Alma Mater Studiorum · Università di Bologna

facoltà di scienze matematiche, fisiche e naturali

Corso di Laurea Triennale in Informatica

BLENDER: IL DIETRO LE QUINTE

Tesi di Laurea in Calcolo Numerico

Relatore:
Chiar.mo Prof.
GIULIO CASCIOLA

Presentata da:
SIMONA ARMA

Sessione I
Anno Accademico 2011-2012

Alla mia famiglia

Introduzione

La Computer Graphics 3D è rientrata ormai in numerosissimi campi: dalla progettazione di edifici e macchinari, alla riproduzione di immagini particolareggiate per la medicina o la geologia, fino al puro intrattenimento attraverso videogiochi, cartoni animati ed effetti speciali per il cinema.

Per realizzare un immagine tridimensionale è necessaria una grande quantità di tempo, cura dei particolari, e soprattutto la conoscenza di software per la modellazione e resa 3d.

Tutti i software di questo tipo fanno dell'utente una sorta di regista, permettendogli di aggiungere oggetti alla scenografia, accendere luci, muovere telecamere al fine di ottenere la scena, così come era stata pensata.

Un sistema di questo tipo andrebbe bene solo per pochissimi scopi, poichè le forme utilizzabili possono essere scelte solo tra una serie limitata di primitive messe a disposizione dal sistema. È necessaria quindi una ulteriore fase che viene chiamata **Editing della primitive** per trasformare cubi di default in personaggi con gambe e braccia e cilindri, sfere, curve e superfici in elementi scenografici.

Durante questa fase è possibile intervenire su una sola geometria per volta, ma sarebbe comodo in alcune situazioni fare interagire più oggetti per calcolare la distanza di uno dall'altro, per esempio, o per posizionare un vertice di una geometria alle stesse coordinate di un vertice di una seconda geometria, senza doversi appuntare valori nel passaggio dall'editing di una geometria all'altra.

Proprio questo sarà la base della mia tesi.

Cercherò di capire se l'edit contemporaneo di più geometrie sia realizzabile, andando a studiare il codice sorgente di un programma di modellazione 3D.

Mi servirò per questo compito di uno dei più noti software in questo campo: **Blender**.

Nel mio lavoro andrò ad esplorare l'arrivo tumultuoso di Blender in **Open Source**, caratteristica fondamentale per la mia ricerca, ed andrò a scavare all'interno del codice e della documentazione estrapolando quelle che per me sono informazioni utili al possibile svolgimento di un progetto in tal senso.

Passerò in rassegna le strutture dati che rappresentano le geometrie: dalle più generali, gli **oggetti**, a quelle più particolareggiate (**mesh, curve e superfici**).

Darò un'occhiata alle funzioni che agiscono sugli oggetti (gli **operatori**), ne analizzerò alcune e cercherò di crearne altre per capire meglio il loro funzionamento.

Infine, vedremo come tutte queste strutture e funzioni siano legate insieme da un modello che rappresenta il funzionamento dell'intero sistema.
Questa piccola guida non è assolutamente da ritenersi esaustiva, conoscere tutto il codice sarebbe un compito inimmaginabile, ma tenterò di rendere più chiare almeno alcune parti.

Indice

Elenco delle figure

Capitolo 1

Cos'è Blender e da dove nasce

Dare una definizione che classifichi il software Blender non è facile, poichè non è soltanto un software di modellazione 3D, non è solo un motore di rendering, non è soltanto uno strumento per la creazione di videogiochi. Per chiarirci le idee, sfruttiamo la definizione di chi lo ha creato!

" Blender: model - shade - animate - render - composite - interactive 3d Blender is the free open source 3D content creation suite, available for all major operating systems under the GNU General Public License. "

www.blender.org

Queste parole ci chiariscono molto bene cosa sia Blender: creato per sviluppare contenuti 3d, ne permette la modellazione, l'aggiunta di ombre, l'animazione, il renderig e la produzione di piccoli videogiochi con l'integrazione di una console python e di un text editor.

È un sistema multipiattaforma gratuito, utilizzabile perciò sia da professionisti che da semplici appassionati.

Il codice sorgente è distribuito sotto la licenza GNU General Public License, quindi è possibile consultarlo e modificarlo.

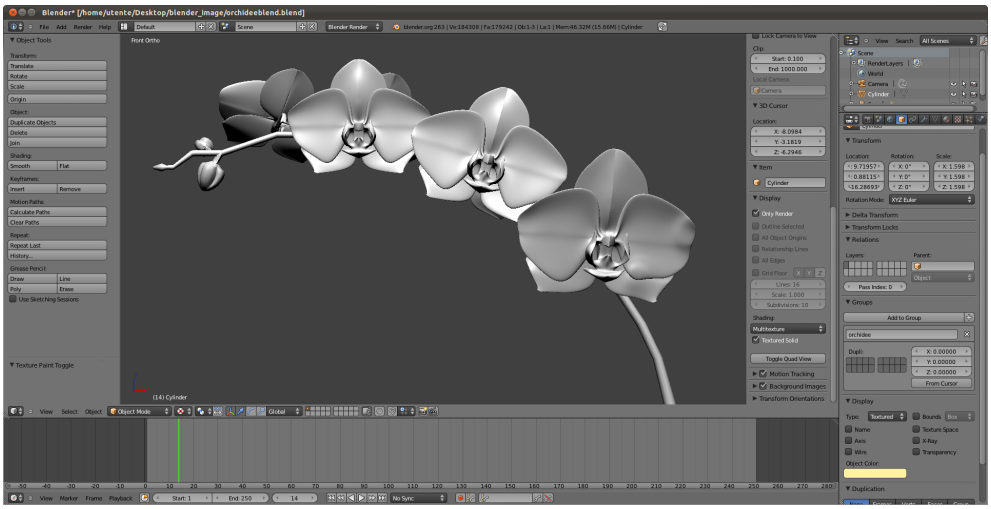

Figura 1.1: Interfaccia utente di Blender 2.62

1.1 Un tormentato arrivo in Open Source

Blender nasce nel 1995 all'interno di uno tra i più importanti studi d'animazione europei: la società olandese NeoGeo.

Dopo un'attenta valutazione, la compagnia decise che lo strumento 3D utilizzato fino a quel momento era troppo scomodo da mantenere e sviluppare, e sarebbe stato necessario riscriverlo dall'inizio.

Il progetto Blender partì sotto la guida di uno dei soci, Ton Roosendaal, che spinto da un notevole spirito d'iniziativa e certo delle potenzialità del nuovo software, aprì una nuova compagnia (la NaN - Not a Number) come distaccamento della NeoGeo, che doveva occuparsi esclusivamente di Blender e di tutto ciò che girava intorno ad esso.

Nella mente di Roosendaal si stava già facendo strada un progetto più grande: la distribuzione della nuova suite di creazione 3D al grande pubblico, installabile su diversi sistemi operativi, ma soprattutto distribuita gratuitamente (idea rivoluzionaria se si pensa che la maggior parte dei modellatori commerciali costava diverse migliaia di dollari).

Il grandissimo interesse dei partecipanti al Siggraph 1999, durante il quale venne presentata l'iniziativa, fece di Blender un successo, tanto che con nuovi finanziamenti ed un team più ampio che comprendeva ora 50 dipendenti, Blender aggiunse l'integrazione di un motore di gioco alla suite 3D.

Ma gli insuccessi di Blender Publisher (versione commerciale per i più affezionati) convinse gli investitori a fermare tutte le operazioni della NaN sul finire del 2001.

Nel Marzo 2002 Ton Roosendaal fondò l'organizzazione no-profit Blender Foundation con lo scopo di continuare lo sviluppo e la promozione di Blender come progetto Open Source. L'unico ostacolo erano i fondi per l'acquisto dei diritti sul codice di Blender e quelli di proprietà intellettuale dagli investitori della NaN. Con un gruppo di volontari entusiasti, tra i quali diversi ex dipendenti della NaN, venne lanciata la campagna per liberare Blender che, tra la sorpresa e il piacere generali, raggiunse l'obiettivo dei 100.000 Euro in sole sette settimane. Domenica 13 Ottobre 2002 Blender fu rilasciato al mondo sotto i termini della GNU General Public License (GPL).

Lo sviluppo oggi è portato avanti da un estesa community di volontari in tutto il mondo, guidata dal creatore originale, Ton Roosendaal.

Capitolo 2

Addentrarsi nel codice

In questo capitolo spiegheremo come ottenere e compilare il codice sorgente di Blender. Inizieremo a navigare al suo interno da un punto di vista strutturale (file e directory che lo compongono) e da un punto di vista funzionale (il modello al quale si ispira).

2.1 Il codice di Blender: come ottenerlo e come compilarlo

La prima cosa da fare è recuperare tutti i pacchetti per l'installazione di Blender. Oltre a subversion che ci permetterà di installare il codice, recuperiamo gli altri pacchetti necessari digitando le seguenti linee da terminale:

```
sudo apt-get update; sudo apt-get install subversion
 build-essential gettext \
 libxi-dev libsndfile1-dev \
 libpng12-dev libfftw3-dev \
 libopenexr-dev libopenjpeg-dev \
 libopenal-dev libalut-dev libvorbis-dev \
 libglu1-mesa-dev libsdl1.2-dev libfreetype6-dev \
 libtiff4-dev libavdevice-dev \
```

```
libavformat-dev libavutil-dev libavcodec-dev libjack-dev \
libswscale-dev libx264-dev libmp3lame-dev python3.2-dev \
libspnav-dev
```

Il secondo passo consiste nello scaricare i sorgenti di Blender dal repository SVN di blender.org.
(Li scarichiamo nella cartella blender-svn dopo averla creata)

```
mkdir blender-svn
cd blender-svn
svn co https://svn.blender.org/svnroot/bf-blender/trunk/blender
```

Per implementare con FFMPEG e Cycles abilitati sono necessarie altre librerie che possono essere scaricate dal repository di blender.org come librerie precompilate in questo modo (il mio sistema è linux 64bit):

```
svn co https://svn.blender.org/svnroot/
bf-blender/trunk/lib/linux64 lib/linux64
```

Ottenuto il codice, controlliamo che sia funzionante compilandolo. Può essere compilato con CMake oppure con Scons che si trova già integrato nel pacchetto scaricato. Per compilare con Scons:

```
cd ~/blender-svn/blender
python scons/scons.py
```

A compilazione eseguita (ci vuole una buona mezz'oretta!) dovremmo visualizzare le seguenti righe:

```
Install file: "/home/myname/blender-svn/blender/
build/linux2/bin/blender" as
 "/home/myname/blender-svn/blender/install/linux2/blender"
scons: done building targets.
*** Success ***
```

Prima di eseguire Blender creiamo un link simbolico al file binario nella cartella /blender-svn/blender per rendere più agevole rintracciarlo.

```
cd ~/blender-svn/blender
ln -s ../install/linux2/blender ./blender
```

ora mandiamo in esecuzione:

```
./blender
```

per ricompilare dopo una modifica digitiamo:

```
python scons/scons.py
```

Tutte le spiegazioni (anche per sistemi operativi differenti) su scaricamento dei sorgenti e compilazione possono essere trovate all'indirizzo:
http://wiki.blender.org/index.php/Dev:Doc/Building_Blender

2.2 Code Overview

Esploriamo ora il codice sorgente che abbiamo ottenuto con le istruzioni della sezione precedente. Se abbiamo anche già compilato i sorgenti, ora all'apertura della cartella blender-svn dovremmo avere le seguenti directory:

blender:
: Contiene i sorgenti, il nostro symbolic link all'eseguibile, il compilatore, alcuni documenti: la pagina del manuale ottenibile attraverso il comando man blender, i testi delle licenze e alcune linee guida per programmatori.

build:
: Contiene i file oggetto

install:
: Contiene gli eseguibili

lib:
: Contiene alcune librerie esterne

La directory blender/source/blender ci fa entrare nei sorgenti di Blender.

