UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Scienze
Matematiche Fisiche e Naturali

Corso di Laurea Specialistica
in Tecnologie Informatiche

Tesi di Laurea

Ricostruzione volumetrica e visualizzazione interattiva di ambienti urbani

Candidato:

Daniele Vacca

Relatori:

Dr. Paolo Cignoni

Dr. Fabio Ganovelli

Controrelatore:

Dr. Giuseppe Prencipe

Anno Accademico 2007/08

Amiga Kickstart v1.3 - Tutti i diritti riservati

Alla mia famiglia,
per essere stata sempre al mio fianco
in questi anni di studio.

Ai relatori e amici del VCG,
per l'infinita pazienza e i preziosi consigli.

Agli amici, quelli veri.
Quelli con cui ho passato momenti indimenticabili,
ma anche quelli su cui ho potuto contare nei momenti difficili.

Ricostruzione volumetrica e visualizzazione interattiva di ambienti urbani

Daniele Vacca

Università di Pisa

Dipartimento di Informatica

Sommario

In questo lavoro di tesi viene affrontato il problema della visualizzazione efficiente di ambienti urbani.

Grazie ai recenti sviluppi tecnologici in materia di digitalizzazione tridimensionale e alla conseguente diffusione di questo strumento di indagine anche al di fuori dell'ambiente della ricerca, la quantità e la qualità dei modelli digitali di ambienti urbani è cresciuta esponenzialmente. Nonostante esista una vasta letteratura ed una serie di tecnologie ben affermate per la visualizzazione efficiente dei modelli 3D generici, le particolari caratteristiche geometrico topologiche di questa classe di modelli ne rendono difficile una loro gestione estremamente efficiente e di alta qualità.

In questa tesi proponiamo una tecnica di codifica compatta e visualizzazione efficiente di ambienti urbani basata su una approssimazione volumetrica della forma e del colore dei vari edifici. Tale approssimazione, che chiameremo V-BlockMap, può essere memorizzata sotto forma di un'immagine 2D e, una volta trasmessa all'hardware, grafico può essere decodificata e visualizzata in maniera estremamente efficiente da un apposito algoritmo di ray-tracing su GPU. Il lavoro di tesi si completa con un implementazione prototipale della tecnica di costruzione e di visualizzazione delle V-BlockMap al fine di dimostrarne la qualità visiva e confrontarla con la classica rappresentazione poligonale.

Indice

Capitolo 1

Introduzione

Negli ultimi anni, grazie alla crescente disponibilità e accessibilità di strumenti di digitalizzazione tridimensionali, è notevolmente aumentata la quantità e la qualità dei modelli virtuali che vengono prodotti con questo tipo di tecnologie.

La tipologia dei modelli digitali ottenibili con tecniche di 3D scanning è molto varia: si possono avere oggetti di media dimensione la cui superficie è campionata in maniera molto densa (circa 9 campioni per millimetro quadrato), oggetti di grandi dimensioni come interi edifici in cui il campionamento è nell'ordine del punto per centimetro fino ad oggetti su scala planetaria per i quali il campionamento è nell'ordine del punto per decine di metri.

Nel primo caso possiamo citare modelli di sculture quali il *David* di Michelangelo [LPC+00], la Minerva di Arezzo [RCM+01, FGM+02], o la Portalada, l'imponente ingresso al monastero di Ripoll [BBC+08]. Esempi di digitalizzazione di edifici comprendono il Duomo di Pisa [BUSM06] e il Partenone, l'antico tempio greco che sorge sull'Acropoli di Atene [Deb04]. Su scala planetaria i dati vengono prodotti sia da scansioni satellitari, come nel caso del progetto MOLA (Mars Orbital Laser Altimeter)[SG02], che da tecniche aeree fotogrammetriche.

Anche la diffusione di modelli relativi ad ambienti urbani è in costante aumento, specialmente con l'avvento di *Google Earth* e *Microsoft Virtual Earth*, i quali hanno reso disponibili ad un vasto pubblico un'elevata quantità di foto satellitari.

Questa disponibilità di dati pone il problema di come questi possano essere visualizzati in tempo reale, in quanto la loro mole è di vari ordini di grandezza superiore a quello che i computer oggi disponibili sono in grado di visualizzare. Questo gap riguarda tutti i passi necessari alla loro visualizzazione, a partire dalla memorizzazione su disco, passando per il loro trasferimento alla memoria on-board della scheda grafica e, in ultimo, la potenza del processore grafico stesso.

La visualizzazione in tempo reale di una tale quantità di dati non è un problema nuovo nella computer grafica ed è da sempre un elemento critico in numerose applicazioni. Questo problema è stato affrontato in letteratura con approcci differenti per ciascun tipo di modello da visualizzare, attraverso soluzioni tipicamente basate su modelli multirisoluzione e/o livelli di dettaglio adattativi.

Gli approcci esistenti sono comunque sviluppati ad hoc per il tipo di dato, per cui la tecnica per la visualizzazione del terreno non sarà ottimale (o persino neppure applicabile) per la statua e viceversa.

In questo contesto il caso degli ambienti urbani è quello meno trattato in letteratura, soprattutto perché solo di recente si ha una vasta disponibilità di questo tipo di dato e presenta peculiarità che sostanzialmente invalidano gli approcci già sviluppati per gli altri casi.

In questa tesi si è sviluppato un sistema innovativo rivolto alla specifica rappresentazione e visualizzazione di ambienti urbani.

L'idea attorno alla quale si sviluppa l'approccio è che gli ambienti urbani possono essere efficacemente approssimati da parallelepipedi uniformemente distribuiti su una griglia regolare, poiché ne catturano la geometria su

larga scala (i profili verticali). In questa verrà mostrato come questo tipo di rappresentazione può essere efficacemente codificato in maniera compatta non solo la forma geometrica, ma anche il colore e altri attributi utili alla visualizzazione, e come sia possibile, utilizzando le funzionalità più avanzate dell'hardware grafico commercialmente disponibile, visualizzarlo in tempo reale.

1.1 Organizzazione della tesi

Nel capitolo 2 viene descritto il problema della visualizzazione di scene tridimensionali complesse e vengono illustrati gli algoritmi e le tecniche utilizzate per risolverlo. Nel capitolo 3 è descritto in dettaglio il lavoro svolto nell'ambito di questa tesi. Questo comprende la realizzazione di una tecnica di ricostruzione volumetrica, interamente eseguita a bordo della scheda grafica, la parametrizzazione del volume ottenuto e, infine, gli algoritmi utilizzati per la visualizzazione interattiva. I risultati ottenuti da queste fasi verranno dunque analizzati all'interno del capitolo 4. Infine, nell'ultimo capitolo saranno discussi i limiti e i possibili miglioramenti del sistema realizzato.

Capitolo 2

Stato dell'arte

2.1 Elementi introduttivi

I modelli tridimensionali possono essere definiti in vari modi: come funzioni parametriche implicite o come insieme di poligoni, ma, al momento del rendering, essi dovranno essere convertiti in un insieme di primitive per le quali l'hardware grafico è stato progettato, cioè punti, linee e triangoli. Quindi, senza perdita di generalità, assumeremo che ogni modello tridimensionale sia definito dall'insieme dei punti e dall'insieme delle facce triangolari che lo compongono.

Il procedimento con il quale i triangoli di una mesh vengono trasformati e visualizzati su uno schermo bidimensionale prende il nome di *pipeline di rendering*. Questo procedimento si costituisce di numerose fasi, le quali possono differire per ordine, numero e caratteristiche a seconda della libreria grafica utilizzata, ma gli stadi principali restano comuni nelle varie implementazioni [Ang05]. Essi sono illustrati in figura 2.1:

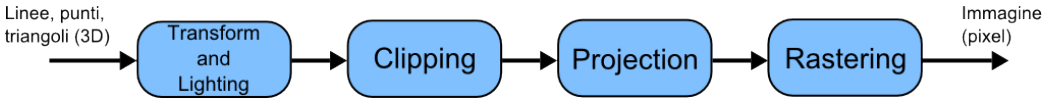


Figura 2.1: *Schema della pipeline di rendering.*

- **Trasformazione e Illuminazione (T&L)**
  In questo stadio le primitive (punti, triangoli, segmenti) vengono trasformate e illuminate secondo i parametri definiti dalla scena virtuale. Per *trasformazione* si intende quell'insieme di operazioni applicate ai vertici delle primitive, definiti nel loro sistema di riferimento, che permette di esprimerli nel sistema di riferimento della scena. Dopo essere state trasformate, le primitive vengono quindi illuminate attraverso specifici algoritmi di illuminazione.

- **Clipping**
  Questo stadio ha lo scopo di scartare le primitive sicuramente non visibili, cioè quelle che non ricadono all'interno del *volume di vista*. Tale volume rappresenta quella porzione tridimensionale della scena virtuale il cui contenuto sarà proiettato sullo schermo. Pertanto, ciò che ricade *interamente* all'esterno del volume sarà sicuramente non visibile, mentre, ciò che vi ricade all'interno risulterà *potenzialmente* visibile. Le primitive che attraversano il volume di vista, che risultano quindi parzialmente interne, vengono ritagliate in modo opportuno: la parte interna prosegue nella pipeline, mentre la restante parte viene scartata.

- **Proiezione**
  Successivamente, le coordinate dei vertici che descrivono le primitive vengono proiettate sul piano di vista (*view plane*) così da poter essere visualizzate sullo schermo. L'intero contenuto del volume di vista viene

quindi *mappato* su un piano attraverso i parametri di proiezione (tipo di prospettiva, lunghezza focale, dimensioni del volume...).

- **Rasterizzazione (o Scan Conversion)**
  Infine, avviene la *rasterizzazione* delle primitive. Questo procedimento trasforma la descrizione bidimensionale dei vertici che compongono le primitive, nell'insieme di pixel *da accendere* sullo schermo. Nel caso di un triangolo, ad esempio, verranno accesi tutti i pixel che ne definiscono il contorno, ma anche tutti quelli interni che ne definiscono l'area.

Per ogni fotogramma, la pipeline di rendering viene attraversata numerose volte, una per ciascuna primitiva che si vuole disegnare. In un'applicazione grafica, la visualizzazione di una scena in movimento avviene per fotogrammi consecutivi, prodotti da numerose elaborazioni svolte sia dalla CPU sia dal processore grafico (GPU). Di conseguenza, maggiore sarà la velocità impiegata nel generare i singoli fotogrammi e più realistica sembrerà la simulazione interattiva. Spesso questo concetto viene utilizzato come parametro di riferimento nel confronto tra applicazioni: la più veloce è quella che produce il miglior *frame rate*, ovvero, quella che offre il maggior numero di frame generati per istante di tempo. Il *frame rate* viene tipicamente misurato in fotogrammi per secondo (*FPS*).

A causa di alcuni fattori biologici, l'occhio umano non riesce a percepire più di 25-30 fotogrammi distinti in un secondo. Quindi, per dare l'impressione di buona fluidità e continuità della visualizzazione, le applicazioni grafiche non devono scendere al di sotto di questo valore.

Di seguito vedremo in che modo è possibile migliorare le prestazioni della pipeline di rendering, con l'obiettivo di produrre il maggior numero di fotogrammi al secondo.

2.1.1 Rimozione delle superfici nascoste

I classici colli di bottiglia della pipeline di rendering sono rappresentati dalla trasformazione e dalla rasterizzazione. L'efficienza del primo stadio è infatti legata al numero di vertici che deve processare. Nella fase di rasterizzazione, invece, oltre al numero dei poligoni, gioca un ruolo fondamentale anche la loro dimensione. Esistono in letteratura numerosi algoritmi volti a snellire il carico di lavoro di questi stadi , alcuni sono implementati direttamente sull'hardware grafico, mentre altri richiedono un piccolo sforzo di programmazione. Essi ricadono sotto il nome di *algoritmi di culling* e giocano un ruolo fondamentale nelle rappresentazioni interattive di scene complesse [Ang05, WDS99]. Di seguito è illustrato il funzionamento di due tecniche classiche, la prima rivolta a ridurre il numero di vertici da elaborare, l'altra rivolta a ridurre il numero di facce da visualizzare.

Frustum Culling

Con questo termine ci si riferisce a quella categoria di algoritmi che evitano di trasformare e illuminare inutilmente gli oggetti di una scena che *sicuramente* verrebbero scartati dalla successiva fase di clipping. Per fare questo, è necessario che un'applicazione memorizzi, in modo esplicito, le caratteristiche del volume di vista (*viewing frustum* appunto) quali dimensioni e forma. A questo punto, prima di inviare i vertici degli oggetti all'API grafica, è sufficiente verificare se una loro *sovrastima* risulti interna o esterna al volume di vista. Per semplicità di calcolo, come sovrastima degli oggetti si utilizzano i *bounding volumes*, come *bounding sphere* o *bounding box*, ovvero la sfera e il cubo che racchiudono un oggetto [Cla76].

Backface Culling

Se si immagina una mesh tridimensionale come un guscio vuoto di cui non si può vedere l'interno, si intuisce che le facce visibili sono solo quelle rivolte verso l'osservatore, mentre le altre non verranno mai viste, in quanto occluse dalle prime e pertanto possono essere scartate. Attraverso il *backface culling* [SSRS74], è quindi possibile evitare di disegnare tutti quei triangoli allineati lungo la direzione di vista. Questo permette di ridurre di circa la metà il numero medio di triangoli disegnati ad ogni fotogramma.

Con opportune modifiche alla pipeline di rendering, è possibile implementare questo algoritmo in modo estremamente efficiente, cioè eseguendo il solo test sul segno della coordinata Z del vettore ortonormale alla superficie di ogni triangolo. Infine, essendo completamente indipendente dal contenuto della scena da rappresentare, il *backface culling* risulta particolarmente semplice da implementare direttamente sull'hardware grafico.

Le tecniche appena illustrate sono orientate a ridurre il numero di primitive (vertici o triangoli) da processare. Tuttavia, nelle scene complesse esse non sono più sufficienti poiché mirano a rimuovere i soli oggetti non visibili lasciando inalterati quelli visibili. Si intuisce facilmente che se un oggetto visibile è comunque costituito da un numero estremamente elevato di vertici e triangoli, la sua visualizzazione risulterà poco fluida.

A tale scopo, sono stati sviluppati degli algoritmi il cui obiettivo è quello di ridurre il numero di triangoli degli oggetti visualizzati, in modo tale che la loro visualizzazione risulti il più fedele possibile alla geometria non semplificata. Questi algoritmi prendono il nome di *tecniche multirisoluzione*.

2.2 Le tecniche multirisoluzione

Tutti gli approcci multirisoluzione si basano sullo stesso principio, cioè quello di creare, a partire dalla geometria di partenza, diverse rappresentazioni dello stesso oggetto, ciascuna con un livello di dettaglio via via decrescente. A differenza delle tecniche basate su *LOD* (*Level Of Detail*), la visualizzazione di modelli attraverso tecniche multirisoluzione non avviene sostituendo l'intero oggetto con una sua versione a dettaglio maggiore (minore); ciò che avviene è una sostituzione *parziale*, delle sole regioni dell'oggetto che richiedono un maggiore (minore) dettaglio.

Ad ogni rendering (frame) viene scelta, o creata in tempo reale, la rappresentazione che meglio si adatta ad essere visualizzata in quel momento, in funzione della distanza dall'osservatore. Gli oggetti lontani, infatti, occupano meno spazio sullo schermo, pertanto, utilizzare un elevato numero di poligoni per visualizzarli sarebbe un'inutile spreco di risorse [Cla76]. Attraverso le tecniche multirisoluzione è dunque possibile scegliere, per ciascun oggetto, la rappresentazione che meglio si adatta ad essere visualizzata su una certa porzione di schermo. Man mano che l'oggetto semplificato si avvicina all'osservatore, e quindi l'area che ricopre sullo schermo diventa sempre più ampia, viene sostituito da un'altra rappresentazione con livello di dettaglio maggiore, fino a che non si raggiunge il livello di dettaglio massimo, rappresentato dalla geometria originale. Come si intuisce, le tecniche multirisoluzione sono basate su una struttura gerarchica, in cui la radice rappresenta la geometria al minimo livello di dettaglio, mentre le foglie rappresentano il livello di dettaglio più alto.

L'utilizzo di queste tecniche, combinate ai classici algoritmi di *culling*, permette di eseguire il rendering di scene complesse in modo decisamente rapido. Esistono diversi categorie di tecniche multirisoluzione, ognuna specializzata nella visualizzazione di determinati tipi di modelli. Di seguito

vedremo due esempi tipici del loro utilizzo: la visualizzazione di terreni e la visualizzazione di statue.

2.2.1 Terrain rendering

La visualizzazione interattiva di terreni gioca un ruolo critico in numerose applicazioni che spaziano dalla visualizzazione scientifica ai simulatori di volo. In queste applicazioni, i terreni devono essere visualizzati in modo tale da garantire un elevato livello di dettaglio quando visualizzati *da vicino* e, nel contempo, garantire un elevato e stabile *frame rate*.

Quando le dimensioni del dataset sono ridotte, è sufficiente applicare le tecniche di culling illustrate precedentemente, ma, nel caso in cui il dataset sia di dimensioni ragguardevoli, si rende necessario l'utilizzo di tecniche multirisoluzione. Di seguito vedremo brevemente due di queste tecniche molto comuni, *ROAM* e *BDAM*.

ROAM

L'acronimo sta per **R**eal-time **O**ptimally-**A**dapting **M**eshes [DWS+97]. In questa categoria di algoritmi si presuppone che la mesh rappresentante il terreno sia definita da un campo di altezze. I campi (mappe) d'altezze sono delle particolari mesh in cui, per ogni vertice, si memorizza un solo valore e cioè la sua altezza. Le altre coordinate sono ricavate implicitamente dalla posizione che il vertice occupa all'interno di una griglia regolare. Grazie a questa regolarità, è molto facile ottenere una mesh triangolare, tuttavia, se si utilizza una triangolazione regolare, una griglia di dimensioni 1024×1024 produrrebbe oltre 2 milioni di triangoli da visualizzare. Con un numero così elevato di poligoni su schermo, sarebbe difficile ottenere un *frame rate* accettabile. Gli algoritmi ROAM risolvono questo problema generando, ad ogni fotogramma, una differente triangolazione *non* regolare del campo d'altezze.

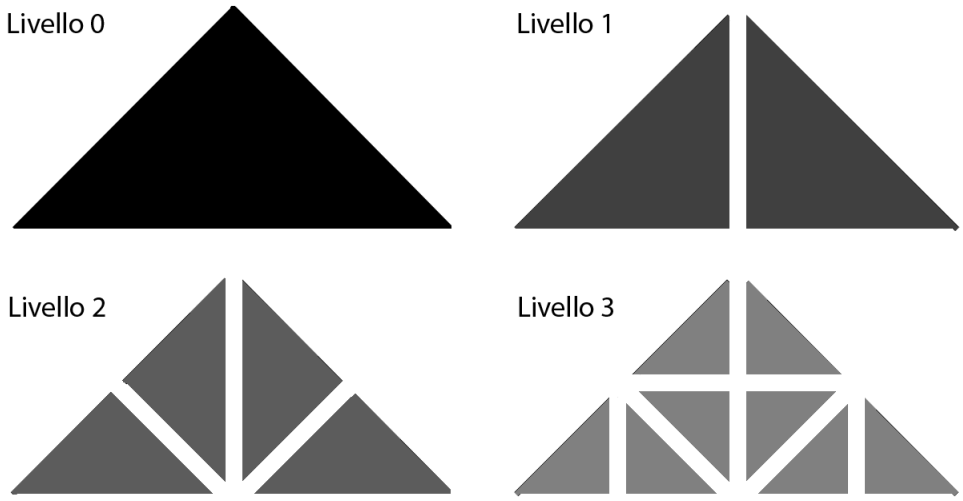


Figura 2.2: *Quattro livelli di un Binary Triangle Tree.*

In questo modo se l'osservatore resta fermo, la visualizzazione del terreno non cambia, mentre, se l'osservatore si muove, la triangolazione sarà differente, così da garantire un *frame rate* costante.

