Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Analisi di software open source per la grafica 3D: dagli algoritmi alla produzione di scene 3D

Relatore:
Chiar.mo Prof.
GIULIO CASCIOLA

Presentata da:
GIULIO LEONE

Sessione II
Anno Accademico 2017/2018

Introduzione

Questo elaborato descrive il lavoro di studio e utilizzo di tre software *open source* per la produzione di una scena 3D. Il suo obiettivo è quello di fornire un esempio di applicazione di tali software in un contesto reale, mostrando un parallelismo tra gli aspetti teorici della *computer graphics* e le tecnologie *open source* disponibili per la loro implementazione.

In particolare, l'attività di progetto qui descritta è la ricostruzione 3D di un appartamento, sfruttando un software di modellazione grafica (*Blender*), un software per l'elaborazione di *mesh* 3D (*MeshLab*) e un software per l'acquisizione di modelli 3D tramite fotogrammetria (*Regard3D*). L'utilizzo di questi software ha permesso la costruzione della scena 3D, dalla quale infine sono stati realizzati alcuni *rendering* e un video.

La caratteristica di questi pacchetti software di essere *open source* ha permesso all'autore di studiare gli algoritmi implementati, giustificando così la scelta o l'esclusione di certe tecniche nel corso del lavoro di progetto.

Gli argomenti trattati in questo lavoro sono strutturati in tre capitoli che rappresentano tre diversi livelli di analisi dello stesso prodotto:

- il primo capitolo tratta in breve l'intero processo produttivo, dall'ideazione della scena alla produzione del video, descrivendo le tecnologie utilizzate e motivando le scelte adottate;

- il secondo capitolo descrive dettagliatamente ciascuna fase, analizzando dapprima i software utilizzati e le singole operazioni svolte e mostrando di volta in volta il risultato della loro esecuzione;

- il terzo capitolo conclude l'elaborato fornendo una descrizione sintetica degli algoritmi implementati nei software *open source* e utilizzati per la produzione della scena 3D.

Infine, la bibliografia contiene i riferimenti agli articoli di ricerca e manuali tecnici utilizzati per approfondire gli argomenti trattati in questo lavoro.

Indice

Elenco delle figure

Capitolo 1

Realizzazione scena 3D

Il primo capitolo espone ad alto livello il processo di modellazione e resa effettuato per la produzione della scena 3D. Dopo una breve introduzione in cui si definisce e motiva la scena da rappresentare, si procede con la descrizione delle attività svolte in forma schematica, concentrandosi principalmente sui prerequisiti di ciascuna fase e sulle motivazioni per le scelte grafiche adottate. Nel capitolo successivo, gli aspetti qui trattati in forma prettamente teorica vengono ripresi, dandone un'esposizione più tecnica.

1.1 Definizione della scena

Gli interni di un appartamento costituiscono uno scenario ideale per l'utilizzo dei software di modellazione grafica in tutte le loro funzionalità. Grazie alla vasta gamma di materiali e forme degli oggetti che si possono trovare in una casa, una rappresentazione di interni è il contesto perfetto per l'applicazione di diversi strumenti di modellazione, *texturing* e *rendering*, fornendo inoltre un valido pretesto per verificare come tali tecniche differiscono nei loro dettagli implementativi.

L'attività di progetto descritta in questo elaborato riguarda pertanto la modellazione 3D e successiva resa di un appartamento, utilizzando varie tecniche per la produzione di una scena che realisticamente mostri gli spazi e

gli arredi. Lo svolgimento di questo lavoro è stato guidato dall'idea di realizzare un prodotto 3D che possa essere utilizzato come punto di partenza per diverse applicazioni, come la resa di alcune inquadrature, la produzione di un filmato per la visualizzazione dell'intero appartamento o la creazione di una visita interattiva tramite l'utilizzo di un *game engine*.

Il lavoro svolto si può inserire in un contesto di produzione di un modello 3D di una casa in fase di costruzione, commissionato ad esempio da un'agenzia immobiliare che vuole mostrare ai propri clienti quali siano le dimensioni e l'aspetto finale dell'appartamento, insieme a un'eventuale ipotesi d'arredo. La costruzione del modello si basa su un appartamento reale, al fine di modellare in scala reale gli spazi e gli oggetti di arredamento e quindi produrre un modello dall'aspetto realistico, mantenendo comunque una certa libertà stilistica nella scelta degli oggetti da modellare e *texture* da applicare su di essi. In particolare, sono stati scelti gli elementi di arredo la cui forma e colore richiedono tecniche di modellazione e *texturing* differenti.

L'appartamento scelto come riferimento per la modellazione è composto da tre ambienti: uno più ampio contenente la cucina e il salotto e due di dimensioni minori corrispondenti alla stanza da letto e il bagno. Si è scelto di fornire una rappresentazione notturna, quindi illuminata artificialmente dalle fonti luminose interne della casa, per simulare un'illuminazione prodotta da più luci e quindi la generazione e sovrapposizione di zone di ombra e penombra.

1.2 Acquisizione modelli 3D

1.2.1 Cos'è la Structure From Motion

La creazione di un modello 3D di un ambiente reale può avvalersi, oltre alle classiche tecniche di modellazione grafica, anche di strumenti per l'acquisizione automatica della forma e del colore di alcuni oggetti; questi utilizzano scansioni laser o fotografie al fine di ottenere una digitalizzazione dell'oggetto in un formato adatto alla visualizzazione 3D.

La tecnica utilizzata per questo scopo è detta fotogrammetria, o *Structure From Motion* (*SFM*) [1], la quale, a differenza di una scansione laser, è economicamente alla portata di tutti e non richiede particolari strumentazioni, se non una semplice fotocamera. Essa è una tecnica di *computer vision* che consente di generare modelli 3D a partire da fotografie che riprendono un oggetto da vari punti di vista e angoli di rotazione (Fig. 1.1). Il modello viene creato estraendo dalle fotografie alcuni descrittori puntiformi (*feature*) i quali, incrociati con i punti di altre immagini, possono essere posizionati in uno spazio 3D sfruttando i parametri fotografici della camera utilizzata per la ripresa, come la lunghezza focale (ossia la distanza tra la lente e il fuoco della camera) e la profondità del sensore.

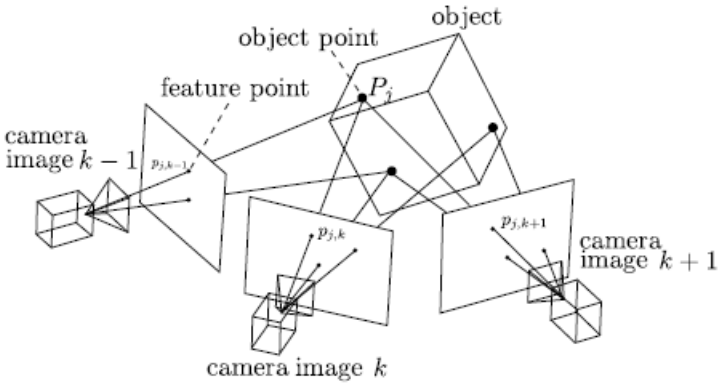


Figura 1.1: *Structure From Motion*. Immagine tratta da [14]

Un vantaggio degli algoritmi di *SFM*, infatti, è quello di poter utilizzare una qualsiasi macchina fotografica, purché se ne conoscano le due caratteristiche tecniche suddette. La maggioranza dei software dedicati alla fotogrammetria sono comunque in grado di estrarre tali informazioni dai metadati delle foto, oppure sfruttano un database contenente vari modelli di macchine fotografiche per recuperare tali dati. Nel corso di questo lavoro, è stata utilizzata una fotocamera *Sony Cybershot DSC-W690* di 16,1 *megapixel* con i seguenti parametri fotografici:

- zoom ottico 10x;

- lente angolare di 25 millimetri;
- lunghezza focale di 4.45 millimetri;
- profondità del sensore di 6.16 millimetri.

Descriviamo ora in breve quali sono i passi fondamentali per una ricostruzione 3D di un oggetto tramite fotogrammetria:

- Tramite euristiche e analisi locale, si estraggono i punti salienti (*keypoint* o *feature*) delle immagini di input. Una *feature* non è altro che un punto di un'immagine con un'alta probabilità di essere individuato in altre immagini dello stesso oggetto, ad esempio angoli, spigoli o cambiamenti netti di colore;
- Descritte le *feature* con un formalismo matematico, si esegue una procedura di *matching* per abbinare i *keypoint* calcolati in ciascuna coppia di immagini;
- Da questi abbinamenti, si calcola la posizione e i parametri fotografici della camera in ciascuno scatto in modo da posizionare il *keypoint* in un spazio 3D, producendo una nuvola di punti sparsa;
- Si aumenta la densità della nuvola sparsa tramite una procedura di densificazione;
- Infine, si costruisce una superficie che interpola i punti della nuvola densa calcolata.

1.2.2 Produzione di un buon picture set

Nel capitolo 2.1 è descritto ad esempio come queste operazioni possano essere eseguite in *Regard3D*, il software orientato alla fotogrammetria utilizzato nel corso di questo lavoro. In questa sezione si vuole invece porre l'attenzione sulla fase preliminare di costruzione di un buon insieme di fotografie (*picture set*) che ritraggono l'oggetto da vari punti di vista.

È fondamentale che questa fase sia eseguita nel modo corretto e con la massima cura, in quanto il risultato della ricostruzione dipende fortemente dalla qualità delle immagini prodotte; infatti, a volte la ricostruzione potrebbe anche fallire se le immagini di input non sono in numero e/o qualità adeguati.

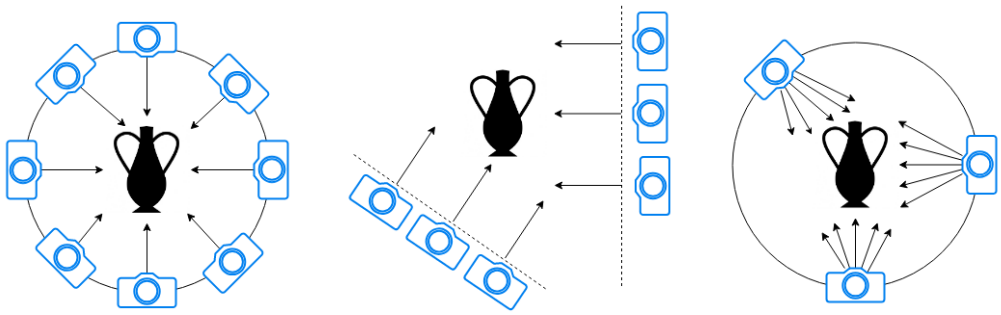
Vi sono una serie di accorgimenti e restrizioni su come creare un buon *picture set*: in generale, un *picture set* con molte immagini dà risultati migliori rispetto ad uno composto da poche fotografie. Anche la qualità delle immagini gioca un ruolo importante: più le foto sono ad alta risoluzione (numero di pixel), più dettagliato sarà il modello estratto, in quanto più punti 3D verranno generati. Un'alta risoluzione implica, tuttavia, un incremento del tempo di elaborazione e costruzione del modello. Una risoluzione di 5 megapixel, ad esempio, è più che sufficiente per garantire una buona ricostruzione.

Ovviamente, anche la qualità della fotocamera influisce sul risultato: una fotocamera con una buona lente riduce gli effetti di distorsione, portando quindi a immagini meno sfocate e di qualità migliore. Bisogna fare inoltre attenzione a eliminare dal *picture set* le immagini sfocate (ad esempio, scattate con la fotocamera non perfettamente ferma) nelle quali alcuni *keypoint* necessari per il *feature matching* potrebbero non essere visibili.

La luce può produrre zone d'ombra e di penombra che possono falsificare il colore e la forma dell'oggetto. Per questo motivo, il tempo ideale durante il quale si consiglia di scattare le fotografie è quello nuvoloso, in modo che il soggetto della ricostruzione sia globalmente illuminato e la luce non produca ombre. Per lo stesso motivo, conviene effettuare la ripresa all'aperto ed evitare un'illuminazione artificiale che può causare vari effetti di luce indesiderati. Per articoli con *self occlusion*, le cui forme concave possono produrre ombre su se stesse, bisogna assicurarsi di riprendere tutte le parti dell'oggetto, anche quelle occluse. In generale, il *picture set* deve contenere fotografie scattate da tutti i possibili angoli di vista per evitare buchi nel modello finale.

Una buona sequenza di ripresa è la seguente: tenendo sempre l'oggetto

inquadrato e mantenendo lo zoom fisso, camminare intorno ad esso seguendo una traiettoria circolare e scattando una fotografia dopo pochi passi (Fig. 1.2 (a)). Per favorire un corretto bilanciamento di punti 3D in tutte le parti del modello, si possono effettuare più riprese circolari a diverse altezze e a diverse distanze. Non bisogna invece effettuare una ripresa variando la distanza della fotocamera dall'oggetto, ad esempio avvicinandosi o allontanandosi o muovendosi ortogonalmente o parallelamente alla scena inquadrata (Fig. 1.2 (b)). Da evitare anche lo scatto di più fotografie dallo stesso punto di vista ma con rotazione differente, come quando si registra un panorama (Fig. 1.2 (c)). Queste ultime tecniche, infatti, non permettono di ottenere sufficienti informazioni sulla forma tridimensionale dell'oggetto.



(a) (b) (c)

Figura 1.2: Sequenza di fotografie corretta (a) e due esempi di sequenze errate (b, c)

Le fotografie devono essere scattate con una leggera variazione di angolo l'una dall'altra. In altre parole, ci deve essere una sovrapposizione parziale (circa 60%) tra fotografie successive, sufficiente da rendere possibile l'abbinamento dei *keypoint*. Ogni *feature* estratta viene collocata in uno spazio 3D solo se essa viene individuata in almeno 3 fotografie e correttamente abbinata tra queste da un algoritmo di *feature matching*. Ad esempio, un angolo di 10-20 gradi garantisce una buona percentuale di sovrapposizione tra le immagini di input.

Infine è importante tener presente che non tutti gli oggetti possono essere

acquisiti con tecniche di fotogrammetria: innanzitutto il requisito fondamentale è che gli oggetti debbano essere fermi. Inoltre, una ripresa a 360 gradi non deve essere eseguita ruotando l'oggetto, ma è la camera che deve ruotare intorno ad esso, altrimenti tutte le fotografie verrebbero scattate dalla stessa direzione e non sarebbe possibile utilizzare i parametri fotografici per il passaggio da *feature* 2D a punti 3D. Superfici bidimensionali o planari devono essere anch'esse escluse, in quanto non possono essere riprese da tutti i possibili angoli di vista.

Gli oggetti devono avere inoltre una *texture* ben definita e sufficiente per distinguerne le singole parti e per separarli dallo sfondo. Sono da escludere quindi articoli monocolore e le superfici riflettenti, che cambiano colore e/o intensità luminosa quando si varia il punto d'osservazione (es. specchi, vetri e così via).

1.2.3 Modelli creati

Un *picture set* realizzato è mostrato in figura 1.3: esso contiene 32 fotografie di un piccolo vaso, la cui colorazione variegata lo rende particolarmente adatto per l'individuazione dei descrittori.

Tramite i software *Regard3D* e *MeshLab*, sono stati realizzati in questo modo quattro modelli (Fig. 1.4), posizionati all'interno della scena come soprammobili. Sono stati scelti un vaso, per la sua *texture* particolarmente colorata, un modello di porcellana di un mercante per la sua notevole complessità, un piccolo modellino in pietra di un trullo e una statua di un cherubino, il cui *picture set* è liberamente scaricabile[1]. Il numero di fotografie per ciascun *picture set* varia a seconda della complessità dell'oggetto ripreso: mentre per il vaso sono state sufficienti 32 fotografie, il modello del mercante ha richiesto ben 137 immagini per una totale copertura dei suoi dettagli e delle parti occluse.

[1] https://www.3dflow.net/it/3df-zephyr-reconstruction-showcase/

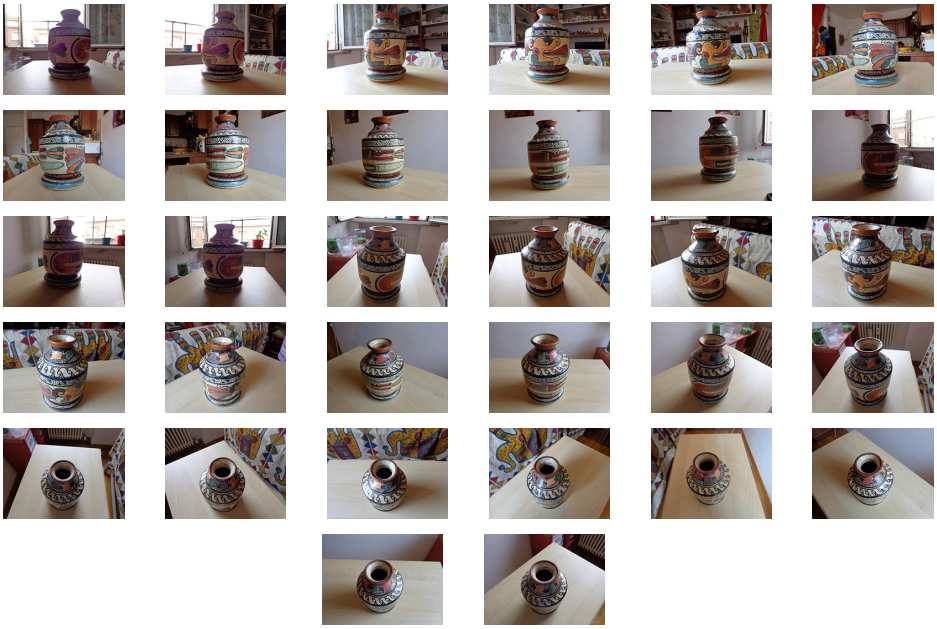


Figura 1.3: Le 32 fotografie utilizzate per la ricostuzione di un vaso

Figura 1.4: Modelli creati tramite *Structure From Motion*

1.3 Modellazione

Tutti gli altri oggetti della scena sono stati realizzati tramite varie tecniche di modellazione grafica, ossia modellando una *mesh* poligonale che approssima la forma dell'oggetto reale. Il software utilizzato per questo scopo è *Blender* (vedi 2.2).

Ogni oggetto scelto per la modellazione è stato dapprima misurato in modo da poter così generare una *mesh* che abbia le medesime dimensioni e proporzioni. Sulla base delle misure effettuate, ogni oggetto è stato temporaneamente rappresentato da un parallelepipedo, in modo da avere un riferimento spaziale per la sua posizione e dimensione. Infatti, nella sua prima versione, la scena era sostanzialmente un insieme di cubi, grazie ai quali è stato possibile suddividere facilmente gli ambienti e descrivere gli spazi.

Nella maggioranza dei casi, le *mesh* modellate sono state realizzate a partire da una primitiva grafica, ossia inserendo un oggetto predefinito 2D (piano, cerchio) o 3D (cubo, cilindro, sfera, cono) dal quale si è provveduto all'aggiunta o rimozione di vertici, lati e facce fino ad ottenere la forma voluta. Alcuni oggetti come l'armadio o la libreria sono, infatti, semplici composizioni di cubi parzialmente modificati nella loro topologia (Fig. 1.5).

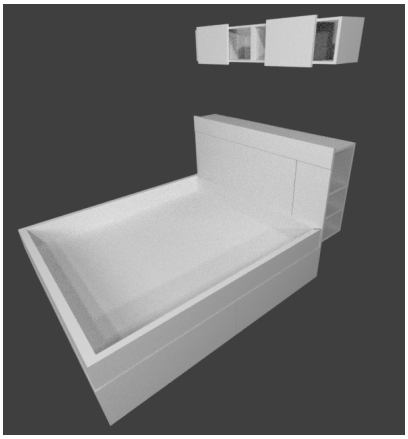
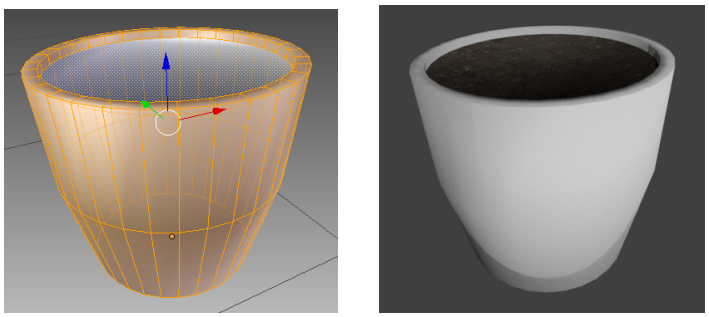


Figura 1.5: La struttura del letto, il comodino e la libreria sono oggetti realizzati usando solo parallelepipedi

La tecnica prevalentemente utilizzata per modificare la forma di una primitiva è l'estrusione, ossia l'allungamento di una porzione della *mesh* lungo una direzione; ad esempio, estrudendo un vertice si ottiene un lato, da uno lato si può ricavare una faccia e da una faccia si può ottenere un solido. Grazie all'estrusione, unitamente alle operazioni di scala, rotazione e traslazione, è stata modellata la maggior parte degli oggetti della scena. Ad

esempio, il vaso per le piante (Fig. 1.6) è stato realizzato a partire da un cerchio, applicando in modo alternato le operazioni di estrusione verso l'alto e modifica della scala. Una volta raggiunta l'altezza voluta, sono state utilizzate contemporaneamente l'estrusione e la riduzione della scala per fornire uno spessore all'oggetto. Infine, applicando un'estrusione verticale, è stata generata la faccia superiore del vaso.



(a) (b)

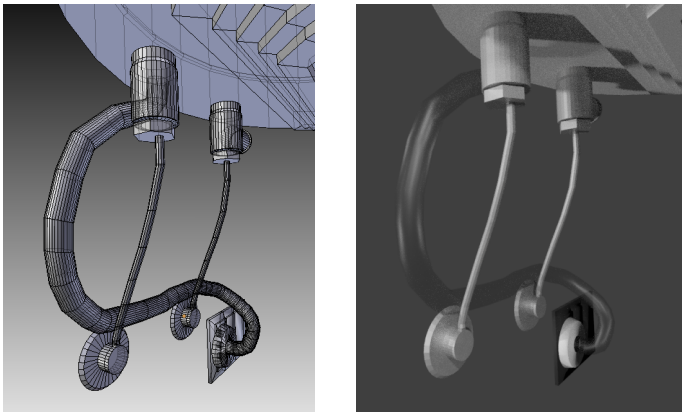
Figura 1.6: Modellazione (a) e resa (b) di un vaso

Gli oggetti che si sono rivelati essere più impegnativi nel disegno sono stati la caldaia (Fig. 1.7) e i sanitari (Fig. 1.8). Mentre il primo è stato realizzato tramite numerose operazioni di estrusione, rotazione e scala per la modellazione di tutte le sue parti, in particolare i cavi, i secondi hanno richiesto, inoltre, l'applicazione di diversi modificatori per produrne la forma, come ad esempio operazioni insiemistiche, suddivisioni ripetute della superficie e fattori di smussatura.

Per la realizzazione degli oggetti più complessi si è fatto ricorso a tecniche più avanzate di modellazione grafica, come l'uso di sistemi particellari o curve matematiche. L'utilizzo di queste tecniche è descritto dettagliatamente nel capitolo 2.

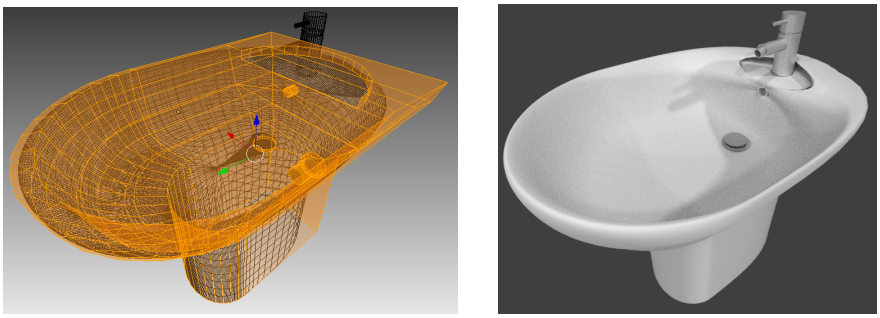
1.4 Resa

Il processo di resa di un modello 3D consiste sostanzialmente in tre fasi:



(a) (b)

Figura 1.7: Modellazione (a) e resa (b) della caldaia



(a) (b)

Figura 1.8: Modellazione (a) e resa (b) dei sanitari

- *Texturing* o *texture mapping*, ossia l'assegnazione di un materiale per la definizione del colore e aspetto di ciascun oggetto della scena;
- *Lighting*, ossia la disposizione delle fonti di luce per la gestione dell'illuminazione, producendo ombre, effetti di luce e riflessioni;
- *Rendering*, ossia il calcolo della luminosità e colore di ciascun pixel della scena inquadrata dall'osservatore per la produzione di un'immagine.

In certi casi, le *texture* utilizzate per la colorazione degli oggetti sono state estratte da alcune fotografie dell'appartamento, come ad esempio i quadri o la coperta del divano (Fig. 1.9 (a)). Per altri oggetti, la cui posizione o

l'assenza di un'illuminazione adeguata hanno reso impossibile effettuare una buona fotografia dei loro materiali, sono state scaricate alcune immagini *texture* visibilmente simili ai corrispondenti materiali da rappresentare (Fig. 1.9 (b)). In particolare, i siti *Textures*[2] e *Poliigon*[3] offrono una notevole quantità di *texture* adatte per tutti i tipi di oggetti. Queste immagini sono scaricabili gratuitamente, previa registrazione di un account. Si è cercato di utilizzare prevalentemente immagini *seamless*, ossia senza soluzione di continuità, in modo da poter arbitrariamente variare la scala del *texture mapping* affiancando più volte l'immagine sulle facce del modello, senza che sia percepibile la ripetizione effettuata.

Figura 1.9: Fotografia di una porzione della coperta del divano (a) e *texture* di un legno scaricata dal sito *Textures* (b)

Come già detto, si vuole fornire una rappresentazione notturna della scena; a tal fine non sono state disposte luci all'esterno dell'appartamento, come anche le finestre sono rappresentate chiuse. Ogni stanza viene illuminata da una fonte luminosa omnidirezionale, insieme ad altre sorgenti luminose di intensità minore per l'illuminazione degli angoli e delle zone più buie. In figura 1.10 sono mostrate le quattro fonti luminose inserite nella scena, rappresentate con piccoli cerchi neri.

A questo punto, si può eseguire la procedura di *rendering*, ossia l'utilizzo del modello 3D per la generazione di un'immagine che ritrae la scena inquadrata dall'osservatore (ossia la camera). La creazione dell'immagine avviene

[2]https://www.textures.com/
[3]https://www.poliigon.com/

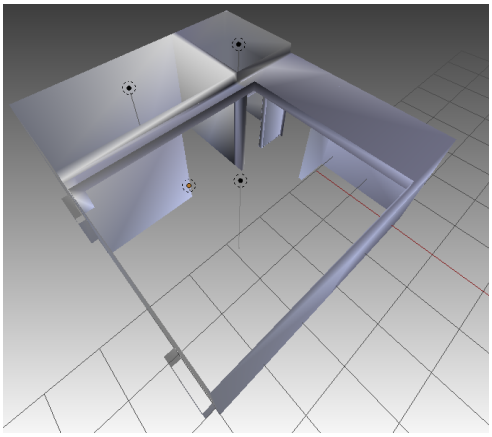


Figura 1.10: Le quattro luci inserite nella scena

un pixel alla volta: per ogni pixel, si calcola il suo colore a partire dal materiale dell'oggetto inquadrato e dalla quantità di luce che riceve direttamente dalle fonti luminose e indirettamente dai raggi riflessi dagli altri oggetti della scena.

Le figure 1.11, 1.12, 1.13, 1.14 e 1.15 mostrano ad esempio alcuni *rendering* del modello confrontati con le fotografie dell'appartamento scattate nelle medesime inquadrature. Come si può notare, a meno di qualche variazione nella scelta delle *texture* e nella quantità di oggetti realizzati, le immagini renderizzate sono piuttosto fedeli alle corrispondenti fotografie.

(a) (b)

Figura 1.11: Confronto tra fotografia (a) e *rendering* (b) della cucina

(a) (b)

Figura 1.12: Confronto tra fotografia (a) e *rendering* (b) del salotto

(a) (b)

Figura 1.13: Confronto tra fotografia (a) e *rendering* (b) del salotto

1.5 Animazione e produzione video

La fase conclusiva dell'attività di progetto riguarda l'inserimento di alcune animazioni, come l'apertura della porta, il movimento dell'orologio e il percorso della camera, per la produzione di un video che dia in pochi minuti un'idea complessiva dell'interno dell'appartamento.

Mentre le animazioni dell'orologio e della porta sono semplicemente delle rotazioni di alcune componenti di tali *mesh* intorno ad un certo asse di riferimento, la realizzazione dei movimenti della camera sulla scena 3D è decisamente più complessa e necessita come fase preliminare la definizione della sceneggiatura del video, prodotto in seguito. Bisogna, quindi, determinare chiaramente l'obiettivo del video, le schermate della scena 3D da inquadrare, il tempo di ciascuna ripresa e la sequenza di riprese da inserire nel video.

(a) (b)

Figura 1.14: Confronto tra fotografia (a) e *rendering* (b) della stanza da letto

(a) (b)

Figura 1.15: Confronto tra fotografia (a) e *rendering* (b) del bagno

Lo scopo del filmato è quello di mostrare l'intero appartamento seguendo un percorso che inizia dalla cucina, prosegue nella stanza da letto e termina nel bagno. In ogni ambiente si è scelto di effettuare dapprima una panoramica generale, quindi alcune riprese di alcuni oggetti particolari.

Per poter definire le inquadrature della scena e i relativi tempi di ripresa, sono stati realizzati alcuni *storyboard* (Fig. 1.16), nei quali per ogni scena si è definito in forma schematica il movimento della camera e il tempo in secondi della ripresa. La sceneggiatura così realizzata è la seguente:

- Inquadrando la porta d'ingresso, la camera si avvicina alla maniglia;
- La chiave viene ruotata un certo numero di volte, la maniglia viene abbassata e la porta si apre;

- La camera entra nell'appartamento, mostrando una panoramica della cucina; si riprendono alcuni dettagli della cucina, come il lavandino, il forno e il frigorifero;
- La camera si sposta verso il salotto e esegue una panoramica del mobile su cui poggia il televisore;
- Si inquadrano alcuni oggetti particolari, come il vaso e l'angelo;
- Si esegue una panoramica della libreria e qualche scorrimento sugli scaffali;
- La camera ruota su se stessa, fino ad inquadrare il divano e i cuscini, quindi ruota intorno al peluche;
- La camera ora si sposta nella stanza da letto. Vengono ripresi gli oggetti sul comodino, il letto e i quadri;
- Dalla stanza da letto, si attraversa la cucina per raggiungere il bagno;
- La camera esegue due panoramiche a 180 gradi, per inquadrare tutti gli oggetti dell'ambiente (specchio, caldaia, lavatrice e sanitari).

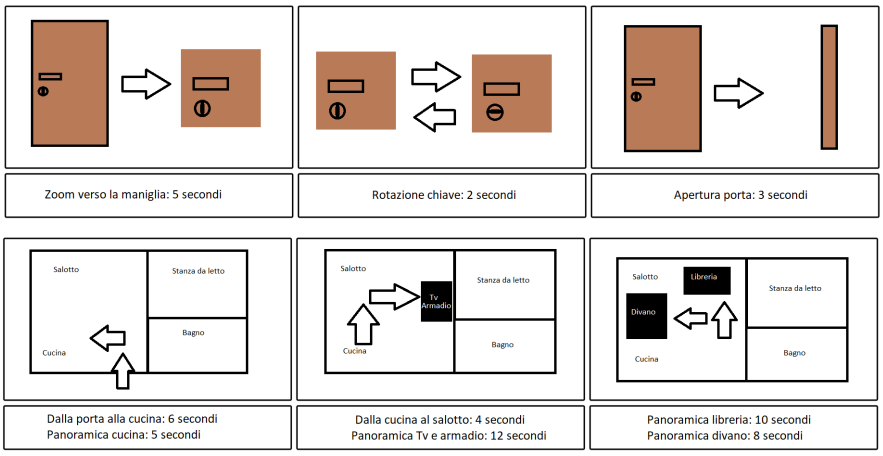


Figura 1.16: Alcuni *storyboard* realizzati

Sulla base della sceneggiatura realizzata, si è definito il percorso della camera all'interno dell'appartamento, producendo un'animazione di 3134 *frame*. Questi ultimi sono stati quindi renderizzati uno alla volta per la produzione del video finale, la cui durata è di circa 2 minuti. Durante il montaggio del video, sono stati inseriti i *frame* prodotti, insieme ad alcuni effetti di transizione e a una musica di sottofondo.

Capitolo 2

Software open source utilizzati

In questo capitolo sono descritti gli strumenti software *open source* utilizzati nel corso di questo lavoro, ossia *Regard3D*, *MeshLab* e *Blender*. Per ognuno di essi è data una breve introduzione sul software riguardo i principali vantaggi nel loro utilizzo e le motivazioni che hanno portato alla scelta di tali software piuttosto che altri concorrenti. Quindi vengono descritte le loro interfacce d'uso e infine l'elenco di attività svolte.

2.1 Regard3D

2.1.1 Introduzione

Regard3D [24] è un software gratuito e *open source* per la fotogrammetria. Esso, quindi, è in grado di realizzare un oggetto 3D a partire da un insieme di fotografie che ritraggono l'oggetto da diversi punti di vista. Tra i tanti software orientati alla *Structure From Motion* (*Meshroom*, *Agisoft Photoscan*, *CloudCompare*, *Zephyr3D* e altri), *Regard3D* è l'unico programma gratuito e open source che non richiede necessariamente un'architettura hardware CUDA (*Compute Unified Device Architecture*) e pertanto una scheda grafica NVIDIA, permettendo di eseguire la computazione direttamente sulla CPU. *Regard3D* può pertanto essere utilizzato su qualsiasi architettura, senza requisiti particolari sulla scheda grafica e per questo motivo è il software scelto

per la produzione di alcuni modelli 3D da inserire nella scena. *MeshRoom*, il miglior software *open source* per la fotogrammetria, ha infatti il vincolo di richiedere CUDA nella fase di densificazione.

Un altro vantaggio di *Regard3D* è l'assenza di un limite massimo di immagini da utilizzare per un singolo progetto; ciò permette di costruire *dataset* di immagini arbitrariamente grandi dai quali estrarre i propri modelli. Al contrario, *Zephyr3D*, un altro software per la fotogrammetria, non permette di utilizzare più di 50 immagini per una singola ricostruzione nella sua versione gratuita, rendendolo inadatto per la generazione di oggetti grandi che possono richiedere molte immagini per una sua completa copertura. In sintesi, i vantaggi principali di *Regard3D* rispetto agli altri software concorrenti sono i seguenti:

- Gratuito e *open source*;
- Eseguibile su piattaforme Windows, OS X e Linux;
- Non richiede necessariamente una scheda grafica NVIDIA;
- Nessuna limitazione sul numero massimo di fotografie per un singolo modello;
- Design semplice e intuitivo;
- Gestione gerarchica delle operazioni che facilita l'organizzazione del lavoro.

In figura 2.1 è mostrata l'interfaccia del programma. Il centro della schermata è occupato dalla *viewport* in cui viene mostrata la ricostruzione effettuata. Il pannello a destra permette di definire i parametri di *rendering*, tra cui la modalità di *shading* (*Smooth* o *Flat*) o la dimensione del singolo punto (*keypoint*) dell'oggetto. A sinistra, invece, è mostrata la vista gerarchica del progetto (*project view*), con la quale è possibile confrontare più esecuzioni di una stessa operazione con diversi parametri di configurazione. Un progetto di *Regard3D* è organizzato nel modo seguente:

- *Project*: nome del progetto;
- *Picture set*: insieme di fotografie da utilizzare per la ricostruzione;
- *Matches*: *feature* estratte dalle fotografie e abbinate tra coppie di immagini;
- *Triangulation*: nuvola di punti sparsa ottenuta dalla triangolazione delle *feature* 2D precedentemente calcolate;
- *Densification*: nuvola di punti densa ottenuta dalla nuvola sparsa dello step precedente;
- *Surface*: superficie calcolata a partire dalla nuvola di punti densa ottenuta.

