



SAPIENZA
UNIVERSITÀ DI ROMA

Fast Digital Sculpting on CPU

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di Laurea in Informatica

Candidato

Andrea Cipollini
Matricola 1804818

Relatore

Prof. Fabio Pellacini

Correlatore

Dr. Giacomo Nazzaro

Anno Accademico 2019/2020

Fast Digital Sculpting on CPU

Tesi di Laurea. Sapienza – Università di Roma

© 2020 Andrea Cipollini. Tutti i diritti riservati

Questa tesi è stata composta con L^AT_EX e la classe Sapthesis.

Versione: 7 dicembre 2020

Email dell'autore: andrea.cipollini98@gmail.com

*Un pensiero speciale a
Gigi Proietti*

Sommario

Il seguente lavoro di Tirocinio Triennale si sviluppa nell'ambito forse più interdisciplinare dell'Informatica, e cioè quello della Computer Grafica. Essa infatti raccoglie in sè molte discipline diverse, come ad esempio l'Algebra, la Geometria e la Fisica, che sono poi sfruttate in vari modi dall'Informatica vera e propria. Tuttavia, quello che distingue la Computer Grafica da altri "rami" dell'Informatica è la sua applicazione più multimediale ed artistica, quindi appunto interdisciplinare, rispetto ad altri ambiti reali di applicazione dell'Informatica. A conferma di ciò, è possibile verificare la sempre crescente diffusione dell'utilizzo di strumenti grafici soprattutto tra coloro che non sono né esperti né tanto meno studiosi della disciplina, ma che al contrario sono spesso perfetti estranei alle meccaniche che sono al di sotto dei supporti che utilizzano: sono veri e propri artisti, che sfruttano la propria capacità di utilizzo di questi strumenti esattamente come un pittore sfrutta la sua capacità di utilizzare il pennello ed il colore sulla tela che sta dipingendo. Nel caso della Computer Grafica, gli strumenti che hanno a disposizione gli artisti sono i software di modellazione, di rigging (Skeletal Animation), di animazione, di montaggio video, di composizione e di rendering sia bi-dimensionali (2D) che tri-dimensionali (3D), che ad esempio permettono di dare forma alle idee modellando oggetti virtuali in uno spazio virtuale, come si farebbe banalmente disegnando su un foglio di carta, e permettono anche ai professionisti di realizzare ciò di cui hanno bisogno, il che potrebbe risultare molto più oneroso se fatto con altri strumenti. E' su questa realtà che si basa il presente progetto, sulla reale necessità di questi strumenti anche nella vita quotidiana e sulla loro effettiva possibilità di realizzazione tramite l'applicazione di leggi, formule e algoritmi di varie discipline che solo in apparenza sono non più che teoretiche.

Ovviamente sono molteplici le tipologie di strumenti sviluppati nell'ambito della Computer Grafica, ed il presente lavoro riguarda la tipologia che simula, forse nel modo più realistico rispetto alle altre tipologie, la modellazione di forme "a mano libera": il Digital Sculpting, ovvero la Scultura Digitale. Il nome non è attribuito a questa pratica casualmente, ma è autoesplorativo e rende quindi l'idea della sua funzione e del suo scopo: riportare la tecnica reale della scultura in uno spazio virtuale, che offre la libertà di creazione che un artista tradizionale ha con la creta, piuttosto che con l'argilla o con qualsiasi altro materiale, anche agli artisti digitali. Oggi, infatti, hanno ampio spazio sulla vita quotidiana tutti i derivati di queste tecniche, come può essere la CGI (Computer-Generated Imagery) nell'industria cinematografica, la Real-Time Graphics nel campo dei videogiochi, o ancora tutti quei software adibiti all'animazione piuttosto che alla pubblicità o all'industria manifatturiera.

Lo scopo di questa relazione di Tirocinio è presentare il processo che ha portato allo sviluppo e alla creazione di un piccolo software di Digital Sculpting che offrisse gli strumenti basilari di software più completi e più complessi che si trovano nell'ampio panorama di programmi analoghi, che vengono utilizzati in contesti reali, implementando algoritmi noti della disciplina e adattandoli per l'implementazione. Questi algoritmi, come si vedrà, sono alla base delle funzionalità fondamentali di

tutti i software di Digital Sculpting e sono ricavati da studi che li descrivono teoricamente; sono poi implementati a seconda del Render Engine del software che si sta realizzando. In particolare, tutto il lavoro che successivamente verrà descritto si basa sul Render Engine Yocto/GL [3], una libreria per la Computer Grafica sviluppata dallo stesso relatore di questa relazione di Tirocinio.

Un altro punto focale di questo lavoro è sicuramente l'attribuzione al software sviluppato di un requisito di calcolo Real-Time. Difatti, il fine ultimo di un programma di Digital Sculpting è la replica di una tecnica, quale la modellazione, in uno spazio virtuale, tale che possa essere utilizzata da un artista come farebbe in un contesto reale, senza quindi limitazioni di sorta; ed è comprensibile come una risposta visiva che arrivi all'utente con un certo ritardo dal suo input (ritardo dovuto ai tempi di calcolo del software) rappresenti un limite enorme per una tipologia di software come il Digital Sculpting. Per questo motivo molti aspetti dell'implementazione del software oggetto di questa trattazione considerano questo requisito come fondamentale per la buona riuscita del progetto, che si fa presente essere sviluppato completamente su CPU, il che è una scelta di progettazione che limita intrinsecamente il raggiungimento della computazione Real-Time: solitamente i calcoli che necessitano applicativi come il Digital Sculpting (in generale i software di Computer Grafica) sono accelerati da hardware specifico, ovvero dalle GPU (Graphics Processing Unit). L'obiettivo di questo lavoro di Tirocinio è stato, al contrario, proprio quello di sviluppare un software di Digital Sculpting (seppur di dimensioni contenute) che potesse essere eseguito praticamente in Real-Time esclusivamente su CPU, senza il supporto di hardware specifico.

Al termine della descrizione del processo di sviluppo del software oggetto di questa trattazione, che è stato chiamato *ysculpting*, verranno esaminati i risultati raggiunti e verranno tratte le conclusioni estrapolate da questa esperienza.

Ringraziamenti

Durante questi ultimi tre anni per me ci sono stati alti e bassi, non solo nella vita universitaria ma spesso e volentieri anche nella vita di tutti i giorni. Se sono riuscito ad andare oltre le difficoltà è perché ho avuto basi solide, a partire dalla mia famiglia fino ad arrivare a vecchie amicizie ritrovate.

Ringrazio mamma e papà, perché mi hanno permesso di continuare gli studi con i loro sforzi quotidiani, perché mi hanno sempre sostenuto, e perché so che continueranno a farlo.

Ringrazio Agnese e Sara, che mi hanno fatto capire il valore della perseveranza, riprendendo gli studi tra molte difficoltà.

Ringrazio Alessandra, che mi ha insegnato l'importanza di inseguire le proprie passioni; ringrazio Marco e Francesca, che sono diventati per me la massima fonte d'ispirazione.

Ringrazio Stefano e Pietro, i miei migliori amici, i miei confidenti, che mi hanno sempre supportato e sopportato.

Ringrazio i miei amici più cari: il Diurbi, Francesco, Emanuele, Daniele e Michele, che sono sempre riusciti a farmi dimenticare il peso delle difficoltà; Luca, Monica e Simona, che hanno intrapreso il percorso insieme a me e lo hanno reso più leggero e accogliente; Flavio, compagno di viaggi, di interessi, di momenti felici e di sventure.

Ringrazio i miei colleghi Simone, Alessandro, Ilaria, Marco e Alessandra: con loro ho condiviso gioie, faide con i professori, confronti, disorganizzazione cronica dei progetti ed ogni tanto (fortunatamente) qualche esame.

Infine ringrazio Luca, Samuele e Luca, vecchi amici ritrovati lungo la via, insieme ai quali ho raggiunto il traguardo.

A tutti voi, Grazie.

Indice

1 Introduzione	1
1.1 Rappresentare il Mondo	1
1.1.1 Camera Virtuale	1
1.1.2 Forme	2
1.1.3 Materiali	3
1.2 Rendering	4
1.3 Modellazione 3D	4
2 Digital Sculpting: elementi di base	7
2.1 Stroke	9
2.1.1 Bounding Volumes	13
2.2 Brush	14
2.2.1 Hierarchical Spatial Hash Grid	15
2.2.2 Multithreading	16
2.2.3 Applicazione del Brush	16
2.3 Interfaccia Utente	18
2.4 Stroke Segmentation	20
3 Gaussian Brush	23
3.1 Densità di probabilità normale	24
3.2 Applicazione al Digital Sculpting	25
4 Opzioni del Brush	33
4.1 Negative Brush	33
4.2 Continuous Brush	34
4.3 Symmetric Brush	36
4.4 Saturation Brush	37
5 Texture Brush	41
5.1 Stroke Parameterization	42
5.1.1 Upwind Averaging	42
5.1.2 Frame-Propagation	44
5.1.3 Combinazione delle modifiche al DEM	44
5.1.4 Parametrizzazione e implementazione	46
5.1.5 Il Geodesic Solver	50
5.2 Dalla texture al Brush	51

6 Smooth Brush	53
6.1 Laplacian Smoothing	54
6.1.1 Problematiche del metodo di smoothing	55
6.2 Cotangent Weights	56
6.3 Applicazione	58
7 Conclusioni	61

Capitolo 1

Introduzione

Prima di iniziare a descrivere il processo che ha portato allo sviluppo del software di Digital Sculpting *ysculpting*, si introducono dei concetti di Computer Grafica basilari, indispensabili alla completa comprensione di tutti i passaggi che verranno di qui in avanti esplicati e che sono parte integrante della libreria Yocto/GL [3], utilizzata come framework di partenza per l'implementazione dello Sculpter nel linguaggio C++.

1.1 Rappresentare il Mondo

La disciplina della Computer Grafica cerca di replicare il mondo reale tramite delle rappresentazioni accurate in un mondo virtuale creato sulla macchina; le rappresentazioni sono racchiuse in "scene", ovvero una collezione di elementi, e gli elementi che stanno alla base delle scene sono essenzialmente i seguenti: camere virtuali, forme, materiali e animazioni (in alcuni ambiti).

1.1.1 Camera Virtuale

La camera virtuale è l'elemento della scena che permette di vedere gli oggetti che la compongono, e dunque proietta ciò che rientra nel suo campo di visione su un piano, perché lo si possa vedere attraverso il display del computer e perché si possa interagire con lo spazio virtuale. La camera virtuale proietta il contenuto della scena su un piano attraverso delle proiezioni ortogonali di vari tipi, ma quella che simula al meglio una camera reale è la proiezione ortogonale prospettica, che è anche quella utilizzata in *ysculpting*. La camera virtuale è definita a partire da una camera reale: possiede una lente all'origine di un sistema di riferimento che ne definisce direzione e orientamento, sul retro della quale è posizionato un sensore virtuale che possiede dimensioni di sensori analoghi reali, che insieme creano una piramide di visione e un piano della messa a fuoco della lente, che è il piano sul quale verrà costruita l'immagine. Il sistema di riferimento sul quale è posizionata la lente è costruito a partire dalla posizione della camera (origine o), dalla direzione di visione (l'asse z capovolto) e dalla direzione "world up", cioè verso l'alto (asse y), mentre l'ultimo asse x è definito per ortogonalità.

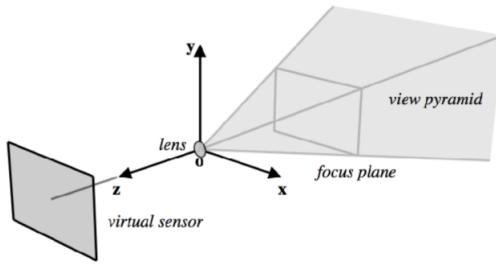


Figura 1.1. La struttura della camera virtuale
(Fabio Pellacini)

1.1.2 Forme

Le forme sono la parte fondamentale per la rappresentazione di oggetti nello spazio virtuale, rappresentazione che se fosse diversa per la descrizione di oggetti diversi comporterebbe una complessità elevata del software per la gestione di diverse tipologie di algoritmi applicabili a diverse rappresentazioni, oltre che per la gestione di diversi tipi di ottimizzazione che sarebbero necessari. Quindi si utilizzano il minor numero di rappresentazioni possibile, tramite l'utilizzo di soli punti, linee e triangoli. I punti sono singoli vertici nello spazio, cui corrispondono quindi tre coordinate, e sono utilizzati per descrivere forme o effetti particolari, soprattutto utilizzandone molti insieme; le linee sono definite da due vertici che ne delimitano inizio e fine, ma per le quali si utilizza una combinazione lineare dei vertici stessi per definirne tutti i punti intermedi:

$$p(t) = (1 - t)v_0 + tv_1 \quad (1.1)$$

con $t \in [0, 1]$. L'utilizzo principale delle linee riguarda capelli, pelliccia ecc... I triangoli sono definiti da tre vertici e le loro collezioni sono utilizzate per approssimare le superfici perché definiscono ognuno un piano, e dunque il loro utilizzo in gran numero può approssimare in maniera più o meno accurata (a seconda del numero di triangoli della collezione) moltissimi tipi di superficie. Anche per i triangoli, ogni punto di essi al di fuori dei vertici può essere definito algebricamente tramite combinazione lineare dei vertici stessi con l'utilizzo di "pesi" chiamati coordinate baricentriche, espresso come:

$$p(w_1, w_2) = v_0 + w_1(v_1 - v_0) + w_2(v_2 - v_0) \quad (1.2)$$

La superficie formata da triangoli connessi è chiamata triangles mesh, ed ogni singolo vertice ha associate delle coordinate (x, y, z) rispetto al sistema di riferimento globale che ne definiscono la posizione all'interno dello spazio virtuale. Per i triangles meshes ci sono anche due diverse definizioni per la topologia e per la geometria: la topologia descrive le modalità di connessione tra i triangoli del mesh, mentre la geometria indica la forma creata dalla collezione di triangoli, quindi la forma dell'oggetto vero e proprio (figura 1.2).

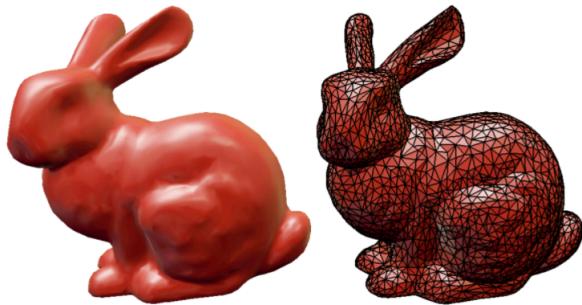


Figura 1.2. Esempio di approssimazione di una superficie tramite una collezione di triangoli, ovvero un triangles mesh
(Fabio Pellacini)

1.1.3 Materiali

Oltre alla forma degli oggetti sono importanti le modalità di interazione della luce con essi per una buona simulazione di oggetti reali nello spazio virtuale. Nel mondo reale la luce si comporta in maniera diversa con oggetti diversi quando sono composti di materiali differenti, e anche nel caso della Computer Grafica è importante riprodurre materiali diversi per ottenere effetti simili a quelli reali. Ci sono tre comportamenti principali che vengono simulati: la diffusione del colore, i riflessi di colore e la nitidezza di queste riflessioni. Questi comportamenti dipendono in particolare dal colore della superficie, da un eventuale texture applicata alla superficie (vedere avanti) e dal vettore normale (cioè il vettore perpendicolare al piano tangente in un certo punto) alla superficie nei punti in cui la luce incontra il mesh.

Il colore può essere applicato alla superficie tramite la definizione "manuale" dei colori per ogni vertice del mesh oppure in maniera automatica tramite l'applicazione di textures: le textures sono immagini che vengono utilizzate per essere applicate ai meshes tramite la definizione di corrispondenze tra ogni vertice del mesh ed ogni rispettivo punto sulla texture. I punti della texture corrispondenti ai vertici del mesh sono individuati tramite una parametrizzazione della texture in un intervallo $[0, 1]^2$, che attribuisce ad ogni punto dell'immagine delle coordinate (u, v) nell'intervallo, e sono chiamate texture coordinates. Per essere applicate all'intero mesh, per prima cosa vengono associate le texture coordinates ai vertici del mesh, e poi anche ai punti sui triangoli, individuati tramite interpolazione baricentrica: le texture coordinates (u, v) associate ad un certo punto con coordinate baricentriche (w_1, w_2) di un triangolo, per il quale sono già state definite le texture coordinates (u_0, v_0) , (u_1, v_1) e (u_2, v_2) ai vertici, sono date da:

$$u = \sum w_i u_i \quad v = \sum w_i v_i \quad (1.3)$$

Una volta associate le texture coordinates al mesh, si valutano i valori corrispondenti della texture che è stata parametrizzata anch'essa nell'intervallo $[0, 1]^2$. La texture però può avere dimensione più piccola o più grande del mesh sul quale di applica, dunque le coordinate (u, v) del mesh saranno adattate alla dimensione della texture:

$$\begin{aligned} s &= (u\%1) * \text{width} \\ t &= (v\%1) * \text{height} \end{aligned} \tag{1.4}$$

1.2 Rendering

L'operazione di "disegno" della scena sul piano dell'immagine, che corrisponde al piano di messa a fuoco della lente della camera virtuale, tramite una proiezione ortogonale di qualche tipo prende il nome di rendering. Questo avviene, nel caso di *ysculpting*, secondo la tecnica di Ray-Tracing, la cui idea di base è tracciare per ogni punto dell'immagine finale (per ogni pixel) uno o più raggi nella direzione della scena, e nel caso questi intersechino qualche oggetto si calcola il colore di questo punto intersecato (colore + texture + illuminazione) e si mette il colore così ottenuto nel pixel da dove è partito il raggio. Nel fare ciò, il piano dell'immagine è parametrizzato nell'intervallo $[0, 1]^2$, in modo tale che si possano far passare i raggi in punti specifici del piano dell'immagine. Un raggio come quelli citati è una semi-linea infinita che ha un origine e ed una direzione d , più due valori t_{min} e t_{max} che indicano la distanza minima e massima dall'origine e lungo la direzione d che delimitano una frazione della semi-linea infinita da considerare: ogni punto del raggio è definibile in funzione della distanza t dall'origine:

$$p(t) = e + td \tag{1.5}$$

Ognuno dei raggi tracciati per il rendering ha l'origine nella posizione della lente, passa per un punto del piano dell'immagine e calcola la sua intersezione con un oggetto della scena (se avviene): poiché gli oggetti della scena sono triangles meshes, l'intersezione del raggio avviene con uno dei triangoli dell'oggetto, e quindi i calcoli effettuati per la definizione del colore del pixel avvengono su questo punto intersecato, tramite l'utilizzo delle coordinate baricentriche. Quando tutti i punti del piano dell'immagine avranno i colori associati, si sarà formata l'immagine.

1.3 Modellazione 3D

Sin qui si sono definite le componenti principali di uno spazio virtuale tri-dimensionale ed è chiaro che la componente più importante in un software di Digital Sculpting siano gli oggetti 3D, e ancora più importanti i relativi triangles meshes. Tutti i software di modellazione infatti operano sui meshes degli oggetti tri-dimensionali, poiché il loro scopo è quello di modificarne la geometria per cambiare forma all'oggetto nel suo insieme, e contemporaneamente di modificarne la topologia in quei casi in cui è necessario gestire appositamente l'interazione del mesh con la luce. Infatti, le modifiche alla topologia di un mesh variano tutti i dati che sono memorizzati sui vertici dei triangoli del mesh, come texture coordinates, normali ecc..., il cui impatto principale è il cambiamento del comportamento rispetto all'illuminazione: la topologia controlla però anche la risoluzione di un mesh (con risoluzione di un mesh si intende la qualità dell'approssimazione della superficie, e quindi il numero dei triangoli), perché può definire il numero di triangoli presenti in aree diverse del mesh. Il controllo della topologia però è solitamente prerogativa dei software di

modellazione 3D classici, ovvero tutti quei software che permettono la modifica di un mesh controllando direttamente tutte le sue componenti, come vertici e spigoli, attraverso un approccio molto tecnico. Al contrario, il Digital Sculpting propone un approccio alla modellazione 3D più simile alla modellazione di un oggetto reale, che non modifica il mesh in modo tecnico a partire dalle sue componenti ma che agisce attraverso la modifica di vertici in zone del mesh scelte arbitrariamente dall'utente in modo "trasparente" rispetto alla modellazione classica, cioè senza che l'utente si preoccupi della topologia dell'oggetto che sta modellando (senza cioè agire singolarmente sulle componenti del mesh). Questo tipo di modellazione è utile per la creazione di quella tipologia di oggetti 3D che simulino il tratto "a mano libera" dell'uomo.

Questo tipo di modellazione quindi agisce sulle posizioni dei vertici tramite trasformazioni che verranno esplicate nei capitoli successivi a seconda dell'effetto che l'utente vuole ottenere. Queste trasformazioni riguardano solitamente un numero molto elevato di vertici, perché la modellazione in Digital Sculpting richiede che il mesh abbia una buona risoluzione per avere risultati realistici, e dunque solitamente tutti i calcoli in virgola mobile sono effettuati su GPU, l'hardware adibito appositamente per fare ciò in Computer Grafica.

Capitolo 2

Digital Sculpting: elementi di base

Conclusa una doverosa introduzione agli elementi principali oggetto del lavoro nell'ambito della Computer Grafica, sono state descritte le nozioni sufficienti per entrare nel vivo della trattazione dello sviluppo del software di Digital Sculpting che è oggetto di questo lavoro di Tirocinio.

