Alma Mater Studiorum
Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA TRIENNALE IN INFORMATICA

XCBool: analisi e aggiornamento di un pacchetto per la composizione di solidi

Relatore:
Chiar.mo Prof. Giulio Casciola

Tesi di Laurea di:
Marco Malentacchi

Anno Accademico 2009-2010

Indice

Introduzione **7**

1 Composizione di solidi **9**

1.1 Modelli per la rappresentazione di forme 9

1.2 Funzioni NURBS 11

1.2.1 Poligonale di controllo 11

1.3 Superfici trimmate 13

1.3.1 Rappresentazione di superfici trimmate 14

1.4 Algoritmo di composizione 17

1.4.1 Intersezione fra superfici (SSI) 17

1.4.2 Composizione finale 19

2 Introduzione a *XCBool* **21**

2.1 Struttura implementativa di XCBool 21

2.2 Interfaccia di XCBool 22

2.3 Creazione di un nuovo progetto 24

2.4 Esecuzione di un'operazione booleana 24

2.5 Opzioni di visualizzazione 27

2.5.1 Resa grafica di superfici trimmate 28

2.6 Esempio di progetto completo 33

3 Test su XCBool **39**

3.1 Raccolta di test 39

3.1.1 Approccio di testing iniziale 40

3.2 Assenza di intersezioni 43
3.3 Problemi di tipo ISE 43
3.4 Problemi di tipo RDE 44
3.5 Problemi di tipo CDE 46
3.6 Panoramica sulla versione attuale di XCBool 49

4 Aggiornamento di XCBool **51**
4.1 Struttura del codice di XCBool 51
4.1.1 Panoramica sui sorgenti 51
4.1.2 Funzioni principali 53
4.2 Gestione dell'assenza di intersezioni 54
4.3 Struttura dei file .int 56
4.4 Studio dei problemi CDE semplici 57
4.5 Studio dei problemi CDE complessi 60
4.5.1 Gestione di una discontinuità 61
4.5.2 Gestione di due discontinuità 62
4.5.3 Gestione di più di due discontinuità 63
4.6 Studio dei problemi RDE 66
4.6.1 Test interno/esterno 67
4.6.2 Nuova versione di *get_inner_point2* 69

5 Conclusioni **71**
5.1 Limiti del *marching cube* 71
5.2 Superfici aperte 72
5.3 Punti singolari 74
5.4 Resa di superfici trimmate critiche 75
5.5 Ipotesi di lavoro future 78
5.5.1 Strumenti di sviluppo 79

Elenco delle figure

1.1 *Una sfera rappresentata con due diverse discretizzazioni.* 10
1.2 *Curve NURBS di grado 2, 3, 4 con stessa poligonale.* 12
1.3 *Variazione del peso di un control point.* 13
1.4 *Effetto dell'algoritmo di trimming* 14
1.5 *Esempio di regione trimmata ottenuta dalla definizione 1 (winding) e 3 (CSG tree)* 16
1.6 *Tre segmenti di intersezione collegati in una curva chiusa* . . . 19

2.1 *Interfaccia principale di XCBool* 23
2.2 *Interfaccia Boolean Operation di XCBool* 25
2.3 *Preferenze di visualizzazione di XCBool* 26
2.4 *Navigazione di un Trimming tree con tre curve allo stesso livello* 28
2.5 *Navigazione di un Trimming tree con due curve a livelli diversi (la prima include la seconda)* 28
2.6 *Griglia originale e modificata* 31
2.7 *Elaborazione di un dominio trimmato per la resa a video* . . . 31
2.8 *Casistiche di triangolazione* 32
2.9 *Operazione booleana impostata* 34
2.10 *Curve di intersezione nei domini parametrici e in 3D* 34
2.11 *Finestra main con il risultato dell'operazione* 35
2.12 *Domini trimmati degli operandi trasl e cono. La parte attiva è quella evidenziata in grigio.* 36
2.13 *Finestra "Preferences" con i parametri di resa* 37

2.14 *Resa dell'oggetto composto in depth cueing, hidden line e shading; tassellazione di default* 37
2.15 *Resa dell'oggetto composto in depth cueing, hidden line e shading; tassellazione più raffinata* 37

3.1 *Risultato errato dell'operazione cono_rot1.db ∩ sfera94.db* . . . 45
3.2 *Risultato errato dell'operazione gcf2.db ∩ toro.db* 45
3.3 *Intersezioni e risultato dell'operazione cono.db ∩ cilindro.db* . 46
3.4 *Errata chiusura della regione di dominio in cilindro.db* 47
3.5 *Errata chiusura della regione di dominio in cubo94_1.db* 47
3.6 *Errore CDE grave nei domini di toro.db e cono_rot1.db* 48

4.1 *Esempio di applicazione dell'errore "No intersection found"* . 55
4.2 *Operazione cono.db ∩ cilindro.db eseguita correttamente* 60
4.3 *Esempio di curva con una discontinuità* 61
4.4 *Esempi di casistiche di curve con due discontinuità* 62
4.5 *Esempio di curve con tre discontinuità* 64
4.6 *Operazione cono_rot1.db ∩ toro.db eseguita correttamente* . . . 66
4.7 *Due regioni di dominio trimmato annidate* 67
4.8 *Operazione cono_rot1.db ∩ sfera94.db eseguita correttamente* . 68
4.9 *Esempi di punti interni a regioni di dominio calcolati con la nuova get_inner_point2* 70

5.1 *Esempi di superfici aperte* 73
5.2 *Errore di valutazione nell'operazione effe.db ∩ sfera_effe.db* . . 73
5.3 *Errore nella regione di dominio con un punto singolare* 74
5.4 *Curve di trimming problematiche per la resa* 75
5.5 *Possibili soluzioni per la resa di curve di trimming particolari* 76
5.6 *Triangolazione di una sub-regione nel nuovo algoritmo proposto* 77
5.7 *Screenshot che illustra alcune funzionalità di KDevelop* 80

Introduzione

I pacchetti per la modellazione di solidi rappresentano ad oggi una delle applicazioni più importanti e in continuo perfezionamento sia nella direzione della qualità dei risultati ottenibili, sia della semplicità di utilizzo degli stessi.

Nel mio lavoro di Tesi intendo concentrarmi su uno specifico strumento, *XCBool*, appartenente al pacchetto di modellazione *XCModel*, la cui funzionalità è combinare oggetti solidi al fine di ottenerne dei più complessi tramite operazioni booleane: intersezione, unione e differenza.

Nel primo capitolo vengono spiegate le strutture matematiche per la modellazione geometrica con cui lavora *XCModel*, quindi una breve introduzione alle curve, superfici e superfici trimmate, note come *NURBS (Non Uniform Rational B-Spline)*. Questo ci consentirà di presentare l'algoritmo di composizione booleana di solidi adottato da *XCBool*.

Il secondo capitolo è interamente dedicato al pacchetto *XCBool*, illustrandone il design e il funzionamento. A partire dall'interfaccia utente, saranno dettagliati i vari passaggi per ottenere solidi composti, con un cenno al funzionamento sottostante all'interfaccia. Verrà infine introdotto l'algoritmo utilizzato per la resa a video di superfici trimmate.

Con il terzo capitolo entriamo nella parte operativa del mio lavoro, in particolare la fase di analisi della versione attuale di *XCBool* tramite una serie di test rappresentativi adottati come benchmark, al fine di rilevare e classificare le casistiche che presentano ancora anomalie o imperfezioni, e ipotizzando un

primo approccio di lavoro.

Il quarto capitolo tratta la fase di studio e debug di *XCBool*, e il suo conseguente aggiornamento. L'ulteriore approfondimento del codice ci consentirà di fare maggiore chiarezza sulle tipologie di malfunzionamento riscontrate nel capitolo precedente, per ognuna delle quali sarà implementata una relativa soluzione.

Nel quinto capitolo riepiloghiamo i miglioramenti presenti in questa nuova versione di *XCBool*. Saranno anche illustrate le casistiche non ancora funzionanti o intrinsecamente problematiche, come la gestione di punti singolari e superfici aperte, ipotizzando di conseguenza alcune possibilità di lavoro future.

Capitolo 1

Composizione di solidi

Nella modellazione geometrica tridimensionale, con "composizione di solidi" si intende la tecnica che consiste nel combinare due o più solidi al fine di ottenerne in output altri più complessi. I primi vengono detti *operandi* e la combinazione consiste nell'applicare operazioni booleane: *intersezione*, *unione* e *differenza*.

1.1 Modelli per la rappresentazione di forme

Prima di addentrarci nel vivo della composizione, è necessario conoscere le basi della modellazione geometrica tridimensionale. Un modello può essere definito seguendo due metodologie principali: attraverso una semplice mesh poligonale, cioè una discretizzazione della forma che si vuole rappresentare, oppure tramite una rappresentazione matematica della superficie che definisce la forma desiderata. Una mesh è definita da un insieme di vertici (geometria) e da una lista di facce che connettono i vertici (topologia).

La memorizzazione di una mesh poligonale viene realizzata tramite opportune strutture dati atte a contenere:

- una lista delle tre coordinate spaziali per ogni vertice del modello;

- una lista delle facce, definita da tre o più vertici;
- una lista di lati che connettono i vertici.

La modellazione di superfici definite tramite mesh consiste nello spostare i vertici, modificando la forma della mesh e quindi della superficie rappresentata.

Le mesh poligonali definiscono modelli discreti, pertanto se si tratta di modellare superfici curve possiamo lavorare solamente su approssimazioni, e per ottenere risultati soddisfacenti bisogna disporre di una mesh con un numero elevato di piccole facce. Per fare un esempio, se vogliamo rappresentare perfettamente un cubo è necessario (e sufficiente) definire sei facce, invece se si vuole modellare una sfera, più facce utilizziamo e più ci avviciniamo all'oggetto reale (per il quale servirebbero teoricamente infinite facce):

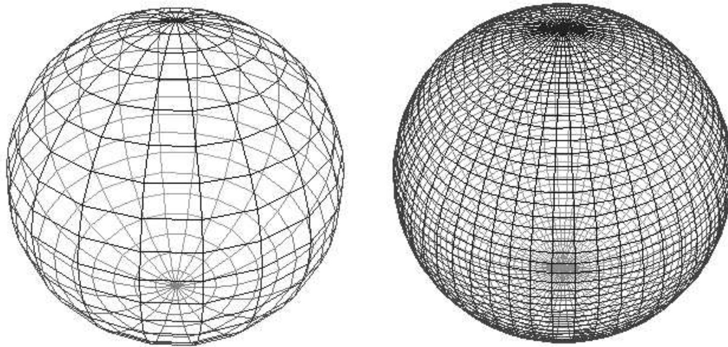

Figura 1.1: *Una sfera rappresentata con due diverse discretizzazioni.*

In alternativa a questa tecnica è possibile lavorare direttamente sulla rappresentazione matematica che descrive la superficie dell'oggetto.

Rispetto ad una mesh si acquisisce il vantaggio di poter rappresentare esattamente la forma. Per quanto riguarda la visualizzazione di una superficie definita in questa maniera, bisogna passare attraverso una discretizzazione del modello matematico.

1.2 Funzioni NURBS

Al momento lo standard nella modellazione a forma libera è costituito dalle funzioni NURBS (*Non-Uniform Rational B-Spline*), funzioni razionali che permettono di descrivere una vasta gamma di forme: curve bidimensionali, curve tridimensionali e superfici.

Le curve NURBS sono funzioni vettoriali in una variabile: ogni punto della curva $C(u)$ è determinato univocamente da un parametro. Questo significa che è possibile scorrere idealmente tutta la curva valutando la funzione $C(u)$ in tutti i punti del dominio, variando $u \in [0, 1]$.

Le superfici NURBS, invece, possiedono un dominio bidimensionale, ovvero sono funzioni *vettoriali in due variabili*. Ogni punto della superficie $S(u, v)$ è determinato univocamente da una coppia di valori $(u_0, v_0) \in [0, 1] \times [0, 1]$.

Un sottoinsieme delle superfici NURBS che ci interessa particolarmente è quello dei *solidi primitivi*. Un solido primitivo ha come boundary una sola superficie NURBS che deve essere chiusa, dividendo lo spazio in due parti (una delle quali limitata, ovvero quella "interna" al solido). Esempi di solidi primitivi sono sfere, tori, superfici chiuse di rotazione e superfici chiuse a forma libera.

Questo tipo di figure può essere composto mediante operazioni booleane, dando luogo ad un *solido composto*.

1.2.1 Poligonale di controllo

Matematicamente le funzioni NURBS sono funzioni razionali (rapporto di polinomi): perciò ad un grado maggiore dei polinomi corrisponde una maggiore "morbidezza" della curva considerata, come si può vedere nella figura 1.2.

Ogni curva può essere governata tramite una particolare linea spezzata,

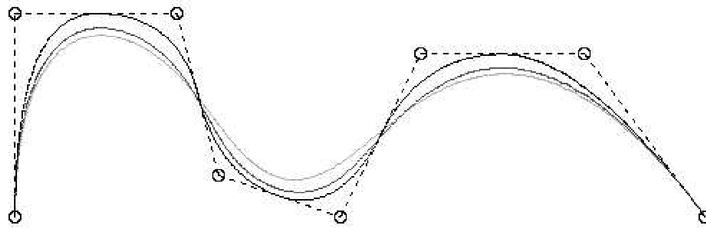

Figura 1.2: *Curve NURBS di grado 2, 3, 4 con stessa poligonale.*

approssimante di forma, che contiene al proprio interno il tracciato della curva: la *poligonale di controllo*.

Si noti che nel caso di curve NURBS di grado 1, la curva è semplicemente una *lineare a tratti*, o "spezzata", che coincide appunto con la sua poligonale di controllo.

I vertici della poligonale sono detti *control point*: l'attività di modellazione consiste in buona parte nella spostamento di questi punti. Inoltre, ad ogni control point viene associato un peso, cioè un valore numerico il cui significato geometrico è quello di avvicinare la curva al control point se il peso aumenta o di allontanarla se il peso diminuisce. Variando i pesi dei control point è quindi possibile cambiare la forma della curva corrispondente (vedi figura 1.3).

L'utente può aggiungere, rimuovere, ruotare, traslare, scalare i control point per definire la curva passo dopo passo: va specificato che la modifica di un control point ha *carattere locale*, cioè interessa soltanto una porzione limitata della curva. Naturalmente più è alto il numero dei control point più è piccola la zona che influenza ognuno di essi.

Nel caso di una superficie NURBS il concetto è analogo, con la differenza che la poligonale di controllo non è una semplice spezzata, ma una griglia tridimensionale a topologia rettangolare.
Per ulteriori approfondimenti: [BBB87], [PITI95], [FARI01].

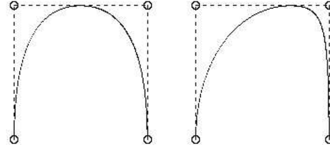

Figura 1.3: *Variazione del peso di un control point.*

1.3 Superfici trimmate

L'esecuzione di una operazione di *trim* su una superficie NURBS ha l'effetto di estrarre una *porzione del dominio parametrico* della superficie di partenza, restringendo la definizione della superficie ad un sottodominio $D \subset (U \times V)$, chiamato *trimming region*. Il dominio D è definito come l'insieme di regioni di $U \times V$ limitate da curve NURBS chiuse $C_k(t)$ orientate e non intersecantesi, dette *curve di trimming*.

I contorni della superficie trimmata sono ottenuti mappando le curve 2D del dominio trimmato sulla superficie 3D. Tali curve nello spazio 3D sono chiamate *ConS: Curves on Surface*, in quanto sono curve sulla superficie. Più formalmente, se $S(u, v)$ è la funzione che definisce una superficie S nel dominio parametrico e $C_k(t) = (x_k(t), y_k(t))$ sono le curve di trimming nel dominio, le *ConS* si ottengono tramite la loro proiezione $S(x_k(t), y_k(t))$.

Quest'operazione genera due superfici complementari: una corrispondente alla regione estratta e l'altra costituita dalla superficie di partenza privata della regione selezionata. Nella figura 1.4 è dato un esempio dell'effetto di un'operazione di trim sulla superficie di una sfera, alla quale sono state "ritagliate" sei calotte.

Figura 1.4: *Effetto dell'algoritmo di trimming*

1.3.1 Rappresentazione di superfici trimmate

La rappresentazione di superfici trimmate, al fine di identificare una superficie ben precisa delle due complementari, può essere implementata in vari modi [CAMO00].

- *Definizione 1: verso di percorrenza (winding)*

  Le regioni di trimming sono definite da curve chiuse orientate in una certa direzione. Quelle esterne sono definite in senso antiorario, viceversa quelle interne in senso orario. Il dominio della superficie trimmata è definito come l'insieme delle regioni contenute nel boundary esterno (corrispondente alle curve esterne). Nella figura 1.5 è illustrata questa definizione, dove l'area facente parte della regione trimmata è ombreggiata.

- *Definizione 2: inclusione pari*

  Le curve di contorno del dominio che identificano regioni incluse od escluse dalla superficie trimmata sono basate sulla parità. In particolare,

le regioni della superficie all'interno di un numero di curve pari sono incluse nella superficie trimmata.

- *Definizione 3: albero CSG*

  Il dominio trimmato è rappresentato da una struttura CSG (Constructive Solid Geometry) ad albero, formata da un insieme di spazi di dominio (con una sola curva di trimming, in modo da definire due sottospazi) legati fra loro da operazioni booleane. Le curve allo stesso livello sono disgiunte fra loro. La regione trimmata da includere nella superficie è determinabile classificando un singolo punto, perché convenzionalmente la condizione di interno/esterno sono assegnate alternando i livelli dell'albero (concetto simile alla definizione precedente di pari/dispari). La figura 1.5 mostra un esempio di albero CSG.

  Il dominio trimmato sembra un insieme di *isole* e *laghi*, dove l'isola rappresenta parte della regione trimmata e il lago è il buco contenuto in essa. L'algoritmo usato per costruire un albero CSG consiste nell'unione di tutte le isole dalle quali sono sottratti tutti i propri laghi.

Le superfici NURBS trimmate sono adottate nell'industria CAD/CAM e sono ormai uno standard utilizzato in molti sistemi di modellazione, come OpenGL, Starbase (Hewlett-Packard Corp.) e Renderman (Pixar). Sono essenziali nella modellazione di oggetti con superfici non regolari, e anche nella descrizione di modelli complessi in cui l'unione di superfici trimmate dà una rappresentazione completa di una *composizione solida*: sono il risultato di *operazioni booleane* su oggetti solidi definiti da superfici NURBS.