In questi algoritmi la struttura dati principale è un albero binario di triangoli (*binary tree triangle, BTT*), in cui ogni nodo è rappresentato da un triangolo isoscele retto. Ciascuno di questi triangoli può essere *tagliato* lungo il segmento che congiunge il punto medio dell'ipotenusa con il corrispondente l'angolo retto, cosicché i due triangoli generati risultino a loro volta isosceli e retti. Nella struttura dati questi rappresentano i figli del triangolo di partenza e possono essere suddivisi a loro volta in modo ricorsivo (come da figura 2.2). L'operazione appena descritta prende il nome di *split* e si contrappone all'inversa operazione di *merge*, la quale, dati due triangoli appartenenti allo stesso livello $l + 1$, li unisce per formare il triangolo di livello l.

Ad ogni fotogramma, si parte dai due triangoli che ricoprono l'intero

campo d'altezze, disposti in modo tale che le loro ipotenuse giacciano sull'ipotetica diagonale della griglia. Questi triangoli vengono quindi suddivisi, attraverso le operazioni di *split*, fino a quando non è soddisfatta una determinata condizione per la terminazione. Tale condizione è differente nelle varie implementazioni: può ad esempio essere in funzione del numero complessivo di triangoli della mesh generata, oppure potrebbe valutare una stima dell'errore geometrico prodotto dalla triangolazione. Ad ogni modo, le zone più vicine all'osservatore saranno visualizzate con un numero maggiore di triangoli (dettaglio migliore), mentre quelle lontane saranno visualizzate con meno triangoli.

Le altre varianti principali di tali algoritmi si differenziano nel tipo di operazioni applicabili ai triangoli: le varianti *split-only* implementano le sole operazioni di taglio, mentre, le varianti *merge-split* implementano sia i tagli sia le fusioni.

BDAM

Gli algoritmi di tipo ROAM, sebbene siano capaci di ridurre enormemente il numero di triangoli da visualizzare per ogni fotogramma, hanno il principale difetto di concentrare troppe risorse di calcolo (CPU) sulla generazione di un singolo triangolo. Paradossalmente infatti, la triangolazione appesantisce notevolmente il processo di rendering, ma, poiché il numero di triangoli generati risulta facilmente gestibile dall'hardware grafico, la loro visualizzazione appare comunque fluida. Questo approccio, inoltre, non sfrutta a dovere le enormi potenzialità degli attuali processori grafici capaci di visualizzare qualche migliaio di triangoli così come se fosse uno solo. A tal proposito sono stati sviluppati gli algoritmi *BDAM* (**B**atched **D**ynamic **A**daptive **M**eshes)[CGG+03, GMC+06].

Il concetto che sta alla base delle tecniche BDAM è quello di dividere il

terreno in porzioni triangolari (esattamente come per i ROAM), ma ciascuna di queste porzioni non rappresenta un singolo triangolo, bensì una vera e propria mesh creata e ottimizzata *off-line* attraverso efficienti algoritmi di semplificazione. In aggiunta, è prevista un'ulteriore struttura multirisoluzione (un albero quaternario) rivolta alla memorizzazione delle informazioni di texture.

La fase di rendering quindi non consiste più in una triangolazione *al volo* di una mappa d'altezze, bensì può essere vista come un procedimento che assembla delle porzioni triangolari di terreno in modo opportuno il cui livello di dettaglio è in funzione della distanza dall'osservatore. La generazione e l'assemblaggio di queste pezze triangolari sono realizzate in modo tale da garantire che le zone adiacenti non si sovrappongano e, inoltre, di contenere l'errore, introdotto dalla semplificazione, entro una certa soglia.

2.2.2 Large scale model rendering

Il processo di visualizzazione interattiva per oggetti quali statue o sculture è sostanzialmente diverso rispetto a quello dei terreni. I campi d'altezze, infatti, rappresentano un particolare caso di mesh triangolari adatte alla sola rappresentazione di oggetti bidimensionali, o facilmente parametrizzabili lungo due dimensioni. Pertanto, essi risultano inadeguati alla memorizzazione di oggetti completamente tridimensionali o non esprimibili in forma parametrica.

Oggetti come statue o artefatti sono memorizzati mediante mesh triangolari. Esse descrivono un oggetto attraverso un insieme di vertici V e un insieme di facce triangolari F. Ciascuna faccia risulterà connessa e adiacente alle altre attraverso un unico lato condiviso chiamato *edge*.

Di seguito verranno illustrati due esempi di algoritmi che operano sulla

Figura 2.3: *Il San Matteo di Michelangelo visualizzato con* 1473 *pezze per un totale di circa* 22000 *triangoli (il dataset originale è composto da circa* 320 *milioni di triangoli).*

struttura di una mesh, in modo da ridurre il numero di vertici e di facce che la compongono.

TetraPuzzle

L'idea che sta alla base di questo algoritmo è la stessa che caratterizza il già visto **BDAM** (2.2.1). Nel *tetrapuzzle* però, la gerarchia di pezze triangolari è sostituita da una gerarchia di regioni tridimensionali tetraedrali [CGG+04]. Ciascuna di queste regioni memorizza una porzione del modello ad un livello di dettaglio che varia in funzione della posizione del nodo nella gerarchia. Anche in questo caso, le porzioni di spazio vengono create e ottimizzate *off-line* attraverso efficienti algoritmi di semplificazione, quindi, così come avviene nei BDAM, la fase di visualizzazione consiste nel scegliere quali di queste rappresentazioni si adattano maggiormente ai parametri della scena e assemblarle per ricostruire il modello originale.

2.3 Urban environment rendering

I modelli tridimensionali degli ambienti urbani sono ben diversi da quelli visti finora. Essi, infatti, hanno la principale caratteristica di non essere *densi* (*non-dense* mesh) come una statua o un terreno, bensì sono costituiti da un elevato numero di componenti connesse, cioè gli edifici, rappresentati da un ristretto numero di poligoni (tipicamente poche decine per ciascun edificio). Inoltre, la geometria che descrive un tipico ambiente urbano è altamente discontinua, auto-occlusiva e *quasi* regolare: i palazzi che lo costituiscono sono spesso disposti in modo più o meno uniforme su una superficie grossomodo planare e spesso hanno forme molto simili. Queste regolarità nella geometria rendono la visualizzazione molto più complessa perché anche la minima approssimazione geometrica, o la piccola discontinuità nell'illuminazione, verrebbe subito notata anche da un occhio poco attento.

Esiste poi un'ulteriore complicazione, derivante dal fatto che in un ambiente urbano anche gli *spazi vuoti*, cioè le strade e le piazze, rappresentano informazione utile. Verrebbe quindi commesso un errore grossolano se si approssimasse un intero quartiere con un singolo parallelepipedo poiché tutte le strade o le piazze sarebbero eliminate. Nemmeno una rappresentazione basata sulle classiche mappe d'altezza risulterebbe adeguata, poiché non sarebbe capace di memorizzare le informazioni di colore relative ai muri verticali.

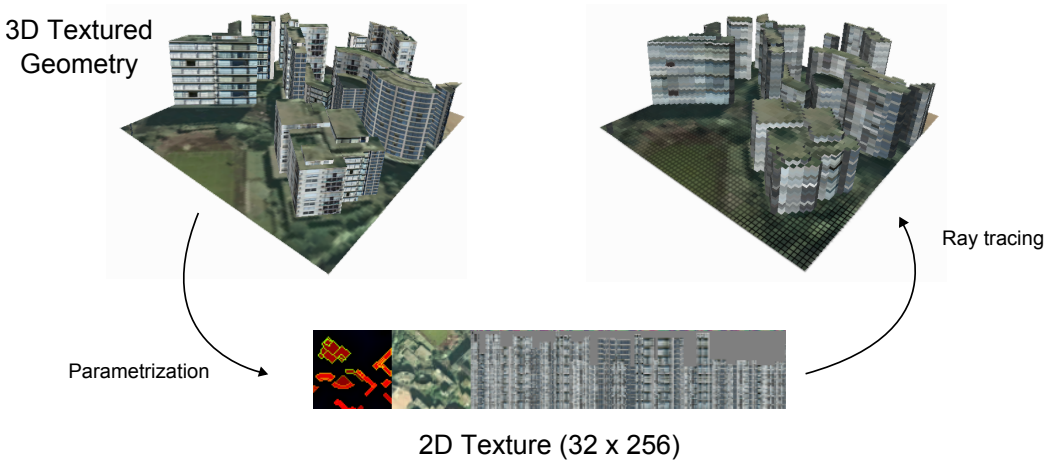
Anche le tecniche multirisoluzione viste precedentemente non sono adeguate per rappresentare ambienti urbani. Gli approcci basati su varianti del ROAM fallirebbero per due motivi: il primo è che la triangolazione al volo introdurrebbe evidenti errori di approssimazione, magari coprendo intere zone di *spazio vuoto* con grossi triangoli; il secondo è che sarebbe necessario utilizzare un'ulteriore struttura dati, collegata in modo non banale al campo d'altezze, dedicata alla memorizzazione delle texture relative ai muri verticali.

Una visualizzazione attraverso BDAM sicuramente sarebbe in grado di gestire le texture meglio del ROAM, ma la creazione delle pezze triangolari non fornirebbe i risultati sperati, poiché, il numero di poligoni per edificio è minimo già nei nodi di alto dettaglio, pertanto la creazione dei nodi a dettaglio inferiore risulterebbe poco significativa.

Per questo motivo, anche le altre tecniche basate sulla semplificazione di mesh attraverso *edge collapsing* (come Progressive Meshes [Hop96] e Tetra-Puzzle) non sarebbero in grado di gestire adeguatamente la visualizzazione di ambienti urbani complessi.

Questi aspetti possono essere parzialmente risolti con l'utilizzo di *impostor* [MS95], cioè delle immagini contenenti il rendering di un gruppo di edifici, visualizzate in sostituzione degli oggetti che rappresentano. Tali texture sono poi rimpiazzate dalla geometria originale man mano che l'osservatore si avvicina. Tuttavia, anche soluzioni di questo tipo introducono alcuni aspetti negativi sul piano pratico, come errori di parallasse in fase di visualizzazione e difficoltà nel calcolare la loro corretta illuminazione.

Buchholz e Döllner, in [BD05], hanno proposto una tecnica multirisoluzione per la gestione di modelli caratterizzati da grandi quantità di dati texture e un limitato numero di poligoni, tipici degli ambienti urbani. La tecnica consiste nell'organizzare tutta l'informazione di texture in una struttura gerarchica (un *quad tree*), costruita bottom up, in cui ad ogni nodo viene associata una versione meno dettagliata delle texture dei nodi figli. Queste sono disposte all'interno di un atlas in modo da non dover replicare le coordinate texture per ogni livello della gerarchia, così da poterle esprimerle mediante una trasformazione affine di quelle originali.



2.3.1 Le BlockMap

Un'altra recente tecnica per la visualizzazione di ambienti urbani è quella proposta in [CDBG+07], che si avvale delle potenzialità delle moderne schede grafiche. Essa è basata su un sistema multirisoluzione, nel quale gli edifici sono codificati in apposite strutture dati chiamate *BlockMap*. In esse la geometria degli edifici viene memorizzata tramite mappe d'altezze, combinate efficientemente alle informazioni di colore relative alle viste laterali. Esse verranno illustrate in modo particolarmente dettagliato, in quanto sono alla base del successivo lavoro svolto.

Una BlockMap è una texture opportunamente suddivisa in regioni distinte, una adiacente all'altra, ciascuna delle quali contiene una diversa codifica della porzione di spazio che chiameremo *dominio* della BlockMap. Per essere più precisi, una BlockMap è suddivisa in tre zone: la parte sinistra codifica informazioni differenti per ciascuno dei tre canali (RGB) dell'immagine, di cui vedremo a breve il significato; la parte centrale memorizza l'informazione colore relativa ai tetti dei palazzi infine, la parte destra dell'immagine contiene le viste laterali. Queste regioni della BlockMap prendono rispettivamente il nome di *Geometry map*, *Roofs map* e *Walls map*.

Creazione

Per trasformare la geometria degli edifici nelle regioni bidimensionali appena menzionate, viene effettuata una discretizzazione nel seguente modo. Sia dato un *dominio* non vuoto; i palazzi che esso contiene vengono dapprima ruotati in modo tale che il loro sviluppo in altezza risulti allineato con l'asse Z positivo. Successivamente, essi vengono proiettati sul piano XY contenente una griglia di $n \times n$ celle. Di seguito, per ogni cella che interseca la geometria proiettata, viene associato il corrispondente valore di Z, cioè l'altezza dell'edificio in quel punto. Tale griglia rappresenta esattamente il contenuto della *Geometry map* e risulta evidente come un valore elevato di n produrrà una discretizzazione più precisa.

A questo punto, è possibile ricostruire la geometria di partenza iterando sulla griglia ed elevando una colonna, di dimensioni $\frac{1}{n} \times \frac{1}{n}$ e altezza z_{ij}, per ogni cella $[i, j]$. Questo procedimento risulterebbe inutilmente laborioso, pertanto la visualizzazione delle BlockMap avviene attraverso un efficiente algoritmo di *raytracing* illustrato nella sezione successiva.

Come detto in precedenza, oltre alla geometria memorizzata come mappa d'altezze nel canale rosso (R) della *Geometry map*, una BlockMap contiene le informazioni di texture relative ai muri verticali e, in aggiunta, le normali alle pareti necessarie per il corretto calcolo della loro illuminazione. Infatti, ogni elemento non vuoto della mappa d'altezze conserva un riferimento univoco ad una precisa zona della *Walls map*. Tali riferimenti sono memorizzati nel canale verde (G) della *Geometry map* sotto forma di *offset* rispetto al primo elemento della *Walls map*. Infine, il rimanente canale blu (B) memorizza delle informazioni utili a velocizzare l'algoritmo di raytracing in fase di rendering. È ora facile intuire il significato della *Roofs map*, che semplicemente rappresenta le informazioni di colore del dominio visto dall'alto.

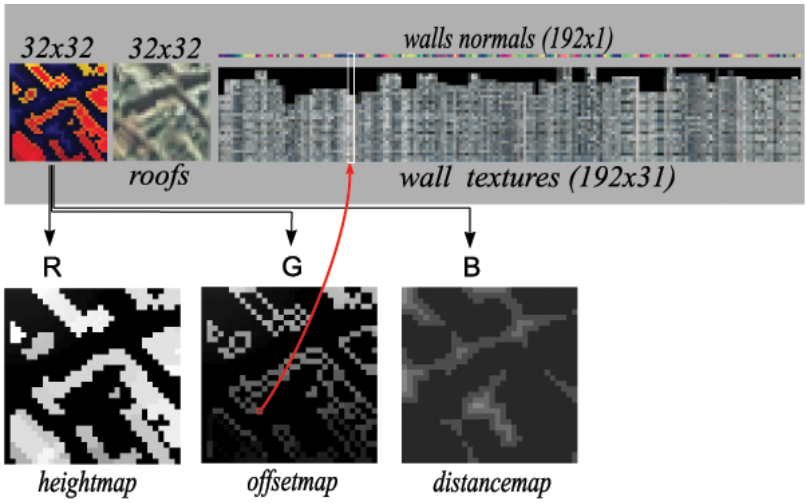


Figura 2.4: *La struttura dati* BlockMap.

Visualizzazione

La visualizzazione della singola BlockMap avviene attraverso un algoritmo di raytracing implementato in modo efficiente ed eseguito interamente dal processore grafico. Come in tutti gli algoritmi di questo genere, vengono tracciati dei raggi che attraversano una certa regione di spazio che si fermano solo quando intersecano geometria o escono da tale regione. Il colore visualizzato è quindi quello della geometria nel punto di intersezione col raggio. Nel caso delle BlockMap i raggi attraversano la *Geometry map* lungo la direzione di vista. Possono verificarsi tre tipi di intersezioni (si veda la figura 2.5), per ognuna delle quali è necessario determinare il colore e la normale da utilizzare per l'illuminazione:

- Il raggio interseca una superficie orizzontale, come un tetto o il terreno. Il punto di intersezione $p(x, y, z)$ avrà normale $n(0, 0, 1)$ mentre il colore verrà prelevato dalla *Roofs map* alle coordinate (x, y).

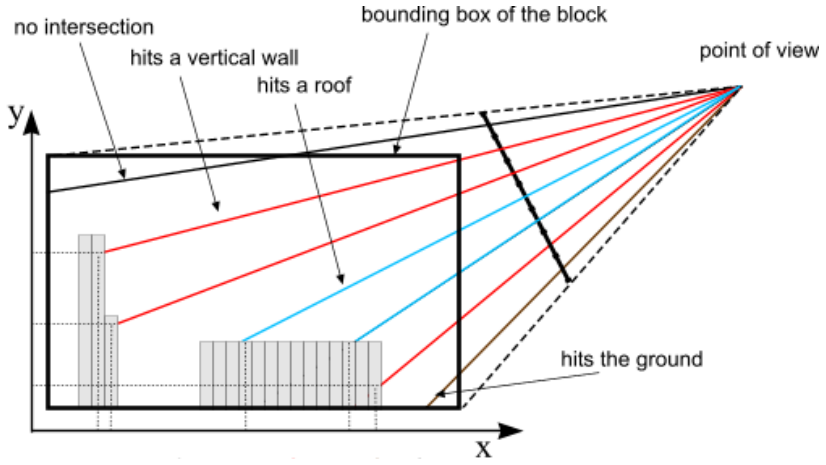


Figura 2.5: *Raytracing: possibili intersezioni.*

- Il raggio interseca una superficie verticale, cioè il muro di un edificio. Il punto di intersezione $p(x, y, z)$ avrà colore e normale entrambi prelevati dalla *Walls map*: l'offset viene letto nel canale verde della *Geometry map* alle coordinate (x, y). Questo valore rappresenta la coordinata u in spazio texture, mentre la coordinata v è data dall'altezza del raggio al momento dell'intersezione, cioè z. Infine, la normale di un muro verticale viene prelevata dall'ultima riga della *Walls map* alla posizione u.

- Il raggio esce dal dominio senza intersecare geometria.

