GVol:
software libero per la manipolazione e visualizzazione di dati volumetrici

Tesi di laurea di Alceste Scalas

Anno Accademico 2002–2003
Corso di Laurea in Informatica
Università degli Studi di Cagliari

Relatore interno: **Prof. Riccardo Scateni**
Relatore esterno: **Dott. Piero Pili, CRS4 (Cagliari)**

Sommario

Il *volume rendering* è una tecnica della grafica tridimensionale computerizzata in grado di rappresentare la conformazione interna degli oggetti. Nata per superare la classica rappresentazione poligonale delle sole superfici, questa tecnica si rivela utile in numerosi campi. Nel settore medico, per esempio, la visualizzazione tridimensionale dei dati volumetrici acquisiti da TAC si rivela un efficace strumento di diagnosi.

Nonostante questi aspetti positivi, i casi di effettiva applicazione della tecnica sono stati finora limitati, e per lo più ristretti al settore della ricerca. Questa situazione è dovuta a due fattori principali strettamente correlati: l'elevato costo computazionale, e la mancanza di studi sull'usabilità e applicabilità dal punto di vista degli utenti finali dei numerosi algoritmi di volume rendering finora sviluppati.

Lo scenario sta tuttavia cambiando: negli ultimi anni i costanti aumenti delle prestazioni dell'hardware, e il contemporaneo abbassamento dei costi, hanno avvicinato l'utilizzo del volume rendering alle architetture di calcolo di fascia media. La possibilità di utilizzare questa tecnologia per applicazioni orientate all'utente finale appare oggi più praticabile e interessante.

In questa tesi viene illustrato il lavoro di analisi e sviluppo di una libreria di componenti per il volume rendering chiamata *GVol*, rilasciata con licenza GNU GPL, e orientata alla prototipazione rapida di applicazioni "general pourpose" e all'ottimizzazione delle loro prestazioni.

Questo lavoro segue la ricerca svolta al CRS4 nell'ambito dello stage formativo, avente come oggetto l'analisi e sviluppo di un software di volume rendering per la sintesi di immagini realistiche chiamato *VolCastIA*.

Questo studio è stato svolto in collaborazione con il gruppo GEMS del CRS4, all'interno delle attività inerenti il Progetto Laboratorio Avanzato per la Progettazione e Simulazione sul Calcolatore.

Indice

1 Introduzione

Il **volume rendering** è una particolare tecnica di visualizzazione di dati tridimensionali in grado di rappresentare non solo la forma esterna degli oggetti, ma anche il loro *volume* e la loro conformazione interna.

L'approccio più classico, e sino ad oggi più diffuso, alla visualizzazione tridimensionale è quello del **rendering poligonale**. Esso si basa su primitive a 0, 1 o 2 dimensioni (punti, linee, poligoni), e riesce a rappresentare efficacemente ed efficientemente le superfici esterne degli oggetti trattati. Le prestazioni offerte dai sistemi per la grafica poligonale (siano essi hardware o software) sono notevoli, e la tecnica può contare su molti anni di esperienze e applicazioni nei settori più diversi; in tempi più recenti si è anche aggiunto l'interesse dell'industria videoludica, ormai trainante nel settore informatico, che spinge a notevoli investimenti in ricerca e sviluppo per l'aumento delle prestazioni dei moderni videogiochi tridimensionali. L'effetto più evidente di questa situazione è la presenza sul mercato di schede di accelerazione 3D estremamente potenti e a basso costo, rese possibili anche dalle economie di scala legate all'enorme richiesta e ai grandi volumi di mercato.

Il rendering poligonale non è tuttavia in grado di rappresentare efficacemente certe tipologie di fenomeni espressi sotto forma di dati tridimensionali. In particolare, si rivela limitato quando è necessario visualizzare non solo le superfici (frontiere) degli oggetti, ma anche il loro interno.

Il volume rendering nasce per superare tali limiti, offrendo anche una rappresentazione di concetti come "consistenza" e "variazione di densità" difficilmente comunicabili all'utente mediante un insieme di superfici. Questo tipo di informazione si rivela estremamente importante in numerosi campi: nel settore medico, per esempio, la visualizzazione tridimensionale dei dati volumetrici acquisiti da **Tomografia Assiale Computerizzata (TAC)** o da **Risonanza Magnetica (RM)** può essere un efficace strumento di supporto alla diagnosi. Il volume rendering infatti permette al medico di esplorare in tre dimensioni i dati del paziente, eventualmente variando la trasparenza di certe zone per poter far risaltare le regioni di interesse (**ROI, Regions of Interest**); oppure permette di verificare le densità dei tessuti, o di variarne i colori, in modo da individuare più facilmente organi e morfologie. Grazie a queste informazioni, il medico ha un maggior numero di elementi di diagnosi e di pianificazione della terapia.

Anche le tecniche di volume rendering presentano tuttavia dei problemi.

La quantità di calcoli necessari per la visualizzazione è uno dei punti critici: interagire con dati volumetrici significa gestire grandi griglie tridimensionali di valori, che possono occupare da centinaia di Kilobyte a centinaia di Megabyte di memoria. La mole di informazioni da trattare tende inoltre a crescere con i progressi dell'hardware di acquisizione dei dati: le TAC di prima generazione, per esempio, sono in grado di generare scansioni composte da circa 2,5 milioni di campioni (risoluzione nell'ordine di 256x256x40) — mentre una TAC moderna arriva generalmente a oltre 67 milioni di rilevazioni (risoluzione nell'ordine di 512x512x256). L'importanza dell'efficienza nella computazione dei dati è dimostrata dalla notevole mole di letteratura in tema, che propone numerose tecniche di ottimizzazione del processo di visualizzazione volumetrica.

Un altro problema che ostacola la diffusione del volume rendering è legato alle modalità di interazione con l'utente finale: fino ad oggi gran parte delle esperienze con la tecnica sono state limitate a settori di nicchia e di ricerca pura, ed il suo utilizzo in applicazioni orientate all'"uso quotidiano" (per esempio, in campo medico per il supporto alla diagnosi e terapia) è ad oggi ridotto. La letteratura presenta infatti numerose tecniche per l'analisi e la rappresentazione di dati volumetrici, ma risulta carente in studi approfonditi sull'usabilità di queste tecniche per applicazioni in settori specifici. Lo sviluppo di un applicativo per il volume rendering orientato ad un utente finale è quindi un argomento scarsamente esplorato.

In questo scenario si inserisce l'argomento della tesi: una libreria di componenti per il volume rendering chiamata **`GVol`**, rilasciata sotto licenza GNU GPL, e orientata alla prototipazione rapida del software applicativo, e all'ottimizzazione delle sue prestazioni.

Si può notare come le necessità nello sviluppo della libreria siano per molti versi contrastanti: in genere la flessibilità si paga con una perdita di prestazioni. Tale contrasto è tuttavia praticamente inevitabile quando si intende creare un applicativo per il volume rendering orientato all'utente finale: in genere, nello scenario attuale, l'utente stesso non sa cosa aspettarsi da uno strumento per lui così innovativo. La definizione degli obiettivi e delle caratteristiche dell'applicazione non può essere effettuata "a monte", ma può nascere solamente dall'interazione costante con l'utente stesso, durante la fase di progettazione e anche di sviluppo dell software. Poichè manca un dominio adeguato di esperienze pratiche per gli utenti e di riferimenti letterari che consentano di creare applicazioni di volume rendering ottimizzate per scopi e usi precisi, è indispensabile

la realizzazione e valutazione di prototipi funzionanti del software, che siano facilmente modificabili, ma siano anche in grado di offrire una stima di quelle che saranno le prestazioni dell'applicazione finale.

Lo svolgimento di questa tesi inizierà con una discussione sui limiti delle classiche tecniche di rendering poligonale (capitolo 2), comprendente un caso esemplificativo in cui la rappresentazione di dati tridimensionali mediante superfici risulta troppo limitata.

Nel capitolo 3 viene illustrato nel dettaglio il processo di volume rendering: viene presentato lo stato dell'arte della visualizzazione volumetrica, attraverso la discussione delle numerose problematiche da affrontare e dei modi in cui esse vengono risolte nella letteratura;

Nel capitolo 4 viene introdotto l'argomento della testi, `GVol`: vengono illustrati l'approccio alla sua progettazione, gli strumenti utilizzati per la realizzazione e le caratteristiche finora implementate.

Il capitolo 5 chiude la trattazione con le considerazioni finali e gli sviluppi futuri.

2 Dal rendering poligonale al rendering volumetrico

Il termine **informatica** nasce dalla contrazione di **informazione automatica**, e descrive in modo sintetico ed efficace l'argomento di questa scienza: il trattamento delle **informazioni**. Ogni tipo di utilizzo degli elaboratori elettronici è riconducibile all'immagazinamento e all'elaborazione dell'informazione; la fase elaborativa, in particolare, riguarda spesso la creazione di una **rappresentazione** dei dati che sia comprensibile dall'utente finale. Le modalità del processo di rappresentazione dipendono essenzialmente dal *tipo* e dalla *quantità* di dati a disposizione, e dalla tipologia di informazioni che si vogliono comunicare all'osservatore.

Il **rendering tridimensionale** non è altro che uno dei possibili metodi di rappresentazione delle informazioni, ed è utilizzato per un grande numero di applicazioni. Il motivo di questo successo si può comprendere analizzando la sua valenza informativa, ovvero la facilità con cui la rappresentazione tridimensionale di un oggetto o un fenomeno può essere recepita dall'utente: avendo a disposizione dei dati come posizione, forma e dimensione, il modo migliore per farli comprendere ad un essere umano è spesso una loro rappresentazione a schermo, più o meno realistica, dotata di una "tridimensionalità" simile a quella sperimentata nel mondo reale.

L'esempio più comune di rappresentazione tridimensionale è il **rendering poligonale**. I dati di partenza consistono in un insieme di punti, linee e poligoni nello spazio, ricavati a partire da oggetti o fenomeni reali o simulati; tali dati sono resi comprensibili mediante la proiezione su un piano (solitamente lo schermo), con opportuni accorgimenti per fornire la sensazione di tridimensionalità (per esempio mediante l'uso della prospettiva). L'informazione offerta da questa tecnica non riguarda l'*intero* oggetto di partenza, ma è limitata alla sua *superficie esterna* (frontiera) — poiché questo è l'unico tipo di struttura ottenibile a partire dalle primitive utilizzate. Questo metodo di rappresentazione è comunque generalmente sufficiente per identificare gli elementi con cui l'utente interagisce, e risulta adeguato in un gran numero di applicazioni: dal gioco alla simulazione di fenomeni fisici. Le prestazioni del sistema sono inoltre notevoli, data la mole generalmente ridotta di dati da trattare, e data la disponibilità sul mercato di hardware dedicato all'accelerazione del processo di rendering poligonale (si pensi alle schede di accelerazione 3D).

Il rendering poligonale si rivela tuttavia una tecnica inadeguata nel momento in cui le informazioni di partenza non possono essere ricondotte a poligoni nello spazio, e nel

momento in cui la sola rappresentazione delle superfici esterne degli oggetti non offre informazioni sufficienti per un determinato campo di applicazione. Si può fare l'esempio delle acquisizioni da TAC o RM: in questi casi i dati da rappresentare consistono in una pila di scansioni bidimensionali del corpo del paziente. Sovrapponendo le scansioni si ottiene una griglia tridimensionale composta dai valori rilevati dalla macchina nelle diverse posizioni dello spazio. Si è così creato un **dataset volumetrico**: in che modo l'informazione contenuta potrebbe essere rappresentata, per poter essere comprensibile dall'utente?

Uno dei metodi, ampiamente utilizzato dal personale medico, consiste nel visualizzare una singola "fetta" bidimensionale di dataset alla volta. È una tecnica che non richiede necessariamente l'interazione con un computer: sono sufficienti dei fogli semitrasparenti e di una lavagna luminosa. Purtroppo questo sistema fa perdere l'informazione relativa ad una delle tre dimensioni, e si rivela dunque non ottimale (anche se l'esperienza permette ai medici di ricostruire mentalmente la dimensione mancante).

Un'altra possibile soluzione per la visualizzazione di un dataset volumetrico e' l'estrazione di informazioni trattabili dai classici algoritmi di rendering poligonale, ovvero la ricerca di superfici di interesse all'interno della griglia di dati. Questo approccio è chiamato **volume rendering indiretto**, ed è utilizzato, per esempio, degli algoritmi denominati **marching cubes** [3] e **dividing cubes** [28]. Essi si basano sull'analisi del datast tridimensionale e sulla ricerca di soglie o di cambiamenti di valore che potrebbero indicare la presenza di superfici da ricostruire e visualizzare. Per esempio: una zona di transizione tra campioni a bassa intensità e campioni ad alta intensità in una acquisizione da TAC potrebbe indicare la presenza della superficie esterna di un osso; essa potrebbe essere quindi ricostruita collegando con linee e poligoni le zone caratterizzate da transizioni analoghe.

Il risultato di queste tecniche è una superficie poligonale (**mesh**) che approssima la superficie esterna delle morfologie rilevate nel dataset volumetrico, talvolta con buoni risultati (fig. 2).

Questo approccio alla visualizzazione di volumi presenta tuttavia dei limiti. Alcuni sono legati al procedimento di ricerca delle informazioni: l'individuazione delle superfici di interesse è basilarmente un processo di selezione binaria (*in questo punto passa una superficie / in questo punto non passa una superficie*), che è inevitabilmente esposto al rilevamento di falsi positivi (superfici inesistenti) o falsi negativi (buchi all'interno delle superfici rilevate). L'approccio risulta inoltre inadeguato per la visualizzazione di

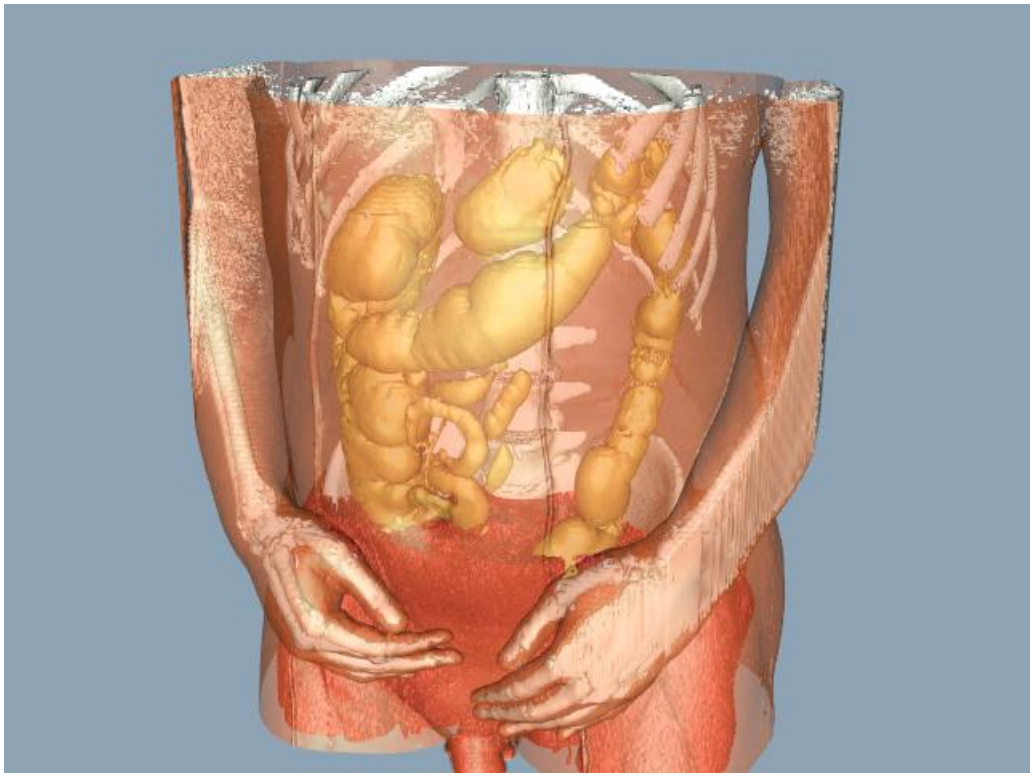


Figura 1: *Rendering di una mesh poligonale ricostruita mediante l'algorimo marching cubes. Immagine tratta da* `http://www.crd.ge.com/esl/cgsp/projects/vm/`.

certe conformazioni dei dati: per esempio, le variazioni di densità all'interno dei tessuti non sono sempre semplificabili in forma di superfici ben definite, e certe strutture contenute nel dataset vengono necessariamente alterate o perse durante creazione della mesh poligonale.

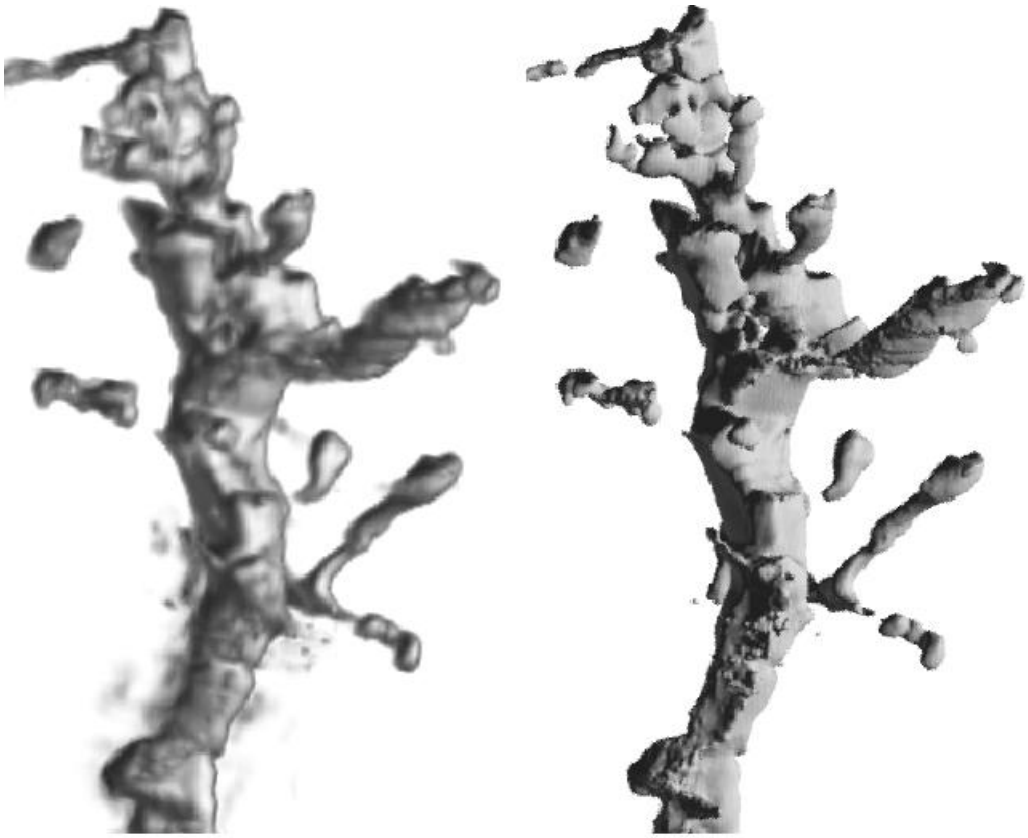
Altri problemi sono riscontrabili nel tipo di rappresentazione dei dati offerta all'utente: non sempre l'approssimazione di un organo come un insieme di superfici poligonali cave è accettabile. Un medico può aver bisogno di esaminare la consistenza e la densità delle zone interne dei tessuti — ma questi tipi di informazione sono difficilmente comunicabili dalla visualizzazione di poligoni bidimensionali. Esiste inoltre la possibilità che una variazione della densità dei tessuti non rilevata (o mal visualizzata) dall'algoritmo possa trarre in inganno l'osservatore (fig. 2)..

Appare quindi chiara la necessità di utilizzare un altro sistema di rappresentazione

dei dati tridimensionali, che sia in grado di comunicare all'utente la presenza e la consistenza del *volume* degli oggetti, senza limitarsi alla semplice rappresentazione delle superfici.

Questo risultato è ottenibile solamente cambiando approccio alla visualizzazione dei dati volumetrici: invece di ricavare una informazione intermedia (il dato poligonale) dal dataset è necessario portare l'informazione contenuta nei dataset volumetrici "direttamente" sullo schermo, evitando ricostruzioni e alterazioni intermedie. Per questo è necessario cambiare le primitive della rappresentazione: non più punti, rette e poligoni, ma direttamente i valori del dataset, che prendono il nome di **voxel** (un termine derivante dalla fusione di *volume* e **pixel**).

