UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

# GENERAZIONE PROCEDURALE DI CONTENUTI E TECNICHE DI RENDERING REAL-TIME

# PROCEDURAL CONTENT GENERATION AND REAL-TIME RENDERING TECHNIQUES

ANDREA VENUTA

Relatore: *Pierluigi Crescenzi*

*Anno Accademico 2014-2015*

# INDICE

# ELENCO DELLE FIGURE

# 1 INTRODUZIONE

Con l'espressione *generazione procedurale di contenuti* si fa riferimento all'insieme di tecniche volte a produrre informazione a partire da una sua rappresentazione in forma algoritmica implicita. Queste tecniche esibiscono la proprietà di *database amplification*: a partire da un insieme di regole relativamente piccolo è possibile ottenere una quantità di informazione arbitrariamente grande. Un'altra proprietà degli algoritmi di generazione procedurale, in particolare quelli basati sulla generazione numerica pseudocasuale, è la serendipità: non è raro, durante la stesura di un algoritmo, trovarsi di fronte a risultati tanto piacevoli quanto imprevisti.

Grazie a queste caratteristiche, la generazione procedurale di contenuti ha trovato largo impiego nell'industria videoludica; ai suoi albori era una necessità dettata dalla scarsa disponibilità di risorse computazionali, mentre attualmente è utilizzata sia come strumento di supporto agli artisti nella creazione di modelli, immagini o suoni, sia come elemento caratterizzante del *gameplay* (alcuni esempi di questo sono titoli quali *Minecraft* e *Terraria*, o pressoché l'intero genere *roguelike*).

Un ambito in cui la generazione procedurale di contenuti ebbe un ruolo cardinale fu quello della *demoscene*, una sottocultura artistica ed informatica nata nei tardi anni '70 con l'obiettivo di produrre presentazioni audiovisive nella forma di software di dimensioni ridotte (*demo*). Validi esempi figurano nell'opera del gruppo tedesco *Farbrausch*; tra questi, il videogioco *.kkrieger*, un *first-person shooter* contenuto in circa 96*kb*, e la produzione *fr-041: debris*.

A quest'ultima è in parte ispirato il progetto descritto in questa tesi: una animazione renderizzata in tempo reale e realizzata, in ogni suo componente, mediante tecniche di generazione procedurale. L'animazione mostrerà una scena urbana notturna, per poi spostarsi all'interno di un appartamento e quindi chiudersi, ciclicamente, su di un monitor che mostra nuovamente la scena urbana.

Nel progetto si sono ricercate e studiate le tecniche di generazione procedurale più adeguate a rappresentare i contenuti dell'animazione. Questo ha portato a definire un formalismo ibrido tra gli *open L-system parametrici* [11] e le *shape grammar* [1] che permettesse di elaborare contemporaneamente informazioni visive a più livelli di astrazione, così da non dover separare l'aspetto di un contenuto dal suo significato. Sulla base del formalismo sono stati progettati e modellati tutti gli oggetti dell'animazione. È stato inoltre realizzato un

motore di *rendering* in tempo reale sfruttando tecniche classiche e moderne e ricercando un equilibrio tra *performance* e resa visiva. Al suo interno sono state implementate funzioni per il disegno procedurale di *texture* da applicare ai modelli; queste sono state analizzate in base ai principi della teoria dell'elaborazione dei segnali al fine di prevenire o correggere fenomeni indesiderati quali l'*aliasing* con il minimo impatto possibile in termini di costo computazionale.

Nel Capitolo 2 si tratteranno gli argomenti relativi alla modellazione procedurale, ovvero tutti gli aspetti che concorrono alla generazione delle *mesh di poligoni* rappresentanti gli oggetti della scena, ed alcune strutture più astratte (ad esempio, i grafi rappresentanti le strade od il modo in cui le stanze di un appartamento sono collegate tra di esse).

Nel Capitolo 3 si affronteranno invece le tecniche ed i principi del *rendering real-time* che consentono di trasformare in *pixel* su schermo l'informazione precedentemente accumulata per mezzo della modellazione procedurale, insieme a quella generata direttamente dagli *shader* contestualmente al *rendering*.

Nel Capitolo 4 verranno analizzate le tecniche analitiche e numeriche di *texturing* procedurale impiegate per definire dettagli, materiali e colori dei modelli generati.

In Appendice A si discuteranno le ragioni per la scelta delle tecnologie impiegate, e si introdurranno i fondamenti di *WebGL* su cui si basano molte delle tecniche e dei principi implementati nel progetto e discussi in questa tesi.

# 2 MODELLAZIONE PROCEDURALE

## 2.1 CONCETTI

Lo scopo di un algoritmo di modellazione procedurale è quello di pervenire ad un'*approssimazione realistica* di ciò che si va a modellare. Naturalmente, non è possibile dare una definizione organica ed obiettiva del concetto di *realismo*; è quindi generalmente difficile quantificare l'accuratezza di un algoritmo di modellazione procedurale. Emergono così due approcci fondamentali alla modellazione procedurale: quello *teleologico* e quello *ontogenetico* [18].

L'approccio *teleologico* si basa sulla modellazione accurata dei fenomeni che hanno generato un oggetto. Costruire un paesaggio montuoso simulando su larga scala la tettonica delle placche e l'erosione da parte degli agenti atmosferici costituisce un processo teleologico. In contrasto, l'approccio *ontogenetico* si limita a studiare l'aspetto e le caratteristiche del risultato, tentando di riprodurlo algoritmicamente. Costituisce un processo ontogenetico l'impiego di una primitiva di *gradient noise*, come quella sviluppata da Ken Perlin nel 1985 [6], per simulare l'altitudine di una catena montuosa.

Entrambi gli approcci hanno pregi e difetti, nonché contesti in cui sono più o meno adeguati. Generalmente, si ricercherà un approccio teleologico laddove sia richiesta maggiore accuratezza - si pensi ad un simulatore di guida - a discapito dell'impiego di risorse, mentre si preferirà un sistema ontogenetico quando si voglia perseguire un risultato non necessariamente corretto, ma sufficientemente adeguato allo scopo, e con minore impiego di risorse; è il caso del progetto qui descritto.

I modelli costruiti nel progetto sono strutturalmente molto simili tra di essi: vi sono essenzialmente solo edifici, balconi, strade, un appartamento ed alcuni oggetti d'arredamento. Per gli oggetti di ciascuna di queste categorie, tuttavia, servono numerose varianti: se così non fosse, si otterrebbe una città composta da edifici identici tra loro. Modellare un elevato numero di oggetti solo leggermente diversi tra loro, inoltre, è un'attività dispendiosa in termini sia di tempo, sia di spazio, ed è una delle ragioni pragmatiche principali per l'adozione di tecniche procedurali.

Nel linguaggio naturale, è solitamente semplice descrivere oggetti riuscendo a comunicare informazioni sufficienti affinché l'interlocutore riesca a figurarsele in maniera più o meno accurata. Si pensi di voler descrivere un palazzo specifico: supponendo che l'interlocutore sappia cosa sia un palazzo, se ne può

comunicare il numero di piani, la disposizione delle finestre, il colore della facciata, la forma, la presenza di un giardino. Pressoché chiunque è familiare con il concetto di palazzo, e con i concetti dei vari costituenti elencati; non altrettanti, invece, avranno competenze di architettura o di ingegneria edile utili a comprendere descrizioni più dettagliate e con maggiori implicazioni. I calcolatori, invece, non hanno nozioni nè architettoniche, nè ingegneristiche, nè di cultura generale; ma sarebbe senz'altro molto utile poter impartire al calcolatore l'istruzione di generare un edificio con 7 piani, con pareti di mattoni, finestre equidistanti ed un balcone su di una facciata per ciascun piano, ed ancor più utile poter generare un numero arbitrario di simili descrizioni in maniera automatica.

La modellazione procedurale nel presente progetto si articola quindi in due fasi: nella prima si formalizza una struttura in grado di costruire oggetti a partire da una qualche descrizione di questi, nella seconda si generano descrizioni adeguate degli oggetti per poi fornirle alla struttura, così da ottenere gli oggetti desiderati.

Una fondamentale problematica di questo approccio è quella di dover esprimere, elaborare ed interpretare concetti *visuali*: è necessario quindi ideare ed implementare un'astrazione adeguata allo scopo.

## 2.2 SHAPE GRAMMAR

Introdotte nel 1972 da G. Stiny e J. Gips, le *shape grammar* [1] sono un tipo di grammatica generativa la cui peculiarità è quella di operare su oggetti geometrici anziché su stringhe.

**Definizione 2.2.1** (Grammatica generativa)**.** Una *grammatica generativa* è una quadrupla $\mathcal{G} = \langle V, T, S, P \rangle$, dove

1. $V$ è l'insieme dei *simboli non terminali*,
2. $T$ è l'insieme dei *simboli terminali*,
3. $S \in V$ è detto *assioma*, e
4. $P \subseteq (V \cup T)^* \times (V \cup T)^*$ è l'insieme delle *regole di produzione*. Una produzione $p \in P$ può essere indicata come $\alpha \to \beta,\ \alpha, \beta \in (V \cup T)^*$.

Il compito di una grammatica generativa è quello di generare tutte e sole le stringhe di un *linguaggio*, ovvero un insieme di stringhe di simboli che seguono regole arbitrarie, specifiche per un dato linguaggio. Ad esempio, indichiamo informalmente il linguaggio $L$ come "l'insieme delle stringhe composte solo dalla lettera *a*", oppure, più formalmente, come $L = \{a^i : i > 0\}$. Una grammatica generativa per questo linguaggio, diciamo $\mathcal{G}_L$, potrebbe essere definita come segue:

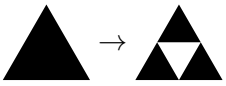
- $V = \{A\}$,
- $T = \{a\}$,
- $S = A$,
- $P = \{A \to aA;\ A \to a\}$

Il linguaggio L viene generato da $\mathcal{G}_L$ applicando ripetutamente le produzioni in P, per prima cosa al simbolo iniziale $S \equiv A$, e successivamente ad ogni stringa di simboli (se non tutti terminali) generata dall'applicazione di una produzione. Applicare una produzione ad una stringa di simboli significa, semplicemente, rimpiazzare le istanze della *parte sinistra* della produzione con la rispettiva *parte destra*. Ad esempio, una possibile stringa in L generata da $\mathcal{G}_L$ è

$$A \to aA \to aaA \to aaaA \to aaaaA \to aaaaa$$

e si ottiene applicando per quattro volte la produzione $A \to aA$, e per una volta la produzione $A \to a$, in seguito alla quale non esistono più simboli non terminali su cui applicare ulteriormente produzioni.

Le *shape grammar* operano in maniera simile ma, anziché basarsi sui simboli, generano forme geometriche a partire da altre forme geometriche. Ad esempio, è possibile generare il *triangolo di Sierpiński* usando una *shape grammar* con una sola regola: "suddividi un triangolo equilatero in tre triangoli equilateri"; oppure, usando una forma più consona,



In figura 1 è rappresentato il risultato della generazione da parte di questa grammatica.

La peculiarità delle *shape grammar* giace nel fatto di operare in un ambito *visuale* molto difficile da ricondurre ad un contesto computazionale od anche, più generalmente, verbale. Alla definizione verbale data per la semplice regola di cui sopra mancano, infatti, alcuni dettagli fondamentali che la rendono

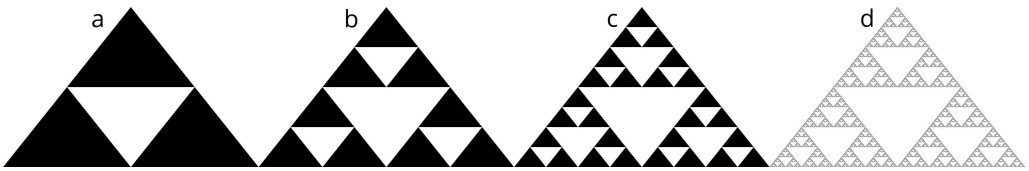



Figura 1: Risultato della *shape grammar* per il triangolo di Sierpiński con (a) 1, (b) 2, (c) 3 e (d) 10 iterazioni

molto ambigua: non specifica, ad esempio, se i tre triangoli risultanti dalla suddivisione debbano mantenere una certa rotazione oppure se debbano rimanere vicini, o distaccarsi, od avere necessariamente proprietà in comune. Sopperire a queste mancanze molto evidenti è naturalmente possibile, ma la percezione delle forme è un fenomeno soggettivo, ed un osservatore a cui venisse fornita solo una descrizione verbale della grammatica potrebbe applicarla in maniera radicalmente differente rispetto ad un altro osservatore, se uno dei due percepisse aspetti visuali della descrizione fornita che all'altro sono sfuggiti, anche a fronte di una descrizione molto dettagliata. Si considerino, inoltre, illusioni ottiche quali il cubo di Necker [25], interpretabili in più modi anche da un unico osservatore: in questo caso anche la rappresentazione grafica presenta delle ambiguità.

L'implementazione di una *shape grammar* nella sua accezione più intuitiva si rivela quindi un'operazione molto complessa, in quanto non è possibile formalizzare tutti gli aspetti della percezione delle forme, nè implementare sistemi di riconoscimento inequivocabilmente accurati senza imporre delle limitazioni.

Il formalismo impiegato nel presente progetto arricchisce perciò i simboli di una *shape grammar* di un significato semantico, rendendone triviale l'implementazione del passo di selezione ed applicazione delle produzioni, in cambio della ridotta flessibilità derivante dalla mancata possibilità di rilevare parti sinistre delle produzioni in senso strettamente geometrico. Esso si ispira agli *open L-system*, introdotti da R. Mech e P. Prusinkiewicz nell'articolo *Visual Models of Plants Interacting with Their Environment* [11], ed è il sistema di riscrittura parallelo delineato nella seguente

**Definizione 2.2.2** (Shape grammar simbolica)**.** $\mathbf{G} = \langle V, T, D, S, P \rangle$ è una *shape grammar simbolica* se e solo se

1. $V$ è l'insieme di *simboli non terminali*. Un simbolo non terminale $v \in V$ è un'arbitraria stringa di caratteri;

2. $T$ è l'insieme di *simboli terminali*. Un simbolo terminale $t \in T$ è un'arbitraria stringa di caratteri;

3. $D$ è un dizionario di coppie $\langle v, i \rangle$, definite *simboli estesi*, tali che $v \in V \cup T$, ed $i$ è il metadato che descrive la codifica di un arbitrario insieme di informazioni semanticamente collegato a $v$. Ad ogni istante $j$ della computazione, $D_j$ è lo *stato globale* della computazione, ovvero un insieme ordinato di coppie $\langle d_v, d_i \rangle$. Se $d \in D_j$, allora $d_v$ rappresenta la componente $v$ e $d_i$ rappresenta un'istanza di informazione codificata secondo la componente $i$; inoltre, $d_v$ non è necessariamente unico in $D_j$ (mentre, per ogni $v \in V \cup T$, $d_v$ è unico in $D$).

4. $S \in D_0,\ D_0 = \{S\}$ è detto *assioma*;

5. P è un insieme ordinato di triple $\langle d, c, f \rangle$, dette *produzioni*, in cui
   a) $d \in D$, $d_v \in V$,
   b) $c$ è un predicato che, con *input* $D_j$ e $d_i$, calcola la *condizione* di applicazione della produzione, ovvero decide se la produzione verrà applicata o scartata,
   c) $f$ è un algoritmo che, con *input* $D_j$ e $d_i$, produce in *output* zero o più coppie $d^k = \langle v_k, i_k \rangle$ tali che $d^k \in D_{j+1}$.

   Si indica con $p^k$ la k-esima produzione.

6. Se $n$ è il numero di iterazioni parallele necessario a terminare una computazione, allora $\exists d \in D_n : d_i$ è una *mesh*[1].

La definizione è volutamente poco restrittiva in proposito al formato delle codifiche nei *simboli estesi*. Questo concede molta libertà nella definizione ed implementazione delle regole di produzione, ma richiede particolare attenzione, da parte dell'implementatore di una grammatica, alla coerenza dei dati. È auspicabile progettare la grammatica pianificando accuratamente le strutture dati di ogni simbolo esteso.

Segue l'algoritmo ricorsivo che implementa la *shape grammar simbolica*.

```
1:  function ShapeGrammar(D_j)
2:      for all d ∈ D_j do
3:          if d_v ∈ T then
4:              D_{j+1} = D_{j+1} ∪ {d}
5:          else
6:              P' = {p ∈ P : d_v = d_v^p ∧ c^p(D_j, d_i)}
7:              p = min_k p^k ∈ P'
8:              D_{j+1} = D_{j+1} ∪ f^p(D_j, d_i)
9:          end if
10:     end for
11:     if ∃d ∈ D_{j+1} : d_v ∈ V then return ShapeGrammar(D_{j+1})
12:     else return D_{j+1}
13:     end if
14: end function
```

L'implementazione del sistema si avvale del concetto di funzioni *first-class*, ed è un *domain-specific language* interno al linguaggio di programmazione utilizzato per implementarlo. In questo modo, predicati ed algoritmi generatori dei successori nelle produzioni possono essere definiti mediante una sintassi Turing completa, e possono avvalersi sia di tutti gli algoritmi e le strutture dati *built-in* del linguaggio, sia di librerie esterne. È stato così possibile arricchire

[1] Si veda la Definizione A.2.2.

le grammatiche rendendo disponibili alcuni algoritmi utili a manipolare forme geometriche.

Questi algoritmi operano su due tipi di dati primitivi: il *cammino* ed il *quadrilatero*.

**Definizione 2.2.3** (Cammino)**.** $P$ è un *cammino* (*path*) di lunghezza $n+1$ se e solo se

$$P = p_0 p_1 \cdots p_n p_{n+1}, \quad p_i \text{ è un vertice } \forall\, i = 0 \ldots n, \quad p_0 \equiv p_{n+1}$$

**Definizione 2.2.4** (Quadrilatero)**.** $Q$ è un *quadrilatero* (*quad*) se e solo se $Q$ è un *cammino* di lunghezza 4 ed i vertici di $Q$ sono coplanari.

Gli algoritmi sono i seguenti:

- EXTRUDE($v, P, l, \mathbf{n}$): genera un quadrilatero $q_i$ estrudendo ogni segmento $p_i p_{i+1}$ in direzione del vettore normale $\mathbf{n}$ per una lunghezza $l$, e restituisce in *output* $n$ istanze di simboli estesi $\langle v, q_i \rangle$.

- SPLIT($V, Q, S, S', \mathbf{t}$): suddivide il quadrilatero $Q$ in $|S \times S'| \equiv |V|$ quadrilateri. $S$ ed $S'$ sono due insiemi ordinati di numeri in $[0, 1]$ tali che

  $$\sum_{s \in S} s = 1, \quad \sum_{s \in S'} s = 1.$$

  $\mathbf{t}$ è il vettore, tangente alla superficie di $Q$, nella cui direzione viene partizionato il quadrilatero secondo la distribuzione definita da $S$; $\mathbf{t}' = \mathbf{t} \times \mathbf{n}$, dove $\mathbf{n}$ è il vettore normale alla superficie di $Q$, è il vettore nella cui direzione viene partizionato il quadrilatero secondo la distribuzione definita da $S'$. A ciascuno dei risultanti $|S \times S'|$ quadrilateri $q_i$ viene associato un $v \in V$, ordinatamente prima secondo $S'$, poi secondo $S$, andando a creare le istanze di simboli estesi $\langle v_i, q_i \rangle$.

- FIT($v, Q, \mathbf{t}, r$): suddivide il quadrilatero $Q$ nella direzione del vettore $\mathbf{t}$, tangente alla sua superficie, nel numero di parti necessario per ottenere quadrilateri uguali il cui rapporto tra altezza e ampiezza si avvicini il più possibile a $r$, sia esso $n$. Restituisce quindi $n$ istanze di simboli estesi $\langle v, q_i \rangle$.

- INSET($P, l$): costruisce e restituisce un nuovo cammino $P'$ ottenuto "avvicinando" tra loro i segmenti $p_i p_{i+1}$ di una quantità $l$.

