

Tecniche per applicazioni interattive



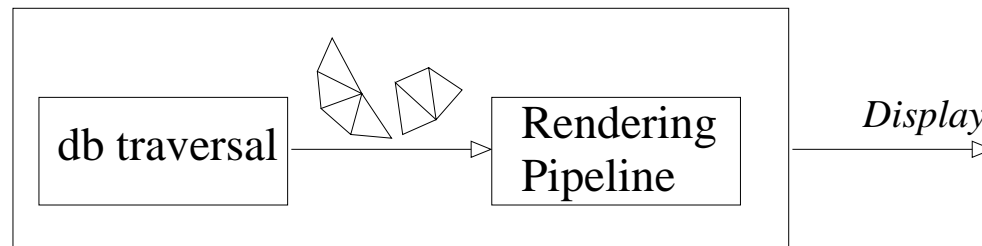
Dove si affronta il problema del rendering dalla parte della applicazione grafica interattiva.

- **Introduzione**
- **Rappresentazione della scena**
- **Tecniche di abbattimento della complessità**

Introduzione

- Possiamo vedere **l'applicazione grafica** come un *front-end* per il motore di rendering (pipeline), costruita per alimentare (di poligoni) quest'ultima.

Graphic application



- L'applicazione grafica ha, tra gli altri:
- il compito di costruire e visitare la struttura dati che rappresenta la scena, dove le correlazioni naturali tra le primitive sono rese esplicite (p.es. raggruppate in oggetti).
- il compito di inviare alla pipeline di rendering un carico di poligoni da disegnare adeguato alle capacità di quest'ultima.

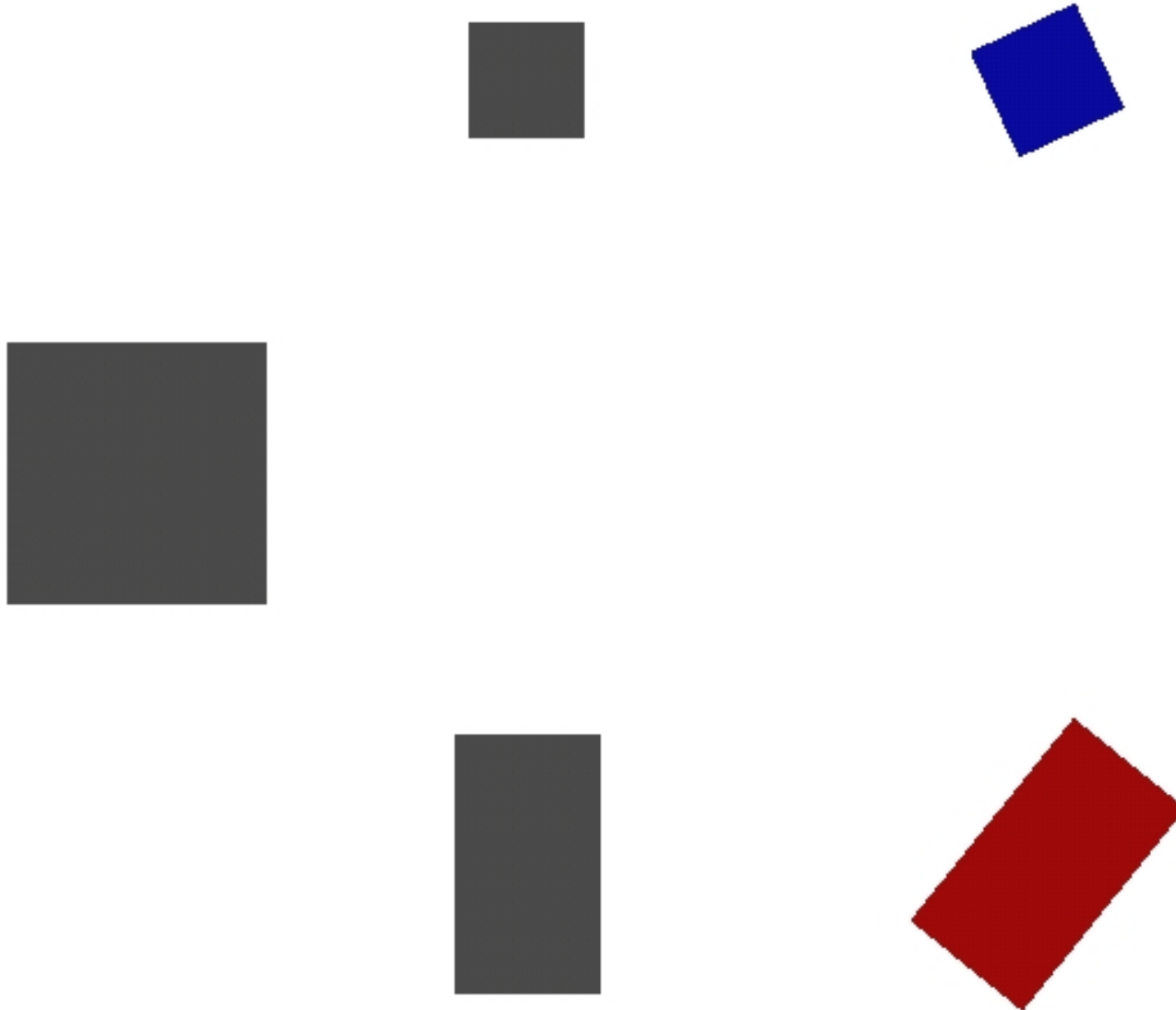
Rappresentazione della scena

- La struttura dati che contiene le primitive della scena è bene che ne renda esplicite le naturali **correlazioni**.
- La correlazione tra primitive può esprimersi sia tra i poligoni (**raggruppamento in oggetti**), sia per gli oggetti stessi (**raggruppamento degli oggetti**)
- Spesso tali raggruppamenti prendono la forma di **gerarchie**, utilizzando o alberi o grafi. Si avranno essenzialmente tre tipi di raggruppamenti
 1. **Gerarchie di poligoni:** in tal caso i poligoni della scena vengono distribuiti su una struttura gerarchica (ad albero in genere). L'uso è tipicamente nell'ottimizzazione di certe operazioni (ricerca geometrica) o per risolvere in maniera più agevole il **visibility culling** (rimozione delle facce che non sono visibili);
 2. **Gerarchie di oggetti:** in tal caso gli oggetti vengono distribuiti su un albero (o un DAG). Si usa per esplicitare qualche **relazione di contenimento** (la bottiglia nel frigo nella cucina nella casa nella città ...) oppure in **animazione** per agevolare il calcolo della postura di oggetti composti complessi.
 3. **Gerarchia della scena:** (scene graph) in tal caso la scena è descritta da una gerarchia che contiene sia gli oggetti da disegnare, che alcuni stati che determinano come disegnarli (trasformazioni, shading etc..). Per fare il renderig della scena basta visitare in modo opportuno tale struttura.

Oggetti Logici ed Istanze

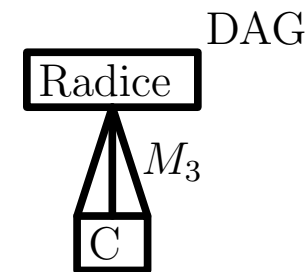
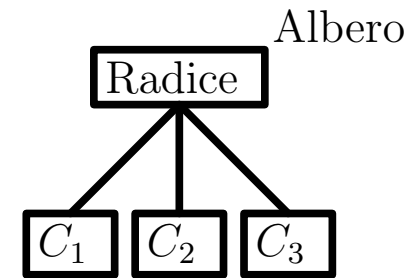
- Prima di cominciare è bene vedere brevemente una importante distinzione che viene spesso usata in linguaggi o applicazioni di grafica al calcolatore
- La distinzione è quella tra un **oggetto logico** ed una sua **istanza fisica**
- Con il primo si intende spesso un oggetto di dimensioni standar, posto in una posizione standard, con proprietà di shading (colore etc.) standard
- Con il secondo si intende una serie di operazioni (trasformazioni affini, coloritura) che, applicate all'oggetto standard, ne specificano una particolare configurazione
- Detto in altri termini l'oggetto logico è una astrazione di un oggetto, le istanze esplicitano tale astrazione nell'universo dell'applicazione grafica

- Ad esempio, si supponga di avere una applicazione che debba disegnare centinaia di cubi di varie dimensioni e posti in varie parti dello spazio con varie posture e colori
- È inutile definire una mesh poligonale per ciascun cubo
- Conviene invece definire una mesh C per un **cubo logico** (ad esempio un cubo di lato unitario, posto con il baricentro nell'origine ed orientato perpendicolarmente ai tre assi cartesiani)
- Ogni cubo da disegnare, una **istanza**, sarà specificato dando una matrice di trasformazione M_i , che conterrà una scalatura, una rotazione ed una traslazione tali da avere il cubo desiderato semplicemente applicando M_i a C (che indichiamo con $M_i(C)$)
- Bisogna stare attenti all'ordine in cui le trasformazioni affini entrano nella matrice M_i ; ritorneremo sull'argomento nel prossimo capitolo (pipeline grafica)



- I vantaggi sono almeno due
 1. Da un punto di vista dello **spazio occupato** dagli oggetti occupa di meno una matrice di trasformazione affine che non una mesh per ogni oggetto (cosa non evidente per il cubo)
 2. Se devo specificare a mano i cubi da disegnare, è più semplice trovare la matrice M_i che non la mesh $M_i(C)$
- Tale distinzione (logico vs istanza) si riflette nei pacchetti di grafica ed in alcuni linguaggi di rendering
- In genere quando si crea un oggetto primitivo con questi (ad esempio un cubo) tale oggetto viene creato nell'origine delle coordinate ed orientato in maniera standard
- L'utente cambia quindi il cubo scalandolo, ruotandolo o traslandolo nel modo desiderato
- Alternativamente (il caso di RenderMan, delle OpenGL e di VRML) il programma mantiene una **pila** (o **stack**) di trasformazioni e quando si richiama la funzione che crea il cubo logico (ad esempio), ad esso viene automaticamente applicata la prima trasformazione dello stack attuale