Questo tipo di rendering tridimensionale è chiamato **volume rendering diretto**, e prevede la rappresentazione diretta della griglia di voxel su un piano bidimensionale. Esistono numerosi algoritmi che affrontano questo problema, differenziandosi per prestazioni, resa grafica e quantità di informazioni fornite all'utente: nel prossimo capitolo verranno illustrate e discusse le tecniche principali illustrate dalla letteratura.



(a) Direct volume rendered (b) Isosurface rendered

Figura 2: *Rendering volumetrico (a sinistra) e poligonale (a destra) di un neurone. La parte in basso dell'immagine a sinistra mostra una sorta di "nebbia", dovuta alla mancanza di definizione del dataset di partenza. L'immagine a destra, invece, non visualizza questa informazione: la superficie della zona è simile a quelle visualizzate per il resto dell'oggetto, e l'utente non può stabilire se si tratti di un dato "reale" o di un semplice artefatto. Immagine tratta da* `http://www.cs.utah.edu/~gk/MS/html/node3.html`.

3 Direct volume rendering

In questo capitolo verrà illustrato il processo di direct volume rendering (d'ora in avanti, semplicemente volume rendering), con una panoramica sui problemi da affrontare e su alcuni algoritmi. Una introduzione più ampia all'argomento è disponibile in [24], [14], [13].

3.1 Alcune definizioni

Prima di iniziare, è utile definire alcuni termini che verranno spesso utilizzati da qui in avanti.

Innanzi tutto, un **dataset volumetrico** può essere definito come un insieme S di voxel (x, y, z, p) distribuiti nello spazio, dove:

- $x, y, z \in \mathbb{N}^0$ indicano la posizione del voxel nelle 3 dimensioni (si assumerà d'ora in avanti che i voxel siano disposti su una griglia regolare, con coordinate intere; la letteratura si occupa anche di disposizioni non strutturate);
- $x < \dim_x$, dove $\dim_x$ è la dimensione del dataset lungo l'asse X;
- $y < \dim_x$, dove $\dim_y$ è la dimensione del dataset lungo l'asse Y;
- $z < \dim_z$, dove $\dim_z$ è la dimensione del dataset lungo l'asse Z.
- p indica una qualsiasi proprietà del dataset.

p indica solitamente una proprietà di tipo numerico; quando essa è limitata ai valori 0 e 1, si parla di **dataset binario** [14].

Un voxel (x, y, z, p) verrà indicato, per brevità, anche con il simbolo $\overline{\boldsymbol{v}}$.

Con $f(x, y, z, p)$, $f(\overline{\boldsymbol{v}})$, o semplicemente f, si indicherà una generica funzione, definita in S, a valori in un codominio variabile in base al contesto (che verrà definito di volta in volta). L'individuazione di questo tipo di funzioni dipendenti unicamente dal valore del voxel è estremamente importante per l'ottimizzazione del processo di volume rendering: esse possono essere sostituite, per esempio, dalla ricerca in un elenco di valori (**lookup table**) generato con il precalcolo della funzione stessa per ogni valore di $\overline{\boldsymbol{v}}$ — e per questo motivo si è scelto di evidenziarle con un simbologia uniforme. Un esempio di utilizzo di tali funzioni si trova nella sez. 3.3.

L'**opacità** di un voxel sarà indicata dal simbolo $\alpha(\overline{\boldsymbol{v}}) \in [0,1]$. Essa rappresenta la frazione di raggi luminosi trattenuti: un voxel con opacità $0,5$, per esempio, risulterà non attraversabile dalla metà dei raggi, e quindi semitrasparente agli occhi dell'osservatore.

Il **colore** di un voxel sarà indicato dal simbolo $c(\overline{\boldsymbol{v}}) \in C_X$, dove C_X indica un qualsiasi spazio di colori (come C_{RGBA} o $C_{\text{CIE-Luv}}$ (per maggiori dettagli, si veda la sez. 3.9.

Il **gradiente** è un vettore che indica la direzione e l'intensità della variazione dei valori dei voxel attorno a un punto di applicazione. Solitamente gli algoritmi di volume rendering prevedono il calcolo del gradiente normalizzato in corrispondenza di ogni voxel. Nel seguito verrà usata la seguente simbologia:

- $\nabla f(\overline{\boldsymbol{v}})$ è il gradiente normalizzato calcolato in corrispondenza del voxel $\overline{\boldsymbol{v}}$ utilizzando la funzione f (dipendente quindi dalla proprietà p). Occorre notare che questa simbologia potrebbe risultare ambigua, poichè potrebbe suggerire che ∇f sia funzione di $\overline{\boldsymbol{v}}$ (cosa generalmente non vera). Questa rappresentazione del gradiente (o sue varianti) è tuttavia ampiamente utilizzata nella letteratura (per esempio in [24]), e si è preferito non discorstarsi eccessivamente da essa;

- $\nabla f(\overline{\boldsymbol{v}})_x$, $\nabla f(\overline{\boldsymbol{v}})_y$ e $\nabla f(\overline{\boldsymbol{v}})_z$ sono le componenti del gradiente calcolato in corrispondenza del voxel $\overline{\boldsymbol{v}}$ in base alla funzione f;

- $\|\nabla f(\overline{\boldsymbol{v}})\|$ è la norma euclidea del gradiente calcolato alla posizione (x,y,z) in base alla proprietà p dei del dataset, normalizzata in modo tale che $0 \leq \|\nabla f(\overline{\boldsymbol{v}})\| \leq 1 \; \forall \; (\overline{\boldsymbol{v}}) \in S$. Quindi, dato $\overline{\nabla f}$ estremo superiore dell'insieme delle norme dei gradienti calcolati all'interno di un dataset, si ha che:

$$\|\nabla f(\overline{\boldsymbol{v}})\| = \frac{\sqrt{\nabla f(\overline{\boldsymbol{v}})_x^2 + \nabla f(\overline{\boldsymbol{v}})_y^2 + \nabla f(\overline{\boldsymbol{v}})_z^2}}{\overline{\nabla f}} \tag{1}$$

3.2 Dal voxel al pixel (rendering)

Uno degli approcci più semplici alla visualizzazione di volumi consiste nel considerare i voxel come punti tridimensionali da proiettare direttamente sullo schermo, mediante una trasformazione geometrica basata su una **matrice di visualizzazione**. Gli algoritmi di questo tipo possono essere suddivisi in due classi:

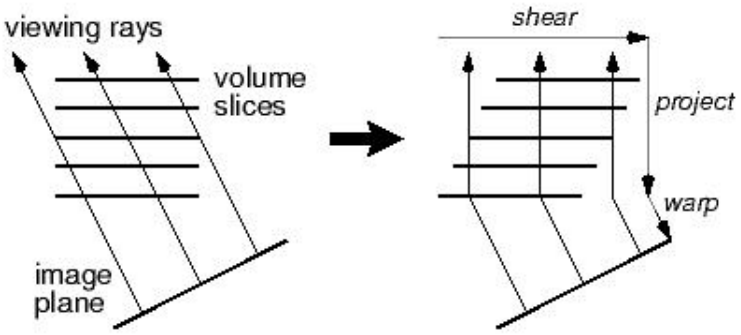


Figura 3: *Trasformazione delle fette di dataset nell'algoritmo di shear-warping, con raggi visuali paralleli. Immagine tratta da [19].*

back-to-front (BTF) i voxel vengono attraversati partendo dal punto più lontano della scena, procedendo verso la posizione dell'osservatore. Durante la rappresentazione dei pixel a video, i voxel più vicini sovrascrivono quindi quelli più lontani, in modo analogo a quanto avviene nel classico "algoritmo del pittore". Le prestazioni di questa tecnica dipendono dalla semplicità del processo di proiezione, che in certi casi può essere applicato rapidamente a una ragionevole quantità di voxel.

front-to-back (FTB) i voxel vengo attraversati a partire dall'osservatore, procedendo verso il punto più lontano della scena. Questo approccio permette di stabilire facilmente quali voxel siano effettivamente influenti nella visualizzazione finale, e consente di alleggerire l'elaborazione ignorando, per esempio, i voxel coperti da altri voxel completamente opachi.

Altri algoritmi seguono un approccio più complesso, basato sulla trasformazione del dataset volumetrico in un sistema di riferimento tridimensionale intermedio detto **pixel space**. Attraverso una trasformazione affine le fette di valori costituenti il dataset vengono traslate e scalate in modo tale da poter essere proiettate a video utilizzando rette perpendicolari al piano visuale, ottenendo comunque un risultato visivo analogo ad una proiezione assonometrica o prospettica. In questo modo la visualizzazione può essere ottenuta sovrapponendo una fetta di voxel dopo l'altra in ordine front-to-back. Un esempio di questo approccio è fornito dall'algoritmo di **shear-warping** sviluppato da Lacroute e Levoy [19] (fig. 3.2, 3.2, 3.2).

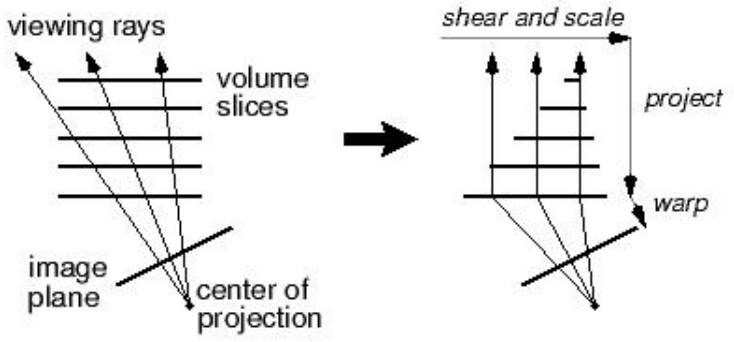


Figura 4: *Trasformazione delle fette di dataset nell'algoritmo di shear-warping, con raggi visuali non paralleli (proiezione prospettica). Immagine tratta da [19].*

Un altro approccio al volume rendering è quello del **volumetric ray casting**, che si basa sull'utilizzo di raggi visuali che attraversano direttamente il dataset. La griglia di campioni viene trattata come una sorta di gel semitrasparente, nella quale ogni raggio può penetrare e raccogliere informazioni (come il colore) dai voxel adiacenti. L'opacità assegnata ai voxel (si veda sez. 3.4) influisce sulla profondità di penetrazione di ogni raggio.

Anche gli algoritmi di **volumetric ray tracing** lavorano utilizzando raggi visuali, ma prevedono la simulazione dei fenomeni fisici legati al processo di visualizzazione (come rifrazione e riflessione). Questo tipo di rendering è solitamente utilizzato per la rappresentazione fotorealistica dei dati volumetrici.

3.3 Informazioni associate ai voxel

Fino a questo momento la definizione del tipo di dati memorizzabile all'interno di un voxel (ovvero, il valore di p definito nella sezione 3.1) è stata mantenuta volutamente generica. Questa informazione è infatti estremamente variabile, e può dipendere dai dati a disposizione, dall'algoritmo di rendering utilizzato o dal tipo di visualizzazione richiesta dall'utente.

Nel caso più semplice, p indica un colore e un'opacità, che possono essere utilizzati direttamente per ricavare il colore dei pixel su schermo.

In altri casi p è un semplice valore numerico (per esempio nei dati provenienti da

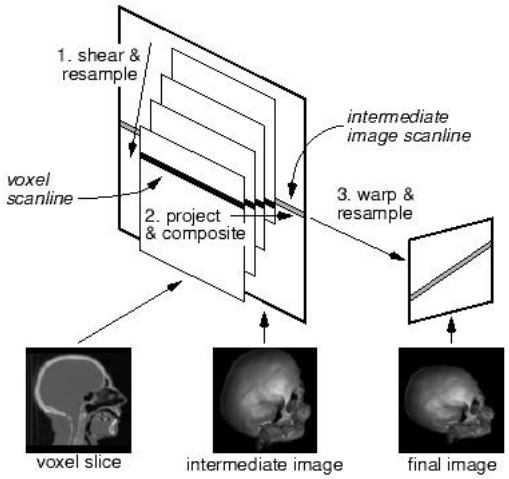


Figura 5: *I tre passaggi dall'algoritmo di shear-warping: ridisposizione delle fette, scansione dei voxel, proiezione finale per la correzione della deformazione. Immagine tratta da [19]*

TAC o RM), che in fase di rendering deve essere associato ad altre informazioni (quali appunto un colore e un'opacità) prima di poter essere proiettato a video. L'associazione di nuove informazioni a un voxel $(\overline{\boldsymbol{v}})$ è ottenuta attraverso funzioni $f(\overline{\boldsymbol{v}})$ di vario tipo: l'applicazione può andare dall'insieme dei voxel a un qualsiasi spazio di colori (per esempio, $f : S \to C_{\text{CIE-XYZ}}$ — si veda la sez. 3.9), o dall'insieme dei voxel all'insieme dei reali positivi compresi nell'intervallo $[0, 1]$ ($f : S \to [0, 1]$, situazione generalmente riscontrabile nell'assegnamento dell'opacità $\alpha_{\overline{\boldsymbol{v}}}$).

p può indicare anche valori non numerici — per esempio un'etichetta (**label**) destinata a segnalare certe caratteristiche del voxel (come l'appartenenza a un certo materiale). Il processo di assegnamento delle label è detto **segmentazione**. Nei casi più semplici il valore di un'etichetta indicata dalla proprietà p'' in un voxel (x, y, z, p'') può essere funzione del valore di un'altra proprietà p'; in altri casi il dataset può essere sottoposto ad una fase di pre-processamento più articolata, basata su diversi parametri: la segmentazione di dati medicali, per esempio, potrebbe osservare delle regole tese ad assicurare che i voxel marcati come "osso" seguano una morfologia compatibile con la struttura di un osso reale. Nei casi più complessi il processo di assegnamento dell'etichetta ai voxel viene effettuato manualmente, o comunque con l'interazione di un

utente.

3.4 Calcolo (mapping) dell'opacità dei voxel

La comprensione delle informazioni contenute in un dataset volumetrico è generalmente un problema di *visibilità*: nella scelta dei parametri di rendering si deve fare in modo che l'utente possa distinguere i voxel contenenti informazioni di interesse da tutti gli altri voxel contenuti nel dataset. Occorre dunque evitare che zone prive di informazione utile coprano altre zone più importanti, o che dettagli importanti vengano esclusi dall'immagine finale. Nella gran parte dei casi la soluzione a questo problema non è semplice.

La visibilità di un voxel è legata alla sua **opacità**: ai voxel più importanti viene generalmente assegnata una opacità maggiore, mentre quelli trascurabili sono di solito resi semitrasparenti (o invisibili) nell'immagine finale. L'operazione di assegnamento di una opacità ai voxel è detta **classificazione**.

Lo scenario più semplice da affrontare è quello in cui occorre visualizzare un dataset composto da livelli di colore e opacità (per esempio, quando il termine p indica valori di tipo RGBA — Red–Green–Blue–Alpha): in tal caso una apposita fase di classificazione non è necessaria, e l'informazione contenuta in ogni voxel intersecato da un raggio visuale può essere accumulata e trasferita direttamente a video.

Se p indica una label, l'assegnamento delle opacità può risultare ugualmente semplice: in tali casi si può stabilire che l'opacità debba essere funzione del valore del voxel ($\alpha_{\overline{\boldsymbol{v}}} \sim f(\overline{\boldsymbol{v}})$), e ciò permette di assegnare e modificare la trasparenza di intere zone di dataset contraddistinte dalla medesima etichetta. La significatività delle immagini così ottenute dipende in primo luogo dalla correttezza della fase di segmentazione.

Nei casi in cui p rappresenti un valore numerico (per esempio, nelle scansioni acquisite da TAC), il criterio di assegnamento dell'opacità può variare notevolmente. L'approccio più semplce è ancora quello di ottenere l'opacità in funzione del solo valore del voxel ($\alpha_{\overline{\boldsymbol{v}}} \sim f(\overline{\boldsymbol{v}})$), in modo da facilitare il rilevamento di zone omogenee all'interno del dataset (**isovalori**).

Un esempio di questo tipo di classificazione dato dalla seguente funzione, derivata dal sistema di assegnazione dei materiali di Drebin, Carpenter e Hanrahan illustrato nella sez. 3.5. Dati i valori di interesse val_n e val_{n+1}, e dato il parametro $0 \leq t_v \leq 1$ che indica la rapidità della transizione da un valore di interesse all'altro, e innanzi tutto

possibile individuare n regioni di transizione di ampiezza $\text{tr}_n = \text{val}_n + \text{tr}_n(\text{val}_{n+1} - \text{val}_n)$. Quindi, date le opacità associate ai valori di interesse α_n e α_{n+1}, l'opacità $\alpha(\overline{\boldsymbol{v}})$:

$$\alpha(\overline{\boldsymbol{v}}) = \begin{cases} \alpha_n & \text{se } \text{val}_n \leq f(\overline{\boldsymbol{v}}) < \text{tr}_n \\ \alpha_{n+1}\left(\frac{f(\overline{\boldsymbol{v}}) - \text{tr}_n}{\text{val}_{n+1} - \text{val}_n}\right) + & \\ \quad \alpha_n\left(\frac{\text{val}_{n+1} - f(\overline{\boldsymbol{v}})}{\text{val}_{n+1} - \text{val}_n}\right) & \text{se } \text{tr}_n \leq f(\overline{\boldsymbol{v}}) < \text{val}_{n+1} \\ 0 & \text{altrimenti} \end{cases} \tag{2}$$

Purtroppo questo semplice sistema di classificazione risente molto della conformazione e del livello di rumore del dataset: esso si rivela generalmente efficace solo per dataset composti da isovalori ben definiti. I dataset reali (per esempio, quelli medicali) spesso non rispettano queste condizioni, e questo spesso rende difficile la ricerca di valori ottimali per i parametri tr, val_{v_n} e α_{v_n}.

Il sistema ha in ogni caso il vantaggio di preservare la percezione dei volumi nell'immagine finale, come mostrato nella figura 3.4.

Un approccio più avanzato al processo di classificazione consiste nel far dipendere l'opacità di ogni voxel $(\overline{\boldsymbol{v}})$ da $\|\nabla f(\overline{\boldsymbol{v}})\|$, norma del gradiente calcolato nella medesima posizione: in questo modo è infatti possibile evidenziare le zone di dataset in cui sono presenti maggiori cambiamenti (indice della presenza di maggiore informazione). Questa idea è applicata in due noti algoritmi di classificazione proposti da Marc Levoy in [18], che verranno illustrati nelle prossime due sezioni.

3.4.1 Classificazione orientata alle superfici esterne degli isovalori

Questo algoritmo di classificazione permette di far risaltare le superfici esterne degli isovalori contenuti in un dataset, e di rendere al contempo visibili le zone caratterizzate da rapidi cambiamenti nei valori dei voxel. Il procedimento è stato sviluppato per migliorare la visualizzazione delle mappe di densità elettronica rilevate da molecole complesse.

La prima fase per l'utilizzo dell'algoritmo consiste nell'individuazione dei valori (o degli intervalli di valori) associabili a isovalori di interesse all'interno del dataset, e nell'assoociazione dei livelli di opacità desiderati.

L'associazione diretta valore–opacità è tuttavia una operazione di selezione binaria

Figura 6: *Rendering volumetrico eseguito utilizzando la classificazione lineare delle opacità. Si può notare come l'immagine permetta di percepire il volume la consistenza dei tessuti. Immagine generata con il programma* VolCastIA*, sviluppato internamente al CRS4.*

(*questo voxel contiene questo valore / questo voxel non contiene questo valore*), e come tale soggetta alla creazione di artefatti o buchi nell'immagine finale. È quindi necessario assicurarsi che la transizione da un isovalore all'altro non sia brusca, ma progressiva all'interno di un certo intervallo r (denominato **thickness** negli articoli di Levoy).

L'algoritmo prevede inoltre l'esaltazione delle zone di dataset caratterizzate da variazioni nelle intensità: per questo è necessario che nella formula compaia la norma di gradiente $\|\nabla f(\overline{\boldsymbol{v}})\|$.

