Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Una libreria OpenGL per la selezione e editing di mesh poligonali

Tesi di Laurea in Grafica

Relatore:
Chiar.mo Prof.
Giulio Casciola

Presentata da:
Davide Aguiari

Sessione III
Anno Accademico 2012/2013

"I would love to change the world,
but they won't give me the source code"
Unknown

Introduzione

Nel mondo Open Source, la libreria grafica *OpenGL* è oggi ampiamente utilizzata in svariati settori come l'animazione 2D/3D, la modellazione *CAD*[1] o nello sviluppo di videogiochi.
A causa dei suoi innumerevoli usi e dell'astrazione che OpenGL permette di ottenere su diversi ambienti grafici, lo sviluppatore - che la utilizza - è vincolato a cercare librerie di supporto al fine di sfruttarne al meglio le potenzialità.

Questa tesi si configura su questi presupposti, presentando una libreria di selezione e editing di mesh 3D basata su OpenGL.
La libreria, chiamata *libEditMesh*, sfrutta il meccanismo geometrico del *RayPicking* permettendo all'utilizzatore di identificare col mouse punti, facce e lati di solidi in scena.

La tesi si articola sostanzialmente in due parti:
nella prima vengono proposte alcune soluzioni ad-hoc sviluppate su applicazioni già esistenti nel panorama openSource, e non;
nella seconda vengono esposti gli algoritmi e funzioni implementate in libEditMesh.

[1] Computer-Aided Design

Indice

Introduzione **i**

1 OpenGL **1**

1.1 Un po' di storia 1
1.2 Come funziona 2
1.3 GLUT - openGL Utility Toolkit 7

2 Mesh - Selezione e editing **9**

2.1 Le mesh 9
2.1.1 Mesh .m 11
2.1.2 Mesh .ply 12
2.1.3 Mesh .mesh 13
2.1.4 Mesh .obj 14
2.1.5 Mesh .off 15
2.2 Interagire con le mesh 15
2.2.1 Blender 15
2.2.2 Maya 19
2.2.3 Rhino (Rhinoceros) 22
2.2.4 MeshLab 23
2.3 Algoritmi di selezione 25
2.3.1 GL_SELECT 25
2.3.2 Color index Picking 26
2.3.3 Occlusion Queries 27
2.3.4 Ray Picking 28

3 Il progetto **33**

3.1 La libreria Glm 33

3.2 La libreria libEditMesh 37

3.2.1 Funzioni di selezione 39

3.2.2 Funzioni di disegno 46

3.2.3 Funzioni di editing 47

3.3 Interfacciarsi a libEditMesh 53

3.3.1 Compilazione e linking 56

3.4 Sviluppi futuri 57

Conclusioni **59**

Bibliografia **62**

Capitolo 1

OpenGL

OpenGL[1] è una interfaccia software indipendente dal sistema operativo e dal *Window System*, utilizzata nel mondo Open Source e non, per creare applicazioni interattive, per gestire oggetti geometrici 2D/3D o semplicemente per interagire con l'hardware grafico.

Le API[2] hanno il compito di renderizzare una geometria in scena, cioè di produrre una rappresentazione di qualità di oggetti 3D, ma non forniscono comandi ad alto livello per descrivere modelli: senza l'utilizzo di librerie più sofisticate (es. GLU OpenGL Utility Library), è solamente possibile definire punti, linee e poligoni.

1.1 Un po' di storia

Negli anni 80 *Silicon Graphics* diede inizio a *IRIS GL*, API grafica proprietaria, fortemente legata all'hardware delle workstation IRIS-based. Solo negli anni 90 SGI decise di dare vita a OpenGL, aprendo il codice di IRIS GL e svincolandolo dall'architettura sottostante.

[1]Open Graphics Library

[2]Application Programming Interface: insieme di procedure disponibili al programmatore raggruppate a formare un set di strumenti specifici

Nonostante qualche iniziale tentativo, fallito, di unificare OpenGL e *Microsoft DirectX*, il 1 luglio 1994 OpenGL uscì con la versione 1.0 basata interamente su pipeline fissa. La versione attuale è la 4.4 (22 luglio 2013) frutto di 20 anni di correzioni e innovazioni portate avanti dalla *ARB*[3], un insieme di compagnie, al fine di migliorarne struttura e prestazioni; nel corso degli anni in ARB troviamo 3DLab, ATI, Dell, Evans e Sutherland, SGI, HP, IBM, Intel, Matrox, NVIDIA e Sun e altre aziende minori.
La natura fortemente estendibile di OpenGL ha permesso, versione dopo versione, di poterne aumentare l'efficienza, accettando come standard anche moduli sviluppati da terzi, soprattutto ATI e NVIDIA.

1.2 Come funziona

OpenGL offre un insieme di diverse centinaia di funzioni che consentono di accedere a quasi tutte le caratteristiche dell'hardware video. Internamente, agisce come una **macchina a stati finiti**: alla libreria vengono dati diversi input che ne alterano lo stato, inducendo alla produzione di differenti output visibili.

Utilizzando le funzioni API è possibile attivare o disattivare i vari aspetti di questa macchina, come gli stili di resa, lo shading, le luci o il texture mapping.
La maggior parte delle implementazioni di OpenGL calcola la scena svolgendo una serie di differenti stadi; questa serie prende il nome di **pipeline grafica**.
Le prime versioni offrivano agli sviluppatori la **pipeline fissa**: fortemente legata all'hardware, non offriva nessuna libertà di modifica sulla trasformazione della geometria o sul calcolo dei colori e della profondità.
Il seguente diagramma mostra una versione semplificata, ma esaustiva, dei passaggi più importanti del funzionamento della pipeline fissa.

[3] Architecture Review Board

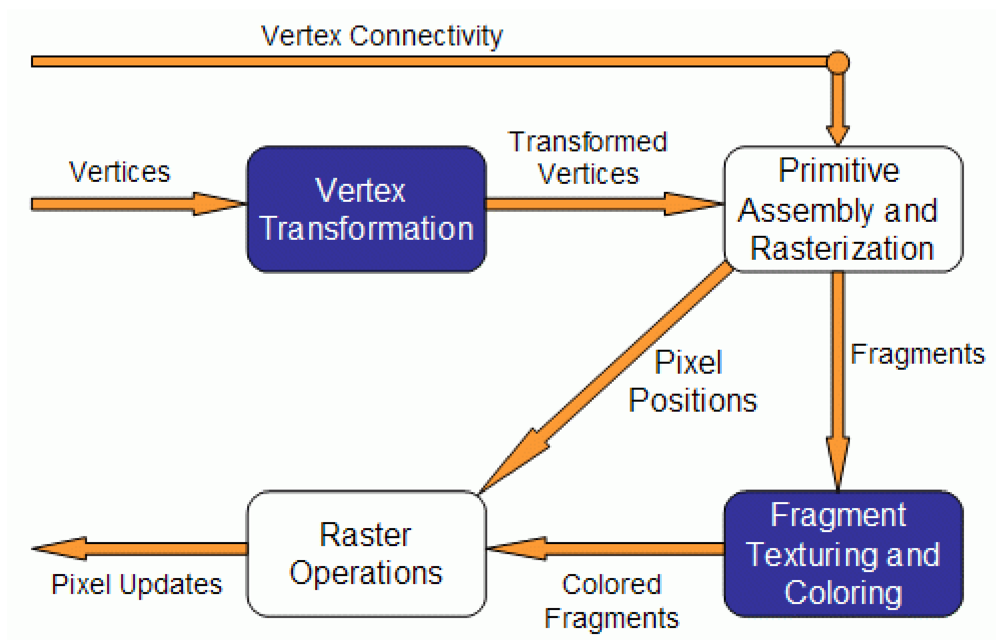


Figura 1.1: Pipeline fissa

1. **Vertex Transformation**: definiti i vertici in scena, questi vengono trasformati calcolando le rototraslazioni delle loro posizioni, le coordinate finali sullo schermo, l'illuminazione e le coordinate di mapping delle texture.

2. **Primitive Assembly and Rasterization**: i vertici vengono raggruppati in modo da formare le primitive, passando per i check di *frustum culling* e di *clipping* della finestra. Successivamente la primitiva viene rasterizzata generando dei *fragments*. Nei fragments troviamo informazioni come la locazione nel *framebuffer*, il colore o la profondità.

3. **Fragment Texturing and Coloring**: i fragment vengono quindi interpolati e combinati insieme ai *texel* (texture element).

4. **Raster Operations**: in questa ultima fase i fragment vengono sottoposti a diversi test (*Scissor Test, Alpha Test, Stencil Test, Depth buffer Test, Blending e Dithering*) fino al salvataggio finale nel framebuffer, pronti per essere visualizzati a video. Il fallimento di uno di questi test, preclude la visualizzazione del fragment sullo schermo.

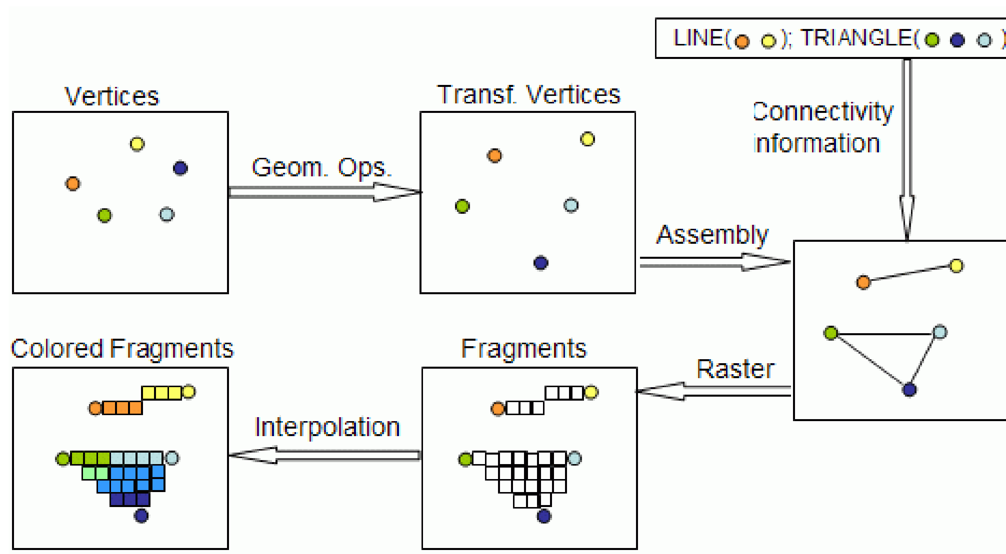


Figura 1.2: Effetti della pipeline sui vertici

Con l'avanzare delle specifiche OpenGL (2.0+), si è preferito superare la rigidità della pipeline fissa a favore di quella **programmabile**: lo sviluppatore, non più così vincolato, ha la possibilità di specificare un codice ad-hoc per alcuni stadi della pipeline.

Di seguito alcune delle fasi più importanti:

- **Display List**: tutte le informazioni sulla geometria e/o i pixel possono essere salvati in particolari strutture chiamate *display list*. Quello che rende le display list uno dei metodi più veloci di rendering, è il fatto di poter minimizzare il trasferimento di dati all'hardware grafico, riutilizzando innumerevoli volte le stesse informazioni, senza doverle rivalutare a ogni computazione. L'unico svantaggio di queste struttura è la scarsa flessibilità alle modifiche: una volta che una display list viene compilata, non può essere modificata dinamicamente.

- **Evaluators**: permette di definire in scena le normali alla superficie, le coordinate texture, i colori e le coordinate spaziali, partendo dai control points.

- **Per-Vertex Operations and Primitive Assembly**: in questa fase i vertici sono convertiti in primitive; inoltre è in questa fase che avvengono le trasformazioni geometriche dalle coordinate mondo alle coordinate schermo. Se richiesto vengono trasformate le coordinate texture e le luci, i materiali e i colori.
  Nella seconda fase viene applicato il clipping della scena, definendo quale parte di geometria verrà visualizzata.

- **Pixel Operations**: mentre le informazioni sui vertici vengono valutate, parallelamente vengono analizzate anche quelle sui pixel; se i pixel vengono letti dal frameBuffer, vengono sottoposti a diverse operazioni come *scale, bias, mapping e clamping*, per poi essere memorizzati in apposite strutture in memoria. La maggior parte di queste fasi sono rimaste invariate rispetto alla pipeline fissa, oppure sono state raggruppate nel *Fragment Shader*.

- **Texture Assembly**: OpenGL può applicare immagini texture alla geometria per rendere gli oggetti più realistici.

- **Rasterization**: qui informazioni pixel e vertex vengono unificate nei fragments visti in precedenza. Ogni quadrato fragment corrisponde a un pixel nel frameBuffer.

- **Fragment Operations**: creati i fragments, questi possono essere accoppiati con i texel se richiesto; qui sono combinati i colori primari e secondari e calcolata la profondità. Come per la pipeline fissa, i fragments vengono sottoposti a una serie di test, il cui fallimento preclude l'inserimento nel buffer finale.

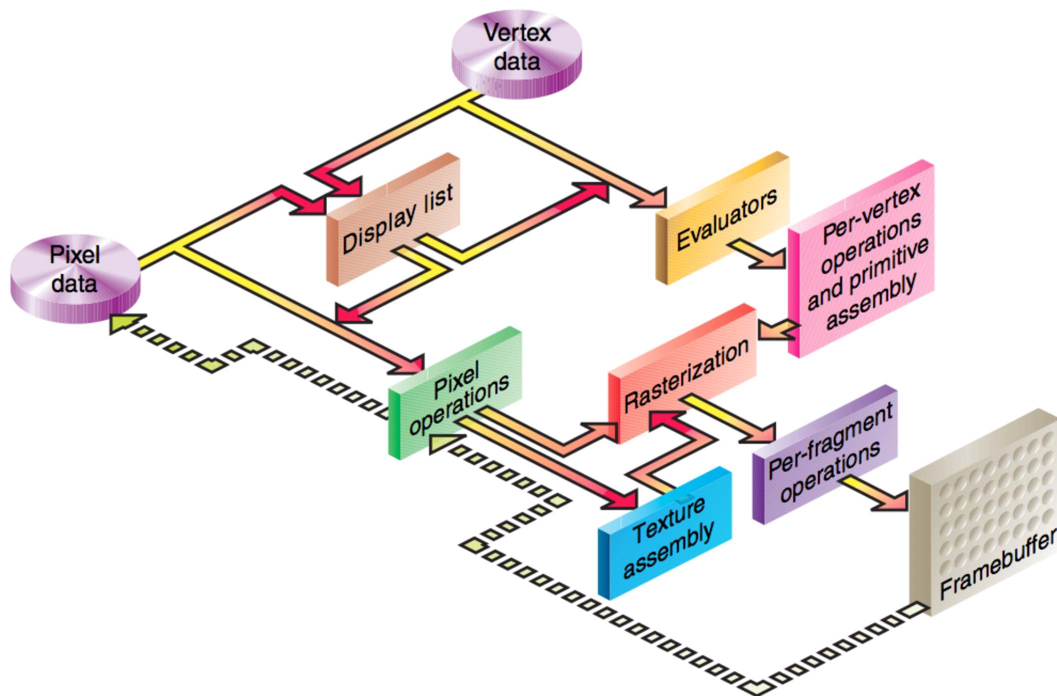


Figura 1.3: Pipeline Grafica Programmabile

Per facilitare il controllo delle varie trasformazioni su vertici e pixel sono stati introdotti tre Shaders: *Vertex Shader, Fragment Shader e Geometry Shader*. Sostanzialmente vengono così accorpate in tre uniche fasi le modifiche da apportare sui tre diversi oggetti.