- Ritorniamo nel dettaglio su tale questione nel prossimo capitolo
- Qui basta notare come questa distinzione tra oggetti logici ed istanze possa influenzare le strutture gerarchiche che discuteremo
- Ad esempio si immagini di avere una struttura gerarchica che descriva l'applicazione precedente, ovvero un insieme elevato di cubi di varia forma, posizione e postura
- Se ciascun cubo viene visto come oggetto a se stante, tale gerarchia può ad esempio essere un albero, in cui le foglie contengono alcuni dei cubi
- D'altra parte se la distinzione tra logico ed istanza è esplicitata, si potrà mettere nelle foglie il cubo logico, ottenendo così un DAG
- I lati di tale DAG possono contenere la matrice M_i associata all'istanza
- In generale non è molto diverso lavorare con alberi o con DAG, è solo questione di gusti (e di efficienza); alle volte si usa una via di mezzo



Gerarchia per gli oggetti

- Iniziamo con alcune considerazioni generali
- Gli oggetti in una gerarchia possono essere in due tipi di relazione
 1. Gli oggetti della gerarchia sono **sottoparti** di un oggetto più complesso (ex. le dita che formano la mano)
 2. Gli oggetti sono separati, ma vi è una **relazione di contenimento** (ex. la bottiglia che è nel frigo)
- Il primo tipo di gerarchia viene usato tipicamente in animazioni poiché, come vedremo in un attimo, semplifica la gestione di un oggetto composto
- Il secondo caso viene spesso usato in termini di ottimizzazioni della pipeline grafica (se non devo disegnare il frigo, sicuramente non disegno nemmeno il suo contenuto)

Oggetti composti

- Si immagini di voler descrivere una macchina
- Si immagini che tale descrizione comprenda le quattro ruote, e la carrozzeria
- Posso immaginare di trattare questi cinque oggetti in modo del tutto indipendente
- In questo caso per specificare la posizione della macchina, io devo specificare cinque trasformazioni affini, una per la carrozzeria e quattro per le ruote; chiamiamole M_c ed M_i con $i = 1 \dots 4$
- Ogni volta devo verificare la **coerenza** della rappresentazione; siccome le ruote sono attaccate alla carrozzeria, non potranno essere messe ovunque
- Esiste dunque una relazione ben precisa tra le matrici
- La mia rappresentazione degli oggetti però non mi permette di utilizzare tale relazione e quindi sono costretto ogni volta a verificare che sia soddisfatto un certo vincolo

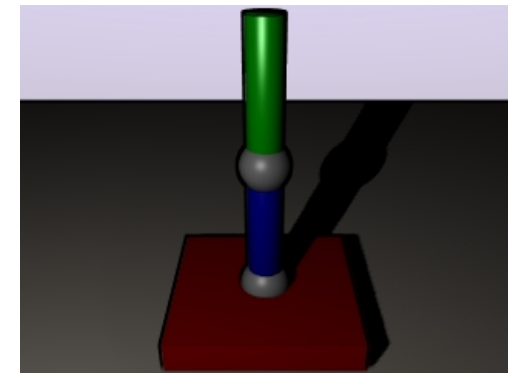
- In realtà il vincolo è molto semplice
- In un riferimento centrato nel baricentro della carrozzeria e tale che questa risulti orientata in modo semplice, ($M_c = I$), le quattro ruote saranno posizionate con quattro semplici traslazioni T_i (non tengo conto della rotazione di queste)
- In generale quindi si avrà $M_i = M_c T_i$
- Si vede che si può esplicitare questa relazione (e quindi evitare di fare un test di coerenza spaziale ogni volta) usando un albero
- La radice sarà la carrozzeria, con quattro figli che contengono le ruote
- La gerarchia significa che le ruote sono vincolate alla carrozzeria e quindi qualsiasi matrice affine trasformi questa, deve essere applicata anche alle ruote, dopo aver applicato le trasformazioni T_i
- Tra parentesi, essendo le ruote tutte uguali, questo è un caso in cui si possono usare istanze di un oggetto logico e quindi avere un DAG al posto di un albero.

- Del tutto in generale, una **gerarchia di oggetti** è definita da un **albero** in cui ogni nodo α contiene uno degli oggetti ed una matrice di trasformazione affine M_α , detta **matrice di trasformazione locale** di α
- Per sapere come trasformare l'oggetto α basta trovare l'attraversamento dell'albero dalla radice ad α
- Supponiamo che la sequenza di nodi $r, r_0 \cdots r_n, \alpha$ rappresenti tale attraversamento, che parte dalla radice r , attraversa i nodi da r_0 fino ad r_n ed arriva in α
- Allora la matrice di trasformazione da applicare all'oggetto in α per posizionarlo, o **matrice globale** di α , sarà data da

$$M_r M_{r_0} \cdots M_{r_n} M_\alpha$$

- Quando quindi cambiamo la posizione o la posa di un oggetto nella gerarchia, automaticamente tutti gli oggetti nel sottoalbero generato da tale oggetto vengono cambiati nello stesso modo
- Ruotando o traslando la radice, ad esempio, ruoto o traslo nella sua totalità l'oggetto composto che la gerarchia rappresenta
- Per disegnare la scena basta visitare l'albero; in generale si usa una previsiteda

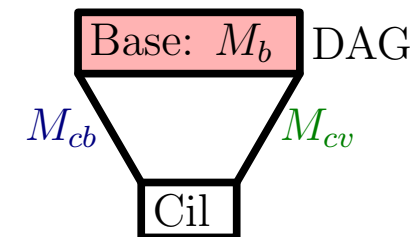
- Ad esempio, si consideri l'oggetto rappresentato in figura, costituito (con un po' di fantasia) da una base (rossa), un avanbraccio (blu) ed un braccio (verde)
- In questo caso la gerarchia è molto semplice e può essere rappresentata sia come albero che come DAG (trascuriamo le due giunture grigie)
- Se M_b , M_{cb} e M_{cv} sono le matrici locali dei tre elementi, per quanto visto fino ad ora il braccio verde avrà la matrice globale $M_b M_{cb} M_{cv}$, l'avambraccio la matrice $M_b M_{cb}$ e la base semplicemente M_b
- **Attenzione:** le rotazioni nelle matrici in generale non saranno rispetto ad un asse passante per l'origine (da dove passa?)