Quindi, dato il valore val_n, l'opacità associata α_n e lo spessore della regione di transizione r, l'opacità per il voxel $\overline{\boldsymbol{v}}$ sarà α_n quando $f(\overline{\boldsymbol{v}}) = \text{val}_n$ e $\nabla f(\overline{\boldsymbol{v}}) = 0$. Se il valore di $f(\overline{\boldsymbol{v}})$ è invece diverso da val_n, l'opacità tenderà a diminuire ad una velocità proporzionale a $(\text{val}_n - f(\overline{\boldsymbol{v}}))$, e inversamente proporzionale a $\|\nabla f(\overline{\boldsymbol{v}})\|$ e a r.

Si utilizza di conseguenza la seguente formula:

$$\alpha(\overline{\boldsymbol{v}}) = \alpha_n \begin{cases} 1 & \text{se } \|\nabla f(\overline{\boldsymbol{v}})\| = 0 \wedge f(\overline{\boldsymbol{v}}) = \text{val}_n \\ 1 - \frac{1}{r}\left|\frac{\text{val}_n - f(\overline{\boldsymbol{v}})}{\|\nabla f(\overline{\boldsymbol{v}})\|}\right| & \text{se } \|\nabla f(\overline{\boldsymbol{v}})\| > 0 \wedge \\ & f(\overline{\boldsymbol{v}}) - r\|\nabla f(\overline{\boldsymbol{v}})\| \leq \text{val}_n \leq \\ & \leq f(\overline{\boldsymbol{v}}) + r\|\nabla f(\overline{\boldsymbol{v}})\| \\ 0 & \text{altrimenti} \end{cases} \tag{3}$$

Quando è necessario visualizzare più di un isovalore, l'espressione precedente deve essere applicata separatamente ad ognuno di essi, ed i risultati devono essere combinati. Dati i valori di interesse $\text{val}_n, 1 \leq n \leq N, n \in \mathbb{N}^0$, con le opacità associate α_n, l'opacità complessiva per il voxel $(\overline{\boldsymbol{v}})$ è data da:

$$\alpha_{\text{tot}}(\overline{\boldsymbol{v}}) = 1 - \prod_{n=1}^{N}(1 - \alpha_n(\overline{\boldsymbol{v}})) \tag{4}$$

Dove $\alpha_n(\overline{\boldsymbol{v}})$ è l'opacità calcolata per il valore di interesse n utilizzando la formula precedente.

Un esempio di applicazione di questo algoritmo può essere osservato nella figura 3.4.1.

3.4.2 Classificazione orientata alle superfici di confine tra isovalori

Questo algoritmo segue un approccio in parte opposto rispetto a quello appena visto: viene infatti ridotta la visibilità delle zone di dataset composte da valori omogenei (anche se esse sono composte da valori di interesse), e viene incrementata la visibilità delle zone di transizione caratterizzate da alti valori di gradiente.

Questo sistema di classificazione si rivela particolarmente utile in campo medicale, poichè permette di far risaltare le *superfici* di contatto tra i diversi organi: esse sono infatti caratterizzate da un cambiamento sensibile nei valori rilevati da TAC o RM (si pensi alla variazione di densità nelle regioni di transizione muscolo–osso). Le superfici così ricostruite agli occhi dell'utente non vengono individuate con una procedura di selezione binaria (come accade invece nell'algoritmo marching cubes, si veda la sez.

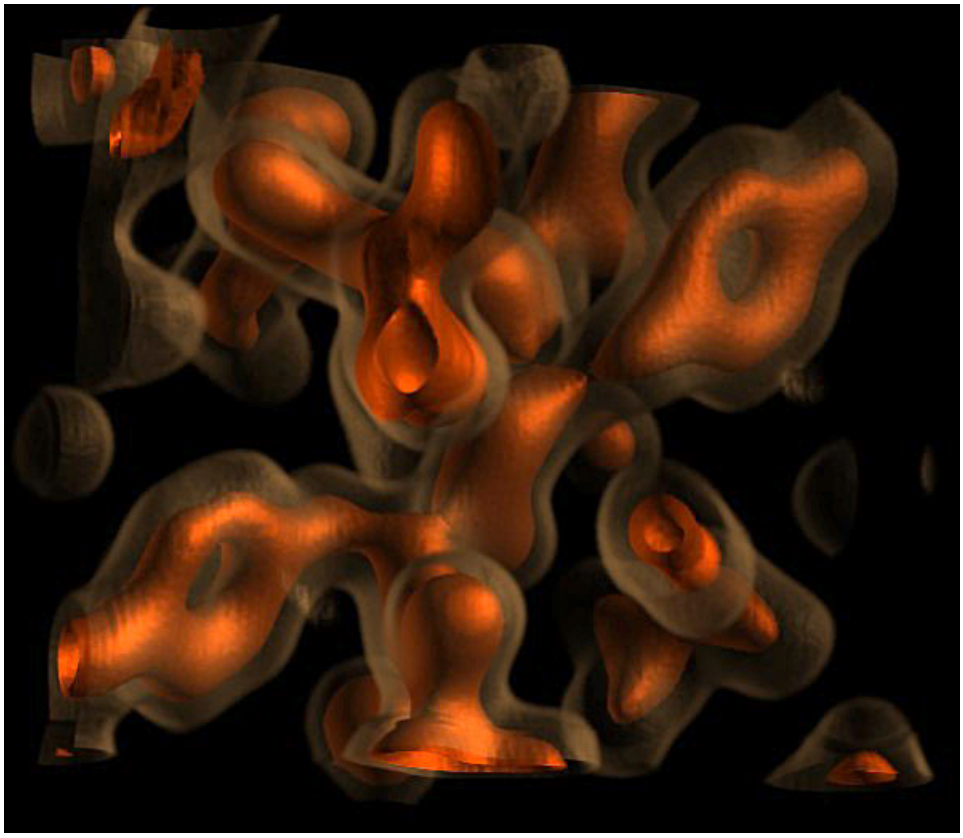


Figura 7: *Rendering volumetrico eseguito utilizzando la classificazione delle opacità orientata alle superfici esterne degli isovalori. Immagine tratta da* `http://graphics.stanford.edu/projects/volume/volrend.html`.

2), e l'immagine finale non presenta artefatti. Inoltre l'informazione sul volume degli organi non viene persa, e rimane a disposizione per l'eventuale visualizzazione.

Per togliere risalto alle zone di dataset omogenee (e quindi delle parti interne degli organi in un dataset medicale) sarà necessario rendere l'opacità $\alpha(\overline{\boldsymbol{v}})$ direttamente proporzionale alla norma di gradiente $\|\nabla f(\overline{\boldsymbol{v}})\|$. Nella formula finale, i valori di interesse non indicheranno quindi gli isovalori da visualizzare, ma gli isovalori dei quali si vuole evidenziare la superficie di confine.

Dati i due valori di interesse val_n e val_{n+1}, con le opacità associate α_n e α_{n+1},

Figura 8: *Rendering volumetrico eseguito utilizzando la classificazione delle opacità orientata alle regioni di confine tra gli isovalori. Si può notare come il volume dei tessuti sia quasi scomparso rispetto alla figura 3.4, e come vengano evidenziate le superfici di contatto tra le zone di diversa densità. Immagine generata con il programma* VolCastIA, *sviluppato internamente al CRS4.*

l'opacità per il voxel $(\overline{\boldsymbol{v}})$ sarà data da:

$$\alpha(\overline{\boldsymbol{v}}) = \|\nabla f(\overline{\boldsymbol{v}})\| \begin{cases} \alpha_{v_{n+1}} \left(\frac{f(\overline{\boldsymbol{v}}) - \text{val}_n}{\text{val}_{n+1} - \text{val}_n} \right) + & \\ \quad \alpha_n \left(\frac{\text{val}_{n+1} - f(\overline{\boldsymbol{v}})}{\text{val}_{n+1} - \text{val}_n} \right) & \text{se val}_n \leq f(\overline{\boldsymbol{v}}) \leq \text{val}_{n+1} \\ 0 & \text{altrimenti} \end{cases} \tag{5}$$

Si può osservare un esempio di applicazione di questo algoritmo nella figura 3.4.2.

Come si può notare, questo tipo di classificazione si basa sull'assunzione che un certo isovalore val_n confini al più con altri due isovalori val_{n-1} e val_{n+1}. Questa semplificazione è generalmente corretta nel caso di dati medicali (è raro che un tessuto confini con più di due differenti tessuti), ma può rivelarsi una fonte di errori di visualizzazione se applicata senza controlli ad altri tipi di volume.

3.5 Calcolo (mapping) del colore dei voxel

L'assegnamento del colore ai voxel è una operazione di classificazione non dissimile da quella riguardante i livelli di opacità.

Anche in questo caso, lo scenario più semplice sono i dataset formati da informazioni sul colore, che non richiedono alcuna operazione di classificazione.

L'assegnamento del colore a voxel (x, y, z, p) caratterizzati da label rientra ancora nell'ambito delle operazioni semplici ($c(\overline{\boldsymbol{v}}) \sim f(\overline{\boldsymbol{v}})$); anche la visualizzazione di voxel a cui sono associate più etichette risulta agevole, poichè il colore finale può essere ottenuto dalla composizione dei colori corrisponenti ad ognuna di esse.

Quando invece p indica un semplice valore numerico, gli algoritmi di assegnamento del colore diventano più complessi, anche se meno vari rispetto a quelli di classificazione delle opacità: il colore finale è solitamente funzione unicamente del valore del voxel ($c(\overline{\boldsymbol{v}}) \sim f(\overline{\boldsymbol{v}})$), e non dipende da altre variabili (quali il gradiente). Un approccio di questo tipo è stato utilizzato con successo da Drebin, Carpenter and Hanrahan [10], in particolare per la visualizzazione di dati medicali. Dati i valori di interesse val_n e val_{n+1}, e dato il parametro $0 \leq t_n \leq 1$ che indica la rapidità della transizione da un valore di interesse all'altro, e innanzi tutto possibile individuare n regioni di transizione di ampiezza $\text{tr}_n = \text{val}_n + t_n(\text{val}_{n+1} - \text{val}_n)$. Quindi, dati i colori associati ai valori di interesse c_n e c_{n+1}, il colore $c(\overline{\boldsymbol{v}})$ sarà:

$$c(\overline{\boldsymbol{v}}) = \begin{cases} c_n & \text{se } \text{val}_n \leq f(\overline{\boldsymbol{v}}) < \text{t}_n \\ c_{n+1}\left(\frac{f(\overline{\boldsymbol{v}})-\text{tr}_n}{\text{val}_{n+1}-\text{val}_n}\right) + & \\ \quad c_n\left(\frac{\text{val}_{n+1}-f(\overline{\boldsymbol{v}})}{\text{val}_{n+1}-\text{val}_n}\right) & \text{se } \text{tr}_n \leq f(\overline{\boldsymbol{v}}) < \text{val}_{n+1} \\ 0 & \text{altrimenti} \end{cases} \tag{6}$$

Anche qui vale l'assunzione che un isovalore val_n confini al più con altri due isovalori val_{n-1} e val_{n+1}.

I colori visibili nelle figure 3.4 3.4.2 sono stati assegnati utilizzando questa formula.

3.6 Ricostruzione dei valori dei voxel

Come detto in apertura, la visualizzazione di dati volumetrici richiede la rappresentazione a schermo di voxel disposti su una griglia nello spazio. Nella trattazione non si è tuttavia sottolineato come il voxel sia una entità *a-dimensionale*: un dataset volumetrico è in effetti equiparabile a un inseme di punti nello spazio, la cui rappresentazione "diretta" più naturale consiste indubbiamente nell'illuminazione di un pixel in corrisponenza di ognuno di essi.

Ovviamente la visualizzazione di una nuvola di punti colorati su schermo non è generalmente in grado di offrire sufficienti informazioni all'osservatore. La creazione di immagini solide come quelle osservabili nelle figure presenti in queste pagine richiede la **ricostruzione** dei valori che dovrebbero trovarsi nelle zone vuote tra un voxel e l'altro.

Il termine "ricostruzione" non è casuale, poichè i dataset volumetrici sono generalmente ottenuti tramite processi di **campionamento** per loro natura distruttivi: essi infatti rilevano l'informazione nei punti campionati, e perdono quella presente nelle altre zone. La visualizzazione di immagini solide richiede dunque della ricostruzione dell'informazione perduta — e proprio questo è uno degli aspetti centrali di ogni algoritmo di volume rendering.

3.6.1 Il problema della ricostruzione

I dati sottoposti a campionamento possono essere considerati una funzione continua nello spazio. Il concetto è facilmente visualizzabile limitandosi a una sola dimensione: per esempio, i valori di densità dei tessuti d lungo una semiretta r possono essere considerati funzione di t, distanza dall'origine della retta:

$$d = g(t) \tag{7}$$

Se i valori dei tessuti vengono campionati, per esempio, da una TAC, si ottiene una funzione $g_c(t)$, che assume il valore campionato d_t se t appartiene a T_C (insieme delle distanze di campionamento dall'origine della semiretta), e 0 negli altri punti:

$$g_c(t) = \begin{cases} d_t & \text{se } t \in T_C \\ 0 & \text{se } t \notin T_C \end{cases} \tag{8}$$

La ricostruzione dei campioni mancanti è dunque problema di **interpolazione**: dati i

campioni d_j rilevati alle distanze t_j con $j = 0, 1, \ldots, n$, e data una famiglia di funzioni, $\Phi(t, \beta_0, \beta_1, \ldots, \beta_n)$, si cercano i valori di $\beta_0, \beta_1, \ldots, \beta_n$ (gradi di libertà) tali che:

$$\Phi(t_j, \beta_0, \beta_1, \ldots, \beta_n) = d_j, \qquad j = 0, 1, \ldots, n \tag{9}$$

Una volta individuata una funzione che soddisfi questa relazione, è possibile stimarne il valore nel punto da ricostruire, ottenendo un risultato utilizzabile per la visualizzazione.

Purtroppo esistono infinite famiglie di funzioni Φ in grado di soddisfare il vincolo (9); per ogni famiglia, inoltre, la funzione identificata dai valori di $\beta_0, \beta_1, \ldots, \beta_n$ fornisce risultati di interpolazione diversi rispetto a quelli ottenibili utilizzando una funzioni proveniente da un'altra famiglia. Quale criterio utilizzare per la scelta?

Per rispondere a questa domanda vengono in aiuto dei concetti legati alla **teoria dei segnali** [30], che verranno brevemente enunciati di seguito.

La funzione $g(t)$ in (7) può essere rappresentata come un **segnale** composto dalla somma di un certo numero (possibilmente infinito) di onde sinusoidali e cosinusoidali di diversa frequenza, ampiezza e fase: questa è la cosiddetta **risposta in frequenza** di $g(t)$, rappresentata dal simbolo $G(\omega)$, e ottenibile mediante la **trasformata di Fourier**:

$$G(\omega) = \int_{-\infty}^{\infty} g(t) e^{2\pi i \omega t} dt \tag{10}$$

La funzione "originale" può essere ricavata dalla risposta in frequenza utilizzando la **trasformata inversa di Fourier**:

$$g(t) = \int_{-\infty}^{\infty} G(\omega) e^{-2\pi i \omega t} d\omega \tag{11}$$

Grazie alla trasformazione da o verso la risposta in frequenza è possibile osservare il comportamento di una funzione/segnale in due diversi "ambienti": nel **dominio spaziale** (quello utilizzato per rappresentare una funzione sul piano cartesiano) e nel **dominio delle frequenze**. La rappresentazione in un certo dominio non aggiunge informazioni nè modifica quelle disponibili, ma permette di comprendere più facilmente diverse caratteristiche e proprietà della funzione studiata: le variazioni nello spazio/tempo, per esempio, sono individuabili più facilmente nel dominio spaziale, mentre le frequenze che compongono la funzione stessa sono individuabili più facilmente dalla trasformata di Fourier.

Tra i due domini esistono (ovviamente) delle relazioni matematiche, che permettono di stabilire in che modo una certa operazione in un dominio si rifletta nell'altro. Per esempio: cosa avviene nel dominio spaziale quando le risposte in frequenza di due funzioni h_1 e h_2 (indicate con H_1 e H_2) vengono moltiplicate tra loro? Si ottiene la **convoluzione** di h_1 e h_2, operazione definita come:

$$(h_1 * h_2)(t) = \int_{-\infty}^{\infty} h_1(u) \cdot h_2(t-u)du \tag{12}$$

Il **teorema della convoluzione** evidenzia inoltre una interessante simmetria tra convoluzione e moltiplicazione di funzioni nei due domini:

$$h_1 * h_2 \Leftrightarrow H_1 \cdot H_2 \tag{13}$$

$$H_1 * H_2 \Leftrightarrow h_1 \cdot h_2 \tag{14}$$

Anche il processo di campionamento usato per la creazione dei dataset rivela interessanti caratteristiche, se osservato nel dominio delle frequenze. Nel dominio spaziale, il campionamento di un segnale non è altro che la moltiplicazione della funzione di partenza per un **treno di impulsi** emessi ad una frequenza pari a quella di campionamento. Il treno di impulsi è detto anche funzione comb (pettine):

$$\text{comb}_T(t) = \sum_{n=-\infty}^{\infty} \delta(t - nT) \tag{15}$$

dove $n \in \mathbb{N}$, T è il periodo di campionamento, e δ è la **funzione impulso**, o funzione di Dirac:

$$\delta(t) = \begin{cases} \infty & \text{se } t = 0 \\ 0 & \text{se } t \neq 0 \end{cases} \tag{16}$$

Quindi, il campionamento g_c a frequenza $1/T$ della funzione g è:

$$g_c(t) = g(t) \cdot \text{comb}_T(t) \tag{17}$$

Questa definizione di g_c è analoga a (8), assumendo che T_C contenga campioni rilevati con periodo (distanza) regolare.

Da (17) si può ricavare che se il campionamento è una operazione di moltiplicazione

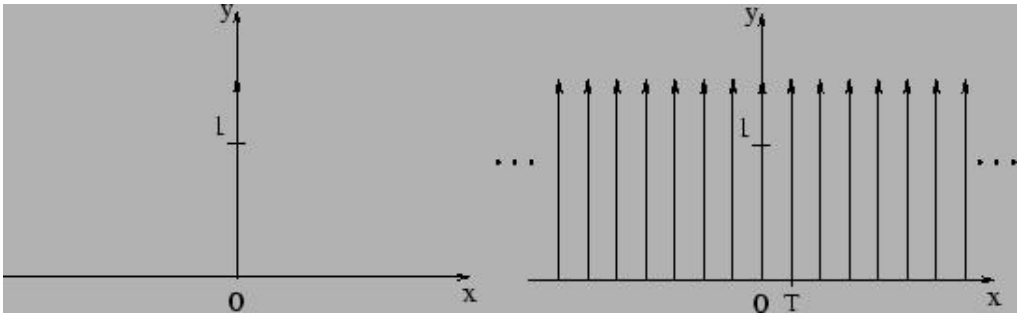


Figura 9: *Funzione impulso (a sinistra) e funzione comb (a destra). Immagine tratta da [30].*

nel dominio spaziale, dovrebbe corrispondere a una convoluzione nel dominio delle frequenze; più precisamente, una convoluzione tra la risposta in frequenza della funzione originale e la risposta in frequenza della funzione comb — che è ancora una funzione comb, ma con periodo inverso:

$$G_c(\omega) = G(\omega) * \text{comb}_{1/T}(\omega) \tag{18}$$

Ora appare chiaro (almeno intuitivamente) che per ricostruire esattamente la funzione G a partire da G_c sarebbe necessario invertire in qualche modo l'operazione di convoluzione in (18).

Il metodo di ricostruzione è effettivamente spiegato dal **teorema del campionamento** [26]. Esso richiede, innanzi tutto, che il segnale $G(\omega)$ da ricostruire sia a **banda limitata**, ovvero composto da frequenze comprese in un intervallo finito: $-u_m \leq \omega \leq u_m$. Se questa condizione è verificata, u_m è detta **larghezza di banda** di G.

Il teorema afferma che una funzione a banda limitata può essere esattamente ricostruita a partire dai suoi campioni solo se la frequenza di campionamento $1/T$ è maggiore o uguale al doppio della sua larghezza di banda u_m. Il valore $\omega_N = 2u_m$ è detto **frequenza di Nyquist**.

