inoltre sono utilizzati la distribuzione sparsa, detta "a piano quotato", e il metodo a curve di livello e linee di discontinuità. Ci si può trovare anche di fronte a schemi misti, in funzione sia del territorio che di eventuali rilevamenti parziali o totali precedenti.

L'elaborazione dei dati è la fase di costruzione vera e propria del modello digitale. Essa implica la determinazione di una funzione analitica in grado di rappresentare la superficie nella forma $z = f(x, y)$, dove z indica la quota di ciascun punto di coordinate planimetriche x, y. La costruzione viene generalmente effettuata con procedure interpolative locali, come il metodo a matrice e la triangolazione.

Con la prima, l'interpolazione avviene secondo funzioni splines bilineari e bicubiche, su punti disposti sulle maglie di una griglia regolare.

Nel metodo a triangolazione invece, l'interpolazione avviene mediante piani, su punti la cui disposizione può essere qualsiasi. Essi sono collegati tra loro a formare dei triangoli non sovrapposti. In questo caso è il dato rilevato che definisce il singolo triangolo, quindi non si tratta di una vera interpolazione.

I modelli a matrice hanno il vantaggio di avere una struttura regolare dei dati, la quale semplifica l'uso del DTM, ma, d'altra parte, la struttura triangolare è più precisa e comporta minore memorizzazione.

L'archiviazione del dato ha ormai una serie di requisiti di standardizzazione che rende qualsiasi DTM interfacciabile con i programmi di gestione più diffusi. Essa differisce logicamente in funzione al fatto che la fase di elaborazione sia stata effettuata con il metodo a matrice o quello a triangolazione.

Il file, nel metodo a matrice, è concettualmente una bitmap. Nell'header sono salvati il numero di righe (*n*) e di colonne (*m*), i valori minimi e massimi delle coordinate planimetriche e il passo della griglia. Il corpo del file è una serie di valori di quota disposti in una matrice *n* per *m*.

Nel metodo a triangolazione, invece, la struttura del file è costituita dall'insieme di parametri che definiscono i triangoli. La struttura è quindi meno immediata da comprendere e da gestire, ma è sicuramente un modello numerico che meglio si adatta a terreni con differente morfologia, ad esempio, in pianura i triangoli saranno meno che in collina.

La rappresentazione dei dati può essere scelta tra una vasta gamma di possibilità: curve di livello, profili, reticoli e punti isolati qualsiasi. Nella fase di trattamento dei dati esistono opportuni algoritmi che permettono di passare dall'uno all'altro tipo di rappresentazione.

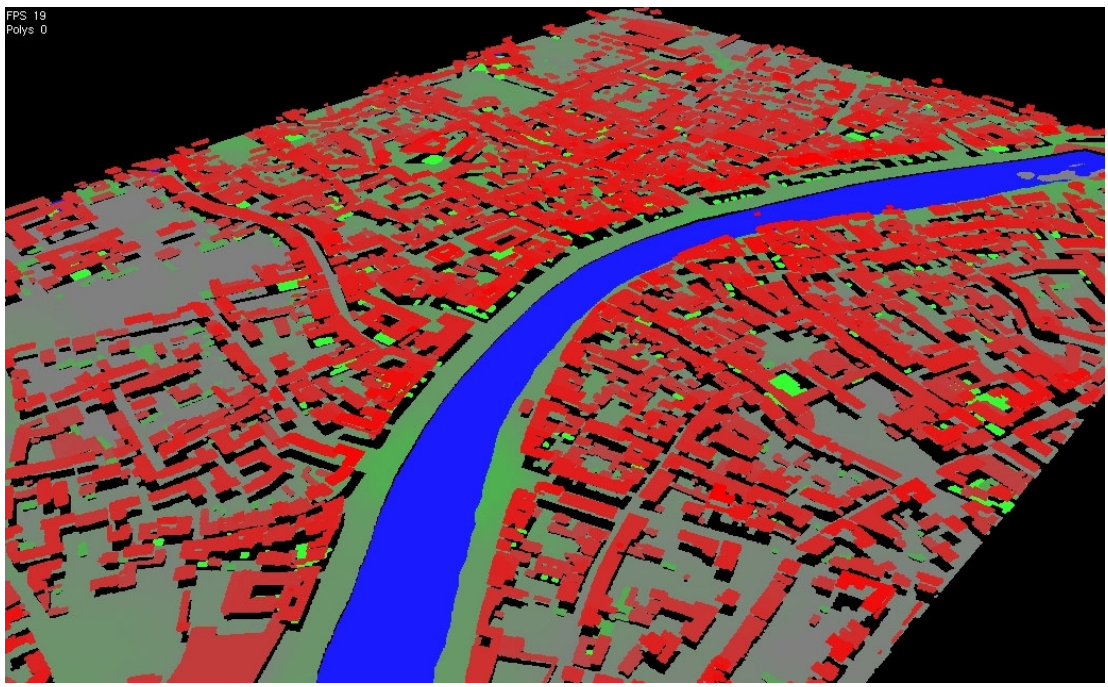


fig. 4 Rappresentazione del DTM di Pisa in XVR

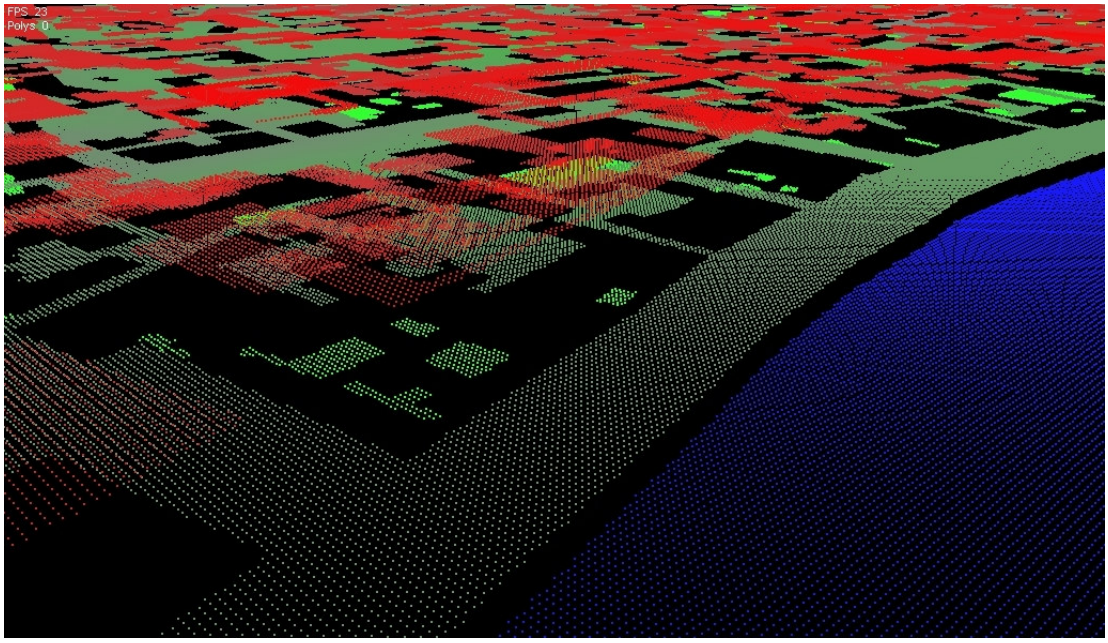


fig. 5 Rappresentazione a griglia regolare

Come detto in precedenza il DTM può contenere inoltre una serie di dati di tipo tematico-quantitativo di carattere ambientale, antropico, sociale, economico ecc..
Insomma, il DTM è evidentemente una buona base di partenza per la costruzione procedurale di una città. Con esso si ottengono dei buoni risultati in termini di fedeltà con la realtà, il suo punto debole è però il costo elevato del rilevamento dovuto all'avanzata tecnologia degli strumenti.

2.6 Analisi dell'immagine e semplificazione dei modelli

Esistono una grande quantità di algoritmi basati sull'elaborazione di immagini aree, sia normali che stereo (un'immagine stereo viene creata scattando due foto della stessa porzione di superficie da due angolazioni diverse).
Al contrario del DTM, questi algoritmi si basano sull'analisi dei pixel dell'immagine, essi sono in grado di fornire in output un insieme di linee rappresentanti il perimetro degli edifici visualizzati nelle foto. Si basano sul riconoscimento di alcuni pattern e primitive relative alle costruzioni (in questo caso può aiutare la classificazione, come visto in 2.3).

fig. 6 Analisi dell'immagine di una città

Come la maggior parte degli artefatti umani, gli edifici hanno delle proprietà come ad esempio la simmetria. Per questa ragione, durante il riconoscimento, si cercano di preservare strutture e simmetrie come la perpendicolarità, la complanarità e il parallelismo. Nel gergo del settore questa operazione è chiamata *model generalisation*, a cui solitamente è affiancata l'operazione di *surface simplification* (semplificazione della superficie), quest'ultima già utilizzata in domini diversi, come la visualizzazione di modelli altamente complessi.

1. *Model generalisation* – è la trasformazione di un oggetto in una geometria, topologia e semantica semplificata. Un primo approccio descritto in (Staufenbiel, 1973), era basato sull'intersezione di linee rette e su un insieme di regole in grado di risolvere i casi in cui le linee si presentavano troppo corte. Una soluzione più recente (Barrault et al., 2001) è data da un sistema *multi-agent*, in cui ogni *agent* si occupa di risolvere un dato problema geometrico. Questi compiti sono solitamente la retifica di un angolo, la cancellazione di edifici troppo piccoli e la fusione di edifici vicini. Un altro approccio (Meyer, 2000) suggerisce l'uso di una sequenza di operazioni per collegare dati semplificati che possono essere usati per costruire una struttura di livelli di dettaglio.
2. *Surface simplification* – è solitamente utilizzato su modelli poligonali come le mesh di triangoli. Il più importante algoritmo è basato sul raggruppamento dei vertici e sulla loro fusione (*edge collapse*). Ad esempio l'algoritmo di (Rossignac e Borrel, 1993) divide gli oggetti in una griglia regolare di cubi, in seguito tutti i vertici che appartengono allo stesso cubo vengono fusi in

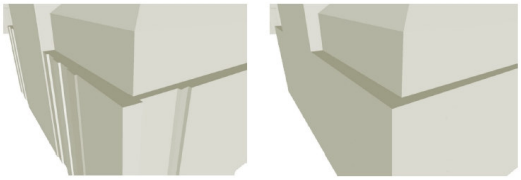


fig. 7 Semplificazione delle superfici

uno solo. Gli algoritmi presentati da (Hoppe, 1996) e (Garland e Heckbert, 1997) invece controllano che l'errore subito da una fusione sia inferiore di una certa soglia. Infine l'approccio di (Ribelles et al., 2001) si basa sul riconoscimento di piccole protuberanze e buchi e sulla loro eliminazione.

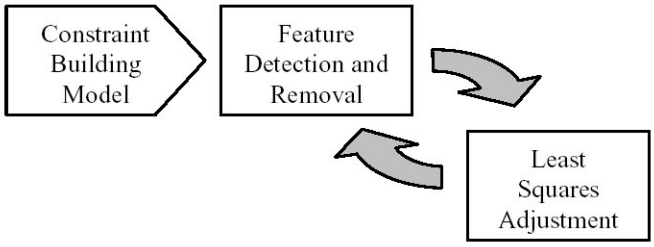


fig. 8 Schema generale di generazione e semplificazione degli edifici

2.7 L-System

L-System è una grammatica per un sistema a regole in grado di rappresentare in maniera semplificata la crescita di una forma biologica. Anche gli edifici possono essere considerati come organismi biologici, poiché, come le piante, seguono un determinato algoritmo di crescita (fondamenta, primo piano, eventuali piani superiori, tetto…) anche se questa deve essere contenuta con l'ausilio di ulteriori forzature.

Per semplicità vediamo un piccolo esempio di generazione di piante, comunque lo stesso ragionamento può essere esteso anche agli edifici digitali.

Inizialmente si determina un insieme di simboli alfanumerici (ad esempio "F", "+" e "-").

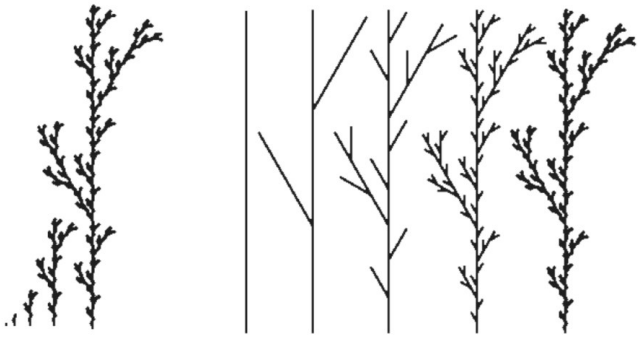


fig. 9 Esempio di L-System su piante

In seguito grazie alle produzioni dettate dalla grammatica i simboli non terminali vengono sostituiti con uno o più simboli (compresa la stringa vuota). Alla fine di ogni passo, cioè dopo che tutte le possibili

produzioni sono state applicate, la parola originale viene trasformata e utilizzata per il passo di sostituzione successivo. Ad esempio una produzione potrebbe essere:

$$\mathbf{F \rightarrow F + F - - F + F}$$

la quale significa che ogni F nella stringa in input deve essere espansa in F+F--F+F.
Questo incrementa il numero di simboli non terminali nella stringa e causa la crescita di ogni simbolo. Applicando un'altra volta la regola, la stringa diventa:

$$\underline{\mathbf{F + F - - F + F}} \mathbf{+} \underline{\mathbf{F + F - - F + F}} \mathbf{- -} \underline{\mathbf{F + F - - F + F}} \mathbf{+} \underline{\mathbf{F + F - - F + F}}$$

dove sono state aggiunte le sottolineature per rendere più agevole la leggibilità.
Le produzioni L-System sono applicate in parallelo espandendo più simboli possibile, oppure non oltre un certo limite prestabilito dall'utente. In questo modo la stringa in output non dipende dall'ordine in cui le produzioni vengono elaborate.
Infine, per creare un oggetto grafico, viene assegnato ad ogni simbolo un significato geometrico, ovvero una forma o una trasformazione.
Purtroppo l'uso di questi simboli non è ancora sufficiente per la sintetizzazione di una pianta. Infatti, mentre l'operazione di espansione viene fatta in parallelo, quella di lettura della stringa di output è seriale, perciò bisogna aggiungere dei caratteri speciali che consentano di tornare indietro. In questo caso specifico sono utili, ad esempio, quando si completa un ramo e si deve tornare al tronco.
Possiamo usare ad esempio i caratteri "[" "]" per indicare le operazioni di *store* e *restore*. In questo caso la produzione potrà essere:

$$\mathbf{F \rightarrow F [+ F] F [- F] F}$$

La figura mostra due modi per ottenere l'albero, a sinistra l'albero è costruito seguendo la vera crescita di un albero, mentre a destra è costruito in modo gerarchico (metodo più efficiente per la fase di processing e di rendering).
Sinora le produzioni viste erano utilizzate per la costruzione di oggetti grafici in due dimensioni, ma è molto semplice estendere questo metodo al mondo tridimensionale.

Cambiamo ad esempio i significati grafici dei simboli finora usati. La rappresentazione grafica di "F" diventa la forma di un cilindro. Aggiungendo il simbolo "&" a "+" e "-" possiamo assegnare loro il significato delle tre rotazioni sui tre assi, chiamati in inglese *yaw, pitch* e *roll* dai loro significati in gergo aeronautico, e con i simboli "\" e "/" possiamo esprimere la rotazione in senso orario o antiorario.
Con la seguente produzione:

$$F \rightarrow F[\&F][\&/-F][\&\backslash F]$$

è possibile costruire un albero ternario.'albero al primo passo e dopo più iterazioni.

fig. 10 Albero al primo passo e dopo più iterazioni

Osserviamo che l'albero generato è troppo regolare, questo perché ogni simbolo di rotazione ha un angolo prestabilito, la parametrizzazione di questi controlli permette invece una maggiore flessibilità e varietà delle forme.
Ad esempio il simbolo +(30) descrive una rotazione del disegno di trenta gradi sull'asse X. E' possibile aggiungere anche simboli per la scalatura della forma, "!" scala l'oggetto successivo della metà. Perciò la stringa "F[!+(30)F]F" disegnerà due cilindri uno sopra

l'altro con un cilindro più piccolo fra i due che spunta lateralmente, inclinato di trenta gradi. Naturalmente anche quest'ultimo simbolo può essere parametrizzato con "!(d)" dove d indica la percentuale di scalatura o il diametro del cilindro.

Come possiamo notare è facile che il numero di simboli superi lo schema ASCII, quindi spesso si usano simboli con più di un carattere, i quali solitamente sono preceduti dal prefisso "@".

Sinora abbiamo utilizzato forme predefinite come il tratto e il cilindro, ma grazie alla grammatica possiamo dare dei comandi che definiscano anche poligoni. Con "F" adesso descriviamo un *edge* (o spigolo), con "{" e "}" delimitiamo la sottostringa che occorre per la sola creazione del poligono, con "." definiamo un vertice e con "G" ci spostiamo.

Possiamo creare anche *patch* utilizzando il simbolo "@PC(n,i,j)", etichettata "n" con il *control point* in posizione (i,j). L-System è anche in grado di creare o caricare texture e di eseguire comandi del sistema operativo.

Sinora sono state descritte le funzionalità generali che riguardano i simboli di un L-System. Ma un L-System può essere ancor più potente con l'aggiunta di particolari comandi nelle produzioni.

L-System visto sinora è deterministico. Ciò significa che derivando più volte la stessa stringa d'origine, si ottiene sempre lo stesso risultato. Questo significa che L-System può descrivere ogni volta una pianta individuale e non una specie di piante (un palazzo e non un tipo di palazzo). Questo è un problema se immaginiamo di dover creare un prato di margherite (o una città). Per far sembrare il prato realistico dovremmo dare ad ogni margherita un L-System diverso, ma queste non dovrebbero differire molto tra loro per sembrare della stessa specie.

In questo caso si usano L-System stocastici, i quali descrivono più produzioni alternative per uno stesso simbolo. Ognuna di queste produzioni ha la sua probabilità (la somma delle probabilità deve essere uguale a 1), che è scritta sopra o davanti alla freccia. Quando, durante la derivazione, il simbolo verrà trovato, sarà generato un numero random secondo la probabilità delle sue produzioni, che determinerà la strada da seguire.

Qui sotto un esempio:

$$\mathbf{F}_{0,7} \rightarrow \mathbf{F\,[\,+\,F\,]\,F\,[\,-\,F\,]\,F}$$

$$\mathbf{F}_{0,3} \rightarrow \mathbf{F\,[\,+\,F\,]\,F}$$

Talvolta però può essere necessario ricreare la stessa identica pianta. Questo è comunque possibile perché le funzioni random in realtà sono *pseudo-random*, quindi basta richiedere come parametro il seme della serie (cioè il numero di partenza) e utilizzare lo stesso quando vogliamo riprodurre la stessa pianta.

2.8 Altri metodi procedurali

In questo paragrafo considereremo alcune tecniche procedurali che non si basano su dati fotogrammetrici e analisi di foto aeree, ma su regole di costruzione generali.
La "LaHave House" (RauChaplin, MacKayLyons e Spierenburg, 1996) ha creato un'applicazione per aiutare la progettazione di case semplici e funzionali. Essa si basa su una grammatica delle forme da cui si può ricavare una libreria di elementi architetturali. La generazione della libreria richiede molto tempo, ma si ottengono risultati molto realistici.
BatiMan (Champciaux, 1998) invece è un modellatore dichiarativo. Si parte da un piccolo numero di edifici, il quale viene poi espanso tramite un'intelligenza artificiale in grado di apprendere. Le soluzione sono classificate e mostrate all'utente, il quale può agire all'incirca su una ventina di parametri.
La tecnica delle *split grammars* descritta in (Wonka, WimmerSillion e Ribarsky, 2003) è una grammatica parametrica basata sul concetto di forma. Il sistema utilizza inoltre una seconda grammatica detta di controllo (*control grammar*), che verifica la correttezza della propagazione degli attributi.
Il modulo GenVillage dell'applicazione AGETIM (Lager, Goff, Chamsseix, Chetala e Larive, 2005) permette di creare gli edifici a partire dalla loro base (in modo automatico o manuale) usando template predefiniti. Gli edifici così creati hanno altezze e tipi di tetto variabili e si adattano automaticamente all'ambiente (stile, orientazione e texture) permettendo una buona integrazione.

2.9 Caso studio 1: Parish - Muller

Prendiamo in considerazione il documento "*Procedural modeling of cities*" di Parish e Muller e il loro *CityEngine System*, più precisamente approfondiamo le parti riguardanti la divisione in lotti, la creazione delle geometrie e delle texture.

Ritornando alla suddivisione in fasi del paragrafo 2.2, la fase detta *Lots* consiste nel dividere gli isolati in aree (*allotments*) destinate ai singoli edifici. Nel metodo suddetto si suppone che gli *allotments* siano geometrie convesse di forma rettangolare. Un isolato viene diviso in tante piccole unità utilizzando un semplice algoritmo ricorsivo che divide il lato più lungo e più parallelo all'orientamento degli edifici, fino a che l'area dell'*allotment* non supera inferiormente una certa soglia specificata dall'utente. Dopo la suddivisione degli isolati, tutti gli *allotments* troppo piccoli o che non hanno accesso diretto alla strada vengono eliminati. In molte grandi città, dove sono presenti grattacieli, esistono piani regolatori per il controllo delle altezze, poiché gli edifici più bassi possono rimanere per troppo tempo in ombra. Questo sistema permette di restringere la generazione di grattacieli solo a certe zone, tramite una bitmap creata dall'utente, in cui ad ogni colore è associata un'altezza.

fig. 11 Composizione dei block e dei lots

La successiva fase detta *Building* crea le geometrie degli edifici grazie all'utilizzo di un L-System stocastico (in cui alcuni valori sono casuali) e parametrico. Come detto in precedenza per ogni allotments è generato un unico edificio. Per creare varietà di stili, gli edifici sono divisi in tre categorie: grattacieli, edifici commerciali e case residenziali. Anche questo fattore, legato comunque alle altezze, è controllato tramite una bitmap.

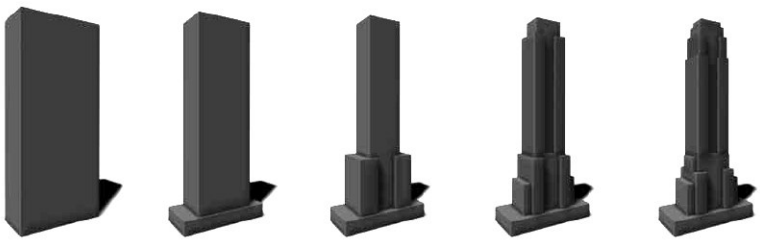


fig. 12 Diversi livelli di dettaglio di un edificio

Per ogni tipo di edificio esiste un insieme diverso di produzioni da eseguire. I comandi sono: scalatura, spostamento, estrusione, biforcazione e terminazione. Infine i tetti sono

costruiti tramite template. La forma finale degli edifici è determinata dal suo primo piano, il quale viene trasformato secondo l'output de L-System.