- **Vertex Shader**: allo shader vengono fornite informazioni sui vertici, come la posizione o le normali. Inoltre è in questa fase che viene trasformata la geometria attraverso le matrici *ModelView* e *Projection*, vengono trasformate le normali, vengono generate le coordinate texture e calcolate luci e colori. Tutte queste operazioni vengono fatte individualmente su ogni vertice, mentre non vi sono ancora informazioni sulla relazione tra di essi.

- **Fragment Shader**: lo shader prende come input le informazioni interpolate dei vertici, come la posizione, il colore, le normali ecc. Si occupa quindi della resa dei pixel associati ai vertici. Qui vengono calcolati gli effetti come il *bump mapping*, le ombre, gli effetti di diffrazione e di rifrazione.

- **Geometry Shader** (Opzionale): a differenza del *Vertex Shader*, questo trasforma primitive al fine di crearne di nuove, sfruttando anche informazioni aggiuntive come le adiacenze.

L'uso della pipeline programmabile è obbligatoria dalla versione 3.1 di OpenGL e, a differenza della 2.0, non risulta più retrocompatibile.

1.3 GLUT - openGL Utility Toolkit

Come scritto all'inizio di questo capitolo, OpenGL è potente, ma non offre costrutti per gestire il *Windows System*, limitandosi alla sola renderizzazione della scena geometrica.
Grazie però alla sua **natura estendibile** e all'impegno di numerosi sviluppatori, *GLUT* (openGL Utility Toolkit) colma questa lacuna.
GLUT introduce innumerevoli supporti alla gestione degli eventi, degli inputs, all'interazione uomo/macchina, al disegno di solidi complessi.
La versione inizialmente sviluppata da *Mark Kilgard*, dipendente di *Silicon Graphics Inc.*, non è più mantenuta, ma parallelamente sono state sviluppate alternative open come *freeGlut*.

Capitolo 2

Mesh - Selezione e editing

2.1 Le mesh

Col termine *Mesh poligonali* si intende indicare la collezioni di poligoni o **facce**, che insieme formano la **superficie di un oggetto**.
Sostanzialmente sono insiemi di vertici, lati e facce distribuite in scena; formalmente una mesh è una tupla (K,V) dove V è l'insieme dei punti nello spazio (vertici), mentre K è l'insieme dei complessi simpliciali[1]. La mesh contiene le informazioni sulla connettività (topologia) tra i punti contenuti in V (ossia lati e facce).
Le facce possono assumere qualsiasi poligono, purchè piano e convesso.
In pratica vengono utilizzati i triangoli perchè:

- Matematicamente semplici
- Sempre piani
- Sempre convessi
- Facili da rasterizzare (l'hardware grafico è basato su triangoli)
- Strutture dati semplici

[1]Il simplesso di dimensione zero è un singolo punto, il simplesso bidimensionale è un triangolo e quello tridimensionale è un tetraedro. Il simplesso n-dimensionale ha n + 1 vertici.

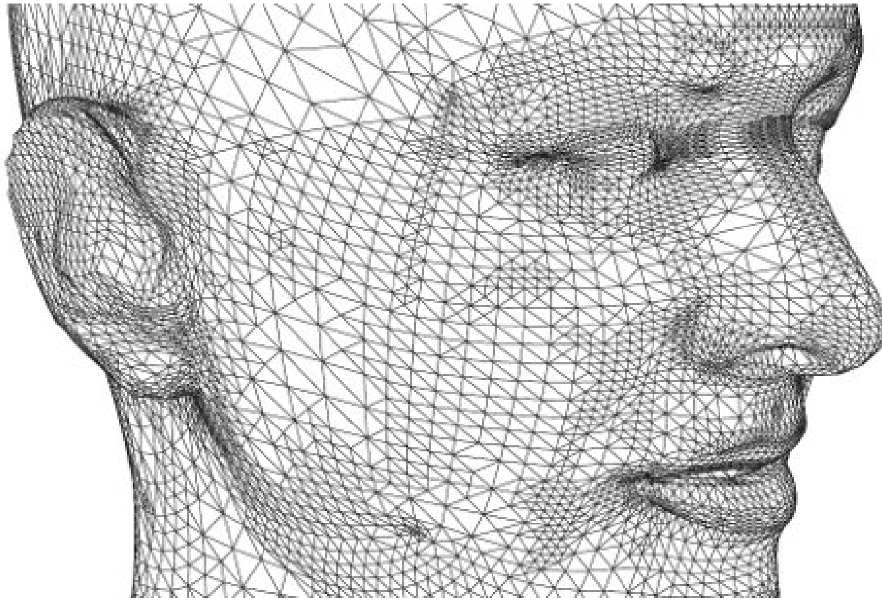


Figura 2.1: Faccia 3D ricostruita tramite mesh

Una mesh può essere definita:

Geometricamente mediante la posizione dei vertici nello spazio 3D.

Topologicamente mediante informazioni su come sono connessi i vertici, lati e facce. Si differenziano ulteriormente in mesh strutturate/non strutturate, strutturate e a blocchi e poliedriche.

Proprietà

- **Solidità**: una mesh rappresenta un oggetto solido se le sue facce racchiudono uno spazio positivo e finito.
- **Connettività**: una mesh si dice connessa se esiste un percorso di edge senza interruzioni, dati due vertici qualsiasi su di essa.
- **Semplicità**: una mesh si dice semplice se l'oggetto rappresentato non presenta buchi.
- **Convessità**: una mesh rappresenta un oggetto convesso se, unendo due vertici, il segmento giace interamente dentro l'oggetto stesso.

Tipicamente le mesh vengono memorizzate e indicizzate in memoria attraverso liste di vertici (coordinate X,Y,Z) e mediante liste di facce orientate (per ogni faccia, indici dei vertici).

2.1.1 Mesh .m

Questo tipo di mesh è presente solo in *ASCII* (testo) ed è di origine accademica (*New York University*); formato per mesh esclusivamente triangolari.

```
#
# Comments
#
Vertex 1 0.500000 -0.500000 0.500000
Vertex 2 -0.500000 -0.500000 0.500000
Vertex 3 -0.500000 0.500000 0.500000
....
Face 1 4 2 1
Face 2 4 3 2
Face 3 8 3 4
Face 4 8 7 3
Face 5 7 8 5
...
Edge 4 2
Edge 2 1
Edge 1 4
...
```

Numerazione vertici
Numerazione facce
Opzionali

Figura 2.2: Esempio di file mesh .m

2.1.2 Mesh .ply

Può essere in binario, o in *ASCII* (testo); è stato ideato alla *Stanford University* nell'ambito del *Progetto Michelangelo Digitale*. E' un formato per mesh generiche anche miste.

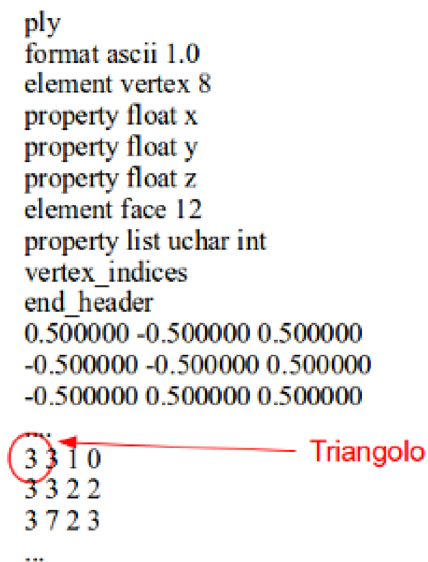


Figura 2.3: Esempio di file mesh .ply

2.1.3 Mesh .mesh

E' un formato *ASCII* (testo); viene proposto nell'ambito del progetto *GAMMA* alla *INRIA* francese; è un formato per mesh generiche.

```
MeshVersionFormatted 1
#
# Comments
#
Dimension
        3
        Vertices
        1224
        -.27789300000000000  -.748523999999999999e-0.1  -.334544999999999998e-0.1  0
        -.473765999999999997e-0.1  .590861999999999998e-0.1  -.221067999999999998  0
        ...
Triangles
        1620
        574 571 573 6
        572 573 571 6
        572 576 573 6
        572 575 576 6
        578 576 575 6
        ....
End
```

Attributo colore vertice

Attributo colore faccia

Figura 2.4: Esempio di file mesh .mesh

2.1.4 Mesh .obj

Può essere in binario, o in *ASCII* (testo); è un formato commerciale della *Alias-Wavefront* molto diffuso. Permette la memorizzazione di mesh generiche, ma anche curve e superfici.
Questo è il formato utilizzato dalla libreria *GLM* descritta nel capitolo 3.1.

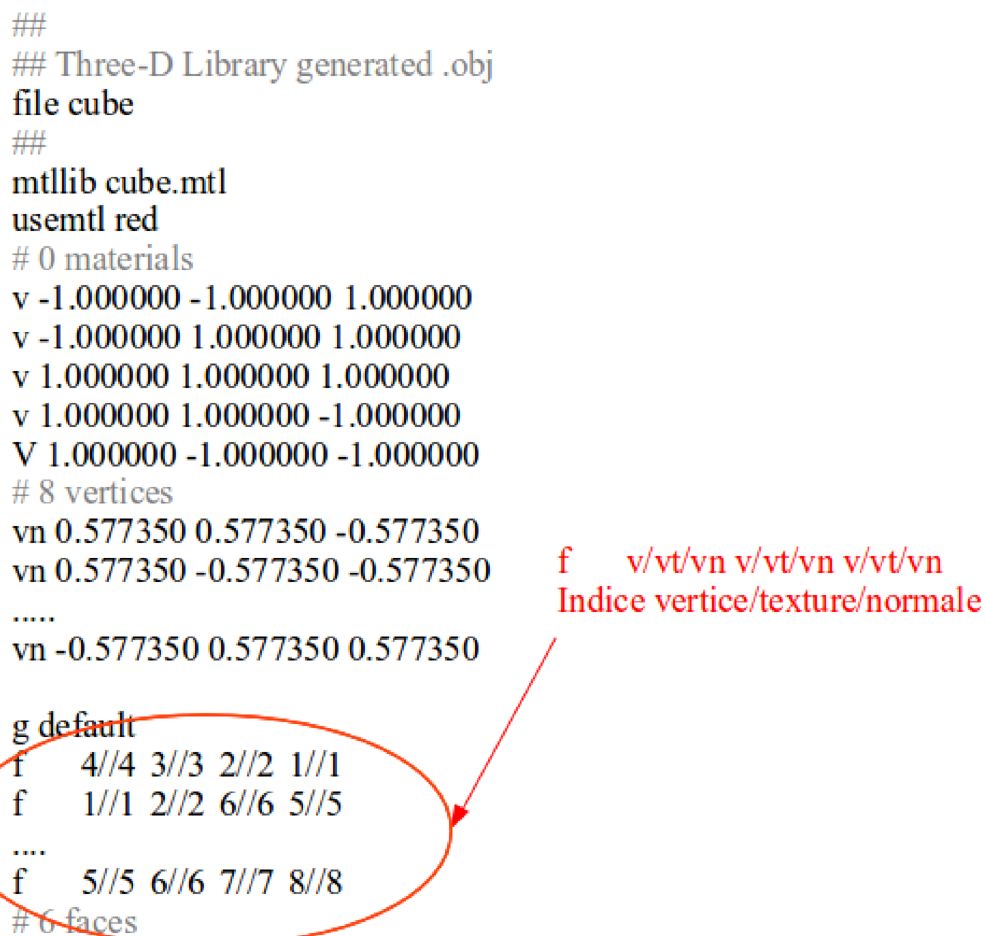


Figura 2.5: Esempio di file mesh .obj

2.1.5 Mesh .off

Puo’ essere in binario, o in *ASCII* (testo.)

N° vertici
N° lati
N° facce
4° vertice: Coord. X,Y,Z
Prima faccia:
Lati/vertici: 4
Indici: 3,2,1,0

```
OFF
5 3 12
0 0 0
3 0 0
3 1 0
1 1 0
1 5 0
4 3 2 1 0
4 0 4 3 1
4 6 2 8 7
```

Figura 2.6: Esempio di file mesh .off

2.2 Interagire con le mesh

Nel corso degli anni, numerosi programmi sono stati progettati *ad-hoc* per interagire e modificare mesh e oggetti 3D in scena.

2.2.1 Blender

L’ambiente di editing più famoso nel mondo Open Source è *Blender*, realizzato dalla *Blender Foundation* nei primi anni 2000; è multipiattaforma e vanta innumerevoli funzionalità, tipiche dei prodotti più commerciali: si passa dall’interazione con mesh in scena, alla gestione delle animazioni, come la cinematica inversa e la possibilità di realizzare applicazioni real time mediante il *Blender Game Engine* o per fare *raytracing*.

Per quanto riguarda l'editing 3D Blender permette due modalità: *Object Mode* ed *Edit Mode*.

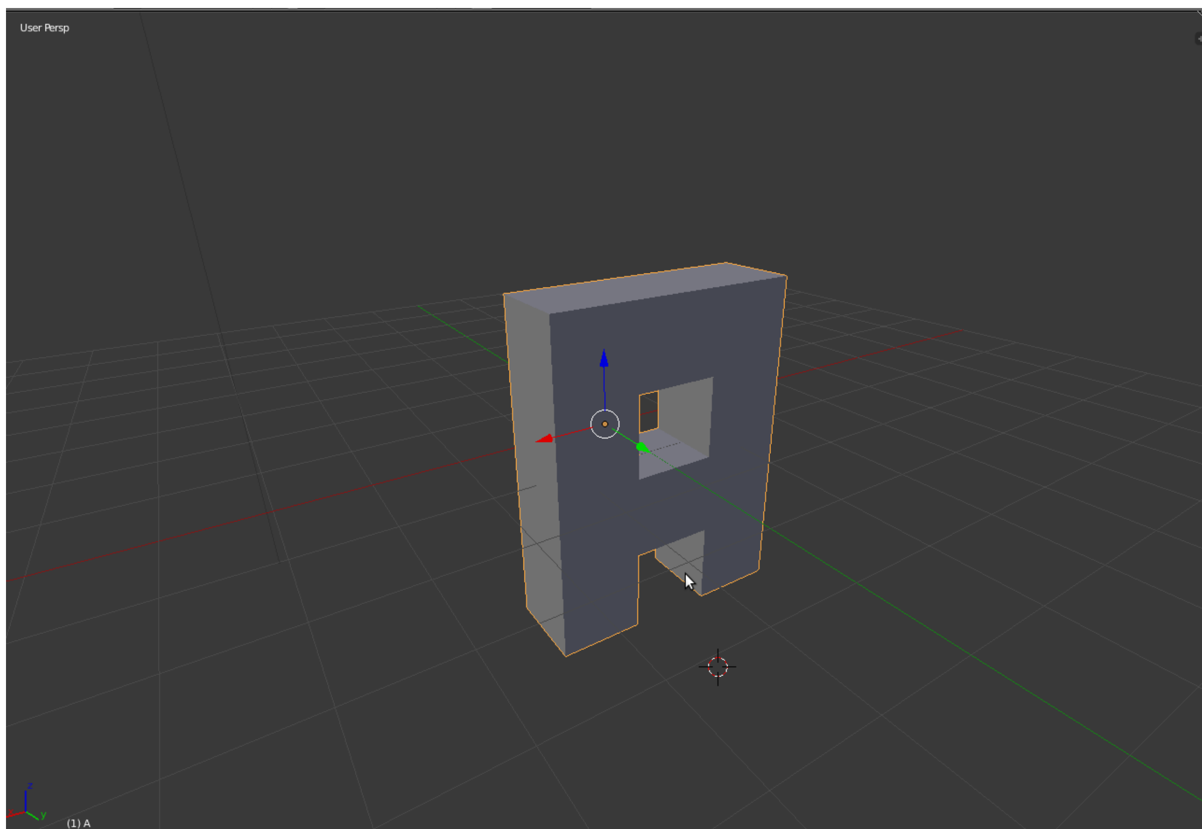


Figura 2.7: Modalità Object Mode di Blender

Object Mode Come suggerisce il nome, questa modalità aiuta l'utente a interagire con l'intera geometria in scena, permettendo di selezionare un solido piuttosto che un altro, se presente.