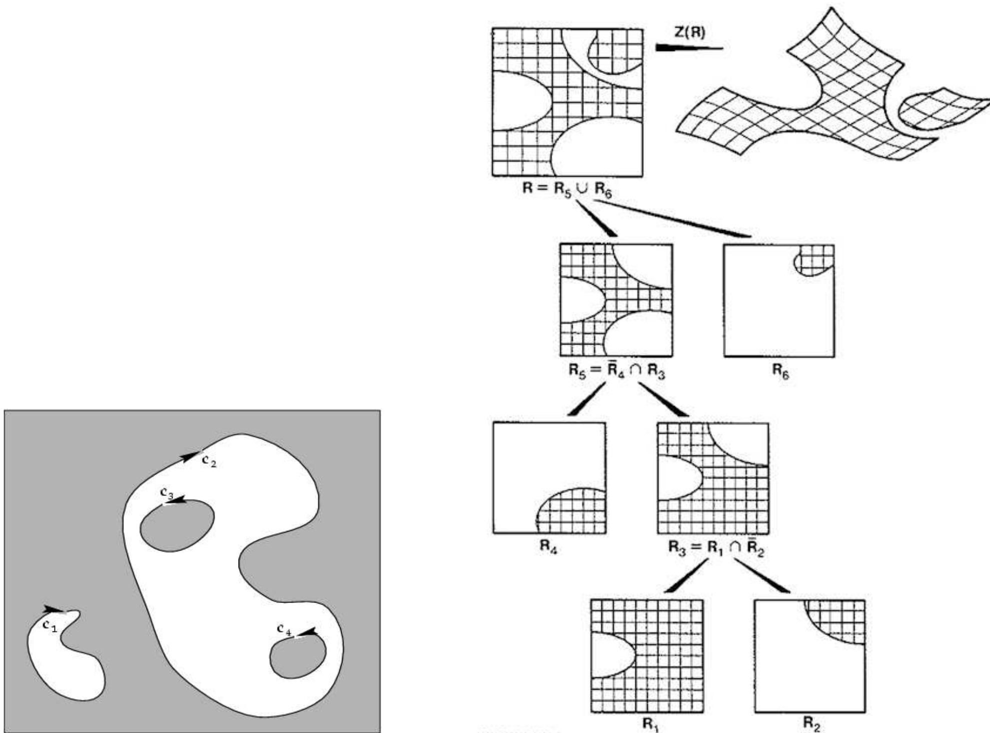

Figura 1.5: *Esempio di regione trimmata ottenuta dalla definizione 1 (winding) e 3 (CSG tree)*

1.4 Algoritmo di composizione

Ora che abbiamo definito le superfici NURBS, i solidi primitivi e l'operazione di trim, siamo in possesso degli strumenti necessari per comprendere l'algoritmo di composizione di solidi. Tale algoritmo presenta difficoltà e costi computazionali elevati, ma grazie all'introduzione delle superfici trimmate sono stati in parte superati, perché permettono appunto una rappresentazione completa del boundary di un solido attraverso l'unione di superfici ristrette ad opportuni domini. Infatti il boundary di un solido composto S è definibile in questo modo:

$$bS = \bigcup_{i=1}^{k} \mathbf{r}_i(D_i)$$

Dove D_i è la trimming region associata alla superficie $\mathbf{r}_i$. Si noti che il boundary del solido composto è nuovamente una superficie chiusa. Dati due solidi A e B definiti dalle rispettive superfici di bordo, il problema della composizione consiste nel determinare i bordi dei solidi $b(A \cup B)$, $b(A \cap B)$ e $b(A \setminus B)$. Nel caso più complesso A e B sono anch'essi solidi composti, quindi

$$bA = \bigcup_{i=1}^{k} \mathbf{r}_i(D_i) \ e \ bB = \bigcup_{j=1}^{h} \mathbf{s}_j(E_j)$$

dove $\mathbf{r}_i$ ed $\mathbf{s}_j$ sono superfici trimmate definite sulle regioni di trimming D_i e E_j. Il problema è che alla base dell'algoritmo di composizione vi è l'operazione di intersezione fra superfici, e l'intersezione fra superfici trimmate è molto complicata, meglio quindi riformularla eseguendo l'intersezione fra le superfici non trimmate (definite sull'intero dominio parametrico).

1.4.1 Intersezione fra superfici (SSI)

Abbiamo appena visto come il primo passo fondamentale per costruire un solido composto consiste nell'individuare le curve di intersezione fra i due solidi di partenza, o in altri termini, ottenere un trimming delle superfici definito dalle loro intersezioni reciproche. Si noti che questa operazione non è calcolabile tramite formule applicate ai modelli parametrici delle superfici

NURBS, perché risulterebbe matematicamente troppo complessa. I metodi più interessanti proposti in letteratura per risolvere il problema SSI (Surface Surface Intersection) sono fondamentalmente due:

- il metodo *geometrico*: si basa sulla discretizzazione delle mesh fino a ridurre il problema in intersezioni elementari (del tipo triangolo/piano). E' un metodo robusto ma lento.

- il metodo *numerico*: consiste nella ricerca di una sequenza di *starting point*, dai quali risalire alle curve di intersezione che saranno finalizzate nella fase di ordinamento. E' un approccio iterativo, veloce ma non sempre affidabile.

E' possibile sfruttare i vantaggi dell'uno e dell'altro metodo tramite un terzo metodo, detto *ibrido*, che consiste nell'effettuare una discretizzazione come nel metodo geometrico per localizzare gli starting point, raffinati poi successivamente mediante tecniche numeriche [CAMO00], [SPAG98].

Nel nostro approccio ibrido distinguiamo le seguenti fasi:

- una prima fase in cui si genera una *griglia adattiva*, cioè si approssima ciascuna superficie mediante una serie di curve isoparametriche, approssimate da lineari a tratti. Questo consente di ottenere una triangolazione della superficie.

- Nella seconda fase generiamo gli *starting point*, mediante intersezioni elementari (segmento di una superficie/triangolo dell'altra) per ottenere almeno uno starting point per ciascuna curva di intersezione.

- A partire da questi punti, procediamo con un algoritmo detto *marching cube* per la ricerca dei punti successivi nelle curve di intersezione. La strategia consiste nel percorrere il tratto di curva in direzione della sua *tangente*, quindi siamo agevolati se lavoriamo su superfici *regolari*, cioè continue almeno fino alla derivata prima. Nel caso di superfici con *spigoli vivi*, il rilevamento della curva è soggetto ad interruzioni, dando origine a più tratti, che dovranno essere successivamente ordinati.

- Nella fase conclusiva, ordiniamo i tratti di curva trovati, fino alla determinazione delle curve di intersezione 3D.

Come in ogni procedimento adattivo è necessario tenere conto di alcune tolleranze, in particolare:

- *Search Refinement Tolerance (SRT)*: regola la qualità della suddivisione adattiva della superficie nella prima fase dell'algoritmo.

- *Curve Refinement Tolerance (CRT)*: se due punti appartenenti ad una curva di intersezione sono ad una distanza inferiore a CRT, il segmento che li unisce è preso come approssimazione del tratto di curva che li separa.

Trovate le coordinate dei punti che approssimano le curve di intersezione, è necessario *chiuderle*, ovvero collegare i segmenti di intersezione in curve chiuse (figura 1.6) [CASA87].

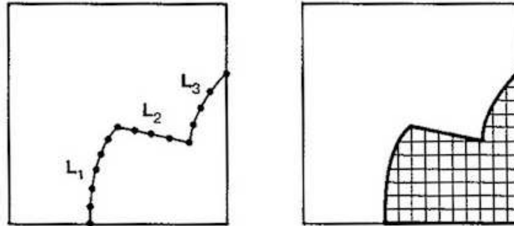

Figura 1.6: *Tre segmenti di intersezione collegati in una curva chiusa*

Il passaggio finale consiste nel memorizzare le regioni di dominio in un albero CSG, rilevandone la loro profondità.

1.4.2 Composizione finale

Al termine dell'algoritmo di SSI, dobbiamo stabilire quali regioni della superficie trimmata di ogni operando includere nel solido risultante. Le formule base della composizione booleana sono [CABO89]:

$$b(A \cup B) = (bA \cap cB) \cup (bB \cap cA)$$
$$b(A \cap B) = (bA \cap iB) \cup (bB \cap iA)$$
$$b(A \setminus B) = (bA \cap cB) \cup (bB \cap iA)$$

dove $b(X)$ è il *boundary di X*, $i(X)$ è l'*interno di X* e $c(X)$ è il *complementare di X*.

Per scendere nel dettaglio implementativo di queste regole, occorre effettuare dei test di tipo interno/esterno, in particolare:

- si cerca arbitrariamente un punto appartenente alla regione di dominio più annidata del primo operando, tramite il teorema di Jordan: se la semiretta uscente dal punto oggetto del test incontra un numero dispari di intersezioni con la curva, allora il punto è interno alla stessa;
- si verifica se quel punto, che si troverà sulla superficie del primo operando, è interno o esterno al secondo operando. Il ragionamento è simile al precedente ma applicato al tridimensionale, farà quindi uso di un algoritmo di raytracing per proiettare la semiretta dal punto di origine, per poi procedere con la regola di Jordan.

Riprenderemo questo procedimento nei capitoli successivi quando parleremo più in dettaglio del codice di XCBool.

Capitolo 2

Introduzione a *XCBool*

Il programma di modellazione *XCBool* è il compositore di solidi del pacchetto *XCModel*. Analogamente agli altri applicativi che fanno parte di XCModel, anche XCBool è stato progettato con un design intuitivo per consentire all'utente di seguire passo-passo la procedura di composizione di solidi. In questa introduzione esploreremo gli aspetti fondamentali del pacchetto, descrivendone l'interfaccia utente e le fasi principali di funzionamento.

2.1 Struttura implementativa di XCBool

Il pacchetto XCBool in realtà non è costituito da un unico programma, il core dell'applicazione fa uso di altri due programmi eseguibili e indipendenti: *xcssi* e *xcdbe*, che gestiscono rispettivamente il problema Surface/Surface Intersection e i file .dbe, formato per la rappresentazione di superfici trimmate. Oltre a questi, XCBool (come gli altri programmi di XCModel) fa uso della libreria *trim* per la resa in tempo reale di superfici trimmate, e della libreria *xtools* che consiste in una collezione di oggetti widget (pulsanti, caselle di testo, barre di scorrimento...) per facilitare la creazione di GUI (Graphical User Interface). Per approfondimenti: [XCMODEL], [XTOOLS].

Nota sulla terminologia *con il nome "XCBool" ci riferiamo al pacchetto applicativo completo, mentre con "xcbool" indicheremo l'eseguibile, quindi il programma a sé stante, che fa uso degli altri programmi e librerie di supporto per completare il calcolo delle operazioni booleane.*

Per quanto riguarda la filosofia di implementazione, XCBool è nato per sopperire ai limiti di molti altri programmi di modellazione, che utilizzano approssimazioni poliedriche per la rappresentazione di solidi e la relativa composizione, i quali algoritmi risultano inefficienti e anche imprecisi.

XCBool invece permette all'utente di ottenere solidi composti partendo da oggetti primitivi, definiti da una singola superficie NURBS non trimmata, per poi costruire l'albero CSG dell'operazione booleana scelta. Gli oggetti finali (e anche quelli intermedi) ottenuti dal progetto CSG sono liste di superfici NURBS trimmate. L'intersezione fra superfici, eseguita da xcssi, è ottimizzata adottando il metodo ibrido illustrato nel capitolo precedente.

2.2 Interfaccia di XCBool

L'area di lavoro di XCBool è suddivisa in varie sezioni, come si può osservare nella figura 2.1:

- i pulsanti nella sezione **Tools**, per richiamare separatamente i programmi ausiliari per la gestione delle superfici trimmate
- la sezione **New operation**, per definire una nuova operazione booleana
- la **Object list** che contiene gli oggetti su cui lavorare o risultanti da operazioni effettuate
- le due righe di pulsanti in basso a sinistra gestiscono le funzionalità principali per lavorare con i progetti e manipolare gli oggetti grafici
- la sezione **View** consente di visualizzare dettagli importanti sugli oggetti elaborati

- i pulsanti **Distance**, **Camera pos** e **Camera dir** servono per navigare nell'area di visualizzazione degli oggetti

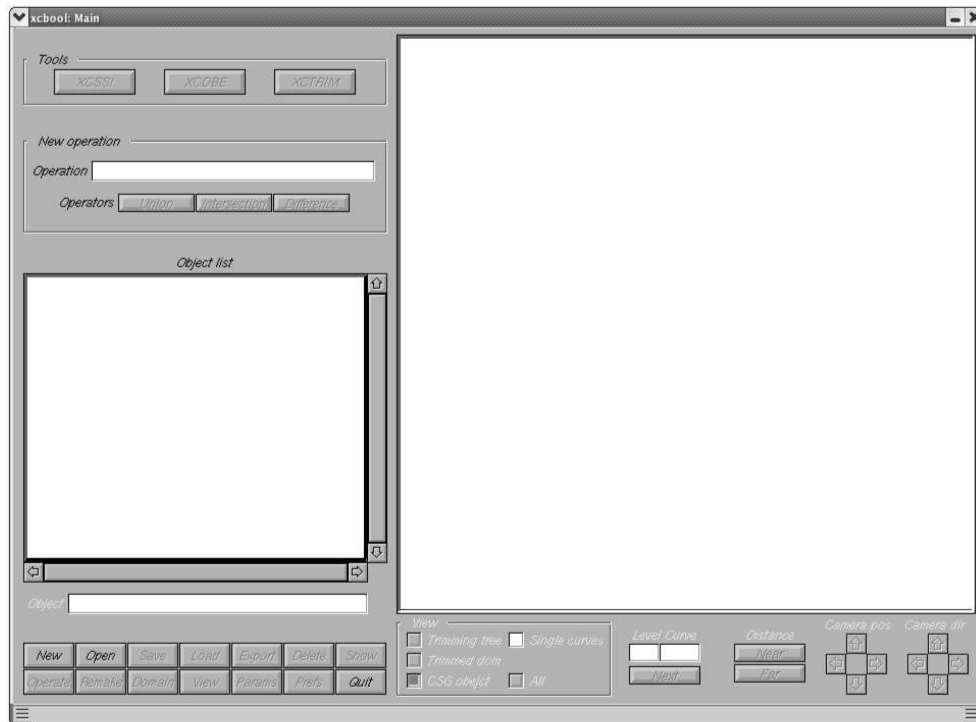

Figura 2.1: *Interfaccia principale di XCBool*

E' possibile distinguere subito i controlli della GUI che sono attivi in un dato momento: per convenzione quelli abilitati hanno le scritte nere, mentre in grigio chiaro significa che non sono disponibili in quel momento. Questo dipende dagli oggetti selezionati e dalle operazioni eseguibili dipendentemente dal contesto in cui ci troviamo. Appena aperto XCBool è possibile solamente: creare un nuovo progetto (**New**), aprirne uno esistente (**Open**) o uscire dall'applicazione (**Quit**).

2.3 Creazione di un nuovo progetto

Il pulsante **New** consente di creare un nuovo progetto, che avrà estensione *.csg*. E' consigliabile utilizzare una directory per ogni progetto, in quanto l'applicazione andrà a creare gli altri file prodotti durante il lavoro nella stessa directory in cui si trova il file .csg. Una volta inizializzato un progetto nuovo, sarà disponibili il pulsante **Load** per caricare oggetti grafici. Si noti la differenza con **Open** che serve a caricare progetti già creati in precedenza. Gli oggetti importabili possono essere del seguente tipo:

- file .db: *primitivi*, rappresentati da superfici NURBS non trimmate e chiuse (le superfici aperte possono indurre XCBool ad errori che verranno approfonditi in seguito);

- file .dbt: *temporanei*, ottenuti da primitivi e utilizzabili per composizioni, ma non essendo nel loro formato definitivo non sono visualizzabili;

- file .obj: *composti*, cioè oggetti finali ottenuti a partire da primitive e/o oggetti temporanei.

2.4 Esecuzione di un'operazione booleana

Una volta caricate almeno due superfici, sarà possibile scegliere quale operazione effettuare e in quale ordine, cliccando nella object list sul primo operando, poi sull'operazione desiderata (pulsanti **Union**, **Intersection** e **Difference**) e infine sul secondo operando. Ora il pulsante **Operate** diventa attivo, e consente di aprire la finestra *Boolean operation* (figura 2.2).

Questa interfaccia permette di gestire l'intersezioni delle superfici, il passaggio intermedio necessario per comporre i solidi. Sono presenti diversi comandi e opzioni: oltre ai pulsanti di navigazione in basso a destra (analoghi a quelli della finestra principale), abbiamo la possibilità di regolare alcuni

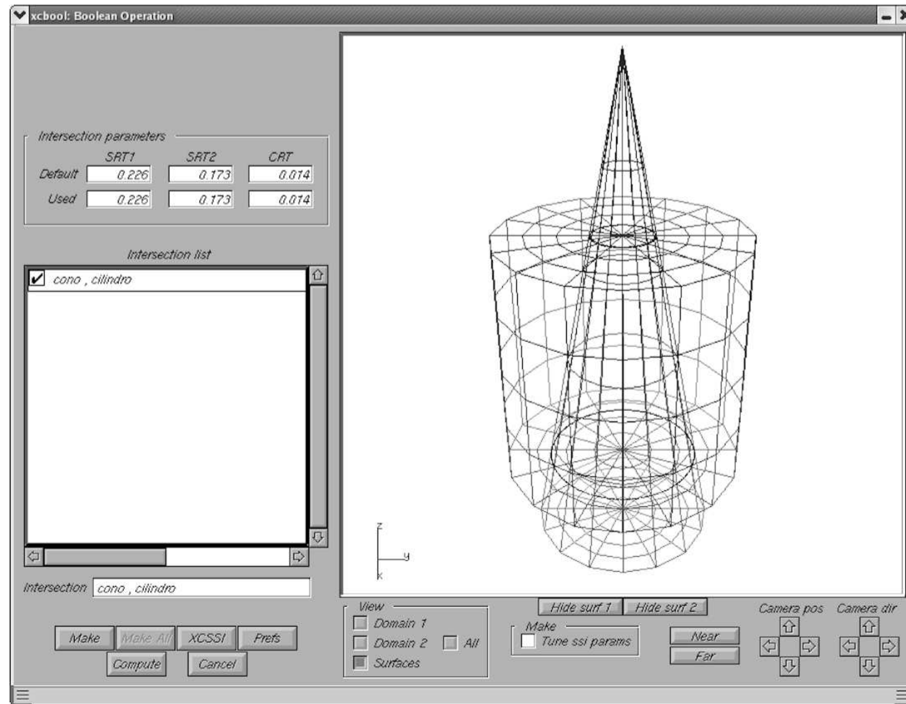

Figura 2.2: *Interfaccia Boolean Operation di XCBool*

parametri (SRT e CRT) utilizzati dall'algoritmo SSI e già descritti nel capitolo precedente.

