Tecniche per applicazioni interattive

Dove si introducono alcune tecniche di rendering adatte alla realizzazione di applicazioni interattive.

- **Introduzione**
- **Visibility culling**
- **Livello di dettaglio continuo**

Introduzione

- Rassegna di tecniche indispensabili per effettuare il pipeline rendering in tempo reale.
- Sono tecniche a carico della applicazione, la quale ha il compito di inviare alla rendering pipeline un carico di poligoni da disegnare "adeguato" alle capacità di quest'ultima.
- In un contesto interattivo (real-time) e con scene complesse, non è pensabile di inviare alla rendering pipeline tutti i poligoni che compongono la scena e lasciare che si arrangi.
- **Idea**: evitare di spedire alla pipeline di rendering i poligoni che non contribuiranno alla immagine, o perchè nascosti o perché troppo piccoli.
- "Se qualcosa non si vede, non va disegnato".
  - **Visibility culling**: sfrondare quanto prima e quanto più efficientemente posibile i poligoni che non dovranno essere disegnati.
  - **Livello di dettaglio**: aggiustare il livello di dettaglio ($\backsim$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.
- Tutte si basano su opportune strutture dati. In generale, si baratta il tempo speso off-line per la costruzione di strutture dati con il tempo risparmiato nel rendering on-line.

Visibility culling

- In generale, l'idea del **visibility culling** è evitare di processare poligoni che non verranno disegnati perchè non visibili (occlusi o esterni al view frustum).
- Gli algoritmi devono essere **conservativi** , ovvero non scartare mai un poligono che dovrebbe essere visualizzato.
- Sono algoritmi di sfoltimento (o broad phase) e non lavorano al livello del singolo poligono ma di gruppi di poligoni o di oggetti.
- Gli algoritmi devono essere **efficienti**, nel senso che devono scartare un insieme di poligoni ad un costo inferiore alla scansione di ciascun poligono (altrimenti il lavoro lo poteva fare la rendering pipeline).
- Comprende:
  - **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
  - **Contribution culling**: scarto di oggetti la cui proiezione è troppo piccola.
- Per ciascuno di questi possiamo avere algoritmi che hanno complessità lineare nel numero

dei poligoni (li devono testare tutti, uno alla volta).

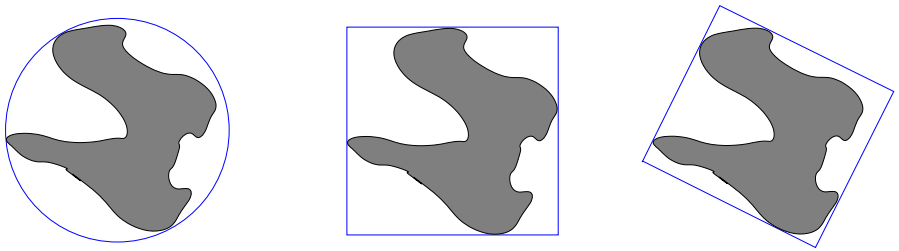
- L'idea del visibility culling è di pre-processare (off-line) la scena, costruendo strutture dati opportune che possano consentire di scartare poligoni non visibili in tempo sub-lineare in fase di rendering.

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 intersezioni tra oggetti, in cui un oggetto è il il view frustum.
- Vedremo ora in generale come affrontare il problema della intersezione tra oggetti.
- Questo problema è connesso con quello delle **collisioni** tra oggetti (*collision detection*) che è importantissimo in scene dinamiche.
- Non tenendo conto della coerenza temporale del moto, il probelema delle collisioni si riduce a quello delle intersezioni. In generale è più complesso, tuttavia.
- Anche il **ray casting** (e ray tracing) è connesso al problema delle intersezioni, poiché si devono calcolare in modo efficiente molte intersezioni oggetto-raggio.