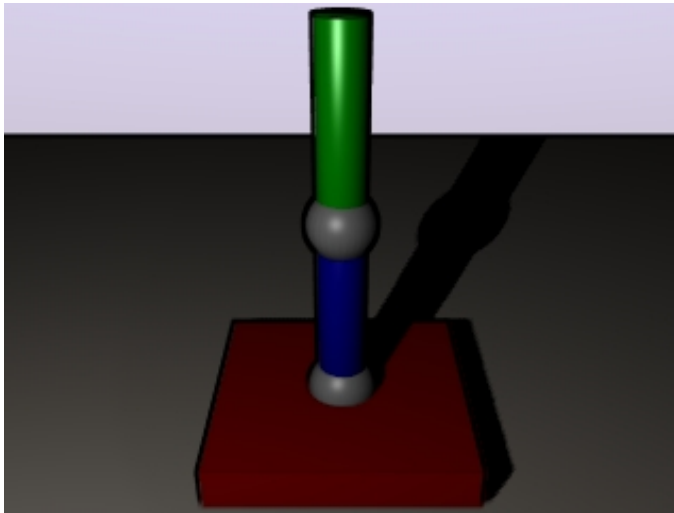


Base: M_b Albero

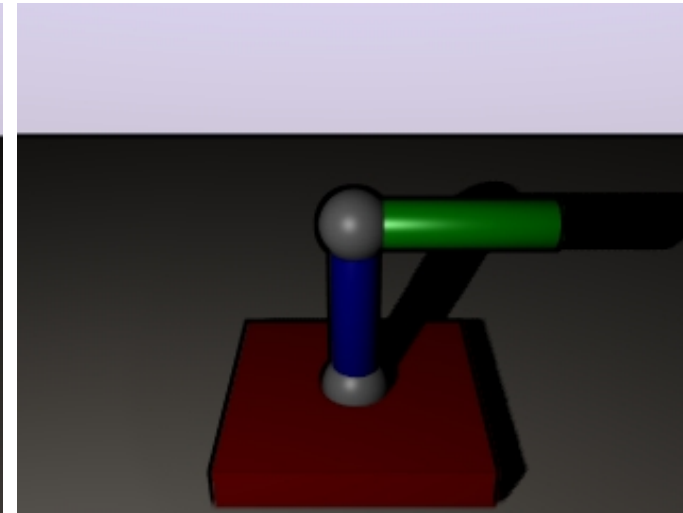
Cil-Blu: M_{cb}

Cil-Ver: M_{cv}

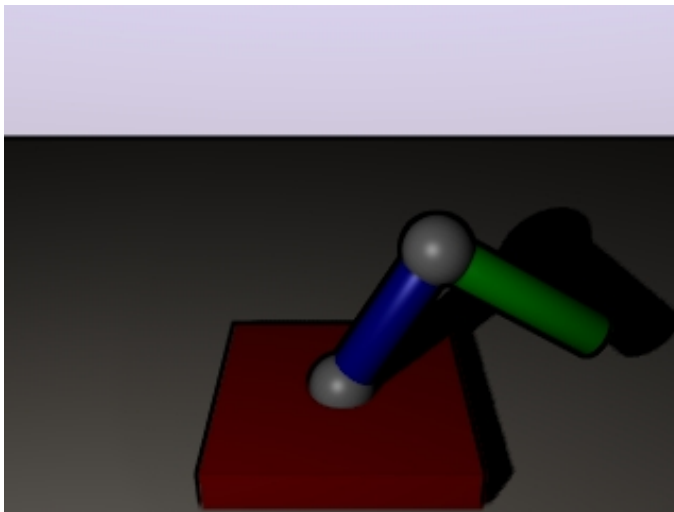




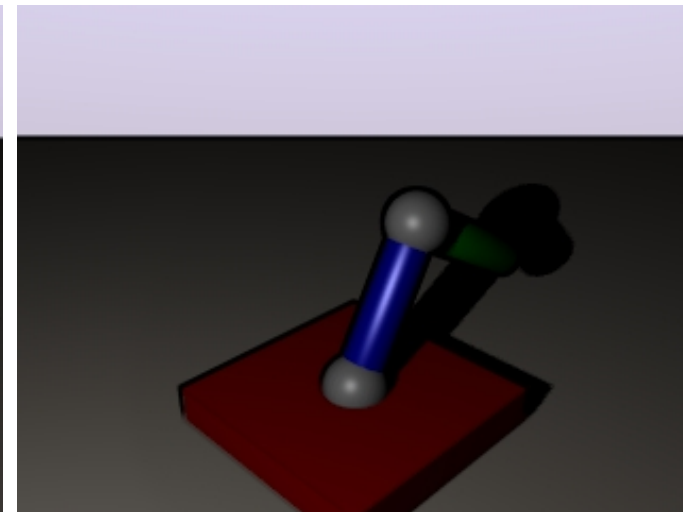
(1) Partenza



(2) $M_{cv} = T_{cv}^{-1} R_y(90) T_{cv}$



(3) $M_{cb} = T_{cb}^{-1} R_y(30) T_{cb}$



(4) $M_b = R_z(45)$

- Come si fa in pratica la visita di un albero di questo tipo?
- Vediamo un po' di pseudocodice
- Supponiamo di avere una funzione che disegni ciascun elemento (ad esempio un cilindro, un cubo etc) che chiamiamo genericamente `disegna` e che accetta come argomento una matrice 4×4
- Supponiamo poi date due funzioni per manipolare le matrici di trasformazione
- `Copia-Matrice(float *M1, float *M2)` copia la matrice contenuta nell'array (16 componenti) `M1` in `M2`
- `Moltiplica(float *M1, float *M2, float *M3)` che mette in `M3` il prodotto riga-colonna tra `M1` ed `M2`
- Ogni elemento della gerarchia deve contenere un **puntatore alla funzione adatta per disegnarlo**, una **matrice locale di trasformazione** ed un puntatore al **primo figlio** ed al **fratello destro** (supponendo di aver ordinato i fratelli da sinistra a destra)

```
typedef struct{
    float M[16];
    void (*disegna)(float *M);
    struct treenode *fratello_destro;
    struct treenode *primo_figlio;
} treenode;

void Attraversa(float *Mat, treenode *radice)
{
    float M_t[16];
    if (radice == NULL) return;
    Copia-Matrice(Mat, M_t); // Copia Mat in M_t
    Moltiplica(Mat,radice->M,Mat) // Mette in Mat radice->M*Mat
    radice->disegna(Mat);
    if (radice->primo_figlio != NULL)
        Attraversa(Mat, radice->primo_figlio);
    Copia_Matrice(M_t, Mat); // Ripristina Mat
    if (radice->fratello_destro != NULL)
        Attraversa(Mat, radice->fratello_destro);
}
```


Oggetti contenuti

- Abbiamo visto le gerarchie per descrivere **oggetti composti**
- Gerarchie del tutto analoghe, in termini di strutture dati ed algoritmi, vengono usate per esplicitare **relazioni di contenimento spaziale** di oggetti indipendenti
- Ad esempio se vi è un oggetto treno che contiene degli oggetti passeggeri, è chiaro che qualsiasi trasformazione avviene sul treno deve essere applicata anche ai passeggeri
- Si usa lo stesso principio di attraversamento di un albero
- Inoltre tale albero può essere usato per fare un visual-culling grossolano
- Se un oggetto non è contenuto nel campo di vista della camera virtuale non viene disegnato e non vengono disegnati (ovviamente) anche gli oggetti che esso contiene
- Dunque ad ogni nodo dell'albero gerarchico si può fare un test per capire se l'oggetto deve essere o meno disegnato; se il nodo viene scartato si può scartare tutto il sottoalbero che esso genera
- Non è in ogni caso un metodo dei più efficienti, ne vedremo altri

Tecniche di abbattimento della complessità

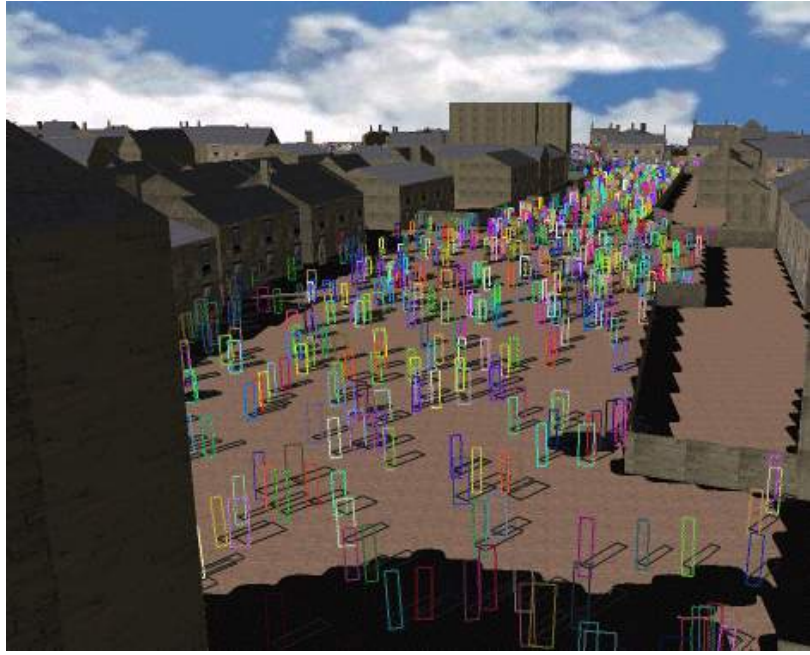
- Per soddisfare requisiti di interattività con scene/modelli complessi, l'applicazione grafica deve applicare tecniche efficienti per filtrare (ridurre) i dati, in modo da non sovraccaricare la rendering pipeline.
- **Semplificazione geometrica**: se l'oggetto è troppo complicato, rendilo più semplice!
- **Livello di dettaglio**: aggiustare il livello di dettaglio (\sim numero di poligoni) della maglia poligonale rispetto alla distanza del punto di vista, in modo da non processare inutilmente poligoni praticamente invisibili perché troppo piccoli.
- **Visibility culling**: filtrare e i poligoni che non dovranno essere disegnati perché non visibili.

Semplificazione geometrica

- Un modo banale per non sovraccaricare la rendering pipeline è gettare via un po' di triangoli semplificando i modelli.
- Sembra semplicistico ma può servire specialmente per modelli sovracampionati come quelli ottenuti da laser scanning.
- **Semplificazione di maglie poligonali.** (Vedi indietro).
- **Impostori planari.** La semplificazione estrema di un oggetto è ridurlo ad un piano (un rettangolo) su cui è mappata (come texture) una immagine pre-calcolata dell'oggetto.
- Utile per rappresentare oggetti visti da lontano.

Impostori planari o sprites

- Spesso le scene di cui bisogna fare il rendering risultano “spoglie” se non si aggiungono abbastanza particolari geometrici
- Abbiamo già discusso come l’aggiunta di particolari geometrici alla superficie di un oggetto spesso sia proibitiva ed abbiamo già visto come sia possibile utilizzare dei “trucchi” (texture mapping, bump mapping) di shading per evitare questo sovraccarico, pur riuscendo a raggiungere buoni livelli fotorealistici
- Un’altra occasione in cui è necessario ricorrere a tali trucchi è quando un **numero elevato** di oggetti geometricamente complicati devono essere inclusi nella scena.
- Esempio: un terreno anche molto realistico, perde completamente di credibilità se non si vedono alberi (ad esempio)
- Un albero è un oggetto piuttosto complesso da un punto di vista geometrico
- Se il terreno è visto da molto lontano allora è sufficiente applicare la giusta texture al terreno per dare l’impressione che questo sia ricoperto da foreste
- Se si è molto vicini, per esempio all’interno della foresta, allora in genere bisogna per forza inserire dei modelli geometrici di alberi.



© Slater et al.

- Il problema è estremamente complesso in una situazione intermedia, in cui siamo abbastanza lontani dal terreno da poter abbracciare con lo sguardo una porzione rilevante di foresta (magari centinaia o migliaia di alberi), ma troppo vicini perché una semplice texture sul terreno basti (si vede che è piatto)
- Altro esempio: in una esplosione vi sono svariate decine di pezzi che volano da tutte le parti (tipicamente in fiamme)
- Se l'applicazione deve introdurre una loro descrizione geometrica, il carico di calcolo sarebbe proibitivo, soprattutto se vi è la possibilità di più esplosioni in sequenza
- Altro esempio: si supponga di voler fare il rendering di una folla di persone. Il problema è del tutto analogo a quello degli alberi discusso sopra, con la differenza che le persone si possono anche muovere
- In queste situazioni si introducono i cosiddetti **impostori planari** o **sprite**.