Anche le texture sono create proceduralmente. In altri sistemi le foto dei veri edifici vengono scansionate, modificate e proiettate sulla superficie della geometria. Questo metodo produce delle texture molto dettagliate, a discapito di una grande quantità di lavoro, mentre la creazione procedurale è assegnata al computer.

Le immagini delle facciate vengono sintetizzate in maniera semi-automatica usando un sistema a livelli e una semplice funzione di composizione detta *layered grid*. Essa è costituita da una struttura gerarchica di griglie, basate su gruppi di intervalli costituiti da una serie di intervalli non sovrapposti. Il risultato finale dell'algoritmo è lo schema dei piani e delle finestre.

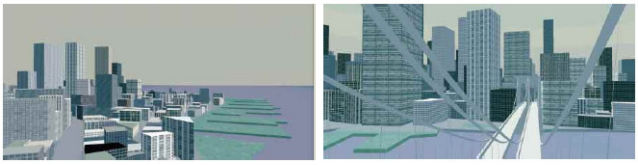


fig. 13 Esempi di resa finale

Il punto di forza del *CityEngine* è la sua capacità di creare molto velocemente un infinito numero di città diverse partendo dalla stessa immagine in input senza l'ausilio di fotografie aeree.

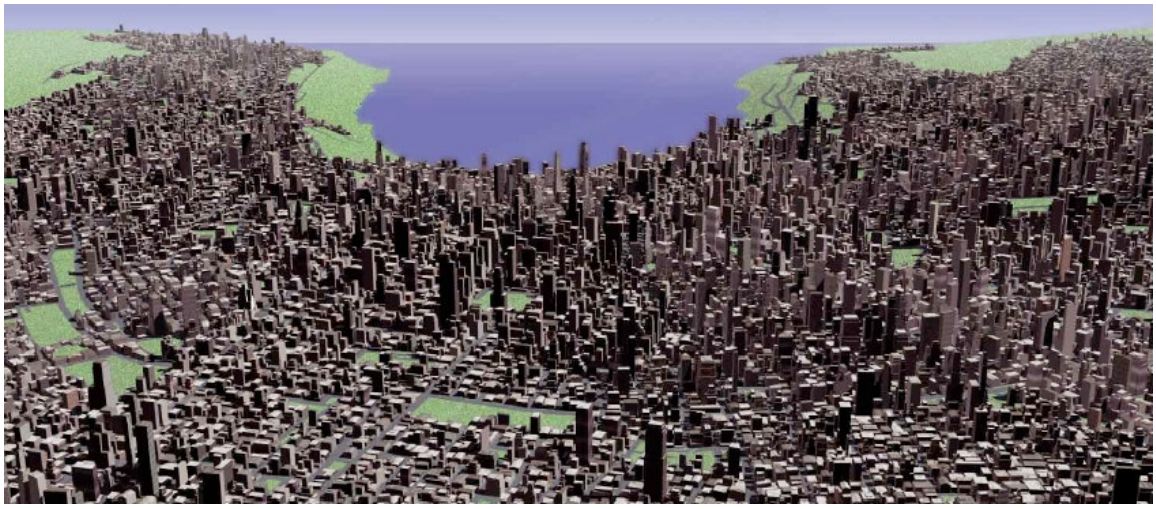


fig. 14 Panoramica di una città col metodo di Parish e Muller

2.10 Caso studio 2: Greuter – Parker – Stewart – Leach

Il framework descritto nel documento "*Real-Time Procedural Generation of Pseudo-Infinite cities*" è progettato in maniera specifica per applicazioni in tempo reale. Esso è formato da tre componenti: *view frustum filling*, *geometry caching* e *geometry generation*.

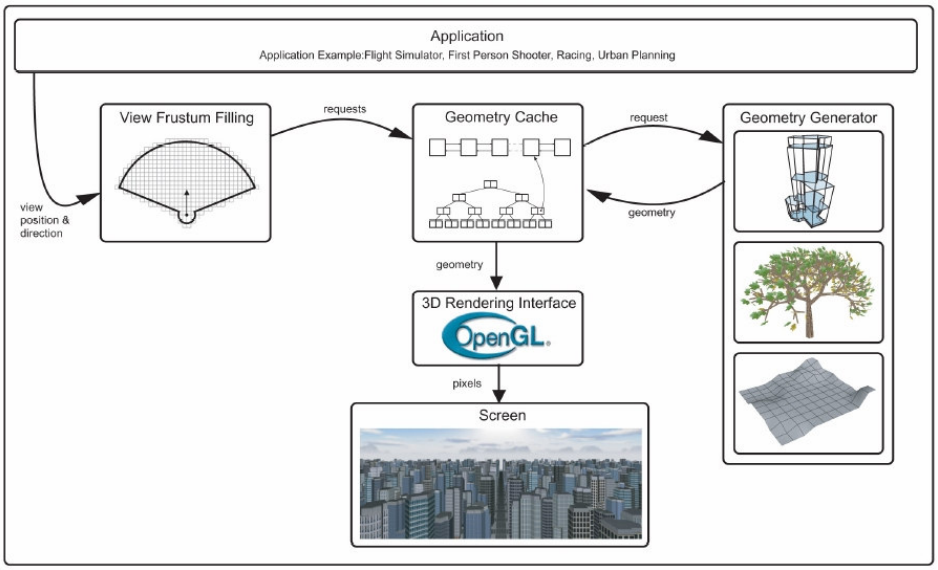


fig. 15 Schema dell'applicazione

Durante una navigazione interattiva solo una frazione del mondo è visibile all'utente, è perciò inutile gestire gli oggetti che non si trovano nell'inquadratura. Nella pipeline di rendering di una applicazione tridimensionale non procedurale esiste una fase chiamata *view frustum culling*, essa si occupa di eliminare tutti gli oggetti (pre-esistenti) fuori vista. Il *view frustum filling* è la tecnica duale. La scena all'inizio della pipeline è vuota, e il *view frustum filling* individua gli oggetti visibili, poiché solo essi verranno generati.

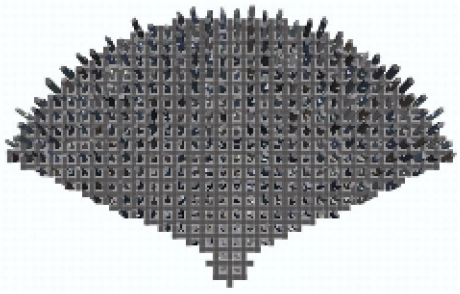


fig. 16 Porzione della città generata tramite view frustum

Il componente di *geometry caching* si occupa di mantenere in memoria gli edifici generati, in

modo da non doverli generare nuovamente ad ogni frame. L'algoritmo utilizzato nella cache è LRU (*least recently used*): ogni volta che viene generato un nuovo edificio, questo è memorizzato al posto di quello visto meno recentemente.

Infine la generazione è eseguita dal *geometry generator*. Ogni tipo di edificio è specificato tramite un "seme" e dei parametri opzionali che influiscono sull'altezza, la lunghezza e la profondità. Gli edifici creati da uno stesso tipo saranno soggetti all'insieme di regole di generazione di quel tipo. Un generatore di palazzi con uffici, ad esempio, può creare solamente una varietà di palazzi del tipo uffici, anche se il risultato non è prevedibile e alcune volte inaspettato.

La costruzione di un edificio è costituita da due fasi: generazione della pianta del palazzo, e generazione delle sue facciate.

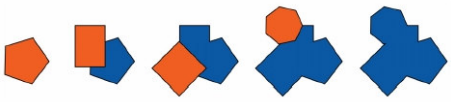


fig. 18 Pianta dei piani

La pianta è realizzata attraverso una funzione iterativa che ad ogni passo aggiunge una forma regolare (quadrato, pentagono esagono) di diversa dimensione. Le forme possono sovrapporsi.

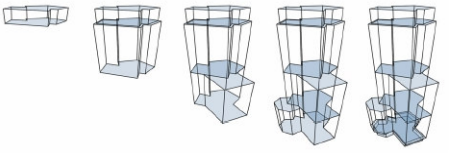


fig. 17 Generazione delle facciate

Nella seconda fase le facciate vengono create partendo dalla pianta dell'ultimo piano, essa viene estrusa verso il basso secondo un numero di piani arbitrario. Il processo è ripetuto per tutte le altre piante.

La città virtuale utilizza un set limitato di texture, le quali però potrebbero essere anche generate proceduralmente.

Questo framework genera una città virtuale in piano, formata da una griglia regolare di strade con edifici di tipo "grattacielo".

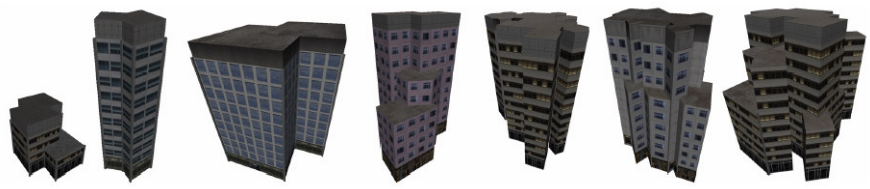


fig. 19 Esempi di edifici creati con l'applicazione

Consente la creazione di città veramente vaste, formate da $4x10^{18}$ edifici geometricamente differenti, circa seicento milioni di volte la corrente popolazione mondiale. La morfologia degli edifici è ispirata ai palazzi di Melbourne, ma cambiando parametri come la scalatura, le texture e l'algoritmo di generazione possono essere creati luoghi che non hanno nulla a che vedere con le comuni città.
L'utente che si addentra nella città creata da questo framework non ne vedrà mai la fine. Purtroppo la regolarità delle strade può, dopo un certo periodo, rendere noiosa la navigazione. Altro problema è la non staticità degli oggetti (se dovessimo ripercorrere all'indietro la strada fatta non ritroveremmo gli stessi edifici).

fig. 20 Panoramica della città (500 edifici)

2.11 Caso studio 3: Wonka – Wimmer – Sillion – Ribarsky

Il documento "*Instant Architecture*" presenta una procedura basata sulle *split grammars*, offrendosi come tecnica alternativa agli L-System.
Gli L-System funzionano molto bene nel caso di modellazione di forme naturali, poiché sono in grado di simularne efficacemente la crescita, mentre non si adattano facilmente a quella di edifici, i quali devono sottostare a molte più forzature spaziali e che spesso non riflettono un processo di crescita.

Al contrario le *split grammars* partono da una forma precisa, la quale viene suddivisa iterativamente in maniera gerarchica secondo dei pattern predefiniti, cosicché la modellazione rimane all'interno della forma iniziale. Le forme di base dell'applicazione sono cuboidi, cilindri e prismi.
La tecnica presentata introduce un sistema di confronto degli attributi (attribute matching system) e una grammatica di controllo, le quali offrono la flessibilità richiesta dalla modellazione degli edifici, usando una varietà di stili differenti.

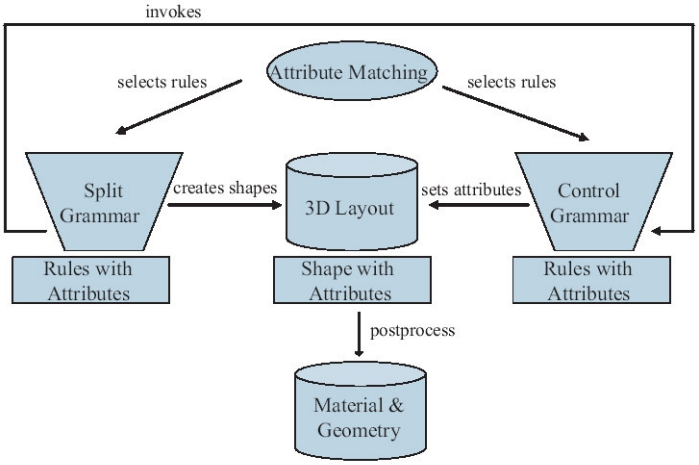


fig. 21 Schema dell'applicazione

Differentemente da paradigmi in cui viene creata una grammatica diversa per ogni oggetto che deve essere modellato, questo si basa sulla creazione di un grande database di regole appartenenti alla *split grammar*, con le quali modella tutti i tipi di edificio.
Il database può essere esteso dall'utente, tuttavia la complessità inerente dell'architettura richiede una certa esperienza.
L'utilizzo più comune di questa grammatica è quello di scegliere manualmente, ad ogni passo di derivazione, le regole più indicate. Anche se questa è una buona strategia, esiste la possibilità di creare il modello automaticamente tramite un processo stocastico che comunque cerca di mantenere la consistenza del modello. Il controllo della consistenza diventa più complicato con l'aumentare delle regole nel database.
La *split grammar* descritta in questo documento risolve il problema poiché riesce a fondere diversi componenti dell'edificio tramite restrizioni sulle regole permesse.

Inoltre, il sistema di confronto dei parametri permette all'utente di specificare ad alto livello gli obiettivi, e controlla in maniera casuale la consistenza dell'output.

La grammatica di controllo è una semplice grammatica libera da contesto, che gestisce la distribuzione spaziale secondo i principi architetturali.

Riassumendo, ecco come viene eseguito un passo di derivazione, l'algoritmo parte dalle facciate:

1. la *split grammar* cerca nel database tutte quelle regole che sono compatibili con la forma corrente.
2. Il sistema di confronto degli attributi viene invocato per scegliere la migliore fra queste regole, esso si basa sul confronto tra gli attributi della forma corrente e quelli propri delle regole. Gli attributi servono inoltre in fase di post processing per la definizione delle geometrie e delle texture.
3. Le forme generate ereditano tutti gli attributi di quella corrente.
4. La grammatica di controllo viene invocata per una buona distribuzione spaziale. La derivazione della grammatica di controllo è soggetta anch'essa al sistema di confronto degli attributi, come la split grammar.
5. La *split grammar* è invocata ricorsivamente su tutte le nuove forme generate.

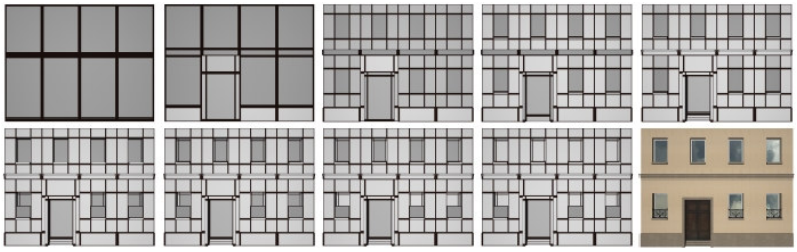


fig. 22 Fasi di divisione della geometria

fig. 23 Esempi di facciate

L'applicazione è in grado di dare un ottimo livello di dettaglio delle facciate poiché gli elementi, come ad esempio le finestre, fanno parte del modello e non della texture. Questo però influisce su una maggiore pesantezza dei modelli. Inoltre il sistema ha poca flessibilità per quanto riguarda la forma esteriore dei palazzi.

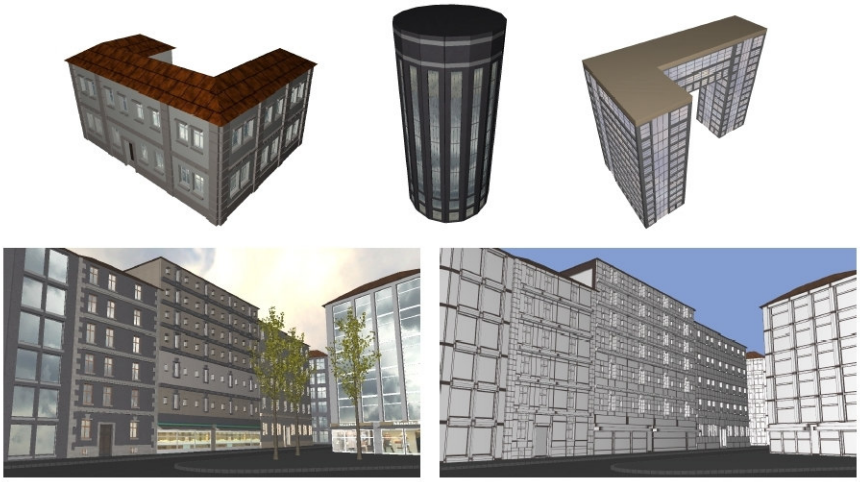


fig. 24 Esempi di edifici generabili

2.12 Caso studio 4: NetLogo

In questo caso l'applicazione proposta nel documento "*Procedural Modeling of Land Use in cities"* (Lechner, Watson, Ren, Wilensky e Tisue) non produce una rappresentazione tridimensionale della città virtuale, ma è comunque degna di nota in quanto il programma considera anche l'elemento tempo, ovvero visualizza in maniera isonometrica (tramite il motore grafico di SimCity) la crescita della città passo dopo passo.
L'utente inserisce la mappa delle quote del terreno e un insieme di parametri globali, che possono comunque essere generati dal programma.
Il sistema offre vari stili di generazione della rete stradale: Manhattan, Parigi, ecc..
Il territorio accessibile dalla rete stradale viene popolato tramite una mappa di densità inserita dall'utente. La simulazione, come detto in precedenza, è continua, ma l'utente può

fermarla in qualsiasi istante e modificare le regole prima che la costruzione di un nuovo quartiere abbia inizio.

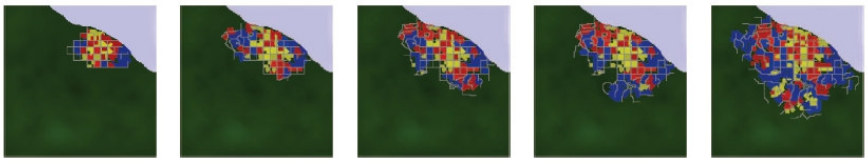


fig. 25 Crescita della città nel tempo

Questa simulazione utilizza diversi tipi di *agent* (programmi "intelligenti") per generare i più comuni componenti della città: edifici residenziali, commerciali e industriali.

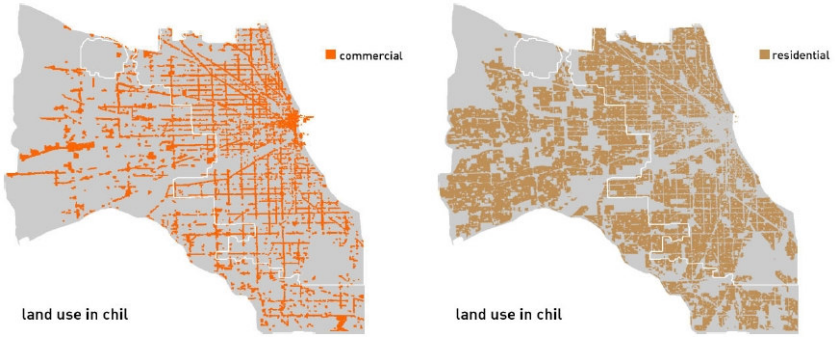


fig. 26 Distribuzione di edifici commerciali e residenziali nella realtà (Chicago)

Gli *agents* descrivono il comportamento e la semantica locale, interagiscono direttamente con l'ambiente circostante e indirettamente tra loro. Il mondo in cui agiscono è rappresentato tramite una griglia di *patches* (pezze).

Le *patches* possono essere occupate dalla strada, la quale può essere di due tipi: terziaria o primaria. Le strade terziarie sono le più piccole e le più diffuse, mentre le primarie sono le più grandi e le più attraversate.

Se le *pacthes* non contengono una strada sono raggruppate in *parcels*, le quali sono proprietà di un determinato agent costruttore.

L'agent costruttore acquisisce tutte le informazioni della *parcel* e gestisce gli attributi dell'edificio, come popolazione, età e valore. Inoltre, gli agents, tengono traccia degli edifici vicini poiché l'edificio in costruzione dipende dal tipo di quelli già esistenti.

Le *patches* mantengono informazioni riguardanti la loro interazione con gli agents. Molto importante è il *value*, un parametro che descrive il prezzo e l'importanza della *patch* agli occhi dell'*agent* costruttore. Per velocizzare i calcoli le informazioni della *patch* vengono ricalcolati solo dopo che un *agent* li richiede.

fig. 27 Viste della città realizzata con il motore grafico di Sim City

Ogni *building agent* segue lo stesso paradigma algoritmico. Trascorre un determinato periodo alla ricerca di un buon posto dove iniziare a costruire. Una volta trovato, l'*agent* genera un'ipotetica soluzione e la propone alla città (come avviene nella realtà tra imprese e comuni). La città testa la proposta, e se la proposta passa l'*agent* acquista il terreno e ci costruisce.

Per la generazione delle parcels e la costruzione degli edifici esiste un altro tipo di agent: developers. Ogni tipo di developers ha un differente metodo di gestire gli attributi del terreno. I valori sono determinati tramite una somma pesata di questi.

I developers residenziali preferiscono regioni con strade poco trafficate, evitano zone industriali e preferiscono stare vicino all'acqua. Quelli industriali invece scelgono solitamente zone povere e evitano zone residenziali. Infine quelli commerciali possono stare sia vicino a zone residenziali che a quelle industriali e sono interessati particolarmente alle strade con molto traffico.

fig. 28 Panoramica della città generata da NetLogo

2.13 Caso studio 5: MapCube

"*Automatic Generation of 3d City Models and Related Application*" di Takase, Sho, Sone e Shimiya, descrive un'applicazione commerciale giapponese implementata dalla *Cad Center Corporation*, la *MapCube*, la quale è in grado di generare automaticamente una città tridimensionale utilizzando laser profiler data, mappe 2d digitali e immagini aeree.
Già dal 2002 sono state riprodotte tutte le maggiori città del Giappone.

fig. 29 Scorcio di Ginza

Il sistema è suddiviso in diversi componenti: *3d city model generation program*, *database management program*, *material data output program* e un visualizzatore *3D CG/VR*.

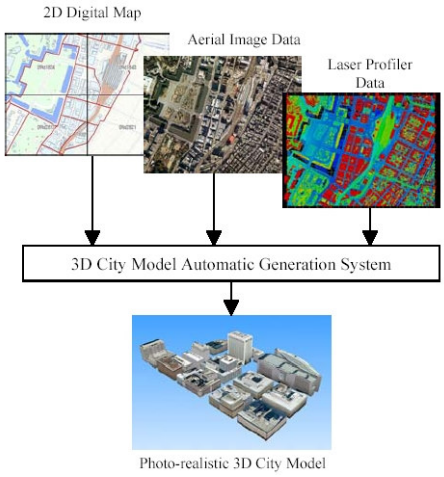


fig. 30 Schema del sistema

Il primo componente è anch'esso composto da più sotto programmi:

1. *Building model generation* – produce automaticamente i dati per la costruzione dell'edificio definendo i contorni tramite mappe 2d digitali e la geometria interna tramite laser profiler data (LPD).
2. *Building model edition* – è un editor supplementare che permette di creare tutte quelle geometrie che il building model generator non è in grado di modellare.
3. *Terrain modeling* – per la generazione automatica dei terreni tramite LPD.

