# Studio delle tecnologie grafiche 3D per applicazioni VR in ambiente PC/Linux (I)

Leonardo Massei

Rapporto Interno
CNUCE-B4-2002-010

## 1. Introduzione

Questo rapporto è il risultato di uno studio sulle possibili tecnologie grafiche 3D, sia hardware che software, per lo sviluppo di applicazioni di realtà virtuale in ambiente a basso costo, in particolare in ambiente PC/Linux. La struttura del rapporto prevede, oltre a questa breve introduzione, altri quattro paragrafi. Il paragrafo 2 dà le ragioni della necessità di VR a basso costo e poi presenta prima una possibile soluzione hardware e poi delle soluzioni software che s'integrano con quella hardware. I paragrafi 3 e 4 analizzano il primo le caratteristiche di un componente dell'hardware (la scheda grafica), mentre il secondo alcune caratteristiche avanzate di una soluzione software presentata nel paragrafo 2. Infine l'ultimo paragrafo, il 5, trae le conclusioni sullo studio svolto.

## 2. VR a basso costo

Gli ambienti per lo sviluppo di applicazioni di realtà virtuale, sono generalmente molto costosi. Essi prevedono un costo iniziale non indifferente più dei costi successivi per la manutenzione, ad opera di personale molto qualificato. In genere tali sistemi si trovano in centri di ricerca o in ambienti specifici. Un esempio tutto italiano è il "Virtual Theatre" allestito presso il CINECA di Bologna [1]; per dare un'idea delle caratteristiche hardware di ambienti del genere, le componenti del Virtual Theatre sono:

- Un sistema SGI Onyx2 configurato con 8 processori, 4 Gbyte di Ram, 3 pipeline grafiche.
- Un sistema di videoproiezione ad alta risoluzione costituito da 3 videoproiettori di tipo BARCO-GRAPHICS 1209s e un tavolo immersivo BARON, un sottosistema di controllo automatico della convergenza, un sottosistema di sovrapposizione dei bordi delle immagini e un sottosistema di commutazione e controllo.
- Uno schermo di proiezione BARCO di tipo cilindrico concepito appositamente per essere utilizzato in ambienti di realtà virtuale.
- Un sistema audio che risponde a caratteristiche di qualità professionale di tipo Dolby Pro Logic Sorround per affiancare all'immersione visiva, quella acustica.

Un altro esempio è il "CyberStage" del GMD di Bonn in Germania [2]. Il CyberStage è in poche parole una stanza, dotata di un sistema di visualizzazione stereo che crea l'illusione di immersione in un ambiente virtuale. L'intera struttura è costituita da tre sistemi di proiezione, da una workstation IR Onyx, sensori per rilevare la posizione dell'utente, quattro schermi e da un sistema sonoro di tipo sorround con 8 canali. In più la stanza ha un pavimento acustico che permette di generare il senso delle vibrazioni.

La tecnologia della realtà virtuale trova applicazione in diverse aree come ad esempio nel campo dell'ingegneria, principalmente per la rappresentazione grafica di fenomeni non visuali; ad esempio in simulazioni tridimensionali della dinamica dei fluidi (utilizzate dall'industria dell'automobile), nella simulazione del flusso d'aria in un motore di un aeroplano, nella modellazione molecolare per lo studio e/o la manipolazione di molecole complesse come le proteine. Un'altra interessante area di applicazione è quella dei beni culturali; ad esempio un ente potrebbe allestire un ambiente di realtà virtuale rappresentante un determinato patrimonio artistico, visitabile da più persone guidate da una guida virtuale. E' soprattutto in un contesto del genere che può essere inquadrato l'utilizzo di VR a basso costo; infatti, un ente dei beni culturali, come un museo, non può certo permettersi di allestire delle strutture simili al Virtual Theatre del CINECA o al CyberStage del GMD.

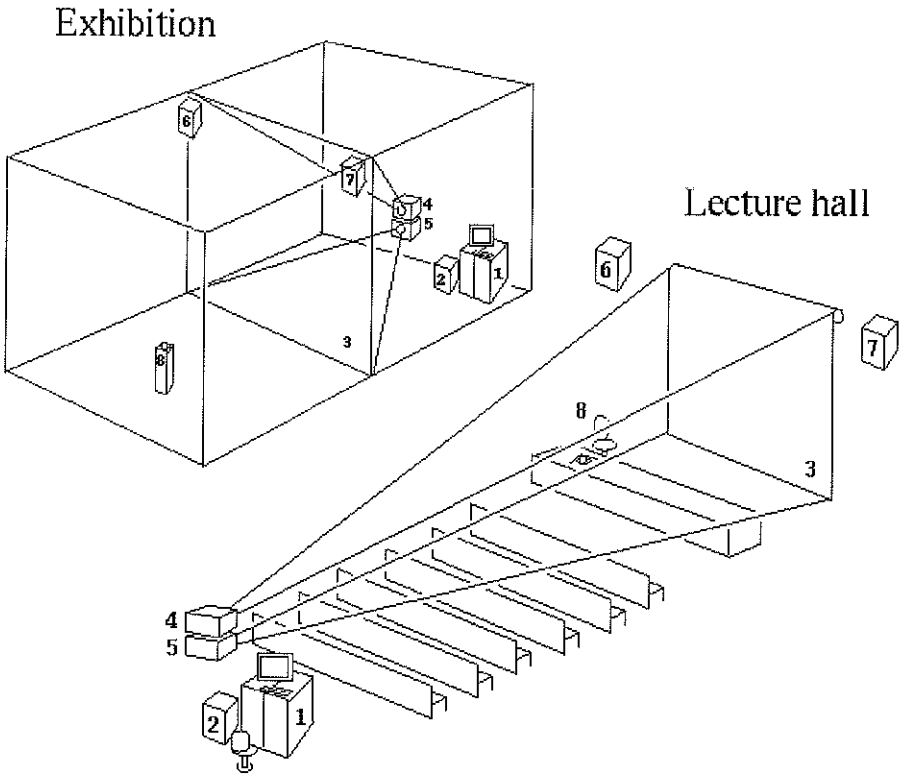
Ovviamente una struttura di realtà virtuale prevede l'impiego sia di tecnologie hardware che software, per cui i prossimi due sottoparagrafi presentano ciascuno una possibile tecnologia a basso costo.

## 2.1. L'hardware

Un'installazione hardware a basso costo che renda possibile immergersi in un mondo virtuale prevede, dal punto di vista funzionale, gli stessi componenti di un'istallazione ad alto costo e cioè un sistema di elaborazione, uno di video proiezione, uno schermo ed un sistema audio.
Una possibilità è la seguente:

- Due PC ciascuno con un'appropriata scheda grafica che costituiscono il sistema di elaborazione (non più quindi una workstation della Silicon), (1).
- Due proiettori per una visione stereo, (4, 5).
- Uno schermo, (3).
- Un PC con un'appropriata scheda sonora, (2).
- Due casse sonore, (6, 7).
- Trackball (mouse), cavi, occhiali stereo passivi, (8).

Tutto questo hardware può essere montato secondo due modi differenti: il primo detto "Exhibition" (per un solo utente), il secondo "Lecture hall" (per più utenti stile cinema). Entrambe i modi sono rappresentati in figura:




In un'istallazione di questo tipo rivestono una particolare importanza le schede grafiche montate sui due PC, che devono essere le più performanti possibili tra quelle presenti sul

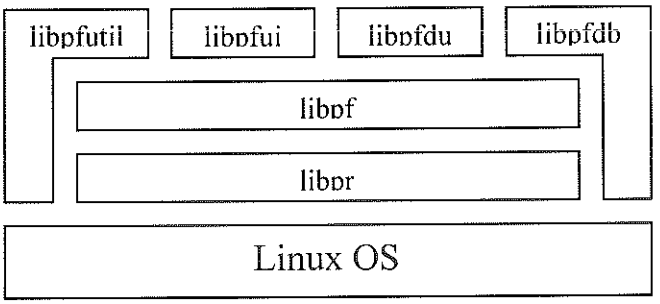
mercato del personal computer. Questo rapporto dedica un intero paragrafo ad analizzare le prestazioni del processore grafico GeForce3 Ti 500 della nVIDIA [3].