Il codice può essere suddiviso in tre grandi parti:

- strutture dati: oggetti, mesh, curve, ecc...
  sorgenti rintracciabili in: blender/source/blender/makesdna
- funzioni che operano sulle strutture dati
  sorgenti rintracciabili in: blender/source/blender/editors
  e in: blender/source/blender/modifiers
- windowmanager: gestione centralizzata degli eventi
  sorgenti rintracciabili in: blender/source/blender/windowmanager

La figura 2.1 spiega alcune delle directory base all'interno dei sorgenti.
Altre parti interessanti sono:

- blender/source/creator
  Al suo interno, nel file creator.c si trova il la funzione main.
- blender/source/gameengine
  Tutto ciò che riguarda il motore di gioco integrato
- /blender/source/blender/render
  Il motore di rendering

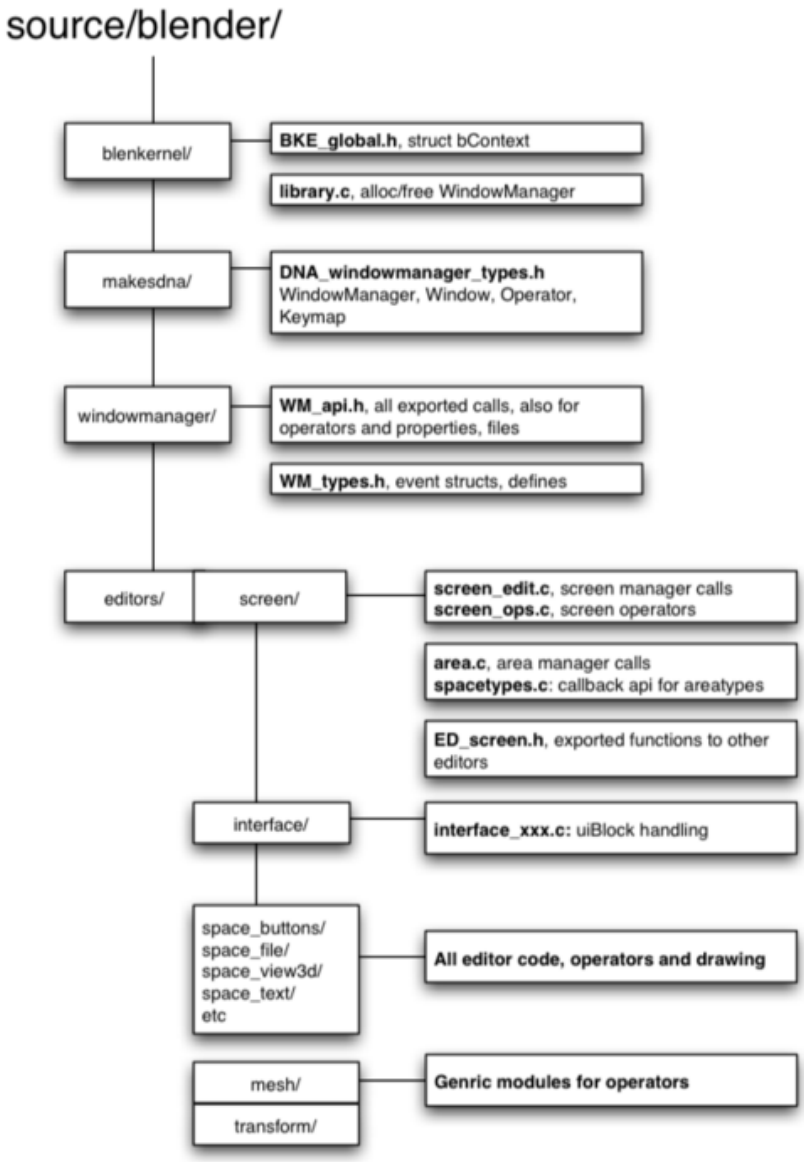

Figura 2.1: Code Layout

2.3 Il modello dei dati

Quando si iniziò la costruzione di Blender, si rese necessario trovare un modello che riunisse tutte le risposte ai bisogni dei membri della NeoGeo, sfruttando il fatto che gli utenti del nuovo sistema sarebbero stati gli stessi animatori e modellatori che avrebbero sviluppato il software. Le esigenze erano le seguenti:

- doveva essere possibile interagire con i dati attraverso un database di funzioni
- i dati dovevano essere visualizzati in modo flessibile attraverso la selezione della vista più consona all'operazione da effettuare

Vigeva anche una regola d'oro: L'editing doveva sempre essere fatto sui dati, mai sulla visualizzazione degli stessi!

Il netto distaccamento delle varie componenti del programma portò alla creazione di un'interfaccia utente per ogni vista. Non c'era una registrazione o una gestione degli eventi centralizzata, quindi era compito di ogni interfaccia gestire gli eventi attraverso una propria coda e un proprio handler. Nella versione 2.5 il modello Data-View-Edit (veniva chiamato così all'interno di NeoGeo) venne rivisto e integrato con il pattern MVC Model (Model View Controller) per dare vita al Blender 2.5 MVC Model che è utilizzato tutt'ora. Questo nuovo modello apportava alcune modifiche:

- la View (UI) ora è l'unico luogo in cui avviene la gestione degli eventi operata dal **Window Manager**
- il Controller è stato diviso in due parti: **gestione degli eventi** e **Operatori.**
  Gli operatori sono funzioni distaccate che vengono richiamate dall'handler. Questo è fondamentale per il riutilizzo in diverse parti del codice come undo, redo e chiamate da console python.

- Gli eventi sono separati dalle notifiche: i primi sono dovuti ad input dell'utente e devono essere gestiti nell'ordine in cui sono sollevati, mentre le notifiche vengono mandate dagli operatori.

L'idea che gli sviluppatori di Blender hanno del concetto di dato può essere riassunta dalla seguente frase: **Tutti i dati sono uguali**.
Questo non significa che una camera sia effettivamene uguale a una curva o ad una mesh, ma semplicemente che deve essere possibile, ad un certo livello, trattarli allo stesso modo.

Ogni blocco di dati specifico (Mesh, Camera, Oggetto, Scena, ecc..) inizia con una struttura ID che contiene, tra gli altri, un indentificativo univoco. Questa struttura permette di manipolare i dati in maniera uniforme senza avere particolari conoscenze sul tipo di dato rappresentato. I blocchi di dati possono essere copiati, modificati, e collegati ad altri dati come desiderato. Ad esempio uno stesso materiale può essere utilizzato per molte mesh, creando così una serie di link tra il primo e tutte le mesh che ne usufruiscono. Tutte le modifiche dell'utente devono essere apportate sui dati, mai sulla loro visualizzazione. Ma come sono organizzati questi dati?
Vengono salvati in un a struttura chiamata Main che non è altro che un blocco di liste (vedi appendice A). Qui risiedono tutti i dati indipendentemente da come l'utente li abbia collegati.
La struttura Main[1] si trova in source/blender/blenkernel/BKE_main.h
Ecco uno stralcio dai sorgenti:

```
typedef struct Main {
        ...
    ListBase scene;
    ListBase library;
    ListBase object;
    ListBase mesh;
    ListBase curve;
```

[1]per una introduzione a datablocks e Main vedere
http://wiki.blender.org/index.php/Dev:Source/Architecture/Overview

```
    ListBase mball;
    ListBase mat;
    ListBase tex;
    ListBase image;
    ListBase latt;
    ListBase lamp;
    ListBase camera;
    ListBase ipo; // XXX depreceated
    ListBase key;
    ListBase world;
    ListBase screen;
    ListBase script;
        ...
} Main;
```

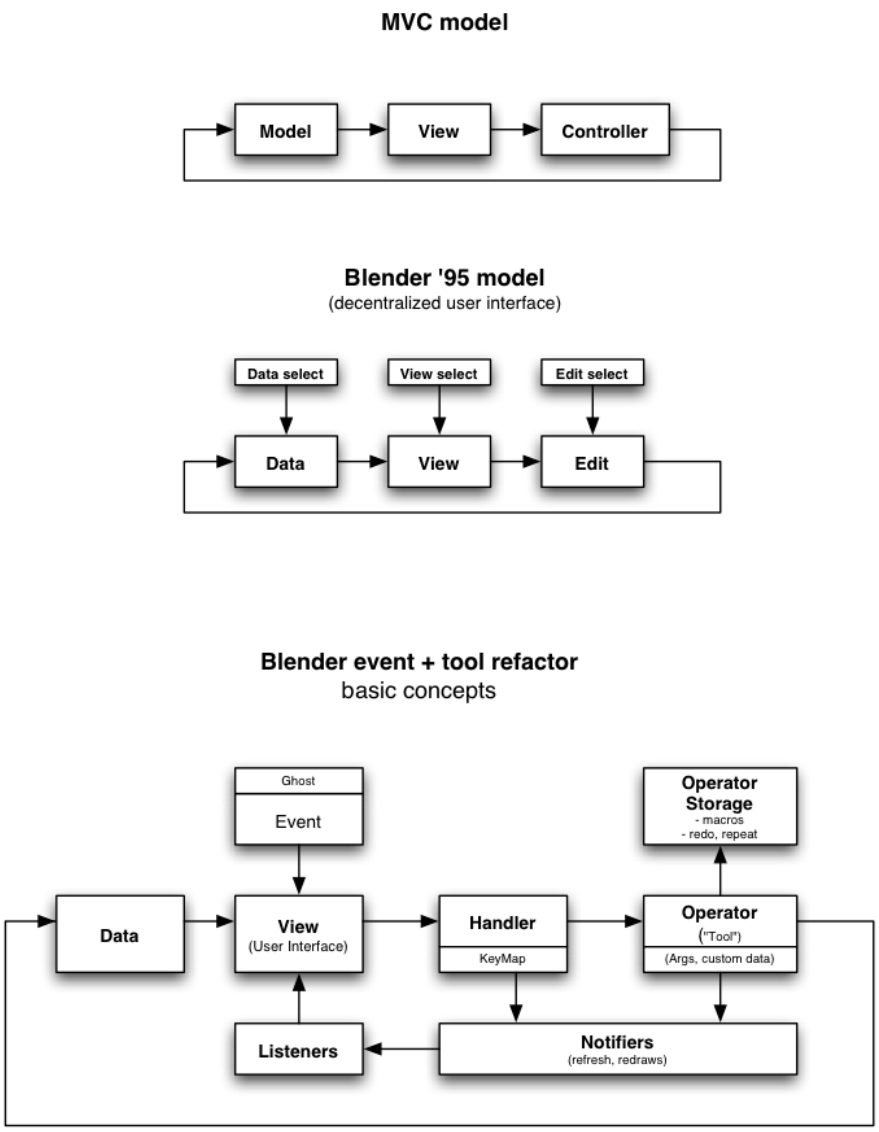

Figura 2.2: Modelli a confronto:
L'integrazione di Data-View-Edit ed MVC model ha portato al modello tutt'ora utilizzato

Capitolo 3

Oggetti

Tutto ciò che possiamo inserire in una scena (mesh, superfici, curve, telecamere, luci, ecc...) è per Blender una struttura dati **Object** poichè questi elementi possiedono tutti una posizione all'interno del sistema di assi e possono essere spostati, ridimensionati o ruotati.

Queste informazioni sono riassunte come vettori di float.
In particolare:

- *loc* individua la posizione dell'oggetto all'interno del sistema di riferimento
- *size* la grandezza dell'oggetto
- *rot* la rotazione

I seguenti attributi sono collegati tra loro:

short type
: un numero intero che individua il genere di oggetto che stiamo trattando, a scelta tra: OB_EMPTY, OB_MESH, OB_CURVE, OB_SURF, OB_LAMP, OB_CAMERA ed altri [1].

[1] definiti in DNA_object_types.h insieme alla struttura dati Object

void *data

un puntatore all'oggetto la cui struttura viene individuata in base al valore dell'attributo type. Esistono strutture dati specifiche per tutti gli oggetti all'interno di una scena, quelle che tratteremo ora sono: struct Mesh (definita in DNA_mesh_types.h) struct Curve (definita in DNA_curve_types.h)

Ammettendo di avere accesso ad un puntatore ad un oggetto (chiamato obj nell'esempio) possiamo recuperare la struttura puntata dal campo data in questo modo:

Nel caso di mesh:

```
(Mesh *)obj-> data;
```

Nel caso di curve:

```
(Curve *)obj-> data;
```

per sapere in quale caso ci troviamo possiamo testare l'attributo type.

Da notare che sia per curve che per superfici la struttura dati associata è una struct Curve.

Altri attributi importanti:

ID id

La struttura dati ID viene utilizzata per gestire l'inserimento dei data blocks nella Main structure (già descritta nel Capitolo 1). Qui troviamo un puntatore al blocco precedente e al successivo nella lista, un identificatore univoco e altre informazioni.

ListBase modifiers

Stack di modificatori che agiscono sull'oggetto, verranno spiegati in seguito.

struct BoundBox *bb;

definito in DNA_object_types.h, definisce un parallelepipedo nel quale è inscritta la mesh o la curva. La struttura BoundBox contiene una matrice 8x3 (le coordinate dei vertici), nonchè alcuni flag.

La struttura dati Object, insieme ad altre più specifiche, funge da interfaccia per le modifiche effettuate in **Object Mode**.

Come accennato nell'introduzione, Object Mode è la vista di Blender in cui possiamo comportarci come registi aggiungendo oggetti, spostando luci e camere e guardare la scena da diverse angolazioni.

Ci serviremo della struttura Object per gli spostamenti, le rotazioni e il ridimensionamento degli oggetti, ma non possiamo aggiungere ad una scena un oggetto generico, dobbiamo sceglierne la tipologia; per questo nei prossimi capitoli parleremo di mesh, curve e superfici.

Capitolo 4

Mesh

Gli oggetti solidi vengono comunemente rappresentati nei sistemi 3d attraverso il disegno del loro contorno che si assume essere una superficie 2-dimensionale. Per semplificarne la descrizione, si usa una approssimazione poligonale della superficie che viene chiamata **Mesh 3D poligonale**.
Una Mesh 3D poligonale viene descritta da un insieme di informazioni:

- Facce poligonari
- Lati
- Verici

I vertici sono punti 3d, vengono rappresentati quindi attraverso le coordinate che li identificano. Ogni lato congiunge due vertici, mentre ogni faccia poligonale è un pligono chiuso (solitamente un triangolo o un quadrilatero) individuato da lati e vertici.