Solitamente, è consigliabile aggiustare quelli predefiniti, che si trovano nella riga **Default**, solo se non danno risultati soddisfacenti o producono addirittura errori, inserendo i nuovi valori nella riga **Used**. In generale non c'è un metodo per calcolare a priori la miglior combinazione di queste tolleranze, in termini di efficienza, robustezza e precisione. A questo scopo, è possibile eseguire xcssi in *modalità interattiva*, tramite il pulsante **XCSSI** di XCBool, al fine di determinare più agevolmente le tolleranze migliori in fase preliminare.

Ora possiamo fare uso del pulsante **Make** che lancia l'eseguibile *xcssi* in background. Trattandosi di un processo separato, l'interazione con il chiamante xcbool è gestita tramite scambio di segnali fra processi. Al termine dell'operazione, in assenza di errori, xcssi genera un file *.int* contenente le definizioni delle curve di intersezione.

A questo punto XCBool riprende la sua esecuzione, ripristina lo stato

attivo della finestra *Boolean operation* e legge il file .int, disegnando le intersezioni sia nella rappresentazione in prospettiva dei solidi sia dei loro domini; si rendono disponibili le funzionalità nella sezione **View** per mostrare separatamente le curve di intersezione in prospettiva (**Surfaces**), le corrispondenti curve nel dominio parametrico di uno dei due solidi (**Domain 1** e **Domain 2**) o tutte in contemporanea (**All**). Per avere una visione più chiara delle intersezioni in 3D può essere utile anche nascondere l'una o l'altra superficie, tramite i pulsanti **Hide surf 1** e **Hide surf 2**. Per accedere invece a preferenze più avanzate usiamo il pulsante **Prefs** (figura 2.3).

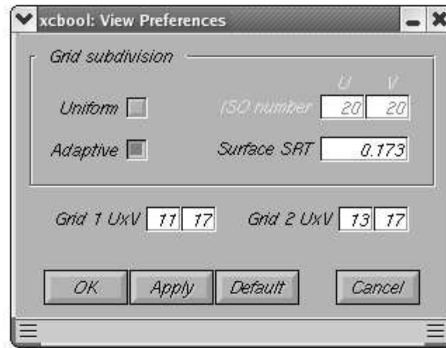

Figura 2.3: *Preferenze di visualizzazione di XCBool*

I checkpoint **Uniform** e **Adaptive** ci permettono di scegliere la suddivisione della griglia. Nell'uniforme è possibile settare direttamente la suddivisione del dominio nei parametri U e V, mentre nell'adattivo possiamo cambiare la tolleranza *SRT*.

Il pulsante **Apply** serve per applicare istantaneamente le preferenze scelte ed osservarne il risultato, per poi confermarle con **OK**, annullarle con **Cancel** o reimpostare le predefinite con **Default**.

Ora è possibile procedere con la fase finale della composizione, tramite il pulsante **Compute** che risulta abilitato dopo il calcolo delle intersezioni. Questo lavoro è esclusivamente a carico di XCBool, che deve implementare

la parte di algoritmo descritto nel paragrafo 1.4.2. Una volta creato il solido composto, si chiude automaticamente la finestra *Boolean operation* tornando a quella principale.

2.5 Opzioni di visualizzazione

Conclusa l'operazione booleana, nella object list si troverà il nuovo oggetto, al quale XCBool assegna un nome di default con indice progressivo (*objectX*).

Subito sotto troviamo i nomi dei due solidi di partenza, scritti *indentati* rispetto a quello composto, perché legati ad esso, infatti quelle voci rappresentano i due operandi trimmati. Selezionandone uno, si rende attivo il pulsante **Domain**, che ci illustra maggiori dettagli su come è stato elaborato il suo dominio trimmato:

- il checkpoint **Trimmed dom** ci mostra la tassellazione effettuata nel dominio, necessaria per la sua rappresentazione grafica;
- il checkpoint **Trimming tree** ci mostra il dominio evidenziando le regioni trimmate attive;
- la checkbox **Single curves** ci permette di visualizzare singolarmente le regioni di dominio, sfogliabili tramite il pulsante **Next**: le caselle di testo **Level** e **Curve** indicano rispettivamente per ogni curva il livello dell'albero CSG a cui appartengono e il loro progressivo all'interno di quel livello (figure 2.4 e 2.5).

I parametri che regolano la suddivisione della griglia sono modificabili tramite il pulsante **Params**, e sono analoghi a quelli utilizzabili dalla finestra *Boolean operation* descritti in precedenza. L'unica differenza è l'ulteriore parametro *Curve CRT*, che definisce il dettaglio di resa delle curve trimmate, da cui segue un maggiore o minor numero di punti per rappresentarla a video.

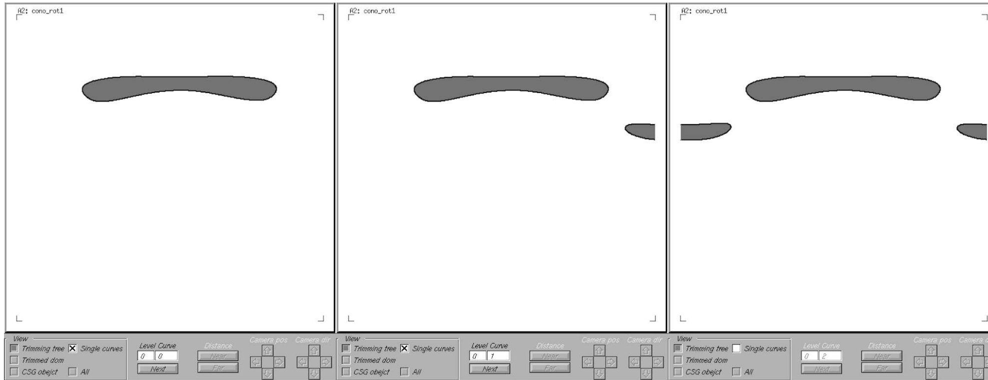

Figura 2.4: *Navigazione di un Trimming tree con tre curve allo stesso livello*

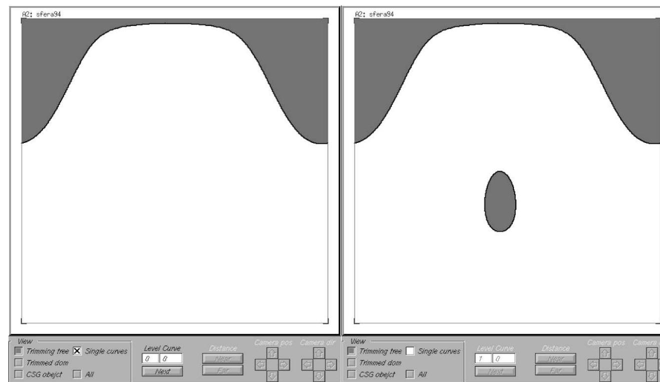

Figura 2.5: *Navigazione di un Trimming tree con due curve a livelli diversi (la prima include la seconda)*

2.5.1 Resa grafica di superfici trimmate

Vogliamo entrare maggiormente in merito al problema della resa di superfici trimmate [DEMA99].

Come sappiamo, gli algoritmi di visualizzazione differiscono nella qualità di resa, che è inversamente proporzionale al costo computazionale. Un sistema di modellazione ha bisogno sia di algoritmi di livello medio/basso per consentire una visione in tempo reale, quindi algoritmi come il *depth cueing*, *hidden line* o *z-buffer*, sia di alto livello per la resa realistica finale, come il *raytracing*.

Gli algoritmi di qualità medio/bassa utilizzano un'*approssimazione lineare*

a tratti (tassellazione o triangolazione) della superficie trimmata in base ad una certa tolleranza. Questo consente una resa real-time anche *via software*, come nel caso di XCModel, senza ricorrere all'accelerazione video presente negli hardware di oggi.

L'algoritmo adottato in XCModel è stato progettato in seguito ad un'analisi di alcuni algoritmi presenti in letteratura per la tassellazione di superfici trimmate. Non è euristico, ma basato su un'*esaustione* dei casi possibili nell'ipotesi che ogni faccia della griglia rettangolare di tassellazione contenga al più *una* delle curve di trimming: questa condizione porta ad una riduzione dei casi da considerare per la triangolazione dei punti di ogni poligono del dominio. Quindi la griglia iniziale viene modificata finché non soddisfa il precedente vincolo.

Nel caso in cui le tolleranze impostate nelle opzioni di visualizzazione di XCBool non consentano la grigliatura con quel vincolo, il programma non riesce a completare la resa della geometria, informando l'utente con l'errore **"invalid trimming grid"**. Nell'ultimo capitolo (sezione 5.4) approfondiremo i limiti di questo algoritmo proponendo alcuni miglioramenti.

I seguenti passaggi descrivono a grandi linee l'algoritmo utilizzato.

- *Suddivisione della superficie NURBS*: la superficie viene divisa lungo curve isoparametriche in entrambe le direzioni u e v, dette *u-line* e *v-line*. La suddivisione può essere di due tipi: *uniforme*, in cui la distanza fra le isoparametriche in una direzione è sempre la stessa, o *adattiva*, regolata da una tolleranza prefissata (SRT), in modo da ottenere una griglia più fitta in corrispondenza delle zone in cui la superficie è più curva. La grigliatura 3D della superficie associata alla griglia del dominio rappresenta un'approssimazione "piana" a tratti della superficie.

- *Tracing delle curve di trimming*: per ogni curva di trimming, vengono cercate le intersezioni con le u-line e v-line. In particolare, le intersezioni

sono calcolate considerando coppie di punti curva consecutivi e intersecando il segmento che li unisce con le opportune u-line e v-line; queste intersezioni vengono memorizzate in modo da tenere traccia facilmente dell'appartenenza o meno dei punti griglia alle regioni di trimming.

- *Modifica della griglia*: si effettua una scansione della griglia considerando prima le colonne poi le righe, al fine di individuare i lati dei rettangoli griglia che presentano più di una intersezione. Prendendo in considerazione l'intera *slice* (colonna o riga), si individuano opportuni punti dove inserire nuove isoparametriche, cercando di minimizzare il numero di interventi. Questo meccanismo è illustrato nella figura 2.6.

- *Ricostruzione delle curve di intersezione*: ora che abbiamo i punti di intersezione e i vertici dei rettangoli griglia, potremmo già procedere con la triangolazione, ma l'approssimazione poligonale delle regioni trimmate è ulteriormente migliorabile aggiungendo punti originali delle curve di trimming. Questo raffinamento è regolato da una tolleranza CRT data, che più è piccola e più induce ad utilizzare un maggior numero di punti originali delle curve.

- *Tassellazione*: si procede alla triangolazione di ogni rettangolo griglia contenente segmenti di curve di trimming, utilizzando i punti griglia e i punti ulteriori del segmento di curva aggiungi nel passaggio precedente, costruendo opportunamente poligoni nel dominio. Nella figura 2.8 sono illustrate le possibili casistiche di triangolazione.

- *Mapping dei poligoni sulla superficie*: tutti i vertici dei poligoni costruiti in precedenza vengono mappati sulla superficie, in modo da ottenere i corrispondenti poligoni 3D.

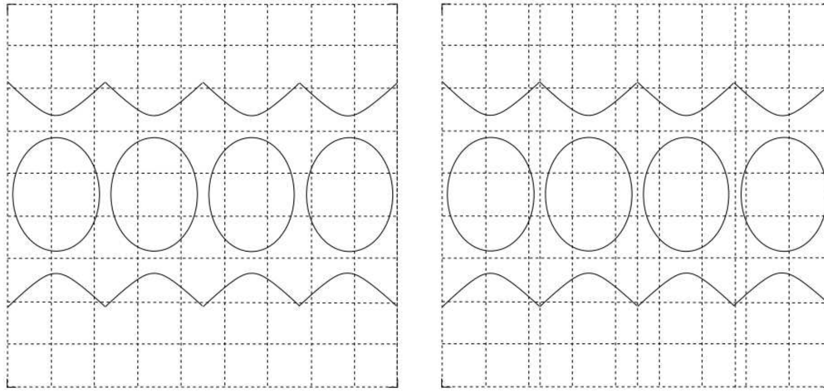

Figura 2.6: *Griglia originale e modificata*

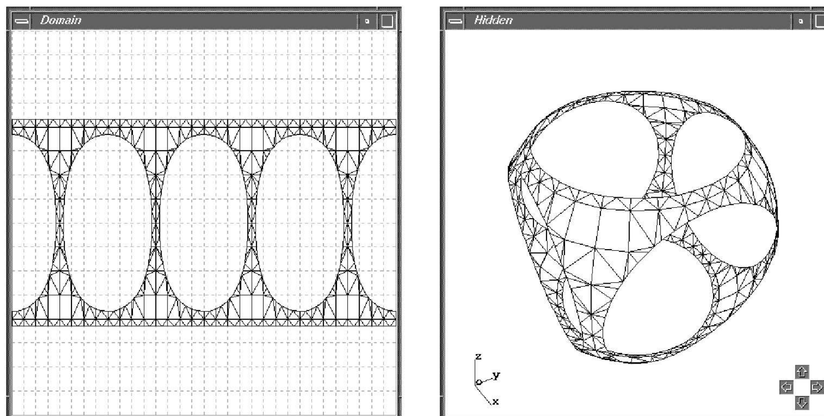

Figura 2.7: *Elaborazione di un dominio trimmato per la resa a video*

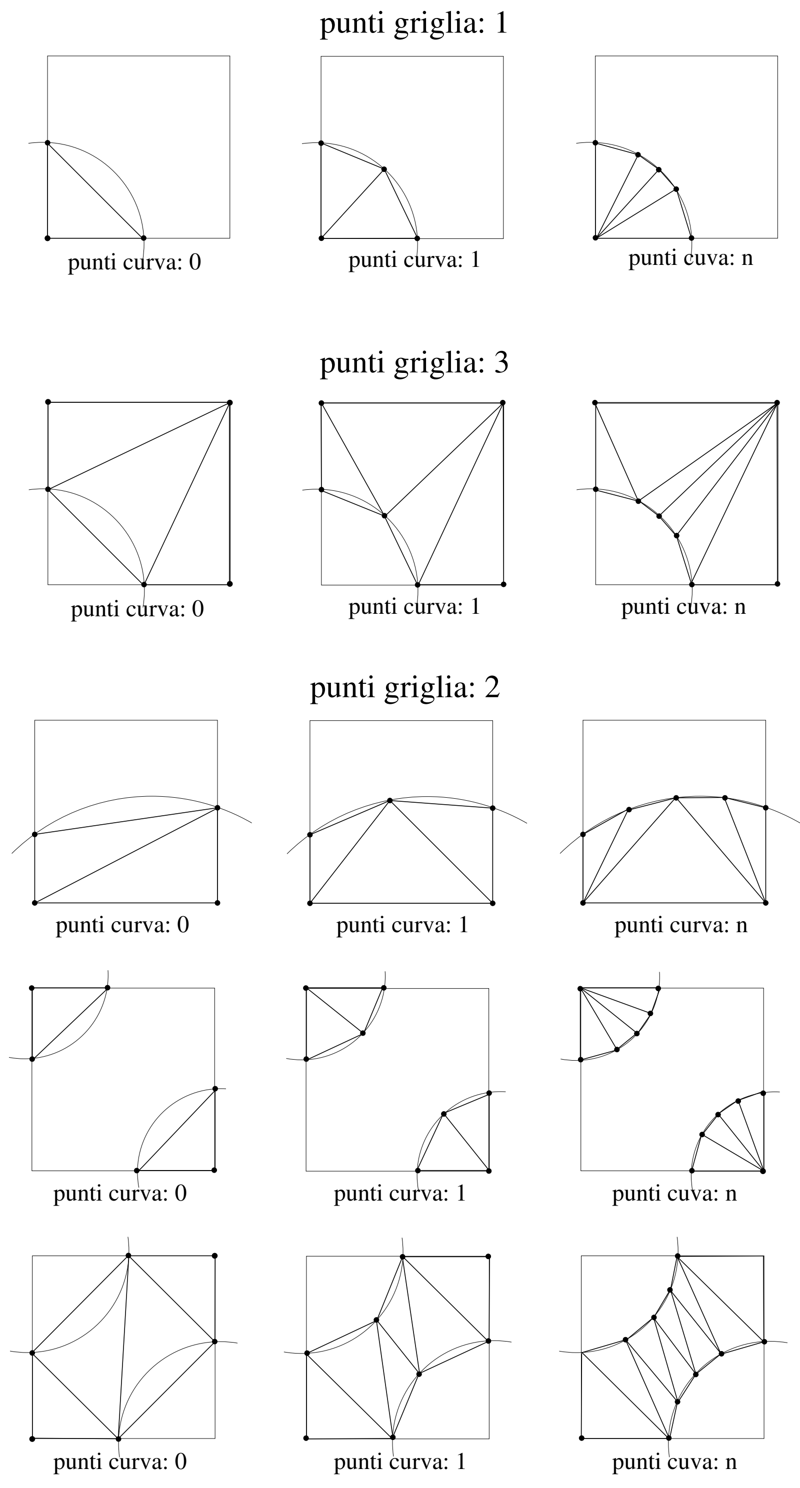

Figura 2.8: *Casistiche di triangolazione*

2.6 Esempio di progetto completo

Per chiarire ulteriormente l'utilizzo del pacchetto XCBool, presentiamo un tutorial completo in cui calcoleremo l'operazione di differenza fra due solidi: *trasl.db*, un cilindro traslato rispetto al suo asse di origine, e *cono.db*, un semplice cono ottenuto per rotazione.

Per prima cosa creiamo il progetto con il pulsante **New**. Usiamo la finestra di dialogo che compare per scegliere la directory in cui creare il nuovo progetto. Quella di default è *objects*, che ci aspettiamo si trovi allo stesso livello della directory *bin* di XCModel. Assegnamo al progetto un nome arbitrario con estensione *.csg*, scrivendolo nella casella di testo *File name*.