## 2.2. Il software

Il progettista, per sviluppare un ambiente virtuale che prevede un'interazione con l'utente finale, ha bisogno di un pacchetto software per lo sviluppo di grafica 3D in tempo reale. Considerando il tipo di hardware a disposizione, due possibili scelte sono: OpenGL Performer della SGI [4][5] e AVANGO sviluppato dal GMD di Bonn [6]. Il primo è fondamentalmente un insieme di librerie grafiche ed è un prodotto commerciale, mentre il secondo è un framework derivante da un ambiente di ricerca, nato da esigenze interne.
OpenGL Performer è stato sviluppato sopra OpenGL, mentre AVANGO poggia sopra OpenGL Performer; quest'ultimo è disponibile sia in ambiente workstation che in ambiente PC con sistema operativo Linux. Per tale motivo, sia che si usi OpenGL Performer sia che si usi AVANGO, la scelta del sistema operativo cade su Linux ed in particolare sulla versione RedHat come raccomandato dagli sviluppatori di Performer.

### 2.2.1 OpenGL Performer

Come è già stato accennato, OpenGL Performer è uno strumento software estensibile per la creazione di grafica 3D in tempo reale. Le componenti principali sono quattro librerie, un supporto di file per tali librerie (come i file header) e del codice sorgente per semplici applicazioni. In più OpenGL Performer fornisce un insieme di caricatori di database, nella forma di codice oggetto del tipo DSO; ad esempio il file libpfobj.so contiene il codice oggetto del caricatore di database nel formato "obj". La figura seguente illustra la gerarchia delle librerie in relazione al sistema operativo (libpfdb è in realtà una collezione di librerie una per ogni tipo di caricatore):




Tale struttura non limita l'accesso di un'applicazione OpenGL Performer ai vari strati, infatti un programma può accedere direttamente ad ognuno di essi, compreso anche quello del sistema operativo. Da notare che all'inizio sono state menzionate quattro librerie mentre dalla figura sembra di capire che ce ne siano cinque; in realtà la libreria libpr non esiste come libreria a se stante ma fa parte della libpf. Segue adesso una sezione che sottolinea le caratteristiche più importanti delle due librerie principali.

La prima è la libpf, per lo sviluppo di simulazioni visive; le funzioni di tali libreria fanno delle chiamate alle funzioni della libpr. Gli elementi fondamentali della libpf sono:

- Un framework per gestire un ambiente multi processo. In OpenGL Performer il rendering di una scena avviene in tre fasi: APP, CULL e DRAW. La prima fase fa

l'elaborazioni richieste dall'applicazione, come ad esempio l'acquisizione dei dati provenienti dai dispositivi di ingresso e l'aggiornamento del database. La seconda consiste nell'attraversamento della scena per la creazione di un'opportuna display list, mentre l'ultima usa tale lista per il disegno della scena (aggiornamento del frame buffer). Queste tre fasi possono essere eseguite da processi distinti, adattando così l'applicazione al numero di CPU disponibili; in più la fase CULL può essere multi thread. OpenGL Performer fornisce delle fasi asincrone addizionali che sono la INTERSECTION per cose come la determinazione delle collisioni, la COMPUTE per calcoli asincroni generali e la DATABASE per il caricamento asincrono di files e l'aggiunta/eliminazione di nodi nel/dal grafo della scena. I problemi più importanti di un ambiente multi processo, come ad esempio la temporizzazione dei processi, la sincronizzazione e la mutua esclusione, sono trasparenti al programmatore.

- Visualizzazione. La libpf fornisce dei costrutti software per facilitare il rendering del grafo della scena. Una pfPipe è l'astrazione software della pipeline hardware, che renderizza uno o più pfChannel in una o più pfPipeWindow; un pfChannel è una vista del grafo della scena, equivalente ad una viewport, dentro una pfPipeWindow.
- Controllo del frame rate. OpenGL Performer è progettato per "correre" ad un determinato frame rate, specificato dall'applicazione; esso misura il carico grafico e usa tale informazione per calcolare un valore di stress. Lo stress è utilizzato per ridurre la complessità della scena quando ci si avvicina a delle condizioni di sovraccarico.
- Grafo della scena. Un grafo della scena è un grafo aciclico direzionale che l'applicazione costruisce (di solito, ma non necessariamente) quando sono caricate le informazioni da un database. Esso è la rappresentazione del mondo virtuale all'interno della memoria.

La seconda è la libpr che è una libreria grafica di basso livello, utile per tutte quelle applicazioni che devono avere delle prestazioni elevate. Gli elementi fondamentali di tale libreria sono:

- Rendering della geometria ad alte prestazioni. La maggior parte delle applicazioni grafiche sono limitate nell'invio di comandi grafici alla pipeline geometrica, dal sovraccarico della CPU. Un pfGeoSet è una collezione di primitive come ad esempio punti, linee, triangoli e strisce di triangoli; attraverso i pfGeoSet OpenGL Performer elimina il collo di bottiglia della CPU.
- Efficiente gestione dello stato grafico. OpenGL Performer ottimizza le prestazioni della libreria grafica attraverso la gestione dei cambiamenti di stato, e fornisce funzioni per controllare gli aspetti dello stato della libreria grafica come l'illuminazione, la texture e la trasparenza.
- Display list. La display list è una lista di comandi geometrici e di stato. Tali liste sono molto utili per situazioni del tipo produttore/consumatore in cui un processo genera una display list di una scena, mentre un altro la renderizza.
- Supporto matematico e intersezioni. Sono fornite semplici funzioni geometriche e funzioni per svolgere intersezioni di segmenti con cilindri, sfere, boxes e piani.
- Punti luminosi. I punti luminosi, definiti dall'oggetto pfLPointState, possono simulare oggetti estremamente emissivi come le luci di una pista di atterraggio, le luci stradali, ecc. La dimensione, la direzione, la forma, il colore e l'intensità di queste luci può essere controllata.
- I/O di file, asincrono. Viene fornito un semplice metodo di accesso ai file non bloccante per permettere alle applicazioni di recuperare i dati all'interno di un file durante operazioni in tempo reale.

- Allocazione della memoria. OpenGL Performer permette di allocare i dati nella memoria heap del processo dell'applicazione o nella cosiddetta "arena"; l'arena deve essere usata quando più processi hanno la necessità di condividere dei dati (un esempio di dati che devono essere condivisi sono quelli che costituiscono il grafo della scena).

Fino ad ora è stata presentata a grandi linee l'architettura di OpenGL Performer, sottolineando le caratteristiche principali delle librerie libpf e libpr. Vediamo i concetti generali di OpenGL Performer.

In OpenGL Performer un mondo virtuale è descritto da un grafo della scena ed una sua vista da un "channel". Tale vista viene renderizzata da una "pipe" in una finestra su un schermo selezionato. Un grafo della scena non è altro che un grafo aciclico direzionale (DAG) e la sua struttura determina l'ordine delle operazioni sui suoi dati. I nodi del grafo sono disposti secondo una gerarchia che può avere diversi significati:

- I nodi figlio possono essere le parti costitutive di un nodo padre; ad esempio il padre rappresenta una lampadina, mentre i suoi due figli rappresentano il bulbo di vetro e la base avvitante.
- Il nodo padre ha il semplice significato di nodo raggruppatore dei suoi nodi figlio.
- I nodi figlio sono tutti delle viste della stessa geometria, ognuna con un certo livello di risoluzione.