Bounding volume

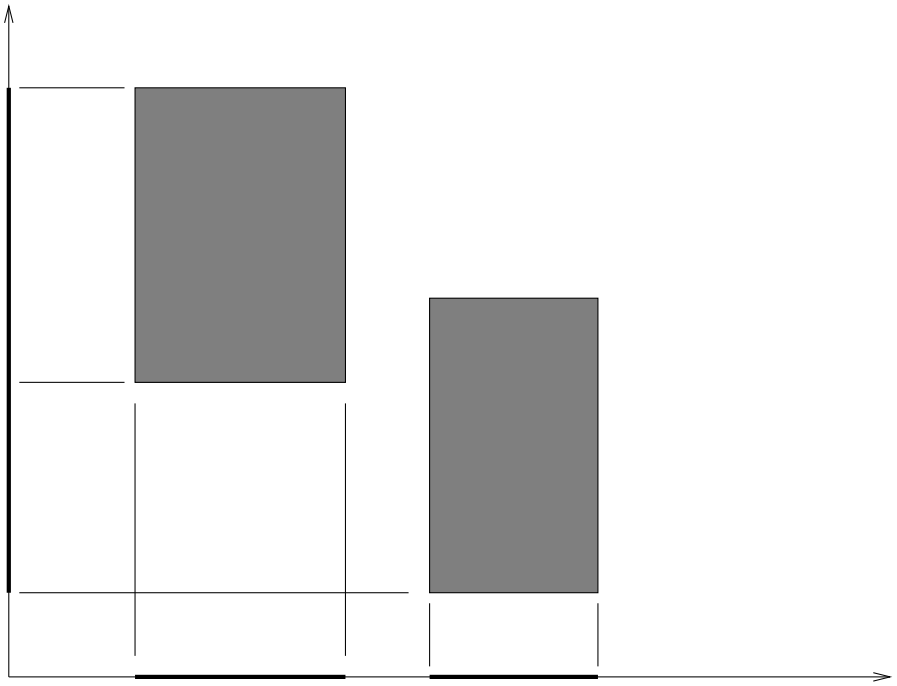
- Una prima tecnica per "sgrezzare" le intersezioni consiste nel racchiudere gli oggetti in volumi che li contengono, con i quali sia facile testare l'intersezione: se non c'è intersezione con il bounding volume non c'è intersezione con l'oggetto racchiuso.
- Questo non rende sub-lineare la complessità 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

Intersezione di AABB

- Determinare se due AABB si intersecano è molto più semplice che fare il test di intersezione per poliedri.
- L'algoritmo che vedremo è simile (ma molto più semplice) alla tecnica di sweep per intersezione di segmenti. L'idea base è che due AABB si intersecano se e solo se le loro proiezioni su ciascun asse coordinato si intersecano.