Se queste condizioni sono rispettate, è possibile ricostruire un segnale moltiplicando i suoi campioni $G_c(t)$ per la cosiddetta **box function**:

$$\Pi_{\omega_N}(t) = \begin{cases} 1 & \text{se } |t| \leq \frac{\omega_N}{2} \\ 0 & \text{se } |t| > \frac{\omega_N}{2} \end{cases} \tag{19}$$

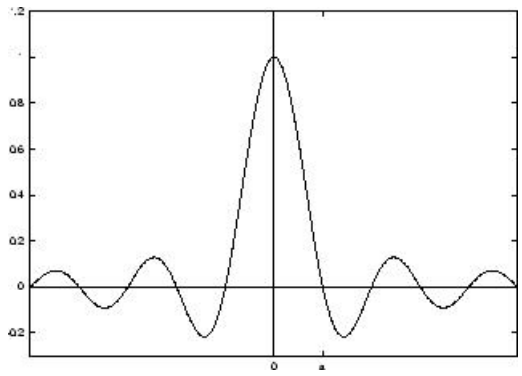


Figura 10: *Funzione sinc. Immagine tratta da [30].*

L'operazione di ricostruzione di G diventa quindi:

$$G(\omega) = G_c(\omega) \cdot \Pi_{\omega_N}(\omega) \tag{20}$$

Se tale moltiplicazione viene convertita in una convoluzione nel dominio spaziale, si ottiene:

$$g(t) = g_c(t) * (\omega_N \text{sinc}(\omega_N t)) \tag{21}$$

Dove $\omega_N \text{sinc}(\omega_N t)$ è la trasformata inversa di Fourier della box function di ampiezza ω_N, e la funzione sinc è definita come:

$$\text{sinc}(t) = \begin{cases} \frac{\sin(\pi t)}{\pi t} & \text{se } t \neq 0 \\ 1 & \text{se } t = 0 \end{cases} \tag{22}$$

Quindi, tornando alla domanda di partenza: come scegliere la miglior funzione di interpolazione per il dataset? Dovrebbe ora apparire chiaro come il candidato ideale sia la funzione sinc, *teoricamente* in grado di ricostruire esattamente i dati di partenza.

Poichè la funzione campionata g_c è diversa da 0 solo per valori discreti, l'integrale nella definizione di convoluzione (12) può essere convertito in una sommatoria, che applicata in (21) fornisce l'equazione di ricostruzione finale:

$$g(t) = \sum_{u=-\infty}^{\infty} d_u \cdot \omega_N \text{sinc}(\omega_N (t - d_u)) \tag{23}$$

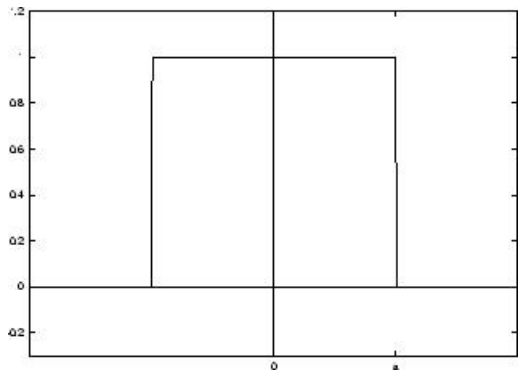


Figura 11: *Funzione box. Immagine tratta da [30].*

dove d_u è il valore dell'u-esimo campione (si veda la definizione di d_t in (8)).

L'adattamento di questa formula a un dataset tridimensionale composto da un insieme finito S di voxel (x, y, z, p') campionati alla frequenza di Nyquist è abbastanza semplice: è innanzi tutto necessario sostituire i termini $+\infty$ e $-\infty$ della sommatoria con l'estenzione massima del dataset, e applicare la ricostruzione nelle tre dimensioni (sfruttando la proprietà di *separabilità* della convoluzione). Quindi, se f è una generica funzione applicata a un voxel (come definito nella sez. 3.1), allora f_r (la medesima funzione calcolata in un punto arbitrario mediante interpolazione) sarà:

$$\begin{aligned} f_r(x_r, y_r, z_r, p') = \sum_{z=0}^{\dim_z - 1} \sum_{y=0}^{\dim_y - 1} \sum_{x=0}^{\dim_x - 1} & f(x, y, z, p') \cdot \\ & \cdot \omega_N \text{sinc}(\omega_N (x_r - x)) \cdot \\ & \cdot \omega_N \text{sinc}(\omega_N (y_r - y)) \cdot \\ & \cdot \omega_N \text{sinc}(\omega_N (z_r - z)) \end{aligned} \tag{24}$$

Dove:

- p' che indica una proprietà numerica reale del dataset (la condizione è necessaria affinchè $f(x, y, z, p') : S \to \mathbb{R}$);
- $x_r \in \mathbb{R}^+, x_r < \dim_x$;
- $y_r \in \mathbb{R}^+, y_r < \dim_y$;
- $z_r \in \mathbb{R}^+, z_r < \dim_z$.

3.6.2 La ricostruzione in pratica

La ricostruzione dei valori mancanti in un dataset discreto appare dunque un compito semplice: supponendo che il dataset contenga funzioni a banda limitata, e che sia stato campionato sopra la frequenza di Nyquist, sarebbe sufficiente effettuare la convoluzione tra i campioni e la funzione sinc descritta nell'equazione (22), utilizzando la formula (24).

Vi sono tuttavia diversi problemi pratici a questo approccio:

1. difficilmente i dataset derivanti da dati reali sono campionati a una frequenza maggiore o uguale a quella di Nyquist;

2. anche quando i dataset sono campionati alla frequenza ottimale, occore considerare che la funzione sinc si estende all'infinito: di conseguenza i termini della sommatoria nell'equazione (23) non dovrebbero essere ridotti (come è stato invece fatto nell'equazione (24)). La *vera* ricostruzione dei dati richiederebbe un numero di campioni infinito, e dovrebbe essere risolta una sommatoria infinita: si tratta di un compito irrealizzabile, specialmente con strumenti informatici.

Il primo problema, se presente, non è risolvibile nè attenuabile: l'informazione necessaria alla ricostruzione del segnale è andata perduta, e l'immagine risultante sarà necessariamente soggetta alla presenza di artefatti (**aliasing**) e ricostruzioni arbitrarie più o meno visibili.

Anche il secondo problema non è risolvibile, ma è attenuabile utilizzando il più alto numero di voxel possibile per la ricostruzione del segnale (cosa che in effetti avviene nell'equazione (24)).

Da qui nascono altri due problemi:

1. il numero di voxel che compongono un dataset è molto alto (mediamente 256 per ogni lato, ovvero 16.777.216 voxel in totale), ed è impensabile utilizzare questa mole di valori per ogni interpolazione da effettuare durante il processo di visualizzazione;

2. la funzione sinc è computazionalmente dispendiosa, poichè prevede il calcolo di un seno e una divisione.

Entrambi i problemi possono essere attenuati utilizzando una funzione di interpolazione alternativa, che entro certi limiti approssimi il comportamento della funzione sinc, e fornisca buoni risultati con un basso numero di campioni. L'equazione di convoluzione da applicare ai voxel diventa quindi del tipo:

$$\begin{aligned} f_r(x_r, y_r, z_r, p') = \sum_{\text{lower}_i(z_r)}^{\text{upper}_i(z_r)} \sum_{\text{lower}_i(y_r)}^{\text{upper}_i(y_r)} \sum_{\text{lower}_i(x_r)}^{\text{upper}_i(x_r)} & f(\overline{\boldsymbol{v}}) \cdot \\ & \cdot \omega_X \text{interp}_i(\omega_X(x_r - x)) \cdot \\ & \cdot \omega_Y \text{interp}_i(\omega_Y(y_r - y)) \cdot \\ & \cdot \omega_Z \text{interp}_i(\omega_Z(z_r - z)) \end{aligned} \tag{25}$$

Dove:

- ω_X, ω_Y e ω_Z sono le frequenze di campionamento rispettivamente sull'asse X, Y e Z del dataset;
- $\text{interp}_i(t)$ è la funzione di interpolazione da utilizzare al posto di $\text{sinc}(t)$;
- $\text{lower}_i(x_r)$ e $\text{upper}_i(x_r)$ sono due funzioni che restituiscono il termine di sommatoria rispettivamente più basso e più alto da utilizzare per la ricostruzione, tali che valga la relazione $\text{lower}_i(x_r) \leq x_r \leq \text{upper}_i(x_r)$.

La funzione di interpolazione prescelta dovrebbe avere una risposta in frequenza non troppo dissimile da quella della funzione sinc: ogni differenza, infatti, darà come risultato distorsioni e artefatti nei dati ricostruiti.

Com'è facile intuire, questo campo di ricerca è decisamente ampio: esistono diversi tipi di funzioni di interpolazione, che hanno comportamenti diversi al variare della conformazione del dataset, e prevedono costi computazionali più o meno elevati.

Nonostante la sua importanza, è peraltro interessante notare come il tema della ricostruzione dei valori abbia avuto una attenta considerazione solamente negli ultimi anni. Una trattazione approfondita si può trovare in [30], [7] e [21]; qui di seguito verrà fornito un breve elenco dei sistemi di interpolazione più utilizzati per il volume rendering.

L'interpolazione più semplice è la **nearest neighbour interpolation**, che assegna al punto da interpolare lo stesso valore rilevato nel voxel più vicino. L'utilizzo di questa funzione corrisponde ad una convoluzione tra i voxel e la box function definita in (19). Tale interpolazione richiede quindi due soli punti per ogni direzione del dataset (per un

totale di 8 voxel); tuttavia la funzione utilizzata ha un comportamento estremamente diverso dalla funzione $sinc$, e questo è invevitabilente fonte di artefatti nell'immagine ricostruita.

La necessità di limitare il numero di campioni utilizzati per l'interpolazione porta in modo naturale all'utilizzo di **funzioni interpolanti a tratti**, in particolare di tipo **spline**. Dato un intervallo $[a, b], a, b \in \mathbb{R}$ suddiviso in n nodi $a = x_0, x_1, \ldots, x_n$, una funzione spline di grado p con nodi $a = x_0, x_1, \ldots, x_n$ è una funzione su $[a, b]$ indicata con il simbolo $s_p(t)$ tale che:

1. su ogni intervallo $[x_i, x_{i+1}], n = 0, 1, \ldots, n-1$ è un polinomio di grado p;
2. la funzione $s_p(x)$ e le sue prime $p-1$ derivate sono continue in $[a, b]$.

Le spline più semplici sono le spline di grado 1, che vengono utilizzate per quella che è comunemente (e impropriamente) chiamata **interpolazione lineare**. Essa equivale a una convoluzione tra i valori dei voxel e la **funzione tent**, definita come:

$$\text{tent}_T = \begin{cases} 1 - \frac{|t|}{T} & \text{se } |t| < T \\ 0 & \text{altrimenti} \end{cases} \tag{26}$$

Anche la risposta in frequenza della funzione tent è decisamente diversa da quella della funzione sinc, e ciò causa la presenza di evidenti artefatti e aliasing nell'immagine ricostruita (osservabili specialmente ad alti livelli di ingrandimento). Tuttavia, la semplicità del calcolo (che richiede solamente 2 nodi per ogni direzione, per un totale di 8 voxel) rende le spline di grado 1 il metodo di interpolazione più utilizzato per il volume rendering.

Anche le spline di grado 3 (o **cubic spline**) sono ampiamente utilizzate per il volume rendering, anche se la loro complessità computazionale le rende generalmente sconsigliabili per processi di visualizzazione interattiva: esse infatti richiedono 4 nodi di interpolazione in ogni direzione, per un totale di 64 voxel; questo generalmente limita il loro utilizzo alle fasi di resampling o pre-processamento dei dati, che generalmente non hanno requisiti di interattività da rispettare. I risultati ottenibili sono assai migliori rispetto all'uso delle spline di grado 1.

Un altro approccio al processo di ricostruzione consiste nell'utilizzare come interpolatore la funzione sinc, limitando però la sua estensione a un numero finito di campioni: ciò richiede la **finestrazione (windowing)** di sinc, ovvero la sua moltiplicazione per

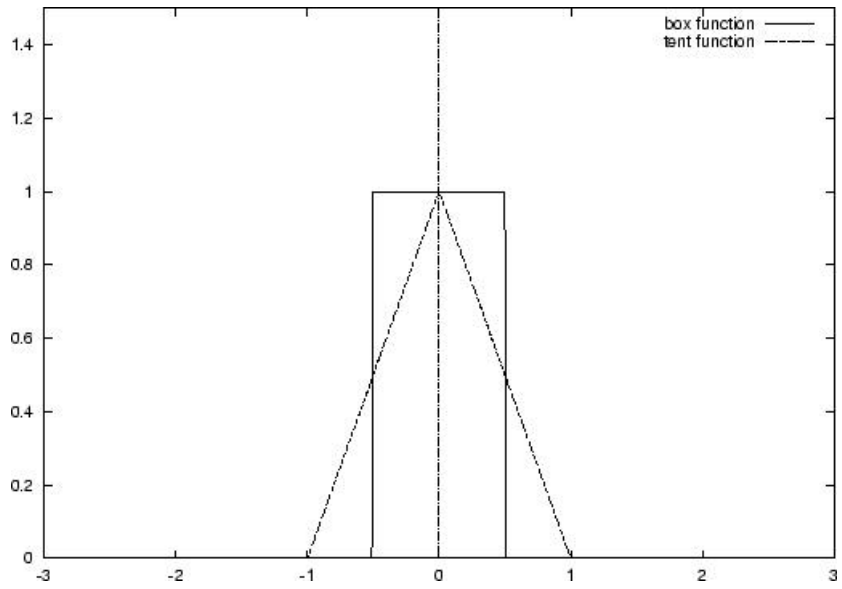


Figura 12: *La funzione tent e la funzione box, corrispondenti rispettivamente all'interpolazione lineare e nearest-neighbour. Immagine tratta da [30].*

un'altra funzione (detta finestra) dall'estensione finita. Questo approccio fornisce generalmente buoni risultati, ma richiede un inevitabile aumento dei costi computazionali; inoltre la scelta della funzione di windowing è un problema dalla soluzione non sempre semplice, dato che diverse finestre creano diverse alterazioni nella risposta in frequenza della funzione sinc originaria. Tra le finestre più comuni si possono citare la funzione box (che di fatto genera a un troncamento della funzione sinc) e la funzione tent definita in (26) (che utilizzata nel processo di windowing prende il nome di finestra di Bartlett). Una trattazione approfondita dell'argomento si può trovare in [32], [33], [31] e [34].

3.7 Ricostruzione del gradiente

Come già illustrato nella sezione 3.1, il vettore gradiente indica l'intensità e la direzione delle variazioni dei valori dei voxel. Anche il calcolo dei gradienti in un dataset è una operazione di ricostruzione, riguardante la *derivata prima* della funzione dei valori dei voxel. Le basi teoriche sono le medesime enunciate nella sezione 3.6.1; ma al posto della funzione sinc, l'equazione di ricostruzione utilizza la sua derivata prima — la

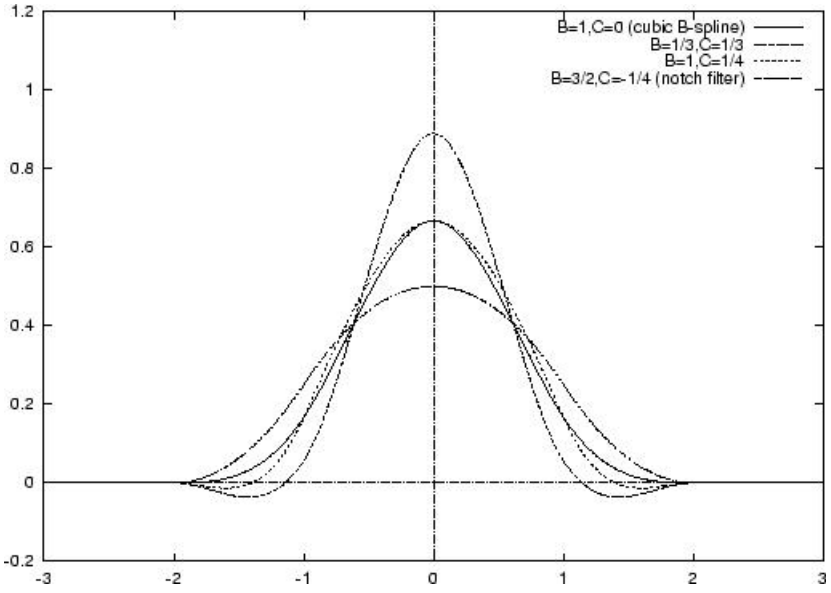


Figura 13: *Il grafico di diversi tipi di spline di grado 3. Immagine tratta da [30].*

funzione cosc:

$$\text{cosc}(t) = \begin{cases} \frac{\cos(\pi t)}{t} - \frac{\sin(\pi t)}{\pi t^2} & \text{se } t \neq 0 \\ 0 & \text{se } t = 0 \end{cases} \tag{27}$$

Purtroppo la funzione di interpolazione cosc ha gli stessi problemi della funzione sinc: ha una estensione spaziale infinita, e il suo utilizzo corretto per il processo di interpolazione richiederebbe la risoluzione di una sommatoria infinita. È quindi necessario utilizzare una formula alternativa per la stima dei gradienti.

La ricostruzione dei gradienti può risultare tuttavia più semplice rispetto alla ricostruzione dei valori: mentre nel secondo caso l'interpolazione avviene in punti arbitrari nelle zone vuote tra i voxel stessi, la stima dei gradienti viene solitamente effettuata soltanto in corrispondenza dei voxel; i gradienti nelle zone intermedie vengono successivamente interpolati a partire da quelli calcolati in prima istanza.

Questo significa che per la stima dei gradienti non è necessario ricalcolare i pesi di ogni voxel, poichè la loro distanza sarà costante rispetto al centro della funzione di convoluzione. Diventa quindi possibile usare le **matrici di convoluzione**, formate da un peso precalcolato per ogni elemento. Il loro utilizzo è estremamente semplice:

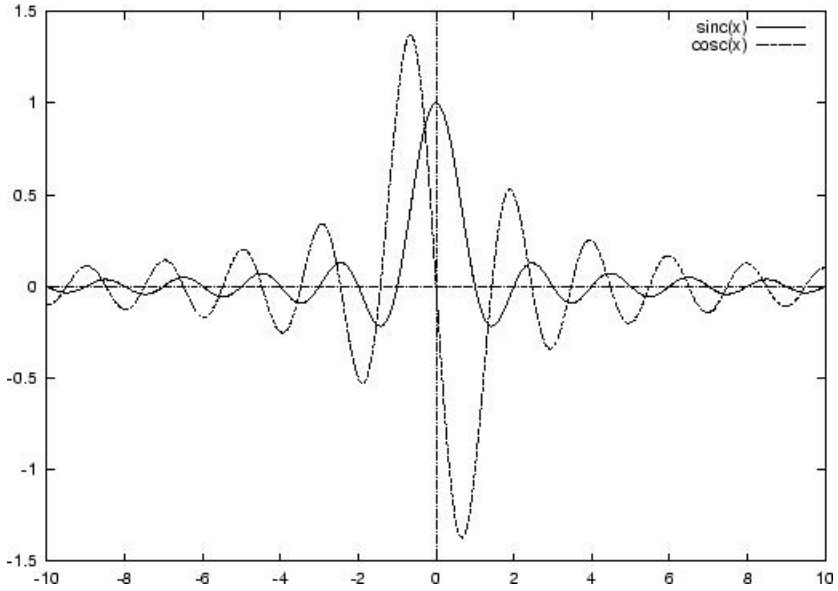


Figura 14: *Grafico della funzione cosc e della funzione sinc. Immagine tratta da [30].*

1. la matrice viene centrata sul voxel in corrispondenza del quale si intende calcolare il gradiente;

2. gli elementi vengono moltiplicati per i voxel corrispondenti;

3. i risultati vengono sommati, ottenendo la componente del gradiente in una delle direzioni X, Y o Z;

4. la matrice viene ruotata, e il procedimento ripetuto per ottenere la componente del gradiente nelle direzioni restanti.

Esistono diverse matrici di convoluzione, che calcolano gradienti caratterizzati da diverse reazioni alle variazioni dei valori del dataset. Tai differenze di comportamento sono particolarmente importanti quando si ricorre alla classificazione orientata alle superfici di confine tra gli isovalori (sez. 3.4.2), o all'ombreggiatura del volume (che verrà affrontata nella sezione 3.8): in questi casi gran parte della resa grafica della rappresentazione dipende dalla norma e dalla direzione del gradiente, che influenzano la continuità e la regolarità delle superfici evidenziate.