- Essenzialmente si tratta di rappresentare l'oggetto da inserire nella scena come una immagine
- Tale immagine viene usata come texture per un poligono (in genere un rettangolo)
- In genere l'immagine ha un canale α di trasparenza tale che solo l'oggetto che si vuole rappresentare risulti visibile, ovvero non si vede il bordo della texture
- Se si posiziona tale poligono in modo opportuno, l'oggetto sembrerà inserito geometricamente nella scena, non semplicemente una immagine 2D
- Per ogni istanza basta aggiungere, da un punto di vista della geometria, un semplice poligono. Si possono così aggiungere centinaia di elementi senza sovraccaricare il motore di rendering grafico.
- Come va posizionato il poligono?
- Dipende dall'applicazione

- In genere, se la telecamera si può spostare liberamente nella scena, bisogna far si che il poligono non venga mai visto di taglio (altrimenti risulta palese la sua natura bidimensionale)
- La tecnica più nota va sotto il nome di **billboarding**
- Nella pratica il poligono viene ruotato intorno ad un suo asse per far si che la faccia del poligono punti sempre in direzione della camera
- Se \mathbf{n} è la normale al poligono e \mathbf{v} è il vettore di vista, ovvero il vettore che specifica la direzione dalla camera al billboard (entrambi i vettori si suppongono normalizzati), allora bisognerà fare in modo che $\mathbf{n} \cdot \mathbf{v} = -1$
- Se l'impostore possiede, grosso modo, una simmetria sferica, allora sarà possibile ruotarlo per soddisfare la precedente condizione
- Viceversa se l'oggetto da simulare ha simmetrie più restrittive (ad esempio cilindrica nel caso di un albero) allora la tecnica ha senso solo se la telecamera è vincolata a stare su un piano perpendicolare all'impostore, o comunque molto vicino a tale piano

- Inoltre muovendo la telecamera si può notare come l'oggetto mostri sempre la stessa faccia all'osservatore, il che può risultare strano
- In presenza di molti impostori ed in scene viste molto velocemente comunque non vi si fa caso.
- Una variante prevede di cambiare immagine a seconda dell'angolo di vista (tipico il caso di 8 immagini)
- L'esempio degli alberi è la più nota applicazione del billboard. Se la telecamera è vincolata a rimanere vicina al terreno e a puntare in direzione parallela a questo, allora gli alberi possono rappresentarsi efficacemente tramite billboard
- Una variante, meno efficace, è quella di costruire l'albero come due poligoni intersecantesi posti perpendicolari e proiettare l'immagine su entrambi i poligoni; in questo caso, per ottenere l'effetto 3D non c'è bisogno di ruotare l'impostore (è quindi più semplice da realizzare)
- Un altro esempio tipico di billboard è la simulazione delle **lens flare**, ovvero le riflessioni interne tipiche di alcuni obiettivi di telecamere o gli effetti di rifrazione (molto di moda, ma poco significativi)

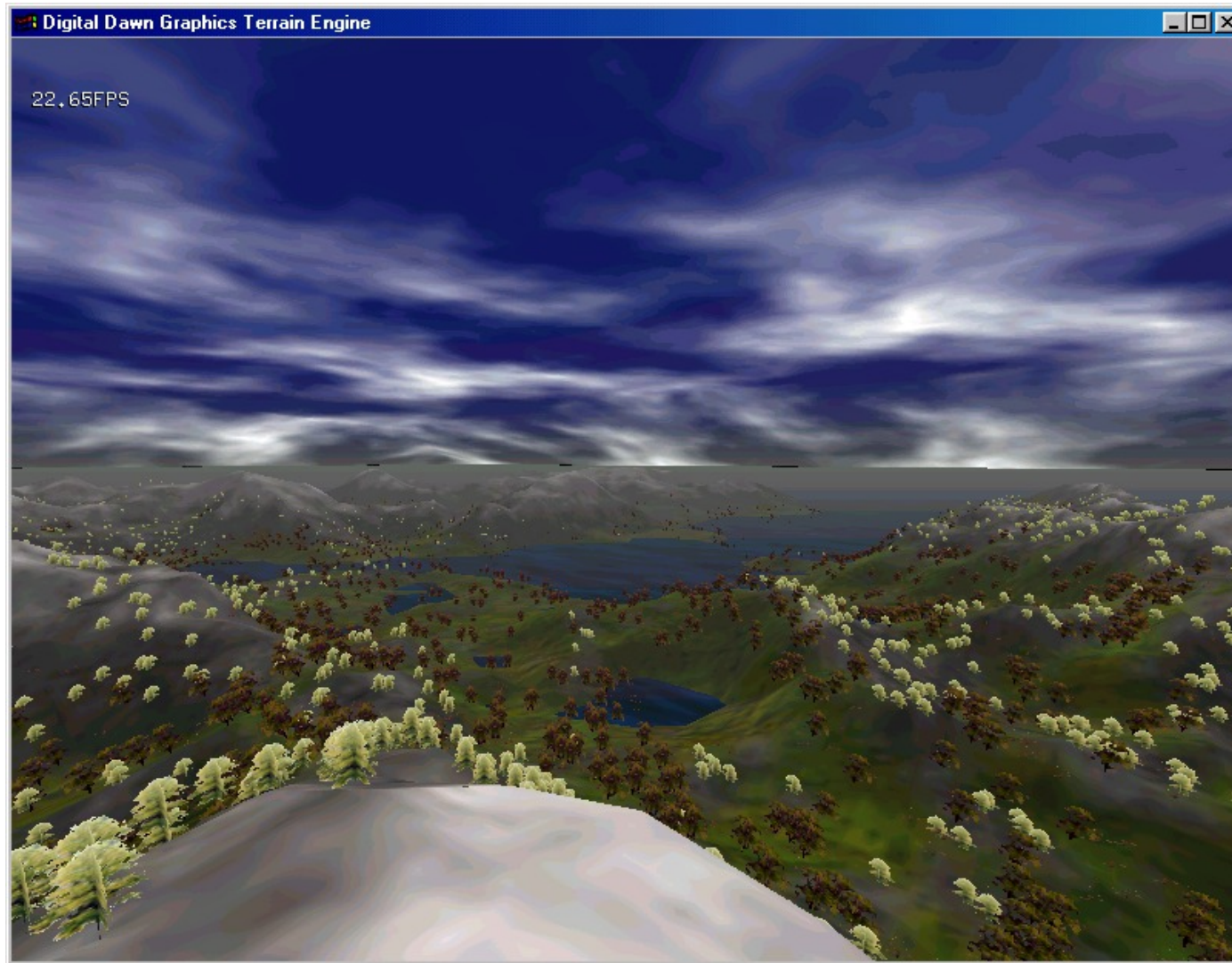


Figura 1: Esempio di billboard per rendering di alberi su terreno

© Alex Pfaffe

- Un altro uso tipico dei billboard è la simulazione di esplosioni e, più in generale, per effetti che coinvolgano numerose **particelle**
- Ciascuna particella viene rappresentata da un poligono con texture opportuna e viene spostata nella scena avendo sempre cura che la faccia del poligono sia sempre rivolta in direzione dell'osservatore.
- Si possono usare sistemi di particelle siffatti per simulare, ad esempio, effetti atmosferici (neve, pioggia, etc)

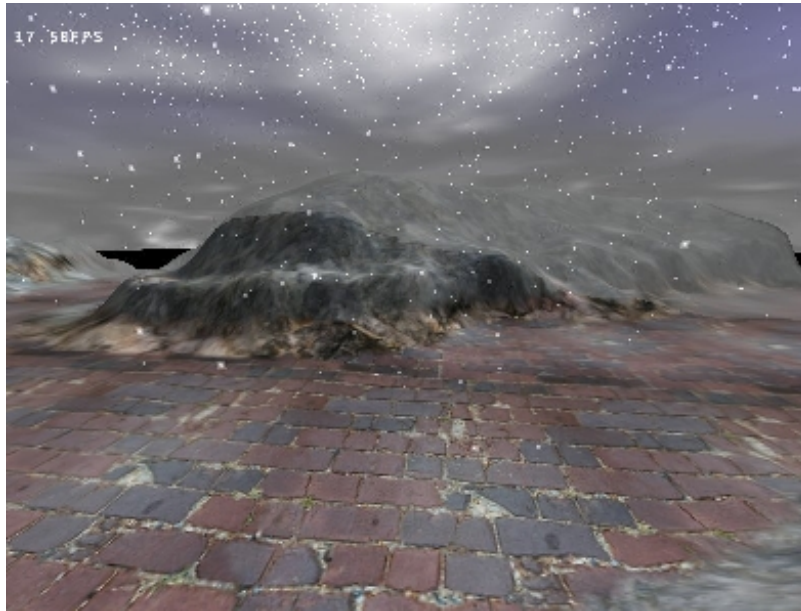


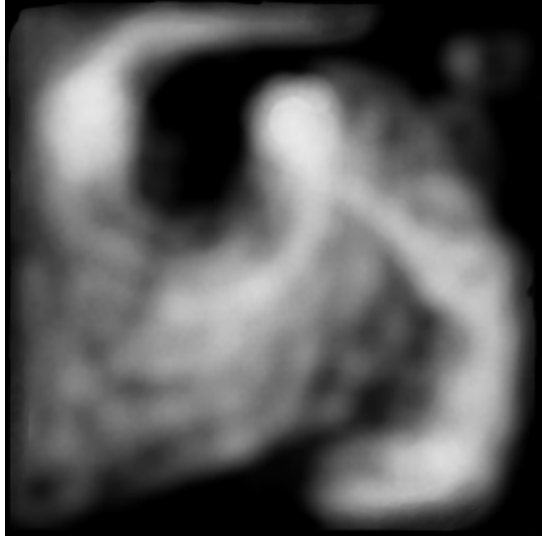
Figura 2: Esempio di billboard per effetti di particelle