Dato che abbiamo assunto che il contorno dell'oggetto sia una superficie bidimensionale, si richiede che anche la mesh che lo approssima sia two-manifold. Vengono quindi imposte le seguenti condizioni:

- Un lato deve appartenere al massimo a due facce

- Se due o più facce hanno un vertice in comune, queste devono formare un ventaglio

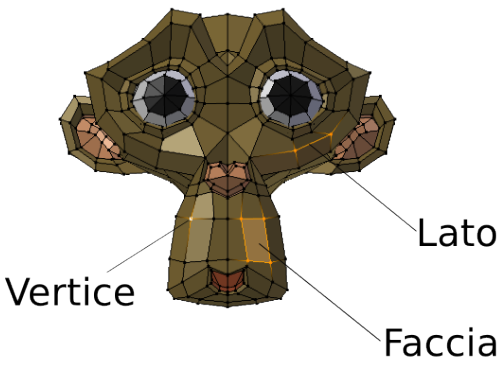

Figura 4.1: Lati, vertici e facce di una mesh

4.1 Le Mesh in Blender 2.62

In realtà le assunzioni che valgono per le mesh in generale, non sono sempre applicate in Blender. Per capirne l'organizzazione ed il distaccamento dal modello originale, dobbiamo parlare di svariate strutture dati che le riguardano:

- Mesh (già introdotta parlando degli oggetti)
- BMEditMesh e BMesh
- DerivedMesh (solo qualche cenno)
- e alcune altre strutture ausiliarie di cui fanno uso quelle sopra citate.

4.1.1 Struct Mesh

Definita in DNA_mesh_types.h

La struttura dati Mesh contiene le informazioni necessarie per il salvataggio della geometria su disco.

Campi che definiscono la geometria[1]:

struct MVert *mvert
: array di strutture MVert. Ogni struttura contiene le coordinate del vertice e le coordinate della normale.

struct MEdge *medge
: array di strutture MEdge. Ogni struttura contiene due interi (v1 e v2) che identificano gli indici degli estremi del'edge all'interno dell'array di MVert associato alla mesh.

struct MPoly *mpoly
: array di strutture MPoly. É stato introdotto solo di recente per permettere la creazione di facce con più di quattro vertici. Ogni struttura contiene il numero di loop di cui è composta la faccia e l'indice all'interno dell'array di loop.

struct MLoop *mloop
: array di strutture MLoop. Ogni struttura contiene due interi (v ed e) che permettono di associare un verso all'edge.

int totvert, totedge, totface, totpoly, totloop
: numero di vertici, edge, poligoni, loop nella mesh.

In realtà tutti i campi citati sono una ripetizione, mantenuta per semplicità di utilizzo, infatti le stesse informazioni sono salvate nei seguenti attributi:

[1] le strutture MLoop, MPoly, MEdge, MVert sono definite in DNA_meshdata_types.h

struct CustomData vdata, edata, fdata, pdata, ldata.

É necessario tenere aggiornati i puntatori in modo che i valori puntati siano quelli realmente utilizzati. Questo viene fatto attraverso la funzione mesh_update_customdata_pointers. La API CustomData provvede a un modo per manipolare i dati nelle mesh ed è definita cosı: Ogni unità di CustomData è chiamata livello. Ogni livello è identificato da un tipo, da una dimensione e da un set di funizoni ad esso associate per manipolarlo. Ogni livello possiede un campo data che punta ad un array di strutture (es. un array di strutture MVert, MEdge...)

struct BMEditMesh *edit_btmesh

puntatore alla struttura dati BMEditMesh che vedremo in seguito.

float loc[3], float size[3], float rot[3]

relativi alla textrure.

Nella figura 3.2 possiamo vedere le primitive Mesh che sono messe a disposizione da Blender. Tra le altre appare anche Suzanne, una scimmietta, regalo da parte della vecchia NaN alla Blender Community. È una sorta di easter egg, ma viene usata per le prove su oggetti complessi, come i più famosi Utah Tea Pot e lo Stanford Bunny.
Da notare anche la base del cono e la cima del cilindro rappresentati da un unico poligono. Questo è possibile poichè viene data la possibilità all'utente di lavorare con il numero di lati e vertici desiderato e di connetterli a piacimento. La tassellazione in faccette triangolari viene calcolata da Blender e mantenuta all'interno di un altro array chiamato Face. Se l'utente decide di lavorare con faccette triangolari, quindi, i due array sono lo stesso array, ma i valori sono comunque duplicati.

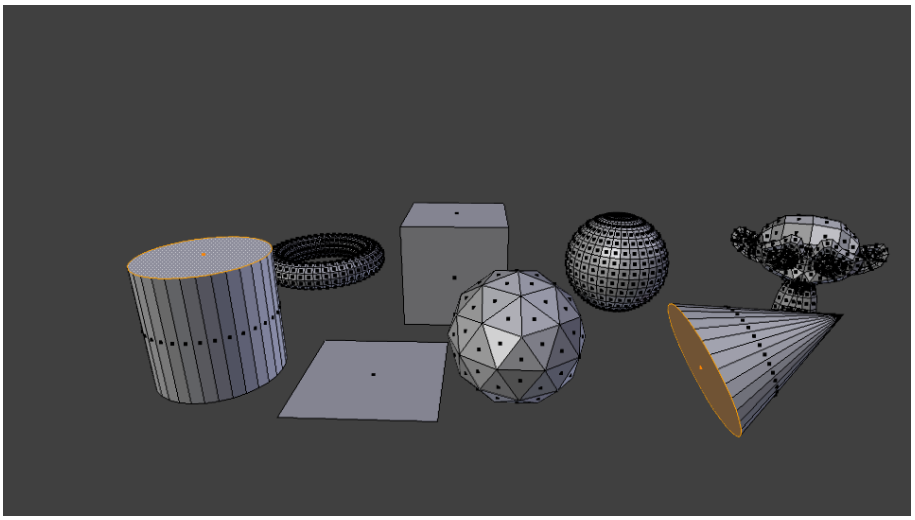

Figura 4.2: Primitive mesh

4.1.2 Editare le Mesh

Blender da qualche anno mette a disposizione oltre alle Mesh, altre strutture per editarle in modo più agevole: BMesh e BMeditMesh[2]. Lo scopo è quello di creare tre livelli di astrazione:

Low-level Mesh Editing API:
: funzioni per attraversare BMesh cycles (spiegati in seguito) e primitive per l'editing locale (Euler operators nella documentazione).

Mid-level Mesh Editing API:
: BMesh Operators (es. mirror, bevel, similar faces) e iteratori. Gli operatori possono essere annidati.

Top-level Mesh Editing API:
: Mesh Tools azionabili direttamente dall'utente tramite bottoni, shortcuts o script python.

[2]Riferimenti nella documentazione:
http://wiki.blender.org/index.php/Dev:2.6/Source/Modeling/BMesh/Design

Le funzioni che appartengono a Low-level e a Mid-level API, ricevono in input una struttura BMesh, mentre quelle di più alto livello una BMEditMesh.

Struct BMesh: definita in bmesh_class.h

Questa struttura contiene vertici, edge, loop e poligoni sottoforma di CustomData e informazioni sull'allocazione della memoria.

BMEditMesh: definita in BKE_tessmesh.h

Quando entriamo in EditMode la Mesh viene convertita, all'occorrenza, in una struttura BMEditMesh. La differenza più significativa tra le due sta nel metodo con cui vengono memorizzati i dati: mentre nella struttura Mesh abbiamo array statici per vertici, lati e facce, nella BMEditMesh troviamo liste circolari, molto più comode se si pensa di agire su una geometria modificando il numero dei vertici eliminando ed aggiungendo elementi. BMVert, BMEdge, BMFace, BMLoop nelle BMEditMesh sono gli analoghi di MVert, MEdge, MFace, MLoop, nelle Mesh.

Altra differenza significativa è l'aggiunta di alcuni elementi, chiamati Cycles, che permettono di navigare tra vertici, edge e poligoni definendo per ogni edge, vertice o poligono un precedente ed un successivo secondo un ordine definito dal cycle stesso. Ne esistono di tre tipi:

Disk Cycles (un ciclo di edges intorno a un vertice)
: vengono memorizzati all'interno della struttura dati BMedge. Nella struttura si tiene traccia dei due vertici dell'edge che stiamo analizzando (BMVert *v1, *v2) e di un Disk Cycle per ognuno dei due vertici (BMDiskLink v1_disk_link, v2_disk_link;)

Loop Cycles (un ciclo di face e edge intorno a un poligono)
: se prendiamo come oggetto della nostra analisi una faccia della nostra geometria (f1), allora possiamo pensare di volere scoprire quali sono i vertici e le edge che compongono quella faccia e quali altre facce confinano con f1.

Radial Cycles (un ciclo di facce intorno a un edge)

Se consideriamo come centro di analisi l'edge, allora possiamo volere conoscere quali sono i poligoni che condividono l'edge in questione. I radial Cycles sono salvati all'interno della struttura BMLoop.

Queste ed altre strutture sono definite in bmesh_class.h

Per accedere alle strutture dati citate è necessario utilizzare le funzioni il cui codice si trova in editmesh_utils.c.
In particolare:
Le strutture BMVert, BMEdge e BMFace vengono generate su richiesta attraverso un'apposita funzione (EDBM_index_arrays_init) che permette anche di inizializzare uno, più o tutti gli array settando un apposito flag. Le strutture possono poi essere lette attraverso le funzioni EDBM_vert_at_index, EDBM_edge_at_index ed EDBM_face_at_index. É possibile ottenere gli indici degli elementi all'interno dell'array attraverso la funzione BM_elem_index_get.

Sia le Mesh che le BMEditMesh sono convertibili in DerivedMesh per permettere l'applicazione dei modificatori.

DerivedMesh: definita in BKE_DerivedMesh.h

Questa struttura è stata creata per essere utilizzata in diversi casi, ad esempio Rendering, export system, modifier stack (uno stack di funzioni applicate sull'oggetto in modo che la mesh ottenuta come ouput della prima funzione sia l'input per la seconda funzione e così via fino ad esaurimento dello stack).

É composta da alcuni campi ad uso privato e da una serie di puntatori a funzioni. Poichè non parleremo di modificatori questa struttura ci interessa relativamente.

Capitolo 5

Curve e superfici

In Computer Graphics, non si utilizzano solo strutture mesh, o meglio, per generare mesh di alta qualità ci si avvale a volte di curve e superfici che, espresse in termini matematici, sono comode perchè sono definite da pochi parametri e le normali possono essere definite correttamente in ogni punto. Come si descrivono curve e superfici?

- Si definisce uno spazio parametrico a una dimensione per curve e a due per superfici
- Si definisce un mapping tra lo spazio parametrico e punti 3D: una funzione che prende valori parametrici e restituisce punti 3D.

Quello che otteniamo è una curva o una superficie parametrica.

Le coordinate x, y e z di un punto 3D sulla curva sono determinate ciascuna da un'equazione che coinvolge:

- il **parametro t**
- alcuni punti specificati dall'utente: i **punti di controllo**

Le singole componenti della curva sono espresse mediante **funzioni base**.

La scelta delle funzioni base determina l'influenza dei punti di controllo sulla forma della curva, dando origine a diverse curve al variare della scelta delle funzioni base. Una di queste basi è la base polinomiale di Berstein.

Curve di Bezièr

Le curve di Bezièr sono descritte da d+1 punti di controllo p_i (che definiranno un poligono di controllo).

La curva di scrive come:

$$F(t) = \sum_{i=0}^{d} p_i B_i^d(t)$$

$$dove : B_i^d(t) = \binom{d}{i} t^i (1-t)^{d-i}$$

B_i^d sono i polinomi base di Bernstein di grado d (detti anche Blending functions).

Le curve di Bezièr godono di alcune proprietà:

- iniziano dal primo punto di controllo e finiscono nell'ultimo
- la tangente alla curva nel primo punto ha la stessa direzione del primo segmento del poligono di controllo
- la curva giace interamente dentro il guscio convesso definito dai punti di controllo.

Per creare curve dalle forme più complesse, si possono unire più segmenti ciascuno dei quali rappresentabile con una di queste curve.

Curve B-Spline

Le curve B-spline sono una rappresentazione matematica compatta per segmenti di curve polinomiali definiti su una sequenza di intervalli parametrici, detta partizione nodale. In base al grado della curva possiamo ottenere B-spline lineari, quadratiche, cubiche e così via, mentre la partizione nodale può essere uniforme o non uniforme. Le curve B-spline automaticamente garantiscono un certo ordine di continuità: solitamente uno in meno del grado del segmento di curva (B-spline lineari hanno continuità C0, le quadratiche

C1, le cubiche C2, ecc...).

$$F(t) = \sum p_i B_{i,d}(t)$$

Curve NURBS

Una curva NURBS 3D può essere vista come una proiezione di una curva B-Spline 4D in uno spazio 3D:

$$[x(t), y(t), z(t), w(t)] \rightarrow [\frac{x(t)}{w(t)}, \frac{y(t)}{w(t)}, \frac{z(t)}{w(t)}]$$

x(t), y(t), z(t) e w(t) sono funzioni B-splines non uniformi.
w(t) è il **peso**: consente di attrarre localmente, in corrispondenza di un Punto di Controllo, una porzione di curva.
Le curve NURBS hanno alcuni vantaggi:

- sono invarianti per proiezione prospettica, così che possono essere valutate nello spazio del piano di proiezione.
- Possono rappresentare esattamente sezioni coniche: parabola, ellisse, circonferenza, iperbole. Le curve B-spline polinomiali possono solo approssimare le coniche.