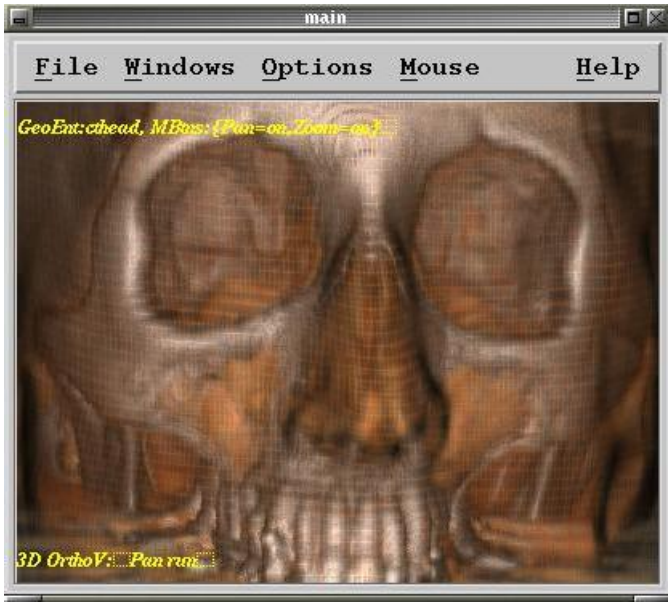


Figura 15: *Rendering volumetrico eseguito utilizzando il filtro gradiente "central difference". Immagine generata con il programma* VolCastIA*, sviluppato internamente al CRS4.*

La matrice di convoluzione più comunemente utilizzata è quella per il calcolo della cosiddetta **central difference**:

$$\begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \tag{28}$$

Essa corrisponde al calcolo del valor medio tra i due voxel situati in prossimità del punto di applicazione del gradiente, lungo la direzione della stima. Questo operatore presenta un buon comportamento nel caso comune, ed è per questo il filtro più utilizzato.

Un'altra matrice di convoluzione è stata creata da Irwin Sobel, e deriva dall'omoni-

mo filtro da lui sviluppato per la rilevazione dei contorni in immagini bidimensionali:

$$\begin{bmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -2 & 0 & 2 \end{bmatrix} \quad (\text{per } x = -1)$$

$$\begin{bmatrix} -3 & 0 & 3 \\ -6 & 0 & 6 \\ -3 & 0 & 3 \end{bmatrix} \quad (\text{per } x = 0) \tag{29}$$

$$\begin{bmatrix} -2 & 0 & 2 \\ -3 & 0 & 3 \\ -3 & 0 & 3 \end{bmatrix} \quad (\text{per } x = 1)$$

Questa matrice genera gradienti più omogenei rispetto a (28), ed è in grado di eliminare gran parte delle variazioni più brusche e del rumore nel dataset. Le superfici generate dall'utilizzo di questo gradiente sono quindi generalmente continue e poco scabrose; il notevole arrotondamento generato non è tuttavia sempre positivo, e può far perdere dettagli importanti nell'immagine finale.

Al contrario, il filtro denominato **intermediate difference** riesce a rilevare le minime variazioni all'interno dei dati:

$$\begin{bmatrix} -1 & 1 \end{bmatrix} \tag{30}$$

Questa matrice di convoluzione richiede solamente due voxel (quello di applicazione del gradiente, ed uno adiacente). La sua estrema reattività ai cambiamenti dei valori tende a generare superfici molto scabrose che, sebbene non sempre apprezzabili visivamente, riescono a rappresentare efficacemente l'irregolarità e la variazione dei valori all'interno del volume.

L'utilizzo delle matrici di convoluzione non è l'unico sistema disponibile per il calcolo dei gradienti: come nel caso della ricostruzione dei valori del dataset, la ricerca della migliore funzione di approssimazione è un argomento vasto, oggetto di trattazione estensiva. Maggiori dettagli sono disponibili in [30], [7], [29], [20].

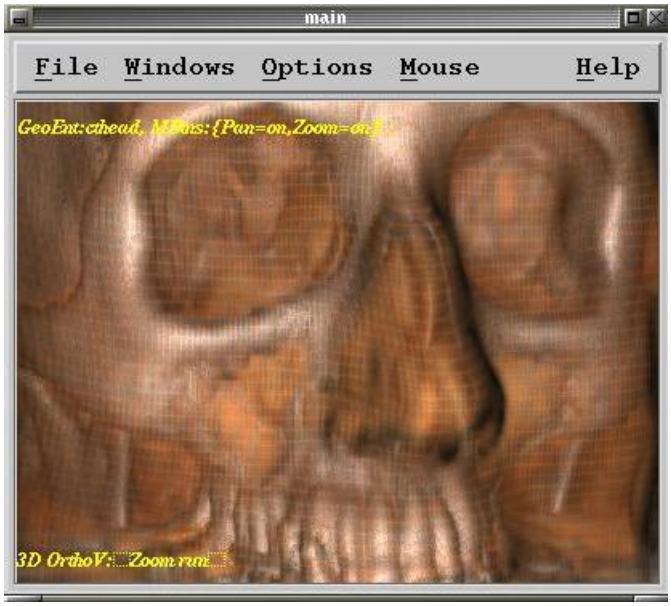


Figura 16: *Rendering volumetrico eseguito utilizzando il filtro gradiente sviluppato da Irwin Sobel. Il risultato è più omogeneo e gradevole alla vista rispetto a 3.7, ma si può notare una sensibile perdita di dettaglio. Immagine generata con il programma* VolCastIA*, sviluppato internamente al CRS4.*

3.8 Luci e ombreggiatura (shading)

Le rappresentazioni tridimensionali generate al computer sono rese spesso più realistiche e comprensibili attraverso l'utilizzo di effetti di luce e ombreggiatura. Al di là degli aspetti estetici, l'utilizzo corretto delle luci è infatti in grado di comunicare più facilmente la sensazione di tridimensionalità degli oggetti trattati, con un conseguente aumento dell'informazione comunicata all'utente.

Il modello più semplice di "illuminazione" utilizzabile nel volume rendering (e non solo) è il **depth shading**, ovvero il progressivo oscuramento della scena in base alla distanza dall'osservatore. Questa tecnica simula in realtà la presenza di una sorta di nebbia in grado di attenuare la penetrazione dei raggi luminosi e la visibilità degli oggetti più lontani — ma una impostazione adeguata dei parametri può (a larghe linee) simulare una illuminazione frontale degli oggetti.

L'equazione più utilizzata per la "vera" simulazione dell'illuminazione e dell'ombreggiatura degli oggetti è sicuramente l'equazione di Phong [2]. Essa è generalmente

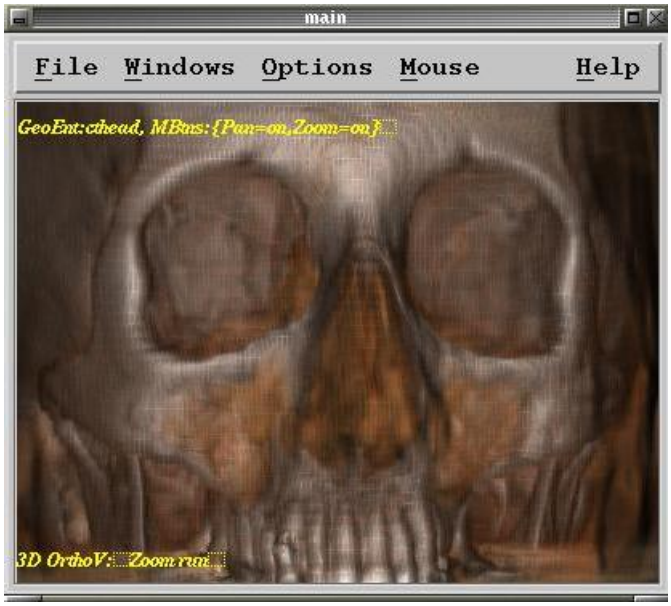


Figura 17: *Rendering volumetrico eseguito utilizzando il filtro gradiente "intermediate difference". Le superfici generate sono visibilmente più scabrose e irregolari di quelle nelle figire 3.7 3.7, ma il livello di dettaglio è maggiore. Immagine generata con il programma* VolCastIA*, sviluppato internamente al CRS4.*

utilizzata per lo shading di superfici analitiche o poligonali, e tra i parametri richiede il vettore normale al punto di illuminazione. Il concetto stesso di "normale della superficie" è decisamente estraneo al processo di volume rendering — ma tale parametro può essere sostituito dal gradiente calcolato nel punto da ombreggiare. La tecnica è estremamente efficace (come si può osservare nelle diverse immagini presenti in queste pagine), ma i risultati grafici dipendono largamente dal sistema utilizzato per la ricostruzione del gradiente.

3.9 Spazi di colore

Com'è noto, le immagini sul monitor di un computer nascono dalla composizione di pixel formati da tre sorgenti di luce monocromatica di diversi colori: rosso, verde e blu. Questa tecnica per la creazione delle immagini deriva dagli studi sul sistema di percezione visiva dell'essere umano: la retina dell'uomo ospita tre tipi di recettori per il colore (chiamati **coni** di tipo S, M ed L), sensibili a tre diversi spettri luminosi, separatamente

percepiti come luce rossa, verde e blu.

Tali spettri luminosi non sono tuttavia monocromatici (ovvero, composti da onde di una sola frequenza), e ciò ha reso necessaria la ricerca di una semplificazione — ovvero di tre frequenze di luce monocromatica che, sommate in diversi modi, siano in grado di comunicare adeguatamente il colore all'occhio umano. Gli esperimenti svolti dalla CIE (Commission Internationale d'Eclairage) hanno evidenziato come l'occhio sia grado di riconoscere efficacemente i colori dalla composizione di onde alla frequenza di 700 nm (rosso), $546,1$ nm (verde) $435,8$ (blu). Queste frequenze dovrebbero essere idealmente utilizzate, per esempio, in tutti i monitor.

Il sistema RGB per la memorizzazione e gestione del colore usato in campo informatico (e non solo) deriva da questi esperimenti; esso non è altro che uno **spazio di colori** in tre coordinate positive (R, G e B), che, usate come coefficienti delle frequenze definite da CIE, individuano la posizione di un determinato colore all'interno del sistema di riferimento.

L'utilizzo del formato RGB per la memorizzazione e gestione dei colori è estremamente efficiente, poichè esso rispecchia la gestione del colore da parte dell'hardware. Questo approccio presenta tuttavia dei limiti:

1. la resa cromatica di un certo colore espresso in coordinate RGB dipende strettamente dal dispositivo in uso: un monitor che non emetta le "giuste" frequenze R, G e B, per esempio, creerà un'immagine con colori diversi rispetto a un'altra periferica dotata di altre frequenze;

2. lo spazio RGB non è **percettivamente uniforme**: uno spostamento lineare nello spazio RGB non è sempre percepito come una variazione di colore uniforme dall'occhio umano, che ha purtroppo una risposta al tristimolo estremamente complessa.

Per risolvere il primo problema è possibile utilizzare il modello dell'**osservatore standard** sviluppato nel 1931 da CIE: esso si basa su uno spazio di colore chiamato **CIE-XYZ**, modellato matematicamente in modo indipendente dai dispositivi di input/output grafico. Sono previste tre coordinate, X, Y e Z, usate come coefficienti di tre **curve di tristimolo** ($\overline{x}(\lambda)$, $\overline{y}(\lambda)$ e $\overline{z}(\lambda)$) disegnate sulla base delle risposte alle differenti frequenze luminose dell'occhio umano.

Questo spazio di colori non può essere utilizzato direttamente dai dispositivi grafici: è necessaria una trasformazione verso il modello RGB. L'utilità del sistema CIE-XYZ

appare ora evidente: se i produttori dell'hardware grafico (monitor, stampanti, ecc.) dichiarano quale sia il **profilo** del propria prodotto (ovvero la coordinata RGB corrispondente a una certa coordinata XYZ), diventa possibile la trasformazione tra spazi diversi tenendo conto dell'intervallo di colori rappresentabile (**gamut**) e delle distorsioni introdotte dalla periferica; il medesimo colore XYZ può essere quindi visualizzato allo stesso modo su qualsiasi dispositivo. Questo sistema è già in uso: i profili di gestione del colore delle periferiche grafiche seguono oggi uno standard chiamato **CMS (Color Management System)**.

Resta da risolvere il problema 2, ovvero la mancanza di uniformità percettiva del sistema RGB — che purtroppo si può riscontrare anche nel sistema CIE-XYZ (fig. 3.9). Questo aspetto può risultare estremamente importante per la significatività del processo di visualizzazione volumetrica: la ricostruzione dei valori nel dataset (illustrata nella sezione 3.6.1) porta in genere alla necessità di eseguire una interpolazione tra i colori di voxel adiacenti — operazione equivalente ad uno spostamento nello spazio RGB. Poichè non vi è garanzia che la transizione venga ritenuta "progressiva" anche dall'occhio umano, vi è il rischio che nuovi colori, percepiti come "estranei", compaiano tra un voxel e l'altro. Questo potrebbe trarre in inganno l'osservatore, specie se tali colori risultano simili ad altri già assegnati a specifici materiali all'interno del dataset.

Questo inconveniente è risolto dai modelli **CIE-Luv** e **CIE-Lab**, derivati dal sistema CIE-XYZ, che introducono nello spazio di colore delle distorsioni studiate in modo da far corrispondere uno spostamento progressivo nel sistema di riferimento ad un cambiamento di colore percepibile come "corretto" dall'occhio umano [6].

Per garantire una maggiore correttezza visiva, quindi, le operazioni sul colore nel processo di volume rendering dovrebbero essere eseguite in spazi come CIE-Luv o CIE-Lab, per essere riportate nel sistema RGB (eventualmente passando per CIE-XYZ) solo alla fine dell'elaborazione.

Per il massimo dell'accuratezza, le operazioni potrebbero essere addirittura eseguite sulle **distribuzioni spettrali di potenza del colore**, ovvero sugli istogrammi che rappresentano fedelmente le frequenze luminose contenute in un raggio luminoso, o assorbite da un certo materiale. La mole di operazioni necessarie per il volume rendering, in questo caso, cresce enormemente — e l'applicabilità e convenienza di queste tecniche deve essere ovviamente valutata in base ai casi.

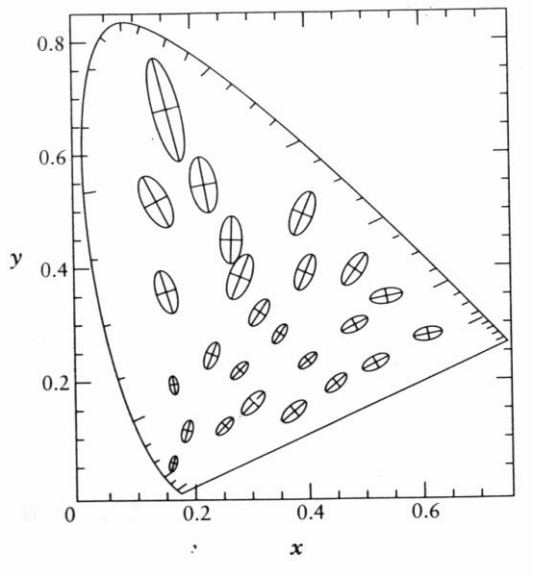


Figura 18: *Ellissi di MacAdam nello spazio di colore CIE-XYZ (qui mostrato in sezione). La mancanza di uniforme percettività è evidenziata dalla forma delle ellissi, che indicano delle regioni di colore percepite come omogenee da soggetti sottoposti a prove sperimentali: nel caso ideale, esse dovrebbero risultare delle circonferenze. Immagine tratta da [6].*

3.10 Prestazioni

Come già illustrato nell'introduzione, le prestazioni sono cruciali nel processo di volume rendering. La necessità di dover gestire griglie tridimensionali di diversi milioni di voxel è fonte di due problemi:

1. il trasferimento di enormi quantità di dati dalla memoria, che diventa il principale collo di bottiglia nel processo di elaborazione;

2. l'altissimo numero di operazioni da eseguire;

La letteratura presenta numerosi metodi nati per risolvere questi problemi, con costi e prestazioni estremamente differenti. I tre metodi principali utilizzati per l'accelerazione del processo di volume rendering comprendono il ricorso ad hardware dedicato, l'utilizzo di schede di accelerazione grafica già presenti sul mercato, o l'ottimizzazione software.

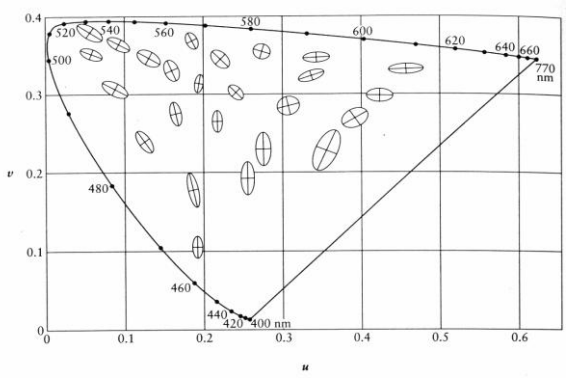


Figura 19: *Ellissi di MacAdam nello spazio di colore CIE-Luv (qui mostrato in sezione). La forma più arrotondata delle ellissi indica una maggiore uniformità percettiva rispetto al modello CIE-XYZ nella figura 3.9. Immagine tratta da [6].*

3.10.1 Hardware dedicato

La letteratura presenta numerose proposte di hardware dedicato per il volume rendering. L'approccio comune nella progettazione consiste nell'utilizzo di diversi canali (**pipeline**) di rendering, e nell'ottimizzazione del flusso di dati dalla memoria.

I primi esempi risalgono agli anni '80; uno dei casi più interessanti è sicuramente l'architettura **CUBE** [12], che si è evoluta fino al modello **CUBE-4** nel 1996. Da questi sistemi sono poi nate delle schede di accelerazione per PC, commercializzate da Mitsubishi™ con il nome di VolumePro™ [25]. Altri esempi più recenti, implementati su schede per PC ma non immessi sul mercato, sono stati sviluppati dall'Università di Tubingen in Germania: **VIZARD** [15] e **VIZARDII** [11] [23].

Sebbene questi sistemi offrano prestazioni notevoli, essi soffrono di due problemi fondamentali. Il primo è sicuramente il costo, che per il momento non può che essere decisamente elevato a causa dell'elevato livello tecnologico e del basso volume di mercato che caratterizza questo tipo di hardware. Il secondo problema è la flessibilità, che non può che essere ridotta: è ben difficile creare un hardware programmabile e performante, in grado di adattarsi a tutte le situazioni di utilizzo del volume rendering.

3.10.2 Acceleratori grafici non dedicati

Come già detto nell'introduzione, negli ultimi anni l'industria videoludica è diventata uno dei settori trainanti della ricerca e sviluppo in campo informatico. La presenza sul

mercato di schede di accelerazione 3D a basso costo è il sintomo più evidente di questo scenario — e le prestazioni di tali schede sono cresciute tanto da permettere approcci al volume rendering fino a poco tempo fa ristretti alle architetture di calcolo di fascia alta.

La tecnica più semplice da implementare con queste schede è sicuramente il **texture mapping volume rendering**: esso prevede la creazione (diretta, o con eventuale ricampionamento) di una serie di texture RGBA a partire dai valori delle fette del dataset; tali texture vengono quindi sovrapposte utilizzando il sistema di composizione (blending) fornito dall'hardware.

Le prestazioni della tecnica sono notevoli, e rendono estremamente semplice la visualizzazione contemporanea di primitive volumetriche e poligonali (un compito assai più complesso da affrontare con strumenti software [18]). Il sistema, tuttavia, offre una flessibilità decisamente ridotta: generalmente i valori RGBA che formano le texture devono essere ricalcolati in blocco per ogni modifica dei parametri di visualizzazione. Non è quindi possibile, per esempio, osservare in tempo reale l'effetto visuale della variazione dei parametri di classificazione dei voxel.