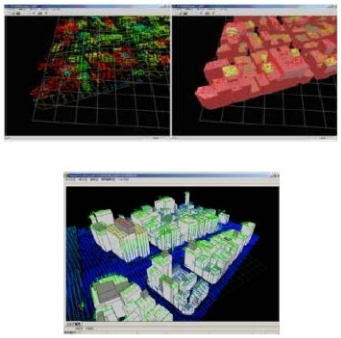


fig. 31 Tools del sistema

4. *Overpass and elevated object modeling* – produce le geometrie di tutti quegli oggetti che risultano sopraelevati, come ad esempio autostrade, ponti, binari. Utilizza la mappa 2D e LPD.
5. *Texture edition* – questa parte non è automatica, è un editor per l'assegnamento delle texture alla geometria.

Esiste comunque un insieme di edifici già texturizzati, preparati manualmente dagli autori che possono essere adoperati in maniera casuale.

Il secondo componente, il *database operation program*, controlla il trasferimento dei vari dati tra il database della città e il 3d city model generation program. Il suo lavoro consiste nella registrazione dei dati del database, nell'estrazione della geometria, delle texture e di tutti gli attributi di una determinata area.

Il terzo componente, il *material data input program*, gestisce e coordina i file, riassegnadoli a directory diverse e cambiando loro i nomi.

Infine, il *3D CG/VR data output program* riceve dal database della città i dati controllati dal *database operation program* e li converte in un formato comune, come ad esempio *WaveFront OBJ*.

Il programma produce un esatto e abbastanza dettagliato modello della città con l'ausilio di una grande quantità di dati. Gli autori si propongono di accorciare i tempi di modellazione, tuttavia rimangono i tempi relativi alla fase di raccolta dei dati. Si dimostra comunque un efficiente tool rispetto alla modellazione totalmente manuale tramite piattaforma Cad.

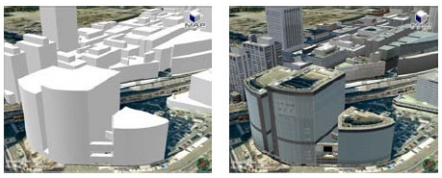


fig. 32 Esempio di modello con e senza texture

3 AB-Block: il progetto

3.1 Introduzione

AB-Block è il risultato dello studio e della ricerca di questi miei ultimi mesi di lavoro compiuti nel settore della modellazione procedurale di edifici. Benché esistano, come descritto nel capitolo precedente, vari paradigmi, la scelta è stata quella di concepire un nuovo modo di sintetizzare la città, dando particolare attenzione alla disposizione degli edifici e alla loro geometria, prendendo come base di partenza gli isolati (*blocks*).

fig. 33 Logo di AB-Block

AB-Block fa parte di un progetto molto più ampio sulla modellazione procedurale e l'analisi dell'immagine realizzato dal laboratorio Percro della scuola superiore Sant'Anna, a cui hanno fatto parte altri laureandi, finalizzato alla riproduzione automatica della città di Pisa tramite una semplice cartografia stradale e una mappa dei rilievi del terreno. Il sistema

è comunque generale, in grado di creare svariati tipi di città, da quelle medievali alle metropoli di questi ultimi decenni.

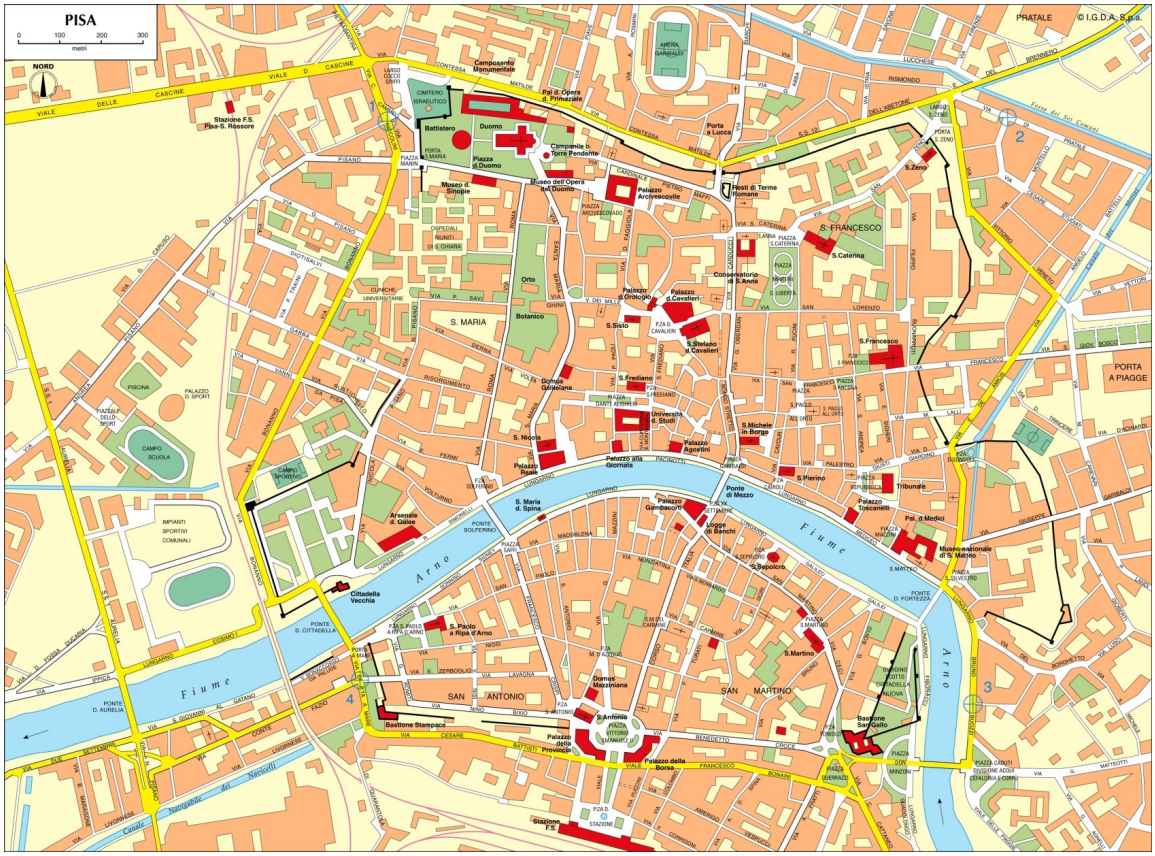


fig. 34 Cartina stradale di Pisa

L'obiettivo del progetto non è la riproduzione esatta della città, ma una copia il più possibile verosimile; d'altronde i dati richiesti in input non sarebbero comunque soddisfacenti per qualsiasi programma che si occupi di ricostruzioni identiche.

AB-Block riceve dati da un programma di elaborazione di immagine, che trasforma le scansioni delle cartografie in mappe vettoriali, in cui sono contenuti i dati relativi ai contorni degli isolati e le quote del terreno. L'isolato elaborato viene salvato in un file *lob*, che sta per *line of block*, ovvero perimetro dell'isolato. Nel file sono contenute le coordinate tridimensionali dei vertici.

Solitamente la costruzione delle città virtuali è demandata ad un unico programma che può mantenere informazioni globali sulla conformazione e la distribuzione degli isolati. Queste informazioni aiutano l'elaborazione nei passi successivi.

AB-Block invece, come abbiamo detto, ha solo la visione di un isolato, non conosce l'ambiente che lo circonda, ma, a differenza di altri sistemi, si focalizza maggiormente nella creazione dei cosiddetti *lots*, ovvero i siti in cui sorgeranno gli edifici.

I *lots* generati sono dei quadrilateri: questa semplificazione degli edifici, utilizzata anche dalla maggior parte delle applicazioni di questo settore, non limita in modo pesante la varietà di edifici che è possibile creare, in quanto le costruzioni umane sono solitamente quadrilateri o composizioni di essi.

AB-Block-Code è l'insieme di funzionalità e di algoritmi elaborati tramite euristiche concepite attraverso lo studio delle mappe urbane. E' inoltre una libreria di funzioni geometriche su linee intese come segmenti di retta, le quali sono le strutture alla base degli algoritmi.

Si è data particolare attenzione anche alla complessità delle geometrie poiché gli isolati generati possano essere utilizzati sia in applicazioni *batch*, come rendering di filmati, che quelle in *real time* come i videogiochi.

AB-Block inoltre offre una comoda interfaccia 2D contenete un visualizzatore 3D OpenGL.

Il tipo di modellazione procedurale utilizzato per l'elaborazione è quello parametrico stocastico. Una volta calcolato il lots, l'utente può modificare a suo piacimento tutti i parametri, i quali sono stati selezionati per dare un'ampia varietà di stili con un piccolo numero di scelte.

L'interfaccia include inoltre un editor per la creazione e l'applicazione veloce di texture prodotte proceduralmente, in maniera specifica, per le facciate delle abitazioni.

Il codice è stato realizzato in linguaggio C++ con l'ausilio di Visual Studio .net con l'integrazione del QT Designer della TrollTech, utilizzato per la composizione della parte 2D dell'interfaccia, e con le librerie di OpenGL (opengl32.lib e glu32.lib) per la parte tridimensionale.

3.2 Data Amplification e Lazy Evaluation

Molti sistemi di modellazione manuale seguono lo stesso standard di flusso dei dati: il modello concettuale articolato dall'utente viene passato al *modeler* che lo interpreta e lo converte in una rappresentazione intermedia utilizzabile nella fase di *processing* e di *rendering*. Il render accetta quest'ultima rappresentazione e sintetizza l'oggetto in immagine. Infine l'utente controlla il modello renderizzato e lo confronta con quello concettuale per apportare, nel caso sia necessario, le opportune modifiche.
La sintesi procedurale segue lo stesso percorso, sebbene l' implementazione delle specifiche della rappresentazione intermedia possono seguire due diversi paradigmi: Data Amplification e Lazy Evaluation.
Nel caso del paradigma Data Amplification una piccola quantità di dati viene trasformata in un oggetto grafico molto dettagliato visualizzato grazie ad un algoritmo di espansione. I modellatori che seguono questo paradigma sintetizzano alcuni tipi di rappresentazione intermedia. Quest'ultima non è altro che la descrizione della scena costituita da poligoni e altre primitive. Purtroppo, un effetto collaterale è dato da una possibile amplificazione esagerata della geometria, che può non essere supportata dall'applicazione o dall'hardware, è necessario perciò un controllo sul numero dei poligoni generati durante il processo di espansione.

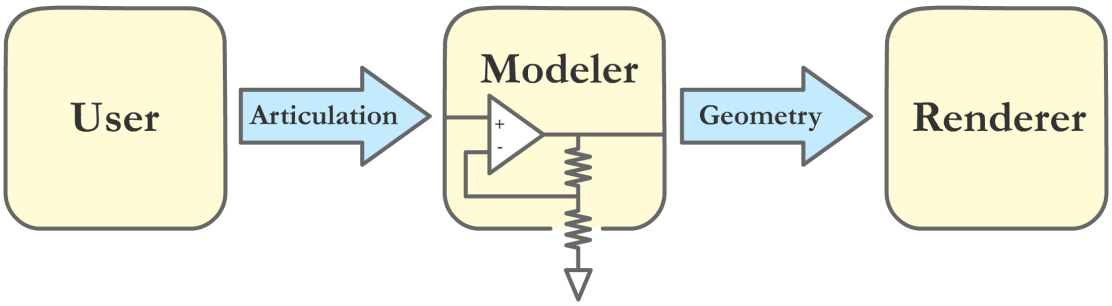


fig. 35 Schema del paradigma Data Amplification

Nel paradigma Lazy Evaluation il problema dell'amplificazione dei dati è evitato eseguendo la sintesi solamente se necessario. Lazy Evaluation è utilizzato soprattutto nelle comunicazioni *client-server* tra il modeler e il render, permettendo al modeler di generare solo

la geometria richiesta per la fase di *rendering* (solitamente la parte visibile della scena), risparmiando quindi sia tempo che spazio di memoria.
Il render deve sapere come richiedere la geometria che serve, mentre il modeler deve sapere quale geometria deve generare, quindi è necessario uno scambio di informazioni tra i due.
AB-Block supporta entrambi i paradigmi.

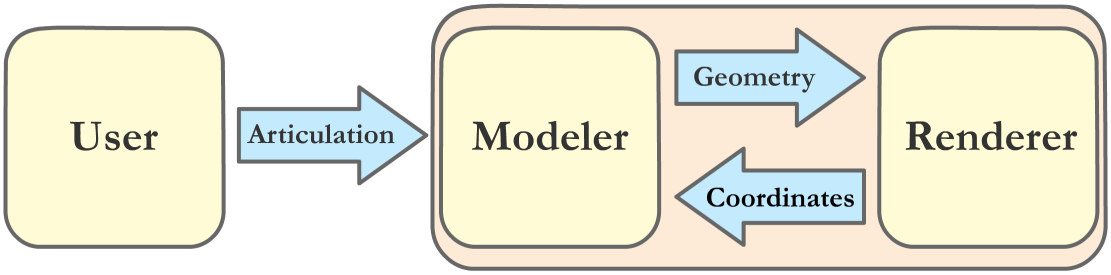


fig. 36 Schema del paradigma Lazy Evaluation

L'algoritmo di espansione degli edifici tramite procedura stocastica permette di creare un isolato contenete anche decine di palazzi attraverso l'utilizzo di un basso numero di dati in input (i vertici del perimetro), inoltre può essere specificato il livello di dettaglio in modo da poter controllare la pesantezza della geometria.
Nel caso si vogliano ricreare edifici non casuali, i dati per la generazione rimangono sempre contenuti, infatti l'algoritmo ha bisogno esclusivamente dei vertici della base dell'abitazione e di pochi suoi attributi.
Questo rende AB-Block-Code uno strumento ideale per applicazioni in real-time attraverso rete WAN, come ad esempio giochi di ruolo di massa, in cui è decisiva la quantità di dati da scambiare tra il server e il client.

3.3 Strumenti di sviluppo: Qt

Qt della TrollTech è un framework per lo sviluppo di applicazioni in C++ composto da una libreria di classi e un designer per la creazione visuale delle finestre. La libreria dispone approssimativamente di 400 classi completamente object-oriented, e di molte delle infrastrutture di cui si ha bisogno per creare una piattaforma server e un'applicazione client.

La libreria contiene classi per la composizione di GUI (Graphic User Interface), layout database, networking, XML e molto altro ancora.

fig. 37 Logo di QT

Gli oggetti dell'interfaccia, i widget, sono gestiti tramite il meccanismo dei cosiddetti *Signal* e *Slot*.

Solitamente la sorgente dei problemi, quando si sviluppa un'interfaccia è la gestione della comunicazione fra i diversi componenti. La soluzione di Qt è data appunto dal meccanismo dei segnali e degli slot, i quali facilitano in maniera sicura la comunicazione inter-object. Infatti in molte applicazioni accade che si voglia notificare lo stato di un widget ad un altro widget.

Nei vecchi toolkit questo tipo di comunicazione viene svolto attraverso le callbacks, ovvero puntatori a funzione che vengono passati alle funzioni chiamanti in risposta a determinati eventi. Il sistema delle callbacks ha due difetti fondamentali: non è sicuro che una funzione chiami la callback con i giusti argomenti e le callbacks sono fortemente accoppiate alla funzione chiamante che deve sapere quale funzione chiamare.

Nel meccanismo utilizzato da Qt viene emesso un segnale ogni volta che occorre un particolare evento. Esistono già una serie di segnali predefiniti, ma possono essere modificati tramite il meccanismo dell'ereditarietà. Un slot è la funzione chiamata in risposta al segnale. Anche in questo caso è possibile crearne dei propri.

Questo meccanismo è sicuro, in quanto la firma del segnale deve coincidere esattamente con la firma dello slot (anche se lo slot può avere parametri in meno, poiché può ignorare argomenti extra). Al contrario delle callbacks, i segnali non sono fortemente legati agli slot, infatti la classe che emette il segnale non conosce assolutamente lo slot che lo riceve; non solo, la stessa classe che contiene gli slot non sa di averli in quanto sono trattati come normali metodi.

Tutte le classi che ereditano da QObject o da sue sottoclassi possono contenere segnali e slot.

E' possibile connettere più segnali ad un solo slot, un segnale ad un altro segnale o un segnale a più slot. In quest'ultimo caso gli slot vengono eseguiti in ordine casuale.

I segnali non hanno mai variabili di ritorno, sono generati attraverso i file *moc* e non devono essere implementati nei file *cpp*.

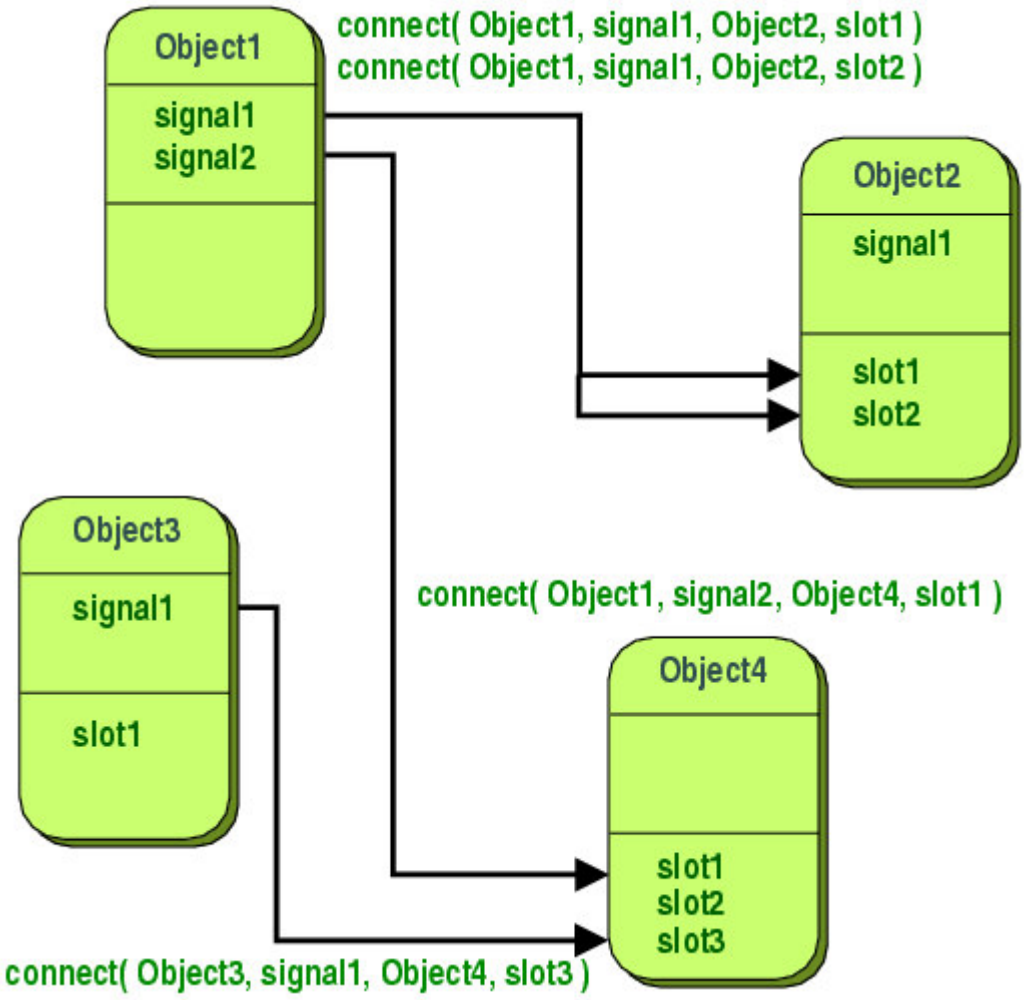


fig. 38 Schema delle connessioni dei segnali agli slot

Questo sistema è efficiente, ma più lento rispetto alle callbacks. In generale emettere un segnale è dieci volte più lento di una chiamata ricevuta direttamente. Questo è l'overhead richiesto per la ricerca dell'oggetto connesso.

3.4 Strumenti di sviluppo: OpenGL

E' difficile parlare di grafica tridimensionale senza nominare OpenGL. OpenGL è il più usato ambiente di sviluppo portabile per applicazioni grafiche interattive in due e tre dimensioni sin da quando è stato introdotto, nel 1992, dalla Silicon Graphics.

Il consorzio indipendente "OpenGL Architecture Review Board" (i cui membri permanenti sono: Digital Equipment, Intel, Microsoft, IBM e Silicon Graphics) controlla le specifiche, ma OpenGL è comunque uno standard open source.

Ogni volta che deve essere aggiunto o modificato qualcosa, vengono avverti in tempo tutti gli sviluppatori affinché possano correggere e apportare le dovute modifiche. Inoltre viene eseguito il controllo di compatibilità all'indietro in modo che nessuna applicazione diventi obsoleta.

fig. 39 Logo di OpenGL

OpenGL permette l'accesso alle innovazioni hardware attraverso l'estensione delle API (Application Programming Interface). In questo modo le innovazioni appaiono rapidamente, lasciando, agli sviluppatori e ai venditori, il tempo di incorporare le nuove funzionalità nel loro prodotto.

Le routine di OpenGL sono efficienti e non occupano molte linee nel codice dell'applicazione, inoltre i drivers OpenGL incapsulano le informazioni sull'hardware sottostante, rendendo il tutto trasparente al programmatore.

L'utilizzo di questa libreria permette l'accesso alle primitive geometriche e di immagine, le display list, trasformazioni, illuminazione e texture, anti-aliasing, blending e molte altre.

Mediante alcune estensioni (glu32 ad esempio) è in grado di gestire anche superfici complesse come le NURBS (Non Uniform B-Spline), le quali vengono comunque scomposte in primitive più semplici (poligoni).

Le operazioni sulle geometrie vengono specificate dall'applicazione a OpenGL mediante tre matrici: la *model_matrix*, la *view_matrix* e la *perspective_matrix*.

Le prime due sono utilizzate nella fase di modelviewing, la terza consente di effettuare la proiezione prospettica delle geometrie. Dopo la fase di clipping, in cui vengono eliminate le primitive che sono al di fuori del volume di vista, viene applicata la trasformazione di viewport, in cui si ottengono le coordinate pixel per la visualizzazione a schermo.