Il cuore del sistema multirisoluzione è costituito da una struttura dati gerarchica, per la precisione un albero quaternario, i cui nodi rappresentano porzioni quadrate dello spazio da visualizzare e possono contenere geometria texturizzata o BlockMap. I primi sono quelli che risiedono nei livelli più profondi dell'albero, pertanto sono utilizzati nel rendering degli oggetti ad un maggiore livello di dettaglio. Invece, gli edifici che si trovano ad una *certa distanza* dall'osservatore, quindi con dettaglio minore, vengono visualizzati

tramite BlockMap, cioè visitando i nodi meno profondi dell'albero. La scelta di utilizzare una struttura dati ibrida è motivata principalmente dal fatto che il raytracing delle BlockMap troppo vicine, nelle macchine più lente, costerebbe più del rendering della semplice geometria.

Ciò è dovuto principalmente al fatto che il costo della visualizzazione di una BlockMap risiede nell'elevato numero di operazioni eseguite per ogni *frammento*. Man mano che l'osservatore vi si avvicina, il numero di frammenti da processare aumenta velocemente, riducendo così le prestazioni della visualizzazione.

La distanza oltre la quale risulta conveniente visualizzare le BlockMap, in sostituzione dei poligoni, è misurata sulla base dell'errore che queste produrrebbero se visualizzate su schermo; se tale errore è inferiore ad una certa soglia (proporzionale alla risoluzione della BlockMap), allora il nodo verrà visualizzato tramite raytracing, altrimenti usando la geometria dei livelli più profondi.

Considerazioni

Il punto di forza di un sistema multirisoluzione basato su BlockMap risiede sicuramente nella sua velocità di rendering. Da questo punto di vista, infatti, i risultati sono notevoli. Il problema principale, invece, si può riassumere nell'eccessiva semplicità della rappresentazione dei palazzi. Innanzitutto, non è prevista la possibilità di visualizzare degli edifici aventi tetti inclinati, questo perché ogni palazzo viene approssimato da un prisma. Questa rappresentazione può essere sufficiente per alcuni ambienti urbani moderni (come New York) o se gli edifici sono visti da molto lontano, ma non è più accettabile se i palazzi sono visti da vicino oppure se l'ambiente urbano descrive una città ricca di tetti inclinati (come la maggior parte delle citta europee).

In secondo luogo, la perdita di dettaglio dovuta alla semplicità della rap-

presentazione, riguarda anche le informazioni di colore. Infatti, lo spazio in cui vengono memorizzate le texture dei muri verticali è costante per tutte le BlockMap, indipendentemente dal dominio che esse codificano (tipicamente le *Walls map* sono ampie 192×32 pixel). Questo fa si che le regioni con pochi palazzi, quindi con poche viste laterali, abbiano a disposizione di uno spazio per le texture egualmente ampio a quello delle regioni con un elevato numero di edifici. Ciò significa che, in fase di creazione del dataset, per far rientrare tutte le viste laterali di un dominio nella rispettiva *Walls map*, potrebbe rendersi necessaria un'operazione di scalatura, la quale introdurrebbe un evidente effetto di *aliasing*. Questo inconveniente è tipico nei nodi meno profondi della gerarchia, proprio perché il loro dominio racchiude un maggior numero di palazzi.

Pertanto, un sistema multirisoluzione basato su BlockMap fornisce il miglior livello di dettaglio solo in combinazione della geometria, utilizzata per la visualizzazione degli edifici vicini.

2.3.2 Ambient-Occluded BlockMap

Esiste una versione sperimentale, riveduta e corretta delle BlockMap appena illustrate. Essa non solo è rivolta a superarne i limiti della rappresentazione (il vero punto debole delle BlockMap), ma propone inoltre delle innovazioni tali da rendere più realistica e appagante la loro visualizzazione su schermo, in modo da essere utilizzate anche per gli edifici vicini.

Creazione

La struttura dati è sostanzialmente invariata, suddivisa nelle tre regioni principali *Geometry map*, *Roofs map* e *Walls map*. Ciò che cambia è la dimensione di tali regioni: se prima erano costituite da pixel descritti da soli tre canali (RGB - 24bit per pixel), in questa versione viene utilizzato un ulte-

riore canale, il canale Alpha, per memorizzare informazioni utili. Il ruolo di questo canale è duplice: laddove si memorizza informazione colore, quindi nella *Roofs map* e nella *Walls map*, esso conterrà un coefficiente di occlusione ambientale per ciascun pixel. Nella *Geometry map* invece, il quarto canale viene utilizzato per memorizzare, in ogni pixel, la componente n_y del vettore $n(n_x, n_y, n_z)$ corrispondente alla normale della superficie del tetto. La componente n_x invece, viene memorizzata nel canale blu, in sostituzione della *Distance map* utilizzata precedentemente per velocizzare l'algoritmo di raytracing. La terza componente, n_z, verrà determinata facilmente in fase di raytracing. Avendo a disposizione le normali dei tetti, risulta quindi possibile utilizzarle correttamente in fase di visualizzazione, senza assumere, come in precedenza, che siano tutte pari a $n_{flat}(0, 0, 1)$.

Per quanto riguarda la *Walls map*, essa è realizzata in modo tale da consentire una parametrizzazione adattativa delle texture relative ai muri verticali. Per essere più precisi, nel caso in cui il numero di viste laterali ecceda la dimensione della *Walls map* (ad esempio 192×32 pixel), l'intera BlockMap viene suddivisa in due porzioni orizzontali di pari altezza, ognuna delle quali dispone di una parametrizzazione indipendente per le viste laterali. Tali porzioni verranno suddivise in modo ricorsivo fintanto che la loro parametrizzazione eccederà le dimensioni della rispettiva *Walls map*. In poche parole, una BlockMap di dimensioni 32×256 pixel verrà divisa in n righe, ognuna delle quali sarà dotata di una *Walls map* indipendente di dimensioni $\frac{32}{n} \times 192$.

Visualizzazione

Avendo eliminato la *Distance map*, si intuisce che anche l'algoritmo di raytracing sarà sostanzialmente differente. In questa versione, i raggi attraversano una griglia regolare secondo una variante semplificata dell'algoritmo [AW87].

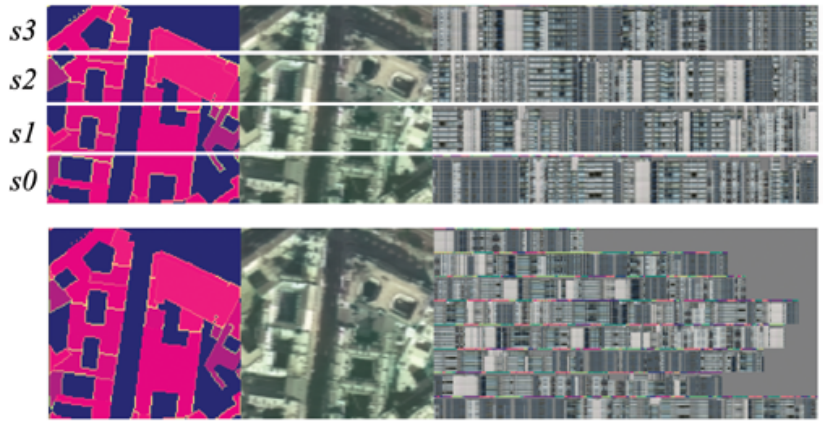


Figura 2.6: *La* Walls map *viene suddivisa in slices orizzontali.*

Inoltre, per essere in grado di distinguere l'intersezione con un tetto o con una parete verticale, per ogni raggio si calcola la derivata parziale dell'altezza lungo la rispettiva proiezione sul terreno. In questo modo, quando si interseca un muro verticale nella height map, si confrontata l'altezza attuale del raggio con l'altezza corrispondente al pixel precedente lungo la direzione del raggio stesso. Se tale differenza risulta minore di una certa soglia, il pixel sarà interpretato come un tetto, altrimenti come parete.

L'ultima modifica riguarda l'applicazione delle texture sui muri verticali. Dal momento che la *Walls map* può essere suddivisa in più strisce orizzontali, sarà quindi necessario eseguire una scalatura e una traslazione alle coordinate dell'intersezione per accedere alla posizione corretta della *Walls map*.

Grazie a questi accorgimenti, è possibile sfruttare le *Ambient-Occluded BlockMap* anche per la visualizzazione di edifici che si trovano vicini all'osservatore. Questo permette inoltre di realizzare un sistema multirisoluzione costituito interamente da soli nodi di BlockMap, evitando quindi l'uso di una struttura dati ibrida come nella precedente versione.

Considerazioni

Grazie alla potenza di calcolo delle moderne schede grafiche, è possibile visualizzare gli edifici vicini all'osservatore tramite raytracing di BlockMap, senza andare a peggiorare la velocità di visualizzazione. Pertanto, un sistema multirisoluzione basato su *Ambient-Occluded BlockMap* riesce nel duplice intento di fornire una visualizzazione interattiva che risulti appagante dal lato puramente estetico, ma capace anche di mantenere elevate le prestazioni di velocità.

2.4 Streaming di città

Gli algoritmi di appena illustrati rappresentano lo stato dell'arte per quanto riguarda la visualizzazione in tempo reale di ambienti urbani. Come abbiamo visto, il loro scopo è quello di creare una parametrizzazione degli edifici che possa essere facilmente visualizzata in sostituzione della geometria originale. Grazie alla compattezza di tale rappresentazione, la loro visualizzazione risulta particolarmente efficiente e ben si adatta ad essere eseguita interamente dal processore grafico. Inoltre, la regolarità della struttura dati BlockMap permette di implementare efficienti strategie di *caching* e di gestione della memoria. Questo si traduce in un'agevolata trasmissione su rete, che permette quindi la realizzazione di architetture *client-server* particolarmente efficienti [CDBG+07].

Capitolo 3

Volumetric BlockMaps

In questo capitolo verrà presentata un'estensione al caso tridimensionale delle BlockMap.Questa nuova versione introduce una differente rappresentazione della geometria basandosi su una variante dei campi d'altezza. Le BlockMap originali, infatti, soffrono dell'impossibilità di rappresentare oggetti sospesi o sovrapposti. Tale limitazione è intrinseca di tutte le rappresentazioni basate su campi d'altezza; in essi, infatti, per ogni punto (x, y) della mappa viene associato un unico valore di altezza z. Ne segue che, con questo tipo di parametrizzazione, non è possibile rappresentare ambienti urbani in cui sono presenti elementi architettonici sopraelevati o sovrapposti quali ponti, portici, balconi, ecc...

L'obiettivo principale di questa tesi è stato, quindi, quello di studiare e implementare una strategia di campionamento e visualizzazione differente da quello delle BlockMap originali, capace di rappresentare anche gli oggetti sospesi o, in generale, quelli con molteplici valori d'altezza. Inoltre, l'intero processo di creazione e visualizzazione è stato realizzato in modo da ridurre l'effetto di *aliasing* che caratterizzava le BlockMap delle precedenti versioni.

Le fasi principali che portano alla creazione di una BlockMap sono due: campionamento e parametrizzazione. Durante la prima fase (trattata in 3.4),

la geometria degli edifici viene *immersa* in una griglia regolare tridimensionale e convertita in una rappresentazione *raster*, cioè costituita da celle cubiche chiamate *voxel*. Nella fase di parametrizzazione (trattata in 3.3), le informazioni per il rendering di questo volume vengono opportunamente codificate all'interno dell'immagine bidimensionale chiamata *Volumetric BlockMap* (in seguito V-Blockmap). Per una migliore comprensione delle scelte fatte in fase di campionamento, verrà illustrata per prima la parte di parametrizzazione.

Data la differente rappresentazione della geometria, si è reso necessario implementare una nuova procedura di ray-tracing da utilizzare nella fase di visualizzazione. Essa è stata scritta su misura e implementata in modo tale da essere eseguita interamente dalla scheda grafica (trattata in 3.5).

3.1 Oltre i campi d'altezza

Per superare i limiti delle BlockMap originali si è arricchita l'espressività della loro codifica. A tal proposito, si è cercato il giusto compromesso tra semplicità ed efficacia di rappresentazione che rendesse possibile la codifica di elementi architettonici quali balconi, portici, ponti, ecc....

Per fare questo, si è assunto che la geometria degli edifici fosse espressa in forma *raster*, costituita, cioè, da un insieme di celle cubiche. Si supponga, inizialmente, di voler rappresentare, nel miglior modo possibile, le informazioni relative ad una singola *colonna* appartenete a tale rappresentazione. Il modo più fedele è, senza dubbio, quello di memorizzare, o indicizzare, le caratteristiche di ogni cella non vuota ad essa appartenenti. Questa rappresentazione risulta, tuttavia, troppo costosa e poco compatta.

Una rappresentazione basata sulle classiche mappe d'altezza, invece, sarebbe fin troppo scarna. In esse, infatti, l'intera colonna verrebbe convertita e compressa in un unico valore che ne rappresenta la sua altezza. È come se, di una colonna, si memorizzasse solo la cella non vuota situata nella posizione

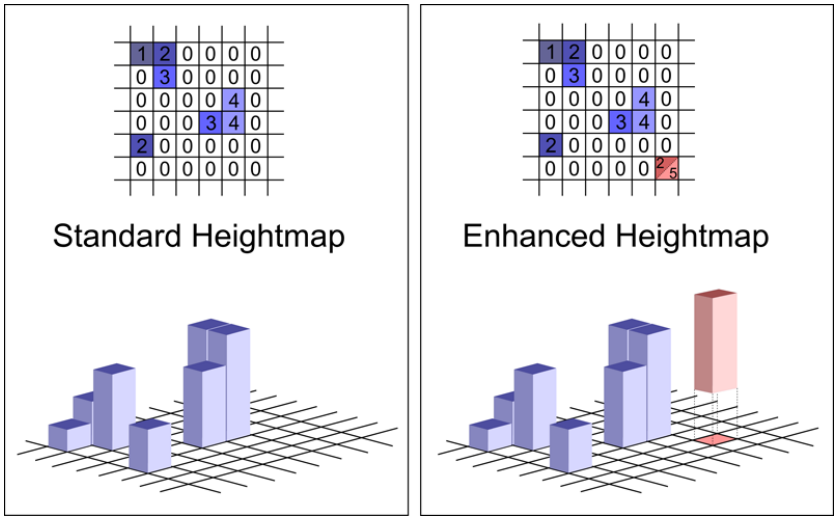


Figura 3.1: *Standard Heightmap: ad ogni elemento della mappa corrisponde un singolo valore di altezza. Enhanced Heightmap: ad ogni elemento della mappa corrispondono due valori di altezza.*

più alta, perdendo così le informazioni relative alle celle sottostanti. Per riuscire a rappresentare gli elementi architettonici sopraelevati si è applicato lo stesso ragionamento, ma da un altro punto di vista. Infatti, se la colonna in esame venisse osservata da *sotto*, si potrebbero ottenere e comprimere anche le informazioni relative al voxel non vuoto situato più in basso.

A questo punto, rimarrebbero da gestire le celle che si trovano nel *mezzo* della colonna in esame. Infatti, può accadere che questa sia costituita in parte da celle vuote. L'introduzione di ulteriori valori le altezze intermedie si tradurrebbe in un'indicizzazione completa del dataset volumetrico e, pertanto, risulterebbe inadeguata.

Un buon compromesso è quello di indicizzare due soli elementi per colonna, il più alto e il più basso, e memorizzare le eventuali celle intermedie vuote, sotto forma di *geometria trasparente*. In questo modo è possibile ottenere una rappresentazione più fedele al dataset originale, pur mantenendo

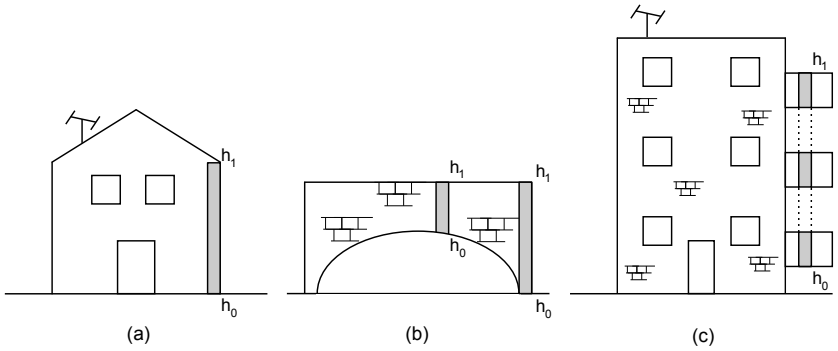


Figura 3.2: *Con soli due valori di altezza è possibile rappresentare numerosi elementi architettonici differenti: (a) un edificio che giace sul terreno; (b) un ponte sopraelevato; (c) una serie di balconi sovrapposti. Lo spazio vuoto viene codificato come geometria trasparente.*

la compattezza delle mappe d'altezza (si veda la figura 3.1).

La nuova codifica risulta, quindi, più espressiva, in quanto permette la rappresentazione di oggetti sopraelevati e sovrapposti. Tutto ciò al costo di dover memorizzare un solo valore addizionale per ogni elemento della mappa. Tale scelta, tuttavia, ben si adatta alla rappresentazione di ambienti urbani, ma, in altri tipi di ambienti risulterebbe una scelta del tutto arbitraria.

Si è passati, dunque, da una rappresentazione poco versatile, i campi d'altezza appunto, ad una rappresentazione specifica per ambienti urbani, aggiungendo solo un ulteriore valore per ciascun elemento della mappa.

Per comprendere meglio l'uso del nuovo valore di altezza si faccia riferimento alla figura 3.2: nel caso **(a)** il valore di altezza minima è pari a zero, ciò significa che l'edificio in esame giace sul terreno; nel caso di un ponte **(b)**, invece, il valore di altezza minima nei punti sopraelevati sarà diverso da zero.

Il caso **(c)**, infine, raffigura una colonna costituita parzialmente da elementi vuoti. Questi verranno convertiti in geometria trasparente all'interno della V-BlockMap.

3.1.1 Rappresentazione mediante colonne

Prima di addentrarsi nei dettagli della realizzazione, è opportuno illustrare i tipi di informazioni che andremo a trattare. Come anticipato brevemente, l'ambiente urbano viene trasformato in una sua rappresentazione volumetrica, costituita dunque da voxel e non più poligoni (figura 3.3). Ognuno di questi voxel contiene tutti gli attributi necessari alla visualizzazione approssimata del dato originale.

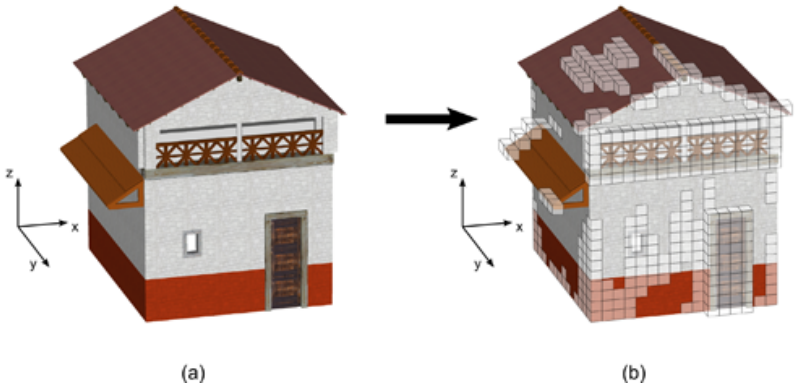


Figura 3.3: *(a) Rappresentazione geometrica. (b) Rappresentazione volumetrica (parziale).*

Successivamente, questi voxel contribuiranno alla creazione della V-BlockMap attraverso il processo di parametrizzazione. Esso consiste nell'*appiattire* la rappresentazione volumetrica scartando le informazioni relative alle altezze dei voxel. Tuttavia, per essere in grado di ricreare l'aspetto della geometria originale, queste informazioni non possono essere completamente eliminate, pertanto, verranno espresse in modo differente, più compatto.

