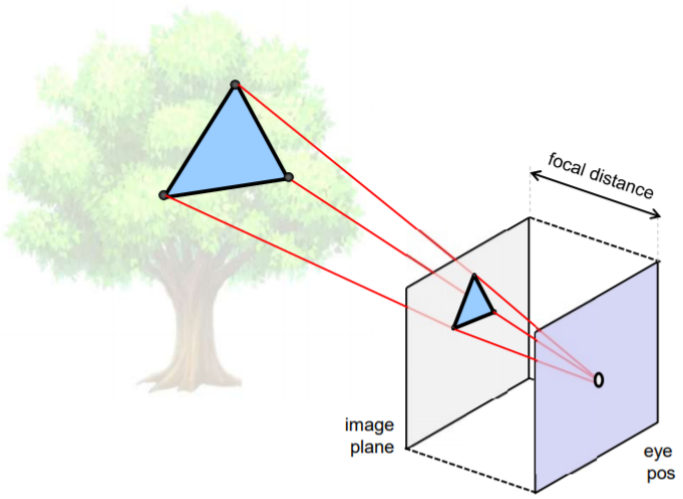
H1 Rasterization based

H2 An opposite approach

Gli algoritmi **rasterization based** sono quelli più usati per il rendering real-time al momento, e hanno un approccio opposto rispetto al ray-tracing.



Mentre nel ray-tracing, era il POV che proiettava raggi sulla scena per capire come colorare un pixel, in questo caso è la scena che viene "proiettata" sul piano immagine, in modo da passare da 3D a 2D. Dopodiché la scena 2D viene **rasterizzata**, cioè l'immagine 2D viene convertita in pixel.

H2 Descrizione dell'algoritmo

Uno pseudocodice è il seguente:

```
FOREACH primitive o{
    find where o falls on screen
    rasterize 2D shape
    FOREACH produced pixel p{
        find color of o
        color p with it
    }
}
```

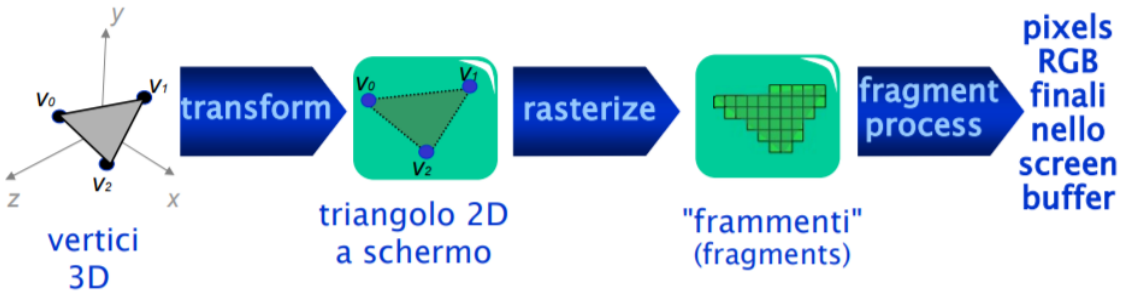
Il processo di **rasterizzazione**, che vedremo in dettaglio in seguito, è però applicabile in modo efficiente solo a determinati oggetti "semplici", ad esempio i triangoli. Questo è il motivo per cui ci interessa come convertire un qualsiasi modello 3D in una *tri-mesh*, infatti la GPU è specializzata nel rasterizzare triangoli.

Un altro modo di chiamare questa classe di algoritmi è **Transform & Lighting**, in quanto le due macro-fasi dell'esecuzione possono essere divisi in:

- Transform : proietto la scena 3D in uno spazio schermo 2D
- Lighting : illumino il pixel prodotto dalla rasterizzazione in base al colore della primitiva e al computo dell'illuminazione

H2 Pipeline di rendering

Uno schema più dettagliato dell'algoritmo è il seguente:

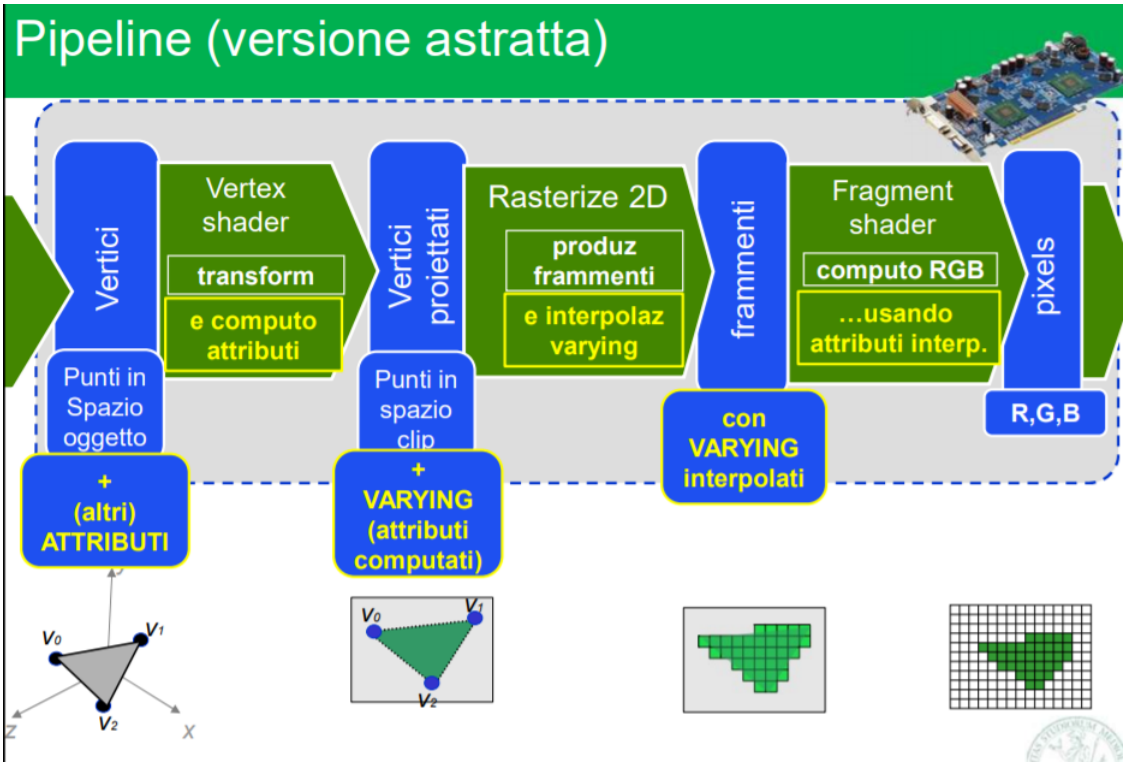


come si nota, l'algoritmo ha uno schema a pipeline, quindi è possibile parallelizzarlo a livello HW facilmente. Ricapitolando le 3 fasi del pipeline di rendering, avremo:

1. **Transform**:
    - *Input* : vertici 3D
    - I vertici in 3D vengono proiettati attraverso una serie di trasformazioni spaziali in spazio schermo, ora quindi in 2D
    - *Output* : vertici in posizione 2D in spazio schermo (precisamente in spazio clip)
2. **Rasterize**:
    - *Input* : vertici in posizione 2D in spazio schermo (precisamente in spazio clip)
    - I triangoli 2D vengono rasterizzati , cioè viene identificato un *frammento* per ogni pixel coperto (vedremo come) dal triangolo
    - *Output* : frammenti per ogni triangolo 2D
3. **Fragment process**:
    - *Input* : frammenti per ogni triangolo 2D
    - Per ogni frammento, si computa il colore da assegnare al pixel corrispondente, in base al colore della primitiva(attributi) e spesso anche all' illuminazione della scena
    - *Output* : valori finali da inserire nello screen buffer

---

H3 Pipeline di rendering - details



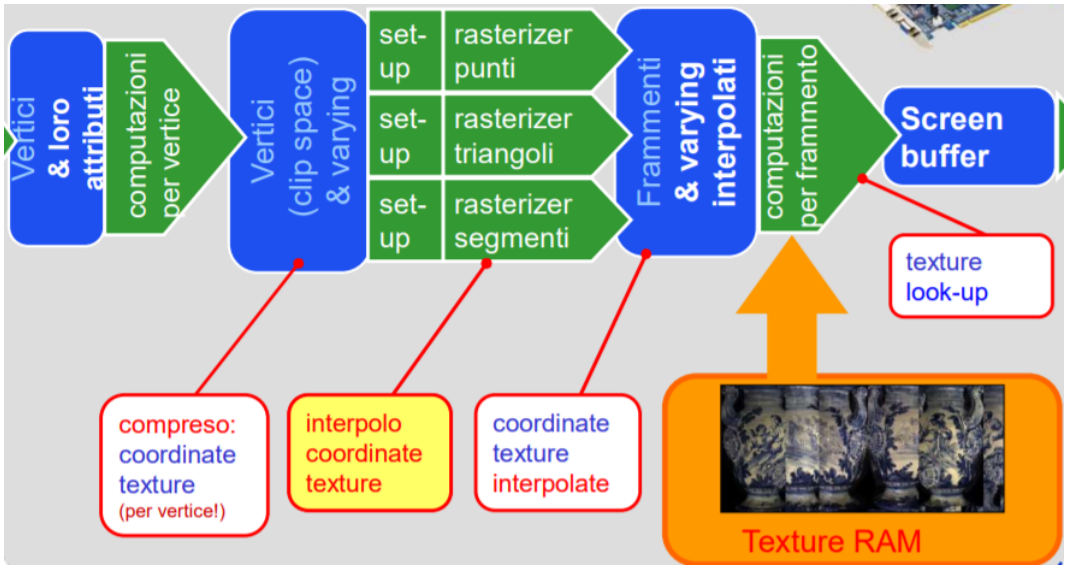
Quello sopra è uno schema più dettagliato del pipeline di rendering rasterization based. La cosa da notare è che vi sono due fasi all'interno che sono programmabili, e i programmi che coprono quelle due fasi sono il ***vertex shader*** e il ***fragment shader***.

- **Vertex Shader**:
  - *Transform* : trasformazione spazio oggetto in spazio clip
  - computo degli attributi, gli attributi computati sono detti *varying*
- **Fragment Shader**
  - *Fragment Process* : computo del colore da inserire nello screen buffer e del depth value da inserire nel depth buffer, il quale verrà usato dall' *output combiner* per il depth test

La fase di **rasterizzazione** è invece *hardwired*, e svolge le seguenti operazioni:

- *Rasterize* : produzione dei frammenti
- Interpolazione degli attributi per vertice

H3 Texture mapping nel pipeline



Percorriamo il texture mapping fase per fase:

1. Le coordinate UV fanno parte degli attributi dei vertici dell'oggetto
2. Il vertex shader elabora le coordinate UV, che ora sono attributi varying
3. Il rasterizzatore interpola le coordinate UV e le assegna a ogni frammento individuato
4. Il fragment shader assegna a ogni frammento il colore corretto in base alle coordinate UV di quel frammento, e per farlo va in memoria della GPU a leggere la texture assegnata a quella primitiva (*texture look-up*).