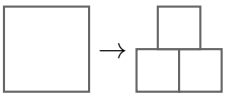
- BOUNDS($P$): calcola il volume di delimitazione del cammino $P$.

Con il formalismo così definito, è possibile riscrivere la *shape grammar* per il triangolo di Sierpiński:

- $V = \{\textbf{Triangolo}\}, T = \{\textbf{TriangoloTerm}\}$
- $D = \{\langle \textbf{Triangolo}, \{x, y, w, h\}\rangle, \langle \textbf{TriangoloTerm}, \{p_0, p_1, p_2\}\rangle\}$
- $S = \langle \textbf{Triangolo}, \{0, 0, 1, 1\}\rangle$
- $P = \{$
  - $p^1 : \langle \textbf{Triangolo}, \{x, y, w, h\}\rangle$ : *condizione di terminazione non soddisfatta* $\rightarrow$
$$\left\langle \textbf{Triangolo}, \left\{ x + \frac{w}{4}, y, \frac{w}{2}, \frac{h}{2} \right\} \right\rangle,$$
$$\left\langle \textbf{Triangolo}, \left\{ x, y + \frac{h}{2}, \frac{w}{2}, \frac{h}{2} \right\} \right\rangle,$$
$$\left\langle \textbf{Triangolo}, \left\{ x + \frac{w}{2}, y + \frac{h}{2}, \frac{w}{2}, \frac{h}{2} \right\} \right\rangle$$
  - $p^2 : \langle \textbf{Triangolo}, \{x, y, w, h\}\rangle$ : *condizione di terminazione soddisfatta* $\rightarrow$
$$\left\langle \textbf{TriangoloTerm}, \left\{ (x, y + h), (x + w, y + h), \left( x + \frac{w}{2}, y \right) \right\} \right\rangle$$

$\}$

La condizione di terminazione può essere, ad esempio, un limite sul numero di passate; finché questa non è verificata, ogni **Triangolo** continuerà a produrre tre simboli non terminali **Triangolo**, ed i valori per $x, y, w, h$ (ovvero posizione verticale, posizione orizzontale, larghezza ed altezza del minimo rettangolo circoscritto al triangolo equilatero) saranno calcolati opportunamente per ciascuno dei tre, in modo tale da averne uno in basso a sinistra, uno in basso a destra ed uno in alto al centro.

$p^1 :$  $\rightarrow$ 

Quando la condizione di terminazione è verificata, ogni simbolo non terminale contenuto in $D_k$ (se $k$ è il momento in cui la condizione si verifica) viene trasformato in un triangolo, i cui vertici sono in basso a sinistra $(x, y + h)$, in basso a destra $(x + w, y + h)$ ed in alto al centro $\left(x + \frac{w}{2}, y\right)$ rispetto al rettangolo.

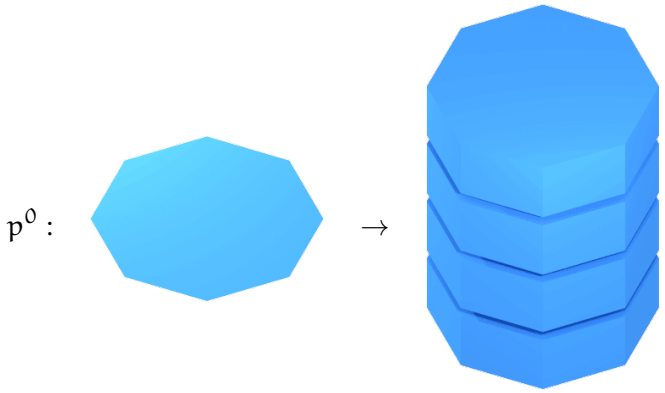
$p^2 :$  $\rightarrow$ 

Trattando dunque le forme come coppie tra un simbolo semantico ed un insieme di dati fondamentali alla sua rappresentazione, è possibile astrarre il meccanismo di calcolo visuale mantenendo una gerarchia robusta, il che può essere una limitazione – è difficile riconoscere figure singole generate da insiemi di altre figure – oppure un vantaggio – si previene l'ambiguità tipica del ragionamento visuale.

### 2.2.1 *Grammatica per gli edifici*

Un *edificio* può essere descritto, in linguaggio naturale, come una serie di *piani* sovrapposti, due dei quali hanno un significato speciale: quello più vicino al suolo, che contiene un ingresso, e quello più in alto, che ha un tetto; a vari livelli si possono trovare inoltre dei *cornicioni* che sporgono leggermente dall'edificio. Ciascun piano è una figura solida il contorno della cui base segue quello delle fondamenta dell'edificio (ad esempio, in un edificio rettangolare ogni piano avrà base rettangolare). Ciascuna faccia laterale di questa figura è un quadrilatero, e prende il nome di *facciata*. Questa, a sua volta, può contenere un numero arbitrario di *finestre*, che sono anch'esse quadrilateri ricavati dentro la facciata, la cui altezza ed ampiezza sono generalmente in proporzioni tra 1 : 1 e 2 : 1 (valori inferiori o superiori di questo intervallo sono possibili, ma insoliti) e la cui posizione verticale è al centro della facciata. Almeno una facciata del piano terreno deve presentare una *porta d'ingresso*, la quale somiglia ad una finestra ma è significativamente più grande e posizionata in basso, per permettere l'accesso. Una facciata può anche presentare dei *balconi*, o delle scale d'emergenza.

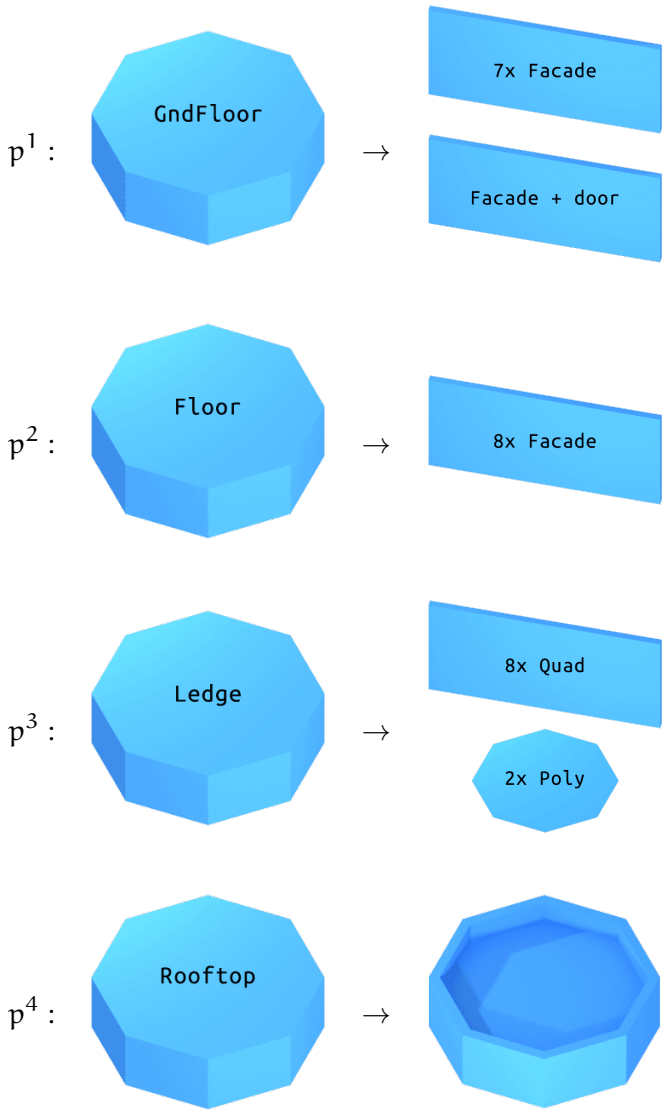
I sostantivi evidenziati rappresentano dunque i simboli della grammatica generatrice dei palazzi. L'assioma è il simbolo **Building**, a cui è associata l'informazione relativa alla forma poligonale delle fondamenta (un *cammino*[2]) e la lista dei piani con relative caratteristiche, e produce un numero di piani formati come le sue fondamenta; di questi, il primo e l'ultimo saranno sicuramente simboli **GndFloor** e **Rooftop**, ovvero piano terreno e tetto, la maggior parte saranno **Floor**, ovvero piani regolari, ed alcuni saranno **Ledge**, ovvero cornicioni (che sono strutturalmente identici ai piani, ma più bassi e con quadrilateri semplici al posto delle facciate).




Ognuno dei quattro simboli relativi ai piani, ovvero **GndFloor**, **Floor**, **Ledge** e **Rooftop**, produce tante facciate quanti sono i lati del poligono delle fonda-

[2] Nelle immagini della grammatica viene rappresentato come un ottagono al fine di rappresentarne meglio alcune peculiarità, ma può essere arbitrario.

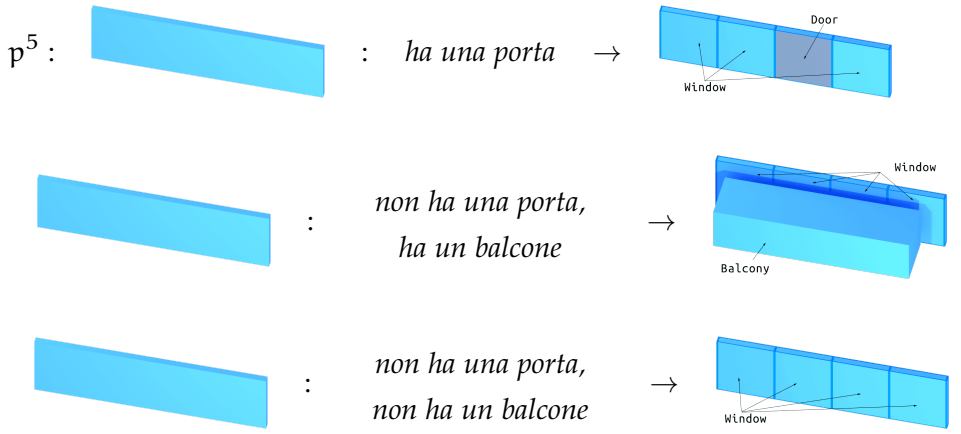
menta. Ciascuno di essi lo fa in maniera leggermente diversa: **GndFloor** e **Floor** generano simboli **Facade**, con la differenza che una di queste generate dal primo ha anche una porta; **Ledge** genera simboli **Quad** e **Poly**, ovvero semplici quadrilateri (per i lati) e poligoni (per le parti alla base ed alla sommità) terminali; **Rooftop**, invece, crea un rialzo mediante gli operatori INSET ed EXTRUDE, e genera simboli **Quad** e **Poly**[3].



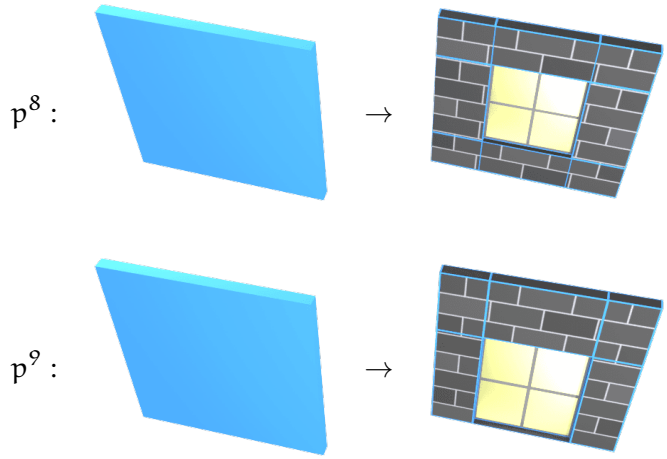

Una facciata è essenzialmente costituita dall'informazione di un quadrilatero, e produce pannelli circa quadrati mediante l'operazione FIT. È parte sinistra di tre produzioni, corrispondenti a tre possibili combinazioni dei predicati *ha una*

[3] Questi non sono stati rappresentati direttamente nella produzione, poiché sono in numero elevato rispetto ad altri simboli.

*porta* ed *ha un balcone*, controllati dal simbolo predecessore della facciata. Specificamente, *ha una porta* è vero solo per uno dei successori di **GndFloor**, mentre *ha un balcone* è vero solo per uno dei successori di ciascun **Floor**. Se la facciata *ha una porta*, allora uno ed uno solo dei pannelli generati, più vicino possibile al centro, sarà un simbolo di tipo **Door**, mentre gli altri saranno **Window**. Se la facciata *ha un balcone*, viene aggiunto un balcone davanti ad essa. Se, infine, la facciata non *ha una porta*, tutti i pannelli generati saranno simboli **Window**.




Le ultime due produzioni hanno come parti sinistre **Window** e **Door**. Entrambe utilizzano l'operatore SPLIT per suddividere il pannello sia orizzontalmente, sia verticalmente, ed in seguito l'operatore EXTRUDE per ricavare una nicchia che fungerà da porta o finestra.

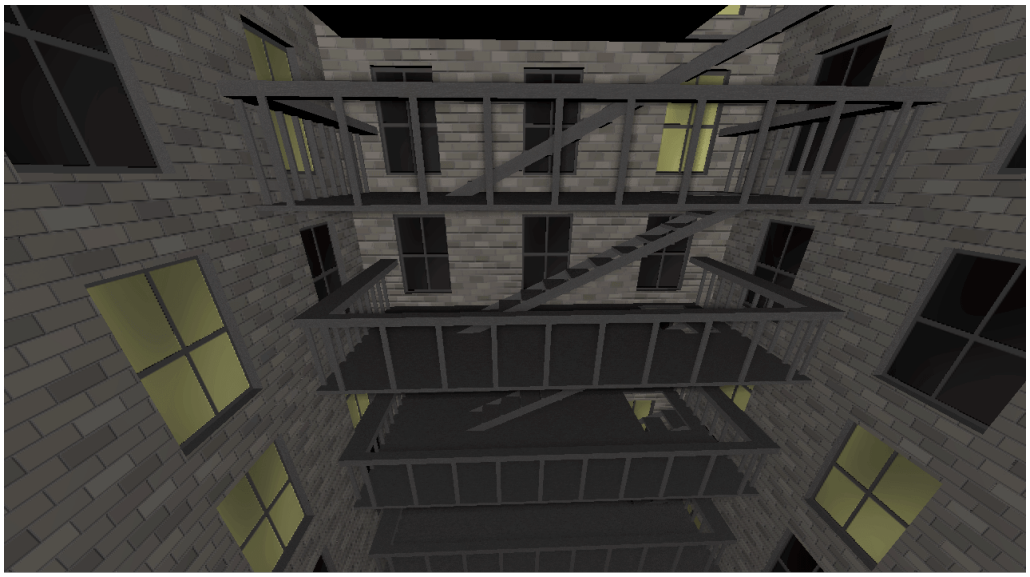


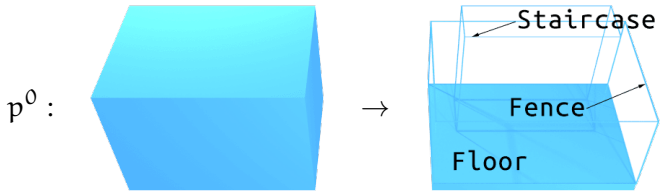

### 2.2.2 *Grammatica per i balconi*

Un balcone è costituito da un *pavimento* rettangolare, tre dei cui lati sono protetti da una *ringhiera*, mentre il quarto è vuoto e dà sulla facciata. Il pavimento

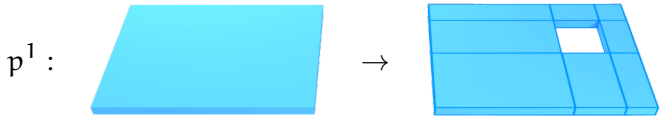
presenta un'apertura dalla quale discende una *scala* che lo collega al balcone del piano inferiore (se ve ne è uno). La ringhiera è composta da *barre* di metallo strette ed alte, unite in cima da un *corrimano*.

L'assioma di questa grammatica è **Balcony**, al quale si associano un volume delimitante e informazioni sui vari componenti, quali l'altezza del parapetto o lo spessore del pavimento. Esso produce i simboli **Floor**[4], **Staircase** e **Fence**.



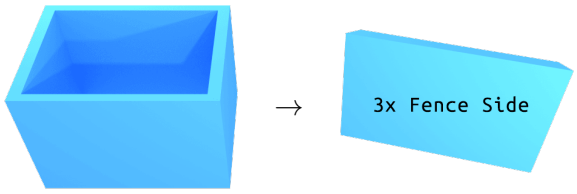



Il simbolo **Floor** si avvale quindi degli operatori SPLIT ed EXTRUDE per ricavare l'apertura per la scalinata, e produce simboli terminali **Quad**.

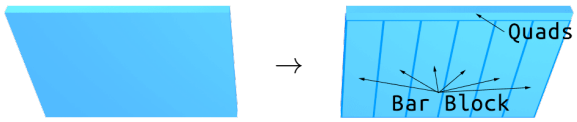



Tre delle facce laterali del parallelepipedo delineato dal parapetto, simbolo **Fence**, vengono estruse verso l'interno, andando a creare tre volumi sottili, ognuno dei quali rappresenta un segmento del parapetto; i simboli risultanti sono tre **FenceSide**.

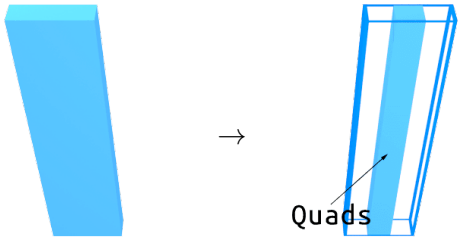
[4] Distinto dall'omonimo simbolo nella grammatica per gli edifici.

$p^2$ : 


Ciascun lato della ringhiera viene prima suddiviso verticalmente mediante l'operatore SPLIT, andando a creare un corrimano composto da simboli terminali, quindi orizzontalmente mediante l'operatore FIT, creando un adeguato numero di simboli **BarBlock** alti e stretti.

$p^3$ : 


Questi verranno ulteriormente suddivisi orizzontalmente in tre parti, delle quali verranno scartate le due più esterne. La parte rimanente verrà prodotta come insieme di simboli terminali **Quad**.

$p^4$ : 


Il volume del simbolo **Staircase**, infine, viene suddiviso sia orizzontalmente, sia verticalmente in maniera adeguata tramite l'operatore FIT, e vengono scartati tutti i blocchi esterni alla diagonale; ciascuno di questi viene ulteriormente suddiviso, tramite l'operatore SPLIT, lungo l'asse verticale, e vengono scartati i due blocchi più esterni. Questi blocchi vengono esportati come simboli terminali **Quad**; in aggiunta a questi, si costruiscono altri due **Quad**, i quali andranno a rappresentare il telaio che mantiene insieme gli scalini, unendo vertici del primo ed ultimo scalino.

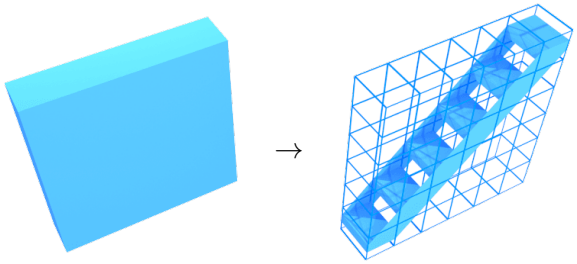
$p^5$ : 

Figura 2: Fotogramma dell'appartamento

### 2.2.3 *Grammatica per l'appartamento*

Parte dell'animazione è ambientata all'interno di un appartamento, con la telecamera che si muove per le varie stanze. La *shape grammar* per l'appartamento non produce, perciò, solamente la geometria di questo, ma anche informazioni utili a generare proceduralmente un percorso per la telecamera, nella forma di un grafo connesso e non orientato.

**Definizione 2.2.5** (Grafo)**.** La coppia ordinata $G = (V, E)$ è un *grafo* se e solo se $E = V \times V$. $V$ si dice insieme dei *nodi*, $E$ si dice insieme degli *archi*.