Carichiamo i due operandi *trasl.db* e *cono.db* con il pulsante **Load**: come impostazione di default, XCBool cerca di aprire la directory chiamata *surfaces*, anche questa allo stesso livello di *bin*, che ci aspettiamo contenga i file .db caricabili. Se tale cartella non esiste viene restituito un messaggio di errore, senza però impedire all'utente di caricare i file .db scegliendo manualmente altre directory.

Per impostare l'operazione *trasl.db* \ *cono.db* selezioniamo il primo operando, premiamo il pulsante **Difference** e selezioniamo il secondo operando. Osserviamo che ora la casella di testo *Operation* contiene la direttiva:
Difference(A0, A1), figura 2.9.

Procediamo con il pulsante **Operate** che predispone la già nota finestra *Boolean Operation*, dove troviamo una rappresentazione in prospettiva dei due solidi interessati e i parametri per l'intersecatore.

Diamo il via all'operazione SSI con **Make**, accettando le tolleranze di default. Terminata l'operazione possiamo osservare la curva di intersezione 3D. Se questa non fosse molto chiara a causa della contemporanea rappre-

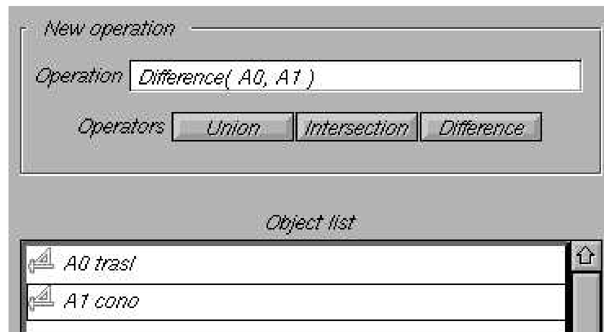

Figura 2.9: *Operazione booleana impostata*

sentazione dei solidi, possiamo nasconderli tramite i pulsanti **Hide surf 1** e **Hide surf 2**.

Attivando la checkbox **All** osserviamo, oltre alla curva di intersezione 3D, le corrispondenti curve nel dominio parametrico di *trasl.db* e *cono.db*.

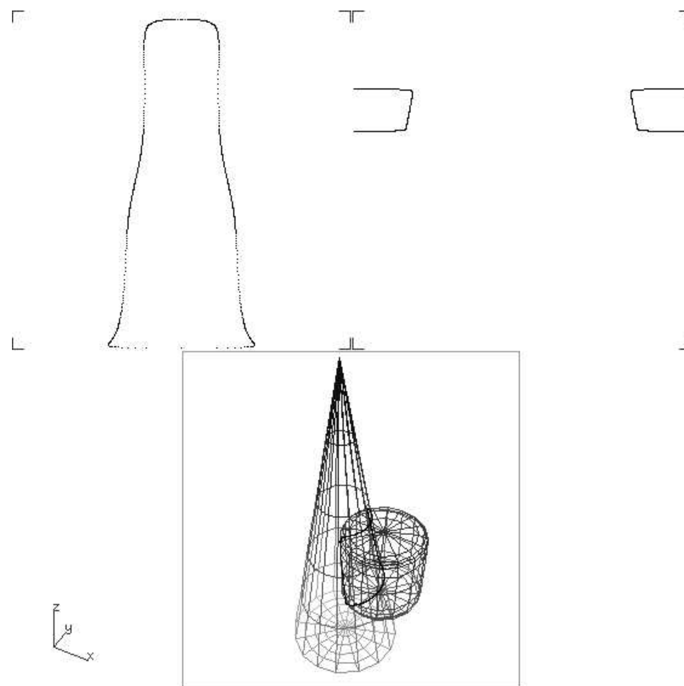

Figura 2.10: *Curve di intersezione nei domini parametrici e in 3D*

Il calcolo dell'intersezione ha avuto successo, quindi possiamo continuare con **Compute**. XCBool calcola l'operazione booleana e al termine riattiva la finestra *Main* con il risultato dell'operazione nella *Object list*, rappresentato dal nuovo oggetto composto a cui è stato assegnato il nome generico *objectX*. Selezioniamolo e diamo i comandi **Show** e **View** in sequenza per visualizzarlo. Per evidenziare la parte sottratta al cilindro in seguito all'operazione booleana, possiamo traslarlo e ruotarlo tramite i pulsanti direzionali nelle sezioni **Camera pos** e **Camera dir**, oltre ad avvicinarlo o allontanarlo tramite i pulsanti **Near** e **Far**.

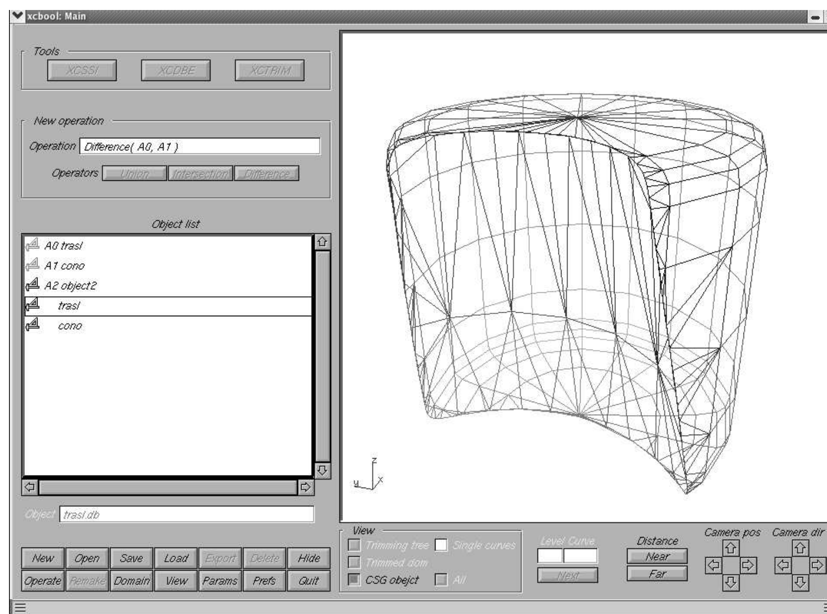

Figura 2.11: *Finestra main con il risultato dell'operazione*

Ora che abbiamo terminato l'operazione, può essere interessante visualizzare le *regioni di dominio trimmate attive* nei due operandi che danno luogo all'oggetto composto. Selezioniamo l'operando interessato (sotto al nuovo oggetto nella *Object list*) e visualizziamone il dominio con il comando **Domain**.

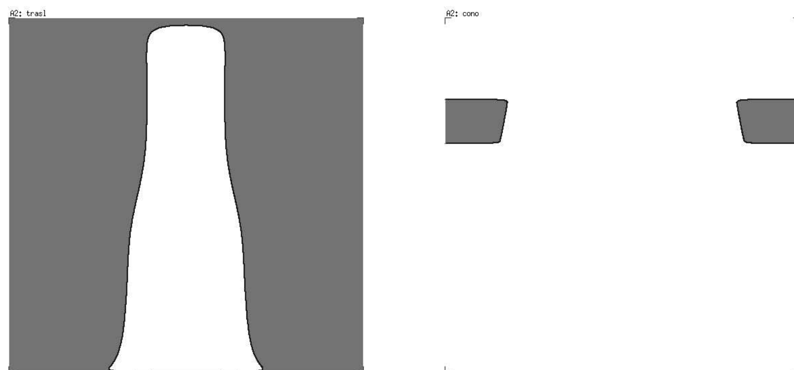

Figura 2.12: *Domini trimmati degli operandi trasl e cono. La parte attiva è quella evidenziata in grigio.*

Per quanto riguarda le potenzialità di resa, abbiamo la possibilità di modificare diversi parametri grazie alla finestra di dialogo **Preferences** (pulsante **Prefs**), scegliendo:

- *wire frame*: una semplice rappresentazione in prospettiva della rete dell'oggetto discretizzato
- *depth cueing*: leggermente più evoluto del wire frame perché disegnato con linee di diversa luminosità, in base alla lontananza dal punto di vista dell'osservatore
- *hidden line*: come il wire frame ma senza disegnare le facce in secondo piano
- *shading*: è il più realistico, perché fa una resa della superficie dell'oggetto in funzione di una *sorgente luminosa*

Oltre alla resa dell'oggetto discretizzato è possibile specificare la bontà della discretizzazione stessa, tramite il pulsante **Params** già descritto in precedenza. Queste opzioni si applicano singolarmente alle superfici trimmate, infatti nel nostro esempio, finché è selezionato l'oggetto composto, il pulsante non è attivo. Si abilita solo selezionando *trasl* o *cono*, per i quali possiamo scegliere tassellazioni diverse, con griglia uniforme o adattiva.

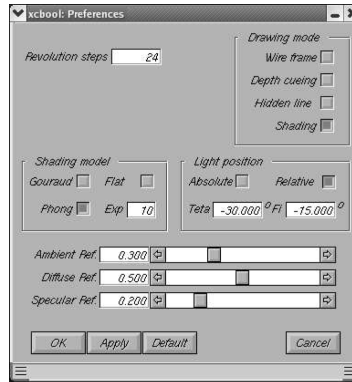

Figura 2.13: *Finestra "Preferences" con i parametri di resa*

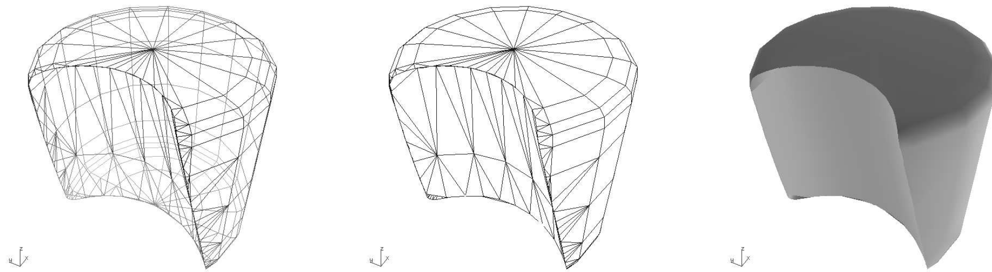

Figura 2.14: *Resa dell'oggetto composto in depth cueing, hidden line e shading; tassellazione di default*

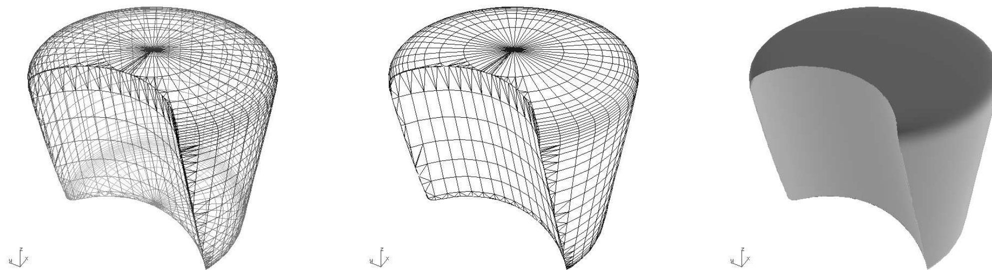

Figura 2.15: *Resa dell'oggetto composto in depth cueing, hidden line e shading; tassellazione più raffinata*

Per ulteriori dettagli sull'utilizzo di XCBool rimandiamo alla lettura di [XCBOOL].

Capitolo 3

Test su XCBool

La fase di testing per analizzare il comportamento di XCBool in termini di stabilità e accuratezza è stata affrontata mediante l'esecuzione di un certo numero e varietà di operazioni, cercando di riprodurre buona parte delle casistiche possibili [BORT96], [SUZZ95]. Vedremo come, in seguito al risultato dei test, è stata pianificata una prima ipotesi di lavoro sul codice.

3.1 Raccolta di test

Il materiale adottato dal nostro programma di analisi consiste in una raccolta di solidi di vario tipo, pertanto (per questioni di ordine) i test sono stati suddivisi in gruppi, in ognuno dei quali è costante un certo solido primitivo, che si combina con altri:

- 18 test per il gruppo *cono*;
- 15 test per il gruppo *cilindro*;
- 11 test per il gruppo *cubo*;
- 15 test per il gruppo *sfera*;
- 9 test per il gruppo *toro*;

- 9 test per il gruppo *effe* (la lettera F ottenuta per estrusione);
- 21 test per il gruppo *varie*, costituito da operazioni su superfici utilizzate in alcuni progetti di modellazione.

Ogni tipo di figura inoltre si presenta in più varianti, di dimensione e posizionamento, per un totale di circa 70 solidi distinti. Il formato dei file invece è lo stesso per tutti gli operandi: si tratta di *file .db*, che secondo la convenzione di XCModel descrivono superfici NURBS non trimmate.

3.1.1 Approccio di testing iniziale

In questa prima fase, nei nostri test eseguiremo principalmente l'operazione di *intersezione* fra solidi. E' una decisione arbitraria: per ora ci è sufficiente per avere un'idea della robustezza del programma. Infatti, in virtù dell'algoritmo illustrato nel paragrafo 1.4.2, la differenza a livello implementativo fra le varie operazioni booleane è di una sola formula, semplice ed altrettanto importante, quindi per il momento supponiamo che le eventuali difficoltà incontrate da questa versione del programma siano altrove, sapendo inoltre che questa versione si comporta correttamente in diverse situazioni già verificate in passato.

Ogni operazione è stata definita con un preciso ordine, quindi tutte le operazioni sono del tipo

$$\textbf{operando1} \cap \textbf{operando2}$$

seguendo la lista di test decisa arbitrariamente. Qualora XCBool esegua correttamente un'operazione, sarà testata la sua inversa, cioè

$$\textbf{operando2} \cap \textbf{operando1}$$

perché nonostante l'intersezione sia un'operazione commutativa, questo non vale più a livello esecutivo. La non commutatività deriva da questioni

numeriche, per le quali non valgono neppure alcune proprietà più banali, come l'associativa dell'addizione o della moltiplicazione.

In base alle nostre conoscenze sull'algoritmo di composizione e su XC-Bool, abbiamo anche gli strumenti per definire *macroscopicamente* quali tipi di problemi possiamo aspettarci dall'esecuzione di un test, o almeno a quale fase appartengono. Sapendo infatti che i due passaggi principali sono:

- intersezione fra superfici, a carico di *xcssi*
- composizione dei domini trimmati, a carico di *xcbool*

possiamo attribuire fin da subito eventuali errori all'uno o all'altro programma. Operativamente, una volta caricati i solidi nel progetto di XCBool, dobbiamo seguire questo percorso:

scelta dell'operazione fra solidi → **Operate** → **Make** → **Compute**

Poiché il pulsante Make lancia il processo xcssi, se dopo avervi cliccato il programma restituisce un errore o non risponde ai comandi (*si congela*, in gergo), sappiamo con certezza che c'è stato un problema nella ricerca delle curve di intersezione fra le superfici.

A questo punto, prima di abbandonare quel test contrassegnandolo come fallito a causa dell'intersecatore, può essere utile variare le tolleranze usate da xcssi, costituite dai già noti valori *SRT1*, *SRT2* e *CRT*. Per questo scopo è utile la già citata possibilità di eseguire xcssi in modo *interattivo*: mentre nel primo tentativo di esecuzione accettiamo i default proposti da xcssi, se questi non funzionano proviamo per tentativi a diminuirli, vale a dire ad aumentare il livello di dettaglio richiesto nella ricerca delle curve, fino ad impostare tolleranze molto piccole (raggiungeremo infatti soglie come SRT1=0.008, SRT2=0.008, CRT=0.003).

Una volta terminata correttamente l'elaborazione di xcssi ed osservata la correttezza delle curve di intersezione ottenute, la seconda possibilità di errore si può verificare in seguito al comando **Compute**, stiamo parlando quindi dell'elaborazione da parte del processo xcbool dei dati forniti da xcssi. Anche in questo caso, potremmo aspettarci vari comportamenti:

- un *congelamento* del programma, ad esempio un loop infinito a causa di test di tolleranza non superati;

- un *crash*, cioè una terminazione del programma inaspettata a causa di un errore, come un "segmentation fault";

- un *solido composto errato* risultante dall'elaborazione, nonostante l'esecuzione del programma non si interrompa.

Come ultima nota, prima di passare alla descrizione del lavoro effettivo, è doveroso segnalare che non tutti i casi di errore sono necessariamente dovuti a *bug* di XCBool, potrebbero anche derivare da situazioni particolari o non previste a cui sottoponiamo il programma.

Discorso analogo per xcssi, anche se il nostro lavoro è dedicato interamente ad xcbool, quindi gli interventi per risolvere eventuali problemi a carico dell'intersecatore si limiteranno ad aggiustamenti delle tolleranze come descritto in precedenza.

Ora che abbiamo sufficiente know-how per attuare e capire le operazioni di test su XCBool, andiamo ad illustrare alcuni esempi che hanno portato a condizioni di errore diverse, per ognuna delle quali è stata identificata una tipologia ben precisa. Questo ci consentirà di ottenere una formalizzazione effettiva degli errori. Distingueremo in tutto tre tipi di problematiche, che per convenzione abbiamo chiamato **ISE**, **RDE** e **CDE**, elencati in ordine crescente di gravità.

3.2 Assenza di intersezioni

Prima delle tre categorie di problemi principali, parleremo di un quarto tipo di errore concettualmente molto banale che si è presentato nei nostri test: l'assenza di curve di intersezione fra le superfici degli operandi.

Se consideriamo gli esempi *cono 6* e *7*, vediamo chiaramente come questi due solidi non abbiano intersezioni. Nel primo caso il toro è completamente *esterno* al cono, nel secondo caso il cilindro è completamente all'*interno* del cono. Questa condizione, che evidentemente non è mai stata prevista e gestita da xcbool, porta ad un crash del programma nella fase di *Compute*, mentre xcssi restituisce correttamente l'assenza di intersezioni.

Formalmente, volendo calcolare le due operazioni di intersezione (in senso di composizione booleana), trattandosi nel primo caso di due insiemi disgiunti, e nel secondo caso $lungo.db \subset cono.db$, dovremmo avere come risultato:

$toro.db \cap cono.db = \emptyset$
$cono.db \cap lungo.db = lungo.db$

ma questo ai fini della modellazione solida non è di grande utilità e costituirebbe una complicazione non giustificata, preferiamo quindi considerare la presenza di intersezioni fra gli operandi come condizione necessaria per eseguire un'operazione.