L'unico editing possibile è la traslazione dell'intero solido sui 3 assi per mezzo di 3 frecce colorate (X rosso, Y verde e Z blu) o in una posizione a piacere, trascinando l'oggetto dopo aver premuto il tasto destro del mouse.

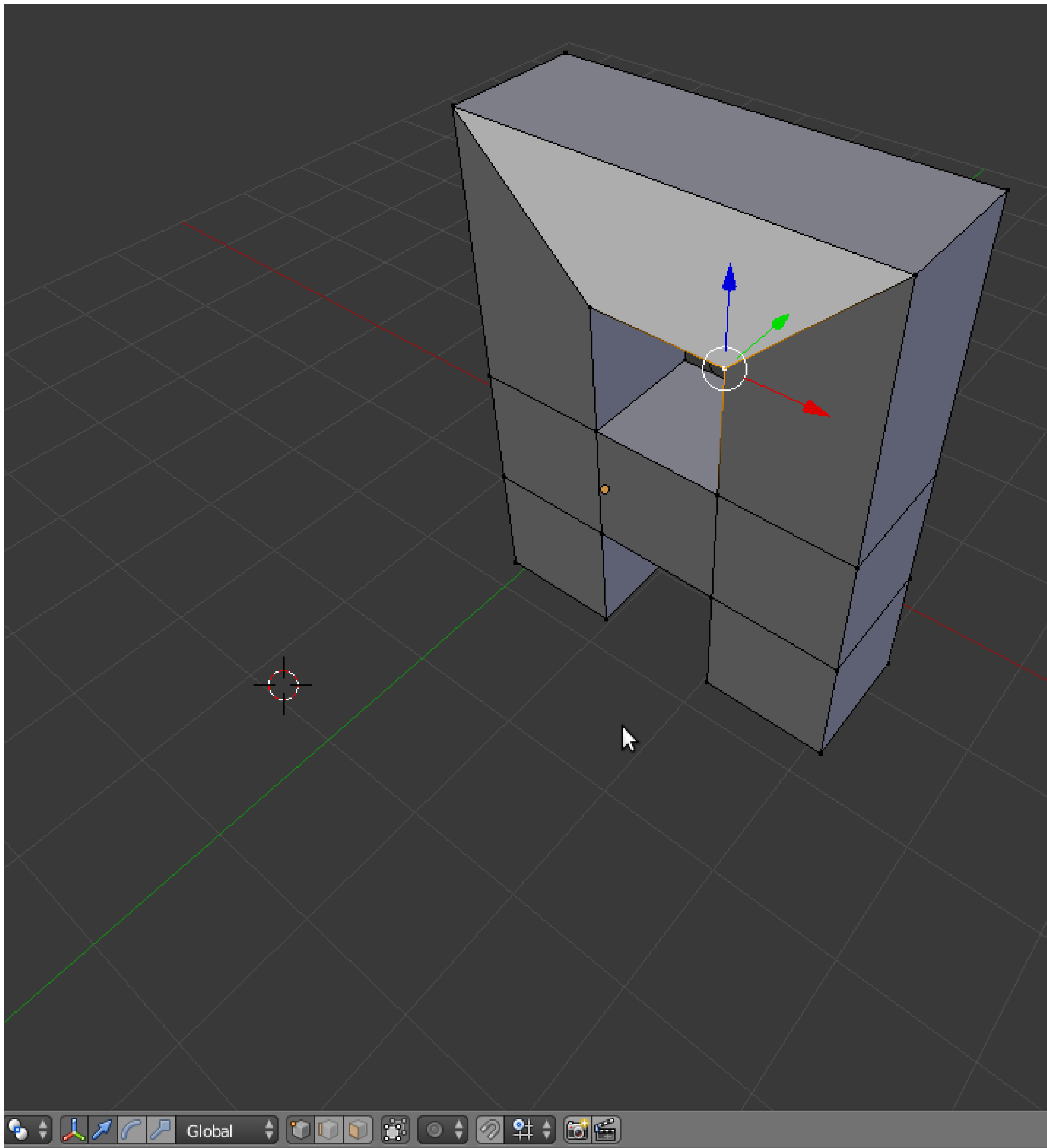


Figura 2.8: Modalità Edit Mode di Blender

Edit Mode Il vero approccio con le mesh si evince solo nella modalità '*Edit mode*', la quale fornisce un diverso set di selezione nell'interfaccia e evidenzia le diverse mesh poligonali nella scena.

Tre diverse tipologie di selezione sono lasciate all'utente: punti, lati e facce.

La selezione dei singoli elementi è sempre lasciata al tasto destro del mouse che, se si trova in un intorno di un oggetto della tipologia scelta, lo seleziona evidenziandone il punto preciso, il punto medio o il baricentro.
Il riferimento agli assi viene centrato nel punto scelto e permette all'utilizzatore di editare la mesh, traslando l'elemento lungo l'asse selezionato o in un punto dello spazio a piacere.
Blender in questa modalità offre anche la possibilità di ruotare e scalare singoli elementi: nella rotazione comparirà una *trackball* circoscritta all'elemento, mentre nel caso della scalatura, le frecce dei tre assi diventano quadrate.

Dalla versione 2.63, Blender ha introdotto le *bMesh*, un nuovo sistema di interazione con mesh n-poligonali, a differenza delle versioni precedenti più limitato a mesh triangolari o quadrate. Inoltre le strutture bMesh tengono conto delle adiacenze, informazione indispensabile al fine di migliorare le performance e il risparmio di memoria.

2.2.2 Maya

Nel mondo commerciale, *Maya* è senza dubbio il più completo e performante software per editing 3D.

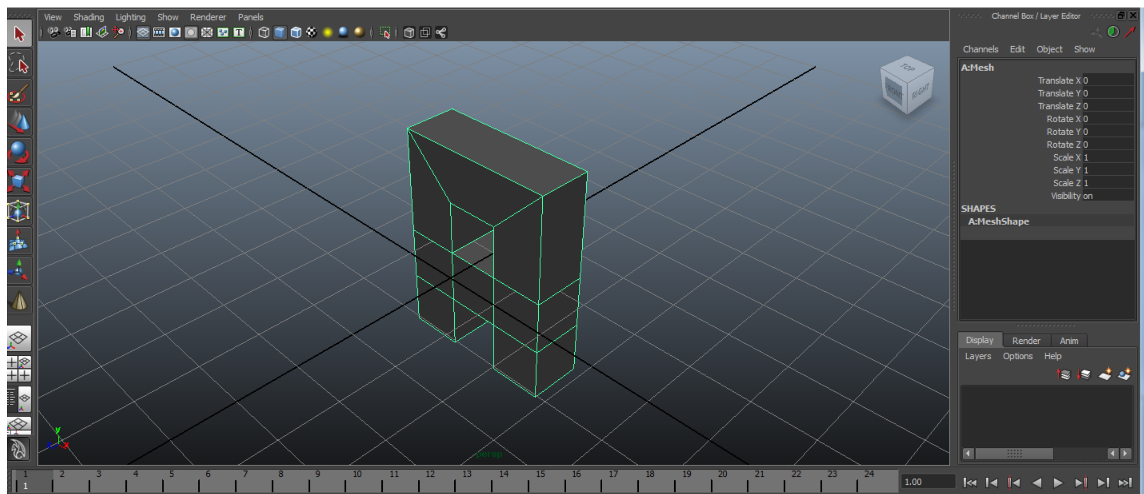


Figura 2.9: Maya User Interface

L'interfaccia, simile a quella vista per Blender, offre un vasto set di tool per interagire con le mesh in scena, previa selezione della modalità '*Polygons*'. Tenendo premuto il tasto destro del mouse, l'utente è invitato a scegliere una delle tipiche tipologie di elementi o una delle funzioni offerte mediante il menù a tendina.

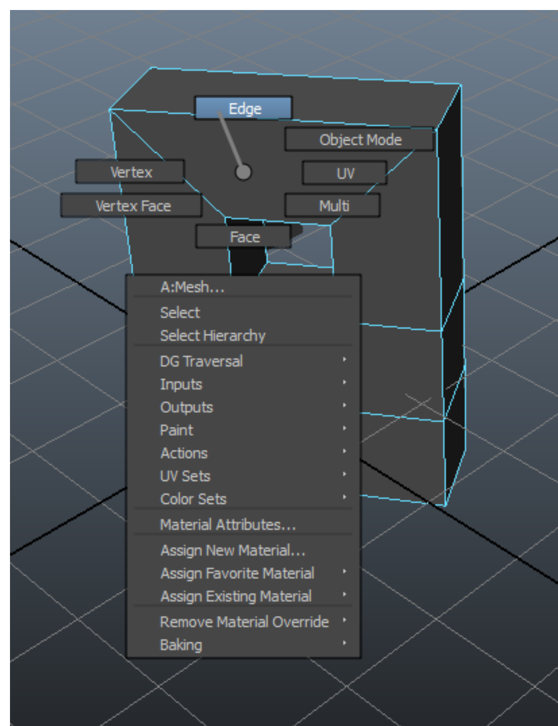
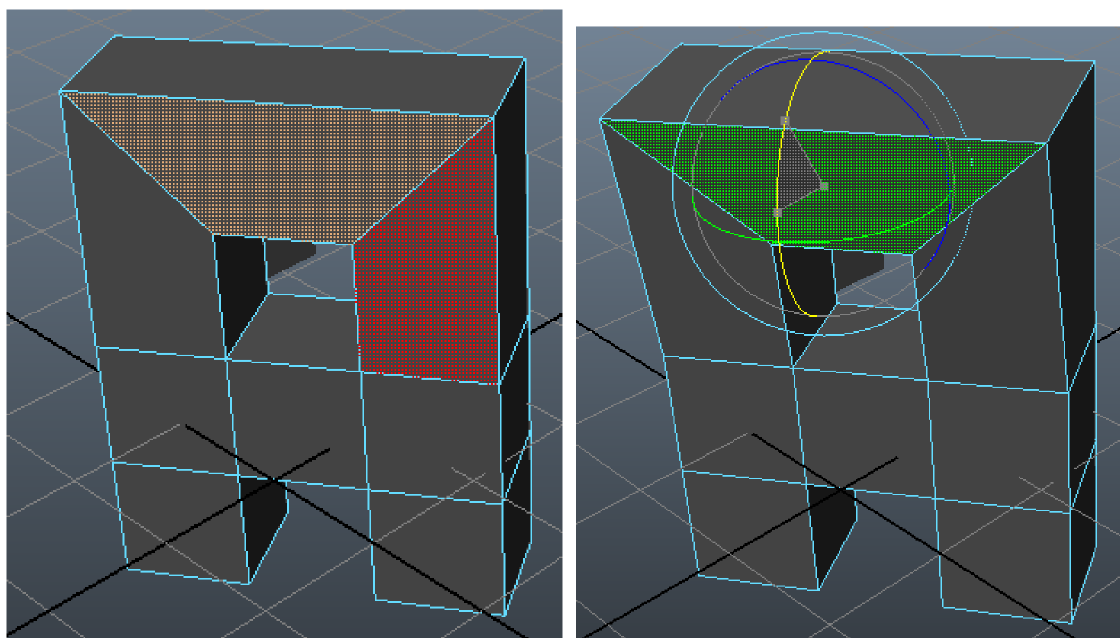


Figura 2.10: Maya - Menù di selezione



(a) Maya - Selezione delle facce (b) Maya - Rotazione delle facce

Figura 2.11: Maya - Selezione e rotazione

Nelle figure 2.11a/b è possibile vedere come Maya evidenzia la selezione di mesh e come viene presentata la possibilità di ruotare l'elemento selezionato (in questo caso una faccia) lungo i tre assi o a 360°; anche qui viene proposta una trackball con la quale l'utente interagisce.

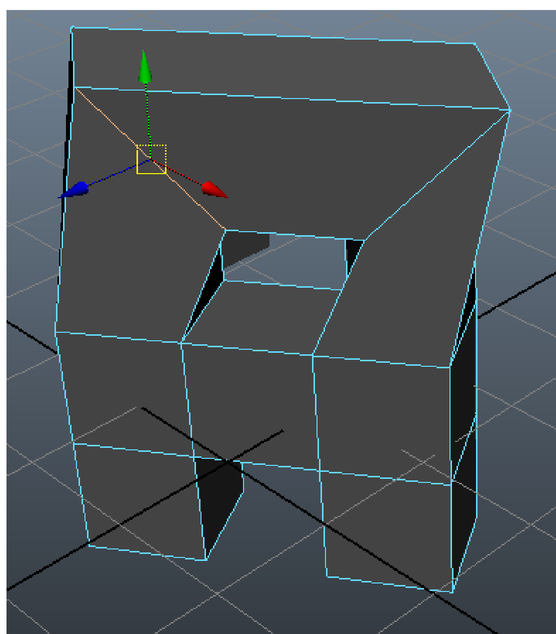
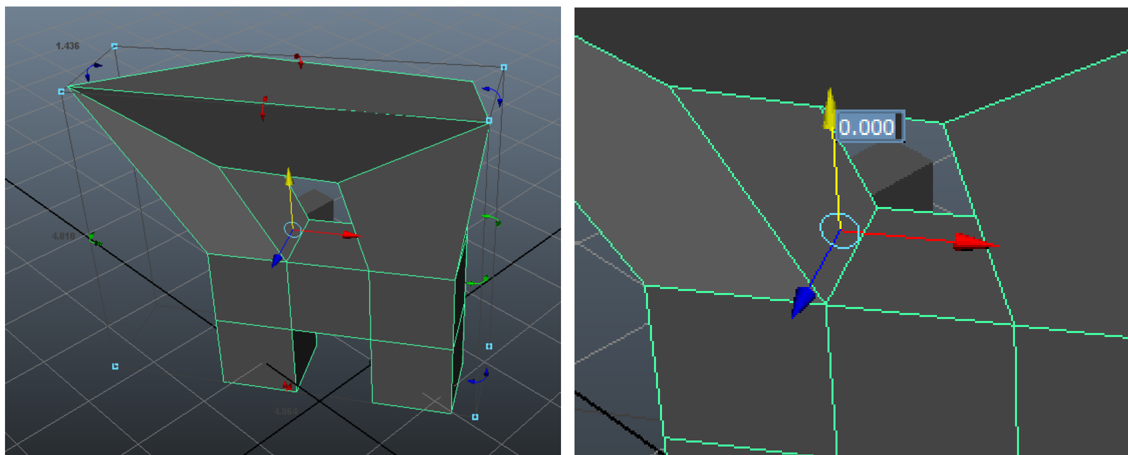


Figura 2.12: Maya - Traslazione di un lato

Immediata è anche la traslazione di elementi di mesh (punti/-lati/facce).