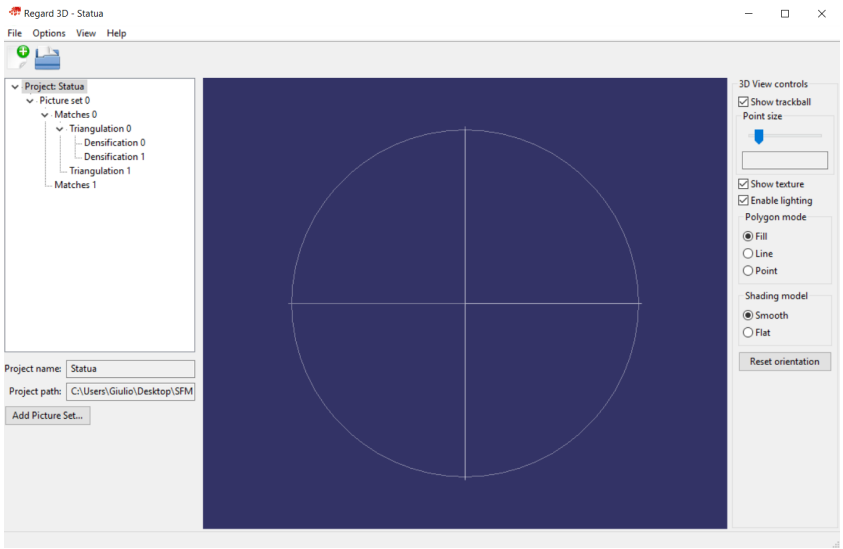


Figura 2.1: Interfaccia di *Regard3D*

Per lo scopo di questo lavoro, le superfici degli oggetti ricostruiti non sono state realizzate in *Regard3D* ma in un altro software *open source*, *MeshLab*, il quale, a differenza di *Regard3D*, consente di modificare la nuvola di punti in modo da eliminare gli *outlier* e lo sfondo prima di produrre una superficie. Inoltre, *MeshLab* mette a disposizione diversi algoritmi di decimazione

e semplificazione che permettono di ridurre il numero di punti della *mesh*, senza alterarne la qualità. Pertanto *Regard3D* è stato utilizzato fino alla fase di *Densification*, cioè fino alla produzione delle nuvole di punti dense, le quali sono state quindi esportate in *MeshLab* per l'elaborazione finale.

Di seguito è descritta la procedura completa per la produzione di una nuvola di punti 3D densa tramite *Regard3D*.

2.1.2 Creazione di un progetto e inserimento di un picture set

La prima operazione da eseguire è la creazione di un nuovo progetto. Cliccando sul pulsante "*Create new project...*" viene visualizzata la finestra mostrata in figura 2.2, nella quale si può definire il percorso (*path*) del progetto e un nome identificativo, diverso dai nomi degli altri progetti esistenti. Cliccando su "*OK*", il progetto viene salvato in un file con estensione *.r3d* ed inserito nella *project view*.

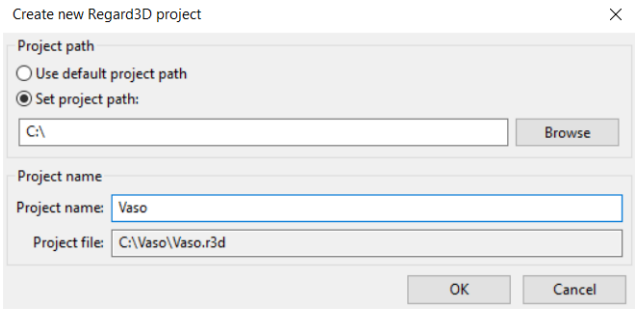


Figura 2.2: Creazione di un nuovo progetto

A questo punto, bisogna inserire l'insieme di fotografie che ritraggono l'oggetto da ricostruire, detto *picture set*. Selezionando la voce del progetto dalla *project view* e cliccando su "*Add Picture Set...*", si apre la maschera per l'inserimento delle immagini. Per caricare il *picture set*, si può cliccare su "*Add files...*" scegliendo quindi le fotografie dal proprio *file system*.

Caricate le immagini, se nella tabella "*Image list*" le colonne "*Focal length*" e "*Sensor width*" mostrano i valori "*N/A*" (*Not Available*), allora la

camera utilizzata non è inserita all'interno del database di *Regard3D*. Questi valori sono fondamentali nella fase di triangolazione, la quale sfrutta i parametri della camera per una corretta collocazione tridimensionale delle *feature* estratte. Si richiede, pertanto, di individuare i valori della lunghezza focale e la dimensione del sensore della camera utilizzata e di inserire manualmente tali valori nel programma. Ad esempio, un sito dedicato alla fotografia digitale, in cui è possibile visualizzare le schede tecniche di varie macchine fotografiche, è *DPReview* (*Digital Photography Review*)[1]. Ottenuti questi due parametri, è possibile definire il valore in millimetri della lunghezza focale direttamente dalla tabella "*Image list*", selezionando un'immagine con il tasto destro e cliccando su "*Set focal length...*". Per il campo "*Sensor width*" bisogna invece aprire il file "*sensor_database_csv*", il cui percorso in Windows è $C:\backslash Users\backslash < Nomeutente > \backslash AppData\backslash Local\backslash Regard3D$, e inserire la marca, il modello e la dimensione del sensore della camera, mantenendo l'ordine alfabetico.

Nel caso in cui la camera è invece nota, questi campi sono popolati dai dati estratti dai metadati *EXIF* delle fotografie.

In figura 2.3 è mostrata un esempio di schermata per l'aggiunta di un *picture set*, dove il pannello a sinistra contiene l'elenco delle fotografie mentre a destra è mostrata una *preview* dell'immagine selezionata. È possibile dare un nome (opzionale) al *picture set* nel campo "*Picture set name*", quindi si può chiudere la schermata cliccando su "*OK*".

2.1.3 Estrazione delle feature e calcolo dei matches

Il prossimo step è calcolare i punti chiave (*feature* o *keypoint*) di ciascuna immagine ed eseguire il *feature matching* tra ciascuna coppia di immagini. Selezionato il *picture set* dalla *project view*, si può cliccare su "*Compute matches...*" per visualizzare la finestra di dialogo mostrata in figura 2.4.

Questa finestra permette di definire non solo i parametri per l'estrazione dei *keypoint*, ma anche l'algoritmo di *matching* da utilizzare per abbinare i

[1] https://www.dpreview.com/

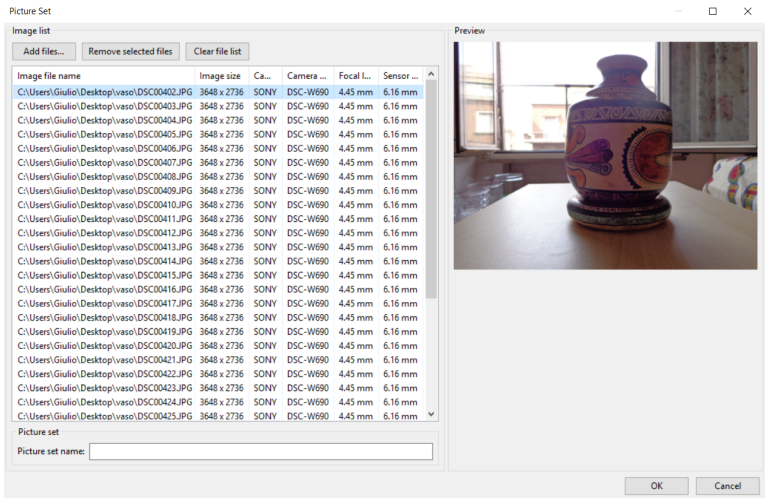


Figura 2.3: Inserimento di un *picture set*

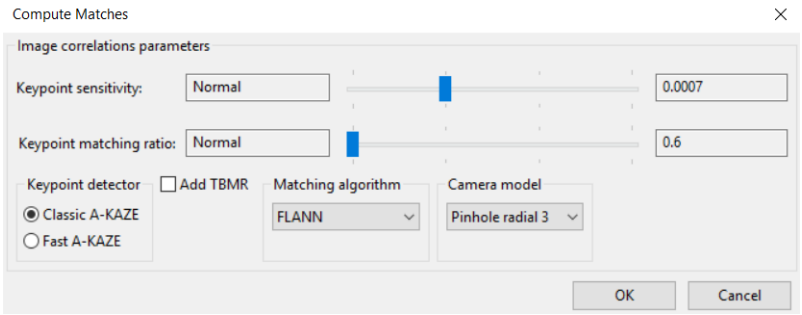


Figura 2.4: Estrazione delle *feature* e calcolo dei *matches*

punti tra ciascuna coppia di immagini. Di seguito sono descritti i parametri di configurazione:

- *Keypoint sensitivity*: definisce la quantità di *keypoint* da individuare. Più alto è il suo valore, maggiore è la sensibilità e, di conseguenza, il numero di *feature* calcolate;
- *Keypoint matching ratio*: definisce la qualità richiesta di ciascun *keypoint* per poter essere utilizzato;
- *Keypoint detector*: permette di scegliere la versione dell'algoritmo *A-KAZE* (vedi 3.1.1 e 3.1.2) da utilizzare per l'estrazione delle *feature*.

Regard3D fornisce l'algoritmo originale (*Classic*) e una sua versione accelerata (*Fast*);

- *Add TBMR*: permette di utilizzare un ulteriore algoritmo di *feature detection*, detto *TBMR* (*Tree-Based Morse Regions*) in coppia con *A-KAZE*;

- *Matching algorithm*: algoritmo di *feature matching*, a scelta tra *FLANN* (vedi 3.3.1), *KGraph* (vedi 3.3.2), *MRPT* (vedi 3.3.3) e una ricerca a forza bruta;

- *Camera model*: modello della camera.

Data la buona qualità delle immagini usate come *picture set*, il campo *Keypoint sensitivity* è stato impostato al valore massimo (*Ultra*) in modo da estrarre il numero minimo di *keypoint* per ogni immagine. Anche per il campo *Keypoint matching ratio* è stato utilizzato il valore massimo (*Ultra*) per massimizzare la qualità dei *keypoint* estratti. Come algoritmo di *feature detection* è stato utilizzato *A-KAZE* nella versione *Classic*, senza utilizzare *TBMR* in coppia. Per il *feature matching* è stata scelta la libreria *FLANN*, che automaticamente individua il miglior algoritmo in base alla tipologia dei dati che riceve in input. Infine il campo *Camera model* è stato lasciato nel suo valore di default ("*Pinhole radial 3*").

Cliccando su "*OK*", inizia la procedura di ricerca e *matching* dei *keypoint*, che in breve consiste nelle seguenti fasi:

- Estrazione dei *keypoint* con *A-KAZE*;

- Descrizione dei *keypoint* con il formalismo *LIOP* (vedi 3.2.1);

- *Feature matching* utilizzando l'algoritmo migliore scelto dalla libreria *FLANN*;

- *Feature tracking* sfruttando l'algoritmo *Union-Find* (vedi 3.4.1).

Terminata la computazione, è possibile vedere gli abbinamenti tra i *keypoint* calcolati selezionando la voce "*Matches*" dalla *project view* e cliccando su "*Show matching result...*" In figura 2.5 è mostrata una coppia di immagini di un *picture set* e i loro *keypoint* abbinati.

Figura 2.5: *Feature* abbinate tra due immagini

2.1.4 Produzione di una nuvola di punti 3D sparsa

Il passaggio dalle immagini 2D ad un modello 3D avviene tramite la procedura di triangolazione (*triangulation*) o di *Structure From Motion* (*SFM*), che consente di generare una nuvola di punti 3D sparsa grazie ai *keypoint* precedentemente estratti e abbinati. A partire dai *matches* di tutte le coppie di immagini, si calcolano le posizioni 3D e gli orientamenti della camera (quindi il punto di vista e l'angolo dal quale ogni foto è stata scattata) e la posizione 3D di ciascuna traccia. Per iniziare la fase di triangolazione, si deve dapprima selezionare la voce "*Matches*" nella *project view*, quindi cliccare su "*Triangulation...*" per visualizzare la schermata mostrata in figura 2.6.

Regard3D offre due possibilità per eseguire la fase di *Structure From Motion*:

- *SFM incrementale* (vedi 3.5.1): partendo da una coppia di immagini iniziale, si aggiunge iterativamente un'immagine alla volta per determinare la singola posizione della camera e delle tracce;

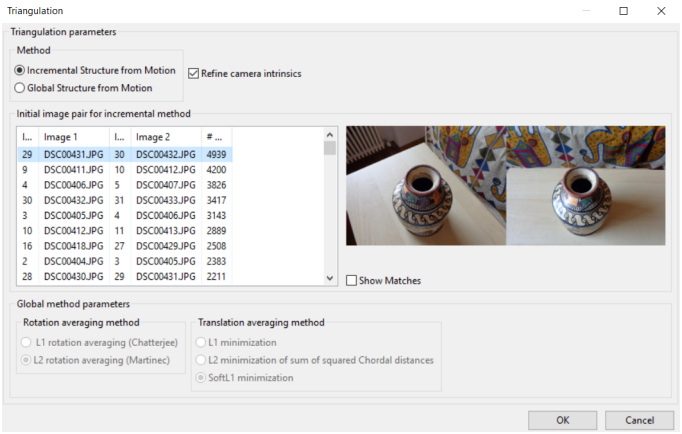


Figura 2.6: Finestra per la triangolazione

- *SFM globale* (vedi 3.5.2): si utilizzano contemporaneamente tutte le immagini per calcolare le posizioni della camera e dei *keypoint* di tutte le fotografie in un unico passo.

Mentre l'algoritmo incrementale è sempre disponibile, la controparte globale è abilitata solo se la camera utilizzata per le fotografie è inserita nel database di *Regard3D* o, più precisamente, se per ogni immagine sono stati definiti i valori *Sensor width* e *Focal length* nella fase di creazione del *picture set*. Gli algoritmi globali implementati in *Regard3D*, infatti, sono in grado di ricostruire oggetti a partire da fotografie scattate da una stessa camera e con gli stessi parametri ottici. La ricostruzione incrementale può, invece, utilizzare immagini provenienti da camere diverse e anche fotografie con i parametri *Sensor width* e *Focal length* non definiti.

D'altra parte, l'approccio globale è nettamente più veloce di quello incrementale, ma richiede molti abbinamenti per una buona ricostruzione e può anche fallire nel caso in cui i *matches* non sono in numero sufficiente. Per utilizzare l'algoritmo incrementale bisogna, tuttavia, tener presente che la sua performance dipende fortemente dalla coppia di immagini iniziale; è pertanto opportuno scegliere inizialmente le due immagini nelle quali è stato individuato il maggior numero di corrispondenze tra le loro *feature*.

Tutti i modelli prodotti in questo lavoro sono stati realizzati sfruttando l'algoritmo di triangolazione incrementale, in quanto, pur essendo più lento di quello globale, è in grado di produrre nuvole sparse di buona qualità, anche utilizzando *picture set* di dimensioni esigue. Selezionato l'algoritmo e cliccato su "*OK*", inizia la fase di triangolazione al termine della quale una finestra di dialogo mostra il numero di camere correttamente calibrate. Nel caso in cui alcune camere non siano state correttamente posizionate, si consiglia di rieseguire la triangolazione iniziando da un'altra coppia di partenza. In figura 2.7 è mostrato un esempio di nuvola sparsa ottenuta dall'esecuzione dell'algoritmo di triangolazione incrementale sul *picture set* di un vaso.

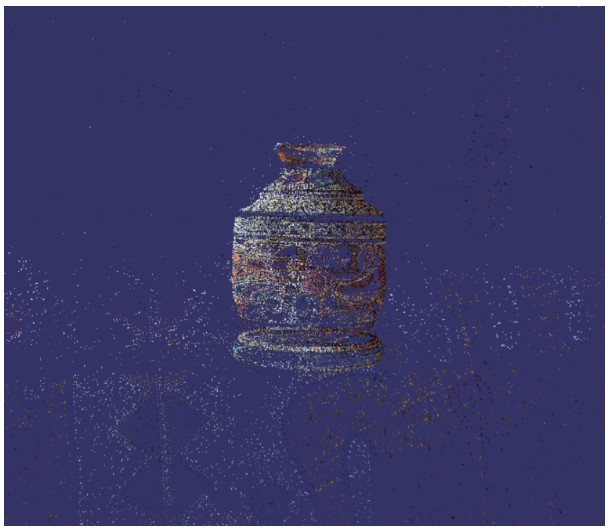
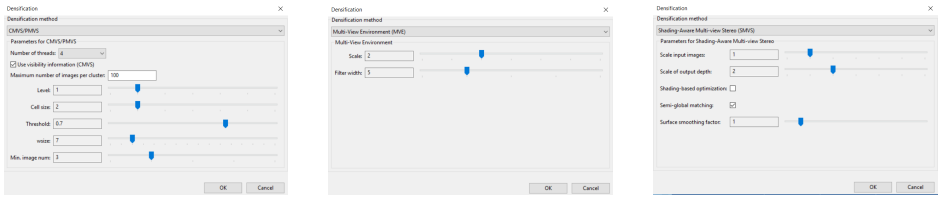


Figura 2.7: Nuvola di punti sparsa

2.1.5 Aumentare la densità della nuvola di punti 3D

Il risultato della fase di triangolazione è una nuvola di punti sparsa, che non consente di effettuare una ricostruzione realistica di un oggetto. Per aumentare la densità di tale nuvola, si deve eseguire la procedura di densificazione (*densification*) che genera una nuvola di punti più densa. Selezionata la voce "*Triangulation*" nella *project view*, si può cliccare su "*Create dense pointcloud...*" per visualizzare la schermata mostrata in figura 2.8 (a).



(a) (b) (c)

Figura 2.8: Schermate per la configurazione di *CMVS/PMVS* (a), *MVE* (b) e *SMVS* (c)

Gli algoritmi di densificazione implementati in *Regard3D* sono i seguenti:

- *CMVS/PMVS* (*Clustering Views for Multi-View Stereo/Patch-based Multi-View Stereo*, vedi 3.6.1);
- *MVE* (*Multi-View Environment*, vedi 3.6.2);
- *SMVS* (*Shading-Aware Multi-View Stereo*).

Di default, l'algoritmo selezionato è *CMVS/PMVS*, utilizzato nel corso di questo lavoro con i suoi valori prestabiliti. Di seguito sono descritti i suoi parametri di configurazione:

- *Number of threads*: numero di *threads* eseguiti durante il processo;
- *Use visibility information (CMVS)*: flag che abilita o meno l'esecuzione dell'algoritmo di *clustering CMVS*, il quale suddivide la fase di densificazione eseguita da *PMVS* in *cluster*. È consigliato abilitare *CMVS* per *picture set* grandi, mentre per modelli medio-piccoli si può anche eseguire *PMVS* da solo;
- *Maximum number of images per cluster*: dimensione massima di ciascun *cluster* creato da *CMVS*;
- *Level*: numero di volte in cui la risoluzione delle immagini viene dimezzata. Aumentando questo valore, il tempo di computazione diminuisce come anche la densità della ricostruzione;

- *Cell size*: dimensione della regione di pixel in cui *PMVS* cerca di ricostruire almeno un *patch*. Più il valore è basso, maggiore è la densità della ricostruzione;
- *Threshold*: soglia che definisce la qualità minima di un *patch* per essere accettato e mantenuto;
- *wsize*: definisce il numero di colori calcolati da ogni immagine, necessari per calcolare le misure di qualità;
- *Min. image num*: numero minimo di immagini in cui un punto deve essere visibile per essere ricostruito.

Gli altri due algoritmi (*MVE* e *SMVS*) non sono stati utilizzati in quanto richiedono un tempo di computazione notevolmente maggiore rispetto a *CMVS/PMVS*. I parametri di *MVE* sono due:

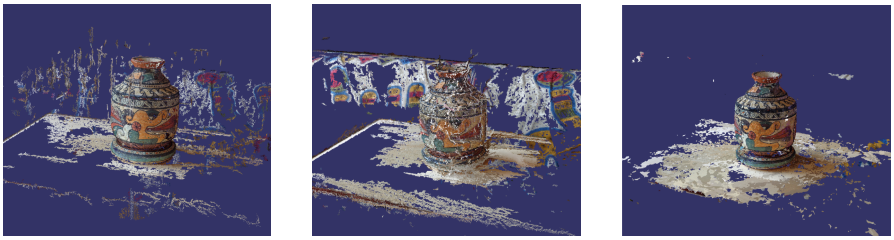
- *Scale*: fattore di scala della ricostruzione. Valori bassi portano a maggiore definizione e a maggior tempo di computazione;
- *Filter width*: dimensione della regione di pixel in cui si calcola il valore *NCC* (*Normalized Cross Correlation*) per il confronto tra due immagini.

Infine, i parametri di *SMVS* sono i seguenti:

- *Scale input images*: fattore di scala di ricostruzione che definisce il valore di dettaglio e, conseguentemente, il tempo di computazione;
- *Scale of output depth*: grado di risoluzione dei singoli *patches* prodotti in output;
- *Shading-based optimization*: flag che abilita o meno l'ottimizzazione *shading-based*;
- *Semi-global matching*: flag che abilita il *matching semi-global*;

- *Surface smoothing factor*: grado di smussatura del modello prodotto in output.

Scelto l'algoritmo, si può far partire la procedura cliccando su "*OK*". Al termine del processo, viene visualizzato il risultato nella *viewport*. In figura 2.9 sono mostrate le nuvole dense prodotte con i tre algoritmi a partire dalla nuvola sparsa della figura 2.7. Come si può notare, la nuvola densa prodotta da *CMVS/PMVS* ha un livello di fedeltà leggermente inferiore a quella di *SMVS*, ma performance nettamente migliori.



(a) t = 16 minuti (b) t = 52 minuti (c) t = 88 minuti

Figura 2.9: Nuvole di punti dense prodotte con CMVS/PMVS (a), MVE (b) e SMVS (c) e tempi di computazione.

La nuvola così ottenuta presenta, tuttavia, molti *keypoint* relativi allo sfondo e non può essere pertanto utilizzata direttamente per la creazione di una superficie. Dato che *Regard3D* non ha a disposizione uno strumento per la rimozione dei *keypoint* estranei all'oggetto, bisogna utilizzare un altro software per effettuare la pulizia della nuvola densa. Il software utilizzato per questo scopo è *MeshLab*, con il quale *Regard3D* è in grado di interfacciarsi esportando il modello come una vera e propria scena di *MeshLab*. Per eseguire questa operazione basta selezionare la voce "*Densification*" dalla *project view*, cliccare su "*Export scene to MeshLab*" e scegliere la cartella in cui salvare il modello.

È possibile anche salvare la nuvola di punti in un file con formato *.ply* (*Stanford Polygon File*) cliccando su "*Export point cloud*", nel caso in cui si voglia utilizzare un software diverso da *MeshLab*. Si consiglia, tuttavia, di

esportare il modello in una scena *MeshLab* in quanto solo in questo modo vengono mantenute le informazioni relative alle camere utilizzate durante la ricostruzione che, come verrà descritto in seguito, sono fondamentali durante la fase di *texturing*. Nel caso in cui, invece, non si intende eseguire il *texturing* del modello a partire dalla fotografie del *picture set*, allora si può procedere con l'esportazione in un file *.ply*.

2.2 MeshLab

2.2.1 Introduzione

MeshLab [2] [25] è un software *open source* per l'elaborazione di *mesh* 3D. Esso dispone di un vasto insieme di strumenti per la modifica, pulizia, riparazione, analisi, resa e conversione di modelli 3D. Esso è particolarmente indicato per gestire *mesh* di grandi dimensioni e non strutturate (*raw data*), che possono derivare da digitalizzazioni 3D (es. scanner 3D o *Structure From Motion*). *MeshLab* è disponibile su varie piattaforme, come Linux, Mac OS X e Windows, ed è in grado di importare/esportare file in vari formati, tra cui PLY, STL, OBJ, 3DS e COLLADA.

La scelta di *MeshLab* rispetto ad altri software è dipesa da due principali motivazioni: da un lato, la sua efficienza nell'elaborazione di *mesh* di grandi dimensioni, dall'altro la capacità di importare le nuvole di punti prodotte con *Regard3D* conservando le informazioni relative alla camera e alle singole fotografie utilizzate per la produzione del modello. Quest'ultima funzionalità è fondamentale per poter eseguire il *texturing* dell'oggetto, generando un'immagine *texture* composta da alcune porzioni delle fotografie.

In figura 2.10 è mostrata l'interfaccia di *MeshLab*: il centro è occupato dalla *viewport*, mentre la parte superiore contiene le varie barre degli strumenti che gestiscono la visualizzazione e l'elaborazione del modello. A destra, invece, è mostrato l'elenco dei *layer* (livelli di disegno) e l'insieme delle fotografie.

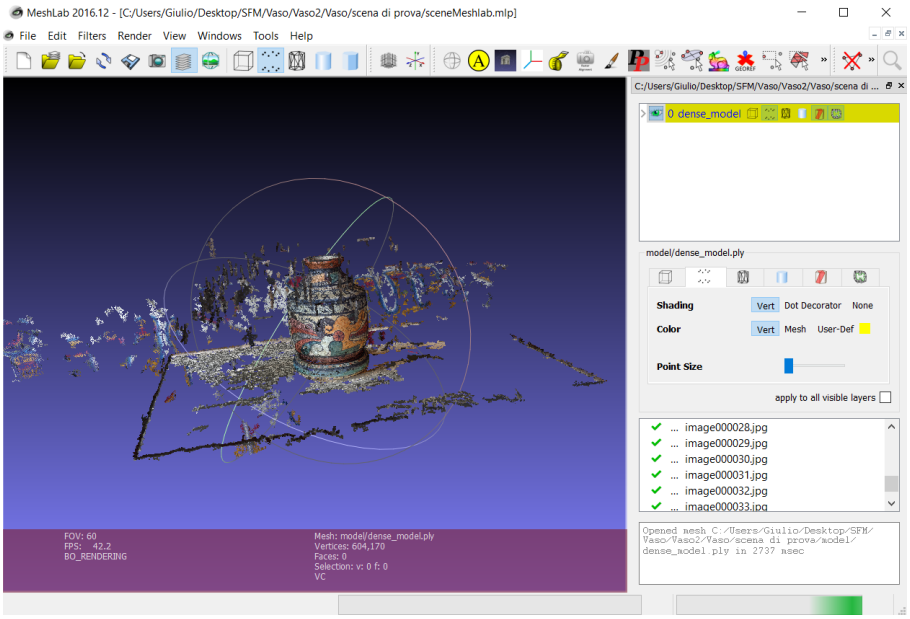


Figura 2.10: Interfaccia di *MeshLab*

MeshLab è un software pieno di funzionalità. In seguito sono descritte solo quelle utilizzate per questo lavoro, ossia la generazione di una superficie, il *texturing* e il campionamento (*sampling*).

2.2.2 Importazione nuvola densa e rimozione punti

Il primo passo è l'importazione della scena contenente la nuvola densa calcolata da *Regard3D*, cliccando su "*Open project*" e aprendo il file "*sceneMeshlab.mlp*" salvato nella cartella scelta durante l'esportazione.

Una volta visualizzata la nuvola nella *viewport* (Fig. 2.11 (a)), è possibile intervenire su di essa per rimuoverne le parti esterne. Il pulsante "*Select vertexes*" permette di selezionare i vertici tracciando dei rettangoli con il mouse e, tenendo premuto il tasto "*Ctrl*", è possibile effettuare selezioni multiple consecutive. Infine, cliccando l'apposito tasto nella barra superiore degli strumenti, i vertici selezionati vengono rimossi. In questo modo si ottiene una nuvola densa pronta per la fase di generazione della superficie (Fig. 2.11 (b)).

(a) (b)

Figura 2.11: Nuvola di punti densa prima (a) e dopo (b) la rimozione dei punti esterni all'oggetto.

2.2.3 Generazione della superficie

Ora si può procedere con la generazione di una superficie, ossia con la creazione della *mesh* vera e propria a partire dai punti della nuvola densa. Gli algoritmi di *surface generation* o *surface reconstruction* sono accessibili dal menu "*Filters* → *Remeshing, Simplification and Reconstruction*". I due algoritmi principali per la produzione di una superficie sono i seguenti:

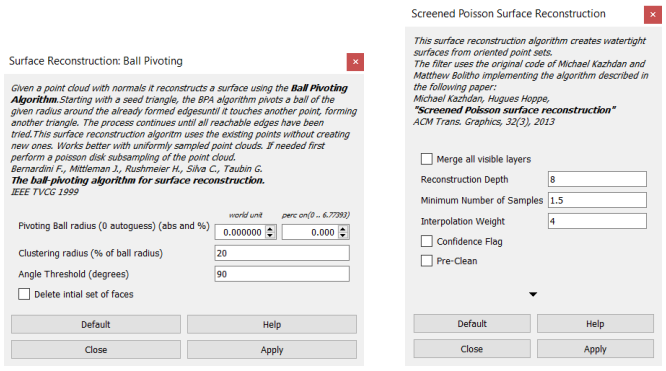
- *Ball Pivoting* (vedi 3.7.1);
- *Screened Poisson Surface Reconstruction* (vedi 3.7.2);

In figura 2.12 sono mostrate le schermate per la configurazione dei due algoritmi. In dettaglio, i parametri di configurazione di *Ball Pivoting* sono:

- *Pivoting Ball radius*: raggio della sfera che viene fatta ruotare sulla nuvola di punti per la creazione di triangoli;
- *Clustering radius*: percentuale del raggio della sfera che indica quanto due vertici devono essere vicini per venire fusi in un unico *cluster*;
- *Angle threshold*: angolo massimo di rotazione per la sfera;
- *Delete initial set of faces*: flag che permette di eliminare le facce della *mesh* e generare la superficie dall'inizio, oppure di utilizzare le facce presenti come punto di partenza per l'algoritmo.

I parametri di configurazione di *Screened Poisson Surface Reconstruction* sono i seguenti:

- *Merge all visible layers*: flag che permette di utilizzare i punti di tutti i *layer* per la generazione della superficie, oppure solo quelli del *layer* selezionato;
- *Reconstruction depth*: profondità d massima dell'albero utilizzato per la ricostruzione. Definisce la risoluzione della griglia tridimensionale (*voxel grid*) che partiziona lo spazio;
- *Minimum Number of Samples*: numero minimo di punti per ogni nodo dell'albero di ricostruzione;
- *Interpolation Weight*: fattore di interpolazione;
- *Confidence Flag*: flag che permette o meno di considerare la qualità come informazione di confidenza;
- *Pre-Clean*: flag che permette o meno di effettuare una pulizia preliminare della nuvola per eliminare i vertici non referenziati o con normali nulle.



(a) (b)

Figura 2.12: Schermate per la configurazione di *Ball Pivoting* (a) e *Screened Poisson Surface Reconstruction* (b)

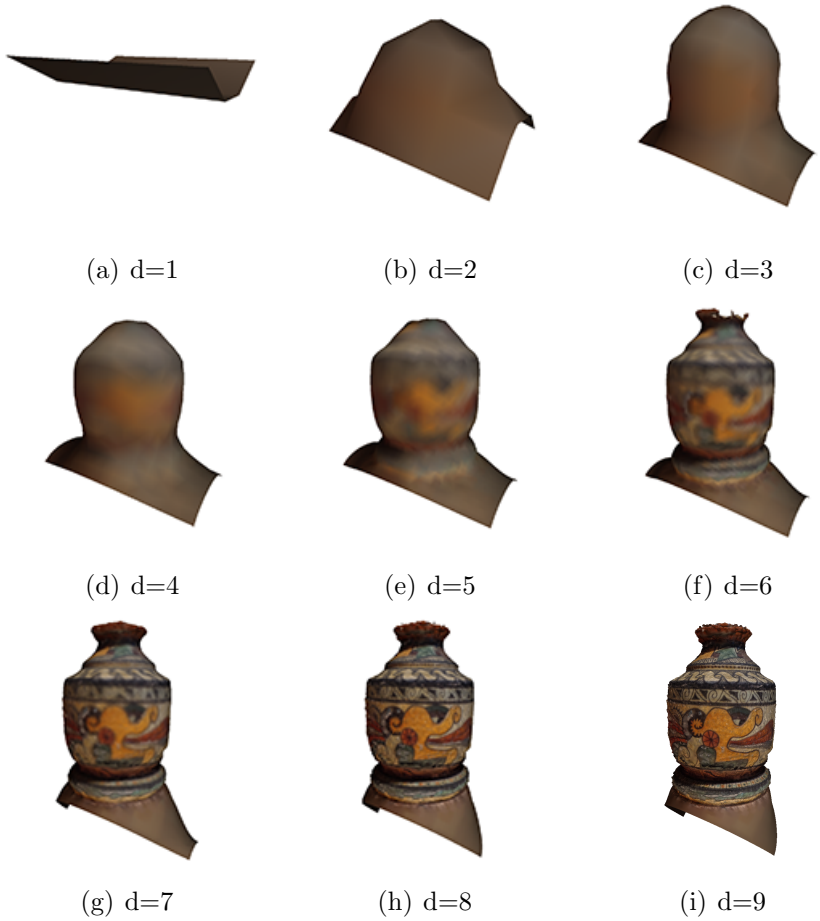
Le superfici degli oggetti ricostruiti in questo lavoro sono state realizzate utilizzando l'algoritmo *Screened Poisson Surface Reconstruction*, il quale non solo è più veloce di *Ball Pivoting* ma porta a modelli di qualità maggiore. Inoltre, il parametro "*Reconstruction depth*" permette di individuare facilmente e in modo intuitivo la risoluzione finale voluta, piuttosto che trovare il miglior raggio della sfera per l'algoritmo *Ball Pivoting*. In figura 2.13, ad esempio, è mostrata la ricostruzione di un oggetto a diverse profondità. Per valori di profondità maggiori di 9, il guadagno in qualità del modello è trascurabile contro un aumento considerevole del numero di vertici. In seguito alla generazione della superficie, può essere richiesta nuovamente una fase di pulizia per rimuovere alcune porzioni errate della *mesh*.

2.2.4 Parametrizzazione UV

Il modello ora mantiene per ogni vertice l'informazione RGB ma non ha una *texture* associata. Se esportassimo il modello in un software di modellazione grafica, come *Blender*, l'informazione del colore non verrebbe mantenuta. Bisogna pertanto generare un file immagine da associare come *texture* alla *mesh* prodotta. Il metodo indicato per questo scopo è sfruttare le fotografie per parametrizzare la *mesh*; in pratica, si creano alcuni *patch* che corrispondono alla proiezione delle porzioni di superficie sull'insieme di immagini registrate, denominate *raster*. Per eseguire la parametrizzazione UV a partire dai *raster*, si può selezionare "*Parametrization + texturing from registered rasters*" dal menu "$Filters \rightarrow Texture$" per visualizzare la schermata in figura 2.14.

È possibile configurare la parametrizzazione UV con i seguenti parametri:

- *Texture size*: dimensione del file immagine da creare;
- *Texture name*: nome del file immagine da creare;
- *Color correction*: flag che abilita o meno la correzione del colore affinché vi sia una transizione *seamless* (cioè senza soluzione di continuità) tra ogni coppia di facce;



(a) d=1 (b) d=2 (c) d=3

(d) d=4 (e) d=5 (f) d=6

(g) d=7 (h) d=8 (i) d=9

Figura 2.13: Creazione di una superficie a diversi valori di profondità.

- *Color correction filter*: tipologia del filtro per la correzione del colore;
- *Use distance weight*: flag che permette di tener conto o meno della distanza del *patch* della camera in ciascuna fotografia;
- *Use image border width*: flag che permette di tener conto o meno della distanza del *patch* dai bordi dell'immagine;
- *Use image alpha weigth*: flag che abilita o meno il canale *alpha* dell'immagine;

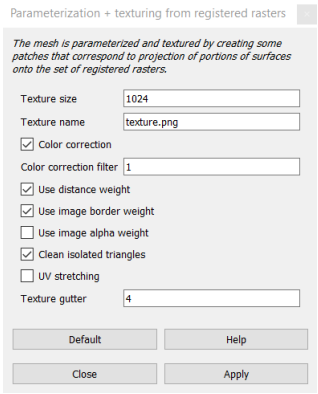


Figura 2.14: Schermata per la parametrizzazione UV

- *Clean isolated triangles*: flag che permette di aggregare o meno i *patch* composti da un solo triangolo con i *patch* adiacenti;
- *UV stretching*: flag che abilita o meno l'estensione delle coordinate UV nell'intervallo [0,1] in entrambe le direzioni;
- *Texture gutter*: confine in pixel aggiunto in ciascun *patch*.

In figura 2.15 è mostrata la *mesh* originale e la sua versione con parametrizzazione UV, insieme all'immagine *texture* generata dalle fotografie.