**Definizione 2.2.6** (Grafo orientato)**.** Un grafo $G = (V, E)$ si dice *orientato* se e solo se esiste almeno un $e \in E,\ e = (u, v)$ tale che $(v, u) \notin E$.

**Definizione 2.2.7** (Cammino di un grafo[5])**.** Sia $G = (V, E)$ un grafo. Allora, la successione

$$\langle v_0, v_1, \cdots v_n \rangle, \quad v_i \in V$$

è detta *cammino* se e solo se $(v_i, v_{i+1}) \in E,\ 0 \leqslant i \leqslant n$.

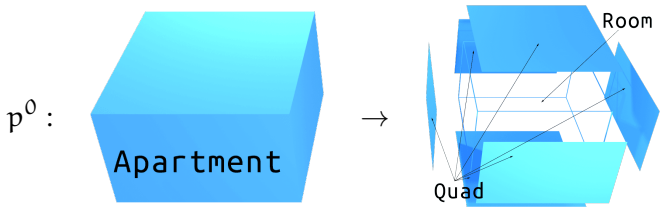
**Definizione 2.2.8** (Grafo connesso)**.** Sia $G = (V, E)$ un grafo. Allora, $G$ è detto *connesso* se e solo se per ogni $u, v \in V$ esiste almeno un cammino da $u$ a $v$.

[5] Concettualmente affine, ma distinta, dalla definizione 2.2.3.

La sola geometria dell'appartamento, infatti, non contiene informazioni semantiche sui concetti di *stanza* o di *porta*, fondamentali per concettualizzare le proprietà della telecamera di *essere in una stanza* ed *attraversare una porta*.

In questa grammatica, pertanto, l'insieme dei simboli terminali sarà più ricco: mentre le grammatiche viste finora producevano solo quadrilateri e poligoni (**Quad** e **Poly**), la grammatica per l'appartamento produrrà anche nodi ed archi di un grafo.

L'assioma della grammatica è **Apartment**, a cui è associato un volume delimitante ed informazioni sui limiti in termini di dimensioni e rapporto tra profondità e larghezza delle stanze di cui è composto (poiché, ad esempio, non è desiderabile avere stanze molto lunghe e strette, nè avere alcune stanze molto spaziose ed altre molto anguste). La prima produzione genererà dunque un quadrilatero per il pavimento, un quadrilatero per il soffitto, quattro quadrilateri per le mura esterne come simboli terminali **Quad**, ed un simbolo nonterminale **Room**. A questo verranno associati, inizialmente, lo stesso volume e gli stessi vincoli dell'appartamento.




Il compito del simbolo **Room** è quello di suddividersi il più possibile, fino a che il suo volume incontra condizioni di dimensione e proporzione che rispettano quelle imposte [22]. Vi sono quindi quattro produzioni aventi come parte sinistra il simbolo **Room**. Le prime due sono *ricorsive*, ossia producono simboli **Room** nella parte destra, ed il loro compito è quello di suddividere la stanza. Le restanti due produzioni, invece, producono simboli terminali e non terminali diversi da **Room**, ed interrompono il processo ricorsivo.

Le due produzioni ricorsive vengono selezionate a fronte del verificarsi di un predicato. Sia $c_s$ il predicato "*la stanza ha dimensioni superiori a quelle minime imposte*"; sia $c_r$ un predicato casuale, la cui veridicità può cambiare ad ogni valutazione, e che è vero nell'80% dei casi e falso nel 20%[6]. Sia $c_M$ il predicato "*la stanza ha dimensioni superiori a quelle massime imposte*". Sia infine $c_{w>d}$ il predicato "*la stanza ha larghezza superiore della profondità*".

La differenza tra la prima e la seconda regola è solo la direzione in cui suddividere la stanza; una delle due viene selezionata se è necessario suddividere la stanza, e si seleziona quella che suddivide in larghezza se la larghezza della

[6] Valori empirici che presentano consistentemente un risultato plausibile.

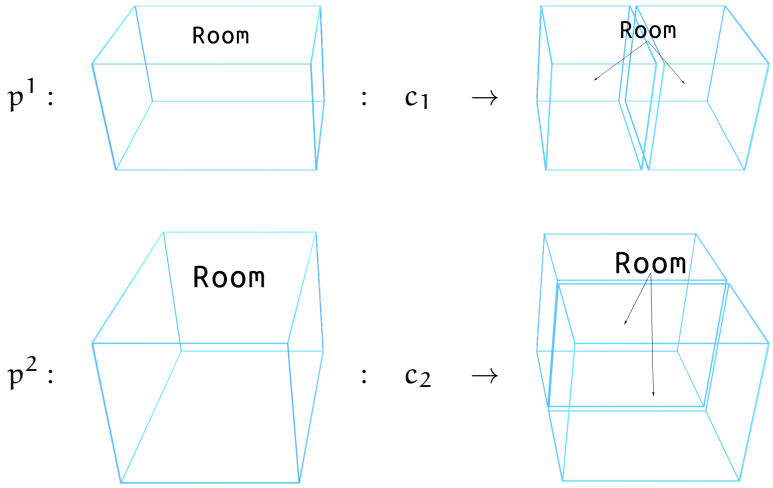
stanza è superiore alla profondità, e quella che suddivide in profondità altrimenti. In questo modo si evitano i casi, precedentemente delineati, di stanze eccessivamente strette o lunghe.

La terza e la quarta regola accettano invece la stanza; la differenza tra queste due, selezionate tramite un predicato $c_f$, "*la stanza supera le dimensioni minime per cui è accettabile arredarla*", che decide se inserire o meno alcuni elementi d'arredamento nella stanza (eccetto una lampada a soffitto, di cui tutte le stanze sono fornite).

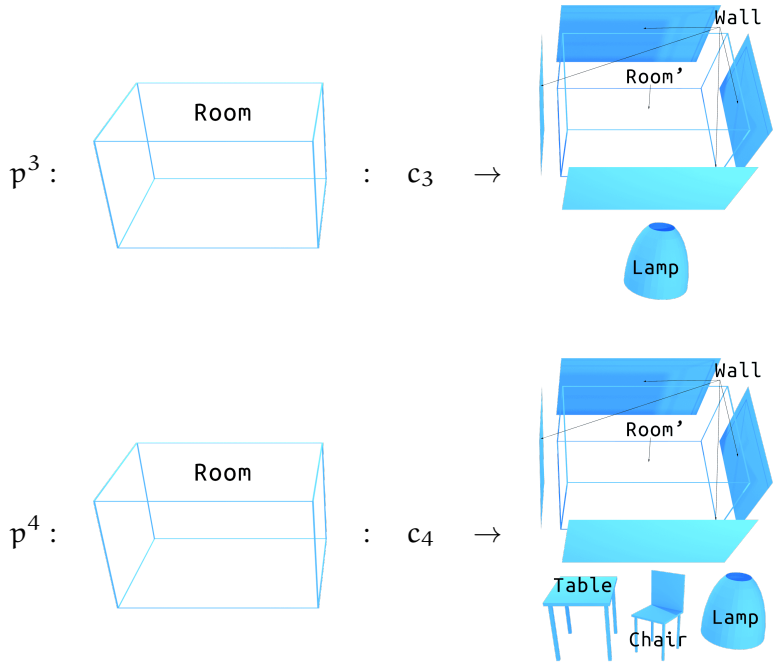
I predicati di selezione delle produzioni sono quindi dati da

$$c_1 = c_{w>d} \wedge ((c_s \wedge c_r) \vee c_M)$$
$$c_2 = \neg c_{w>d} \wedge ((c_s \wedge c_r) \vee c_M)$$
$$c_3 = \neg c_f \wedge \neg (c_1 \wedge c_2)$$
$$c_4 = c_f \wedge \neg (c_1 \wedge c_2)$$

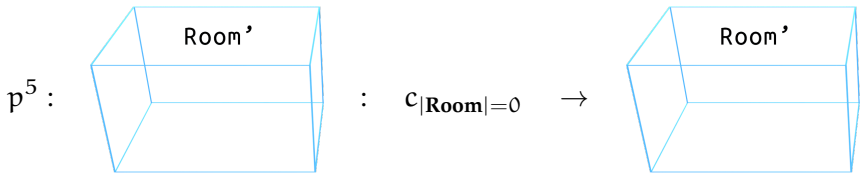
laddove $c_1$ seleziona la prima produzione, la quale suddivide la stanza nel verso della larghezza, $c_2$ seleziona la seconda produzione, la quale suddivide la stanza nel verso della profondità, e $c_3$ e $c_4$ selezionano la terza e quarta produzione, che accettano la stanza così com'è (ed, eventualmente, la arredano).




Le produzioni $p^3$ e $p^4$ generano anzitutto un simbolo **Room'**, concettualmente rappresentante anch'esso una stanza, ma che si comporta in maniera diversa; associato ad esso vi saranno le informazioni sul volume della stanza ed un nodo del grafo, rappresentante il centro di questa. Successivamente, generano quattro simboli **Wall**, rappresentanti i muri che circondano la stanza. Infine, generano entrambe un simbolo **Lamp** per la lampada a soffitto, e solo $p^4$ genera un simbolo **Table** e zero o più simboli **Chair** per tavoli e sedie.

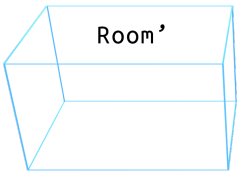



Il simbolo **Room'** compare in quattro parti sinistre. La prima produzione è un meccanismo di sincronizzazione a barriera: attende che tutti i possibili simboli **Room'** siano stati generati. Questa condizione equivale al non avere, nello stato globale della computazione, alcun simbolo **Room**, giacché questi sono gli unici a poter generare **Room'**. Si indica questa condizione col predicato $c_{|\mathbf{Room}|=0}$. L'attesa è implementata semplicemente producendo come parte destra esattamente la parte sinistra.

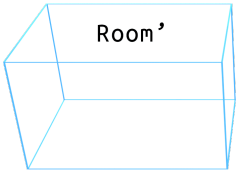



L'animazione avrà bisogno di essere a conoscenza di quale siano la prima e l'ultima stanza da visitare. Per fare questo, si definiscono due predicati $c_{\text{first}}$ e $c_{\text{last}}$ che indicano, rispettivamente, se la parte sinistra è la prima o l'ultima ad apparire nello stato globale. La produzione $p^6$, selezionata per la prima stanza, genererà un simbolo **Node** a cui sarà associato il nodo del grafo della parte sinistra **Room'** e marcato come primo. La produzione $p^7$, selezionata invece per l'ultima stanza, produrrà un simbolo speciale **LastRoom**, associando anche ad esso il nodo del grafo della parte sinistra. La produzione $p^8$, selezionata invece per tutte le stanze intermedie, produrrà semplicemente un elemento **Node** con le informazioni relative al nodo del grafo della parte sinistra, ma senza alcuna

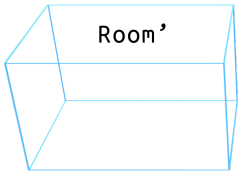
marcatura speciale. Si indica con **p** il punto nello spazio associato al nodo del grafo.

$p^6$ : 

: $c_{first} \rightarrow \langle \textbf{Node}, \{\textbf{p}, isFirst\}\rangle$

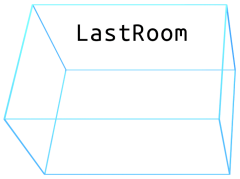
$p^7$ : 

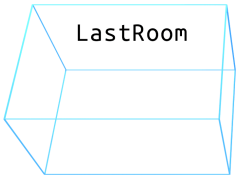
: $c_{last} \rightarrow \langle \textbf{LastRoom}, \textbf{p}\rangle$

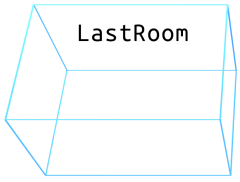
$p^8$ : 

: $\neg(c_{first} \vee c_{last}) \rightarrow \langle \textbf{Node}, \textbf{p}\rangle$

Il simbolo **LastRoom** compare come parte sinistra su due produzioni. La prima è una sincronizzazione a barriera come la precedente vista, e attende che il grafo sia stato interamente generato, controllando che non vi siano nello stato globale simboli **Wall** od **OccludedWall** (questo verrà discusso in seguito). Si indica questa condizione col predicato $c_{gg}$. La seconda produzione genererà un nodo marcato come finale (simbolo **Node**) ed un simbolo **Monitor** che rappresenta lo schermo dal quale verrà proiettato l'inizio dell'animazione ed al quale la telecamera dovrà avvicinarsi al termine dell'animazione.

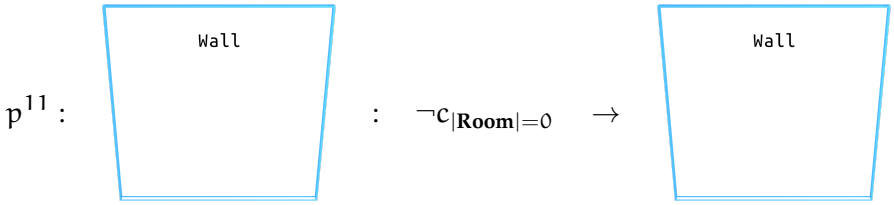
$p^9$ : 

: $\neg c_{gg} \rightarrow$ 


$p^{10}$ : 

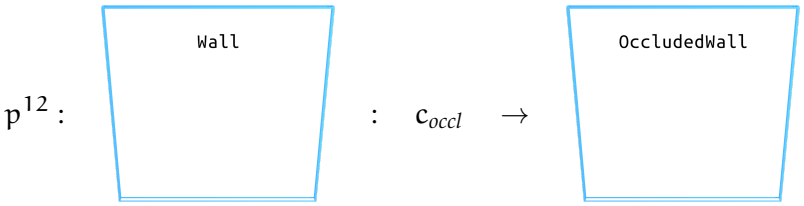
: $c_{gg} \rightarrow \langle \textbf{Node}, \textbf{p}\rangle, \langle \textbf{Monitor}, g\rangle$

Anche nel caso del simbolo **Wall** si presenta una sincronizzazione a barriera, che attende che tutte le stanze siano state generate, il che equivale ad attendere che tutti i muri siano stati generati.

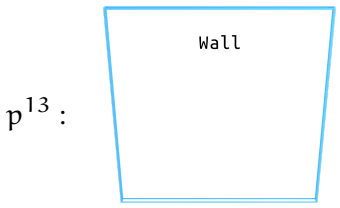



$$p^{11} : \quad : \quad \neg c_{|\mathbf{Room}|=0} \quad \rightarrow$$

A questo punto, si valuta una condizione, diciamo $c_{occl}$, su ciascuna parte sinistra; questa è vera se e solo se il muro della parte sinistra *collide*, o si sovrappone in parte, con un muro più grande di esso. Se questa condizione si verifica, la parte sinistra viene trasformata in un simbolo **OccludedWall** avente le stesse informazioni, e questo viene prodotto; altrimenti, si produce un simbolo vuoto. La ragione per questo è che ogni muro collide con almeno un altro muro, ad eccezione di quelli più esterni che sono comunque già stati generati, e che mantenendo solo i muri più corti si possono ricavare da questi semplicemente le porte, mentre è difficile stabilire la posizione delle porte partendo da muri più grandi [22]. Il simbolo **OccludedWall** prodotto contiene i due nodi corrispondenti alla stanza del muro occludente ed alla stanza del muro occluso.

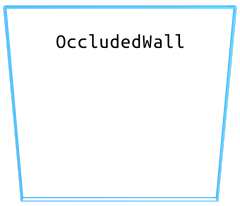



$$p^{12} : \quad : \quad c_{occl} \quad \rightarrow$$




$$p^{13} : \quad : \quad \neg c_{occl} \quad \rightarrow \quad \lambda$$

Il simbolo **OccludedWall** genera simboli terminali. Dalle informazioni geometriche del muro viene generato un insieme di **Quad** che rappresenta il muro all'interno del quale è stato ricavato lo spazio per una porta. Dai nodi delle due stanze adiacenti, invece, vengono generati quattro archi $(r_1, d)$, $(r_2, d)$, $(d, r_1)$, $(d, r_2)$ (simbolo **Arc**) che collegano le due stanze ($r_1$ ed $r_2$) ad un nuovo simbolo **Node** corrispondente alla porta ($d$).

$p^{13}$ : 

$\rightarrow$  ,

$\langle \textbf{Node}, d \rangle$ ,
$\langle \textbf{Arc}, (r_1, d) \rangle$ , $\langle \textbf{Arc}, (d, r_1) \rangle$ ,
$\langle \textbf{Arc}, (r_2, d) \rangle$ , $\langle \textbf{Arc}, (d, r_2) \rangle$

Gli elementi d'arredamento, definiti dai simboli **Lamp**, **Table** e **Chair**, generano direttamente insiemi di **Quad**; nel primo caso si genera il solido per rotazione di un segmento di parabola e se ne estrae un tronco, nel secondo e terzo caso si modellano ed uniscono semplici parallelepipedi.

### 2.2.4 *Grammatica per la rete stradale*

Il posizionamento degli edifici è diretto dalla conformazione della rete stradale. In generale, ciascun edificio si affaccia su di una strada, ed in particolare verso di essa è orientata la facciata con il portone. Gli edifici sono organizzati in *isolati*, i quali sono delimitati da un poligono formato dai *segmenti di strada* che uniscono due incroci. Per rappresentare la rete stradale, la scelta più naturale è quella del grafo, che si suppone planare (ovvero privo di archi che si intersecano fuori dai nodi) per semplicità. Ogni isolato è suddiviso in *lotti*, ciascuno dei quali di dimensioni e proporzioni adeguate ad ospitare un solo edificio.

Ciascun nodo del grafo delle strade rappresenta un punto sul piano orizzontale. Ad un isolato corrisponde un *ciclo senza corde* del grafo, ed un segmento di strada equivale ad un arco.

**Definizione 2.2.9** (Ciclo in un grafo)**.** Sia $G = (V, E)$ un grafo e $c = \langle v_0, \cdots, v_n \rangle$ un cammino di $G$. Allora, $c$ è un *ciclo* se e solo se $v_0 \equiv v_n$.

**Definizione 2.2.10** (Corda di un grafo)**.** Sia $G = (V, E)$ un grafo e $c = \langle v_0, \cdots, v_n \rangle$ un cammino di $G$. Allora, $e \in E$ è una corda del cammino $c$ se e solo se $e = (v_a, v_b)$, con $v_a$ e $v_b$ nodi sul cammino $c$, e $v_a$ e $v_b$ non sono successivi nel cammino.

Le intrinseche proprietà geometriche del grafo delle strade, a differenza di un grafo normale, permettono di semplificare l'interpretazione di alcuni aspetti basandosi non sulla teoria dei grafi, ma sulla geometria. Ad esempio, i cicli senza corde di cui fa parte un determinato nodo possono essere trovati mediante il seguente algoritmo.

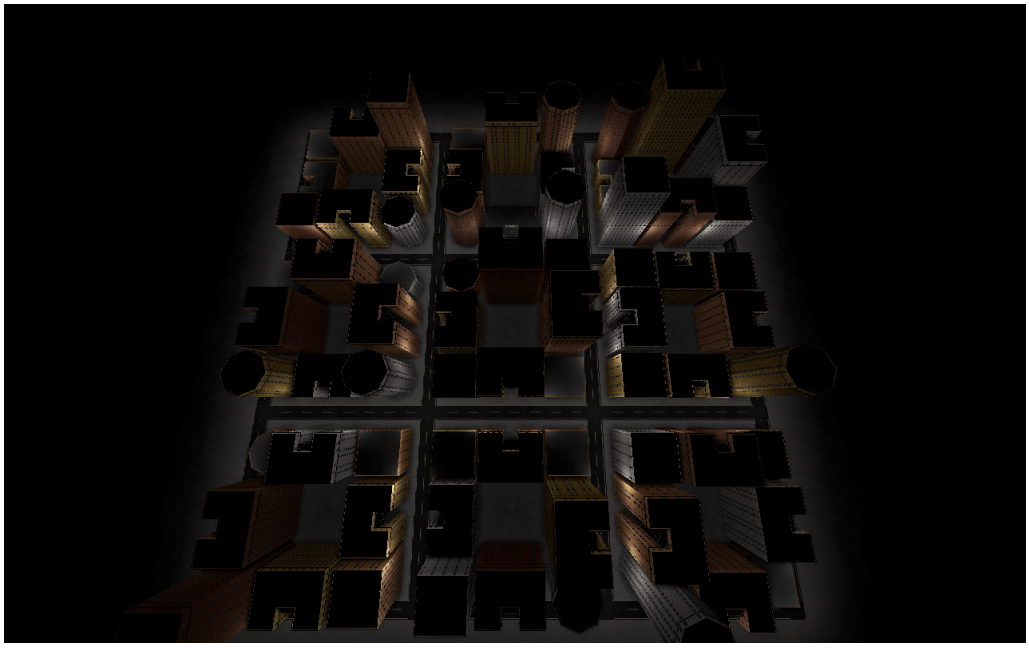


Figura 3: Vista dall'alto della città. Si osservi la struttura delle strade e dei lotti.