Per cominciare a definire e descrivere quali siano gli strumenti che è necessario fornire ad un software di Digital Sculpting, si può partire da un'analisi teorica della tecnica reale che si vuole emulare: la modellazione "a mano libera". Come visto nel capitolo precedente, esistono vari tipi di modellazione che differiscono molto l'uno dall'altro e di conseguenza ognuno di essi ha la necessità di fornire specifici strumenti all'utente, strumenti che possono risultare indispensabili per alcune tipologie ed inutili per altre. Sono necessari strumenti diversi non solo quando si parla di una differente tipologia di modellazione, ma banalmente anche nel caso in cui le modellazioni siano fatte in spazi virtuali di dimensioni diverse, ovvero bi-dimensional e tri-dimensional, o nel caso in cui si debba modellare uno stesso oggetto ma con modalità diverse, come mostrato nelle figure 2.1 e 2.2 che presentano un esempio di quanto detto nel software di modellazione Blender [4]: nella figura 2.1 l'utente può solo selezionare l'oggetto nella sua interezza e effettuare un certo tipo di operazioni su di esso, mentre nella figura 2.2 la tipologia di selezione permette la scelta di alcune parti specifiche del mesh e la possibilità di effettuare delle operazioni esclusivamente su di esse.

Nel caso di un software di Digital Sculpting il primo strumento da fornire all'utente è quello che gli permetta di interagire con l'oggetto che voglia modellare, cioè che gli permetta di selezionare una zona del mesh che voglia andare a "scolpire". Questo strumento deve "tracciare" in qualche modo il movimento dell'input dell'utente (che può essere il semplice mouse ma anche una tavoletta grafica o una penna digitale, le quali aumentano la componente di simulazione durante l'utilizzo) in modo tale che si possa riportare sull'oggetto una serie di modifiche, scelte anch'esse dall'utente, esattamente sulle porzioni di mesh interessate dalle azioni in input. Da ciò, si può poi derivare il secondo strumento che è necessario fornire in un software di Digital Sculpting, ovvero quello che permetta di definire quali siano i modificatori da

applicare al mesh. Nei software di Digital Sculpting più completi e complessi sono moltissime le tipologie di modificatori che si possono applicare al mesh una volta che si è stabilita la zona di applicazione; in *ysculpting* sono sviluppati i modificatori di base, cioè solo quelli più comuni ed utilizzati che permettono comunque ad un artista di avere un buon grado di libertà nelle operazioni che può effettuare. Un esempio di quanto sia elevato il numero di questi modificatori in software di alto livello è presentato nella figura 2.3, che oltre a ciò suggerisce anche quale siano altri strumenti fondamentali da fornire all'utenza in *ysculpting*: un'interfaccia che sia il più chiara possibile e un feedback visivo sulla posizione e parametri attuali dell'input dell'utente.

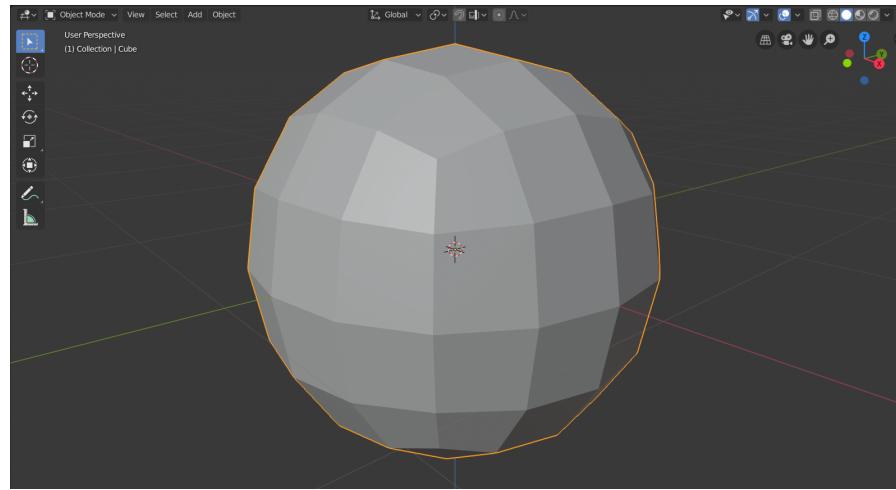


Figura 2.1. Blender: selezione completa dell'oggetto (in arancione) e gli strumenti disponibili (sulla sinistra)

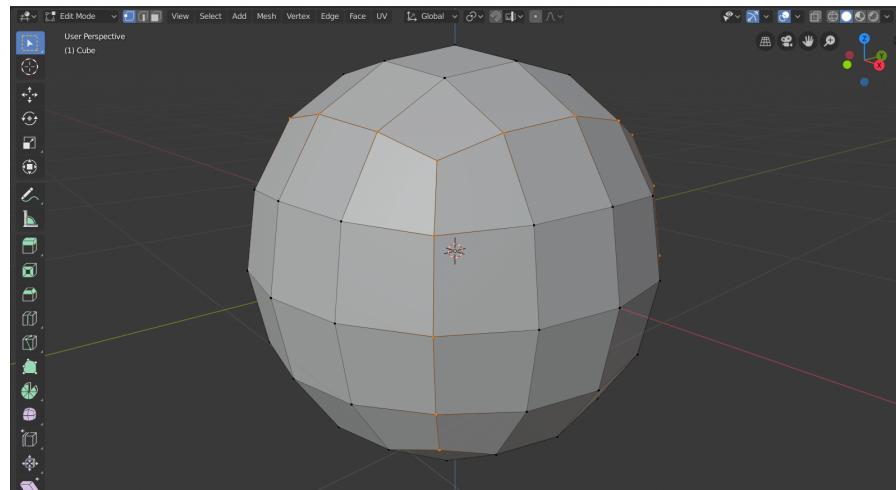


Figura 2.2. Blender: selezione di alcuni vertici e spigoli specifici (in arancione) e gli strumenti disponibili (sulla sinistra)

Infine, ma non meno importante, deve essere fornito il controllo della telecamera virtuale nello spazio virtuale, ed in particolare la possibilità di spostarla, allontanarla,

avvicinarla o ruotarla attorno all'oggetto per garantire all'utente la possibilità di modellarlo nella sua interezza.

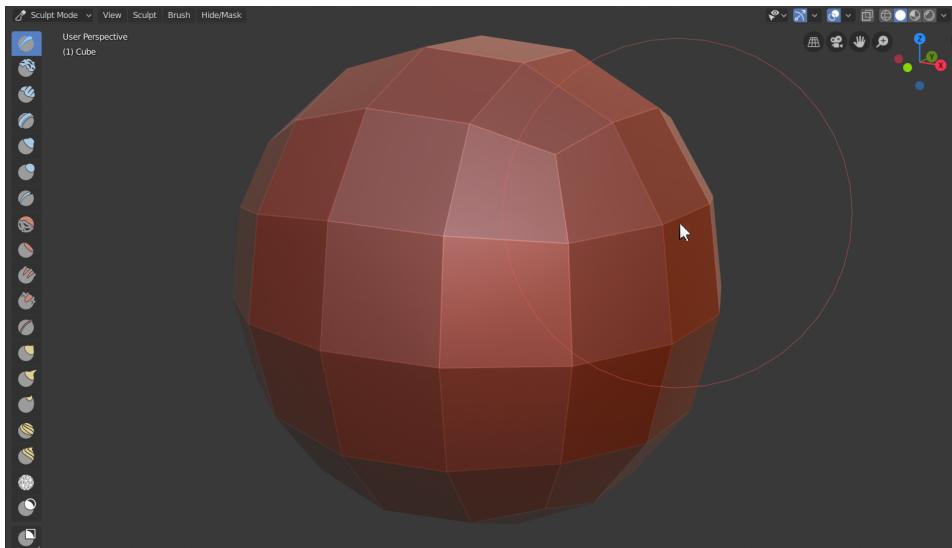


Figura 2.3. Blender: interfaccia di Digital Sculpting, con l'elenco dei modificatori o Brush (sulla sinistra) e la visualizzazione del puntatore del mouse con l'attuale parametro di raggio (la circonferenza in rosso)

2.1 Stroke

Ciò che è stato definito grezzamente come "tracciamento" dell'input dell'utente si traduce nell'ambito del Digital Sculpting con il termine Stroke, che in questo contesto assume il significato di "tratto". Difatti è esattamente questo che rappresenta, il tratto che l'utente compie sull'oggetto con il quale interagisce spostando il cursore, di qualsiasi tipo si tratti, e che rappresenta sul mesh esattamente il percorso creato dalla proiezione del cursore sul mesh stesso (sia in spazi virtuali 2D che 3D). Si parla di proiezione del cursore sull'oggetto perché, come spiegato nel Capitolo 1, la telecamera virtuale mostra in due dimensioni, cioè l'ordine di grandezza del display attraverso il quale si visualizza l'output dei computer, ciò che esiste in tre dimensioni nella scena virtuale, e questo tramite la proiezione ortogonale prospettica ottenuta proiettando dei raggi verso la scena 3D attraverso ogni pixel che compone la telecamera virtuale (nel caso del rendering basato su Ray-Tracing). Lo stesso principio è sfruttato per la gestione dell'interazione dell'utente col mesh: si proietta la posizione del cursore dallo spazio 2D della telecamera virtuale verso scena 3D attraverso la definizione di un raggio che parte dalla lente della camera e che passi per la posizione del cursore sul piano dell'immagine in direzione della scena 3D. Eventuali intersezioni di questo raggio con l'oggetto corrispondono alla posizione del cursore sulla superficie del mesh. Ovviamente il raggio in questione deve essere proiettato a partire dalla posizione corretta del cursore sul piano dell'immagine della telecamera virtuale, che però si ricorda essere parametrizzato nell'intervallo $([0, 1], [0, 1])$. Dunque, per ottenere le coordinate del cursore relative al piano, si

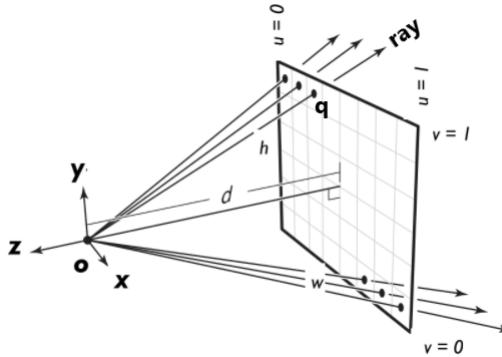


Figura 2.4. Rappresentazione della posizione di un punto q sul piano dell’immagine di una telecamera prospettica e visualizzazione del raggio passante per q e con l’origine in o (Fabio Pellacini)

prendano le coordinate in pixel (x, y) della posizione del cursore e le si dividano per la dimensione, sempre in pixel, della telecamera virtuale:

$$(u, v) = \frac{(x, y)}{(width, height)} = \left(\frac{x}{width}, \frac{y}{height} \right) \quad (2.1)$$

Una volta ottenute le coordinate del cursore parametrizzate nell’intervallo corretto, si crei il raggio che ha come origine la posizione della lente della telecamera virtuale, quindi l’origine del sistema di riferimento della telecamera in coordinate globali, e come direzione il vettore che parte dall’origine e che passa per il punto sul piano dell’immagine con le coordinate ottenute. Posto il piano dell’immagine ad una distanza d dall’origine o del sistema di riferimento della telecamera composto dagli assi x , y e z , data la posizione q del cursore con coordinate (u, v) sul piano dell’immagine, dato fov il field-of-view della telecamera, dato il parametro w tale che:

$$w = 2d \tan\left(\frac{fov}{2}\right) \quad (2.2)$$

le coordinate globali del punto q sono così ottenute:

$$q = o + (u - 0.5)wx + (v - 0.5)hy - dz \quad (2.3)$$

e quindi, seguendo la struttura definita nel Capitolo 1, il raggio è così definito:

$$ray(u, v) = \{o, \frac{q - o}{|q - o|}\} \quad (2.4)$$

con la direzione normalizzata (figura 2.4).

A questo punto si calcoli l’eventuale intersezione tra il raggio descritto e il mesh: in particolare l’intersezione tra il raggio ed i triangoli che compongono il mesh, attraverso un metodo algebrico che è meno preciso rispetto ad altri metodi ma più veloce, e come già spiegato la computazione Real-Time è fondamentale in un software

come *ysculpting*. Come descritto nel Capitolo 1, ogni punto del triangolo può essere algebricamente espresso tramite le sue coordinate baricentriche:

$$p(w_1, w_2) = v_0 + w_1(v_1 - v_0) + w_2(v_2 - v_0) \quad (2.5)$$

quindi risolvere l'intersezione significa dover risolvere il sistema di equazioni composto dal raggio e dai punti del triangolo:

$$\begin{cases} p(t) = e + dt \\ p(w_1, w_2) = v_0 + w_1(v_1 - v_0) + w_2(v_2 - v_0) \end{cases} \quad (2.6)$$

risolvibile per sostituzione scrivendo un sistema di tre equazioni in tre incognite in forma matriciale:

$$\begin{aligned} e + dt &= v_0 + w_1(v_1 - v_0) + w_2(v_2 - v_0) \\ &= \\ w_1(v_0 - v_1) + w_2(v_0 - v_2) + dt &= v_0 - e \\ &= \\ \begin{bmatrix} v_0 - v_1 & v_0 - v_2 & d \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ t \end{bmatrix} &= \begin{bmatrix} v_0 - e \\ \\ \end{bmatrix} \end{aligned} \quad (2.7)$$

Questo sistema scritto in forma matriciale è risolvibile con la regola di Cramer, che permette l'ottenimento della distanza sul raggio alla quale è avvenuta l'intersezione (t) e le coordinate baricentriche del punto intersecato sul triangolo (w_1 e w_2):

$$t = \frac{[(v_2 \times e_1)e_2]}{(d \times e_2)e_1} \quad w_1 = \frac{(d \times e_2)v_2}{(d \times e_2)e_1} \quad w_2 = \frac{(v_2 \times e_1)d}{(d \times e_2)e_1} \quad (2.8)$$

$$\text{con: } v_2 = e - v_0 \quad e_1 = v_1 - v_0 \quad e_2 = v_2 - v_0$$

Al termine del calcolo di t , w_1 e w_2 , l'intersezione è avvenuta se si verificano le seguenti condizioni:

$$t_{min} \leq t \leq t_{max} \quad \wedge \quad 0 \leq w_1 \leq 1 \quad \wedge \quad 0 \leq w_2 \leq 1 \quad \wedge \quad w_1 + w_2 \leq 1 \quad (2.9)$$

Completati questi calcoli, trovata la primitiva del mesh intersecata ed il punto di intersezione su di essa, successivamente è necessario che si calcoli questo stesso punto in coordinate globali; come si vedrà nei capitoli successivi le coordinate globali del punto intersecato servono ad applicare i modificatori del mesh dei quali si è accennato in precedenza. Dunque, avendo memorizzate tutte le coordinate globali dei tre vertici di tutti i triangoli del mesh e conoscendo qual è il triangolo che è stato intersecato, si può accedere ai valori di queste coordinate per il triangolo in questione, e poiché si sono ottenute le coordinate baricentriche w_1 e w_2 del punto di intersezione, lo stesso punto del quale vogliamo le coordinate globali, basti effettuare un'interpolazione lineare delle coordinate dei vertici rispetto alle coordinate baricentriche. L'interpolazione in questione è già stata espressa sotto forma di equazione con la formula 2.5, con la quale di esprime ogni punto del triangolo tramite le coordinate baricentriche. Dunque si ottengano le coordinate globali sostituendo in questa

formula i parametri w_1 e w_2 con le coordinate baricentriche calcolate nell'intersezione.

Si è ottenuto così il primo punto dello Stroke, che per definizione però è un tratto, cioè l'intero movimento del cursore e l'intero insieme delle intersezioni derivate da questo movimento. Idealmente quindi, lo Stroke dovrebbe essere un insieme infinito di punti di intersezione che vanno a formare la curva corrispondente al movimento del cursore sul mesh, ma questo ovviamente non è riproducibile sulla memoria limitata di un computer. Ciò che viene fatto quindi è un point sampling della curva dello Stroke, ovvero un campionamento dei punti che costituiscono lo Stroke sulla superficie del mesh, in modo tale da ottenere effettivamente una polilinea, che di fatto è un'approssimazione di una curva. Anticipando leggermente la prossima sezione, i modificatori del mesh sono operazioni che agiscono sui vertici di esso entro un certo raggio da un punto centrale, e quest'ultimo è proprio uno dei punti campionati sullo Stroke. Questa piccola anticipazione è necessaria perché si definisca come questi punti vengano campionati: un buon criterio utilizzato per decidere quando campionare un punto dello Stroke e quando invece ignorarlo è la distanza del punto in esame con l'ultimo punto campionato. Questa distanza deve essere tale da rendere la sequenza di circonferenze che rappresentano il raggio in cui agiscono i modificatori e centrate nei punti campionati un'approssimazione soddisfacente del tratto ideale che l'utente ha tracciato con il cursore. Sono stati fatti vari studi al riguardo, raccolti in vari articoli: questo lavoro di Tirocinio si è basato sul lavoro di Erika Jansson [2], che assume come distanza tra i punti campionati il 20/25% del raggio di azione dei modificatori applicati a partire dallo Stroke. In figura 2.5 sono visibili le differenze tra alcuni Strokes con distanze tra i punti centrali differenti.

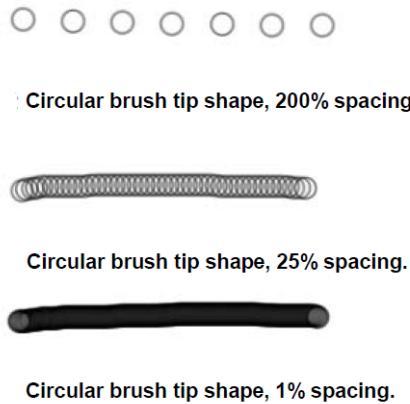


Figura 2.5. Esempi di spaziature diverse per lo Stroke (Erika Jansson)

L'implementazione dello Stroke segue esattamente quanto detto, con degli accorgimenti prettamente pratici, ed è descritto dall'Algoritmo 1. Questo algoritmo viene chiamato ad ogni aggiornamento della finestra del programma di *ysculpting*, che avviene più volte al secondo, e restituisce un nuovo punto che si aggiunge al campionamento dello Stroke oppure viene scartato.

Algorithm 1 stroke

```

1: Input: camera, mouse_uv, mesh, intersection, last_position, radius
2: Output: stroke_sample
3: position = eval_position(mesh, intersection.element, intersection.uv)
4: delta_position = distance(position, last_position)
5: stroke_distance = radius * 0.2
6: if last_position == null then
7:     last_position = position
8:     return null
9: end if
10: if delta_position >= stroke_distance then
11:     last_position = position
12:     return position
13: end if
14: return null

```

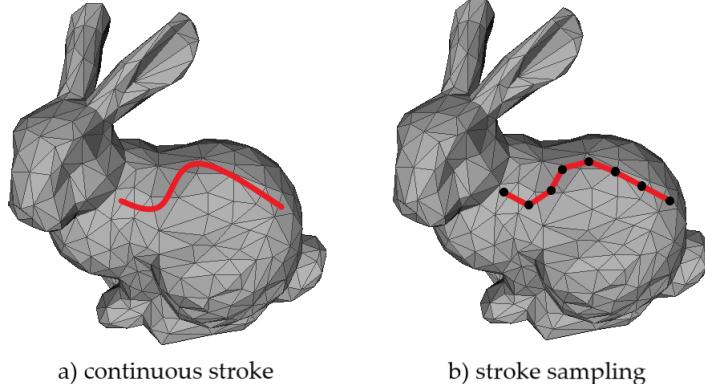


Figura 2.6. Esempi di Stroke su un triangles mesh: continuo (a) e campionato (b)
(base di Maks Ovsjanikov)

2.1.1 Bounding Volumes

Sin qui è stata presentata la teoria necessaria ad eseguire intersezioni tra raggi e triangoli del mesh. In realtà questa è una tecnica sì efficace ma poco efficiente, soprattutto nel caso di software che debba fornire output Real-Time e soprattutto se l'intersezione debba essere calcolata più volte al secondo (perché viene calcolata ad ogni aggiornamento della UI) e per tutti i triangoli del mesh. Infatti non si conosce in partenza la primitiva del mesh che verrà intersecata, e dunque occorrerà calcolare l'intersezione tra il raggio e tutti i triangoli del mesh, finché non si trovi quello intersecato. Dunque sarebbe altamente utilizzare questo metodo per l'intersezione tra raggio e mesh, anche perché il suo costo computazionale aumenterebbe con l'aumentare del numero di triangoli del mesh, un caso molto comune e anzi quasi obbligato dalla tecnica stessa di Digital Sculpting, per la quale si ottengono risultati migliori su mesh ad alta risoluzione. Per evitare che si calcoli un così alto numero di intersezioni sono utilizzate delle strutture di accelerazione chiamate Bounding

Volumes, che sono dei volumi che racchiudono gli oggetti: in particolare essi riducono drasticamente i costi della computazione di un'intersezione, soprattutto nella loro versione gerarchica, ovvero la Bounding Volume Hierarchy. Questa tipologia consiste nel racchiudere in volumi non solo l'oggetto nella sua interezza ma anche parti più piccole di esso, stabilendo una gerarchia tra Bounding Volumes contenitori e contenuti e permettendo così di ottenere la primitiva intersecata (se è effettivamente avvenuta un'intersezione) tramite intersezioni con questi volumi. Con la Bounding Volumes Hierarchy le intersezioni sono più semplici perché i volumi sono creati appositamente come geometrie semplici da intersecare e inoltre la gerarchia è definita in modo tale da far convergere le intersezioni verso la parte di mesh che è quella che potenzialmente contiene la primitiva intersecata, ignorando tutto il resto del mesh sul quale così non viene provata l'intersezione.

Grazie all'utilizzo di queste strutture, l'intersezione in *ysculpting* è calcolata su CPU quasi istantaneamente.