(a) (b) (c)

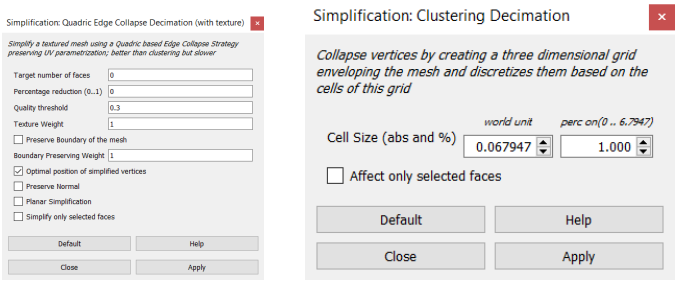
Figura 2.15: *Mesh* (a), *mesh* con parametrizzazione UV (b) e *texture* generata (c)

2.2.5 Semplificazione della mesh

L'ultima fase che precede l'inserimento dell'oggetto sulla scena è la sua semplificazione, ossia un suo campionamento a bassa risoluzione per ridurne la complessità in termini di numero di vertici complessivo. Come gli algoritmi di *surface generation*, gli algoritmi di decimazione sono presenti nel menu "*Filters* → *Remeshing, Simplification and Reconstruction*". I due principali algoritmi di semplificazione disponibili in *MeshLab* sono i seguenti:

- *Clustering Decimation* (vedi 3.8.1);
- *Quadric Edge Collapse Decimation* (vedi 3.8.2).

In figura 2.16 sono mostrate le finestre per la configurazione dei due algoritmi:



(a) (b)

Figura 2.16: Finestre per la configurazione di *Quadric Edge Collapse Decimation* (a) e di *Clustering Decimation* (b)

L'algoritmo "*Clustering Decimation*" richiede solo due parametri: *Cell size*, ossia la dimensione di ciascuna cella della griglia di *clustering*, e il flag "*Affect only selected faces*" con cui l'utente può scegliere se effettuare la decimazione di tutta la *mesh* o solo delle facce selezionate. L'algoritmo *Quadric Edge Collapse Decimation*, invece, ha i seguenti parametri di configurazione:

- *Target number of faces*: numero di facce della *mesh* ridotta;
- *Percentage reduction (0..1)*: percentuale di riduzione della *mesh*;

- *Quality threshold*: soglia di qualità minima per ciascuna faccia;
- *Texture weight*: peso delle coordinate UV di ciascun vertice;
- *Preserve boundary of the mesh*: flag che permette di conservare o meno i confini della *mesh*;
- *Boundary Preserving Weight*: valore che definisce l'importanza dei confini della *mesh*;
- *Optimal position of simplified vertices*: flag con cui l'utente può scegliere se il vertice collassato debba essere posizionato nella posizione ottimale che minimizza il *quadric error*, oppure debba essere fuso con uno dei due vertici del lato;
- *Preserve Normal*: flag che permette di preservare o meno l'orientamento delle facce;
- *Planar Simplification*: flag che abilita o meno la semplificazione delle porzioni planari della *mesh*;
- *Simplify only selected faces*: flag con cui l'utente può scegliere se effettuare la decimazione di tutta la *mesh* o solo delle facce selezionate.

Tra i due algoritmi, quello utilizzato in questo lavoro è il secondo, grazie alla sua capacità di mantenere inalterata la topologia della *mesh* e di conservare la parametrizzazione UV. La decimazione basata su *clustering*, pur essendo notevolmente più veloce dell'altra, non ha queste caratteristiche. Come si può vedere in figura 2.17, l'algoritmo *Quadric Edge Collapse Decimation* porta a risultati decisamente migliori rispetto all'algoritmo *Clustering Decimation*, richiedendo tuttavia un tempo di esecuzione maggiore.

L'utilizzo di *MeshLab* si conclude con l'esportazione del modello in formato OBJ in modo da poter essere inserito come geometria nella scena 3D in realizzazione. Per esportare la *mesh*, basta selezionare la voce corrispondente nella barra dei *layer*, quindi dal menu "*File* → *Export Mesh As..*" si

(a) 600.000 vertici (b) 60.000 vertici (c) 420.000 vertici

Figura 2.17: *Mesh* (a), *mesh* ridotta con *Quadric Edge Collapse Decimation* (b) e con *Clustering Decimation* (c). Si può notare che la *mesh* in (b) preserva la qualità, la topologia e la parametrizzazione UV, a differenza della *mesh* in (c).

può indicare il formato di output e il nome del file. La *mesh* è ora pronta per essere inserita in *Blender*, il software di computer grafica utilizzato per la modellazione della scena.

2.3 Blender

2.3.1 Introduzione

Blender [26] è un software *open source* per la modellazione e animazione 3D. Oltre alla vasta quantità di funzionalità messe a disposizione, *Blender* è anche in grado di supportare file in diversi formati, come OBJ, 3D STUDIO e FBX, rendendo il programma adatto per l'elaborazione centralizzata di modelli 3D prodotti con tecniche differenti e, quindi, salvati in formati diversi. Quest'applicazione è stata scelta e utilizzata per tutta l'attività lavorativa, in quanto permette di gestire l'intero processo produttivo, dall'importazione di modelli realizzati tramite fotogrammetria alla modellazione di altre geometrie, fino alla produzione di un video a partire da un insieme di *frame* renderizzati.

L'interfaccia di *Blender* è composta da una serie di finestre, dette anche *Editor*. In figura 2.18 è mostrata l'interfaccia di default, ossia l'insieme di finestre che *Blender* mostra al suo avvio:

- *3D View*: area in cui viene mostrata la scena 3D;
- *Outline*: elenco gerarchicamente organizzato degli oggetti presenti sulla scena;
- *Properties*: insieme di parametri per modificare le proprietà dell'oggetto selezionato, della scena attuale o dell'intero progetto;
- *Timeline*: flusso temporale con cui definire il comportamento dinamico degli oggetti nel tempo.

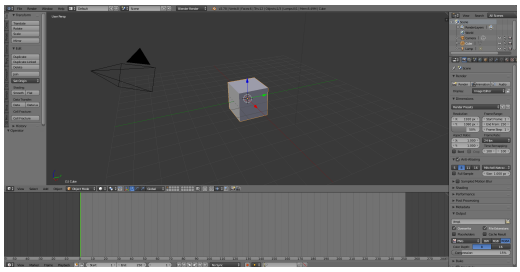


Figura 2.18: Schermata di default di *Blender*

E' possibile modificare il *layout* del programma, cioè la disposizione delle finestre, come anche la tipologia delle finestre stesse. Cliccando sul pulsante di editor di una finestra, che si trova nella sua intestazione o *header*, è possibile scegliere quale finestra visualizzare. Altri *Editor* non presenti nell'interfaccia di default, ma ampiamente utilizzati nel corso di questo progetto, sono i seguenti:

- *Node Editor*: area per la definizione dei materiali, dove ciascun materiale è implementato tramite una rete di nodi;
- *UV/Image Editor*: finestra per il *texture mapping* e la definizione delle *UV map*;

- *Graph Editor*: editor per la gestione delle animazioni tramite F-curve;

- *Video Sequence Editor*: editor di video che consente di generare un filmato a partire da un insieme di immagini.

2.3.2 Creazione progetto

Per cominciare, bisogna creare un nuovo progetto di *Blender* in cui verranno caricate le *mesh* prodotte con i due software precedentemente descritti. All'apertura di *Blender*, viene creato un progetto base di default, contenente un cubo, una camera e una fonte luminosa che possono essere eliminati in quanto sia la camera che le sorgenti luminose verranno inserite in seguito.

A questo punto, per importare i modelli realizzati con *Regard3D* e *MeshLab* si può cliccare su *File* → *Import* → *Wavefront (.obj)* e selezionare il file OBJ di ciascun modello da importare.

In figura 2.19 sono mostrati quattro oggetti realizzati tramite *SFM* inseriti in una scena di *Blender*.

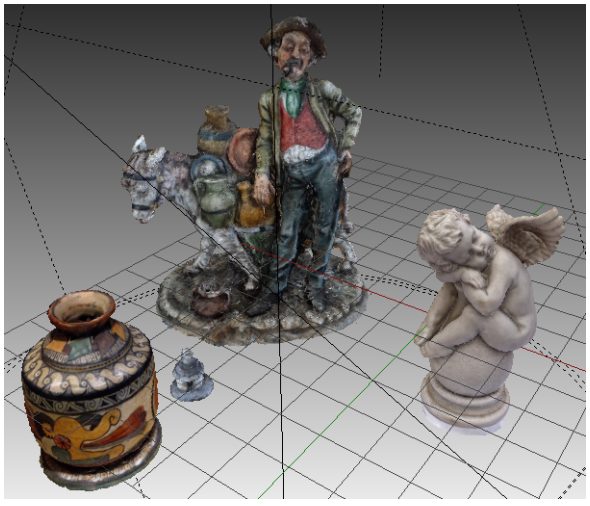


Figura 2.19: I quattro modelli realizzati tramite *SFM* importati in *Blender*

2.3.3 Modellazione

3D View

La fase di modellazione avviene nella *3D View*, la finestra principale di *Blender* che contiene la scena 3D modellata. In Figura 2.20 è mostrata la *3D View* della schermata di default. La navigazione all'interno di tale finestra avviene tramite la rotella del mouse: tenendo premuto la rotella, si può ruotare la scena; muovendola in avanti o indietro si può effettuare lo zoom in o lo zoom out; infine tenendo premuti contemporaneamente il tasto "Shift" e la rotella e muovendo il mouse si può effettuare una traslazione.

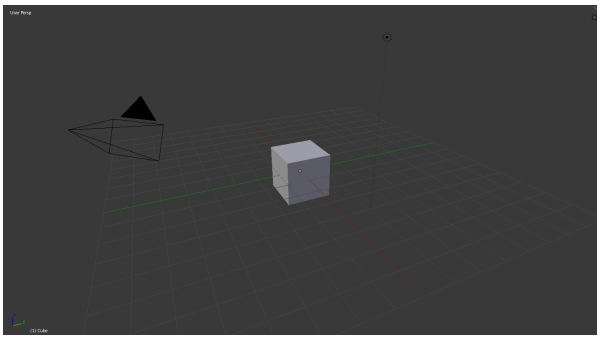


Figura 2.20: *3D View*

Blender mette a disposizione sei diverse tecniche di *viewport shading*, ossia le modalità in cui tutti gli oggetti sulla scena vengono mostrati nella *3D View*. Ogni oggetto sulla scena può essere rappresentato in sei diverse modalità (vedi 3.14), ordinate per complessità di visualizzazione crescente:

- *Bounding Box*: mostra solo i contorni dei parallelepipedi che approssimano la forma degli oggetti;
- *Wireframe*: ogni oggetto viene rappresentato mostrandone i vertici e i lati;
- *Solid*: visualizza gli oggetti, utilizzando un colore comune per le facce e un'illuminazione di default;

- *Texture*: mostra le *mesh* con le corrispondenti *texture* applicate;
- *Material*: visualizza gli oggetti con i loro materiali assegnati;
- *Rendered*: utilizza il motore di *rendering* per rappresentare l'oggetto con le luci della scena.

In figura 2.21 è mostrato ad esempio un oggetto visualizzato nelle sei diverse modalità.

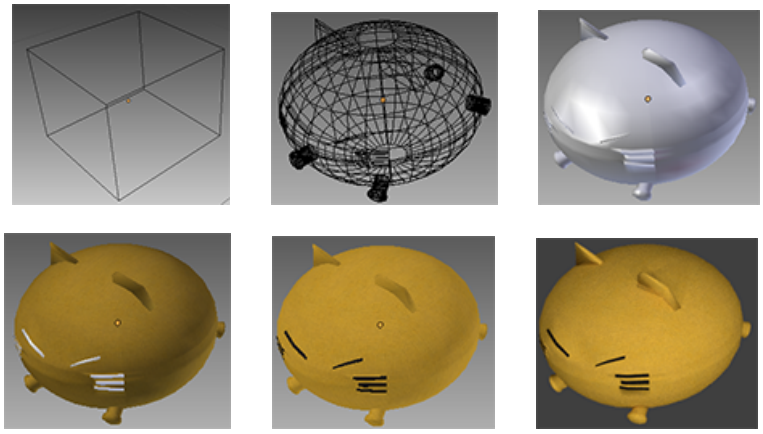
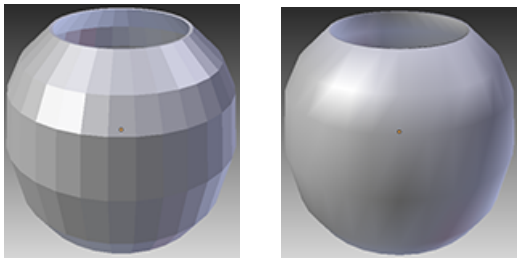


Figura 2.21: *Mesh* mostrata nelle sei modalità di *viewport shading*

Oltre alle tecniche di *viewport shading*, globali a tutti gli oggetti, vi sono inoltre due modalità di *object shading*, che invece permettono di gestire il comportamento del singolo oggetto rispetto alle sorgenti luminose:

- *Flat*: assegna un colore a ciascuna faccia della *mesh* calcolandone la luminosità in base alla sua normale. Nel caso di superfici curve, può portare a *mesh* con spigoli evidenti;
- *Smooth*: assegna un colore a ciascun vertice della *mesh*. Il colore della faccia deriva dall'interpolazione delle normali dei suoi vertici.

In figura 2.22 è mostrato, ad esempio, un oggetto visualizzato nelle due tecniche suddette.



(a) (b)

Figura 2.22: *Flat shading* (a) e *Smooth shading* (b)

All'interno della *3D View* è possibile aggiungere un nuovo oggetto sulla scena premendo la combinazione di tasti "Shift+A" e selezionando quindi la tipologia di oggetto voluta. Il nuovo oggetto sarà creato nella posizione attuale del *3D cursor*, che rappresenta il punto di riferimento di *Blender*. Per definire la nuova posizione del *3D cursor* basta cliccare con il tasto sinistro del mouse su un punto della *3D View*. Si può eliminare un oggetto selezionandolo con il tasto destro, premendo "X" e quindi confermando l'eliminazione. Per centrare, invece, la scena su un particolare oggetto basta selezionare l'oggetto e digitare il tasto "." del tastierino numerico. *Blender* mette a disposizione diversi tipi di oggetti, oltre alle *mesh*, per la realizzazione di un progetto 3D:

- *Curve*: curve matematiche con cui definire percorsi, sui quali ad esempio far muovere una telecamera o moltiplicare in successione una *mesh*;
- *Camera*: osservatore della scena, il cui tronco di vista costituisce l'area da renderizzare;
- *Lamp*: fonte di luce che illumina gli oggetti sulla scena.

Le *mesh* possono essere manipolate principalmente in due modalità. In *Object Mode* non si può modificare la topologia della *mesh*, ma è possibile applicarvi le trasformazioni geometriche di traslazione, rotazione e scala. In *Edit Mode*, invece, si può modificare la topologia della *mesh* e, quindi, aggiungere/rimuovere vertici, spigoli e facce. È possibile passare da una modalità all'altra premendo il tasto "TAB", previa selezione di una *mesh*.

Il tasto "N" permette di aprire o nascondere la finestra delle informazioni dell'oggetto selezionato, tra cui le coordinate x, y e z. Le trasformazioni geometriche fondamentali (traslazione, rotazione e scala) possono essere effettuate all'interno di tale pannello, oppure tramite le seguenti combinazioni di tasti:

- G (*Grab*): traslazione. Le combinazioni "G+X", "G+Y", "G+Z" permette di limitare la traslazione ad un solo asse;
- R (*Rotate*): rotazione. Le combinazioni "R+X", "R+Y", "R+Z" permette di limitare la rotazione ad un solo asse;
- S (*Scale*): scala uniforme. Le combinazioni "S+X", "S+Y", "S+Z" permette di limitare la scala ad un solo asse.

Una combinazione di tasti precedente seguita da un numero permette di definire l'intensità della trasformazione. Ad esempio, "G+X+2" trasla la *mesh* sull'asse X di due unità.

Mura, porte e finestre

La modellazione 3D di un ambiente reale richiede necessariamente la definizione preliminare del sistema di unità di misura da utilizzare per la misurazione delle lunghezze. Al fine di rispettare le proporzioni e le distanze tra i vari oggetti della scena, si è impostato in *Blender* il sistema metrico (*Metric*), in modo tale che le operazioni di creazione, spostamento, scala e modifica di una *mesh* possano essere eseguite utilizzando i centimetri e metri come unità di misura.

La fase di modellazione è iniziata con il disegno delle mura e del pavimento in modo da ripartire immediatamente lo spazio 3D in tre parti, corrispondenti alle tre stanze dell'appartamento: cucina/salotto, stanza da letto e bagno (Fig. 2.23). Dopo aver misurato ogni spigolo delle tre stanze, si è provveduto a collocare i piani (*plane*) delle mura, ortogonali tra di loro e ortogonali al piano del pavimento. Ad ogni piano delle mura interne della

casa è stata dato un spessore di 10 centimetri, tramite estrusione lungo la direzione parallela alla sua normale. L'estrusione in *Blender* avviene tramite il tasto "E", eventualmente seguito da "X", "Y"o "Z" se si vuole eseguire l'estrusione lungo un asse diverso da quello ortogonale. Infine, digitando una quantità insieme all'unità di misura (in questo caso, 10cm), si può definire l'intensità della trasformazione.

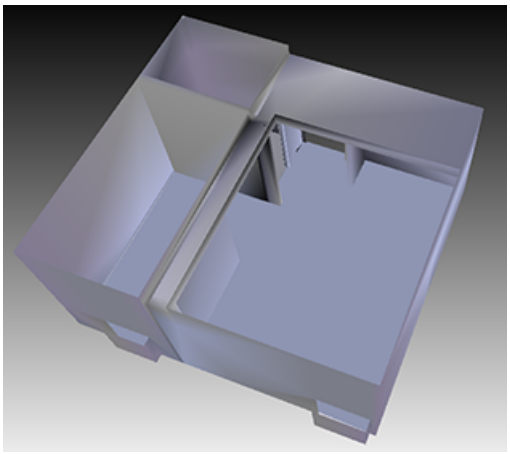


Figura 2.23: Modellazione delle mura

Nella fase successiva si è proceduto all'inserimento delle porte e delle finestre. Mentre le porte sono state modellate con due *mesh* fedeli alle loro controparti reali, per le finestre si è utilizzato l'oggetto "Rail Window" disponibile nell'Add-on *Archimesh*, il quale contiene molteplici oggetti per l'arredamento di interni, come porte, finestre, scale, scaffali e così via.

L'inserimento delle porte e delle finestre ha richiesto l'utilizzo del modificatore *Boolean* per inserire due aperture nelle mura dove collocare le due *mesh*. In particolare, *Boolean* è un modificatore di *Blender* che prende in input due *mesh* e produce in output una nuova *mesh* prodotta dall'applicazione di un'operazione insiemistica (unione, intersezione o differenza) ai due input. L'utilità di questo operatore è la semplificazione del processo di disegno: piuttosto che realizzare una *mesh* completamente dettagliata, è possibile disegnare più componenti distinte e utilizzare un modificatore *Boolean* per generare il modello finale.

Per ogni porta e finestra si è creato un parallelepipedo che ne approssima la dimensione e lo si è collocato in modo da intersecarsi con le mura nel punto in cui dovrà essere collocata la *mesh*. Attivando il modificatore *Boolean* ($Modifiers \rightarrow Add\ Modifier \rightarrow Boolean$) all'edificio e applicando in questo caso l'operazione di differenza con i parallelepipedi creati, si ottiene la creazione delle aperture nelle pareti. Applicando quindi il modificatore ed eliminando i cubi delle finestre non più necessari, si è così creata una casa con finestre e porte senza dover modificare manualmente l'edificio in *Edit Mode* (Fig. 2.24).

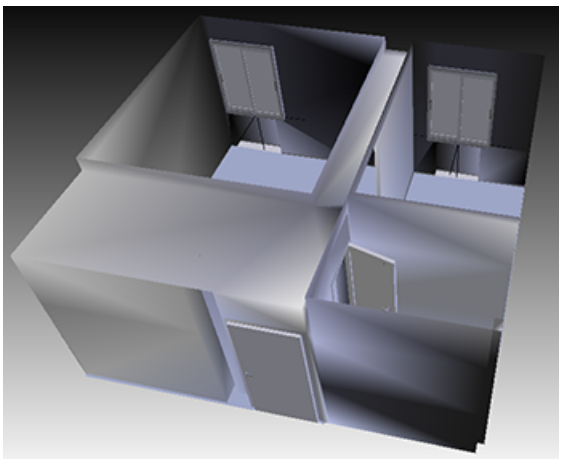
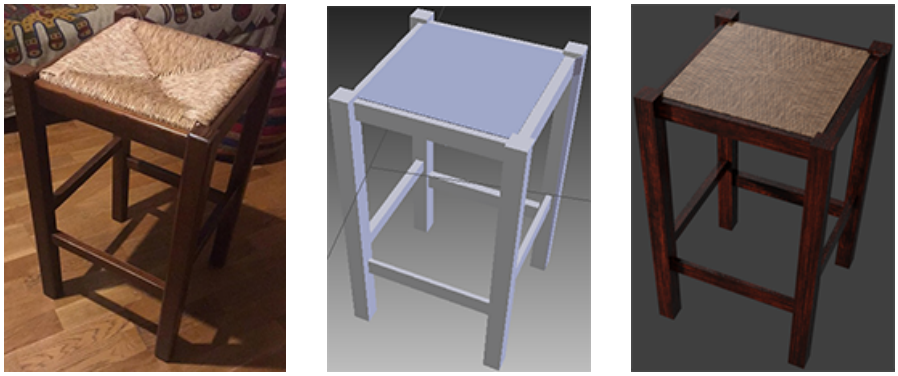


Figura 2.24: Inserimento porte e finestre

Oggetti low-poly

Tutti gli oggetti della scena sono stati modellati realizzando *mesh low-poly*, cioè composte da un numero esiguo di poligoni, garantendo la resa realistica del modello con *texture* realistiche piuttosto che con *mesh* dettagliate, al fine di minimizzare il tempo di *rendering* senza ripercussioni sulla qualità del prodotto. In figura 2.25 è mostrato, ad esempio, lo sgabello realizzato: come si può notare, esso non è altro che una composizione di parallelepipedi; grazie alla fase successiva di *texturing* e all'utilizzo delle *normal map*, la *mesh* acquisisce in seguito maggiore realismo.



(a) (b) (c)

Figura 2.25: Fotografia (a), modello (b) e resa (c) di uno sgabello

Rimanendo nell'ottica della riduzione del numero di vertici complessivo del progetto, si è ridotto al minimo l'utilizzo del modificatore *Subdivision Surface*, il quale fornisce certamente più realismo al modello, ma aumenta notevolmente il numero di poligoni e quindi si ripercuote sulle prestazioni di resa. Infatti, ogni volta che è stato applicato tale operatore per la modellazione di alcune geometrie complesse, come il divano e il letto, si è fatto in seguito ricorso ad un algoritmo di decimazione.

I tre algoritmi di decimazione presenti in *Blender* (vedi 3.8) sono accessibili come modificatori nel pannello *Properties* (*Properties* → *Add Modifier* → *Decimate*). Utilizzando, ad esempio, l'algoritmo di decimazione *Collapse* (vedi 3.8.5) si può definire la percentuale di riduzione della *mesh* tramite uno *slider*; nel progetto è stata applicata una percentuale di riduzione tra il 10% e il 50% in modo da ridurre il numero di poligoni per le *mesh* più elaborate, senza alterarne la qualità. Cliccando sul pulsante "*Apply*" la modifica viene applicata e la *mesh* viene quindi sostituita da una sua versione semplificata.

Avendo minimizzato il numero di *mesh* differenti per la produzione della scena, bisogna porre attenzione su quale tecnica utilizzare per moltiplicare un oggetto, nel caso in cui siano necessarie più istanze della stessa *mesh*. La duplicazione di oggetti in *Blender* può avvenire principalmente in due modalità:

- "Shift+D": crea un duplicato della *mesh* selezionata;
- "Alt+D": crea un duplicato "linkato" della *mesh* selezionata.

Nel caso in cui gli oggetti duplicati debbano essere completamente indipendenti dalla *mesh* sorgente, allora è necessario utilizzare la combinazione di tasti "Shift+D". Essa è utile, ad esempio, se si vogliono creare più versioni di una stessa *mesh*, dove ogni versione si differenzia dalle altre per qualche particolare. In questo modo, è possibile modificare una copia, senza che la modifica si ripercuota sugli altri duplicati. Se invece si vuole creare delle istanze tutte collegate tra di loro in modo che la modifica di una copia influenzi tutte le altre copie, allora bisogna creare duplicati "linkati" tramite la combinazione di tasti "Alt+D". Questa modalità può essere utile, ad esempio, se è noto a priori il numero di duplicati necessario per ciascuna *mesh*; in tal caso, è preferibile posizionare dapprima tutti i duplicati nella loro posizione definitiva e quindi migliorare il modello raffinando un oggetto campione per ciascuna tipologia di *mesh*. Nel corso di questo lavoro, si sono creati principalmente duplicati collegati e la variabilità delle singole istanze è stata realizzata applicando diversi valori di scala e rotazione (Fig. 2.26 (a)). Per utilizzare *texture* differenti si richiede, invece, di duplicare gli oggetti in modo non linkato, altrimenti tutte le istanze condividerebbero gli stessi materiali; ad esempio, i quadri devono avere *texture* differenti, pertanto devono essere istanze di una stessa *mesh* ma non collegate tra di loro (Fig. 2.26 (b)).

In altre situazioni in cui gli oggetti da moltiplicare devono essere posizionati ad intervalli regolari lungo una o più direzioni (ad esempio, gli scaffali della libreria), si consiglia di utilizzare il modificatore *Array* (vedi 3.9), il quale consente di generare sequenze di oggetti equamente distanziati. Impostato tale modificatore dal pannello *Modifiers* nella scheda *Properties*, è possibile definire il numero di copie da creare e l'offset tra una copia e l'altra lungo gli assi x, y e z. Come per le copie collegate, questa soluzione è preferibile nel caso in cui è noto il numero di copie da creare. Volendo apporre una modifica ad una o più copie dell'array, bisogna dapprima applicare il modificatore

(a) (b)

Figura 2.26: Duplicati linkati (a) e duplicati non linkati (b)

tramite il pulsante "*Apply*" in modo da convertire la trasformazione in una vera e propria geometria. In figura 2.27 è mostrata ad esempio una libreria divisa in quattro parti, realizzata a partire da un cubo moltiplicato in due direzioni tramite due modificatori *Array*.

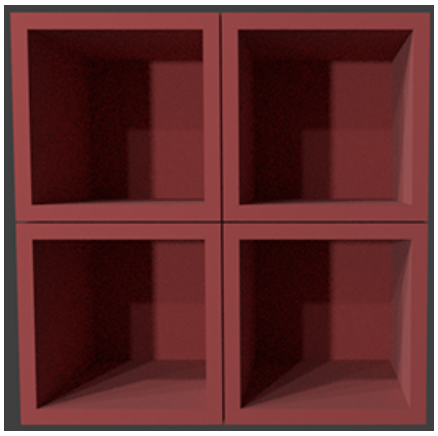


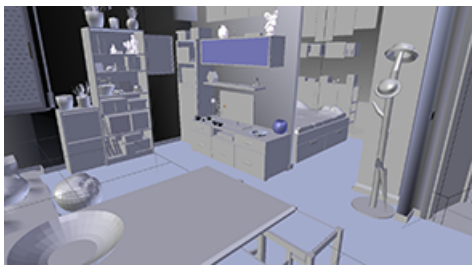
Figura 2.27: Libreria realizzata tramite il modificatore *Array*

Riassumendo, le tecniche di modellazione principalmente utilizzate per la realizzazione degli oggetti della scena sono le seguenti:

- Disegno di *mesh* a partire da primitive grafiche (cubi, cilindri e così via) per la produzione di modelli *low-poly*;

- Scarso utilizzo del modificatore *Subdivision Surface*;
- Decimazione delle *mesh* più complesse;
- Moltiplicazione di un oggetto tramite creazione di duplicati prevalentemente linkati.

In questo modo è stata realizzata la maggior parte degli oggetti della scena (Fig. 2.28). Menzione a parte richiedono alcune *mesh* più complesse, come il divano, la spirale di rame o le piante, realizzate con le tecniche descritte nei prossimi paragrafi.



(a)

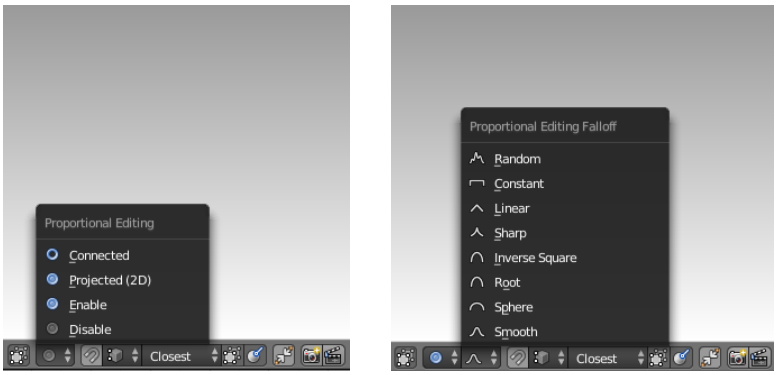
(b)

Figura 2.28: Modellazione (a) e resa (b) della scena

Proportional editing

Il *proportional editing* (vedi 3.10) è una tecnica di modellazione che consente di applicare una trasformazione agli oggetti selezionati (es. *mesh*, vertici, lati o facce) in modo che tale modifica influenzi gli elementi vicini in modo proporzionato. Ad esempio, selezionando un vertice e muovendolo lungo un asse, i suoi vertici vicini vengono anch'essi traslati nella stessa direzione in base ad un certo fattore di proporzione, consentendo così di realizzare superfici smussate che difficilmente potrebbero essere realizzate modificando un vertice alla volta. Questa tecnica è stata utilizzata, infatti, per la produzione di alcune superfici complesse, come il divano, i cuscini e il letto, simulando così in modo realistico le pieghe del tessuto che li ricopre.

Per abilitare il *proportional editing*, si può utilizzare l'apposito menu presente nell'*header* della *3D View* (Fig. 2.29 (a)), oppure si può semplicemente premere il tasto "O" per attivare o disattivare la funzione. In figura 2.29 (b) è mostrato, invece, il menu per la scelta della funzione di *falloff*, ossia la funzione che definisce il modo in cui una modifica viene trasferita ai vicini dell'elemento selezionato.



(a) (b)

Figura 2.29: Menu per il *proportional editing*

Tra le tante funzioni di *falloff* a disposizione, quella utilizzata in questo lavoro è la funzione *Smooth*, la quale applica un fattore di smussatura che previene la formazione di aree spigolose. Un esempio di realizzazione di una *mesh* tramite *proportional editing* è mostrata in figura 2.30. La procedura in dettaglio è costituita dai seguenti passaggi:

- Si parte da una primitiva grafica che approssima il volume dell'oggetto da realizzare. Ad esempio, per realizzare un letto si può partire da un parallelepipedo;

- Si suddivide la *mesh* fino alla risoluzione desiderata. Tanto maggiore è il numero di suddivisioni, tanto migliore sarà il grado di smussatura. Per effettuare una suddivisione, basta selezionare la *mesh*, passare in modalità *Edit Mode*, selezionare tutti i vertici premendo "A", digitare "W" e cliccare la voce "*Subdivide*";

- Abilitato il *proportional editing* e definita la funzione di *falloff*, si può procedere con la modellazione dell'oggetto. Mentre si sta eseguendo una trasformazione (ad esempio, una traslazione di un punto), si può muovere la rotella del mouse per aumentare o diminuire l'area di influenza del *proportional editing*, rappresentata da un cerchio intorno al cursore del mouse.

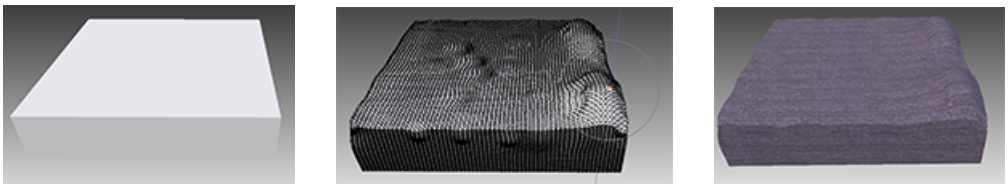


Figura 2.30: Fasi di realizzazione di una *mesh* tramite *proportional editing*

Gli oggetti realizzati con la tecnica del *proportional editing* sono mostrati in figura 2.31.

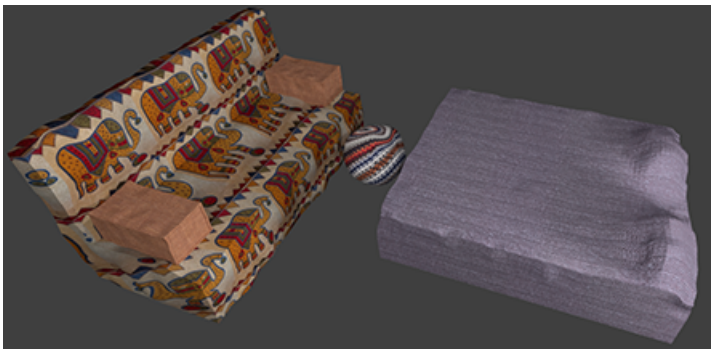


Figura 2.31: Oggetti realizzati tramite *proportional editing*

Sistemi particellari

Un sistema particellare (*particle system*, vedi 3.11) è un metodo di emissione di una grande quantità di *mesh* (in genere centinaia o migliaia), sfruttando una superficie come area dal quale emettere ogni singolo oggetto (o particella) in modo casuale. Vi sono due tipologie di sistemi particellari disponibili in *Blender*:

- *Emitter*: genera un'animazione in cui in ogni *frame* una particella viene creata e posizionata in modo casuale sulla superficie di emissione;
- *Hair*: le particelle vengono tutte disposte sulla *mesh* e possono essere renderizzate. È consigliato quando le particelle sono di tipo filiforme (ad esempio, erba, capelli, paglia e così via).

Tramite i sistemi particellari sono state realizzate alcune decorazioni della scena, come le piante. Dato che esse sono composte da elementi non filiformi, per la loro modellazione si è optato per il sistema particellare *Emitter*. In particolare, una volta generata l'animazione in cui le particelle vengono emesse una alla volta, è stato individuato il *frame* corrispondente alla configurazione esteticamente migliore per il modello, quindi in tale *frame* è stata effettuata la conversione del *particle system* in geometria; in questo modo le particelle emesse diventano *mesh* statiche, adatte per la successiva fase di produzione del video.

La costruzione di un sistema particellare richiede innanzitutto la definizione del gruppo di oggetti che costituirà la singola unità emessa come particella. Ad esempio, nel caso di una pianta, il gruppo sarà composto dalle diverse tipologie di foglie (in questo caso, 5) che si vuole generare in modo casuale. In altri casi (ad esempio, il vaso con le cannucce colorate) il gruppo sarà composto da una sola *mesh* (un cilindro). Per creare un gruppo, bisogna selezionare le *mesh* da raggruppare, digitare la combinazione di tasti "Ctrl+G" e dare un nome al gruppo.

A questo punto bisogna definire quali siano i vertici e le facce della superficie dove le particelle verranno generate. È possibile definire l'area di emissione del *particle system* cliccando sulla *mesh* sorgente, selezionando parte dei vertici in *Edit Mode*, creando un *Vertex Group* nella scheda *Properties* → *Object Data* e assegnando tali vertici a tale *Vertex Group* cliccando sul pulsante "*Assign*". In questo lavoro è stata invece utilizzata una tecnica alternativa, per cui la superficie di emissione è semplicemente un piano (*plane*) nascosto all'interno della *mesh* e quindi non visibile all'esterno. Questo piano deve ovviamente essere suddiviso molteplici volte, in quanto le particelle

verranno emesse dalle sue facce e pertanto un piano di default, avendo solo una faccia, genererebbe solo una particella.

Eseguiti questi due passaggi preliminari, si può ora procedere con la creazione del sistema particellare, selezionando il piano di emissione e cliccando sul pulsante "+" nella scheda *Particles* della finestra *Properties*. Vengono, quindi, mostrati i parametri per la sua configurazione, tra cui la tipologia del *particle system* e il numero totale di *mesh* da emettere. Scelto il sistema *Emitter*, si ha la possibilità di decidere inoltre se le particelle debbano essere emesse dai vertici, dalle facce o dai volumi selezionati (in questo caso si è scelto di emettere le particelle dalle facce). Nel campo *Dupli group* della sezione *Group* si deve ora inserire il nome del gruppo precedentemente creato. Se è stato definito un *Vertex Group* come sorgente di emissione, bisogna inoltre inserire tale gruppo nel campo *Density* della sezione *Vertex Groups*.