Sia c un cammino, inizialmente costituito da due soli vertici $\langle v_0, v_1 \rangle$ collegati da un arco a partire dai quali si vuole trovare un ciclo senza corde. Allora si ripeta il seguente passo.

Siano $v_{i-1}, v_i$ gli ultimi due vertici in c, e $V_i = \{v_j \in V : (v_i, v_j) \in E\}$. Allora, per ogni $v_j \in V_i$, si calcoli

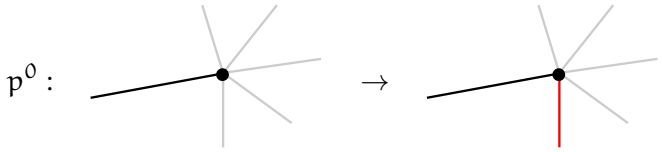
$$\theta_j = \arctan \frac{y_{v_i} - y_{v_{i-1}}}{x_{v_i} - x_{v_{i-1}}}$$

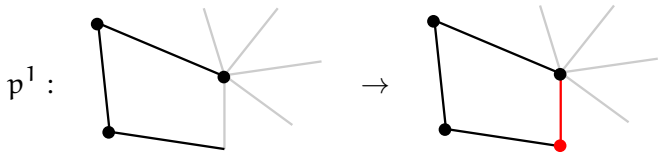
E si calcoli inoltre

$$\theta_i = \arctan \frac{y_{v_i} - y_{v_{i-1}}}{x_{v_i} - x_{v_{i-1}}}$$

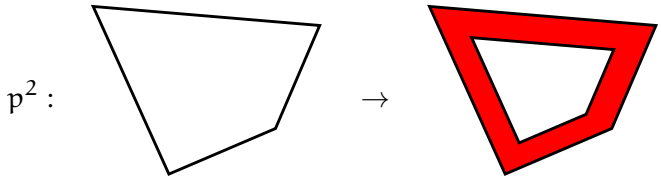
Sia, quindi, k il valore per cui è minimo $\theta_k - \theta_i$ modulo $2\pi$. Allora, si aggiunge $v_k$ al cammino c e si ripete, finché $v_k \equiv v_0$.

Informalmente, con questo algoritmo si ricerca, ad ogni passo e per ogni arco, l'arco successivo più vicino in senso orario:

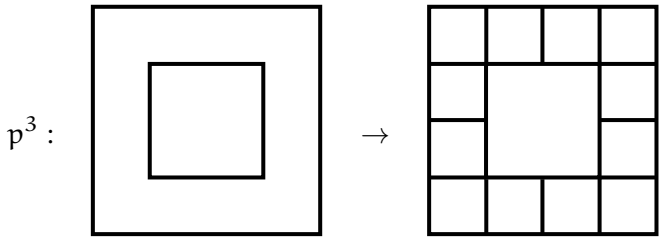





Una volta ottenuti tutti i cicli senza corde, ovvero tutti gli isolati, si procede a suddividerli in *lotti*. Per fare questo, si impiega anzitutto l'operatore INSET per ricavare un poligono della stessa forma dell'isolato, ma più piccolo ed interno all'isolato stesso, delineando così uno spazio lungo il quale disporre gli edifici.




Da una ulteriore suddivisione dello spazio sopra generato, si ricavano infine i singoli lotti che andranno ad ospitare un edificio ciascuno. Il calcolo dei punti di suddivisione dei lotti è eseguito mediante una versione modificata dell'operatore SPLIT, che introduce un aspetto legato in parte ad euristiche basate sulla geometria degli isolati, in parte ad una fonte di numeri pseudocasuali, al fine di variare le dimensioni dei lotti pur mantenendole coerenti e realistiche – ad esempio, è opportuno scartare suddivisioni che generino lotti troppo larghi, o troppo stretti, o troppo piccoli.




## 2.3 CENNI SULLA GENERAZIONE NUMERICA PSEUDOCASUALE

Alcuni dei processi di generazione procedurale delineati dalle precedenti grammatiche necessitano di processi casuali per guidare in maniera organica l'espressione di determinati tratti. Ad esempio, ogni edificio avrà bisogno di differenziarsi nella forma delle fondamenta, nella disposizione delle finestre, nel numero dei piani, e l'appartamento avrà bisogno di introdurre del disturbo nei criteri di suddivisione delle stanze, al fine di non generare una planimetria troppo regolare od uniforme, i cui *pattern* emergerebbero troppo evidentemente, sottraendo realismo alla scena.

Per far fronte a questa necessità, si è impiegato un algoritmo di generazione numerica pseudocasuale (*PRNG*) in grado di generare sequenze di numeri approssimativamente casuali. Questo algoritmo è il Mersenne Twister [12] ed è impiegato come generatore pseudocasuale standard in molti linguaggi di programmazione.

Un generatore di numeri pseudocasuali è un sistema deterministico: a fronte di uno stesso *input*, detto *seed*, verrà generata sempre la medesima sequenza di numeri. Mentre questo aspetto può rivelarsi uno svantaggio in ambiti quali la sicurezza informatica, nel contesto della generazione procedurale è di fondamentale importanza per fornire controllo sui risultati.

Il progetto impiega numerose istanze di *PRNG*, una per ciascun aspetto che esibisce comportamenti pseudocasuali. Ognuna di queste istanze riceve un *seed* ricavato da qualche proprietà del contesto in cui un dato aspetto si presenta. Ad esempio, il numero dei piani, la conformazione delle fondamenta ed il numero e la posizione delle finestre illuminate rispetto a quelle non illuminate ricevono *seed* da un'operazione di somme e moltiplicazioni (senza un particolare criterio) tra i valori numerici che descrivono il lotto su cui si edifica il palazzo (ad es. larghezza, profondità, posizione di un punto, ...). Oppure, ancora, il predicato $c_r$ usato per suddividere le stanze dell'appartamento ha un *seed* costante mediante il quale viene generata sempre la stessa planimetria.

# 3

# TECNICHE DI RENDERING

Con il termine *rendering* si indica la resa grafica di una scena a partire da una sua descrizione astratta. Un *rendering fotorealistico* è quanto risulta dall'impiego di tecniche di *rendering* basate sulla simulazione, in qualche senso, dei fenomeni fisici legati al senso della vista.

## 3.1 ILLUMINAZIONE

Il processo della percezione visiva si articola in tre fenomeni fondamentali: l'emissione di luce, l'interazione della luce con gli elementi dell'ambiente, e l'assorbimento della luce da parte di un sensore (ad esempio un occhio umano, o la pellicola di una macchina fotografica).

### 3.1.1 *Equazione di Rendering*

J. T. Kajiya pubblica, nel 1986, un articolo che descrive l'*equazione di rendering* [7], un'equazione integrale che generalizza alcuni algoritmi noti di *rendering*, sintetizzando un modello fisico dell'illuminazione. Essa si basa sulla *legge di conservazione dell'energia*, ed è[1]

**Definizione 3.1.1** (Equazione di *rendering*)**.**

$$L_o(\mathbf{p}, \mathbf{v}) = L_e(\mathbf{p}, \mathbf{v}) + \int_\Omega f(\mathbf{l}, \mathbf{v}) \otimes L_o(r(\mathbf{p}, \mathbf{l}), -\mathbf{l}) \cos\theta_l d\mathbf{l} \tag{3.1}$$

laddove

- $L_o(\mathbf{p}, \mathbf{v})$ misura la radiazione uscente dal punto $\mathbf{p}$ in direzione $\mathbf{v}$.
- $L_e(\mathbf{p}, \mathbf{v})$ misura l'*emittanza*, ovvero la quantità di energia emessa, dal punto $\mathbf{p}$ in direzione $\mathbf{v}$.
- $\Omega$, il dominio di integrazione, è un emisfero unitario orientato secondo il vettore normale alla superficie a cui appartiene $\mathbf{p}$.
- $f(\mathbf{l}, \mathbf{v})$ è detta *bidirectional reflectance distribution function*, o *BRDF*, e misura l'intensità dell'energia *diffusa* (o *dispersa*) dal punto $\mathbf{p}$ in direzione $\mathbf{v}$ per la componente della luce entrante su $\mathbf{p}$ dalla direzione $\mathbf{l}$.

[1] La forma dell'equazione riportata in [7] differisce lievemente da quella, qui riportata, di [17]. Si è scelta questa forma in luogo di quella dell'articolo originale per coerenza di notazione con le altre formule citate nel seguito, molte delle quali tratte da [17].

- $\otimes$ è un operatore di moltiplicazione componente-componente, usato poiché $f$ ed $L_o$ sono rappresentati come *colori* (definiti come 3.1.2).
- $r(\mathbf{p}, \mathbf{v})$ è la *ray casting function*, che ritorna il primo punto incontrato su una superficie proiettando un raggio da $\mathbf{p}$ in direzione $\mathbf{v}$.
- $\theta_i$ è l'angolo tra $\mathbf{l}$ e la normale alla superficie $\mathbf{n}$.

**Definizione 3.1.2** (Colore)**.** La tripla $\mathbf{c} = (\mathbf{c}_r, \mathbf{c}_g, \mathbf{c}_b)$ è un *colore*, oppure *vettore RGB*, se e solo se $\mathbf{c}_r, \mathbf{c}_g, \mathbf{c}_b \in [0, 1]$. Le tre componenti indicano, rispettivamente, la *radianza* corrispondente alle lunghezze d'onda dei colori rosso, verde e blu; sovrapponendole, si ottiene $\mathbf{c}$.

Informalmente, l'intensità della luce in un punto dello spazio è data dalla somma della luce *emessa* da un oggetto in quel punto, della luce *diretta* proveniente da altre fonti luminose, e della luce *indiretta*, ovvero riflessa e rifratta dalla geometria circostante.

Risolvere l'equazione di *rendering* è l'obiettivo fondamentale perseguito nell'ambito del *rendering* fotorealistico. Tuttavia, il termine $L_o$ interno all'integrale è *ricorsivo* a causa della *ray casting function*, e potrebbe rendere indecidibile la terminazione di un algoritmo di risoluzione diretta. È necessario, quindi, ricorrere a soluzioni approssimate. Queste si dividono in due categorie: gli algoritmi di *rendering offline* e quelli di *rendering real-time*.

I primi approssimano l'equazione di *rendering* in maniera più fedele, ma hanno costi computazionali elevati, e la loro esecuzione può impiegare da pochi minuti fino a diversi giorni anche per un singolo fotogramma; vengono principalmente impiegati nell'industria cinematografica e, più in generale, per produrre animazioni ed immagini statiche. Uno tra gli algoritmi più noti di questa categoria è il *ray tracing*.

Ai secondi, invece, è imposto un limite superiore sul tempo d'esecuzione: essendo principalmente impiegati in contesti interattivi e dinamici quali l'industria videoludica, un algoritmo *real-time* dovrà necessariamente terminare entro un ammontare di tempo relativamente breve, in accordo con il numero di *frames* per secondo posto come obiettivo. Questo è fondamentale: valori troppo inferiori a 25 *fps* rovinano l'esperienza dell'utente, giacché il passaggio da un *frame* al successivo diventa evidente; serve pertanto un algoritmo che termini al più in 40 *ms*. Un obiettivo qualitativamente migliore è quello dei 60 *fps*, ovvero un tempo di esecuzione massimo di $16.\overline{6}$ *ms*. L'*hardware* di consumo disponibile al tempo della stesura di questa tesi è generalmente in grado di eseguire una varietà di algoritmi di *rendering real-time* ampiamente entro questi limiti. È tuttavia necessario compromettere, in parte, il fotorealismo, e progettare accuratamente la distribuzione del carico computazionale al fine di ottimizzare l'equilibrio tra risorse impiegate e risultato estetico, e considerare che generalmente

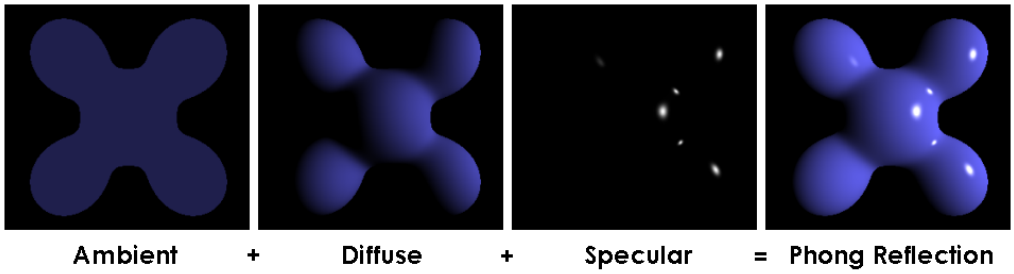



Figura 4: Modello di riflessione di Phong.
Brad Smith, Wikimedia Commons, CC BY-SA 3.0

i calcoli relativi al *rendering* dovranno condividere il tempo computazionale a loro disposizione con altro codice applicativo (ad esempio, quello relativo alla gestione dell'*input* dell'utente).

Per la natura dinamica del processo di generazione della geometria, per il fatto che questa, effettivamente, non esista prima dell'avvio dell'applicazione del progetto, e per le tecnologie scelte, sono stati impiegati algoritmi *real-time*.

### 3.1.2 *Phong shading e modello di riflessione Blinn-Phong*

Introdotto nel 1975 da Bùi Tuong Phong [3], il *Phong shading* è un metodo di interpolazione dei vettori normali alla superficie su di una *mesh* di poligoni. Un solido modellato come una *mesh* di poligoni presenta una discontinuità della normale alla superficie lungo i bordi di un poligono, tanto più evidente quanto più ampia è la superficie del poligono considerato; questo porta ad un calcolo errato e dall'aspetto innaturale dell'illuminanza risultante su di un punto. Secondo Phong, la normale alla superficie nel punto $\mathbf{p}_t$, $t \in [0, 1]$, posto lungo il segmento definito da due punti $\mathbf{p}_0$ e $\mathbf{p}_1$, distante $t$ da $\mathbf{p}_0$ e $t-1$ da $\mathbf{p}_1$, è calcolata come

$$\mathbf{n}_t = t\mathbf{n}_1 + (1-t)\mathbf{n}_0 \tag{3.2}$$

dove $\mathbf{n}_0$ ed $\mathbf{n}_1$ sono le normali calcolate, rispettivamente, nei punti $\mathbf{p}_0$ e $\mathbf{p}_1$.

A questo metodo di interpolazione si affianca il *modello di riflessione di Phong*, un modello empirico di illuminazione locale divenuto *standard de facto* nell'ambito del *rendering real-time*. Phong descrive lo *shading* di una superficie al punto P come la somma di tre termini: il termine di riflessione *ambientale*, un coefficiente costante e generalmente piccolo che modella la luce riflessa dall'intera scena; il termine di riflessione *diffusiva*, che modella la quantità di luce diffusa dalla superficie; il termine di riflessione *speculare*, che modella la quantità di luce riflessa dalla superficie. Il modello è formalizzato come segue[2]:

[2] Notazione di [3].

$$S_P = C_P\left((1 - p_a)\cos\theta + p_a\right) + W(\theta)\,[\cos s]^n \tag{3.3}$$

laddove

- $C_P$ è un coefficiente di riflessione della superficie al punto $P$ per una determinata lunghezza d'onda,
- $\theta$ è l'angolo incidente alla superficie,
- $p_a$ è il coefficiente di *riflessione diffusa ambientale*,
- $s$ è l'angolo tra la direzione della luce riflessa e la direzione da $P$ al sensore ottico,
- $W(\theta)$ e $n$ sono due termini empirici, il primo funzione dell'angolo incidente alla superficie, necessari a modellare le caratteristiche del *materiale* della superficie; non hanno un significato fisico intrinseco.

Nel 1977, James F. Blinn pubblica l'articolo "Models of light reflection for computer synthesized pictures" [4], espandendo sul lavoro di Phong. Qui viene espressa una versione migliorata dell'algoritmo di Phong, sulla base della *legge di Lambert*. L'equazione di *shading* di Blinn-Phong viene descritta, per ogni punto della superficie $\mathbf{p}$, come[3]:

$$\begin{aligned} L_o(\mathbf{p}, \mathbf{v}) &= c_{amb} \otimes L_A + \left(c_{diff}\ \overline{\cos}\ \theta_i + c_{spec}\ \overline{\cos}^m\ \theta_h\right) \otimes E_L \\ &\equiv c_{amb} \otimes L_A + \left(c_{diff} \max\{\mathbf{l}\cdot\mathbf{n}, 0\} + c_{spec} \max\{\mathbf{h}\cdot\mathbf{n}, 0\}^m\right) \otimes E_L \end{aligned} \tag{3.4}$$

laddove

- $\overline{\cos}\ \theta = \max\{\cos\theta, 0\}$
- $L_A$ è l'*irradianza* della *riflessione ambientale*,
- $c_{amb}$ è il colore della *riflessione ambientale*,
- $c_{diff}$ è il colore della *riflessione diffusiva* al punto $\mathbf{p}$,
- $c_{spec}$ è il colore della *riflessione speculare* al punto $\mathbf{p}$,
- $E_L$ è il colore della luce entrante,
- $m$ è una costante empirica,

[3] Notazione di [17].

- $\mathbf{v}, \mathbf{l}, \mathbf{n}$ rappresentano rispettivamente la direzione dal punto $\mathbf{p}$ al sensore, la direzione dalla fonte luminosa al punto e la normale alla superficie sul punto,
- $\mathbf{h} = \frac{\mathbf{l}+\mathbf{v}}{\|\mathbf{l}+\mathbf{v}\|}$ è la direzione che massimizza la riflessione speculare, rispetto alla direzione $\mathbf{v}$ verso il sensore, della luce entrante dalla direzione $\mathbf{l}$; è chiamato *halfway vector*, dal momento che indica la direzione "a metà strada" tra quella della luce e quella del sensore,
- $\theta_i$ è l'angolo compreso tra i vettori $\mathbf{l}$ ed $\mathbf{n}$,
- $\theta_i$ è l'angolo compreso tra i vettori $\mathbf{h}$ ed $\mathbf{n}$.

I valori per $m$, $c_{diff}$ e $c_{spec}$ sono proprietà empiriche intrinsecamente dipendenti dal *materiale* di cui è composta la superficie, come $C_P$, $W(\theta)$ ed $n$ nella formulazione originale di Phong.

Per la natura additiva della luce, ricaviamo l'approssimazione dell'equazione di *rendering* secondo il modello di Blinn-Phong, per il punto $\mathbf{p}$ nella direzione del sensore $\mathbf{v}$ e per $n$ fonti luminose come

$$L_o(\mathbf{p}, \mathbf{v}) = c_{amb} \otimes L_A + \left( \sum_{k=1}^{n} \left( c_{diff} \, \overline{\cos} \, \theta_{i_k} + c_{spec} \, \overline{\cos}^m \, \theta_{h_k} \right) \right) \otimes E_{L_k}. \quad (3.5)$$

### 3.1.3 *Screen-space ambient occlusion*

Il modello di *Blinn-Phong* è uno dei più diffusi nell'ambito del *rendering real-time*, sia per la semplicità d'implementazione, che garantisce *performance* ottime su gran parte dell'*hardware* di consumo contemporaneo, sia per l'efficacia dei risultati, che trasmettono con successo la dimensione di profondità della scena. Tuttavia, questo modello approssima il contributo di tutta l'illuminazione indiretta tramite una costante; a causa di ciò, gli oggetti non proiettano ombra e spesso distinguere le relazioni tra di essi nell'ambito di una scena diventa complesso.