Con una logica molto simile a quella precedentemente descritta per Blender, Maya permette di modificare l'intera geometria, aggiornando di volta in volta le mesh durante l'editing.



(a) Maya - Universal manipulator (b) Maya - Spostamento offset

Figura 2.13: Maya - Universal manipulator

L'*Universal Manipulator* di Maya permette di modificare l'oggetto globalmente; combina le funzioni di *Move Tool, Rotate Tool e Scale Tool* in un'unica visione d'insieme.
Durante la trasformazione, a video viene visualizzato l'offset di spostamento o l'angolo di rotazione; inoltre viene data all'utente la possibilità di inserire valori precisi direttamente nella scena. (Fig. 2.13b).

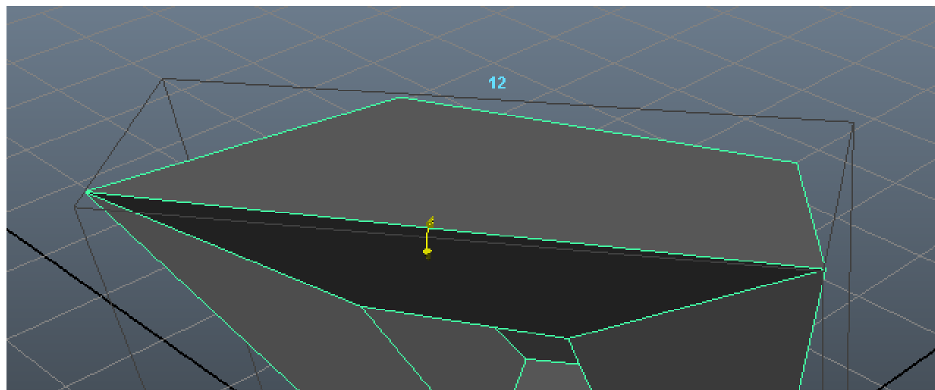


Figura 2.14: Maya - Rotazione di lati

La natura *Close-source* di Maya non ne permette un'analisi più approfondita e non è possibile verificare come la selezione di mesh sia stata implementata.

2.2.3 Rhino (Rhinoceros)

Rhinoceros, comunemente chiamato *Rhino*, è un'applicazione commerciale *Windows/Mac OSX* per l'elaborazione e editing di *NURBS* (Non Uniform Rational B-Splines).
Nonostante questa sua natura legata alle NURBS, Rhino permette di creare mesh a partire da esse, ma non è possibile riconvertire tutto in NURBS, una volta concluso l'editing.
Rhyno fornisce, quindi, un set limitato di operazioni come l'estrazione, il collassamento o la suddivisione di mesh.

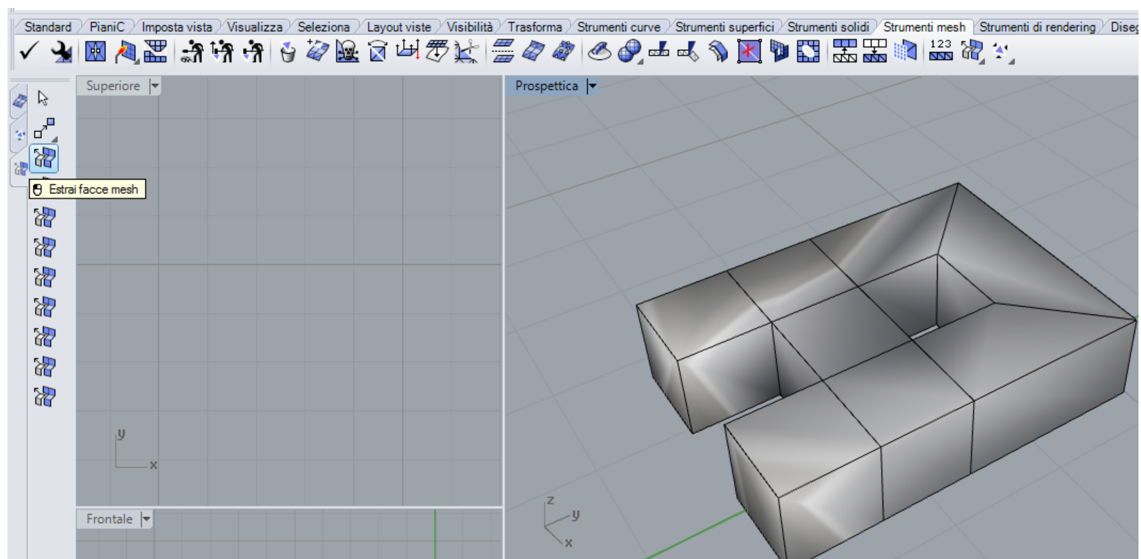


Figura 2.15: Rhino - User interface

2.2.4 MeshLab

Meshlab è un sistema openSource, portabile ed estendibile per la selezione e editing di mesh 3D triangolari non strutturate.
Sviluppato partendo da un progetto italiano dell'università di Pisa nel 2005, oggi è disponibile anche su mobile (Android e iOS).

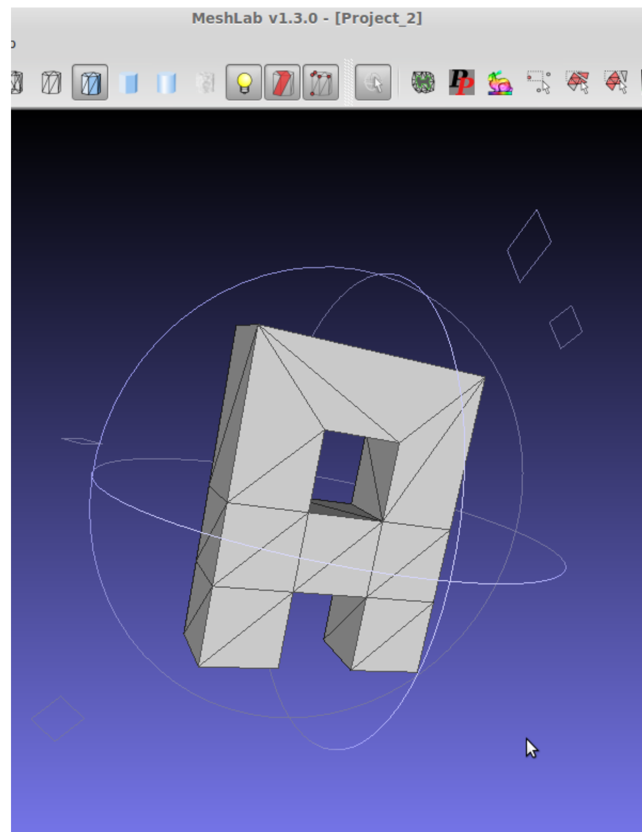


Figura 2.16: MeshLab - User interface

Nonostante offra all'utente una vastissima gamma di tool di editing su facce e vertici, MeshLab non presenta la possibilità di editing libero su vertici/lati/facce, come esposto per Blender o Maya.
La selezione degli elementi nella scena avviene tramite selezione ad area, evidenziando con un colore diverso gli oggetti compresi.

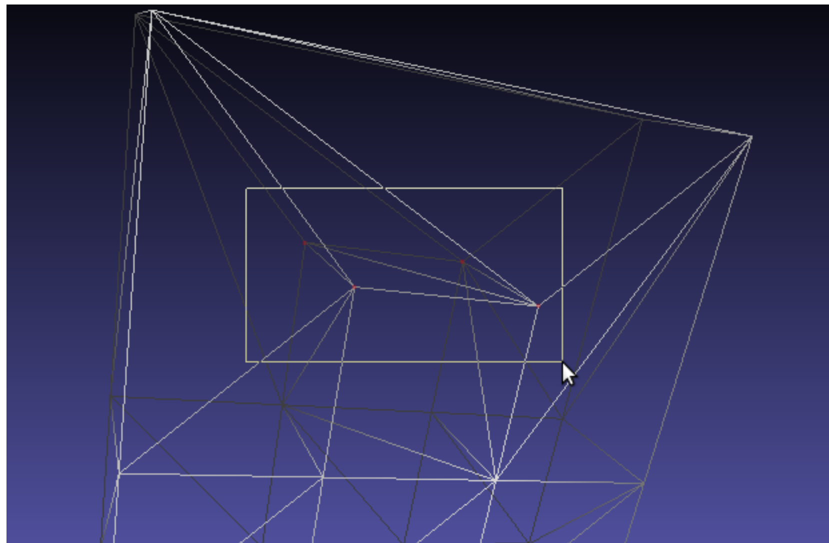


Figura 2.17: MeshLab - Selezione di punti

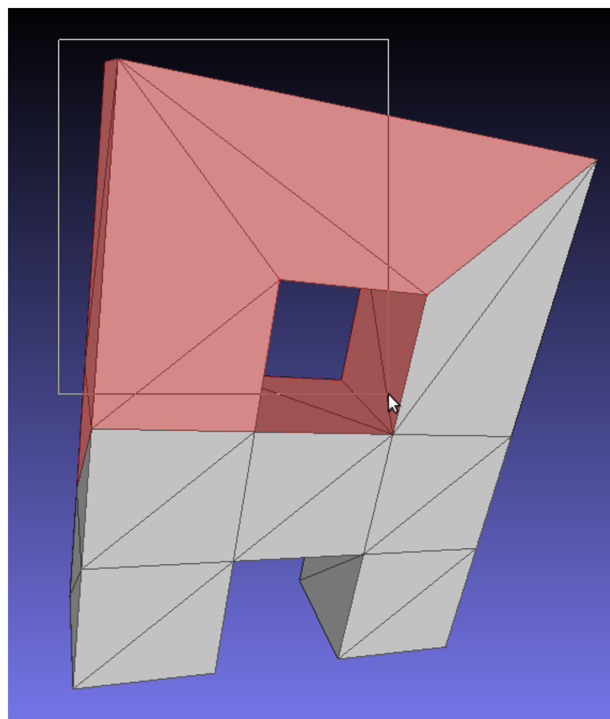


Figura 2.18: MeshLab - Selezioni di facce

2.3 Algoritmi di selezione

OpenGL è una **libreria di rendering della scena**, di conseguenza non offre costrutti e funzioni per interagire con gli elementi nello spazio.
La necessità di sfruttare OpenGL per l'editing ha però spinto la comunità a introdurre metodi e algoritmi, per bypassarne la natura di pura resa grafica.

2.3.1 GL_SELECT

Un primo approccio possibile a quello che viene chiamato '**picking**' o **selezione degli oggetti in scena**, fu permesso dalla *glRenderMode()*.
Questa utilizza tre possibili diversi parametri:

- GL_RENDER: opzione di rendering di default; è l'opzione che disegna gli oggetti nel frameBuffer.

- GL_FEEDBACK: permette di produrre informazioni su cosa si sta disegnando.

- GL_SELECT: ogni qualvolta si tenti di riprodurre in scena oggetti in questa modalità, questi vengono 'virtualmente' renderizzati, mentre alla *glRenderMode()* viene ritornato un contatore, pari al numero di primitive disegnate.
  La particolarità di GL_SELECT è che il contatore viene incrementato solo se l'oggetto è nella *viewport*. Restringendola a un pixel è quindi possibile verificare, tramite il contatore, se vi è un object nella coordinata X,Y che rappresenta temporaneamente la viewport di scena.
  Sfruttando quindi un *name stack*, cioè una struttura appositamente pensata per attribuire a ogni primitiva un identificativo univoco, è possibile iterare la selezione GL_SELECT e verificare se una primitiva è stata selezionata e quale essa sia.

GL_SELECT e GL_FEEDBACK sono oggi deprecati e non più utilizzabili dalla versione 3.0; questo è dovuto soprattutto a performance e problematiche legate all'architettura hardware: per anni la selezione via GL_SELECT ha creato seri problemi su schede video ATI, fino alla rimozione totale della modalità a partire dalle schede *Radeon x800*, a favore di selezioni software.
Anche il team NVIDIA ha tolto questa strategia nelle architetture *Fermi*, rendendola di fatto obsoleta.

2.3.2 Color index Picking

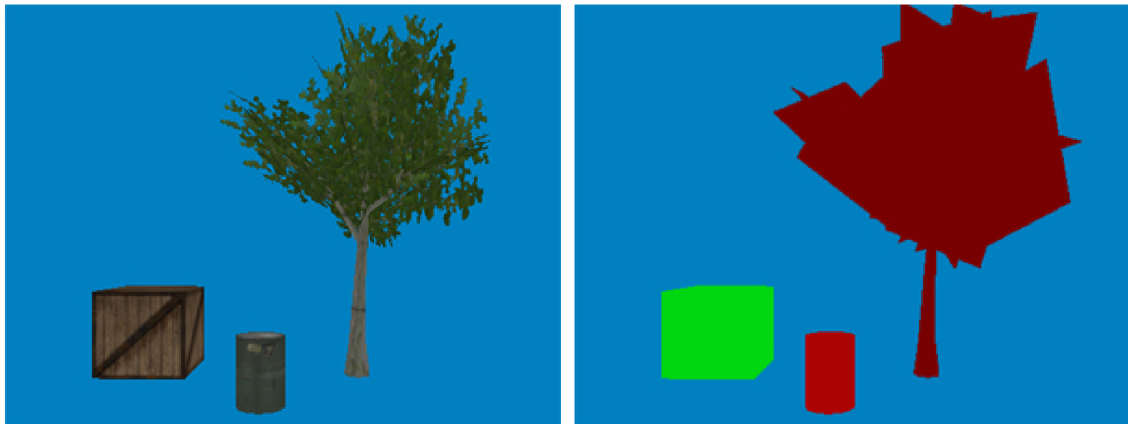
Una tecnica alternativa alla GL_SELECT è l'utilizzo del *backBuffer* e dei colori.
L'idea, che sta alla base di questa modalità, è quella di renderizzare la scena normalmente, ma attribuendo a ogni mesh (o elementi di mesh) un colore univoco, salvando l'informazione nel backBuffer. Quest'ultimo non viene scambiato con il frontBuffer, il quale contiene il vero colore del solido, così da renderizzarlo a video sempre il colore corretto.
Di base si utilizza uno stack simile a quello visto in precedenza, ma anzichè usare stringhe univoche, si utilizzano combinazioni *RGB*[2]. Viene quindi assegnato un intero univoco a punti, facce o lati[3].
Una volta associati i colori, è necessario recuperare quale coordinata X,Y è stata selezionata dall'utente, per poi leggere il backBuffer attraverso la chiamata

```
void glReadPixels(GLint x, GLint y, GLsizei width,
   GLsizei height, GLenum format, GLenum type, GLvoid *
   data);
```

Il puntatore '*data*' conterrà l'id del colore selezionato in esadecimale.