Come si vede nella parte **(a)** dell'immagine 3.4, il processo di parametrizzazione genera un'immagine bidimensionale corrispondente alla vista dall'alto dell'ambiente urbano. In essa, ciascun elemento (x, y) viene creato con il contributo di tutti quei voxel di posizione (x, y, z_i), con $i = 0, 1, 2...Z_{max}$.

In altre parole, ad ogni pixel di una V-BlockMap corrisponde una precisa *colonna di voxel*, estratta dalla relativa rappresentazione volumetrica.

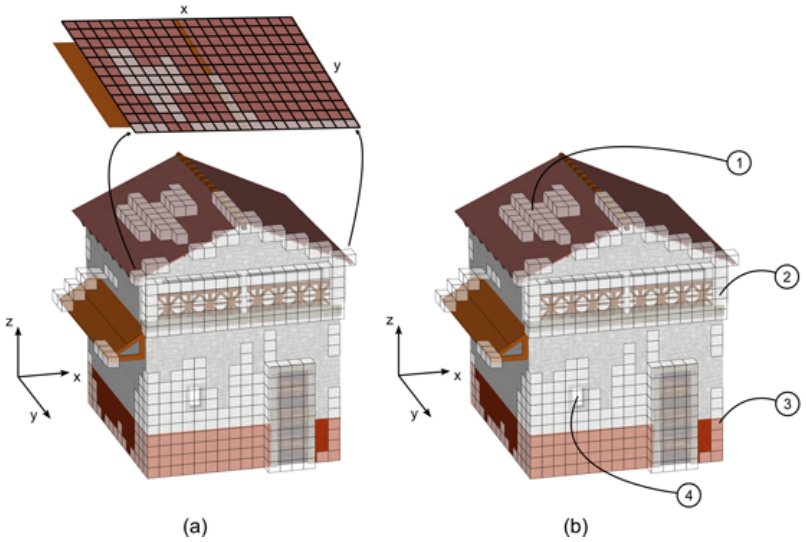


Figura 3.4: *(a) Esempio di parametrizzazione (parziale). (b) 1.Colonna non esposta. 2.Colonna sospesa. 3.Colonna esposta. 4.Colonna con elementi trasparenti.*

Queste colonne sono il concetto fondamentale attorno al quale è stato realizzato l'intero *framework*. Gli attributi di una colonna dipendono dal tipo di voxel che essa contiene, pertanto, possiamo distinguere quattro tipi fondamentali di colonne, sintetizzati nella parte **(b)** in figura 3.4: le colonne che risultano caratterizzate da un solo voxel visibile, posizionato in cima prendono il nome di *colonne non esposte* o *tetti*. Questo tipo di colonne è contrassegnato dal numero **1** in figura. Le colonne di tipo **2** e **3** sono praticamente identiche tra loro; in entrambi i tipi, infatti, il voxel in cima non è l'unico visibile, ma lo sono anche quelli sottostanti. Questo tipo di colonne prende il nome di *colonne esposte* o *pareti*; le pareti di tipo **2** differiscono da quelle di tipo **3** per il semplice fatto che le prime risultano sopraelevate, mentre, le seconde sono adagiate sul terreno. Infine, le colonne di tipo **4** sono

anch'esse di tipo *parete*, ma alcuni loro voxel sono vuoti, cioè non contengono informazioni né geometriche né di colore. Questi voxel verranno quindi codificati come trasparenti.

Esiste, inoltre, un altra tipologia di voxel, la quale non contribuisce direttamente alla generazione della V-BlockMap. Questi sono i voxel che si trovano *all'interno* della rappresentazione volumetrica e dunque non risultano mai visibili. I voxel e le colonne di questo tipo prendono rispettivamente i nomi di *voxel interni* e *colonne interne*.

3.1.2 Gli attributi di un voxel

Come detto precedentemente, i voxel della rappresentazione volumetrica contengono tutti gli attributi utili alla visualizzazione approssimata della geometria originale. Distingueremo tali attributi in due categorie: *attributi geometrici* e *attributi di shading*. Il primo gruppo consente di ricreare le forme degli edifici, mentre il secondo rappresenta quell'insieme di informazioni che contribuiscono alla corretta colorazione della geometria, ovvero, colore, vettore normale alla superficie, fattore di occlusione ambientale e trasparenza.

Vediamoli in dettaglio:

- **Colore**
  È il classico attributo in una rappresentazione volumetrica: ogni voxel memorizza un colore unico per la geometria approssimata.

- **Normale**
  Con questo termine si indica il vettore ortogonale alla superficie approssimata da un voxel. Ad esempio, si supponga di esprimere un parallelepipedo in forma volumetrica. Per ogni sua faccia, i voxel che le attraversano avranno tutti la stessa normale. I voxel di *spigolo*, invece, avranno un vettore normale calcolato come media delle normali delle facce che incidono su di esso.

- **Occlusione ambientale**

  Per migliorare il realismo della visualizzazione, ogni voxel memorizzerà il relativo fattore di occlusione ambientale. Questo fattore è calcolato come la percentuale delle direzioni dalle quali può giungere la luce sul punto di una superficie. In altre parole, serve per indicare da quante direzioni tale punto risulta illuminato. Si faccia riferimento alla figura 3.5: la zona compresa tra la casetta e il palazzo risulterà sempre in penombra, quindi più scura, perché la luce non può giungere da tutte le dirazioni in quanto occlusa dai due edifici.

- **Trasparenza**

  Attraverso questo coefficiente, saremo in grado di rappresentare la geometria trasparente indispensabile nella visualizzazione delle colonne di tipo **4**, viste nella sezione precedente. Inoltre, tale valore di trasparenza potrà essere sfruttato per implementare tecniche di antialiasing.

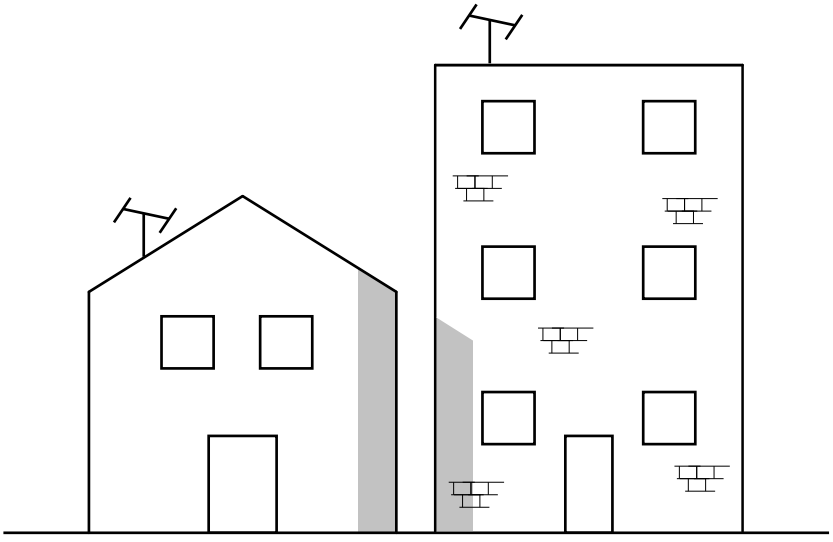


Figura 3.5: *Occlusione ambientale: nelle zone in cui la luce risulta parzialmente occlusa dagli edifici, la geometria risulterà più scura.*

3.2 La struttura dati

Le informazioni appena illustrate, memorizzate nei voxel della rappresentazione volumetrica, verranno successivamente trasformate in pixel in modo tale da essere memorizzate all'interno dell'immagine bidimensionale detta V-BlockMap. Essa è costituita da tre parti principali, ciascuna dedicata alla memorizzazione di uno specifico tipo di dato.

Le informazioni di tipo geometrico, come la forma e la disposizione dei palazzi, verranno memorizzate nella regione che chiameremo *Geometry map*. Le informazioni di *shading*, invece, verranno scomposte e memorizzate nelle restanti due regioni, la *Roofs map* e la *Walls map*. Nella prima troveranno luogo gli attributi di *shading* relativi alle colonne di tipo *tetto*; nella seconda vi saranno quelli relativi alle colonne di tipo *parete*.

3.2.1 La Geometry Map

Questa regione della V-BlockMap non è altro che l'implementazione dell'*enhanced heightmap* visto in 3.1. Pertanto, ogni suo elemento (pixel) conterrà due valori di altezza che chiameremo Z_{max} e Z_{min}. Durante la fase di visualizzazione, questi elementi verranno estrusi in modo da ricreare il profilo della geometria originale.

Rispetto alle precedenti versioni delle BlockMap, è stata introdotta un'ulteriore modifica. Essa nasce da un'innovazione riguardante la *Walls map* (che vedremo a breve), cioè dalla scelta di memorizzare, per ogni colonna di tipo *parete*, un numero di pixel proporzionale al numero di voxel che la costituiscono. Infatti, nelle precedenti versioni il numero di pixel delle viste laterali era uguale per tutte le colonne di una stessa BlockMap; in questo modo l'accesso ad una vista laterale poteva avvenire attraverso la sola coordinata u poiché la coordinata v si poteva ricavare implicitamente dall'algoritmo di

ray-tracing. Questa facilità di accesso veniva tuttavia scontata con un livello di dettaglio non adattativo, a discapito delle colonne più alte.

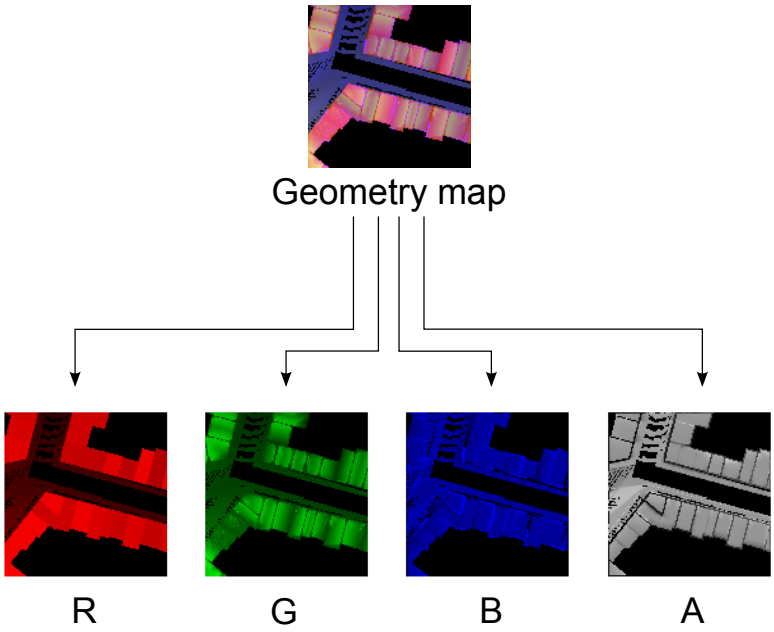


Figura 3.6: *Ciascun canale della* Geometry map *memorizza un'informazione differente: (R) Altezza massima; (B) Altezza minima; (G) Coordinata U / Normale n_x; (A) Coordinata V / Normale n_y.*

In questa nuova versione, si è scelto di dedicare alle viste laterali un numero di pixel proporzionale alla dimensione della rispettiva colonna, così da ottenere un sampling isotropico del dato. Questa scelta comporta, dunque, la modifica dell'intera *Geometry map*, in quanto si è reso necessario memorizzare esplicitamente entrambe le coordinate di texture (u, v), laddove prima ne veniva memorizzata una sola.

Trattandosi di un'immagine, ciascuno dei pixel che compone la *Geometry map* è costituito dai quattro canali standard *RGBA*. Sfrutteremo questa suddivisione a nostro vantaggio, memorizzando informazioni differenti per ciascun canale (si faccia riferimento alla figura 3.6):

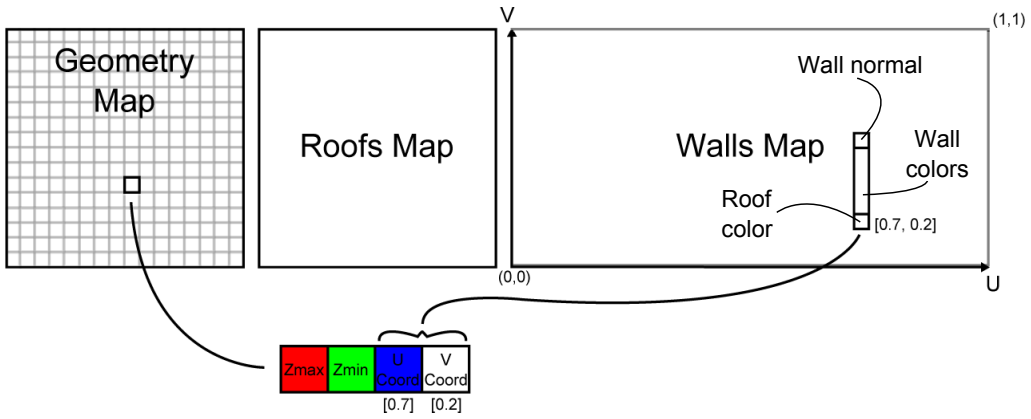


Figura 3.7: *Mapping delle colonne esposte.*

Il canale rosso (R) memorizza i valori Z_{max} per ciascuna colonna, così come il canale verde (G) ne memorizza i corrispondenti valori Z_{min}. I canali blu (B) e alfa (A) hanno invece un duplice significato, a seconda che si stia codificando una colonna di tipo *parete* o di tipo *tetto*. Le prime, infatti, producono la vista laterale che andrà memorizzata nella *Walls map*; accederemo a questa vista attraverso le relative coordinate (u, v) memorizzate rispettivamente nei canali blu e alfa della *Geometry map* (figura 3.7).

Le colonne di tipo tetto, invece, non producono viste laterali. Pertanto, i canali B e A verranno utilizzati per memorizzare le componenti del vettore normale $n_{roof}(x, y, z)$, ortogonale alla superficie del tetto.

La tabella 3.1 riassume il significato *polimorfo* dei pixel memorizzati *Geometry map*.

3.2.2 Roofs Map

In questa regione sono memorizzate le informazioni di colore relative ai tetti, come se gli edifici fossero catturati in una foto dall'alto. La struttura *a griglia* della *Geometry map* è pressente anche in questa regione: ogni suo pixel

	Colonne *parete*	**Colonne *tetto***
Canale R	Altezza massima	Altezza massima
Canale G	Altezza minima	Altezza minima
Canale B	Coordinata texture (u)	Normale tetto (componente n_x)
Canale A	Coordinata texture (v)	Normale tetto (componente n_y)

Tabella 3.1: *Significato dei pixel nella* Geometry map

rappresenta il colore delle colonne viste dall'alto. Tuttavia, poiché quelle di tipo *parete* utilizzano i quattro canali della *Geometry map* esclusivamente per la memorizzazione di geometria e coordinate texture (come in figura 3.7), è necessario trovare dello spazio addizionale per memorizzare i due vettori ad esse ortogonali, cioè n_{wall}, ortogonale alla parete verticale, e n_{roof}, ortogonale al tetto. Vedremo a breve come il vettore n_{wall} trovi una facile collocazione nella *Walls map*, pertanto ci focalizzeremo solo sulla memorizzazione del vettore n_{roof}.

Per le colonne esposte, cioè quelle di tipo *parete*, si è deciso di spostare l'informazione di colore, relativa alla vista dall'alto, all'interno della *Walls map*, che sarebbe servita alla memorizzazione delle sole viste laterali. Questo spostamento permette di accedere al colore del tetto in modo meno rapido ma consente di memorizzare le componenti del vettore n_{roof} all'interno della *Roofs map*. Per la precisione, verranno utilizzati i canali *GBA*, mentre il canale R verrà sfruttato per discriminare tali colonne in fase di visualizzazione attraverso un'opportuna codifica.

3.2.3 Walls Map

Questa regione contiene sia le informazioni del colore sia informazioni delle normali riguardanti esclusivamente le colonne di tipo *parete*. La sua struttura

prevede che le colonne vengano disposte una accanto all'altra, codificate secondo la figura 3.8: il primo pixel, quello alla base, rappresenta il colore della colonna vista da sopra; proseguendo verso l'alto, si trovano i pixel prodotti dalla vista laterale; infine, l'ultimo pixel memorizza il vettore n_{wall} normale alla parete, calcolato come media di tutti i vettori normali che incidono su di essa.

Qualora una colonna non possa essere disposta accanto alle precedenti per ragioni di spazio, essa verrà posizionata al di sopra di una delle precedenti, mantenendo invariata la propria codifica.

Come detto precedentemente, il numero di pixel dedicati alle viste laterali di ciascuna colonna sarà proporzionale alla dimensione della colonna stessa. In questa regione che vengono codificate anche le colonne di tipo **4**, ovvero, quelle che contengono uno più voxel vuoti. Questo tipo di voxel verrà tradotto in un insieme di pixel, la cui codifica li renderà trasparenti in fase di visualizzazione.

3.2.4 Vincoli della rappresentazione

Come detto precedentemente, ogni pixel della V-BlockMap è costituito dai quattro canali *RGBA*. La dimensione di ciascun canale è fissata a 8 bit, questo significa che tutte le informazioni utili a descrivere una colonna, devono prima essere mappate nell'intervallo chiuso $(0, 255)$ e poi salvate nell'apposito canale. Ciò comporta sicuramente una perdita di precisione che, tuttavia, diventa trascurabile quando gli edifici sono visualizzati ad una certa distanza.

Tale limitazione, invece, si fa molto più stringente per quanto riguarda la memorizzazione delle coordinate di texture (u, v). Infatti, avendo a disposizione soli 256 valori per canale, e quindi per coordinata, le dimensioni della *Walls map* non possono essere maggiori di 256×256 pixel. Pertanto, qualora le dimensioni del dataset siano tali che la parametrizzazione ecceda

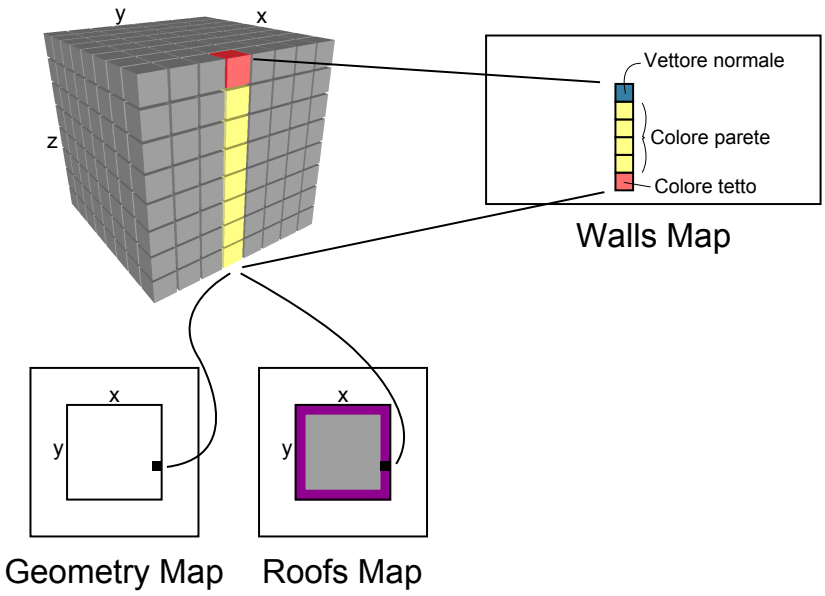


Figura 3.8: *Parametrizzazione di una colonna di tipo* parete. *Nell'immagine questa tipologia di colonne è indicata con il colore viola all'interno della* Roofs Map. *Nella* Walls Map *vengono memorizzate, dal basso verso l'alto, le seguenti informazioni: colore del tetto, colore parete e vettore normale alla parete.*

le dimensioni massime della *Walls map*, sarà allora necessario ricorrere ad opportune operazioni di scalatura (si veda la sezione3.3).