Con il termine *ambient occlusion* si fa riferimento all'ombreggiatura proiettata da parte della luce *ambientale* [17]. Nel modello di Phong, la luce ambientale è supposta provenire uniformemente da ogni direzione. Considerando però una *mesh* presentante delle concavità, si intuisce che ciò non possa essere corretto, in quanto parte delle direzioni della luce entranti su un punto della superficie interno alla concavità saranno *occluse* dalla *mesh* stessa (si veda la Figura 5).

Se si definisce la luce ambientale come $L_i(\mathbf{l}) = L_A, \ \forall \ \mathbf{l}$, e la si applica all'equazione di *rendering*, si ottiene un'irradianza pari a

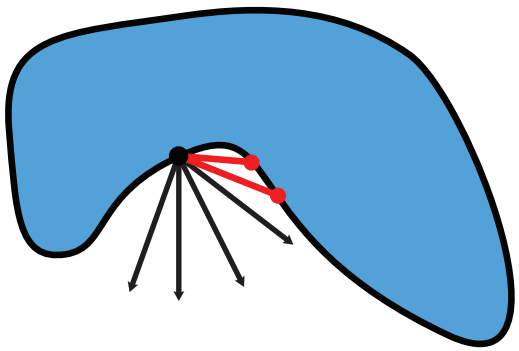


Figura 5: Occlusione di alcune direzioni della luce in una figura concava

$$E(\mathbf{p}, \mathbf{n}) = \int_{\Omega} L_A \cos\theta_i \, d\mathbf{l} = \pi L_A \tag{3.6}$$

Si definisce quindi, formalmente, il termine di *ambient occlusion* al punto $p$ come

$$k_A(\mathbf{p}) = \frac{1}{\pi} \int_{\Omega} v(\mathbf{p}, \mathbf{l}) \cos\theta_i \, d\mathbf{l} \tag{3.7}$$

dove $v(\mathbf{p}, \mathbf{l})$ è una *funzione di visibilità* il cui valore è $0$ se un raggio tracciato da $\mathbf{p}$ in direzione $\mathbf{l}$ è bloccato da un poligono, ed $1$ altrimenti. Il valore di $k_A(\mathbf{p})$ è compreso tra $0$, il caso degenere di una superficie completamente occlusa, ed $1$, che indica una superficie aperta, come quella di una *mesh* convessa.

Crytek sviluppa, intorno al 2007, un approccio in *screen-space* all'*ambient occlusion* [16] usato nel videogioco *Crysis*. Questo algoritmo stima il valore di $k_A$ per

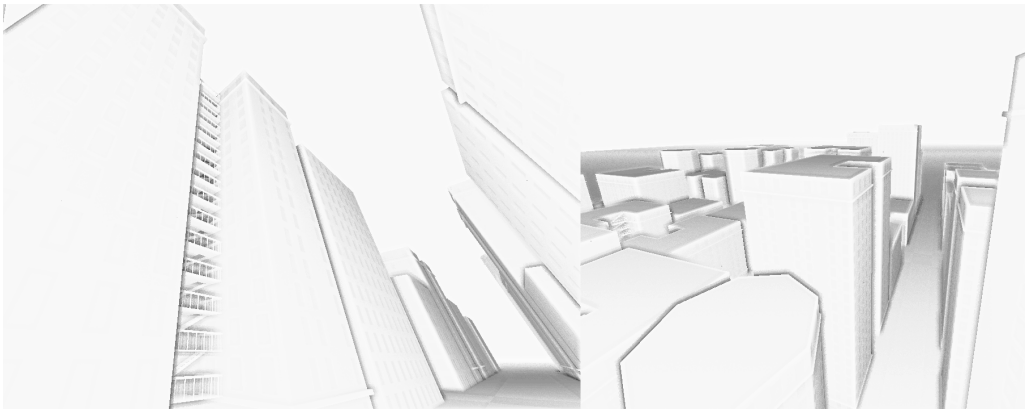


Figura 6: *Screen-space ambient occlusion buffer* di due fotogrammi dell'animazione

un punto $\mathbf{p}$ campionando un insieme di punti distribuiti all'interno di una sfera centrata su $\mathbf{p}$. La profondità di ciascuno dei punti dell'insieme è confrontata con quella di $\mathbf{p}$: punti dai valori di profondità inferiori passano il test, vengono pesati in maniera inversamente proporzionale alla loro distanza da $\mathbf{p}$ ed il loro contributo viene aggiunto a $k_A$, mentre valori superiori vengono scartati. Un numero inferiore di punti passanti il test si traduce quindi in un valore di $k_A$ inferiore, indice di maggiore occlusione.

L'algoritmo di *screen-space ambient occlusion* di Crytek risulta fisicamente scorretto, ma visivamente piacevole; inoltre, l'*overhead* in termini di *performance* non è eccessivo. Per queste ragioni è stato scelto per essere implementato nell'ambito del progetto.

### 3.1.4 *Fonti luminose dirette*

Il termine della direzione della luce nel modello di Blinn-Phong può essere modellato a partire dal modo in cui si definisce la sorgente stessa della luce. Nel caso generale, una sorgente luminosa è modellata come un punto nello spazio, diciamo $\mathbf{p}_L$, irradiante energia in maniera uniforme in tutte le direzioni. Pertanto, la direzione in cui $\mathbf{p}_L$ illumina il punto sulla superficie $\mathbf{p}_S$ è calcolata come

$$d = \|\mathbf{p}_L - \mathbf{p}_S\|$$
$$\mathbf{l} = \frac{\mathbf{p}_L - \mathbf{p}_S}{d}$$

Il termine al denominatore è la *distanza* tra la fonte luminosa ed il punto illuminato, ed è variabile nella *funzione di attenuazione*,

$$f_{dist}(d) = \frac{1}{a_c + a_l d + a_q d^2}$$

dove $a_c$ è il parametro di *attenuazione costante* (indipendente dalla direzione, $a_l$ è il parametro di *attenuazione lineare*, e $a_q$ è il parametro di *attenuazione quadratica*. Questi modellano in modo empirico la diminuzione d'intensità della luce con la distanza ($a_l$ ed $a_q$), e la sua intensità massima ($\frac{1}{a_c}$ quando $d = 0$).

Il valore della funzione di attenuazione per la luce $\mathbf{p}_L$ al punto $\mathbf{p}_S$ definisce il parametro di irradianza della luce entrante nel modello di Blinn-Phong:

$$E_L = I_L f_{dist}(d)$$

dove $I_L$ indica l'intensità della luce.

Le luci puntiformi sono un modello coerente con fonti luminose locali, ad esempio i lampioni o le lampade a soffitto usate nel progetto. Per modellare

una fonte luminosa quale la luce solare o quella lunare, si usa un caso particolare delle luci puntiformi: la *luce direzionale*. Si suppone che la posizione di queste sia un punto all'infinito, e che l'intensità sia arbitrariamente elevata: conseguentemente, la direzione della luce sarà indipendente dal punto illuminato, e sarà uniforme su tutta la scena, allo stesso modo dell'intensità. La luce sarà quindi modellata come la sua direzione ed intensità, senza interazioni con l'ambiente.

### 3.1.5 *Deferred shading e fonti luminose multiple*

L'illuminazione nel progetto è gestita combinando il modello di illuminazione descritto da Blinn-Phong con la *screen-space ambient occlusion* di Crytek; approssimando l'equazione di *rendering* con questi due strumenti, si ottiene un risultato ragionevolmente realistico con *performance* di elaborazione che raggiungono consistentemente l'obiettivo stabilito.

L'approssimazione finale, usata nel progetto, dell'equazione di *rendering*, è dunque

$$L_o(\mathbf{p}, \mathbf{v}) = k_A(\mathbf{p}) \sum_{k=1}^{n} \left( \left( c_{diff} \, \overline{\cos} \, \theta_{i_k} + c_{spec} \, \overline{\cos}^{m} \, \theta_{h_k} \right) \otimes E_{L_k} \right). \tag{3.8}$$

Si noti che il termine di *ambient occlusion* moltiplica tutte le componenti della luce; questo è formalmente scorretto. Tuttavia, impiegare in questo modo l'*ambient occlusion* costituisce un'approssimazione, seppur grossolana, per l'implementazione delle ombre. Queste potrebbero essere state modellate tramite algoritmi di *shadow mapping*, che sono però stati scartati poiché computazionalmente troppo onerosi: è necessaria infatti una passata di *rendering* della geometria per ciascuna fonte luminosa (in ciascuna passata $\mathbf{v}$ è la posizione della luce) per una complessità in $\mathcal{O}(gl)$, $g$ = numero di frammenti, $l$ = numero di luci. Il numero di luci è generalmente elevato durante tutta l'animazione (ve ne sono fino a 64 per ogni frame) e per questo non è possibile ottenere *frame rate* interattivi.

È stato inoltre escluso il termine costante $c_{amb} \otimes L_A$ di illuminazione ambientale, poiché il contributo da esso fornito all'approssimazione dell'illuminazione globale è stato giudicato trascurabile rispetto al contributo fornito dal solo termine di *ambient occlusion*, e non vi era giustificazione per i calcoli aggiuntivi.

Per realizzare il sistema di *rendering* con gli algoritmi delineati, si è implementata una *pipeline* a più passate; si è ovvero separato l'algoritmo di determinazione delle superfici visibili dall'algoritmo di illuminazione. La *pipeline* così realizzata è detta *deferred*, per il fatto che la computazione dell'illuminazione è *differita* rispetto ai calcoli geometrici.

Il *deferred shading* nell'ambito di *WebGL* consiste nell'impiego di più *programmi shader*. A differenza del *forward rendering*, descritto in Appendice A, il primo programma si occupa unicamente del calcolo delle superfici visibili. Il suo output, anzichè un'immagine, è un *buffer* bidimensionale il cui spazio corrisponde allo spazio NDC dello schermo, ma i cui contenuti non sono (necessariamente) colori; questo *buffer* è denominato *G-buffer* poiché codifica informazioni geometriche. Il formato dei dati del *G-buffer* va progettato in funzione delle informazioni che saranno necessarie alle passate successive del *rendering*; nel caso più semplice si codificheranno, in ogni punto NDC, posizione del vertice in *view space* e normale alla superficie.

Il *G-buffer* del progetto è molto compresso: tutte le informazioni relative ad un punto sullo schermo sono codificate in quattro numeri *floating point* a 32 *bit*. In due di questi è contenuta la codifica del vettore normale alla superficie: essendo noto che la lunghezza di questo vettore è unitaria, se ne registrano solo due delle tre componenti, e se ne ricostruisce la terza [20]. In un altro dei numeri *floating point* è registrato il colore della riflessione diffusiva, codificato registrando una componente nella parte frazionaria (7 *byte* della mantissa) e le altre due nella parte intera (8 *byte* ciascuno della mantissa). Infine, è registrata la *profondità* in *NDC* del frammento: questa, insieme alla posizione del frammento in *NDC*, è necessaria a ricostruire la posizione in *view space* del vertice originale invertendo la proiezione prospettica (approfondimento in Appendice A).

I dati così codificati sono sufficienti a ricostruire tutte le informazioni necessarie a calcolare l'equazione di *rendering* ai passi successivi della *pipeline*.

# 4

# TEXTURING PROCEDURALE

Le tecniche descritte finora permettono di rappresentare accuratamente la geometria fornita alla *pipeline* di *rendering* nella forma di *mesh* di poligoni accompagnate solo dalla descrizione di alcune proprietà della loro superficie, ad esempio il colore della riflessione diffusiva, da considerarsi valide su tutto il poligono. Per ottenere sufficiente dettaglio con questo approccio, è necessario impiegare *mesh* con un elevatissimo numero di poligoni anche per oggetti apparentemente molto semplici. I costi computazionali che ne conseguono sono molto elevati, sia in termini di tempo, sia di spazio.

Originariamente introdotto da Edwin Catmull [2], il processo di *texture mapping* associa una proprietà, od un insieme di proprietà, ad ogni punto su di una superficie, in funzione della posizione di questo rispetto alla superficie stessa, ed a partire da un sistema di riferimento predeterminato. Informalmente, si può pensare il *texture mapping* come l'applicazione di un'immagine su di una superficie. Il processo consiste nell'associazione di una coordinata ad ogni vertice, nella sua interpolazione attraverso tutta la superficie, nel recupero dell'informazione relativa alla coordinata interpolata a partire da una fonte dati arbitraria, e nell'impiego di questa informazione nel calcolo dello *shading*.

Nel caso tradizionale del *diffuse mapping*, si associa ad ogni punto $\mathbf{p}$ della superficie di un poligono una coordinata, detta $(u, v) \in [0, 1]^2$, a partire dalla quale è possibile calcolare la posizione di un *pixel* (o, più propriamente, *texel*) su di una immagine precaricata; il colore di questo *pixel* verrà assegnato alla

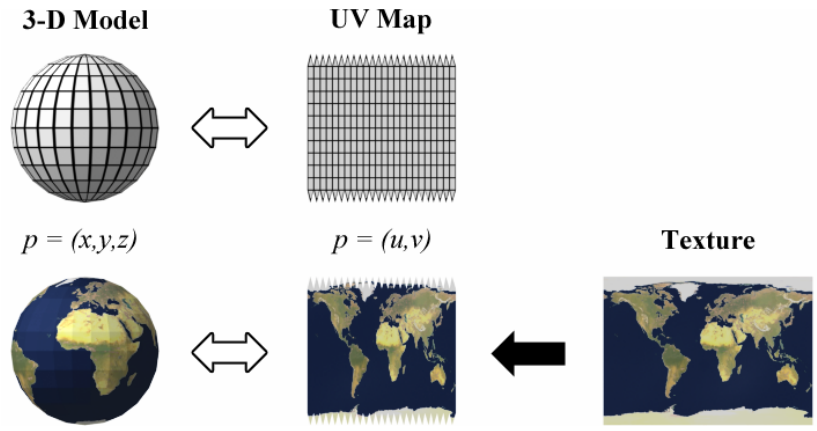



Figura 7: Un esempio di *UV mapping*. Tschmits, Wikimedia Commons, CC BY-SA 3.0.

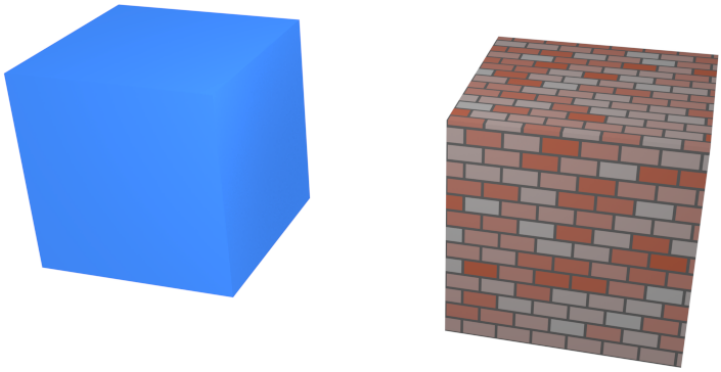


Figura 8: Un esempio di *diffuse mapping*. Sulla sinistra, un cubo senza *texture*; sulla destra, un cubo a cui è stata applicata una *texture* di mattoni.

componente $c_{diff}$ di riflessione diffusiva ed a partire da questo si potrà calcolare il colore risultante $L_o(\mathbf{p}, \mathbf{v})$.

Ad esempio, sia data una *texture* nella forma di un'immagine *bitmap* della dimensione di $1024 \times 512$ *pixel*, e sia $\mathbf{p}$ un generico punto su una superficie, avente parametri $(u, v)$. Allora, il colore del punto $\mathbf{p}$ corrisponderà al colore del *pixel* dell'immagine situato al punto $(1024 \cdot u, 512 \cdot v)$.

Il colore non è l'unico tipo di informazioni che è possibile codificare in una *texture*. Nel caso del *bump mapping* [5], ad esempio, è possibile interpretare il contenuto di un *texel* come una perturbazione della normale alla superficie; applicando la normale perturbata $\mathbf{n}' = \mathbf{n} + \mathbf{n}_{tex}$ nell'equazione di *rendering*, si ha la possibilità di simulare dei rilievi su di una superficie senza modificarne la geometria.

Nella maggior parte dei casi, le *texture* vengono create in anticipo nella forma di immagini (ad esempio, un disegno od una fotografia) e fornite al sistema di *rendering*. Nel presente progetto, invece, tutte le *texture* impiegate sono state realizzate nella forma di funzioni di coordinate $(u, v)$: al *lookup* di una *texture map* si sostituisce la valutazione della *texture procedurale* in un punto. A fronte di un maggior costo computazionale, le *texture* procedurali presentano i vantaggi di una risoluzione arbitraria, una memorizzazione estremamente compatta e la possibilità di ottenere, da una singola base, innumerevoli varianti per mezzo della parametrizzazione. Nel seguito verrà descritta ciascuna delle *texture* procedurali presenti nel progetto.

## 4.1 ANTIALIASING

Un fenomeno intrinseco ad ogni tipo di rappresentazione in computer grafica è l'*aliasing*. A tutti gli effetti, un'immagine digitale è il risultato di un *campionamento*, ovvero la conversione di un segnale continuo in un segnale discreto. La

scansione di un documento, od uno scatto fotografico digitale, esemplificano bene il processo: dalla luce entrante sul sensore, continua nello spazio, vengono "estratti" (campionati) punti in numero finito disposti come un reticolo quadrato, corrispondenti ai *pixel* dell'immagine digitalizzata. Similmente, con l'atto di calcolare il colore di ogni punto del contesto di *rendering* si esegue un campionamento del segnale continuo costituito dalla scena. La fruizione di un segnale discreto richiede che esso sia *ricostruito*, ovvero nuovamente trasformato in un segnale continuo (nel caso delle immagini, la ricostruzione avviene anche a livello di percezione visiva umana).

Il campionamento di un segnale continuo implica quasi sempre una perdita d'informazione. Quando questa perdita è eccessiva, la ricostruzione può dare origine ad un segnale fondamentalmente diverso da quello originale. Una ricostruzione corretta del segnale è possibile, secondo il *teorema di campionamento di Nyquist-Shannon*, solo impiegando una frequenza di campionamento superiore al doppio della massima frequenza del segnale originale. Questo si può verificare solo in presenza di un segnale *limitato in banda*, ovvero il cui spettro non presenta frequenze sopra un certo limite, il che non avviene quasi mai nell'ambito del *rendering*: le discontinuità cromatiche tra il bordo di un poligono nero ed il suo sfondo bianco possono ad esempio presentare frequenze arbitrariamente elevate, teoricamente infinite, come avviene nel caso di un'onda quadra; in questi casi non esiste una frequenza di campionamento fisicamente impiegabile e sufficientemente adeguata a ricostruire il segnale.

L'impossibilità di ricostruire fedelmente segnali dalle frequenze troppo elevate dà luogo all'*aliasing*, che nel caso del *rendering* si presenta nella forma di artefatti visivi (si veda la figura 11). Le tecniche per il trattamento di questi artefatti si basano sul *filtraggio* del segnale a qualche passo del processo di campionamento e ricostruzione e ricadono sotto il nome di *antialiasing*. L'operazione matematica dell'applicazione di un filtro prende il nome di

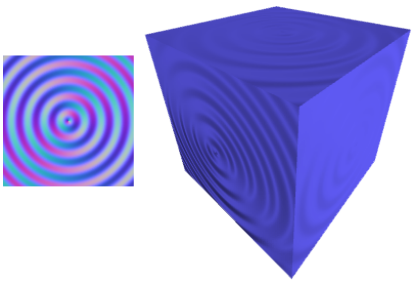


Figura 9: Un esempio di *normal mapping*: le normali alla superficie del cubo sono perturbate secondo la *texture* sulla sinistra, i cui colori sono composti codificando ciascuna componente direzionale della normale perturbata rispettivamente in ciascuno dei tre canali *RGB*.