3.3 Problemi di tipo ISE

*Con problemi di tipo **ISE (Intersezione Superfici Errata)** ci riferiamo all'impossibilità da parte di xcssi di calcolare le curve di intersezione fra gli operandi.*

E' questo il caso di numerosi test del gruppo *cubo*, in cui la fase di *Make*

dà messaggi di errore come *"Failed ssi: curve not closed"*, o cade in un loop infinito, nonostante siano state provate numerose tolleranze.

Di solito la dinamica che abbiamo riscontrato in questi tipi di blocco è la seguente: xcssi non riesce a chiudere le curve di intersezione con le tolleranze di default, ma diminuendo progressivamente le tolleranze succede che dall'errore "curve not closed" passiamo direttamente ad una situazione di congelamento del processo. In tal caso chiudiamo la valutazione del test contrassegnandolo con l'errore ISE.

In altri casi invece, il fallimento dell'intersecatore con i valori di default è stato corretto cambiando le tolleranze, annotandoci di conseguenza i valori funzionanti per quell'operazione. Sono comunque casi poco frequenti perché solitamente, se l'intersecatore funziona, ha immediatamente successo con i valori di default per un buon 90% dei nostri test.

3.4 Problemi di tipo RDE

Per illustrare questo tipo di problematica prendiamo come esempio il test *cono 4*, dove abbiamo *sfera94.db* ∩ *cono_rot1.db*. Questa operazione è eseguita subito correttamente da XCBool, ma la sua inversa, vale a dire *cono_rot1.db* ∩ *sfera94.db*, presenta un errore nella parte di superficie trimmata della sfera, com'è evidente nella figura 3.1, restituendo *sfera94.db* \ *cono_rot1.db*.

Se osserviamo i domini trimmati elaborati da xcbool, vediamo che la lettura delle curve di trimming importate da xcssi è corretta, ed altrettanto la suddivisione in regioni del dominio. C'è stato quindi un errore di valutazione che porta a contrassegnare come attiva la parte di dominio complementare a quella che ci aspettavamo.

Provvediamo quindi a classificare questo tipo di problematica ed identificarla nel caso si ripresenti in futuro:

Definiamo problemi di tipo ***RDE (Regione di Dominio Errata)*** *i casi in cui xcbool, in seguito ad una corretta chiusura delle curve di dominio fornite da xcssi, sceglie la parte di superficie trimmata sbagliata in uno o più operandi nel comporre il solido CSG risultante.*

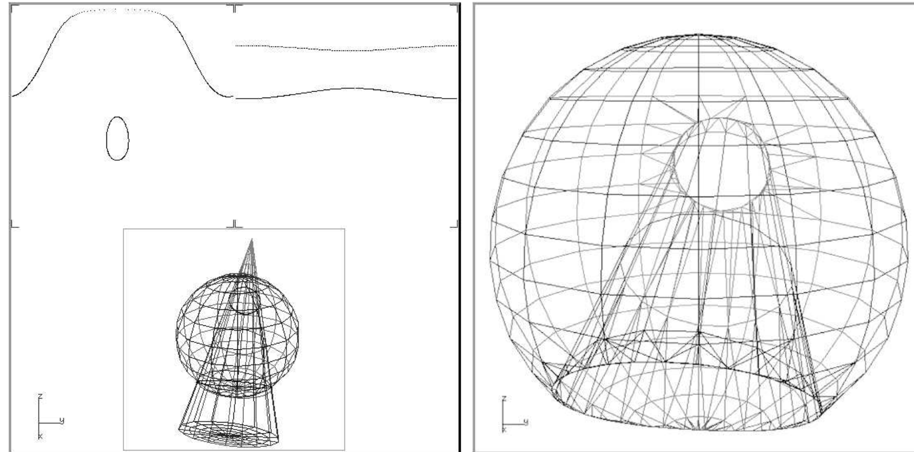

Figura 3.1: *Risultato errato dell'operazione cono_rot1.db* $\cap$ *sfera94.db*

Altri problemi RDE li troviamo ad esempio in alcuni test in cui compare il solido F, come *effe 1* e *toro 8* (figura 3.2).

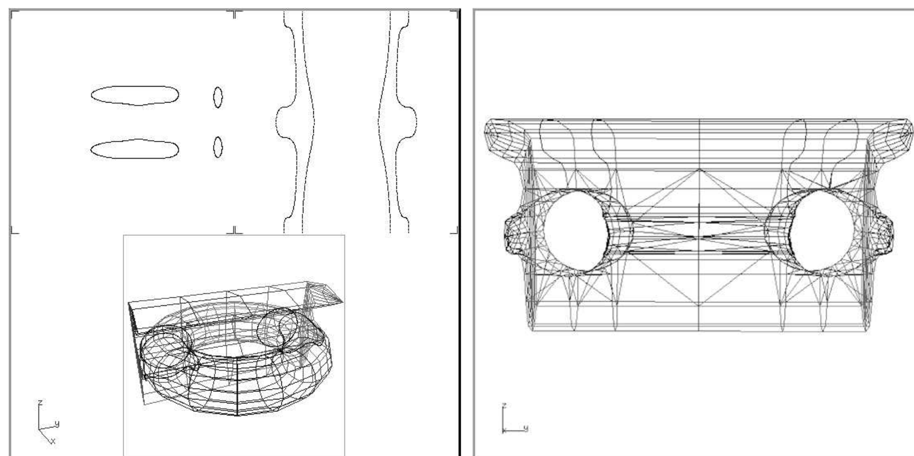

Figura 3.2: *Risultato errato dell'operazione gcf2.db* $\cap$ *toro.db*

3.5 Problemi di tipo CDE

Nel primo test del gruppo *cono*, vale a dire l'operazione *cono.db* $\cap$ *cilindro.db*, osserviamo un risultato inatteso: il solido ottenuto dall'operazione è *cilindro* $\setminus$ *cono*, le curve di intersezione fra i solidi però sono calcolate correttamente (figura 3.3). Abbiamo quindi a che fare con un problema RDE? Apparentemente sì, ma andiamo a visualizzare in che modo xcbool ha elaborato le curve di intersezione dei domini (figura 3.4): risultano chiaramente errate, in particolare la regione inferiore del dominio trimmato del cilindro non è stata chiusa correttamente negli estremi, infatti nel CSG risultante è proprio la parte attiva del cilindro trimmato ad essere sbagliata.

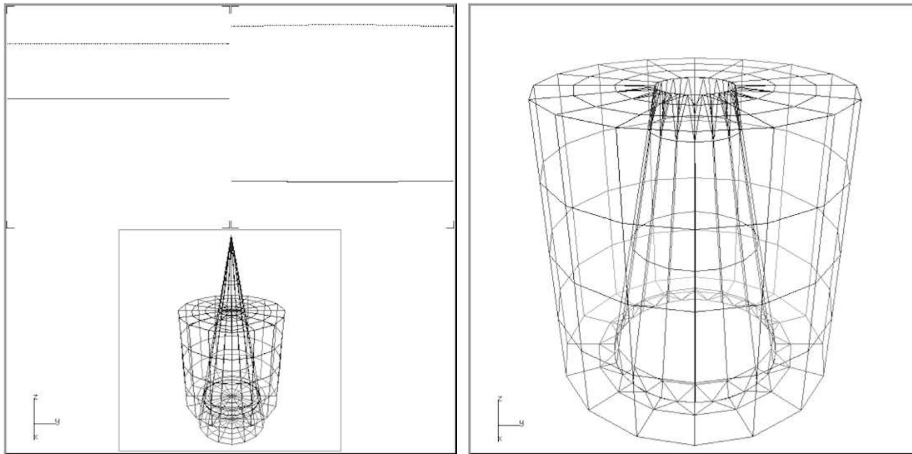

Figura 3.3: *Intersezioni e risultato dell'operazione cono.db* $\cap$ *cilindro.db*

Trattandosi quindi di un errore diverso dal precedente RDE, diamone una definizione propria:

Chiameremo problemi di tipo ***CDE (Chiusura Domini Errata)*** *gli errori di elaborazione da parte di xcbool delle curve di intersezione fra solidi, nonostante queste siano calcolate correttamente da xcssi, portando ad un'errata definizione delle regioni di dominio trimmato di uno o più operandi.*

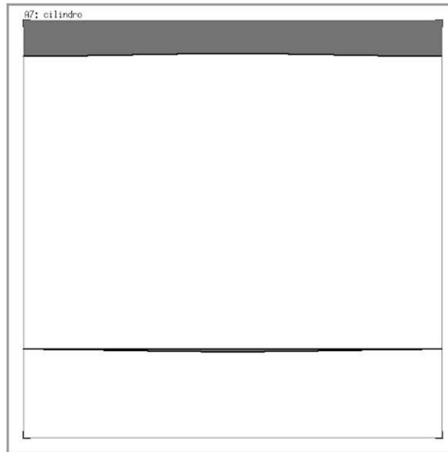

Figura 3.4: *Errata chiusura della regione di dominio in cilindro.db*

Altro esempio di errore CDE si trova nel test *cono 8*, cioè *cubo94_1.db* $\cap$ *cono.db* (figura 3.5), che presenta una chiusura di dominio analogamente errata negli estremi. In questo esempio, a differenza del primo caso CDE descritto, XCBool non consente neppure la visualizzazione dei solidi composti, perché a causa delle troppe intersezioni nella regione di dominio sbagliata otteniamo il messaggio di errore *"Invalid trimming grid"*.

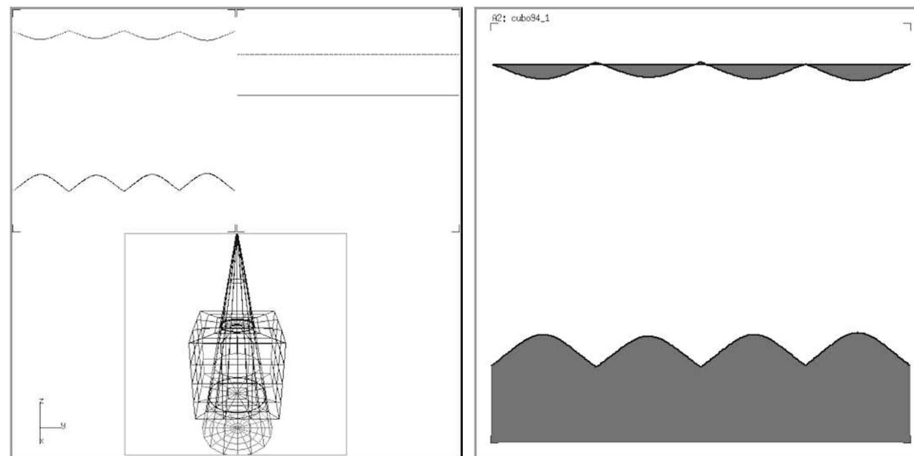

Figura 3.5: *Errata chiusura della regione di dominio in cubo94_1.db*

Problema più complesso, ma sempre di tipo CDE, si è presentato al test *cono 14*. Eseguito nell'ordine originale, cioè *cono_rot1.db* ∩ *toro.db*, xcbool va in loop nell'operazione di Compute; in ordine inverso otteniamo un risultato completamente sbagliato (figura 3.6): in questo caso l'errata chiusura non è presente solo agli estremi della curva di trimming, ma c'è una confusione più ampia nella lettura/rielaborazione dei punti che definiscono le curve di trimming.

Ci riferiremo a questo tipo di problemi definendoli ***CDE gravi****.*

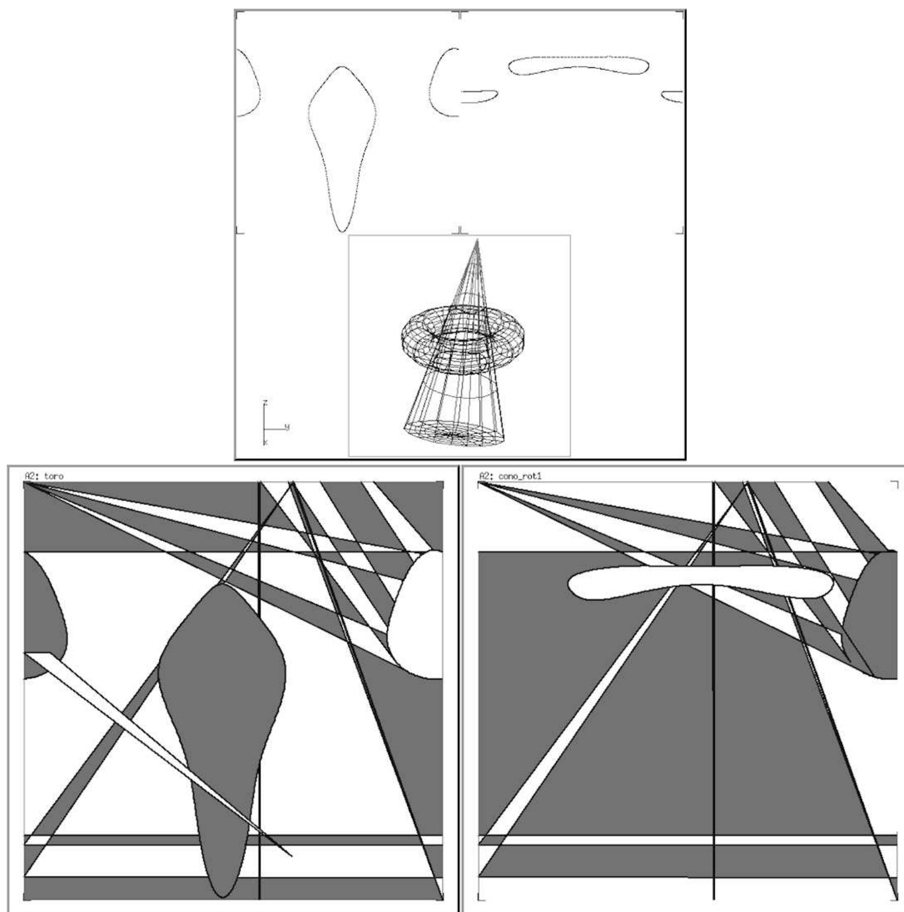

Figura 3.6: *Errore CDE grave nei domini di toro.db e cono_rot1.db*

3.6 Panoramica sulla versione attuale di XC-Bool

Inquadrati i problemi e le inesattezze che sono venuti alla luce in seguito ai nostri test, possiamo supporre che a questa versione di XCBool manchi qualche raccordo che le consenta di calcolare correttamente il solido composto in situazioni specifiche, legate principalmente a:

- figure solide particolari;
- composizioni di solidi con bordi a contatto, che generano *punti singolari*;
- valutazione delle regioni di dominio errate a causa di tolleranze numeriche non ottimali (intuibili dagli errori RDE e CDE);
- incompletezza nell'importazione delle curve da xcssi (intuibili dagli errori CDE gravi).

Con la fase di studio e debug che illustreremo nel prossimo capitolo cercheremo di fare maggiore chiarezza e risolvere più problemi possibili, oltre ad identificare le casistiche non contemplate nella progettazione stessa di XCBool.

Capitolo 4

Aggiornamento di XCBool

Cercheremo ora di capire e approfondire l'origine delle problematiche rilevate nel precedente lavoro di testing, attraverso lo studio dei *sorgenti* di xcbool, associando le varie parti di codice alle funzionalità che implementano.

In seguito affronteremo la fase di *debug e aggiornamento* nella quale saranno apportati alcuni miglioramenti, sia in termini di stabilità che di capacità di elaborazione.

4.1 Struttura del codice di XCBool

Prima di seguire passo-passo l'esecuzione del programma, nonostante le facilitazioni dello strumento di debug utilizzato, è necessario conoscere la struttura del codice quantomeno nei punti salienti. Nel mio percorso ho preferito iniziare con un'analisi *top-down*, identificando l'ambito implementativo di ogni file sorgente in C.

4.1.1 Panoramica sui sorgenti

Segue la lista di sorgenti di xcbool con una breve descrizione (in ordine alfabetico):

- `csg-objects.c`: gestione delle strutture per la lettura e manipolazione degli oggetti CSG del progetto;
- `error.c`: gestione degli errori ad alto livello;
- `formats.c`: utility di conversione fra formati numerici e lettura di dati da file;
- `int-controlGUI.c`: definizione dell'interfaccia utente (finestra e widget) della finestra Operate;
- `intlist.c`: utility per le strutture che memorizzano le intersezioni fra superfici;
- `isinside.c`: algoritmo di raytracing per il test interno/esterno di un punto in un solido;
- `nurbs-bool.c`: esecuzione del processo *xcssi* e del calcolo delle operazioni booleane;
- `nurbs-curves.c`: elaborazione delle curve di trimming del dominio parametrico, importate da xcssi;
- `nurbs-surfaces.c`: caricamento e salvataggio di superfici NURBS, valutazione di un punto sulla superficie date le coordinate u e v;
- `points.c`: funzioni relative a punti, segmenti e bounding-boxes;
- `region.c`: gestione delle regioni di dominio negli alberi CSG;
- `trimmed-surfaces.c`: funzioni principali per il calcolo dell'operazione booleana;
- `visual.c`: resa grafica degli oggetti;
- `xcbool.c`: funzioni di startup (contiene il `main`) per il lancio di XCBool in modalità a riga di comando o interattiva;
- `xcboolGUI.c`: definizione dell'interfaccia utente principale;

4.1.2 Funzioni principali

Scendiamo ora più in dettaglio nel codice studiando le *funzioni* principali.

Poiché il processo di creazione di un nuovo progetto, il caricamento degli operandi e in generale tutta la serie di operazioni prima di cliccare sul pulsante **Operate** esula dal mio piano di lavoro (non presentando alcun problema), cerchiamo di identificare direttamente l'evento generato da tale pulsante, che trattandosi di un elemento della GUI principale ci aspettiamo si trovi in `cboolGUI.c`. Infatti nella funzione `CreateWMain` troviamo una lunga serie di widget definiti con l'ausilio della libreria *xtools*, fra i quali il pulsante `BTOPERATE_WM`, che è associato alla funzione `WMain_operate` tramite la riga: `SetButtonBRScript(WMain, BTOPERATE_WM, WMain_operate)`. In `WMain_operate`, dopo un controllo sui tipi di superfici caricate come operandi, troviamo `nurbs_bool_init(file1, type1, file2, type2, &boolop)`, la quale carica le superfici degli operandi nella finestra *Operate*, che viene inizializzata e mostrata a video rispettivamente da:
`WOperate_init()` e `ShowWindow(&WOperate)`.
Segue direttamente la chiamata a `WMain_select()` che ripristina la finestra principale: significa che tutta l'esecuzione del programma, finché non si è conclusa l'operazione booleana, è ora a carico della finestra *Operate*.