Il grafo della scena non è altro che la rappresentazione di tutti i dati presenti nel database ed i suoi nodi, possono rispondere ad alcuni tipi di "attraversatori". Un attraversatore scorre su una parte o su tutta la gerarchia dei nodi, però non tutti i nodi rispondono a tutti i tipi di attraversatori. Esempi di attraversatori sono gli attraversatori CULL e DRAW. La gerarchia dei nodi di un grafo della scena, determina l'ordine in cui i nodi devono essere elaborati da un attraversatore; tale ordine è dall'alto verso il basso e da sinistra verso destra. Un attraversatore applicato al nodo radice, si propaga attraverso l'intero grafo; è tuttavia possibile applicare l'attraversatore ad un sottoinsieme del grafo, applicandolo ad un nodo che non sia la radice.

Un channel equivale ad una macchina fotografica spostabile attraverso la scena. Esso contiene solo quelle informazioni visuali che sono visibili ad un osservatore; in poche parole un channel mostra una porzione di scena da una determinata prospettiva. La vista selezionata da un channel è definata da:

- Una posizione ed un orientamento della macchina fotografica.
- Un "tronco di visualizzazione".

Un tronco non è altro che il tronco della piramide di visualizzazione (geometricamente è una piramide troncata) ed è definito da 4 parametri: piano di taglio vicino, piano di taglio lontano, vista di campo verticale e vista di campo orizzontale. Le geometrie del grafo della scena sono invisibili quando sono oltre il piano di taglio lontano, tra l'osservatore ed il piano di taglio vicino o al di fuori delle viste di campo verticali ed orizzontali. In poche parole le geometrie visibili sono solo quelle che stanno dentro il tronco di visualizzazione.
Ogni channel è associato ad un grafo della scena, mentre ad un grafo della scena possono essere associati più channel.

La pipe renderizza i dati delimitati da un tronco di visualizzazione in una finestra; la pipe è l'astrazione software della pipeline hardware. Il rendering della scena avviene in tre fasi:

- APP: aggiorna gli oggetti (forma e posizione) e la camera (posizione ed orientamento) della scena.
- CULL: determina quali geometrie della scena sono visibili (nel tronco di visualizzazione), eliminando il resto.
- DRAW: renderizza tutte le geometrie visibili.

Ogni fase è potenzialmente un processo separato. Per massimizzare le prestazioni, ognuno di tali processi può essere eseguito su differenti CPU. Se si stanno utilizzando tre CPU, OpenGL Performer è in grado di elaborare tre frame contemporaneamente (ogni CPU esegue una delle tre fasi). L'attraversatore CULL genera una lista (display list) che descrive la scena vista da un channel; l'attraversatore DRAW attraversa tale lista ed invia i comandi grafici in essa presenti, alla pipeline geometrica. Attraversare una display list è più veloce che attraversare il database gerarchico, poiché la lista livella la gerarchia in una semplice ed efficiente struttura. Cambiamenti alla scena non provocano effetti fino a che non si ha una successiva chiamata dell'attraversatore CULL.

Le parti fondamentali di un programma OpenGL Performer sono:

- Inizializzazione: consiste in una chiamata del metodo pfInit(). L'inizializzazione serve a preparare l'arena per i tre processi (APP, CULL e DRAW) ed a inizializzare lo stato grafico di OpenGL Performer. Poiché tutti e tre i processi lavorano sullo stesso frame, è richiesta una memoria (l'arena) che possa essere acceduta da processi distinti. Tutti i processi OpenGL Performer necessitano di accedere al grafo della scena, così tutti i dati del grafo devono stare nell'arena. Il metodo pfInit() crea per te tale zona di memoria.
- Creazione della pipe, del channel e della pipe window: consiste nel creare prima di tutto una pipe, poi un channel da associare alla pipe creata, una pipe window da associare alla pipe ed infine nell'associare il channel alla pipe window.
- Caricamento del grafo della scena: consiste nel caricare una scena da un database.
- Posizionamento del channel: consiste nel chiamare due metodi per posizionare il channel e nella chiamata del metodo pfFrame() che fa partire i processi CULL e DRAW.
- Creazione del simulation loop: il simulation loop guida l'applicazione e si ripete fino a che l'applicazione non termina; esso si sviluppa nel processo dell'applicazione. Consiste di tre passi: aggiornamento della forma e della posizione degli oggetti nella scena, aggiornamento della posizione e dell'orientamento della camera ed infine ridisegno della scena.

### 2.2.2 AVANGO

Per progettare un sistema software nel contesto della realtà virtuale come AVANGO, i ricercatori del GMD di Bonn hanno sviluppato un modello astratto per la descrizione di mondi virtuali. Qualsiasi cosa compresa in un mondo virtuale è incapsulata in oggetti, e può essere manipolata variando gli attributi degli oggetti. Per descrivere le relazioni tra gli oggetti, si fa uso di un grafo aciclico direzionale (modellazione gerarchica) e gli oggetti sono nodi di tale grafo. Lo stato dei nodi assieme alle loro relazioni, definisce lo stato del mondo virtuale. Tale stato deve essere completo e ciò significa che tutti i diversi sistemi di rappresentazione (visuali, auditivi o tattili), puntati ai canali sensoriali della percezione umana, devono essere indirizzati attraverso nodi del grafo. Ogni nodo ha un suo modo di valutazione o meglio di

rendering per tutti i sistemi di rappresentazione disponibili. I nodi sono riferiti come nodi genitore o nodi figlio. Un nodo può avere più di un genitore, perciò il grafo in generale non è un albero; ciò è vantaggioso per riferimenti multipli allo stesso oggetto, perché si evita di copiare tutti i dati relativi a quell'oggetto. Per sostenere ambienti multi utente, gli oggetti ed i loro stati devono essere condivisi tra differenti siti su una connessione di rete. Per ridurre la quantità di dati che devono essere trasferiti, vengono trasmessi solo i cambiamenti di stato. Trattare un mondo virtuale come una macchina a stati è un grosso vantaggio per la risoluzione del problema della distribuzione, inoltre è possibile memorizzare una copia del mondo sul disco in ogni momento.

AVANGO è un framework orientato agli oggetti per lo sviluppo di applicazioni di realtà virtuale interattive e distribuite. Il sistema è formato da una libreria, che implementa le funzionalità base di AVANGO e da un insieme di strumenti, costruiti su di essa, usati per configurare le applicazioni. La distribuzione dei dati è ottenuta attraverso una riproduzione trasparente del grafo della scena, tra i processi costituenti l'applicazione distribuita (concetto di grafo della scena condiviso). Ogni processo possiede una copia locale del grafo della scena e dell'informazione di stato in esso contenuta, la quale è mantenuta sincronizzata. AVANGO permette la creazione di classi specifiche dell'applicazione, che ereditano tali proprietà di distribuzione; inoltre il grafo della scena è potenziato con un grafo "dataflow" distribuito.