[2]Red Green Blue
[3]Con 1 byte per canale si possono rappresentare $255^3 = 16$ milioni di colori

Punto forte di questa modalità è la semplicità d'implementazione e l'indipendenza da librerie esterne o da complicati calcoli matematici; di contro fa un uso massiccio della GPU[4] e, senza un modello a supporto delle mesh, non è possibile avere più informazioni sul punto cliccato.



(a) Color picking - Scena iniziale (b) Applicazione color picking

Figura 2.19: Esempio di applicazione del color picking

2.3.3 Occlusion Queries

Se è necessario selezionare mesh o oggetti in una scena dinamica, la soluzione migliore è sfruttare le '*occlusion queries*'.
Le '*query object*' permettono all'applicazione di 'prelevare' dalla GPU certe informazioni della scena in modo asincrono.
Si chiamano Occlusion Queries l'insieme dei query object GL_SAMPLES_PASSED, GL_ANY_SAMPLES_PASSED e GL_ANY_SAMPLES_PASSED_CONSERVATIVE.

[4]Graphics Processing Unit

- GL_SAMPLES_PASSED: indica il numero di primitive che non falliscono il *depth-test* definito dalla query.
- GL_ANY_SAMPLES_PASSED: ritorna GL_FALSE se nessuno dei comandi di rendering, passati alla query, supera il *depth-test*, GL_TRUE altrimenti.
- GL_ANY_SAMPLES_PASSED_CONSERVATIVE: analogo al precedente, nettamente più veloce, ma meno accurato.

Nel paper *'Fast and Reliable Mouse Picking Using Graphics Hardware'* è stato mostrato come la combinazione del *RayPicking* con le *Occlusion Queries* è risultato da 2 a 14 volte più veloce dei trazionali metodi di picking basati solamente sulla GPU, permettendo una selezione ottimale anche tra 12 milioni di facce triangolari nella stessa scena.

2.3.4 Ray Picking

L'ultima tecnica esaminata, ma non in ordine di importanza, è quella del '*Ray picking*'. L'idea è quella di creare un raggio dal punto di vista dell'utente fino al punto cliccato in scena; successivamente si verifica se il raggio interseca un oggetto in scena, oppure no.

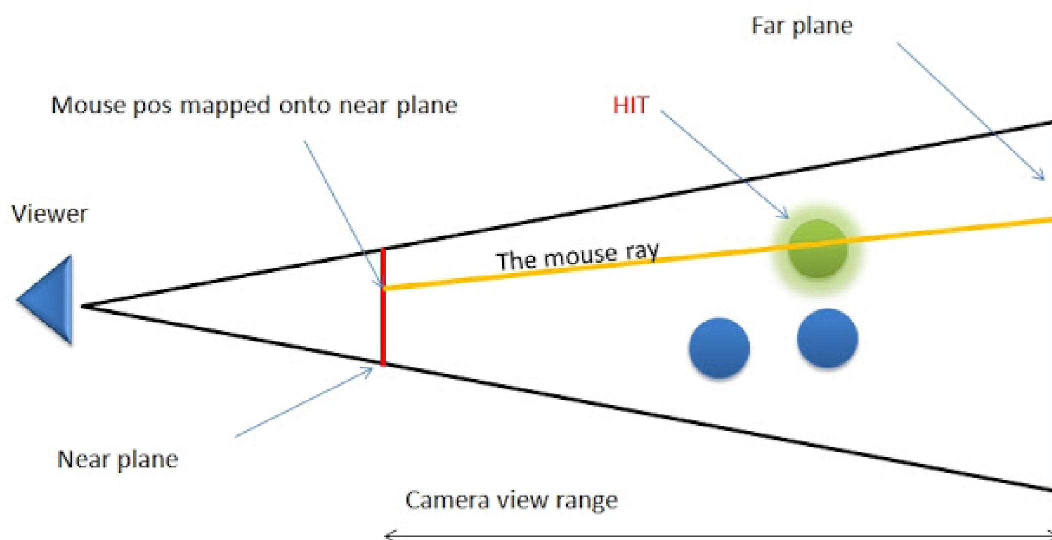


Figura 2.20: Ray Picking - Idea

Calcolare il punto 3D cliccato

Il primo problema da porsi è come calcolare quale punto della scena 3D è stato cliccato, se la viewport è in 2D. Dal punto di vista geometrico, un click del mouse corrisponde a un **raggio** che parte dall'osservatore verso il *far plane* di scena; vi sono quindi **infiniti punti**, tutti appartenenti alla stessa retta.
La libreria GLU permette di ricavare un punto 3D specificandone la componente Z (profondità), mediante la funzione

```
GLint gluUnProject(GLdouble winX, GLdouble winY, GLdouble
    winZ, const GLdouble *model,const GLdouble *proj, const
    GLint *view,  GLdouble *objX, GLdouble *objY, GLdouble *
    objZ);
```

- **winX** e **winY** corrispondono alla coordinata x,y del mouse. La coordinata y deve essere opportunatamente invertita, poichè l'asse Y di OpenGL ha la direzione opposta rispetto all'asse Y della window;
- **winZ** sarà opportunatamente calcolata con valore 0.0, se vogliamo calcolare la coordinata 3D sul near Plane, altrimenti 1.0 per la rispettiva coordinata sul far Plane;
- ***model**, ***proj**, ***view** rappresentano puntatori alle matrici ModelView, Projection e Viewport attualmente in scena;
- gluUnProject() salverà rispettivamente in ***objX**, ***objY**, ***objZ** la coordinata X,Y e Z del punto 3D calcolato.

Per calcolare le coordinate (**objX**, **objY**, **objZ**), *gluUnProject()* moltiplica le coordinate window normalizzate per la matrice inversa di (**model** * **proj**):

$$\begin{pmatrix} objX \\ objY \\ objZ \\ W \end{pmatrix} = INV(PM) \begin{pmatrix} \frac{2(winX - view[0])}{view[2]} - 1 \\ \frac{2(winY - view[1])}{view[3]} - 1 \\ 2(winZ) - 1 \\ 1 \end{pmatrix}$$

INV denota l'inversione di matrice, mentre W è una variabile - inutilizzata - di supporto.
Con due successive chiamate *gluUnProject()* sarà possibile ricavare due punti del raggio di vista, al fine di poterne definirne l'entità come differenza vettoriale.

Verificare l'intersezione con gli oggetti in scena

Definito il raggio, ora verifichiamo se questo interseca, o no, le primitive in scena. Esistono qui molteplici algoritmi chiamati *'Intersection Tests'*, al fine di verificare la collisione; il più immediato è l'intersezione raggio/sfera.

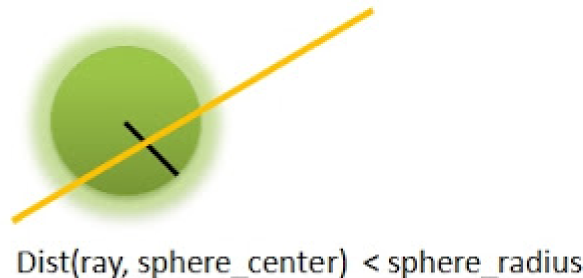


Figura 2.21: Ray Picking - Intersection test

L'idea è quella di inscrivere gli elementi che formano la mesh dentro dei *bounding-solid* come le sfere o dei cubi; gli algoritmi di bounding più avanzati fanno uso di *AABB* (Axis Aligned Bouding Boxes) o *OBB* (Oriented Bouding Boxes).
Nel caso di mesh non troppo complesse, le sfere rappresentano il compromesso complessità/performance migliore.
A questo punto il raggio ε della sfera rappresenta la tolleranza di errore nel test di collisione; mediante la formula di distanza punto/retta è possibile verificare se la distanza raggio/centroSfera è minore o maggiore di ε, cioè del raggio della sfera. Nel caso questa sia minore **vi è dunque intersezione**, altrimenti il test fallisce.

Nel caso vi siano più primitive colpite dallo stesso raggio, sarà necessario calcolarne la **distanza con l'osservatore**, individuando così la più vicina alla vista.

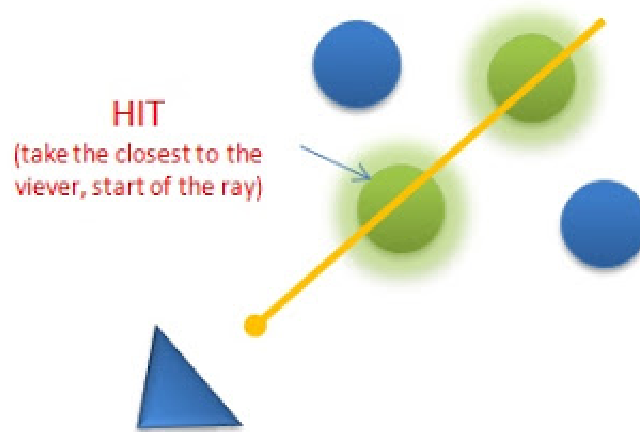


Figura 2.22: Ray Picking - Test di vicinanza

Iterando il seguente test su ogni primitiva in scena, è possibile verificarne la selezione.

Capitolo 3

Il progetto

Il progetto che questa tesi si pone come obiettivo, è lo sviluppo di una libreria OpenGL per la selezione e l'editing di mesh poligonali in una scena 3D.
Si è scelto di utilizzare OpenGL 2.0+ per sfruttare, al meglio, la sua compatibilità con la maggior parte delle applicazioni grafiche, oggi presenti.
Dopo un'attenta analisi delle tecniche precedentemente viste per la selezione di mesh, si è preferito utilizzare il **RayPicking**: l'utilizzo di algoritmi geometrici ne permette l'estendibilità e il miglioramento, senza modificarne la struttura logica.
La libreria prende il nome di **libEditMesh**.

3.1 La libreria Glm

Al fine di conoscere le geometria in scena, la *libEditMesh* fa uso e migliora la libreria Glm sviluppata da *Nate Robins*, ex dipendente di SGI.
La libGLM offre una vasta interfaccia per parsare e memorizzare mesh di tipo OBJ, generare *display list* per la visualizzazione, il calcolo delle normali e delle texture. Mediante la procedura

```
GLMmodel* glmReadOBJ(char* filename);
```

la Glm parsa il file .obj passato come parametro e inizializza una struttura chiamata *GLMModel*.

```
/* GLMmodel: Structure that defines a model. */
typedef struct _GLMmodel {
 char*    pathname;          /* path to this model */
 char*    mtllibname;        /* name of the material library */

 GLuint   numvertices;       /* number of vertices in model */
 GLfloat* vertices;          /* array of vertices  */

 GLuint   numnormals;        /* number of normals in model */
 GLfloat* normals;           /* array of normals */

 GLuint   numtexcoords;      /* number of texcoords in model */
 GLfloat* texcoords;         /* array of texture coordinates */

 GLuint   numfacetnorms;     /* number of facetnorms in model */
 GLfloat* facetnorms;        /* array of facetnorms */

 GLuint       numtriangles;  /* number of triangles in model */
 GLMtriangle* triangles;    /* array of triangles */

 GLuint       nummaterials;  /* number of materials in model */
 GLMmaterial* materials;     /* array of materials */

 GLuint       numgroups;     /* number of groups in model */
 GLMgroup*    groups;        /* linked list of groups */

 GLfloat position[3];        /* position of the model */

 GLuint   numedge;         /* NEW: number of edges in model */
 GLMedge* edges;          /* NEW: array of edges */

 GLuint numCentroid;       /* NEW: number of centroids in model */
 GLMcentroid* centroids;          /* NEW: array of centroids */

} GLMmodel;
```

GLMmodel contiene tutte le informazioni necessarie alla selezione/editing del solido in scena: inizialmente soltanto dei vertici/facce/normali, ma, durante il progetto, è stata affinata permettendo di inizializzare il vettore dei lati e dei baricentri. Sono, quindi, state introdotte 24 nuove funzioni ausiliarie al calcolo del modello, tra cui:

```
GLvoid initEdge(GLMmodel* model);

GLvoid insertEdge(GLMmodel* model, GLMtriangle*
   currentTriangle, GLMedge **edgeHead, GLint indexA, GLint
   indexB);

GLvoid middlePoint(GLMmodel* model);

GLvoid middleScale(GLMmodel* model, GLfloat scale);

GLvoid centroidScale(GLMmodel* model, GLfloat scale);

GLMedge* searchEdge(GLMmodel* model, GLuint index);

GLvoid deleteEdge(GLMmodel* model, GLint index);

GLvoid getTriangleFromPoint(GLMmodel* model, GLint index);

GLint getEdgeFromPoint(GLMmodel* model, GLint index);

GLvoid centroid3D(GLMmodel* model, GLMtriangle* pTriangle,
   GLMcentroid* pCentroid);

GLvoid initCentroid(GLMmodel* model);
```

InitCentroids() calcola, nel modello, i baricentri delle facce in scena, mentre *initEdge()* in tempo $O(n \log n)$ inizializza i lati dei poligoni: viene inizialmente allocata abbastanza memoria per gestire 3 * *numTriangoli* lati in tutto il modello, ma non sarà necessario utilizzarli tutti; infatti si noti come lo stesso lato può essere presente per più facce. E' necessario quindi verificarne la duplicità, evitando di inserire lo stesso lato due volte.

```
model->edges = (GLMedge*)calloc(model->numtriangles * 3,
                    sizeof(GLMedge));
memset(model->edges, 0, model->numtriangles * 3 * sizeof(
    GLMedge));
```

Quindi, dato il numero di edges per ogni poligono, si riempie la lista:

```
for(z=0; z<currentTriangle->nv;z++){
 int indexA=currentTriangle->vindices[(0 + z)%
    currentTriangle->nv];
 int indexB=currentTriangle->vindices[(1 + z)%
    currentTriangle->nv];

 //Dati i vertici A,B,C inserisco i (NV-1) lati nel modello
 insertEdge(model, currentTriangle, &model->edges, indexA,
    indexB);
}
```

InsertEdge() inserisce ogni lato nella lista *model->edges*, ordinandola basandosi man mano sul minor indice associato ad ogni vertice. Ad esempio il lato 3-5 (dove 3 e 5 sono gli indici dei vertici, nell'array degli indici del GLMmodel), verrà inserito precedentemente a 6-4, ma successivamente a 7-2.
La priorità nell'ordinamento permette una veloce scansione dell'array, ogni qual volta si tenti di trovare un duplicato: se, ad esempio, si vuole inserire il lato 6-3, basterà eseguire la scansione dei soli lati con indice minore 3; se vi si trova il 6 come indice maggiore, allora il lato è già presente nel modello.

InitCentroids() invece calcola per ogni faccia il proprio baricentro, salvandolo nell'array dei baricentri *model->centroids* e aggiungendo alla faccia un riferimento ad esso.