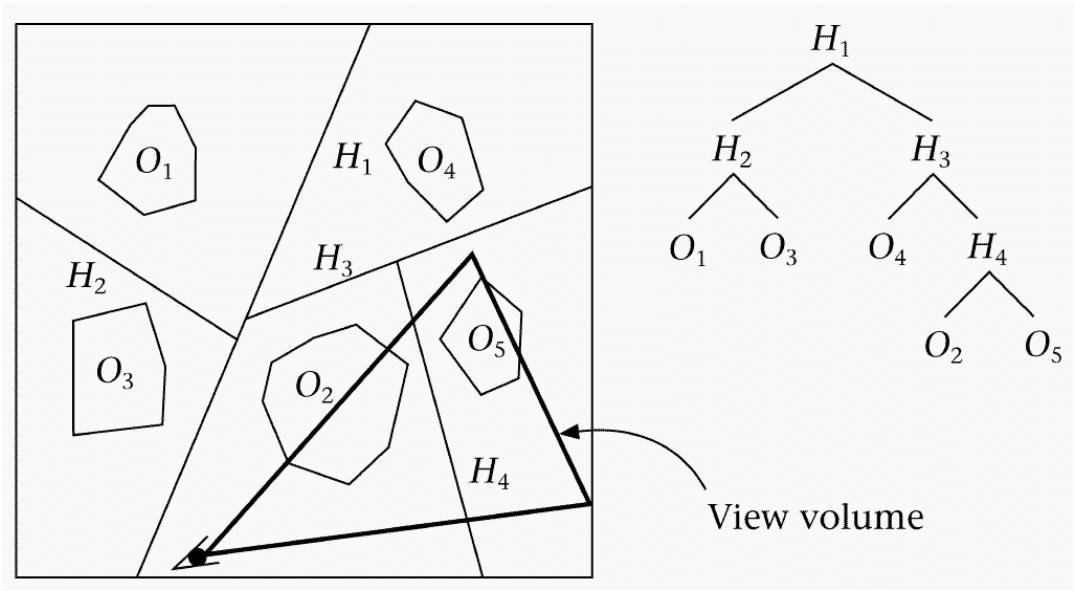


- Il problema si scompone quindi in tre problemi unidimensionali: trovare se un insieme di intervalli sulla retta si intersecano.

- Si ordinano gli estremi degli intervalli in una lista, mantendo l'informazione circa il tipo di estremo (inferiore o superiore) e l'identità dell'intervallo.
- Si scandisce la lista:
  - ogni volta che si incontra un estremo inferiore si pone l'intervallo corrispondente in una lista di intervalli attivi e si stabilisce che gli intervalli attivi si sovrappongono.
  - ogni volta che si incontra un estremo superiore si toglie l'intervallo dalla lista.

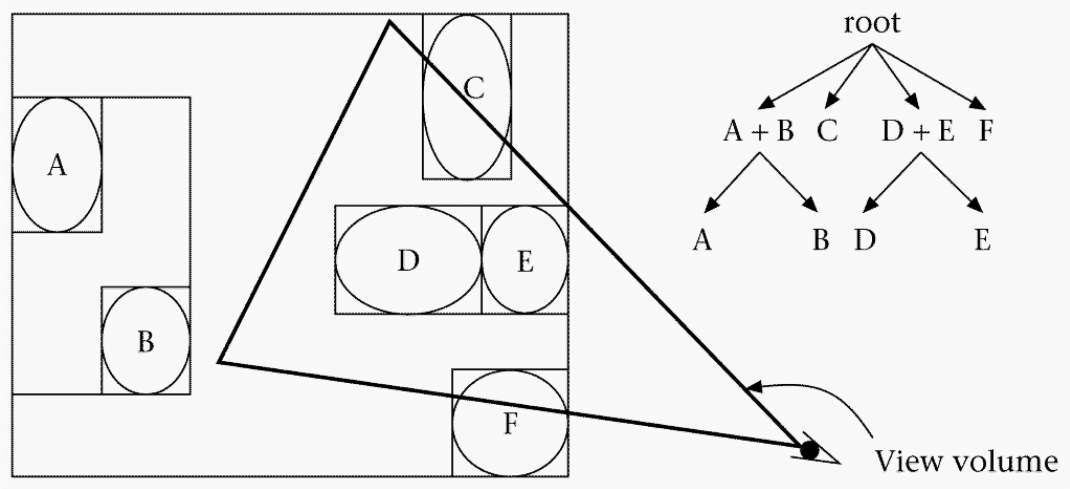
Space partitioning

- 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 scene, ed al livello più basso si hanno bounding volumes per i singoli oggetti.



- È stata introdotta 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 attrecerso **portali**.
- L'idea di fonodo 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 attaverso di esso, ovvero si vuole calcolare il **Potentially Visible Set** (PVS). Le celle che rientrano nel PSV vengono disegnate.
- Il PVS si può calcolare staticamente (off-line) o dinamicamente (on-line).

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 poligono vengono raggruppati in cluster 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 run time, 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.