La seguente tabella riassume i modi in cui le informazioni sulle colonne vengono scomposte nella struttura dati. Per ciascun tipo di colonna è indicata la regione della V-BlockMap in cui è memorizzato il dato corrispondente:

Per i pixel che memorizzano informazioni colore, sia nella *Roofs map* sia nella *Walls map*, è necessario dedicare i tre canali *RGB* per tale scopo. Il quarto canale A è invece dedicato alla memorizzazione del coefficiente di trasparenza, pertanto, l'unico modo per memorizzare il fattore di occlusione ambientale rimane quello di premoltiplicarlo al valore di colore di ciascun

	Colonne *esposte*	**Colonne *non esposte***
Dimensioni	Geometry map (canali R e G)	Geometry map (canali R e G)
Coordinate Walls map (u, v)	Geometry map (canali B e A)	-
Colore (laterale)	Walls map	-
Colore (dall'alto)	Walls map (base pixel)	Roofs map
Normale (laterale)	Walls map (top pixel)	-
Normale (dall'alto)	Roofs map	Geometry map (canali B e A)

canale. Con questa operazione, tale fattore diventa a tutti gli effetti un attributo statico del rispettivo voxel, precalcolato in fase di creazione del dataset.

3.3 Parametrizzazione del volume

Il procedimento di creazione di ogni singola V-BlockMap assume di avere in ingresso la rappresentazione *volumetrica* della geometria originale. La creazione di tale dataset verrà illustrata nel dettaglio in 3.4. Al momento, è sufficiente immaginare che esso sia costituito da una griglia regolare di $n \times n \times n$ cubetti, chiamati *voxel*.

La parametrizzazione di questo volume produrrà la relativa V-BlockMap, la cui risoluzione dipenderà direttamente dalle dimensioni del volume di partenza. Prima di cominciare, è però necessario definire alcune regole per l'identificazione dei voxel. Indicheremo come *pieni* quei voxel in cui possiamo trovare il colore della geometria originale da loro approssimata; in caso contrario, essi saranno identificati come voxel *vuoti*. Vi è un caso particolare di voxel pieni, ovvero quelli che si trovano all'interno del volume. Essi

Tipo Voxel	Descrizione
Vuoto	Non contiene nessuna informazione
Pieno	Non Vuoto
Interno	Geometria e *shading* inconsistenti
Visibile	Non Interno e almeno un vicino Vuoto

Tabella 3.2: *Il volume è costituito da due tipi principali di voxel:* pieni *e* vuoti*. Tra i quelli pieni si distinguono quelli non visibili (*interni*) e quelli* visibili*.*

non contengono informazioni né geometriche né di shading, pertanto, non contribuiranno in alcun modo alla creazione della V-BlockMap.

Infine, un voxel si dice *visibile* se non è *interno* e almeno uno dei suoi 26 vicini è *vuoto*, cioè risulta visibile anche parzialmente.

Sulla base di queste semplici regole, riassunte nella tabella seguente, è possibile iterare sul volume per creare le colonne di voxel che verranno parametrizzate nelle apposite regioni della V-BlockMap.

Prima di procedere con la parametrizzazione del volume, è necessario un passo iniziale durante il quale il volume verrà attraversato alla ricerca di quei voxel che contengono informazioni utili.

Attraversamento

L'attraversamento avviene per colonne verticali, cioè lungo l'asse Z. Si parte dalla prima colonna, quella di posizione $(0, 0)$, e si analizzano i voxel uno per uno a partire da quello più alto. Non appena si trova un voxel pieno, la sua coordinata Z viene salvata, in una struttura dati d'appoggio, come valore Z_{max} per tale colonna. Successivamente, si prosegue con la ricerca del valore Z_{min}, ciò significa che la colonna viene analizzata allo stesso modo,

ma a partire dal voxel più basso.

Al termine di questi passi, se Z_{max} risulta pari a zero, allora l'intera colonna in esame viene scartata in quanto vuota; in caso contrario, vengono contati quanti dei suoi voxel sono visibili. In base a tale numero, viene quindi stabilito se essa potrà essere parametrizzata nella sola vista dall'alto (colonna non esposta, un solo voxel è risultato visibile) o anche nella vista laterale (colonna esposta).

L'attraversamento riprende quindi con la successiva colonna, quella di posizione $(1, 0)$, fino ad arrivare all'ultima in posizione $(n-1, n-1)$ (si faccia riferimento alla figura 3.9 per una migliore comprensione).

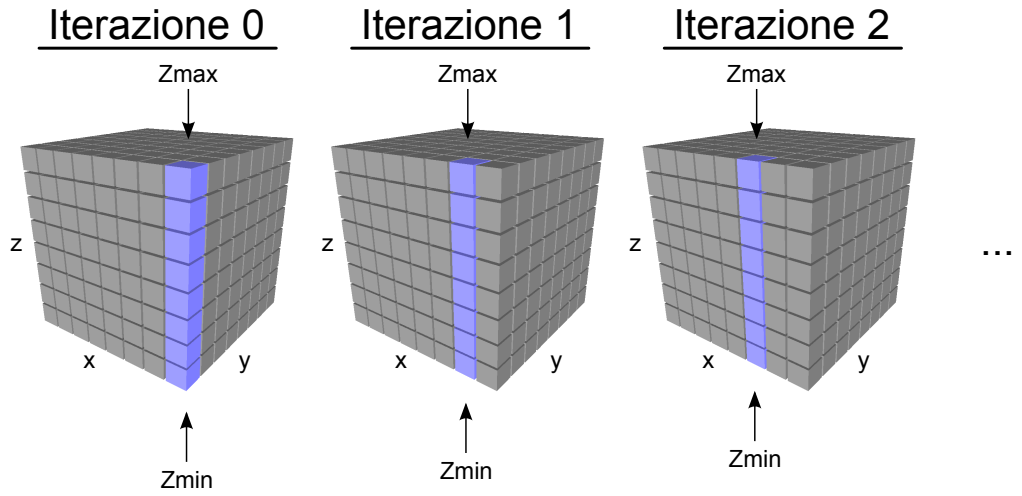


Figura 3.9: *Schema di attraversamento del volume. Per ogni colonna, i valori Z_{max} e Z_{min} sono individuati seguendo la corrispondente direzione delle frecce.*

Al fine di rappresentare correttamente le superfici inclinate, come dei tetti spioventi ad esempio, l'intero volume viene analizzato per individuare e codificare questo tipo di informazione. Nelle BlockMap precedenti, infatti, questa caratteristica veniva calcolata in tempo reale, durante la fase di ray-tracing. Il nostro intento è quello di identificare tali superfici in fase di

creazione e trasformare questa informazione in un attributo statico per le colonne.

Per individuare le superfici inclinate, ciascuna colonna non vuota viene confrontata con le otto colonne adiacenti, calcolando le differenze di altezza con la colonna corrente. Se almeno una di queste differenze risulta maggiore di una certa soglia, allora si assume che la colonna in esame faccia parte di una superficie inclinata. Questa informazione rappresenta un attributo addizionale per le colonne, pertanto, la loro parametrizzazione non avverrà in modo differente. Ciò che sarà diverso è la loro codifica nella *Geometry map*, che risulterà tale da essere riconosciuta e decodificata in fase di ray-tracing.

Parametrizzazione

A questo punto, la struttura dati d'appoggio contiene tutte le colonne che devono essere parametrizzate. Il loro numero dipende dall'aspetto degli edifici e dalle dimensioni del dataset volumetrico; ad ogni modo, non potranno mai essere più di $n \times n$.

Per prima viene creata la *Geometry map*: a ciascuna colonna del volume, in posizione $v(x, y)$ con $x, y = 0, 1, ..., n - 1$, corrisponderà un singolo pixel nella mappa in posizione $p(x, y)$. In esso vengono scritti i valori Z_{max} e Z_{min} rispettivamente nei canali R e G dopo la necessaria conversione in 8 bit. Per ogni colonna, inoltre, deve essere salvato il colore del voxel di posizione Z_{max}. La regione in cui salvare tale informazione dipende, come abbiamo visto, dal tipo di colonna:

- **Colonna non esposta (tetto)**
  Se si tratta di una colonna non esposta, i canali B e A della *Geometry map* verranno utilizzati per memorizzare le componenti del vettore normale alla superficie del tetto. Essendo espresse nell'intervallo $(-1, 1)$, dovranno prima essere mappate nell'intervallo $(0, 255)$.

La *Roofs map* si trova, di fatto, adiacente alla *Geometry map*, quindi è sufficiente traslare le coordinate della colonna $p(x, y)$ di un valore pari al lato della *Geometry map* per scrivere le informazioni di colore relative al voxel Z_{max} alle coordinate corrette $p(x + n, y)$. Come anticipato, il colore del voxel verra premoltiplicato per il fattore di occlusione ambientale prima di essere scritto.

La scrittura di questi pixel avviene durante la creazione della *Geometry map*.

- **Colonna esposta (parete)**
  Trattandosi di una colonna esposta, i canali B e A della *Geometry map* verranno utilizzati per memorizzare le coordinate (u, v), pertanto, le componenti del vettore normale al tetto verranno salvate nei tre canali *GBA* della *Roofs map* alle coordinate $p(x + n, y)$, ovviamente dopo la mappatura nell'intervallo $(0, 255)$. Il canale R di tale pixel assumerà, invece, una particolare configurazione, facilmente riconoscibile in fase di visualizzazione, in modo tale da trattare correttamente le colonne esposte e le colonne non esposte. La scrittura di queste ultime avviene in un secondo momento, precisamente durante la creazione della *Walls map*.

La scelta di utilizzare un particolare valore di rosso per codificare delle informazioni che non siano colore, comporta il fatto che *qualunque* pixel della *Roofs map* debba rispettare tale codifica. Pertanto, si avrà l'effetto di rappresentare il colore delle viste dall'alto con una tonalità di rosso *in meno* rispetto al colore originale. Per questioni di praticità, la tonalità di rosso scelta per distinguere le colonne esposte da quelle non esposte, è quella corrispondente al valore 255, cioè il massimo valore di rosso rappresentabile.

Il passo finale consiste nel posizionare le colonne esposte all'interno della *Walls map* e quindi, salvare nella *Geometry map* le coordinate che permet-

teranno di recuperare le informazioni appena scritte. Le parametrizzazioni appena illustrate, consistono nella semplice mappatura di $n \times n$ elementi (voxel) in due aree uguali e adiacenti, costituite entrambe da altrettanti elementi (pixel). La *Walls map*, invece, è di dimensioni prefissate, $256 \times n$, pertanto la mappatura non risulterà altrettanto banale.

Per semplicità, si ipotizzi inizialmente di parametrizzare un ristretto numero di colonne, ciascuna con altezza differente ma mai superiore all'altezza della *Walls map*. Queste colonne verranno disposte una accanto all'altra replicando lo schema in figura 3.8. Ogni volta che ne viene aggiunta una, vengono scritti anche i corrispondenti pixel nella *Geometry map* e nella *Roofs map*. Si noti che, essendo disposte una accanto all'altra, le coordinate (u, v) di queste colonne differiranno lungo u di una sola unità l'una dall'altra, mentre la v rimane pari a zero.

A questo punto abbiamo creato correttamente una *striscia* di viste laterali, costituita, supponiamo, da 256 colonne. La successiva colonna, per via della dimensione prefissata della *Walls map*, non può più essere memorizzata accanto alle precedenti, pertanto deve trovare spazio *al di sopra* di quelle già salvate.

Vi sono vari modi per scegliere sopra quale colonna andremo a posizionare la nuova vista laterale, ad esempio individuando la più piccola tra quelle già memorizzate, oppure analizzandole una per una fintanto che non si trova la prima disponibile. Qualunque criterio venga utilizzato, esso deve essere in grado di mantenere integra la codifica delle viste laterali, evitando cioè di spezzarle per mancanza di spazio. Può accadere, infatti, che il numero di colonne da parametrizzare sia tale che la loro disposizione in strisce sovrapposte non sia sufficiente a memorizzarle tutte all'interno della *Walls map* (ampia $256 \times n$). In questi casi, peraltro piuttosto frequenti, l'intera rappresentazione delle viste laterali viene scalata lungo l'altezza di un fattore 2^{-i}, con $i = 0, 1, 2, ...$ in modo da contenere gli errori di rappresentazione numerica e

di approssimazione.

La scalatura viene eseguita campionando il dato dalla colonna effettuando una lettura ogni 2^i voxel, con $i = 0, 1, 2, ...$ pari al precedente. Questi voxel verranno tradotti nei pixel della vista laterale e, pertanto, saranno gli unici che contribuiranno al calcolo del vettore normale alla parete, il quale è ottenuto semplicemente come media delle loro normali.

Le colonne di tipo **4**, cioè quelle che contengono anche elementi trasparenti, sono memorizzate allo stesso modo. L'unica differenza sussiste nella gestione dei voxel vuoti: innanzitutto, questi non contribuiranno al calcolo della normale; in secondo luogo, verranno codificati come pixel in cui tutti canali conterranno il valore zero.

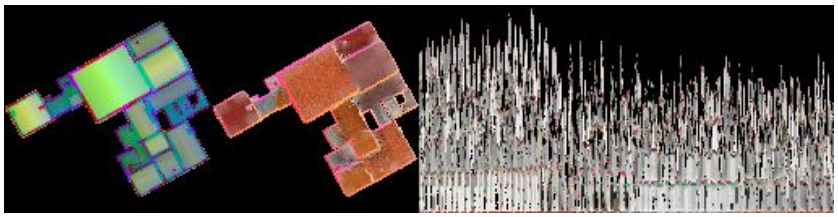


Figura 3.10: *L'immagine mostra la parametrizzazione di un gruppo di edifici.*

3.4 Sampling volumetrico

Il sampling è un procedimento di campionamento che permette di passare da una rappresentazione di tipo *continuo*, ad una di tipo raster [Ang05]. La versione 3D di questo procedimento prende il nome di sampling volumetrico o *voxelizzazione*. Esso è l'analogo 3D della rasterizzazione 2D comunemente usata per visualizzare le immagini su monitor e stampanti. La rasterizzazione consiste nell'approssimare la forma di un oggetto inserendolo all'interno di una griglia, colorando opportunamente le sole celle che l'oggetto attraversa. Tali celle nel caso bidimensionale sono di forma quadrata e sono ben note

come pixel. Nel caso tridimensionale sono chiamate voxel e sono di forma cubica.

In seguito si assumerà che la rappresentazione *continua* di un oggetto tridimensionale avvenga per mazzo di *mesh*, ovvero, attraverso vertici e triangoli.

La strategia proposta in questo lavoro di tesi, usata per generare una rappresentazione volumetrica a partire da una mesh, ha molte analogie con la tecnica denominata *shape-from-silhouette* [NB94], molto conosciuta nell'ambito delle acquisizioni tridimensionali. In essa vengono effettuate numerose fotografie di uno stesso oggetto, ciascuna da un punto di vista differente dalle altre. Per ogni foto, l'oggetto viene quindi estratto dalla scena circostante delineandone i contorni; le sagome così generate vengono poi utilizzate, una per una, per creare quelli che prendono il nome di *conoidi*, ovvero quei solidi dalla forma di cono ottenuti congiungendo il punto di vista (vertice del cono) con i punti appartenenti alle sagome estratte precedentemente. Come si vede in figura 3.11, l'intersezione di tutti i *conoidi* darà luogo ad un'approssimazione volumetrica dell'oggetto originale.

Il principio alla base della nuova procedura di campionamento trae spunto dalla tecnica appena descritta, ma viene applicata su oggetti sintetici piuttosto che su oggetti reali. In questo modo, è possibile ottenere una rappresentazione volumetrica degli oggetti relativa al solo volume esterno, cioè visibile da fuori. Ne segue che, con l'algoritmo adottato, le parti interne di un edificio non verranno rasterizzate.

La procedura prevede di eseguire diversi rendering dell'oggetto, ma da essi non verrà estratta alcuna sagoma, come nel caso dello *shape-from-silhouette*, quindi l'approssimazione del volume non sarà generata dall'intersezione dei conoidi. Infatti, i rendering verranno utilizzati per realizzare quello che prende il nome di *space carving*.

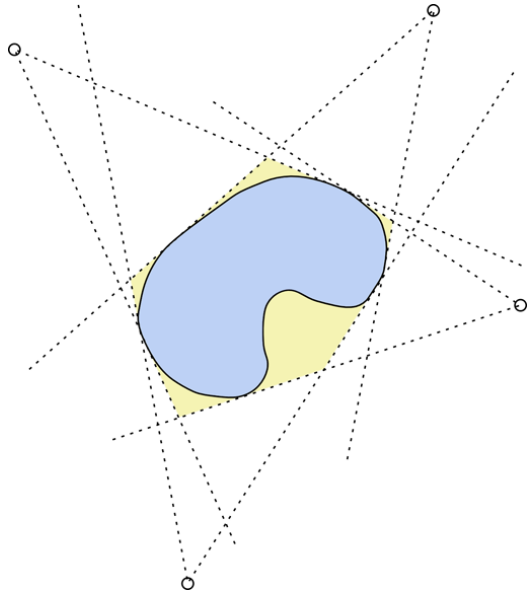


Figura 3.11: *Shape-from-silhouette in 2D. L'intersezione di quattro conoidi da luogo all'approssimazione (giallo) dell'oggetto originale (azzurro). Maggiore è il numero di conoidi e migliore sarà tale approssimazione.*

3.4.1 Space Carving via Shadow Mapping

Lo *space carving* è un algoritmo che, dato un volume iniziale all'interno del quale è racchiuso un oggetto tridimensionale, provvede a rimuovere delle porzioni di tale volume fintanto che questo non converge con la forma dell'oggetto in esso racchiuso [KS00]. Per analogia, così come il Maestro estraeva le sue opere imprigionate all'interno dei blocchi di marmo, il carving estrae gli oggetti racchiusi all'interno di un volume sintetico.