Superfici

Le superfici parametriche 3D seguono le stesse regole delle curve, ma la parametrizzazione viene eseguita mediante due parametri: s e t. Le superfici parametriche hanno una topologia rettangolare, questo significa che per la loro definizione si usa una griglia rettangolare di punti di controllo. hanno la seguente forma:

$$F(s,t) = \sum_{i=0}^{1} \sum_{j=0}^{1} P_{i,j} B_{i,s}(s) B_{j,t}(t)$$

per certe funzioni $B_{i,s}$ e $B_{j,t}$

5.1 Curve e superfici in Blender 2.62

Blender mette a disposizione solo curve di Bezièr e curve NURBS. Per le superfici invece, esistono solamente NURBS all'interno del programma. Andiamo a vedere più da vicino le strutture dati che le riguardano.

Struct Curve

La struttura dati Curve contiene le informazioni necessarie per il salvataggio della geometria su disco e per i cambiamenti effettuati in ObjectMode.

I campi che possono interessarci sono i seguenti:

short type
: contiene un intero che permette di distinguere curve da superfici (OB_CURVE vs OB_SURF)

float loc[3], float size[3], float rot[3]
: relative alla textrure.

EditNurb *editnurb
: Strutture che accolgono le Nurbs in EditMode.

ListBase[1] nurb
: Lista di Nurb structures necessarie per la visualizzazione in objectMode.

Struct Nurb

Stranamente gli sviluppatori hanno deciso di chiamare la struttura dati Nurb anzi che Nurbs. Gli stessi commenti all'interno del codice indicano perplessità e non si conosce il motivo della scelta.

[1] Per le liste vedi appendice A le liste di Blender

All'inteno di questa struttura troviamo i campi che ci definiscono una curva o una superficie come i punti di controllo o le maniglie.
Analizziamo ora i campi in comune alle Bezièr curve e alle NURBS:

short type;
: permette di distinguere tra i seguenti tipi: CU_POLY, CU_BEZIER, CU_BSPLINE, CU_CARDINAL, CU_NURBS.

short hide
: Individua se la Nurbs in questione è visibile.

short pntsu, pntsv
: numero di punti nella direzione U o V

short resolu, resolv
: risoluzione nella direzione U o V (per tassellazione)

short orderu, orderv
: ordine nella direzione U o V

short flagu, flagv
: A scelta tra: CU_NURB_CYCLIC, CU_NURB_ENDPOINT, CU_NURB_BEZIER. Vedremo in seguito come Blender utilizza questi flag.

A questo punto è d'obbligo fare una distinzione tra le curve di Bezier e Nurbs perchè gli altri campi della struttura sono utilizzati in modi differenti.

Curve di Bezièr

Se ci troviamo di fronte a curve di Bezièr dovremo memorizzare le triple di Bezièr per le quali è stata definita una struttura a parte, invece non saranno utilizzati i seguenti campi: BPoint *bp; float *knotsu, *knotsv.

BezTriple *bezt;

Array di strutture BezTriple. Questa struttura si occupa di tenere traccia delle maniglie e dei punti di controllo. Al suo interno troviamo:

float vec[3][3];

si occupa proprio di questo. Riporto il commento esplicativo nel file DNA_curve_types.h.

```
/* vec in BezTriple looks like this:
 * - vec[0][0]=x location of handle 1
 * - vec[0][1]=y location of handle 1
 * - vec[0][2]=z location of handle 1 (not used for FCurve Points(2d))
 * - vec[1][0]=x location of control point
 * - vec[1][1]=y location of control point
 * - vec[1][2]=z location of control point
 * - vec[2][0]=x location of handle 2
 * - vec[2][1]=y location of handle 2
 * - vec[2][2]=z location of handle 2 (not used for FCurve Points(2d))
 */
```

char h1, h2;

tipi di maniglie per le due handle.

char f1, f2, f3

usati per il selection status.

char hide

usato per indicare se la BezTriple è nascosta.

Curve NURBS

Se usiamo le curve Nurbs, il campo BezTriple *bezt sarà settato a NULL. Mentre saranno utilizzati i seguenti:

float *knotsu, *knotsv

Array che tengono gli knots nelle direzioni U e V.

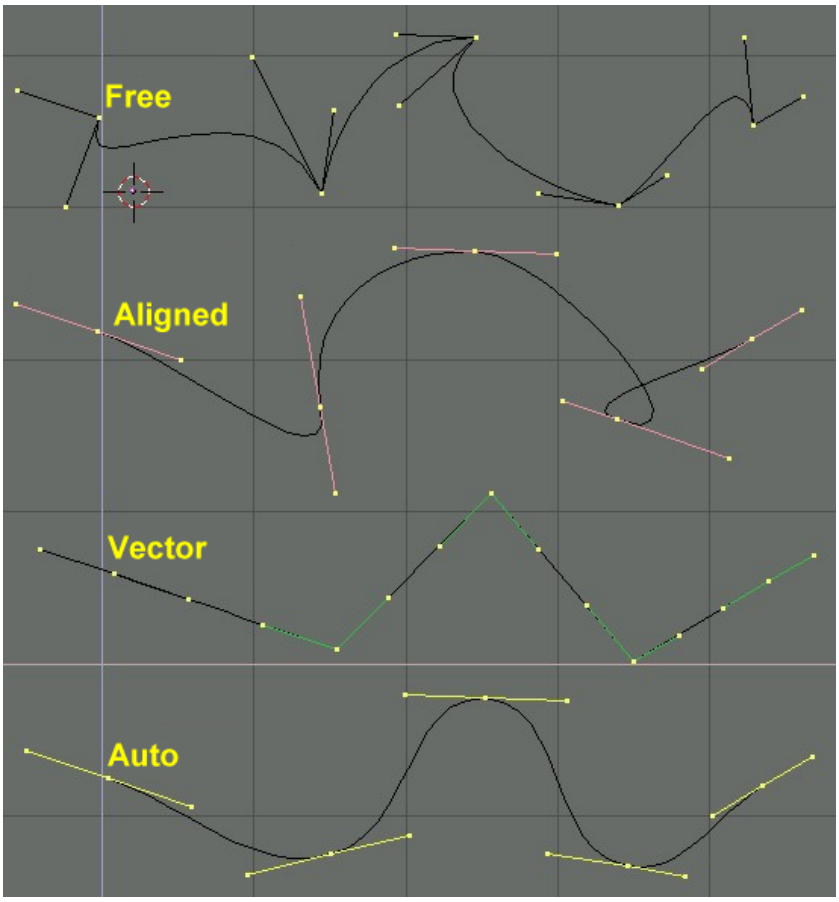

Figura 5.1: Curve di Bezièr con diversi tipi di maniglie (attributo h1 ed h2)

BPoint *bp
: Array di strutture Bpoint.

La struttura Bpoint si compone dei seguenti campi:

float vec[4]
: tiene le coordinate del punto di controllo: x, y, z e il peso associato.

short f1
: selection status

short hide
: usato per indicare se il punto di controllo è nascosto.

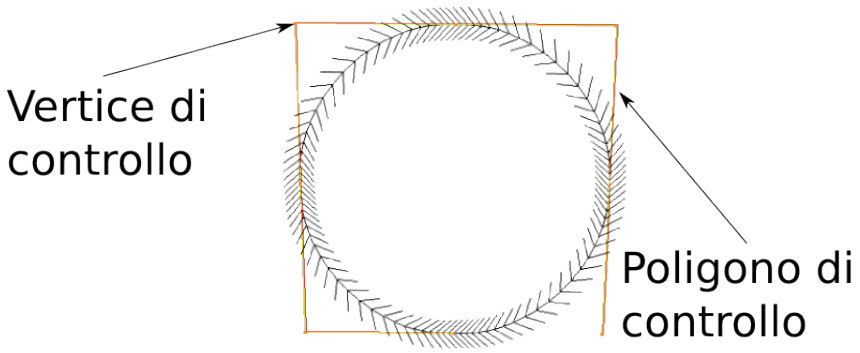

Figura 5.2: Cerchio NURBS

5.2 Disegnare curve e superfici

il file drawobject.c contiene numerose funzioni per disegnare tutte le componenti di una scena: dagli assi cartesiani, alle immagini di fondo, agli oggetti veri e propri.
La funzione
void draw_object(Scene *scene,ARegion *ar,View3D *v3d,Base *base,int flag)
si occupa del disegno degli oggetti in scena. Al suo interno vengono chiamate differenti funzioni a seconda dell'attributo type dell'oggetto da disegnare. Noi seguiremo ora alcune funzioni per capire come avvenga il disegno di una curva NURBS. Innanzi tutto si distingue tra:

- Curve in editMode:
  viene chiamata drawnurb(scene, v3d, rv3d, base, nurbs− >first, dt)

- Curve in objectMode:
  viene chiamata drawDispList(scene, v3d, rv3d, base, nurbs− >first, dt)

Cercheremo di seguire la prima chiamata, poichè al suo interno, come vedremo, effettuerà una chiamata alla drawDispList.

drawnurb

Poichè ci troviamo in editMode, dovranno essere disegnate anche le maniglie per le curve di Bezier, e il poligono di controllo per le curve Nurbs, quindi si effettuano i seguenti passi:

1. viene chiamata la funzione DrawDispList per il disegno vero e proprio della curva

2. vengono disegnate le maniglie se stiamo trattando curve di Bezièr

3. vengono effettuate due chiamate draw_editnurb:
   una con ultimo argomento uguale a zero (per disegnare i lati del poligono di controllo non selezionati), l'altra con ultimo argomento uguale a uno, per disegnare i lati del poligono selezionati. Queste due chiamate non hanno effetto se si sta disegnando una curva di Bezièr. Si tratta semplicemente di prendere le coordinate dei vertici di controllo e di tracciare una linea tra un punto di controllo ed il successivo; il colore viene settato in base ai punti selezionati.

   ```
   glBegin(GL_LINE_STRIP);
       glVertex3fv(bp->vec);
       glVertex3fv(bp1->vec);
   glEnd();
   ```

4. Vengono disegnate le linee per indicare le normali (danno anche il senso di lettura della curva)

drawDispList

Questa è la funzione che disegna la curva vera e propria.
dt rappresenta la visualizzazione: a scelta tra OB_BOUNDBOX, OB_WIRE, OB_SOLID, OB_TEXTURE, OB_MATERIAL, OB_RENDER. Nel caso in cui sia selezionato OB_SOLID verrà chiamata la funzione drawDispListsolid per le superfici, ma visto che noi ci stiamo occupando delle curve, che restano identiche nonostante gli venga applicato un materiale, verrà chiamata sia nel caso di OB_SOLID che nel caso di OB_WIRE la stessa funzione: static int drawDispListwire(ListBase *dlbase).
A questo punto viene disegnata la curva, ma prima dobbiamo introdurre una struttura che viene utilizzata per il disegno: la struttura DispList.

All'interno della struttura Object esiste un campo ListBase disp utilizzato solo dalle curve e dalle superfici. Ogni entry nella lista è legata ad una curva dell'oggetto.

I campi della struttura DispList che ci interessano sono:

DispList *next, *prev
: per navigare all'interno della lista

short type
: a scelta tra DL_SEGM e DL_POLY.

int parts, nr
: indicano in quanti parti viene divisa la curva (per le curve parts è sempre 1, mentre nr cambia a seconda della curva).

float *verts, *nors
: sono un array di vertici e normali che vengono utilizzati per il disegno.

Il disegno viene fatto testando l'attributo type per ogni curva da disegnare, e si attingono dall'array verts le coordinate necessarie in questo modo:

```
...
```

```
data = dl->verts;
    ...
switch (dl->type) {
    case DL_SEGM:

        glVertexPointer(3, GL_FLOAT, 0, data);

        for (parts = 0; parts < dl->parts; parts++)
            glDrawArrays(GL_LINE_STRIP, parts * dl->nr, dl->nr);

    break;
    case DL_POLY:

        glVertexPointer(3, GL_FLOAT, 0, data);

        for (parts = 0; parts < dl->parts; parts++)
            glDrawArrays(GL_LINE_LOOP, parts * dl->nr, dl->nr);

    break;
    ...
```

dl è un puntatore alla struttura DispList della curva che vogliamo disegnare. Nel primo caso la curva è aperta, nel secondo caso è chiusa.

Ma dove e come viene riempita la struttura DispList?
La struttura viene riempita con la funione curve_to_displist in questo modo:

1. nr viene calcolato tenendo conto di risoluzione e numero di punti di controllo
   es. per un cerchio NURBS (di default):
   $nr = resolu * pntsu$ (12 * 8 = 96)
   parts=1

2. viene testato l'attributo flagu della curva: se è CU_NURB_CYCLIC dl $->$ type verrà settato a DL_POLY (curva chiusa) altrimenti verrà settato a DL_SEGM

3. viene allocato un vettore per contenere i vertici grande:
$(resolu * pntsu) * 3 * sizeof(float)$

4. La curva viene valutata in nr punti (non solo negli knot) questi vengono convertiti in coordinate 3d e salvati all'interno del vettore vert.

Capitolo 6

Scena e Contesto

Nel primo capitolo abbiamo visto che all'interno della struttura Main ci sono una serie di liste, tra le quali figura anche la lista delle **scene**, ognuna è definita da una struct Scene.