Ora, per dare un effetto ancora più caotico e casuale al sistema, è possibile inserire alcune forze che agiscono su ciascuna particella, alterandone la dimensione e la rotazione. Nella sezione *Physics* → *Netwonian*, sono stati infatti definiti i valori *Size* (dimensione iniziale della particella), *Random Size* (variazione casuale per il campo *Size*), insieme al valore *Brownian* che altera casualmente l'orientamento applicando un moto *browniano* ad ogni oggetto emesso. In questo modo, non solo gli oggetti vengono distribuiti in modo casuale sul piano, ma ogni oggetto si distingue dagli altri per differenti valori di rotazione e scala. In figura 2.32 è mostrato un esempio di configurazione di un *particle system*.

Il sistema così realizzato produce un animazione, visibile cliccando sul pulsante *Play* nella finestra *Timeline*. Individuato il *frame* in cui il sistema ha la configurazione migliore, è possibile convertire il sistema in geometria selezionando il *frame* nella *Timeline* e cliccando su "*Convert*" nel sistema particellare presente in *Properties* → *Modifiers*.

Gli oggetti realizzati tramite sistemi particellari sono mostrati in figura 2.33.

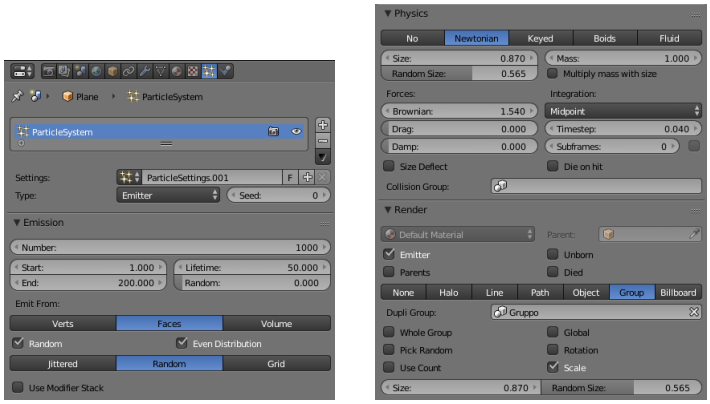


Figura 2.32: Esempio di configurazione di un *particle system*

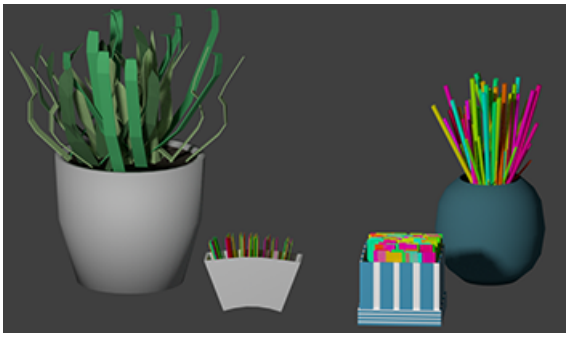


Figura 2.33: Oggetti realizzati tramite sistemi particellari

Curve di Bèzier

Alcuni oggetti particolari, come la spirale di rame e il portagioielli, sono stati realizzati utilizzando oggetti di tipo *Curve*, in particolare curve di Bèzier. Il motivo è la presenza di particolari curvature per le quali la modellazione per estrusione non è quella ideale.

Il disegno di una curva di Bèzier avviene in modo diverso dalla modellazione di una *mesh*: la forma di una curva è data dai suoi vertici, o punti di controllo, ognuno dei quali ha due manopole (*handle*) che permettono di gestire la forma della curva prima e dopo il vertice. Come avviene per le *mesh*, si può effettuare un'estrusione con il tasto "E" per aggiungere un nuovo punto di controllo e proseguire con il disegno. Le *mesh* modellate tramite

curve di Bèzier sono mostrate in figura 2.34.

Figura 2.34: Oggetti realizzati tramite curve di Bèzier

Mentre per i primi due oggetti mostrati sono state inserite due *Bèzier Curve*, il terzo è, invece, una composizione di *Bèzier Circle*, una curva di Bèzier con la particolarità di essere chiusa e di avere una forma circolare.

Modellata la forma degli oggetti, nella scheda *Properties* → *Data* è possibile definire il fattore di estrusione e di *Bevel* per la produzione di una superficie 3D. In questo caso, è stato dato un valore di 5 centimetri per entrambi i campi "*Extude*" e "*Bevel depth*"

2.3.4 Generazione dei materiali

Node Editor

Per poter proseguire con la fase di *texturing*, bisogna dapprima definire i materiali che opportunamente combinano gli *shader*[2] e le immagini *texture* per la colorazione realistica delle *mesh* modellate. La creazione dei materiali avviene nella finestra *Node Editor*. Utilizzando il motore di *rendering Cycles*, ciascun materiale può essere realizzato come una rete di nodi, dove un nodo non è altro che un oggetto che riceve uno o più input e produce uno o più output, tramite l'applicazione di diverse operazioni matematiche. Esistono varie tipologie di nodi, che permettono di manipolare il colore, lo *shader*, le

[2]In Blender uno *shader* è una composizione di nodi il cui output definisce l'aspetto della superficie di un oggetto quando viene illuminato da una o più sorgenti luminose, producendo ad esempio effetti di riflessione, trasparenza e rifrazione

texture e così via. In Figura 2.35 è mostrato un materiale di esempio costruito all'interno del *Node Editor*.

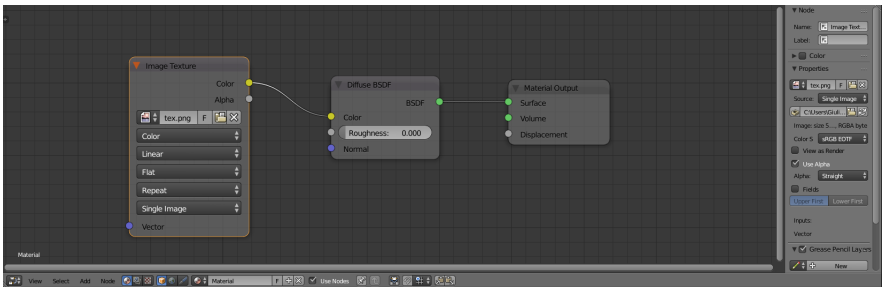


Figura 2.35: *Node Editor*

Come per la *3D View*, l'aggiunta di un nuovo nodo avviene con la combinazione di tasti "Shift+A" e scegliendo nel menù a tendina la tipologia del nodo voluta. Allo stesso modo, la rimozione avviene con il tasto "X". Un nodo è rappresentato come un rettangolo con le sue proprietà all'interno e alcuni cerchi colorati (*sockets*), che rappresentano i canali di comunicazione tra il nodo e gli altri nodi: un nodo riceve gli input dai *sockets* a sinistra ed emette gli output attraverso i *sockets* a destra. Il nodo *Material Output* è il nodo che rappresenta il risultato finale del materiale, pertanto non ha punti di connessione in uscita.

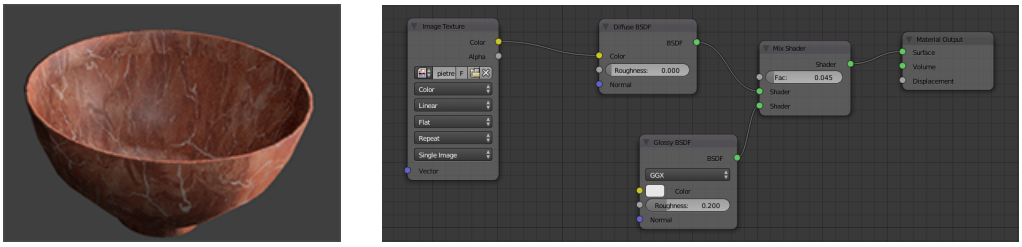
Le connessioni possono avvenire tra *sockets* dello stesso colore (ossia dello stesso tipo): ad esempio, nella figura 2.35 il nodo *Diffuse BSDF* riceve l'informazione "*Color*" dal nodo *Image Texture*. Per creare una connessione tra due nodi si può cliccare con il tasto sinistro del mouse sul *socket* di output del primo nodo e, tenendo premuto, spostare il cursore in corrispondenza del *socket* di input del secondo nodo. All'interno della finestra *Properties* è possibile vedere in ogni momento l'aspetto finale del materiale nella sezione *Preview*.

Texture

La maggioranza degli oggetti della scena sono stati texturizzati utilizzando una sola immagine, che si è rivelata sufficiente per i fini di realismo

prefissati. Un esempio di oggetto texturizzato in questo modo è mostrato in figura 2.36. Tutti questi materiali condividono la stessa architettura di nodi:

- *Diffuse BSDF*: nodo che gestisce la diffusione del colore dell'oggetto. Riceve in input il colore dal nodo *Image Texture*;
- *Image Texture*: nodo che carica l'immagine che si vuole utilizzare come *texture*;
- *Glossy BSDF*: nodo che aggiunge effetti di riflessione della luce;
- *Mix Shader*: nodo che unisce l'effetto dei due *shader* utilizzati.



(a) (b)

Figura 2.36: Oggetto (a) e rete di nodi del relativo materiale (b)

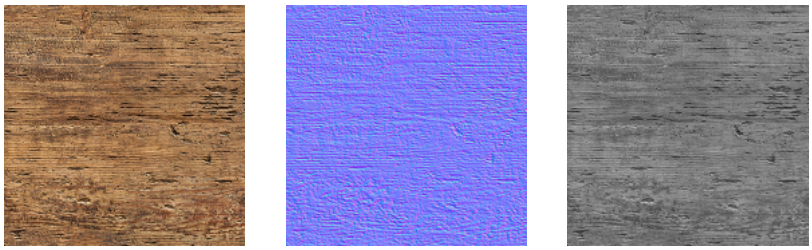
Legno

La tecnica di *texturing* utilizzata per la realizzazione dei diversi tipi di legno è nota come PBR (*Physically-Based Rendering*). Il PBR è un metodo per conferire realismo ad oggetti *low-poly* senza aumentarne il numero di poligoni, ma sovrapponendo più *texture* in modo da simulare l'effetto realistico di un materiale, come ad esempio la ruvidezza, le asperità, la quantità di luce riflessa e così via. Data un'immagine, è possibile calcolare un insieme di immagini derivate, dette "mappe", per la produzione di un materiale realistico. Le tre mappe utilizzate sono le seguenti:

- *Diffuse map*: definisce il colore di un materiale (Fig. 2.37 (a));

- *Normal map*: simula la complessità di una superficie calcolando le normali a partire da un modello poligonale dettagliato e applicando tale mappa ad una versione *low-poly* della stessa *mesh* (Fig. 2.37 (b));

- *Roughness map*: definisce il grado di ruvidezza del materiale. Superfici ruvide tendono a riflettere la luce in direzioni casuali, mentre superfici lisce forniscono una riflessione speculare al raggio di luce (Fig. 2.37 (c));



(a) (b) (c)

Figura 2.37: *Diffuse Map* (a), *Normal Map* (b) e *Roughness Map* (c) di una *texture*

Il software utilizzato per la generazione delle *normal*, *diffuse* e *roughness map* è *Awesome Bump*, il quale, caricata un'immagine di partenza, genera le mappe desiderate in base ad un insieme di parametri definiti dall'utente, tra cui il livello di dettaglio, il contrasto, il fattore di scala e così via.

Realizzate le tre mappe per ciascuna *texture*, si è creato in *Blender* un materiale che sovrappone le tre immagini, il quale è stato infine assegnato alla *mesh* da texturizzare. L'architettura di un materiale con PBR (Fig. 2.38) realizzato in *Blender* è composta dai seguenti nodi:

- Tre nodi *Image Texture*: nodi con cui caricare la *diffuse map*, la *normal map* e la *roughness map*. I nodi relativi alla *normal map* e alla *roughness map* devono essere impostati come "*non-color data*", in quanto l'informazione del colore deve essere calcolata solo dalla *diffuse map*;

- Un nodo *Normal map*: nodo che prende in input i dati della *normal map* e da in output il valore "*Normal*" al nodo PBR;
- Un nodo PBR: Riceve in input i valori della *diffuse map*, della *roughness map* e dalla *normal map*, producendo il valore di *shader* in output;
- Un nodo *Material Output*: nodo che rappresenta il risultato finale. Riceve il valore "*Surface*" dal nodo PBR.

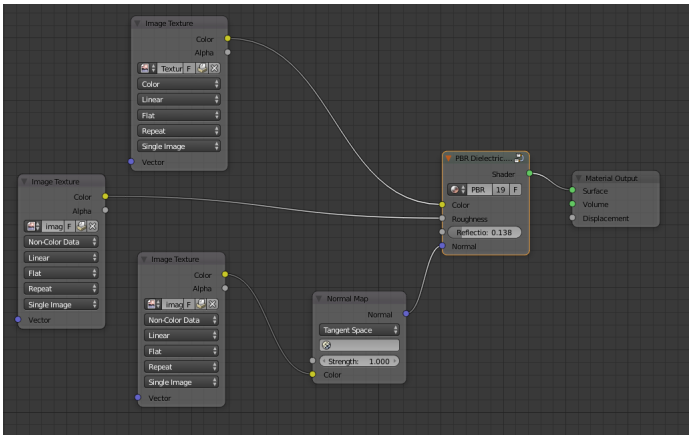


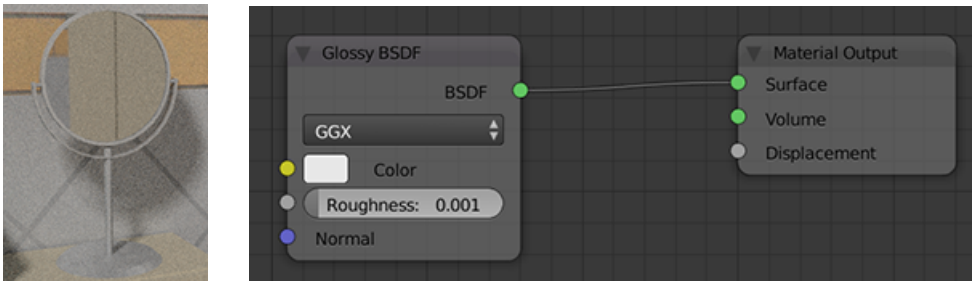
Figura 2.38: Materiale con PBR

Figura 2.39: Confronto tra una scena con materiale PBR (sopra) e senza materiale PBR (sotto)

In figura 2.39 sono mostrate due rappresentazioni della stessa scena che differiscono per il materiale assegnato al legno: nell'immagine superiore, si è applicata la tecnica PBR; in quella inferiore, invece, si è utilizzata una sola *texture* come fonte del colore. Come si può vedere, la prima scena è molto più realistica della seconda e inoltre l'incremento in tempo e memoria richiesto per l'elaborazione di un materiale PBR è trascurabile.

Specchio

Il materiale per i tre specchi presenti sulla sena è molto semplice ed è gestito da un solo nodo *Glossy BSDF*. Se il fattore *roughness* è impostato ad un valore molto piccolo (ad esempio, 0.001), allora il raggio luminoso che irradia la superficie viene completamente riflesso senza che esso fornisca un fattore di illuminazione all'oggetto, realizzando così un effetto a specchio (Fig. 2.40).



(a) (b)

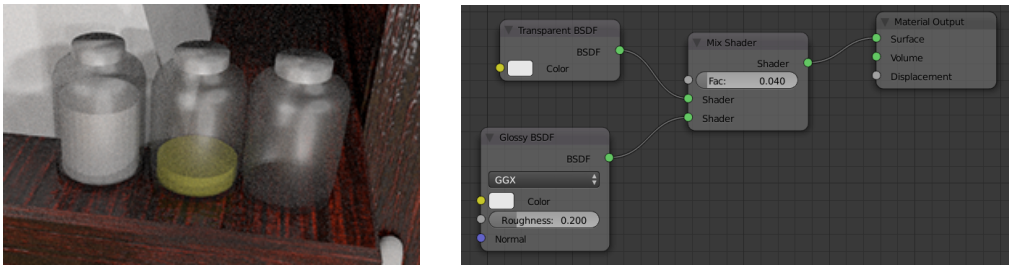
Figura 2.40: Specchio (a) e rete di nodi del relativo materiale (b)

Vetro

Per gli oggetti trasparenti (Fig. 2.41), come i bicchieri e le bottiglie di profumo, bisogna usare un nodo *Mix Shader* che permette di unire gli effetti di trasparenza e di riflessione della luce; tramite un *slider* è possibile definire quale *shader* debba prevalere sull'altro e in che proporzione. In particolare, i due *shader* utilizzati sono:

- *Transparent BSDF*: fornisce l'effetto trasparente all'oggetto;
- *Glossy BSDF*: permette di aggiungere effetti di riflessione all'oggetto;

In questo caso, è la componente trasparente che deve essere preponderante, quindi lo *slider* del *Mix Shader* deve essere impostato ad un valore basso per far prevalere il primo *shader* collegato. Per rappresentare i vari tipi di vetro colorati, il materiale è stato semplicemente duplicato modificando l'attributo *Color* in entrambi gli *shader*.



(a) (b)

Figura 2.41: Oggetti trasparenti (a) e rete di nodi del relativo materiale (b)

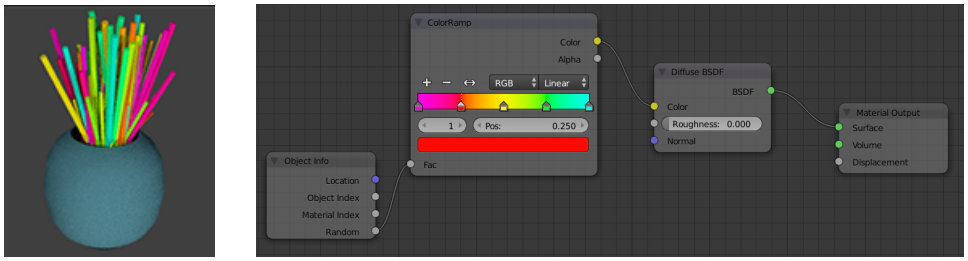
Casuale

Gli oggetti emessi dal *particle system* richiedono un colore che sia casuale per ogni particella (Fig. 2.42). Per ottenere questo risultato sono necessari due nodi ulteriori, oltre al nodo *Diffuse BSDF*: *ColorRamp* e *Object Info*. In particolare nel primo nodo, tramite i pulsanti "+" e "-" è possibile modificare l'intervallo di colori RGB dal quale verrà selezionato un colore casuale. Il secondo nodo, invece, è necessario per far sì che la scelta del colore sia gestita dall'attributo *Random* dell'oggetto.

2.3.5 Texturing

UV/Image Editor

L'*UV/Image Editor* è la finestra che permette di effettuare il *texture mapping* tra immagini 2D e oggetti 3D (Fig. 2.43).



(a) (b)

Figura 2.42: Colori casuali (a) e rete di nodi del relativo materiale (b)

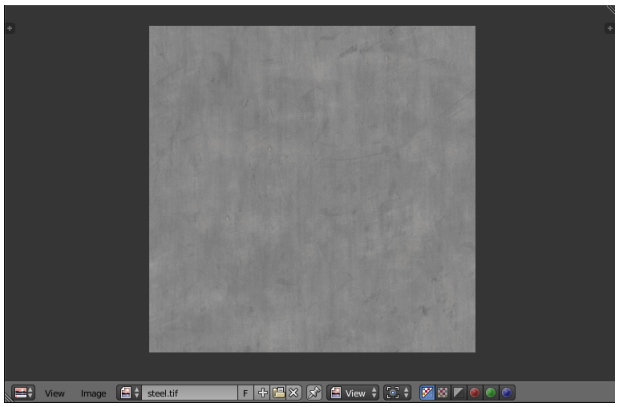


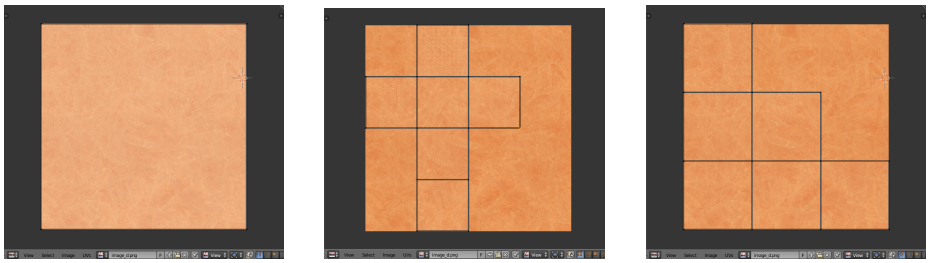
Figura 2.43: *UV/Image Editor*

Per poter texturizzare una *mesh*, bisogna innanzitutto creare un materiale da assegnare all'oggetto interessato nella finestra *Properties* → *Material*. Quindi, in modalità *Edit Mode* si devono selezionare le facce della *mesh* da texturizzare e premere il tasto "U"; a questo punto *Blender* mette a disposizione diverse modalità di *texture mapping*, che sfruttano alcuni spigoli marcati (*seams*) per "aprire" la *mesh* in modo che essa venga proiettata in due dimensioni:

- *Unwrap* (vedi 3.12.1): sfrutta i *seams* definiti dall'utente per eseguire l'*UV mapping*. Per creare un *seam* basta selezionare uno spigolo della *mesh*, digitare la combinazione di tasti "Ctrl+E", quindi selezionare la voce "*Mark Seam*";

- *Smart UV Project* (vedi 3.12.2): esegue l'*UV mapping* sfruttando *seams* generati automaticamente, in modo che ciascuna faccia sia texturizzata con una porzione differente della *texture*.

In figura 2.44 sono mostrare le due modalità di *texture mapping* eseguite su un cubo. Come si può notare, l'*Unwrap* eseguito su un cubo senza la definizione dei *seams* porta ad avere tutte le facce texturizzate con la stessa area della *texture*. Una volta eseguito il *texture mapping*, è possibile modificare individualmente le *UV map* delle singole facce della *mesh*. Selezionata una faccia, nella finestra *UV/Image Editor* viene mostrata la porzione di *texture* ad essa assegnata, che può essere ruotata, traslata o scalata come se fosse una geometria nella *3D View*.



(a) (b) (c)

Figura 2.44: *Unwrap* senza *seams* definiti (a), *Unwrap* con *seams* definiti (b) e *Smart UV Project* (c)

Parametrizzazione UV

La scelta del tipo di *texturing* da eseguire su una *mesh* è stata guidata dalla sua forma: per forme semplici, come un piano (ad esempio, i quadri) o un insieme di parallelepipedi (ad esempio, lo sgabello), le facce sono state texturizzate una alla volta tramite la tecnica *Unwrap* e la parametrizzazione UV dei corrispondenti vertici è stata modificata all'interno della finestra *UV/Image Editor*, aumentando o diminuendo la scala della faccia proiettata.

Oggetti sferici e circolari (ad esempio, la pallina da tennis), come anche gli oggetti più complessi, come il peluche, sono stati texturizzati invece con

la tecnica *Smart UV Project*, grazie alla capacità dell'algoritmo di esaminare la forma dell'oggetto e di individuare il miglior insieme di *seams* per la proiezione di ciascuna faccia.

Infine, per il divano si è dovuto ricorrere ad una terza soluzione, sfruttando l'attuale angolo di vista nella *3D View* come angolo di proiezione (*Project From View*). In questo modo la *mesh* viene appiattita così come appare nella *viewport*. Il motivo di questa scelta è la necessità di utilizzare una fotografia dell'oggetto come *texture* per il modello. Inquadrando la *mesh* dallo stesso angolo di vista dal quale è stata scattata la fotografia, si ottiene così un modello texturizzato in modo fedele all'oggetto reale.

2.3.6 Animazione

Timeline e Graph Editor

La *Timeline* è la finestra per la creazione delle animazioni. L'animazione degli oggetti avviene tramite la definizione di *keyframes* (vedi 3.13), ossia punti chiave del flusso temporale che determinano la posizione, rotazione e scala di un oggetto in precisi istanti di tempo; l'animazione complessiva viene calcolata come l'interpolazione di tali *keyframes*.

La finestra *Timeline* mostra al centro la sequenza di *frame*, dove il cursore verde indica il *frame* attualmente selezionato. Nella parte inferiore della finestra si possono definire i *frame* iniziali e finali dell'animazione e, di conseguenza, la sua durata. Per poter inserire un *keyframe* per l'animazione di una *mesh*, innanzitutto si deve selezionare il *frame* da trasformare in *keyframe*; effettuata quindi una trasformazione sulla *mesh*, in *Object Mode* si può premere il tasto "I" per visualizzare il menu per l'inserimento di un *keyframe*. A questo punto si deve indicare cosa si vuole salvare in tale *frame*, ad esempio la posizione e la scala dell'oggetto. Premendo il tasto "*Play*", è possibile visualizzare nella *3D View* tutte le animazioni create nell'intervallo di *frame* definito. In figura 2.45 è mostrata un'animazione di 46 *frame* con 9 *keyframes*.

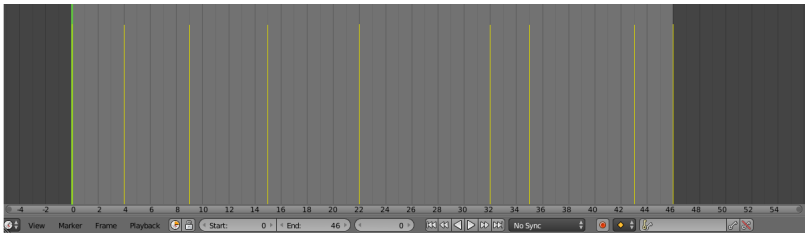


Figura 2.45: *Timeline*

La gestione delle animazioni realizzate nella *Timeline* avviene all'interno della finestra *Graph Editor*. Ogni animazione viene gestita da una particolare curva, detta F-curva, che utilizza i *keyframes* di un'animazione come suoi punti di controllo. Essa non è altro che una funzione di Bèzier o una funzione nella base di Bernstein: in questo modo è possibile avere per ogni istante temporale in ascissa un unico valore per la F-curva.

Ogni *keyframe* o vertice della F-curva ha due manopole (*handle*) che consentono di modellare la forma della F-curva tra due *keyframe* consecutivi. Essi sono rappresentati come linee tangenti al *keyframe* con estremi circolari, i quali possono essere traslati con il tasto "G" come se fossero vertici di una geometria nella *3D View*. In figura 2.46 è mostrata una schermata del *Graph Editor*, in cui un'animazione viene rappresentata con due F-curve che rappresentano ognuna la trasformazione dell'oggetto lungo una direzione.

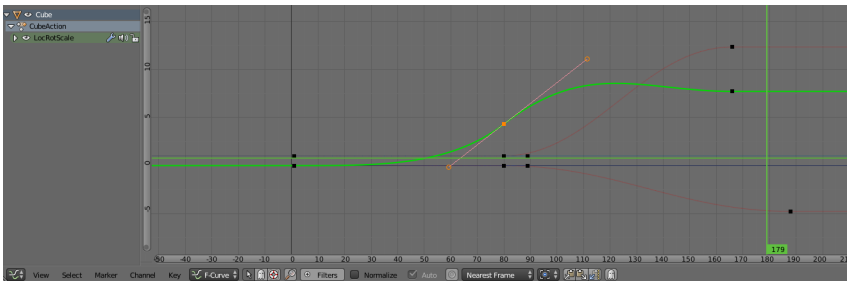


Figura 2.46: *Graph Editor*

Percorso camera

Sulla scena è stato inserito un oggetto Camera di tipo "Perspective" e con lunghezza focale pari a 15 millimetri, posizionata all'esterno dell'edificio e rivolta verso la porta d'ingresso. Quindi, a partire dagli *storyboard* realizzati, è stato definito il suo percorso all'interno dell'appartamento.

Per facilitare la realizzazione dell'animazione, si consiglia di modificare il *layout* di *Blender* per ospitare due *3D View*: una per spostare e ruotare la camera e un'altra per visualizzare in ogni momento la scena inquadrata dalla camera. In particolare, per impostare una *3D View* in vista camera, basta digitare il tasto "0" del tastierino numerico (Fig. 2.47).

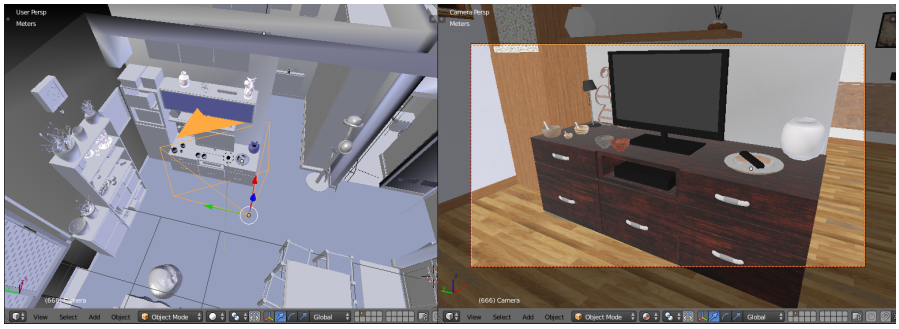
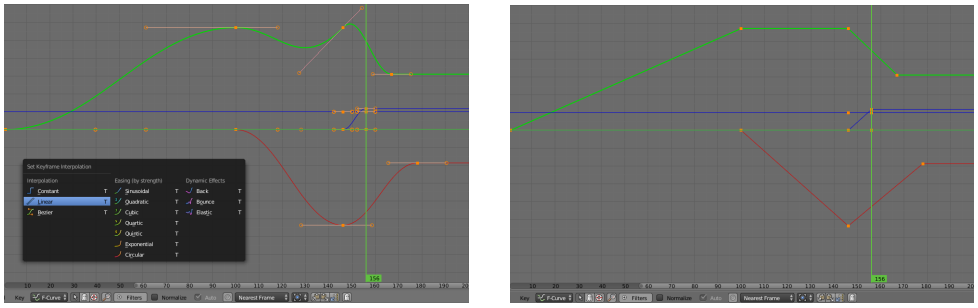


Figura 2.47: Due *3D View* per realizzare l'animazione della camera

Di default, *Blender* applica un'interpolazione di Bèzier ai *keyframes* inseriti, per cui la trasformazione è più lenta nei *frame* estremi dell'animazione, mentre è più veloce nei *frame* intermedi. In questo caso, si vuole invece che la trasformazione avvenga in modo graduale. Per cambiare tipo di interpolazione, nel *Graph Editor* bisogna selezionare i *keyframes* che si vuole alterare (in questo caso, tutti), premere"T" e selezionare la voce "*Linear*" tra le possibili tecniche di interpolazione. In figura 2.48 è mostrato ad esempio il passaggio da un'interpolazione di Bèzier ad una lineare.

L'animazione complessiva della camera consta di circa 3000 *frame* (Fig. 2.49), ossia circa 2 minuti quando il *frame rate* è pari a 24 *frame* al secondo. Le concentrazioni maggiori di *keyframe* sono in corrispondenza delle riprese con rotazione intorno ad un oggetto, ossia quando il punto di rotazione (*pivot*)



(a) (b)

Figura 2.48: Esempio di interpolazione di Bèzier (a) e lineare (b)

non è il centro della camera, ma l'origine dell'oggetto ripreso. In questo caso non si può sfruttare l'interpolazione di alcuni istanti chiave, in quanto la rotazione della camera comporta anche una modifica della sua posizione; posizione e rotazione verrebbero interpolate diversamente, producendo così un effetto altalenante della ripresa. Per questo motivo, ogni *frame* di tali sequenze deve essere un *keyframe*.

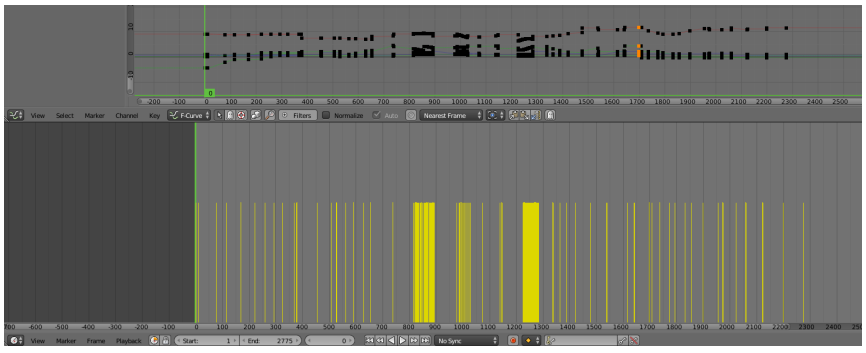


Figura 2.49: Animazione camera

Porta d'ingresso

L'animazione che costituirà l'inizio del video è l'apertura della porta d'ingresso, in cui la chiave viene ruotata tre volte di 180 gradi, la maniglia viene abbassata e la porta viene ruotata di 90 gradi. Mentre la rotazione della chiave avviene intorno al suo centro, le rotazioni della maniglia e della porta

avvengono intorno ad un punto di rotazione (*pivot point*) di confine della *mesh*. In questi ultimi due casi, la rotazione comporta anche una traslazione dell'oggetto, pertanto in ciascun *keyframe* bisogna inserire sia la posizione che la rotazione dell'oggetto.

Per modificare il punto di rotazione, bisogna dapprima selezionare un vertice della *mesh* e posizionarvi il *3D cursor*, digitando "Shift+S" e selezionando la voce "Cursor to Selected"; quindi, bisogna modificare il punto di pivot per la rotazione, presente nell'*header* della *3D View*, da "*Median Point*" a "*3D Cursor*".

In figura 2.50 sono mostrati a sinistra 3 *frame* dell'animazione e a destra la configurazione delle finestre *Timeline* e *Graph Editor*.

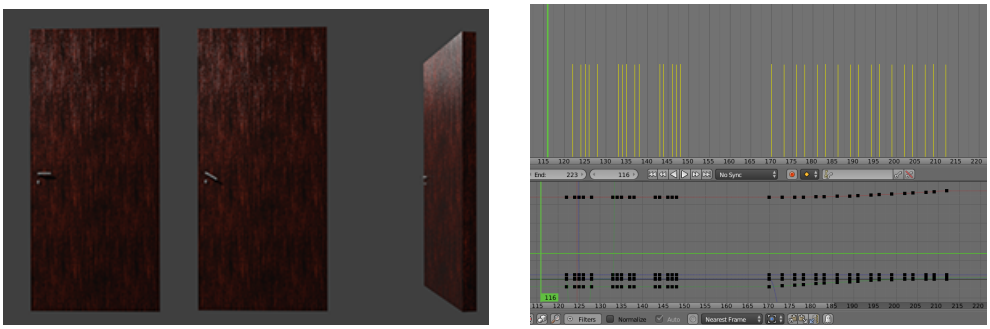


Figura 2.50: Animazione della porta

Orologio

L'orologio a pendolo è stato animato con 28 *keyframes* ripetuti ciclicamente per tutta la durata dell'animazione della scena.

L'animazione consiste in una rotazione lungo l'asse y del pendolo di 5 gradi in più o in meno rispetto al *frame* precedente, raggiungendo un angolo massimo di 35 e -35 gradi (Fig. 2.51). Come perno di rotazione, si è utilizzato l'estremo superiore del pendolo; come già detto in precedenza, l'animazione così realizzata porta anche ad una modifica della posizione dell'oggetto, in quando la rotazione non avviene intorno al suo centro. Quindi in ogni *keyfra-*

me bisogna salvare non solo la rotazione ma anche la posizione dell'oggetto, tramite il comando *LocRot* nel menu di inserimento del *keyframe*.

Figura 2.51: *Frame* 1, 8, 15, 22 e 28 dell'animazione dell'orologio

Per fare in modo che l'animazione prodotta venga eseguita ripetutamente in modo ciclico, bisogna selezionare tutti i *keyframes* nella finestra *Graph Editor* con il tasto "A", digitare la combinazione di tasti "Shift+E", quindi selezionare la voce "*Make Cyclic* ($F\_Modifier$)". In figura 2.52 è mostrato l'esito di questa sequenza di operazioni, con la quale i *keyframes* definiti vengono ripetuti all'infinito.