Gli acceleratori 3D più moderni presenti sul mercato consumer permettono tuttavia di superare (almeno in parte) questo problema: i modelli più recenti sono infatti sono dotati di **GPU (Graphic Processing Unit** estremamente potenti, e soprattutto dotate di pipeline di calcolo programmabili. È il caso delle schede prodotte, per esempio, da nVidia™ e ATI™, che sono in grado di eseguire almeno parte della mole di calcoli necessari per le tecniche di volume rendering non completamente basate su texture mapping [5].

3.10.3 Ottimizzazioni software

Il processo di volume rendering si presta all'utilizzo di varie ottimizzazioni al livello software, che possono incrementare le prestazioni della visualizzazione anche su hardware non dedicato e privo di acceleratori per la grafica 3D.

L'approccio più comunemente usato per l'incremento delle prestazioni è il ricorso al parallelismo, e quindi ad architetture multiprocessore o a cluster di computer che si occupino di renderizzare singole parti di dataset.

Un altro approccio (non esclusivo rispetto al precedente) è la ricerca della **cache coherency**. Sulle architetture di fascia medio/bassa, il principale collo di bottiglia per l'elaborazione è l'accesso alla memoria. Mentre i processori hanno raggiunto elevate

prestazioni, la velocità di trasferimento dei dati tra il processore stesso e la RAM deve essere limitata per ragioni di contenimento dei costi. Questo spiega come mai si utilizzino le **cache**, ovvero buffer di memoria ad alte prestazioni che mantengono una copia dei dati usati più recentemente: si tratta di una via di mezzo che consente di ridurre il numero di accessi alla RAM, assai più lenta ma più a buon mercato.

L'unità di memorizzazione della cache è chiamata **cacheline**, e la sua lunghezza varia in base all'architettura in uso: essa ammonta, per esempio, 32 byte su processori Intel™ della famiglia ia32, e 64 byte su processori AMD™ e Intel™ ia64 (Itanium™). Ogni lettura dalla memoria, quindi, copierà in cache non solo i byte richiesti, ma anche quelli adiacenti, per un totale pari alla lunghezza della cacheline.

Tenendo conto di questi fattori, nello sviluppo di un algoritmo di volume rendering ottimizzato devono essere seguite due linee guida nella pianificazione degli accessi alla memoria:

coerenza spaziale i dati che vengono solitamente utilizzati in combinazione dovrebbero trovarsi in locazioni di memoria attigue, contenibili in poche cacheline, in modo da poter essere caricati contemporaneamente in cache con poche letture;

coerenza temporale tutte le elaborazioni che riguardano un certo insieme di dati dovrebbero essere svolte in un breve intervallo di tempo, in modo che l'esecuzione di altre istruzioni non elimini i dati richiesti dalla cache.

Questo tipo di accesso alla memoria è usato in modo estremamente efficiente dall'algoritmo di shear-warping a cui si è accennato nella sezione 3.2: poichè le fette di dataset vengono visualizzate sequenzialmente, ed anche l'accesso ai voxel è sequenziale, si ha la certezza che la lettura di un voxel provochèrà il caricamento in cache anche dei valori successivi, i quali verranno utilizzati entro breve tempo nei calcoli per la visualizzazione.

Per questo motivo, la tecnica di shear-warping risulta la tecnica attualmente più efficiente per la visualizzazione dei volumi — a condizione che si utilizzi una proiezione assonometrica dell'ambiente tridimensionale (ovvero, si utilizzino raggi visuali paralleli tra loro). Purtroppo la proiezione assonometrica è fonte di errori nell'interpretazione delle profondità del dataset, e non è sempre utilizzabile; inoltre, l'utilizzo dello shear-warping con proiezione prospettica risulta assai meno efficiente.

Queste limitazioni della tecnica sono state affrontate dal sistema **UltraVis**, sviluppato da Gunter Knittel[16], che implementa un motore estremamente efficiente per il

rendering prospettico di dati volumetrici. Le prestazioni del sistema si basano su un gran numero di ottimizzazioni rivolte al processore Intel™ Pentium™ III, riguardanti in particolare:

gestione della cache la disposizione in memoria dei dati (voxel, lookup table, ecc.) è ottimizzata sulla base degli algoritmi di popolazione della cache dell'architettura Intel™. In questo modo è possibile assicurarsi che la lettura di nuovi voxel andrà a sostituire i valori dei vecchi voxel in cache, e non andrà per esempio a eliminare una struttura di accelerazione (p. es. un octree) che dovrebbe restare sempre in cache;

uso estensivo dell'assembly gran parte del codice di volume rendering di UltraVis è stato scritto in assembly, e sfrutta le estensioni SIMD (Single Instruction, Multiple Data) introdotte da Intel (in particolare quelle dedicate al calcolo in virgola mobile, chiamate SSE).

4 La soluzione proposta: GVol

Da quanto esposto nel precedente capitolo, dovrebbe apparire chiaro come la visualizzazione di dati volumetrici sia un argomento decisamente ampio: esistono infatti un gran numero di algoritmi utilizzabili per estrarre, visualizzare e ricostruire informazioni da un dataset; la scelta stessa degli algoritmi da utilizzare per un dato scopo è un argomento complesso, poichè è spesso necessario trovare un giusto compromesso tra velocità, accuratezza e quantità di informazioni fornite all'utente.

4.1 Applicativi per il volume rendering: usabilità e requisiti

La realizzazione di una applicazione per la visualizzazione volumetrica comporta anche un altro problema, non affrontato nel precedente capitolo poichè non legato direttamente alle specifiche degli algoritmi: la necessità di comprendere *quale* sia la forma più adatta di visualizzazione per un certo tipo di dati volumetrici, e *come* l'interfaccia debba presentarsi all'utente finale per una interazione ottimale.

La visualizzazione volumetrica è utilizzabile per lo studio di tipologie di dati estremamente differenti tra loro: dalle misurazioni geologiche alle mappe di densità elettronica, dalle rilevazioni metereologiche ai dataset medicali. La progettazione e lo sviluppo di una applicazione per di questo tipo richiedono la selezione di algoritmi in grado di far emergere dal dataset l'informazione a cui si è interessati; ma spesso il *modo* in cui dovrebbe essere presentata l'informazione non è chiaramente comprensibile, neppure per gli stessi utenti (generalmente non abituati all'interazione con dati volumetrici). Le più classiche tecniche di ingegneria del software (per esempio, la raccolta dei casi d'uso) vengono messe in crisi da questo scenario, poichè esse generalmente prevedono che gli utenti finali siano in grado di descrivere ciò che si attendono da un sistema informatico.

Un esempio illuminante di questo scenario riguarda l'utilizzo delle tecniche di visualizzazione volumetrica in campo medicale: un medico che abbia sempre svolto il suo lavoro esaminando su una lavagna luminosa i lucidi ricavati da TAC o RM è difficilmente in grado di fornire i requisiti di una applicazione che visualizzi gli stessi dati in tre dimensioni. In molti casi il personale medico ha già accesso a sistemi per il volume rendering (per esempio, perchè forniti insieme alla TAC acquistata dall'ospedale), ma non li ritene di ausilio alla propria attività. Questo può essere indice della mancanza

di formazione del personale nel settore delle nuove tecnologie — o, più probabilmente, potrebbe essere indice dell'inadeguatezza del software proposto agli occhi dell'utente.

Questo problema è legato all'attuale scarsa diffusione delle tecniche di volume rendering: esse sono utilizzate principalmente all'interno della comunità scientifica teorica e informatica nel senso più stretto, poichè solo in questo ambito esiste una reale comprensione delle loro potenzialità ed una reale esperienza nel loro utilizzo; all'esterno di questo settore è ancora poco diffuso il ricorso "naturale" a tecniche volumetriche, ovvero la prassi di utilizzare software per la visualizzazione di volumi come supporto alla propria attività quotidiana.

L'usabilità dei sistemi di volume rendering per l'utente finale è insomma un argomento di ricerca ancora aperto. Allo stato attuale, vista la difficoltà nell'individuazione dei prerequisiti, per poter creare un buon software di visualizzazione volumetrica lo sviluppatore dovrebbe avere una certa padronanza nel campo di applicazione del prodotto. Il programmatore di una applicazione per il rendering di dati medicali, per esempio, dovrebbe avere anche buone conoscenze mediche, per poter definire i requisiti e auto-valutare il proprio lavoro.

Oppure, più realisticamente, il programmatore dovrebbe interagire costantemente con il personale medico, non solo durante le fasi di progettazione, ma anche durante le fasi di sviluppo del software; in questo modo, egli potrebbe essere guidato attraverso la scelta degli algoritmi di segmentazione, classificazione e interpolazione più adatti a soddisfare i requisiti qualitativi e prestazionali dell'applicazione finale.

Questo, all'atto pratico, significa che lo sviluppo di applicazioni di volume rendering richiede la creazione rapida di **prototipi** funzionanti, dotati di prestazioni quantomeno indicative di quelle che saranno ottenibili nel programma finale.

L'esperienza maturata durante lo stage formativo al CRS4 (riguardante lo sviluppo di una applicazione per il volume rendering realistico chiamata *VolCastIA*) ha offerto ottime indicazioni sulle tecniche da adottare per progettare una libreria con questi requisiti. È stato infatti possibile osservare quali siano le difficoltà che si incontrano quando occorre modificare una sistema di volume rendering per adattarlo a scopi inizialmente non previsti (evento molto comune nella ricerca), ed è stato possibile individuare quali porzioni di codice debbano essere create puntando alla massima riusabilità (tenendo comunque a mente il bisogno di non degradare eccessivamente le prestazioni).

Queste osservazioni ed esperienze sono stati i punti di partenza nella progettazione di **GVol**, la libreria di manipolazione e visualizzazione di dataset volumetrici oggetto di

questa tesi.

4.2 Obiettivi

Gli obiettivi fondamentali nella progettazione di `GVol` sono stati due:

- flessibilità e riusabilità del codice
- ottimizzazione delle prestazioni

Questi due obiettivi sono notoriamente contrastanti: la creazione di codice flessibile comporta generalmente una perdita (almeno parziale) di prestazioni, mentre il codice ottimizzato è generalmente poco flessibile e manutenibile.

La ricerca di un punto di equilibrio è peraltro inevitabile nella realizzazione di qualsiasi programma di rendering volumetrico: vista la mole di calcoli richiesti, l'argomento prestazioni non può essere trascurato — mentre d'altra parte l'aumento delle prestazioni porta quasi inevitabilmente alla rinuncia a qualche caratteristica del processo di visualizzazione, che potrebbe rivelarsi necessaria per il campo di applicazione.

La necessità aggiuntiva di sviluppare rapidamente dei prototipi dell'applicazione finale richiede che il codice sia flessibile, e che permetta di realizzare versioni intermedie già indicative (almeno a grandi linee) di quelle che saranno le prestazioni finali. Solo in questo modo è possibile valutare la bontà delle scelte compiute durante lo sviluppo, e capire se la direzione intrapresa nell'ottimizzazione del codice sarà in grado di soddisfare i anche i requisiti qualitativi richiesti dall'utente finale.

4.3 Software esistente

Prima di intraprendere la realizzazione di un nuovo programma o libreria, è ovviamente necessario valutare se il software già disponibile soddisfi (almeno in parte) i requisiti di partenza.

Il panorama del software per il volume rendering è per la gran parte formato da soluzioni proprietarie, che non forniscono la possibilità di accedere liberamente al codice sorgente, e di poter adattare il il programma alle proprie necessità.

Una notevole eccezione in questo scenario è il **Visualization Toolkit (Vtk)**, una libreria in C++ per il rendering tridimensionale e la manipolazione di immagini. Essa è

rilasciata sotto una licenza libera, è ampiamente utilizzata in un gran numero di applicazioni scientifice, e comprende anche un buon numero di algoritmi per la visualizzazione volumetrica dalle prestazioni generalmente buone. È anche supportato il rendering misto di primitive volumetriche e geometriche — tanto che la visualizzazione ibrida è un requisito richiesto per ogni classe di volume.

La libreria presenta (ovviamente) anche dei limiti: essa supporta un gran numero di caratteristiche non sempre necessarie nel processo di visualizzazione volumetrica (per esempio il rendering ibrido poligoni-volumi), che finiscono per pesare sulle prestazioni finali; più in generale, le routine per la visualizzazione volumetrica sono integrate in un sistema per la visualizzazione grafica decisamente ampio, che risulta difficile da modificare, ridurre e ottimizzare per applicazioni specifiche.

Non si tratta, ovviamente, di aspetti negativi in senso assoluto: essi rivelano semplicemente come Vtk sia un toolkit grafico di uso generico, non specializzato nella visualizzazione e manipolazione di dati volumetrici.

Un sistema di componenti flessibili e ottimizzati per il volume rendering dovrebbe essere più semplice e facilmente gestibile, e orientato alla sola gestione dei volumi.

Appare quindi sensata l'idea di sviluppare un simile toolkit partendo da zero, cercando ovviamente di riutilizzare quanto più codice possibile da applicazioni libere giù esistenti.

4.4 Paradigma e linguaggio di programmazione

Il modello a oggetti è uno dei paradigmi di analisi e sviluppo del software oggi più utilizzati. Il motivo del suo successo è da ricercarsi nella flessibilità e riusabilità del codice che (se correttamente utilizzato) riesce a garantire. Esso appare dunque ideale per lo sviluppo di `GVol`.

La scelta del linguaggio di programmazione è un argomento più complesso.

La stragrande maggioranza dei linguaggi orientati agli oggetti è di alto livello (è il caso per esempio di Python [9], Ruby [22] o Java™), e cioò li rende non adatti per lo sviluppo di una applicazione di volume rendering con stringenti requisiti prestazionali.

I linguaggi di alto livello, tuttavia, hanno generalmente una caratteristica estremamente utile per la prototipazione di applicativi di qualsiasi tipo: la possibilità di esplorare durante l'esecuzione il contenuto degli oggetti, e di rilevare il tipo ed il valore degli attributi. Questa caratteristica, chiamata **runtime object inspection**, rende estremamente

agevole e rapido lo sviluppo di interfacce utente: invece di creare codice specializzato per visualizzazione dello stato di un certo tipo di oggetto, diventa possibile scrivere interfacce orientate al *tipo di attributo*, riutilizzabili in base alle esigenze.

Nel campo del volume rendering, in particolare, la modifica dei parametri di un dataset e l'interazione con il processo di visualizzazione possono essere ricondotte ad operazioni di modifica di attributi di pochi tipi predefiniti (per esempio numeri interi o a virgola mobile, variabili booleane, punti, vettori, matrici, valori in una lookup table). L'approccio "standard" allo sviluppo delle interfacce utente richiederebbe la creazione di nuove schermate di modifica dei parametri per ogni nuova classe di oggetti da trattare — nonostante gli attributi supportati siano in genere di pochi tipi già noti. Le capacità di object inspection dei linguaggi di alto livello potrebbero essere invece utilizzate per realizzare una interfaccia utente generica, indipendente dalla classe dell'oggetto da gestire, e capace di auto-adattarsi agli attributi rilevati in run-time.

Questo approccio, portato all'estremo, prevede che l'utente possa "vedere" direttamente gli oggetti gestiti dall'applicazione, e modificarne gli attributi di interesse (o per lo meno quelli resi visibili dal programmatore). L'idea è stata proposta (e implementata come software libero) per la realizzazione di programmi di gestione finanziaria; tra i vantaggi di questo paradigma (chiamato **naked objects**) sono citati la rapidità nello sviluppo dell'applicazione, l'adattabilità del codice risultante a requisiti non previsti, e la semplificazione nell'interazione tra programmatore e utente (dato che entrambi possono parlare linguaggi simili nel descrivere il comportamento dell'applicazione) [8].

Tutte queste caratteristiche possono rivelarsi positive anche per la realizzazione di prototipi di applicazione per il volume rendering — e quindi risultano desiderabili per la creazione di `GVol`. Questo approccio, tuttavia, richiederebbe la possibilità di utilizzare la runtime inspection anche in linguaggi di basso livello.

4.4.1 Linguaggi di basso livello orientati agli oggetti

Un linguaggio di basso livello dotato del supporto per il paradigma a oggetti è il **C++**. Esso si rivela spesso una scelta naturale, specie quando si devono sviluppare applicazioni con dei vincoli sulle prestazioni; ma anche il C++ presenta diversi aspetti negativi:

1. alcune sue caratteristiche (come l'overloading degli operatori) rendono difficile prevedere il comportamento del codice, specie quando si utilizzano molte librerie esterne. Un semplice operatore `+`, per esempio, potrebbe nascondere una chia-

mata a funzione non prevista; anche l'allocazione e deallocazione degli oggetti è spesso causa chiamate "invisibili" a costruttori e distruttori: il controllo delle prestazioni e del flusso del codice può diventare quindi estremamente complesso;

2. la gestione interna del sistema a oggetti del C++ è demandata interamente al compilatore, ed e inaccessibile per il programmatore. Questo è ovviamente un aspetto positivo quando il programmatore richiede solamente che il sistema a oggetti funzioni — ma può essere un limite quando si vuole controllare e modificare il comportamento del sistema in situazioni particolari, o quandosi vuole valutare esattamente la complessità e il costo computazionale introdotti da astrazioni come ereditarietà e polimorfismo;

3. il codice assembly generato dai compilatori C++ tende ad essere complesso (a causa del supporto per gli oggetti), e durante la fase di ottimizzazione delle prestazioni risulta difficilmente analizzabile o integrabile con codice assembly scritto manualmente;

4. il sistema a oggetti del C++ non supporta la runtime object inspection.

Un altro linguaggio di basso livello con supporto per gli oggetti è **Objective C**, ideato da Apple Computer™ per lo sviluppo dei propri sistemi operativi [4]: esso supporta la runtime inspection, ma si rivela anche un linguaggio dalla diffusione decisamente ridotta (pur essendo compatibile con il C), che soffre degli stessi aspetti negativi del C++ legati al supporto nativo per gli oggetti (difficoltà nel seguire il flusso del codice, nel creare routine assembly, ecc.).

4.4.2 C a oggetti

Il modello a oggetti è un paradigma di risoluzione dei problemi che può essere utilizzato con qualsiasi linguaggio di programmazione, anche privo del supporto nativo per ereditarietà, polimorfismo e incapsulamento. In questo caso, il prezzo da pagare consiste in una maggiore autodisciplina del programmatore.

Esistono, per esempio, numerosi esempi di software sviluppato in C con un approccio a oggetti. Essi si basano su un'idea di partenza generalmente simile: le strutture di base vengono incluse come primo membro nelle strutture derivate. Un puntatore ad una struttura derivata può quindi essere sottoposto a cast espliciti, che rendono possibile

l'accesso ad una struttura base, e simulano una semplice forma di ereditarietà. Se anche le funzioni per la manipolazione dei membri vengono memorizzate all'interno delle strutture stesse (sotto forma di puntatori), diventa possibile anche una semplice forma di polimorfismo. Il tipo `struct` del C viene insomma trasformato in una rudimentale forma di classi.

Questo accorgimento è utilizzato (ed esteso) da **GObject** [17], un sistema a oggetti in C facente parte delle librerie **GLib** [27]. La grande mole di servizi offerti da GObject e GLib si è rivelata estremamente utile per lo sviluppo di `GVol`:

Ereditarietà Gobject supporta l'ereditarietà singola, con controllo in run-time della correttezza dei cast.

Polimorfismo Ogni classe di oggetti nel sistema GObject è rappresentata da due strutture: una per l'istanza (che contiene gli attributi), e una per i metadati riguardanti la classe (che contiene i puntatori ai metodi sottoponibili a polimorfismo).

Interfacce Il sistema GObject supporta le interfacce, che possono essere addirittura modificate in run-time.

Iteratori Grazie ad una semplice patch disponibile su Internet [1], è possibile estendere GObject e GLib con il supporto ai tipi `GIterator` e `GValueIterator`, che implementano degli iteratori utilizzabili per l'accesso sequenziale agli elementi in un qualsiasi container di valori.

Reference counting L'allocazione/deallocazione della memoria delle istanze derivate dal tipo `GObject` è automatica, e basata sul conteggio dei riferimenti: essi sono incrementati con una chiamata al metodo `GObject::ref`(), e decrementati con una chiamata a `GObject::unref`(). Quando il reference count di un oggetto giunge a 0, vengono automaticamente invocati i distruttori della classe.

Weak references È possibile creare dei "riferimenti deboli", ovvero dei puntatori a oggetti derivati da `GObject` che vengono automaticamente impostati a `NULL` quando l'istanza viene deallocata.