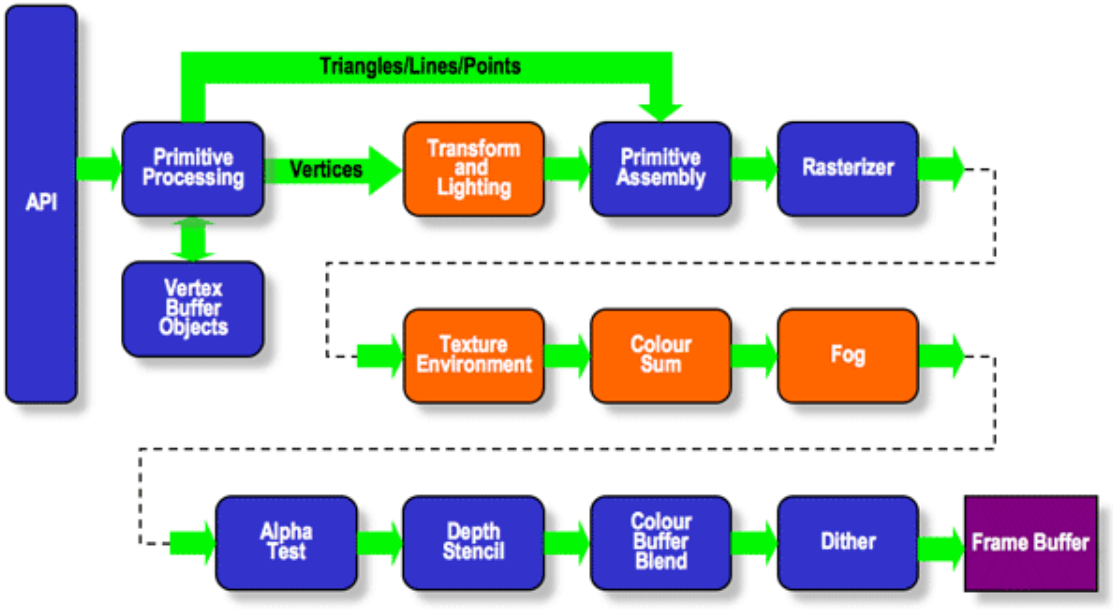


fig. 40 Pipeline di rendering di OpenGL

Sebbene le specifiche di OpenGL definiscano un certo tipo di pipeline di elaborazione, chiunque ha la possibilità di aggiungere particolari implementazioni al fine di raggiungere particolari obiettivi. Ad esempio possono essere eseguite determinate chiamate su un hardware dedicato.

Molte estensioni, come GLU, GLX, WGL sono state definite dai produttori e gruppi di essi e risiedono nel "OpenGL Extension Registry", il quale definisce anche convenzioni dei nomi e le linee guida per la creazione di una nuova estensione.

4 AB-Block: strutture dati

4.1 Introduzione

Le strutture dati ideate per questo progetto possono essere suddivise in tre categorie: strutture di memorizzazione permanente, strutture puramente geometriche, e componenti relativi agli edifici.

fig. 41 Suddivisione concettuale delle strutture dati

La prima categoria è costituite da file: i line of block (lob), di cui abbiamo accennato nel precedente capitolo, e l'aam, quest'ultimo è il formato proprietario usato dalla scuola Sant'Anna per visualizzare scene 3D.
Le strutture dati puramente geometriche sono: vertici (di linee e di facce), linee, facce.
Infine l'ultima categoria è composta dalle classi relative hai componenti degli edifici: Building, Facade, Roof e SideWalk.

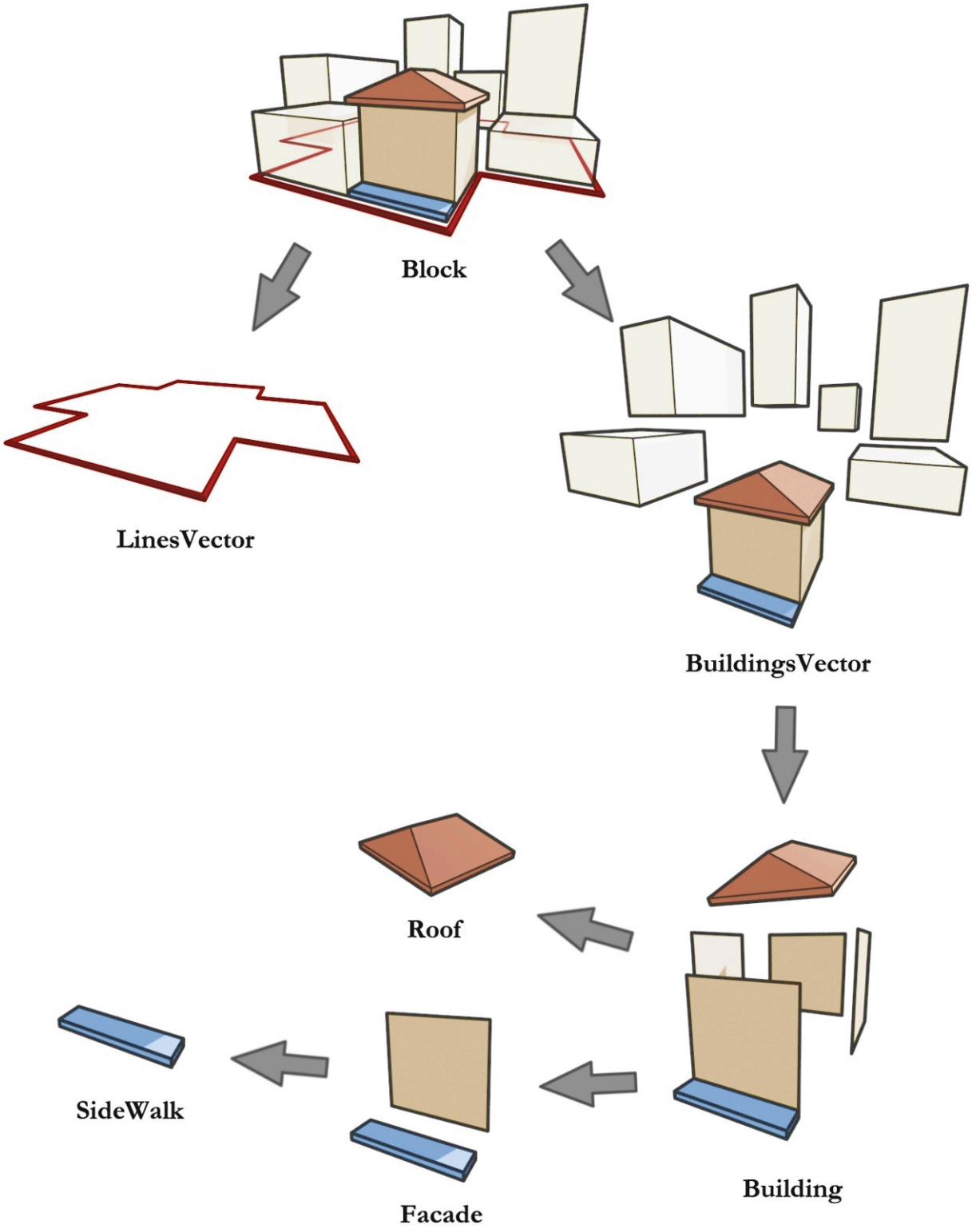


fig. 42 Schema dei componenti dell'isolato

4.2 File .aam

L'AAM, come detto precedentemente, è il formato proprietario del Sant'Anna per ambiente XVR. Esso può essere sia binario che ascii, permette di descrivere la composizione di una scena, in particolare le proprietà riguardanti: geometrie, animazioni, camere e luci. Attualmente le ultime due proprietà vengono esportate in un file separato (.cam) in modo da scindere le proprietà degli oggetti da quelle della scena.
AB-Block esporta in AAM ascii le informazioni riguardanti gli isolati generati e non tiene conto della camera e delle luci. L'isolato può essere salvato in coordinate locali (centrato nell'origine del mondo) o globali. Quest'ultima opzione è utile nel processo di creazione della città, poiché gli isolati salvati in questo modo vengono disposti automaticamente nel loro sito reale.
La versione testuale è suddivisa in due parti: la lista dei materiali e quella degli oggetti.

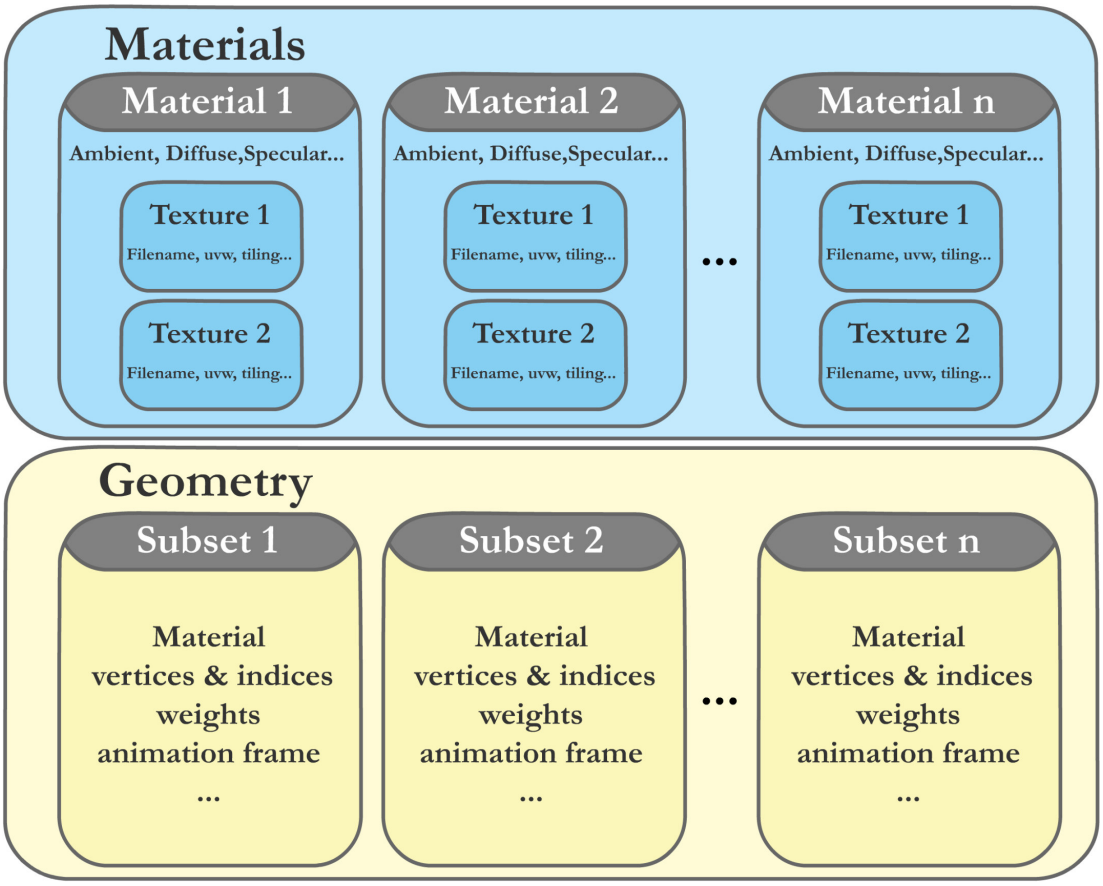


fig. 43 Schema del formato AAM

4.3 File .lob

Il file .lob (line of block) contiene tutte le informazioni di input per la generazione dell'isolato. Il file è generato attraverso un ulteriore programma di analisi dell'immagine, il quale estrapola i dati dalla cartografia della zona da riprodurre (mappa stradale e mappa delle quote del terreno). E' possibile, se necessario, editare a mano il file in quanto le informazioni non sono scritte in binario, ma in un più leggibile formato testo.
Ecco l'esempio di un semplice file .lob:

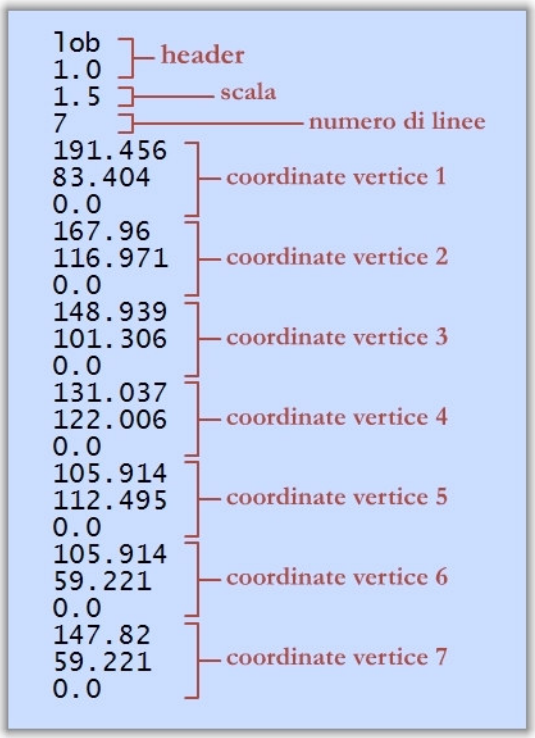


fig. 44 Esempio di file .lob

L'header è costituito dall'identificatore del file e dalla versione. Successivamente è memorizzata la scala in cui sono state salvate le misure (se la scala è 1.5 allora un metro è dato da 1.5 unità di OpenGL). Segue il numero di linee appartenenti al perimetro dell'isolato (in questo caso sette). Le triple sottostanti rappresentano le coordinate tridimensionali dei vertici delle linee. L'isolato è appunto un poligono chiuso, quindi ogni vertice è condiviso tra due linee adiacenti, in particolare il vertice uno è il primo vertice della prima linea, ma è anche il secondo dell'ultima.

4.4 Vertici

Esistono due tipi di vertice a seconda che questo appartenga ad una linea (*Vertex*) o ad una faccia (*Gvertex*). Entrambi contengono un array di 3 float atti alla memorizzazione delle coordinate del vertice nello spazio a tre dimensioni. La struttura Gvertex contiene inoltre un intero *index,* che viene adoperato nel momento del salvataggio delle geometrie.
Durante il salvataggio infatti, è necessario conoscere la posizione del vertice nella lista globale dei vertici, in quanto le facce fanno riferimento ad essa per la loro definizione. In questo modo il salvataggio risulta più semplice e veloce, poiché non bisogna creare un ulteriore struttura dati per associare ogni vertice ad una lista di facce in cui esso è presente.

4.5 Linee

La linea (*Line* nel codice) è la struttura dati fondamentale del AB-Block-Code, è usata in quasi tutti gli algoritmi. Essa è gestita come un segmento di retta, ovvero memorizzata tramite i coefficienti della funzione esplicita della retta e i vertici che la delimitano.

$$y = \mathbf{m}\,x + \mathbf{q}$$

Qui sopra è rappresentata la formula esplicita della retta, ricordiamo che m è il coefficiente angolare e q è l'ordinata all'origine. Quest'espressione rende più agevole la definizione e la manipolazione delle linee, poiché è possibile usare tutte le regole e le trasformazioni della geometria euclidea; purtroppo esistono due casi particolari in cui bisogna prestare attenzione: la linea orizzontale e quella verticale, è bene perciò avere un intero (kind) che indichi il tipo di retta (HORIZZONTAL, VERTICAL, OBLIQUE). Il problema è dato soprattutto dalla linea verticale, la quale non può essere definita tramite un coefficiente angolare, poiché questo è infinito. La linea orizzontale invece, può essere rappresentata ponendo il coefficiente angolare a zero; tuttavia la linea orizzontale è strettamente legata a quella verticale in quanto l'una è la perpendicolare dell'altra. La creazione della

perpendicolare ad una linea orizzontale deve essere trattata perciò come caso particolare, infatti il calcolo per derivare la perpendicolare di una retta include l'operazione $m_p = -1/m$, che nel caso di $m = 0$ dà infinito. Nella struttura sono memorizzati ulteriori dati di utilità generale come il coefficiente angolare della perpendicolare, e la lunghezza del segmento.

```
struct Line
  {
   Vertex point[2];
   int   kind;      /* OBLIQUE, VERTICAL or HORIZZONTAL*/
   float m;
   float q;
   float mPerpendicular;
   float lenght;
  };
```

fig. 45 Codice della struct Line

4.6 Facce

In AB-Block le facce generate sono solitamente dei quadrati, ma capitare che le geometri e richiedano l'utilizzo dei triangoli, per questo motivo la struttura dati *Face* contiene il booleano *isTriangle*, vero se è un triangolo. Esso è utilizzato in fase di disegno e in fase di salvataggio. A seguire un array di quattro puntatori a *Gvertex*, il cui ultimo è preso in considerazione solo se la faccia è un quadrato.

Altri due array sono contenuti nella struct. Il primo (*texCoord*) costituito da otto float memorizza le coordinate *u* e *v* dei vertici sulla texture (il primo vertice ha la coordinate u in posizione zero e la v in posizione uno, e così via).

Il secondo contiene i valori della normale alla faccia, la quale viene utilizzata da OpenGL, ad esempio, per il calcolo della luminosità.

```
struct Face
{
    bool isTriangle;
    Gvertex* point[4];
    float texCoord[8];
    float normal[3];
};
```

fig. 46 Codice della struct Face

4.7 Building

La classe building è la struttura dati più complessa dell'AB-Block-Code, poiché questa deve contenere tutte le informazioni riguardanti gli attributi estetici dell'edificio e qualche altro dato ausiliare.

```
class Building
{

public:
    int name;
    bool isSelected;
    bool isExtra;

    int nFloor;
    int floorType;
    int groundFloorType;
    bool hasRoof;

    Gvertex base[4];
    Gvertex ceiling[4];
    Gvertex roofBase[4];
    Facade  facade[4];
    Roof    roof;
    Line    line[4];
    Line    roofLine[4];
    GLuint  texture;
```

fig. 47 Variabili istanza della classe Building

Gli attributi del building sono semplici: la presenza del tetto *(hasRoof)*, numero dei piani *(nFloor)*, se è un edificio extra (*isExtra*), tipo di altezza (LOW, MEDIUM, HIGH) del piano terra *(groundFloorType)* e tipo di altezza di tutti gli altri *(floorType)*.
Abbiamo detto che l'edificio è un quadrilatero, perciò sono utili array di quattro valori per la memorizzazione dei vertici della base *(base)* e quelli del soffitto *(ceiling)*, e per mantenere le informazioni sulle facciate *(facade)*. Inoltre la classe contiene le linee del perimetro della base *(line)* e del soffitto *(roofLine)*, molto utili in fase di costruzione del tetto e dei marciapiedi. La presenza di uno o più portici in un edificio comporta l'ausilio di un ulteriore array che contenga nuovi vertici del soffitto *(roofBase)*, questo è utilizzato per la creazione del tetto (*roofLine* considera i vertici di *roofBase*, non quelli di *ceiling*), mentre *ceiling* viene usato nella generazione delle facciate.
La classe contiene in seguito un tetto *roof*, e un intero *texture* che rappresenta la posizione della texture nella lista globale di OpenGL e di AB-Block.
Infine, due variabili utili per il picking in AB-Block sono un intero *name* che identifica la posizione dell'edificio all'interno del vettore degli edifici dell'isolato *(buildingsV)* e un booleano *isSelected* per capire se l'edificio deve essere evidenziato.
Qui sotto l'elenco dei metodi della classe.

```
public:
    Building();
    ~Building();
    void  build(bool firstTime);
    void  buildRoof();
    void  buildSideWalk(int nFacade);
    void  buildPortico(int i, float ground, float firstFloorHeight);
    void  drawBuilding();
    void  getNormal(Face* face);
    float getGround();
    float getLowestPoint();
    void  pointOutFacades(bool on);
    void  destroySideWalk(int nFacade);
    float perimeter();
    float maxHeight();
    float height();

};
```

fig. 48 Metodi della classe Building

4.8 Facade

La facade contiene le informazioni relative alla facciata dell'edificio. Le operazioni consentite su una facciata sono la selezione (*isSelected* come nel Building), l'eliminazione *(isDestroyed)* e la creazione di un marciapiede *(hasSideWalk)*. L'eliminazione delle facciate può essere utile per diminuire la pesantezza della geometria nei casi in cui queste risultino nascoste perché coincidenti. Questa operazione è reversibile poiché la facciata continua a essere memorizzata, anche se non viene visualizzata a schermo. Durante il salvataggio le facciate che avranno *isDestroyed* vera non verranno considerate.
La generazione di un marciapiede può avvenire solo se la facciata è esterna, ovvero se è di fronte alla strada, questa proprietà viene settata al momento della creazione degli edifici e non può essere cambiata dall'utente. Il tipo di marciapiede (LARGE,MEDIUM,TIGHT) è settato tramite *sideWalkType*, mentre *hasPortico* indica la presenza di un portico. La struttura mantiene un puntatore *(sidewalk)* al marciapiede generato.
Infine, un vettore per la memorizzazione dei puntatori ai vertici e uno per quelli delle facce.

```
class Facade
{

public:
    bool isSelected;
    bool isDestroyed;
    bool isExtern;
    bool hasSideWalk;
    bool hasPortico;
    SideWalk* sidewalk;
    int sideWalkWidthType;
    vector <Gvertex*> vertexesV;
    vector<Face*> facesV;

public:
    Facade();
    ~Facade();
    void drawFacade(GLuint texture);
    void destroyFacade();
    void destroySideWalk();

};
```

fig. 49 Variabili e metodi della classe Facade

4.9 Roof

Gli attributi del tetto sono l'altezza *(roofHeight)*, lo stile *(roofStyle)*, orientamento *(orientation)* e l'aggetto (la sporgenza del tetto rispetto alle facciate) nei due assi dell'edificio(roofSlope1, roofSlope2). E' possibile scegliere fra 8 stili diversi (0..7). L'orientamento è utile per quei tipi di tetto che non sono simmetrici, come ad esempio, quello a due falde. La presenza dell'aggetto rende la geometria molto più raffinata, ma ne aumenta il numero di poligoni. Essendo un componente dell'edificio, anche il tetto contiene i due vettori di puntatori ai vertici e alle facce.

```
class Roof
{

public:
    int roofStyle;
    int orientation;
    float roofHeight;
    float roofSlope1;
    float roofSlope2;
    vector <Gvertex*> vertexesV;
    vector <Face*> facesV;

public:
    Roof();
    ~Roof();
    void drawRoof();
    void destroyRoof();

};
```

fig. 50 Variabili e metodi della classe Roof

4.10 SideWalk

Il SideWalk è la struttura più semplice tra i componenti dell'edificio, e, come gli altri, contiene i due vettori di puntatori ai vertici e alle facce. In più una struttura Line *(line)* utile per la generazione delle colonne del portico e per la sua cancellazione.

```
class SideWalk
{

public:
    Line line;
    vector <Gvertex*> vertexesV;
    vector <Face*> facesV;

public:
    SideWalk();
    ~SideWalk();
    void drawSideWalk();
    void destroySideWalk();

};
```

fig. 51 Variabili e metodi della classe SideWalk

5 AB-Block: algoritmi

5.1 Introduzione

Così come le strutture dati, anche gli algoritmi possono essere suddivisi in tre classi. I primi, puramente geometrici, sono alla base del funzionamento di tutti gli altri, ad esempio, la creazione della parallela o della perpendicolare ad una linea; vi sono poi quelli che dispongono i lots nell'isolato; ed infine quelli atti alla costruzione dei componenti dell'edificio.