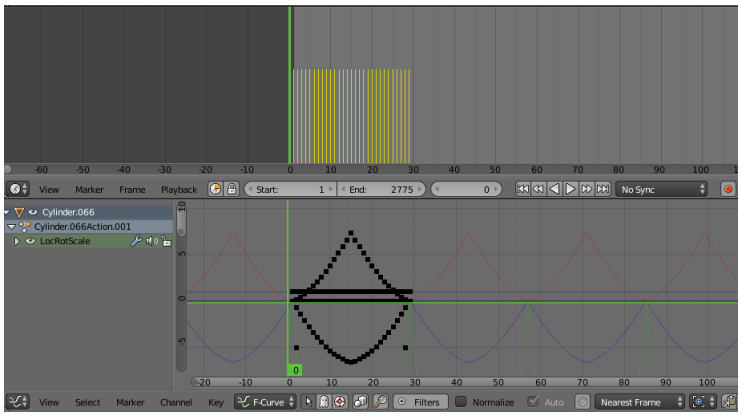


Figura 2.52: Animazione ciclica

2.3.7 Rendering

Terminata la modellazione e l'animazione del modello, si è proceduto con l'inserimento delle fonti luminose all'interno della scena, per poi proseguire

con la fase di *rendering*. In particolare, sono stati inseriti tre oggetti *Lamp* di tipo *Point*, uno per ogni stanza, posizionato in prossimità del soffitto. Inoltre, un'ulteriore fonte luminosa, sempre di tipo *Point*, è stata aggiunta all'interno della lampada vicina al televisore, la quale, essendo stata definita trasparente, in fase di *rendering* permetterà di simulare la diffusione della luce nell'area circostante (Fig. 2.53).

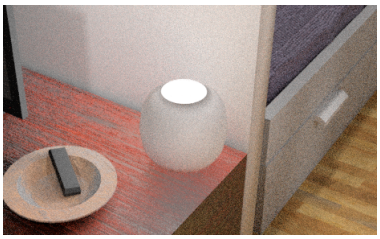


Figura 2.53: Lampada

Vi sono vari metodi per poter eseguire il *rendering* su *Blender*. Innanzitutto, è possibile effettuare un'anteprima (*preview*) del *rendering* della scena visualizzata nella *3D View*, tramite la combinazione di tasti "Shift+Z". Se si vuole ridurre l'area della *viewport* da renderizzare, si può definire un confine rettangolare del *rendering* (*Render Border*) cliccando su $View \rightarrow Render\ Border$ e tracciando l'area desiderata. La qualità del risultato di anteprima dipende dal numero di *samples* inseriti nel campo $Sampling \rightarrow Preview$. In genere, un numero di *samples* pari a 10 permette di avere una buona approssimazione del *rendering* finale (Fig. 2.54).

Figura 2.54: Render di *preview* del *frame* 609 con 10 *samples*

La resa vera e propria dell'area inquadrata dalla camera nel *frame* attualmente selezionato può essere eseguita con il tasto "F12" oppure cliccando sul pulsante "*Render*" nella scheda *Properties* → *Render*. Nella finestra *UV/Image Editor* viene mostrato lo stato di computazione del *Render Result*, ossia dell'immagine prodotta. Il *rendering* complessivo di un'animazione richiede, invece, la definizione preliminare del range di *frame* da renderizzare, quindi con il pulsante "*Animation*" nella scheda *Properties* → *Render* si può far partire il *rendering* dell'intera scena.

Tutti i parametri di configurazione per la resa del modello sono presenti nella scheda *Properties* → *Render*. Innanzitutto, il primo fattore da tenere in considerazione è il formato di output (*Resolution*) che si vuole ottenere per ogni singola immagine. Di default, *Blender* utilizza una risoluzione di 1920 x 1080 e applicando un fattore di scala di riduzione del 50%, producendo quindi immagini di 960x540 pixel. Dopo aver prodotto un video di prova con fattore di scala al 10%, sono stati realizzati quindi altri due video a maggior risoluzione, in particolare al 50% e al 80%.

Come già detto, il *Frame Range* definisce l'intervallo di *frame* dell'animazione che si vuole renderizzare. In questo caso, il range va da 1 a 3134, pari alla durata dell'animazione della camera.

Nella sezione *output* si può definire la cartella in cui verranno salvati i *frame* resi, insieme ad alcuni parametri di codifica, come il formato di output, colore e *color depth*, ossia la profondità in bit di ciascun canale di colore. Per questo lavoro, ogni *frame* è stato salvato come un'immagine PNG a colori (RGB) con *color depth* pari a 16.

Le sezioni successive permettono, invece, di definire gli algoritmi di *lighting* e di *sampling* necessari per calcolare i percorsi della luce e rendere ogni pixel per campioni successivi. Il motore di *rendering* adottato è *Cycles*, il quale mette a disposizione due algoritmi di *rendering* per il calcolo dell'illuminazione (*integrator*):

- *Path Tracing*: ogni raggio luminoso che colpisce una superficie può rimbalzare in una direzione casuale (*diffuse ray*) o essere riflesso (*glossy*

ray) o essere influenzato da altri fattori, come la rifrazione (*transmission ray*). È un algoritmo veloce da calcolare, ma richiede un gran numero di campioni (*samples*) per simulare realisticamente il colore e la luminosità di ciascun pixel della scena inquadrata;

- *Branched Path Tracing*: il raggio luminoso viene dapprima suddiviso in tanti raggi quante sono le componenti *shader* da simulare e ogni raggio così ottenuto viene gestito come un *sample* indipendente. In genere, è un algoritmo più lento del *Path Tracing*, ma può portare a risultati migliori.

Entrambe le versioni appartengono alla categoria di algoritmi *ray tracing* di tipo *backward* [23]: mentre gli algoritmi *Forward Ray Tracing* simulano il funzionamento della luce reale, generando migliaia di fotoni al secondo che partono dalla sorgente luminosa e rimbalzano sulla scena, il *Backward Ray Tracing* inverte il problema, seguendo i raggi luminosi a ritroso: in questo modo, ci si può concentrare solo su una regione dello spazio (cioè quella inquadrata dall'osservatore) e verificare se vi siano dei fotoni che contribuiscono all'illuminazione dell'area ripresa.

In pratica, il *Backward Path Tracing* simula un certo numero di raggi luminosi che partono dall'osservatore e rimbalzano su ciascun pixel dell'immagine. Essendo un algoritmo di tipo Monte Carlo[3], si esegue una valutazione probabilistica dei vari percorsi della luce: per ogni pixel si definiscono casualmente alcuni raggi, detti *sample rays* o *samples*, per simulare i vari percorsi che la luce può seguire. Un numero maggiore di *samples* porta a risultati di qualità migliore: matematicamente la luminosità di un oggetto deve tener conto dei raggi luminosi provenienti da tutte le direzioni, portando ad una somma infinita (integrale) di fattori di luminosità nell'equazione di *rendering*. Maggiori sono i *samples*, migliore sarà pertanto la discretizzazione dell'integrale come somma finita dei contributi luminosi che ciascun raggio porta ai pixel dell'immagine.

[3]Gli algoritmi di tipo Monte Carlo sono una classe di algoritmi il cui risultato deriva da ripetuti campionamenti casuali delle variabili di input.

L'algoritmo scelto per questo scopo è il *Path Tracing* (vedi 3.15.1), usando 60 *samples* per la resa di ogni pixel, ottenendo così immagini ad alta qualità. I parametri successivi della scheda permettono di configurare l'*integrator* scelto: ad esempio, il campo *Seed* (impostato a 3) contiene il valore che l'algoritmo utilizza per la produzione di diversi pattern di rumore, mentre i campi *Clamp Direct* e *Clamp Indirect* (entrambi impostati a 1) consentono di limitare l'intensità massima di ciascun campione.

Infine, nella sezione *Light Paths*, si può definire il numero di rimbalzi (*bounces*) di un raggio luminoso all'interno della scena, cioè quante volte esso colpisce una superficie e viene riflesso. In questo caso, sono stati definiti 3 massimi rimbalzi per i raggi di diffusione (*diffuse ray*) e un solo rimbalzo per i raggi di riflessione (*glossy ray*).

Tutti gli altri parametri presenti nella scheda e non precedentemente elencati sono stati lasciati nella loro configurazione di default. Riassumendo, le impostazioni per il *rendering* sono mostrate in figura 2.55.

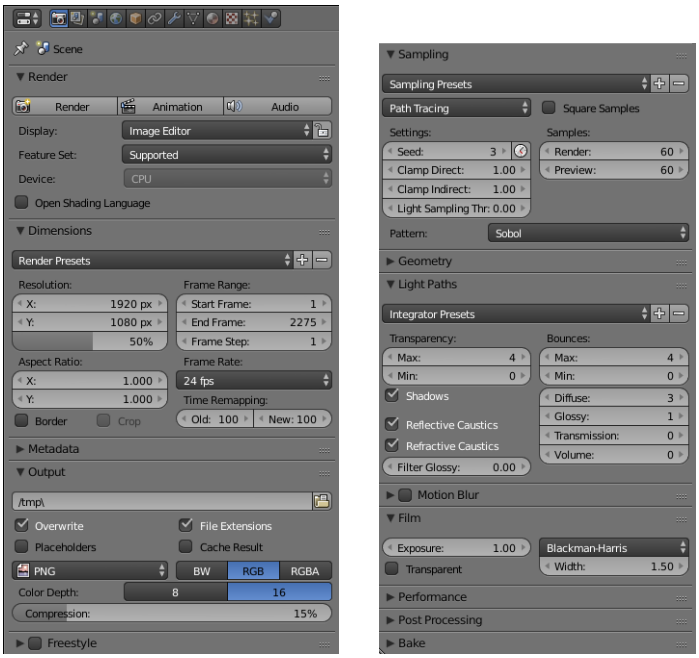


Figura 2.55: Impostazioni di *rendering*

In figura 2.56 sono mostrate infine alcune inquadrature della scena ren-

derizzate con i parametri precedentemente descritti.

Figura 2.56: Resa di alcune inquadrature della scena

2.3.8 Produzione filmato

Video Sequence Editor

La finestra *Video Sequence Editor* consente la produzione di un filmato a partire da un insieme di immagini *frame* renderizzate. La finestra è strutturata in canali (*channels*), ognuno dei quali può contenere *strip*, ossia sequenze di immagini, animazioni o effetti di transizione. Cliccando sul menu *Add* è possibile caricare un insieme di immagini o video o audio per importarli in *Blender* ed inserirli come *strip* nel canale selezionato. Le *strip* possono essere concatenate lungo lo stesso canale (per eseguirle in sequenza), oppure impilate su canali differenti nello stesso intervallo di *frame* (per eseguirle contemporaneamente). In figura 2.57 è mostrata un esempio di creazione di un filmato.

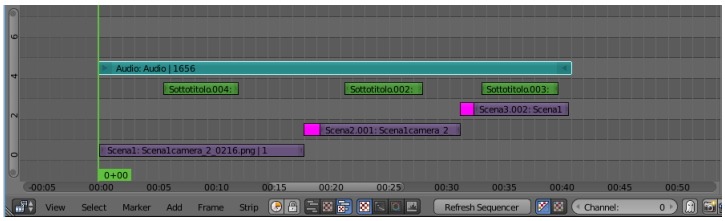


Figura 2.57: *Video Sequence Editor*

Blender usa colori differenti per rappresentare le diverse tipologie di *strip*:

- Viola: sequenza di immagini;
- Verde: scena 3D;
- Blu: video;
- Rosa: transizione tra due *strip* senza sovrapposizione. In questo modo lo *strip* successivo inizierà solo quando sarà ultimata la *strip* precedente;
- Arancione: transizione tra due *strip* con sovrapposizione. La sua dimensione determina quanta parte dei due *strip* viene tagliata per la creazione dell'effetto di transizione;

- Celeste: suoni.

E' fondamentale mantenere un corretto ordinamento verticale delle varie *strip*: l'ordine di *stack* dei canali procede dal basso verso l'alto, quindi il canale più in basso costituirà lo sfondo del filmato, mentre nel canale più in alto bisogna inserire tutto ciò che deve essere mostrato in primo piano. Quindi, i *frame* devono costituire la base del video e caricati quindi nel canale inferiore, mentre gli effetti grafici, i suoni, i sottotitoli e così via devono essere posti nei canali superiori.

Montaggio video

Il montaggio del video deve essere eseguito in un nuovo progetto di *Blender*, in cui i *frame* precedentemente renderizzati vengono caricati come *strip* all'interno di un canale del *Video Sequence Editor*. Cliccando su "*Add* → *Image*" e selezionando l'insieme di immagini volute, esse vengono inserite come un'unica sequenza (*strip*) all'interno del canale selezionato. Si è provveduto, quindi, a suddividere la *strip* in più sequenze, organizzate secondo gli *storyboard* realizzati. Per suddividere una *strip*, bisogna dapprima selezionare il *frame* su cui deve avvenire la suddivisione, quindi cliccare con il tasto destro del mouse sulla *strip* da dividere e infine selezionare la voce "*Strip* → *Cut (soft) at frame*". Le sequenze così ottenute sono state disposte in modo alternato su due canali e con una parziale sovrapposizione verticale tra ciascuna coppia in modo da poter inserire una transizione "*Cross*" tra due *strip* consecutive, la cui durata dipende dalla percentuale di sovrapposizione tra le due sequenze. Per inserire una transizione di tipo *Cross*, bisogna selezionare le due sequenze da collegare e selezionare la voce *Add* → *Effect Strip ...* → *Cross*.

Alla fine del video è stata inserito un testo di ringraziamento, realizzato con un oggetto di tipo *Text* inserito in una scena di *Blender*. Al testo è stato assegnato un materiale di colore bianco; inoltre esso è stato posizionato su un *plane* nero per una sua maggiore visibilità. In tale scena è stata disposta una camera di tipo ortografica (*Orthografic*) che riprende il testo senza l'ap-

plicazione di effetti prospettici. Infine, la scena prodotta è stata caricata nel *Video Sequence Editor* cliccando su "*Add* → *Scene*" e posizionando la *strip* al termine dell'intervallo di *frame*.

Nei canali superiori è stata aggiunta infine la componente sonora del video, cliccando su "*Add* → *Sound*" e caricando i file audio. Al di sopra dei *frame* sono stati inseriti alcuni effetti sonori in corrispondenza della rotazione della chiave e apertura della porta, mentre nel canale superiore è stata inserita una musica di sottofondo per tutta la durata del video (Fig. 2.58).

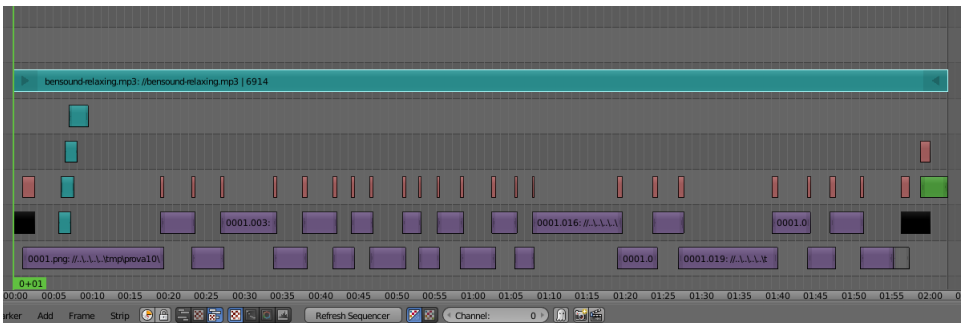


Figura 2.58: Montaggio del video

Terminato il montaggio, è possibile vedere un'anteprima del video realizzato cliccando sul pulsante "*Image Preview*" e facendo partire l'animazione con il tasto "*Play*" della *Timeline*. In alternativa, si può cliccare su "*Sequencer and Image Preview*" per visualizzare contemporaneamente l'anteprima del video e le sequenze di *strip* realizzate.

Similmente a quanto eseguito nella fase di *rendering*, bisogna ora definire i parametri per la codifica del file di output, contenuti nella scheda *Properties* → *Render*. In questo caso, gli unici campi da tenere in considerazione sono l'intervallo di *frame* (da 1 a 3134), la cartella di salvataggio del file, il colore di ciascun *frame* (RGB) e il formato del video. Per quest'ultimo parametro, si è scelto il valore "H.264" che applica una compressione *lossless* dei *frame*, producendo in output un unico file.

È possibile ora creare il filmato cliccando sul pulsante "*Animation*" nella stessa scheda.

Capitolo 3

Algoritmi

Questo capitolo contiene una descrizione "a basso livello" delle principali funzionalità di *Regard3D*, *MeshLab* e *Blender* utilizzate per la produzione della scena 3D. Si descrivono e analizzano, quindi, gli algoritmi di *computer vision* e *computer grafica* che implementano tali funzionalità. Il capitolo non vuole essere un manuale esaustivo per l'interpretazione del codice dei tre programmi, ma vuole fornire una descrizione schematica e generale per gli algoritmi implementati, rinviando il lettore alla lettura degli articoli presenti nella bibliografia per un'analisi più approfondita.

3.1 Estrazione dei keypoint

L'algoritmo utilizzato da *Regard3D* per individuare i *keypoint* di un'immagine è *A-KAZE* (*Accelerated KAZE*). Esso è la versione accelerata di *KAZE*, algoritmo sviluppato nel 2012 e di pubblico dominio. *Regard3D* mette a disposizione due versioni dell'algoritmo: la versione originale (*Classic A-KAZE*), la quale fornisce un grande numero di *keypoint* a un tempo di computazione elevato, e una versione più veloce (*Fast A-KAZE*) che, tuttavia, fornisce un numero di punti leggermente inferiore rispetto alla versione classica.

3.1.1 KAZE

KAZE [3] è un algoritmo di estrazione di *feature* 2D per immagini rappresentate a diversi livelli di scala (o risoluzione). L'idea alla base dei metodi multi-scala come *KAZE* è quella di costruire una rappresentazione spazio-scala di un'immagine, in cui quest'ultima viene filtrata con una funzione appropriata rispetto ad una variabile crescente (ad esempio, il tempo o la scala); con una rappresentazione multi-scala dell'immagine è possibile individuare e descrivere le *feature* a diversi livelli di risoluzione.

La rappresentazione spazio-scala più diffusa è quella lineare (gaussiana), che applica un filtro gaussiano rispetto alla deviazione standard crescente. Lo svantaggio delle rappresentazioni spazio-scala lineari è la riduzione della precisione all'aumentare del numero dei livelli di astrazione. Il *Gaussian blurring* (offuscamento) utilizzato in alcuni algoritmi di riconoscimento di oggetti, come *SIFT*, non rispettano i bordi naturali degli oggetti, in quanto smussa con lo stesso fattore sia i dettagli che il rumore di un'immagine, riducendo la precisione nella localizzazione dei *keypoint* e la loro distinguibilità.

A differenza di *SIFT*, *KAZE* esegue un *blurring* localmente adattivo all'immagine, in modo che solo il rumore sia smussato, lasciando invariati i dettagli e i bordi dell'immagine. Per ottenere questo risultato, *KAZE* costruisce una rappresentazione spazio-scala non lineare ed elabora le immagini secondo i seguenti passi [4]:

- Costruzione di una rappresentazione spazio-scala non lineare;
- Localizzazione dei *keypoint*;
- Calcolo dell'orientamento dominante.

La costruzione della rappresentazione spazio-scala sfrutta la funzione di conducibilità di Perona e Malik, la quale calcola il gradiente di una versione *gaussian blurred* dell'immagine originale ∇L_σ in funzione del tempo:

$$c(x, y, t) = g(|\nabla L_\sigma(x, y, t)|) \tag{3.1}$$

dove t è il tempo e L è l'immagine originale.

Per individuare i *keypoint*, *KAZE* applica un determinante di Hessian (DoH) a diversi valori di scala σ. I valori massimi corrispondono ai possibili *keypoint* di un'immagine, in quanto *DoH*, essendo uno metodo di *blob detection*, individua le regioni di un'immagine con proprietà di luminosità e colore diverse rispetto a quelle dell'ambiente.

$$L_{Hessian} = \sigma^2(L_{xx}L_{yy} - L_{xy}^2) \tag{3.2}$$

dove L_{xx} è la derivata seconda orizzontale, Lyy è la derivata seconda verticale, L_{xy} è la derivata seconda incrociata e σ è il valore di scala.

Per calcolare l'orientamento dominante di ciascun punto, si definisce un'area circolare di raggio $6\sigma_i$ al livello di scala σ_i, incentrata nella posizione del *keypoint*. Per ogni punto contenuto in tale area, si calcolano le derivate prime L_x e L_y che vengono pesate con una gaussiana centrata nel punto interessato e quindi rappresentate come punti in uno spazio vettoriale. Suddivisa l'area in segmenti circolari di 60° ciascuno, i vettori contenuti in ciascun segmento vengono sommati. L'orientamento del *keypoint* corrisponde al vettore somma più lungo.

3.1.2 A-KAZE

A-KAZE, a differenza di *KAZE*, usa un metodo più veloce per creare la rappresentazione spazio-scala non lineare, detto FED (*Fast Explicit Diffusion*). Il FED applica i filtri non lineari per iterazioni successive, utilizzando l'immagine originale nel primo ciclo, quindi l'immagine risultante nel ciclo successivo.

Di seguito è riportato lo schema generale dell'algoritmo utilizzato in *Regard3D*:

```
1 AKAZESemaLocker akazeLocker;
2
3 cv::Mat cvimg;
4 cv::eigen2cv(img.GetMat(), cvimg);
5
```

```
AKAZEOptions options;
options.omin = 0;
options.verbosity = false;
options.dthreshold = params.threshold_;
options.img_width = img.Width();
options.img_height = img.Height();

libAKAZE::AKAZE evolution(options);
evolution.Create_Nonlinear_Scale_Space(cvimg);
evolution.Feature_Detection(vec_keypoints);

for(int i = 0; i < static_cast<int>(vec_keypoints.size()); i++)
{
  evolution.Compute_Main_Orientation(vec_keypoints[i]);
  (vec_keypoints[i].angle) *= 180.0 / CV_PI;
  vec_keypoints[i].angle += 90.0f;
  while(vec_keypoints[i].angle < 0)
    vec_keypoints[i].angle += 360.0f;
  while(vec_keypoints[i].angle > 360.0f)
    vec_keypoints[i].angle -= 360.0f;
}
```

L'algoritmo dapprima utilizza un semaforo per ammettere un solo *thread* in tale sezione del codice. Quindi, l'immagine viene convertita in matrice per poter essere manipolata dai metodi della libreria *OpenCV*. Creata la rappresentazione spazio-scala, si estraggono le *feature* e si calcola l'orientamento di ciascun *keypoint*, convertendo infine gli angoli da radianti in gradi.

3.2 Descrizione dei keypoint

Una volta calcolati i punti chiave delle immagini di input, essi devono essere descritti in un formalismo matematico per poter essere manipolati dai successivi algoritmi di *matching* e di triangolazione. Il formalismo utilizzato da *Regard3D* è *LIOP*.

3.2.1 LIOP

LIOP (*Local Intensity Order Pattern*) [5] è il metodo utilizzato da *Regard3D* per descrivere matematicamente le *feature* estratte dall'algoritmo *A-*

KAZE. Questo descrittore si basa sul concetto di "*local order pattern*", ossia l'ordine ottenuto ordinando i *keypoint* per intensità crescente. Formalmente, dato un pixel x e n pixel vicini $x_1, x_2, \ldots, x_n$, il pattern di ordine locale in x è la permutazione σ che ordina i pixel vicini per intensità crescente: $I(x_{\sigma(1)}) \leq I(x_{\sigma(2)}) \leq \cdots \leq I(x_{\sigma(n)})$.

LIOP è un descrittore locale che codifica l'informazione di una piccola regione di un'immagine (*patch*). Calcolando l'informazione di intensità generale del *patch*, è possibile suddividere la regione da analizzare in sottoregioni che vengono utilizzate per calcolare i rispettivi descrittori. In questo modo, *LIOP* sfrutta l'informazione di intensità locale e generale per ottenere un descrittore altamente discriminante.

Oltre ad essere un descrittore fortemente distintivo, *LIOP* è inoltre invariante ai cambianti monotoni di intensità e alle rotazioni dell'immagine. Affinché i pattern siano invarianti rispetto alla rotazione, i punti $x_1, \ldots, x_n$ vicini a x devono essere presi in modo covariante rispetto alla rotazione, in particolare individuando in senso antiorario i punti all'interno di un cerchio di raggio r intorno al punto x.

Calcolati i pattern per ogni pixel x, essi vengono utilizzati per costruire un istogramma che descrive la distribuzione di intensità nella regione che circonda x. Suddividendo il *patch* in un certo numero di regioni $R_1, \ldots, R_m$ e calcolandone i rispettivi istogrammi, si può calcolare il descrittore *LIOP* dell'intero *patch* combinando gli istogrammi ottenuti. Le regioni, tuttavia, devono essere selezionate in modo che il descrittore sia invariante ai cambiamenti monotoni di intensità. Ciò può essere ottenuto suddividendo il *patch* in modo che le singole regioni contengano lo stesso numero pixel: se m è il numero di regioni da generare, basta ordinare i pixel per intensità crescente e partizionare la lista in m parti uguali (eventualmente inserendo i pixel restanti nell'ultima partizione).

Per calcolare l'istogramma dei pattern calcolati, le permutazioni devono essere mappate nelle classi dell'istogramma, ordinando le permutazioni in ordine lessicografico. Ad esempio, considerando $n = 4$ vicini per ciascun

pixel, vi sono $n! = 24$ permutazioni possibili, ad ognuna delle quali si può associare un numero ordinale progressivo (Tab. 3.1)

σ	Ind(σ)
1 2 3 4	1
1 2 4 3	2
1 3 2 4	3
1 3 4 2	4
1 4 2 3	5
⋮	⋮
4 3 1 2	23
4 3 2 1	24

Tabella 3.1: *Index table* per $n = 4$

Tramite la *index table*, la permutazione σ può essere mappata in un vettore $n!$-dimensionale v, i cui elementi sono tutti 0, eccetto l'elemento $Ind(\sigma)$ che è pari a 1. Chiamiamo tale vettore $LIOP(x)$, ossia il descrittore *LIOP* per la regione incentrata nel punto x. Questo vettore n!-dimensionale corrisponde ad una delle $n!$ classi che costituiscono l'istogramma dell'intensità del *patch*. Il descrittore *LIOP* dell'intero *patch* è calcolato accumulando i $LIOP(x)$ di ciascun punto nelle corrispondenti classi dell'istogramma e concatenandoli insieme. L'aggregazione dei pattern assegna un peso a ciascun pattern che ne descrive la sua stabilità, la quale misura in modo proporzionale il numero di coppie di pixel del vicinato con differenza di intensità maggiore di una certa soglia Θ:

$$w(x) = \sum_{i,j} [|I(x_i) - I(x_j)| > \Theta) \tag{3.3}$$

Il descrittore *LIOP* dell'intero *patch* è dato dalla concatenazione degli m descrittori $LIOP(x)$, ognuno con peso $w(x)$ e dimensione $n!$:

$$LIOP = (des_1, des_2, \ldots, des_m) \tag{3.4}$$

$$des_i = \sum_{x \in R_i} w(x) LIOP(x) \tag{3.5}$$

Il vettore ottenuto ha dimensione $mn!$, dove m è il numero di classi dell'istogramma e n è il numero dei vicini per ciascun punto.

3.3 Matching dei keypoint

Tutti gli algoritmi di *keypoint matching* presenti in *Regard3D* (*FLANN*, *KGraph* e *MRPT*) sono algoritmi che ricercano *nearest neighbors* (vicini più vicini) dei *keypoint* precedentemente estratti, al fine di individuare delle corrispondenze tra le *feature* calcolate dalle diverse immagini. Una ricerca *nearest neighbors* può essere divisa in due fasi separati: una fase offline e una online. In quella offline si costruisce un indice (ad esempio, una tabella, un grafo o un albero), mentre in quella online si eseguono query veloci sull'indice costruito.

Definiamo un problema k-NN (*k-nearest neighbors*) come un problema di ricerca di k punti più vicini ad un punto q in un *dataset* X secondo una metrica di distanza m. Formalmente, k-NN individua un sottoinsieme $K \subseteq X$ tale che $|K| = k$ e $m(q, x) \leq m(q, y) \quad \forall x \in K, y \in X \setminus K$.

In particolare, *FLANN* e *MRPT* sono algoritmi di ricerca di *nearest neighbors* approssimati. Essi ammettono, infatti, una certa probabilità di errore nel calcolo dei k-NN di un punto particolare (*query point*), ma sono molto più efficienti degli algoritmi di ricerca dei k-NN esatti, i quali tendono ad avere prestazioni non migliori di un algoritmo a forza bruta se eseguiti su insiemi di dati multidimensionali. La bontà di un algoritmo di ricerca approssimata può essere misurata con una misura di *accuracy*; ad esempio, la metrica *recall* è definita come la proporzione dei veri k-NN di un punto restituiti dall'algoritmo:

$$Recall = \frac{A \cap K}{k} \tag{3.6}$$

dove A è l'insieme di k-NN approssimati calcolati dall'algoritmo e K è l'insieme dei veri k-NN di un punto.

3.3.1 FLANN

FLANN (*Fast Library for Approximate Nearest Neighbors*) è una libreria *open source* per la ricerca di *nearest neighbors* approssimati, che è stata incorporata in *OpenCV* ed è attualmente una delle più popolari librerie per il *nearest neighbor matching*. Essa fornisce algoritmi efficienti per l'indicizzazione e ricerca di *nearest neighbors* approssimati, come anche un metodo di scelta automatica del miglior algoritmo di ricerca da utilizzare in base alla struttura dei dati in input e performance desiderata.

Gli algoritmi di *feature matching* presenti nella libreria *FLANN* indicizzano i dati di input creando delle strutture ad albero, sulle quali vengono quindi eseguite le query di ricerca. La performance di questi algoritmi dipende da molti fattori, come la dimensionalità dei dati, la grandezza e la struttura dei dati, la precisione di ricerca desiderata e un insieme di parametri tipici di ogni algoritmo, come ad esempio il fattore di *branching* (ramificazione) o il numero totale di alberi da creare. *FLANN* propone, quindi, una tecnica per la scelta automatica dell'algoritmo più performante per un particolare tipo di dati in input.

L'idea è quella di considerare gli algoritmi di ricerca come parametri θ di una generica funzione A. L'obiettivo è, pertanto, quello di individuare i parametri $\theta \in \Theta$ che permettono di ottenere la soluzione migliore. La performance di A configurata con i parametri θ sui dati di input viene misurata da una funzione costo c che combina il tempo di ricerca, il tempo di indicizzazione (ossia la costruzione degli alberi di ricerca) e l'occupazione di memoria. Per individuare il miglior algoritmo, bisogna risolvere il seguente problema di ottimizzazione in uno spazio di parametri Θ:

$$\min_{\theta \in \Theta} c(\theta) \qquad c : \Theta \to \mathbb{R} \tag{3.7}$$

Definiamo $c(\theta)$ come:

$$c(\theta) = \frac{s(\theta) + w_b b(\theta)}{\min_{\theta \in \Theta}(s(\theta) + w_b b(\theta))} + w_m m(\theta) \tag{3.8}$$

dove $s(\theta)$, $b(\theta)$ e $m(\theta)$ rappresentano rispettivamente il tempo di ricerca, il tempo di costruzione degli alberi e l'occupazione di memoria. I pesi w_b e w_m permettono di definire l'importanza relativa del tempo di costruzione e dell'occupazione di memoria sul costo totale. Il processo di ottimizzazione consta di due passi:

- Un'esplorazione globale dello spazio dei parametri;
- Un'ottimizzazione locale a partire dalla soluzione migliore trovata nello step precedente.

Di seguito sono presentati in breve due algoritmi di *feature matching* presenti nella libreria *FLANN*.

K-D Tree Randomized

Un *k-d tree* (*k-dimensional tree*) [6] è un albero binario che consente di trovare velocemente *nearest neighbors* approssimati di dati definiti in k dimensioni.

L'algoritmo opera partizionando ricorsivamente l'insieme di input in base al valore mediano di un certo attributo (dimensione) dei dati. La costruzione dell'albero avviene nel seguente modo: la radice dell'albero corrisponde all'intero *dataset*; quindi i dati vengono suddivisi in due parti da un piano ortogonale alla dimensione in cui i dati hanno la varianza maggiore. Il piano separa i dati in base ad un certa soglia, che corrisponde alla mediana (valore centrale) dell'insieme da partizionare. L'algoritmo procede per passi successivi, individuando ogni volta la dimensione nella quale i dati hanno la varianza maggiore e suddividendo i dati in due sottoinsiemi, producendo così un albero binario. In figura 3.1 è mostrata un'applicazione dell'algoritmo descritto.

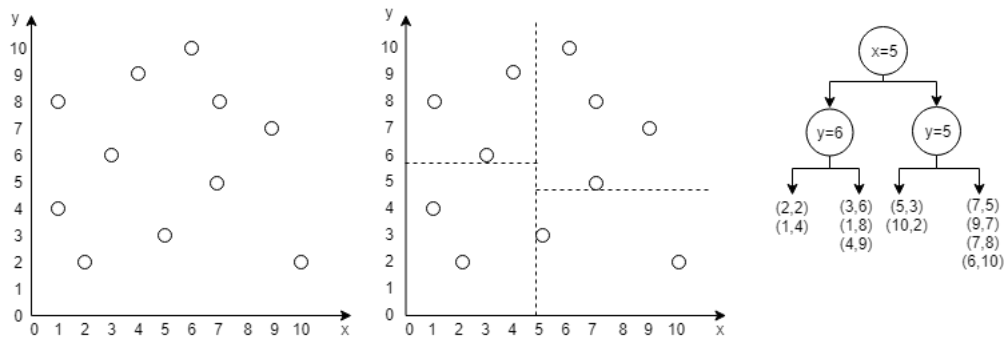


Figura 3.1: Un esempio di applicazione dell'algoritmo k-d tree con $k = 2$. I dati vengono suddivisi lungo i valori $x = 5$, $y = 6$ e $y = 5$, che corrispondono ai valori mediani dei corrispondenti insiemi di dati. A destra è mostrato l'albero k-d risultante.

Dovendo ricercare i vicini di un certo dato, si individua la foglia dell'albero k-d contenente il dato e si calcola la distanza tra quel valore e gli altri punti contenuti in quella foglia. La ricerca, essendo approssimata, potrebbe non restituire i reali vicini di un punto, che potrebbero trovarsi in un'altra foglia dell'albero.

La versione randomizzata dell'algoritmo [7], implementata in *FLANN*, costruisce molteplici alberi k-d sui quali la ricerca viene eseguita in parallelo. Gli alberi vengono costruiti in modo simile ai classici *k-d tree* con la differenza che, invece di considerare a ogni step la dimensione con la varianza maggiore, la dimensione di suddivisione viene scelta ogni volta casualmente tra le prime n dimensioni con la varianza maggiore. Le decomposizioni multiple casuali aumentano la probabilità che un *query point* e i suoi vicini siano in una stessa partizione. Se, infatti, il *query point* è vicino ad uno dei piani di suddivisione, i suoi vicini possono trovarsi con equa probabilità in entrambi i lati del piano e se si trovano dal lato opposto rispetto al *query point* è necessario visitare nuovamente l'albero per poterli individuare.

La ricerca dei *nearest neighbors* di un punto in una foresta k-d randomizzata sfrutta una coda con priorità contenente i nodi non esplorati, ordinata per distanza crescente di ciascun nodo dal piano di decisione. La ricerca

esplorerà, quindi, prima le foglie più vicine di tutti gli alberi. Una volta esaminato un punto, esso viene marcato per non essere riesaminato negli altri alberi.

Priority Search K-Means Tree

Questo algoritmo [7] [8] suddivide i dati eseguendo il *clustering* dei punti, usando come metrica di distanza la somma delle distanze su tutte le dimensioni, a differenza dell'algoritmo *k-d tree* che invece partiziona i dati considerando una dimensione alla volta.

Il processo di costruzione dell'albero inizia dividendo i dati di input in K regioni distinte (*cluster*) tramite l'esecuzione dell'algoritmo *k-means*[1] e ricorsivamente ogni *cluster* viene ripetutamente suddiviso finché ogni regione ha un numero di punti minore di K, detto fattore di *branching* (ramificazione).

Per individuare i *nearest neighbors* di un punto si parte dalla radice dell'albero e si prosegue di nodo in nodo fino a raggiungere il nodo foglia più vicino. Il criterio di scelta che guida la navigazione nell'albero è percorrere il ramo che porta ad un *cluster* il cui centro è più vicino al *query point* rispetto agli altri cluster raggiungibili dagli altri rami; i percorsi non esplorati vengono inseriti in una coda con priorità, ordinata per distanza crescente del *cluster* dal *query point*. L'algoritmo, quindi, procede l'esplorazione estraendo il ramo sulla cima della coda con priorità e proseguendo da esso la ricerca.