AVANGO usa il linguaggio di programmazione C++ per definire due categorie di classi. I *nodi* foniscono un API del grafo della scena orientata agli oggetti che permette la rappresentazione ed il rendering di geometrie complesse. I *sensori* costituiscono l'interfaccia con il mondo reale e sono usati per importare i dati dai dispositivi esterni. Tutti gli oggetti di AVANGO sono "fieldcontainers", cioè l'informazione di stato di un oggetto è rappresentata come una collezione di campi. Gli oggetti hanno un'interfaccia generica di "streaming" che permette di scrivere l'informazione di stato in uno "stream" e successivamente la ricostruzione dell'oggetto da quello stream. AVANGO usa i collegamenti tra campi per costruire un grafo dataflow ortogonale al grafo della scena. La scelta di costruire AVANGO sopra OpenGL Performer deriva dalla necessità di ottenere le massime prestazioni possibili dalle applicazioni. Compiti avanzati come il cambiamento del livello di dettaglio, la comunicazione con l'hardware grafico ed il "culling" sono gestiti da Performer. In aggiunta all'API C++ AVANGO presenta un linguaggio completo, di collegamento al linguaggio Scheme. Scheme è un linguaggio interpretato ad alto livello per scopi generali, derivante dall'Algol e dal Lisp. Tutti gli oggetti di alto livello di AVANGO possono essere creati e manipolati con Scheme. Per tale motivo, un approccio allo sviluppo di una applicazione, può essere quello di implementare con il C++ (estendendo o derivando le classi esistenti di AVANGO) le funzionalità più complesse e critiche dal punto di vista delle prestazioni; poi di per se l'applicazione è una collezione di script scritti nel linguaggio Scheme che istanziano gli oggetti AVANGO desiderati, chiamano i metodi su tali oggetti, settano i valori dei loro campi e definiscono le relazioni tra di essi. Gli script possono essere scritti, testati e se ne può fare il debug mentre l'applicazione è in esecuzione.

Performer usa delle funzioni membro per accedere agli attributi di stato di un oggetto; per ogni attributo esiste una coppia simmetrica di funzioni "set" e "get". AVANGO usa i *campi* come contenitori per gli attributi di stato dell'oggetto. I campi incapsulano i tipi di dato fondamentali e forniscono una generica interfaccia di streaming. Essi sono implementati come membri pubblici e sono perciò ereditati dalle classi derivate; inoltre sono direttamente accessibili dalle classi cliente. Ci sono due tipi di campi: "single-fields" e "multi-fields". I primi contengono un solo valore mentre i secondi contengono un vettore di un arbitrario numero di valori. Come già accennato gli oggetti AVANGO sono fieldcontainers, che sta a

significare che lo stato di un oggetto è rappresentato da una collezione di campi. Per ognuno di tali oggetti esistono dei metodi pubblici per sapere il numero dei campi presenti e per ottenere un loro riferimento.

AVANGO usa il concetto di "fieldconnections": i campi di tipo compatibile possono essere collegati così che se il valore di un campo sorgente cambia, il nuovo valore viene immediatamente inoltrato al campo destinatario. Usando i collegamenti tra i campi, può essere costruito un grafo dataflow che è concettualmente ortogonale al grafo della scena. AVANGO utilizza questo meccanismo per specificare relazioni supplementari tra i nodi, che non possono essere espresse in termini di grafo della scena. La valutazione del grafo dataflow avviene una volta per rendering del frame. I collegamenti di campo inoltrano i cambiamenti di valore immediatamente, in modo tale che non c'è ritardo di propagazione per percorsi con collegamenti in cascata; gli anelli vengono scoperti e correttamente risolti. Un fieldcontainer può implementare effetti secondari su cambiamenti di campo, attraverso l'overloading delle funzioni membro notify() e evaluate(). Tutte le volte che il valore di un campo cambia, viene chiamato sul fieldcontainer il metodo notify() con un riferimento al campo modificato come argomento. Dopo che tutte le notifiche per tutti i campi su tutti i fieldcontainer sono state fatte per un frame, viene chiamato il metodo evaluate() su ogni fieldcontainer che ha avuto almeno uno dei suoi campi notificati.

Tutti i nodi di Performer sono stati adattati a fieldcontainers (cioè le classi pfGroup, pfDCS, pfLOD, ecc.) e così la maggior parte delle classi di Performer (pfGeoState, pfMaterial, ecc.); cioè tutte quelle classi che assieme rappresentano l'API del grafo della scena di Performer. Per convenzione i nodi di AVANGO sostituiscono il prefisso "pf" con il prefisso "av". L'abilità di mescolare nodi AVANGO con nodi Performer per costruire il grafo della scena, può essere convenientemente usata per definire nuovi nodi con funzionalità aggiuntive.

I sensori rappresentano l'interfaccia di AVANGO con il mondo reale. Essi sono derivati dalla classe avFieldContainer (non da nodi di Performer) e non possono essere inseriti nel grafo della scena. I sensori incapsulano il codice necessario per accedere ai dispositivi di input di vario tipo. I dati generati da un dispositivo sono mappati nei campi del sensore. Tutte le volte che un dispositivo genera nuovi dati, i campi del corrispondente sensore sono aggiornati di conseguenza. I collegamenti tra i campi di un sensore ed i campi di un nodo del grafo della scena, sono usati per incorporare i dati di un dispositivo in una applicazione. AVANGO utilizza un processo demone dispositivo, che tratta l'interazione diretta con i dispositivi attraverso la linea seriale o una connessione di rete. Il demone aggiorna i valori dei dati del dispositivo in un segmento di memoria condiviso, dove la classe av*Device*Sensor (cioè il sensore) può accedere. Sono state definite molte classi av*Device*Sensor per supportare una grande varietà di dispositivi di input, come mouse, tastiere, guanti, ecc.

Lo scopo principale che ha guidato il progetto di AVANGO nei confronti della distribuzione, era di rendere lo sviluppo di applicazioni distribuite così semplici come lo sviluppo di applicazioni indipendenti. Per ottenere questo grado di trasparenza, il concetto di distribuzione è una parte integrante del modello a oggetti di AVANGO.

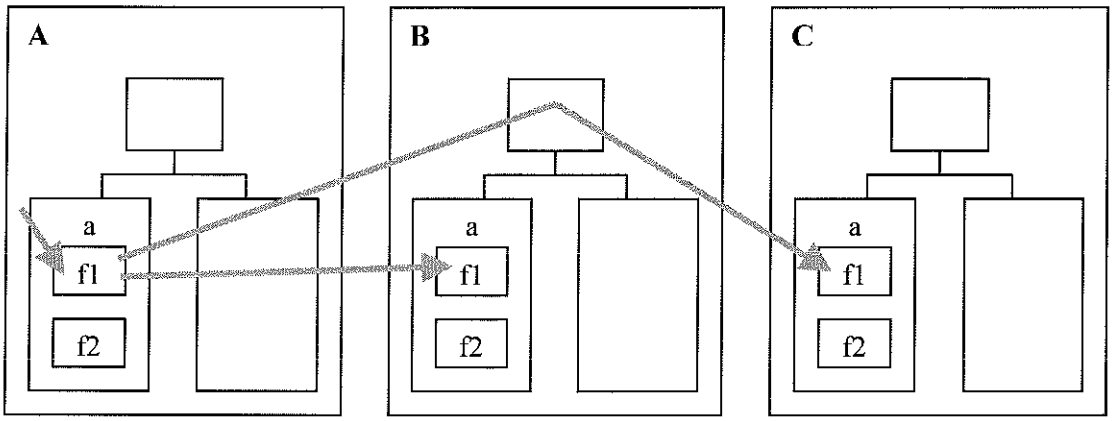
Le grosse richieste di larghezza di banda durante l'accesso agli oggetti per il rendering, rende necessario che ogni oggetto esista nella memoria locale. Per tale motivo in AVANGO il supporto per la distribuzione è basato sulla replica dell'oggetto e sul modello della memoria condivisa distribuita. Memoria condivisa significa un'area di memoria locale che può essere acceduta contemporaneamente da più di un processo su una singola macchina. Ogni processo assegnato ad un segmento di memoria condivisa vedrà tutti i cambiamenti che altri processi

assegnati applicano al segmento. Il concetto di memoria condivisa distribuita (DSM) estende ciò a processi distribuiti sopra una rete. Ogni processo mantiene una copia locale del segmento DSM, che è tenuto sincronizzato nei confronti dei cambiamenti che altri processi possono applicare alla loro copia locale. I processi si possono assegnare ad uno o più gruppi di distribuzione. Gli oggetti possono essere istanziati sia localmente come oggetti locali, o in uno dei gruppi di distribuzione assegnati, come oggetti distribuiti. Un oggetto locale esiste solo nello spazio di indirizzi del processo che lo ha creato, mentre un oggetto distribuito istanzierà in modo trasparente copie nello spazio di indirizzi di ogni processo assegnato al gruppo di distribuzione. Cambiamenti successivi allo stato di un oggetto, saranno instradati a tutte le copie distribuite dell'oggetto.