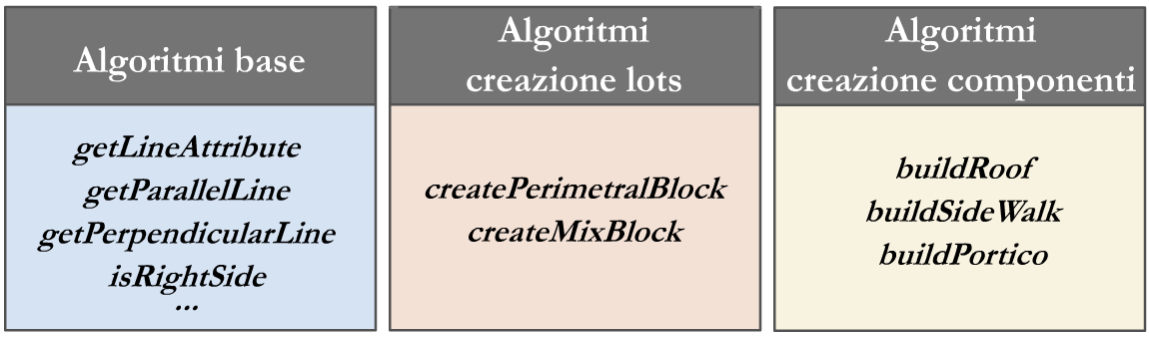


fig. 52 Suddivisione algoritmi di AB-Block

5.2 Algoritmi base

La costruzione degli edifici in AB-Block segue un procedimento basato sulle linee paragonabile a quello utilizzato nel disegno tecnico. Ogni forma è creata a partire da una o più linee di riferimento, d'altronde, lo stesso file LOB da cui vengono estratti i dati sull'isolato contiene, solo ed esclusivamente, le linee del perimetro. Effettivamente queste informazioni non bastano affinché AB-Block possa comprendere le fattezze di un isolato nella sua interezza, questa operazione infatti, risulterebbe molto complessa in termini di strutture ausiliarie e tempo. AB-Block ha perciò una visione locale alla linea. Ciò nonostante esiste un grosso problema: capire quali punti dello spazio sono all'interno o all'esterno della linea. E' necessario introdurre perciò una convenzione, utilizzata anche per il calcolo della normale ad una faccia, cioè la memorizzazione delle linee in ordine antiorario.
In questo modo ogni linea è orientata, l'esterno è dato dal semipiano destro della retta coincidente alla linea. In AB-Block esiste la funzione *isRightSide* che calcola se un punto è nel semipiano destro o sinistro di una linea, restituendo vero o falso a seconda dell'esito.

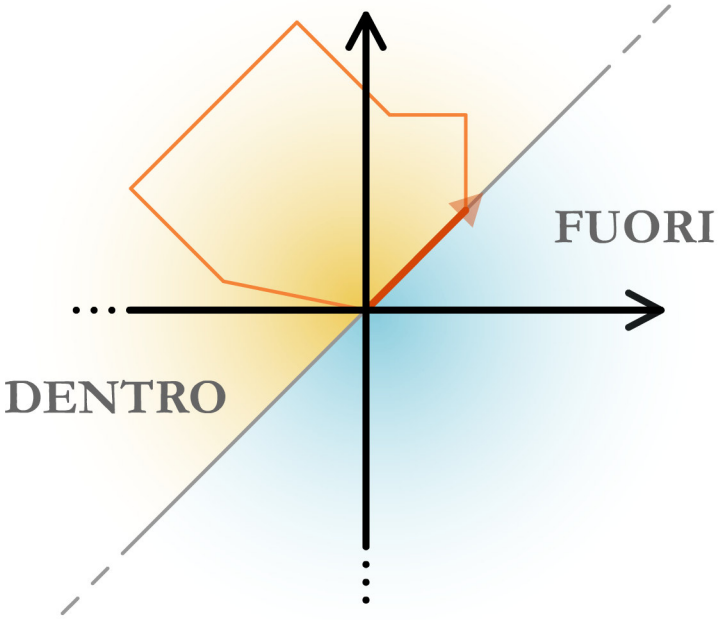


fig. 53 Calcolo del semipiano destro e sinistro

Esistono molte altre funzioni base, un' altra per esempio, essenziale per tutte le costruzioni di AB-Block, è quella che permette la generazione della parallela di una data linea ad una certa distanza, questa può essere creata sia a sinistra che a destra. Il metodo è fondamentalmente simile al procedimento con l'utilizzo delle righe a squadra del disegno tecnico. Innanzitutto si calcola la perpendicolare alla linea, ovvero si ricava il suo coefficiente angolare (il quale è già stato calcolato e memorizzato nella struttura dati Line) e la sua ordinata all'origine. La perpendicolare viene posizionata sul secondo punto della linea (il primo vertice della perpendicolare è il secondo vertice della linea). Sappiamo che l'arcotangente del coefficiente angolare è l'angolo che la linea forma con l'asse x positivo. Calcoliamo il coseno dell'angolo per trovare il rapporto tra la distanza che vogliamo ottenere e quella effettiva sull'asse x.

$$\Delta x = \mathrm{abs}(\cos(\mathrm{arctg}(m_{\mathrm{perpendicolare}}))) * \mathrm{distanza}$$

A questo punto troviamo il Δx, ma non sappiamo ancora se questo deve essere aggiunto o sottratto alla coordinata x del secondo vertice della linea. Il secondo punto della perpendicolare viene comunque calcolato sommando Δx, controllando successivamente, tramite la funzione *isRightSide*, se questo è nel lato desiderato, altrimenti viene calcolato nuovamente, questa volta sottraendo il Δx.

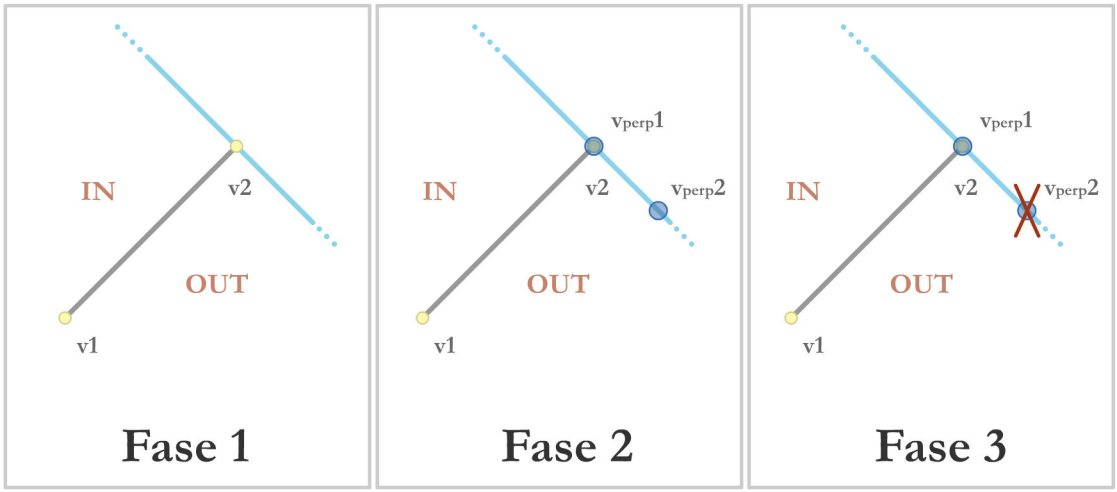


fig. 54 Prime fasi della creazione della parallela
(nell'esempio la si vuole all'interno, perciò il primo vertice calcolato è errato)

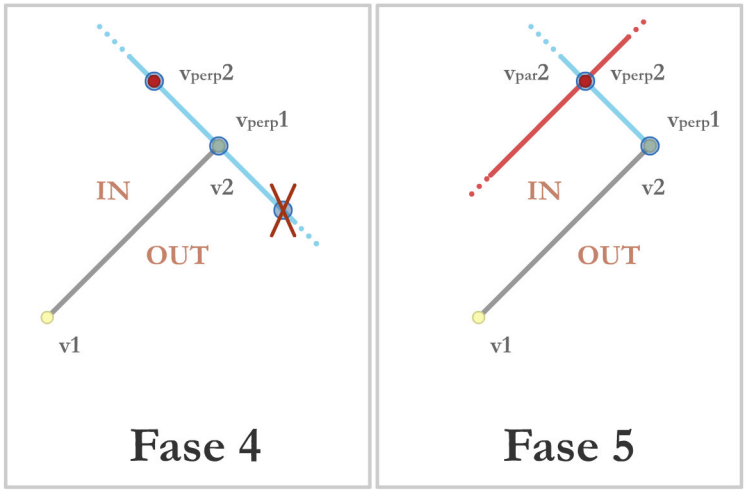


fig. 55 Ultime fasi della creazione della parallela

Trovato il punto giusto, sappiamo che questo è il secondo vertice della parallela da creare, sappiamo inoltre che la parallela ha lo stesso coefficiente angolare della linea iniziale perciò l'unica cosa che rimane da calcolare è la sua ordinata all'origine (le coordinate del secondo vertice sono calcolate nello stesso modo del primo).

5.3 Algoritmo Perimetral

Perimetral è uno dei due algoritmi atti alla generazione dei lots dell'isolato, ed è scelto dall'utente tramite l'interfaccia. L'isolato generato con questo metodo assume la connotazione tipica di una corte, ovvero di un cortile delimitato da edifici dello stesso spessore. Quest'ultimo è calcolato come la lunghezza del lato più piccolo del perimetro fratto 3,1 (questo particolare valore permette eliminazione delle possibili sovrapposizioni degli edifici).

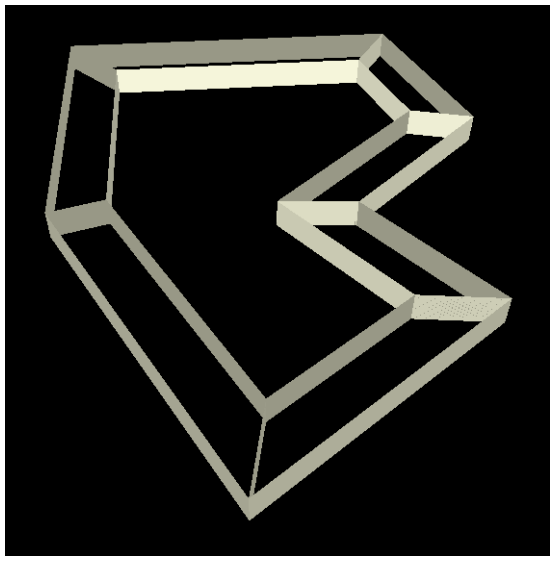


fig. 56 Generazione dei lots tramite algoritmo Perimetral senza suddivisione delle linee

In entrambi gli algoritmi viene creato un edificio per ogni linea del perimetro. L'utente può comunque dividere queste linee impostando la loro lunghezza media. In questo modo, prima che abbia inizio la creazione dei lotti, viene richiamato un ulteriore algoritmo ricorsivo di bisezione delle linee.

La lunghezza massima accettata è calcolata come 5/4 la lunghezza media inserita dall'utente. Ogni volta che una linea è maggiore della lunghezza massima viene divisa a metà. Per evitare la creazione di linee della stessa lunghezza viene deciso in maniera random se aggiungere o sottrarre una quantità pari a 1/12 della lunghezza massima.

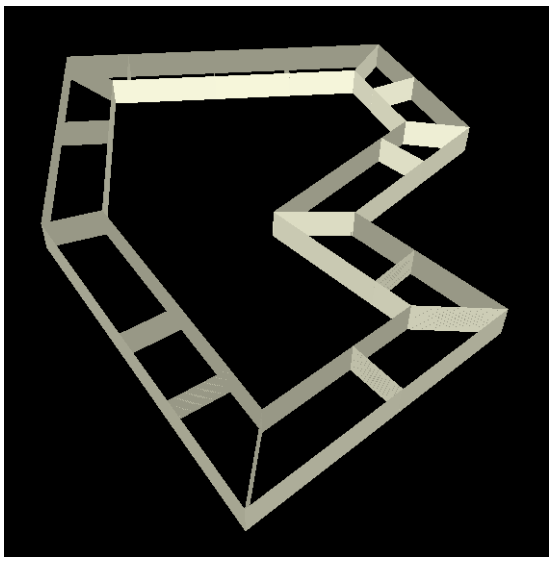


fig. 57 Generazione dei lots tramite algoritmo Perimetral con suddivisione delle linee

Una volta che le linee sono divise può iniziare la creazione dei lots. L'algoritmo, semplice e veloce, procede nella maniera seguente: inizialmente vengono create, all'interno dell'isolato, tutte le parallele, alle linee del perimetro impostando la distanza uguale allo spessore, i due vertici della linea del perimetro sono i primi due vertici della base dell' edificio, successivamente per ogni parallela viene calcolato il punto di intersezione con la successiva, il quale sarà il terzo vertice della base. Si esegue infine un ulteriore ciclo, assegnando alle coordinate del quarto vertice di ogni edificio quelle del terzo vertice del precedente.

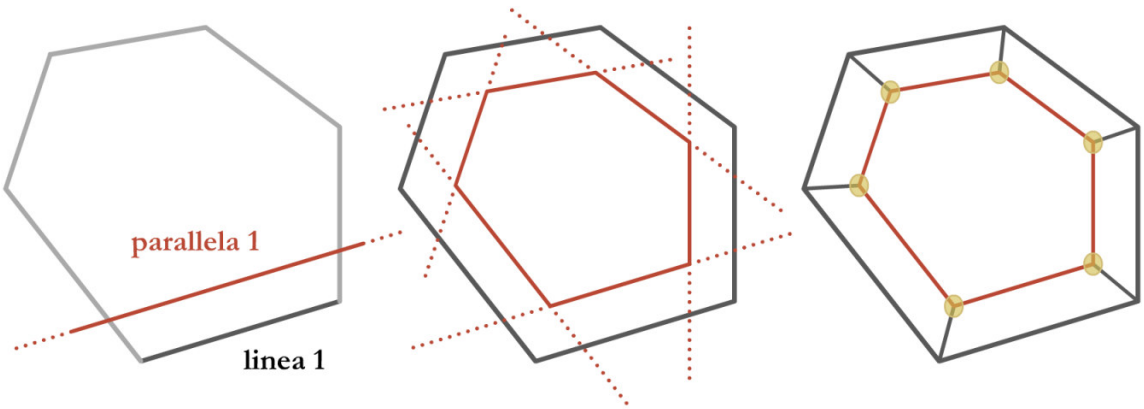


fig. 58 Procedimento dell'algoritmo Perimetral

5.4 Algoritmo Mix

L'algoritmo Mix è più intelligente del precedente poiché è in grado di generare isolati molto più complessi e variegati in base alla morfologia dell'isolato. Anch'esso genera un edificio per ogni linea del perimetro, la quale può essere, anche in questo caso, suddivisa precedentemente tramite l'algoritmo ricorsivo descritto nel paragrafo precedente.

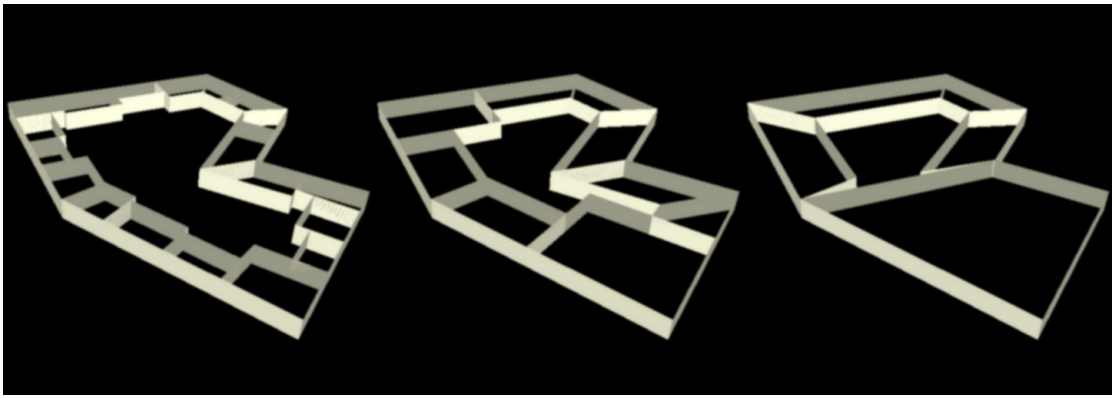


fig. 59 Generazione dei lots tramite algoritmo Mix con diverse lunghezze medie

La disposizione dei lots è calcolata in base agli angoli fra le linee, esiste perciò un vettore di interi che contiene, per ogni linea, l'identificatore dell'angolo formato con la successiva (COINCIDENT angolo piatto, PERPENDICULAR angolo compreso tra 70 e 110 gradi, ACUTE angolo inferiore a 70, OBTUSE angolo superiore a 110 e inferiore a 180 e RIGHT angolo superiore a 180 gradi).

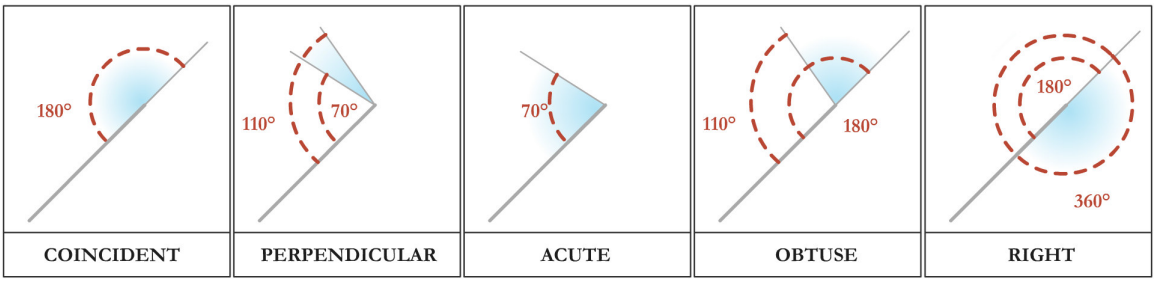


fig. 60 Angoli

Per ogni conformazione degli angoli esiste un'euristica, ovvero una soluzione data. Questo metodo è tanto più raffinato quanto si riesce a conoscere la disposizione delle linee vicine a quella corrente, sia quelle precedenti che quelle successive (si potrebbe arrivare a controllare sino a tre linee di distanza in entrambe le direzioni). Purtroppo l'algoritmo diventa molto complesso poiché i casi possibili e i confronti sono veramente tanti.

Per ovviare a questo problema l'algoritmo si compone in passi successivi, una soluzione molto più efficiente in grado di eliminare alcuni controlli.

Come abbiamo detto AB-Block ha sempre una visione locale alla linea perciò gli angoli considerati ogni volta sono solamente il precedente e il successivo (tranne in un caso). La potenza dell'algoritmo è data scelta della sequenza in cui vengono creati i lots, a seconda della conformazione locale delle linee.

Ad ogni ciclo si eliminano le linee già utilizzate. Nel primo si generano i lots per quelle linee i quali angoli, precedente e successivo, sono entrambi COINCIDENT o entrambi PERPENDICULAR. Nel secondo ciclo le conformazioni sono, partendo dal precedente, COINCIDENT PERPENDICULAR COINCIDENT, e l'altra con il successivo uguale a OBTUSE o ACUTE o RIGHT. Nel terzo, tutte le rimanenti, ovvero quelle con angolo successivo COINCIDENT o PERPENDICULAR.

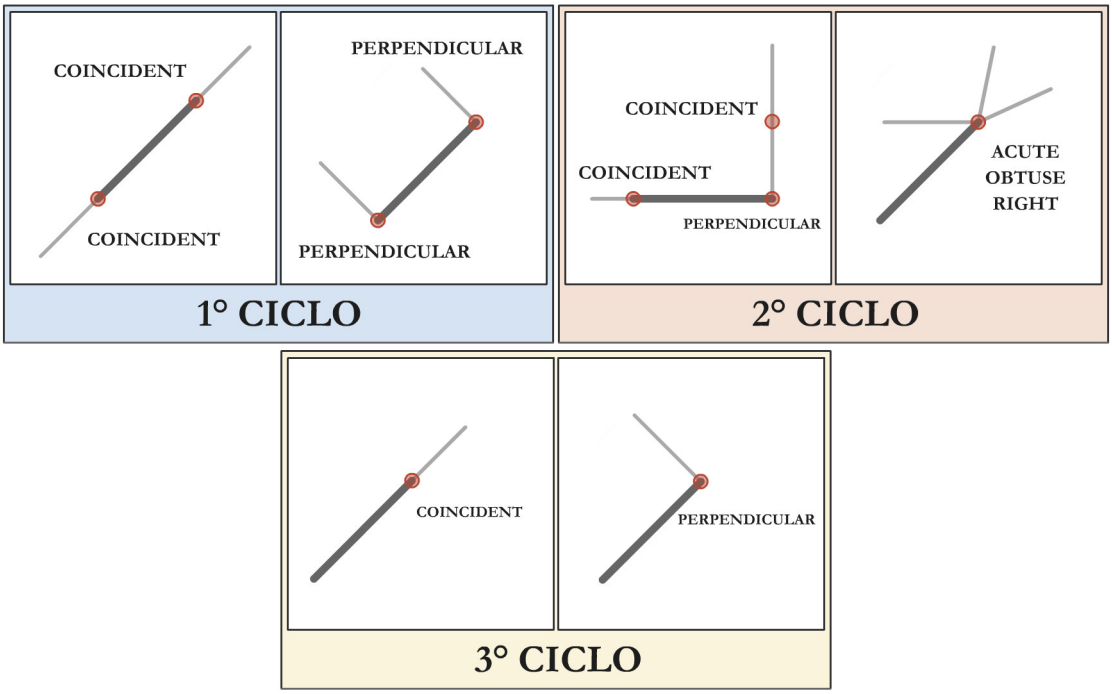


fig. 61 Cicli dell'algoritmo Mix

Esistono quattro tipi di chiusura del lots a seconda dei casi, come mostrato in figura.