2.2 Brush

Stabilito l'algoritmo per la definizione dei punti del mesh sui quali l'utente voglia agire, si possono descrivere quelli che per semplicità sono stati chiamati precedentemente "modificatori". Il nome corretto in Computer Grafica per ognuno di essi è Brush, e la loro funzione è indicare la tipologia di modifica che debba essere effettuata nel momento della loro applicazione. Nei software di elaborazione di immagini digitali, come ad esempio Photoshop, un utente può utilizzare una Brush digitale esattamente come un pittore userebbe un pennello per dipingere: in questo caso specifico la Brush rappresenta la tipologia di "pennellata" che si effettua, a seconda della quale il colore viene distribuito nel raggio di azione del pennello secondo modalità differenti (uniforme, mescolato al colore già presente, tramite blending, in semitrasparenza, ...). In generale, il Brush applica un certo ragionamento (un algoritmo a tutti gli effetti) alle unità che compongono il suo dominio di applicazione. Nel caso delle immagini digitali questo dominio è rappresentato dai pixel, ai quali viene cambiato colore; nel caso del Digital Sculpting le unità alle quali sono applicate le Brushes sono i vertici del mesh, per i quali viene calcolata la nuova posizione nello spazio. Dunque si può intuire che più sia alta la risoluzione del mesh più potrebbe essere alto il numero dei vertici sui quali applicare il Brush per lo stesso valore del raggio di azione, e più potrebbe essere elevato il costo computazionale di queste operazioni. Rendere quest'ultimo minimale è una condizione necessaria al successo del progetto, poiché ad un costo computazionale molto elevato non può corrispondere un output in *ysculpting* che sia Real-Time; questa condizione applicata al caso particolare del Brush assume ancora più importanza dal momento che la pratica del Digital Sculpting acquisisce rilevanza per oggetti costituiti di un mesh con un numero di vertici importante, nell'ordine di centinaia di migliaia come caso minimo fino ad arrivare a mesh ad altissima risoluzione con un numero di vertici nell'ordine dei milioni. Infatti i vertici sono le unità sulle quali agisce il Brush per cambiare la geometria di un mesh, e così la loro densità si rispecchia nel grado di libertà e nella profondità delle modifiche applicabili.

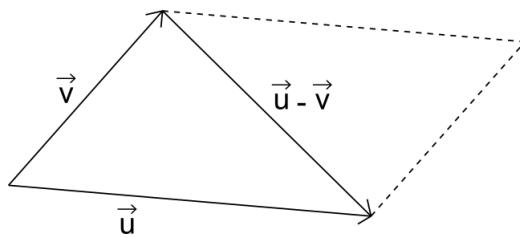


Figura 2.7. Visualizzazione del vettore differenza

Il primo passo una volta ottenuto lo Stroke sia ricavare tutti i vertici sui quali occorra applicare il Brush. Come si è potuto vedere in precedenza nella figura 2.3, i software moderni di Digital Sculpting applicano un Brush all’interno dell’area delimitata dalla circonferenza che ha il centro nel punto in cui si trova il cursore dell’utente, e cioè uno dei punti campionati dello Stroke. L’idea più semplice alla quale di potrebbe pensare è ovviamente il controllo della distanza di ogni singolo vertice del mesh dal punto sullo Stroke. In uno spazio vettoriale in tre dimensioni, la distanza tra due punti p_1 e p_2 è calcolata come segue:

$$\text{distance}(p_1, p_2) = |p_1 - p_2| \quad (2.10)$$

Infatti i punti p_1 e p_2 sono interpretabili anche come due vettori, e la differenza tra vettori è il vettore che ha l’origine in uno dei due punti e termina nell’altro, come mostrato in figura 2.7. Dunque prendendo il modulo di questo vettore di ottiene la distanza tra i due punti p_1 e p_2 . L’iterazione su tutti i vertici del mesh per il controllo della distanza data dalla formula 2.10 circa un certo punto dello Stroke è effettivamente la soluzione più semplice ma anche più inefficiente: la complessità è lineare, ma aumenterebbe con l’aumentare della risoluzione del mesh, e si è già spiegato in precedenza come un numero molto elevato di vertici debba essere considerato il caso base per un applicativo come *ysculpting*. Necessitando di un’altra soluzione per risolvere questo problema che eviti l’iterazione su tutti i vertici del mesh, in questo lavoro di Tirocinio si sono prese in considerazione strutture dati e tecniche di accelerazione del processo: l’Hash Grid e la computazione parallela in multithreading.

2.2.1 Hierarchical Spatial Hash Grid

Una Hierarchical Spatial Hash Grid è una struttura dati sparsa ed organizzata in gerarchie realizzata per essere molto efficiente per molti dei calcoli che richiedono l’accesso da un certo punto dello spazio a degli spazi contigui, vicini, che normalmente sarebbero difficili da ricavare. Questa struttura dati divide ed organizza lo spazio in "celle" contigue di una dimensione prefissata e con ordinamento gerarchico, in modo tale che ogni cella abbia accesso diretto alle celle vicine. In Computer Grafica è una struttura dati molto utilizzata soprattutto per quanto riguarda la simulazione ed il collision detection tra oggetti della scena: infatti, una volta creata questa griglia di celle, data una posizione iniziale si può accedere a tutte le celle vicine entro un certo raggio per trovare eventuali altri vertici (magari appartenenti ad un altro mesh) e

calcolarne la relativa distanza.

Nel caso di un Brush, questa struttura dati garantirebbe esattamente ciò che serve: data una posizione sullo Stroke (la posizione iniziale), si potrebbero ricavare tutti i vertici che sono ad una distanza minore o uguale al raggio (la distanza massima) del Brush, senza considerare tutti i restanti vertici del mesh. Un piccolo compromesso di questa struttura dati che si avrebbe nel caso in cui si utilizzasse in *ysculpting* sarebbe la necessità di ricalcolo della struttura ogni volta che venissero modificate le posizioni dei vertici del mesh (anche se ad uno solo tra essi), in modo tale che possa aggiornarsi alla loro nuova disposizione.

2.2.2 Multithreading

Il multithreading è una tecnica molto utilizzata sia in ambito della programmazione in ambito Computer Grafica che in altri, perché permette di sfruttare al massimo la potenzialità della CPU in uso per ridurre di un fattore uguale al numero di core fisici del processore il costo computazionale delle operazioni. Purtroppo non è sempre possibile sfruttare tutta la potenza di calcolo del processore in maniera completa e parallela: in molti casi in cui si utilizzino risorse condivise potrebbe non essere possibile parallelizzare al massimo il carico di lavoro gravante sulla CPU.

Nel caso del Brush, il multithreading potrebbe essere applicato nell'iterazione di tutti i vertici del mesh solo nel caso in cui essi vengano correttamente divisi in modo che i vari threads non debbano scrivere dati condivisi con gli altri threads: l'insieme dei vertici potrebbe essere diviso in n sottoinsiemi di vertici, con n il numero di threads utilizzati, ed ogni sottoinsieme potrebbe esser processato da un thread diverso perché calcoli la distanza di ogni vertice del suo sottoinsieme dalla posizione centrale, ed eventualmente li aggiunga all'insieme dei vertici che devono esser modificati dal Brush. Il compromesso da gestire con l'utilizzo del multithreading sarebbe il costo di spawning e mantenimento di ogni thread, che essendo elevato necessiterebbe di un trattamento apposito, come ad esempio un ThreadPool.

2.2.3 Applicazione del Brush

La decisione dell'approccio da utilizzare tra Hash Grid e Multithreading è ricaduta sulla prima in base a delle valutazioni pratiche effettuate su meshes di varie risoluzioni, durante le quali l'Hash Grid è risultata più efficiente rispetto al multithreading. E' stata implementata anche una versione di *ysculpting* che combinasse le due tecniche utilizzando il supporto del multithreading per calcolare il modificatore del Brush sui vertici del mesh nel raggio di azione, individuati tramite l'Hash Grid. Tuttavia, questo terzo approccio combinato non ha portato a risultati che ne incoraggiassero la preferenza rispetto al singolo utilizzo dell'Hash Grid.

Dunque, ottenuti i vertici del mesh, il compito del Brush è quello di applicare ad ognuno di essi un'operazione che ne modifichi la posizione e di conseguenza la forma del mesh cui appartengono. Per raggiungere questo obiettivo, banalmente si

sposti ogni vertice v_i di una certa quantità h_i lungo una direzione d_i data da un vettore, sommando alla posizione $p(v_i)$ del vertice v_i il vettore della direzione d_i normalizzato e poi moltiplicato per lo spostamento h_i :

$$p'(v_i) = p(v_i) + \frac{d_i}{|d_i|} * h_i \quad (2.11)$$

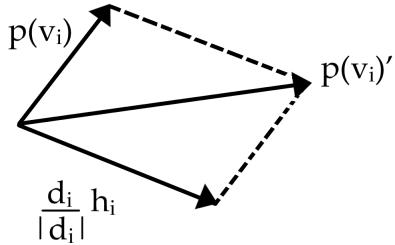


Figura 2.8. Visualizzazione del vettore somma della formula 2.11

Per la regola del parallelogramma, infatti, la posizione che si ottiene è quella dell'origine del sistema di riferimento $p(o)$ traslata del vettore $p(v_i)$, che a sua volta viene traslato dal vettore d_i il cui modulo si imposta uguale a h_i , e il risultato è la posizione $p'(v_i)$ (figura 2.8). Il valore h_i ha un indice i perché può variare da vertice a vertice in base a certi criteri sono propri di differenti Brushes e che li differenziano l'uno dall'altro. Infatti, agendo sul valore h_i si potrebbero creare comportamenti diversi, e di conseguenza si modificherebbe diversamente il mesh. Inoltre, in molti casi la direzione d che viene utilizzata è la normale d_i del vertice (figura 2.9) o la normale alla superficie nel punto dello Stroke cui il Brush si riferisce, ma ovviamente potrebbe essere anche una direzione qualsiasi.

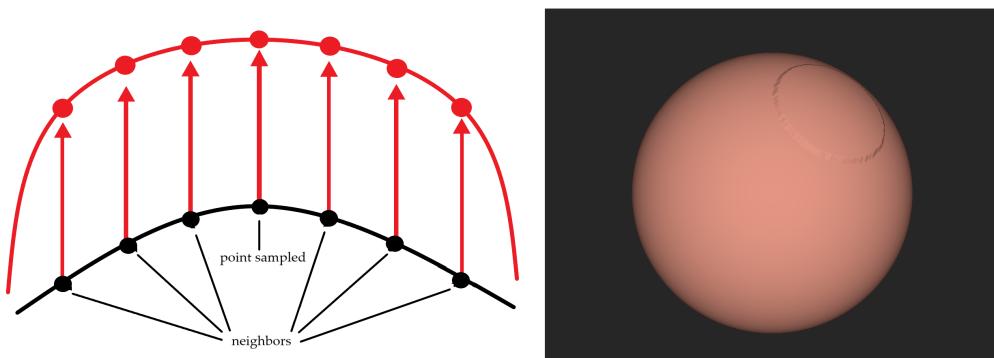


Figura 2.9.

Sinistra: uno schema di Brush uniforme, dove i punti neri indicano le posizioni iniziali dei vertici del mesh; quelli rossi la loro posizione dopo l'applicazione del Brush lungo la normale del punto campionato (frecce rosse)

Destra: un esempio di Brush uniforme in *ysculpting*

La figura 2.9 mostra un esempio di un Brush uniforme implementato in *ysculpting*, una tipologia di Brush che però è poco interessante e non molto utile in un contesto

di Digital Sculpting, ma a questo punto è utile per avere un feedback visivo del discorso. Comunque, nella maggior parte dei Brush, è questo il ragionamento alla base delle modifiche delle posizioni dei vertici, anche se ci sono sempre delle eccezioni. Nel codice di *ysculpting* l'algoritmo di Brush lavora, tra le altre cose, con l'output dell'algoritmo di Stroke visto in precedenza e viene qui di seguito descritto, con l'aggiunta dell'utilizzo di *bvh*¹:

Algorithm 2 brush

```

1: Input: mesh, stroke_sample, radius, hash_grid, bvh
2: if stroke_sample == null then
3:   return
4: end if
5: neighbors = find_neighbors(hash_grid, stroke_sample, radius)
6: for neighbor : neighbors do
7:   neighbor = modifier(neighbor, direction, lenght)
8: end for
9: apply_brush(mesh, neighbors, hash_grid, bvh)
```

2.3 Interfaccia Utente

Con la definizione dell'Algoritmo 1 per lo Stroke e dell'Algoritmo 2 per il Brush, *ysculpting* possiede già gli elementi basilari di un qualsiasi altro software di Digital Sculpting, ma ancora non si è trattata l'implementazione né di un'interfaccia utente che permetta di settare tutti i parametri per il controllo degli strumenti Stroke e Brush né di un qualche tipo di feedback visivo che possa aiutare nell'utilizzo degli strumenti. Come visto nella figura 2.3 un cursore personalizzato è molto importante perché sia garantito all'utente controllo su ciò che possa fare. Dunque anche in *ysculpting* si implementa un cursore analogo, che visualizzi in modo chiaro il raggio di azione del Brush e la posizione del punto campionario sullo Stroke.

Si vuole utilizzare la circonferenza per rappresentare il limite massimo di raggio di azione del Brush come cursore personalizzato, e questa viene creata a partire dal punto campionario sullo Stroke che è quindi il suo centro, "disegnata" sul piano tangente di questo stesso punto. In questo modo la zona di azione del Brush è più chiara anche quando parte di essa non è completamente visibile dalla telecamera virtuale. L'equazione della circonferenza dato il centro e il raggio è la seguente:

$$(x - x_0)^2 + (y - y_0)^2 = r^2 \quad (2.12)$$

ed è espressa in due dimensioni, mentre lo spazio virtuale è in tre dimensioni. Si utilizzi quindi il piano tangente alla posizione (x_0, y_0) , che è il punto centrale della circonferenza, rispetto alla superficie del mesh. Per calcolare questo piano tangente si ha a disposizione la normale del centro della circonferenza, e la si calcoli tramite interpolazione delle normali sui vertici della primitiva cui appartiene la posizione

¹È il Bounding Volume Hierarchy introdotto nella sezione Stroke

campionata sullo Stroke, in modo analogo a quanto fatto con la formula 2.5 con l'uso delle normali n_i al posto dei vertici v_i :

$$n(w_1, w_2) = n_0 + w_1(n_1 - n_0) + w_2(n_2 - n_0) \quad (2.13)$$

Dunque si necessita nuovamente delle coordinate baricentriche della posizione campionata sullo Stroke che si trovano nella stessa struttura dati per l'intersezione che si ha nell'Algoritmo 1. La normale n_p così ottenuta è il vettore ortogonale al piano tangente (per la definizione stessa di normale): calcolando un vettore ortonormale n'_p a n_p e il vettore n''_p normale ottenuto da $(n_p \times n'_p)$, si ottiene una base ortonormale con l'origine in p e con i vettori normali n'_p e n''_p che formano gli assi x ed y del piano che è tangente a p stesso. La direzione dei vettori n'_p e n''_p non è importante, poichè la circonferenza, che si costruirà sul piano formato da essi, non ha un orientazione ma infiniti assi di simmetria.

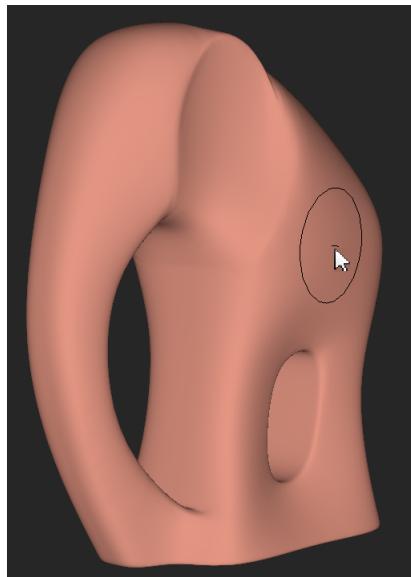


Figura 2.10. Circonferenza tangente al punto di intersezione in ysculpting, che rappresenta il raggio d'azione del Brush

Anche la circonferenza come lo Stroke è una curva, dunque analogamente si potrebbe attuare un campionamento dei punti ed ottenere un'approssimazione della circonferenza costituita di molti segmenti a formare una polilinea: più è elevato il numero di segmenti, maggiore è accurata l'approssimazione. I punti di partenza possono essere ottenuti utilizzando i valori $x = 0$ e $y = 0$ nella formula 2.12, che restituisce quattro posizioni corrispondenti ai punti della circonferenza sugli assi x ed y . Questi quattro punti iniziali possono essere ricavati direttamente dalla base ortonormale in fase di implementazione: $(0, n'_p * radius)$, $(0, n'_p * -radius)$, $(n''_p * radius, 0)$, $(n''_p * -radius, 0)$. Da questi quattro punti c_1 , c_2 , c_3 , c_4 si può ottenere la poli-linea formata dai quattro segmenti d_1 , d_2 , d_3 , d_4 che uniscono questi punti, che sono considerabili vettori:

$$d_1 = (c_2 - c_1) \quad d_2 = (c_3 - c_2) \quad d_3 = (c_4 - c_3) \quad d_4 = (c_1 - c_4) \quad (2.14)$$

Questi quattro vettori possono essere divisi ognuno in m segmenti, con $m * 4$ il numero di segmenti totali (cioè la risoluzione della circonferenza), e ogni vertice di ogni segmento così ottenuto si può imporre alla distanza *radius* sulla direzione costituita dal vettore che parte dall'origine della circonferenza ed arriva al vertice del segmento:

$$c_i = \frac{(c_i - p)}{|c_i - p|} * radius \quad (2.15)$$

Di seguito è riportato lo pseudocodice dell'implementazione di *ysculpting*:

Algorithm 3 view_pointer

```

1: Input: mesh, intersection, radius, resolution
2: Output: circle_mesh
3: if intersection.hit == false then
4:   return null
5: end if
6: position = eval_position(mesh, intersection.element, intersection.uv)
7: normal = eval_normal(mesh, intersection.element, intersection.uv)
8: basis = basis_from_z(normal)
9: return make_circle(position, basis, radius, resolution)
  
```

Oltre alla visualizzazione della raggio di azione del Brush si mostri come parte della UI anche il comando per la scelta del raggio (le modalità di creazione della UI vera e propria sono variabili, dunque non sono trattate).

2.4 Stroke Segmentation

Negli algoritmi di *ysculpting* sin qui descritti persiste una problematica che è resa visibile solo quando la risoluzione del mesh in elaborazione raggiunge una soglia molto elevata. Ricordando che l'implementazione è basata su CPU, che non è in grado di effettuare la tipologia di calcoli richiesti da un Digital Sculpter alla stessa velocità di una GPU, questo comportamento è comprensibile e questa soglia è molto variabile poiché dipende moltissimo dal processore della macchina sulla quale si esegue *ysculpting*: a numeri di vertici molto elevati il software inizia ad avere un frame-rate sempre più basso (il frame-rate è il numero di fotogrammi al secondo che la UI riesce a garantire) e anche le strutture di accelerazione utilizzate come il Bounding Volume Hierarchy e l'Hash Grid iniziano a vacillare, causando un rallentamento al programma che diventa sempre più netto col crescere del numero dei vertici del mesh. Questo rallentamento rimane tollerabile e permette comunque l'utilizzo di *ysculpting* fino ad un'altra soglia, ancora più elevata della precedente, di numero di vertici (ma come l'altra dipende molto dall'hardware in uso); il software però presenta un difetto particolare quando la risoluzione del mesh è molto elevata:

la Stroke Segmentation, ovvero una segmentazione vera e propria dello Stroke che lo rende in molte situazioni discontinuo e disomogeneo. Questo difetto si traduce nella difficoltà di realizzazione anche del più semplice tratto sulla superficie del mesh, perché anche se l'utente tracciasse uno Stroke continuo lungo il mesh avrebbe come risultato vari Stroke più piccoli e segmentati. Questo fenomeno è dovuto al costo computazionale crescente che le funzioni di Brush acquisiscono per numeri di vertici oltre le soglie menzionate, che porta ad un rallentamento dell'output del software. Infatti, il rallentamento delle funzioni di Brush che ricevono in input l'output della funzionalità di Stroke causa un ritardo dell'aggiornamento della UI di *ysculpting*, poiché questo non può essere effettuato prima della conclusione dei calcoli delle nuove posizioni dei vertici del mesh. Si ricordi poi come la chiamata alla funzione di Stroke coincida proprio con l'aggiornamento della UI, che rallentando causa lo stesso rallentamento anche alla chiamata per la funzionalità di Stroke. Rallentare lo Stroke significa avere la possibilità che non vengano calcolate tutte le proiezioni delle posizioni del cursore durante il suo movimento. La visibilità della Stroke Segmentation dipende anche dalla velocità del cursore: più questa è elevata più sono le posizioni che non verranno considerate nel campionamento dello Stroke.