Passiamo quindi al file `int-controlGUI.c` dove si trova la gestione dell'evento *Make*, a cui è associata la funzione `WOperate_make`. Seguendo il flusso esecutivo, ci imbattiamo nella funzione in cui viene chiamato il processo xcssi: la `make_intersection`, nel file `nurbs-bool.c`. Terminata l'interazione con xcssi, viene abilitato il pulsante **Compute** con `WOperate_compute_enable()`.

Nell'evento *Compute* abbiamo un'importante successione di chiamate:

`boolop_compute` → `nurbs_bool_compute` → `trimmed_surface_compute`

Quest'ultima, situata in `trimmed-surfaces.c`, implementa il nocciolo dell'*algoritmo di composizione booleana*.

Per ora come livello di dettaglio ci è sufficiente per affrontare il primo problema: l'assenza di intersezioni fra gli operandi.

4.2 Gestione dell'assenza di intersezioni

Per evitare il crash dell'applicazione quando si tenta di comporre due solidi che non hanno parti in comune, è necessario aggiungere una *patch* che comunichi all'utente l'impossibilità di proseguire, nel caso in cui questi tenti di dare il comando *Compute* nonostante la fase di *Make* precedente non abbia effettivamente prodotto alcuna curva di intersezione.

Per implementare l'errore "assenza di intersezioni" è bene seguire il paradigma già sviluppato in XCBool per la gestione degli errori, pertanto lavoreremo innanzitutto sul file `error.h`, definendo il nuovo tipo di errore:

aggiungiamo all'enumerazione `_ERR_Tag` la nuova costante `_ERR_NOINTERSECTION` e il corrispondente messaggio di errore verbale `"No intersection found"` nel vettore `_ERR_Msg[]`. Infine, la macro per inizializzare la struttura globale che contiene l'errore attivo da gestire:

```
#define ERROR_NOINTERSECTION() error_setmsg( __FILE__,__LINE__,
_ERR_NOINTERSECTION, _ERR_Msg[_ERR_NOINTERSECTION] )
```

Ora dobbiamo individuare il punto del codice dal quale rilevare questa condizione di errore. Abbiamo visto come tutta la gestione della composizione booleana sia a carico della `trimmed_surface_compute`. Al suo interno, fra le operazioni iniziali, troviamo la chiamata a:
`get_int_curves(pi->surf, pj->surf, &pc)`
funzione che si occupa appunto di ottenere le coordinate delle curve di intersezione che xcssi ha prodotto nel file *.int*. I parametri `pi` e `pj` sono puntatori a superfici NURBS, mentre `pc` è un vettore che contiene le curve di trimming nei domini parametrici delle superfici.

Prima di uscire dalla funzione restituendo `_ERR_NONE`, controlliamo che le liste di curve di intersezione dei due operandi non sia nulla, restituendo in tal caso l'errore specifico:

```
if (curves[0].c1 == NULL || curves[0].c2 == NULL)
{
.     ERROR_NOINTERSECTION();
.     return;
}
```

Tale macro, analogamente al costrutto *try ... catch* dei linguaggi più ad alto livello, sarà propagato dalle funzioni chiamanti con `RETURN_ON_ERROR()`, fino a raggiungere `boolop_compute` che si trova direttamente nella sezione GUI:

```
if (ISERROR())
.     nurbs_bool_free(&boolop);
ERROR_HANDLER(TRUE);
```

in cui la macro `ERROR_HANDLER` mostra la finestra con il messaggio di errore da noi definito. Nella figura 4.1 possiamo osservare come reagisce ora il programma nel test *cono 6*, che in precedenza portava al crash:

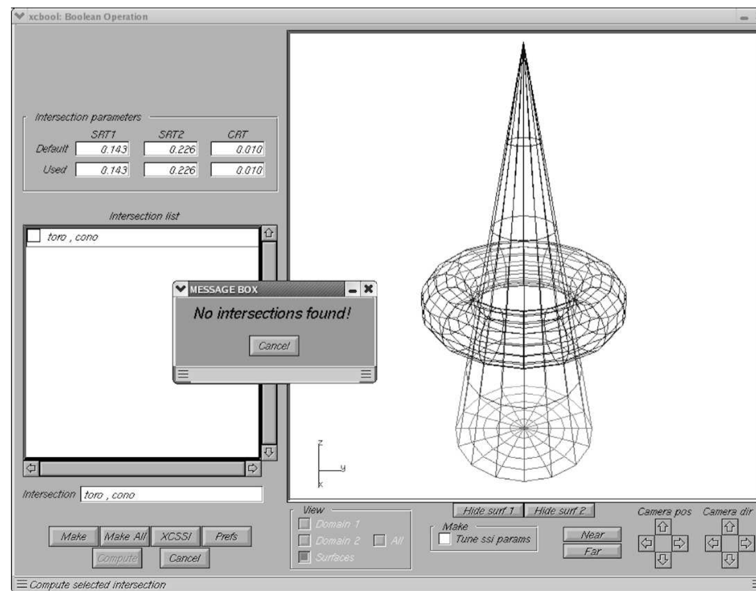

Figura 4.1: *Esempio di applicazione dell'errore "No intersection found"*

4.3 Struttura dei file .int

Competenza fondamentale per l'analisi dei problemi CDE e RDE, è conoscere la struttura dei file *.int* prodotti da xcssi, che contengono le curve di intersezione. Sono scritti in formato testo (*ASCII*), questo ne facilita anche la lettura aprendo il file con un semplice editor.

Ogni file .int contiene i seguenti campi:

- **Surface Names:** nome delle superfici coinvolte
- **Intersection Params**: parametri di tolleranza usati
- **Number of Curves**: numero di curve di intersezione
- per ogni curva di intersezione x abbiamo:
  - **Number of Points for Curve *x***: numero di punti appartenenti alla curva
  - **Break on Surface Domain 1**: indici dei punti della curva in cui si hanno *discontinuità* nel dominio 1
  - **Break on Surface Domain 2**: indici dei punti della curva in cui si hanno *discontinuità* nel dominio 2
  - **Domain Points of Curve *x***: coordinate dei punti che costituiscono la curva di trimming numero x

Le coordinate dei punti sono scritte in questa forma matriciale:

$$\begin{matrix} x_{1,1} & y_{1,1} & x_{2,1} & y_{2,1} \\ \vdots & \vdots & \vdots & \vdots \\ x_{1,n} & y_{1,n} & x_{2,n} & y_{2,n} \end{matrix}$$

dove n è il numero di punti della curva; le coppie $x_{1,x}$, $y_{1,x}$ sono le coordinate dei punti della curva nel dominio del *primo* operando; $x_{2,x}$, $y_{2,x}$ sono

le coordinate dei punti della corrispondente curva di intersezione nel dominio del *secondo* operando.

Il concetto di *discontinuità* consiste in un'*interruzione* nel flusso di punti che descrivono una curva. Infatti le coordinate dei punti sono elencate in ordine ben preciso nella matrice, seguendo la direzione della curva stessa, ma quando xcssi nella sua ricerca di intersezioni termina un segmento di curva, segnala la discontinuità aggiungendo l'indice del punto corrente alla lista "Break on Surface Domain x". Nei casi più complicati una curva può essere suddivisa anche in tre segmenti distinti, che xcbool ha il compito di riordinare. Per ora ci è sufficiente considerare il caso più semplice di un segmento unico di curva.

Ricordiamo infine che le curve di trimming sono *NURBS di grado 1*, quindi non si ha alcuna approssimazione lineare ma linee spezzate. E' compito dell'intersecatore trovare un numero sufficiente di punti in modo da ottenere una buona approssimazione di tali curve.

4.4 Studio dei problemi CDE semplici

Cerchiamo ora di capire l'origine del problema CDE più banale, che consiste in una chiusura scorretta dei bordi delle curve di trimming nel dominio, generando regioni di trimming errate. Nella fase di analisi precedente abbiamo incontrato un esempio tipico nel test *cono 1*.

Trattandosi di un'errata elaborazione delle coordinate lette dal file .int, iniziamo con l'esaminare questo file nel caso dell'operazione *cono.db* ∩ *cilindro.db*, vale a dire il file `cono_cilindro.int`:

```
Surface Names:
cono.db
cilindro.db
Intersection Params SRT1=0.050000 SRT2=0.050000 CRT=0.005000
```

```
Number of Curves
2
Number Points for Curve 1
1260
Break on Surface Domain 1
1259
Break on Surface Domain 2
1259
Domain Points of Curve 1
1.000000e+00 5.833337e-01 1.000000e+00 2.114180e-01
9.989004e-01 5.833337e-01 9.989164e-01 2.114196e-01
9.974940e-01 5.833337e-01 9.975312e-01 2.114217e-01
⋮
1.754023e-03 5.833338e-01 1.728498e-03 2.114194e-01
8.770194e-04 5.833337e-01 8.642598e-04 2.114195e-01
0.000000e+00 5.833337e-01 3.738744e-08 2.114196e-01
```

Questa prima curva di trimming, nel dominio di *cilindro.db* (quindi la seconda coppia di coordinate, terza e quarta colonna), crea problemi ad xcbool nel chiudere la corrispondente regione di dominio trimmato. Invece la seconda curva:

```
Number Points for Curve 2
503
Break on Surface Domain 1
502
Break on Surface Domain 2
502
Domain Points of Curve 2
4.967080e-09 8.333338e-01 0.000000e+00 9.135805e-01
2.170916e-03 8.333337e-01 2.093560e-03 9.135807e-01
```

```
4.341626e-03 8.333337e-01 4.187312e-03 9.135814e-01
⋮
9.951414e-01 8.333337e-01 9.953140e-01 9.135816e-01
9.977502e-01 8.333337e-01 9.978303e-01 9.135807e-01
1.000000e+00 8.333337e-01 1.000000e+00 9.135805e-01
```

non dà alcun problema. Poniamo la nostra attenzione sul flusso di punti: si noti che nella prima curva, la coordinata x parte da 1.0 e decresce fino a $\sim 3.7 * 10^{-8}$, cioè quasi zero.

Se proviamo ad editare manualmente il file .int, sostituendo il $3.738744e-08$ con $0.000000e+00$ per poi proseguire normalmente con **Compute**, vediamo che questa volta xcbool non ha problemi a "capire" che anche il secondo estremo della curva si trova sul bordo esatto del dominio, chiudendo correttamente la regione e *producendo il risultato corretto dell'operazione.*

Notiamo però che anche la *seconda* curva di trimming nel dominio di *cono.db* presenta una situazione analoga: il valore dell'ascissa va da 1.0 a $\sim 5.0*10^{-9}$, senza raggiungere lo 0 esatto. Per quale motivo questa situazione non dà invece alcun problema?

Se studiamo il codice in `nurbs-curves.c` troviamo la costante:

```
#define SOGLIA 1.0e-8
```

che viene utilizzata nella funzione `nurbs_curve_close` prima di chiudere la regione di dominio trimmato. In particolare, le coordinate degli estremi di una curva di trimming sono approssimate a 0 o 1 nel caso in cui la distanza da questi valori siano inferiori a `SOGLIA`. Questo ci fa comprendere immediatamente come $5.0 * 10^{-9}$ sia approssimato a 0, ma questo non accada per $3.7 * 10^{-8}$.

In seguito a numerosi tentativi sui test che presentavano il problema CDE semplice, abbiamo stabilito una tolleranza ottimale di $1.0*10^{-3}$. Con tolleranze inferiori permane il problema in alcuni test del gruppo *cubo*, con tolleranze

superiori il problema nasce in alcuni test del gruppo *varie*.

Nonostante non ci siano regole precise per calcolare la tolleranza ottimale (e neppure dimostrare che questa esista), possiamo affermare di aver trovato un buon compromesso che ci ha consentito di *correggere gli errori CDE semplici da tutti i nostri test*.

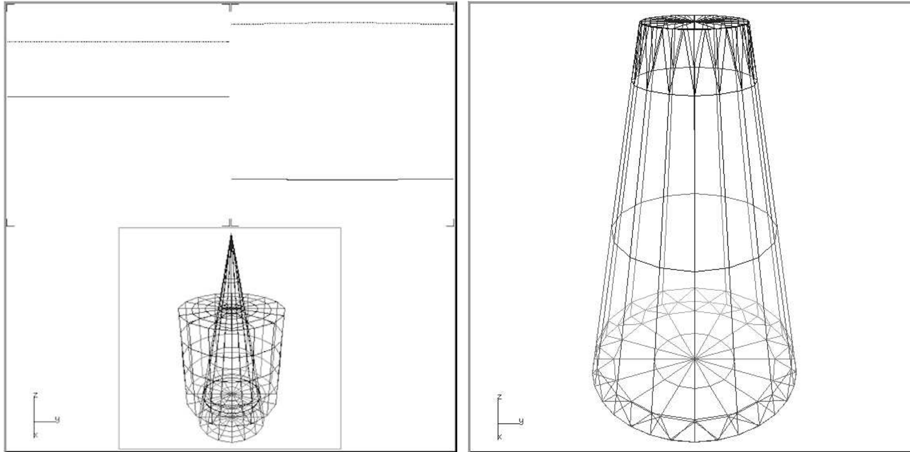

Figura 4.2: *Operazione cono.db* $\cap$ *cilindro.db eseguita correttamente*

4.5 Studio dei problemi CDE complessi

La soluzione a questo tipo di problematica è chiaramente più difficile da attuare rispetto ad un valore di tolleranza, e va ricercata nelle funzioni che lavorano sulle curve di trimming prima della loro chiusura da parte della `nurbs_curve_close` che abbiamo incontrato nel problema precedente.

Nella funzione `get_int_curves`, dove sono letti e rielaborati i dati dal file .int, vediamo che le liste di curve di trimming sono restituite dalla funzione:

```
nurbs_curve *nurbs_curve_adjust(nurbs_curve *c, long *sing)
```

che si trova in `nurbs-curves.c`. E' una funzione piuttosto articolata il cui compito consiste nel *riordinare le curve di trimming dei domini parametrici*,

in base alle *discontinuità* prodotte nel calcolo delle coordinate da parte di xcssi.

4.5.1 Gestione di una discontinuità

Una curva di trimming caratterizzata da *una discontinuità* nella matrice delle coordinate dei suoi punti può identificare *una e una sola curva continua.* E' questo il caso dell'esempio citato nel problema CDE semplice (*cono 1*). Possiamo generalizzare questa situazione come in figura 4.3, dove p_1 ... p_n sono i punti della matrice di coordinate, che coincidono con gli estremi della curva stessa; d_1 è il primo (e unico) punto di discontinuità. Nel caso in cui d_1 coincida con il penultimo punto della lista, l'ultimo è una ripetizione del primo e viene eliminato da xcbool. Le ragioni di questa ripetizione derivano dall'algoritmo di xcssi che cerca di chiudere le curve di intersezione.

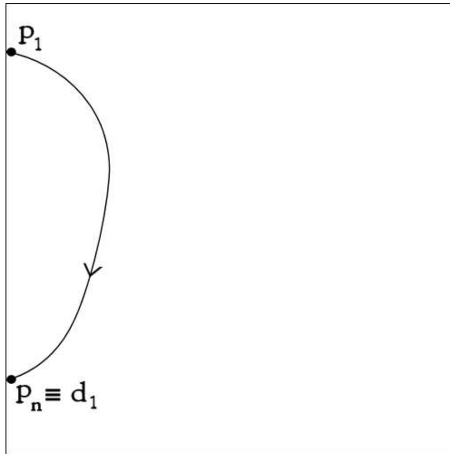

Figura 4.3: *Esempio di curva con una discontinuità*

4.5.2 Gestione di due discontinuità

Una curva di trimming caratterizzata da *due discontinuità* nella matrice delle coordinate dei suoi punti può identificare *una o due curve continue distinte.* Possiamo generalizzare le due casistiche come in figura 4.4.

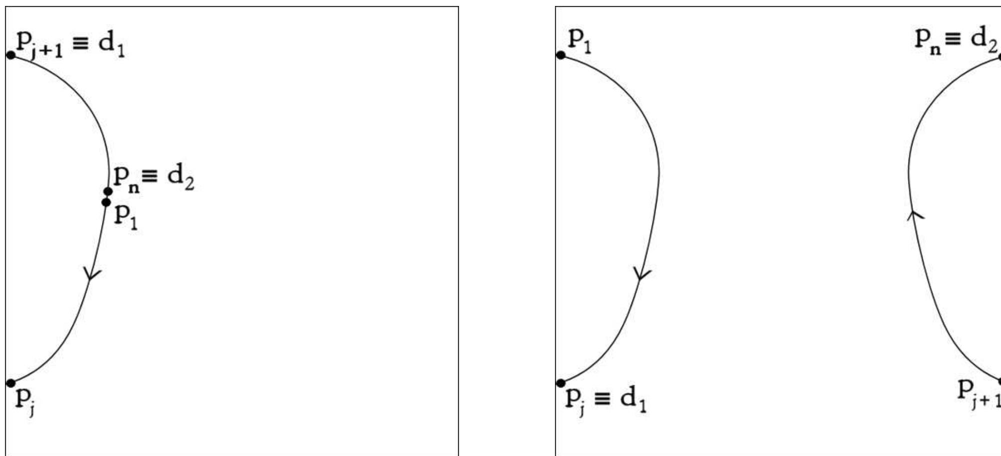

Figura 4.4: *Esempi di casistiche di curve con due discontinuità*

Nel primo caso, la duplice discontinuità nasce dal fatto che lo starting point p_1 non è stato rilevato da xcssi all'estremo della curva, ma al suo interno. Di conseguenza, p_1 e p_n non si troveranno sul bordo del dominio: è proprio questa la strategia algoritmica adottata per il riconoscimento di questa casistica:

```
if (!ONBORDER(CP(c,0)) && !ONBORDER(CP(c, CPN(c)-1)))
{
.     clist = nurbs_curve_reorder(c, sing[1], sing[2]);
.     clist->next = NULL;
}
```

dove `CP(c,0)` e `CP(c, CPN(c)-1)` sono rispettivamente p_1 e p_n; il vettore

di coordinate `c` viene riorganizzato con la `nurbs_curve_reorder`, che applica un algoritmo di *swap* dei due segmenti di curva, in particolare il range indicato (cioè dalla discontinuità 1 alla 2) viene spostato all'inizio del vettore, in modo che il primo punto sia l'estremo iniziale della curva, e l'ultimo sia l'estremo finale, proprio come il caso di una sola discontinuità. La *lista di curve* `clist` restituita contiene quindi una sola curva.
Nota: `sing` è il vettore delle discontinuità, perché nel codice viene utilizzato il termine *singolarità*. Qui abbiamo preferito *discontinuità* per una maggiore correttezza e per evitare confusione con il concetto di *punto singolare* di cui parleremo nell'ultimo capitolo.