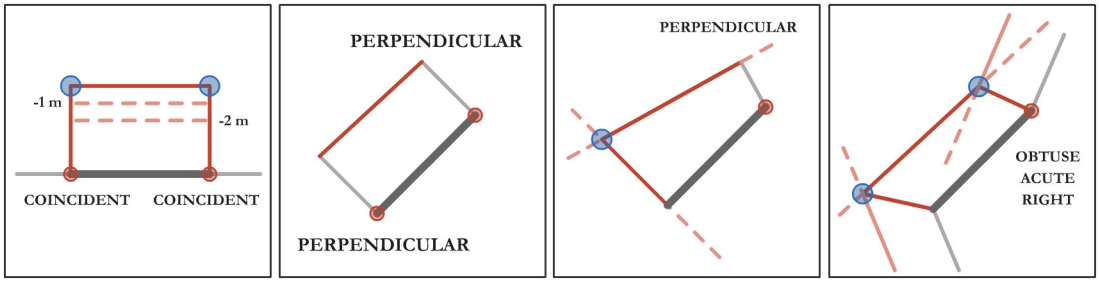


fig. 62 Chiusure dei lots

Nel primo viene creata una parallela con distanza uguale allo spessore (lunghezza della linea minima fratto 2), alla quale viene sottratta, in maniera random, una quantità pari a zero, un metro o due metri, così da variegare le dimensioni degli edifici; i due vertici di chiusura (in blu) sono quelli della linea proiettati sulla parallela. Il secondo caso è il più semplice in quanto vengono collegate le due estremità della spezzata. Nel terzo vengono create le perpendicolari alle linee, il vertice di chiusura è dato dalla loro intersezione. Infine, in tutti gli altri casi si calcolano le parallele della spezzata, i vertici, anche in questo caso, sono dati dalle loro intersezioni (le parallele hanno distanza fissa lunghezza minima fratto 2).

5.5 Algoritmi di generazione dei componenti

Sinora abbiamo generato solamente le piante degli edifici. Per la loro completa costruzione esistono gli algoritmi: buildRoof, buildSidewalk e buildPortico, grazie ai quali vengono ricostruiti i template dei tetti e dei marciapiedi, con o senza portico.

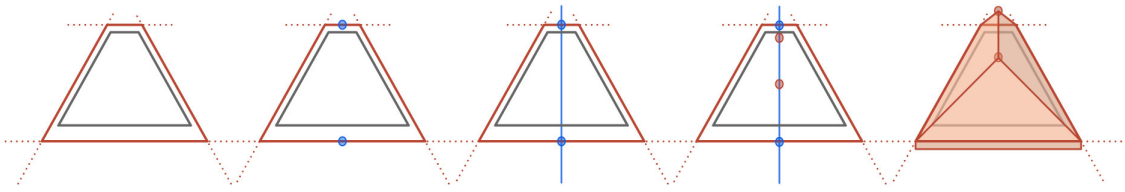


fig. 63 Esempio di costruzione di un tetto

Come sempre il procedimento è basato sulla generazione e intersezione di parallele e perpendicolari. Nell'esempio il tetto sporge dai muri perciò il primo passo è quello di creare le parallele a distanza uguale allo spessore dell'aggetto. Poiché il tetto è del tipo "a quattro falde" si trovano i punti medi delle linee (in blu). Si crea successivamente la retta passante per i punti medi, e si trovano i due vertici interni, i quali vengono traslati verso l'alto secondo l'altezza del tetto. Qui sotto alcuni esempi in cui sono segnate le posizioni dei vertici nei vettori del tetto delle facciata con portico (colonna separata) e del marciapiede.

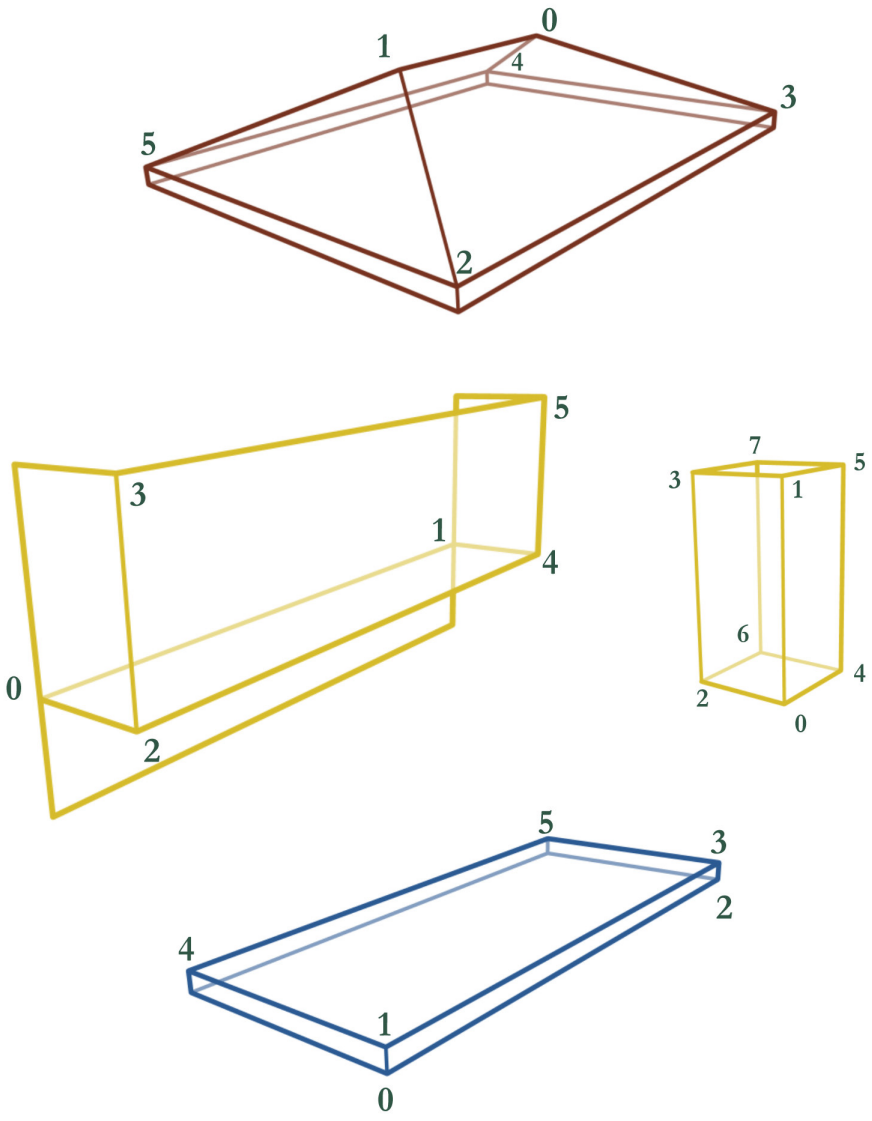


fig. 64 Ordine di creazione dei vertici nelle costruzioni

6 AB-Block: interfaccia

6.1 Introduzione

L'AB-Block-Code è integrato in un'interfaccia realizzata tramite le API di QT. L'editor offre all'utente un facile inserimento, sia dei parametri dell'isolato, che delle proprietà degli edifici. L'utente ha inoltre la possibilità di scegliere il livello di automatizzazione della generazione. L'interfaccia è studiata in modo da essere intuitiva e usabile, tutte le funzionalità sono reversibili in caso di errore.

I componenti più importanti dell'interfaccia sono l'openGLViewer (visualizzatore dell'ambiente OpenGL) e il textureViewer (visualizzatore e generatore delle texture procedurali). Il primo componente eredita dalla classe QGLWidget, mentre il secondo dalla QScrollViewer, poiché questo deve essere in grado di gestire lo scrolling nel caso che la texture generata sia più grande della finestra di visualizzazione.

La cattura dei segnali è gestita interamente dalle finestra principale e da quella secondaria dell'editor delle texture. Queste si occupano di prelevare i dati inseriti dall'utente, di copiarli nelle proprietà degli edifici per poi richiamarne la rigenerazione, o di inserirli come parametri di input per gli algoritmi dell'AB-Block Code.

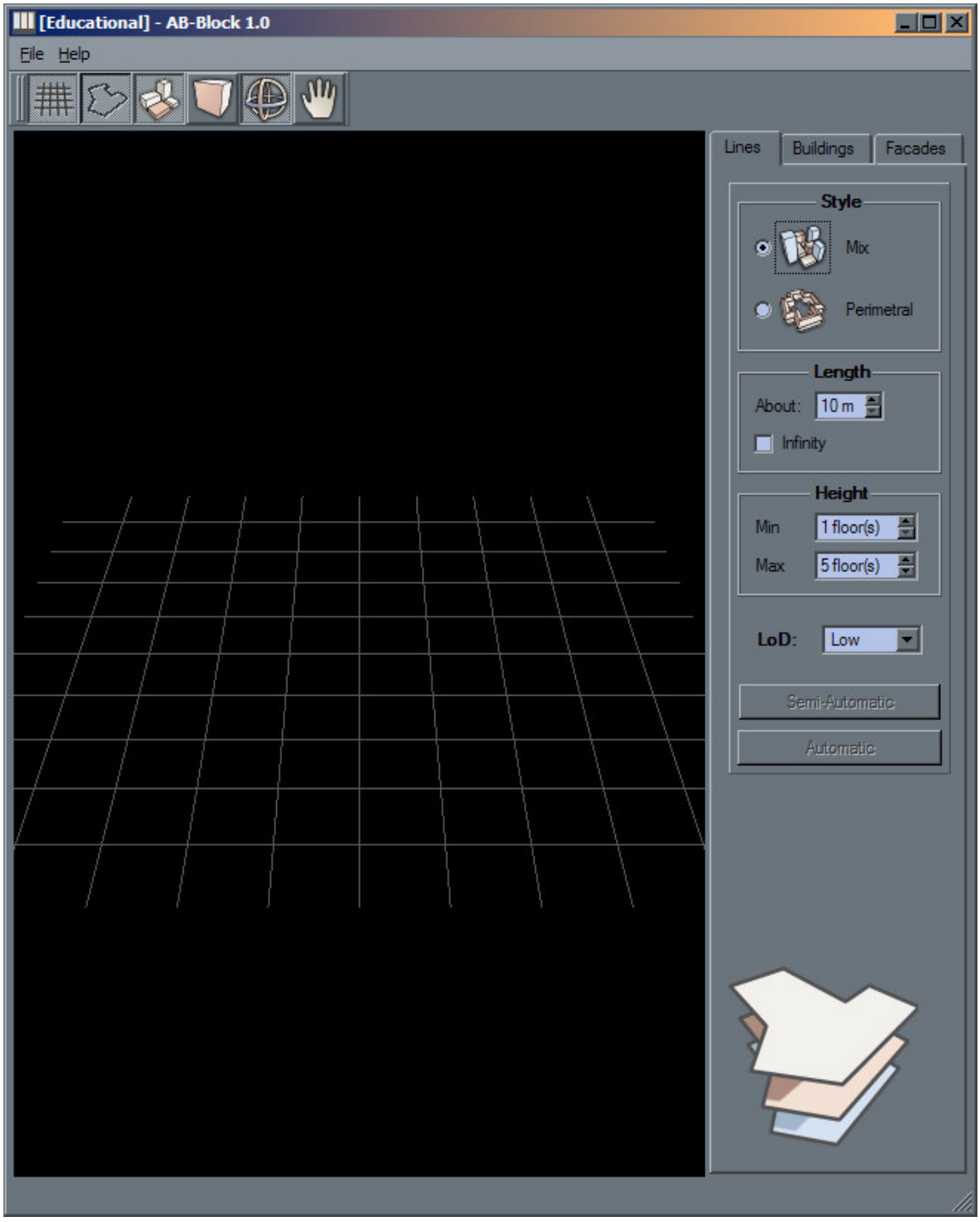


fig. 65 Schermata iniziale di AB-Block

6.2 Menù e Toolbar

Dal menù è possibile aprire i file LOB ed esportare il lavoro fatto con l'editor nel formato AAM. Ogni isolato aperto viene automaticamente ridimensionato secondo la scala descritta nel file e centrato nella openGLViewer (le coordinate rimangono quelle originali) per una più semplice visualizzazione. Nel caso venga aperto un file non ben formattato il programma avverte con una finestra in quale riga del file c'è stato un errore (questo può accadere se il file è stato editato a mano). Anche per l'esportazione l'utente può scegliere se salvare l'isolato nelle coordinate globali o in quelle globali.

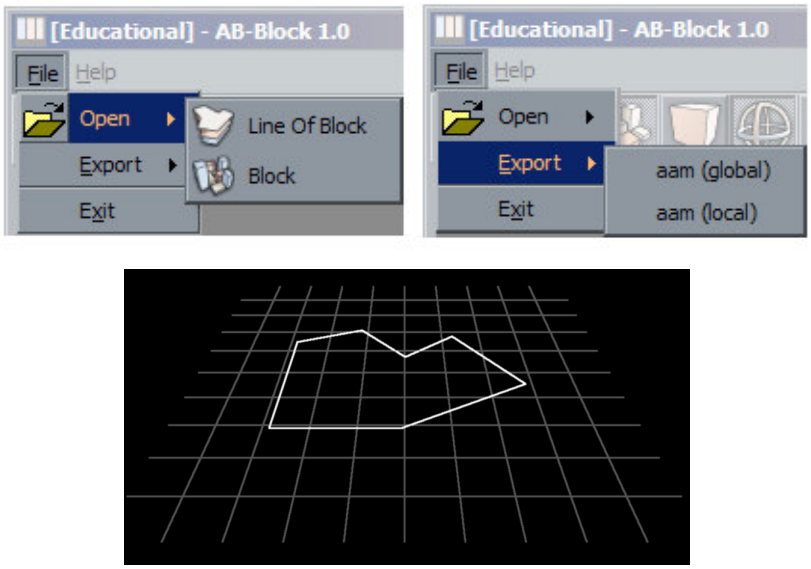


fig. 66 Menù di AB-Block e visualizzazione di un file LOB

Gli oggetti che si possono gestire in AB-Block sono: il perimetro dell'isolato, gli edifici generati e le loro facciate. Le prime due icone della toolbar sono i pulsanti di Mostra/Nascondi griglia e Mostra/Nascondi linee del perimetro. I due pulsanti successivi sono utilizzati in mutua esclusione e servono per la selezione degli edifici o delle loro facciate, questa è effettuata per mezzo del pulsante sinistro del mouse. Non esistono operazioni manuali sulle singole linee del perimetro perciò non esiste nessun pulsante di selezione per queste. Le ultime due icone sono relative alle funzioni del tasto destro del mouse, ovvero la

fig. 67 Toolbar di AB-Block

trackball (per la rotazione della scena) e il panning (per la traslazione). Lo zoom viene effettuato tramite la rotella del mouse.

6.3 Sezioni

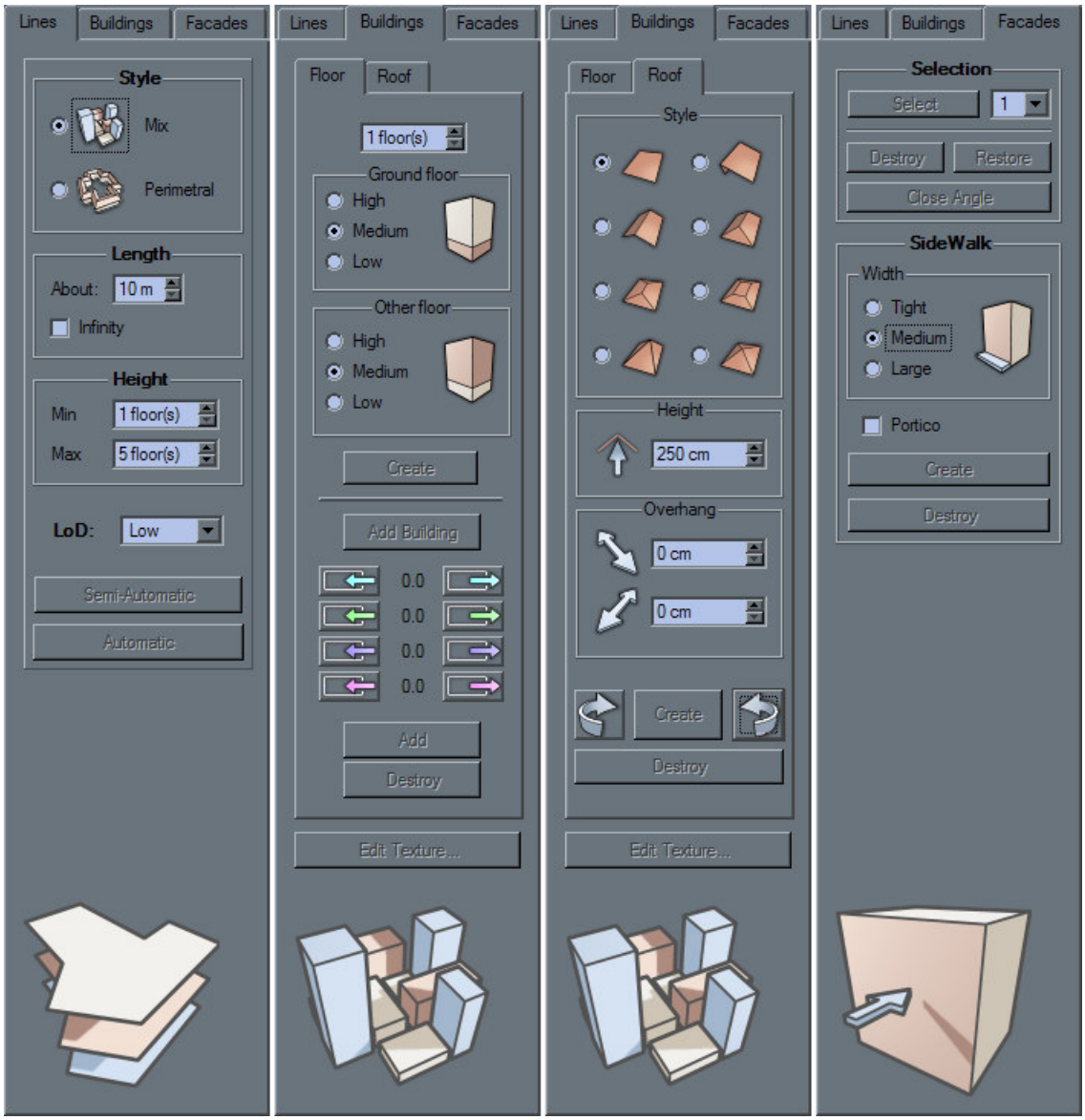


fig. 68 Sezioni di AB-Block

In AB-Block esistono quattro sezioni: una per le proprietà dell'isolato, due per quelle degli edifici (una per i piani e l'altra per i tetti) e l'ultima per le facciate e i relativi marciapiedi.

Nella sezione delle *Lines* l'utente può scegliere l'algoritmo da usare per la creazione del lots, la loro lunghezza media (mettendo un check su *infinity* si richiede di non effettuare il controllo), l'altezza minima e massima degli edifici e il loro livello di dettaglio (*Lod*). Le ultime due proprietà devono essere indicate esclusivamente nel caso l'utente voglia utilizzare la generazione completamente automatica, altrimenti non vengono considerate.
I comandi della sezione Building sono accessibili dopo aver effettuato la selezione di un edificio. Nella sottosezione *Floor* è possibile scegliere il numero di piani dell'edificio, l'altezza del piano terra e quella degli altri piani. Più in basso vediamo lo strumento *Add Building*. Con esso è possibile creare una varietà molto più ampia e complessa di edifici, aumentando la sfera d'azione di AB-Block. Abbassando il pulsante *Add Building* compaiono quattro linee di colore diverso sui bordi superiori dell'edificio. Sotto il pulsante vediamo due serie di frecce, quelle sulla sinistra servono per spostare la linee verso l'interno dell'edificio mentre quelle a destra verso l'esterno. Le frecce sono facilmente associabili alle corrispondenti linee poiché queste sono dello stesso colore. Cliccando successivamente il pulsante *Add* viene creato un edificio i cui vertici della base sono i punti di intersezione delle linee. Durante la fase di spostamento delle linee viene effettuato un controllo affinché la distanza tra quelle opposte non sia minore di quaranta centimetri, in modo tale da evitare, da parte dell'utente, la creazione di edifici con pianta nulla o opposta.

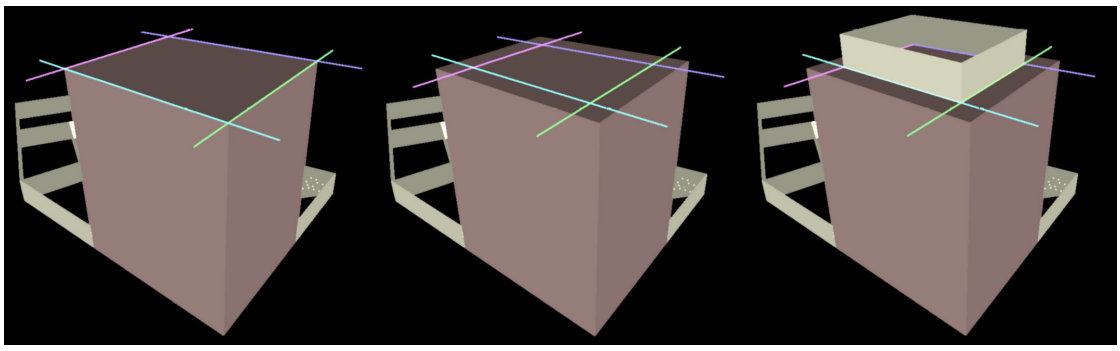


fig. 69 Fasi della Add Building

Ogni edificio creato con questo metodo è marcato come extra e può essere distrutto (in questo caso la distruzione è definitiva, comunque lo si può sempre ricreare).
La funzione *Add Building* è applicabile anche agli edifici extra, è così possibile creare edifici "a matriosca" e molti altri tipi ancora.

Nella sottosezione *Roof* vi sono i parametri per la creazione dei tetti. Di ogni tetto è possibile cambiare lo stile, l'altezza e la sporgenza dalle mura (aggetto) nei due assi, inoltre il tetto può essere ruotato sui lati dell'edificio.
Nell'ultima sottosezione, quella delle *Facades*, troviamo uno strumento utile per la selezione delle facciate, nel caso che questa non sia applicabile tramite il picking con il mouse. Selezionando una facciata accessibile dell'edificio si può scegliere la facciata desiderata cliccando, nella combo box, il numero della facciata da selezionare (ricordiamo che queste sono sempre numerate in senso antiorario). Una volta selezionate possono essere cancellate o ripristinate. Questa funzionalità è importante nel caso si volessero eliminare facciate comunque nascoste. Le pareti esterne, ovvero quelle di fronte alla strada, non possono essere cancellate poiché deve essere preservato il perimetro dell'isolato.
Durante l'editing dell'isolato, può capitare che l'utente abbia la necessità di "chiudere un angolo", cioè di voler passare dalla configurazione 1 (sinistra) a quella 2 (destra).