Per realizzare il procedimento di *space carving* appena descritto, ogni rendering viene processato secondo una tecnica nota come *shadow mapping*, normalmente utilizzata per proiettare le ombre in una scena generata al computer, incrementandone così il realismo visivo [Wil78]. Il procedimento di generazione dell'immagine consiste nell'effettuare più volte il rendering della scena. Ogni rendering viene eseguito da un punto di vista differente, cor-

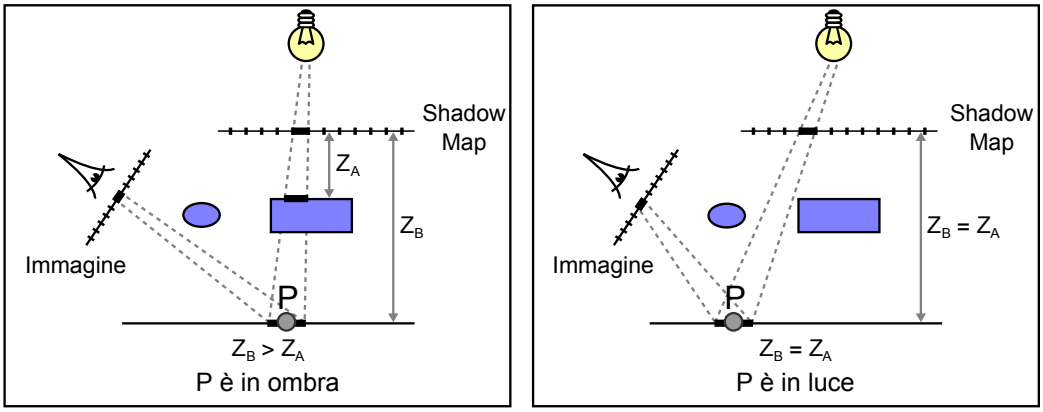


Figura 3.12

rispondente alle posizioni delle sorgenti luminose presenti nella scena (per semplicità, si supponga che vi sia una sola luce). Di questi rendering, viene creata la mappa di profondità (*depth map*), ma, poiché è stata generata dal punto di vista di una luce, prenderà il nome di *shadow map*. Una shadow map memorizza, per ogni suo pixel, la distanza che intercorre tra la luce e la geometria rasterizzata.

Una volta effettuati i rendering dal punto di vista di tutte le luci, viene eseguito il rendering dal punto di vista dell'osservatore. Durante questa fase, ogni pixel dell'immagine principale viene confrontato con il pixel corrispondente nella *shadow map*, appena generata, per stabilire se esso si trovi in luce o in ombra. Questo confronto è illustrato in figura 3.12.

Lo *shadow mapping* è una tecnica molto efficace, veloce e, soprattutto, *image based*, cioè del tutto indipendente dal contenuto della scena, pertanto viene implementato interamente sul processore grafico.

Combinando i princìpi dello *shape-from-silhouette* con la semplicità dello *shadow mapping* è stato quindi possibile realizzare un innovativo l'algoritmo di *space carving* eseguito interamente dal processore grafico. In questo algoritmo, analogamente allo *shape-from-silhouette*, vengono effettuati numerose

riprese (rendering) di un oggetto da altrettanti punti di vista. Questi saranno disposti uniformemente su una superficie sferica e rivolti verso l'oggetto che giace al centro di tale sfera.

Tali punti di vista sono l'analogo dei punti luce relativi allo *shadow mapping*, pertanto, i rendering da essi generati produrranno la corrispondente *shadow map*. In aggiunta, poiché di un oggetto risultano di interesse anche altri attributi come il suo colore o l'orientamento dei vettori normali alle superfici, ciascun rendering produrrà anche questo tipo di informazione.

A questo punto, può iniziare l'algoritmo di *carving*. Si faccia riferimento alla figura 3.13, in cui viene illustrato il funzionamento di tale algoritmo nel caso 2D. L'approssimazione volumetrica dell'oggetto è inizialmente costituita da una griglia regolare di voxel che viene scandita a piccoli passi di dimensione prefissata. Ad ogni passo, si estrae una porzione del volume chiamata *slice*, la quale viene *sovrapposta* alla *shadow map* corrente e analizzata pixel per pixel: quelli che risultano all'interno della *slice*, o lo sono entro una certa soglia, vengono candidati come voxel pieni. Questo valore di soglia determina la dimensione dei voxel nel volume finale.

I voxel candidati verranno quindi riempiti secondo l'attributo corretto (colore, normale...), ottenuto dalle informazioni addizionali relative al punto di vista corrente.

Il procedimento di *carving* prosegue analizzando, per ogni *shadow map*, tutte le slice estratte dal volume. Al termine di questo procedimento, cioè quando tutti i rendering saranno stati processati, i voxel pieni saranno quelli che sono risultati *visibili* da tutti i punti di vista.

La forza di questo metodo risiede nella sua capacità di essere eseguito interamente dall'hardware grafico. Questo fatto porta due principali vantaggi:

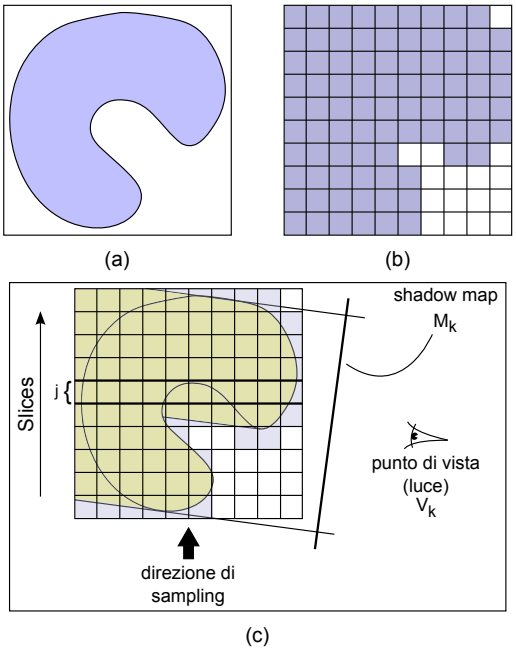


Figura 3.13: *In figura viene illustrato un oggetto nella sua rappresentazione geometrica (a), e nella sua rappresentazione volumetrica parziale (b). Tale rappresentazione è generata dal punto di vista mostrato in (c), il quale permette di eliminare, attraverso il* carving*, i voxel di colore bianco. Sulla sinistra, la freccia indica il verso con il quale le slices vengono scandite per il confronto con la vista corrente* v_k*.*

- **Efficienza**

  Essendo eseguito interamente dal processore grafico, questo metodo consente di ottenere una rappresentazione volumetrica di un oggetto in breve tempo, proporzionale al tempo necessario per visualizzarlo k volte, dove k è il numero di viste effettuate.

- **Robustezza**

  Essendo basato sul rendering, questo metodo permette di creare la rappresentazione raster di un qualunque oggetto tridimensionale, a patto che ne si sappia fare il rendering. Ciò significa che anche i casi di geo-

metria complessa, degenere o auto-intersecante non richiedono computazioni addizionali. Infatti, dato che si è in grado di visualizzarli su schermo, con un adeguato numero di viste anche gli oggetti complessi verranno rasterizzati con semplicità.

Figura 3.14: *L'immagine di sinistra mostra un edificio nella sua rappresentazione geometrica. L'immagine di destra mostra lo stesso edificio, in versione volumetrica, generata con l'algoritmo di* Space Carving *descritto in questo capitolo.*

Infine, un ulteriore vantaggio di tale tecnica risiede nella possibilità di calcolare il fattore occlusione ambientale, per ogni voxel, senza alcun costo addizionale. Infatti, è sufficiente tenere conto del numero di volte che ogni voxel è risultato visibile. Se questa condizione è sempre stata verificata, significa che il voxel non è mai stato occluso, dunque, tale contatore coinciderà con il numero di rendering effettuati. Se la condizione è stata verificata solo alcune volte, significa che il voxel in esame è parzialmente occluso, pertanto dovrà essere visualizzato più scuro. Ne segue che i voxel *interni* al volume saranno caratterizzati da un valore di occlusione ambientale pari a zero.

Nonostante in letteratura si possano trovare numerosi algoritmi di sampling volumetrico basato su GPU, come in [HLTC05, ED08], l'algoritmo ap-

pena proposto si differenzia nella capacità di campionare la sola parte di volume visibile dall'esterno.

3.4.2 Antialiasing via supersampling

La procedura di sampling appena descritta crea un volume i cui voxel sono considerati o pieni o vuoti. Questo tipo di campionamento porta ad inevitabili effetti di *aliasing*, notabili come *scalettature* in fase di visualizzazione.

Il supersampling è una tecnica di *antialiasing*, cioè che permette di ridurre l'effetto di scalettatura appena descritto [Ang05].

Nel caso 2D, ciò avviene effettuando numerosi sample all'interno di uno stesso pixel. Per fare questo, viene creata prima un'immagine ad una risoluzione molto maggiore di quella necessaria alla visualizzazione e, successivamente, questa viene ristretta alla dimensione desiderata, attraverso un operazione detta *downsampling*, in modo tale che ogni pixel dell'immagine risultante nasca dal contributo di più pixel dell'immagine più grande. Il numero di campionamenti effettuati all'interno di ogni pixel determina la qualità del risultato finale e indica anche la *risoluzione* del supersampling.

Analogamente, il supersampling nel caso 3D consiste nell'effettuare numerosi sampling all'interno di uno stesso voxel. Come nel caso 2D, prima si crea un volume ad una risoluzione maggiore e poi, attraverso il downsampling, si genera il volume alla risoluzione desiderata.

Ciò significa che ciascun voxel non vuoto del volume risultante è ottenuto con il contributo di più voxel provenienti da un volume più grande. Il numero di voxel che prendono parte alla generazione di tale contributo prende il nome di *coverage*.

Tale valore verrà memorizzato come coefficiente di trasparenza per ciascun voxel non vuoto. Per comprenderne meglio il suo utilizzo, si faccia riferimento allo schema in figura 3.15: nel gruppo di voxel **(a)** vi sono 8 cubi

pieni, pertanto la *coverage* sarà pari a 1; ciò significa che il voxel risultante sarà completamente opaco. Il gruppo successivo **(b)** possiede soli 3 voxel pieni, quindi la *coverage* del voxel generato sarà pari a $\frac{3}{8}$. Il terzo gruppo **(c)** è costituito da soli voxel vuoti, quindi darà origine ad un voxel vuoto.

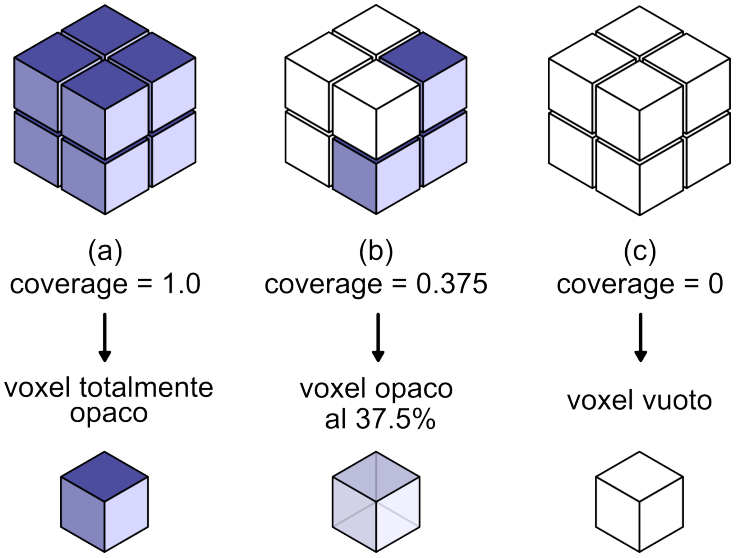


Figura 3.15: *Downsampling a partire da un super-volume 8 volte più grande.*

Come è stato illustrato nella fase di parametrizzazione, è possibile codificare come trasparente lo spazio vuoto che esiste tra voxel sovrapposti. Tale codifica viene fatta sfruttando il canale alpha delle V-BlockMap. Questa scelta non è stata casuale, ma è stata guidata dalla possibilità di utilizzare questo stesso canale per memorizzare i valori di *coverage* in ogni pixel delle V-BlockMap che contengono informazioni di colore.

A causa delle limitazioni hardware, non è possibile tenere in memoria video dei super-volumi di dimensioni troppo elevate. Per questo motivo si è implementato un supersampling partizionato.

Si supponga di voler campionare un volume di dimensioni $n \times n \times n$ con risoluzione di supersampling α. In questo caso la procedura di sampling verrà

ripetuta per α^3 volte (secondo lo schema in figura 3.16, $\alpha = 2$), ciascuna delle quali sarà rivolta al campionamento di una differente porzione della geometria originale. Per la precisione, tale geometria viene partizionata in α^3 regioni cubiche di uguali dimensioni, ognuna delle quali verrà campionata alla risoluzione $n \times n \times n$.

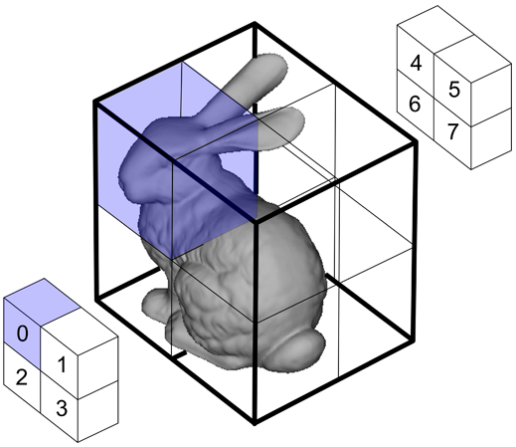


Figura 3.16: *L'ordine con il quale viene eseguito il sampling dei sottovolumi.*

Al termine di questo procedimento, le regioni vengono assemblate in modo da creare un super-volume di dimensioni $(n \times \alpha)^3$ che, attraverso il procedimento di downsampling, verra trasformato in un volume di dimensioni n^3.

3.5 Visualizzazione della struttura V-BlockMap

Al fine di visualizzare su schermo il risultato della parametrizzazione, l'applicazione che effettuerà il rendering di una V-BlockMap si preoccuperà di inviare alla scheda grafica i soli vertici del *bounding box* relativo alla V-BlockMap; è all'interno di questo volume che opererà il ray-tracer, implementato sul processore grafico come *shader program*.

Uno *shader program* si compone generalmente di due parti principali chiamate *vertex program* e *fragment program* [Ang05, WDS99, Ros06]. Per implementare il ray-tracer, nella prima fase, oltre a eseguire le trasformazioni geometriche che posizionano e adattano la V-BlockMap secondo i parametri della scena, verranno effettuati alcuni calcoli preparativi per l'avanzamento dei raggi. Tale avanzamento, cioè il ray-tracer vero e proprio, sarà eseguito interamente durante la seconda fase, il *fragment program*. In essa, per ogni *frammento*, viene creato un che raggio attraversa la *Geometry map*. L'origine di questi raggi coincide con la posizione dell'osservatore, mentre la loro direzione è data dalla direzione di vista.

Ciascun pixel della *Geometry map* che interseca il cammino di uno dei raggi viene analizzato per stabilire se si tratta o meno di una colonna da estrudere. L'attraversamento lungo la griglia avviene secondo l'algoritmo proposto in [AW87] il quale permette di visitare tutti quei pixel che intersecano il cammino di un raggio, anche se solo parzialmente.

La figura 3.17 mostra il funzionamento di tale algoritmo. Le celle colorate sono quelle attraversate dal raggio e verranno visitate secondo l'ordine $(a...h)$.

Durante l'attraversamento della *Geometry map*, la visita di un pixel consiste nel verificare se esso contiene un valore di $Zmax$ maggiore di zero. Se questa condizione non è verificata il raggio proseguirà il suo cammino, altrimenti verranno eseguiti due tipi di test di intersezione. Il primo è il più immediato, perché si occupa semplicemente di confrontare l'altezza corrente del raggio con i valori $Zmax$ e $Zmin$ letti dal pixel corrente. Il secondo test ha luogo solo se fallisce il primo, ovvero se il raggio si trova al di sopra o al di sotto della colonna corrente. Le possibili alternative sono tre (si faccia riferimento alla figura 3.18).

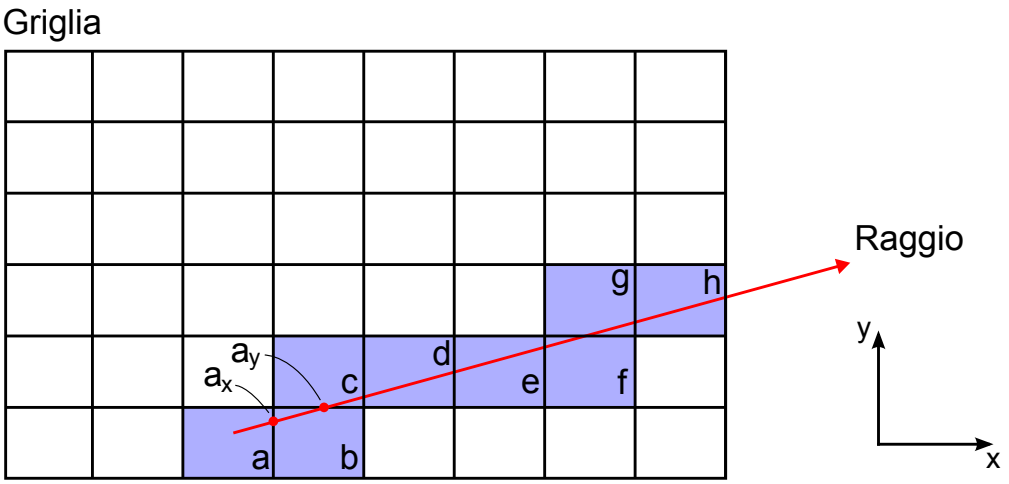


Figura 3.17: *Attraversamento del raggio nel caso 2D.*

- **A. Il raggio non interseca alcuna parete**
  In questo caso si procederà semplicemente con l'avanzamento del raggio, cioè visitando il prossimo pixel.

- **B. Il raggio interseca una parete orizzontale**
  Se il raggio non ha intersecato una parete verticale è possibile che abbia intersecato un tetto o la parte inferiore di un oggetto sopraelevato. A tale scopo è necessario verificare se le coordinate del punto di intersezione tra il raggio e il piano $z = Zmax$ ricadono all'interno del pixel corrente. In caso affermativo, si prelevano le informazioni di shading in base al tipo di colonna: dalla *Roofs map* se si tratta di una colonna non esposta; dalla *Walls map* se la colonna è esposta.

  Questo procedimento è identico anche nel caso in cui il raggio intersechi la parte inferiore di un oggetto sopraelevato. L'unica differenza risiede nel fatto di testare l'intersezione del raggio con il piano di equazione $z = Zmin$. Si intuisce, quindi, che le informazioni di *shading* relative

alle parti sottostanti degli edifici, saranno le stesse della parte superiore.

- **C. Il raggio interseca una parete verticale**
  Se le coordinate attuali del raggio sono tali da intersecare una parete verticale, si utilizzano le coordinate (u, v) prelevate dal pixel corrente per individuare la base della corrispondente colonna nella *Walls map*. L'altezza corrente del raggio, opportunamente scalata, viene dunque utilizzata come offset dalla base della colonna per accedere al pixel contenente le informazioni di *shading* corrette. Come sappiamo, la normale si trova nel pixel situato in cima alla colonna appena individuata. Per accedere a tale pixel è sufficiente calcolare e scalare opportunamente l'altezza della colonna corrente. In esso troveremo il vettore normale da utilizzare per il calcolo della corretta illuminazione.