Nel caso di *ysculpting* sarebbe molto complicato o infattibile abbassare ulteriormente il costo computazionale del Brush, quindi si sceglie una soluzione che risolve completamente il problema ma che ha come compromesso la parziale perdita di precisione dello campionamento dello Stroke rispetto al movimento reale del cursore. Questa soluzione consiste nel garantire sempre che la distanza tra il campionamento di un punto ed il successivo sia uguale tra tutti i campionamenti e che sia sempre la distanza corretta. Infatti il comportamento della funzione di Stroke che causa la sua segmentazione è dovuta all'Algoritmo 1 che effettua l'intersezione col mesh per le posizioni del cursore anche molto distanti dall'ultimo punto campionato; tutti le posizioni intermedie vengono perse a causa del rallentamento. Come mostrato nella sezione Stroke, la scelta della corretta distanza minima tra un campionamento e l'altro è vitale per una buona approssimazione del movimento reale del cursore; si imponga allora anche una distanza massima ad esso, per la quale il punto intersecato venga aggiunto allo Stroke e oltre la quale la distanza venga calcolata non più dal punto considerato inizialmente ma da quest'ultimo punto aggiunto. Per fare ciò è possibile modificare l'Algoritmo 1 in modo tale che se l'ultima posizione ricevuta del cursore dovesse generare un'intersezione a distanza maggiore di quanto stabilito (nel caso di *ysculpting* il 20% del raggio del Brush), allora questa distanza verrebbe divisa in segmenti di lunghezza pari a quella stabilita per il campionamento dello Stroke, a partire dall'ultimo punto campionato. Il resto di questa divisione della distanza, se presente, non verrebbe considerato, e così l'ultimo punto campionato non sarebbe più quello ricavato dall'intersezione dell'ultima posizione del cursore ma quello corrispondente alla posizione u del cursore tale che:

$$u = \frac{u_{i+1} - u_{i-1}}{|u_{i+1} - u_{i-1}|} * stroke_uv_dist * n \quad (2.16)$$

con u_{i+1} la posizione corrente del cursore, u_{i-1} l'ultima posizione del cursore la cui intersezione corrispondente è stata aggiunta al campionamento nella precedente chiamata alla funzione di Stroke e n il risultato della divisione intera tra la distanza

delle posizioni u_{i+1} e u_{i-1} e $stroke_uv_dist$, il quale è la distanza scelta per il campionamento dello Stroke proporzionata alla distanza delle posizioni u_{i+1} e u_{i-1} sul piano dell'immagine della telecamera virtuale. Date le posizioni sul mesh s_{i+1} e s_{i-1} corrispondenti alle posizioni del cursore sul piano dell'immagine u_{i+1} e u_{i-1} :

$$stroke_uv_dist = \frac{(|u_{i+1} - u_{i-1}|) * (radius * 0.2)}{|s_{i+1} - s_{i-1}|} \quad (2.17)$$

Ottenuta così la posizione finale u del cursore, conoscendo u_{i-1} e un certo numero di segmenti (o steps), per ogni posizione sul piano dell'immagine corrispondente alla fine di ogni segmento (compresa la posizione finale) si generi un raggio del quale si calcola l'intersezione con il mesh: ogni posizione sul mesh così ricavata venga aggiunta al campionamento dello Stroke, impedendo la Stroke Segmentation.

Algorithm 4 new_stroke

```

1: Input: camera, mouse_uv, last_uv, mesh, intersection, last_position, radius,
   bvh
2: Output: stroke_sampling
3: position = eval_position(mesh, intersection.element, intersection.uv)
4: delta_position = distance(position, last_position)
5: stroke_distance = radius * 0.2
6: if last_position == null then
7:   last_position = position
8:   last_uv = mouse_uv
9:   return null
10: end if
11: delta_uv = distance(mouse_uv, last_uv)
12: steps = int(delta_position / stroke_distance)
13: if steps == 0 then
14:   return null
15: end if
16: stroke_uv = delta_uv * stroke_distance / delta_position
17: mouse_dir = normalize(mouse_uv - last_uv)
18: for step : steps do
19:   last_uv += stroke_uv * mouse_dir
20:   ray = camera_ray(camera, last_uv)
21:   intersection = intersect_triangles_bvh(bvh, mesh, ray)
22:   position = eval_position(mesh, intersection.element, intersection.uv)
23:   normal = eval_normal(mesh, intersection.element, intersection.uv)
24:   last_position = position
25:   stroke_sampling.insert(position, normal)
26: end for
27: return stroke_sampling

```

Capitolo 3

Gaussian Brush

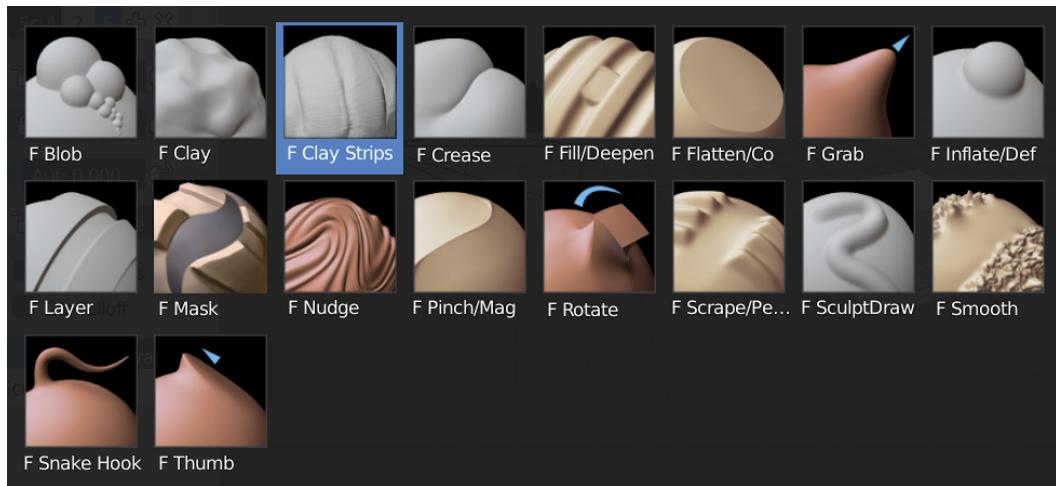


Figura 3.1. Blender: esempi di Brushes

Nel capitolo precedente sono stati introdotti gli elementi di base che un software di Digital Sculpting dovrebbe avere per essere tale, e ne è stata esplicata l'implementazione in *ysculpting*. Se per quanto concerne lo Stroke è stato descritto tutto il necessario, del Brush è stato spiegato solo il concetto generale sia in grafica bi-dimensionale che tri-dimensionale, con un semplice esempio di Brush uniforme (figura 2.9). Tuttavia questo è lo strumento fondamentale che permette all'artista che lo utilizza di dare una forma al tratto che effettua sull'oggetto piuttosto che un'altra; il Brush infatti modifica le posizioni di alcuni vertici del mesh secondo delle regole che variano in base alla tipologia di Brush utilizzata. Un Brush uniforme da questo punto di vista non è funzionale a creare qualcosa di diverso dall'oggetto di partenza, perché si limita a spostare i vertici di una stessa quantità in una sola direzione. In realtà esistono moltissime tipologie di Brush, come ad esempio lo Smooth Brush che smussa le linee più "sporgenti", il Flatten Brush che sposta i vertici a formare un piano, il Clay Brush che crea sulla superficie del mesh delle imperfezioni per simulare specifici materiali, e moltissimi altri mostrati come esempio in figura 3.1. Tutte queste tipologie di Brush spostano i vertici del mesh di una quantità variabile, secondo un qualche criterio che le caratterizza, ed è esattamente il comportamento

che si vuole replicare in *ysculpting*. Per fare ciò si possono utilizzare delle funzioni che calcolino le nuove posizioni dei vertici in input in base alle posizioni iniziali degli stessi rispetto al punto centrale campionato sullo Stroke: è il caso del Gaussian Brush che utilizza proprio la funzione di distribuzione della probabilità gaussiana per computare le nuove posizioni dei vertici. Questo Brush può essere considerato l'analogo in *ysculpting* del Brush standard di molti altri software di Digital Sculpting (nel caso di Blender, in figura 3.1, il Brush standard è quello denominato SculptDraw) ed utilizza la funzione gaussiana per via della forma "morbida" che riesce a conferire alla nuova disposizione dei vertici sul mesh.

3.1 Densità di probabilità normale

È stata scelta la funzione gaussiana a causa della rappresentazione grafica della relativa funzione di densità di probabilità: essa infatti descrive una cupola che ha il suo picco massimo nel punto centrale e che scende simmetricamente sull'asse x in maniera graduale fino a tendere a zero per ($x \rightarrow +\infty$) e ($x \rightarrow -\infty$).

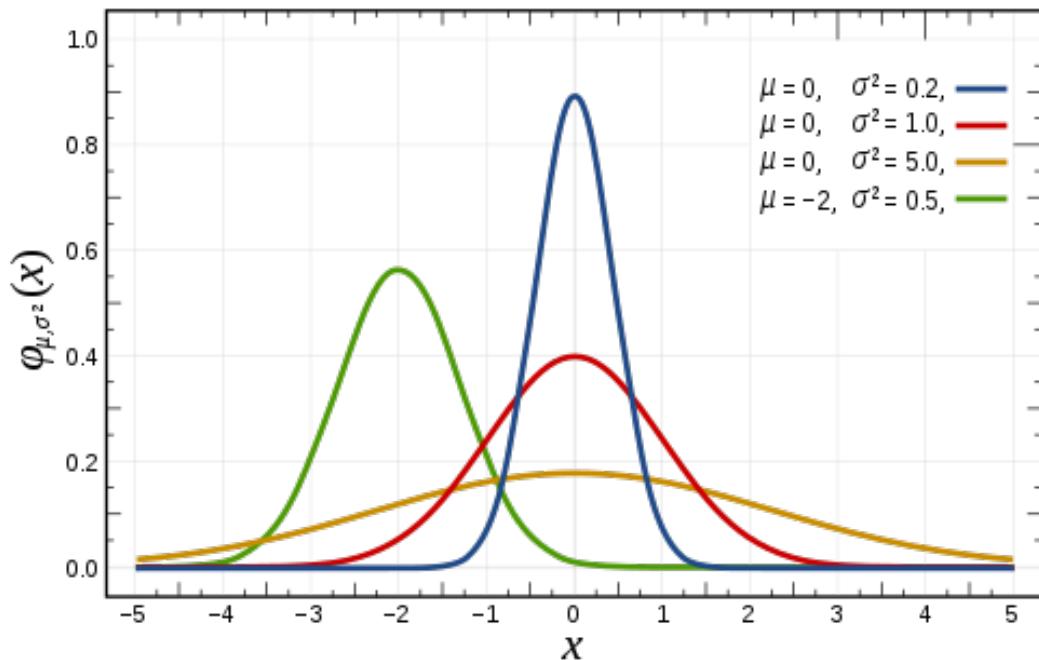


Figura 3.2. Il grafico della funzione di densità relativa alla distribuzione normale di probabilità
(Wikipedia by Inductiveload user)

$$\varphi_{\mu, \sigma^2} = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\} \quad (3.1)$$

Come si può notare, la formula 3.1, che descrive la funzione di densità di probabilità gaussiana, è parametrizzata in μ e σ che sono due parametri che ne definiscono la posizione centrale (μ) e la pendenza (σ), che sono esattamente gli

aspetti che interessano il Brush: il primo parametro potrebbe essere utilizzato per "posizionare" il centro della curva su un punto campionario dello Stroke, ed il secondo potrebbe essere modificato dall'utente per variare la forza da applicare al tratto. Se si osserva la figura 3.2 si può notare che le curve blu, rossa e gialla hanno il centro di simmetria nel punto $x = 0$ poiché il parametro $\mu = 0$; la curva verde invece è traslata rispetto alle altre ed ha il suo centro di simmetria in $x = -2$, corrispondente al valore del parametro $\mu = -2$; il parametro σ invece varia su tutte le curve rappresentate, che di conseguenza hanno tutte una pendenza e altezza massima differenti. L'utilizzo di questa funzione ammetterebbe così la possibilità di spostare di quantità differenti i vari vertici del mesh interessati dalle modifiche del Brush; questa possibilità è già stata introdotta implicitamente nella formula 2.11, ed è rappresentata dal parametro h_i relativo al vertice v_i che indica la lunghezza che assume il vettore di spostamento d_i nel momento dell'applicazione del Brush. Nel caso della funzione di densità di probabilità gaussiana il parametro h_i corrisponde al valore della funzione stessa in x_i , cioè la distanza del vertice v_i dalla posizione x centrale, ovvero:

$$h_i = \varphi_{\mu, \sigma^2}(x_i) \quad (3.2)$$

e dunque si può esprimere il Gaussian Brush per un vertice v_i :

$$p'(v_i) = p(v_i) + \frac{d_i}{|d_i|} * \varphi_{\mu, \sigma^2}(v_i) \quad (3.3)$$

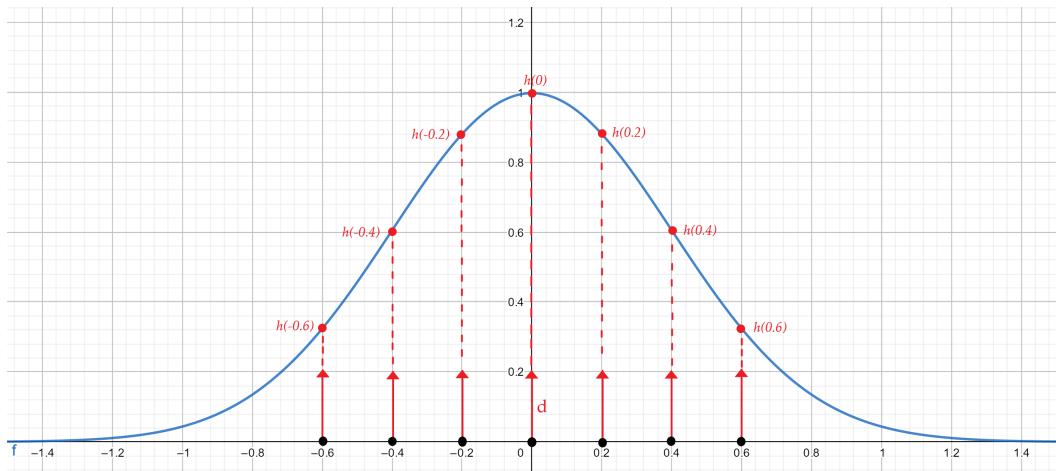


Figura 3.3. I valori corrispondenti sull'asse y a dei punti specifici sull'asse x ; nell'approccio del Gaussian Brush i primi corrispondono al valore h_i per le posizioni dei vertici del mesh, rappresentati dai punti sull'asse x , rispetto al punto centrale, con d il vettore direzione dello spostamento
(Graphing Calculator by GeoGebra)

3.2 Applicazione al Digital Sculpting

L'applicazione della formula 3.3 al contesto di Digital Sculpting potrebbe sembrare banale, ma in realtà cela delle problematiche da gestire per garantire un efficace

adattamento di questa formulazione al Gaussian Brush di *ysculpting*.

La prima di queste problematiche è che la funzione di densità di probabilità normale vista nella formula 3.1 è espressa in una sola dimensione perché il suo dominio è il solo asse x , mentre il Digital Sculpting viene effettuato in uno spazio virtuale a tre dimensioni dove ogni vertice del mesh ha tre valori distinti per le coordinate x , y e z , che devono costituire dunque il dominio: la formula non dovrà essere del tipo $\varphi_{\mu,\sigma^2}(x)$ ma piuttosto del tipo $\varphi_{\mu,\sigma^2}(x, y, z)$. Risolvere questo ostacolo non è però molto complicato, poiché è sufficiente ottenere una generalizzazione della formula 3.1 tale che sia espressa in tre dimensioni, in modo da poterla utilizzare per processare input in tre coordinate per ogni vertice su cui agisce il Gaussian Brush. In generale, se si vuole ottenere una generalizzazione della formula 3.1 che sia in n dimensioni e che sia del tipo $(N * \exp\{-\frac{x^2+y^2+z^2+\dots}{2\sigma^2}\})$, allora la costante di normalizzazione N dipenderà direttamente dalla dimensione n dello spazio:

$$N = \frac{1}{\sigma^n (2\pi)^{\frac{n}{2}}} \quad (3.4)$$

quindi nel caso tri-dimensionale, con $n = 3$:

$$N = \frac{1}{\sigma^3 (2\pi)^{\frac{3}{2}}} \quad (3.5)$$

e di conseguenza la formula di densità di probabilità normale espressa in 3D:

$$\varphi_{\mu,\sigma^2}(x, y, z) = \frac{1}{\sigma^3 (2\pi)^{\frac{3}{2}}} \exp\left\{-\frac{x^2 + y^2 + z^2}{2\sigma^2}\right\} \quad (3.6)$$

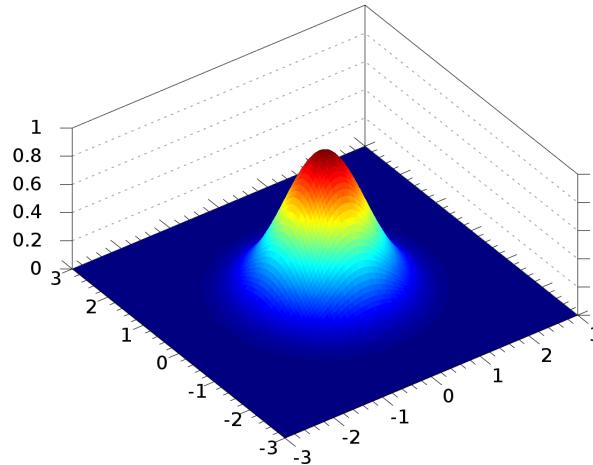


Figura 3.4. La curva gaussiana definita su un dominio bi-dimensionale; la curva nel caso tri-dimensionale è quasi la stessa, con la differenza che i punti del dominio non giacciono tutti sullo stesso piano, ma possono avere diversi valori per la coordinata z (Wikipedia by Krishnavedala user)

La seconda delle problematiche da affrontare è la necessità di adattare l'intervallo del dominio della funzione di densità di probabilità normale a quello del Brush: il primo è definito sull'asse x da $-\infty$ a $+\infty$, mentre il dominio del Brush, come si è già spiegato, è limitato ad un intervallo di distanza che va da $-radius$ a $+radius$. Dal grafico rappresentato nella figura 3.2 si può evincere che anche dalla funzione di densità di probabilità normale si può estrapolare una sorta di diametro sull'asse x , entro il quale il valore y della funzione non tende completamente a zero: un esempio visivo di ciò che si intende è mostrato nella figura 3.5. Dunque si potrebbero ridimensionare proporzionalmente il diametro di applicazione del Brush ($radius * 2$) e le coordinate dei vertici sui quali il Brush agisce rispetto a questo diametro della funzione di densità di probabilità gaussiana, in modo che si possa ottenere per ogni vertice nel area di azione definita dal diametro del Brush il valore corretto di spostamento, applicando la funzione di densità di probabilità normale alle coordinate ridimensionate dei vertici stessi. In altre parole, si tratta di scalare un intervallo del dominio della funzione di densità di probabilità standard in un altro intervallo, che è quello dato dal *radius* del Brush (figura 3.6).

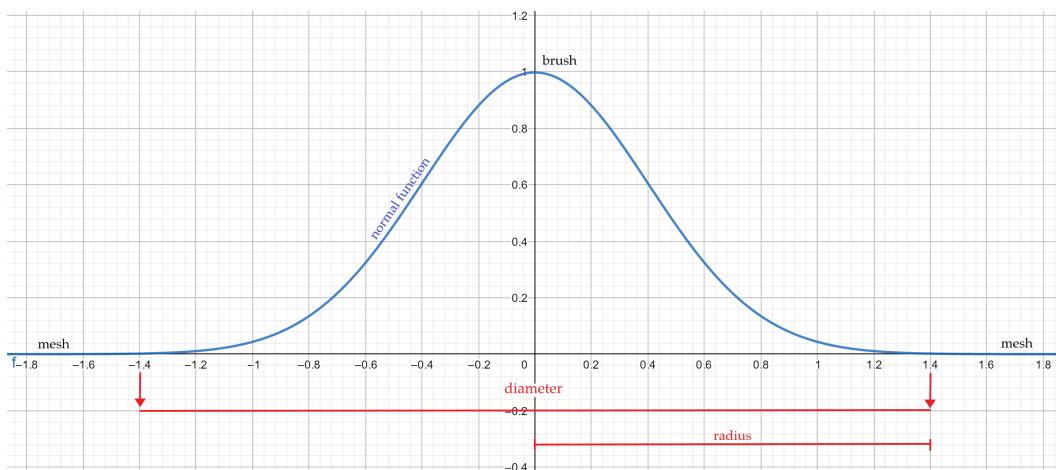


Figura 3.5. Interpretazione della funzione normale in *ysculpting*: la linea rossa più lunga rappresenta il diametro di azione del Brush, il cui risultato sul mesh è la creazione di una protuberanza secondo la normale gaussiana
(Graphing Calculator by GeoGebra)

Nella pratica, quindi in *ysculpting*, tutto ciò si traduce nello scegliere un raggio fissato sulla funzione di densità di probabilità normale che sia di riferimento per trovare il fattore di scala (*scaled_factor*) con il *radius* del Brush, grazie al quale si possano scalare anche le coordinate di tutti i vertici del mesh da modificare per calcolare su di esse il valore corretto della funzione normale. Questa scelta del raggio sulla funzione di densità di probabilità normale è cruciale perché il risultato dell'applicazione del Brush risulti sul mesh in una superficie smussata e continua, che non abbia discontinuità (figura 3.7): il raggio deve essere tale che il relativo diametro copra l'intervallo di dominio della normale al quale corrisponda un intervallo di codominio strettamente maggiore di zero (la funzione in realtà tende a zero, dunque con "strettamente maggiore di zero" qui si intenda un numero maggiore di zero di

almeno una certa quantità).