A partire da questa struttura si riescono ad ottenere gli oggetti che compongono la scena (in una lista), la struttura camera che rappresenta l'osservatore in fase di rendering, l'oggetto selezionato per ultimo e l'oggetto editabile (se ci troviamo in Edit Mode). Con queste informazioni e seguendo i link a materiali e texture contenuti in ogni oggetto, otteniamo uno Scene graph.

```
typedef struct Scene {
    ID id;
    struct Object *camera;
    struct World *world;
    struct Scene *set;
    ListBase base;
    struct Base *basact; /* active base */
    struct Object *obedit; /* name replaces old G.obedit */
    float cursor[3]; /* 3d cursor location */
    float twcent[3]; /* center for transform widget */
    float twmin[3], twmax[3]; /* boundbox of selection
```

```
                                    for transform widget */
        ...
} Scene;
```

Il **contesto** è una struttura aggiuntiva che tiene informazioni riguardanti il window manager (ed esempio i menu a cui si può accedere) e informazioni riguardanti i dati su cui intervenire (ad esempio la scena). Le informazioni al suo interno sono prese direttamente dalla scena corrente.
Un set di funzioni permette di interrogare il contesto:

Scene *scene= CTX_data_scene(C);
per ottenere un puntatore alla scena corrente

Object *obedit= CTX_data_edit_object(C);
per ottenere un puntatore all'oggetto in Edit Mode (restituisce NULL se non siamo in Edit Mode)

Object *obactive=CTX_data_active_object(C);
permette di ottenere un puntatore all'oggetto attivo.

Interrogare il contesto è molto importante per gli operatori che, come vedremo tra poco, vengono eseguiti sempre sull'oggetto attivo o sull'oggetto portato in Edit Mode; deve quindi essere possibile accedere ad essi.

Capitolo 7

Operatori

Gli operatori sono funzioni che interagiscono con i dati, permettendo di dare all'oggetto ai quali sono applicati la forma e gli effetti desiderati. Definire un operatore significa registrarlo presso il Window Manager per fare in modo che quando facciamo partire il programma, il nostro operatore possa essere chiamato dall'utente in relazione a qualche evento.
Tutti i tipi di operatori seguono lo stesso template:

```
static void PREFIX_OT_do_some_operation(wmOperatorType *ot)
{
    /* identifiers */
    ot->name= "human readable name";
    ot->idname= "PREFIX_OT_do_some_operation";

    /* api callbacks */
    ot->invoke= invoke_function;
    ot->exec= exec_function;
    ot->modal= modal_function;
    ot->poll= poll_funcion;

    /* flags */
    ot->flag= OPTYPE_REGISTER|OPTYPE_UNDO ;
```

```
    /* properties */
}
```

Il prefisso indica il campo d'azione dell'operatore (abbiamo MESH, OBJECT, CURVE, VIEW3D, ecc.)
_OT_ sta per operator type e viene seguito da un nome che indica cosa la funzione si propone di fare.

Name e description sono utilizzati nell'interfaccia utente, quindi devono essere stringhe leggibili. Idname è un identificatore univoco per l'operatore, per convenzione si usa lo stesso nome della funzione. Le callback functions definiscono il funzionamento dell'operatore:

poll
: viene chiamata per testare se l'operatore può essere eseguito. Se l'operatore deve agire esclusivamente in Edit Mode è facile che vengano fatti controlli del tipo CTX_data_edit_object(C)!=NULL

exec
: esegue l'operatore

invoke
: viene eseguita quando l'operatore viene chiamato dall'utente. Se non è stata definita viene utilizzata la funzione exec

modal
: viene chiamata ogni volta che accade un evento. É questa funzione che deciderà poi se gestirlo oppure no

I flag danno informazioni al window manager: nell'esempio è stato settato il flag OPTYPE_REGISTER che permette all'operatore di essere registrato nell'history stack, e OPTYPE_UNDO che indica che deve essere fatta una undo push dopo che l'operatore ha terminato.

Gli operatori possono poi definire un numero di proprietà. Queste possono essere settate dall'utente e usate dall'operatore per modificare il suo comportamento.

Ora che abbiamo un background generale su Blender possiamo entrare un pò di più nello specifico andando a vedere come lavorano alcuni operatori. Ricordo che questa sorta di piccola guida ha come scopo il comprendere se sia possibile modificare in qualche modo Blender per ottenere un Edit Multiplo delle geometrie e per fare questo dobbiamo conoscere come funziona l'entrata attuale in Edit Mode. Tratteremo poi un operatore interessante (Join) che a prima vista sembra essere molto vicino allo scopo, ma vedremo che non sarà esattamente così.

7.1 Entrare in Edit Mode

L'operatore preposto all'entrata in Edit Mode si chiama OBJECT_OT_editmode_toggle.

```
void OBJECT_OT_editmode_toggle(wmOperatorType *ot)
{

/* identifiers */
ot->name = "Toggle Editmode";
ot->description = "Toggle object's editmode";
ot->idname = "OBJECT_OT_editmode_toggle";

/* api callbacks */
ot->exec = editmode_toggle_exec;

ot->poll = editmode_toggle_poll;

/* flags */
```

```
ot->flag = OPTYPE_REGISTER|OPTYPE_UNDO;
}
```

La funzione **editmode_toggle_poll** è molto semplice: testa che l'oggetto selezionato sia una geometria, non una luce o una camera.

La funzione **editmode_toggle_exec** è un pò più complessa, viene descritta dai seguenti passi:

1. Prende dal contesto la scena corrente e l'oggetto attivo (ob)
2. (Fa alcuni controlli per vedere che non si trovi su un layer diverso da quello su cui stiamo operando)
3. nel campo mode di ob salva il valore OB_MODE_EDIT
4. si comporta diversamente a seconda del tipo di oggetto (testando ob− >type):

   per MESH:

   1. mette ob all'interno di obedit nella scena
   2. crea una EditBmesh
   3. manda una notifica

   per CURVE E SUPERFICI:

   1. mette ob all'interno di obedit nella scena
   2. crea una editNurb
   3. manda una notifica

5. DAG_id_tag_update(ob− >id, OB_RECALC_DATA);

7.2 Join

L'operatore Join eseguibile solamente in Object Mode, rende apparentemente editabili contemporaneamente due o più oggetti, ma non è effettivamente così. Vediamo come lavora.

Selezioniamo una serie di oggetti; ovviamente l'ultimo selezionato apparirà di un colore arancione perchè è l'oggetto attivo, mentre gli altri saranno contornati da un colore più scuro. Ora possiamo applicare il Join.

Visivamente notiamo che, se stiamo lavorando soltanto con oggetti mesh (o soltanto con curve o solo con superfici), tutti sono ora contornati di arancione. Quello che ci aspettiamo è che tutti questi oggetti siano diventati oggetti attivi e, poichè come abbiamo visto nella sezione precedente, gli oggetti attivi sono trasportabili in Edit Mode, possiamo pensare di premere tab e di editare così contemporaneamente tutte le geometrie evidenziate. Effettivamente questo è possibile, ma facendo qualche prova e guardando il codice, ci accorgiamo che il Join è semplicemente un trucco.

```
void OBJECT_OT_join(wmOperatorType *ot)
{
    /* identifiers */
    ot->name = "Join";
    ot->description = "Join selected objects into active object";
    ot->idname = "OBJECT_OT_join";

    /* api callbacks */
    ot->exec = join_exec;
    ot->poll = join_poll;

    /* flags */
    ot->flag = OPTYPE_REGISTER|OPTYPE_UNDO;
}
```

Come si vede nella descrizione all'interno dell'operatore, il Join non rende più oggetti attivi contemporaneamente, ma fonde tutti quelli selezionati nell'oggetto attivo, eliminando le altre strutture oggetto.

Questo per il nostro scopo potrebbe anche andare bene, ma come si comporta se selezioniamo oggetti di tipo diverso? Riesce a fonderli ugualmente in una unica struttura Object? Andiamo a vedere nel codice!

```
static int join_exec(bContext *C, wmOperator *op)
{
    Scene *scene= CTX_data_scene(C);
    Object *ob= CTX_data_active_object(C);

    if (scene->obedit) {
        BKE_report(op->reports, RPT_ERROR,
         "This data does not support joining in editmode");
        return OPERATOR_CANCELLED;
    }
    else if (object_data_is_libdata(ob)) {
        BKE_report(op->reports, RPT_ERROR,
         "Can't edit external libdata");
        return OPERATOR_CANCELLED;
    }

    if (ob->type == OB_MESH)
        return join_mesh_exec(C, op);
    else if (ELEM(ob->type, OB_CURVE, OB_SURF))
        return join_curve_exec(C, op);
    else if (ob->type == OB_ARMATURE)
        return join_armature_exec(C, op);

    return OPERATOR_CANCELLED;
}
```

Come si vede dalla funzione join_exec [1] non è possibile fondere strutture con tipo diverso in un unico oggetto, infatti vengono chiamte funzioni differenti eseguendo proprio un test su ob− >type.

In particolare:

join_mesh_exec(C, op)[2]

Fonde tutte le Mesh selezionate in un unico oggetto (quello attivo) e in un'unica Mesh che conterrà nei vettori di vertici, edge e poligoni nuovi array contenenti tutti i vertici, edge e poligoni di tutte le mesh sulle quali è stato applicato il join.

join_curve_exec(C, op)[3]

Fonde tutte le NURBS selezionate in un unico oggetto e in un'unica struttura Curve che conterrà nel campo nurb una lista di strutture Nurb integrata con le nuove arrivate. Questo procedimento vale sia per curve che per superfici, le due però non possono essere mischiate.

[1] OBJECT_OT_join e join_exec sono definite in blender/source/blender/editors/object/object_add.c

[2] source: blender/source/blender/editors/mesh/meshtools.c

[3] source: blender/source/blender/editors/curve/editcurve.c

Capitolo 8

Integrare Blender

Per capire meglio come funzionano gli operatori ho provato a crearne alcuni utilizzando i due metodi che mette a disposizione Blender:

- integrare i sorgenti attraverso aggiunte al codice C
- utilizzare il codice Python (molto sponsorizzato dalla comunità di sviluppatori)

I listati degli operatori spiegati in questo capitolo possono essere trovati nell'APPENDICE B.

8.1 Integrare il codice sorgente

Per aggiungere un nuovo operatore ai sorgenti di Blender dobbiamo creare un nuovo operator type (esattamente come spiegato nel capitolo precedente) e le callback functions ad esso associate.

La scelta del file in cui inserire il nuovo operatore, deve essere fatta con un certo criterio: se pensiamo di agire su oggetti generici il codice andrà inserito in ../blender/source/blender/editors/object/object_edit.c, se opereremo sulle mesh in uno dei file nella cartella ../blender/source/blender/editors/mesh ecc... La posizione determinerà quando rendere disponibile all'utente l'operatore e la sintassi utilizzata nella console python.

Stampare le strutture

Spiegazione del Codice

Il primo operatore che vado a creare è una funzione per stampare le strutture dati.
È stata creata con lo scopo di dare un'occhiata più da vicino ai campi delle strutture Oggetto, Mesh, Curve e di tutte quelle ad esse collegate, stampando il loro contenuto in un file che ho chiamato strutture.txt.

Esegue i seguenti passi:

1. Apre il file strutture.txt in scrittura

2. Prende dal contesto l'oggetto attivo e stampa tutti i campi ad esso associati

3. Vengono chiamate funzioni differenti a seconda del tipo di oggetto:
   static void printmesh (Mesh *me, FILE *fd) per mesh
   static void printcurve (Curve *cu, FILE *fd) per curve e superfici
   Ognuna di queste funzioni stampa sul file strutture.txt i campi della struttura Mesh o Curve e, se l'oggetto si trova in EditMode, anche le strutture specifiche utilizzate per l'EditMode come la struttura EditBMesh.

Eseguire l'operatore

Questo operatore è stato scritto in C, quindi dopo avere ricompilato possiamo mandare in esecuzione Blender per vedere quali collegamenti sono stati creati in automatico.
Il risultato è il seguente: L'operatore che abbiamo creato può essere rintracciato nel menu attivabile attraverso la pressione della spacebar, selezionando la voce "Print Structure".

Se siamo invece più esperti possiamo raggiungere la console python integrata e mandare in esecuzione l'operatore digitando:

```
bpy.ops.object.printstructure()
```

se l'operatore va a buon fine verrà stampata la stringa 'FINISHED' in azzurro.
Ecco una piccola descrizione del comando utilizzato:

bpy
: sta per "blender python" l'API per interagire con Blender utilizzando script python.

ops
: sta per "operators" Gli operatori di Blender

object
: identifica il tipo di operatore che vogliamo mandare in esecuzione (in questo caso un operatore che agisce in generale sugli oggetti, ma al suo posto poteva esserci la parola "mesh", "curve", "view3d").
Printstructure è stato inserito in un file all'interno della directory ../blender/source/blender/editors/object, quindi la sottocategoria di operatore da richiamare è per l'appunto object.

printstructure()
: è il nome dell'operatore che vogliamo eseguire. Non prende argomenti perchè agisce direttamente sull'oggetto attivo prendendolo dal contesto.

Conoscere la sintassi python per accedere ad operatori e oggetti è utile perchè ci permetterà di utilizzare gli stessi comandi digitati da console anche integrandoli in script più complessi.