Depth-sort

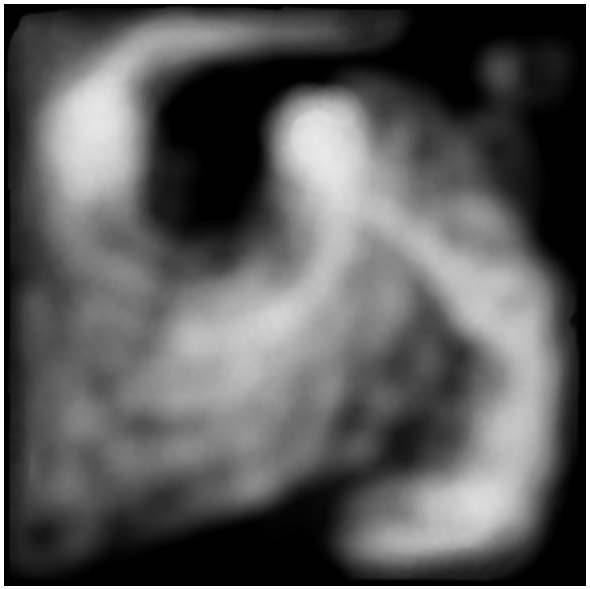
- Il metodo depth-sort per HSR richiede che i poligoni vengano ordinati.
- Poiché l'ordine dipende dal punto di vista, bisogna ricalcolarlo ad ogni spostamento.
- Non è praticabile in una applicazione interattiva (es: simulatore di volo).
- Un albero BSP, invece, una volta costruito (richiede pre-processing), consente di ottenere rapidamente l'ordine giusto al cambiare del punto di vista.
- Costruisco un albero BSP auto-partitioning con i triangoli (divido usando i piani che contengono i triangoli).
- Vediamo come si ottiene l'ordinamento:
  - consideriamo la radice dell'albero
  - il punto di vista si trova (diciamo) a destra dell'iperpiano associato alla radice.
  - allora gli oggetti che stanno a sinistra si possono disegnare prima di quelli che stanno a destra dell'iperpiano.
  - l'ordine per gli oggetti che si trovano nei due semispazi si ottiene ricorsivamente considerando separatamente i due sottoalberi.
- In sintesi: attraversando l'albero ottengo i poligoni ordinati back-to-front.

- Non fa parte, a rigore, delle tecniche di Visibility Culling, poiché non elimina i triangoli ma li ordina.
- Si basa però sull'idea comune alle tecniche precedenti di pre-processare la scena usando strutture dati opportune (BSP trees) per ridurre il tempo di rendering.

Livello di dettaglio

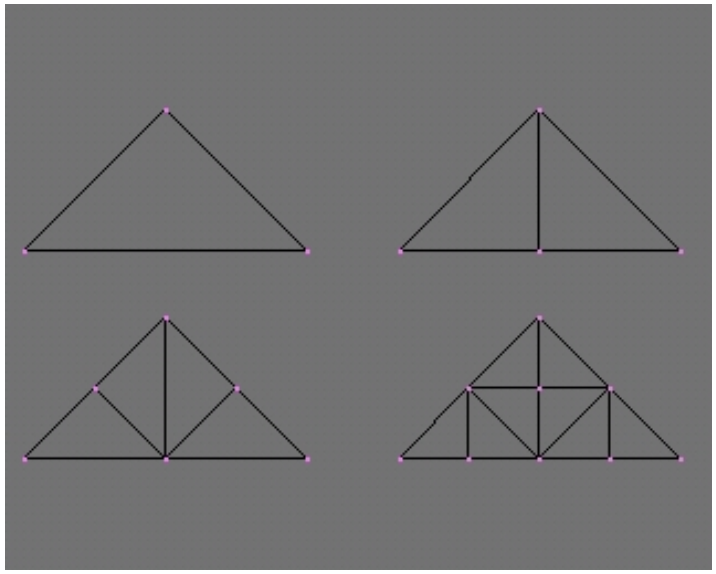
- Per descrivere un oggetto complesso con una mesh mantenendo un sufficiente dettaglio geometrico sono necessari tipicamente molti poligoni. Altrimenti vi sarà una **mancanza di dettaglio** quando l'oggetto viene analizzato da vicino.
- Viceversa vi sarà uno **spreco di risorse** quando l'oggetto è lontano 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**.