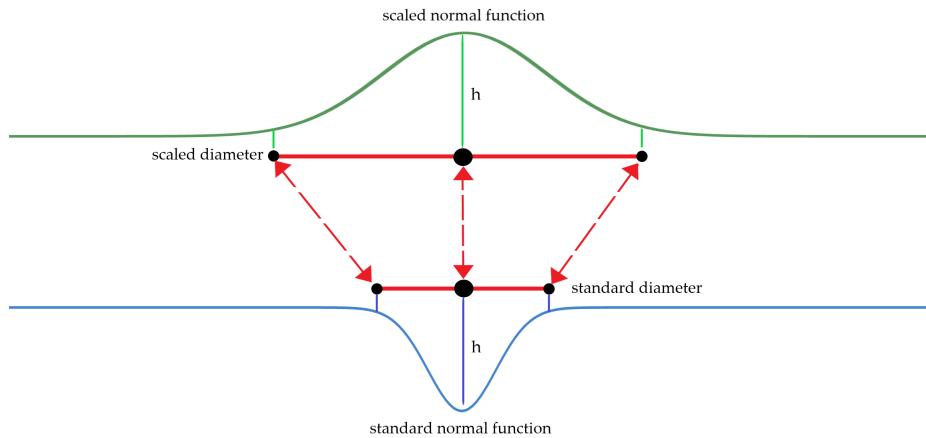


Figura 3.6. Ridimensionamento proporzionale della funzione normale
(Graphing Calculator by GeoGebra)

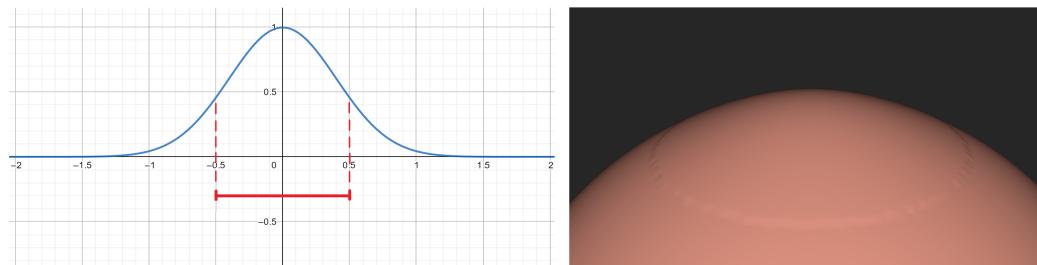


Figura 3.7.

Sinistra: una pessima scelta di raggio

Destra: il risultato di una cattiva scelta di raggio in *ysculpting*
(Graphing Calculator by GeoGebra)

L'ultima problematica da affrontare per concludere la trattazione dell'applicazione della funzione di densità di probabilità normale nel Digital Sculptor *ysculpting* è la gestione del parametro σ , ovvero la deviazione standard. Riprendendo il discorso introduttivo del capitolo, questo parametro è definito nell'intervallo $\sigma \in (0, +\infty)$ ed al suo variare corrisponde la variazione della pendenza della curva della funzione normale: facendo scegliere all'utente tramite UI il valore di questo parametro, esso potrebbe variare la "forza" del tratto, creando sul mesh delle curve con sporgenza variabile. Modificando questo parametro però non cambia solamente la pendenza della curva normale ma anche l'intervallo del dominio della funzione per il quale il suo codominio non tende nettamente a zero. Questo comportamento è nettamente visibile nella figura 3.2, mettendo a confronto le curve blu e rossa: la prima tende a zero già dai valori $x = -2$ e $x = 2$, mentre la seconda ha un valore di molto maggiore di zero per i punti $x = -2$ e $x = 2$ e tende a zero solo a partire dai punti $x = -3.5$ e $x = 3.5$ più o meno. Questo significa che se si controllasse la pendenza della curva normale tramite il parametro σ , non si potrebbe scegliere un valore fisso

per il raggio della funzione di densità di probabilità normale così come si è scelto nel trattare le precedenti problematiche, perché esso varierebbe con il variare del parametro σ . Nonostante questo, si potrebbe comunque decidere di utilizzare il parametro σ per controllare la pendenza della curva, ma ciò implicherebbe trovare un'altra soluzione alla problematica precedente. Nel caso di *ysculpting* è stato scelto di parametrizzare ulteriormente la funzione di densità di probabilità normale tramite un parametro *strength* che fosse direttamente modificabile dall'utente per gestire la "forza" da imprimere al tratto (corrispondente all'altezza massima della curva gaussiana). L'obiettivo è parametrizzare la funzione normale in modo tale che si possa variare la pendenza senza intaccarne l'intervallo di dominio per il quale essa è strettamente maggiore di zero. Come punto di partenza per esplicare il metodo alla base della parametrizzazione tramite *strength*, si analizzi il punto centrale della funzione normale, il cui valore è massimo rispetto a tutti gli altri valori possibili del dominio. Nella formula 3.6 le variabili x , y e z rappresentano la distanza dal punto centrale; si impongano uguali a zero per trovare il valore della gaussiana nel suo punto centrale (cioè massimo):

$$\begin{aligned}\varphi_{\mu,\sigma^2}(0,0,0) &= \frac{1}{\sigma^3(2\pi)^{\frac{3}{2}}} \exp\left\{-\frac{0^2+0^2+0^2}{2\sigma^2}\right\} \\ &= \\ \varphi_{\mu,\sigma^2}(0,0,0) &= \frac{1}{\sigma^3(2\pi)^{\frac{3}{2}}} \exp\{0\} \\ &= \\ \varphi_{\mu,\sigma^2}(0,0,0) &= \frac{1}{\sigma^3(2\pi)^{\frac{3}{2}}}\end{aligned}\tag{3.7}$$

Si ottiene così il valore $\frac{1}{\sigma^3(2\pi)^{\frac{3}{2}}}$ (abbreviato con N) che non dipende dalle variabili x , y e z , ed è quindi costante. Quindi modificando questa costante N si può modificare l'altezza massima della funzione normale e di conseguenza la sua pendenza. In questo modo è possibile mantenere una scelta fissata per il valore del raggio sulla funzione di densità di probabilità normale con la quale ridimensionare proporzionalmente il *radius* del Brush, anche al variare del parametro *strength* (figura 3.8). Trattandosi però di una costante, il valore N parametrizzato con una certa scelta di *strength* rimarrebbe costante anche al variare di *radius* del Brush, e questo non è un comportamento desiderabile: un Brush che agisse su un'area minore dovrà modificare il mesh in maniera meno netta rispetto a quanto farebbe il Brush con lo stesso valore per *strength* ma che agisse su un'area maggiore. La soluzione è parametrizzare anche lo stesso parametro *strength* con il valore di *radius*, in modo che il valore del primo sia influenzato da quello del secondo; una possibile applicazione di questo potrebbe essere una semplice moltiplicazione (*strength*radius*).

Risolte tutte queste problematiche può essere definita la nuova formulazione dell'equazione 3.6:

$$\varphi_{\mu,\sigma^2}(x,y,z) = (\textit{strength} * \textit{radius}) \frac{1}{\sigma^3(2\pi)^{\frac{3}{2}}} \exp\left\{-\frac{x_s^2 + y_s^2 + z_s^2}{2\sigma^2}\right\}\tag{3.8}$$

con $x_s = \textit{scaled_factor} * x$, $y_s = \textit{scaled_factor} * y$, $z_s = \textit{scaled_factor} * z$

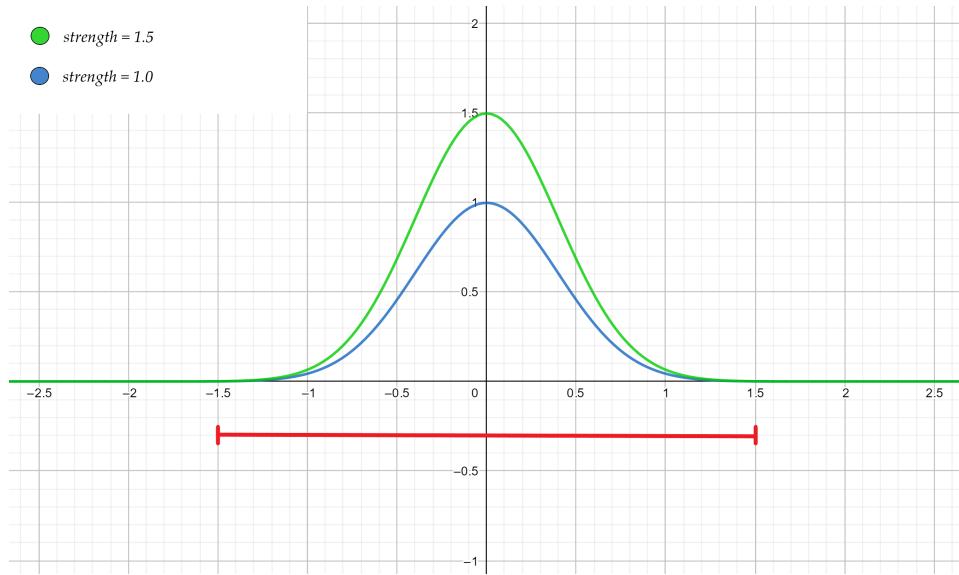


Figura 3.8. Controllo della pendenza della funzione normale tramite il parametro *strength* senza modifica della scelta per il raggio
(Graphing Calculator by GeoGebra)

Con la formula 3.8 si ottiene la quantità dello spostamento dei vertici interessati dal Brush, che quindi deve essere moltiplicata per il vettore normalizzato della direzione, come nella formula 2.11. Solitamente (e anche nel caso di *ysculpting*) il vettore direzione che viene utilizzato è la normale al piano del mesh nel punto campionato dello Stroke, il cui ottenimento è già stato discussso nella sezione Interfaccia Utente del Capitolo 2 tramite la formula 2.13. Infine, si imposti il parametro μ sempre a zero, perché la funzione di densità di probabilità normale venga sempre calcolata a partire dal punto centrale del Brush; il parametro σ venga scelto liberamente, in base alla pendenza di partenza che si voglia dare al Brush. In *ysculpting* è stato imposto σ uguale a 0.7 sulla base di test pratici in fase di sviluppo e gusto personale; la scelta del raggio sul grafico della funzione di densità di probabilità normale con deviazione standard uguale a 0.7 è stato scelto pari a 3.5.

Lo pseudocodice del Gaussian Brush è strutturato a partire dall’Algoritmo 2 e descritto nell’Algoritmo 5.

Algorithm 5 new_brush

```

1: Input: mesh, stroke_sampling, radius, hash_grid, bvh
2: if stroke_sampling == null then
3:   return
4: end if
5: scale_factor = 3.5 / radius
6: for sample : stroke_sampling do
7:   neighbors = find_neighbors(hash_grid, sample, radius)
8:   position = sample.position
9:   normal = sample.normal
10:  for neighbor : neighbors do
11:    gauss_height = gaussian_distribution(position, neighbor,
12:      0.7, scale_factor, strength, radius)
13:    neighbor += normal * gauss_height
14:  end for
15: end for

```

Algorithm 6 gaussian_distribution

```

1: Input: center, position, standard_dev, scale_factor, strength, radius
2: scaled_strength = strength * radius
3: N = 1 / ((standard_dev * scaled_strength)3 *  $\sqrt{2\pi}$ )
4: dx = (center.x - position.x) * scale_factor
5: dy = (center.y - position.y) * scale_factor
6: dz = (center.z - position.z) * scale_factor
7: E = (dx2 * dy2 * dz2) / (2 * standard_dev2)
8: return N * exp{-E}

```



Figura 3.9. Due esempi di Gaussian Brush su *ysculpting*, con stesso valore per *strength* e valori diversi per *radius*

Capitolo 4

Opzioni del Brush

Nell'ambito del Digital Sculpting sono moltissime le tipologie di Brush disponibili dei software più famosi (un esempio dei Brushes di Blender in figura 3.1), ed ognuna di queste ha un effetto diverso sul mesh. Inoltre possono esserci variazioni al comportamento di uno stesso Brush, tramite opzioni che ne determinano aspetti particolari e che sono diverse a seconda della tipologia di Brush al quale appartengono. In generale servono a gestire dei comportamenti "secondari" del Brush. Queste opzioni possono essere molteplici nei software di Digital Sculpting completi, ma nel caso del software *ysculpting* che si sta presentando in questa trattazione sono state implementate solo alcune di esse, che nonostante siano le più basilari sono comunque molto importanti perché permettono delle sfumature al comportamento standard del Brush che possono risultare utili agli artisti per la creazione di effetti particolari sul mesh.

4.1 Negative Brush

Il Negative Brush è probabilmente l'opzione più semplice ma allo stesso tempo più utile che un Brush possa avere: si tratta infatti dell'opzione che permette al modificatore di avere una direzione opposta a quella standard. Questo significa che tramite un Brush non si creano protuberanze di qualche tipo sul mesh (per esempio seguendo la funzione di densità di probabilità normale, come nel caso del Gaussian Brush) ma che al contrario si "scava" la superficie, creando delle formazioni convesse su di essa.

Come detto è un'opzione molto semplice, perché l'unica cosa che cambia rispetto all'implementazione standard del Brush è la sua direzione. Si è visto nei capitoli precedenti come lo spostamento dei vertici del mesh causato dall'applicazione di un Brush segua una certa direzione (in alcune tipologie di Brush, mentre in generale ogni vertice può avere la sua direzione di spostamento), espressa nella formula 2.11: ad esempio nel Gaussian Brush di *ysculpting* il vettore direzione d_i è la normale alla superficie nel punto di intersezione tra cursore e mesh, calcolato tramite interpolazione delle normali ai vertici della primitiva intersecata; il vettore d_i , nel caso del Negative Brush viene ribaltato di 180° , cioè viene reso opposto semplicemente imponendo $d_i = -d_i$. Infatti, cambiando segno a tutte le coordinate di un vettore si calcola il

vettore simmetrico rispetto a tutti e tre gli assi del sistema di riferimento, ottenendo così esattamente il suo opposto $-d = (-d_x, -d_y, -d_z)$. Di conseguenza, anche sul mesh lo spostamento dei vertici avviene in maniera opposta a quella standard, cioè in direzione negativa. Dunque il Negative Gaussian Brush di *ysculpting* computa il vettore direzione opposto al vettore normale alla superficie nel punto campionato sullo Stroke (che è il centro del Brush), che è in direzione del mesh e al quale causa una deformazione come quella in figura 4.1.

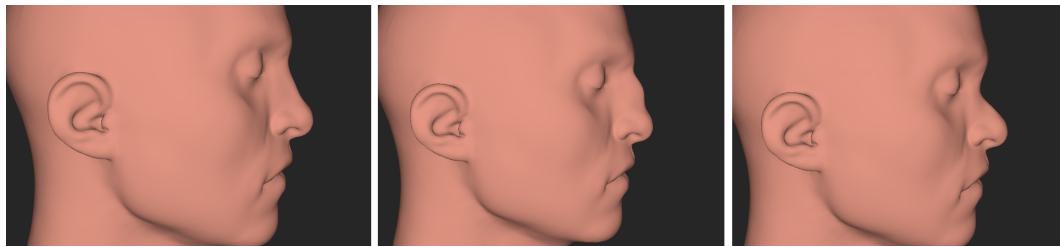


Figura 4.1. Differenze tra modello di base (sinistra), Gaussian Brush (centro) e Negative Gaussian Brush (destra) in *ysculpting*
(base model by niyoo: <https://www.turbosquid.com/it/Search/Artists/niyoo>)

4.2 Continuous Brush

Il Continuous Brush è un'altra opzione che può risultare molto utile nella modellazione in Digital Sculpting e consiste nell'aggiramento del meccanismo di campionamento dello Stroke. Infatti, normalmente, nel muovere il cursore sul mesh l'utente causa la creazione dello Stroke sampling di cui si è parlato nel Capitolo 2; ad una posizione fissa del cursore o quando il suo movimento non soddisfa i requisiti per il campionamento delle intersezioni (Algoritmo 4) non corrisponde invece alcuna memorizzazione del punto intersecato. Dato questo caso generale, possono poi verificarsi situazioni in cui l'utente necessiti che l'applicazione di un Brush avvenga o in un punto preciso del mesh oppure in modo continuo (da cui il nome dell'opzione), per la creazione di deformazioni particolari dell'oggetto.

L'elusione del meccanismo di campionamento dello Stroke può essere effettuata con la semplice cancellazione del controllo della distanza tra l'ultimo punto campionato e quello attuale, in modo tale che venga considerata ogni intersezione con la posizione del cursore, qualunque essa sia. Così facendo, però, non si aggirerebbe solo il meccanismo di campionamento dello Stroke ma anche quello riguardante la problematica della Stroke Segmentation: infatti la funzione di Stroke calcolerebbe sempre e solo l'intersezione attuale tra cursore e mesh, senza considerazione delle intersezioni precedenti, e nel caso di rallentamenti della funzione di Brush dovuti alla crescita del suo costo computazionale si ripresenterebbe il fenomeno della Stroke Segmentation che normalmente è risolto proprio grazie a dei calcoli effettuati a partire dall'ultimo punto campionato per la computazione dei successivi. Di conseguenza non è possibile applicare il Continuous Brush solamente rimuovendo il controllo della distanza tra punti campionati ed intersezioni, ma è necessaria una via di mezzo, un comportamento mediano tra quello standard e quello appena descritto. In *ysculpting*

il Continuous Brush è quindi definito aggiungendo un controllo all'Algoritmo 4 che raggiri il campionamento dello Stroke solo quando l'opzione "continuous" è attiva e la distanza tra l'ultimo punto campionato sullo Stroke e l'intersezione successiva tra cursore e mesh è minore della distanza standard dello Stroke sampling: in questo modo, a distanze minori rispetto a quella prefissata, il Brush è applicato ad ogni intersezione, mentre per distanze maggiori viene eseguito il comportamento standard per evitare la Stroke Segmentation. La modifica riguarda dunque l'algoritmo di Stroke:

Algorithm 7 options_stroke

```

1: Input: camera, mouse_uv, last_uv, mesh, intersection, last_position, radius,
   bvh, continuous
2: Output: stroke_sampling
3: position = eval_position(mesh, intersection.element, intersection.uv)
4: delta_position = distance(position, last_position)
5: stroke_distance = radius * 0.2
6: if last_position == null then
7:     last_position = position
8:     last_uv = mouse_uv
9:     return null
10: end if
11: if continuous ∧ (delta_position < stroke_distance) then
12:     normal = eval_normal(mesh, intersection.element, intersection.uv)
13:     stroke_sampling.insert(position, normal)
14:     return stroke_sampling
15: end if
16: delta_uv = distance(mouse_uv, last_uv)
17: steps = int(delta_position / stroke_distance)
18: if steps == 0 then
19:     return null
20: end if
21: stroke_uv = delta_uv * stroke_distance / delta_position
22: mouse_dir = normalize(mouse_uv - last_uv)
23: for step : steps do
24:     last_uv += stroke_uv * mouse_dir
25:     ray = camera_ray(camera, last_uv)
26:     intersection = intersect_triangles_bvh(bvh, mesh, ray)
27:     position = eval_position(mesh, intersection.element, intersection.uv)
28:     normal = eval_normal(mesh, intersection.element, intersection.uv)
29:     last_position = position
30:     stroke_sampling.insert(position, normal)
31: end for
32: return stroke_sampling

```

4.3 Symmetric Brush

L'opzione tra le più importanti nella modellazione in Digital Sculpting è senza dubbio il Symmetric Brush, che permette all'utente di replicare in maniera simmetrica ad un certo asse di simmetria lo Stroke e di conseguenza il Brush. Ad esempio, i meshes sui quali questa opzione è molto utile sono solitamente quelli che rappresentano manufatti, che in quanto tali sono molto spesso simmetrici, o anche quando si modella l'anatomia umana. Comunque, l'implementazione del Symmetric Brush in *ysculpting* è molto basilare e potrebbe non essere adatta all'utilizzo su ogni tipo di mesh; dopo la trattazione della sua implementazione sarà più facile spiegarne il motivo.

Trovare il punto del mesh simmetrico rispetto ad un certo asse potrebbe sembrare banale, ma non lo è. Infatti si potrebbero ricavare le coordinate simmetriche di una certa posizione rispetto ad un certo asse, ma queste non corrisponderebbero necessariamente ad un punto sulla superficie del mesh: questo sarebbe vero se il mesh sul quale si applica il Symmetric Brush fosse simmetrico rispetto all'asse di simmetria già prima della modifica, ma ovviamente questo non è da prendere come caso base. Dunque, data un posizione, le coordinate simmetriche rappresentano effettivamente la corretta simmetria rispetto ad un certo asse, ma nel caso del Digital Sculpting a queste coordinate va associato poi un punto sul mesh. Sin qui, le posizioni sul mesh sono sempre state individuate tramite intersezioni e anche nel caso del Symmetric Brush di *ysculpting* l'approccio è lo stesso: intersecare il raggio passante per il punto dello spazio corrispondente alle coordinate simmetriche con il mesh, con l'origine di questo raggio stabilita in base all'asse di simmetria scelto. Il punto di intersezione così ottenuto è un'approssimazione del punto simmetrico sul mesh rispetto alla posizione data.

Dato un punto p sul mesh, che corrisponde ad un punto campionato sullo Stroke, la prima computazione da effettuare riguarda le coordinate simmetriche di p rispetto ad un certo asse a : questo punto è indicato come p'_a . Le coordinate simmetriche sono facilmente calcolabili:

$$\begin{aligned} p'_x &= (-p_x, p_y, p_z) \\ p'_y &= (p_x, -p_y, p_z) \\ p'_z &= (p_x, p_y, -p_z) \end{aligned} \tag{4.1}$$

sulle quali occorre derivare il vettore direzione del raggio passante per il punto p'_a . L'idea implementata in *ysculpting* è quella di far partire questo raggio dall'interno del mesh che si sta modellando e farlo passare per il punto simmetrico. Per fare ciò si impone che l'oggetto da modellare sia sempre posizionato con il centro di massa all'origine del sistema di riferimento globale, in modo che l'origine del raggio possa essere definito sugli assi x , y o z globali. Inoltre, per fare in modo che il raggio intersechi il meno possibile altre parti del mesh, si stabilisce il suo origine su un piano, che sarà dunque:

$$\begin{aligned} o &= (0, p_y, 0) \text{ se l'asse di simmetria scelto è } x \\ o &= (p_x, 0, 0) \text{ se l'asse di simmetria scelto è } y \\ o &= (0, p_y, 0) \text{ se l'asse di simmetria scelto è } z \end{aligned} \tag{4.2}$$

La definizione del raggio è dunque $\{o, \frac{p-o}{|p-o|}\}$, e la sua intersezione con il mesh sarà l'approssimazione del punto simmetrico sul mesh stesso a quello dello Stroke. Ovviamenete nel caso di particolari mesh questo approccio non restituisce il vero punto simmetrico che si vuole ottenere con questa opzione, e quindi la sua applicazione non è sempre corretta; tuttavia è un approccio molto semplice che in alcuni casi può risultare molto utile e nonostante tutto è quindi implementato in *ysculpting*.