Selezionando i centri dei *cluster* come punti casuali, è possibile rieseguire l'algoritmo più volte per ottenere diversi alberi e quindi diverse partizioni dei dati. Una ricerca eseguita in parallelo su tali alberi migliora significativamente la performance dell'algoritmo.

[1] Il *k-means* è un algoritmo di *clustering* partitivo che suddivide i dati in K gruppi. Definiti K punti iniziali (centroidi), iterativamente si assegnano ai centroidi i punti corrispondenti più vicini, quindi ogni centroide è ricalcolato come la media dei punti ad esso assegnati.

3.3.2 KGraph

KGraph [9] [10] è una libreria che sfrutta la costruzione di grafi k-NN (*k-NNG*, o *k-Nearest Neighbor Graph*) come indici sui quali eseguire le ricerche di *nearest neighbors* di un particolare punto rappresentato come un nodo del grafo.

Un k-NNG di un certo insieme di dati V è un grafo orientato (V, E) dove ogni dato $v \in V$ è connesso con i K nodi più simili, rispetto ad una certa misura di similarità. Formalmente, sia V un insieme di dati di dimensione $N = |V|$ e sia $\sigma : V \times V \to \mathbb{R}$ una misura di similarità. Per ogni $v \in V$, si possono definire i seguenti insiemi:

- $B_K(v)$: sono i k-NN diretti di v, ossia i k oggetti di V (escluso v) più simili a v;
- $R_K(v)$: sono i k-NN inversi di v, ossia i punti di V per i quali v è un k-NN, cioè $R_K(v) = \{u \in V | v \in B_K(u)\}$;
- $B[v]$ e $R[v]$: sono le approssimazioni di $B_K(v)$ e $R_K(v)$ insieme con i valori di similarità;
- $\bar{B}[v]$: è l'insieme dei vicini generici di v, cioè $\bar{B}[v] = B[v] \cup R[v]$.

La costruzione di un grafo k-NN esatto è costosa e l'algoritmo a forza bruta che lo produce ha complessità $O(n^2)$, rendendo difficile una sua applicazione per *dataset* grandi. L'euristica sulla quale si basa *KGraph* per la costruzione approssimata di un grafo k-NN si basa su un semplice principio di *small world*, secondo il quale "il vicino di un vicino è probabilmente un vicino". In altre parole, data un'approssimazione del grafo contenente k nodi scelti casualmente per ogni dato, si può iterativamente migliorare la soluzione confrontando ogni punto con i vicini dei suoi vicini, considerando sia i k-NN diretti che quelli inversi. L'algoritmo termina quando il grafo non può più essere migliorato. Vi sono una serie di possibili migliorie [10] per velocizzare l'indicizzazione e ridurre le computazioni ridondanti:

- *Join locale*: dato un punto v e i suoi vicini $\bar{B}[v]$, un join locale in $\bar{B}[v]$ è l'operazione con cui si calcolano le similarità tra ciascuna coppia di punti $p, q \in \bar{B}[v]$, aggiornando quindi $B[p]$ e $B[q]$ con i valori di similarità calcolati;

- *Ricerca incrementale*: si utilizzano flag booleani per evitare di riconsiderare coppie di dati già confrontate nelle iterazioni precedenti. Si considerano solo coppie in cui almeno uno dei dati non è stato mai considerato. Quando un oggetto viene confrontato, il suo flag viene marcato in modo tale che esso non venga riconsiderato nelle iterazioni future;

- *Campionamento*: per ridurre la complessità dei *join* locali, si può utilizzare una strategia di *sampling* per costruire le liste di dati k-NN dirette e inverse composte da ρK elementi, invece che K, dove ρ è un valore di *sampling* compreso tra 0 e 1;

- *Terminazione anticipata*: per evitare di eseguire le ultime iterazioni dell'algoritmo le quali portano a miglioramenti trascurabili, si può far terminare l'algoritmo quando l'iterazione successiva non porta a miglioramenti significativi dell'*accuracy* di ricerca, ossia il numero di modifiche eseguite nel grafo è minore di una certa tolleranza.

Una volta costruito il k-NNG approssimato, la ricerca dei *nearest neighbors* di un *query point* q avviene tramite una procedura di ricerca *greedy*: a partire da n dati scelti casualmente, si mantiene una lista di nodi in cui vengono salvati i k nodi migliori ordinati per distanza crescente da q. Ricorsivamente si calcolano le distanze tra q e i dati non esplorati presenti nella lista. Quest'ultima viene aggiornata con i vicini più vicini del punto q. La ricerca termina quando ogni punto della lista è più vicino a q rispetto ai suoi vicini.

3.3.3 MRPT

MRPT (*Multiple Random Projection Trees*) [11] è una libreria per la ricerca di *nearest neighbors* approssimati. Esso utilizza alberi di proiezione per suddividere gerarchicamente i dati e quindi individua velocemente i *nearest neighbors* di un punto, analizzando le strutture ad albero create. Come i precedenti algoritmi, anche *MRPT* opera in due fasi:

- Fase offline: indicizzazione dei dati con una collezione di alberi di proiezione casuali;
- Fase online: calcolo dei *nearest neighbors* eseguendo query sulla struttura indicizzata.

Inizialmente si dispongono i dati come punti in uno spazio d-dimensionale e li si proietta su un vettore casuale r d-dimensionale; il vettore r costituisce la radice dell'albero di proiezione. I dati vengono quindi suddivisi in due nodi figli usando la media dei punti proiettati su r: i punti con valori proiettati minori o uguali alla media sono spostati nel figlio sinistro, mentre quelli con valore maggiore della media sono spostati nel figlio destro. L'algoritmo procede ricorsivamente per ciascun sottoinsieme di dati in entrambi i nodi figli, fino a raggiungere una certa profondità massima l predefinita o un numero massimo di punti per ciascuna foglia dell'albero (Fig. 3.2).

Nel calcolare i *nearest neighbors* di un punto, *MRTP* considera solo i punti che appartengono alla stessa partizione (ossia la stessa foglia dell'albero) del *query point* e ne restituisce quelli più vicini. Essendo un algoritmo di ricerca approssimata, vi sono casi in cui un *nearest neighbors* reale non viene individuato, in quanto ricade in un'altra partizione. Rieseguendo l'algoritmo, i vettori di proiezione casuali portano a diverse suddivisioni dello spazio. Pertanto, costruire più alberi con la stessa profondità o dimensione massima per le foglie può migliorare significativamente l'*accuracy* dei risultati. D'altro canto, il tempo di esecuzione di *MRPT* ha dipendenza lineare nel numero di alberi utilizzati. Il tempo di ciascuna query consta di due parti: il tempo per

attraversare ogni albero e raggiungere il *query point* e il tempo per eseguire una ricerca lineare dei k-NN nelle foglie contenenti il punto.

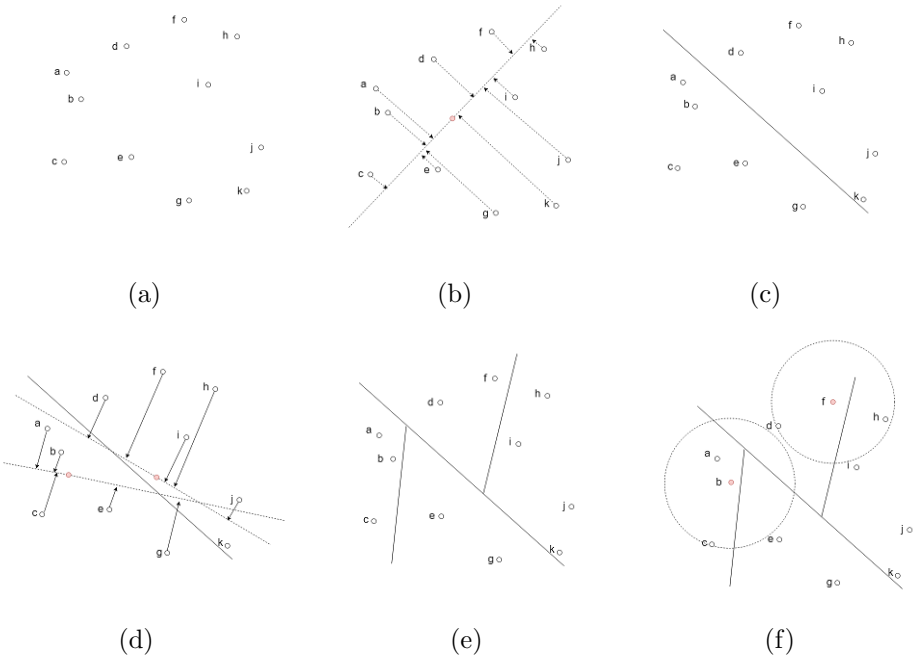


Figura 3.2: Esempio di esecuzione dell'algoritmo $RMPT$ per una ricerca 2-nn con profondità massima $l = 2$. (a) *Dataset* iniziale. (b) Creazione vettore casuale e proiezione dei dati. La media è rappresentata dal cerchio rosso. (c) I dati vengono suddivisi. (d) Creazione di due vettori casuali e proiezione dei dati. (e) I dati vengono suddivisi. (f) Esecuzione di due query, una su f e una su b. Come si può vedere, la ricerca non dà sempre il risultato corretto. La query in b fornisce correttamente a e c, ma la query in f fornisce d ma non h, il quale fa parte di un'altra partizione.

Nel caso di *dataset* definiti in molte dimensioni, il calcolo delle proiezioni casuali è lento e richiede un alto quantitativo di memoria per salvare i vettori casuali e gli alberi di proiezione. Per rendere l'algoritmo più efficiente, vi sono due possibili miglioramenti:

- Non utilizzare vettore casuali d-dimensionali densi, ma sparsi;

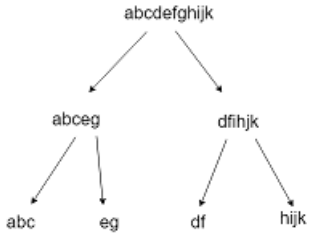


Figura 3.3: Albero creato dall'esecuzione di *MRPT* mostrata in figura 3.2

- Non creare un vettore casuale diverso per ogni nodo, ma riutilizzare lo stesso vettore per tutti i nodi di uno stesso livello dell'albero.

3.4 Feature tracking

Gli algoritmi di *feature tracking* prendono in input corrispondenze binarie di *feature* per produrre in output tracce coerenti tra più immagini (*multi-view tracks*). L'algoritmo utilizzato da *Regard3D* è detto *Union-Find*.

3.4.1 Union-Find

L'algoritmo *Union-Find* [12] è in grado di risolvere un problema di *feature tracking* in modo semplice, veloce ed efficiente: si crea un *singleton* (insieme con un solo elemento) per ogni *feature*, quindi per ogni corrispondenza binaria si effettua l'unione (*Union*) dei due insiemi che contengono le due *feature*. Consideriamo un grafo $G = (V, E)$ dove i vertici $v \in V$ sono insiemi singleton di *feature*, mentre gli archi $e \in E$ indicano le corrispondenze tra due *feature*. Le tracce che l'algoritmo restituisce in output sono le componenti connesse del grafo. In figura 3.4 è mostrato un esempio di esecuzione dell'algoritmo.

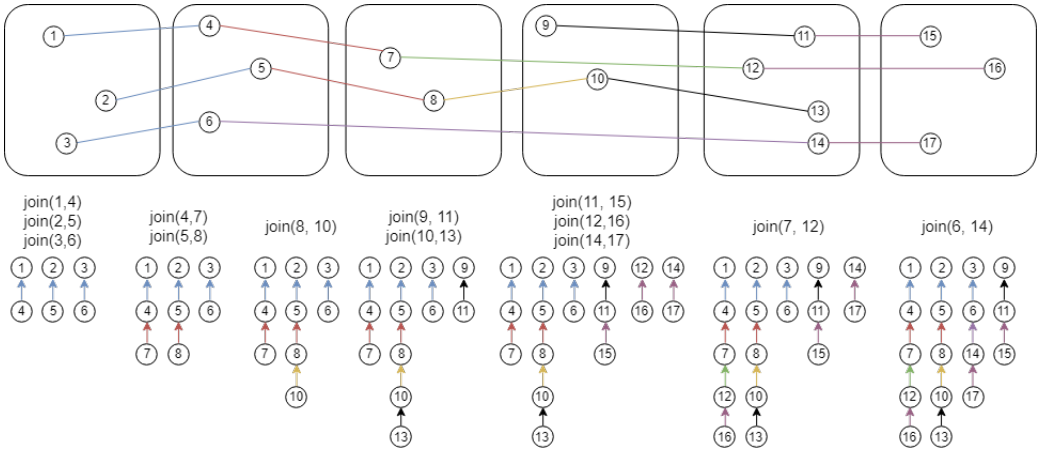


Figura 3.4: Esempio di esecuzione dell'algoritmo *Union-Find*. Le tracce calcolate sono [1,4,7,12,16], [2,5,8,10,13], [3,6,14,17] e [9,11,15].

3.5 Triangolazione

Gli algoritmi di triangolazione [13], o di *Structure From Motion*, prendono in input immagini *multi-view stereo* (ossia immagini prese da punti di vista differenti) e informazioni sulla calibrazione della camera per produrre in output una nuvola di punti 3D e orientamenti/pose della camera in un comune sistema di coordinate 3D. Gli algoritmi di *SFM* possono essere suddivisi in due principali categorie:

- *Algoritmi incrementali o sequenziali*: iniziano da un coppia di immagini e ripetutamente aggiungono immagini per determinare in modo incrementale la posizione 3D dei *keypoint*;
- *Algoritmi globali*: utilizzano tutte le immagini in un unico step.

3.5.1 Incremental Structure From Motion

Gli algoritmi incrementali [13] creano una nuvola 3D sparsa partendo da una coppia di immagini iniziale (quella con maggior numero di abbinamenti

tra le *feature* calcolate) e iterativamente migliorando il modello considerando in ogni passo un'immagine in più.

I punti estratti da ciascuna immagine vengono abbinati a punti 3D del modello, permettendo di stimare la posizione e orientamento della camera in tale scatto. Ogni volta che viene utilizzata una nuova immagine per aggiungere punti 3D al modello, si esegue una procedura di ottimizzazione globale detta *bundle adjustment* per minimizzare l'errore accumulato in ogni passo.

Un algoritmo di *SFM* incrementale è *RANSAC* (*RANdom SAmple Consensus*) [14]. *RANSAC* è una procedura randomizzata che sceglie ripetutamente insiemi di dati casuali, la cui dimensione è sufficiente per stimare i parametri del modello. L'algoritmo iterativamente individua i parametri che massimizzano il numero di *inliers* (punti interni del modello) all'interno di una certa soglia di errore T, definita a priori. Tuttavia, se T è troppo piccola, il modello risulta impreciso perché composto da pochi *inliers*, mentre se T è troppo grande, il modello viene contaminato dagli *outliers*.

Regard3D utilizza l'implementazione *OpenMVG* di *AC-RANSAC* (*A Contrario RANSAC*). Mentre *RANSAC* utilizza soglie di errore globali e definite empiricamente per la scelta del modello, *AC-RANSAC* adatta le soglie di errore ai dati di input, rendendo l'algoritmo versatile per diverse tipologie di dati. Queste soglie vengono calcolate in modo da allontanare correttamente gli *inliers* e gli *outliers*.

3.5.2 Global Structure From Motion

Gli algoritmi di *Structure From Motion* globali [15] considerano tutte le coppie di immagini con *feature* abbinate in un unico step con un'unica fase finale di *bundle adjustment* per ottimizzare i parametri di ricostruzione. Si possono distinguere due principali categorie di questi algoritmi:

- I metodi *rotation averaging* calcolano simultaneamente tutti gli orientamenti della camera dalle rotazioni tra ciascuna coppia di immagini di input;

- I metodi *translation averaging* calcolano le posizioni della camera, generalmente quando gli orientamenti sono già stati calcolati precedentemente.

Regard3D offre quattro possibili algoritmi globali di triangolazione, due per ogni categoria. In generale, gli algoritmi globali sono più veloci di quelli incrementali, ma richiedono un numero maggiore di immagini di input per ottenere un buona ricostruzione. D'altro canto, gli algoritmi incrementali hanno lo svantaggio di dover individuare una buona coppia di immagini di partenza. Tutti gli algoritmi di *SFM* globali seguono in genere la medesima struttura:

- Trovare la matrice di rotazione R_i che mappa le coordinate del mondo nelle coordinate di ciascuna camera i;
- Calcolare la traslazione relativa t^l_{ij} tra le camere i e j nel sistema locale di coordinate;
- Data una stima per R_i, calcolare $t_{ij} = R_i t^l_{ij}$, cioè la stessa traslazione nel sistema di coordinate del mondo;
- Applicare un algoritmo di ottimizzazione, insieme ad una *loss function*, come step finale di *bundle adjustment*.

3.6 Densificazione

Gli algoritmi di *densification*, o *multi view stereo* (*MVS*), prendono in input immagini con pose e la corrispondente nuvola di punti sparsa e producono modelli densi 3D. In seguito sono descritti due algoritmi *MVS* presenti nella suite di algoritmi di *Regard3D*.

3.6.1 CMVS/PMVS

CMVS/PMVS è una coppia di algoritmi che consentono di aumentare la densità di una nuvola di punti sparsa. Da un lato *CMVS* (*Clustering Views*

for Multi-View Stereo) suddivide le immagini in gruppi (*cluster*) gestibili da un algoritmo *MVS*, dall'altro *PMVS* (*Patch-based Multi-View Stereo*) sfrutta i *cluster* generati da *CMVS* per costruire un insieme di *patch* rettangolari densi che ricoprono le superfici visibili nelle immagini di input. Mentre altri algoritmi *MVS* usano tutte le immagini simultaneamente per costruire un modello 3D, rendendo inapplicabile una ricostruzione quando l'insieme dei dati di input è molto grande, *CMVS/PMVS* effettuano una selezione e *clustering* delle immagini e ogni gruppo viene gestito in modo indipendente, rendendo questa tecnica da un lato parallelizzabile, dall'altro scalabile verso insiemi massicci di foto.

CMVS

Assumiamo che le immagini di input $\{I_i\}$ siano state elaborate da un algoritmo *SFM* per la generazione di una nuvola sparsa di punti $\{P_j\}$, ognuno dei quali è visibile in un insieme di immagini $\{V_j\}$. L'algoritmo *CMVS* individua un certo numero di *cluster* $\{C_k\}$ di immagini con sovrapposizione parziale tali che valgono le seguenti tre proprietà [16]:

- *compattezza*: le immagini ridondanti devono essere escluse dalla computazione. Formalmente, si vuole minimizzare $\sum_k |C_k|$;
- *dimensione*: ogni *cluster* deve avere una dimensione gestibile da un algoritmo MVS. Formalmente:

$$\forall k, |C_k| \leq \alpha \tag{3.9}$$

  dove α è una costante che definisce le risorse computazionali e di memoria;

- *copertura*: la ricostruzione con *cluster* deve portare a perdite di dettaglio minime rispetto al risultato ottenibile dall'elaborazione dell'intero insieme di immagini. Formalmente:

$$\forall i \frac{\{\#\text{ di punti coperti in } I_i\}}{\{\#\text{ di punti in } I_i\}} \geq \delta \tag{3.10}$$

cioè la quantità di punti coperti (ossia ricostruibili in almeno un *cluster*) rispetto al numero totale di punti deve essere maggiore o uguale a una certa costante δ.

L'algoritmo esegue 4 passi, dei quali gli ultimi due vengono ripetuti iterativamente finché le proprietà precedentemente dette non sono rispettate:

1. *Fusione di punti*: partendo da un insieme di punti, casualmente si fonde un punto con i suoi vicini, eliminando il punto e i vicini dall'*input set*, finché quest'ultimo non è vuoto. Questa fase riduce significativamente la dimensione dell'*input set* e il tempo di esecuzione dei tre passi successivi;

2. *Rimozione di immagini ridondanti*: si verifica per ogni immagine se il vincolo di copertura vale ancora se l'immagine viene scartata; in caso affermativo, essa viene eliminata dall'insieme delle immagini di input. Le immagini sono ordinate per risoluzione crescente, in modo che quelle a bassa qualità vengano eliminate per prime;

3. *Divisione dei cluster*: Ogni *cluster* che viola il vincolo di dimensione viene suddiviso in componenti più piccole. Si rappresenta un *cluster* come un grafo, dove i nodi sono le immagini, mentre gli archi misurano con i loro pesi il contributo di ciascuna coppia di immagini per la ricostruzione *MVS* di alcuni punti. Quest'operazione elimina quindi gli archi con pesi più bassi di una certa soglia;

4. *Inserimento di immagini*: si aggiungono immagini in ciascun *cluster* per garantire il vincolo di copertura, che può essere stato violato nella fase precedente. Si costruisce una lista di azioni di inserimento di immagini, ognuna delle quali misura l'efficacia nell'inserire un'immagine

in un *cluster* per incrementare la copertura. Le azioni sono quindi ordinate in ordine decrescente di efficacia ed eseguite finché il vincolo di copertura non è soddisfatto.

PMVS

Calcolati i *cluster*, l'algoritmo *PMVS* [17] ricostruisce i punti 3D per ciascun *cluster* indipendentemente. Esso sfrutta i *cluster* prodotti da *CMVS* per costruire *patch* rettangolari coi quali aumentare la densità della nuvola di punti. Definiamo un *patch* p come un rettangolo con centro $c(p)$ e normale $n(p)$ orientata verso la camera che lo osserva. Per ogni *patch* si può definire:

- un'immagine di riferimento $R(p)$;
- un insieme di immagini $S(p)$ dove p dovrebbe essere visibile ma può non esserlo in pratica a causa di *motion blur*, *highlights* o ostacoli in movimento;
- un insieme di immagini $T(p)$ in cui il *patch* è individuato ($R(p) \in T(p)$).

Per ogni immagine, invece, si definisce una griglia regolare di celle $C_{i,j}$, in ognuna delle quali l'algoritmo cerca di ricostruire almeno un *patch*, adottando una procedura *match-expand-and-filter*:

- *Matching*: i *keypoint* calcolati vengono collegati (*matched*) tra le diverse immagini tramite gli operatori *Harris* e *Difference-of-Gaussians* (*DoG*), producendo un insieme sparso di *patch*;
- *Expansion*: una tecnica di espansione aumenta il numero di pixel in ogni regione, portando a un insieme denso di *patch*;
- *Filtering*: i filtri di qualità e visibilità eliminano gli abbinamenti errati che non sarebbero visibili nella *mesh* finale.

Nella prima fase, si utilizzano gli operatori *Harris* e *DoG* per individuare gli angoli e i *blob* in ciascuna immagine. Per ogni cella di un'immagine,

si calcolano gli n massimi locali dei due operatori con le risposte maggiori. Individuate queste *feature*, esse vengono collegate in modo da costruire un insieme sparso di *patch*.

Nella fase di espansione, iterativamente si aggiungono nuovi vicini ai *patch* esistenti finché tutte le superfici del modello non sono coperte. Intuitivamente, due *patch* p e p' sono considerati vicini se appartengono a due celle adiacenti $C(i, j)$ e $C(i', j')$ della stessa immagine. Il *patch* p' viene inizializzato con i valori di p, quindi $R(p') = R(p)$, $T(p') = T(p)$ e $n(p') = n(p)$. Il centro $c(p')$ corrisponde al punto in cui il raggio di vista che passa attraverso il centro di $C(i', j')$, cioè la cella a cui appartiene p', si interseca con il piano contenente il *patch* p. Il centro e la normale di p' sono raffinate tramite una procedura di ottimizzazione, quindi si calcolano gli insiemi $S(p')$ e $T(p')$. Il *patch* p' viene accettato e inserito nella nuvola densa di punti se $|T(p')| \geq \gamma$, cioè se il *patch* viene individuato in almeno γ immagini.

Nell'ultima fase, vengono applicati due filtri: un filtro di qualità e uno di visibilità [16]:

- Il filtro di qualità filtra i *cluster* più distanti dal modello che hanno una densità di punti minore rispetto ai cluster più vicini. Il filtro calcola per ogni *cluster* un istogramma $\{H_l\}$ dove H_l è la somma delle precisioni di ricostruzione dei punti contenuti. Vengono quindi filtrati i punti i cui valori nell'istogramma sono minori della metà del valore massimo;

- Il filtro di visibilità calcola quante volte la ricostruzione di un punto da parte di un *cluster* entra in conflitto con le ricostruzioni di altri *cluster*. Il punto viene quindi scartato se tale numero è maggiore di una certa tolleranza.

3.6.2 MVE

MVE (*Multi-View Environment*) [18] è un algoritmo *MVS* in grado di costruire una nuvola di punti densa a partire da un insieme di immagini con diversi valori di luminosità e scala. L'idea alla base dell'algoritmo è una

scelta oculata delle immagini da utilizzare per lo *stereo matching*, eseguendo una selezione delle viste (*view selection*) a due livelli:

- Una selezione globale eseguita a livello di immagine;
- Una selezione locale eseguita a livello di pixel.

La selezione globale avviene identificando per ogni immagine (o vista) di riferimento I un insieme di immagini vicine da utilizzare per lo *stereo matching*. In questa fase si utilizzano le *feature* condivise come indicatore di compatibilità tra due viste, poiché immagini con molte *feature* condivise generalmente coprono una porzione simile della scena. Data l'immagine di riferimento I, si calcola per ogni altra immagine vicina V un punteggio globale $g_I(V)$ come la somma pesata delle *feature* condivise tra I e V:

$$g_I(V) = \sum_{f \in F_V \cap F_I} w(f) \tag{3.11}$$

dove F_X è l'insieme delle *feature* individuate nella vista X e $w(f)$ assegna un peso alla *feature* f in base alla similarità delle due viste R e V in termini di risoluzione e alla differenza di angolo di vista tra tutte le immagini vicine (cioè all'interno di un vicinato N). La funzione peso favorisce, quindi, le viste con risoluzione uguale o maggiore alla vista di riferimento e con angolo di vista simile a quello delle immagini vicine. Bisogna ora stabilire la dimensione del vicinato N in termini di somma dei punteggi globali delle viste contenute: $\sum_{V \in N} g_R(v)$. Ad esempio, si può utilizzare un approccio *greedy* per aumentare il vicinato in modo incrementale aggiungendo iterativamente a N la vista con il punteggio globale più alto.

Con la selezione locale si individua un insieme $A \subset N$ di immagini che hanno un'alta coerenza fotometrica (ossia con simili distribuzioni di luce) e alta distribuzione angolare, necessarie per uno *stereo matching* stabile. Data un'immagine di riferimento I e una nuova vista V da valutare, si calcola il valore *NCC* (*Normalized Cross Correlation*) tra due viste in una stessa regione di pixel, quindi si definisce per V un punteggio locale $l_I(V)$:

$$l_I(V) = g_I(V) \cdot \prod_{V' \in A} w_e(V, V') \tag{3.12}$$

dove $w_e(V, V')$ è determinato dalla differenza in angoli di vista tra V e V'. L'algoritmo, quindi, individua la vista V con punteggio $l_I(V)$ massimo. Se V ha punteggio NCC sufficientemente alto, viene aggiunto in A, altrimenti viene scartato. Si ripete l'algoritmo finché l'insieme A non raggiunge una dimensione desiderata o non ci sono più viste da considerare.

Una volta individuato l'insieme di immagini da usare per lo *stereo matching*, *MVE* esegue l'operazione di ricostruzione in due fasi:

- *Region-growing*: definisce una coda con priorità Q dei punti candidati per il *matching*;
- *Matching*: usa la coda con priorità Q per calcolare profondità, normale e confidenza di match per ciascun punto.

La prima fase sfrutta l'idea che un campione abbinato con successo fornisce una buona stima iniziale per la profondità, la normale e un valore di confidenza per il match per i pixel vicini. I punti candidati per il match vengono quindi inseriti in una coda con priorità per considerare inizialmente prima i campioni con confidenza di *match* attesa più alta. Nel caso in cui viene calcolato un abbinamento per un pixel già precedentemente abbinato, la nuova informazione di match va a sostituire quella salvata, se la nuova confidenza è maggiore di quella vecchia.

Nella seconda fase si ottimizzano le informazioni di profondità e normale stimate nella fase precedente per massimizzare la coerenza fotometrica. Un sistema di *matching* prende i candidati in Q e calcola profondità, normale e confidenza di match usando le viste vicine fornite dalla selezione locale. Se il *matching* ha successo, i dati sono salvati in mappe di profondità, normale e confidenza e i pixel vicini sono aggiunti ai nuovi candidati nella coda con priorità.

3.7 Ricostruzione della superficie

La fase di *surface reconstruction* utilizza una nuvola di punti 3D densa per costruire una superficie. In seguito sono descritti due algoritmi di ricostruzione che è possibile utilizzare in *MeshLab*.

3.7.1 Ball Pivoting

BPA (*Ball Pivoting Algorithm*) [19] è uno degli algoritmi che *MeshLab* mette a disposizione per la costruzione di una *mesh* triangolare a partire da una nuvola di punti. Esso è un algoritmo efficiente in termini di tempo e memoria ed è sufficientemente robusto da gestire il rumore presente in dati acquisiti tramite scansioni 3D.

L'idea alla base di *BPA* è la seguente: tre punti vengono collegati per formare un triangolo se una sfera (*ball*) di un certo raggio ρ li tocca senza che altri punti ricadano al suo interno. Partendo da un triangolo iniziale (*seed triangle*) e da una sfera di raggio ρ (ρ-sfera) che tocca i tre punti, si fa ruotare la sfera (*ball pivoting*) intorno ad un lato mantenendo contatto con i suoi due estremi, finché la sfera non tocca un altro punto; quest'ultimo viene collegato agli altri due punti toccati dalla sfera per formare un nuovo triangolo. Si procede in questo modo finché tutti i lati raggiungibili non sono stati provati; in tal caso si ricomincia da un altro triangolo iniziale, finché tutti i punti non sono stati considerati.

Vi sono due requisiti per poter utilizzare *BPA* come algoritmo di costruzione di una superficie:

- La nuvola di punti non deve presentare curve troppo accentuate: in particolare, se il raggio di curvatura è minore di ρ, alcuni punti possono essere irraggiungibili e quindi alcune *features* non verrebbero considerate nella costruzione della *mesh*;

- Ogni sfera di raggio ρ centrata in M deve contenere almeno un campione. Se infatti la nuvola di punti è troppo sparsa, la sfera può attraver-

sare la superficie senza toccare alcun punto e alcuni lati non verrebbero quindi prodotti, lasciando buchi nella *mesh* risultante. Se la nuvola di punti ha densità maggiore di ρ, tutti i punti saranno toccati dalla ρ-sfera. È possibile comunque eseguire l'algoritmo con differenti valori di ρ per gestire le nuvole di punti con densità irregolari.

Di seguito è mostrato lo schema generale dell'algoritmo. *BPA* dapprima individua un triangolo di partenza $(\sigma_i, \sigma_j, \sigma_k)$ tale che una sfera di raggio ρ che li tocca non contiene altri punti e quindi aggiunge un triangolo alla volta, eseguendo ripetutamente l'operazione di *ball pivoting*:

```
1  while(true)
2    while(e_(i,j) = get_active_edge(F))
3      if(σ_k = ball_pivot(e_(i,j)))
4        if(not_used(σ_k) || σ_k ∈ F))
5          create_triangle(σ_i, σ_j, σ_k)
6          join(e_(i,j), σ_k, F)
7          if(e_(k,i) ∈ F) glue(e_(i,k), e_(k,i), F)
8          if(e_(j,k) ∈ F) glue(e_(k,j), e_(j,k), F)
9        else
10         mark_as_boundary(e_(i,j))
11
12   if((σ_i, σ_j, σ_k) = find_seed_triangle())
13     create_triangle(σ_i, σ_j, σ_k)
14     add_edge(e_(e,j), F)
15     add_edge(e_(j,k), F)
16     add_edge(e_(k,i), F)
17   else
18     return
```

L'algoritmo utilizza una struttura dati F, detta fronte, che contiene una collezione di liste collegate di archi. Inizialmente F contiene solo i 3 archi del primo *seed triangle*, il quale viene calcolato nel seguente modo:

- Dato un punto σ non ancora utilizzato, si considerano tutte le coppie di punti σ_a, σ_b nel vicinato di σ in ordine di distanza;
- Si costruiscono i potenziali triangoli iniziali $\sigma, \sigma_a, \sigma_b$ e si verifica per ognuno di essi che la relativa normale sia coerente con le normali dei punti corrispondenti;

- Si verifica quindi se esiste una sfera di raggio ρ che tocca tutti e tre i vertici, senza che altri punti ricadano nella sua area;

- Si iterano i passi precedenti finchè non viene individuato un *seed triangle* valido.

Gli archi di F da utilizzare per il *pivoting* sono detti attivi; i lati sui quali non è possibile far ruotare la sfera sono marcati come *boundary*. Per ogni lato attivo, BPA utilizza due operatori topologici, *join* e *glue*, che generano nuovi triangoli, aggiungendo e rimuovendo alcuni lati dal fronte F.

L'operazione *join* è la più semplice e viene utilizzata quando la sfera, ruotando intorno ad un lato $e_{(i,j)}$, tocca un nuovo vertice σ_k. In questo caso, l'operatore crea il triangolo $(\sigma_i, \sigma_j, \sigma_k)$, rimuove da F il lato $e_{(i,j)}$ e aggiunge in F i nuovi lati $e_{(i,k)}$ e $e_{(k,j)}$.

L'operazione *glue* è invece applicata quando viene creato un arco identico a un lato già presente in F, ma con orientamento opposto. In tal caso, le coppie di lati coincidenti vengono rimosse. Ad esempio, se il lato $e_{(i,k)}$ viene aggiunto al fronte dall'operazione *join* e il lato $e_{(k,i)}$ è in F, l'operatore *glue* rimuove le coppie di lati $e_{(i,k)}, e_{(k,i)}$, aggiustando il fronte opportunamente.

3.7.2 Poisson Surface Reconstruction

L'algoritmo *Poisson Surface Reconstruction* [20] è uno degli algoritmi messi a disposizione da *MeshLab* per generare una *mesh* da una nuvola di punti. Come requisito fondamentale per poter applicare questo algoritmo, tutti i punti di input devono avere le corrispondenti normali correttamente orientate, in quanto si utilizza il *normal field* (spazio definito dalle normali) per approssimare il modello creando una isosuperficie[2].

L'idea di base è quella di individuare una funzione caratteristica che permette di distinguere i punti all'interno o all'esterno del modello, ossia il suo valore è 1 se la funzione è applicata all'interno dell'oggetto, 0 altrimenti.

[2]Superficie definita da una insieme di equazioni del tipo $f(x, y, z) = 0$

Assumendo che i punti calcolati formino la superficie del modello, una isosuperficie può essere approssimata da tale funzione, il cui gradiente è uguale al campo definito dalle normali vicino alla superficie e zero altrove.

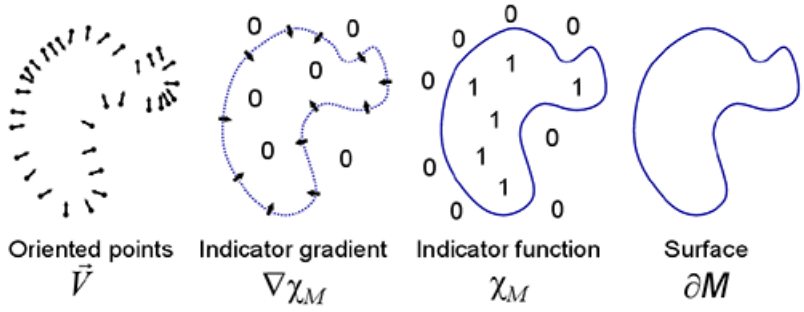


Figura 3.5: Estrazione di una isosuperficie da un insieme di punti orientati. Immagine tratta da [20]

Per l'estrazione della superficie si usa l'algoritmo *Marching Cubes* che divide la nuvola di punti in una griglia di cubi (*voxel grid*) e quindi, analizzando un cella alla volta, verifica quali siano i punti che definiscono la superficie dell'oggetto.

3.8 Semplificazione

La semplificazione [21] o decimazione è il processo col quale una *mesh* poligonale può essere trasformata in un'altra *mesh* con un numero minore di facce, lati e vertici. È quindi un metodo che consente di ridurre la complessità del modello realizzato nelle fasi precedenti. Formalmente, data una *mesh* $M = (V, F)$, dove V è l'insieme dei vertici e F è l'insieme delle facce, il processo di semplificazione di una *mesh* M consiste nel calcolare una *mesh* $M' = (V', F')$ tale che:

- $|V'| = n < |V|$ e $||M - M'||$ sia minima oppure
- $||M - M'|| < \epsilon$ e $|V'|$ sia minima.