© Alex Pfaffe

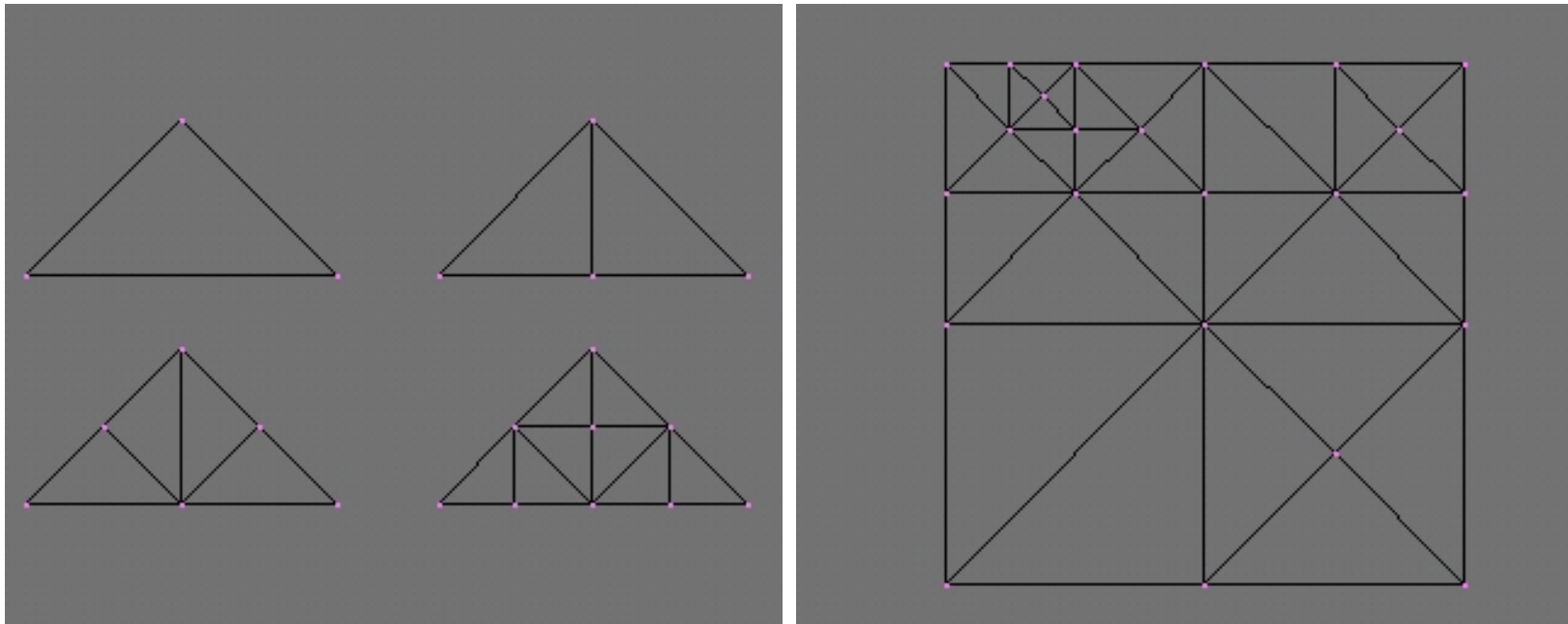
Livello di dettaglio (LOD)

- Per descrivere un oggetto complesso con una mesh mantenendo un sufficiente dettaglio geometrico sono necessari tipicamente molti poligoni. Altrimenti vi sarà una **mancanza di dettaglio** quando l'oggetto viene analizzato da vicino.
- Viceversa vi sarà uno **spreco di risorse** quando l'oggetto è lontano dal punto di vista (viene proiettato su pochi pixel dell'immagine).
- Una soluzione è quella di rappresentare l'oggetto con una mesh diversa a seconda della distanza dall'osservatore; tipicamente se ne usano tre o quattro.
- Questo approccio però non è efficiente quando si ha una variazione del punto di vista continua (la telecamera si sposta) e l'oggetto da rappresentare è molto grande rispetto al campo di vista (es. terreno). In tal caso alcune parti dell'oggetto possono risultare vicine ed alcune lontane.
- È necessario introdurre strutture dati **multirisoluzione** che consentano di modellare la scena in modo che la sua rappresentazione (triangoli) possa variare in maniera continua e dinamica.
- Vedremo l'algoritmo **ROAM**, basato sui **binary triangle tree**.

Real-time Optimally Adapting Meshes

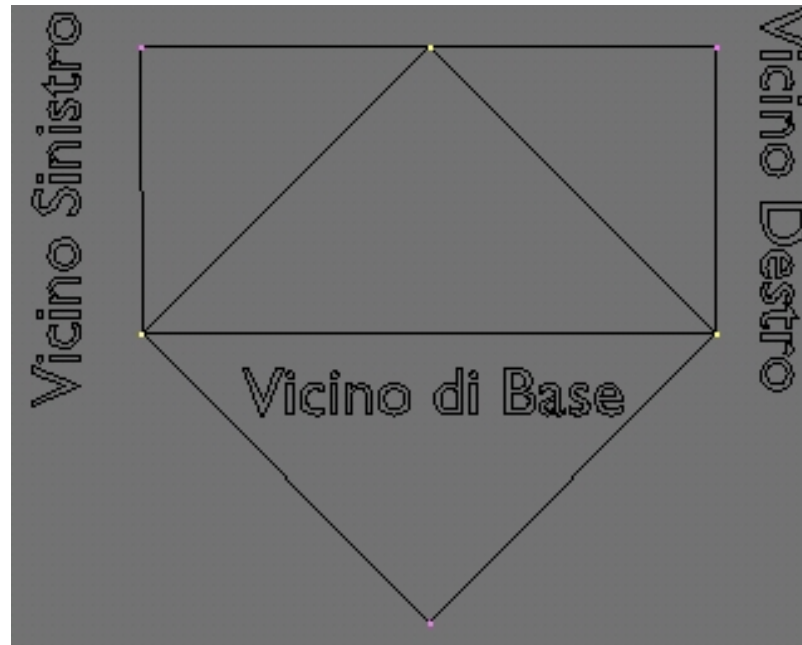
- Si immagini di voler costruire una tessellazione con triangoli che approssimi un certo **heightfield**
 - Un heightfield è semplicemente una matrice (di solito quadrata, ma non è necessario) i cui elementi (numeri interi di solito) vengono interpretati come altezze su un livello di riferimento
- 
- La triangolazione più semplice è data dal rappresentare il terreno da una griglia quadrata i cui vertici siano proprio alle altezze date dalla mappa (quindi la griglia ha le stesse dimensioni dell'array), e poi suddividere ciascun quadrilatero così ottenuto in due triangoli con una diagonale
 - Sebbene molto veloce da fare, questa mesh non è dinamica, ma statica.
 - L'algoritmo **ROAM** usa un **binary triangle tree** per ottenere una triangolazione dinamica che implementi un livello di dettaglio continuo.

- I **binary triangle tree** sono una struttura dati molto simile al quadtree.
- Si parte dalla considerazione geometrica che è possibile dividere un triangolo rettangolo in due triangoli rettangoli
- Iterando tale suddivisione si ha una gerarchia di triangoli, in modo del tutto simile alla gerarchia di quadrati del quadtree



- L'albero associabile a tale struttura è binario (a differenza del quadtree)
- Come nel quadtree la suddivisione può non essere omogenea, alcuni triangoli possono essere divisi, altri no

- Come vedremo è una struttura ideale per rappresentare una mesh di triangoli con una risoluzione non omogenea
- Tipicamente per ogni triangolo si tengono dei puntatori ai due figli (destro e sinistro) ed ai tre triangoli adiacenti dello stesso livello, ovvero il triangolo **vicino di base** (quello che incide sull'ipotenusa) al triangolo vicino destro ed a quello sinistro



- Torniamo al ROAM.
- Supponiamo per semplicità che l'heightfield sia quadrato ($N \times N$)
- Si parte con il creare quattro vertici nei quattro angoli della mappa e nel dividere tale quadrato con una diagonale
- A questo punto si potranno suddividere i due triangoli risultati con il metodo di suddivisione visto per i binary triangle tree
- La divisione può procedere su profondità elevate dove sia richiesto un livello di dettaglio alto, mentre può fermarsi a basse profondità (nel senso dell'albero) in zone in cui non sia richiesto un alto dettaglio
- Un commento prima di proseguire: la triangolazione nello spazio 3D sarà composta, in generale, da triangoli non rettangoli; per quanto riguarda le operazioni sul triangle binary tree, si immaginerà sempre questo proiettato sul piano $z = 0$, in tal modo i suoi triangoli sono rettangoli ed ha senso parlare di ipotenusa di un triangolo in modo univoco

- Ma come si decide quando suddividere un triangolo e quando non suddividerlo?
- Bisogna avere una funzione errore (detta alle volte **metrica**) che ci dica quanto la superficie triangolata è lontana dalla superficie definita dall'heightfield in un dato triangolo
- Discutiamo una scelta della metrica semplice, ma efficace
- Dato un triangolo dell'albero, definiamo la sua **varianza** come la differenza tra il valore dell'heightfield nel centro dell'ipotenusa ed il valore medio dell'heightfield sui due vertici del triangolo incidenti sull'ipotenusa