In AVANGO la creazione di un oggetto distribuito è un processo che avviene in due fasi. Per prima cosa viene creato un oggetto locale, il quale poi, in un secondo momento, viene migrato al gruppo di distribuzione desiderato. La migrazione implica l'annuncio di un nuovo oggetto al gruppo di distribuzione e la disseminazione dello stato dell'oggetto corrente a tutti i membri del gruppo. L'interfaccia di streaming dell'oggetto è usata per trasmettere (un po alla volta) lo stato dell'oggetto in un buffer, il cui contenuto viene inviato a tutti i membri del gruppo; quest'ultimi a sua volta useranno le informazioni di stato inviate per crearsi una copia dell'oggetto. L'oggetto distribuito appena creato esisterà come una copia locale in ognuno dei processi partecipanti. Tutte le volte che il valore di un campo di un oggetto distribuito viene modificato localmente, il nuovo valore viene trasmesso al buffer e di conseguenza inviato a tutti i partecipanti del gruppo; in tal modo tutte le copie dell'oggetto distribuito risultano sincronizzate. La relazione genitore figlio tra oggetti del grafo della scena, è rappresentata da dei riferimenti in un campo di tipo multi-fields nel nodo genitore. Poiché tutti i valori di campo sono inviabili, lo schema di distribuzione descritto sopra non solo distribuirà e sincronizzerà i singoli oggetti, ma anche le loro relazioni genitore figlio. In tal modo è possibile replicare un completo grafo della scena a tutti i processi di un gruppo di distribuzione. Questo è un passo chiave verso la semplificazione dello sviluppo di applicazioni distribuite, ma non è sufficiente. Consideriamo il caso di un gruppo di distribuzione con due processi membro A e B. Entrambi i processi hanno già creato diversi oggetti distribuiti in quel gruppo. Ora un terzo processo C si unisce al gruppo. Da ora in poi, tutti e tre i processi saranno notificati di future creazioni e manipolazioni di oggetti, ma il processo C non sa quali sono gli oggetti che A e B hanno creato prima che si unisse. AVANGO supera questo problema attraverso un trasferimento atomico di stato; quando un nuovo processo si unisce ad un gruppo di distribuzione già popolato, uno dei membri più vecchi si assume la responsabilità di trasmettere lo stato corrente del gruppo al nuovo membro. Ciò implica l'invio di tutti gli oggetti, correntemente distribuiti nel gruppo, con tutti i loro valori di campo al nuovo arrivato. A trasferimento di stato avvenuto, il nuovo membro avrà l'esatta copia dell'insieme di oggetti del gruppo. Per evitare problemi di consistenza, durante il trasferimento di stato vengono sospese tutte le altre comunicazioni (il trasferimento viene svolto come un'azione atomica). La capacità di replicare l'intero grafo della scena, unita al trasferimento di stato, risolve effettivamente il problema della duplicazione del database.

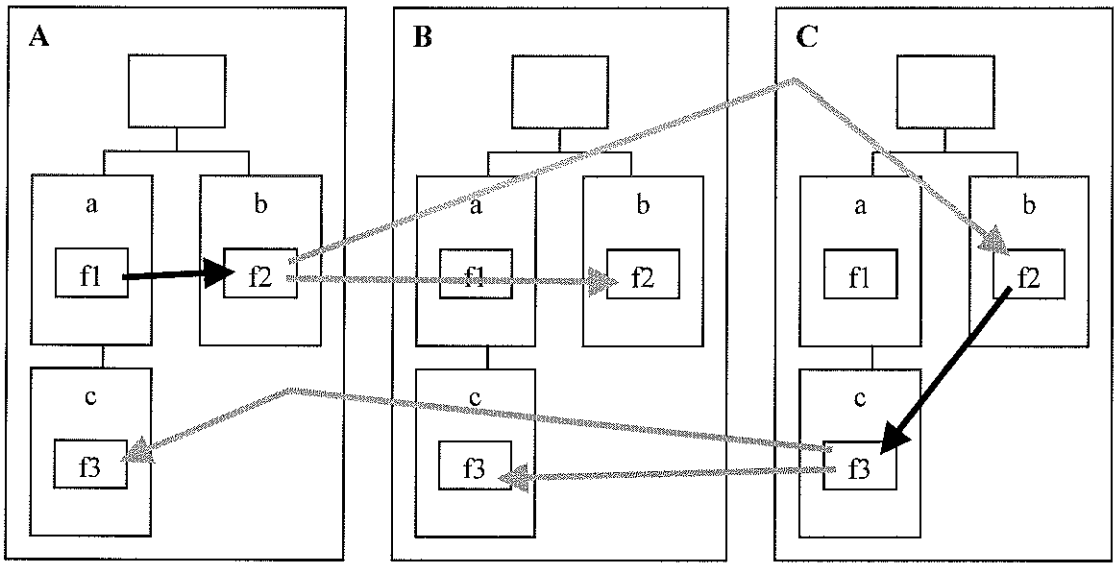
Consideriamo la configurazione seguente (vedi figura). Tre processi A, B e C si sono uniti ad un gruppo di distribuzione ed è stato costruito un piccolo grafo della scena. Ora il processo A cambia il valore del campo f1 sul nodo "a". Poiché il nodo è distribuito, il nuovo valore viene comunicato al campo f1 delle corrispondenti copie. Supponiamo che la funzione notify() sul nodo "a", modifichi il valore del campo f2 in conseguenza del cambiamento di valore del campo f1. A causa del meccanismo di comunicazione, la notify() viene chiamata su ogni copia di "a"; essa cambierà il contenuto del campo f2 della copia, che sarà a sua volta distribuito. Segue che il campo f2 viene aggiornato esattamente N volte su ogni copia del

nodo "a", dove N è il numero delle copie distribuite implicate, in questo caso tre. Ciò è chiaramente non desiderabile:




Se tentiamo di risolvere il problema non chiamando la notify() sulle copie del fieldcontainer modificato, non sarebbe possibile implementare il nodo avField come nodo distribuito. Il nodo avField carica la geometria specificata come effetto secondario della modifica del campo Url. Se un processo cambia il campo Url su un nodo avField distribuito, noi vogliamo che ciò succeda su tutte le copie distribuite del nodo, così che tutte le copie rappresentino la stessa geometria. Per risolvere il problema dividiamo ogni funzione di notifica (non solo la notify() ma anche la evaluate()) in due parti, una per effetti secondari locali ed una per effetti secondari distribuiti. Quelle per effetti secondari distribuiti (notify() ed evaluate()) vengono chiamate solo su una copia distribuita del nodo. Tali funzioni possono svolgere modifiche ad altri campi. Le funzioni per effetti secondari locali invece (notify_lse() ed evaluate_lse()), saranno invocate su tutte le copie distribuite del nodo; la sola restrizione è che non devono modificare altri campi.