Nel primo caso l'obiettivo è minimizzare l'errore tra la *mesh* originale e quella semplificata, dato un valore costante n che rappresenta il numero di vertici desiderato. Nel secondo caso, invece, si utilizza una costante ϵ che costituisce l'errore massimo tra i due modelli e pertanto la minimizzazione riguarda il numero di vertici della *mesh* ridotta. La semplificazione è generalmente controllata da un insieme di criteri di qualità definiti dall'utente, grazie ai quali è possibile preservare alcune proprietà specifiche della *mesh*, come ad esempio la sua topologia o la parametrizzazione UV.

Gli algoritmi di decimazione possono essere racchiusi principalmente in due categorie:

- *Algoritmi di clustering*, che riducono i vertici di una *mesh* fondendoli. Sono generalmente veloci con complessità O(n), dove n è il numero di vertici, ma la qualità della *mesh* di output può non essere soddisfacente;
- *Algoritmi di rimozione incrementale*, in cui in ogni step viene eliminato un vertice o lato. Permettono di produrre *mesh* ad alta qualità, ma hanno complessità maggiore rispetto agli algoritmi di *clustering*.

Di seguito sono descritti gli algoritmi di semplificazione implementati in *MeshLab* e in *Blender*.

3.8.1 Clustering

L'algoritmo *Clustering Decimation* [21] è un algoritmo di *vertex clustering* che raggruppa i vertici vicini di una *mesh* in *cluster* e sostituisce ogni gruppo con un singolo vertice. Data una tolleranza $\epsilon > 0$, lo spazio circostante la *mesh* viene partizionato in celle con lato o diametro minore o uguale a ϵ e tutti i vertici all'interno di una stessa cella costituiscono un *cluster*. Per ogni area viene calcolato un vertice rappresentativo e i punti contenuti in una stessa cella vengono sostituiti da tale vertice. Vi sono varie possibilità per calcolare il punto rappresentativo di una cella: esso può coincidere con uno dei vertici contenuti, oppure può corrispondere alla media dei vertici del *cluster* o al centro geometrico del *cluster*, oppure alla posizione che minimizza

una certa funzione di errore. *MeshLab* in particolare permette di scegliere tra le strategie "*average*" o "*closest to center*", con cui il vertice rappresentativo viene calcolato rispettivamente come la media dei punti della cella o come il centro del *cluster*.

I centri di ciascun *cluster* così calcolati costituiscono il nuovo insieme di vertici della *mesh*. In generale, se P e Q sono i vertici rappresentativi rispettivamente di p_0, p_1, ..., p_m e q_0, q_1, ..., q_n, P e Q sono collegati da un lato se esiste almeno una coppia di vertici (p_i, q_j) connessi nella *mesh* originale. Un esempio di *Clustering Decimation* è mostrato in figura 3.6.

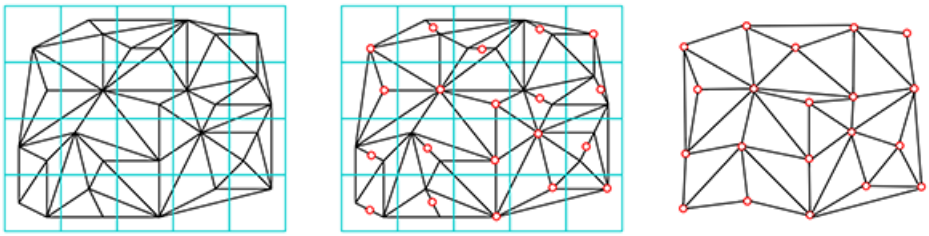


Figura 3.6: *Clustering Decimation*

Questo algoritmo ha il vantaggio di poter essere eseguito molto velocemente e di produrre un'approssimazione globale di una *mesh*. D'altro canto, esso non preserva sempre la topologia della *mesh* originale; la *mesh* derivata potrebbe non essere *2-manifold* anche se lo è quella originale, poiché una porzione della superficie potrebbe collassare in un punto.

Esistono altri algoritmi di *Clustering Decimation*, non implementati in *MeshLab*, che utilizzano strutture dati diverse dalle griglie uniformi 3d, come ad esempio gli *octrees*, ossia alberi in cui ogni nodo interno ha esattamente otto nodi figli. Queste strutture dati permettono di adattare la risoluzione del *clustering* in base alla distribuzione geometrica della *mesh*, grazie alla capacità di poter suddividere lo spazio 3D ricorsivamente in otto sezioni, dette ottanti. In questo modo, è possibile partizionare lo spazio in modo non uniforme, aumentando o diminuendo la dimensione del singolo *cluster* in base alla geometria da decimare.

3.8.2 Quadric Edge Collapse

L'algoritmo *Quadric Edge Collapse Decimation* [22] è un algoritmo di rimozione incrementale che sfrutta l'operatore *edge collapse* e una misura di errore per individuare ad ogni step il lato da collassare o, in altre parole, la coppia di vertici da fondere. La metrica di errore utilizzata, detta *quadric error distance measure*, calcola per ogni vertice della *mesh* una matrice di errore simmetrica 4x4 e un *quadric error* (errore di secondo grado). Quindi, per ogni lato si individua un nuovo vertice con valore minimo di errore, nel quale il lato viene fatto collassare. La versione implementata in *MeshLab* è in grado di semplificare la *mesh* preservando la parametrizzazione UV e quindi le *texture* applicate.

Per calcolare l'errore quadratico di un vertice v, si considera il piano associato a ciascun triangolo della *mesh*, del quale si calcola la normale tramite il prodotto vettoriale:

$$ax + by + cz + d = 0 \qquad \vec{n} = \frac{AB \times AC}{|AB \times AC|} \tag{3.13}$$

La distanza (cioè, l'errore) tra il vertice v e un piano p è definita dalla seguente equazione:

$$\Delta_p(v) = av_1 + bv_2 + cv_3 + d \tag{3.14}$$

L'errore in v rispetto a p può essere quindi calcolato inserendo le coordinate di v nell'equazione di p. Riscrivendo il vertice v e il piano p come matrici, il valore $\Delta_p(v)$ corrisponde al prodotto delle due matrici:

$$p = \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \qquad v = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \\ 1 \end{pmatrix} \qquad \Delta_p(v) = \begin{pmatrix} av_1 \\ bv_2 \\ cv_3 \\ d \end{pmatrix} = p^T \cdot v \tag{3.15}$$

$\Delta_p(v)$ è la distanza con segno da v verso p. Tale distanza può essere un valore negativo, quindi calcoliamo il suo quadrato:

$$\begin{aligned}\Delta_p(v)^2 = (p^T v)^2 = (p^T v)^T (p^T v) = (v^T p)(p^T v) = v^T (pp^T) v \\ = v^T \begin{pmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{pmatrix} v = v^T M_p v\end{aligned} \tag{3.16}$$

Sia M_p la matrice data dal prodotto della matrice p per se stessa trasposta. Quindi, per ogni vertice v di una *mesh*, la sua matrice simmetrica di errore, che denominiamo Q_v, è data dalla somma delle matrici $M_p(v)$ di tutti i piani che contengono un triangolo incidente in v:

$$Q_v = \sum_{\forall \text{ p incidente in v}} M_p \tag{3.17}$$

L'errore quadratico di un vertice è dato dalla somma dei quadrati delle distanze tra il vertice e i piani con triangoli incidenti in v:

$$\begin{aligned}\Delta_v = \sum_{\forall \text{ p incidente in v}} \Delta_p(v)^2 = \sum_{\forall \text{ p incidente in v}} v^T M_p v \\ = v^T \left(\sum_{\forall \text{ p incidente in v}} M_p \right) v = v^T Q_v v\end{aligned} \tag{3.18}$$

Calcolati i valori di errore per ciascun vertice, bisogna individuare il vertice v con valore di errore Δ_v minimo. La formula $v^T Q_v v$ è un'equazione di secondo grado in v, pertanto il suo minimo può essere calcolato impostando le derivate parziali a 0 e risolvendole rispetto a x, y e z. Un punto v è definito da tre variabili, cioè $v = (x, y, z)$ e il suo valore di errore $v^T Qvv$ ha la forma di un'equazione di secondo grado:

$$f(x, y, z) = ax^2 + by^2 + cz^2 + 2dxy + 2exz + 2fyz + 2gx + 2hy + 2iz + j \tag{3.19}$$

Impostando le derivate parziali a 0 e risolvendo per x, y e z si ottiene il vettore v:

$$\begin{aligned}\frac{\partial f}{\partial x} &= 2ax + dy + ez + g = 0 \\ \frac{\partial f}{\partial y} &= 2dx + by + fz + h = 0 \\ \frac{\partial f}{\partial z} &= 2ex + fy + cz + i = 0\end{aligned} \tag{3.20}$$

Essendo un algoritmo di *edge collapsing*, bisogna stabilire un criterio per la scelta del lato da collassare in ogni iterazione. Si considerano valide per la rimozione le coppie di vertici (p, q) per cui una delle seguenti condizioni sia rispettata:

- p e q sono collegati da un lato;
- la distanza tra p e q è inferiore ad una certa tolleranza ϵ definita dall'utente.

L'algoritmo quindi iterativamente fonde coppie di vertici, non necessariamente connessi da un lato. Data una coppia (p, q) valida, essa può essere collassata in uno dei due estremi (p o q) o nel punto intermedio $(p + q)/2$ o in un nuovo punto v che minimizza la seguente formula, dove Q_p e Q_q sono le matrici di errore rispettivamente di p e q:

$$\delta_v = \frac{(v^T Q_p v + v^T Q_q v)}{2} = \frac{[v^T (Q_p + Q_q) v]}{2} \tag{3.21}$$

Calcolato il vertice v, i due vertici p e q vengono fusi in v, il quale riceve il valore di errore δ_v e la matrice di errore $Q_p + Q_q$. Riassumendo lo schema generale dell'algoritmo è il seguente: dapprima si calcola l'insieme delle possibili coppie di vertici con relativi valori di errore. Quindi, in ogni iterazione si individua la coppia di vertici con errore minimo, facendola collassare nel punto che minimizza l'errore. Quindi si ricalcolano i nuovi valori di errore per i vertici vicini.

3.8.3 Dissolve

L'algoritmo di decimazione *Dissolve* implementato in *Blender* riduce le aree piatte in poligoni in base al parametro "*angle_limit*", definito dall'utente, che rappresenta l'angolo massimo tra due lati e tra due facce. La procedura opera in due fasi, gestendo dapprima gli archi e poi i vertici.

Nella prima fase, viene creato un *heap* di tante celle quanti sono gli archi. Quindi gli archi *wire* (archi senza facce connesse) vengono marcati con il valore -1 per la successiva fase di rimozione:

```
BM_ITER_MESH (e_iter, &iter, bm, BM_EDGES_OF_MESH)
BM_elem_flag_set(e_iter, BM_ELEM_TAG, BM_edge_is_wire(e_iter));
BM_elem_index_set(e_iter, -1);
```

L'*heap* viene popolato con i valori di *dissolve error* di ciascun arco. Dato un arco, il suo valore di errore viene calcolato come il coseno negativo dell'angolo formato dalle normali delle due facce collegate:

```
float angle_cos_neg = dot_v3v3(e->l->f->no, e->l->radial_next->f->no);
```

L'algoritmo, quindi, itera finché l'*heap* non è vuoto e l'elemento in cima ha valore minore dell'angolo limite. Ad ogni iterazione, si estrae un arco dall'*heap*: se l'arco è *manifold* (cioè, collega 2 facce), si crea una nuova faccia fondendo le due facce toccate dall'arco e se ne calcola la normale, quindi si ricalcolano i costi degli archi della nuova faccia, aggiornando i valori nello *heap*.

```
do
const int j = BM_elem_index_get(l_iter->e);
if (j != -1 && eheap_table[j])
const float cost = bm_edge_calc_dissolve_error(l_iter->e, delimit, &
    delimit_data);
BLI_heap_node_value_update(eheap, eheap_table[j], cost);
while ((l_iter = l_iter->next) != l_first);
```

Successivamente, si rimuovono gli archi e i vertici precedentemente marcati per la rimozione. Considerando tutti gli archi, se un arco è *wire* ma non è marcato con valore "-1" nell'*heap*, significa che inizialmente l'arco non

era *wire* ma lo è diventato durante la computazione. L'arco viene quindi eliminato insieme ai suoi vertici, se non sono connessi ad altri archi.

```
for (i = bm->totedge - 1; i != -1; i--)
  e_iter = earray[i];
  if (BM_edge_is_wire(e_iter) && (BM_elem_flag_test(e_iter, BM_ELEM_TAG) ==
    false))
    int vidx_reverse;
    BMVert *v1 = e_iter->v1; BMVert *v2 = e_iter->v2;
    BM_edge_kill(bm, e_iter);
    if (v1->e == NULL)
      vidx_reverse = vert_reverse_lookup[BM_elem_index_get(v1)];
      if (vidx_reverse != -1) vinput_arr[vidx_reverse] = NULL;
      BM_vert_kill(bm, v1);
    if (v2->e == NULL)
      vidx_reverse = vert_reverse_lookup[BM_elem_index_get(v2)];
      if (vidx_reverse != -1) vinput_arr[vidx_reverse] = NULL;
      BM_vert_kill(bm, v2);
```

Anche nella fase successiva di gestione dei vertici viene utilizzato un *heap* creato a partire dai vertici della *mesh*. Per ogni vertice si calcola il suo *dissolve error* e il vertice, insieme al suo errore, viene inserito nell'*heap*. Il costo di un vertice è calcolato come il prodotto tra l'angolo formato dai due lati toccati dal vertice e l'angolo tra le due facce collegate. Gli angoli vengono dapprima convertiti da radianti in valori compresi tra 0 e 1, quindi moltiplicati tra loro e infine convertiti nuovamente in radianti.

```
#define UNIT_TO_ANGLE DEG2RADF(90.0f)
#define ANGLE_TO_UNIT (1.0f / UNIT_TO_ANGLE)

const float angle = BM_vert_calc_edge_angle(v);
if (v->e && BM_edge_is_manifold(v->e))
  return ((angle * ANGLE_TO_UNIT) * (BM_edge_calc_face_angle(v->e) *
    ANGLE_TO_UNIT)) * UNIT_TO_ANGLE;
else
  return angle;
```

Nuovamente, l'algoritmo itera finché l'*heap* non è vuoto e il vertice in cima ha valore minore dell'angolo limite. Ad ogni step, si estrae un vertice dallo *heap*: se esso incide su due archi, questi ultimi vengono fusi in un nuovo

arco. Il vertice viene quindi rimosso dall'*heap*. Dato il nuovo arco, si itera radialmente sulle facce vicine per aggiornare le normali di tali facce.

```
if (BM_vert_is_edge_pair(v))
  e_new = BM_vert_collapse_edge(bm, v->e, v, true, true);
  if (e_new)
    BLI_heap_remove(vheap, vnode_top);
    vheap_table[i] = NULL;
    if (e_new->l)
      BMLoop *l_first, *l_iter;
      l_iter = l_first = e_new->l;
      do
        BM_face_normal_update(l_iter->f);
      while ((l_iter = l_iter->radial_next) != l_first);
```

Infine, si ricalcolano i costi dei vertici del nuovo lato, aggiornando i valori dello *heap*.

```
BM_ITER_ELEM (v_iter, &iter, e_new, BM_VERTS_OF_EDGE)
  const int j = BM_elem_index_get(v_iter);
  if (j != -1 && vheap_table[j])
    const float cost = bm_vert_edge_face_angle(v_iter);
    BLI_heap_node_value_update(vheap, vheap_table[j], cost);
```

3.8.4 Unsubdivide

La decimazione *Unsubdivide* implementata in *Blender* è una procedura iterativa, il cui numero di iterazioni è definito dal parametro *iterations* scelto dall'utente durante la configurazione del modificatore.

In ciascuna iterazione, si considerano tutti i vertici della *mesh* e si verifica per ognuno di essi se il ventaglio (*fan*) di triangoli incentrato nel vertice possa essere collassato. La procedura *bm_vert_dissolve_fan_test* esegue il test seguente: se il test ha esito positivo, il vertice viene marcato con i flag ELE_VERT_TAG e $VERT\_INDEX\_INIT$. Se invece il *fan* di triangoli non può essere collassato, il vertice viene marcato con il flag BM_INDEX_IGNORE e non sarà quindi considerato durante la rimozione.

```
BM_ITER_MESH (v, &iter, bm, BM_VERTS_OF_MESH)
  if (BM_elem_flag_test(v, BM_ELEM_TAG) && bm_vert_dissolve_fan_test(v))
    BMO_vert_flag_enable(bm, v, ELE_VERT_TAG);
```

```
4     BM_elem_index_set(v, VERT_INDEX_INIT);
5   else
6     BM_elem_index_set(v, VERT_INDEX_IGNORE);
```

In dettaglio, il test itera sugli archi incidenti sul vertice v per inizializzare le seguenti variabili:

- *tot_edge*: numero di archi incidenti sul vertice v;
- *tot_edge_boundary*: numero di archi di confine (una faccia connessa)
- *tot_edge_wire*: numero di archi *wire* (nessuna faccia connessa)
- *tot_edge_manifold*: numero di archi *manifold* (2 facce connesse);
- $varr[]$: per ogni arco incidente nel vertice v, mantiene un riferimento all'altro vertice dell'arco.

Il test restituisce esito positivo se il *fan* è composto dalle seguenti combinazioni di archi:

- 2 archi *wire*;
- 4 archi *manifold*;
- 3 archi *manifold*;
- 2 archi *boundary* e 1 arco *manifold*.

Negli ultimi tre casi, si verifica inoltre se esiste una faccia composta esattamente dai vertici presenti in $varr[]$. In caso negativo, il *fan* può essere collassato. Questo controllo è necessario per verificare che non vi siano facce in sovrapposizione. In tutti gli altri casi non precedentemente descritti, il test restituisce un esito negativo.

Eseguito il test per tutti i vertici e impostati opportunamente i rispettivi flag, l'algoritmo itera sui vertici per individuare un vertice v_first con i flag $VERT\_INDEX\_INIT$ e ELE_VERT_TAG (quindi un vertice che ha

passato il test). Si inizializza una procedura di visita (*walker*) per visitare solo gli elementi con il flag ELE_VERT_TAG a partire dal vertice v_first. La ricerca procede per ampiezza (*breadth first*) e pertanto ignora gli elementi che non sono ad una certa profondità (nth); i vertici visitati vengono invece marcati con il flag $VERT\_INDEX\_DO\_COLLAPSE$, che corrisponde al valore -1.

```
BMW_init(&walker, bm, BMW_CONNECTED_VERTEX, ELE_VERT_TAG, BMW_MASK_NOP,
    BMW_MASK_NOP, BMW_FLAG_NOP, BMW_NIL_LAY);
BLI_assert(walker.order == BMW_BREADTH_FIRST);
for (v = BMW_begin(&walker, v_first); v != NULL; v = BMW_step(&walker))
  if (BM_elem_index_get(v) == VERT_INDEX_INIT)
    if ((offset + BMW_current_depth(&walker)) % nth)
      BM_elem_index_set(v, VERT_INDEX_DO_COLLAPSE);
    else
      BM_elem_index_set(v, VERT_INDEX_IGNORE);
BMW_end(&walker);
```

Terminate le iterazioni, i vertici che devono essere rimossi sono stati marcati con $VERT\_INDEX\_DO\_COLLAPSE$. Iterando su tutti i vertici e individuato un vertice con tale flag impostato, viene chiamata la funzione $bm\_vert\_dissolve\_fan$, la quale collassa il *fan* incentrato nel vertice.

```
iter_done = false;
BM_ITER_MESH_MUTABLE (v, v_next, &iter, bm, BM_VERTS_OF_MESH)
  if (BM_elem_index_get(v) == VERT_INDEX_DO_COLLAPSE)
    if (bm_vert_dissolve_fan(bm, v))
      iter_done = true;
if (iter_done == false)
  break;
```

La funzione $bm\_vert\_dissolve\_fan$ collassa il *fan* incentrato in un vertice v se esso è composto dalle seguenti combinazioni di archi:

- 4 archi *manifold*;
- 3 archi *manifold*;
- 2 archi *boundary* e 1 arco *manifold*.

Se invece il *fan* è composto da 2 archi *wire*, allora i due archi semplicemente collassano nel vertice v:

```
if (tot_edge == 2)
if (tot_edge_wire == 2)
  return (BM_vert_collapse_edge(bm, v->e, v, true, true) != NULL);
```

Negli altri casi i *loop* del vertice vengono suddivisi, quindi il vertice v viene rimosso. Nel caso in cui non vi siano *loop*, la procedura restituisce esito negativo e il vertice non può essere rimosso.

3.8.5 Collapse

L'algoritmo *Collapse* implementato in *Blender* esegue la decimazione di una *mesh* eseguendo un metodo di *edge collapse* (vedi 3.8.2). Inizialmente si alloca un *heap* contenente gli archi della *mesh* e un *array* dei valori di errore quadratici (*Quadric*) per ogni vertice. Come già detto, il calcolo dell'errore quadratico di un vertice richiede la costruzione di una matrice simmetrica, ottenuta dalla matrice colonna dei coefficienti del piano incidente nel punto moltiplicata per se stessa trasposta; essendo il risultato una matrice simmetrica, è possibile definirla completamente con la sua parte triangolare superiore.

```
typedef struct Quadric {
double a2, ab, ac, ad,
           b2, bc, bd,
               c2, cd,
                   d2;
} Quadric;
```

Sommando tutte queste matrici per ogni piano incidente nel vertice, si ottiene la matrice di errore del punto.

```
Quadric q;

BM_face_calc_center_mean(f, center);
copy_v3db_v3fl(plane_db, f->no);
plane_db[3] = -dot_v3db_v3fl(plane_db, center);
BLI_quadric_from_plane(&q, plane_db);
l_iter = l_first = BM_FACE_FIRST_LOOP(f);
do
BLI_quadric_add_qu_qu(&vquadrics[BM_elem_index_get(l_iter->v)], &q);
```

```
10 while ((l_iter = l_iter->next) != l_first);
```

Si calcolano quindi i costi di tutti i lati della *mesh*, sommando le matrici *Quadric* dei due punti e valutando l'errore quadratico nel punto intermedio dell'arco.

```
optimize_co[0] = 0.5 * ((double)e->v1->co[0] + (double)e->v2->co[0]);
optimize_co[1] = 0.5 * ((double)e->v1->co[1] + (double)e->v2->co[1]);
optimize_co[2] = 0.5 * ((double)e->v1->co[2] + (double)e->v2->co[2]);

const Quadric *q1, *q2;
q1 = &vquadrics[BM_elem_index_get(e->v1)];
q2 = &vquadrics[BM_elem_index_get(e->v2)];
cost = (BLI_quadric_evaluate(q1, optimize_co) + BLI_quadric_evaluate(q2,
    optimize_co));
```

Il valore numerico del costo ottenuto dalla matrice di errore è calcolato dalla funzione *BLI_quadric_evaluate* che moltiplica i valori del punto medio dell'arco per i coefficienti della matrice *Quadric* in input.

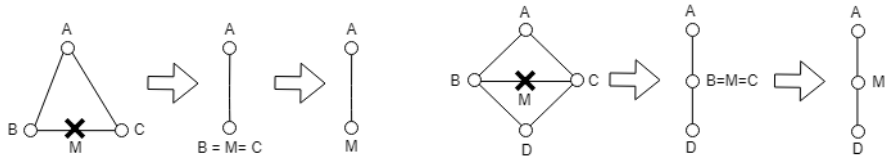
```
return ((q->a2 * v[0] * v[0]) + (q->ab * 2 * v[0] * v[1]) + (q->ac * 2 * v
    [0] * v[2]) + (q->ad * 2 * v[0]) + (q->b2 * v[1] * v[1]) + (q->bc * 2 *
    v[1] * v[2]) + (q->bd * 2 * v[1]) + (q->c2 * v[2] * v[2]) + (q->cd * 2 *
     v[2]) + (q->d2));
```

Calcolati i valori di errore per ogni lato, si costruisce un *heap* dei lati e si itera finché l'*heap* non è vuoto e non è stato raggiunto un numero predefinito di facce. Ad ogni iterazione si estrae dallo *heap* il lato con costo minimo e se ne verifica la tipologia:

- Se il lato è *boundary* (ha 1 faccia vicina), si verifica se la faccia incidente è un triangolo. In caso affermativo, il lato viene eliminato e i suoi due vertici vengono fusi dalla funzione *BM_vert_splice*. Dalla fusione dei due vertici, si forma una coppia di lati coincidenti: supponiamo di eliminare il lato BC da un triangolo di vertici ABC e quindi di unire i punti B e C in un nuovo punto M; i lati AB e AC diventano entrambi AM e AM. La coppia di lati coincidenti viene congiunta in un unico lato dalla funzione *BM_edge_splice* (Fig. 3.7 (a));

- Se il lato è *manifold* (ha 2 facce vicine), la procedura è la stessa ma in questo caso le facce incidenti sul lato sono 2. Verificato che entrambe siano dei triangoli, il lato viene eliminato e i suoi vertici vengono fusi, generando in questo caso due coppie di lati coincidenti, che vengono infine unite da *BM_edge_splice* (Fig. 3.7 (b));

- In tutti gli altri casi, la procedura dà esito negativo.



(a) (b)

Figura 3.7: Rimozione di un lato *boundary* (a) e di un lato *manifold* (b). Nel primo caso, vi è una coppia di lati coincidenti (AB=AC), mentre nel secondo caso vi sono due coppie (AB=AC e BD=CD).

Ogni volta che un lato viene eliminato, vengono ricalcolati i pesi dei vertici vicini, quindi si applica una procedura di normalizzazione affinché i valori siano compresi nell'intervallo $[0, 1]$:

```
1 float v_other_weight = interpf(vweights[v_other_index], vweights[
      v_clear_index], customdata_fac);
2 CLAMP(v_other_weight, 0.0f, 1.0f);
3 vweights[v_other_index] = v_other_weight;
```

3.9 Array

Il modificatore *Array* di *Blender* moltiplica un oggetto lungo una o più direzioni (assi).

Di default, il modificatore crea un duplicato dell'oggetto lungo l'asse x con un offset pari ad 1 unità di distanza. La struttura *ArrayModifierData* contiene tutti i parametri per la configurazione del modificatore:

- *start_cap*: oggetto che rappresenta l'estremo iniziale della sequenza;
- *end_cap*: oggetto che rappresenta l'estremo finale della sequenza;
- *curve_ob*: curva che il modificatore segue per moltiplicare un oggetto;
- $offset\_ob$: oggetto che il modificatore usa come offset;
- $offset[3]$: offset costante per la duplicazione;
- *length*: direzione sulla quale effettuare la duplicazione;
- *count*: numero di oggetti, compreso l'oggetto iniziale, creato dal modificatore. Di default, $count = 2$, ossia il modificatore crea una sola copia dell'oggetto al quale viene applicato;
- fit_type: tipologia di duplicazione, che può essere a numero di oggetti duplicati fisso, a lunghezza fissa o guidata da una curva;
- $offset\_type$; tipologia di offset, che può essere costante o relativo;
- $merge\_dist$: limite al di sotto del quale i vertici vicini vengono fusi; Il suo valore di default è 0.01;
- $scale[3]$: fattori di scala per le direzioni; di default $scale[0] = 1$, mentre $scale[1]$ e $scale[2]$ sono impostati a 0;
- $flags$: flag generici.

La funzione che applica il modificatore *Array* è $arrayModifier\_doArray$ i cui parametri sono i seguenti:

- *amd*: struttura *ArrayModifierData* che contiene i dati di configurazione del modificatore;
- *scene*: scena attuale contenente l'oggetto da modificare;
- *ob*: oggetto soggetto al modificatore;

- *dm*: struttura *DerivedMesh* contenente i dati dell'oggetto prodotto in output;
- *flag*: flag generici.

La prima operazione eseguita è il calcolo dell'offset, cumulando nel vettore $offset[3]$ le informazioni dell'offset costante, dell'offset relativo e dei fattori di scala delle tre dimensioni. Si calcola, quindi, il numero massimo di copie che entrano all'interno nella lunghezza prefissata; se l'offset è maggiore di una certa soglia *eps*, il nuovo duplicato viene creato ad una certa distanza dal duplicato precedente, pari a $(length + eps)/dist + 1$, altrimenti se la distanza non viene specificata (quindi minore di *eps*), il nuovo duplicato viene creato con offset pari ad 1 unità:

```
float dist = len_v3(offset[3]);
if (dist > eps)
  count = (length + eps) / dist + 1
else
  count = 1
```

Individuata la distanza in cui creare il duplicato, bisogna calcolare il numero di vertici, archi, *loops* (coppie di vertici) e *polys* (poligoni) della *mesh* risultante, prima di eseguire un'eventuale operazione di *merging doubles* (fusione dei vertici molto vicini). Ad esempio, il numero dei vertici corrisponde al numero di vertici del singolo oggetto moltiplicato per il numero di duplicati, aggiungendo i vertici delle *mesh* che costituiscono gli estremi della sequenza. Gli altri valori di inizializzazione vengono calcolati in modo simile. Viene pertanto creata una nuova geometria a partire da tali valori:

```
result_nverts = chunk_nverts * count + start_cap_nverts + end_cap_nverts;
result_nedges = chunk_nedges * count + start_cap_nedges + end_cap_nedges;
result_nloops = chunk_nloops * count + start_cap_nloops + end_cap_nloops;
result_npolys = chunk_npolys * count + start_cap_npolys + end_cap_npolys;

result = CDDM_from_template(dm, result_nverts, result_nedges, 0,
    result_nloops, result_npolys);

DM_copy_vert_data(dm, result, 0, 0, chunk_nverts);
DM_copy_edge_data(dm, result, 0, 0, chunk_nedges);
```

```
DM_copy_loop_data(dm, result, 0, 0, chunk_nloops);
DM_copy_poly_data(dm, result, 0, 0, chunk_npolys);
```

Per eseguire la successiva fase di *merging*, viene creata una mappa completa dei vertici duplicati ($full\_doubles\_map$) composta da tanti valori -1 quanti sono i vertici della *mesh*:

```
if (use_merge)
  full_doubles_map = MEM_malloc_arrayN(result_nverts, sizeof(int), "mod
    array doubles map");
  copy_vn_i(full_doubles_map, result_nverts, -1);
```

Per ogni duplicato, vengono copiati i dati della geometria originale e ricalcolati gli offset relativi:

```
for (c = 1; c < count; c++)
  DM_copy_vert_data(result, result, 0, c * chunk_nverts, chunk_nverts);
  DM_copy_edge_data(result, result, 0, c * chunk_nedges, chunk_nedges);
  DM_copy_loop_data(result, result, 0, c * chunk_nloops, chunk_nloops);
  DM_copy_poly_data(result, result, 0, c * chunk_npolys, chunk_npolys);

  mv_prev = result_dm_verts;
  mv = mv_prev + c * chunk_nverts;

  mul_m4_m4m4(current_offset, current_offset, offset);
  for (i = 0; i < chunk_nverts; i++, mv++, mv_prev++)
    mul_m4_v3(current_offset, mv->co);
```

L'ultima operazione è la fusione dei vertici *doubles*. Per ogni punto della geometria, si verifica se vi siano altri vertici sufficientemente vicini da poter eseguire l'operazione di fusione. Ricordando che i vertici della geometria di partenza sono stati indicizzati in una matrice $full\_doubles\_map$ con valori -1, l'obiettivo è quello di individuare i vertici i cui valori in tale matrice sono diversi da -1, ossia i nuovi vertici prodotti dal modificatore.

Trovato un vertice con valore diverso da -1, si risale la catena dei *doubles* definita da $full\_doubles\_map$, ossia $new\_i = full\_doubles\_map[new\_i]$, fino a raggiungere il valore new_i tale che $full\_doubles\_map[new\_i] = -1$. Se $i = new\_i$, allora non vi sono altri vertici new_i nella posizione di i, pertanto il vertice i è senza duplicati e può essere marcato con il valore -1. Diversamente, se $i \neq new\_i$, allora i due vertici sono *doubles* che dovranno essere fusi

e il contatore *tot_doubles* viene incrementato di un'unità. Se il contatore dei *doubles* è maggiore di 0, si esegue la fusione dei vertici della *mesh* "*result*", la quale viene infine restituita in output.

```
tot_doubles = 0;
if (use_merge)
  for (i = 0; i < result_nverts; i++)
    int new_i = full_doubles_map[i];
    if (new_i != -1)
      while (!ELEM(full_doubles_map[new_i], -1, new_i))
        new_i = full_doubles_map[new_i];
    if (i == new_i)
      full_doubles_map[i] = -1;
    else
      full_doubles_map[i] = new_i;
      tot_doubles++;
  if (tot_doubles > 0)
    result = CDDM_merge_verts(result, full_doubles_map, tot_doubles,
    CDDM_MERGE_VERTS_DUMP_IF_EQUAL);
```

3.10 Proportional Editing

Il *Proportional Editing* è una tecnica di modellazione 3D che permette di applicare una modifica ad un vertice selezionato, in modo tale che l'operazione si ripercuota in modo proporzionato sui vertici vicini. L'effetto dell'operazione dipende da due fattori: da un lato, il cursore circolare permette di delimitare l'area circostante il vertice selezionato; dall'altro, una funzione di *falloff* definisce l'intensità della trasformazione. In dettaglio, la funzione *calculatePropRatio* utilizza la struttura dati *TransData*, della quale le variabili interessate per il *proportional editing* sono le seguenti:

- *dist*: distanza minima per influenzare un vertice vicino;
- *rdist*: distanza tra il vertice selezionato e il suo vertice più vicino;
- *factor*: fattore della trasformazione.

I vertici vengono dapprima ordinati in base alla loro distanza (*dist*) dal vertice selezionato, escludendo in questo modo gli elementi che non rientrano

all'interno del cursore circolare. Si utilizza quindi il valore *rdist* per i calcoli di *falloff*, la cui funzione definisce l'intensità della trasformazione (*factor*) nel modo seguente:

```
switch (t->prop_mode)
  case PROP_SHARP:
    td->factor = dist * dist;
    break;
  case PROP_SMOOTH:
    td->factor = 3.0f * dist * dist - 2.0f * dist * dist * dist;
    break;
  case PROP_ROOT:
    td->factor = sqrtf(dist);
    break;
  case PROP_LIN:
    td->factor = dist;
    break;
  case PROP_CONST:
    td->factor = 1.0f;
    break;
  case PROP_SPHERE:
    td->factor = sqrtf(2 * dist - dist * dist);
    break;
  case PROP_RANDOM:
    td->factor = BLI_frand() * dist;
    break;
  case PROP_INVSQUARE:
    td->factor = dist * (2.0f - dist);
    break;
  default:
    td->factor = 1;
    break;
```

In breve, le funzioni di *falloff* disponibili per il *proportional editing* sono le seguenti:

- *Random*: la modifica proporzionata è definita da un numero casuale $BLI\_frand()$;
- *Constant*: il fattore di trasformazione è costante per tutti i punti, a prescindere dalla distanza dal vertice centrale;

- *Linear*: la trasformazione cresce linearmente all'aumentare della distanza dal vertice centrale;
- *Sharp*: la modifica proporzionata è pari al quadrato della distanza del vertice dal punto centrale. *Root* è la sua funzione inversa;
- *Root*: la modifica proporzionata è pari alla radice quadrata della distanza del vertice dal punto centrale. *Sharp* è la sua funzione inversa;
- *Sphere*: la proporzione della trasformazione è definita da una sfera;
- *Smooth*: la proporzione della trasformazione è definita da una funzione sigmoide;
- *Invsquare*: il fattore di trasformazione è pari alla differenza tra il doppio della distanza tra i due vertici e tale distanza al quadrato;

3.11 Sistemi particellari

I sistemi particellari implementati in *Blender* consentono la generazione automatica di una grande quantità di *mesh* con la stessa topologia, ma con differenti valori di scala, rotazione, posizione e colore. Di seguito si descrive il sistema particellare "*Emitter*" utilizzato nel corso di questo lavoro.

La funzione principale che gestisce l'evoluzione dinamica del sistema particellare è *system_step*. Essa calcola lo stato di tutte le particelle del sistema in ogni *frame*, eseguendo i seguenti quattro passi:

- Emissione, distribuzione e inizializzazione delle particelle;
- Verifica presenza del *frame* nella *cache*;
- Gestione della fisica;
- Salvataggio del frame nella *cache*.