Runtime inspection Il sistema `GParam` fornito da GLib permette di specificare quali attributi di un oggetto possano essere sottoposti a runtime inspection, e secondo quali termini (solo lettura, o lettura e scrittura). Il sistema permette anche di porre

dei vincoli dipendenti dal tipo di attributo (per esempio, è possibile specificare l'intervallo di valori accettabili per un attributo `gfloat`). È inoltre possibile una completa runtime inspection degli attributi gestiti da `GParam`, e addirittura una loro aggiunta o rimozione in run-time.

Segnali Gli oggetti derivanti dalla classe `GObject` possono emettere dei segnali, ovvero dei messaggi di vario tipo per comunicare variazioni nel proprio stato; essi possono essere catturati dal resto dell'applicazione mediante l'allocazione di un *signal handler* (una funzione che verrà richiamata in caso di emissione del segnale di interesse).

Valori generici GLib comprende un sistema generico per il trasporto di tipi di base e oggetti, chiamato `GValue`. Esso permette, per esempio, di passare qualunque tipo di dato come argomento di una funzione, senza dover gestire direttamente i dettagli riguardanti l'eventuale copia o deallocazione della memoria associata.

Funzioni come tipo di dato (closures) Il tipo `GClosure` rappresenta una funzione generica, della quale è possibile stabilire in runtime il numero e il tipo di parametri, e il tipo di valore di ritorno.

L'utilizzo di GObject presenta tuttavia due aspetti negativi:

Prolissità del codice Specie all'inizio dello sviluppo, il tempo necessario a scrivere del codice a oggetti che segua il sistema GObject è generalmente maggiore rispetto a quello necessario a produrre lo stesso codice in C++;

Autodisciplina GObject è interamente basato sul C, che non ha supporto nativo per gli oggetti. La mancanza di un sistema di controllo a livello compilatore (come quello presente, per esempio, in C++) costringe il programmatore ad una rigorosa autodisciplina e attenzione nella stesura del codice, poichè la maggior parte dei vincoli nella programmazione a oggetti sono auto-imposti.

Entrambi questi aspetti negativi risultano tuttavia attenuabili con l'esperienza nell'utilizzo di GObject. Questa libreria è stata quindi scelta per l'implementazione di `GVol`.

4.5 Struttura del prototipo di software implementato

L'utilizzo di GObject per lo sviluppo di `GVol` ha permesso la creazione di un sistema a oggetti minimale ed estremamente flessibile, riducendo al minimo il numero di interfacce o i metodi richiesti da ogni classe del sistema. L'esperienza nello sviluppo di *VolCastIA* (si veda la sez. 4.1) ha infatti evidenziato come interfacce troppo rigide possano rendere estremamente complesso l'adattamento del software a situazioni non previste.

Un esempio indicativo è il sistema di rendering: l'unico interfaccia richiesta da GVol è il metodo `GVolCamera::render()`(). Il resto dei dettagli (per esempio come associare uno o più volumi ad una classe `GVolCamera`(), o come impostare i parametri di rendering) non è specificato, e può essere basato sul sistema di runtime object inspection offerto da GObject (fig. 4.5).

Tutti gli oggetti del sistema `GVol`derivano dalla classe `GVolObject`, che implementa:

- alcuni attributi di uso comume (per esempio, il nome dell'oggetto);
- il segnale `"progress"`, che un oggetto può emettere per indicare il suo stato di attività: una classe di volume, per esempio, può ricorrere ad esso per indicare la percentuale di rendering completata. `progress` prevede due argomenti: una stringa di caratteri `msg` (per la descrizione dello stato dell'oggetto) e un numero intero `percentage`, che indica lo stato dell'operazione. In particolare, `percentage` può essere maggiore di 0 (e compreso nell'intervallo $[0, 100]$) per indicare la percentuale di completamento dell'operazione, o minore di 0 per indicare il semplice avanzamento dell'elaborazione (utilizzabile, per esempio, qualora non sia possibile stabilire la percentuale di lavoro completata).

La classe astratta `GVolContainer` rappresenta un contenitore astratto per dati generici (grazie all'uso di `GValue` per i metodi di accesso). È possibile estrarre o inserire valori in sequenza, con l'utilizzo di iteratori di tipo `GIterator` e `GValueIterator`. La classe base viene specializzata in classi di container monodimensionali e tridimensionali (`GVolContainer1D` e `GVolContainer3D`), a loro volta specializzati in base al tipo di dati effettivamente contenuto (per esempio, `GVolContainer1Dgfloat`) (si veda la fig. 4.5).

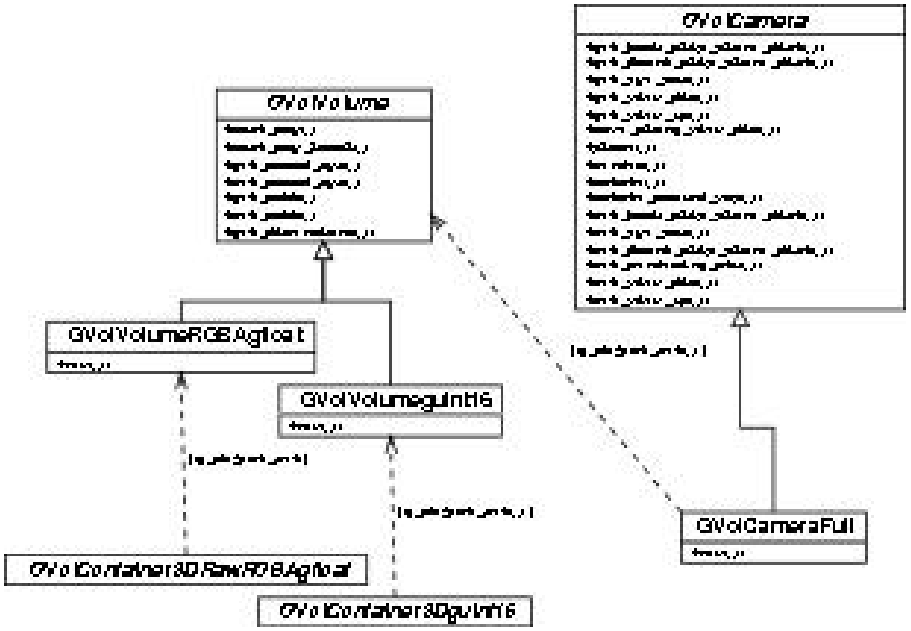


Figura 20: *Schema di uno scenario di rendering (non ottimizzato) utilizzabile con* `GVol`*: La classe* `GVolCameraFull` *utilizza il metodo* `::cast_ray()` *della classe* `GVolVolume`*, e può in questo modo gestire diversi tipi di volumi e di dataset. Le frecce tratteggiate indicano che la modalità di assegnamento delle componenti indicate non è vincolata dalla struttura a oggetti di* `GVol`*: non esistono metodi obbligatori per associare, per esempio, un certo dataset alla classe volume. L'operazione può avvenire sfruttando la run-time inspection: la classe* `GVolVolumeguint16` *segnala di supportare un attributo* `"dataset"` *di tipo* `GVolContainer3Dguint16`*, che puo essere impostato al valore desiderato mediante utilizzando la funzione* `g_object_set()`*.*

I valori dei voxel in un dataset volumetrico vengono memorizzati all'interno di container derivati da `GVolContainer3D`. La gestione dei volumi avviene attraverso la classe `GVolVolume`, che di fatto trasforma uno o più dataset in entità geometriche, dotate di un punto di applicazione nello spazio e di un proprio sistema di riferimento. I metodi di `GVolVolume` permettono di traslare, ruotare e scalare i voxel, e implementano le operazioni di rendering.

Il rendering volumetrico viene gestito attraverso la classe `GVolCamera`, utilizzata per:

- impostare i parametri di visualizzazione (posizione del punto di vista, direzione,

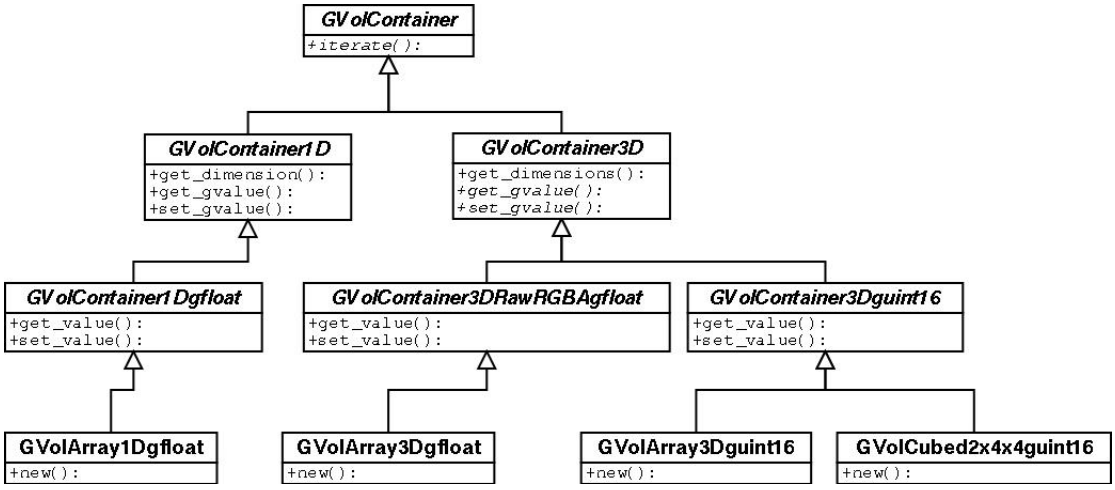


Figura 21: *Gerarchia (parziale) della classe* `GVolContainer`.

posizione di back e front clipping plane);

- stabilire il contesto di rendering (finestra a schermo, pixmap offscreen, file su disco...);
- effettuare l'operazione di rendering, con il metodo `::render`().

Un'altra classe fondamentale nella struttura di `GVol` è `GVolFunc`, che rappresenta una funzione generica. Le classi derivate sono specializzate in base a parametri e valori di ritorno: per esempio, `GVolFunc_gfloat__gint32` è una classe astratta per una funzione con un parametro intero a 32bit, che restituisce un valore float a singola precisione. Quando necessario, è possibile determinare il tipo di valore di ritorno e il tipo di parametri di una funzione `GVolFunc` esaminando la `GClosure` restituita dal metodo `GVolFunc::get_gclosure`().

Nelle prossime sezioni verranno brevemente ripercorsi gli argomenti trattati nel capitolo 3, illustrando in che modo l'attuale implementazione di `GVol`affronti ognuno di essi.

4.6 Dal voxel al pixel

Nel caso più generale e meno ottimizzato, il rendering di un dataset può essere realizzato utilizzando il metodo `::render` di una classe derivata da `GVolCamera`, che

a sua volta può generare l'immagine da visualizzare con una serie di chiamate al metodo `GVolVolume::cast_ray`(). Questo è stato l'approccio seguito fino a questo momento, e messo in pratica con la classe **GVolCameraFull**.

`GVol`non è tuttavia basato su un particolare modello di proiezione dei voxel a schermo, anche se ogni classe derivata da `GVolVolume` deve implementare il metodo `::cast_ray`(). Tale metodo, infatti, è richiesto principalmente per il sondaggio (*probing*) all'interno del dataset — per esempio, per la verifica del valore di colore restituito da un raggio visuale avente una certa direzione. Questo tipo di operazioni non richiedono prestazioni estremamente ottimizzate, poichè non vengono generalmente eseguite in blocco, e la mole di calcoli richiesta per il processamento di un singolo raggio è ridotta. Il probing del volume è utile per la comprensione e la verifica di correttezza del processo di visualizzazione, e per questo motivo si è deciso di richiedere l'implementazione del metodo `::cast_ray`() per ogni classe di volumi gestita da `GVol`.

Al di là di questo requisito, comunque, è sempre possibile creare delle classi derivate da `GVolCamera` e `GVolVolume` particolarmente ottimizzate, che lavorino utilizzando altri sistemi di rendering (per esempio shear-warping, o texture mapping volume rendering) mediante altri metodi dedicati; dall'esterno di queste classi il processo di rendering risulterà comunque accessibile solamente attraverso il metodo `GVolCamera::render`.

4.7 Informazioni contenute nei voxel

`GVol`supporta per il momento il rendering di due tipi di dataset:

- dataset formati da valori unsigned short a 16 bit (`guint16`);
- dataset formati da valori RGBA in virgola mobile, a singola precisione (tipo `GVolRawRGBAgfloat`);

Ciò ha richiesto l'implementazione di due classi derivate da **GVolVolume** (chiamate `GVolVolumeguint16` e `GVolVolumeRawRGBAgfloat`), e di differenti tipi di container con diverse ottimizzazioni (per i dettagli, si veda la sezione 4.14). La classe `GVolCameraFull` descritta precedentemente non ha tuttavia richiesto modifiche, poichè l'utilizzo di metodi di rendering generici (`GVolVolume::cast_ray`()) la rende adatta alla gestione di entrambi i tipi di dataset.

4.8 Calcolo (mapping) dell'opacità dei voxel

La classificazione dei voxel gestiti dalla classe `GVolVolumeguint16` viene attualmente eseguita utilizzando una funzione di classe `GVolFunc_gfloat__gint32`, in grado di restituire un livello di opacità dipendente dal valore del voxel (si veda la definzione di $f(\overline{v})$ nella sezione 3.1). In particolare viene usata una sottoclasse, `GVolLUT_gfloat__gint32`, che implementa tale funzione mediante una ricerca in una lookup table basata su un container di tipo `GVolContainer1Dgfloat`.

La classificazione dipendente dal gradiente è attualmente demandata alla classe `GVolVolume`, che si limita a moltiplicare l'opacità restituita dalla funzione per la norma del gradiente rilevato alla posizione del voxel.

4.9 Calcolo (mapping) del colore dei voxel

La classificazione dei colori dei voxel non è stata ancora completamente implementata nel sistema di rendering. È disponibile un tipo di volume che opera direttamente su valori RGBA (`GVolVolumeRGBAgfloat`), e che utilizza direttamente il sistema di gestione dei colori di `GVol` senza la necessità di una loro classificazione.

Quando implementata, la classificazione dei colori nel caso di dataset formati da valori numerici potrà seguire un approccio simile a quello utilizzato per la classificazione delle opacità. Una funzione di tipo `GVolFunc_RawRGBAgfloat__gint32`, per esempio, potrà restituire un colore di tipo RGBA a singola precisione dipendente dal valore del voxel. Questo tipo di funzione può quindi prevedere diversi algoritmi di assegnamento, invisibili al motore di volume rendering.

4.10 Ricostruzione dei valori dei voxel

La classe `GVolInterp` rappresenta un interpolatore generico su una dimensione. È attualmente implementate una classe di interpolatori chiamata `GVolInterpGSL`, che utilizza la libreria **GSL (GNU Scientific Library)** ed è in grado di effettuare interpolazioni utilizzando tutte le routine fornite dalla libreria (attualmente comprendenti 6 varianti, tra cui spline lineari e cubiche).

L'interpolazione lineare fornita da GSL è stata per il momento isolata in una classe separata (`GVolInterpGSLLinear`), ma essa verrà rimossa nelle successive versioni di `GVol`.

4.11 Ricostruzione del gradiente

La ricostruzione dei gradienti in GVol è realizzata con la classe `GVolConvFilter3D`. La sottoclasse `GVolConvMatrix3Dguint16`, in particolare, applica delle matrici di convoluzione a valori all'interno di un container del tipo `GVolContainer3Dguint16`. Sono attualmente supportate le matrici per filtri gradiente del tipo intermediate difference, central difference, e per l'operatore di Sobel (definiti nella sezione 3.7.

4.12 Luci e ombreggiatura (shading)

L'unico sistema (improprio) di illuminazione attualmente disponibile da `GVol`è il depth shading, descritto nella sezione 3.8.

4.13 Spazi di colore

`GVol` è stato disegnato per supportare differenti spazi di colore. La classe astratta `GVolColor` rappresenta un generico colore, che può essere interpolato con altri colori della medesima classe utilizzando il metodo `::interpolate`(). Tra le specializzazioni della classe vi sono `GVolColorColorRGBguchar` (RGB, precisione 8 bit per canale), `GVolColorRGBAguchar` (RGB + canale A per l'opacità, precisione 8 bit per canale), `GVolColorRGBAgfloat` (RGB + canale A per l'opacità, singola precisione in virgola mobile). Ogni classe di colore deve inoltre supportare la conversione nei formati RGB e RGBA con precisione a 8 bit per canale (formati scelti per la loro utilità e ampio utilizzo nelle operazioni di rendering).

Le classi derivanti da `GVolColor` non sono altro che delle astrazioni cha nascondono il reale sistema di gestione dei colori di `GVol`: `GVolColorRGBAgfloat`, per esempio, offre un'interfaccia a oggetti per l'utilizzo delle funzioni che operano sul tipo `GVolRawRGBAgfloat` (una semplice struttura contenente quattro valori `float`). Le routine di basso livello sono comunque accessibili al programmatore, e in questo modo è possibile evitare l'utilizzo del sistema a oggetti quando non richiesto (per esempio, per salvaguardare le prestazioni).

Le strutture di basso livello per la memorizzazione del colore hanno formati "comuni", che possono essere utilizzati in modo flessibile. La struttura `GVolRawRGBguchar`, per esempio, può essere sottoposta a cast e utilizzata come un array di 3 valori `char`; le

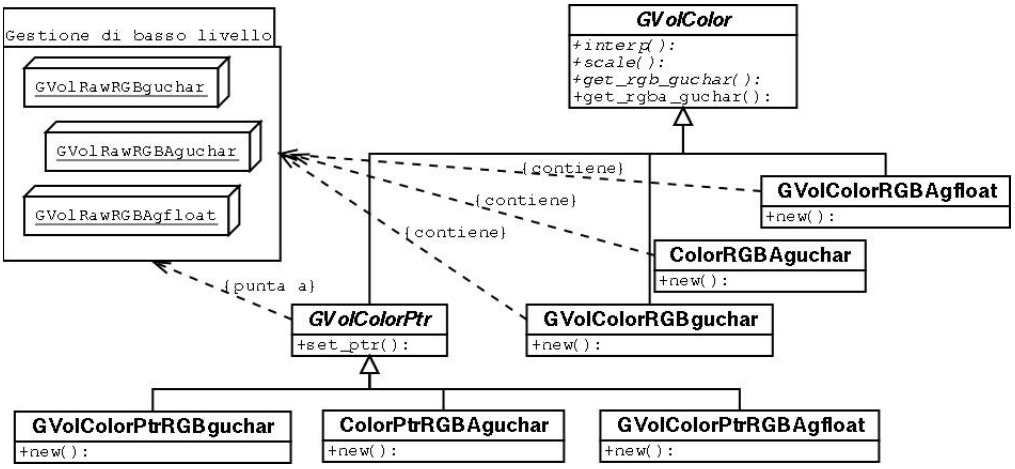


Figura 22: *Struttura delle classi per il supporto del colore in* `GVol`.

routine di gestione del colore possono quindi essere utilizzate, per esempio, per gestire buffer di valori RGB da condividere con qualsiasi toolkit grafico.

`GVol` comprende infine una classe astratta chiamata `GVolColorPtr`, che richiede esplicitamente che il valore del colore da gestire venga memorizzato in una struttura di basso livello esterna. Una volta impostato il puntatore a tale struttura (con il metodo `::set_ptr`), è possibile modificare il colore memorizzato in una specifica locazione di memoria attraverso un'interfaccia a oggetti. Ciò permette di scrivere del codice generico che si occupi della gestione colore durante il volume rendering (con le classiche operazioni di attenuazione, interpolazione...); tale codice può essere quindi "pilotato" verso le giuste locazioni di memoria in base al tipo di colore effettivamente utilizzato (fig. 4.13)

Si è inoltre cercato di astrarre il più possibile la gestione del colore isolando l'accesso in scrittura ai buffer di rendering mediante le classi `GVolRenderingCtx` e `GVolRenderingRgn` (fig. 4.13).

4.14 Analisi delle prestazioni

Lo sviluppo di `GVol` non è ancora giunto alla fase di ottimizzazione delle prestazioni. Per adesso tutto il codice di volume rendering segue rigidamente le interfacce previste dal modello a oggetti, in modo da garantire un funzionamento corretto del sistema, ed avere un modello di riferimento con cui confrontare gli sviluppi futuri.