I collegamenti di campo non sono distribuiti; esistono solo nel processo che ha collegato i campi implicati. Consideriamo l'esempio dato in figura:




tre processi A, B e C sono assegnati ad un gruppo di distribuzione e condividono un piccolo grafo della scena di quattro nodi. Il processo A ha collegato il campo f1 del nodo "a" al campo f2 del nodo "b", mentre il processo C ha collegato il campo f2 del nodo "b" al campo

f3 del nodo "c" (frecce più scure). Adesso seguiremo la catena di notifiche che iniziano nel momento in cui il processo A scrive un nuovo valore nel campo f1 del nodo "a". A causa del collegamento di campo, il campo f2 sul nodo "b" del processo A, riceverà immediatamente il nuovo valore. Poi, tutte le copie distribuite del nodo "b" riceveranno il nuovo valore per il campo f2. Come conseguenza del collegamento di campo presente nel processo C, il campo f3 del nodo "c" riceverà il nuovo valore; questo, a sua volta, farà scattare un aggiornamento di f3 su tutte le copie distribuite del nodo "c". E' facile osservare che il comportamento del nostro grafo dataflow distribuito, coincide esattamente con il comportamento che ci aspetteremo da un grafo indipendente con i collegamenti da f1 a f2 e da f2 a f3 (per tale motivo i collegamenti di campo non sono distribuiti).

## 3. La scheda grafica

In questo paragrafo andiamo ad analizzare le caratteristiche principali di una scheda grafica 3D, basata sul processore GeForce3 Ti 500 della nVIDIA. Prima di iniziare vediamo alcuni concetti fondamentali (di base) di una scheda grafica 3D. Essenzialmente le schede grafiche sono la via di comunicazione tra la scheda madre del computer ed il monitor. Il cuore di una scheda grafica è la GPU (processore grafico) il cui compito è, in poche parole, quello di disegnare i poligoni, mappare le texture su tali poligoni, calcolare i valori di illuminazione della scena ed infine tracciare i pixel sul monitor (la GPU è realmente un co-processore). L'intero processo di elaborazione appena accennato, dai poligoni fino alla visualizzazione dei pixel, è chiamato "pipeline grafica 3D". L'applicazione inoltra le informazioni relative ad una porzione della scena, al driver della GPU il quale le invia all'hardware grafico. Da questo momento in poi iniziano una serie di passi che costituiscono la pipeline grafica 3D:

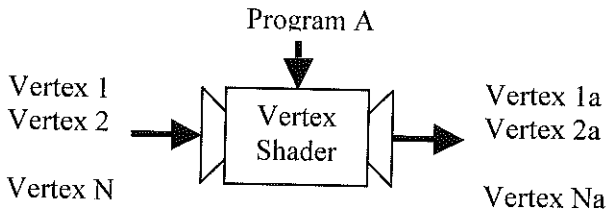
- Tassellazione delle superfici "High-order": la maggior parte degli oggetti 3D sono formati da triangoli perché più facili da elaborare per le GPU. Alcuni oggetti invece sono costituiti da linee curve che possono essere molto complesse da un punto di vista matematico, perché descritte da formule high-order (formule con variabili elevate a potenza). Oggetti formati da superfici high-order devono essere suddivisi in triangoli, prima di essere inviati alla successiva unità funzionale della GPU. Per tale motivo la tasselazione delle superfici high-order costituisce il primo passo della pipeline.
- "Vertex shading" (compreso il "T&L"): una volta che l'oggetto è costituito da un insieme di triangoli, la funzionalità di vertex shading della GPU è pronta a fare il suo lavoro eseguendo operazioni di "transform" (T) e "lighting" (L). La prima operazione consiste nello scalare, traslare o ruotare gli oggetti mentre la seconda riguarda il calcolo degli effetti di illuminazione per ogni vertice di ogni triangolo.
- "Setup" del triangolo: l'unico obbiettivo di questa funzionalità è quello di convertire matematicamente i dati, così come prodotti dal passo precedente, per renderli comprensibili alla funzionalità successiva.
- "Pixel shading" e rendering (compresa la "texturing"): il pixel shading ed il rendering comprendono tutti i calcoli per determinare il colore finale di ogni pixel; tali operazioni devono anche tener conto di eventuali texture, al fine del calcolo del colore. Una texture può descrivere modifiche del colore, modifiche dell'illuminazione, proprietà dei materiali e molte altre cose ancora. L'ultima mansione di questa funzionalità è quella di immagazzinare i pixel nel frame buffer.
- Visualizzazione: il controller del display legge l'informazione dal frame buffer e la invia al driver del display selezionato.

La famiglia GeForce3 dell'nVIDIA comprende 3 GPU: GeForce3, GeForce3 Ti 200 e GeForce3 Ti 500. Tutte e tre hanno come caratteristiche chiavi il motore "nFiniteFX" e l'architettura della memoria "lightspeed"; in più le ultime offrono all'utente le tecnologie "shadow buffer" e "texture 3D".

### 3.1. Il motore nFiniteFX

Il motore nFiniteFX è un'unità programmabile della GPU che fornisce l'elaborazione del vertice ("vertex shader") e la capacità di gradazione del pixel ("pixel shader"). Segue una descrizione delle due tecnologie.

Un vertex shader è una funzione di elaborazione grafica che fa delle operazioni matematiche sui dati che definiscono un vertice (ad esempio la posizione x, y e z, il colore, la texture, ecc. sono tutte informazioni che definiscono un vertice). Tali operazioni matematiche non modificano il tipo dei dati ma ne cambiano semplicemente i valori; si ha così che un vertice appare in una posizione diversa, oppure di un colore diverso. Questo tipo di elaborazione grafica prima poteva essere fatta solo off-line cioè non in tempo reale; ciò perché l'hardware della GPU non permetteva di definire (programmare) un vertex shader (offriva solo un menu fisso di effetti visivi). Se uno sviluppatore aveva bisogno di definire un'elaborazione personalizzata era costretto ad implementarla via software. Il motore nFiniteFX offre dei vertex shader programmabili ed ognuno di essi può essere immaginato come una funzione il cui corpo è definibile dal programmatore attraverso un insieme di istruzioni. Tale funzione prende in ingresso un insieme di vertici (di una geometria), e restituisce gli stessi vertici eventualmente modificati (può darsi che modifichi solo alcuni vertici in base ad un criterio stabilito dallo sviluppatore); ovviamente per "vertici in ingresso" si intende l'informazione ad essi associata. Uno stesso vertex shader, che ha compiuto una determinata elaborazione su un oggetto grafico, può essere in seguito riprogrammato per compiere un'elaborazione diversa.




In passato i vertex shader programmabili non erano disponibili all'interno di una GPU perché troppo complessi e le GPU avevano solo delle unità T&L; una unità T&L fa solo una ben precisa elaborazione grafica (non è programmabile). Le operazioni di un vertex shader venivano fatte fare alla CPU. Ciò comporta che per applicazioni di resa off-line non c'è bisogno di strutture hardware costose mentre per applicazioni di resa in tempo reale si. Per problemi di compatibilità con le vecchie applicazioni (che utilizzano solo unità T&L), all'interno della GPU, i vertex shader sono affiancati alle unità T&L. Con i vertex shader programmabili sono possibili un'infinità di effetti come:

- Animazione di soggetti elaborati: realizzare pelle ed abiti più realistici che si stendono o si raggrinzano a livello delle articolazioni (ad esempio l'animazione facciale può creare fossette o rughe). Calcolare l'intera animazione che sta tra due key-frame.
- Effetti ambientali: effetto nebbia/fumo; ad esempio una collina/tavolo si innalza dalla nebbia/fumo perché un vertex shader può utilizzare l'effetto selettivamente a seconda

dell'altezza dei vertici di un oggetto. Effetto ondata di calore. Rifrazione della luce (ad esempio nell'acqua).

- "Procedural deformation": i vertex shader possono calcolare la procedural deformation; essa può essere sfruttata ad esempio per animare una bandiera (effetto dinamico) o per creare delle ammaccature su un oggetto metallico (effetto statico).
- Rendere indistinto il movimento: i vertex shader possono rendere indistinto il movimento di un oggetto (ciò è utile per avere l'impressione che esso abbia una velocità notevole).
- Effetti lente: i vertex shader possono essere programmati per fare delle trasformazioni su misura al fine di riprodurre effetti di lenti ottiche.
- Effetti di illuminazione: è possibile dare una caratteristica di illuminazione al lato "posteriore" di un triangolo. Ciò implica che si possono avere superfici con caratteristiche di illuminazione differenti su entrambi i lati, senza ricorrere ad un raddoppio dei triangoli.