La prima fase verifica innanzitutto se il numero di particelle presenti nel sistema sia diverso dal numero attuale impostato dall'utente; in tal caso le particelle vengono allocate nuovamente dalla funzione *realloc_particles* che elimina il vecchio sistema particellare dalla memoria.

```
ParticleSystem *psys = sim->psys;
int oldtotpart = psys->totpart;
int totpart = tot_particles(psys, pid);
if (totpart != oldtotpart)
  realloc_particles(sim, totpart);
return totpart - oldtotpart;
```

Le particelle (*ParticleData*) vengono quindi distribuite in modo casuale sulla superficie e per ognuna di esse si impostano alcuni parametri di inizializzazione, come ad esempio il tempo di creazione (*birth time*) o il suo indice nel sistema particellare, che rimangono fissi durante la computazione.

Nella fase successiva, si controlla se il *frame* è stato già precedentemente calcolato ed inserito nella *cache*. In caso affermativo, il sistema viene aggiornato in tale *frame* da *cached_step*, la quale applica una modifica casuale a ciascuna particella e ne ricalcola i tempi di vita. Se il tempo massimo di vita di una particella (*dietime*) è minore o uguale al *frame* attuale ($cfra$), allora la particella viene eliminata, altrimenti essa viene mantenuta per lo step successivo. Se invece la variabile *time* è maggiore di $cfra$, la particella non è ancora "nata" ($PARS\_UNBORN$) e verrà inserita nel sistema nei *frame* successivi.

```
if (pa->time > cfra)
  pa->alive = PARS_UNBORN;
  if (part->flag & PART_UNBORN && (psys->pointcache->flag & PTCACHE_EXTERNAL
    ) == 0)
    reset_particle(sim, pa, 0.0f, cfra);
else if (dietime <= cfra)
  pa->alive = PARS_DEAD;
else
  pa->alive = PARS_ALIVE;
```

Effettuata tale modifica, tutti i *frame* precedenti al *frame* attuale vengono inseriti nella *cache*. L'algoritmo quindi termina e procede con il *frame*

seguente. Se invece il *frame* non è presente nella *cache*, allora si procede con la fase successiva.

La terza fase permette di gestire le forze fisiche da applicare a ciascuna particella. Per ognuna di esse, si applica una modifica casuale e si ricalcola il suo tempo di vita, eventualmente resettando la particella se ha raggiunto il suo tempo massimo. Quindi si applica la forza impostata dall'utente nella scheda *Physics*. Ad esempio, se l'utente seleziona la fisica newtoniana, le particelle vengono ruotate e scalate in base alla forza di gravità, alle collisioni con gli altri oggetti e ad un fattore casuale definito dalle forze *browniane* applicate al sistema.

```
case PART_PHYS_NEWTON:
  LOOP_DYNAMIC_PARTICLES
    basic_integrate(sim, p, pa->state.time, cfra);
    if (sim->colliders)
      collision_check(sim, p, pa->state.time, cfra);
    basic_rotate(part, pa, pa->state.time, timestep);
  break;
```

L'ultima fase è la scrittura del *frame* nella *cache*.

```
if (pid)
  BKE_ptcache_validate(cache, (int)cache_cfra);
  if ((int)cache_cfra != startframe)
    BKE_ptcache_write(pid, (int)cache_cfra);
```

3.12 Texturing

Gli algoritmi di *texturing* implementati in *Blender* consentono di eseguire la parametrizzazione UV per ciascun vertice di una *mesh*. La differenza sostanziale tra i due algoritmi disponibili è il diverso utilizzo dei *seams*: mentre l'algoritmo *Unwrap* sfrutta i *seams* definiti dall'utente, nella procedura *Smart UV project* i *seams* vengono creati automaticamente.

3.12.1 Unwrap

L'algoritmo *Unwrap* consta di due fasi distinte:

- proiezione della *mesh* su una superficie 2D;
- correzione delle coordinate proiettate.

Mentre la prima fase differisce in base alla *mesh*, la seconda è analoga per tutte le primitive e permette di eseguire tre tipi di correzione:

- *Correct aspect*: permette di aggiustare l'*aspect ratio* dell'*UV mapping*. In particolare, se il rapporto tra i due lati dell'immagine è uguale a 1 ($aspx = aspy$, quindi è un quadrato), allora l'algoritmo termina. Se invece una dimensione è maggiore dell'altra (es. $aspx > aspy$) si calcola il fattore di scala $scale = aspy/aspx$ che viene moltiplicato per le coordinate x dei vertici delle facce selezionate. Nell'altro caso ($aspy > aspx$) il fattore di scala è invece $scale = aspx/aspy$, moltiplicato per le coordinate y dei vertici selezionati;
- *Scale to bounds*: Se l'intera *UV map* è al di fuori del range (0, 1), l'intera mappa vine scalata per entrare all'interno di tale intervallo. Per ogni UV si calcolano i valori minimi e massimi in entrambe le direzioni x e y, dai quali vengono calcolati i fattori di scala *dx* e *dy*. Questi ultimi, moltiplicati per le coordinate UV dei vertici selezionati, fanno sì che la mappa raggiunga un *aspect ratio* 1/1:

```
1  dx = (max[0] - min[0]);
2  dy = (max[1] - min[1]);
3
4  if (dx > 0.0f)  dx = 1.0f / dx;
5  if (dy > 0.0f)  dy = 1.0f / dy;
6
7  BM_ITER_MESH (efa, &iter, em->bm, BM_FACES_OF_MESH)
8    if (!BM_elem_flag_test(efa, BM_ELEM_SELECT)) continue;
9    BM_ITER_ELEM (l, &liter, efa, BM_LOOPS_OF_FACE)
10     luv = BM_ELEM_CD_GET_VOID_P(l, cd_loop_uv_offset);
11     luv->uv[0] = (luv->uv[0] - min[0]) * dx;
12     luv->uv[1] = (luv->uv[1] - min[1]) * dy;
```

- *Clip to bounds*: Tutti i valori UV all'esterno dell'intervallo (0, 1) sono troncati in tale intervallo.

```
  CLAMP(luv->uv[0], 0.0f, 1.0f);
  CLAMP(luv->uv[1], 0.0f, 1.0f);
```

La fase di proiezione, invece, dipende dalla tipologia di *mesh* sulla quale viene applicata. Di seguito si descrive, ad esempio, come avviene la proiezione di un cubo, di un cilindro e di una sfera.

Cube projection

Iterando su tutte le facce del cubo, si verifica quali siano le facce selezionate. Solo su quelle selezionate si esegue la proiezione della faccia calcolando in base alla normale i due assi dominanti.

```
const float xn = fabsf(axis[0]);
const float yn = fabsf(axis[1]);
const float zn = fabsf(axis[2]);

if      (zn >= xn && zn >= yn)  *r_axis_a = 0; *r_axis_b = 1;
else if (yn >= xn && yn >= zn)  *r_axis_a = 0; *r_axis_b = 2;
else                            *r_axis_a = 1; *r_axis_b = 2;
```

Per ogni *loop* della faccia, si calcolano i due valori UV per i due assi precedentemente ottenuti. Ogni valore viene definito in base alla dimensione del cubo, impostata come parametro dall'utente, e la distanza unidimensionale tra il vertice del *loop* e il centro della faccia.

```
BM_ITER_MESH (efa, &iter, bm, BM_FACES_OF_MESH)

  if (use_select && !BM_elem_flag_test(efa, BM_ELEM_SELECT))
    continue;

  axis_dominant_v3(&cox, &coy, efa->no);

  BM_ITER_ELEM (l, &liter, efa, BM_LOOPS_OF_FACE)
    luv = BM_ELEM_CD_GET_VOID_P(l, cd_loop_uv_offset);
    luv->uv[0] = 0.5f + 0.5f * cube_size * (l->v->co[cox] - loc[cox]);
    luv->uv[1] = 0.5f + 0.5f * cube_size * (l->v->co[coy] - loc[coy]);
```

Cylinder projection

Per ogni faccia del cilindro, si verifica se essa è selezionata. In caso affermativo, si itera sui *loop* corrispondenti, calcolando dapprima la distanza tra il vertice e il centro della *mesh* e mappando i vertici in un tubo (*tube*). La mappatura è eseguita dalla funzione *map_to_tube*, mostrata nel codice seguente.

```
void map_to_tube(float *r_u, float *r_v, const float x, const float y, const
     float z)
  float len;
  *r_v = (z + 1.0f) / 2.0f;
  len = sqrtf(x * x + y * y);
  if (len > 0.0f)
    *r_u = (1.0f - (atan2f(x / len, y / len) / (float)M_PI)) / 2.0f;
  else
    *r_v = *r_u = 0.0f;
```

Sphere projection

Per ogni faccia della sfera, si verifica se essa è selezionata. In caso affermativo, si itera sui *loop* corrispondenti, calcolando dapprima la distanza tra il vertice e il centro della *mesh* e mappando le coordinate uv come punti di una sfera, definita matematicamente nella funzione *map_to_sphere*.

```
void map_to_sphere(float *r_u, float *r_v, const float x, const float y,
    const float z)
  float len;
  len = sqrtf(x * x + y * y + z * z);
  if (len > 0.0f)
    if (UNLIKELY(x == 0.0f && y == 0.0f))
      *r_u = 0.0f;
    else
      *r_u = (1.0f - atan2f(x, y) / (float)M_PI) / 2.0f;
    *r_v = 1.0f - saacos(z / len) / (float)M_PI;
  else
    *r_v = *r_u = 0.0f;
```

3.12.2 Smart UV project

L'algoritmo *Smart UV project* genera una lista di proiezione contenente le normali delle facce della *mesh*, ordinate per area. Vengono, quindi, eliminate le facce con area nulla, in particolare con area minore di una costante $SMALL\_NUM$ che rappresenta la dimensione minima di una faccia.

Iterando sulle facce della *mesh*, si inserisce la media delle normali delle facce vicine nel vettore di proiezione $projectVecs$, quindi si calcola il prodotto scalare tra ciascuna faccia e i vettori calcolati in modo da ottenere l'angolo di proiezione. Si calcolano quindi le coordinate UV di una faccia a partire dai vertici proiettati:

```
MatQuat = VectoQuat(projectVecs[i])

for f in faceProjectionGroupList[i]:
  f_uv = f.uv
  for j, v in enumerate(f.v):
    f_uv[j][:] = (MatQuat * v.co).xy
```

Le coordinate UV delle facce vengono calcolate considerando gruppi o isole (*islands*) di facce e corrispondenti isole di coordinate UV. La prima operazione è il calcolo dei *seams*, in modo da non oltrepassarli durante la computazione e utilizzarli come confini per le isole.

```
edge_seams = {}
for ed in me.edges:
  if ed.use_seam:
    edge_seams[ed.key] = None
```

Per ciascuna faccia si traccia lo stato della computazione utilizzando un codice numerico a tre valori:

- 0 : faccia non ancora considerata;
- 1 : faccia considerata, ma non ancora utilizzata nella fase di ricerca;
- 2 : faccia considerata e utilizzata nella fase di ricerca.

Tutte le facce vengono inizializzate con il valore 0, quindi la prima faccia viene inserita nella lista delle isole e pertanto marcata con 1. Per ogni faccia

con codice pari a 1, si considerano le facce confinanti non ancora considerate (codice 0) e non connesse da un *seam*. In tal caso le facce vicine vengono marcate con 1 e inserite nella stessa isola della faccia confinante. Considerati tutti i vicini di una faccia, quest'ultima è stata completamente analizzata e viene quindi marcata con 2. L'isola viene inserita nella lista delle isole, quindi l'algoritmo ricomincia da una faccia con codice 0, ossia una faccia che corrisponde ad un'altra sezione della *mesh*.

```
ok = True
while ok:
  ok = True
  while ok:
    ok = False
    for i in range(len(faces)):
      if face_modes[i] == 1:
        for ed_key in faces[i].edge_keys:
          for ii in edge_users[ed_key]:
            if i != ii and face_modes[ii] == 0:
              face_modes[ii] = ok = 1
              newIsland.append(faces[ii])
        face_modes[i] = 2
  islandList.append(newIsland)

  ok = False
  for i in range(len(faces)):
    if face_modes[i] == 0:
      newIsland = []
      newIsland.append(faces[i])
      face_modes[i] = ok = 1
      break
```

3.13 Keyframe

L'animazione per *keyframes* consiste nel definire la posizione, rotazione e scala di un oggetto in diversi istanti di tempo, grazie ai quali esso viene animato tramite l'interpolazione di tali *keyframes*. Descriviamo ora come avviene la creazione di un *keyframe* in *Blender*.

Innanzitutto l'algoritmo esegue una validazione del cursore della *Timeline* che rappresenta il *frame* selezionato: se questo puntatore non esiste, l'algoritmo termina. Altro controllo preliminare è la verifica dell'esistenza della F-curva, ossia la curva 2D che interpola i *keyframes* rappresentati come punti in un piano 2D, permettendo così di gestire l'animazione dell'oggetto. Se la F-curva esiste ed è editabile (cioè non è stata bloccata per le modifiche), allora è possibile inserire un nuovo *keyframe* nel *frame* selezionato.

```
1 if (fcu == NULL)
2   BKE_report(reports, RPT_ERROR, "No F-Curve to add keyframes to");
3   return false;
4 if (fcurve_is_keyframable(fcu) == 0)
5   BKE_reportf(reports, RPT_ERROR, "F-Curve with path '%s[%d]' cannot be
      keyframed, ensure that it is not locked or sampled, " "and try removing
      F-Modifiers", fcu->rna_path, fcu->array_index);
6   return false;
```

La funzione *insert_vert_fcurve* inserisce il *keyframe* come un nuovo punto della F-curva, implementata come una curva di Bèzier. La struttura *BezTriple* permette, infatti, di gestire sia i *keyframe* delle F-curve che i punti delle curve di Bèzier. Ogni *keyframe* è definito dalla struttura *BezTriple*, il cui campo *vec* contiene le posizioni x, y e z del punto e delle manopole (*handle*) usate per modificare la curva:

- vec[0][0] = coordinata x del primo *handle*;
- vec[0][1] = coordinata y del primo *handle*;
- vec[0][2] = coordinata z del primo *handle* (non utilizzato nei punti 2D delle F-curve);
- vec[1][0] = coordinata x del punto di controllo;
- vec[1][1] = coordinata y del punto di controllo;
- vec[1][2] = coordinata z del punto di controllo;
- vec[2][0] = coordinata x del secondo *handle*;

- vec[2][1] = coordinata y del secondo *handle*;
- vec[2][2] = coordinata z del secondo *handle* (non utilizzato nei punti 2D delle F-curve).

Inizializzata opportunamente la struttura *BezTriple*, si impostano i valori per l'interpolazione, quali il periodo, l'ampiezza e la tipologia di interpolazione da utilizzare. Quindi l'interpolazione tra il segmento della F-curva e quello successivo viene definito a partire dal fattore di interpolazione precedente, solo se non ci sono stati cambiamenti nel numero di vertici della curva, ossia se non sono stati sostituiti alcuni *keyframe*; in tal caso, l'utente può aver specificato dei nuovi parametri di interpolazione che devono essere mantenuti.

3.14 Shading

Blender offre sei diverse possibilità per il *viewport shading*: *Bounding Box*, *Wireframe*, *Solid*, *Texture*, *Material* e *Render*. La voce scelta dall'utente viene salvata nella variabile *dt* (*draw type*) utilizzando un valore intero progressivo:

- $OB\_BOUNDBOX = 1$
- $OB\_WIRE = 2$
- $OB\_SOLID = 3$
- $OB\_MATERIAL = 4$
- $OB\_TEXTURE = 5$
- $OB\_RENDER = 6$

In base al valore della variabile *dt*, viene chiamata l'opportuna funzione che disegna gli oggetti con la modalità di *shading* scelta. Altre due modalità

di *shading* presenti in *Blender* sono il *Flat shading* e lo *Smooth shading* che, invece, governano il modo in cui le facce dell'oggetto vengono rappresentate in base all'illuminazione ricevuta. Di seguito sono descritti in breve i codici che gestiscono le diverse tipologie di *shading* suddette.

3.14.1 Bounding Box

Se $dt = OB\_BOUNDBOX$, per ogni oggetto viene designata la corrispondente *bounding box*. La funzione *draw_bounding_volume* restituisce, infatti, la componente *bb* (*bounding box*) dell'oggetto, se esistente. Nel caso in cui l'oggetto non abbia il campo *bb* definito, esso viene ricalcolato dalla funzione $BKE\_mesh\_texspace\_calc$, che calcola le coordinate minime e massime nelle tre direzioni e restituisce così gli estremi del parallelepipedo che contiene l'oggetto.

```
Mesh *me = ob->data;

if (ob->bb)
  return ob->bb;

if (me->bb == NULL || (me->bb->flag & BOUNDBOX_DIRTY))
  BKE_mesh_texspace_calc(me);

return me->bb;
```

3.14.2 Wireframe

Se $dt = OB\_WIRE$, gli oggetti vengono rappresentati solo con i loro lati, non disegnando quindi le facce. La funzione *cdDM_drawEdges* disegna quindi i contorni delle facce degli oggetti, con la possibilità di escludere i lati *loose*, ossia i lati senza facce adiacenti.

```
gdo = dm->drawObject;
if (gdo->edges && gdo->points)
  if (drawAllEdges && drawLooseEdges)
    GPU_buffer_draw_elements(gdo->edges, GL_LINES, 0, gdo->totedge * 2);
  else if (drawAllEdges)
```

```
6       GPU_buffer_draw_elements(gdo->edges, GL_LINES, 0, gdo->loose_edge_offset
         * 2);
7     else
8       GPU_buffer_draw_elements(gdo->edges, GL_LINES, 0, gdo->tot_edge_drawn *
        2);
9       GPU_buffer_draw_elements(gdo->edges, GL_LINES, gdo->loose_edge_offset *
        2, dm->drawObject->tot_loose_edge_drawn * 2);
```

3.14.3 Solid

Se $dt = OB\_SOLID$, gli oggetti vengono rappresentati come solidi, composti da vertici, lati e facce e illuminati dalle luci di default di *OpenGL*. La funzione $cdDM\_drawFacesSolid$, che si occupa della visualizzazione delle facce degli oggetti, esegue il disegno delle facce per triangoli successivi, rappresentando ognuno di essi con il colore di default di *Blender* per gli oggetti solidi.

```
1 GPU_vertex_setup(dm);
2 GPU_normal_setup(dm);
3 GPU_triangle_setup(dm);
4 for (a = 0; a < dm->drawObject->totmaterial; a++)
5   if (!setMaterial || setMaterial(dm->drawObject->materials[a].mat_nr + 1,
      NULL))
6     GPU_buffer_draw_elements( dm->drawObject->triangles, GL_TRIANGLES, dm->
      drawObject->materials[a].start, dm->drawObject->materials[a].totelements
      );
7 GPU_buffers_unbind();
```

Le facce possono essere rappresentate con due diverse tipologie di *shading*:

- *Flat shading*: ad ogni triangolo viene assegnato un colore uniforme, dato dall'angolo compreso tra la normale alla superficie e il raggio luminoso che irradia la faccia;

- *Smooth shading*: assegna un colore ad ogni pixel di un triangolo, dato dall'interpolazione delle normali dei tre vertici e dall'angolo compreso tra la normale interpolata e il raggio luminoso.

In base alla scelta dell'utente, il codice inizializza il corrisponde modello di *shading* tramite la primitiva *OpenGL* $glShadeModel()$, prima di procedere con il disegno del modello.

3.14.4 Texture

Se $dt = OB\_TEXTURE$, tutti gli oggetti vengono visualizzati come solidi insieme alle *texture* ad essi associate. La funzione *tex_mat_set_texture_cb* si occupa di assegnare una immagine come *texture* ad una *mesh*. In particolare, si verifica innanzitutto l'esistenza dell'immagine di input da assegnare come *texture*; in caso affermativo, l'eventuale materiale assegnato alla *mesh* viene disabilitato per poi assegnare la *texture* all'oggetto.

```
1 if (ED_object_get_active_image(data->ob, mat_nr, &ima, &iuser, &node, NULL))
2   ...
3   GPU_object_material_unbind();
4   glBindTexture(GL_TEXTURE_2D, ima->bindcode[TEXTARGET_TEXTURE_2D]);
```

Usando la parametrizzazione UV configurata, la *texture* viene assegnata come un materiale con massimo valore di diffusione (*diffuse*). Il disegno vero e proprio della *mesh* è eseguito dalla funzione *draw_mesh_textured*.

3.14.5 Material

Se $dt = OB\_MATERIAL$, ogni oggetto viene visualizzato con i materiali assegnati. La funzione *GPU_object_material_bind* si occupa dell'assegnazione del materiale. Se nessun materiale è stato definito, l'oggetto verrà rappresentato con il materiale di default di *Blender*, altrimenti i materiali assegnati vengono sovrapposti in base al canale *alpha*. Il disegno vero e proprio della *mesh* è eseguito dalla funzione drawMappedFacesGLSL che opera in due fasi:

- verifica degli attributi necessari per l'implementazione di ciascun materiale;
- generazione e riempimento degli *array* di dati necessari per la visualizzazione della *mesh*.

Infine, ogni materiale viene istanziato con i parametri precedentemente calcolati e disegnato sui singoli oggetti.

```
for (a = 0; a < tot_active_mat; a++)
  new_matnr = dm->drawObject->materials[a].mat_nr;
  do_draw = setMaterial(new_matnr + 1, &gattribs);
  if (do_draw)
    if (matconv[a].numdata)
      GPU_interleaved_attrib_setup(buffer, matconv[a].datatypes, matconv[a].
    numdata, max_element_size);
    GPU_buffer_draw_elements(dm->drawObject->triangles, GL_TRIANGLES,
    dm->drawObject->materials[a].start, dm->drawObject->materials[a].
    totelements);
    if (matconv[a].numdata)
      GPU_interleaved_attrib_unbind();
```

3.14.6 Render

Se $dt = OB\_RENDER$, *Blender* esegue un *rendering* di *preview* sulla scena o sulla porzione di scena selezionata dall'utente. Quest'operazione è eseguita dalla funzione *view3d_main_region_draw_engine*: essa individua dapprima il motore di *rendering* impostato in *Blender*, quindi imposta l'area da renderizzare come quella attualmente inquadrata nella *viewport* ed eventualmente ristretta al rettangolo (*Render Border*) inserito dall'utente. Infine viene eseguito il *rendering* sull'area calcolata.

```
RenderEngine *engine;
type = RE_engines_find(scene->r.engine);
if (!(type->view_update && type->view_draw))
  return false;
...
view3d_main_region_setup_view(scene, v3d, ar, NULL, NULL, NULL);
...
if (clip_border)
  if (border_rect->xmax > border_rect->xmin && border_rect->ymax >
    border_rect->ymin)
    glGetIntegerv(GL_SCISSOR_BOX, scissor);
    glScissor(border_rect->xmin, border_rect->ymin,
    BLI_rcti_size_x(border_rect), BLI_rcti_size_y(border_rect));
  else
    return false;
```

```
15 ...
16 type = rv3d->render_engine->type;
17 type->view_draw(rv3d->render_engine, C);
```

3.15 Rendering

Il principale algoritmo di *rendering* implementato in *Blender* è detto *Path Tracing*, il quale calcola le riflessioni dei raggi luminosi per verificare la quantità di luce che irradia ogni oggetto. *Blender* mette a disposizione anche una variante di questo algoritmo, detta *Branched Path Tracing*.

3.15.1 Path Tracing

Il *Path Tracing* è una tecnica di *rendering* 3D che integra le singole fonti luminose presenti sulla scena per verificare quali punti delle superfici degli oggetti sono illuminate. In particolare, i raggi luminosi che attraversano la scena vengono riflesse dalle superfici colpite e l'illuminazione di un oggetto dipende dalla sua capacità di riflettere la luce e dalla quantità di luce riflessa verso la camera. Questa procedura è pertanto *view dependent*, ossia l'aspetto di ciascun oggetto dipende dall'angolo di vista e dalla posizione della camera.

L'algoritmo implementato in *Blender* costruisce dapprima una gerarchia *BVH* (*Bounding Volume Hierarchy*), una struttura dati ad albero che organizza in modo gerarchico le geometrie della scena, raggruppandole in rettangoli (*bounding volumes*) che rappresentano le foglie dell'albero. Ricorsivamente questi rettangoli sono racchiusi in rettangoli più grandi, fino a avere un unico grande *bounding volume* nella radice dell'albero. I *BVH* sono usati spesso in algoritmi di *rendering* per escludere alcune geometrie localizzate in diversi *bounding volumes* da quello intersecato dal singolo raggio.

Costruita la struttura *BVH*, si attraversa l'albero ricorsivamente per individuare la foglia corrispondente all'oggetto "colpito" dal raggio luminoso.

```
1 Node *stack[MAX_STACK_SIZE];
2 int hit = 0, stack_pos = 0;
```

```
3
4  if (!TEST_ROOT && !is_leaf(root))
5    bvh_node_push_childs(root, isec, stack, stack_pos);
6  else
7    stack[stack_pos++] = root;
8
9  while (stack_pos)
10   Node *node = stack[--stack_pos];
11   if (!is_leaf(node))
12     if (bvh_node_hit_test(node, isec))
13       bvh_node_push_childs(node, isec, stack, stack_pos);
14       assert(stack_pos <= MAX_STACK_SIZE);
15   else
16     hit |= RE_rayobject_intersect( (RayObject *)node, isec);
17     if (SHADOW && hit) return hit;
18 return hit;
```

A questo punto, l'algoritmo procede effettuando il *rendering* per parti, cioè eseguendo il calcolo della luminosità degli oggetti in una porzione rettangolare (*tile* o *RenderPart*) dell'intera immagine (*RenderResult*). Ogni *tile* completamente renderizzato produce un *RenderResult*; ogni volta che viene completato il *rendering* di un *tile*, il suo *RenderResult* risultante viene fuso con quelli precedentemente calcolati, fino alla produzione di un unico *RenderResult* che corrisponde all'intera scena inquadrata dalla camera.

Il *rendering* di un *tile* viene eseguito dalla funzione *render_result_new*, mentre la fusione dei vari *RenderResult* è eseguita da *render_result_merge*. La fusione avviene iterando sui vari livelli di *rendering* (*RenderLayer*) impostati dall'utente e iterando per ogni *RenderLayer* sui *RenderPart* corrispondenti. Ogni *RenderResult* di un *tile* viene fuso con il *RenderResult* generale dalla funzione *do_merge_tile*, la quale semplicemente ricalcola le dimensioni del *RenderResult* generale per includere il nuovo *tile* elaborato.

```
1  RenderLayer *rl, *rlp;
2  RenderPass *rpass, *rpassp;
3
4  for (rl = rr->layers.first; rl; rl = rl->next)
5    rlp = RE_GetRenderLayer(rrpart, rl->name);
6    if (rlp)
7      for (rpass = rl->passes.first, rpassp = rlp->passes.first; rpass &&
       rpassp; rpass = rpass->next)
```

```
8         if (strcmp(rpassp->fullname, rpass->fullname) != 0)
9           continue;
10        do_merge_tile(rr, rrpart, rpass->rect, rpassp->rect, rpass->channels);
11        rpassp = rpassp->next;
```

3.15.2 Branched Path Tracing

La versione *branched* dell'algoritmo *Path Tracing* è simile a quella originale, con la differenza sostanziale che il raggio luminoso viene dapprima suddiviso in tanti raggi quante sono le componenti *shader* che si vuole simulare, quindi ogni raggio viene gestito come un *sample* indipendente. Per questo motivo, è più lento dell'algoritmo originale, ma può portare a risultati migliori.

Questo algoritmo permette di avere maggiore controllo sugli *shader* di una superficie; infatti, l'utente può definire il numero di campioni da dedicare per la resa di ogni componente *shader* dei materiali impiegati. Ad esempio, se i materiali utilizzati nel progetto sono costituiti principalmente da elementi *Diffuse*, si può incrementare il numero di *samples* per il corrispondente *shader*, in modo da dedicare la maggior parte della computazione per la resa di tale componente.

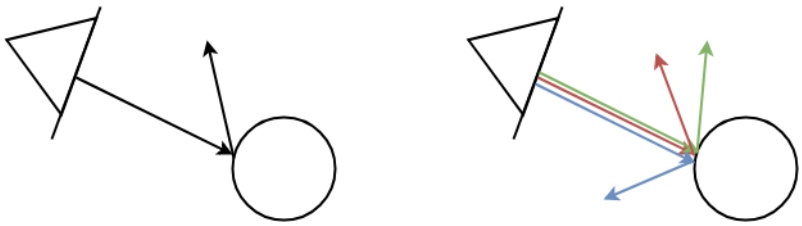


Figura 3.8: *Path Tracing* (a sinistra) e *Branched Path Tracing* (a destra)

Conclusioni

La realizzazione del modello 3D descritta in questo elaborato, che costituisce la naturale prosecuzione dell'attività di tirocinio svolta durante il percorso di studi, ha consentito di conoscere l'intero processo di sviluppo di un prodotto 3D, dalle fasi preliminari di acquisizione e modellazione di *mesh*, alla resa delle stesse fino alla produzione di un filmato. Riprendendo le conoscenze acquisite durante il tirocinio, il lavoro svolto approfondisce tali nozioni, fornendo anche una trattazione dei dettagli implementativi delle varie tecniche di modellazione grafica utilizzate.

Questo elaborato può essere quindi un valido ausilio per uno studente o appassionato di computer grafica che vuole impiegare le conoscenze acquisite nei propri studi in alcuni software *open source*, verificando da un lato come alcune operazioni possano essere eseguite su tali software, dall'altro quali siano gli algoritmi che implementano le rispettive funzionalità. In questo senso, il presente documento si pone come un punto d'incontro tra gli aspetti teorici sulla grafica 3D e alcune loro applicazioni pratiche.

La difficoltà principale nella stesura di questa tesi è stata la comprensione di alcuni porzioni del codice implementativo di *Blender*. Se da un lato le documentazioni di *Regard3D* e *MeshLab* definiscono chiaramente quali siano gli algoritmi impiegati, fornendo tra l'altro gli articoli di ricerca utili per la loro interpretazione, dall'altro si ha scarsa documentazione sugli algoritmi utilizzati da *Blender*, nonostante sia disponibile un suo manuale tecnico completo, la cui dimensione però può spesso essere fuorviante. Come già detto, questo documento non è da considerare come un guida esaustiva per l'interpreta-

zione dei tre programmi utilizzati, ma come un esempio di attività con cui coniugare la teoria e la pratica.

Alcuni possibili sviluppi futuri di questo elaborato possono riguardare, ad esempio, l'applicazione di un *game engine* per la produzione di una visita interattiva nella scena 3D, oppure l'utilizzo di un software per la stampa 3D del modello realizzato. Ad esempio, *Blender* mette a disposizione sia un *game engine* per la gestione dell'interattività, come anche un Add-on, detto "*3D Print Toolbox*", per la verifica della stampabilità di una *mesh*. Essendo *Blender* un software *open source* in tutte le sue componenti, l'utilizzo di queste due tecniche può fornire anche il giusto pretesto per lo studio di tali funzionalità e delle loro implementazioni.

Bibliografia

[1] Callieri M. *3D from photos. Automatic dense photogrammetry*. In *3D Digitalization for Cultural Heritage*

[2] Cignoni P., Callieri M., Corsini M., Dellepiane M., Gavonelli F. and Ranzuglia G. (2008). *MeshLab: an Open-Source Mesh Processing Tool*. In *Sixth Eurographics Italian Chapter Conference*, pp. 129-136

[3] Alcantarilla P.F., Bartoli A. and Davison A.J. (2012). *KAZE features*. In *Fitzgibbon A., Lazebnik S., Perona P., Sato Y., Schmid C. (eds) Computer Vision – ECCV 2012. Lecture Notes in Computer Science*, n. 7577, Springer, Berlin, Heidelberg, pp. 1-8

[4] Andersson O. and Marquez S.R., (2016). *A comparison of object detection algorithms using unmanipulated testing images*. KTH Royal Institute of Technology in Stockholm, pp. 10-17

[5] Zhenhua W., Bin F. and Fuchao W. (2011). *Local Intensity Order Pattern for Feature Description*. In *ICCV '11 Proceedings of the 2011 International Conference on Computer Vision*, pp. 603-610

[6] Silpa-Anan C. and Hartley R. (2008). *Optimised KD-trees for fast image descriptor matching*. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1-8

[7] Muja M. and Lowe D.G. (2014). *Scalable Nearest Neighbor Algorithms for High Dimensional Data*. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 36, pp. 1-7

[8] Muja M. and Lowe D.G. (2012). *Fast Matching of Binary Features*. In *2012 Ninth Conference on Computer and Robot Vision*, pp. 404-410

[9] Wen L., Ying Z., Yifang S., Wei W., Wenjie Z. and Xuemin L. (2016). *Approximate Nearest Neighbor Search on High Dimensional Data - Experiments, Analyses, and Improvement*. In *CoRR*, pp. 1-6

[10] Wei D., Moses C. and Kai L. (2011). *Efficient k-nearest neighbor graph construction for generic similarity measures*. In *WWW 2011 – Session: Clustering*, pp. 1-4

[11] Hyvönen V., Pitkänen T., Tasoulis S., Jääsaari E., Tuomainen R., Wang L., Corander J. and Roos T. (2016). *Fast Nearest Neighbor Search through Sparse Random Projections and Voting*. In *IEEE International Conference on Big Data*, pp. 1-5

[12] Moulon P. and Monasse P. (2012). *Unordered feature tracking made fast and easy*

[13] Strupczewski A. and Czupryński B (2014). *3D reconstruction software comparison for short sequences*. In *Proc. SPIE 9290, Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2014*, pp. 1-6

[14] Moulon P., Monasse P. and Marlet R. (2012). *Adaptive Structure from Motion with a contrario model estimation*. In *Lee K.M., Matsushita Y., Rehg J.M., Hu Z. (eds) Computer Vision – ACCV 2012.*, pp. 1-9

[15] Wilson K. and Snavely N. (2014). *Robust Global Translations with 1DSfM*. In *ECCV*, pp. 1-9

[16] Furukawa Y., Curless B., Seitz S.M. and Szeliski R. (2010). *Towards Internet-scale Multi-view Stereo*. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1-6

[17] Furukawa Y. and Ponce J. (2010). *Accurate, Dense, and Robust Multi-View Stereopsis*. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 32, pp. 1362-1376

[18] Goesele M., Snavely N., Curless B., Hoppe H. and Seitz S.M. (2007). *Multi-View Stereo for Community Photo Collections*. In *Proceedings of ICCV 2007*, pp. 1-6

[19] Bernardini F., Mittleman J., Rushmeier H., Silva C. and Taubin C. (1999). *The Ball-Pivoting Algorithm for Surface Reconstruction*. In *IEEE Transactions on Visualization and Computer Graphics*, v. 5, n. 4, pp. 349-359

[20] Kazhdan M., Bolitho M. and Hoppe H. (2006). *Poisson Surface Reconstruction*. In *Eurographics Symposium on Geometry Processing*, pp. 1-6

[21] Arvo J., Euranto A., Järvenpää L., Lehtonen T. and Knuutila T. (2015). *3D mesh simplification. A survey of algorithms and CAD model simplification tests*. In *University of Turku Technical Reports*, n. 3, pp. 5-17

[22] Garland M. and Heckbert P.S. (1997). *Surface simplification using quadric error metrics*. In *SIGGRAPH*, pp. 1-4

[23] Londei M. (2015). *Motori di Rendering a confronto in Blender: Teoria e Applicazioni*. In *Alma Mater Studiorum - Università di Bologna*, pp. 1-46

[24] Documentazione Regard3D, *http://www.regard3d.org/index.php*

[25] Documentazione MeshLab, *http://www.meshlab.net/*

[26] Documentazione Blender, *https://docs.blender.org/manual/en/latest/index.html*

Ringraziamenti

Un ringraziamento particolare al professore Giulio Casciola, relatore della tesi, per avermi supportato durante la realizzazione di questo lavoro e per avermi fatto conoscere e apprezzare il mondo della *computer graphics*.

Ringrazio l'azienda "*AdArte di Luca Mandolesi & C. s.n.c.*" per la loro disponibilità ad accettarmi come tirocinante e per la collaborazione durante tutto il periodo di tirocinio.

Ringrazio inoltre il dottor Valerio Velino per la sua costante disponibilità e tutto il personale di segreteria.

Ringrazio infine i colleghi di studio Gabriele, Oronzo, Cataldo e Francesco per i bei momenti passati insieme.