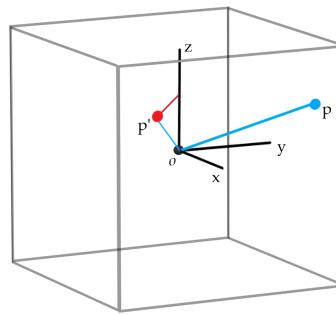


Figura 4.2. Visualizzazione dell'approccio alla simmetria di *ysculpting*: il segmento rosso indica il raggio creato a partire dagli assi del sistema di riferimento globale e il punto p' simmetrico a p .

4.4 Saturation Brush



Figura 4.3. Esempio di Gaussian Brush con opzione "saturazione" attiva in *ysculpting*

Come ultima opzione viene trattato il Saturation Brush, un'opzione che limita la quantità di spostamento massima di ogni vertice del mesh durante uno stesso Stroke (vale a dire per il tempo durante il quale l'utente mantiene il comando per applicare il Brush) per limitare le deformazioni di una particolare zona durante la

modellazione.

L'implementazione del Saturation Brush consiste nel memorizzare un buffer di numeri decimali che abbia lunghezza pari a quella del numero dei vertici del mesh ed inizializzato a zero. Il valore di ogni posizione del buffer sta ad indicare la quantità di spostamento applicata al vertice di riferimento, che inizialmente è nulla: con l'applicazione di un certo Brush però, il valore del buffer relativo ad ogni vertice cui viene modificata la posizione è aggiornato in base alla quantità massima di spostamento che è stata applicata con il Brush attuale. Questa quantità massima nel caso del Gaussian Brush, non è altro che il valore della funzione di densità di probabilità normale calcolato sul punto centrale del Brush, cioè sul punto campionato dello Stroke, come visto nella formula 3.7. La costante N della formula, parametrizzata in base alla trattazione del Capitolo 3, rappresenta la quantità di spostamento massima che il Gaussian Brush può applicare, e dunque corrisponderà ad una saturazione massima: per il vertice al quale è applicata la quantità di spostamento pari a questa costante sarà anche aggiunto il valore massimo di saturazione al relativo valore nel buffer (indicato con s). Per ogni altro vertice il valore che verrà aggiunto al suo corrispondente nel buffer di saturazione è il rapporto tra il valore della costante N ed il valore del Gaussian Brush calcolato su di essi:

$$\begin{aligned}
 s'_i &= s_i + \frac{\varphi_{\mu,\sigma^2}(v_i)}{N} \\
 &= \\
 s'_i &= s_i + \frac{(strength*radius)^{\frac{1}{\sigma^3(2\pi)^{\frac{3}{2}}}} \exp\left\{-\frac{v_{ix}^2 + v_{iy}^2 + v_{iz}^2}{2\sigma^2}\right\}}{(strength*radius)^{\frac{1}{\sigma^3(2\pi)^{\frac{3}{2}}}}} \\
 &= \\
 s'_i &= s_i + \exp\left\{-\frac{v_{ix}^2 + v_{iy}^2 + v_{iz}^2}{2\sigma^2}\right\}
 \end{aligned} \tag{4.3}$$

La memorizzazione di un valore per ogni vertice modificato dal Brush è il meccanismo generale del Saturation Brush per tenere traccia di quanto sia stato spostato a seguito dell'ultima applicazione di un Brush. Tuttavia lo scopo dell'opzione è quella di limitare questo spostamento oltre una certa soglia: essa può essere modificabile a piacere dall'utente, che in questo modo ha la facoltà di gestire la limitazione data dal Saturation Buffer. Quando un vertice del mesh raggiunge questa soglia non deve più essere spostato dal Brush finché il suo valore nel buffer di saturazione non sarà ripristinato allo zero.

Per i vertici del mesh che con il loro ultimo spostamento superino la soglia di saturazione, il valore da sommare ad essi tramite la formula 3.3 sia ridimensionato in base alla differenza tra la soglia di saturazione ed il loro valore attuale del buffer di saturazione:

$$\begin{cases} p'(v_i) = p(v_i) + \frac{d_i}{|d_i|} * \varphi_{\mu,\sigma^2}(v_i) & \text{se } s'_i \leq soglia \\ p'(v_i) = p(v_i) + \frac{d_i}{|d_i|} * (N * (soglia - (s_i))) & \text{se } s'_i > soglia \end{cases} \tag{4.4}$$

perché:

$$(s'_i > soglia) \Rightarrow 0 \leq (soglia - s_i) \leq 1 \tag{4.5}$$

L'implicazione espressa nella formula 4.5 è sempre vera, poiché il valore massimo che possa essere aggiunto al buffer di saturazione dovuto ad una sola applicazione di un Brush è pari a 1: i valori del buffer di saturazione sono sempre un rapporto con la costante N , che è il valore massimo possibile applicabile da una sola applicazione di un Brush, e dunque al massimo questo rapporto è pari a 1 ($\frac{N}{N}$).

In questo modo la quantità di spostamento applicata da un Brush è correttamente ridimensionata in base alla differenza del nuovo valore s'_i e la soglia del buffer di saturazione.¹

L'implementazione in *ysculpting* mette insieme le formule 4.3 e 4.4:

Algorithm 8 new_brush_with_saturation

```

1: Input: mesh, stroke_sampling, radius, hash_grid, bvh, saturation, buffer,
   threshold
2: [...]
3: if saturation then
4:   max_height = gaussian_distribution(position, position,
5:   0.7, scale_factor, strength, radius)
6:   for neighbor : neighbors do
7:     saturation_val = buffer[neighbor]
8:     if saturation_val == threshold then
9:       continue
10:    end if
11:    gauss_height = gaussian_distribution(position,
12:    neighbor, 0.7, scale_factor, strength, radius)
13:    gauss_ratio = gauss_height / max_height
14:    if saturation_val + gauss_ratio > threshold then
15:      gauss_height = max_height * (threshold -
16:      saturation_val)
17:      gauss_ratio = threshold - saturation_val
18:    end if
19:    buffer[neighbor] += gauss_ratio
20:    neighbor += normal * gauss_height
21:  end for
22: else
23:   for neighbor : neighbors do
24:     gauss_height = gaussian_distribution(position,
25:     neighbor, 0.7, scale_factor, strength, radius)
26:     neighbor += normal * gauss_height
27:   end for
28: end if

```

¹Questo è vero solo con un valore minimo della soglia pari a 1, ma imporre per esso un valore minore di 1 equivrebbe a modificare il parametro *strength* del Brush e porre la soglia ad 1; quindi si assume che il valore minimo che possa essere attribuito alla soglia si pari a 1.

Capitolo 5

Texture Brush

A volte nella modellazione 3D, ed in particolare nel Digital Sculpting, può risultare difficile per l'utente modellare forme particolarmente complesse o strutturate sul mesh, anche perché spesso gli strumenti a disposizione hanno delle limitazioni. Cosa può fare allora l'utente nel caso in cui voglia, ad esempio, scolpire le scaglie di un drago come in figura 5.1?



Figura 5.1. Un mesh molto complesso visualizzato in *y sculpting*
(model by Oliver Laric: <https://www.turbosquid.com/it/Search/Artists/oliverlaric>)

Queste sono costituite di geometrie molto piccole, dettagliate, precise e complesse, e molto difficilmente anche un artista esperto riuscirebbe a replicare un effetto simile a mano libera, se non impiegando moltissimo tempo (e con moltissima pazienza!). Per questo tipo di operazioni l'utente necessita di strumenti appositi che ne automatizzino l'utilizzo, spostando il carico di lavoro sulla macchina: è questo il caso del Texture Brush, uno strumento creato appositamente per replicare della geometria complessa in maniera quasi del tutto automatica sulla superficie di un mesh a partire da un'immagine rappresentativa. Questa immagine è una texture, che per essere

applicata correttamente deve soddisfare un requisito: essere un alpha texture (o il alternativa deve essere codificata con il canale alpha oltre che ai canonici RGB). Normalmente le alpha textures sono utilizzate in Computer Grafica nel texture mapping per indicare la quantità di trasparenza della texture, ma nel caso del Digital Sculpting possono essere reinterpretate in modo tale che il canale alpha ne indichi l'altezza: non siano più viste come immagini vere e proprie, costituite di colori, ma come immagini che indichino il rilievo da applicare al mesh tramite il Brush, con 0 (il nero) per "nessun rilievo" e 1 per "rilievo massimo"; tutti i numeri nell'intervallo [0, 1] corrispondano alle gradazioni tra "nessun rilievo" e "rilievo massimo". Le texture inoltre sono applicate ad un intero mesh nelle situazioni classiche, con le modalità viste nel Capitolo 1, mentre nel Digital Sculpting è come al solito l'utente che deve tracciare uno Stroke sull'oggetto 3D per definire i vertici ai quali vada applicato il rilievo derivante dalla texture.

5.1 Stroke Parameterization

Come l'applicazione classica delle textures sull'intero mesh ne richiede la parametrizzazione completa per associare ad ogni punto di esso un punto sull'immagine, tramite quelle che sono chiamate texture coordinates, anche il Texture Brush ha lo stesso requisito; il meccanismo di applicazione è lo stesso, con la grande differenza che nel primo caso la parametrizzazione è calcolata, mentre nel secondo la parametrizzazione è limitata ad un sottoinsieme dei vertici del mesh; la parametrizzazione per il Texture Brush inoltre possiede anche una direzione, che è quella dello Stroke tracciato dall'utente. Nel Digital Sculpting è dunque necessaria una Stroke Parameterization [5] che riesca ad associare ad ogni punto dell'alpha texture un vertice del mesh tra quelli nell'area d'azione del Brush.

5.1.1 Upwind Averaging

L'algoritmo che verrà utilizzato per la parametrizzazione si basa su quello di Discrete Exponential Map (DEM) [6], il cui scopo è quello di mappare un'immagine 2D su una superficie 3D in modo tale che il centro dell'immagine sia in un punto stabilito sulla superficie stessa, attribuendo ai vertici del mesh delle coordinate planari. L'algoritmo di Stroke Parameterization in più, mappa l'immagine non a partire da un singolo punto centrale ma lungo più punti sulla superficie, che corrispondono al campionamento dello Stroke in *ysculpting*, applicando anche una direzione alla mappatura. Per garantire una corretta parametrizzazione direzionata lungo uno Stroke, non basta applicare diverse parametrizzazioni su ognuno dei punti campionati ma occorre una singola parametrizzazione consistente. Per adattare questa parametrizzazione al caso di un mesh 3D, è necessario utilizzare distanze geodesiche al posto delle distanze calcolate con la formula 2.10 per definire ogni punto nel raggio d'azione del Brush rispetto al punto centrale dello Stroke: le distanze geodesiche sono infatti un'approssimazione della distanza che un punto sul mesh avrebbe rispetto al punto centrale di una texture che venisse "appoggiata" idealmente sul mesh stesso, perché rappresentano la distanza tra due punti calcolata seguendo la superficie.

L'algoritmo DEM mappa in maniera incrementale i vertici del mesh attorno ad un

punto p sul suo piano tangente T_p , tramite i loro vicini: assumendo che un punto q sia già stato mappato su T_p , il DEM mappa un vicino x di q prima mappando x sul piano tangente T_q in q tramite proiezione, poi scalando questa proiezione per un fattore $|q - p|$ per preservarne la lunghezza originaria ed infine trasferendo la mappatura ottenuta $T_q(x)$ in $T_p(x)$. Quest'ultimo passaggio è possibile prima calcolando i sistemi di riferimento F_q ed F_p dei punti q e p rispettivamente, e poi allineandoli tramite una rotazione tri-dimensionale $R(q, p)$:

$$T_p(x) = T_p(q) + R(q, p)T_q(x) \quad (5.1)$$

I sistemi di riferimento F_x di cui sopra non sono altro che una base dello spazio vettoriale costruita a partire dall'origine x e dalla normale n_x al vertice del mesh che ne costituisce un vettore.

Nell'approccio del DEM quindi, per parametrizzare un vertice x del mesh su T_p è necessario solo un vicino q per il quale sia già definito $T_p(q)$: questo vicino è determinato in qualche modo dall'applicazione dell'algoritmo Dijkstra sul grafo costruito sulle adiacenze tra vertici del mesh, che calcola gli shortest paths dall'origine (e cioè dal punto p) verso gli altri nodi (cioè gli altri vertici). L'algoritmo di Stroke Parameterization che si sta trattando attua però una modifica a questo approccio, definendo quello che è chiamato "Upwind Averaging": con questo metodo, non viene utilizzato solo un vicino del vertice x per calcolare $T_p(x)$, ma vengono considerati tutti i vicini q_i del vertice x , per i quali sia già calcolato $T_p(q_i)$, per calcolare più valori di $T_p(x)$ corrispondenti ai vari $T_{q_i}(x)$, per poi fare una media dei risultati. Aggiungendo infine dei pesi alla media di questi valori, si ottiene una versione del DEM più robusta [5].

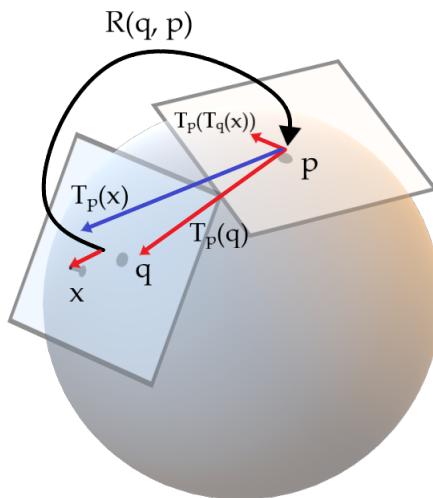


Figura 5.2. Visualizzazione dell'approccio DEM alla base dell'Upwind Averaging

5.1.2 Frame-Propagation

Un altro aspetto per il quale la Stroke Parameterization si discosta dall'algoritmo DEM sul quale si basa è la propagazione dei sistemi di riferimento sui vertici del mesh. Infatti nell'approccio DEM il calcolo delle coordinate planari di un punto x passa per il piano tangente T_q di un suo vicino q (e nel caso della Stroke Parametrization passa per più vicini, come visto), che poi viene ruotato per farlo coincidere con T_p ed ottenere le coordinate planari di x . L'alternativa proposta nell'algoritmo di Stroke Parameterization è di calcolare le coordinate planari del punto x tramite un vicino q_i prima spostando il sistema di riferimento F_p del punto p in q , ottenendo il sistema di riferimento $F_{p \rightarrow q_i}$, e poi calcolando $T_p(q_i) + T_{q_i}(x)$ sulla base del sistema di riferimento ottenuto. Il metodo di ottenimento del sistema di riferimento $F_{p \rightarrow q_i}$ è chiamato Parallel Transport, e consiste nel propagare il sistema di riferimento dal punto iniziale p a tutti i vicini q_i che vengono visitati in qualche ordine dall'Algoritmo Dijkstra attraverso la rotazione minima dei sistemi di riferimento F_p ed F_{q_i} , vale a dire la rotazione che allinea le normali n_p ed n_{q_i} attorno all'asse $n_p \times n_{q_i}$. Questo processo è poi ripetuto dai vicini di p ai loro vicini, e così fino alla fine della visita Dijkstra. Inoltre, per ottenere maggiore robustezza anche nella Frame Propagation, si potrebbe effettuare il Parallel Transport lungo molteplici percorsi tramite l'algoritmo Dijkstra per poi fare una media dei risultati (figura 5.3).

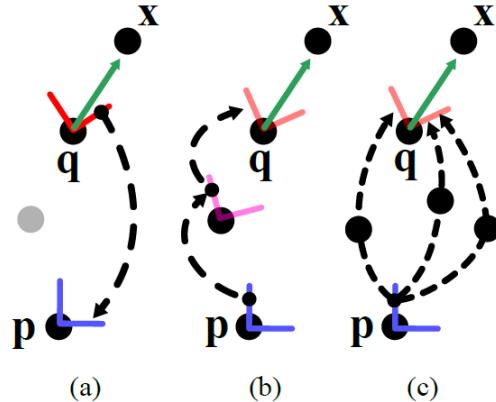


Figura 5.3.

(a) approccio DEM, (b) parallel transport, (c) averaging dei risultati di percorsi multipli
(R. Schmidt)

In realtà non è necessario computare veramente il Parallel Transport lungo più percorsi (sarebbe anche computazionalmente molto costoso), ma è sufficiente aggiungere il passaggio di Frame Propagation a quello precedente di Upwind Averaging: così quando si effettua il calcolo di $T_p(x)$ tramite un vicino q_i , si effettua anche il Parallel Transport $F_{p \rightarrow q_i \rightarrow x}$.

5.1.3 Combinazione delle modifiche al DEM

Partendo dal Discrete Exponential Map si sono definiti gli aspetti di un approccio che è più adatto alle superfici di mesh 3D e la cui computazione è diventata esclusivamente

locale al vertice che volta per volta si considera: verranno successivamente definite le formulazioni per il calcolo delle coordinate planari e dei vari sistemi di riferimento a partire dal vertice di riferimento e considerando esclusivamente i suoi vicini nel grafo delle adiacenze del mesh. Questa caratteristica permette inoltre di adattare l'approccio non solo ad un singolo punto iniziale ma a vari punti, che nel caso del Digital Sculpting corrispondono ai punti campionati dello Stroke. I calcoli da effettuare per questi punti iniziali è ovviamente il caso base, che deve essere gestito a parte rispetto al resto dell'algoritmo. Dato l'insieme di punti che costituiscono lo Stroke $S = \{s_i\}$ ed l'insieme delle corrispondenti normali $N = \{n_i\}$, dato il *radius* scelto per il Brush e data la lunghezza geodesica dall'inizio dello Stroke fino ad un punto di esso $s_i \in S$, indicata con $\alpha(s_i)$, per ognuno di essi occorre calcolare le coordinate planari ($P(s_i)$) ed i frames ($F(s_i)$):

$$\begin{aligned} P(s_i) &= (\alpha(s_i), radius) \\ \left\{ \begin{array}{ll} F(s_i) = (n_i, e_1 = (n_i \times \frac{s_{i+1}-s_i}{|s_{i+1}-s_i|}), e_2 = n_i \times e_1) \\ \quad \wedge \\ F(s_{|S|-1}) = (n_{|S|-1}, e_1 = (n_{|S|-1} \times \frac{s_{|S|-1}-s_{|S|-2}}{|s_{|S|-1}-s_{|S|-2}|}), e_2 = n_{|S|-1} \times e_1) \\ F(s_0) = (n_0, e_1 = (n_0 \times \frac{c-n_0}{|c-n_0|}), e_2 = n_0 \times e_1) \quad se \quad |S| = 1 \end{array} \right. & (5.2) \end{aligned}$$

con c un punto casuale nello spazio virtuale.

Con la formula 5.2, si indica che le coordinate planari dei punti campionati dello Stroke si trovano sempre a metà dell'asse y , mentre sull'asse x hanno coordinata pari alla lunghezza geodesica dello Stroke in quel punto. La formula 5.2 indica anche la formazione delle basi vettoriali sui punti campionati dello Stroke, che hanno come asse z sempre la normale n_i al punto e l'asse x (e_1 nella formula 5.2) individuato dalla direzione del vettore che parte dal vertice dello Stroke considerato ed arriva al prossimo (l'asse y è ottenuto di conseguenza per ortogonalità, ed è e_2 nella formula 5.2). In questo modo i sistemi di riferimento di questi punti iniziali seguono la direzione dello Stroke con l'asse delle x , come mostrato in figura 5.4.

Successivamente è possibile inserire i punti s_i nel grafo delle adiacenze dei vertici del mesh e applicare su di esso la visita Dijkstra, che parta proprio dai punti s_i inizializzati a distanza 0 nel grafo. Come un vertice x viene visitato e la sua distanza dalla sorgente viene stabilita, si calcolano le sue coordinate planari:

$$P(x) = \sum_{q_i \in N_u(x)} w(q_i, x)(P(q_i) + T_{q_i}(x)) \quad (5.3)$$

con $N_u(x)$ l'insieme dei vicini di x nel grafo che sono già stati visitati, $w(q_i, x) = 1/(|q_i - x| + \epsilon)$ un peso sulla distanza tra i vertici x e q_i e ϵ il numero in virgola mobile più piccolo che la macchina può memorizzare.

Fatto ciò è necessario che si calcoli anche la base dello spazio vettoriale con origine in x e con un vettore della base già conosciuto, ovvero la normale n_x . Dunque è

sufficiente ottenere un solo altro vettore e_1 affinché la base possa essere definita; l'ultimo vettore e_2 infatti sarà derivato per ortonormalità da n_x e e_1 :

$$e_1(x) = \sum_{q_i \in N_u(x)} w(q_i, x) R(q_i, x) e_1(q_i) \quad (5.4)$$

con $R(q_i, x)$ la rotazione che porta n_q in n_x , cioè la rotazione attorno all'asse $n_q \times n_x$ di un angolo $\text{acos}(n_q \cdot n_x)$.