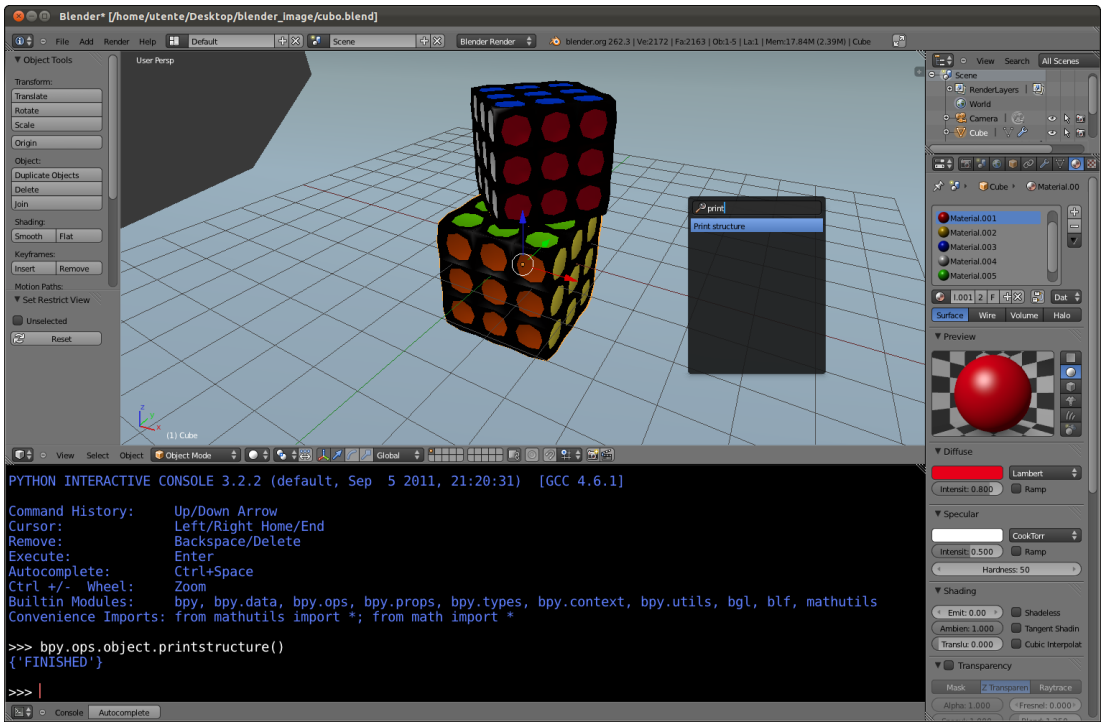

Figura 8.1: Eseguire l'operatore printstructure

8.2 Integrare Blender attraverso codice python

Come abbiamo accennato nella sezione precedente, Blender mette a disposizione una vista dedicata allo scripting.
Al suo interno troviamo:

- Text Editor: una finestra in cui possiamo scrivere il nostro codice python. Dal menu sottostante, alla voce template, è posibile selezionare e visualizzare circa una ventina di script che mostrano vari metodi per implementare operatori, menu, ecc... Un ottimo modo per cominciare a capire è modificarne qualcuno e vedere come si comportano gli oggetti in scena ai quali sono applicati.

- Python Console: una console che ci permette di chiamare operatori e script preesistenti al volo.

- View 3D: per vedere il risultato del nostro lavoro all'interno della scena.

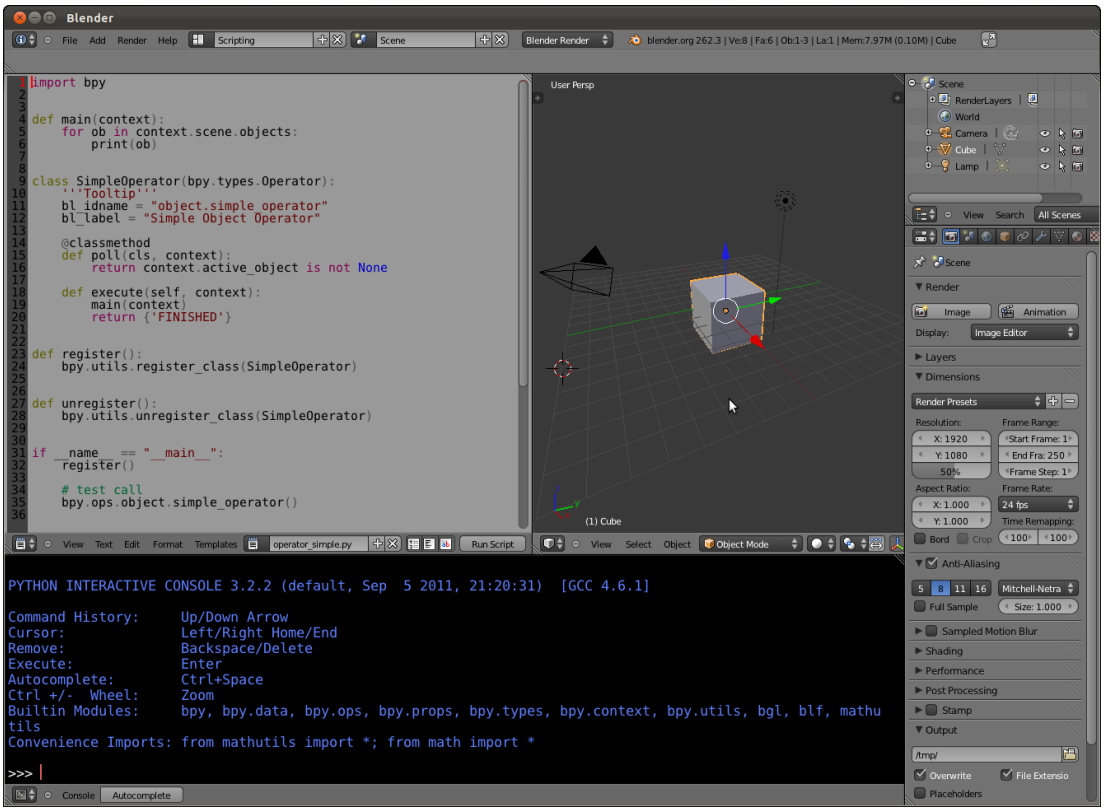

Figura 8.2: L'ambiente di scripting

Possiamo utilizzare questo ambiente per scrivere il codice dei nostri operatori e provarli attraverso il bottone Run Script. È proprio in questo modo che ho operato per implementare l'operatore di cui parleremo tra poco.

Incollare vertici e Punti di controllo

Il nuovo obiettivo, più affine alla mia ricerca, è capire se sia possibile interagire con due o più geometrie contemporaneamente. Per farlo ho provato ad implementare un nuovo operatore che porta i vertici della geometria attiva alle stesse coordinate dei vertici di altre geometrie selezionate.

L'implementazione consiste nei seguenti passi:

1. Si scorrono gli oggetti selezionati (scartando quello attivo) alla ricerca delle nuove destinazioni (vertici selezionati).

2. Quando se ne trova una, si trasformano le sue coordinate (locali) in coordinate globali attraverso la matrice di trasformazione che risiede nell'oggetto stesso e si salvano i nuovi valori in una lista.

3. Si prende dal contesto l'oggetto attivo e la sua matrice di trasformazione del mondo.

4. Si trasformano le coordinate nella lista in coordinate locali all'oggetto attivo in base alla sua matrice.

5. Si scorrono i suoi vertici per trovare quelli selezionati e quando se ne trova uno si inseriscono le nuove coordinate.

Per testare il nuovo operatore basta semplicemente premere ctrl+P dall'ambiente di script interno a Blender. Visto che, una volta creato, vorremmo poter utilizzare il nostro operatore molte volte, è irreale pensare di aprire il sorgente del nostro script nel text editor, e mandarlo in esecuzione ogni volta che ci serve. Perciò proviamo a creare una Add-on.

Installare l'Add-on

Add-On è il termine generico che indica ogni script opzionale che permette di estendere le funzionalità di Blender. Gli script ritenuti più utili sono già integrati in Blender ed installati come add-on, ma altri possono essere trovati sul web (su blender.org esiste una intera sezione dedicata) o implementati dagli utenti stessi. Anche noi ci cimenteremo in questa impresa.

Ora che abbiamo il codice del nostro operatore e che abbiamo visto essere funzionante, possiamo installarlo come Add-On nella nostra versione di Blender o passarlo ad altri utenti che potranno fare gli stessi passaggi per installarlo nelle loro versioni.

Per installare il nuovo operatore è necessario avere salvato il codice dell'operatore in un file del tipo nome_operatore.py (il nostro si chiamerà incolla_vertici.py).

Per far si che Blender riconosca l'operatore come Add-On deve essere aggiunta in capo al file una struttura dati che definisce alcune informazioni necessarie agli utenti che installeranno il nuovo operatore:

```
bl_info = {
    "name": "My Script",
    "description": "Description",
    "author": "Pinco Panco, Panco Pinco",
    "version": (1, 0),
    "blender": (2, 53, 0),
    "location": "Menu 1 > Menu 2",
    "warning": "",
    "wiki_url": "http://wiki.blender.org/index.php/Extensions:2.5/Py/"
                "Scripts/My_Script",
    "tracker_url": "http://projects.blender.org/tracker/index.php?"
                   "func=detail&aid=<number>",
    "category": "My category"}
```

La struttura è così riempita:

name (string)
: Nome dello script. Viene visualizzato nel menu Add-on

description (string)
: Una piccola descrizione per fare capire all'utente se può avere bisogno dello script in questione.

author (string)
: Nome dell'autore

version (tuple of integers)
: Versione dello script

blender (tuple of 3 integers)
: La versione minima di Blender richiesta per eseguire lo script

location (string)
: Dove può essere trovata la nuova funzionalità.

warning (string)
: Informa che è presente un bug nella versione corrente.

wiki_url (string)
: Link alla wiki page dello script. Qui si può inserire un manuale per usare lo script o link esterni.

tracker_url (string)
: Link alla pagina dello script, permette all'utente di fare notare bug incontrati.

category (string)
: Definisce il gruppo di appartenenza dello script.

Ora portiamo il file all'interno della cartella:
blender-svn/install/linux/2.62/scripts/addons
Dalla quale verranno prese le informazioni di tutti gli script presenti per aggiungerle ad un apposito menu.

Compiute queste operazioni preliminari possiamo avviare Blender ed installare l'Add-on.
Andiamo nel pannello delle proprietà (accessibile attraverso File > User Preferences) e accediamo alla voce Addons.

Come mostra la figura 8.3 dovremmo trovare nella categoria object l'operatore che abbiamo appena creato. Selezionandolo e cliccando su Install Addon, l'operatore sarà accessibile all'utente.
Per far sì che sia disponibile ogni volta che eseguiamo Blender, clicchiamo su Save As Default.
L'operatore è ora correttamente integrato nella nostra versione di Blender.

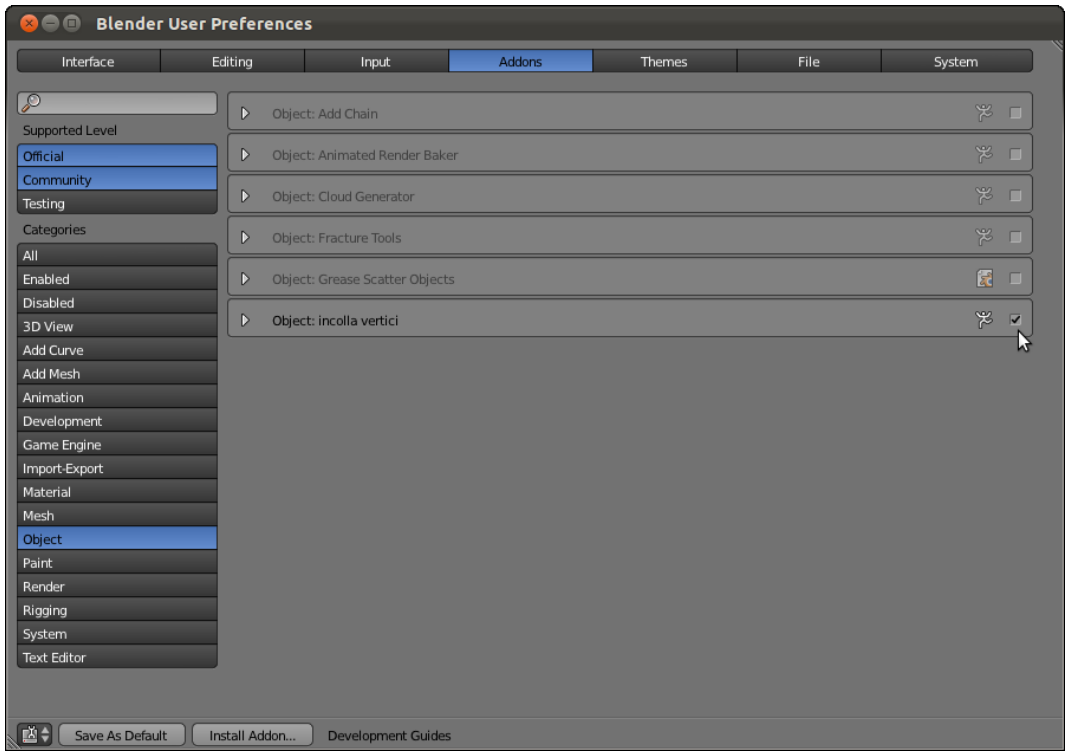

Figura 8.3: Installare l'Add-on "Incolla Vertici"

Esecuzione

L'utente è ora pronto per eseguire l'operatore "incolla vertici" in questo modo:

1. Si selezionano i vertici o i punti di controllo delle geometrie che vogliamo fare combaciare.

2. si selezionano tutte le geometrie che vogliamo fare interagire. L'ultima selezionata sarà quella attiva, quindi sarà quella alla quale verrà applicato il cambiamento.