Nel caso in cui p_1 e p_n si trovino già sul bordo del dominio, la duplice discontinuità individua *due* segmenti di curva disgiunti che hanno entrambi gli estremi sul bordo. La matrice di coordinate in questo caso è già ordinata, non resta quindi che effettuare uno *split* del vettore di coordinate `c` nei punti di discontinuità, separando le due curve da p_0 a d_1 e da d_1 a d_2:

```
clist = curve_split(c, 0, sing[1]);
clist->next = curve_split(c, sing[1], sing[2]);
clist->next->next = NULL;
```

La *lista di curve* `clist` restituita in questo caso contiene due elementi.

4.5.3 Gestione di più di due discontinuità

Una curva di trimming caratterizzata da *m > 2 discontinuità* nella matrice delle coordinate dei suoi punti può identificare *m o m−1 curve continue distinte.* In figura 4.5 abbiamo un esempio di 3 discontinuità.

La sezione di codice responsabile di questa gestione aveva diverse man-

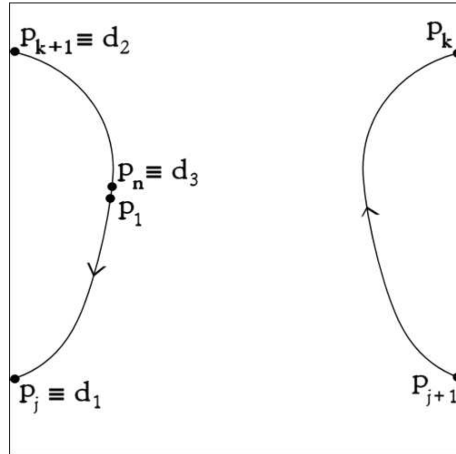

Figura 4.5: *Esempio di curve con tre discontinuità*

canze e imprecisioni, pertanto è stato riscritto completamente l'algoritmo. La strategia con cui elaborare le possibili casistiche è analoga a quella applicata alle due discontinuità:

sia m il numero totale di discontinuità: se p_1 e p_n si trovano sul bordo del dominio, è sufficiente uno *split* del vettore per separare le curve, ottenendo quindi m curve distinte.

Invece, se p_1 e p_n non sono sul bordo, occorre spostare le coordinate dell'ultimo segmento di curva (dai punti d_{m-1} a d_m) all'inizio del vettore, utilizzando nuovamente la funzione `nurbs_curve_reorder`:

```
if (!ONBORDER(CP(c,0)) && !ONBORDER(CP(c, CPN(c)-1)))
{
.     nurbs_curve_reorder(c, sing[(*sing)-1], sing[*sing]);
```

In seguito a questa operazione occorre aggiornare gli indici dei punti di discontinuità d_1 ... d_{m-1} aggiungendo l'offset dovuto allo swap precedente, ed eliminando l'ultima discontinuità d_m:

```
.     for (i=1; i<*sing; i++)
.         sing[i] += sing[*sing] - sing[(*sing)-1];

.     (*sing)--;
}
```

In questo modo otteniamo tutte curve ordinate come nel primo caso, quindi procediamo con lo *split* delle $m-1$ curve distinte:

```
clist = ctmp = curve_split(c, 0, sing[j]);

while (j<*sing)
{
.     i = j;
.     j++;
.     ctmp->next = curve_split(c, sing[i], sing[j]);
.     ctmp = ctmp->next;
}
ctmp->next = NULL;
```

Questa patch del codice ci ha consentito di *correggere quasi tutti gli errori CDE complessi*, fra i quali il test *cono 14* (figura 4.6) di cui avevamo parlato nel capitolo precedente. I rimanenti sono dovuti sostanzialmente a situazioni particolari come la presenza di punti singolari dovuti a bordi coincidenti nei solidi che si intersecano.

Ultima nota nell'aggiornamento di questa sezione di codice riguarda nuovamente le tolleranze e l'ottimizzazione dei dati provenienti da xcssi. In particolare, prima di gestire le discontinuità, sono state aggiunte alcune elaborazioni preliminari:

- l'approssimazione mediante la costante di tolleranza `SOGLIA` (che è stata

di conseguenza tolta dalla successiva `nurbs_curve_close`, ottenendo un'accuratezza numerica superiore)

- eventuali ripetizioni di punti con medesime coordinate sono state cancellate perché possono portare ad errori nelle operazioni successive

- vincoli d'integrità sugli indici dei punti di discontinuità

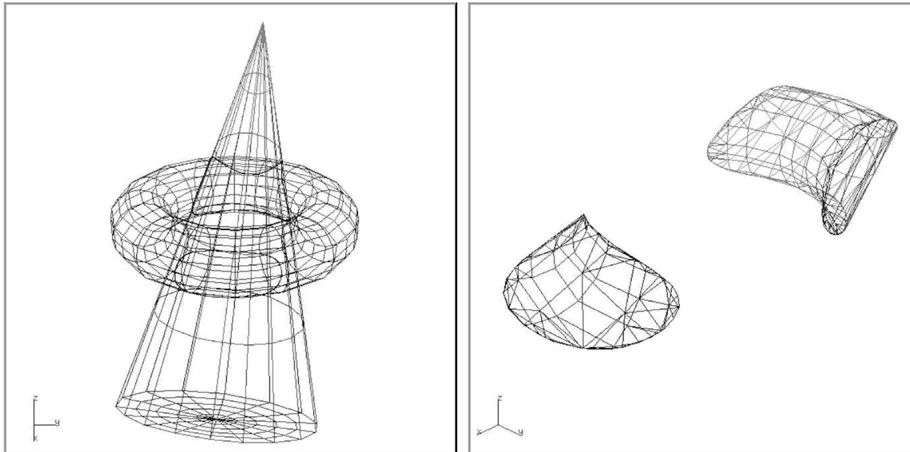

Figura 4.6: *Operazione cono_rot1.db ∩ toro.db eseguita correttamente*

4.6 Studio dei problemi RDE

I problemi di tipo RDE non sono molti ma comunque importanti, perché si sono verificati in casi di operazioni semplici. La ragione per cui a volte xcbool sceglie la parte di solido trimmato sbagliata, può essere di natura algoritmica o numerica.

Vedremo in particolare come lo studio dell'operazione *cono 4* abbia rivelato un bug importante, e come la scelta arbitraria di un punto contenuto in una regione di dominio possa essere ottimizzata per evitare errori numerici.

4.6.1 Test interno/esterno

Nel file sorgente `trimmed-surfaces.c`, che come sappiamo contiene l'implementazione della parte centrale dell'algoritmo di composizione, troviamo la funzione `trimmed_patch_set_domain`. Questa ha il compito di effettuare i test di interno/esterno come descritto nel capitolo 1, rendendo attiva la regione trimmata corretta in ogni operando. Descriviamone la sezione più importante:

```
get_inner_point2(reg->log.last, &p);
nurbs_surface_value(s, p.y, p.x, &p3);
test = trimmed_surface_point_contained(r, &p3);
```

la funzione `get_inner_point2` sceglie arbitrariamente il punto `p` contenuto nella regione di dominio trimmato più in profondità `reg->log.last` dell'operando `s`. Viene poi calcolato il valore della superficie `s` in quel punto `p`, per verificare che sia o meno interna all'operando `r`, tramite la funzione `trimmed_surface_point_contained`.

Algoritmicamente è giusto, ma gli strumenti di debug utilizzati hanno evidenziato subito che nell'operazione *cono_rot1.db* ∩ *sfera94.db*, nel dominio trimmato della sfera, il puntatore `reg->log.last` non fa riferimento alla curva 2 ma alla curva 1 (figura 4.7), quindi non punta alla regione più in profondità come dovrebbe.

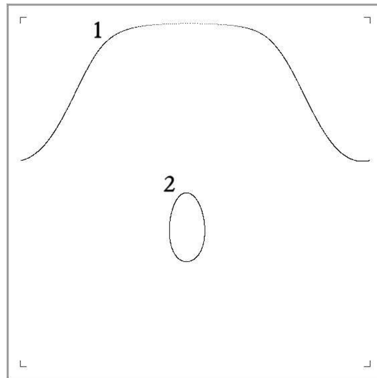

Figura 4.7: *Due regioni di dominio trimmato annidate*

Correggere all'origine il problema che porta ad un valore errato in quel puntatore richiederebbe un certo tempo. Considerando che non vi sono altre sezioni di codice che fanno uso di `reg->log.last`, abbiamo preferito ricercare da zero la curva di maggiore profondità, navigando nell'albero delle regioni di dominio trimmato:

```
it = reg->root;
last = it->curve;

while (it->rg_next != NULL)
{
.     it = it->rg_next;
.     last = it->curve;
.     depth++;
}
```

In seguito a questa patch, l'operazione *cono_rot1.db* ∩ *sfera94.db* è andata a buon fine.

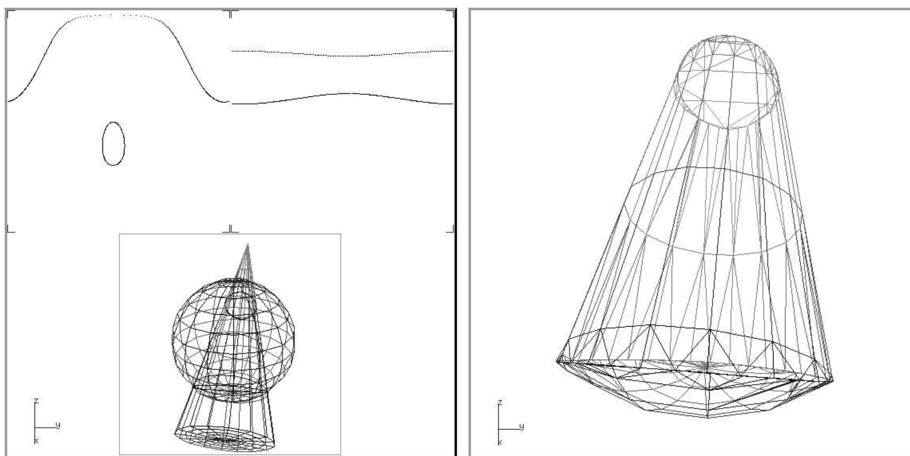

Figura 4.8: *Operazione cono_rot1.db ∩ sfera94.db eseguita correttamente*

4.6.2 Nuova versione di *get_inner_point2*

Abbiamo visto che la funzione `get_inner_point2` cerca un punto interno alla regione di dominio passata come parametro.

L'algoritmo utilizzato consiste nella creazione di una griglia all'interno del *bounding-box* che contiene la regione, testando ciclicamente ogni punto della griglia finché questi non risulta interno (tramite la `nurbs_point_contained`, che applica l'algoritmo di Jordan come descritto nel capitolo 1). Per ottenere una precisione maggiore, la funzione attende di trovare *quattro* punti adiacenti della griglia che risultino tutti interni, restituendo il punto medio a questi come risultato finale.

Tale algoritmo però, oltre a non garantire al 100% la correttezza della risposta, non implementa neppure la soluzione più ottimale. Può capitare infatti che il punto restituito sia molto vicino al bordo della regione, situazione critica nelle operazioni in cui l'accuratezza numerica è importante, come confermato da alcuni test.

La mia versione, pur contemplando l'approccio della griglia, aggiunge un ulteriore vincolo di ottimizzazione: vogliamo scegliere, fra i punti interni, quello con *maggiore distanza dai bordi della regione e del dominio*. Pertanto ogni punto che risulta interno alla regione è sottoposto alla seguente valutazione:

```
for (i=0; i<CPN(c); i++)
{
.     dist = sqrt(SQR(CP(c,i).x - p2.x) + SQR(CP(c,i).y - p2.y));
.     dist_min = MIN(dist_min, dist);
}
```

dove `dist_min` registra la distanza minima del punto interno `p2` dai bordi, cioè da ogni punto che definisce la curva di trimming, che è contenuta nel vettore `c`. Successivamente si controlla se la distanza minima di quel punto è *maggiore* rispetto a quella degli altri punti interni finora trovati:

```
if (max_dist_min < dist_min)
{
.     max_dist_min = dist_min;
.     ix_best = ix; iy_best = iy;
}
```

il punto restituito dalla nuova `get_inner_point2` con coordinate `(ix_best, iy_best)` sarà il più distante dai bordi.

Se non si trovasse alcun punto interno con la prima griglia, questa viene raffinata e il test si ripete. Infine, superato un certo numero di iterazioni senza successo (ad esempio a causa di curve di intersezione non correttamente elaborate), la funzione esce con il messaggio di errore *"get_inner_point2 failed!"* per evitare congelamenti o crash del programma.

Nella figura 4.9 troviamo due esempi di punti interni calcolati con la nuova funzione: il primo proviene dal già noto test cono/cilindro, il secondo è in una regione di dominio a forma di F risultante da *effe.db* $\cap$ *sfera_effe.db*.

Nonostante questo aggiornamento abbia portato a miglioramenti, l'errore RDE in alcuni test non è ancora risolto. Le sue origini infatti non dipendono da questioni di tolleranze numeriche, come illustreremo nel prossimo capitolo.

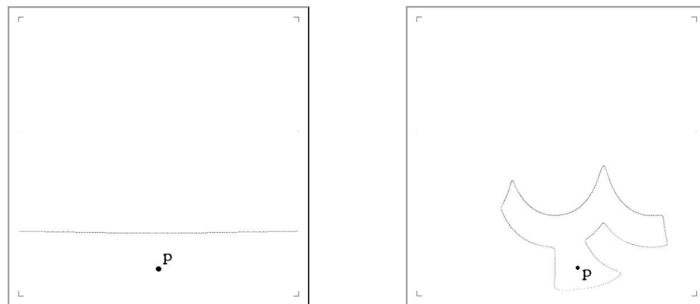

Figura 4.9: *Esempi di punti interni a regioni di dominio calcolati con la nuova get_inner_point2*

Capitolo 5

Conclusioni

Il mio percorso di analisi e aggiornamento del pacchetto XCBool ha portato a diversi miglioramenti. Ripetendo interamente la raccolta di test dopo le modifiche apportate al programma, si è verificato un decremento delle problematiche riscontrate da 40 a 13, ma aldilà del dato quantitativo è importante fare un sunto *qualitativo* delle nuove potenzialità:

- soluzioni *anti-crash*: XCBool non consente di proseguire con **Compute** se la risposta dell'intersecatore non è idonea per eseguire l'operazione booleana, inoltre non entra più in loop infinito nel calcolo dei punti interni alle regioni trimmate;
- l'*interazione di xcbool con xcssi*, anche in caso di intersezioni complesse, è stata ulteriormente integrata;
- l'*accuratezza numerica* ha raggiunto un compromesso ottimale.

Cerchiamo ora di approfondire i problemi rimasti.

5.1 Limiti del *marching cube*

In molti test che non sono stati eseguiti con successo si è riscontrato il tipo di errore **ISE**, pertanto è opportuno giustificare questa problematica.

Abbiamo parlato nel primo capitolo dell'algoritmo utilizzato da xcssi per la ricerca delle curve di intersezione, denominato *marching cube*. Prerogativa per il successo di tale strategia è la *regolarità* delle superfici da intersecare, quindi che siano *continue almeno fino alla derivata prima*.

Nel caso non lo siano, l'unica strategia che ha il programma per determinare le intersezioni è fermarsi nella ricerca nella direzione che fallisce, e tentare di recuperare quel ramo di curva giungendovi dalla direzione opposta. Quindi dai punti determinati inizialmente come *starting point* si avanza in entrambe le direzioni, fino al raggiungimento di un altro starting point.

Questo approccio può risolvere il problema in diversi casi, ma incontra chiaramente più difficoltà rispetto alla condizione base. E' chiaro quindi come gli errori ISE si trovino quasi esclusivamente nei test in cui è coinvolto il *cubo*, figura non regolare, perché lungo la sua superficie abbiamo curve continue, ma con derivata prima discontinua.

5.2 Superfici aperte

Gli errori di tipo **RDE** rimasti sono tutti legati al problema della composizione con *superfici aperte*.

Come sappiamo, con *solido* ci riferiamo ad una superficie *chiusa*. La mancanza di questa caratteristica consente comunque di procedere nel calcolo dell'operazione booleana da parte di xcbool, senza però aspettarsi una corretta risposta dalla funzione `trimmed_surface_point_contained`, perché effettivamente non è corretto chiedersi se "un punto p è interno o esterno ad una superficie aperta", ma solo se è interno o esterno ad un solido.

La suddetta funzione utilizza un algoritmo di *raytracing*, che sostanzialmente consiste nel proiettare una semiretta uscente dal punto p oggetto del test. Se questa semiretta incontra un numero di intersezioni *dispari* con la superficie del solido, significa che il punto è *interno*. Si tratta quindi della

regola di Jordan che abbiamo già visto nella valutazione di punti interni ad una curva.

Alla luce di ciò, l'utilizzo di superfici come *body.db* o *effe.db* (figura 5.1) nella composizione booleana non sono contemplate dalla nostra soluzione, salvo apportare cambiamenti nell'algoritmo utilizzato.