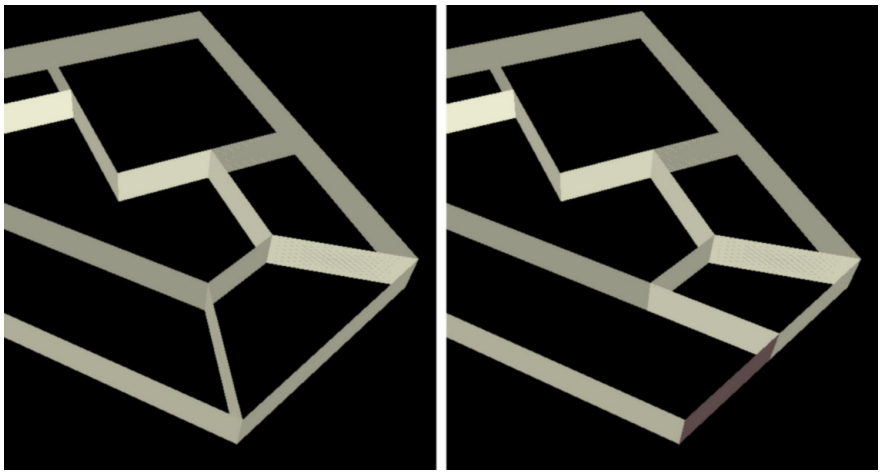


fig. 70 Chiusura dell'angolo

In questo caso l'utente deve selezionare la facciata che desidera portare a ridosso della strada e cliccare sul tasto *Close Angle*. L'algoritmo non funziona nel caso che le facciate adiacenti dei due edifici non siano perfettamente coincidenti.
Ultima operazione disponibile per le facciate è la generazione del marciapiede, il quale può essere stretto, medio, largo, con o senza portico.

6.4 Editor delle texture

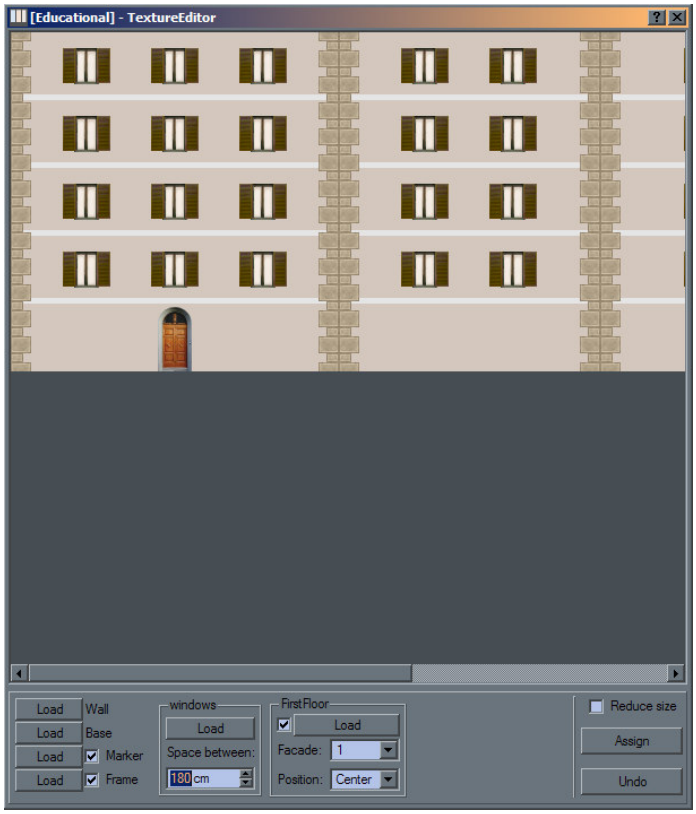


fig. 71 Editor delle texture

Dalla sottosezione *Buildings* è possibile accedere all'editor procedurale delle texture. Ad ogni edificio viene assegnata al massimo una texture. Le informazioni da inserire per la generazione sono: i file del muro, della base, del marcapiano, della cornice, delle finestre e del piano terra. Per base si intende la cornice che solitamente esiste alla base degli edifici che non sono in piano.

I marcapiano, le cornici e la texture del piano terra possono essere rimosse tramite gli appositi checks. Nell'ultimo caso l'algoritmo genera le finestre anche nel piano terra, questa opzione è importante per la texturizzazione degli edifici extra.

E' possibile inoltre definire la facciata in cui disegnare la texture del piano terra (solitamente un portone), la sua posizione in essa (sinistra, centro, destra), e infine la distanza tra le finestre.

Nel caso si desiderino creare texture con un livello di dettaglio più basso è possibile dimezzare le dimensioni dell'immagine tramite il check *Reduce size* (l'operazione è reversibile).

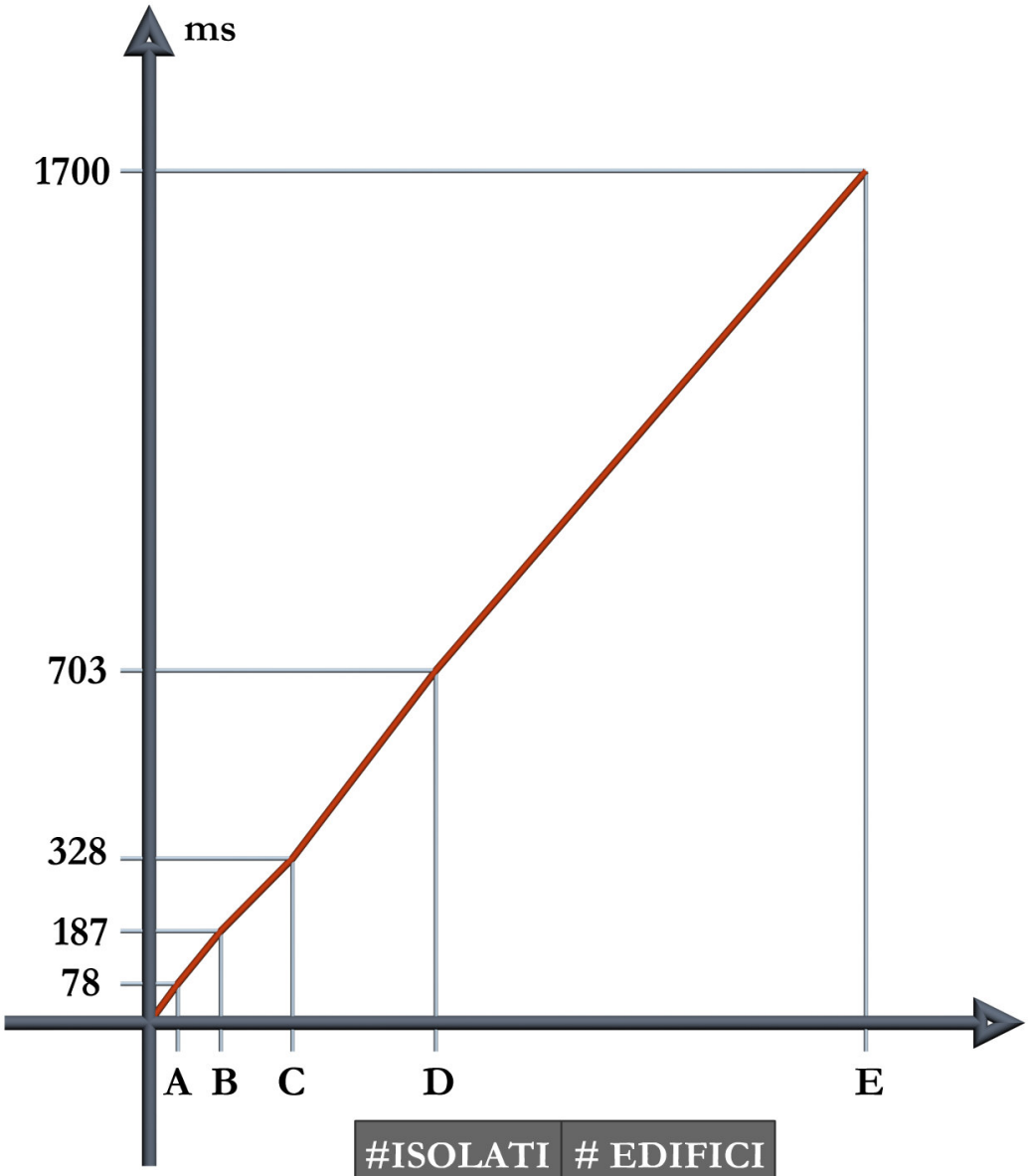
Premendo il pulsante *Assign* si assegna la texture generata all'edificio, inoltre, questa viene salvata su disco nella cartella *GenTexture* dell'applicazione, il nome del file è dato dalla concatenazione della data in cui sono stati generati i buildings, più l'identificativo dell'edificio *(int)*. Qui sotto un esempio:

Feb051034412006-5.png

7 AB-Block: risultati

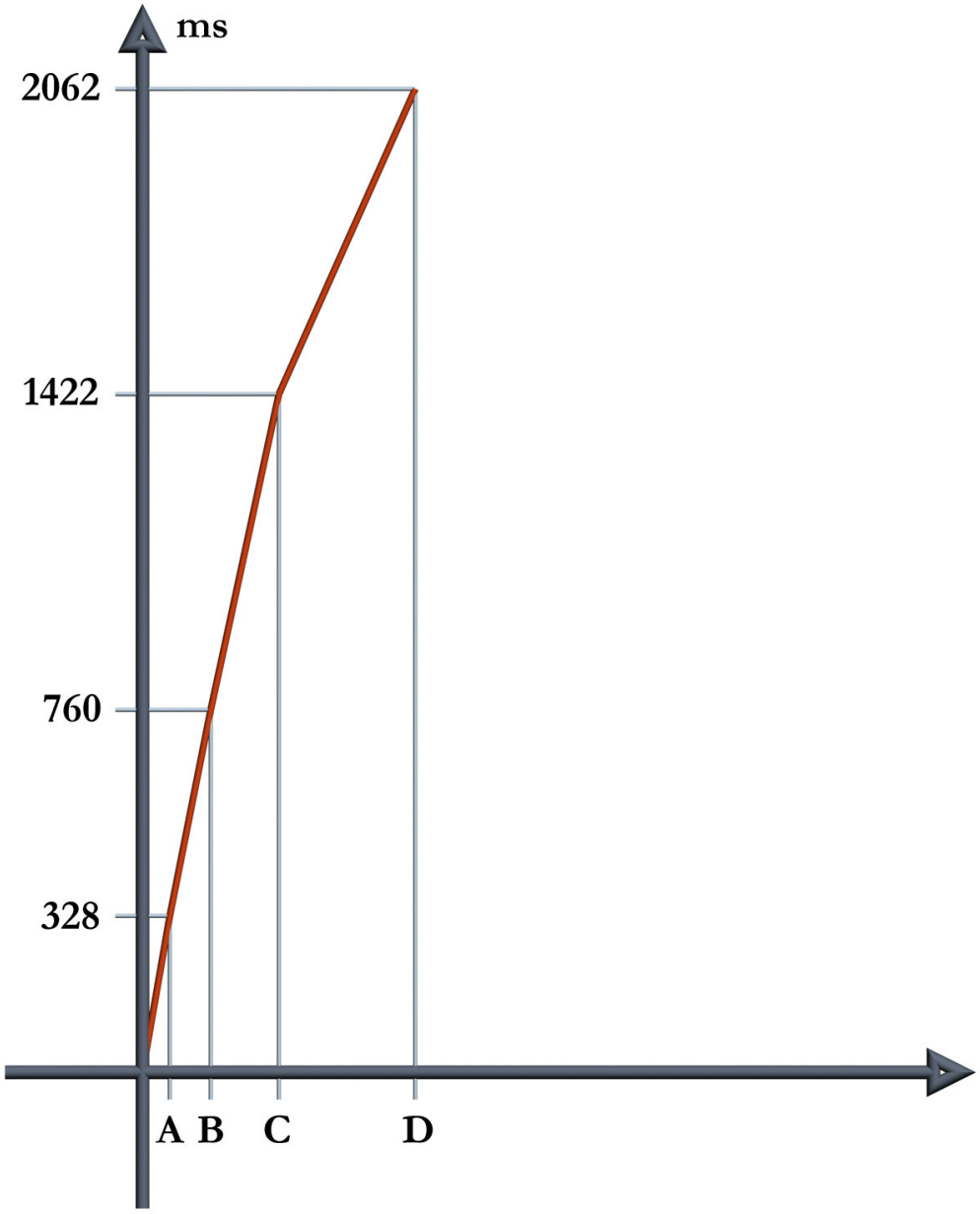
7.1 Introduzione

In questo capitolo sono riportate le misure quantitative e qualitative di AB-Block. Le misure sono state effettuate in un computer con processore Athlon XP 1900+ da 1,6 GHz, con memoria RAM da 768 MB e scheda video GeForce 4 MX 440.
I primi quattro grafici visualizzano i tempi di generazione dei lots per i due algoritmi nei caso in cui vi sia il controllo sulle lunghezze o meno. L'ultimo grafico esprime i tempi di generazione degli edifici nei vari livelli di dettaglio.
Successivamente sono messe a confronto alcune immagini reali di Pisa con l'output di AB-Block. Infine è riportata una serie di screenshots che mostrano la grande varietà degli isolati e degli edifici che è possibile creare con l'applicazione.



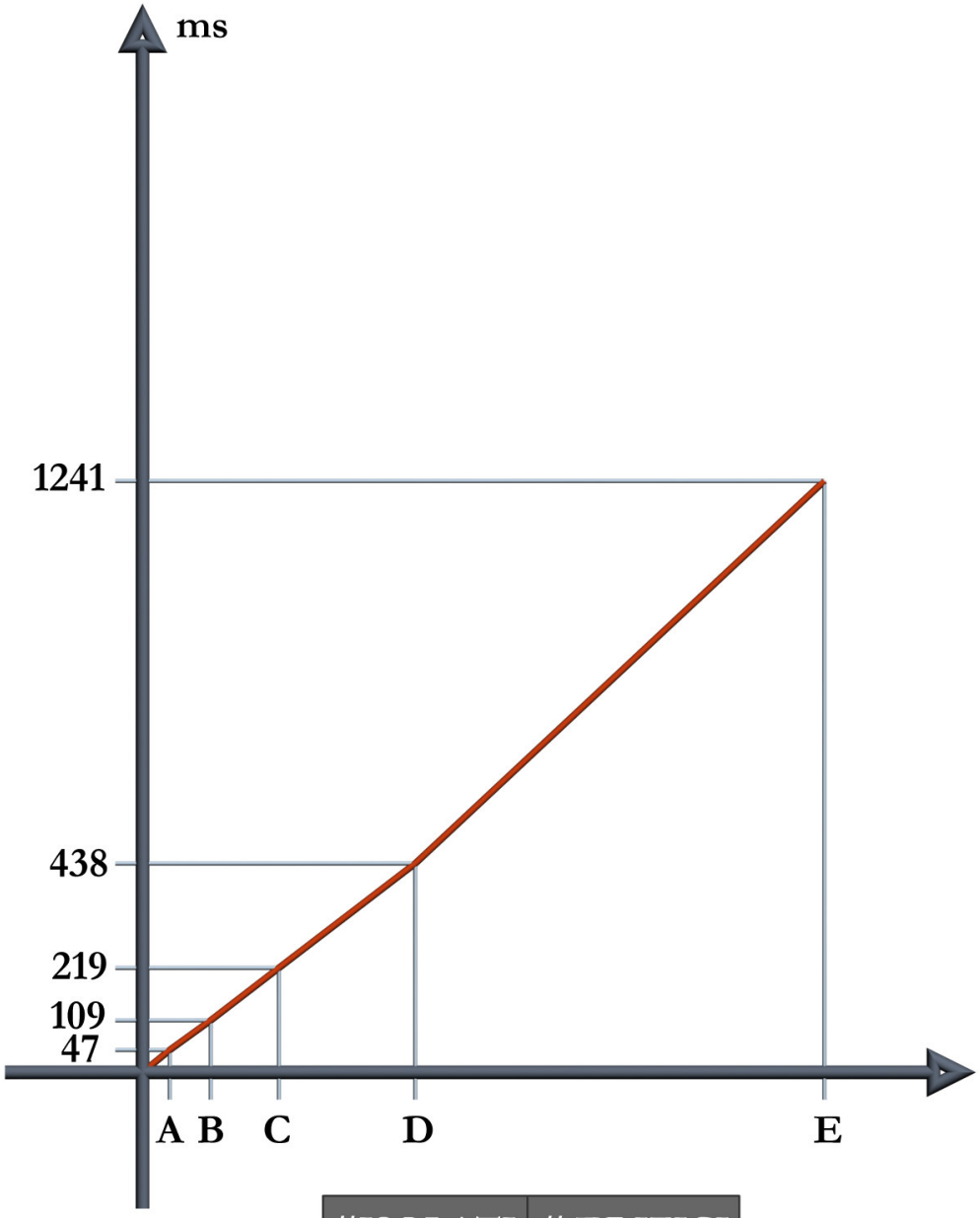
	#ISOLATI	# EDIFICI
A	200	1400
B	500	3500
C	1000	7000
D	2000	14000
E	5000	35000

fig. 72 Grafico dei tempi di generazione del Perimetral senza il controllo della lunghezza delle linee



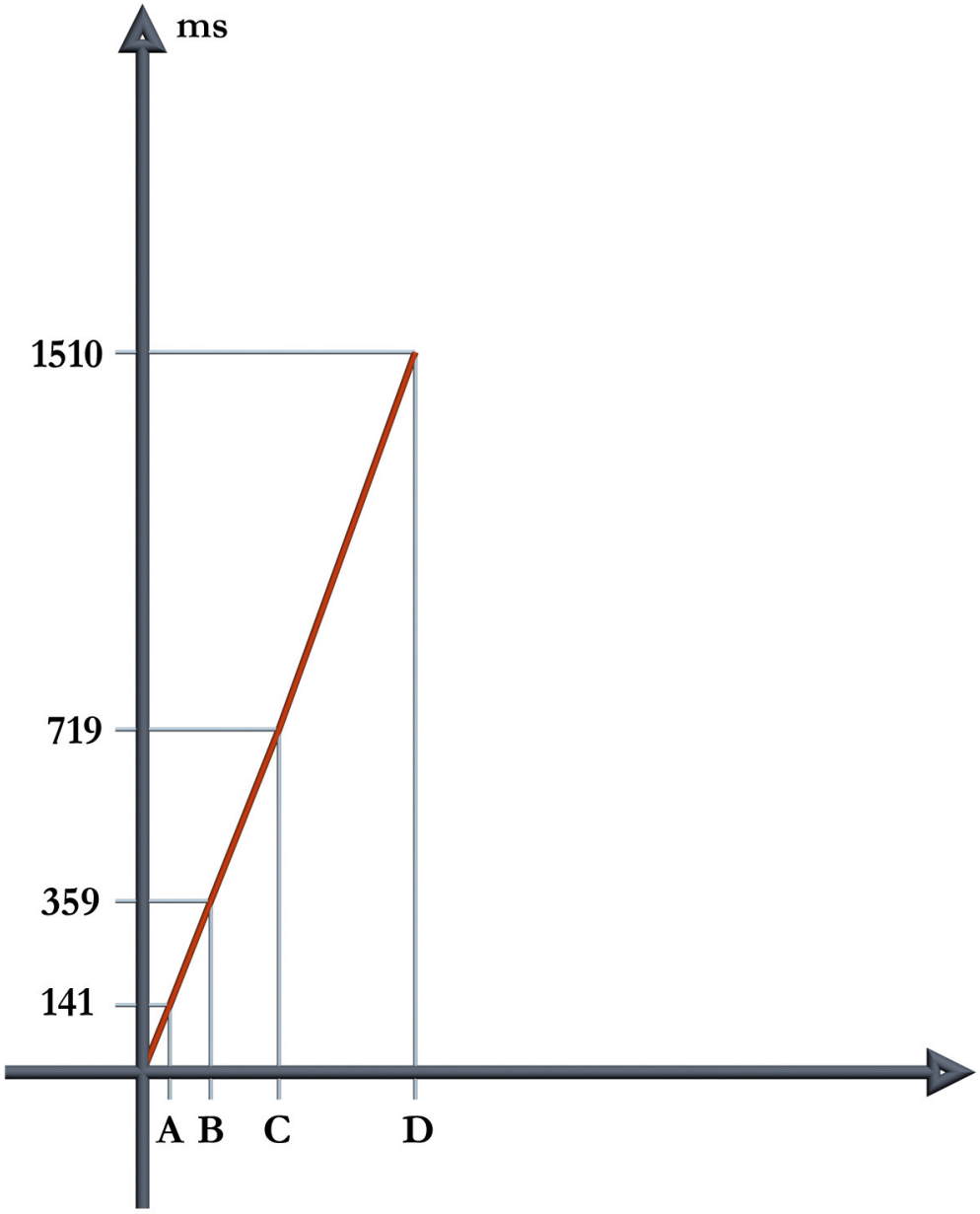
	#ISOLATI	# EDIFICI
A	200	4400
B	500	11000
C	1000	22000
D	2000	44000

fig. 73 Grafico dei tempi di generazione del Perimetral con controllo della lunghezza delle linee



	#ISOLATI	# EDIFICI
A	200	1000
B	500	2500
C	1000	5000
D	2000	10000
E	5000	25000

fig. 74 Grafico dei tempi di generazione del Mix senza il controllo della lunghezza delle linee



	#ISOLATI	# EDIFICI
A	200	3800
B	500	9500
C	1000	19000
D	2000	38000

fig. 75 Grafico dei tempi di generazione del Mix con controllo della lunghezza delle linee

Dai grafici precedenti possiamo notare come ogni algoritmo abbia complessità lineare.
Qui sotto sono rappresentati i tempi di generazione dell'edificio per i tre diversi livelli di dettaglio, le misure si riferiscono alla creazione di 5000 edifici. Il tempo complessivo di generazione dell'isolato è dato dal tempo di allotments, più il tempo di generazione del singolo edificio al livello di dettaglio richiesto per il numero di edifici creati.