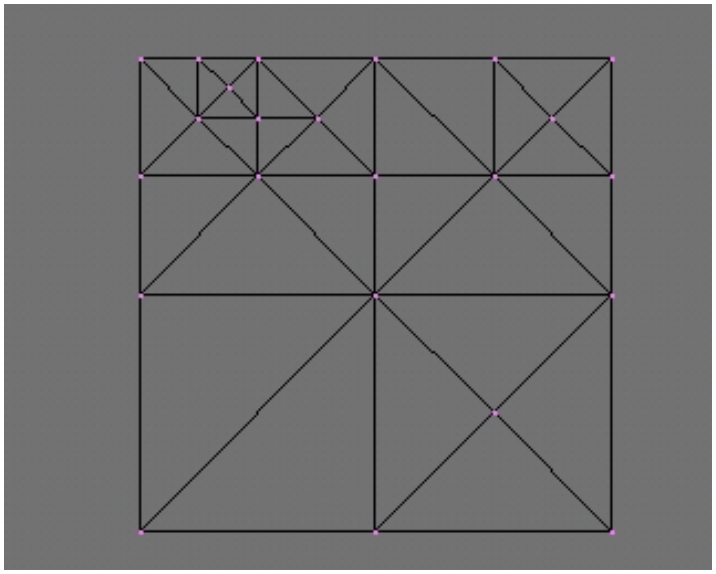
Real-time Optimally Adapting Meshes



- 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.

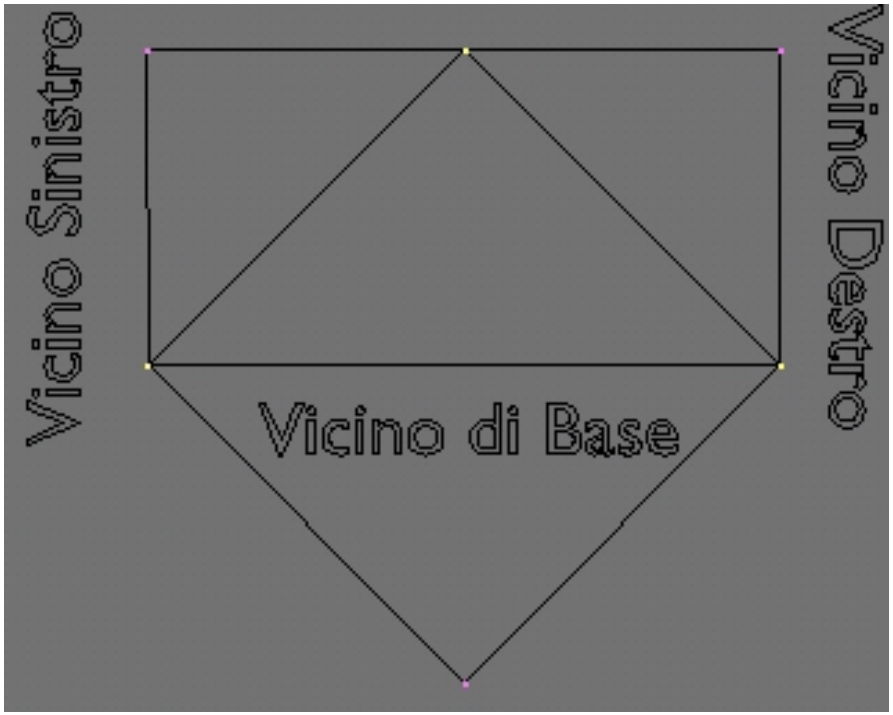
- I **binary triangle tree** sono una struttura dati molto simile al quadtree.
- Si parte dalla considerazione geometrica che è possibile dividere un triangolo rettangolo in due triangoli rettangoli
- Iterando tale suddivisione si ha una gerarchia di triangoli, in modo del tutto simile alla gerarchia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