3. Si esegue lo script cercandolo nel menu azionanile tramite spacebar oppure nella console python.

Il risultato è una geometria con coordinate dei vertici (o punti di controllo per curve e superfici) modificati. Infatti le nuove coordinate dell'oggetto attivo vengono portate alle coordinate dei punti selezionati per le altre geometrie.

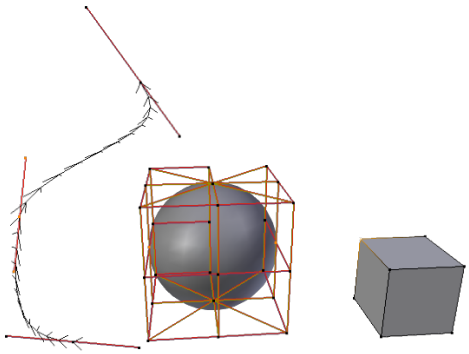

Figura 8.4: Selezione dei punti per l'operatore "Incolla Vertici"

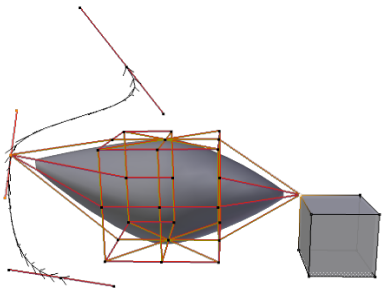

Figura 8.5: Risultato dell'operatore "Incolla Vertici"

8.3 Codice C o Codice python?

La funzione della sezione precedente è stata implementata solamente per le Mesh anche in codice C, soprattutto per capire quali differenze ci siano nell'implementazione.

Come ci si aspettava, è effettivamente più difficile scrivere operatori in linguaggio C, non tanto per l'algoritmo o per la gestione della memoria, ma soprattutto per le strutture dati. Avendo studiato approfonditamente la strutture non ho avuto grandi problemi, ma senza questo background l'implementazione sarebbe risultata particolarmente difficoltosa.

Ad esempio, per modificare le coordinate dei vertici della mesh, è necessario operare sulle strutture Custom Data, non sugli array di MVert, poichè essi sono soltanto una ripetizione e devono essere aggiornati di conseguenza!

Python e l'API bpy ci permettono un certo livello di astrazione che rende più veloce e meno impegnativo il lavoro e attraverso Add-On rende i nostri operatori trasportabili da un computer all'altro e da un utente all'altro. È anche vero, però, che non tutto quello che può essere fatto in C è fattibile in python! Quindi per profondi cambiamenti è necessaria una modifica diretta del codice sorgente.

Nelle figure 8.6 e 8.7 possiamo vedere che il comportamento di "Attacca vertici", implementato in C per Mesh, è lo stesso di Incolla Vertici, implementato invece in python.

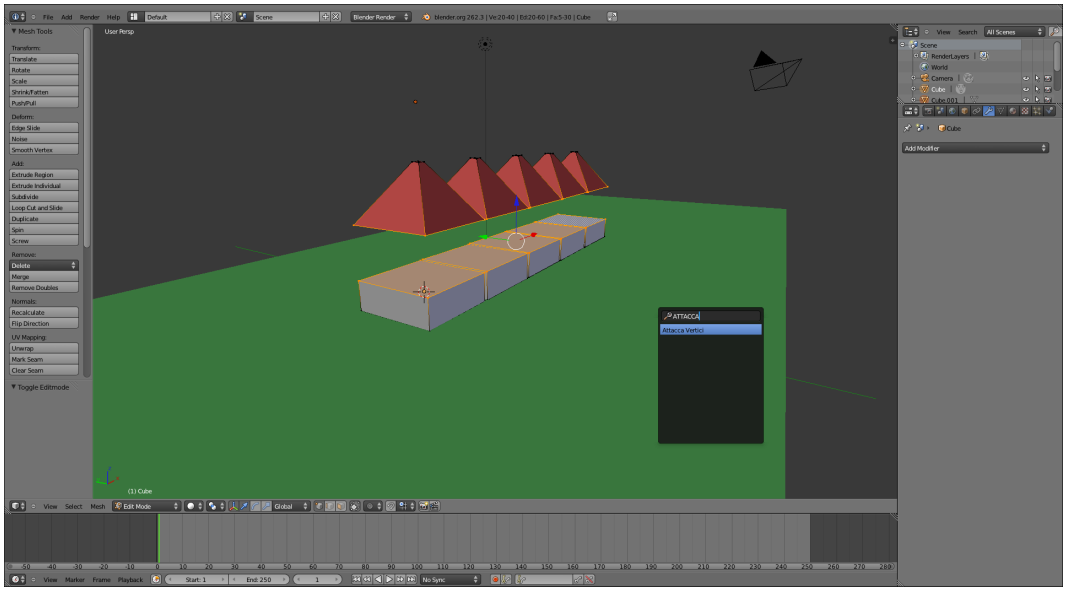

Figura 8.6: Esecuzione dell'operatore C "Attacca Vertici"

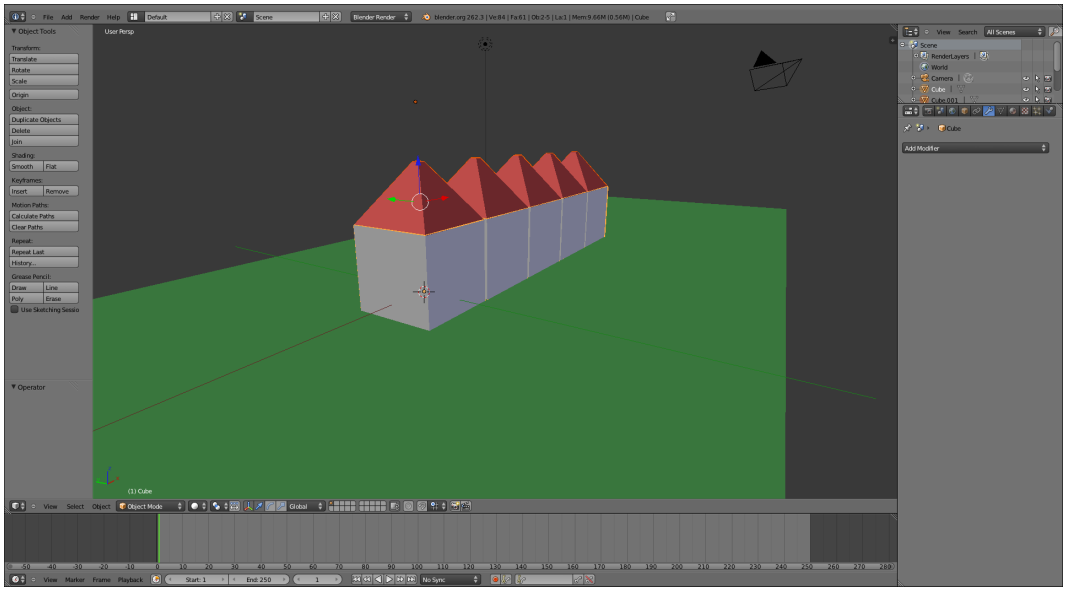

Figura 8.7: Risultato dell'operatore "Attacca Vertici"

Conclusioni

L'analisi portata avanti con questa ricerca ci permette di concludere il lavoro di tesi effettuando alcune considerazioni.

L'interazione tra più geometrie attraverso operatori appositamente studiati è possibile, come dimostrato nel capitolo 8 con la funzione per "incollare" vertici di due geometrie.

Ora resta da dimostrare se sia possibile creare un ambiente come gli attuali Edit Mode e Object Mode nel quale si possano applicare selezione contemporanea di vertici di due geometrie diverse e tutti gli operatori che possono essere applicati ai vertici.
La visualizzazione di più geometrie in Edit Mode non è un problema, infatti è bastata una modifica all'attuale entrata in Edit Mode per far si che quando selezioniamo più geometrie e le portiamo in EditMode queste vengano disegnate con poligono di controllo, maniglie o per le mesh con vertici, edge e facce. Per le immagini nel capitolo 8 si è utilizzata questa modifica.

Poichè, come abbiamo già visto, tutte le funzioni utilizzano scena e contesto come base su cui operare, probabilmente saranno necessarie modifiche a queste strutture dati per trasformare l'oggetto attivo in una lista di oggetti attivi.
Si potrebbe pensare di mettere al posto di Obedit nella scena una lista di oggetti editabili. In questo modo però tutti gli operatori che attingono al puntatore obedit (anche attraverso le funzioni che interrogano il contesto) continuerebbero ad utilizzare solo il puntatore alla prima entry nella lista. Per

rendere tutte le funzioni effettivamente funzionanti con piú oggetti sarebbe necessario un refactoring per aggiungere cicli che scorrano la lista di oggetti editabili, ma rendono pesante l'esecuzione. Questo porta a profondi cambiamenti all'interno del codice, che forse non vale la pena apportare.

Se il presupposto dell'Edit Multiplo è quello di chiamare operatori creati appositamente, la mia proposta è quella di creare un piccolo pannello nella barra degli strumenti per gli operatori che lavorano con più geometrie. Un bottone permette di acquisire la prima geometria alla quale vanno effettuati i cambiamenti, e una serie di Toggle Buttons permettono di scegliere gli altri oggetti coi quali lavorare. Selezionare una geometria la porterà in Edit Mode, e permetterà all'utente di selezionare vertici e utilizzare i normali operatori disponibili in Edit Mode. Potranno anche essere applicati operatori specifici, attraverso appositi bottoni nel pannello.

Qualunque sia la strada intrapresa, si tenga conto che per implementare operatori e pannelli non importa toccare i sorgenti C, ma tutto è fattibile attraverso python, con minore sforzo.

Appendice A

Le liste di Blender

Molte volte, nelle pagine precedenti ci siamo trovati a parlare di liste di oggetti integrati in campi di strutture dati, a scorrere liste di puntatori e ad utilizzare iteratori che indubbiamente agiscono su liste. Se si vuole interagire o apportare modifiche al codice, però, vale la pena spendere due parole sul tipo di liste utilizzate da Blender, sulla loro implementazione e sulle funzioni per operare su di esse.

Proprio perchè devono poter contenere elementi di natura diversa (materiali, oggetti, scene, altre liste, ecc) l'implementazione è la più generica possibile.

Le strutture dati messe a punto per implementarle vengono di seguito riportate e si trovano nel file: *source/blender/makesdna/DNA_listBase.h*

```
/* dynamicaly allocated dat structures has to
   include next and prev pointers */
typedef struct Link
{
    struct Link *next,*prev;
```

```
} Link;

/* ListBase points at the first and last pointer
   in dynamic list */
typedef struct ListBase
{
    void *first, *last;
} ListBase;
```

ListBase è un puntatore alla testa e alla coda della lista, mentre gli elementi al suo interno sono collegati da puntatori next e prev, implementando in questo modo una lista bidirezionale doppiolinkata.

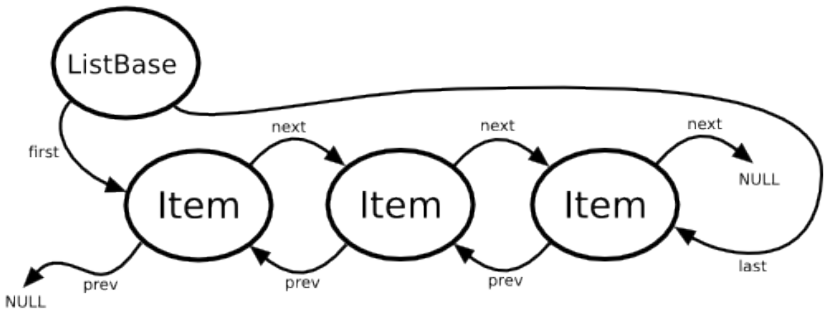

Figura A.1: Implementazione liste di Blender

Per operare su ogni elemento della lista, possiamo scorrerla in avanti utilizzando un costrutto for e dandogli come parametro di ingresso il puntatore alla testa della lista e ad ogni iterazione spostarci sull'elemento successivo attraverso il campo next:

```
for (item = lb.first; item; item= item->next) {
    do_some_operation_with_item();
}
```

Allo stesso modo possiamo scorrerla all'indietro partendo dalla coda della lista e navigando attraverso il campo prev.

Le funzioni che agiscono su liste sono implementate nel file:
source/blender/blenlib/intern/util.c

mentre i prototipi di tali funzioni sono rintracciabili qui:
source/blender/blenlib/BLI_blenlib.h

```
void addlisttolist(ListBase *list1, ListBase *list2);
void BLI_insertlink(struct ListBase *listbase,
                    void *vprevlink, void *vnewlink);
void * BLI_findlink(struct ListBase *listbase, int number);
void BLI_freelistN(struct ListBase *listbase);
void BLI_addtail(struct ListBase *listbase, void *vlink);
void BLI_remlink(struct ListBase *listbase, void *vlink);
void BLI_addhead(struct ListBase *listbase, void *vlink);
void BLI_insertlinkbefore(struct ListBase *listbase,
                          void *vnextlink, void *vnewlink);
void BLI_freelist(struct ListBase *listbase);
int BLI_countlist(struct ListBase *listbase);
void BLI_freelinkN(struct ListBase *listbase, void *vlink);
```

I nomi di queste funzioni spiegano già molto bene cosa si apprestano a fare: troviamo funzioni per aggiungere un elemento in testa o in coda, rimuovere elementi, trovarne, contare il numero di entry, ecc...