  Il procedimento appena descritto è quello responsabile della visualizzazione delle sole colonne esposte.

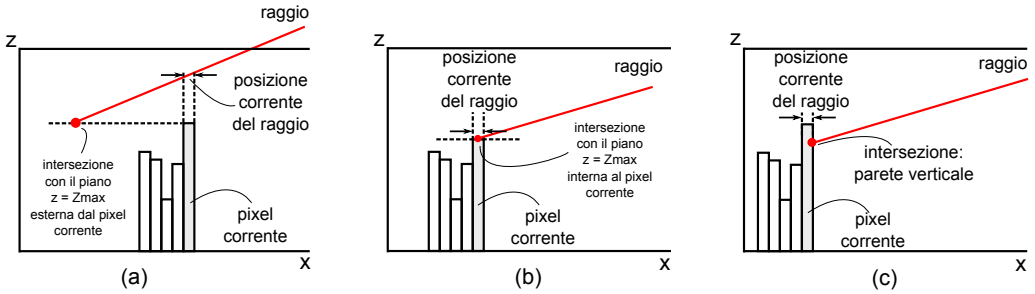


Figura 3.18: *Intersezioni dei raggi: (a) Il punto di intersezione ricade al di fuori del pixel corrente, pertanto il raggio prosegue il suo cammino; (b) Il punto di intersezione ricade all'interno del pixel corrente: il raggio ha incontrato una parete orizzontale; (c) L'altezza del raggio nel punto di intersezione è compresa tra $Zmax$ e $Zmin$: il raggio ha incontrato una parete verticale.*

A questo punto, il ray-tracer ha ottenuto tutte le informazioni di *shading* necessarie per determinare il colore del frammento corrente.

3.6 Integrazione in una struttura multirisoluzione

Al fine di visualizzare un ambiente urbano di ampie dimensioni, è necessario integrare le V-BlockMap all'interno di una struttura multirisoluzione, costituita quindi da diversi nodi, ognuno dei quali verrà utilizzata per rappresentare una differente porzione del dominio.

Il tipo di struttura che si è utilizzata è un albero quaternario (*quad tree*) i cui nodi memorizzano delle porzioni quadrate di terreno sotto forma di V-BlockMap. La radice contiene la rappresentazione dell'intero ambiente urbano, mentre, ciascuno dei suoi quattro figli conterrà esattamente un quarto di tale rappresentazione e così via fino a raggiungere le foglie dell'albero, le quali contengono la parametrizzazione della geometria al massimo livello di dettaglio.

Le V-BlockMap vengono memorizzate esternamente alla gerarchia, precisamente all'interno di texture *contenitori*, dette *atlas*. Essi sono delle vere e proprie texture all'interno delle quali possono trovare luogo più V-BlockMap, in funzione delle dimensioni dell'altlas stesso.

3.6.1 Creazione del *quad tree*

Durante la fase di creazione della struttura multirisoluzione, la geometria dell'ambiente urbano viene suddivisa in base allo schema appena descritto. Si parte con la radice, ne si estrae il *bounding box* e lo si divide a metà lungo gli assi X e Y. Vengono creati così i *bounding box* dei nodi figli, i quali necessitano però di essere riadattati lungo la direzione Z. In seguito,

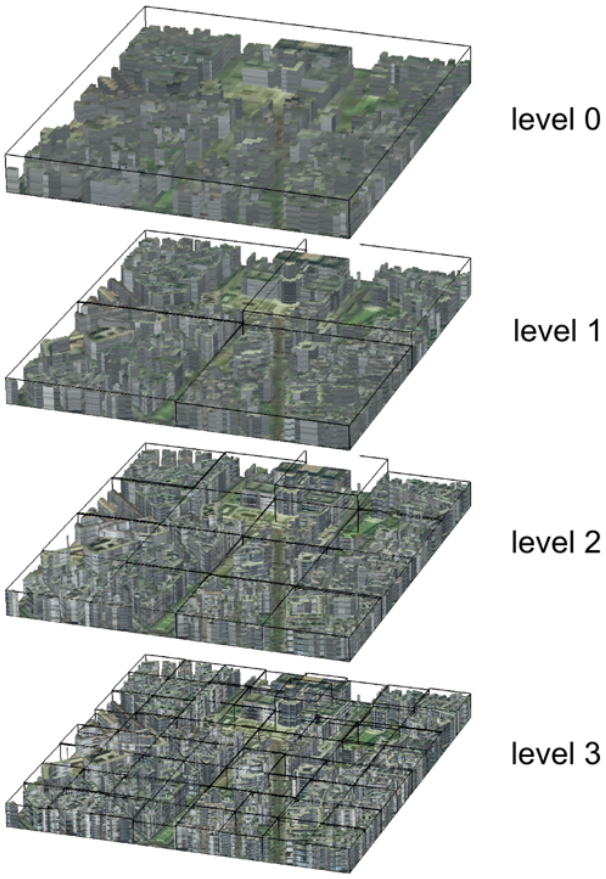


Figura 3.19: *Quattro livelli di un* quad tree *di V-BlockMap.*

per la creazione di ogni nodo, la geometria in essi contenuta viene *tagliata* seguendo il bordo del *bounding box* corrispondente, eccezion fatta per la radice. In questo modo, solo la geometria che ricade completamente al suo interno contribuirà alla creazione della V-BlockMap corrispondente.

In realtà, operando in questo modo, il campionamento dei fattori di occlusione ambientale non risulterebbe corretto, poiché la procedura di sampling avviene in modo separato e indipendente per ciascun nodo. Ciò significa che i voxel situati in prossimità dei bordi del *bounding box* risulterebbero comple-

tamente esposti, cioè mai occlusi e il calcolo dell'occlusione ambientale per questo tipo di voxel non risulterebbe fondamentalmente sbagliato.

Per determinare un corretto valore di occlusione ambientale, si è stabilito che ogni nodo possa effettuare il sampling su una regione geometrica più ampia di quella realmente richiesta. La geometria *in più* sarà comunque utilizzata per il solo calcolo dell'occlusione ambientale, ma, verrà scartata al momento di generare il contenuto geometrico della corrispondente V-BlockMap.

Riassumendo, ogni nodo della struttura multirisoluzione, fatta eccezione della radice, viene creato suddividendo in quattro parti il dominio (*bounding box*) della geometria appartenente al nodo padre. Tale geometria viene prima sostituita dalla relativa rappresentazione volumetrica creata utilizzando la procedura di sampling. In seguito, essa verrà a sua volta sostituita dalla V-BlockMap corrispondente.

3.6.2 Calcolo dello *Screen Space Error*

Come si vedrà a breve (sezione 3.7), la fase di visualizzazione sarà in grado di stabilire se un nodo può essere visualizzato, oppure necessita di ulteriori passi di raffinamento.

Questa decisione viene presa valutando il numero di pixel occupati dalla proiezione su schermo della V-BlockMap relativa al nodo corrente. Se tale numero risulta maggiore di un certo valore di soglia, significa che la proiezione della V-BlockMap occupa una regione di schermo troppo ampia, pertanto, il nodo che la contiene dovrà essere raffinato. In altre parole, si impone che la proiezione in pixel di una V-BlockMap debba essere costante.

Per determinare velocemente il numero di pixel che verranno proiettati su schermo, per ogni nodo si calcola una loro sovrastima individuando la *sfera* che contiene interamente il dominio del nodo corrente. In questo modo, il

numero di pixel proiettati su schermo sarà un valore proporzionale al raggio di tale sfera. Inoltre, tale sovrastima risulta invariante rispetto alle operazioni di rotazione, quindi il raffinamento di un nodo avverrà solo al variare della sua distanza con l'osservatore.

In fase di creazione della struttura multirisoluzione, viene calcolata la sovrastima dell'errore che ogni nodo produrrà una volta proiettato sullo schermo. Tale valore coincide con metà della diagonale del *bounding box* di ciascun nodo e sarà indicato nel seguito con ε_{screen}.

Poiché è necessario garantire che la sovrastima dell'errore di nodo di livello i non sia mai inferiore alla sovrastima di un suo nodo figlio [CGG+03, CGG+04], ad ognuno di essi viene associata una seconda sfera contenitiva di raggio $r \geqslant \varepsilon_{screen}$. Tali sfere sono capaci di espandersi al fine di contenere interamente le sfere dei nodi figli. Queste vengono generate a partire dai nodi foglia, in cui si impone che $r = \varepsilon_{screen}$. Nei nodi di livello superiore, il raggio r viene espanso in modo che la sfera risultante sia capace di contenere interamente le quattro sfere figlie (si veda la figura 3.20).

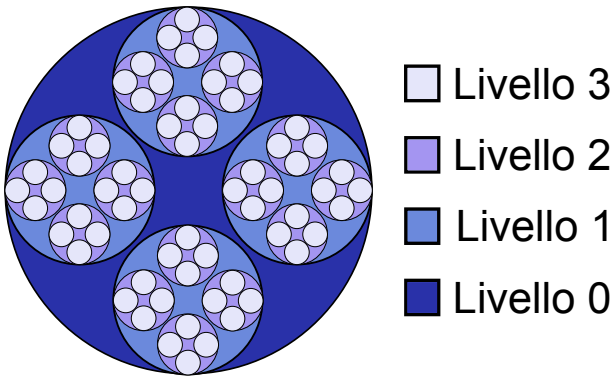


Figura 3.20: *Screen space error. La sovrastima dell'errore di livello i è tale da contenere completamente la sovrastima dell'errore dei nodi figli.*

3.7 Rendering interattivo di ambienti urbani

La fase di rendering costituisce nella visita della struttura multirisoluzione. L'ambiente urbano viene posizionato secondo i parametri del nodo radice e da questo, comincia la visita. Ad ogni frame, si valuta la distanza tra l'osservatore e il centro del nodo in esame. Il valore di tale distanza è d_{obs}. Del nodo corrente si conosce la sovrastima dell'errore ε_{screen} e il raggio r della sfera contenitiva. Infine, dai parametri della vista corrente è possibile determinare la distanza d_{img} tra l'osservatore e il piano immagine.

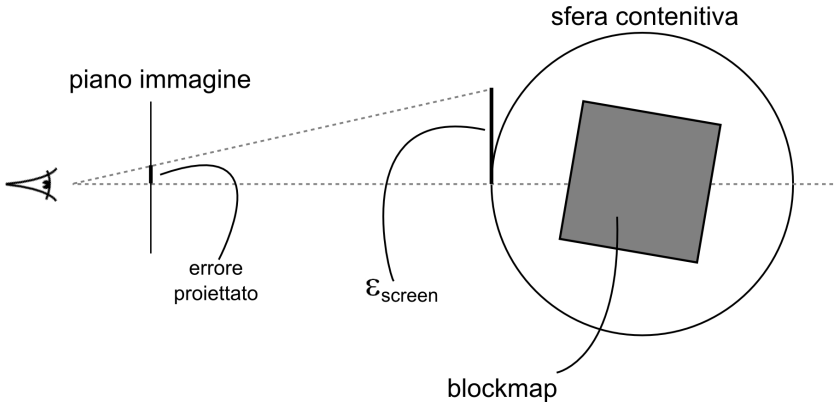


Figura 3.21

Noti questi dati, attraverso la seguente relazione è possibile calcolare la dimensione assunta ε_{screen} sul piano immagine, espressa in pixel:

$$\epsilon = \frac{\varepsilon_{screen} \cdot d_{img}}{d_{obs} - r}$$

L'operazione di raffinamento avrà dunque luogo se la il valore di ϵ supererà un certo valore di soglia impostato dell'applicazione. Su macchine lente tale valore di soglia sarà più elevato, mentre, su macchine veloci si potrà impostare

più basso. Ciò equivale a dire che con un hardware meno potente si è disposti ad accettare un maggior errore su schermo.
Infine, per ridurre il numero di nodi da processare, è stato implementato un algoritmo di *frustum culling* (si veda la sezione 2.1.1), il quale permette di escludere dalla computazione quei nodi che *sicuramente* non verranno visualizzati in quanto *esterni* al volume di vista.

Capitolo 4

Risultati

Il *framework* è costituito da due elementi principali: il generatore e il visualizzatore. Il primo rappresenta la parte di computazione *off-line* e comprende le fasi di ricostruzione volumetrica, parametrizzazione e creazione della struttura multirisoluzione. Il visualizzatore implementa la parte di visualizzazione interattiva.

Ogni fase del generatore produce una diversa rappresentazione della geometria originale. Per valutare i risultati delle singole fasi, sono stati realizzati dei visualizzatori atti a tale scopo. Attraverso questi è stato possibile valutare visivamente l'accuratezza con cui una V-BlockMap approssima il corrispondente dato geometrico.

L'intero *framework*, inclusi i corrispondenti visualizzatori, è stato implementato in *C++*, con il supporto della libreria grafica *OpenGL* per la parte di visualizzazione, l'ausilio della libreria *QT* per la gestione delle interfacce utente e la libreria *VCGLib* per la gestione delle mesh tridimensionali.

Gli algoritmi di *space carving* e *ray-tracing*, eseguiti dall'hardware grafico, sono stati implementati nel linguaggio OpenGL Shading Language (*GLSL*).

Il sistema è stato testato su numerosi dataset di prova, sia artificiali, sia provenienti da situazioni reali. Tra i dataset artificiali, alcuni sono stati

volutamente realizzati con lo scopo di osservare il comportamento delle varie fasi in presenza di casi limite o degeneri.

Per quanto riguarda il test su dataset reali, si è utilizzato una parziale ricostruzione procedurale dell'ambiente urbano di *Pompei*. Tale dataset, è costituito da 50 edifici, descritti da circa 150K vertici e circa 230K triangoli. La sola geometria occupa circa 30 MB su disco mentre le relative texture ne occupano circa 40 MB, per un totale complessivo di circa 70 MB. La versione completa del dataset è stata gentilmente concessa da Pascal Müller [MWH+06].

Il fine principale di tali test è stato quello di valutare la qualità e la compattezza della rappresentazione dell'ambiente urbano.Di seguito, verranno riportati i tempi relativi alle varie fasi di computazione e i risultati di visualizzazione delle singole fasi. I tempi sono stati misurati su una macchina dotata di Intel Core 2 Duo @2.33GHz, equipaggiata con 3GB di RAM e scheda grafica nVIDIA GeForce 8600 GTS con a bordo 256MB di memoria.

Dataset artificiali

I test relativi a dataset artificiali, costituiti per lo più da cubi disposti in vario modo, sono quelli in cui si sono ottenuti i migliori risultati. Ciò è dovuto alla semplicità dei dati di partenza, che hanno permesso di creare rappresentazioni volumetriche ottimali, le quali sono risultate poi di facile parametrizzazione. I risultati, come si può vedere in figura 4.1, sono tali da non far scorgere alcuna differenza tra la visualizzazione del dato geometrico e quella della sua parametrizzazione. Tuttavia, non vi sono grossi vantaggi nell'utilizzo di una BlockMap per visualizzare una mezza dozzina di cubi.

I volumi sono stati creati ad una risoluzione di 64^3 voxel, attraverso 256 viste. I tempi necessari alla creazione sono stati mediamente di 30 secondi, mentre la loro parametrizzazione ha richiesto in media 10 secondi. Per ogni

rappresentazione è riportato il numero di fotogrammi al secondo ottenuti sulla macchina di riferimento.

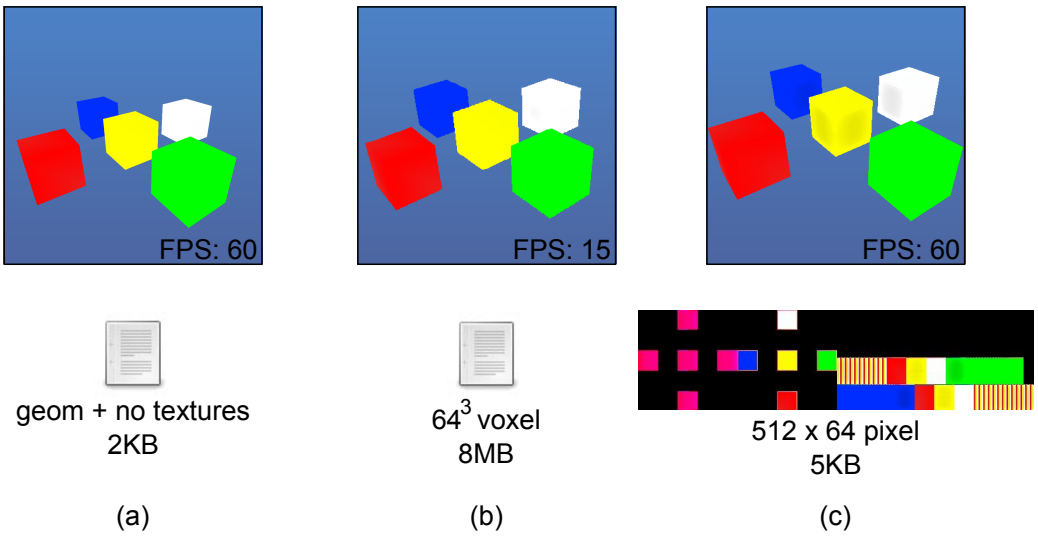


Figura 4.1

Edifici singoli

Le prossime immagini mettono a confronto le varie rappresentazioni relative ad un singolo edificio. Si faccia riferimento alla figura 4.2: il modello relativo alla casetta **(a)** è costituito da 6300 triangoli e occupa su disco circa 6 megabyte. La sua rappresentazione volumetrica **(b)** è stata generata con 256 viste e consiste in un volume di 128^3 voxel le cui dimensioni su disco sono di 64 megabyte. La BlockMap **(c)** risultante dalla parametrizzazione di tale volume, è un'immagine di 512×128 pixel che occupa su disco circa 100 KB.

Il tempo necessario a creare tale dataset è stato di circa trenta secondi. Con un numero inferiore di viste si sarebbero ottenuti tempi minori a discapito della precisione volume risultante. La parametrizzazione di tale volume richiede un tempo trascurabile, nell'ordine di pochi secondi.