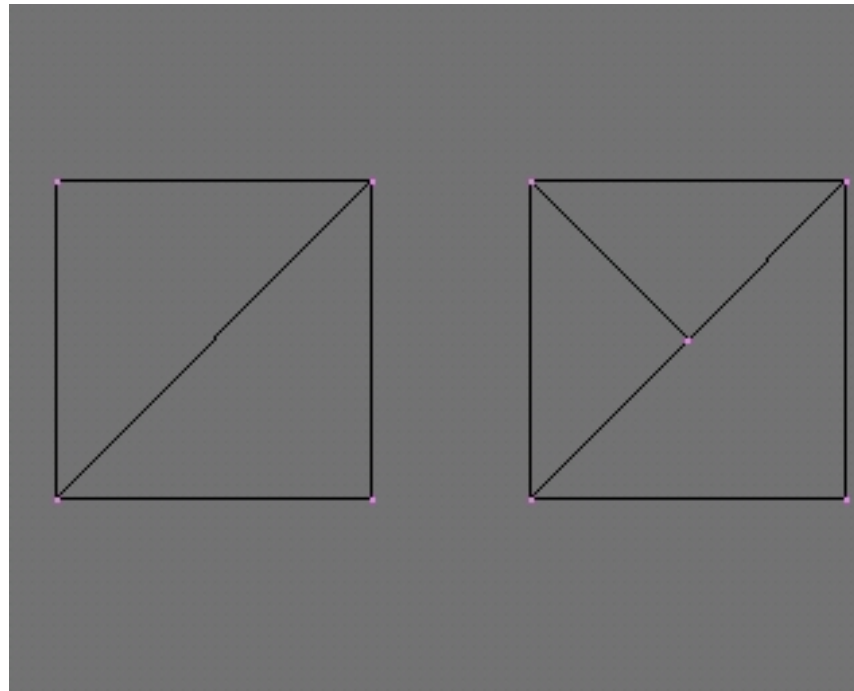
$$var_t = |h(t_i) - 1/2(h(t_2) + h(t_1))|$$

dove t indica il triangolo, t_i il centro dell'ipotenusa del triangolo e t_2 e t_1 i due vertici di t incidenti sull'ipotenusa ed h è la funzione che interpola i valori dell'heightfield per ogni punto del piano xy

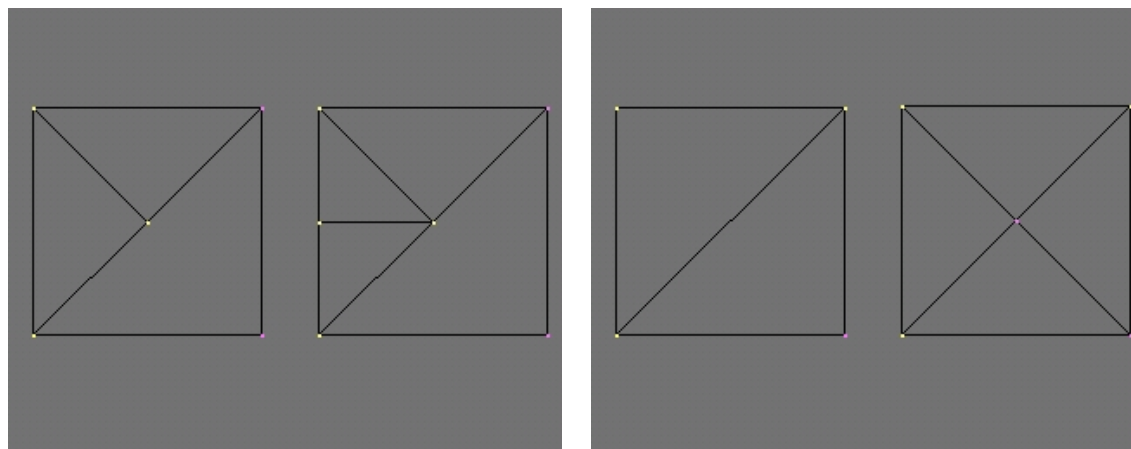
- Ovviamente il punto t_i sul piano xy è dato da $t_i = 1/2(t_1 + t_2)$

- A questo punto si può procedere con l'iterazione: triangoli con varianza superiore ad una certa soglia vengono suddivisi e si itera il test sui figli così ottenuti
- La suddivisione avviene fino a quando tutti i triangoli della mesh hanno varianza inferiore ad una soglia
- Se si cambia il valore di tale soglia a seconda del punto di vista (più alta quando ci si allontana e viceversa), allora si sarà più tolleranti nelle zone lontane dalla telecamera
- L'effetto netto è di avere zone con molto dettaglio e vicine alla telecamera tessellate in modo fitto, zone lontane e con pochi dettagli tessellate in modo sparso
- Allo spostarsi della telecamera cambiano le soglie e quindi la triangolazione si adatta al punto di vista

- Rimane un ultimo punto da discutere
- La suddivisione ricorsiva non deve introdurre inconsistenze nella triangolazione (crack o giunzioni a T)
- Per esempio nella seguente figura, la suddivisione del triangolo superiore e non di quello inferiore introduce una inconsistenza nella mesh risultante (quello inferiore è un quadrilatero)

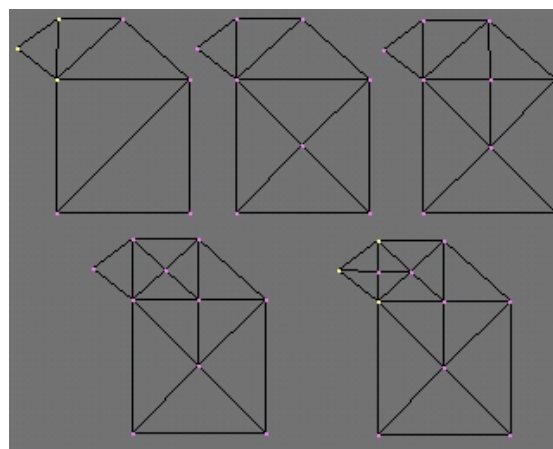


- Risulta utile, per risolvere il problema, classificare i triangoli in questo modo
 1. **Triangoli che hanno l'ipotenusa sul bordo;** in tal caso la loro suddivisione può essere fatta senza alcun problema
 2. **Triangoli appartenenti ad un diamante;** si definiscono così quei triangoli il cui vicino di base sia allo stesso livello di suddivisione. Se un triangolo appartenente ad un diamante deve essere suddiviso, allora anche il suo vicino di base viene suddiviso
 3. **Gli altri;** in tal caso il vicino di base si trova ad un livello inferiore di suddivisione e si opera un algoritmo ricorsivo: si scende lungo la catena dei vicini di base fino a quando non si trova un triangolo appartenente ad un diamante. A quel punto si suddivide tale triangolo (ed il suo vicino di base) e si risale la catena suddividendo i vari triangoli, fino a tornare al triangolo di partenza (operazione che prende il nome di **forced split**)



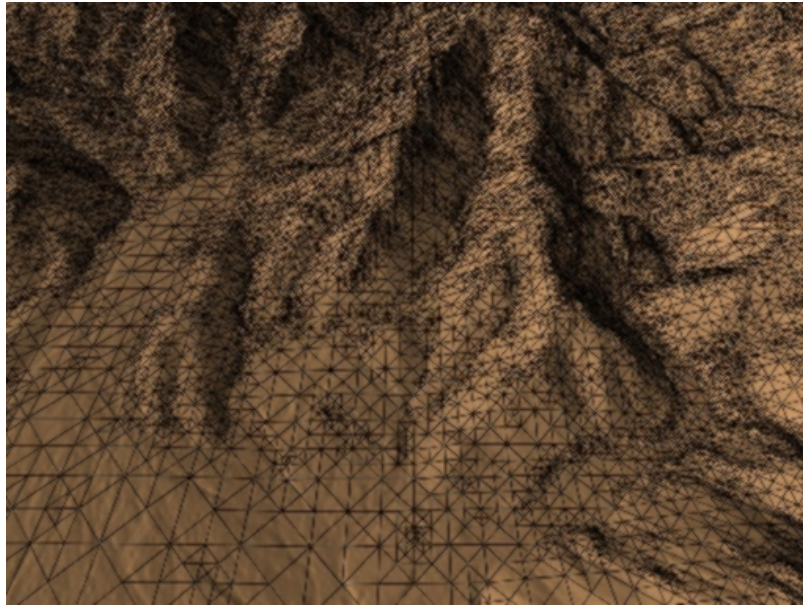
(1) Triangolo di bordo

(2) Diamante

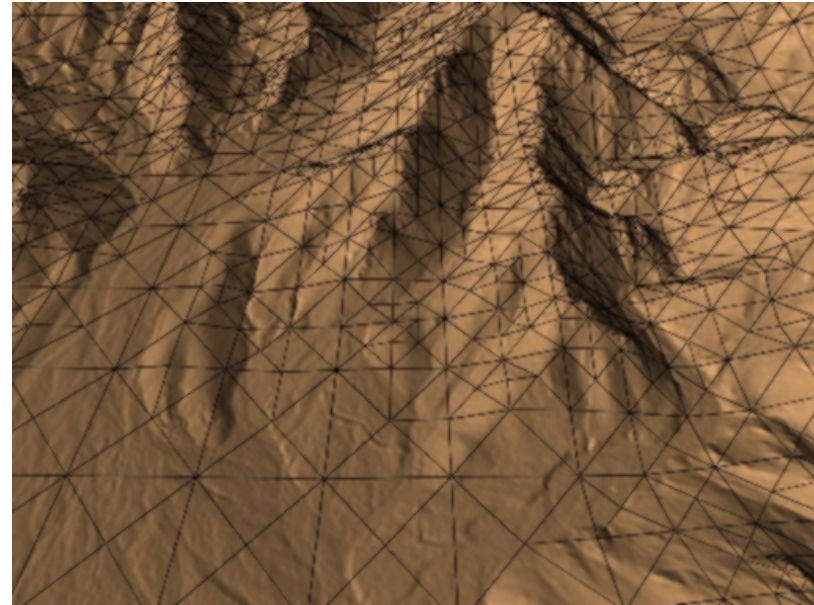


(3) Ricorsione

Ecco qualche esempio concreto.



