Rimozione superfici nascoste

Introduzione

Nel pipeline di rendering rasterization based è necessario inserire un meccanismo che consenta di colorare i pixel a schermo solo degli oggetti più vicini, cioè gli oggetti più vicini devono coprire quelli alle loro spalle. Per farlo ci sono varie tecniche.

Algoritmo del pittore

Idealmente, si tratta di colorare prima i pixel delle primitive più lontane, e poi via via quelli delle primitive più vicine, sovrascrivendo il frame buffer. In sostanza si tratta di ordinare le primitive in base ai loro valori sulla z prima di rasterizzarle.

Questo algoritmo è semplice ma ha molti difetti:

❌ Prevede un sorting, che si sa essere $O(n\,logn)$, con n primitive

❌ Non è inseribile nel pipeline della GPU, bensì è la CPU che dovrebbe inviare i triangoli alla GPU nell'ordine giusto

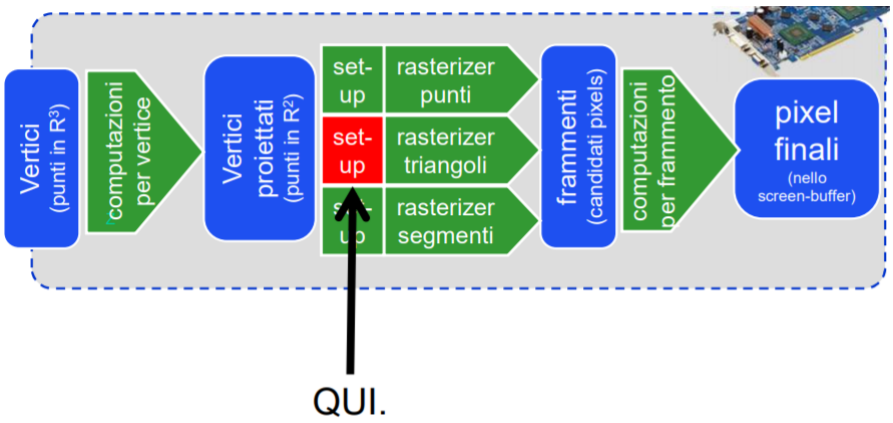
❌ Una primitiva non ha un'unica z, ogni vertice ne ha una. Può capitare di non riuscire a trovare un ordine corretto, ad esempio nelle intersezioni tra primitive

❌ Costringere ad un ordine il rendering delle primitive può rallentare il processo

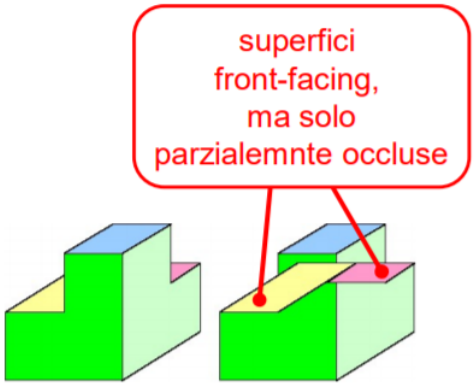
---

Back face culling

Un algoritmo efficiente e funzionante è il **back face culling**, ma che funziona specificatamente su superfici ben orientate e chiuse. Infatti l'idea è quella di non mostrare a schermo i triangoli nascosti all'interno dell'oggetto stesso e che saranno quindi orientati in modo diverso. Ad esempio se noi renderizzassimo una bottiglia, ci accorgeremo che i triangoli della superficie esterna saranno orientati verso l'esterno (*front-facing*) mentre i triangoli interni della bottiglia saranno orientati verso l'interno (*back-facing*). Questo tipo di test è compatibile con la pipeline ed è inseribile prima della rasterizzazione dei triangoli:



Come dicevamo, questo algoritmo non copre però tutti i casi, ad esempio io potrei avere dei triangoli orientati verso l'esterno ma che sono occlusi. In questo caso il back face culling non se ne accorgerebbe:

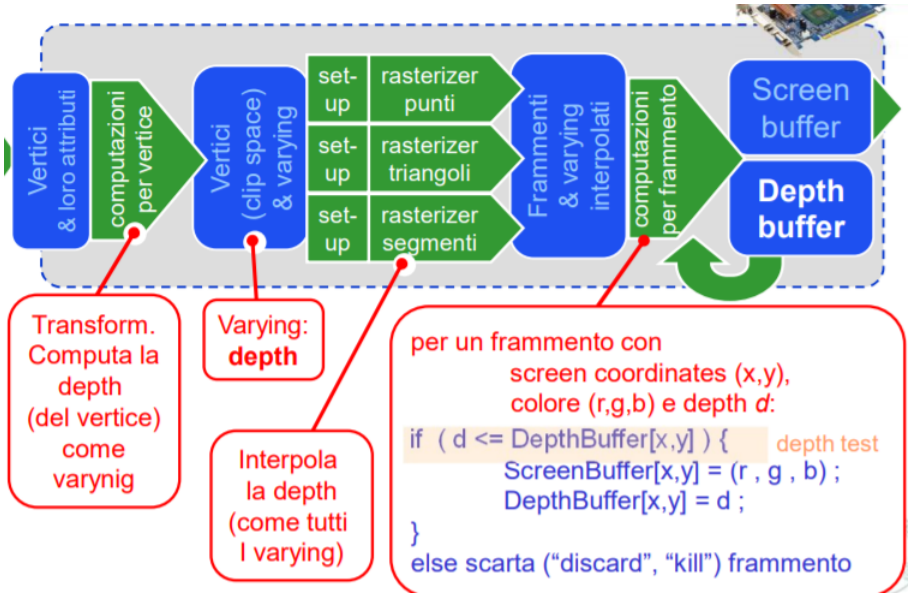


H2 Algoritmo dello z-buffer

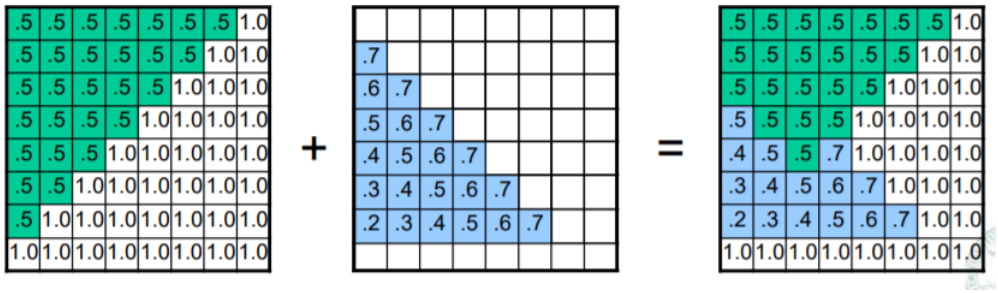
L'idea di base è la seguente: facciamo il test della profondità (***depth test***) a livello di frammento. In questo modo se ho più frammenti che corrispondono allo stesso pixel, prenderò il frammento di profondità minore.
Questa idea deve essere supportata da una struttura dove memorizzare la profondità di ogni pixel. Introduciamo quindi il **depth buffer** (o **z-buffer**), il quale sarà un array bidimensionale con la stessa dimensione dello screen buffer, e ogni cella del buffer, corrispondente a un pixel, non conterrà i valori RGB del pixel come lo screen buffer, bensì la profondità.

La profondità di un frammento è data interpolando i valori di profondità in spazio clip dei vertici del triangolo, così come per tutti gli altri attributi/dati assegnati a un vertice. Il depth test viene quindi fatto nella fase di fragment process, confrontando il valore di profondità del frammento appena prodotto con quello attualmente presente nel buffer.

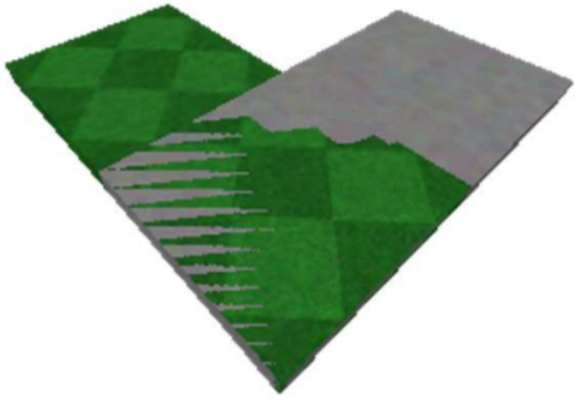
Visualizzando l'algoritmo all'interno del pipeline:



Un esempio di applicazione dell'algoritmo è rappresentato dalla seguente immagine, in cui si vede che i frammenti colorati saranno quelli con valore di profondità inferiore:



Questo algoritmo può avere dei problemi dovuti alla limitatezza dei valori continui memorizzati in un computer. Infatti i numeri reali a un certo punto vengono troncati dal momento che i bit finiscono. Se due valori sono molto vicini quindi, i problemi di approssimazione creano dell'*aliasing* in quanto l'algoritmo non riesce a distinguere chi è davanti e chi è dietro. Questo problema è detto ***z-fighting***:



Aldilà di questo problema, questo algoritmo ha molti vantaggi:

- ✔ il rendering diventa ***order independent***
- ✔ funziona in ogni situazione
- ✔ adatto alla parallelizzazione HW

Invece gli svantaggi sono:

❌ ***z-fighting*** possibile (anche se improbabile)

❌ costa memoria in quanto va aggiunto il depth buffer

❌ i frammenti vengono scartati solo alla fine del pipeline, quindi prima sono stati inutilmente elaborati

❌ problemi con le superfici non opache

❌ il buffer è condiviso in parallelo, quindi è necessario che il test sia HW (*output combiner*) e che questo ne assicuri la sincronia

H3 *z*-buffer - Considerazioni sull'efficienza

- scartare è più veloce di sovrascrivere, sarebbe quindi meglio disegnare prima gli oggetti più vicini, poi quelli lontani (l'opposto dell' algoritmo del pittore).
- il depth test viene fatto a livello HW per motivi di velocità e sincronizzazione, quindi in realtà abbiamo una componente HW, cioè l' *output combiner* , che ha il compito di scrivere sul frame e sul depth buffer.