$$T_{complessivo} = T_{allotments} + (T_{Lod} / 5000) * \#Edifici$$

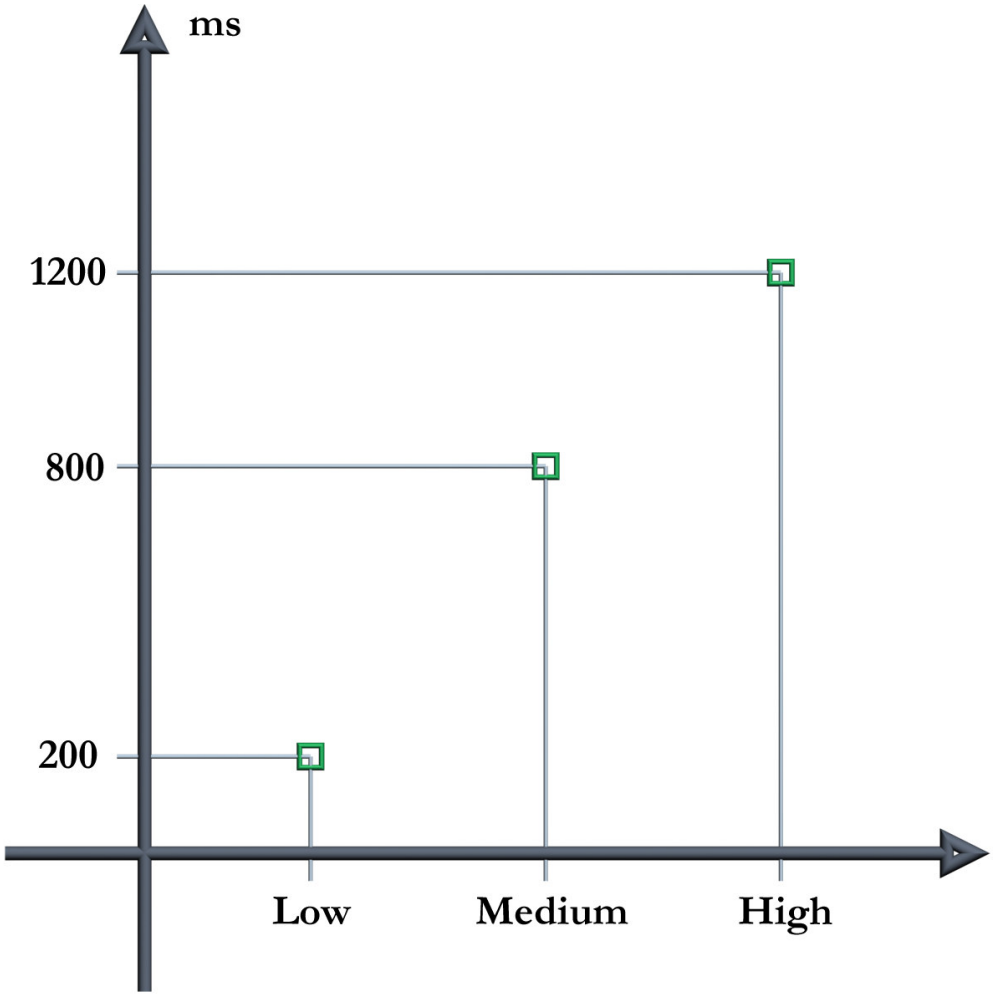


fig. 76 Grafico dei tempi di generazione di 5000 edifici nei tre diversi livelli di dettaglio

fig. 77 Foto aerea di Pisa

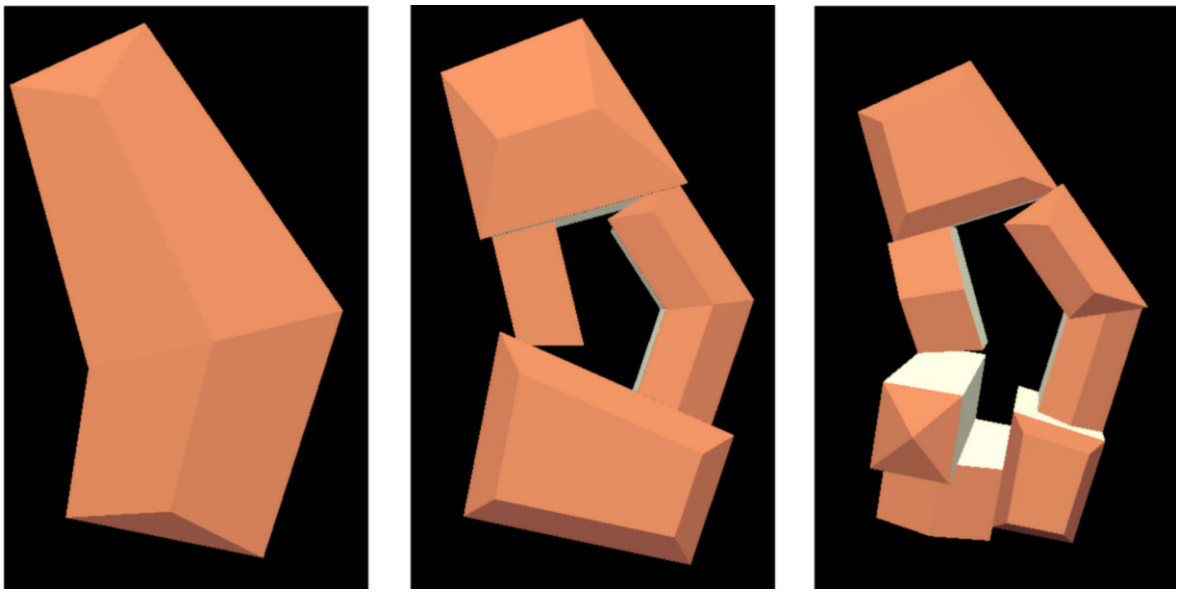


fig. 78 Tre diversi output per lo stesso isolato tramite il Mix (si noti come ricalchino la morfologia degli isolati reali)

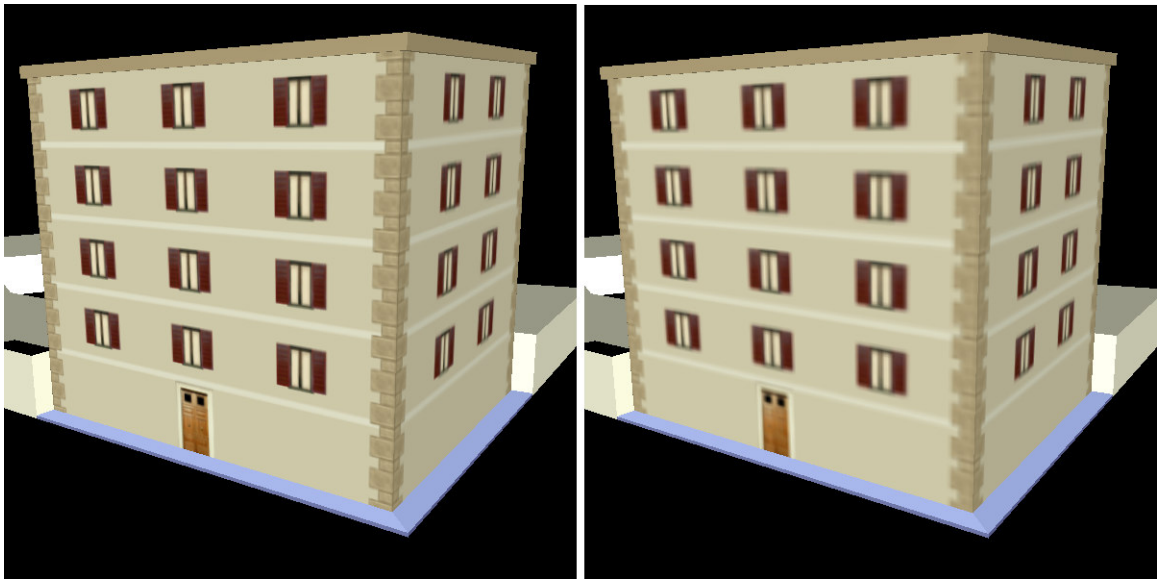


fig. 79 Differenza di qualità tra la texture a grandezza originale e quella ridotta

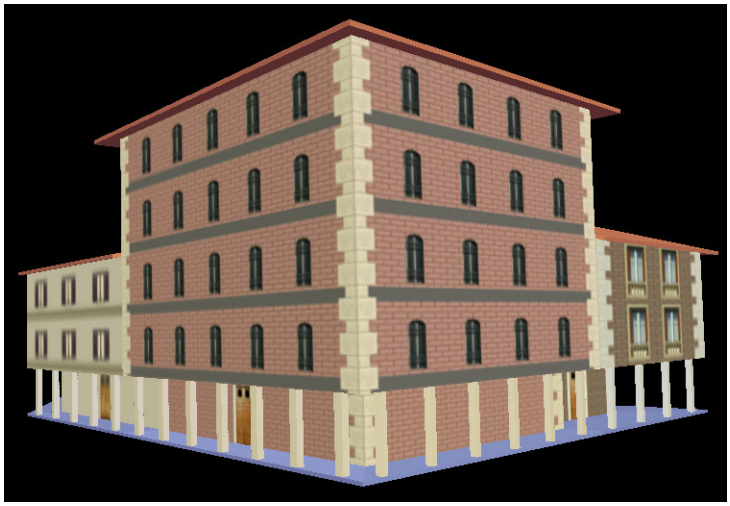


fig. 80 Esempio di portici

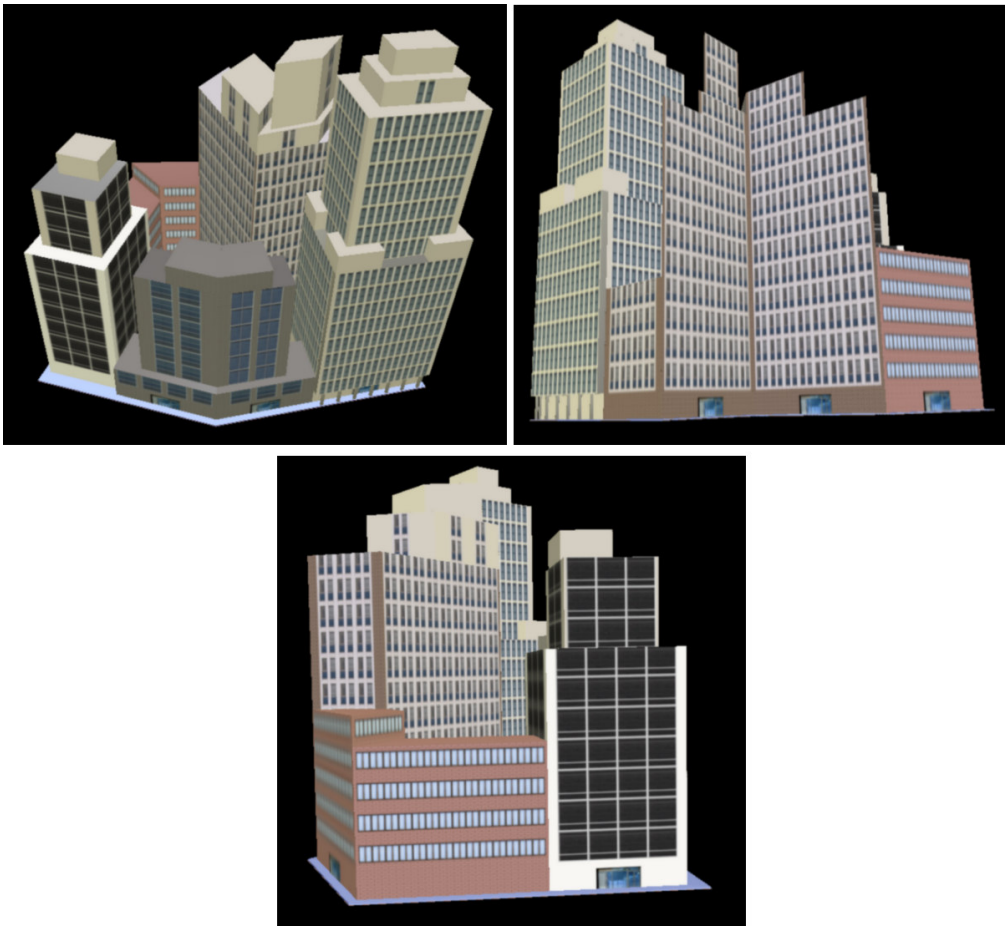


fig. 81 Tripla visuale di un isolato con palazzi

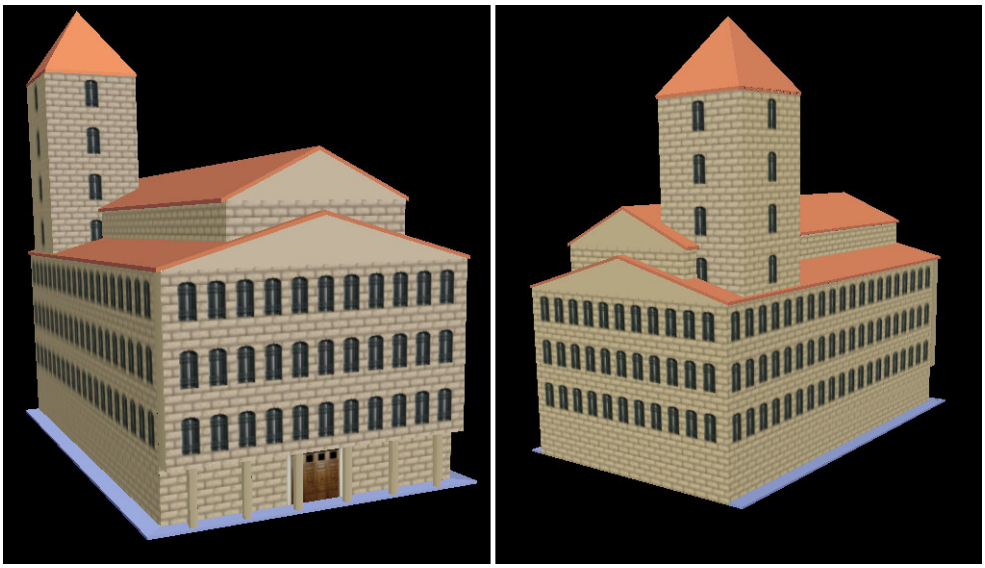


fig. 82 Doppia visuale di una chiesa

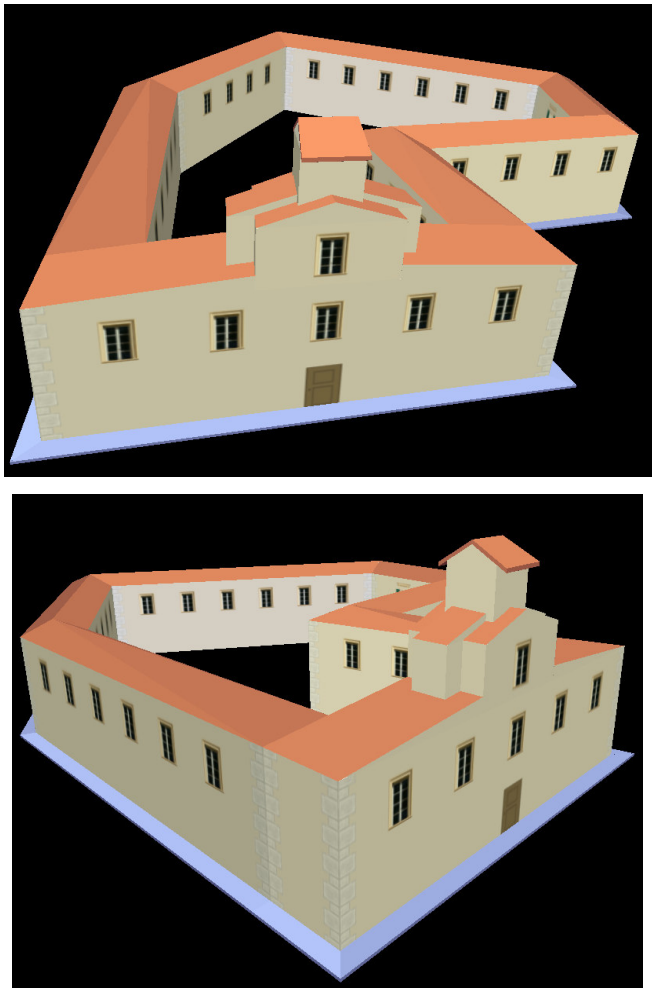


fig. 83 Esempio di corte realizzata con l'algoritmo Perimeter

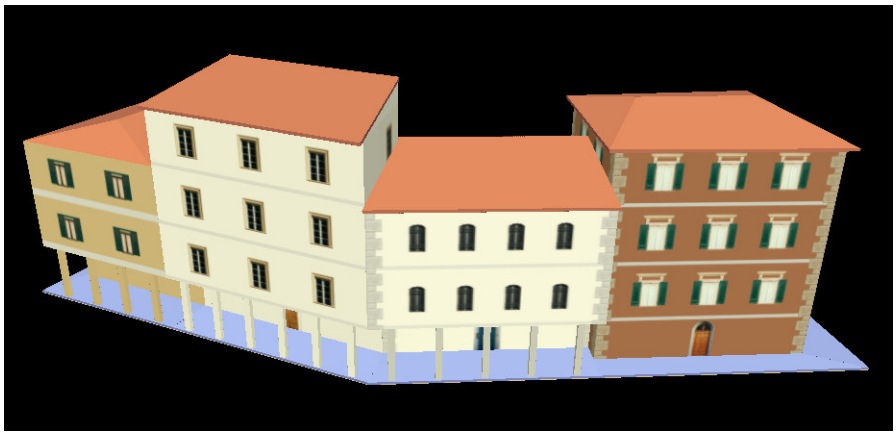


fig. 84 Esempio di edifici tipici pisani

8 Conclusione

8.1 Conclusioni e sviluppi futuri

AB-Block non è, e non vuole essere, una copia di applicazioni esistenti, ma si propone come innovazione nel settore della modellazione procedurale di edifici, più precisamente per ciò che riguarda la fase di allotments e generazione degli edifici.
Ogni algoritmo è originale, creato ad arte per rispettare il paradigma di AB-Block, ossia la località dei controlli sulla conformazione degli isolati e la sintetizzazione delle geometrie tramite linee di riferimento e costruzioni tipiche del disegno tecnico.
L'insieme dei parametri per la generazione è ben bilanciato, studiato in modo da massimizzare la varietà dell'output con il minor numero di inserimenti da parte dell'utente, o di decisioni da parte del programma.
AB-Block-Code è in grado di generare, con pochi dati in input, diverse tipologie di città, dal paesello di provincia sino alla grande metropoli. Inoltre supporta i dislivelli del terreno, la generazione delle geometrie con diversi livelli di dettaglio e i paradigmi del DataAmplification e del Lazy Evaluation.
Il salvataggio viene effettuato in formato AAM, sia con coordinate globali che locali, ogni edificio è memorizzato come oggetto a sé stante per un più facile editing futuro.

AB-Block è facilmente espandibile poiché parametrico, non esiste un limite alla quantità di nuove funzionalità da aggiungere, poiché si può definire un sempre maggiore livello di dettaglio. E' possibile aggiungere nuovi algoritmi per la generazione dei template dei tetti, nuovi stili di portici e di facciate (terrazzi, cornici…) e nuove proprietà per la creazione delle texture.

E' già in progetto una parametrizzazione degli stili degli edifici per la generazione automatica. Questa potrà essere completamente personalizzabile tramite dei file di stile compilabili per mezzo di un semplice editor, questi memorizzeranno le probabilità delle diverse opzioni delle proprietà dell'edificio, le quali verranno estrapolate dal programma per essere utilizzate nella generazione random. Una volta creato, un file di stile potrà essere riutilizzato più volte per diversi isolati.

In progetto anche la possibilità di aprire più isolati in uno stesso file LOB. In questo modo gli isolati con lo stesso stile potranno essere inclusi in un unico file, velocizzando ancor più l'operazione creazione della città.

Altri sviluppi futuri sono: l'implementazione di un nuovo algoritmo di allotments e il salvataggio in formato 3DS.

Il nuovo algoritmo è un'estensione del Mix, ma a differenza di quest'ultimo, tende a coprire maggiormente l'area dell'isolato. Questo non è altro che un Mix iterativo in cui, ad ogni passo, vengono determinate nuovamente le linee dell'isolato prendendo quelle del perimetro dell'area ancora da coprire.

Il salvataggio in formato 3DS permetterà una più grande portabilità dell'output, la visualizzazione e l'ulteriore editing manuale dei singoli edifici. Inoltre le città potranno essere renderizzate con un motore di rendering superiore, come ad esempio quello di 3d Studio Max.

Bibliografia

[1] *"Texturing & Modeling – A Procedural Approach"* David S. Ebert, F. Kenton Musgrave Darwyn Peachey, Ken Perlin, Steven Worley (Morgan Kauffman Publishers, 2003)

[2] *"Automatic Generation of Urban Zone"* Mathieu Larive, Yann Dupuy, Véronique Gaildrat (WSCG, 2005)

[3] *"Procedural Modeling of Cities"* Yoav I. H. Parish, Pascal Müller (ACM, SIGGRAPH 2001)

[4] *"Undiscovered Worlds – Towards a FrameWork for Real-Time Procedural World Generation"* Stephan Greuter, Jeremy Parker, Nigel Stewart, Geoff Leach (MelBourneDAC, 2003)

[5] *"Instant Architecture"* Peter Wonka, Michael Wimmer, François Sillion, William Ribarsky (ACM, Transaction on Graphics 2003)

[6] *"Real-time Procedural Generation of 'Peudo Infinite' Cities"* Stephan Greuter, Jeremy Parker, Nigel Stewart, Geoff Leach (GRAPHITE, 2003)

[7] *"Automatic Generalisation of 3D Building Models"* Martin Kada (Institute of Fotogrammetry, University of Stuttgart, Germany)

[8] *"Procedural Modeling of Land Use in Cities"* Thomas Lechner, Ben Watson, Pin Ren, Uri Wilesky, Seth Tisue

[9] *"Automatic Generation of 3D City Models and Related Applications"* Y. Takase, N. Sho, A. Sone, K. Shimiya (Journal of the visualization Society, 2003)

[10] *"Lowering the Development Time of Multimodal Interactive Application"* Marcello Carrozzino, Franco Tecchia, Sandro Bacinelli, Carlo Cappelletti, Massimo Bergamasco (PERCRO)

[11] *"Using 3D Geometric Constraints in Architectural Design of Support System"* B. DeVries, A. J. Jessurun, R. H. M. C. Kelleners

[12] *"Cartografia Generale Tematica e Numerica"* A. Selvini, F. Guazzetti

[13] *"Algorithmic Beauty of Buildings Methods for Procedural Building Generation "* Jess Martin (2005)

[14] *"Modeling Structure Patterns in Road Network Generalisation"* Qingnian Zhang (ICA WorkShop on Generalisation e Multiple Rappresentation, 2004)

[15] *"Rapid Procedural Modeling of Architectural Structure"* P.J. Birch, S.P. Browne, V.J. Jennings, A.M. Day, D.B. Arnold (University of East Anglia)

[16] *"Real-Time Expressive Rendering of City Models"* Junger Dollner, Maike Walter

[17] *"From Urban Terrain Model to Visibles Cities"* W. Ribarsky, T. Wasilewsky, N. Faust (IEEE Computer Graphics and Applications, 2002)

[18] *"Procedural City Modeling"* Thomas Lechner, Ben Watson, Uri Wilensky, Martin Felsen

[19] *"Modeling and Rendering of Large Urban Environment "* P.J. Birch, S.P. Browne, V.J. Jennings, A.M. Day, D.B. Arnold (University of East Anglia)

[20] *"The Algoritmic Beauty of Plants"* Przemyslaw Prusinkiewicz, Aristid Lindenmayer, James S. Hanan, F. David Fracchia, Deborah Fowler, Martin J.M. De Boer, Lynn Mercer (2004)

[21] *"On efficiently Representing Procedural Geometry"* John C. Hart (Washington State University)

Materiale on-line

[22] *www.vterrain.org*

[23] *www.trolltech.com*

[24] *www.opengl.org*

[25] *www.urbansimulation.com*