(4) 62947 poligoni



(5) 3385 poligoni

Le immagini sono © di Peter Lindstrom, David Koller, William Ribarsky Larry F. Hodges, Nick Faust, Gregory A. Turner

Visibility culling

- Per quanto la determinazione della visibilità sia un problema oramai risolto in hardware con lo z-buffering, anche la scheda più veloce non può gestire una intera scena fatta da milioni di poligoni, garantendo tempi di risposta **interattivi**.
- D'altra parte spesso le scene sono densamente occluse oppure sono troppo grandi per essere esaminate da un solo punto di vista. Questo implica che in molte situazioni solo una frazione della scena è effettivamente visibile.
- Quindi, l'approccio naive (che già abbiamo criticato) di spedire tutti i poligoni della scena alla rendering pipeline 1) non è praticabile e 2) non è intelligente.
- Non ci si può affidare completamente ad algoritmi di visibilità (VSD) che esaminano ciascuna primitiva (o poligono) della scena per risolverne la visibilità (costo lineare nel numero totale delle primitive della scena), come fa la rendering pipeline.
- La complessità dell'intera scena non deve avere un impatto significativo sul tempo di elaborazione richiesto per disegnare (render) una singola immagine.
- Piuttosto questo deve solo dipendere dalla complessità della **porzione** di scena che è effettivamente visibile.

- Algoritmi con questa proprietà si chiamano algoritmi di visibilità **output sensitive** o algoritmi di **visibility culling** (VC).
- Gli algoritmi di VC operano uno sfoltoimento, non lavorano al livello del singolo poligono ma di gruppi di poligoni o di interi oggetti.
- Gli algoritmi di VC devono essere **conservativi**, ovvero non scartare mai un poligono che dovrebbe essere visualizzato.
- Gli algoritmi di VC devono essere **efficienti**, nel senso che devono scartare un gruppo di poligoni ad un costo inferiore alla scansione di ciascun poligono (altrimenti il lavoro lo poteva fare la rendering pipeline).
- In generale, l'idea del VC è cercare di stabilire una approssimazione per eccesso (più stretta possibile) dell'insieme dei poligoni visibili (visible set), scartando "all'ingrosso" i poligoni che non contribuiscono alla immagine corrente, ovvero che non devono essere disegnati, perché non visibili (occlusi o esterni al view frustum).
- Si dovrà pre-elaborare (off-line) la scena, costruendo strutture dati opportune (tip. gerarchiche) che possano consentire di scartare i poligoni non visibili in tempo sub-lineare in fase di rendering.

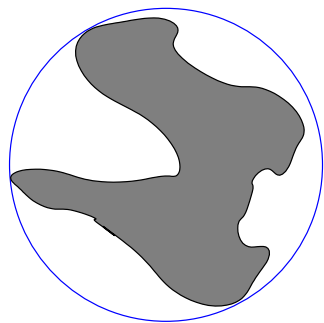
- Tre tipici algoritmi di VC sono:
 - **View frustum culling**: scarto di oggetti esterni a view frustum;
 - **Occlusion culling**: scarto di oggetti nascosti all'osservatore;
 - **Back-face culling**: scarto di gruppi di poligoni che rivolgono la faccia posteriore all'osservatore

View frustum culling

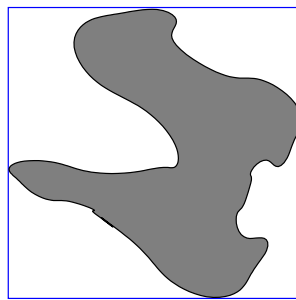
- Il **view frustum culling** consiste nello scartare in modo efficiente e conservativo i poligoni (o gli oggetti) che non intersecano il volume di vista.
- Serve ad alleggerire il **clipping**, ma non va confuso con quest'ultimo.
- È riconducibile al problema di rilevare in modo efficiente le intersezioni tra oggetti, in cui un oggetto è il view frustum.
- Anche il **ray casting** (e ray tracing) si riconduce al problema (analogo) di rilevare intersezioni oggetto-raggio (e le tecniche sono simili).
- **Nota:** abbiamo accennato al problema delle **collisioni** tra oggetti (*collision detection*) che è importantissimo in scene dinamiche.
- Non tenendo conto della coerenza temporale del moto, il problema delle collisioni si riduce a quello delle intersezioni. In generale è più complesso, tuttavia.

Bounding volume

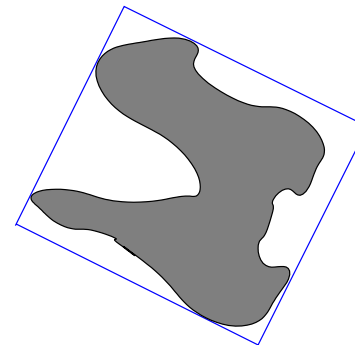
- Una prima tecnica per “sfoltire” le intersezioni consiste nel racchiudere gli oggetti in volumi che li contengono, con i quali sia facile testare l’intersezione: se non c’è intersezione con il bounding volume non c’è intersezione con l’oggetto racchiuso.
- Questo non rende sub-lineare la complessità ma semplifica le operazioni, e dunque sortisce nella pratica un miglioramento dei tempi.
- Tipici bounding volumes sono sfere, parallelepipedi con i lati paralleli agli assi cartesiani (AABB, da axis aligned bounding box), oppure parallelepipedi generici (OBB, da oriented bounding box)



Sphere



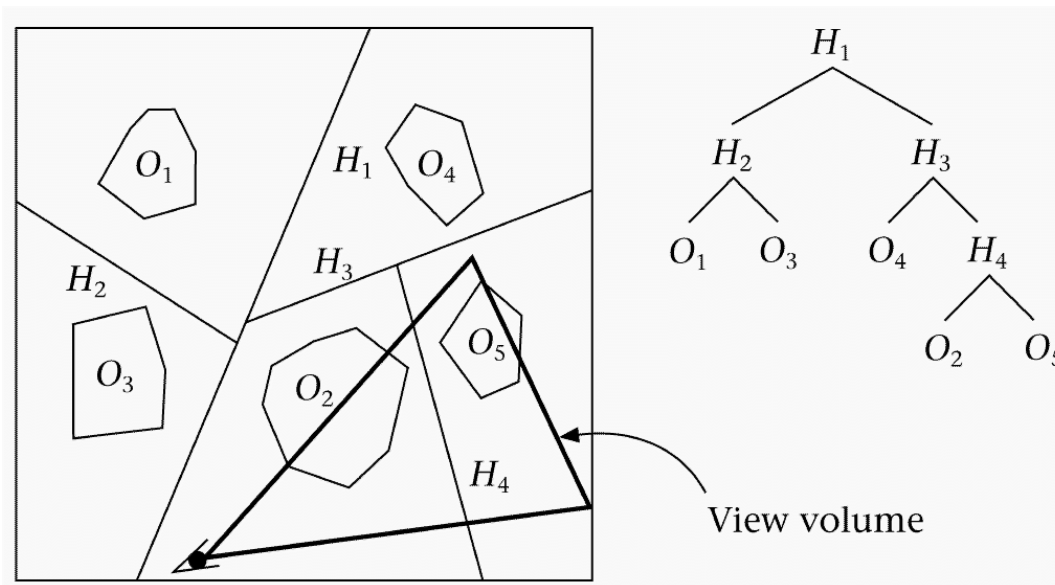
AABB



OBB

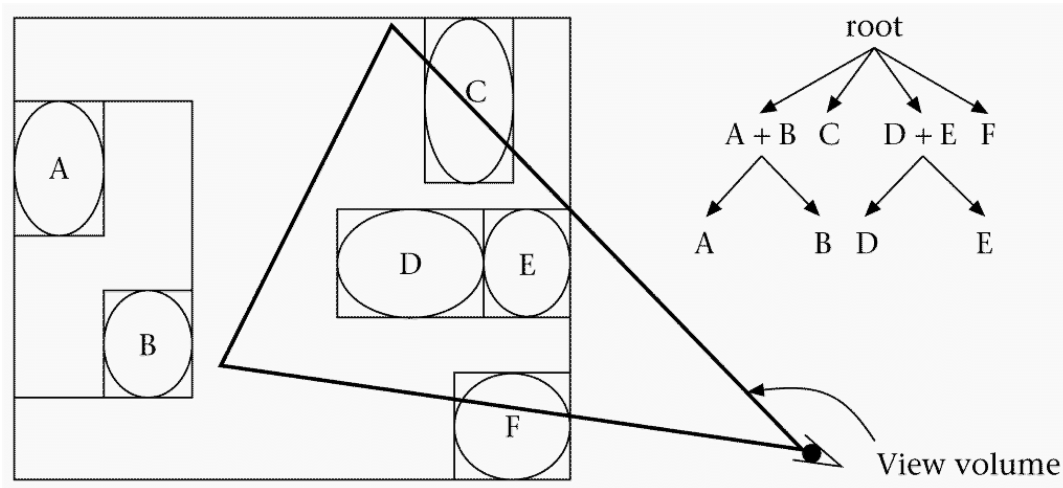
Hierarchical Space partitioning

- La tecnica sfrutta una partizione gerarchica dello spazio in regioni, come quelle offerte da quad-tree, k-d tree e BSP tree.
- BSP tree per View Frustum Culling (VFC). Si visita l'albero ed ad ogni nodo si controlla se il frustum non interseca il piano associato al nodo (sostituendo le coordinate dei 5 vertici del frustum nell'equazione del piano i segni devono essere concordi). Se è così l'intero sottoalbero associato al semispazio che non contiene il frustum può essere eliminato.