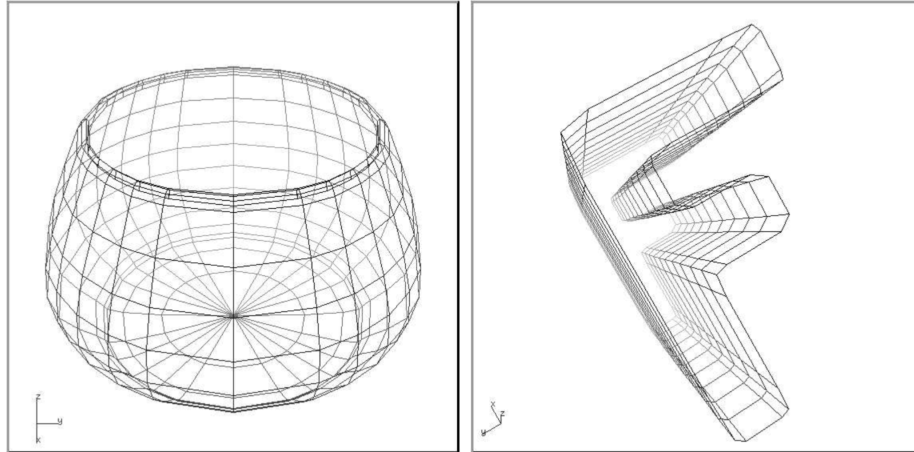

Figura 5.1: *Esempi di superfici aperte*

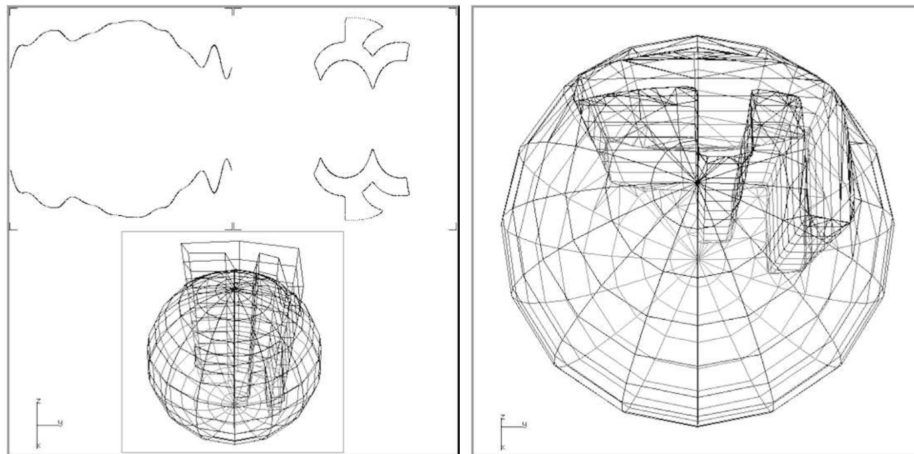

Figura 5.2: *Errore di valutazione nell'operazione effe.db* ∩ *sfera_effe.db*

5.3 Punti singolari

Un problema particolare che può portare all'errore **CDE** è causato dai *punti singolari*, che sono generati da linee di intersezione fra solidi *autointersecantesi*.

Se osserviamo l'esempio *toro 7* vediamo come la superficie del cilindro sia, in un punto, perfettamente adiacente a quella del toro. Questo fa sì che le due curve di intersezione abbiano quel punto in comune. Fornendo le opportune tolleranze, xcssi porta correttamente a termine l'operazione, però xcbool nell'operazione *toro2.db* ∩ *cilindro94_1.db* fa confusione nel determinare le regioni di dominio trimmato del cilindro, vedendo due regioni sovrapposte, come mostra la figura 5.3. L'operazione inversa è calcolata correttamente, fatto che però è deducibile esclusivamente osservando i domini nel CSG, perché con curve di trimming che si intersecano non è sistematicamente possibile fare una *resa* del solido composto utilizzando l'algoritmo adottato da XCBool.

In generale, le elaborazioni che coinvolgono punti singolari non fanno parte delle feature progettate in XCBool.

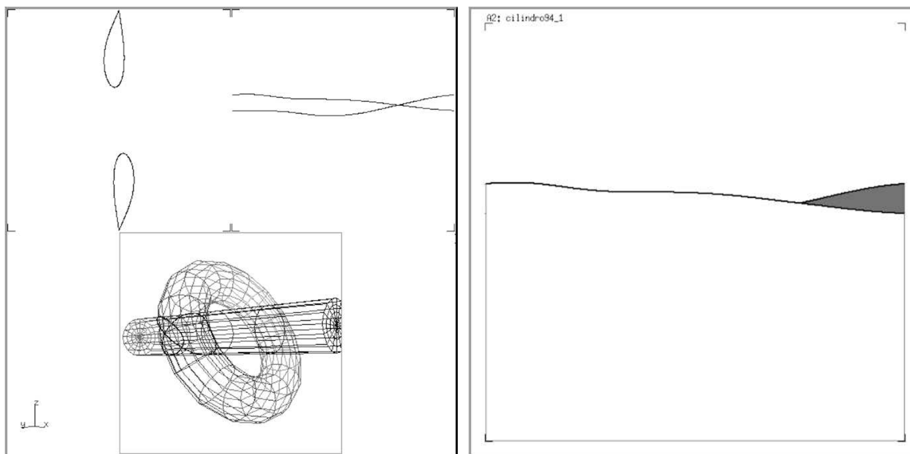

Figura 5.3: *Errore nella regione di dominio con un punto singolare*

5.4 Resa di superfici trimmate critiche

La libreria *trim* per la resa in tempo reale di superfici trimmate, ha dei limiti algoritmici che non hanno consentito la visualizzazione del risultato in alcuni test. Le situazioni problematiche per l'applicazione dell'algoritmo (che è stato illustrato nel capitolo 2), sono sostanzialmente di tre tipi (figura 5.4):

- 1. curve di trimming molto piccole;
- 2. curve di trimming con bordi molto vicini la cui tangente nella zona di vicinanza ha una certa inclinazione;
- 3. punti singolari.

Nei casi 1 e 2, per eseguire la tassellazione e quindi la resa, è necessario un elevato raffinamento della griglia del dominio, proporzionale alla piccolezza della curva nel caso 1 e alla vicinanza delle due curve nel caso 2. Per il caso 3 non c'è alcuna possibilità di applicazione del nostro algoritmo di resa, perché il punto singolare appartiene a due curve.

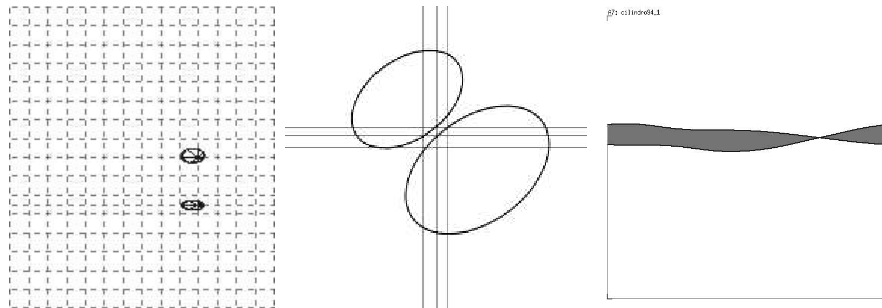

Figura 5.4: *Curve di trimming problematiche per la resa*

Seguono alcune proposte di miglioramento dell'algoritmo esistente, specifiche per i suddetti problemi.

- 1. Procedere alla triangolazione delle regioni molto piccole, anche se sono completamente all'interno di un rettangolo griglia, costruendo triangoli "a ventaglio" che hanno come vertice in comune un punto interno

della regione, mentre gli altri vertici sono costituiti dai punti della curva stessa. Unica prerogativa è che la forma della regione sia abbastanza semplice per essere approssimata in questo modo, ma essendo molto piccola è anche richiesta un'approssimazione inferiore rispetto ad altre parti della superficie trimmata.

- 2. Per non esasperare il raffinamento nelle zone di vicinanza delle curve, suddividere *internamente* i rettangoli griglia critici, in direzione delle tangenti delle curve nel punto di vicinanza. Si rende però necessario prevedere nuove casistiche di triangolazione in modo esaustivo.

- 3. Applicare una specifica suddivisione della griglia tracciando una *u-line* e una *v-line* passanti per il punto singolare. Questo non consente ugualmente di superare il vincolo sulle intersezioni curve/rettangoli imposto dall'attuale algoritmo, perché il punto singolare appartiene ad entrambe le curve. Però, trovandosi sul vertice comune dei quattro rettangoli griglia adiacenti appositamente costruiti con la nuova *u-line* e *v-line*, è possibile applicare un trattamento ad hoc per il punto singolare, escludendolo dal vincolo, per poi procedere alla triangolazione già implementata.

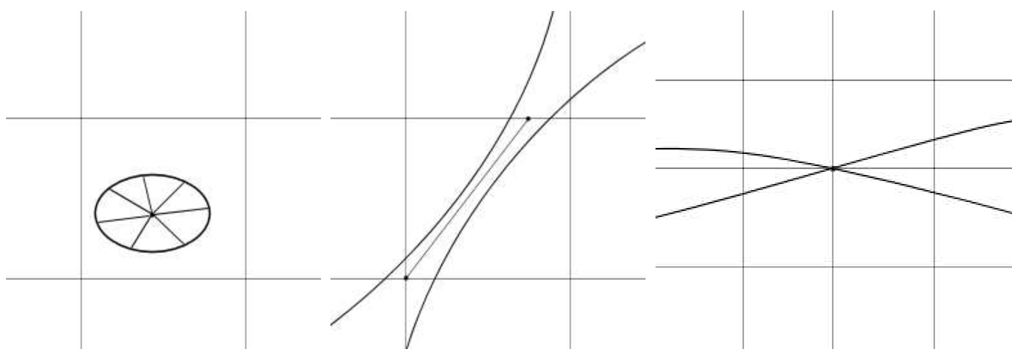

Figura 5.5: *Possibili soluzioni per la resa di curve di trimming particolari*

Una proposta alternativa consiste in un totale cambiamento dell'algoritmo corrente. Ne sintetizziamo l'idea nelle seguenti fasi:

- suddividere il dominio con una griglia uniforme;
- per ogni rettangolo griglia, consideriamo la regione di trimming in esso contenuta o delimitata, che chiameremo *sub-regione*;
- per ogni sub-regione scegliamo un punto interno, il più lontano dai bordi (come avviene nella nuova `get_inner_point2`);
- per ogni curva di trimming consideriamo un certo numero di punti in base ad una tolleranza CRT, che chiameremo *punti curva*;
- per ogni sub-regione deve sussistere la seguente *proprietà*: ogni segmento congiungente il punto interno con i punti curva della sub-regione non deve intersecare altri rami di curva. Qualora non sussista questa proprietà, *raffineremo adattivamente* la griglia;
- quando tutte le sub-regioni sono caratterizzate dal precedente vincolo è semplice procedere alla triangolazione, unendo il punto interno con i punti del bordo.

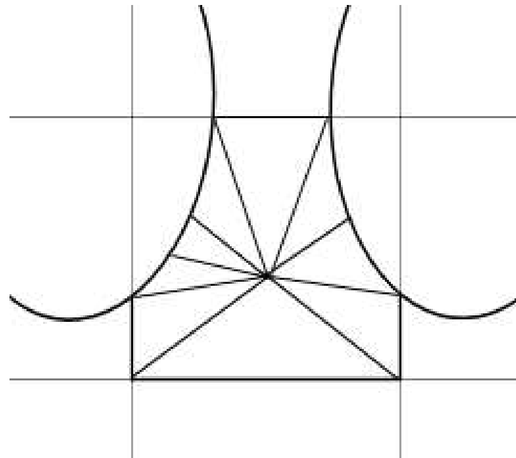

Figura 5.6: *Triangolazione di una sub-regione nel nuovo algoritmo proposto*

5.5 Ipotesi di lavoro future

Abbiamo visto come le operazioni di test dove il pacchetto XCBool non risponde ancora in modo corretto sono, in realtà, situazioni non contemplate dalle specifiche del programma stesso, almeno fino a questa versione.

Segue una lista di ipotetici miglioramenti che potrebbero essere apportati ad XCBool in versioni future.

- Aggiornamenti di **xcssi**:
  - in caso di fallimento dell'algoritmo con le tolleranze di default, provare *adattivamente* valori più accurati;
  - possibilità di stabilire un limite di tempo computazionale, o di interrompere in ogni momento, al fine di evitare situazioni di loop infinito che costringono alla terminazione forzata del programma.

- Aggiornamenti di **xcbool**:
  - rendere stabile il supporto per la composizione di *più di due solidi*, feature che è già stata prevista e almeno parzialmente implementata, ma non ancora terminata;
  - gestire le situazioni che generano *intersezioni irregolari*, ad esempio con *punti singolari*, ma è anche possibile che due solidi abbiano *subpatch* in comune: per far fronte a questi problemi servirebbe un aggiornamento importante nell'algoritmo di composizione [CABO89];
  - migliorare la funzione `trimmed_surface_point_contained` tramite un'*euristica* che consenta di comporre anche solidi *aperti*, fornendo una risposta che rispecchi maggiormente la percezione spaziale umana. Ad esempio con *effe.db*, nonostante sia un prisma senza basi, l'occhio valuta un punto interno o esterno come se fosse un prisma chiuso.

- Aggiornamenti della libreria **trim**:
  - migliorare l'algoritmo di resa di superfici trimmate valutando altre strategie, come una di quelle proposte nel paragrafo precedente.

5.5.1 Strumenti di sviluppo

Come ultima nota segnaliamo anche l'importante ruolo degli *strumenti di sviluppo* utilizzati nella pratica di debug e scrittura di codice.

Per un programma di ampie dimensioni come XCBool è di grande utilità lavorare con un **IDE** avanzato (*Integrated Development Environment*, cioè *Ambiente di Sviluppo Integrato*), come **KDevelop**, nel quale è stato importato per la prima volta l'intero progetto. Grazie a questa scelta infatti è stato possibile riprodurre l'esecuzione del programma *passo-passo*, monitorando il contenuto delle variabili tramite strumenti come i *breakpoint*, cioè interruzioni automatiche in punti del codice stabiliti, associati al *watch*: una tabella che contiene le variabili desiderate affiancate dal loro valore, aggiornato in tempo reale durante l'esecuzione.

Per analizzare rapidamente dati e situazioni più complesse, può essere utile anche la scrittura di funzioni specifiche per il debug, che riportino ad esempio un *log* degli eventi.

Nel nostro caso è stata implementata la `debug_print_curve`, che stampa su file il vettore di punti delle curve di trimming.

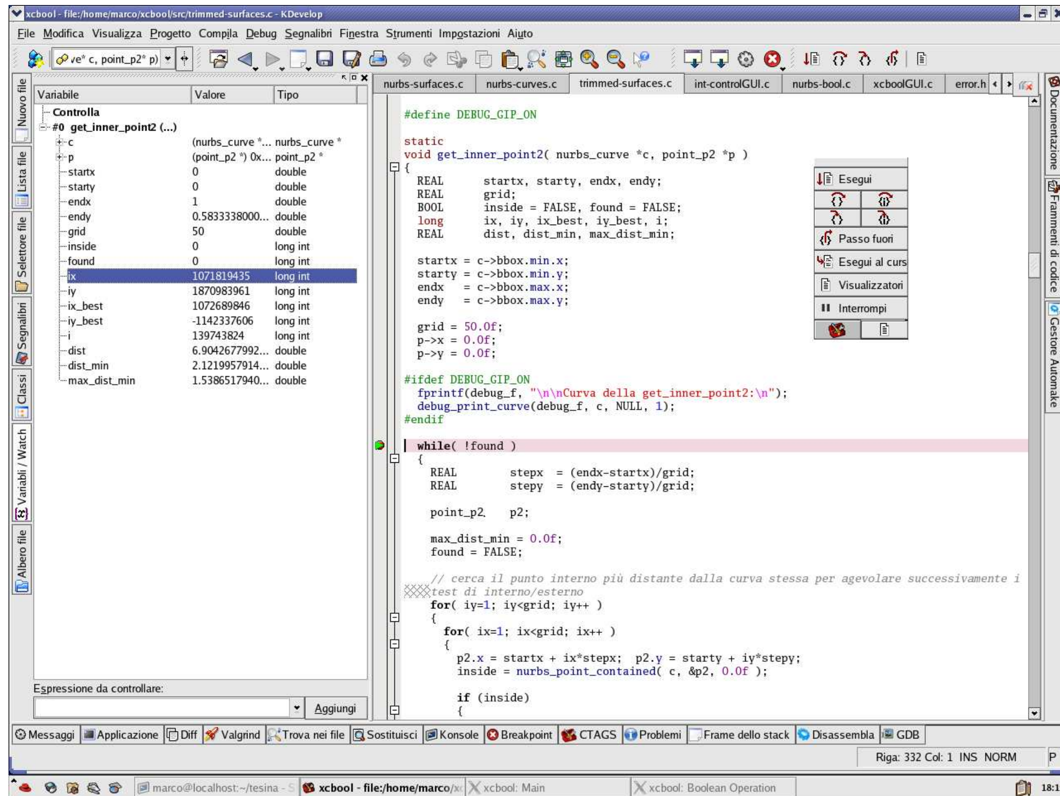

Figura 5.7: *Screenshot che illustra alcune funzionalità di KDevelop*

Bibliografia

[XCMODEL] G.Casciola. *xcmodel: a system to model and render NURBS curves and surfaces* User's guide (1999), `http://www.dm.unibo.it/~casciola/html/xcmodel.html`

[XCBOOL] G.Casciola, G.De Marco. *xcbool: the object composer* User's guide - Version 1.0 (2000), `http://www.dm.unibo.it/~casciola/html/xcmodel.html`

[XCSURF] G.Casciola. *xcsurf: the 3D modeller* User's guide (1999), `http://www.dm.unibo.it/~casciola/html/xcmodel.html`

[XTOOLS] G.Casciola, S.Bonetti. *xtools library: Programming Guide*, Version 2.0 (2001).

[FARI01] G.Farin. *Curves and surfaces for CAGD*, fifth edition, Academic Press (2001).

[CAMO00] G.Casciola, S.Morigi. *The trimmed NURBS age: advances in theory of computational mathematics: recent trends in numerical analysis* vol. III, Ed. D.Trigiante, Nova Science Publishers, Inc. (2000).

[PITI95] L.Piegl, W.Tiller. *The NURBS Book*, Springer Verlag (1995).

[CABO89] M.S.Casale, J.E.Bobrow. *A set operation algorithm for sculptured solids modeled with trimmed patches*, Elsevier Science Publishers B.V., Holland (1989).

[CASA87] M.S.Casale. *Free-Form Solid Modeling with Trimmed Surface Patches*, PDA Engineering, California (1987).

[BBB87] R.H.Bartels, J.C.Beatty, B.A.Barsky. *An introduction to splines for use in computer graphics and geometric modelling*, Morgan Kaufmann publishers, Palo Alto, California (1987).

[DEMA99] G.De Marco. *xcbool: un sistema per la composizione di solidi sculturati* (1999).

[SPAG98] S.Spagna. *Analisi, sviluppo e applicazione di metodi di intersezione per spline razionali* (1998).

[BORT96] M.Bortolani. *Un sistema per la modellazione e resa di scene definite mediante superfici NURBS trimmate* (1995).

[SUZZ95] M.Suzzi. *Un sistema prototipo per la composizione di solidi NURBS* (1995).