Le funzioni *getTriangleFromPoint()* e *getEdgeFromPoint()* accettano entrambe un intero corrispondente all'indice di un vertice; restituiscono rispettivamente tutte le facce e i lati in cui il vertice è presente, salvandole nelle liste dinamiche *pPointTriangles* e *pPointEdge*.

3.2 La libreria libEditMesh

Il cuore del progetto è la libreria **libEditMesh**.
Consta di 37 nuove funzioni al fine di poter selezionare, con più modalità diverse, punti/facce/lati del modello OBJ in scena, per modificarlo eventualmente.

La libreria si struttura in tre tipologie di funzioni, principalmente contenute in *editMesh*:

- Funzioni di **selezione** di elementi di mesh
- Funzioni di **disegno** di elementi di mesh
- Funzioni di **editing** di elementi di mesh

La libreria include anche:

- La libreria Glm,
- la libreria Gltb per la gestione della trackball
- geoUtils, contenente gli '*Intersection test*'

Di seguito le più importanti, il cui funzionamento è illustrato nei paragrafi successivi:

```
GLvoid setSelectionMode(GLint mode);
GLvoid changeSelectionMode();
GLvoid setDeletingMode(GLboolean mode);
GLvoid setSelectionElements(GLboolean mode);
GLvoid meshSelect(GLMmodel* model, GLint mouseX, GLint
   mouseY, GLboolean ctrlClicked);

GLvoid selectPoint(GLMmodel* model, GLint numPoint, GLint
   type,  GLboolean ctrlClicked);

GLvoid changeFaceSelection(GLboolean mode);
GLvoid changeEdgeSelection(GLboolean mode);
GLvoid drawSelectionPoint(Vector3d point, GLboolean
   withArrow);
GLvoid drawSelectionFace(GLMmodel* model, GLint faceIndex,
   GLboolean color);

GLvoid drawSingleEdge(GLMmodel* model, GLint indexP1, GLint
   indexP2);

GLvoid editMesh(GLMmodel* model, GLint mouseX, GLint mouseY,
    GLvoid reDisplay());

GLvoid editPoint(GLMmodel* model, GLint index, GLfloat
   offset);
GLvoid editFace(GLMmodel* model, GLint index_offset, GLfloat
    offset, GLfloat centroid_offset);

GLvoid editEdge(GLMmodel* model, GLint index_offset, GLfloat
    offset, GLfloat middle_offset);

GLvoid deleteFace(GLMmodel* model, GLboolean ctrlClicked,
   GLvoid reDisplay());
```

3.2.1 Funzioni di selezione

Una volta che l'utente ha inizializzato il modello GLMmodel, ha la possibilità di selezionare punti, facce o lati delle mesh visualizzate in modalità **Wireframe**.
La modalità può essere impostata chiamando *setSelectionMode(GLint mode)* che accetta come valori:

- NO_SELECTION
- SELECTION_POINTS
- SELECTION_FACES
- SELECTION_LINES

ChangeSelectionMode() invece cambia elemento di selezione, 'ciclando' automaticamente sulle quattro disponibili.
Impostata la modalità, è necessario ora inizializzare il calcolo appropriato dell'algoritmo:

```
GLvoid meshSelect(GLMmodel* model, GLint mouseX, GLint
    mouseY, GLboolean buttonClicked)
```

- **model** è il puntatore al GLMmodel in scena;
- **mouseX** è la coordinata X del punto cliccato nella viewport;
- **mouseY** è la coordinata Y del punto cliccato nella viewport;
- il flag **buttonClicked** indica la permanenza della selezione, evidenziandola tramite il posizionamento del sistema di editing sulle coordinate specificate.

MeshSelect() inizialmente converte il punto cliccato da 2D a coordinate mondo 3D, come precedentemente illustrato nel paragrafo 2.3.4 .

Selezione di punti su mesh

Calcolato il punto 3D, l'algoritmo continua con la funzione:

```
selectPoint(model, model->numvertices, POINT, buttonClicked);
```

a cui viene passato il modello, il numero totale di punti da analizzare, la tipologia di punto da selezionare[1] e la necessità di evidenziare l'elemento.
SelectPoint() è l'esatta implementazione dell'algoritmo *RayPicking* (descritto in 2.3.4), appositamente adattato per selezionare punti mesh, punti medi o baricentri.
L'algoritmo itera, su ogni set di punti (2° parametro), il calcolo geometrico *RayTest()* che, a sua volta, calcola il punto più vicino tra il raggio di vista e il punto considerato con la funzione:

```
Vector3d closest = ClosestPoint(start, end, center);
```

dove '**center**' è il centro della sfera costruita intorno al vertice.
Nel caso *RayTest()* risulti vero per più di un punto[2], viene rilevato il punto più vicino all'osservatore.

Se **buttonClicked** è GL_TRUE[3], la funzione specificherà in apposite strutture globali, l'indice del vertice selezionato. Successivamente la funzione di disegno provvederà ad evidenziarlo, se precedentemente salvato.

[1] Possibili scelte: POINT, MIDDLE(POINT) o CENTROID
[2] Nel caso in cui più punti siano allineati
[3] True booleano

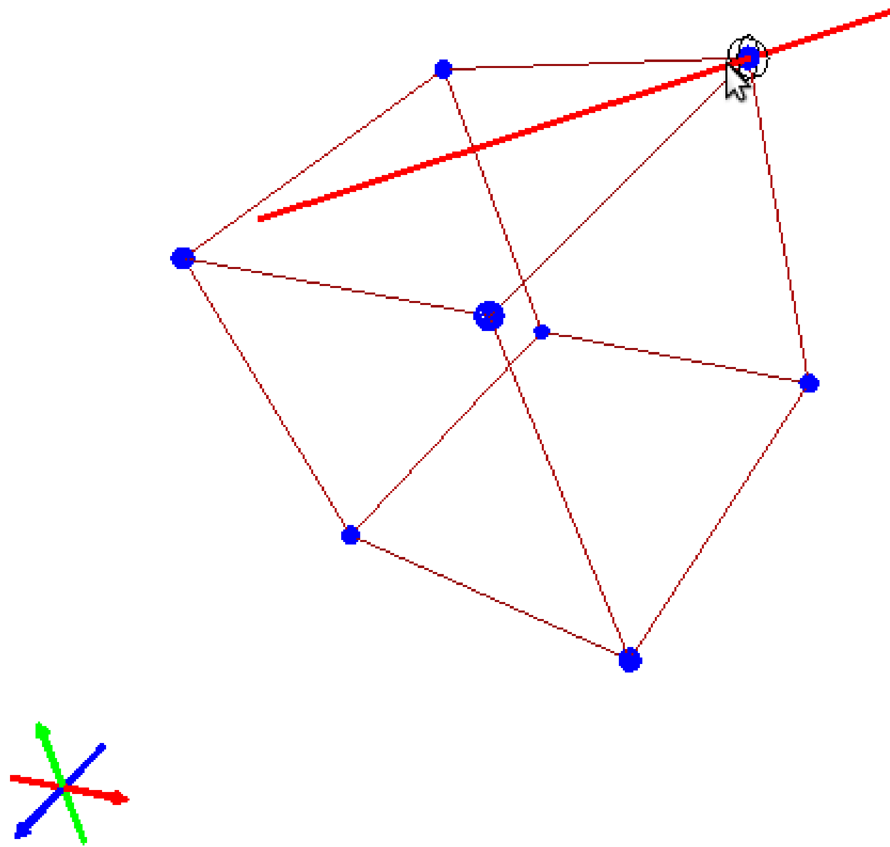


Figura 3.1: libEditMesh - Selezione di un punto[4]

[4]buttonClicked è GL_FALSE e il segmento rosso rappresenta la vista dopo la rotazione della camera

Selezione di facce su mesh

LibEditMesh offre due algoritmi per il calcolo delle facce selezionate:

- Via intersezione **Ray/Triangle**: viene verificata l'intersezione del raggio di vista con tutti i triangoli[5] presenti in scena e viene evidenziato quello più vicino alla vista.

- Via **baricentri**: metodo più performante, ma meno accurato; permette la selezione di facce in secondo piano, se verificata l'intersezione tra il raggio di vista e un baricentro, purchè visibile.

Tramite la funzione *changeFaceSelection()* è possibile commutare tra i due metodi appena illustrati; di default è attiva la selezione tramite intersezione Ray/Triangle. La selezione per baricentri è fornita sempre dalla funzione

```
selectPoint(model,model->numCentroid,CENTROID,buttonClicked)
```

opportunatamente configurata per selezionare baricentri. La selezione mediante intersezione Ray/Triangle ha una procedura più complessa:
inizialmente ogni faccia mesh considerata viene suddivisa in *(numeroVertici - 2)* triangoli.

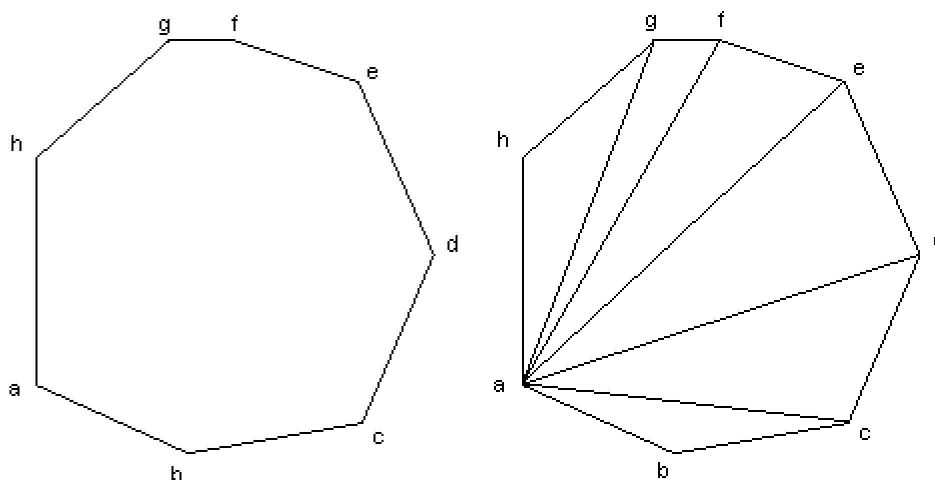


Figura 3.2: libEditMesh - Selezione di un punto[6]

[5]Nel caso di mesh poligonali, questi vengono suddivisi in triangoli

Successivamente, per ogni triangolo, viene calcolato

```
GLint intersect3D_RayTriangle(Vector3d R_P0, Vector3d R_P1,
    Vector3d T_V0, Vector3d T_V1, Vector3d T_V2 , Vector3d* I)
```

il quale verifica se vi è stata incidenza tra il raggio di vista e il triangolo.
Ci sono numerosi modi per verificare questo test, come gli algoritmi [*Badouel*, 1990] e [*O'Rourke*, 1998] che proiettano il punto e il triangolo nel piano di coordinate 2D per verificarne l'intersezione.
L'algoritmo usato [*Dan Sunday*, 2012] sfrutta direttamente le informazioni nei piani 3D e il test di intersezione tra raggio e piano.
L'equazione parametrica del piano è data da:

$$V(s,t) = V_0 + s(V_1 - V_0) + t(V_2 - V_0) = V_0 + su + tv$$

dove s e t sono numeri reali, $u = V_2 - V_0$ e $v = V_2 - V_0$ sono vettori dei lati del triangolo T.
Infine il punto $P = V(s,t)$ è interno al triangolo T se $s \geq 0$, $t \geq 0$ e $s + t \leq 1$.

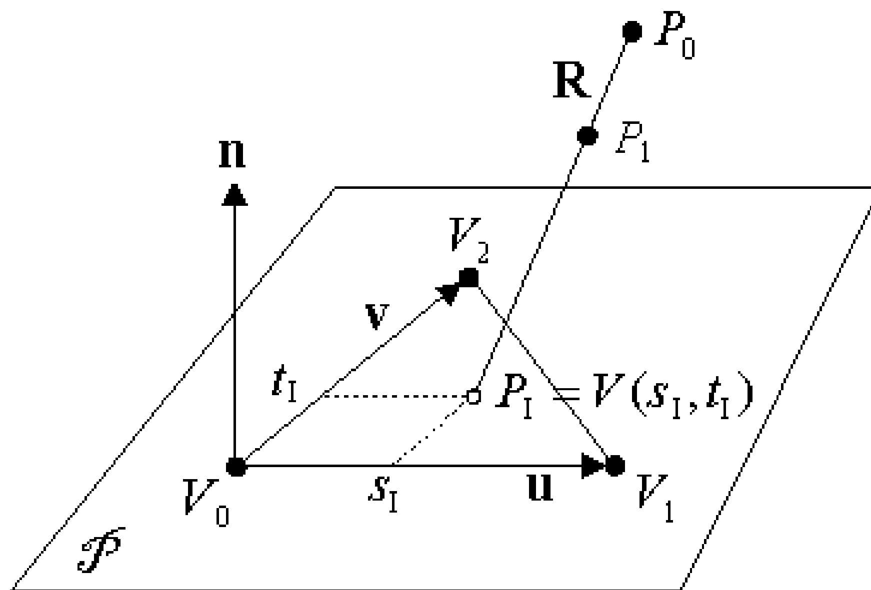


Figura 3.3: Intersection Test: Ray/Triangle

Come spiegato per la selezione dei punti, anche in questo caso viene calcolata la faccia più vicina alla vista, se vi sono stati più triangoli a superare il test di intersezione. Intersect3D_RayTriangle() memorizza nel *Vector3d *I* le coordinate 3D del punto di intersezione tra il Ray e il triangolo.