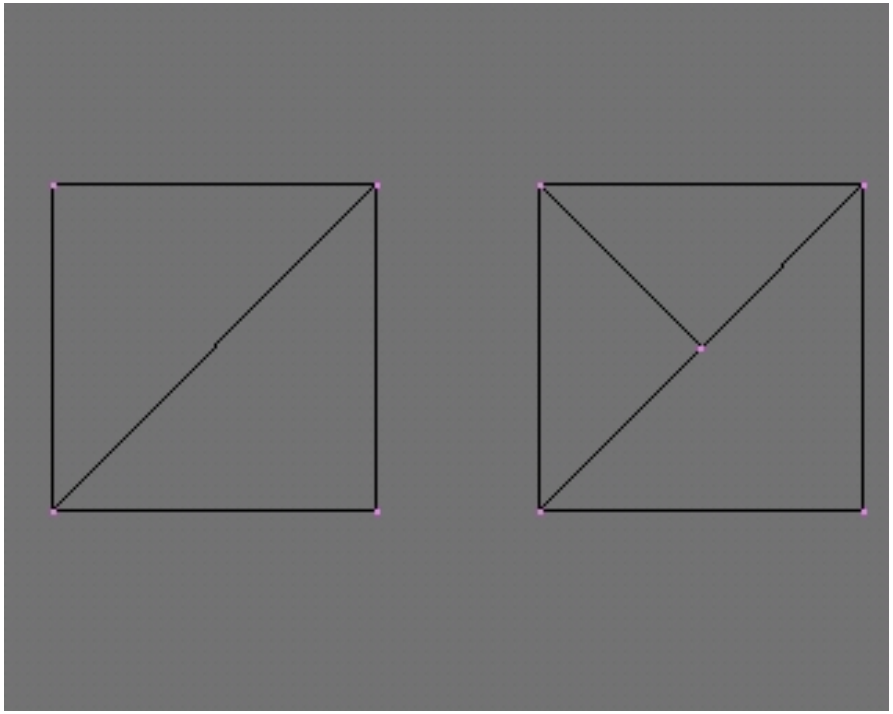
$$var_t = |h(t_i) - 1/2(h(t_2) + h(t1))|$$

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

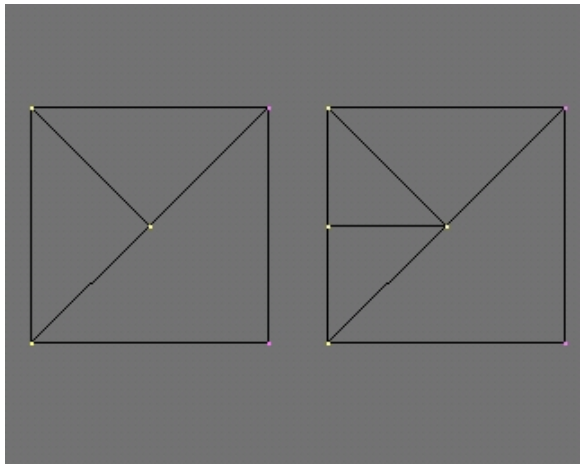
- Ovviamente il punto t_i sul piano xy è dato da $t_i = 1/2(t_1 + t_2)$

- 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

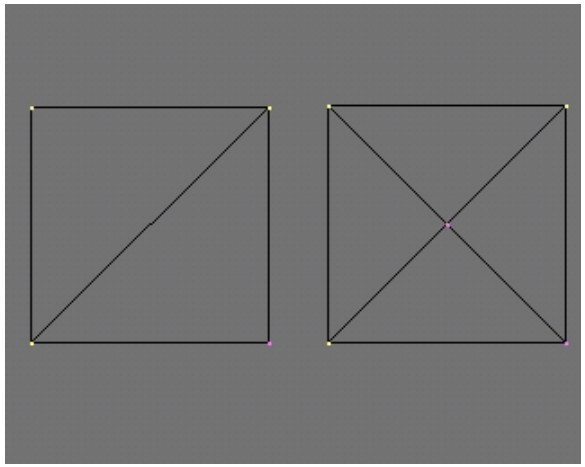
- Rimane un ultimo punto da discutere
- La suddivisione ricorsiva non deve introdurre inconsistenze nella triangolazione (crack o giunzioni a T)
- Per esempio nella seguente figura, la suddivisione del triangolo superiore e non di quello inferiore introduce una inconsistenza nella mesh risultante (quello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