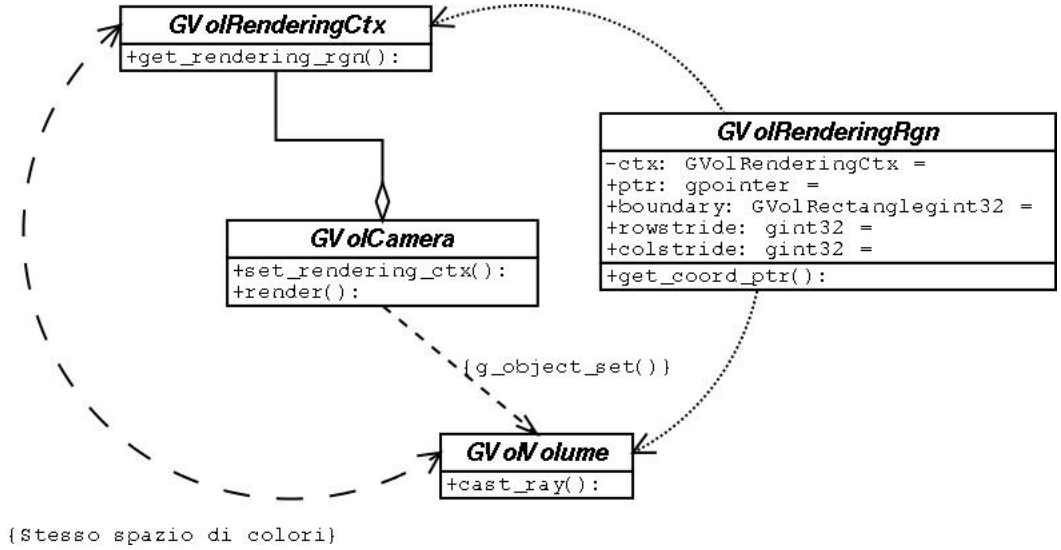


Figura 23: *Astrazione dal contesto di rendering di* `GVol`*. La classe* `GVolCamera` *utilizza un rendering context in modo opaco, senza conoscere i particolari riguardanti lo spazio di colori utilizzato o la disposizione in memoria dei pixel; questi dettagli sono a carico dalla struttura di gestione della regione di rendering (*`GVolRenderingRgn`*), che verrà usata per pilotare la visualizzazione all'interno della classe* `GVolVolume`*.*

Un'eccezione è rappresentata dallo sviluppo di un container tridimensionale (classe `GVolCubed2x4x4guint16` ottimizzato per la coerenza nell'accesso alla memoria durante il volume rendering. I valori contenuti non sono infatti disposti nella RAM in modo sequenziale (come avviene nei semplici array), ma sono disposti in blocchi di 2x4x4 unsigned a 16 bit, ottenuti applicando una serie di trasformazioni alle coordinate per la lettura/scrittura dei valori. La forma dei blocchi è dovuta alla necessità di mantenere la loro dimensione nel limite dei 64 byte, corrispondente alla cacheline dei processori Intel™ Itanium™ e AMD™ Duron™ usati per il test.

Utilizzando un container `GVolCubed2x4x4guint16` per il rendering si è osservato un aumento delle prestazioni prossimo al 20/30%. Tale incremento è limitato dall'attuale sistema di rendering, basato su un semplice raycasting con una chiamata a metodo per ogni raggio: questo significa che la coerenza temporale nell'accesso ai dati è decisamente ridotta. Maggiori indicazioni sull'efficacia dello schema di memorizzazione potranno essere ottenute da ulteriori analisi, in fasi più avanzate dello sviluppo.

4.15 Una applicazione di test: Giave

Il test della libreria `GVol` ha richiesto la creazione di una applicazione di prova, che fosse in grado di mettere in evidenza le caratteristiche della libreria. Tale applicazione è stata chiamata **Giave** — un nome che può essere, a scelta, quello di un paese in provincia di Sassari, o l'acronimo per "Giave Is A Volume Environment".

Gli scopi principali di `Giave` sono due:

- la creazione di un prototipo di applicazione in grado di utilizzare la libreria, evidenziando limiti e problemi nella progettazione dell'interfaccia per il programmatore;
- lo sviluppo di un'interfaccia grafica che semplifichi la gestione dei parametri di rendering, seguendo l'approccio naked objects citato in precedenza.

I risultati possono essere osservati nella figura 4.15:

- sulla sinistra si può osservare il sistema di navigazione degli oggetti. Si tratta di un widget grafico che rappresenta gli oggetti accessibili all'utente, e visualizza i loro attributi utilizzando la runtime inspection. Le azioni disponibili variano in base al tipo di attributo contenuto nell'oggetto:
  - gli attributi di tipo numerico o di tipo stringa possono essere inseriti direttamente cliccando sull'attuale valore;
  - gli attributi di tipo booleano (non presenti nell'immagine) vengono visualizzati con una checkbox, che può essere attivata o disattivata direttamente sull'interfaccia;
  - un doppio click sugli attributi di tipo `GVolFunc_gfloat__gint32` contenenti un valore di tipo `GVolLUT_gfloat__gint32` visualizza la finestra di impostazione dei valori della lookup table visualizzata in alto nell'immagine;
  - gli altri tipi di attributi sono (per il momento) semplicemente visualizzati.
- sulla destra si può osservare la finestra di rendering, che in questo momento gestisce un volume di tipo `GVolVolumeguint16` di dimensione 256x256x133, con classificazione lineare delle opacità e depth cueing.

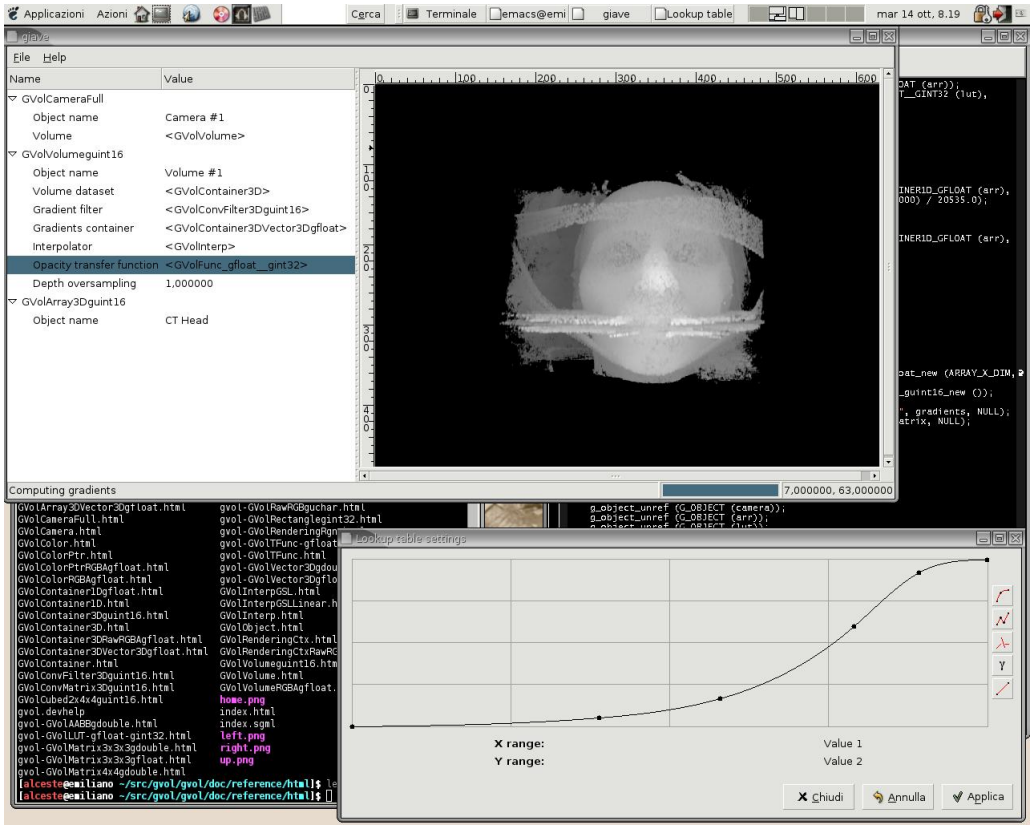


Figura 24: *Schermata tratta da una sessione di esecuzione di* `GVol`.

- in basso si trova la barra di stato, che comunica all'utente le operazioni in corso. Il suo contenuto è stabilito direttamente dai segnali `progress` emessi dagli oggetti di tipo `GVolObject` gestiti dall'applicazione: il testo comunicato dal segnale viene visualizzato in basso a sinistra, mentre la percentuale di completamento viene utilizzata per aggiornare la barra di avanzamento in basso a destra.

Nella figura 4.15 si può osservare un'altra schermata, in cui si utilizza un container di tipo `GVolCubed2x4x4guint16` per la memorizzazione del dataset. Come si può notare, il sistema di navigazione degli oggetti visualizza correttamente il nuovo tipo di oggetto in uso.

Il risultato è ancora sperimentale, ma permette di verificare l'efficacia del sistema `GVol` per la prototipazione di applicazioni.

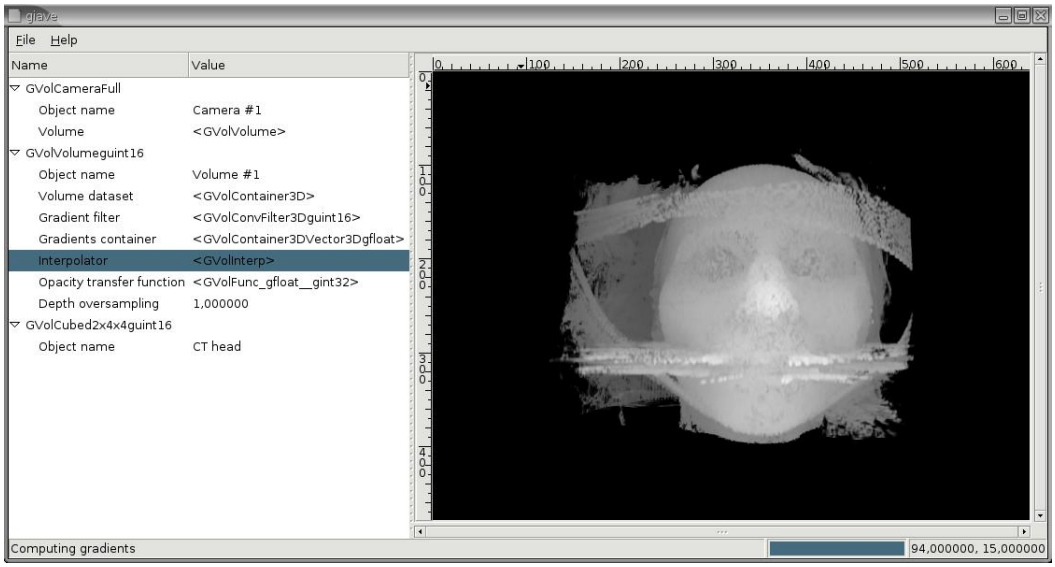


Figura 25: *Schermata tratta da una sessione di esecuzione di* `GVol`*, in cui si utilizza un container di tipo* `GVolCubed2x4x4guint16` *per la memorizzazione del dataset.*

5 Conclusioni e sviluppi futuri

Allo stato attuale, `GVol` comprende un rilevante numero di classi e funzioni già testate e affidabili, utilizzabili quindi come base per la costruzione di applicazioni specializzate di volume rendering. L'appendice A contiene la guida di riferimento alla parte di software proposta nel presente lavoro.

Le prove finora effettuate indicano che la libreria `GVol` è adeguata ai requisiti di flessibilità illustrati durante la progettazione (sezione 4.1). L'applicazione di test implementata (`Giave` , sez. 4.15) illustra una delle modalità di utilizzo della libreria. Come previsto, il suo sviluppo ha richiesto un investimento di risorse limitato alla fase iniziale: dopo la creazione del sistema di navigazione degli oggetti, infatti, è stato possibile controllare alcune delle caratteristiche principali del processo di volume rendering direttamente dall'interfaccia, senza creare altro codice dedicato. Questo importante risultato ha consentito, come da premesse, di concentrare l'attività di sviluppo successiva sulle routine di volume rendering. Inoltre questo prototipo di applicazione mostra come associare al sistema `GVol` una tipica modalità di interazione grafica per l'analisi di dati volumetrici.

L'analisi delle prestazioni del sistema è ancora in fase embrionale. Le prove fin qui eseguite (riguardanti il "container" ottimizzato di classe `GVolCubed2x4x4guint16`) confermano la possibilità di utilizzare il sistema `GVol` per la valutazione delle strategie di ottimizzazione.

I possibili sviluppi futuri riguardano il "profiling" e l'ottimizzazione spinta delle funzioni più utilizzate, la creazione di una suite per il test delle prestazioni, e un confronto tra le caratteristiche offerte da `Giave` e quelle offerte da programmi di volume rendering già esistenti (come il già citato *VolCastIA*).

Un'altra attività futura riguarderà l'utilizzo del sistema `GVol` per lo sviluppo di software specifico per determinati campi applicativi. Per esempio, in ambito medico, il sistema verrà utilizzato per l'analisi di dati acquisiti da RM, e per lo studio delle modalità di interazione utente con personale medico. In particolare, `GVol` verrà utilizzato nello studio di malattie neurodegenerative quali epilessia refrattaria, morbo di Parkinson e sclerosi multipla. Questo lavoro verrà svolto in collaborazione con i ricercatori del CRS4 e con i medici dell'Università degli Studi di Cagliari.

Elenco delle figure

Riferimenti bibliografici

[1] Joel Becker. Iterators for glib and gobject. *GNOME Bugzilla*, 2002. `http://bugzilla.gnome.org/show_bug.cgi?id=83729`.

[2] Phong Bui-Tuing. Illumination for computer-generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[3] W. Lorensen | H. Cline. Marching cubes: A high resolution 3-d surface construction algorithm. *Computer Graphics*, 21:163–169, 1987.

[4] Apple Computer™. The objective-c programming language. *Apple Computer™developer website*, 2002. `http://developer.apple.com/documentation/Cocoa/Conceptual/ObjectiveC/`.

[5] Klaus Engel | Martin Kraus | Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. *University of Stuttgart website*, 2001.

[6] Andrew S. Glassner. *Principles of Digital Image Synthesis*. Morgan Kaufman, 1995. ISBN 1-55860-276-3.

[7] Thomas Theußl | Helwig Hauser | Eduard Gr oller. Mastering windows: Improving reconstruction. *Institute of Computer Graphics — Vienna University of Technology*, 1999.

[8] Naked Objects Group. The naked objects approach. *Naked objects website*, 2003. `http://www.nakedobjects.org/no-approach.html`.

[9] Jr. (editor) Guido Van Rossum | Fred L. Drake. Python tutorial. *Python website*, 2003. `http://www.python.org/doc/current/tut/tut.html`.

[10] R. Drebin | L. Carpenter | P. Hanrahan. Volume rendering. *Computer Graphics (ACM Siggraph Proceedings)*, 22(4):65–74, 1988.

[11] U. Kanus, G. Wetekam, J. Hirche, and M. Meißner. VIZARDII: An FPGA-based Interactive Volume Rendering System. In *Field-Programmable Logic and Applications*, Proc. of the 12th International Conference on Field-Programmable Logic, pages 1114–1117, September 2002.

[12] A. Kaufman. Voxel-based architectures for three-dimensional graphics. In *Proceedings IFIP'86, 10th World Computer Congress*, pages 361–366, Dublin, Ireland, September 1986.

[13] A. Kaufman. Volume visualization. In *Volume Visualization*. IEEE Computer Society Press Tutorial, 1990.

[14] Arie E. Kaufman. Volume visualization: Principles and advances. *Center for Visual Computing and Computer Science Department, State University of New York at Stony Brook*, 1988. `http://www.merl.com/people/pfister/courses/Bonn2000/Papers/KaufmanVolumeVisualization.pdf`.

[15] G. Knittel and W. Straßer. Vizard - visualization accelerator for realtime display. In *Proc. of SIGGRAPH/Eurographics workshop on graphics hardware 1997*, pages 139–146, August 1997.

[16] Gunter Knittel. The ultravis system. *Hewlett-Packard Laboratories, Visual Computing Department*, 2000.

[17] Mathieu Lacage. The glib object system v0.8.0. *Mathieu Lacage website*, 2003. `http://www.le-hacker.org/papers/gobject/`.

[18] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, pages 29–37, May 1988.

[19] P. Lacroute | M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. *Proceedings of SIGGRAPH '94*, pages 451–458, July 1994.

[20] Mark J. Bentum | Berthold B.A. Lichtenbelt | Thomas Malzbender. Frequency analysis of gradients estimators in volume rendering. *HP Labs 1995 Technical Reports*, 1995. `http://www.hpl.hp.com/techreports/95/HPL-95-81.html`.

[21] Stephen R. Marschner and Richard J. Lobb. An evaluation of reconstruction filters for volume rendering. In R. Daniel Bergeron and Arie E. Kaufman, editors, *Proceedings of Visualization '94*, pages 100–107, 1994.

[22] Yukihiro Matsumoto. What's ruby. *Ruby website*, 2002. `http://www.ruby-lang.org/en/20020101.html`.

[23] M. Meißner, U. Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Straßer, M. Doggett, P. Forthmann, and R. Proksa. VIZARDII: A reconfigurable interactive volume rendering system. In *Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 137–146, September 2002.

[24] Berthold Lichtenbelt | Randy Crane | Shaz Naqvi. *Introduction to Volume Rendering*. Hewlett Packard - Prentice Hall PTR, 1998. ISBN 0-13-861683-3.

[25] Hanspeter Pfister | Jan Hardenbergh | Jim Knittel | Hugh Lauer | Larry Seiler. The volumepro real-time ray-casting system. In Alyn Rockwood, editor, *Siggraph 1999, Computer Graphics Proceedings,*, pages 251–260, Los Angeles, 1999. Addison Wesley Longman.

[26] C.E. Shannon. Communication in the process of noise. In *Proceedings the IRE*, volume 37, pages 10–21, 1949.

[27] Glib/GTK+ team. Glib reference manual. *GNOME developer website*, 2003. `http://developer.gnome.org/doc/API/2.0/glib/index.html`.

[28] H. E. Cline | W. E. Lorensen | S. Ludke | C. R. Crawford | B. C. Teeter. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15:320–327, 1988.

[29] Thomas Theußl. On windowing for gradient estimation in volume visualization. In *Contributions in CESCG99*, 1999.

[30] Thomas Theußl. *Sampling and Reconstruction in Volume Visualization*. Tesi di laurea - Technischen Universität Wien, Technisch-Naturwissenschaftliche Fakultät, 1999. `http://www.cg.tuwien.ac.at/~theussl/DA/`.

[31] E. H. W. Meijering | K. J. Zuiderveld | M. A. Viergever. Image reconstruction by convolution with symmetrical piecewise nth-order polynomial kernels. *IEEE Transactions on Image Processing*, 8(2):192–2001, 1999. `http://imagescience.bigr.nl/meijering/publications/abstracts/tip1999.html`.

[32] E. H. W. Meijering | W. J. Niessen | J. P. W. Pluim | M. A. Viergever. Quantitative comparison of sinc-approximating kernels for medical image interpolation. *Medical Image Computing and Computer-Assisted Intervention (MICCAI 1999) — Lecture Notes in Computer Science*, 1679:210–217, 1999. `http://imagescience.bigr.nl/meijering/publications/abstracts/miccai1999.html`.

[33] E. H. W. Meijering | W. J. Niessen | M. A. Viergever. The sinc-approximating kernels of classical polynomial interpolation. *IEEE International Conference on Image Processing - ICIP '99 (Sixth International Conference, Kobe, Japan, October 24-28, 1999)*, III:652–656, 1999. `http://imagescience.bigr.nl/meijering/publications/abstracts/icip1999l.html`.

[34] E. H. W. Meijering | W. J. Niessen | M. A. Viergever. The sinc-approximating kernels of classical polynomial interpolation. *IEEE International Conference on Image Processing - ICIP '99 (Sixth International Conference, Kobe, Japan, October 24-28, 1999)*, III:652–656, 1999. `http://imagescience.bigr.nl/meijering/publications/abstracts/icip1999l.html`.

A GVol API Reference

Le pagine seguenti contengono la guida al programmatore per la parti già sviluppate della libreria `GVol`.