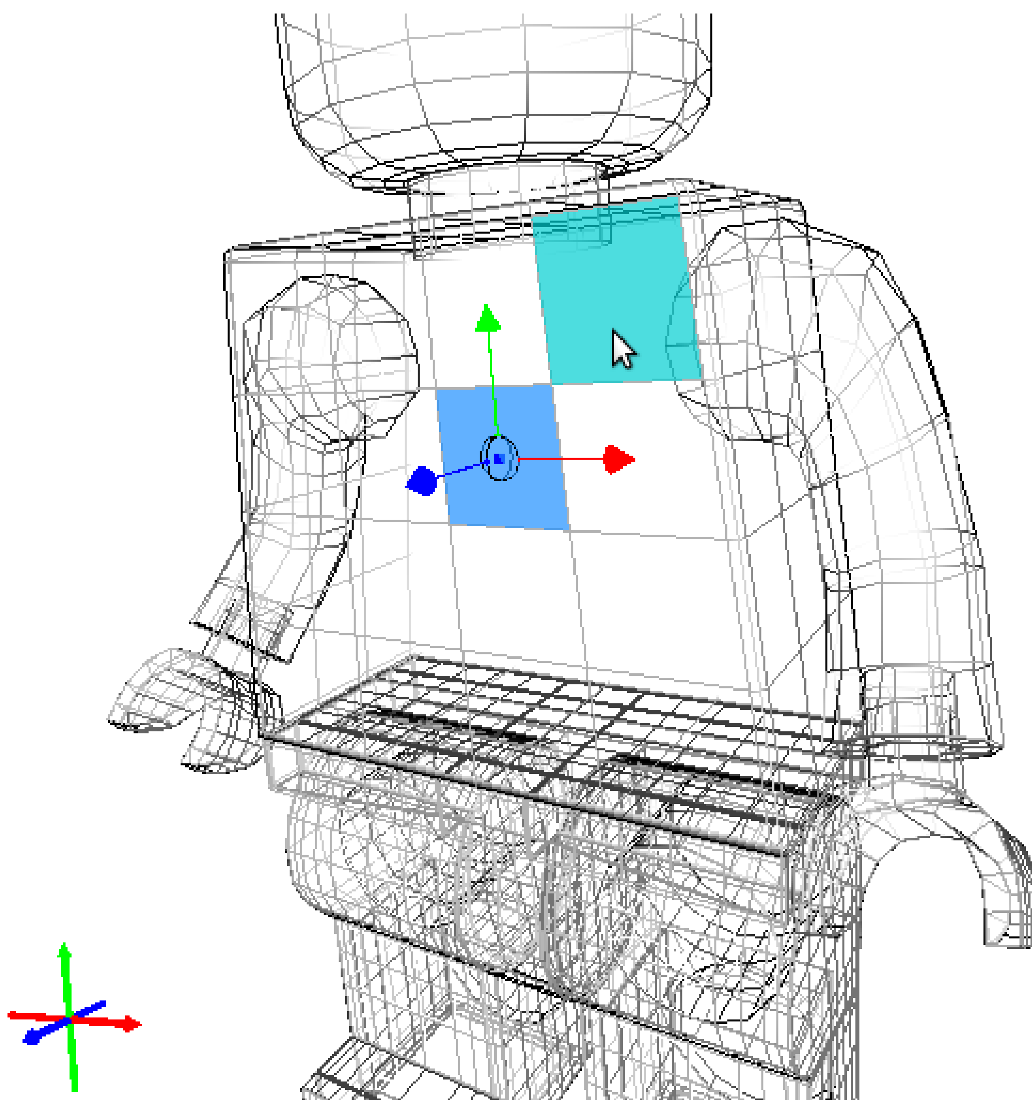


Figura 3.4: Selezione permanente (Blu scuro) e temporanea (Azzurro) di una faccia

Selezione di lati su mesh

LibEditMesh offre due algoritmi per il calcolo dei lati selezionati:

- Via intersezione **Ray/Triangle** e **calcolo della distanza con i punti medi**: viene verificata l'intersezione del raggio di vista con tutti i triangoli; successivamente, calcolato il punto esatto nel triangolo, si verifica a quale punto medio questo è più vicino.

- Via **punti medi**: metodo più performante, ma meno accurato; permette la selezione di lati in secondo piano, se verificata l'intersezione tra il raggio di vista e un punto medio, purchè visibile.

Analogamente alla selezione di facce, la funzione *changeEdgeSelection()* alterna la selezione dei lati; se non viene modificato, verrà utilizzato il primo metodo.
La procedura mediante i punti medi viene calcolata con:

```
selectPoint(model, model->numedge, MIDDLE, buttonClicked);
```

analogamente a punti e facce.
Il primo metodo sfrutta l'algoritmo visto precedentemente per le facce; calcolato l'esatto punto P di intersezione tra Ray e il triangolo, la funzione

```
GLMedge* searchEdge(GLMmodel* model, GLuint index);
```

inizializza un array temporaneo, contenente i lati della faccia selezionata.
A questo punto si identifica la distanza minore tra P e i punti medi individuati, evidenziando il rispettivo lato.

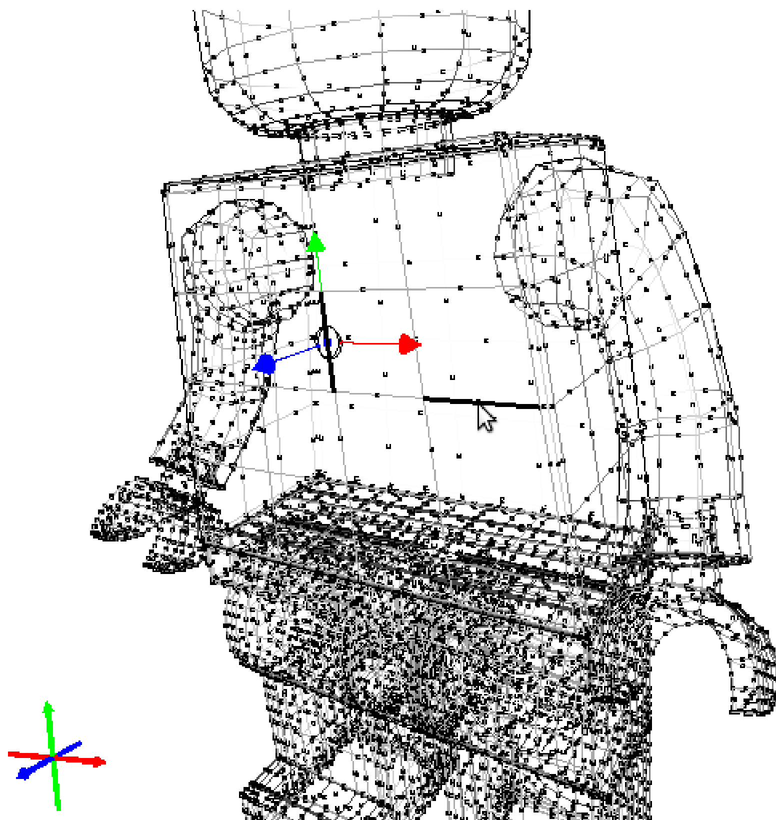


Figura 3.5: Selezione permanente e temporanea di una lato mediante punti medi

3.2.2 Funzioni di disegno

La libreria fornisce tre diverse funzioni di disegno, una per ogni tipologia di primitiva.

```
GLvoid drawPoints(GLMmodel* model);
GLvoid drawTriangle(GLMmodel* model);
GLvoid drawEdges(GLMmodel* model);
```

Si capisce facilmente come queste tre funzioni disegnino in scena il punto selezionato, la faccia selezionata o il lato selezionato.

NOTA: Prima di utilizzare una delle tre funzioni si consiglia di verificare che la scena sia renderizzata in modalità *wireframe*.

- **drawPoints()** verifica che la modalità sia SELECTION_POINTS; successivamente disegna in scena un doppio anello nero, per evidenziare l'eventuale punto mesh selezionato; inoltre, se è stato selezionato permanentemente un punto con *meshSelect()*, drawPoints() visualizza i riferimenti ai tre assi sul vertice appropriato. Infine mediante *getSelectionElements()* si mostrano a video le sfere di debug, utilizzate nell'algoritmo di RayPicking Ray/Sphere.

- **drawTriangle()** verifica che la modalità sia SELECTION_FACES e che sia attiva una delle due diverse modalità di selezione delle facce; successivamente verifica se sia stata richiesta la visualizzazione dei baricentri in scena, mostrati attraverso dei vertici neri; infine viene evidenziata di azzurro la faccia selezionata temporaneamente al passaggio del mouse, mentre di blu quella selezionata permanentemente da *meshSelect()*. Anche qui troviamo i riferimenti degli assi cartesiani, centrati nel baricentro della faccia, in preparazione dell'editing del solido.

- **drawEdges()** verifica che la modalità sia SELECTION_LINES. In un secondo momento disegna in scena i punti medi, similmente a quanto visto per i baricentri. Viene visualizzato l'eventuale lato selezionato e il lato scelto per l'editing del solido, con il riferimento agli assi nel suo punto medio.

3.2.3 Funzioni di editing

LibEditMesh non si limita alla semplice selezione degli elementi mesh in scena, ma ne permette la modifica, aggiornando di volta in volta il modello GLM.
Una volta modificato, la libreria GLM permette di salvare un nuovo file OBJ mediante la chiamata *glmWriteOBJ()*.
Le versione attuale della libreria permette due operazioni:

- **Traslazione** di punti/facce/lati
- **Eliminazione** di facce

La funzione chiave della traslazione è:

```
GLvoid editMesh(GLMmodel* model, GLint mouseX, GLint
    mouseY, GLvoid reDisplay()
```

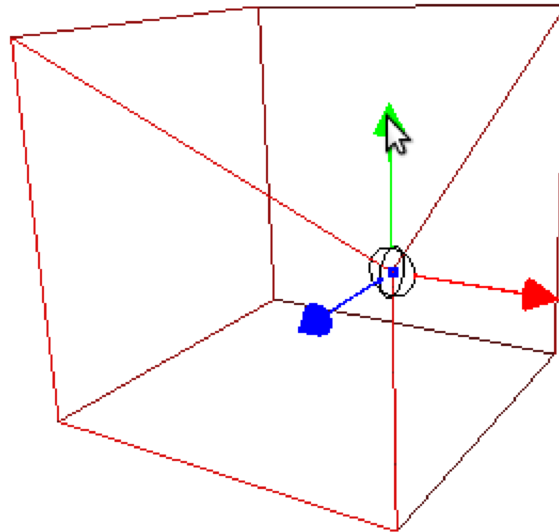
Dopo aver verificato che l'utente stia trascinando uno dei tre assi posizionati su un vertice, su un punto medio o su un baricentro, viene calcolata l'intersezione retta/retta tra X/Y/Z con il raggio di vista.
E' importante notare come l'intersezione tra due rette nello spazio 3D non sia facilmente calcolabile come nel piano 2D: l'introduzione di un terzo piano[7] rende impossibile l'utilizzo del classico teorema di Pitagora.
LineLineIntersect() risolve il problema calcolando la distanza tra il raggio di vista (*nearPoint* - *farPoint*) e la retta lungo l'asse desiderato: se questa risulta essere minore di un dato ε, *LineLineIntersect()* salverà in memoria[8] il *Vector3d*, proiezione del raggio di vista sull'asse.

[7] Il piano Z
[8] Parametro *pa di LineLineIntersect()

In questa fase il punto proiettato è l'*offset* di traslazione della procedura di editing; *editMesh()* procederà quindi con:

- **editPoint()** aggiorna la componente x, y o z del punto selezionato, aggiungendo l'offset passato come argomento. Successivamente verranno aggiornati i baricentri e i punti medi delle facce e dei lati, adiacenti.

```
pLastPointClicked[index] = offset - 0.6;
updateCentroid(model, lastPointIndexClicked);
updateMiddle(model, lastPointIndexClicked);
```



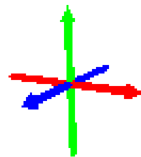
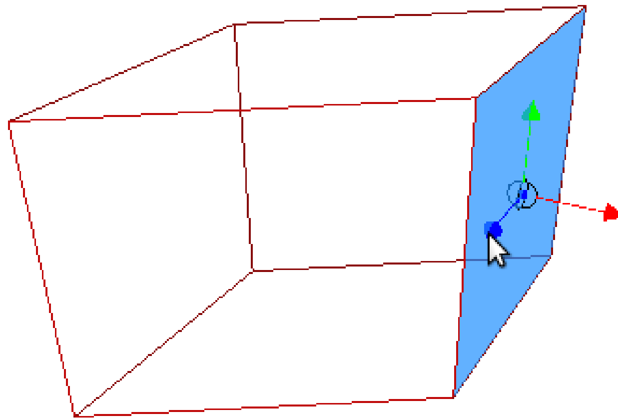


Figura 3.6: Traslazione di un punto lungo l'asse Y

- **editFace()** recupera la faccia selezionata e aggiunge l'offset a tutti i suoi vertici. Per ogni vertice, vengono aggiornati i punti medi e i baricentri vicini.

```
GLMtriangle *pTriangle = &model->triangles[
   lastFaceIndexClicked];
for(i=0; i<pTriangle->nv; i++){
 GLint index = pTriangle->vindices[i];
 model->vertices[index*3+index_offset] += offset -
    centroid_offset - 0.6;
 updateCentroid(model, index);
 updateMiddle(model, index);
}
```



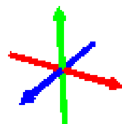


Figura 3.7: Traslazione di una faccia lungo l'asse Z

- Analogamente **editEdge()** modifica le informazioni sui due vertici del lato, aggiornando baricentri e punti medi adiacenti a entrambi i punti.

```
model->vertices[indexA * 3 + index_offset] += offset
    - middle_offset - 0.6;
model->vertices[indexB * 3 + index_offset] += offset
    - middle_offset - 0.6;
// Aggiorno i baricentri adiacenti ai 2 punti
updateCentroid(model, indexA);
updateCentroid(model, indexB);
// Aggiorno i punti medi adiacenti ai 2 punti
updateMiddle(model, indexA);
updateMiddle(model, indexB);
```

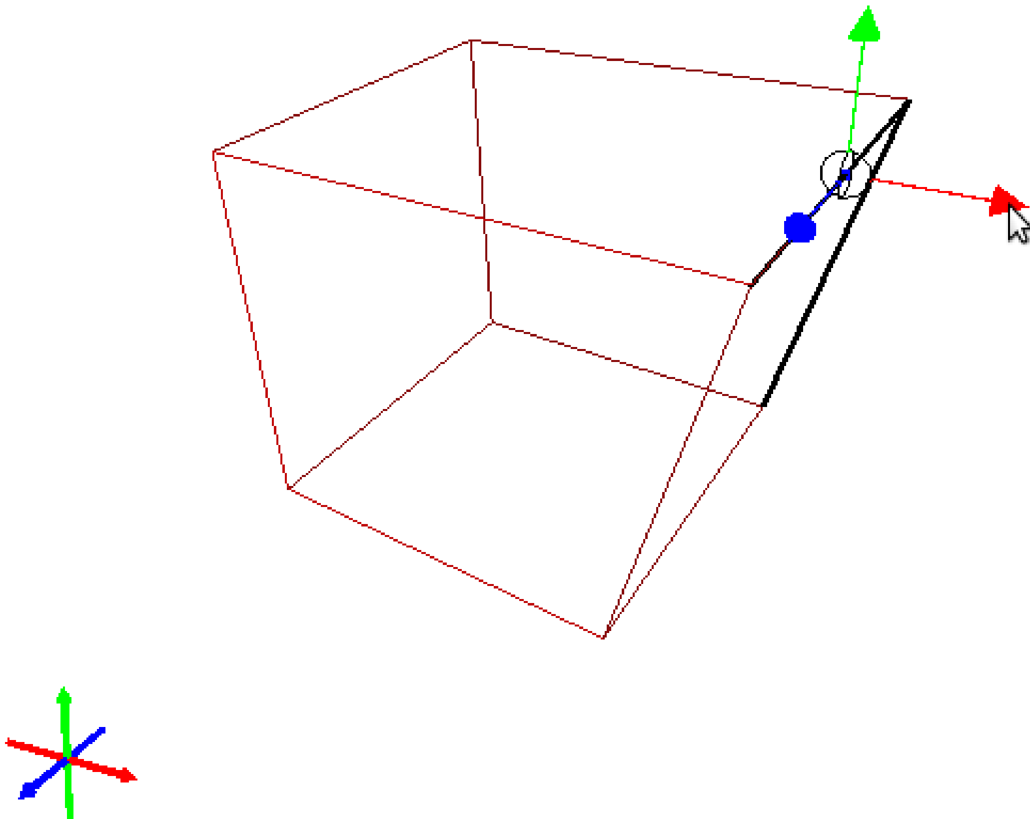


Figura 3.8: Traslazione di un lato lungo l'asse X

Aggiornato il modello, la funzione *editMesh()* chiama *reDisplay()*, passata per argomento, al fine di ridisegnare la scena.