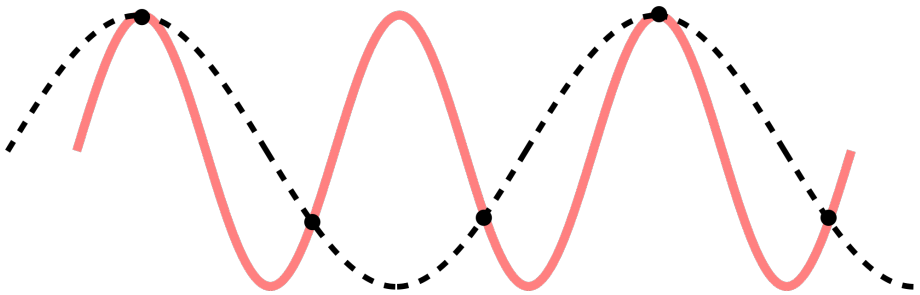


Figura 10: *Aliasing* di un segnale. Pluke, Wikimedia Commons, CC0 1.0.
I punti campionati sulla sinusoide originale (in rosso) possono essere erroneamente ricostruiti come una sinusoide differente (tratteggiata) quando la frequenza di campionamento è inferiore al doppio della massima frequenza del segnale originale.

**Definizione 4.1.1** (Convoluzione)**.** Siano $f, g$ due funzioni. Allora, l'operazione

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) \cdot g(t-\tau)\,d\tau = \int_{-\infty}^{\infty} f(t-\tau) \cdot g(\tau)\,d\tau \tag{4.1}$$

è detta *convoluzione*.

Nell'ambito della computer grafica, si deve operare necessariamente con quantità e funzioni discrete; nell'ambito del *texturing*, in particolare, è noto che queste sono quasi sempre bidimensionali. Introduciamo pertanto la seguente semplificazione:

**Definizione 4.1.2** (Convoluzione lineare discreta)**.** Siano $f, g$ due funzioni. Allora, l'operazione

$$(f * g)(u, v) = \sum_{x=-\infty}^{\infty} \sum_{y=-\infty}^{\infty} f(u-x, v-y) \cdot g(x, y) \tag{4.2}$$

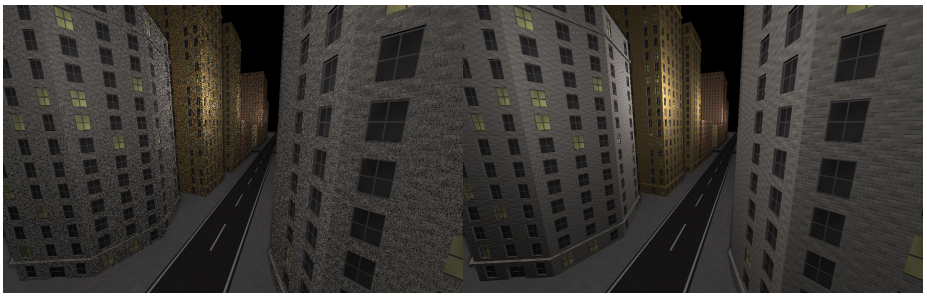


Figura 11: Sulla sinistra: un fotogramma che presenta *aliasing* sulla *texture* dei mattoni. Il filtraggio non corretto del campionamento delle normali alla superficie crea il disturbo. Sulla destra: lo stesso fotogramma con un filtraggio adeguato.

è detta *convoluzione lineare discreta*. Nel seguito, se non diversamente specificato, si indicherà questa col termine *convoluzione*; si indicheranno inoltre con $f$ il segnale sorgente e con $g$ il filtro.

Applicativamente, in generale, un filtro di convoluzione discreta ha solo pochi valori non nulli intorno all'origine, ed è zero altrove. Si può quindi semplificare ulteriormente, esprimendo la convoluzione come

$$(f * g)(u, v) = \sum_{(x,y)\in\mathbb{R}^2_g} f(u-x, v-y) \cdot g(x, y) \tag{4.3}$$

e rappresentando il filtro stesso nella forma di una *matrice di convoluzione* o *kernel*, tale che all'elemento centrale corrispondono i valori di $(0, 0)$ per $(x, y)$. Ad esempio, si consideri il *filtro di sfocatura Gaussiana*, definito come

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{4.4}$$

L'applicazione di questo filtro ad una immagine $f$ genera una nuova immagine $f * g$ in cui il colore di un generico punto $(u, v)$ è dato dalla somma dei nove punti intorno ad $(u, v)$, compreso, pesati secondo la matrice di convoluzione ($\frac{4}{16}$ per $f(u, v)$, $\frac{2}{16}$ per $f(u-1, v)$, $\frac{1}{16}$ per $f(u-1, v-1)$, ...). Il risultato dell'operazione si può osservare in figura 12.

Intuitivamente, in una immagine filtrata, il colore in un dato punto è il risultato di una media pesata dei colori dei punti che lo circondano nell'immagine originale, ed il filtro definisce i pesi.

## 4.2 SINTESI DI TEXTURE BASATA SUL RUMORE

Per introdurre un elemento di organicità nella definizione di una *texture* procedurale, è necessario impiegare una qualche sorgente di dati irregolari. Si potrebbe pensare all'impiego di un generatore di numeri pseudocasuali; tuttavia,

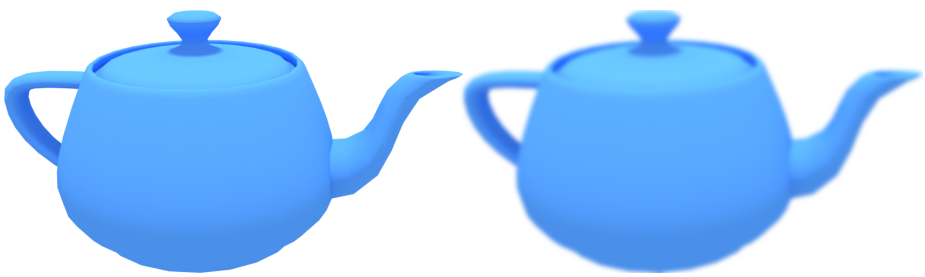


Figura 12: Filtro di sfocatura Gaussiana

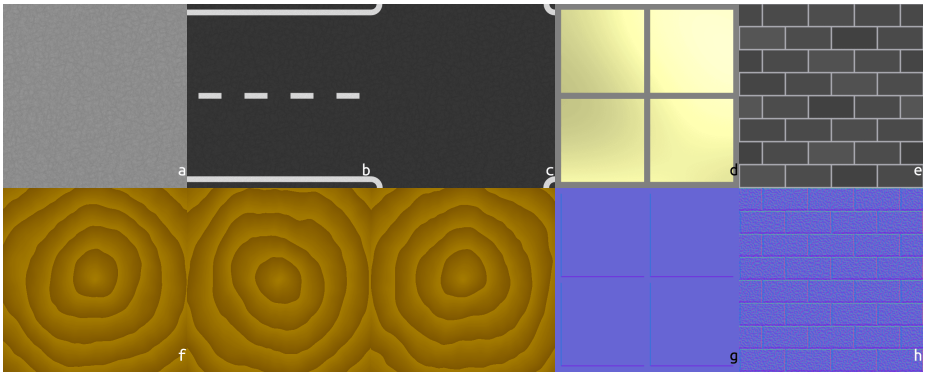



Figura 13: *Texture* procedurali impiegate nel progetto. (a) Materiale impiegato per i marciapiedi e gli interni. (b) Segmento di strada. (c) Incrocio. (d) Finestra, con relativa *normal map* (g). (e) Mattoni, con relativa *normal map* (h). (f) Tre istanze della *texture* solida del legno, con $y$ costante per ciascuna.

questo presenterebbe problematiche che sfuggono ad una analisi superficiale. In generale, infatti, quando qualcuno suppone di aver percepito un evento casuale, intende in realtà di aver riconosciuto l'assenza di uno schema nell'evento [10]. Ad esempio, lanciare cento volte una moneta ed ottenere testa per cento volte è probabile esattamente quanto ottenere una qualsiasi altra sequenza di teste o croci, e tuttavia difficilmente qualcuno potrebbe affermare, di fronte ad un evento simile, di non aver provato dubbi sul fatto che la moneta non fosse in qualche modo truccata. Questo suggerisce che un approccio puramente casuale alla generazione di irregolarità visuali potrebbe essere controproducente.

Lo stesso problema può essere riformulato in termini di elaborazione dei segnali. Al processo di lanciare cento monete, ovvero alla distribuzione uniforme di probabilità, corrisponde il segnale del *rumore bianco*. Questo ha una risposta in frequenza uniforme su tutto lo spettro, il che indica la possibilità di frequenze arbitrariamente alte. Queste portano all'*aliasing*, ovvero rendono indistinguibile il segnale ricostruito da un *altro* segnale, che sarebbe lo schema percepito dall'osservatore del fenomeno – in relazione all'evento precedente dei cento lanci di moneta che producono solo testa, il segnale percepito è quello costante prodotto dal campionare sempre lo stesso numero.

Naturalmente l'esempio dei cento lanci risultanti in testa è eccessivo ed estremamente raro, come d'altronde è la probabilità che si presenti una qualsiasi altra sequenza *specificata* di cento lanci indipendenti con probabilità $\frac{1}{2}$:

$$p_{100t} = \prod_{i=1}^{100} p_i = \prod_{i=1}^{100} \frac{1}{2} = 2^{-100}$$

Tuttavia, eventi particolari locali (ad esempio gruppi di cinque lanci testa consecutivi) sono decisamente meno improbabili, ma al contempo è decisamente più probabile che vengano percepiti ed intepretati come eventi non casuali.

Un evento percepito come casuale il più delle volte ha caratteristiche ben definite. Ad esempio, un insieme di punti distribuiti intorno ad una griglia, in modo tale che intorno a ciascun punto ci sia una quantità simile di spazio vuoto, sembrerà più casuale rispetto ad un insieme di punti generato in maniera realmente casuale ma in cui i punti sono stati generati molto vicini tra loro, producendo spazi bianchi e grappoli di *pixel* neri (Figura 14). Per tale ragione, al fine di produrre risultati percettibilmente casuali è opportuno impiegare la casualità pura solo come base di un sistema più strutturato.

La primitiva Noise ideata da Ken Perlin [14] è una funzione dall'implementazione semplice, dal ridotto costo computazionale e dai risultati soddisfacenti. Essa consiste nel generare casualmente vettori di lunghezza unitaria su di una griglia regolare in $\mathbb{R}^n$, e quindi nell'interpolarli con i punti adiacenti nella griglia mediante il polinomio[1] $3t^2 - 2t^3$ per ogni dimensione. In questo modo è possibile ottenere una funzione dal comportamento localmente prevedibile, ma dall'aspetto globalmente casuale.

Usando la funzione Noise come primitiva, è possibile sintetizzare lo spettro di nuovi segnali di rumore mediante un processo simile a quanto avviene con un'antitrasformata di Fourier: il rumore sintetizzato deriva, cioè, dalla somma di varie istanze del segnale primitivo a frequenze ed ampiezze variabili. In particolare, si parla di *rumore frattale* quando si sintetizza un segnale moltiplicando frequenza ed ampiezza di una quantità costante ad ogni passo. Ad esempio:

$$\text{FractalNoise}(t) = \sum_{\omega=1}^{k} \frac{\text{Noise}(\omega t)}{\omega} \tag{4.5}$$

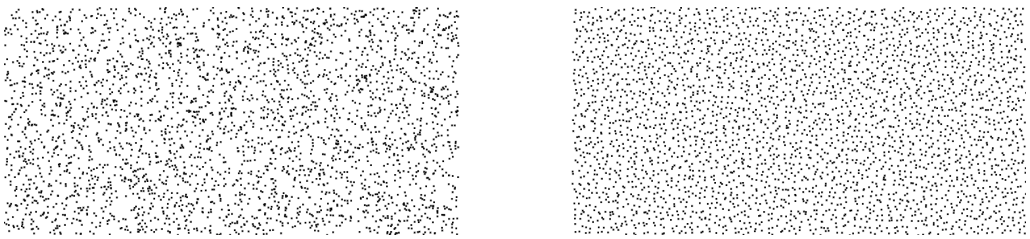


Figura 14: Confronto tra distribuzioni casuali di punti. Sulla sinistra, la posizione di ogni punto è generata in maniera totalmente casuale. Sulla destra, la posizione di ogni punto viene inizialmente generata su una griglia quadrata, per poi essere discostata di pochi *pixel* orizzontalmente e verticalmente.

[1] In una versione migliorata dell'algoritmo [15], questo è stato sostituito dal polinomio $6t^5 - 15t^4 + 10t^3$, avente migliori proprietà di continuità e derivabilità.

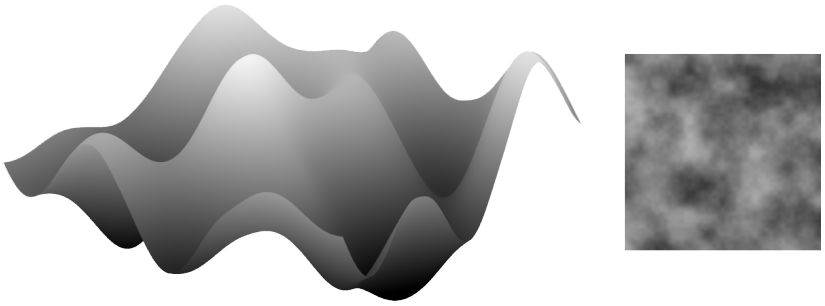


Figura 15: Rumore di Perlin. Sulla sinistra, un *rendering* in cui all'altezza della superficie corrisponde il valore della valutazione della funzione NOISE in quel punto. Sulla destra, la somma di varie istanze di NOISE a frequenze crescenti ed ampiezze decrescenti.

Nel progetto sono stati impiegati segnali sintetizzati a partire dal rumore di Perlin in tutti i casi in cui si è resa necessaria una perturbazione sulle *texture*.

## 4.3 TEXTURE IMPLEMENTATE NEL PROGETTO

### 4.3.1 *Texture per i mattoni e per le finestre*

La *texture* per i mattoni è un esempio classico nella generazione procedurale. Essa consiste, essenzialmente, nell'alternanza tra due colori in funzione della posizione: la malta, generalmente rappresentata con un colore grigio, ed il mattone, generalmente rosso. Diciamo che la funzione finale $\mathbf{c}_{\text{brick}}(u, v)$ associerà ad ogni coordinata $(u, v)$ il colore grigio oppure il colore rosso; per farlo, impiegherà un'altra funzione, diciamo $f_{\text{brick}}(u, v) : [0, 1]^2 \to [0, 1]$, la cui immagine è 0 quando ad $(u, v)$ corrisponde il colore della malta ed 1 quando ad $(u, v)$ corrisponde il colore del mattone.

$$\mathbf{c}_{\text{brick}}(u, v) = \begin{cases} \mathbf{c}_{\text{gray}} & f_{\text{brick}}(u, v) = 0 \\ \mathbf{c}_{\text{red}} & f_{\text{brick}}(u, v) = 1 \end{cases} \tag{4.6}$$

Ai fini dell'*antialiasing*, discusso in seguito, $f_{\text{brick}}$ può in realtà avere un codominio continuo, e quindi si ridefinisce $\mathbf{c}_{\text{brick}}$ come interpolazione lineare tra i due colori:

$$\mathbf{c}_{\text{brick}}(u, v) = \mathbf{c}_{\text{gray}} \cdot (1 - f_{\text{brick}}(u, v)) + \mathbf{c}_{\text{red}} \cdot f_{\text{brick}}(u, v) \tag{4.7}$$

Di seguito è delineato l'algoritmo implementato per ricavare $f_{\text{brick}}$.

1. Si calcola anzitutto $(u', v') = (um_x, vm_y)$, laddove $m_x, m_y$ sono il numero di mattoni in direzione, rispettivamente, orizzontale e verticale. Dato

che $u$ e $v$ hanno valori in $[0, 1]$, la parte intera di $u'$ e $v'$ andrà ad identificare in quale riga e colonna del *pattern* di mattoni si trovano le coordinate, e la loro parte frazionaria indicherà invece se ci si trova in mezzo alla malta (es. $0 \leqslant u \leqslant m_w$, con $m_w$ ampiezza del solco di malta nella direzione orizzontale) oppure in mezzo al mattone ($m_w < u \leqslant 1$).

2. Con le coordinate $(u', v')$ così calcolate i mattoni saranno allineati sia orizzontalmente, sia verticalmente; per effettuare uno scostamento orizzontale del pattern a righe alterne, come avviene nei muri di mattoni nella realtà, si calcolano nuove coordinate[2]

$$(u'', v'') = \begin{cases} (u', v') & \lfloor v' \rfloor \equiv_2 0 \\ (\{u' + 0.5\}, v') & \lfloor v' \rfloor \equiv_2 1 \end{cases}$$

3. Calcolando ancora altre coordinate

$$(u''', v''') = \left( \left\{ u'' + 1 - \frac{m_w}{2} \right\}, \left\{ v'' + 1 - \frac{m_h}{2} \right\} \right)$$

si ottiene l'effetto di centrare un solco di malta sull'origine. Confrontando queste ultime coordinate con $m_w$ e $m_h$ (ampiezza del solco di malta in verticale) si ottiene infine il pattern cercato, nella forma di un predicato che indica se le coordinate indicano un mattone od un solco di malta.

4. Si possono adesso impiegare le parti intere di $u'''$ e $v'''$ per scurire o schiarire leggermente il colore dei mattoni, ed introdurre varietà. Nel progetto, questi due valori sono stati impiegati sfruttando intenzionalmente il concetto di *aliasing*: il coseno della loro somma moltiplicata per un numero molto grande è una funzione periodica ad elevata frequenza ma che viene campionata sparsissimamente (una volta ogni mattone) al punto che diventa molto difficile scorgere uno schema che si ripete, e l'effetto che ne risulta è effettivamente quello di una distribuzione casuale.

In Figura 16 sono raffigurati i quattro passaggi dell'implementazione della *texture* descritti sopra.

Nel caso della *texture* procedurale per i mattoni, il problema dell'*antialiasing* è particolarmente grave, poiché siamo in presenza di una funzione equivalente all'onda quadra, e quindi che produce frequenze teoricamente infinite nello spettro. Tuttavia, la funzione della *texture* ha il vantaggio di essere facilmente integrabile in maniera analitica. Scegliendo un filtro molto semplice, quale la funzione in $\mathbb{R}^2$

[2] Con l'operatore $\{\cdot\}$ si indica la parte frazionaria di un numero.