Appendice B

Codice delle funzioni descritte nel capitolo 8

Stampa strutture - codice C

```
void OBJECT_OT_print_structure(struct wmOperatorType *ot)
{
    /* identifiers */
    ot->name = "Print structure";
    ot->description = "Print data structures attributes in scrivi.txt";
    ot->idname = "OBJECT_OT_printstructure";

    /* api callbacks */
    ot->exec = printobject;
    ot->poll = ED_operator_object_active_editable;
    /* flag */
    ot->flag = OPTYPE_REGISTER|OPTYPE_UNDO;
}

int printobject(bContext *C, wmOperator *UNUSED(op)) {
FILE *fd;
Object *ob = ED_object_active_context(C);
```

```
printf("stampa strutture su file\n");
/* apre il file in scrittura */
fd=fopen("strutture.txt", "a");
if( fd==NULL ) {
    perror("Errore in apertura del file");
    exit(1);
}

fprintf(fd, "-------------------struttura Oggetto------------------\n");
/* rot en drot have to be together! (transform('r' en 's')) */
//float loc[3], dloc[3], orig[3];
fprintf(fd, "locazione: %f %f %f \n", ob->loc[0], ob->loc[1],ob->loc[2]);
fprintf(fd, "dloc: %f %f %f \n", ob->dloc[0], ob->dloc[1],ob->dloc[2]);
//float size[3];              /* scale infact */
fprintf(fd, "\nsize: %f %f %f \n", ob->size[0], ob->size[1],ob->size[2]);
//float rot[3], drot[3]; /* euler rotation */
fprintf(fd, "rotazione: %f %f %f \n", ob->rot[0], ob->rot[1],ob->rot[2]);
fprintf(fd, "drot: %f %f %f \n", ob->drot[0], ob->drot[1],ob->drot[2]);
if(ob->disp.first== NULL)fprintf(fd, "la displist e' NULL\n");
else{
    DispList *disp;
    int i;
    float *a;
    disp = ob->disp.first;
    while(disp){
        fprintf(fd, "DISPLIST\n");
        fprintf(fd, "type: %d\n", disp->type);
        fprintf(fd, "numero: %d parti: %d \n", disp->nr, disp->parts);
        a=disp->verts;
        for (i=0; i<96;i++){
            if (a!=NULL){
```

```
                fprintf(fd, "verts: %f \n", *a);
                a++;
            }
        }
        disp=disp->next;
    }
}
/*Chiamo funzioni diverse a seconda che il tipo di dati da stampare
sia una struttrua Mesh o una struttura Curve.*/

if (ob->type ==OB_MESH)
    printmesh((Mesh *)ob->data,fd);

else if (ob->type ==OB_SURF || ob->type == OB_CURVE)
    printcurve((Curve *)ob->data,fd);

else
    printf("printobject: oggetto non riconosciuto\n");
fprintf(fd, "------------------------------------------------------\n");
fclose(fd);

  return OPERATOR_FINISHED;
}
```

Viene omesso il codice delle funzioni:
static void printmesh(Mesh *me, FILE *fd)
static void printcurve(Curve *cu, FILE *fd)
poichè gli attributi stampati sono stati già ampiamente descritti nei capitoli precedenti.

Incolla Vertici - script python

```
bl_info = {
    "name": "Incolla vertici",
    "description": "Incolla i vertici della geometria
                    selezionata a quelli di altre geometrie",
    "author": "Simona Arma",
    "version": (1, 0),
    "blender": (2, 53, 0),
    "location": "",
    "warning": "",
    "wiki_url": "",
    "tracker_url": "",
    "category": "Object"}

import bpy
import mathutils
from mathutils import Vector
def main(context):
    B=bpy.context.active_object
    Bwmtx = B.matrix_world
    print(Bwmtx)
    NewCoord = list()
    #Raccolgo i punti di arrivo:
    for A in bpy.data.objects:
        if A != B and A.select:
            Awmtx=A.matrix_world
            print(Awmtx)
            if A.type=='MESH':
                for vert in A.data.vertices:
                    if vert.select:
                        coord=vert.co
```

```
                print(coord)
                #ottengo le coordinate globali
                coord= Awmtx*coord
                print(coord)
                NewCoord.append(coord)

    elif A.type == 'CURVE':
         for spline in A.data.splines:
            if spline.type=='NURBS':
                for point in spline.points:
                    if point.select:
                        coord=point.co
                        print('point.co')
                        print(coord)
                        weight=coord[3]

                        coordx=coord[0]*weight
                        coordy=coord[1]*weight
                        coordz=coord[2]*weight
                        print('peso')
                        print(weight)
                        vec=Vector((coordx,coordy,coordz,weight))
                        vec= Awmtx*vec
                        vec=vec/weight
                        NewCoord.append(Vector((vec[0],vec[1],vec[2])))
                        print(vec[0],vec[1],vec[2])


            else: #bezier
                for point in spline.bezier_points:
                    if point.select_control_point:
```

```
                            coord=point.co
                            coord= Awmtx*coord
                            NewCoord.append(coord)
            elif A.type == 'SURFACE':
                 for spline in A.data.splines:
                     for point in spline.points:
                            if point.select:
                                coord=point.co
                                coord=point.co
                                print('point.co')
                                print(coord)
                                weight=coord[3]

                                coordx=coord[0]*weight
                                coordy=coord[1]*weight
                                coordz=coord[2]*weight
                                print('peso')
                                print(weight)
                                vec=Vector((coordx,coordy,coordz,weight))
                                vec= Awmtx*vec
                                vec=vec/weight
                                NewCoord.append(Vector((vec[0],vec[1],vec[2])))
                                print(vec[0],vec[1],vec[2])



print('Dove devo mettere i punti')
print (NewCoord)
i=0
if B.type == 'MESH':
```

```
        for vert in B.data.vertices:
            if vert.select:
                #ottengo le coordinate locali a B
                coord=Bwmtx.inverted()*NewCoord[i]
                print(coord)
                vert.co=coord
                i=i+1
    elif B.type == 'CURVE':
        for spline in B.data.splines:
            if spline.type=='NURBS':
                for point in spline.points:
                    if point.select:
                        coord=Bwmtx.inverted()*NewCoord[i]
                        point.co[0:3]=coord
                        point.co[3]=1
                        i=i+1
            else: #bezier
                for point in spline.bezier_points:
                    if point.select_control_point:
                        coord=Bwmtx.inverted()*NewCoord[i]
                        point.co=coord
                        i=i+1
    elif B.type == 'SURFACE':
         for spline in B.data.splines:
             for point in spline.points:
                    if point.select:
                        coord=Bwmtx.inverted()*NewCoord[i]
                        point.co[0:3]=coord
                        point.co[3]=1
                        i=i+1
```

```
class IncollaVertici(bpy.types.Operator):
    '''Tooltip'''
    bl_idname = "object.incolla_vertici"
    bl_label = "incolla vertici"

    @classmethod
    def poll(cls, context):
        return context.active_object is not None

    def execute(self, context):
        main(context)
        return {'FINISHED'}


def register():
    bpy.utils.register_class(IncollaVertici)


def unregister():
    bpy.utils.unregister_class(IncollaVertici)


if __name__ == "__main__":
    register()

    # test call
    bpy.ops.object.incolla_vertici()
```

Attacca Vertici - Operatore per Mesh in codice C

```
void OBJECT_OT_attacca_vertici(struct wmOperatorType *ot)
```

```
{
    /* identifiers */
    ot->name = "Attacca Vertici";
    ot->description = "Attacca i vertici di due geometrie";
    ot->idname = "OBJECT_OT_attacca_vertici";

    /* api callbacks */
    ot->exec = attacca_vertici_exec;
    //ot->invoke=attacca_vertici_invoke;
    //ot->modal=attacca_vertici_modal;
    ot->poll = ED_operator_object_active_editable;
    /* flag */
    ot->flag = OPTYPE_REGISTER|OPTYPE_UNDO;
}

int attacca_vertici_exec(bContext *C, wmOperator *UNUSED(op)) {
typedef struct coord_entry{
    float vec[3];
    struct coord_entry *next;
} coord_entry;
Scene *scene= CTX_data_scene(C);
Object *ob = ED_object_active_context(C);
Base *base;
coord_entry *entry, *p;
coord_entry *NewCoord =NULL;
int i=0;
MVert mvert; Mesh *me;
BMEditMesh *em= NULL; BMVert *bmvert= NULL;
float mat[4][4];
float coord[3];
```

```
//*MEM_mallocN(size_t len, const char *str)
/*creare una lista*/
printf("ciccio ciaccio\n");
for (base = FIRSTBASE; base; base=base->next) {
    if((base->object!=ob)&&(base->flag & SELECT)){

            printf("%d\n", base->object->type);
            /*prendo la matrice del mondo*/
            switch (base->object->type){
            case OB_MESH:
                 me =(Mesh *) base->object->data;
                 copy_m4_m4(mat, base->object->obmat);
                 print_m4("matrice", mat);
                 EDBM_mesh_make(scene->toolsettings, scene, base->object);
                 em = me->edit_btmesh;
                 EDBM_selectmode_flush(em);
                 EDBM_mesh_free(em);
                 me->edit_btmesh=NULL;
                 printf("\nCoordinate dei vertici:\n");
                 for (i=0;i<me->totvert;i++){
                        mvert=me->mvert[i];
                        if(mvert.flag & SELECT){
                                  printf("select\n");
                                  coord[0]=mvert.co[0];
                                  coord[1]=mvert.co[1];
                                  coord[2]=mvert.co[2];
                                  mul_m4_v3(mat, coord);
                                  entry= (struct coord_entry *)
                                    malloc(sizeof(struct coord_entry));
                                  entry->vec[0] =coord[0];
                                  entry->vec[1] =coord[1];
```

```
                                    entry->vec[2] =coord[2];
                                    entry->next=NULL;
                                    if (NewCoord==NULL) {
                                           NewCoord=entry;
                                           p=NewCoord;
                                    }
                                    p->next=entry;
                                    p=p->next;
                             }
                        }
                    break;
                    }


        }

}

me=(Mesh *)ob->data;
printf("editbmesh non pronta\n");
EDBM_mesh_make(scene->toolsettings, scene, ob);
em = me->edit_btmesh;
EDBM_selectmode_flush(em);
EDBM_mesh_free(em);
me->edit_btmesh=NULL;
copy_m4_m4(mat, ob->imat);
p=NewCoord;

for (i=0;i<me->totvert;i++){
    mvert=me->mvert[i];
    if(mvert.flag & SELECT){
```

```
            mul_m4_v3(mat, p->vec);
            CustomData_set(&me->vdata, i, CD_MVERT, p->vec);
            p=p->next;
        }
    }
    mesh_update_customdata_pointers(me, TRUE);
    for (entry = NewCoord; entry; entry=entry->next) {
        p=entry;
        free(p);
    }
    return OPERATOR_FINISHED;
    }
```

Appendice C

Note sugli strumenti utilizzati

L'oggetto della tesi è l'ultima release di Blender presente al momento in cui si è deciso di intraprendere il lavoro: Blender 2.62.
Poichè la comunità di sviluppatori esegue release bimestrali, sicuramente dovranno essere apportate delle modifiche alla tesi se si confronta con versioni successive di Blender.

Il codice sorgente è stato scaricato utilizzando subversion come consigliato nella documentazione rintracciabile all'indirizzo:
http://wiki.blender.org/index.php/Dev:2.5
/Doc/Building_Blender/Linux/Ubuntu/Scons

Per navigare all'interno del codice sorgente (e per gli operatori in codice C) è stata utilizzata la piattaforma Eclipse integrata da plug-in CDT per sviluppare in C/C++.
Per la compilazione è stato utilizzato scons (già presente nel pacchetto per sviluppatori di Blender). Alcune informazioni sul compilatore possono essere trovate sul sito ufficiale: http://www.scons.org

Gli script python sono stati realizzati direttamente nell'ambiente di scripting integrato in Blender.

Pe la redazione dell'elaborato scritto è stato utilizzato latex attraverso il software Texmaker.

Bibliografia

[1] http://www.blender.org
Introduzione, storia e schemi su Liste di blender, Codice sorgente e Modello

[2] http://wiki.blender.org
Documentazione sul codice sorgente

[3] Slide del corso di Grafica del Prof. Casciola
Per la parte generale su mesh, curve e superfici

[4] http://it.wikibooks.org/wiki/Blender

[5] http://www.blender.org/documentation/blender_python_api_2_63_11
Riferimenti per lo sviluppo del codice python attraverso l'API fornita da Blender

Ringraziamenti

Ringrazio il Professor Casciola per la supervisione e collaborazione.
Grazie al Dottor Flavio Bertini per i suoi suggerimenti.
Un grazie di cuore alla mia famiglia, agli amici e a tutti i ragazzi che mi hanno accompagnato in questo viaggio universitario!!