La visita Dijkstra termina quando si raggiungono i primi vertici la cui distanza geodesica è maggiore del parametro *radius*; terminata la visita, ogni vertice all'interno dell'area di azione del Brush lungo lo Stroke avrà associate delle coordinate planari e una base (figura 5.4).

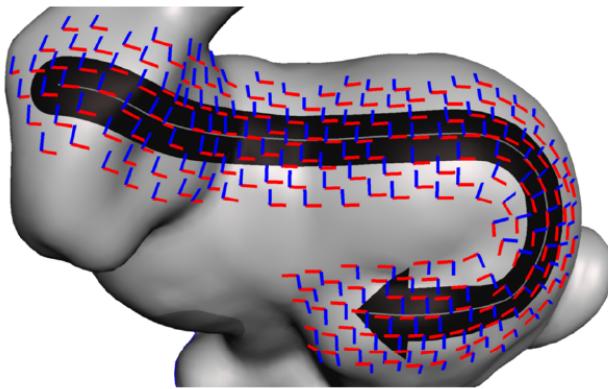


Figura 5.4. Visualizzazione del flusso di sistemi di riferimento creati con la Frame Propagation, a partire dai sistemi di riferimento iniziali mostrati sulla linea bianca al centro dello Stroke, con i quali si possono calcolare le coordinate planari.
(R. Schmidt)

5.1.4 Parametrizzazione e implementazione

Le coordinate planari ottenute dall'algoritmo di Stroke Parameterization sono una parametrizzazione dello Stroke e del campo di azione del Brush, ma non sono ancora nell'intervallo $[0, 1]$ con il quale è parametrizzata una texture: è necessario un piccolo passo successivo, con il quale si convertono le coordinate planari nell'intervallo $[0, 1]$ rispetto ad un parametro scelto. Nel caso di *ysculpting*, questo parametro è *radius*: in questo modo si ottiene una parametrizzazione in $[0, 1]$ nell'area di azione del Brush sul primo punto campionario dello Stroke, che significa che si avrà uno "stampo" completo della texture scelta alla prima applicazione del Texture Brush. Con i successivi campionamenti dello Stroke, si avrà una parametrizzazione che va oltre il valore 1, ma nell'associare questi punti alla texture la loro parametrizzazione viene comunque campionata nell'intervallo $[0, 1]$ attraverso l'operazione modulo. Con questo approccio, l'effetto ottenuto sarà quello di ripetizione continua della texture seguendo la lunghezza dello Stroke, mentre in ampiezza la texture non verrà ripetuta. In figura 5.5 sono rappresentati i risultati del Texture Brush in *ysculpting* utilizzando il parametro *radius* ed altri.

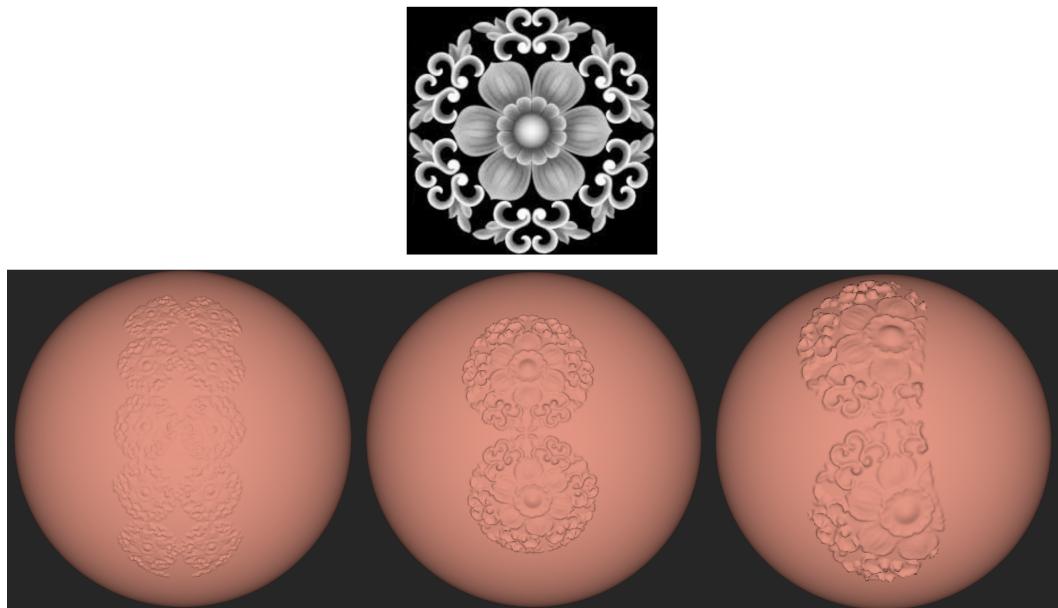


Figura 5.5. Tre scelte differenti per il riferimento della parametrizzazione nell'applicazione al mesh della texture in alto, create in *ysculpting*:
sinistra: valore minore di *radius*, con conseguente ripetizione della texture sia sulla lunghezza dello Stroke che sull'ampiezza del Brush
centro: valore uguale a *radius*, con ripetizione sulla lunghezza dello Stroke ma non sull'ampiezza del Brush
destra: valore maggiore di *radius*, con ripetizione sulla lunghezza dello Stroke ma "taglio" della texture sull'ampiezza del Brush
(texture by Less-Ordinary Designer, Sara Luke)

A questo punto si ha tutto il necessario per applicare al Digital Sculpting la Stroke Parameterization. Il primo passo è adattare l'inizializzazione dei punti dello Stroke al contesto di *ysculpting*: secondo l'algoritmo, una volta calcolate le coordinate planari e i sistemi di riferimento iniziali, i punti che formano lo Stroke, che non sono vertici del mesh (Capitolo 2), vanno aggiunti al grafo delle adiacenze tra vertici che rappresenta il mesh. Fare ciò, significherebbe modificare la topologia del mesh ogni volta che venisse campionato un nuovo punto sullo Stroke e ricalcolare tutte le adiacenze tra vertici, che sono due operazioni abbastanza lunghe e complesse anche per mesh con un numero di vertici contenuto. Dovendo rispettare il requisito di computazione Real-Time, questo approccio diventerebbe controproducente, e dunque la soluzione adottata al suo posto è l'approssimazione del punto dello Stroke con il vertice del mesh più vicino: ottenuto il punto campionato tramite intersezione di un raggio con il mesh, si conosce già la primitiva intersecata ed i relativi vertici; con le coordinate baricentriche (ottenute anch'esse dall'intersezione) è possibile stabilire quale dei tre vertici della primitiva è più vicina al punto d'intersezione seguendo l'Algoritmo 9. In questo modo non sarà necessario ricreare continuamente il grafo.

La fase successiva è il calcolo di coordinate planari e sistemi di riferimento per i punti dello Stroke seguendo la formula 5.2, implementati tramite l'Algoritmo 10 e

Algorithm 9 closest_vertex

```

1: Input: mesh, coords, primitive
2: Output: vertex
3: if  $coords.x < 0.5 \wedge coords.y < 0.5$  then
4:   return primitive.x
5: end if
6: if  $coords.x > coords.y$  then
7:   return primitive.y
8: end if
9: return primitive.z

```

l’Algoritmo 11 rispettivamente. L’Algoritmo 11 inoltre implementa, per ogni punto s_i sullo Stroke tale che $s_i \neq s_0 \wedge s_i \neq s_{|S|-1}$, una media delle direzioni $e_1(s_{i-1})$ e $e_1(s_{i+1})$ per ottenere un flusso di sistemi di riferimento più "smussato".

Algorithm 10 compute_stroke_coords

```

1: Input: coords, stroke_sampling, radius, vertices, positions
2: Output: coords
3: coords[stroke_sampling[0]] = {radius, radius}
4: vertices.insert(stroke_sampling[0])
5: for  $i : from 1 to stroke\_sampling.size()$  do
6:   edge = positions[stroke_sampling[i]] - positions[stroke_sampling[i - 1]]
7:   coords[stroke_sampling[i]] = {coords[stroke_sampling[i - 1]].x + length(edge),
      radius}
8:   vertices.insert(stroke_sampling[i])
9: end for
10: return coords

```

Poi sono da implementare le formule 5.3 e 5.4 per il calcolo di coordinate planari e sistemi di riferimento dei vertici: l’Algoritmo 12 e l’Algoritmo 13 sono l’implementazione indicativa in *ysculpting*.

Nell’Algoritmo 13 la rotazione R definita nella formula 5.4 è calcolata a partire dalle basi dei sistemi vettoriali con origine in q_i ed x , poiché la rotazione che si necessita nella formula è quella tra n_{q_i} ed n_x , e dunque qualsiasi base costruita su questi due vettori permette il calcolo della matrice rotazione corretta. Dunque:

$$\begin{aligned}
RF(x) &= F(q_i) \\
&= \\
R &= F(q_i)F(x)^T
\end{aligned} \tag{5.5}$$

Algorithm 11 compute_stroke_frames

```

1: Input: frames, stroke_sampling, positions, normals
2: Output: frames
3: for  $i : from 0 to stroke\_sampling.size() - 1$  do
4:   curr = stroke_sampling[i]
5:   next = stroke_sampling[i + 1]
6:   dir = positions[next] - positions[curr]
7:   z = normals[curr]
8:   y = cross(z, normalize(dir))
9:   x = cross(y, z)
10:  frames[curr] = {x, y, z}
11: end for
12: if  $stroke\_sampling.size() == 1$  then
13:   frames[stroke_sampling[0]] = random_basis_fromz(normals[stroke_sampling[0]])
14: else
15:   final = stroke_sampling[stroke_sampling.size() - 1]
16:   prev = stroke_sampling[stroke_sampling.size() - 2]
17:   dir = positions[final] - positions[prev]
18:   z = normals[final]
19:   y = cross(z, normalize(dir))
20:   x = cross(y, z)
21:   frames[final] = {x, y, z}
22: end if
23: for  $i : from 1 to stroke\_sampling.size() - 1$  do
24:   curr = stroke_sampling[i]
25:   next = stroke_sampling[i + 1]
26:   prev = stroke_sampling[i - 1]
27:   dir = frames[prev].x + frames[next].x
28:   z = normals[curr]
29:   y = cross(z, normalize(dir))
30:   x = cross(y, z)
31:   frames[curr] = {x, y, z}
32: end for
33: return frames

```

Algorithm 12 compute_coordinates

```

1: Input: coords, frames, positions, node, neighbor, weight
2: current_coord = coords[node]
3: edge = positions[node] - positions[neighbor]
4: projection = project_onto_plane(frames[neighbor], edge)
5: new_coord = coords[neighbor] + projection
6: avg_length = (length(current_coord) + length(new_coord)) / 2
7: new_dir = normalize(current_coord + new_coord)
8: coords[node] = current_coord == zero2f ? new_coord : new_dir * avg_length;
9: return coords

```

Algorithm 13 compute_frame

```

1: Input: frames, normals, node, neighbor, weight
2: Output: frames
3: current_dir = frames[node].x
4: rotation      =      random_basis_fromz(normals[neighbor])           *
   transpose(random_basis_fromz(normals[node]))
5: neighbor_dir = frames[neighbor].x
6: current_dir = current_dir + (rotation * weight * neighbor_dir)
7: frames[node].z = normals[node]
8: frames[node].y = cross(frames[node].z, normalize(current_dir))
9: frames[node].x = cross(frames[node].y, frames[node].z)
10: return frames

```

5.1.5 Il Geodesic Solver

L'unico aspetto rimasto da modellare in forma di algoritmo è il grafo delle adiacenze tra vertici e la visita Dijkstra su di esso. Il grafo utilizzato in *ysculpting* è il Geodesic Solver di Yocto/GL[3], una struttura dati che mantiene per ogni vertice del mesh (che nel grafo acquista la valenza di nodo) tutti i vertici cui è collegato (che nel grafo diventano i vicini), con i rispettivi spigoli per il calcolo delle distanze geodesiche. Questa struttura è già presente nella libreria Yocto/GL, e dunque non si entra nel merito della sua implementazione. La visita Dijkstra invece è stata implementata come segue:

Algorithm 14 dijkstra

```

1: Input: geodesic_solver, stroke_sampling, distances, max_distance, update
2: compare = [&](int i, int j) {return distances[i] > distances[j]}
3: priority_queue<int, vector<int>, decltype(compare)> queue(compare)
4: for sample : stroke_sampling do
5:   queue.push(sample)
6: end for
7: while !queue.empty() do
8:   node = queue.top()
9:   queue.pop()
10:  distance = distances[node]
11:  if distance > max_distance then
12:    continue
13:  end if
14:  for arc : solver.graph[node] do
15:    new_distance = distance + arc.length
16:    update(node, arc.node, new_distance)
17:    if new_distance < distances[arc.node] then
18:      distances[arc.node] = new_distance
19:      queue.push(arc.node)
20:    end if
21:  end for
22: end while

```

Questo algoritmo utilizza una Priority Queue, una struttura dati che rappresenta un contenitore adattivo progettato in modo tale che il primo elemento sia sempre il maggiore rispetto ad un qualche criterio, definito dalla funzione lambda *compare*: in *ysculpting* la visita passerà sempre prima sui vertici la cui distanza geodesica è maggiore, e poichè le distanze iniziali dei vertici che non sono sullo Stroke sono pari al numero in virgola mobile maggiore che la macchina può rappresentare, questo garantisce che la visita passi sempre al più una volta su ogni vertice del grafo che sia a distanza dal punto campionario dello Stroke minore a *radius*. Il parametro *update* poi, è la funzione lambda che viene chiamata ad ogni visita di un vicino di ogni nodo, ed è quella che contiene le operazioni da effettuare, cioè quelle dell’Algoritmo 12 e dell’Algoritmo 13. Infine, l’utilizzo del Geodesic Solver di Yocto/GL implica che i mesh da utilizzare, per garantire un corretto funzionamento del Texture Brush, debbano avere una topologia "isotropica", a causa di come il Geodesic Solver stesso è implementato.

5.2 Dalla texture al Brush

Parametrizzato lo Stroke e i vertici del mesh nel raggio di azione del Brush, occorre valutare il valore della texture per ognuno di questi vertici ed applicarlo alle loro normali per ottenere un rilievo. Come detto nell’introduzione di questo capitolo, al bianco, che è rappresentato come {255, 255, 255} in RGB e {1, 1, 1} in Yocto/GL, corrisponde il rilievo massimo, mentre al nero, rappresentato come {0, 0, 0} sia in RGB che nella rappresentazione di Yocto/GL, corrisponde l’assenza di rilievo. Già dalla rappresentazione di questi colori (e della relativa scala di grigi) in Yocto/GL si può evincere che possano essere considerati come dei vettori in tre coordinate, che corrispondono ai fattori R, G, e B. Il modulo di questi vettori, dunque, può essere solo nell’intervallo [0, 1], con 0 il modulo del vettore del nero e 1 il modulo del vettore del bianco: partendo da una quantità fissata di spostamento *N*, essa può essere moltiplicata per il modulo dei vettori dei colori, ottenendo così una percentuale di questa stessa quantità in [0, *N*], e ottenendo quindi rilievi diversi per vertici cui corrispondono colori diversi. Questa quantità *N* può essere calcolata, come nel caso di *ysculpting*, dalla funzione di densità di probabilità normale nel suo punto centrale, che corrisponde quindi al picco massimo della curva gaussiana. Seguendo questo approccio si può così permettere all’utente di scegliere il fattore *strength* descritto nel Capitolo 3 anche per il Texture Brush, che ne indichi l’altezza massima del rilievo, ottenuta modulando l’altezza massima della curva normale.

Algorithm 15 texture_brush

```

1: Input: mesh, vertices, texture, coords, strength, radius, positions, normals,
   negative
2: scale_factor = 3.5 / radius
3: max_height = gaussian_distribution(zero3f, zero3f, 0.7, scale_factor, strength,
   radius)
4: for i : vertices do
5:   uv = coords[i]
6:   color = eval_image(texture, uv)
7:   height = lenght(color)
8:   normal = normals[i]
9:   if negative then
10:    normal = -normal
11:   end if
12:   height *= max_height
13:   positions[i] += normal * height
14: end for

```

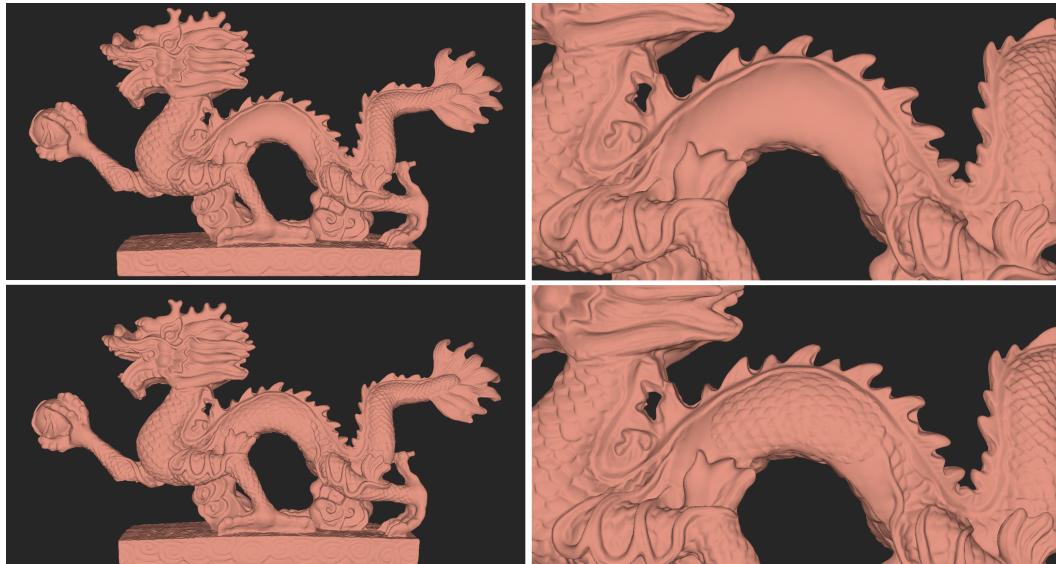


Figura 5.6. Un esempio di applicazione di Texture Brush in *ysculpting* per creare una geometria complessa: delle scaglie sulla pelle di un dragone
 (base model by Oliver Laric:
<https://www.turbosquid.com/it/Search/Artists/oliverlaric>)

Capitolo 6

Smooth Brush

L'ultimo strumento che viene presentato in questa trattazione è lo Smooth Brush, il cui compito è quello di smussare l'area della superficie alla quale viene applicato. È uno strumento molto utile e di conseguenza molto utilizzato, perché può conferire al mesh sul quale si applica un aspetto più omogeneo, andando ad avvicinare alcuni vertici quando le loro posizioni sono molto diverse. Questo strumento è stato incluso nello sviluppo del software *ysculpting* poiché completa l'insieme delle funzionalità di base che un software di Digital Sculpting necessita per fornire all'utente il minimo indispensabile per questa tipologia di modellazione 3D. Inoltre, uno degli scopi principali del Digital Sculpting è l'emulazione dell'analogia tecnica reale in uno spazio virtuale, come introdotto nel Sommario, per cui la funzionalità di smussamento della superficie di un mesh è funzionale proprio a questo proposito, perché è un'operazione quasi obbligatoria in molte categorie di modellazione e di scultura reali per la creazione di certe tipologie di superficie: l'emulazione di queste stesse superfici in ambito virtuale dunque potrebbe necessitare gli stessi strumenti.

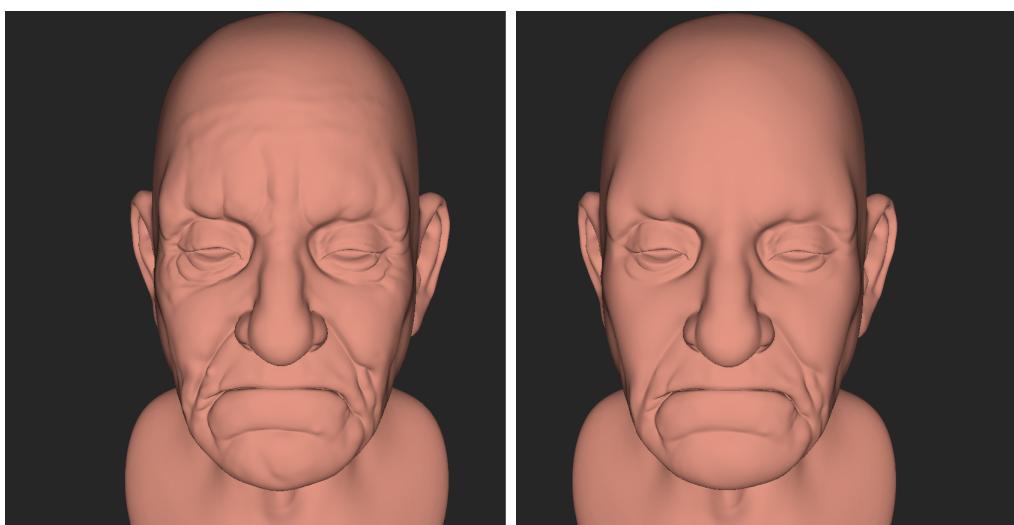


Figura 6.1. Un esempio di caso di utilizzo di Smooth Brush: l'eliminazione delle rughe in alcune zone di un modello di viso di anziano in *ysculpting*
(base model by cvbtruong: <https://www.turbosquid.com/it/Search/Artists/cvbtruong>)

6.1 Laplacian Smoothing

Esistono molti metodi per effettuare lo smussamento della superficie di un mesh, quello utilizzato in *ysculpting* e che viene analizzato successivamente è il metodo chiamato Laplacian Smoothing, basato sull'Operatore di Laplace che prende il nome dal suo ideatore Pierre Simon Laplace. Questo operatore è utilizzato in molti campi della matematica e non solo, ed è un operatore differenziale del secondo ordine definito come la divergenza del gradiente di una funzione in uno spazio euclideo, e può essere applicato da due ad n dimensioni (quindi anche al caso tri-dimensionale) e definito in uno spazio vettoriale, oltre che in uno scalare. Questo operatore è definito su un dominio continuo, quindi quello utilizzato nel caso della Computer Grafica e più nello specifico del Digital Sculpting è il corrispondente Operatore di Laplace Discreto, definito per avere senso su un grafo [7] e quindi su un mesh.