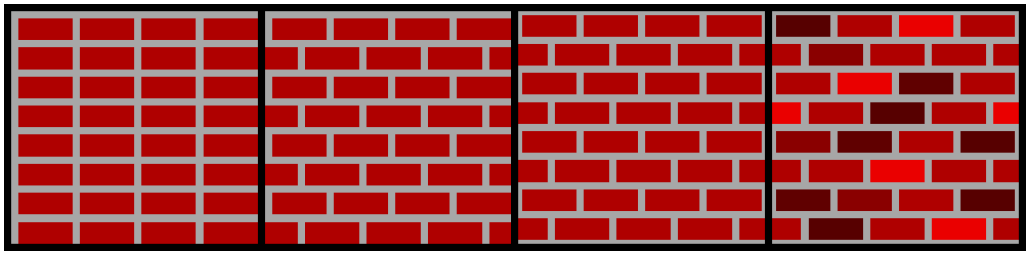


Figura 16: Costruzione della *texture* dei mattoni. Da sinistra: valutazione delle coordinate $(u', v')$, valutazione di $(u'', v'')$, valutazione di $(u''', v''')$ ed infine valutazione di $(u''', v''')$ con i colori alterati.

$$g_w(x, y) = \begin{cases} 1 & x, y \in [-w, w] \\ 0 & \text{altrove} \end{cases} \tag{4.8}$$

diviene possibile campionare direttamente la convoluzione della funzione col filtro, anziché la funzione stessa, ed ottenere risultati più accurati che col caso discreto:

$$\begin{aligned} (f_{\text{brick}} * g_w)(u, v) &= \int_{-\infty}^{\infty} f_{\text{brick}}(u - x, v - y) \cdot g_w(x, y) dx dy \\ &\equiv \int_{[-w,w]^2} f_{\text{brick}}(u - x, v - y) \cdot g_w(x, y) dx dy \end{aligned} \tag{4.9}$$

La forma analitica dell'integrale della funzione della *texture*, rappresentata per semplicità in una sola dimensione (Figura 17), è definita a partire dalle coordinate derivate da quelle originali, ed è

$$I_{\text{brick}}(u) = \lfloor u \rfloor (1 - m_w) + \max\left\{0, \{u\} - m_w\right\} \tag{4.10}$$

Intuitivamente, alla funzione costante a tratti $\lfloor u \rfloor (1 - m_w)$ si somma la componente a dente di sega $\{u\} - m_w$. Questa viene troncata a zero quando $\{u\} <$

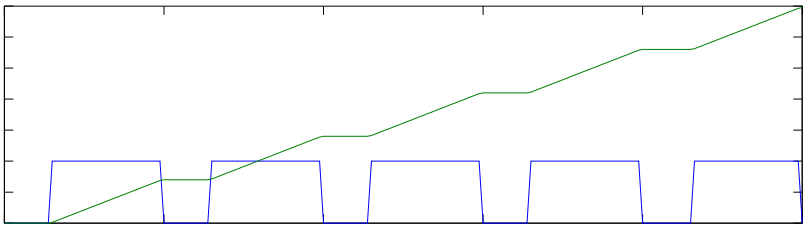


Figura 17: Grafico di un'onda quadra (in blu) ed il suo integrale (in verde).

$m_w$, ovvero quando $u$ è in mezzo al solco di malta e $f_{brick}(u) = 0$, e resta lineare altrimenti, ovvero quando $u$ è in mezzo al mattone e $f_{brick}(u) = 1$.

Ottenuta la forma analitica dell'integrale, se ne può valutare direttamente la convoluzione col filtro $g_w$ come l'integrale definito sull'area del filtro centrata su $u'''$, e ridefinire quindi la funzione

$$f_{brick}(u) = \left[ I_{brick}(u''') \right]_{-w}^{w} \tag{4.11}$$

Passando in due dimensioni, ed avendo cura di distinguere l'integrale in direzione orizzontale e quello in direzione verticale:

$$f_{brick}(u, v) = \left[ I^u_{brick}(u''') \right]_{-w}^{w} \cdot \left[ I^v_{brick}(v''') \right]_{-w}^{w} \tag{4.12}$$

Per simulare la presenza di geometria più ricca, è stata impiegata una *normal map*. Si è anzitutto realizzata una *height map* coi valori di $f_{brick}$: in questa, come nella *texture* piana, a valori bassi (prossimi a 0) corrisponde la malta, ed a valori alti (prossimi ad 1) corrisponde il mattone, che si suppone protrudere leggermente dalla superficie. Si è inoltre simulato un leggero spiazzamento sulla superficie dei mattoni sommando alla *height map* una funzione di rumore, per renderli più irregolari:

$$\text{NOISE}_{brick}(u, v) = \frac{\text{NOISE}(256u, 256v)}{8} \tag{4.13}$$

La *normal map* è quindi calcolata a partire da questa. Si differenzia, per ciascun punto, la *height map* in direzione orizzontale e verticale mediante un rapporto incrementale, ottenendo l'inclinazione della superficie in quel punto e verso le direzioni orizzontale e verticale, rispettivamente. Si procede quindi ricavando il piano inclinato secondo le due direzioni calcolate, e quindi il vettore normale ad esso. Il risultato è riportato in Figura 13h.

La *texture* per le finestre è ottenuta in modo del tutto simile, rimuovendo lo scostamento a file alterne e modificando i parametri che definiscono le dimensioni del mattone e del solco di malta, che ora diventano rispettivamente pannello di vetro e telaio della finestra. È stata simulata una variazione dell'illuminazione proveniente da dietro la finestra mediante una funzione di rumore semplice ed a bassa frequenza:

$$\text{NOISE}_{window}(u, v) = \frac{\text{NOISE}(\frac{u}{4}, \frac{u}{4})}{4} \tag{4.14}$$

### 4.3.2 *Texture per la segnaletica orizzontale*

Anche la *texture* per la segnaletica orizzontale segue la logica di alternare due colori, ovvero il grigio scuro dell'asfalto col bianco della segnaletica; tuttavia, essendo più complesso ricavarne un integrale analitico rispetto alle due precedenti, si affronta l'*antialiasing* in maniera differente, ovvero tentando di prevenire l'*aliasing* in primo luogo. Per fare ciò, è necessario calcolare un segnale limitato in banda. Si usa quindi, in luogo della funzione gradino, un polinomio interpolante cubico di Hermite troncato per valori esterni a $[0, 1]$:

$$\textsc{Smoothstep}(t, a, b) = \begin{cases} 0 & \tau \leqslant 0 \\ 1 & \tau \geqslant 1 \\ 3\tau^2 - 2\tau^3 & \tau \in (0, 1) \end{cases}, \quad \tau = \frac{t - a}{b - a} \tag{4.15}$$

Questa funzione ha l'importante caratteristica di simulare gli effetti del filtraggio, ovvero sfumare il passaggio da 0 ad 1 in una funzione soglia, evitando così le frequenze infinite dello spettro di questa.

La funzione $f_{road}(u, v)$ avrà valore 1, associato al bianco della segnaletica orizzontale, lungo due fasce orizzontali vicine ai bordi della *texture* e lungo una fascia orizzontale centrale, troncata ad intermittenza (si veda la Figura 13b). Il significato matematico delle tre proposizioni "$v$ è nella fascia orizziontale superiore", "$v$ è nella fascia orizzontale inferiore", "$v$ è nella fascia orizzontale centrale ed $u$ è in una delle fasce verticali di numero pari", che indicano il colore bianco della segnaletica orizzontale se almeno una di esse è verificata, ed il colore dell'asfalto se nessuna delle tre lo è, si può tradurre in termini insiemistici:

$$\begin{aligned} I_{markings} = &\left\{(u, v) : a_1 < v < b_1\right\} \cup \\ &\left\{(u, v) : a_2 < v < b_2\right\} \cup \\ &\left\{(u, v) : a_3 < v < b_3 \wedge \frac{1}{4} < \{ku\} < \frac{3}{4}\right\} \end{aligned} \tag{4.16}$$

$$I_{asphalt} = [0, 1]^2 \setminus I_{markings} \tag{4.17}$$

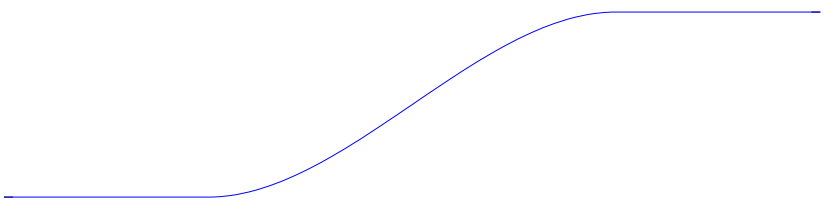


Figura 18: La funzione Smoothstep.

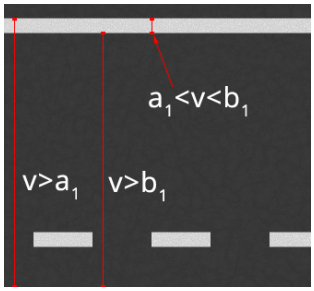



Figura 19: Un intervallo definito in termini di una funzione a gradino.

dove $[a_1, b_1]$, $[a_2, b_2]$, $[a_3, b_3]$ indicano gli intervalli verticali entro i quali sono comprese le strisce di segnaletica orizzontale rispettivamente superiore, inferiore e centrale, e $k$ è un intero che specifica il numero di segmenti verticali in cui è suddivisa la *texture* (nel caso del progetto, $k = 4$).

Si noti adesso che è possibile esprimere il luogo geometrico dei punti all'interno di ciascuna fascia in maniera alternativa, come segue, facendo uso di una funzione a gradino (si veda la figura 19):

$$v_a = \begin{cases} 1 & v \leqslant a_1 \\ 0 & v > a_1 \end{cases}, \quad v_b = \begin{cases} 1 & v \leqslant b_1 \\ 0 & v > b_1 \end{cases}$$
$$\Big\{(u, v) : a_1 < v < b_1\Big\} \equiv \Big\{(u, v) : v_b - v_a = 0\Big\} \tag{4.18}$$

Infatti, $v_b - v_a$ sarà zero se e solo se $v_b = v_a = 0$ oppure se $v_b = v_a = 1$, ovvero se $v < a_1$ oppure $v > b_1$, ed 1 altrimenti. Le definizioni di $v_a$ e $v_b$ si possono esprimere alternativamente, a questo punto, mediante la funzione SMOOTHSTEP:

$$v_a = \text{SMOOTHSTEP}(v, a_1 - \varepsilon, a_1 + \varepsilon),$$
$$v_b = \text{SMOOTHSTEP}(v, b_1 - \varepsilon, b_1 + \varepsilon),$$

con $\varepsilon$ grande quanto la dimensione del filtro di convoluzione desiderato (che dipenderà dalla distanza dei *texel* adiacenti). Si ricava così, infine, dalle 4.16 e 4.17 la formula completa per $f_{road}(u, v)$[3]:

$$\begin{aligned} f_{road}(u, v) = {} & (s(v, b_1 \pm \varepsilon) - s(v, a_1 \pm \varepsilon)) + \\ & (s(v, b_2 \pm \varepsilon) - s(v, a_2 \pm \varepsilon)) + \\ & (s(v, b_3 \pm \varepsilon) - s(v, a_3 \pm \varepsilon)) \cdot \\ & \left(s\left(\{ku\}, \frac{3}{4} \pm \varepsilon\right) - s\left(\{ku\}, \frac{1}{4} \pm \varepsilon\right)\right) \end{aligned} \tag{4.19}$$

[3] Per compattezza, si indica con $s$ la funzione SMOOTHSTEP ed i due parametri $a$, $b$ come intorni col simbolo $\pm$.

Si definisce il colore della *texture* per la segnaletica orizzontale, infine, nello stesso modo della *texture* per i mattoni:

$$\mathbf{c}_{\text{road}}(u,v) = \mathbf{c}_{\text{asphalt}} \cdot (1 - f_{\text{road}}(u,v)) + \mathbf{c}_{\text{markings}} \cdot f_{\text{road}}(u,v) \tag{4.20}$$

La *texture* degli incroci (Figura 13c) è stata realizzata con lo stesso principio. In questa vi sono semplicemente quattro settori di corona circolare agli angoli che si raccordano con i segmenti di strada. Partendo da ciascun angolo, l'equazione della corona è definita limitando in un intervallo la lunghezza di un vettore che parte dall'angolo ed arriva a ciascun punto (Figura 20). Per ottenere il vettore, è sufficiente minimizzare in entrambe le direzioni la distanza dai bordi di ciascuna componente del punto $(u,v)$ da valutare:

$$u' = \min\{u', 1-u'\}$$
$$v' = \min\{v', 1-v'\}$$

Ad esempio, quando $(u,v)$ sarà più vicino all'angolo in basso a destra, $u$ sarà prossimo ad 1 e $v$ sarà prossimo a 0, per cui $u' = 1-u$ e $v' = v$; ma l'angolo in basso a destra ha coordinate $(1,0)$, e quindi $(u',v') = (1-u,v) \equiv (1,0) - (u,v)$ è il vettore cercato. A questo punto, siano $\varepsilon$ l'ampiezza del filtro di convoluzione per l'*antialiasing*, $l = \|(u',v')\|$ la lunghezza del vettore, ed $[a,b]$ l'intervallo in cui si ha la segnaletica per gli angoli; si ottiene la formula

$$f_{\text{inters}}(u,v) = s(l, b \pm \varepsilon) - s(l, a \pm \varepsilon) \tag{4.21}$$

da applicare sostituendo $f_{\text{inters}}$ a $f_{\text{road}}$ nell'Equazione 4.20.

Per conferire un aspetto increspato sia all'asfalto, sia alla segnaletica, sono stati perturbati i colori $\mathbf{c}_{\text{asphalt}}$ e $\mathbf{c}_{\text{markings}}$ in funzione della posizione e mediante un segnale di rumore frattale. Dato che la primitiva Noise ha immagine in $[-1,1]$, si sono introdotte delle discontinuità usando come primitiva il suo valore assoluto; si è così ottenuto un aspetto organico ma con dei picchi. Le funzioni di rumore per asfalto e segnaletica sono, rispettivamente[4]:

$$\text{Noise}_{\text{asphalt}}(u,v) = \left|\frac{N(16u,16v)}{16}\right| + \left|\frac{N(32u,32v)}{16}\right| + \left|\frac{N(128u,128v)}{8}\right| \tag{4.22}$$

$$\text{Noise}_{\text{markings}}(u,v) = \left|\frac{N(128u,128v)}{8}\right| \tag{4.23}$$

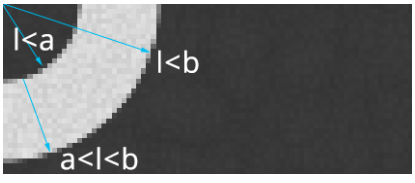



Figura 20: *Texture* per gli incroci

[4] Si indica qui con N, per compattezza, la primitiva Noise.

Figura 21: *Texture* solida per il legno

### 4.3.3 *Texture volumetrica: legno per l'arredamento*

Per rappresentare gli oggetti d'arredamento nella scena è stata realizzata una *texture* procedurale che modella il legno. Usare una *texture* bidimensionale tradizionale in questo caso presenterebbe l'importante difficoltà di far coincidere le venature del legno lungo i bordi dei poligoni, oltre a risultare innaturale su superfici curve. Questi problemi sono però stati evitati impiegando una *texture solida*: il dominio della funzione $f_{wood}$ è $\mathbb{R}^3$ anziché $\mathbb{R}^2$. Valutando quindi la *texture* in relazione alla posizione nello spazio di ciascun frammento, anziché in relazione alla posizione di ciascun frammento sulla superficie del poligono a cui appartiene, crea l'effetto di un oggetto scolpito da un blocco unico di legno.

L'idea alla base della *texture* è quella di simulare gli anelli concentrici di un tronco d'albero lungo il piano orizzontale, e perturbarli senza discontinuità nella direzione verticale. In Figura 13f sono riportate tre superfici orizzontali estratte dalla *texture*, mentre in Figura 21 è rappresentata la *texture* applicata ai tre modelli. Per modellare gli anelli concentrici, si è creato un semplice gradiente di colore governato dalla parte frazionaria della distanza di un punto dal centro della *texture*[5]. Per introdurre irregolarità nella *texture*, si è semplicemente sommata una sorgente di rumore di Perlin alla distanza dal centro prima di estrarne la parte frazionaria:

$$f_{wood}(u,v,w) = \left\{ k \left\| \left( u - \frac{1}{2}, w - \frac{1}{2} \right) \right\| + \textsc{Noise}_{wood}(u,v,w) \right\} \tag{4.24}$$

Segue la formula impiegata per sintetizzare il rumore, in cui la primitiva bidimensionale NOISE è stata valutata sia in orizzontale, per perturbare il con-

[5] Il piano $uw$ è quello orizzontale.

torno degli anelli del legno ad una data altezza, sia in verticale, per modellare la crescita irregolare del tronco.

$$\begin{aligned}\textsc{Noise}_{\text{wood}}(u,v,w) = &\frac{N(\frac{u}{8},\frac{w}{8})\cdot N(\frac{u}{8},\frac{v}{8})}{2} + \frac{N(\frac{u}{4},\frac{w}{4})\cdot N(\frac{u}{4},\frac{v}{4})}{4} + \\ &\frac{N(\frac{u}{2},\frac{w}{2})\cdot N(\frac{u}{2},\frac{v}{2})}{8} + \frac{N(u,w)\cdot N(u,v)}{16}\end{aligned} \tag{4.25}$$

Come per le altre *texture*, la funzione $f_{\text{wood}}$ ha immagine in $[0,1]$ ed è usata per interpolare tra due colori, un marrone chiaro ed uno leggermente più scuro.

$$\mathbf{c}_{\text{wood}}(u,v,w) = \mathbf{c}_{\text{darkbrown}} \cdot (1 - f_{\text{wood}}(u,v,w)) + \mathbf{c}_{\text{brightbrown}} \cdot f_{\text{wood}}(u,v,w) \tag{4.26}$$

# 5

# CONCLUSIONI E SVILUPPI FUTURI

Il progetto ha mostrato come sia possibile "disegnare" col linguaggio matematico, cercando modi di descrivere forme e colori con formalismi e funzioni. Queste tecniche vanno tuttavia impiegate con cautela: l'animazione del progetto, pur accuratamente ottimizzata, necessita di *hardware* performante per essere eseguita adeguatamente in tempo reale, poiché la valutazione di alcune delle formule procedurali è notevolmente più dispendiosa dell'impiego di modelli precostruiti e *texture* disegnate manualmente. Tuttavia si ritiene che studi mirati ed approfonditi relativamente all'architettura delle unità di elaborazione grafica o, più in generale, al calcolo parallelo potrebbero senz'altro fornire idee ed approcci all'ottimizzazione dei costi computazionali migliori rispetto a quelli impiegati nel presente progetto.

Si è inoltre riscontrata la necessità di approfondire aree dello scibile relativamente distanti dall'informatica e dalla matematica, che da sole non sono sufficienti ad ottenere risultati del tutto soddisfacenti dai punti di vista dell'accuratezza e dell'estetica. Nel caso specifico del progetto, maggiori competenze di architettura e *design* avrebbero permesso di arricchire notevolmente le grammatiche, permettendo di esprimere molta più varietà ed in maniera molto più realistica e verosimile, e maggiori competenze artistiche avrebbero permesso di discernere in maniera più efficace la qualità estetica di modelli e *texture*. È quindi auspicabile l'ausilio di figure diversificate nella progettazione di un sistema procedurale, o quantomeno un approfondimento individuale multidisciplinare.

Il lavoro qui presentato non costituisce che una ridotta panoramica sulle possibilità offerte dalla generazione procedurale di contenuti. Il progetto potrebbe essere espanso prendendo in considerazione approcci *teleologici* al fine di migliorare il realismo, ad esempio costruendo la città a partire da informazioni sulla geologia, sull'evoluzione delle strutture urbane nella storia, sull'economia e sulla popolazione [13]. Potrebbe inoltre essere impiegato per realizzare sistemi interattivi quali simulazioni o videogiochi, in virtù del fatto che il *rendering* avviene in tempo reale; i principi visti sulla generazione di numeri pseudocasuali potrebbero inoltre essere sfruttati, in questo contesto, per generare scene che cambiano ad ogni esecuzione.

# A

# TECNOLOGIE IMPIEGATE

Tradizionalmente, le *demo* vengono progettate e sviluppate per singole piattaforme hardware, con lo scopo di sfruttarne le peculiarità ed ottenere il miglior risultato possibile con le limitate risorse a disposizione. Con l'evolversi dell'industria dei calcolatori, il problema della limitazione di risorse è divenuto meno importante, ed al contempo è emerso quello della portabilità del software, ulteriormente esacerbato dalla frammentazione tra i sistemi operativi.

Il progetto si pone l'obiettivo di essere esteticamente gradevole, nonché tecnicamente fruibile. È pertanto da ricercarsi un equilibrio tra il risultato e la portabilità.