I vertex shader sono programmabili attraverso l'uso di istruzioni e registri ed un vertex shader può essere programmato per elaborare più compiti in parallelo, oppure più compiti in serie (un primo compito per una frazione di secondo, poi un secondo e così via).

Il pixel shader è un'unità di elaborazione grafica che determina il colore di ogni singolo pixel del frame, sulla base dei dati provenienti dalla fase di setup del triangolo e sulla base delle texture (può trattare fino a 4 texture contemporaneamente). La grossa novità è che è programmabile e quindi gli sviluppatori non sono più limitati a scegliere tra un numero di effetti preconfezionati. Agli albori degli acceleratori 3D, texture a bassa risoluzione venivano mappate sopra i poligoni di un oggetto con una tecnica di filtraggio bilineare. Nel 1999 l'nVIDIA ha introdotto una tecnologia (NSR: nVIDIA Shading Rasterizer) che permetteva di aggiungere degli effetti (tipico è quello di illuminazione) ai singoli pixel. Attualmente, con l'introduzione dei pixel shader programmabili, gli effetti per pixel sono programmabili, è possibile trattare fino ad un massimo di 4 texture (sia in tecniche di passaggi multipli, sia in tecniche di passaggio unico) ed infine sono possibili letture di texture dipendenti. Da notare che con la possibilità di utilizzare più texture (al max 4) in un unico passaggio si hanno delle prestazioni migliori rispetto all'utilizzo di passaggi multipli; i passaggi multipli rallentano l'intero processo di rappresentazione.

### 3.2. Architettura della memoria lightspeed

Una volta che l'applicazione ha calcolato quale porzione della scena deve essere rappresentata (elaborata), l'applicazione invia il sottoinsieme dal database alla "pipeline geometrica". Storicamente l'esecuzione della pipeline geometrica veniva svolta dalla CPU, ma adesso questi calcoli vengono fatti dalla GPU. Tale spostamento di carico ha reso possibile agli sviluppatori di contenuto di passare da centinaia di poligoni per frame a centinaia di migliaia. Queste scene ricche, mentre sono più irresistibili dal punto di vista visivo, hanno anche una larghezza di banda molto elevata. Ad esempio se consideriamo che in media una scena è formata da 100.000 poligoni e che ogni vertice ha 50 byte di informazione associata, ogni frame contiene 100.000 x 3 x 50 = 15 Mb di informazione geometrica; poiché c'è il requisito di un frame rate di 60 frame al secondo, si ha una larghezza di banda di 900 Mb/sec. Un problema che riveste l'architettura dei PC attuali con questi tipi di carico è la comunicazione fra la CPU principale e le GPU attuali. Il collegamento fra questi due sistemi generalmente è il bus AGP dell'Intel la cui implementazione più avanzata è l'AGP4x. L'AGP4x offre una larghezza di banda di 1.0 Gb/sec che sono a malapena sufficienti (in media però) per scene

ricche di contenuto. Si ha il problema della larghezza di banda della geometria. Per risolvere il problema nVIDIA ha sviluppato la soluzione delle superfici high-order. Le superfici high-order consentono allo sviluppatore di creare superfici curve complesse definite attraverso formule matematiche, invece che attraverso migliaia di triangoli. I processori grafici della famiglia GeForce3 sono in grado di processare queste efficienti (dal punto di vista dell'occupazione della memoria) formule in tempo reale, attraverso l'hardware, per "creare" la geometria all'interno della GPU. In tal modo è evidente che l'occupazione di banda del bus AGP si riduce.

Un secondo problema che si presenta è quello della larghezza di banda per la rappresentazione dei pixel. Normalmente generare un singolo pixel una sola volta, richiede alla GPU una lettura dal "color buffer", una lettura dallo "Z-buffer", una lettura per leggere i dati della texture più due scritture per aggiornare una il color buffer e l'altra lo Z-buffer. Supponendo di trasferirc 4 byte per ogni operazione, la generazione di un pixel richiede il trasferimento di 20 byte (trasferimento che avviene tra la GPU e la memoria che sta sulla scheda grafica). 20 byte possono sembrare pochi, ma se si tiene conto che la complessità della profondità media di un frame è 2.5 (la complessità di profondità è il numero di volte in cui ogni dato pixel deve essere rappresentato prima che l'immagine sia completa; ad esempio l'immagine di una persona davanti ad un muro ha complessità di profondità pari a due), che si presume una risoluzione di 1024 x 768 ed un frame rate di 60 Hz, si ottiene una banda di 1024 x 768 x 2.5 x 20 x 60 = 2.4 Gb/sec. Se si aumenta poi la risoluzione a 1600 x 1200 si ha 5.8 Gb/sec. Una così enorme quantità di larghezza di banda può essere realizzata con GPU/memorie veloci (in grado di lavorare a frequenze elevate) ed in grado di sostenere trasferimenti di grosse quantità di dati alla volta (attualmente viene usata la memoria DDR che supporta trasferimenti di 256 bit per periodo). Tuttavia anche con sistemi così avanzati, la larghezza di banda per la rappresentazione dei pixel è uno dei limiti maggiori all'aumento della risoluzione e/o del frame rate di applicazioni grafiche. I processori grafici della famiglia GeForce3 implementano tre tecnologie al fine di incrementare l'efficienza di rappresentazione dei pixel:

- Controller della memoria "cross-bar": se si fa uso di memoria DDR un tipico controller della memoria accede alle informazioni in blocchi di 256 bit alla volta. Attualmente le scene, a parità di risoluzione, risultano essere sempre più ricche di triangoli e di conseguenza la misura media in pixel del triangolo può essere veramente piccola. Se un triangolo misura 2 pixel e per ogni pixel c'è bisogno di trasferire ad esempio 32 bit, l'ammontare complessivo di dati per un triangolo è di 64 bit. Se il controller della memoria può accedere alle informazioni solo in parti di 256 bit c'è uno spreco di larghezza di banda (relativamente all'esempio si spreca il 75% della banda). Per risolvere il problema e raggiungere una granularità di 64 bit (avendo però sempre la possibilità di trasferire 256 bit alla volta), sono stati implementati quattro controller della memoria indipendenti che possono accedere alla memoria contemporaneamente.
- Compressione Z senza perdite: i processori grafici tradizionali leggono/scrivono i dati dallo/nello Z-buffer per ogni pixel che rappresentano facendo del traffico Z-buffer uno dei più grandi consumatori di larghezza di banda della memoria. Implementando una forma di compressione dati senza perdita di 4:1, la larghezza di banda della memoria utilizzata dal traffico Z-buffer viene ridotta di un fattore 4. Tale compressione è implementata nell'hardware in modo trasparente alle applicazioni con la compressione e la decompressione che avviene in tempo reale.
- "Z-Occlusion Culling": una scena tipica attuale ha una complessità di profondità media di 2, che significa che per ogni pixel che diventa visibile, vengono in media

rappresentati due pixel. Ciò significa che per ogni pixel visibile la GPU deve accedere al frame buffer due volte, essenzialmente per rappresentare pixel che l'utente non vedrà mai. Per mezzo della tecnologia Z-Occlusion Culling, la GPU è in grado di determinare prima se un pixel alla fine sarà visibile. Se un pixel sarà nascosto e l'unità Z-Occlusion Culling lo determina, il pixel non verrà rappresentato, non si accede al frame buffer e la larghezza di banda del frame buffer è salva. Ovviamente più la complessità di profondità media è alta e più si ha un aumento delle prestazioni.