Per comprendere al meglio il funzionamento di questo particolare Brush, si analizzi in prima istanza un caso semplificato di mesh: la polilinea. Una polilinea rappresenta l'approssimazione di una curva come un mesh rappresenta l'approssimazione di una superficie. Una polilinea definita con $P = \{p_i\}$ è un insieme di punti collegati tra loro che possono formare angoli più o meno accentuati: il risultato dello Smooth Brush su una polilinea del genere dovrà essere una nuova polilinea i cui punti p_i vicini non formino angoli eccessivamente acuti. In figura 6.2 viene mostrato un esempio di risultato desiderabile dall'applicazione dello Smooth Brush su una polilinea. Questo risultato è ottenibile per ogni punto p_i tramite una semplice media delle coordinate del punto p_{i-1} precedente e del punto p_{i+1} successivo:

$$p_i = (p_{i-1} + p_{i+1})/2 - p_i \quad (6.1)$$

che corrisponde alla differenza finita discreta della seconda derivata, ovvero all'Operatore di Laplace Discreto in una dimensione [1]:

$$\begin{aligned} p_i &= p_i + \lambda L(p_i) \\ L(p_i) &= \frac{1}{2}(p_{i+1} - p_i) + \frac{1}{2}(p_{i-1} - p_i) \end{aligned} \quad (6.2)$$

con $(0 < \lambda < 1)$ che indica la "velocità" di convergenza verso la posizione calcolata.

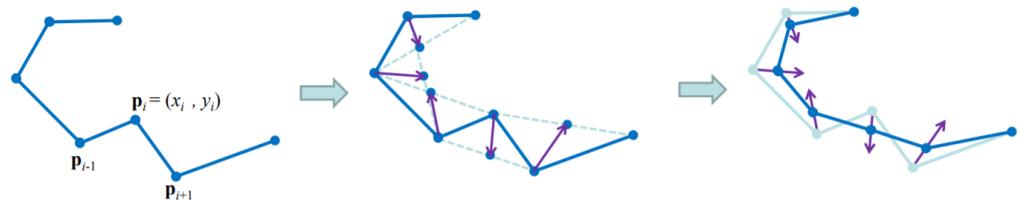


Figura 6.2. Trasformazione di una polilinea sulla quale si applichi lo Smooth Brush (Bruno Levy e Mirela Ben-Chen)

Definito l'Operatore di Laplace in una dimensione per una polilinea, si può ricavare la derivazione in 3D per un mesh. In tre dimensioni, i vicini di un vertice v_i non sono più solamente due vertici (il precedente ed il successivo) ma l'insieme dei vertici $N_u(v_i)$ adiacenti ad esso nel grafo di adiacenze del mesh; anche il peso per la media delle posizioni non sarà dunque $\frac{1}{2}$ ma $\frac{1}{|N_u(v_i)|}$:

$$\begin{aligned} v_i &= v_i + \lambda L(v_i) \\ &= \\ v_i &= v_i + \lambda \left(\frac{1}{|N_u(v_i)|} \left(\sum_{q_i \in N_u(v_i)} q_i \right) - v_i \right) \end{aligned} \quad (6.3)$$

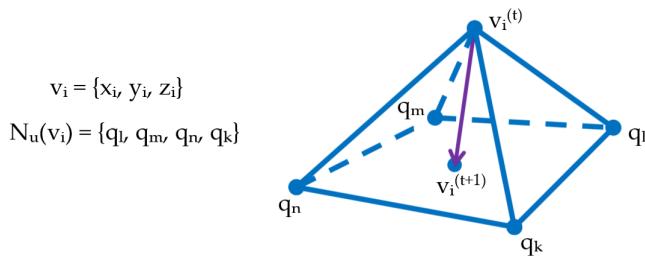


Figura 6.3. Trasformazione di un vertice del mesh sul quale si applichi lo Smooth Brush (Bruno Levy e Mirela Ben-Chen)

Quindi anche nel caso dello Smooth Brush i vertici saranno visitati tramite il grafo delle adiacenze del mesh, e alla visita di ognuno di essi sarà calcolato l'Operatore di Laplace Discreto sui suoi vicini, per ottenere la nuova posizione.

6.1.1 Problematiche del metodo di smoothing

Per quanto questo approccio allo smoothing sia semplice ed effettivamente efficace, porta ad alcuni comportamenti non desiderabili soprattutto da parte della topologia del mesh. In particolare, se si applica lo Smooth Brush come definito precedentemente ad una superficie che è già smussata, questa dovrebbe rimanere la stessa, mantenendo allo stesso tempo geometria e topologia. Similmente, applicare questo Smooth Brush a mesh formati da primitive di dimensioni differenti non dovrebbe portare a risultati differenti: in altre parole, l'operazione di smoothing non dovrebbe dipendere dalla dimensione dei triangoli (o di altre primitive) di cui è costituito un mesh.

Queste due problematiche (mostrate in figura 6.4) sono presenti nello Smooth Brush che implementa la formula 6.3, e dipendono dal fatto che la media tra le posizioni dei vicini per definire quale sia la nuova posizione di un vertice è una media non pesata: in realtà già nella formula 6.2 che descrive lo smoothing di una polilinea è presente un "peso" per i punti p_{i-1} e p_{i+1} , che corrisponde a $\frac{1}{2}$, perché due è il numero dei vicini di ogni punto della polilinea e dunque il peso per ognuno di essi è lo stesso, è uniforme. Allo stesso modo nella formula 6.3 per lo smoothing sul mesh sono presenti dei pesi, e corrispondono a $\frac{1}{|N_u(v_i)|}$, perché $|N_u(v_i)|$ è il numero dei vicini di un vertice sul mesh. Dunque anche questi pesi sono uniformi, ed è a causa di ciò che esistono le problematiche descritte. Il comportamento che si vorrebbe introdurre nello Smooth Brush è il mantenimento delle proporzioni delle distanze

tra un vertice e i suoi vicini: ad esempio, preso un vertice v_i per il quale ci siano due vicini q_m e q_n con il primo più vicino a v_i ed il secondo ad una distanza maggiore, si vorrebbe che dopo l'applicazione dello Smooth Brush la posizione di v_i cambi ma rimanga comunque più vicina a q_m che a q_n . Quello che succede realmente è mostrato in figura 6.3, con la precedente posizione $v_i^{(t)}$ che inizialmente è più vicina ai vertici q_l e q_k ma che poi diventa $v_i^{(t+1)}$ che si trova esattamente nel punto centrale del quadrilatero formato dai suoi vicini. Le proporzioni delle distanze relative ai vicini non vengono mantenute.

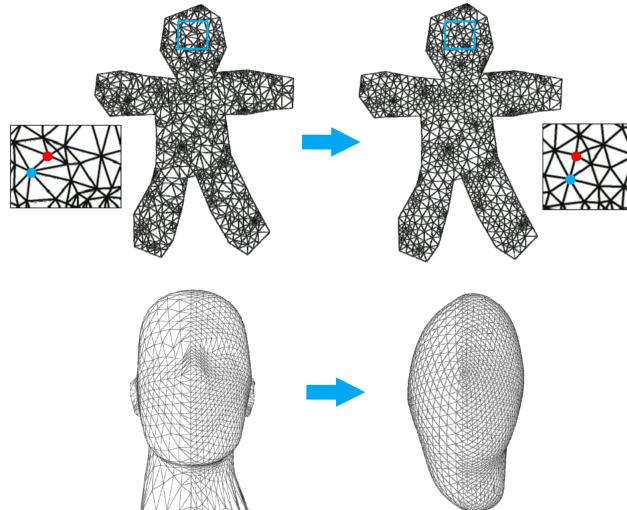


Figura 6.4. Modifica della topologia del mesh (**sopra**) e comportamento non consistente per dimensioni diverse delle primitive (**sotto**) dello Smooth Brush
(Bruno Levy e Mirela Ben-Chen)

6.2 Cotangent Weights

La soluzione alle problematiche descritte di conseguenza è utilizzare una media pesata nella formula 6.3 e determinare quali siano i "pesi" corretti da associare al calcolo. Ancora una volta, per linearità di ragionamento, si analizzi in primo luogo il caso più semplice.

Per una polilinea la scelta dei "pesi" è semplice, perché corrisponde esattamente alla distanza tra due punti vicini: per un punto p_i che abbia due vicini p_j e p_k a distanza l_{ij} ed l_{ik} rispettivamente, i "pesi" possono essere scelti come $w_{ij} = \frac{1}{l_{ij}}$ e $w_{ik} = \frac{1}{l_{ik}}$, e dunque l'Operatore di Laplace Discreto in una dimensione:

$$L(p_i) = \frac{w_{ij}p_j + w_{ik}p_k}{w_{ij} + w_{ik}} - p_i \quad (6.4)$$

L'adattamento al caso tri-dimensionale richiede un parametro per il calcolo dei "pesi" più complesso della semplice distanza tra due vertici vicini v_i e q_i , che solitamente è stabilito essere $\frac{1}{2}(\cotan(\alpha) + \cotan(\beta))$, con α l'angolo individuato dagli spigoli v_i, q_{i-1} e q_i, q_{i-1} e β l'angolo individuato dagli spigoli v_i, q_{i+1} e q_i, q_{i+1}

(figura 6.5). Si utilizza l'operazione cotangente perché questi angoli α e β variano al variare della posizione di v_i rispetto a q_i , e in particolare più i vertici sono vicini minori sono α e β ; per la funzione cotangente più l'angolo è ampio, minore è il valore calcolato dalla funzione, e viceversa; dunque più v_i e q_i sono vicini, minori sono gli angoli α e β e di conseguenza maggiore il valore della cotangente (e quindi del peso).

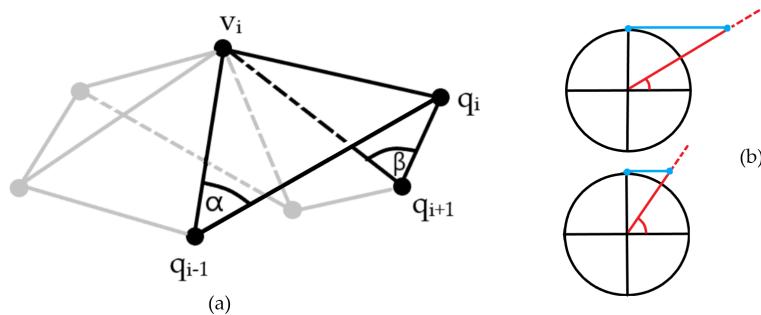


Figura 6.5.

- (a): Raffigurazione del "first ring" di un grafo di adiacenze tra vertici del mesh rispetto ad un vertice v_i , con visualizzazione degli angoli α e β per il calcolo dei cotangent weights
- (b): Rappresentazione della cotangente al variare dell'ampiezza dell'angolo corrispondente

Partendo dalla formula 6.4 riferita al caso uni-dimensionale, si effettui l'adattamento al caso tri-dimensionale tramite l'utilizzo dei "pesi" cotangente e tramite e la fusione con la formula 6.3, il cui difetto è proprio l'utilizzo di "pesi" uniformi [1]:

$$L(v_i) = \frac{1}{\sum_{q_i \in N_u(v_i)} w_{v_i q_i}} \left(\sum_{q_i \in N_u(v_i)} w_{v_i q_i} q_i \right) - v_i \quad (6.5)$$

Grazie alla formula 6.5 e ai "pesi" cotangente, i mesh planari (che sono già smussati) rimangono invariati all'applicazione dello Smooth Brush e il risultato di questa applicazione non dipende più dalla dimensione delle primitive di cui è composto il mesh.

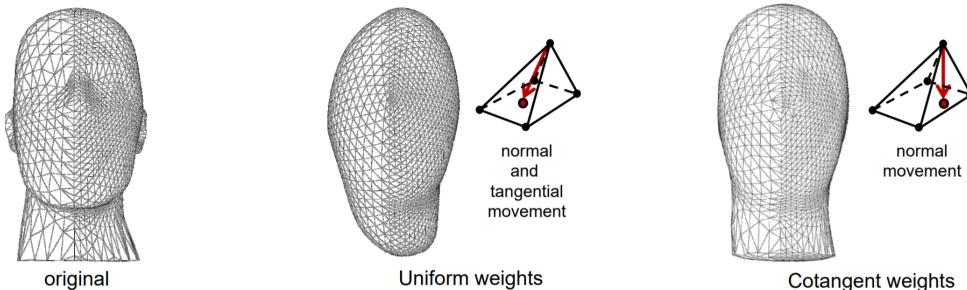


Figura 6.6. Differenze di risultati dello Smooth Brush con "pesi" uniformi e "pesi" cotangente (Bruno Levy e Mirela Ben-Chen)

6.3 Applicazione

Per implementare l'algoritmo di Smooth Brush relativo all'Operatore di Laplace Discreto con "pesi" cotangente, si necessita la possibilità di visitare tutti i vicini di un determinato vertice del mesh per effettuare la media pesata descritta dalla formula 6.5. Come nel caso del Texture Brush si utilizza il grafo delle adiacenze dei vertici del mesh e lo stesso Geodesic Solver già descritto nel Capitolo 5. Per la visita di questo grafo si utilizza nuovamente l'algoritmo Dijkstra, tramite l'implementazione già descritta nell'Algoritmo 14 che fa anche uso della funzione lambda *update* per specificare le operazioni da effettuare ogni volta che si visita un vicino del nodo di partenza. Dovendo utilizzare il Geodesic Solver, come nel caso del Texture Brush è necessario l'utilizzo dell'Algoritmo 9 per l'approssimazione dei punti campionati dello Stroke sui vertici del mesh, che evita di dover ricalcolare il grafo delle adiacenze dei vertici ad ogni aggiunta di un nuovo punto dello Stroke. Altra somiglianza con l'implementazione del Texture Brush è la memorizzazione di un vettore che mantenga le distanze geodesiche calcolate durante la visita Dijkstra e che sia inizializzato con valori pari a zero per i nodi del mesh che sono quelli derivati dall'approssimazione dei punti campionati dello Stroke. Prima di descrivere lo pseudocodice dello Smooth Brush vero e proprio, si definisce quello per il calcolo dei pesi cotangente, che è utilizzato all'interno della funzione lambda *update* durante la visita Dijkstra:

Algorithm 16 laplacian_weights

```

1: Input: positions, adjacencies, node, neighbor
2: Output: weight
3: num_neighbors = adjacencies[node].size()
4: ind = get_index(adjacencies[node], neighbor)
5: prev = adjacencies[node][(ind - 1) ≥ 0 ? ind - 1 : num_neighbors - 1]
6: next = adjacencies[node][(ind + 1) % num_neighbors]
7: v1 = positions[node] - positions[prev]
8: v2 = positions[neighbor] - positions[prev]
9: v3 = positions[node] - positions[next]
10: v4 = positions[neighbor] - positions[next]
11: cot_alpha = cotan(v1, v2)
12: cot_beta = cotan(v3, v4)
13: cotan_max = cos(flt_min) / sin(flt_min)
14: weight = clamp(weight, cotan_max, -cotan_max)
15: return weight

```

Infine, lo pseudocodice per l'intero Smooth Brush con l'utilizzo dell'Algoritmo 16 per il calcolo dei pesi cotangente e dell'Algoritmo 14 per la visita del Geodesic Solver:

Algorithm 17 smooth_brush

```

1: Input: mesh, geodesic_solver, stroke_sampling, positions, strength, radius,
   adjacencies
2: distances = init_distances(stroke_sampling)
3: current_node = -1
4: neighbors = vector<int>{}
5: weights = vector<float>{}
6: update = [&] (node, neighbor, new_distance) {
7:   if current_node == -1 then
8:     current_node = node
9:   end if
10:  if node != current_node then
11:    sum1 = 0
12:    sum2 = 0
13:    for i : from 0 to neighbors.size() do
14:      sum1 += positions[neighbors[i]] * weights[i]
15:      sum2 += weights[i]
16:    end for
17:    positions[current_node] += strength *
18:      ((sum1 / sum2) - positions[current_node])
19:    current_node = node
20:    neighbors.clear()
21:    weights.clear()
22:  end if
23:  neighbors.insert(neighbor)
24:  weights.insert(laplacian_weight(positions, adjacencies, node, neighbor))
25: }
26: dijkstra(solver, stroke_sampling, distances, radius, update)

```

Capitolo 7

Conclusioni

In questo capitolo conclusivo sono esaminati i risultati ottenuti dallo sviluppo di *y sculpting*, tramite le modalità descritte nei capitoli precedenti, rispetto agli obiettivi iniziali.

Gli obiettivi iniziali di questo lavoro di Tirocinio si proponevano di ottenere un software di Digital Sculpting che avesse le funzionalità di base dei programmi completi analoghi. In questo senso, *y sculpting* permette all’utente l’interazione completa con il mesh di un oggetto: può essere caricato un oggetto come base dalla quale partire per effettuare delle modifiche; questo oggetto viene visualizzato nello spazio virtuale e l’utente ha le funzionalità necessarie allo spostamento della camera virtuale in direzioni arbitrarie, per ruotare intorno all’oggetto, per avvicinarsi o allontanarsi da esso, per spostarsi nello spazio; attraverso il cursore del mouse (o di un altro sistema di input del computer, ad esempio con una penna digitale) l’utente può selezionare dei punti sul mesh e definire uno Stroke su di esso, ed ha un feedback visivo sull’area di azione del Brush che ha selezionato tramite una circonferenza che ne indica i bordi; possono essere selezionati tutte le tipologie di Brush di *y sculpting* (Gaussian Brush, Smooth Brush e Texture Brush), ognuno con le relative opzioni disponibili e con la possibilità di selezionare il suo raggio di azione e la forza di applicazione. Tutte queste funzionalità si ritrovano in molti (se non tutti) software di Digital Sculpting, e con la loro combinazione un utente capace ha la possibilità di raggiungere risultati interessanti e di complessità media, anche se ovviamente ciò che sia possibile creare con un software di dimensioni contenute come *y sculpting* è limitato.

Altro requisito fondamentale stabilito all’inizio del processo di sviluppo di *y sculpting* è la risposta Real-Time all’input dell’utente. Per valutare il comportamento del software in analisi dal punto di vista del tempo di calcolo sono necessarie delle considerazioni preliminari. Come già descritto all’inizio di questa trattazione, la parte relativa al Digital Sculpting è implementata esclusivamente su CPU, il che è già un ostacolo intrinseco al raggiungimento di una risposta Real-Time all’input dell’utente, laddove l’utilizzo di un supporto GPU avrebbe risolto già in partenza questo scoglio. Tuttavia questa problematica era stata messa in preventivo, e dunque era ritenuto ragionevole aspettarsi un buon comportamento dal punto di vista del

tempo di calcolo anche su CPU. Gli algoritmi noti implementati in *ysculpting* sono stati accelerati quando possibile attraverso l'utilizzo di strutture dati (Hash Grid, Geodesic Solver) e tecniche precise (Bounding Box) che diminuissero la complessità delle computazioni. La velocità di esecuzione di questi calcoli però, in una tipologia di software qual'è *ysculpting*, può variare molto dall'hardware sul quale si eseguono: la macchina sulla quale è stato eseguito tutto il processo di sviluppo fa parte della tipologia dei "laptop", ed in quanto tale non è tra le soluzioni più performanti per eseguire software di questo tipo. Su questa macchina però, *ysculpting* garantisce una risposta Real-Time nella maggior parte delle situazioni, sin quando agisce su mesh con una risoluzione fino a qualche milione di vertici, che per un software base come quello presentato in questa trattazione (ma anche in moltissimi casi reali) è un numero molto elevato. Di conseguenza è lecito aspettarsi risultati nettamente migliori su macchine più performanti. Il requisito di interazione Real-Time tra utente e programma è dunque considerato soddisfatto.

Il livello del software *ysculpting* raggiunto alla conclusione di questo lavoro di Tirocinio consente in futuro di poter riprendere lo sviluppo e di poter migliorare ulteriormente il programma in termini di ottimizzazione e di completezza delle funzionalità.

Bibliografia

- [1] Ben-Chen Mirela: *Geometry Processing Algorithms* (2012)
Stanford University
- [2] Jansson Erika: *Brush Painting Algorithms* (2004)
Chalmers University of Technology, Gothenburg, Sweden
- [3] Pellacini Fabio: *Yocto/GL tiny C++ Libraries for Data-Driven Physically-based Graphics*
<https://github.com/xelatihy/yocto-gl>
- [4] Roosendaal Ton: *Blender Foundation*
<https://www.blender.org/foundation/>
- [5] Schmidt R.: *Stroke Parameterization* (2013)
Autodesk Research, Toronto, Canada
- [6] Schmidt R. , Grimm C. , Wyvill B. : *Interactive decal compositing with discrete exponential maps. ACM Transactions on Graphics 25, 3 (2006)*
R. Schmidt: University of Calgary
C. Grimm: Washington University in St. Louis
B. Wyvill: University of Calgary
- [7] Wardetzky Max, Mathur Saurabh, Kälberer Felix, Grinspun Eitan: *Discrete Laplace operators: No free lunch* (2007)
Wardetzky Max, Kälberer Felix: Freie Universität Berlin, Germany
Mathur Saurabh, Grinspun Eitan: Columbia University, USA