Il *Web*, in tempi recenti, è stato oggetto di numerosi sforzi, sia da parte dei fornitori dei *web browser*, sia da parte dei comitati incaricati di definire gli standard, diretti verso l'unificazione dell'esperienza utente. Il *World Wide Web Consortium* pubblica come *Recommendation* lo standard **HTML5** in data 28 ottobre 2014 [23], ufficializzando l'evoluzione del *Web* in una piattaforma per applicazioni pressoché totalmente disaccoppiata dall'hardware. Pochi anni prima, nel giugno del 2011, l'associazione *Ecma International* pubblica lo standard **ECMA-262**, la specifica del linguaggio *ECMAScript* nella versione *5.1* [21]. Questo è la base del già ampiamente diffuso linguaggio *Javascript*, originariamente sviluppato insieme al *browser Netscape Navigator 2.0*, rilasciato nel marzo 1996. Lo standard *ECMAScript 5.1* è, al momento della stesura di questa tesi, correttamente implementato da tutti i maggiori fornitori di *browser*, insieme alle *Application Programming Interface* definite in *HTML5*.

Collateralmente, il consorzio industriale *Khronos Group* ha sviluppato, e rilasciato in data 27 ottobre 2014, lo standard **WebGL** nella versione *1.0.3* [24]. Questo standard definisce un'API improntata alla grafica *3D*, basata sullo standard **OpenGL ES 2.0** e da esporsi attraverso l'API *HTML5 Canvas*. Tra i membri del gruppo di lavoro dello standard *WebGL* figurano Apple, Google, Mozilla ed Opera, quattro tra i maggiori fornitori di *web browser*. Rispetto ad altre API per il *Web*, *WebGL* non gode ancora di un supporto universalmente coerente, sia a causa della relativa novità dello standard, sia a causa delle ampie differenze di performance tra le differenti tipologie di hardware video. Si prefigura tuttavia, futuribilmente, come una delle piattaforme grafiche più diffuse e portabili.

La scelta sulle tecnologie da impiegare per il progetto è ricaduta, dunque, sul linguaggio *Javascript* e sull'API *WebGL*. In termini di performance, si è perseguito l'obiettivo di 60 *frames per second* per tutta la durata dell'animazione, su

di un calcolatore dotato di processore Intel® Core™ i7-4770k e scheda video Asus GeForce® GTX 770.

## A.1 JAVASCRIPT

Javascript è un linguaggio di programmazione interpretato multi-paradigma. Supporta uno stile di programmazione *object-oriented*, implementato con un approccio *prototype-based*. Concetti e pattern fondamentali della programmazione orientata agli oggetti classica sono possibili, ma non sempre adeguati al peculiare paradigma di Javascript. Di fondamentale importanza nel linguaggio è la presenza di funzioni *first-class*: le funzioni possono infatti essere trattate come variabili, e passate come argomento ad altre funzioni di ordine superiore. Questo permette di implementare molti concetti della programmazione funzionale, quali le λ-espressioni o l'applicazione parziale.

Un altro pattern importante è quello dei *moduli*. Il gruppo di lavoro *TC39* di *Ecma International* è attualmente al lavoro sulla futura specifica *ECMAScript 6*, la prima ad includere esplicitamente il concetto di modulo; nel frattempo, vari progetti sono stati sviluppati al fine di rendere possibile l'impiego dei moduli ed un ecosistema efficace per la riusabilità del codice. Tra questi figurano *Asynchronous Module Definition*, la cui principale implementazione è *Require.js*, e *CommonJS*, ampiamente usato nel contesto *server-side* da *Node.js* ed il suo gestore di pacchetti *npm* e portato su *browser* tramite lo strumento *Browserify*. Per la possibilità di produrre pacchetti unici, per il ridotto *overhead* di codice dovuto alla compilazione rispetto a *Require.js*, e per la possibilità di utilizzare l'archivio di *npm*, la scelta per il progetto è ricaduta su *CommonJS* e *Browserify*.

I moduli separano il codice in unità indipendenti, ciascuna delle quali può esporre tutte, nessuna o solo alcune delle funzioni o strutture dati definite al suo interno. *Browserify* realizza ciò mediante l'impiego di *chiusure*, nella forma di funzioni anonime immediatamente invocate, che costruiscono uno *scope* inaccessibile dall'esterno e che garantisce l'incapsulamento, per quanto in una forma leggermente diversa da quella della programmazione orientata agli oggetti classica.

## A.2 WEBGL

WebGL è una *Application Programming Interface* il cui obiettivo è quello di generare un'immagine, assegnando un valore cromatico ad ogni *pixel* del suo contesto, a partire dalla descrizione geometrica di questa. WebGL opera essenzialmente in due passi: il primo è quello di valutare la visibilità di una superficie (*visible surface determination*), il secondo è quello di valutare il colore di ogni pixel delle superfici visibili (*shading*).

WebGL presenta una *pipeline* programmabile. Ad una estremità di questa vi è la trasmissione di informazioni geometriche nella forma di una sequenza di punti, a loro volta rappresentati come gruppi di numeri *floating-point* associati alla dichiarazione del tipo di primitiva che ciascun punto della sequenza andrà a rappresentare (punti, linee o triangoli). L'unica primitiva di interesse nell'ambito del progetto è il triangolo; l'insieme di punti così definito è denominato una *mesh di triangoli*. Il formato scelto per il progetto, `GL_TRIANGLES`, è il più semplice, ma anche il più flessibile: consiste in quanto segue.

**Definizione A.2.1** (Vertice)**.** Sia

$$M = m_0, m_1, ..., m_n, \quad m_i \in \mathbb{R},\ 0 \leqslant i < n, \quad n \equiv 8 \pmod 9$$

una successione di numeri reali. Se $i \equiv 0 \pmod 3$, allora la tripla

$$\begin{pmatrix} m_i \\ m_{i+1} \\ m_{i+2} \end{pmatrix}$$

rappresenta un *vertice* in $\mathbb{R}^3$. Di seguito, se $p$ è un vertice, allora si indicheranno con $p_x,\ p_y,\ p_z$ rispettivamente la prima, la seconda e la terza componente del vertice.

**Definizione A.2.2** (Mesh di triangoli)**.** Inoltre, se $i \equiv 0 \pmod 9$, allora i tre vertici

$$\begin{pmatrix} m_i \\ m_{i+1} \\ m_{i+2} \end{pmatrix}, \quad \begin{pmatrix} m_{i+3} \\ m_{i+4} \\ m_{i+5} \end{pmatrix}, \quad \begin{pmatrix} m_{i+6} \\ m_{i+7} \\ m_{i+8} \end{pmatrix}$$

rappresentano un triangolo. L'insieme dei triangoli descritti da $M$ secondo questo formato costituisce la *mesh*.

Una volta preparata una *mesh*, è necessario trasferirne i dati sulla scheda grafica per mezzo di un *buffer*. Di seguito, un esempio del codice *Javascript* necessario a trasmettere un unico triangolo, avente vertici

$$\begin{pmatrix} -1 \\ -1 \\ 0 \end{pmatrix}, \quad \begin{pmatrix} -1 \\ 1 \\ 0 \end{pmatrix}, \quad \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}.$$

```
var position = gl.createBuffer();

gl.bindBuffer(gl.ARRAY_BUFFER, position);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([
  -1, -1, 0,  -1, 1, 0,  1, 1, 0,
]), gl.STATIC_DRAW);
```

Dal momento in cui i dati sono pronti, il metodo `gl.drawArrays()` sancirà l'inizio delle operazioni da parte della GPU. A questo punto verrà invocato, parallelamente su tutti i vertici, il primo componente della *pipeline* programmabile: il *vertex shader*.

Il *vertex shader* è un programma scritto in linguaggio *GLSL*, contrazione di *OpenGL Shading Language*, un linguaggio imperativo di alto livello ispirato al *C* ed ideato per eseguire calcoli paralleli sulla scheda grafica. *GLSL* è dotato di numerosi tipi di dato e funzioni *built-in* ottimizzate per gestire la geometria. Vi sono ad esempio il supporto per l'algebra matriciale, fondamentale per calcolare trasformazioni geometriche, funzioni per calcolare soglie (`step`, `smoothstep`), troncare valori (`clamp`), calcolare interpolazioni lineari (`mix`).

Il *vertex shader*, nella sua forma più semplice, riceve in input un vertice ed ha il compito di produrre in output un altro vertice, portando il suo input in *normalized device coordinates* (*NDC*). Questo è un sistema di coordinate relativo al contesto di *rendering*, il cui volume è delimitato dai vertici $(-1,-1,-1)$ ed $(1,1,1)$. Se $p$ è un vertice in NDC, allora $(p_x, p_y)$ indica la posizione del vertice relativamente al contesto di *rendering*, dove $(-1,-1)$ indica l'angolo in alto a sinistra ed $(1,1)$ quello in basso a destra, e $p_z$ indica la profondità del vertice, ed ha lo scopo di essere sottoposto al *depth test* oppure di contribuire al *blending*.

Quasi mai, tuttavia, un *vertex shader* riceve unicamente un vertice come input: nella maggior parte dei casi, altre variabili concorrono alla trasformazione del vertice in NDC. Queste possono essere fondamentalmente di due tipi. Il primo, caratterizzato dal qualificatore `uniform`, indica una variabile avente il medesimo valore per ciascuno dei vertici su cui lo *shader* viene invocato. È il caso delle matrici di modello, di vista e di proiezione prospettica. Lo scopo della *model matrix* è quello di portare il vertice di una *mesh* dal proprio spazio locale ad un sistema di coordinate comune a tutte le altre *mesh* definito *world space*; intuitivamente, la stessa *mesh* può essere renderizzata più volte con differenti matrici *model* venendo replicata in posizioni o dimensioni diverse nella scena. La *view matrix* funge invece da "telecamera": è comune a tutte le *mesh* di una scena e serve a trasformare tutto il *world space*, cosicché all'origine corrisponda la posizione della telecamera ed al versore $\hat{\mathbf{z}}$ corrisponda la sua direzione. Lo spazio ottenuto a questo punto della trasformazione è denominato *clip space*.

Infine, la *perspective projection matrix* ha il compito di trasformare i punti in *clip space* in coordinate NDC. La matrice di trasformazione prospettica rappresenta un'omografia che associa uno ed un solo punto in coordinate NDC ad ogni punto, in *view space*, che giace all'interno di un tronco di piramide. La forma di questo è definita da quattro parametri: l'angolo del campo visivo (*field of vision*), il rapporto d'aspetto (*aspect ratio*), il *near plane* ed il *far plane*. Questi indicano, rispettivamente, l'angolo verticale alla sommità della piramide, il rapporto tra la dimensione orizzontale e quella verticale della piramide, e la distanza lungo l'asse $\mathbf{z}$ dei due piani che vanno a troncarla.

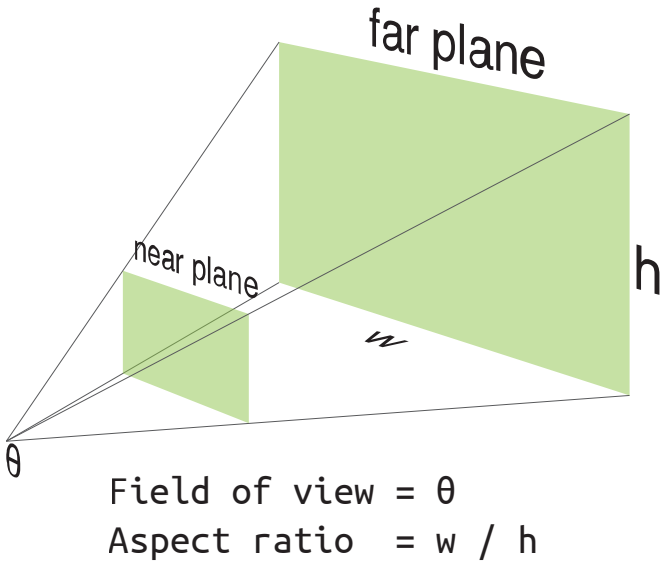



Figura 22: Volume di vista nella proiezione prospettica.

La trasformazione prospettica, a differenza di quelle associate alle matrici *model* e *view*, non opera in uno spazio *affine* ma in uno spazio *proiettivo* i cui punti vengono rappresentati mediante *coordinate omogenee*. Introdotte da August Ferdinand Möbius, le coordinate omogenee rappresentano, mediante quantità finite, i *punti all'infinito* in cui le rette parallele si intersecano.

**Definizione A.2.3** (Coordinate omogenee)**.** Sia $p \in \mathbb{R}^3$. Allora, il punto $\dot{p} \in \mathbb{R}^4$ è una rappresentazione di $p$ in *coordinate omogenee* se e solo se

$$\dot{p} = w \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}, \quad w \in \mathbb{R}$$

A valori tendenti a $0$ di $w$ corrisponderanno punti la cui distanza dall'occhio tende all'infinito; le coordinate così ottenute convergeranno al punto $\mathbf{o}$ in NDC, ovvero al centro del contesto di *rendering*.

Il secondo tipo di variabili che un *vertex shader* può ricevere è caratterizzato dal qualificatore `attribute`; a differenza delle variabili `uniform`, ad ogni vertice di una *mesh* è associata esattamente un'istanza di una variabile `attribute`. Ad esempio, in molti modelli di *shading* è necessario fornire al programma *GLSL* il vettore normale alla superficie di un triangolo, ed ad ogni vertice della *mesh* verrà quindi associato un vettore normale (risultante dal contributo delle normali alle superfici di tutti i triangoli ad esso adiacenti). Il vertice stesso, in realtà, è

una variabile `attribute`, pertanto il metodo di caricamento dell'informazione relativa a tutte le variabili `attribute` sarà il medesimo precedentemente visto.

Segue il codice di un *vertex shader* essenziale: vi si calcolano le coordinate NDC del vertice e si trasferisce al passo successivo, il *fragment shader*, la normale alla superficie in corrispondenza del vertice.

```
uniform mat4 projection, view, model;
attribute vec3 vertex, normal;

varying vec3 vNormal;

void main() {
  vNormal = normal;
  gl_Position = projection * view * model * vec4(vertex, 1.);
}
```

Il *fragment shader* è anch'esso un programma scritto in linguaggio *GLSL*, ma che tratta insiemi di dati sostanzialmente differenti rispetto al *vertex shader*. Al termine dell'esecuzione del *vertex shader*, la *pipeline* di WebGL prevede un passo non programmabile, denominato *rasterizzazione*: questo si occupa di generare effettivamente le posizioni dei *pixel* che dovranno essere colorati, interpolando linearmente i risultati forniti dal *vertex shader* ed invocando il *fragment shader* sull'insieme di dati così ottenuto. L'*input* fondamentale del *fragment shader* è la variabile `gl_FragCoord`, indicante la posizione del pixel sullo schermo. Come nel caso del *vertex shader*, tuttavia, raramente questa è l'unica informazione processata; è possibile infatti aggiungere al processo di rasterizzazione ulteriori variabili, definite col qualificatore `varying`, esportandole dal *vertex shader*; giungeranno così, interpolate linearmente dal processo di rasterizzazione, al *fragment shader*, dove potranno essere lette; concorreranno qui alla decisione dell'*output* finale, ovvero il colore del pixel.

Segue il codice di un *fragment shader* essenziale, compatibile col *vertex shader* precedentemente riportato. Esso calcola una versione semplificata del modello di illuminazione Blinn-Phong per luci direzionali, trattato in dettaglio più avanti: solo il termine di *riflessione diffusa* viene calcolato.

```
uniform vec3 lightDirection;

varying vec3 vNormal;

void main() {
  const vec3 color = vec3(1., 0., 0.);
  float diffuse = max(0., dot(vNormal, lightDirection));
  gl_FragColor = vec4(color * diffuse, 1.);
}
```

# BIBLIOGRAFIA


[1] Stiny, George, Gips, James, "Shape grammars and the generative specification of painting and sculpture", *Information Processing 71, 14601465. North-Holland Publishing Company*, 1972 (Cited on pages 5 and 8.)

[2] Catmull, Edwin Earl, "A Subdivision Algorithm for Computer Display of Curved Surfaces.", *The University of Utah*, 1974 (Cited on page 39.)

[3] Phong, Bùi Tuong, "Illumination for Computer Generated Pictures", *Department of Computer Science, University of Utah, UTEC-CSs-73-129*, 1975 (Cited on page 31.)

[4] Blinn, James F., "Models of Light Reflection for Computer Synthesized Pictures", *SIGGRAPH Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, 1977 (Cited on page 32.)

[5] Blinn, James F., "Simulation of Wrinkled Surfaces", *SIGGRAPH Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, 1978 (Cited on page 40.)

[6] Perlin, Ken, "An Image Synthesizer", *SIGGRAPH Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, 1985 (Cited on page 7.)

[7] Kajiya, James T., "The Rendering Equation", *SIGGRAPH Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, 1986 (Cited on page 29.)

[8] Deering, Michael, Winner, Stephanie, Schediwy, Bic, Duffy, Chris, Hunt, Neil, "The Triangle Processor and Normal Vector Shader: A VLSI System for High Performance Graphics", *SIGGRAPH Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*, 1988

[9] Saito, Takafumi, Takahashi, Tokiichiro, "Comprehensible Rendering of 3-D Shapes", *SIGGRAPH Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, 1990

[10] Bar-Hillel, Maya, Wagenaar, Willem A, "The Perception of Randomness", *Advances in Applied Mathematics, Volume 12, Issue 4*, 1991 (Cited on page 44.)

[11] Měch, Radomír, Prusinkiewicz, Przemyslaw, "Visual Models of Plants Interacting with Their Environment", *SIGGRAPH Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, 1996 (Cited on pages 5 and 10.)

[12] Matsumoto, Makoto, Nishimura, Takuji, "Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator", *ACM Transactions on Modeling and Computer Simulation, Volume 8 Issue 1*, 1998 (Cited on page 28.)

[13] Parish, Yoav I. H., Müller, Pascal, "Procedural Modeling of Cities", *SIGGRAPH Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, 2001 (Cited on page 55.)

[14] Ebert, David S., Musgrave, F. Kenton, Peachey, Darwyn, Perlin, Ken, Worley, Steven, "Texturing and Modeling: A Procedural Approach, 3rd edition", *Morgan Kaufmann Publishers Inc.*, 2002 (Cited on page 45.)

[15] Perlin, Ken, "Improving Noise", *SIGGRAPH Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, 2002 (Cited on page 45.)

[16] Mittring, Martin, "Finding Next Gen - CryEngine 2", note del corso *SIGGRAPH 2007 Advanced Real-Time Rendering in 3D Graphics and Games*, 2007 (Cited on page 34.)

[17] Akenine-Moller, Tomas, Hoffman, Naty, Haines, Eric, "Real-Time Rendering 3rd Edition", *A. K. Peters, Ltd.*, 2008 (Cited on pages 29, 32, and 33.)

[18] West, Mick, "Random Scattering: Creating Realistic Landscapes", `http://www.gamasutra.com/view/feature/130071/random_scattering_creating_.php?page=2`, 2008 (Cited on page 7.)

[19] Burger, Wilhelm, Burge, Mark J., "Principles of Digital Image Processing: Fundamental Techniques", *Springer Publishing Company, Incorporated*, 2009

[20] Aras Pranckeviius, "Compact Normal Storage for Small G-Buffers", `http://aras-p.info/texts/CompactNormalStorage.html`, 2010 (Cited on page 37.)

[21] Ecma International, "EcmaScript® Language Specification, 5.1 edition", `http://www.ecma-international.org/ecma-262/5.1/`, 2011 (Cited on page 57.)

[22] Miguel Cepero, "Building Rooms", `http://procworld.blogspot.com/2012/03/building-rooms.html`, 2012 (Cited on pages 20 and 24.)

[23] World Wide Web Consortium, "HTML5 - A vocabulary and associated APIs for HTML and XHTML", `http://www.w3.org/TR/2014/REC-html5-20141028/`, 2014 (Cited on page 57.)

[24] Khronos Group, "WebGL Specification", https://www.khronos.org/registry/webgl/specs/1.0/, 2014 (Cited on page 57.)

[25] Weisstein, Eric W., "Necker Cube.", http://mathworld.wolfram.com/NeckerCube.html (Cited on page 10.)