Per quanto riguarda l'eliminazione di una faccia, questa avviene per mezzo della funzione:

```
GLvoid deleteFace(GLMmodel* model, GLboolean ctrlClicked,
    GLvoid reDisplay())
```

Prima di cancellare l'elemento dal GLMmodel, *deleteFace()* controlla se '*rimangono orfani*' punti o lati della faccia selezionata.
Per ogni lato viene diminuito di 1 il proprio *reference*, cioè quel numero che indica a quante facce fa riferimento un lato. Nel caso questo sia 0, viene tolto il suo riferimento nel modello e in scena con la funzione:

```
GLvoid deleteEdge(GLMmodel* model, GLint index);
```

che aggiorna opportunamente i puntatori nell'array degli edges.

Viene infine analizzato se anche i vertici sono rimasti isolati, senza lati di riferimento mediante la funzione:

```
GLboolean isPointAlone(GLMmodel* model, GLint index);
```

che verifica, per ogni lato in scena, se il vertice con indice '**index**' nell'array dei vertici, è ancora connesso.

Nel caso risulti sconnesso, le sue coordinate vengono azzerate.[9]
Solo infine vengono eliminati i riferimenti alla faccia in memoria e la geometria viene ridisegnata in scena chiamando *reDisplay()* passato per argomento.

```
memset(&model->centroids[model->triangle[
   faceCandidateIndex].cindice], 0, sizeof(GLMcentroid));
memset(&model->triangles[faceCandidateIndex],0, sizeof(
   GLMtriangle));
model->triangles[faceCandidateIndex].cindice = -1;
```

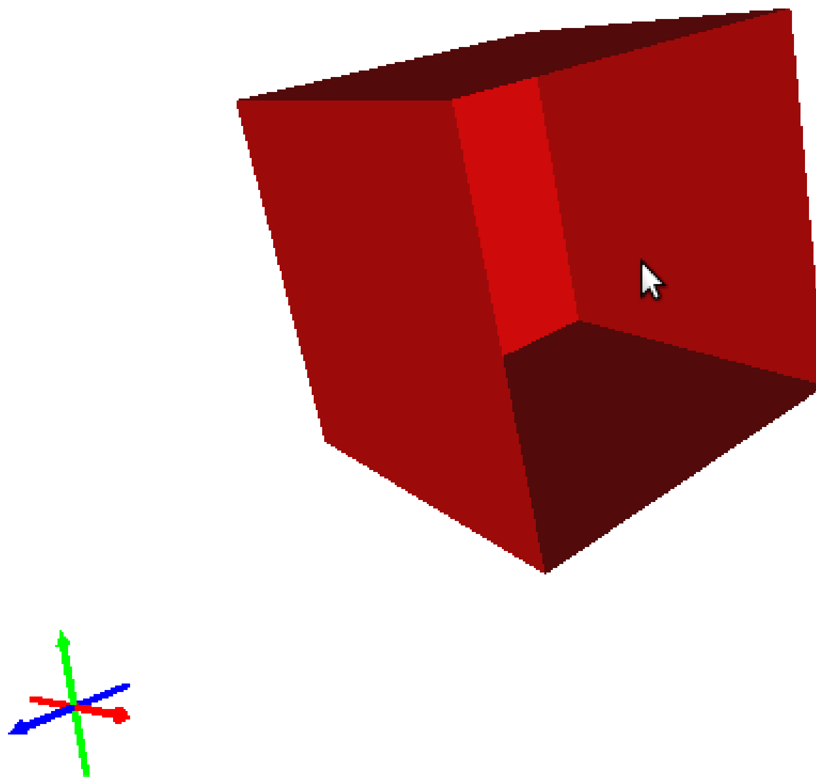


Figura 3.9: Eliminazione di una faccia

[9]E' consigliabile implementare il modello, permettendo l'eliminazioni fisica di punti

3.3 Interfacciarsi a libEditMesh

La natura estendibile di OpenGL e la robustezza di GLM, rendono facilmente usufruibile la libEditMesh in applicazioni per selezione ed editing di mesh in scena.
Una volta che si è inizializzato il modello, il riferimento ad esso diverrà indispensabile, al fine di interagire con la libreria di editing; allo stesso tempo però, con poche funzioni, è possibile interagire con il solido 3D.

Per quanto riguarda l'interazione con l'utente, inizialmente è necessario specificare la tipologia di primitive selezionabili in scena mediante:

```
GLvoid setSelectionMode(GLint mode);
```

oppure

```
GLvoid changeSelectionMode();
```

Successivamente è consigliabile inserire nella funzione linkata a glutPassiveMotionFunc() di GLUT, la chiamata:

```
meshSelect(model, x,y, GL_FALSE);
```

la quale calcolerà, ad ogni spostamento del mouse, l'eventuale selezione di un elemento mesh.
MeshSelect() ricopre un ruolo importante anche nella funzione linkata a *glutMouseFunc()* nel *main()*: l'utente è invitato ad associare una combinazione di tasti a scelta[10] alla selezione permanente di elementi mesh con:

```
meshSelect(model, x,y, GL_TRUE);
```

in preparazione alla fase di editing.

[10] Ad esempio il CTRL+Click del mouse

Sempre nella funzione di gestione del mouse è consigliabile inserire

```
GLvoid deleteFace(GLMmodel* model, GLboolean ctrlClicked,
    GLvoid reDisplay());
```

associando a una combinazione con il mouse, la possibilità di eliminare facce in scena.

Per quanto riguarda l'editing è indispensabile verificare se le frecce degli assi vengono cliccate/trascinate:

```
checkClickArrow(model, getNearPoint(), getFarPoint());
```

la funzione verifica quale freccia è stata eventualmente selezionata, mentre

```
GLvoid checkDragging(GLint state);
```

va utilizzata al fine di delimitare la fase di editing, dall'inizio alla fine del trascinamento del mouse.

A questo punto, nel gestore linkato a *glutMotionFunc()*, si chiamerà:

```
GLvoid editMesh(GLMmodel* model, GLint mouseX, GLint
   mouseY,
       GLvoid reDisplay());
```

che provvederà ad aggiornare il modello e a renderizzarlo in scena.
Per quanto riguarda la resa in scena, una volta impostata la modalità wireframe, si chiami:

```
drawPoints(model);   /*Disegna i punti selezionati*/
drawEdges(model);    /*Disegna i lati selezionati*/
drawTriangle(model); /*Disegna le facce selezionate*/
```

Si invita infine lo sviluppatore a:

- Utilizzare *freeEdges()*, *freeCentroids()* e *resetSelection()* nel caso in cui il modello venga sostituito.
  Le funzioni servono a liberare la memoria utilizzata per le liste dinamiche di edges e baricentri, a ricalcolare le liste e a resettare le selezioni scelte e i puntatori utilizzati.

- Fare uso di *initCentroid()* e *initEdge()* in fase di *init()* al fine di calcolare baricentri e lati nel modello GLM. Si ricorda che il calcolo dei lati aggiunge un overhead in fase di inizializzazione del modello, pertanto initEdge() non è obbligatorio.

- Inserire *setEdgeNextModel(GL_TRUE/GL_FALSE)* nel codice, al fine di permettere/impedire l'inizializzazione dei lati nel successivo modello caricato, nel caso si voglia dare la possibilità di rendere opzionali gli edges nel modello GLM.

- Usufruire di *setSelectionElements()* per la visualizzazione delle primitive di supporto: le bounding-spheres nella selezione dei punti, i punti medi nella selezione dei lati e i baricentri nella selezione delle facce.

3.3.1 Compilazione e linking

LibEditMesh è concepita come libreria dinamica ed è compilabile mediante il proprio *Makefile*.
Una volta estratto il tar.gz in una directory a piacere, si compili con il seguente comando:

```
$ make library
```

il quale provvederà a creare la libreria **libeditmesh.so**[11] e libeditmesh.so.1, link simbolico al .so.
A questo punto si decida se spostare il .so in */usr/lib/yourArchitecture/* o in */lib/yourArchitecture*; infine si aggiorni il LD_LIBRARY_PATH inserendo il path del link simbolico libeditmesh.so.1:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/dirEditMesh
```

Al fine di utilizzare la libreria, si copi *editMesh.h* nella propria directory di lavoro includendola nel progetto:

```
#include "editMesh.h"
```

La libreria è fornita insieme ad un'applicazione di prova chiamata gc_smooth.
Si compili con il comando:

```
$ make gc_smooth
```

[11] Shared Object

3.4 Sviluppi futuri

La struttura modulare della libreria offre numerose vie di miglioramento e sviluppo futuro:

- Gli algoritmi di '*intersection test*' sono in *geoUtils.c*; questo ne permette una facile modifica con algoritmi geometrici più performanti, ove necessario.

- Il modello di memorizzazione utilizzato dalla libreria GLM spesso obbliga l'analisi dell'intera struttura, a causa della mancanza di informazioni come le adiacenze. Una struttura come la '*winged edge*' potrebbe semplificare drasticamente la quantità di calcolo necessario.

- La fase di editing può essere ulteriormente arricchita mediante funzioni di rotazione di facce mesh[12] e/o di scalatura[13], la selezione multipla di elementi mesh[14] o splitting e estrusione di facce[15].

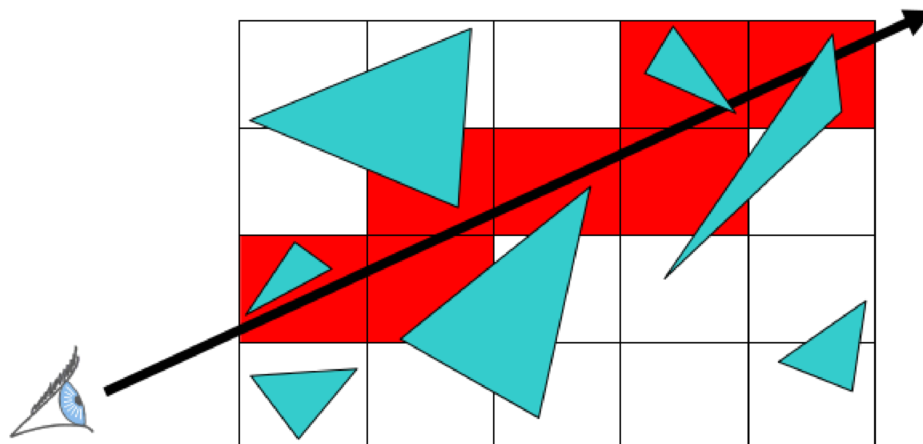
- In fase di selezione, l'algoritmo itera su ogni primitiva memorizzata nel modello. E' possibile migliorare l'interazione con lo spazio, mediante l'uso di tecniche come la '*Regular Grid*'.

[12]Introducendo in editMesh() un meccanismo simile a quello della trackball, implementata nella libreria gltb;

[13]Si sfruttino glmScale(), middleScale() e centroidScale() in un algoritmo simile all'editing per traslazione;

[14]Come visto nella descrizione di *MeshLab*

[15]Mediante traslazione lungo la normale alla faccia simultaneamente al salvataggio dei nuovi vertici nel modello



MIT EECS 6.837, Durand and Cutler

Figura 3.10: Regular grid

Lo spazio viene partizionato in celle di dimensione variabile; il sistema è quindi a conoscenza di quali primitive sono incluse in determinate celle.
A tal proposito il raggio di vista dovrà calcolare l'intersezione con le sole facce nelle celle adeguate.
Algoritmi simili, ma più complessi sono le '*Adaptive Grids*' come le *Nested Grids/OCTREE* o l'uso del '*Bounding Volume Hierarchy*'.
Nonostante la suddivisione renda la selezione più performante, l'editing comporta un overhead significativo ogniqualvolta un solido trasformato non sia più contenuto nella cella di partenza.

- La scelta di utilizzare OpenGL 2.0+ vincola l'uso di funzioni e metodologie oggi deprecate nelle versioni più recenti. Si invita quindi ad effettuare un porting del progetto su OpenGL 3 e 4.

Conclusioni

Il progetto soddisfa, in gran parte, gli obiettivi posti in fase di progettazione, emula efficacemente dinamiche di interazione presenti nei maggiori applicativi commerciali e non commerciali.

La natura OpenSource del progetto ne permette il libero uso e miglioramento, al fine di supportare funzionalità oggi difficilmente presenti nei progetti semi-professionali.

Se ne consiglia quindi l'uso, auspicando che possa migliorare lo sviluppo di applicazioni o che possa essere un buon strumento di studio delle dinamiche OpenGL.

Bibliografia

[1] Dave Shreiner: OpenGL® Programming Guide - Seventh Edition, Addison Wesley, 2009.

[2] Jason L. McKesson: Learning Modern 3D Graphics Programming, Arcsynthesis.org, 2012

[3] History of OpenGL http://www.opengl.org/wiki/History_of_OpenGL

[4] Blender - http://www.blender.org/

[5] Maya - http://www.autodesk.it/products/autodesk-maya

[6] Rhino - http://www.rhino3d.com/

[7] MeshLab - http://meshlab.sourceforge.net/

[8] GL_SELECT: http://www.opengl.org/archives/resources/faq/technical/selection.htm

[9] Color Picking: http://www.opengl-tutorial.org/miscellaneous/clicking-on-objects/picking-with-an-opengl-hack/

[10] Occlusion Queries: http://www.opengl.org/wiki/Query_Object

[11] Hanli Zhao, Xiaogang Jin, Jianbing Shen, Shufang Lu: "Fast and Reliable Mouse Picking Using Graphics Hardware", International Journal of Computer Games Technology Volume 2009 (2009), Article ID 730894

[12] RayPicking: http://www.bfilipek.com/2012/06/select-mouse-opengl.html

[13] Dr.Anton Gerdelan : Mouse Picking with Ray Casting - http://antongerdelan.net/opengl/raycasting.html

[14] Dan Sunday: Intersection of a Ray/Segment with a Triangle - http://geomalgorithms.com/a06-_intersect-2.html

[15] Paul Bourke: Intersection Line/Line - http://paulbourke.net/geometry/pointlineplane/

[16] Giulio Casciola: Fondamenti di Computer Graphics, A.A. 2011/2012

[17] Serena Morigi: Fondamenti di Computer Graphics, A.A. 2012/2013

Ringraziamenti

Mi piacerebbe qui ringraziare uno per uno tutte le persone che mi sono state vicine in questi anni di università, ma la vedo un po' dura... probabilmente una sola pagina non basterebbe.

Voglio ringraziare innanzitutto la mia famiglia che affettivamente e economicamente mi ha sostenuto fino a qui.

Ringrazio il professor. Giulio Casciola per l'attenzione e pazienza riservatami durante questo lavoro di tesi, sapendomi consigliare strategie e metodi al fine di concludere un interessante progetto di grafica.

Ringrazio gli amici e amiche bresciani per essere riusciti a non farmi sentire la distanza e quelli "bolognesi" per aver vissuto con me gioie e dolori di questi anni.

Davide