### 3.3. Shadow Buffer

Si tratta di una tecnologia proveniente dalla grafica professionale, introdotta dall'nVIDIA nei processori grafici GeForce3 Ti. Essa consiste nella creazione di una mappa ("shadow map") che rappresenta l'ombra "portata" dai vari oggetti illuminati nella scena. Tale mappa viene memorizzata in un buffer (detto appunto shadow buffer) per poter essere utilizzata più tardi ed acceduta come una texture attraverso una speciale unità hardware presente all'interno della GPU. Per creare e usare uno shadow buffer la GPU deve avere delle speciali caratteristiche hardware che consistono nel:

- designare un'area della memoria grafica a shadow buffer
- rappresentare la shadow map nello shadow buffer
- leggere i dati dallo shadow buffer necessari ai calcoli matematici per determinare se uno specifico pixel nel frame buffer è completamente oscurato, parzialmente oscurato o non oscurato

Per creare la shadow map il processore grafico renderizza la scena dal punto di vista della sorgente luminosa, come se l'osservatore fosse localizzato nella stessa posizione e guardasse nella stessa direzione della sorgente. Una volta che la mappa è stata creata e memorizzata, il contenuto dello shadow buffer viene utilizzato nella fase di pixel shading e rendering. I vantaggi di questa tecnica, che non è comunque sostitutiva delle altre (e che sono ugualmente supportate in hardware), sono molteplici: oltre alla velocità di esecuzione, la possibilità di poter gestire scene in cui la quantità di ombre multiple risulterebbe improba coi metodi classici, alla possibilità di poter gestire effetti visivi anche sulle ombre stesse (per esempio le ombre diffuse) grazie all'applicazione su di esse di particolari algoritmi, fino alla possibilità del cosiddetto "self-shadowing", ovvero la possibilità di riprodurre l'ombra portata anche su parti stesse dell'oggetto che la genera.

### 3.4. Texture 3D

Le texture 3D, a differenza di quelle 2D, contengono delle informazioni anche lungo la terza dimensione (la profondità) oltre che lungo la larghezza e l'altezza. Esse possono essere pensate come composte da numerose "fette" (texture 2D) poste una sopra l'altra sino ad ottenere un cubo; infatti le texture 3D hanno delle dimensioni del tipo 64 x 64 x 64, 128 x 128 x 128, 256 x 256 x 256, ecc. L'implementazione delle texture 3D in una forma realmente utilizzabile comporta diversi problemi. Alcuni di questi problemi sono:

- le texture 3D sono enormi: il file contenente una texture 3D può raggiungere delle dimensioni notevoli come 64 Mb o addirittura 512 Mb. Ciò significa che una texture 3D caricata completamente in memoria può consumare l'intero frame buffer.

- distorsione: è difficile trovare una texture in grado di avvolgere l'intera geometria di un oggetto senza avere un effetto di distorsione. Per risolvere tale problema molte applicazioni grafiche contengono più risoluzioni e dimensioni di una determinata texture. Nella determinazione del colore di un pixel vengono scelte due differenti risoluzioni della texture sorgente ("mip mapping"). Lo stesso effetto di distorsione lo si ha anche con le texture 3D ed il mip mapping in tre dimensioni non è la stessa cosa di quello in due dimensioni.

La GeForce3 Ti (sia la 200 che la 500) supporta la compressione per le texture 3D fino ad un rapporto di 8:1 ed è in grado di eseguire un vero "3D mip mapping".

## 4. Caratteristiche avanzate di OpenGL Performer

In questo paragrafo analizziamo alcuni aspetti avanzati di OpenGL Performer ed in particolare la gestione del frame rate, la gestione del "LOD" ("Level-Of-Detail") ed il discorso del multiprocessing.

### 4.1. Gestione del frame rate

Il frame rate è una quantità che si misura in Hz ed indica il numero di frame che l'applicazione visualizza al secondo; in altre parole un valore di 20 Hz significa che il display viene aggiornato 20 volte al secondo e che un singolo frame rimane visualizzato per un tempo di 50 ms (tempo che l'applicazione ha a disposizione per generare il frame successivo). Poiché il tempo di elaborazione di un frame è per sua natura variabile, è impossibile che ogni frame sia pronto entro il tempo desiderato; per tale motivo la funzione pfFrameRate() di Performer serve soltanto a specificare un desiderato frame rate ma ciò non garantisce nulla. E' possibile che alcuni frame richiedano molto più tempo di calcolo di quello stabilito; ciò causa che dei frame vengono saltati. Una situazione in cui i frame vengono saltati è detta situazione di "overload" ed un sistema che è vicino a tale situazione è detto essere in "stress". OpenGL Performer ha diverse armi per combattere il problema del frame rate tra cui il LOD e la multi elaborazione, argomenti che verranno discussi più avanti.

Performer calcola il "load" di sistema per ogni frame; il load è calcolato come la percentuale dell'intervallo di frame (l'inverso del frame rate) che serve per elaborare il frame. Poi il load viene usato per calcolare lo stress di sistema che a sua volta serve per sistemare il livello di dettaglio dei modelli visibili. La gestione del LOD è il metodo principale di Performer per gestire il load di sistema.

L'intervallo di frame è sempre un multiplo dell'intervallo di refresh (quanto tempo ci vuole all'hardware per disegnare un quadro video); quando il tempo di calcolo di un frame supera l'intervallo di frame bisogna prendere una decisione: ad esempio se presentare il frame all'istante di inizio del successivo intervallo di refresh, oppure se aspettare e presentarlo al primo istante multiplo dell'intervallo di frame. Con la funzione pfPhase() è possibile specificare quale azione intraprendere:

- "PFPHASE_FREE_RUN": dice all'applicazione di presentare ogni nuovo frame non appena è pronto (ovviamente sempre all'inizio di un intervallo di refresh).

all'interno del buffer un sottografo e finalmente chiamare la funzione pfMergeBuffer() per incorporare il sottografo nella scena dell'applicazione.

## 5. Conclusioni

All'interno di questo rapporto sono state presentate delle possibili tecnologie per lo sviluppo di applicazioni grafiche 3D in tempo reale in ambiente PC/Linux. Tali tecnologie potrebbero tornare utili per installazioni di realtà virtuale in enti dei beni culturali (come ad esempio i musei) in quanto a basso costo. Tuttavia la parola basso costo è relativa ed andrebbe quantificata meglio. Inoltre la soluzione hardware presentata non è detto che riesca a coprire tutte le esigenze, e l'uso del PC invece di una workstation potrebbe imporre dei grossi limiti nello sviluppo dell'applicazione.

## Bibliografia

[1] Sanzio Bassini, Silvia Monfardini, "Caratteristiche e prospettive del progetto di Virtual Theatre", articolo della rivista "notizie dal CINECA" n° 35-36/99

[2] Ping Dai, Gerhard Eckel, Martin Gobel, Frank Hasenbrink, Vali Lalioti, Uli Lechner, Johannes Strassner, Henrik Tramberend, Gerold Wesche, "Virtual Spaces: VR Projection System Technologies ad Applications", pubblicazione presa dal sito viswiz.gmd.de

[3] "NVIDIA nfiniteFX Engine: Programmable Vertex Shaders", "NVIDIA nfiniteFX Engine: Programmable Pixel Shaders", "GeForce3: Lightspeed Memory Architecture", "Realistic Shadows Using NVIDIA Shadow Buffer Technology", "NVIDIA's 3D Texture Technology", note tecniche prese dal sito www.nvidia.com

[4] George Eckel, Ken Jones, Tammy Domeier, "OpenGL Performer Getting Started Guide", scaricata dal sito www.sgi.com

[5] George Eckel, Ken Jones, "OpenGL Performer Programmer's Guide", scaricata dal sito www.sgi.com

[6] Henrik Tramberend, "Avango: A Distributed Virtual Reality Framework", articolo preso dal sito www.avango.de