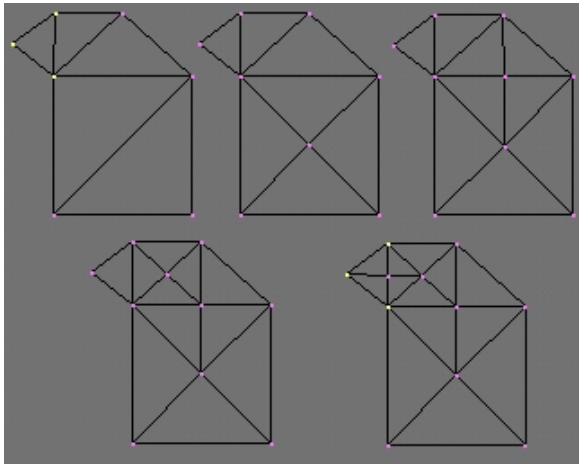
- Risulta utile, per risolvere il problema, classificare i triangoli in questo modo
  1. **Triangoli che hanno l'ipotenusa sul bordo;** in tal caso la loro suddivisione può essere fatta senza alcun problema
  2. **Triangoli appartenenti ad un diamante;** si definiscono così quei triangoli il cui vicino 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**)
- Nulla è meglio di una immagine



(1) Triangolo di bordo

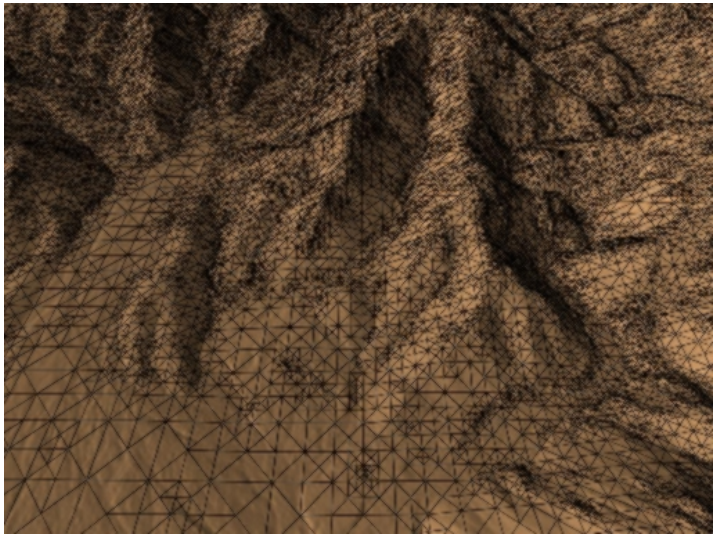


(2) Diamante

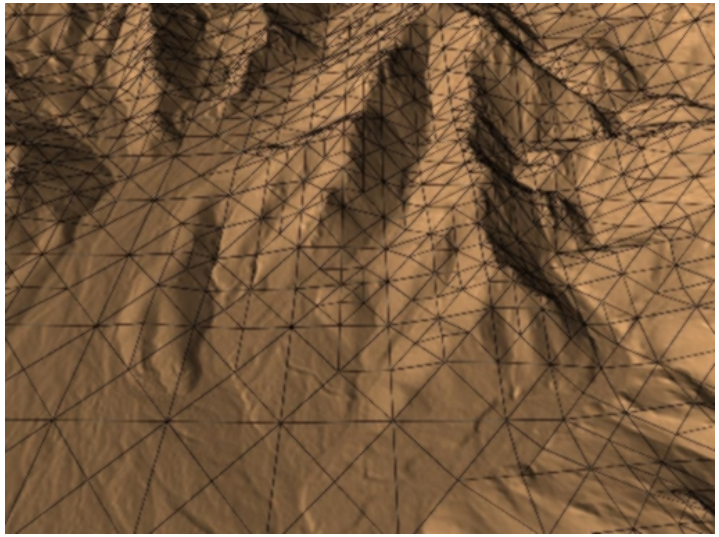


(3) Ricorsione

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