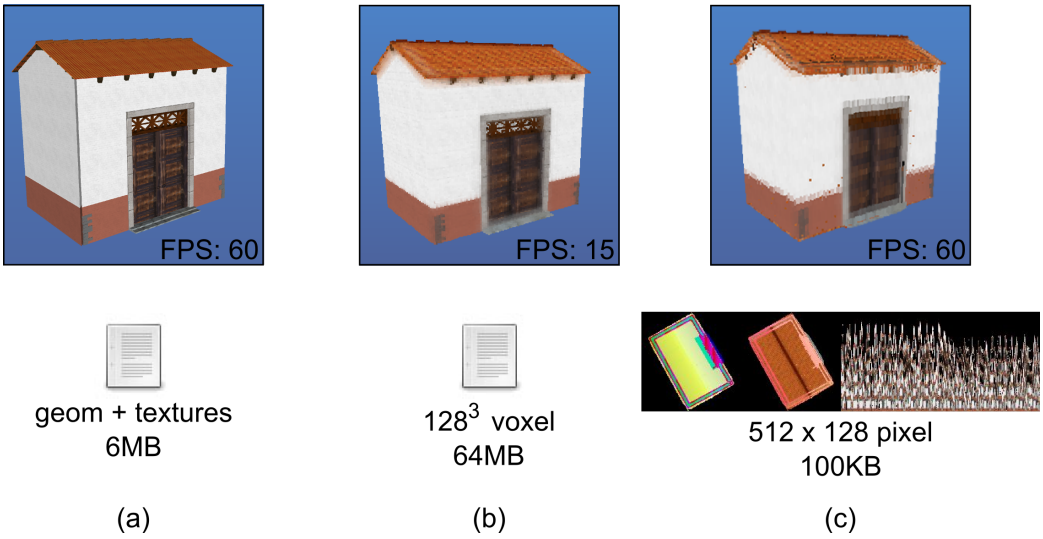


Figura 4.2

Tale dataset volumetrico è particolarmente significativo poiché costituito da tutte le tipologie di colonne illustrate in 3.1.1. La rappresentazione raster risulta molto fedele a quella geometrica, perfino le features situate al di sopra della porta sono state campionate in modo corretto. Tuttavia, sono presenti anche alcuni errori, causati prevalentemente da una risoluzione della griglia non troppo elevata (128 voxel). Questi errori sono i principali responsabili degli errori presenti nella visualizzazione della BlockMap, mostrati nel dettaglio della figura 4.3: i pixel evidenziati dovrebbero essere di colore bianco. Il colore errato è causato dal fatto che la colonna, cui questi pixel appartengono, risulta *sospesa* a causa dell'approssimazione dovuta alla rasterizzazione del volume. Ne segue che, in fase di ray-tracing, tali pixel vengano identificati come *tetto* e, conseguentemente, il colore viene prelevato dalla *Roofs map* e non dalla *Walls map*. Inoltre, a causa dell'elevata compressione in quest'ultima regione, le features situate al di sopra della porta sono andate perdute.

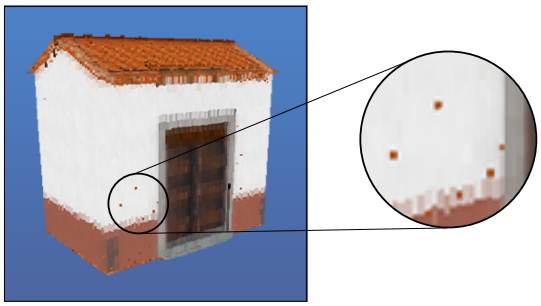


Figura 4.3: *Artefatti dovuti alle approssimazioni della rasterizzazione volumetrica.*

Pompei

Il secondo test è relativo a 20 edifici della versione procedurale di Pompei. Anche la rappresentazione volumetrica di questo dataset è stata generata attraverso 256 viste, in poco più di un minuto.

Attraverso la visualizzazione comparata di questo dataset, è possibile riconoscere i limiti sia della ricostruzione volumetrica che della parametrizzazione. Infatti, come si può notare dall'immagine **(b)** in figura 4.4, la rasterizzazione volumetrica risulta poco fedele alla corrispondente visualizzazione geometrica **(a)**. Ciò è dovuto dal fatto che il volume, costituito anche in questo caso da 128^3 voxel, risulta troppo piccolo per approssimare adeguatamente le informazioni dei 20 edifici. L'area che questi ricoprono è, infatti, più ampia di oltre quattro volte rispetto all'area del caso precedente. Ne segue che, per una visualizzazione ottimale di aree estese, sarebbe necessario incrementare la dimensione del volume oppure ricorrere ad approcci multirisoluzione.

Inoltre, questo rappresenta un caso limite anche per la parametrizzazione. Infatti, maggiore è il numero di edifici all'interno del dominio di una BlockMap e maggiore sarà il numero di colonne esposte da parametrizzare. Questo significa che per memorizzare tali colonne all'interno della *Walls map*, che ha dimensioni prefissate a 256×256 pixel (si veda 3.2.4), esse dovranno essere

Figura 4.4

scalate fino ad occupare pochi pixel. Questa operazione introduce sia effetti di *aliasing*, sia errori di approssimazione numerica, che danno entrambi origine a piccoli difetti in fase di visualizzazione.

Come si vede dai tempi illustrati in tabella 4.1, la parametrizzazione viene eseguita in breve tempo, in funzione della sola dimensione del dato volumetrico. I tempi di creazione del volume, invece, crescono con l'aumentare della complessità del modello processato. Questo è dovuto al fatto che la creazione delle viste, con le relative depth map, richiede un maggiore tempo di rendering.

La tabella 4.1, inoltre, riassume i risultati ottenuti relativamente allo spazio occupato su disco.

	Cubi	**Edificio singolo**	**Pompei**
Numero viste	256	256	256
Creazione volume	$\sim$ 30s	$\sim$ 40s	$\sim$ 60s
Parametrizzazione	$<$ 10s	$<$ 10s	$<$ 10s
Dimensioni geometria con texture	2KB	6MB	30MB
Dimensioni BlockMap	5KB	95KB	97KB

Tabella 4.1

Visualizzazione multirisoluzione

La creazione e la visualizzazione della struttura multirisoluzione sono state testate sulla porzione di Pompei costituita da 50 edifici.

Sono stati effettuati numerosi test di creazione e visualizzazione, impostando differenti risoluzioni e differenti profondità della gerarchia. L'immagine 4.5 mostra il dataset in versione geometrica, mentre le immagini 4.6 e 4.7 mettono a confronto la visualizzazione di due strutture multirisoluzione entrambe di profondità pari a 3. In un caso le ricostruzioni volumetriche dei nodi sono stati realizzati a partire da griglie di 64^3 voxel, indicato con 64x, mentre, nell'altro caso con griglie di 32^3 voxel, indicato con 32x.

Nel modello 64x, il numero di nodi generati è pari a 34, mentre nel secondo è 33. Tale differenza in termini di nodi è determinata proprio dalla differenza di risoluzione: le griglie del modello 64x sono più fitta, quindi permettono di catturare più particolari rispetto alle griglie del modello 32x. Come si può notare dalle immagini, a parità di livello, il modello 64x offre un dettaglio maggiore rispetto al modello 32x.

I confronti tra tempi, dimensioni e risoluzioni sono invece riportati nella tabella 4.2.

	Risoluzione 64 voxel	**Risoluzione 32 voxel**
Nodi	34	33
Profondità	3	3
Numero Atlas	3	1
Dimensione su disco	1.3 MB	$<$ 400 KB
Tempo di creazione	$\sim$ 1h 6min	$\sim$ 58min
FPS (valore medio)	32	32

Tabella 4.2: *La tabella riassume le caratteristiche dei due modelli multirisoluzione. Il dato geometrico dal quale sono stati generati è costituito da circa 150K vertici e circa 230K facce. Lo spazio occupato su disco è di circa 70MB, incluse le textures. Sulla macchina di riferimento, la visualizzazione di tale dataset avviene a 14 FPS.*

Analizzando i risultati in tabella, si può notare che il tempo di creazione tra la versione 64x e quella 32x differisce di soli pochi minuti. Ciò è dovuto al fatto che l'operazione che richiede maggior tempo di elaborazione è proprio il rendering dei 50 edifici. Questo influisce direttamente sul tempo di creazione delle 256 viste utilizzate nell'algoritmo di *space carving*.

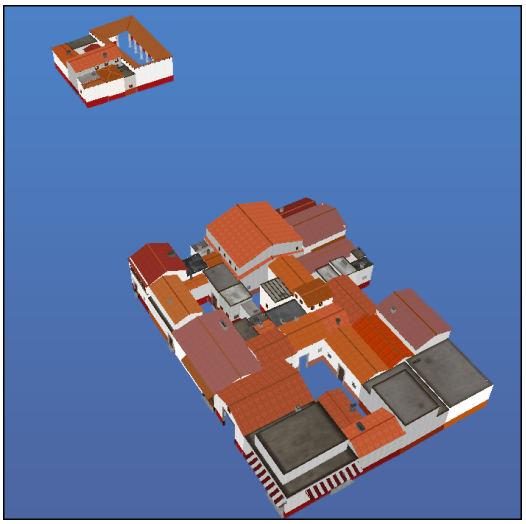


Figura 4.5: *Dataset di Pompei nella versione geometrica con textures.*

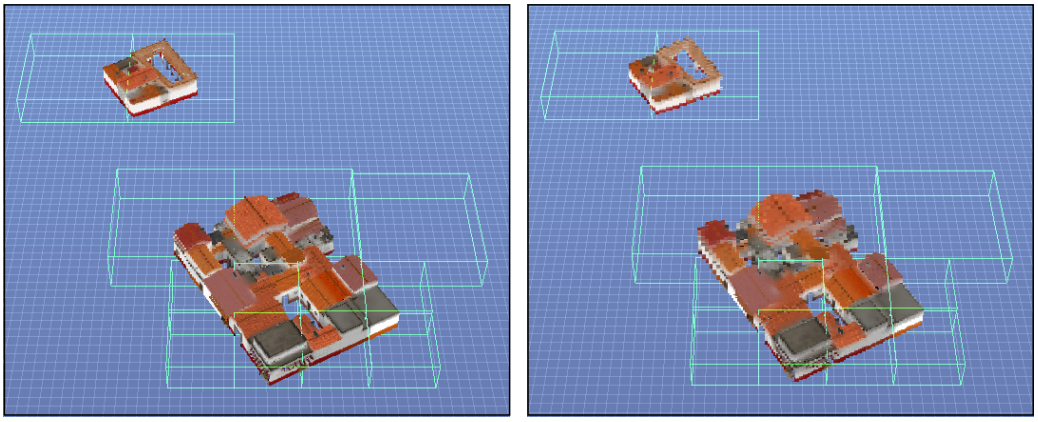


Figura 4.6: *L'immagine mette a confronto due strutture multirisoluzione di pari profondità. I nodi più lontani richiedono ulteriori passi di raffinamento.*

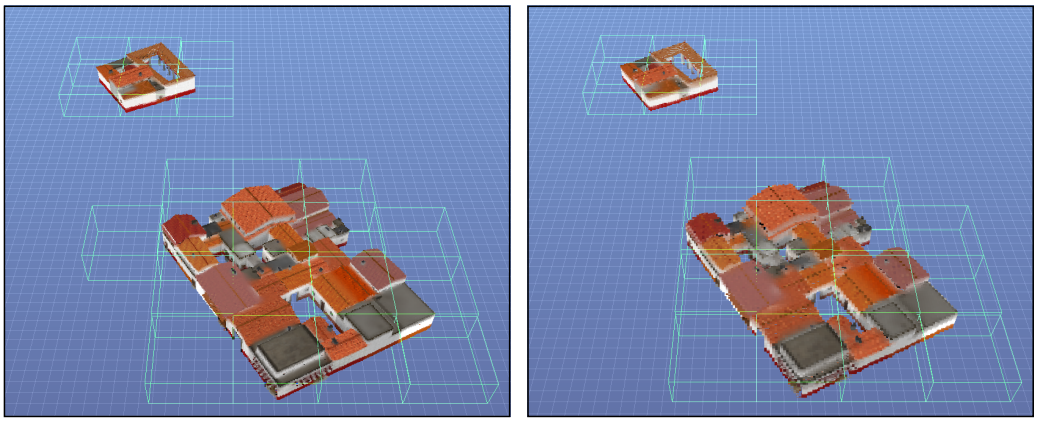


Figura 4.7: *Entrambi i modelli visualizzati al massimo dettaglio.*

Capitolo 5

Conclusioni

In questa tesi è stata presentata una tecnica innovativa per la codifica e visualizzazione di ambienti urbani. Tale tenica, chiamata V-BlockMap, adotta un rappresentazione basata su parallelepipedi distribuiti su griglia regolare che catturano il dettaglio su larga scala, cioè l'altezza e i profili verticali, che è quello principalmente percepito. La V-BlockMap viene efficacemente memorizzata come una immagine di piccole dimensioni e visualizzata per mezzo di un algoritmo di ray-tracing implementato on-board sulla scheda grafica.

Rispetto alla struttura BlockMap dalla quale si è partiti, il contributo innovativo delle V-Blockmap si evidenzia su più fronti:

- **Potenza espressiva**
  Con le V-BlockMap è possibile codificare non solo campi di altezza ma anche parti *sospese* come balconi o pensiline e tetti spioventi.

- **Campionamento della geometria visibile**
  Per rasterizzare la geometria visibile in una griglia 3D è stata sviluppata una tecnica innovativa ed efficiente di *space carving* interamente a carico dell'hardware grafico.

- **Campionamento degli attributi colore**
  Il campionamento del colore è *anisotropico*, cioè lo spazio della texture dedicato alla memorizzazione delle viste laterali rispetta le proporzioni degli edifici.

Sebbene la strategia di sampling volumetrico sia risultata molto versatile, capace di ricostruire qualsiasi oggetto tridimensionale in breve tempo e preservarne i dettagli, si presentano anche alcune nuove problematiche nel caso di dataset complessi, in particolare quando la rasterizzazione presenta errori di approssimazione. In altre parole, il processo di parametrizzazione non è *fault tolerant* rispetto alla qualità della rasterizzazione, cosicché se per esempio capita che uno o due voxel di una parete non compaiono nella rasterizzazione, l'errore si propaga nella parametrizzazione e quindi sulla visualizzazione.

5.1 Sviluppi futuri

Una sicura direzione di miglioramento consiste nell'incrementare la risoluzione della rasterizzazione e introdurre un post processing di rettifica che analizzi la rasterizzazione ed elimini le sorgenti di errore appena citate.

Inoltre, per giungere ad un framework effettivamente utilizzabile su larga scala, occorre affrontare le problematiche relative alla visualizzazione remota. In questo senso, le BlockMap presentano almeno due vantaggi: non essendo una rappresentazione incrementale ma consistendo in piccole immagini indipendenti di dimensione costante, possono giungere in modo completamente asincrono senza introdurre criticità nell'algoritmo di rendering; pur essendo già una codifica compatta, possono essere ulteriormente compresse con tecniche di compressione per immagini, anche lossy (eccetto per la parte denominata *Geometry map* nel testo).

Bibliografia

[Ang05] Edward Angel. *Interactive Computer Graphics: A Top-Down Approach using OpenGL (4th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[AW87] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, pages 3–10, 1987.

[BBC+08] Isaac Besora, Pere Brunet, Marco Callieri, Antoni Chica, Massimiliano Corsini, Matteo Dellepiane, Daniel Morales, Jordi Moyés, Guido Ranzuglia, and Roberto Scopigno. Portalada: A virtual reconstruction of the entrance of the ripoll monastery. In *3DPVT08: Fourth International Symposium on 3D Data Processing, Visualization and Transmission*, pages 89–96, June 2008.

[BD05] Henrik Buchholz and Jürgen Döllner. View-dependent rendering of multiresolution texture-atlases. In *Proceedings of the IEEE Visualization 2005*, pages 215–222, 2005.

[BUSM06] Marcello Balzani, Federico Uccelli, Roberto Scopigno, and Claudio Montani. La cattedrale di pisa nella piazza dei miracoli: un rilievo 3d per l'integrazione con i sistemi informativi di docu-

mentazione storica e di restauro. In *Restauro 2006: Salone dell'Arte del Restauro e della Conservazione dei Beni Culturali e Ambientali*. Acropoli srl, 2006. Ferrara, Aprile 2006.

[CDBG+07] Paolo Cignoni, Marco Di Benedetto, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, and Roberto Scopigno. Ray-casted blockmaps for large urban models streaming and visualization. *Computer Graphics Forum*, 26(3), Sept. 2007.

[CGG+03] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Bdam - batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum*, 22(3):505–514, September 2003.

[CGG+04] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Adaptive TetraPuzzles – efficient out-of-core construction and visualization of gigantic polygonal models. *ACM Transactions on Graphics*, 23(3):796–803, August 2004. Proc. SIGGRAPH 2004.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19(10):547–554, 1976.

[Deb04] Paul E. Debevec. Digitizing the parthenon: Estimating surface reflectance properties of a complex scene under captured natural illumination. In *VMV*, page 99, 2004.

[DWS+97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. Roaming terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages

81–88, Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[ED08] Elmar Eisemann and Xavier Décoret. Single-pass gpu solid voxelization for real-time applications. In *GI '08: Proceedings of graphics interface 2008*, pages 73–80, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society.

[FGM+02] R. Fontana, M. Greco, M. Materazzi, E. Pampaloni, L. Pezzati, C. Rocchini, and R. Scopigno. Three-dimensional modelling of statues: the minerva of arezzo. *Journal of Cultural Heritage*, 3(4):325–331, 2002.

[GMC+06] Enrico Gobbetti, Fabio Marton, Paolo Cignoni, Marco Di Benedetto, and Fabio Ganovelli. C-bdam - compressed batched dynamic adaptive meshes for terrain rendering, sep 2006. To appear in Eurographics 2006 conference proceedings.

[HLTC05] Hsien-Hsi Hsieh, Yueh-Yi Lai, Wen-Kai Tai, and Sheng-Yi Chang. A flexible 3d slicer for voxelization using graphics hardware. In *GRAPHITE '05: Proceedings of the 3rd international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 285–288, New York, NY, USA, 2005. ACM.

[Hop96] Hugues Hoppe. Progressive Meshes. In Holly Rushmeier, editor, *SIGGRAPH96*, CGPACS, pages 99–108, New York, 1996. ACM Press/ACM SIGGRAPH.

[KS00] Kiriakos N. Kutulakos and Steven M. Seitz. A theory of shape by space carving. *Int. J. Comput. Vision*, 38(3):199–218, 2000.

[LPC+00] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The Digital Michelangelo Project: 3D scanning of large statues. In *SIGGRAPH 2000, Computer Graphics Proceedings*, Annual Conference Series, pages 131–144. Addison Wesley, July 24-28 2000.

[MS95] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *In 1995 Symposium on Interactive 3D Graphics*, pages 95–102, 1995.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. 25(3):614–623, 2006.

[NB94] Wolfgang Niem and Ralf Buschmann. Automatic modelling of 3d natural objects from multiple views. In *In European Workshop on Combined Real and Synthetic Image Processing for Broadcast and Video Production*. Springer, 1994.

[RCM+01] C. Rocchini, P. Cignoni, C. Montani, P. Pingi, R. Scopigno, R. Fontana, L. Pezzati, M. Cygielman, R. Giachetti, and G. Gori. 3D scanning the Minerva of Arezzo. In *ICHIM'2001 Conf. Proc., Vol.2*, pages 265–272. Politecnico di Milano, 2001.

[Ros06] Randi J. Rost. *OpenGL(R) Shading Language (2nd Edition)*. Addison-Wesley Professional, January 2006.

[SG02] Neumann G. Arvidson R. E. Smith, D. and E. A. Guinness. Mars global surveyor laser altimeter mission experiment gridded data record. In *NASA Planetary Data System, MGS-M-MOLA-5-MEGDR-L3-V2.0*, 2002.

[SSRS74] Ivan E. Sutherland, Robert F. Sproull, Robert, and A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6:1–55, 1974.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[WDS99] Mason Woo, Davis, and Mary Beth Sheridan. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. In *In Computer Graphics (SIGGRAPH '78 Proceedings*, pages 270–274, 1978.