Hierarchical bounding volumes

- Si costruiscono gerarchie di bounding volumes, dove al livello più alto si ha un volume che racchiude tutta la scena, ed al livello più basso si hanno bounding volumes per i singoli oggetti.



- Come per il ray tracing.

Occlusion culling

- L'**Occlusion culling** serve a scartare in modo efficiente e conservativo poligoni (o oggetti) non visibili perché nascosti (occlusi) all'osservatore da altri oggetti.
- Si consideri per esempio un interno di edificio: le stanze diverse da quella dove si trova la telecamera non sono visibili, salvo quelle (poche, rispetto al totale) che lo sono attraverso porte o finestre.
- Serve ad alleggerire il VSD, ma non va confuso con quest'ultimo.
- Soluzione *cell-based*: lo spazio viene diviso in **celle** (stanze) che comunicano attraverso **portali**.
- L'idea di fondo è che se non si vede un portale, non si vedono nemmeno le celle dietro ad esso.
- Se il portale invece è in vista, bisogna determinare quali altre celle *possono* essere visibili attraverso di esso, ovvero si vuole calcolare il **Potentially Visible Set** (PVS). Le celle che rientrano nel PVS vengono disegnate.
- Il PVS si può calcolare staticamente (off-line) o dinamicamente (on-line).

Rimozione delle superfici nascoste

- Non tutti i poligoni sopravvissuti, però, devono essere disegnati. Alcuni possono non essere visibili dall'osservatore perché nascosti (totalmente o parzialmente) da altri poligoni.
- Problema: dati un insieme di poligoni in 3D ed un punto di vista, si vogliono disegnare solo i poligoni visibili (o porzioni di essi). Ogni poligono si assume essere piatto ed opaco.
- Vi sono essenzialmente due approcci:
 - **object-precision**: l'algoritmo lavora sui poligoni stabilendo relazioni di occlusione reciproca. Il costo è quadratico nel numero dei poligoni. Però la precisione è elevata (precisione macchina).
 - **image-precision**: l'algoritmo stabilisce occlusioni a livello del pixel. È più veloce ma la precisione è limitata.
- La rimozione delle superfici nascoste (*Hidden Surface Removal*, HSR) viene anche indicata come **determinazione delle superfici visibili** (*Visible Surface Determination*, VSD).

Back-face culling

- Non è un algoritmo generale di HSR in senso stretto, ma solo una tecnica (object space) utile per eliminare poligoni ovviamente invisibili.
- L'eliminazione delle facce posteriori (o **back-face culling**), elimina i poligoni che, a causa della loro orientazione, non possono essere visti.
- Viene effettuato nello spazio vista.
- Se \mathbf{v} è la direzione di vista (punta verso l'osservatore) ed \mathbf{n} è la normale al poligono, è facile rendersi conto che il poligono è visibile solo se:

$$\mathbf{n} \cdot \mathbf{v} > 0$$

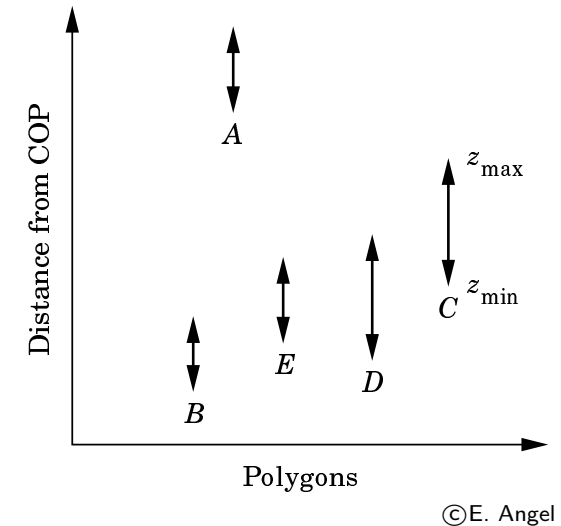
- **Nota:** se la scena è composta da un solo solido convesso, il culling risolve anche il problema della eliminazione delle facce posteriori.

List-priority

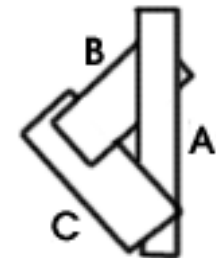
- Gli algoritmi **list-priority** determinano un ordinamento per i poligoni che garantisce che l'immagine che si ottiene è corretta se i poligoni vengono disegnati nell'ordine assegnato.
- Operano in object-precision (ovvero a livello di poligono), nello spazio 3D screen.
- Questa tecnica non scarta i triangoli ma li ordina.
- Infatti, risolve un problema di visibilità, ma implica il rendering back-to-front di *tutti* i triangoli.
- Può costituire la fase di HSR dentro la rendering pipeline, oppure può essere a carico della applicazione, nel qual caso si assume che l'HSR della pipeline sia "spento" e che i triangoli vengano disegnati nell'ordine in cui vengono spediti.
- Ordinare i poligoni può servire anche per ridurre l'overdraw con lo z-bufer attivo, in tal caso procedo front-to-back.

Depth-sort: algoritmo del pittore

- **Algoritmo del pittore** (o **Depth-sort**):
 - i poligoni vengono ordinati per profondità (o pseudo-profondità) decrescente.
 - si effettua il rendering dei poligoni della lista, dal più lontano al più vicino (*back-to-front rendering*).
 - In questo modo i poligoni più lontani vengono sovrascritti da quelli più vicini.



- Problema 1: se gli z-intervalli dei due poligono non sono disgiunti l'algoritmo non si applica (cos'è la profondità' a del poligono?); bisogna controllare altre condizioni per stabilire se uno dei due può essere disegnato prima dell'altro.
- Problema 2: nel caso ciclico è necessario spezzare i poligoni in parti.
- L'algoritmo di **Newell** risolve correttamente questi problemi.



Depth-sort con Newell

- Ordina i poligoni in base al vertice di massima distanza dall'osservatore.
- I poligoni i cui z-intervalli non si sovrappongono possono essere disegnati (in ordine back-to-front) come nell'algoritmo del pittore.
- Dati due poligoni P e Q i cui z-intervalli si sovrappongono, è corretto disegnare P prima di Q solo se almeno uno di questi test è vero:
 1. gli x-intervalli di P e Q sono disgiunte (veloce);
 2. gli y-intervalli di P e Q sono disgiunte (veloce);
 3. Si consideri il piano contenente Q. P è contenuto completamente nel semispazio opposto a quello dove è l'osservatore (abbastanza veloce)?
 4. Si consideri il piano contenente P. Q è contenuto completamente nello stesso semispazio dove è l'osservatore (abbastanza veloce)?
 5. Le proiezioni di P e Q sullo schermo sono disgiunte (pesante)?
- Se tutti i test falliscono, controllo se è corretto disegnare Q prima di P.
- Se anche questo fallisce Q viene tagliato usando il piano contenente P (con clipping), ed i pezzi vengono collocati nella lista al posto giusto.

s

BSP tree

- Poiché l'ordine dei poligoni dipende dal punto di vista, bisogna ricalcolarlo ad ogni spostamento.
- Non è praticabile in una applicazione interattiva (es: simulatore di volo).
- Usiamo un albero BSP per ottenere rapidamente l'ordine corretto dei poligoni al cambiare del punto di vista.
- Costruzione (off-line) di un albero BSP auto-partitioning (divido usando i piani che contengono i poligoni). Le foglie sono poligoni o frazioni.
- Attraversando l'albero si ottiene l'ordinamento desiderato:
 - consideriamo la radice dell'albero
 - il punto di vista si trova (diciamo) a destra dell'iperpiano associato alla radice.
 - allora i poligoni che stanno a sinistra si possono disegnare prima di quelli che stanno a destra dell'iperpiano.
 - l'ordine per i poligoni che si trovano nei due semispazi si ottiene ricorsivamente considerando separatamente i due sottoalberi.
- Questa tecnica si basa sull'idea comune ad altre di pre-processare la scena usando strutture dati opportune per ridurre il tempo di rendering.

Back-face culling

- Come nel caso del view frustum culling, si vuole evitare di processare linearmente tutti i poligoni.
 - A tale fine si costruisce una struttura gerarchica in cui i poligoni vengono raggruppati in **cluster** (gruppi) basati sulle normali e sulla prossimità.
 - Ciascuno dei cluster induce una partizione dello spazio in 3 regioni: quella alla quale tutti i poligoni del cluster danno la faccia, quella alla quale tutti i poligoni del cluster mostrano il retro, e una regione mista.
 - Al tempo di rendering, dato un punto di vista, se questo giace nella regione 1 o 2 il destino dei poligoni del cluster è determinato, mentre è necessario visitare i sottocluster solo se il punto di vista giace nella regione 3.