Ottimizzazione di un renderer 2D mediante GPU

Samuele Esposito

Università degli Studi di Padova

2025-07-22

Outline

Introduzione

Programmazione di GPU

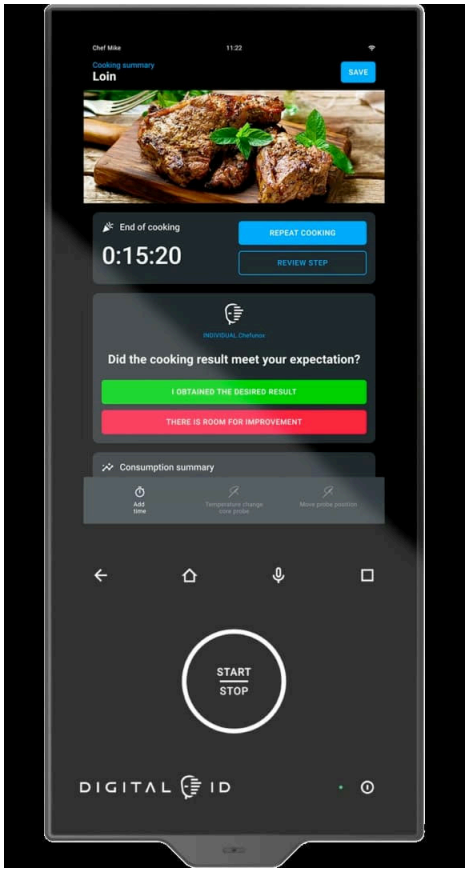
Lavoro svolto

L'azienda: UNOX S.p.A.

UNOX produce forni per gastronomie, ristorazione, centri cottura…

Multinazionale di Cadoneghe (PD), vanta più di 1300 dipendenti, e 42 filiali estere.

Il sistema operativo Digital.ID™



I forni di fascia più alta includono un sistema operativo *smart*, denominato Digital.ID™, realizzato in React Native.

L’azienda desidera però ridurre i costi per l’hardware, ma il sistema precedente era troppo lento; da qui è nata l’idea di creare una soluzione *in house*.

Questa soluzione è tutt’ora in sviluppo, realizzata in Rust.

Outline

Introduzione

Programmazione di GPU

Lavoro svolto

Introduzione alle GPU

Le GPU (Graphical Processing Unit) sono coprocessori capaci di lavorare in maniera altamente parallela.

Introduzione alle GPU

Le GPU (Graphical Processing Unit) sono coprocessori capaci di lavorare in maniera altamente parallela.

Al giorno d'oggi, presentano due *pipeline* differenti, per due scopi differenti:

- compute pipeline — non verrà trattata oggi.
- graphics pipeline.

Introduzione alle GPU

Le GPU (Graphical Processing Unit) sono coprocessori capaci di lavorare in maniera altamente parallela.

Al giorno d'oggi, presentano due *pipeline* differenti, per due scopi differenti:

- compute pipeline — non verrà trattata oggi.
- graphics pipeline.

Per accedere alla *GPU* si possono utilizzare diverse *API* — alcuni esempi sono OpenGL, Vulkan o DirectX.
Durante il progetto, abbiamo deciso di utilizzare *WebGPU*.

Pipeline grafica

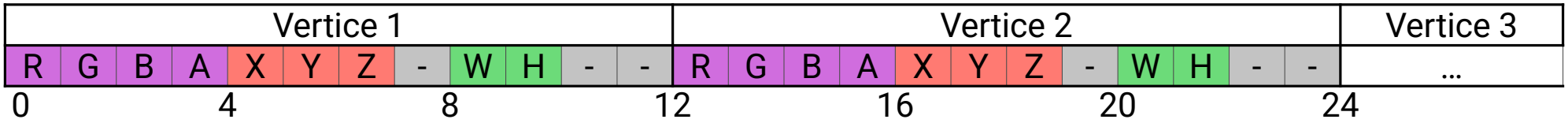
L’elaborazione si divide in più fasi, eseguite in maniera sequenziale:

- configurazione della pipeline (eseguita all’esterno della GPU);
- esecuzione della *vertex shader*;
- *primitive assembly* e *clipping*;
- rasterizzazione;
- esecuzione della *fragment shader*.

Configurazione della pipeline

Oltre a caricare e compilare le *shader*, è necessario definire gli input della *vertex shader* e gli output della *fragment shader*.

L'input principale delle *vertex shader* è un *vertex buffer*, ossia un *array* di input arbitrari, che la *vertex shader* riceverà come input.



Vertex shader

Le *vertex shader* si occupano di generare i vertici. Possono essere utilizzate anche per modificarli.

```
@vertex
fn vert_main(
  @builtin(vertex_index) vertexIndex : u32
) -> @builtin(position) vec4<f32> {
  let pos = array(
    vec2f(0, 0), vec2f(1, 0), vec2f(0, 1),
    vec2f(0, 1), vec2f(1, 0), vec2f(1, 1),
  );

  return vec4f(pos[vertexIndex], 1, 1);
}
```

Vertex shader

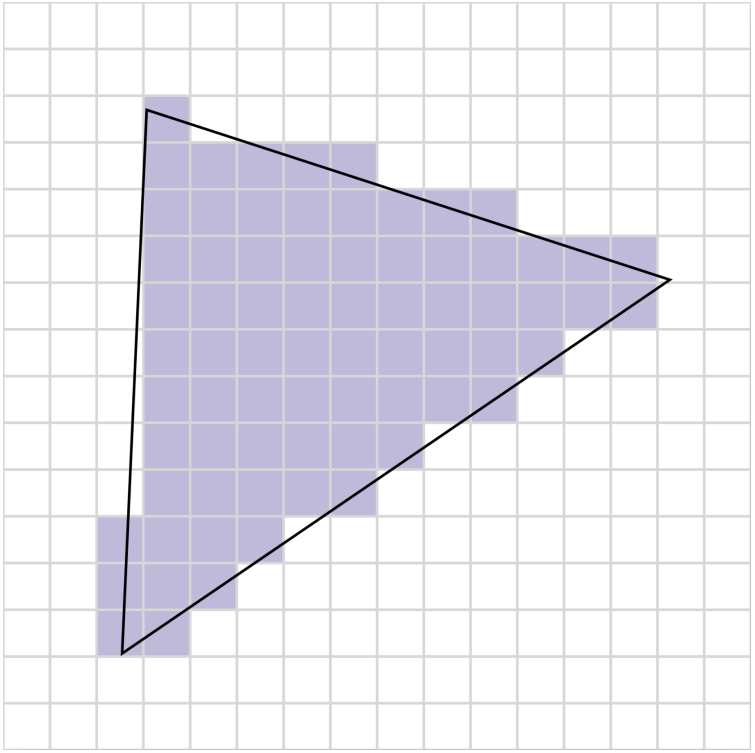
Le *vertex shader* si occupano di generare i vertici. Possono essere utilizzate anche per modificarli.

In *WebGPU* le *shader* sono definite in un linguaggio appositamente sviluppato, chiamato *WGSL*.

```
@vertex
fn vert_main(
  @builtin(vertex_index) vertexIndex : u32
) -> @builtin(position) vec4<f32> {
  let pos = array(
    vec2f(0, 0), vec2f(1, 0), vec2f(0, 1),
    vec2f(0, 1), vec2f(1, 0), vec2f(1, 1),
  );

  return vec4f(pos[vertexIndex], 1, 1);
}
```

Clipping e Rasterizzazione

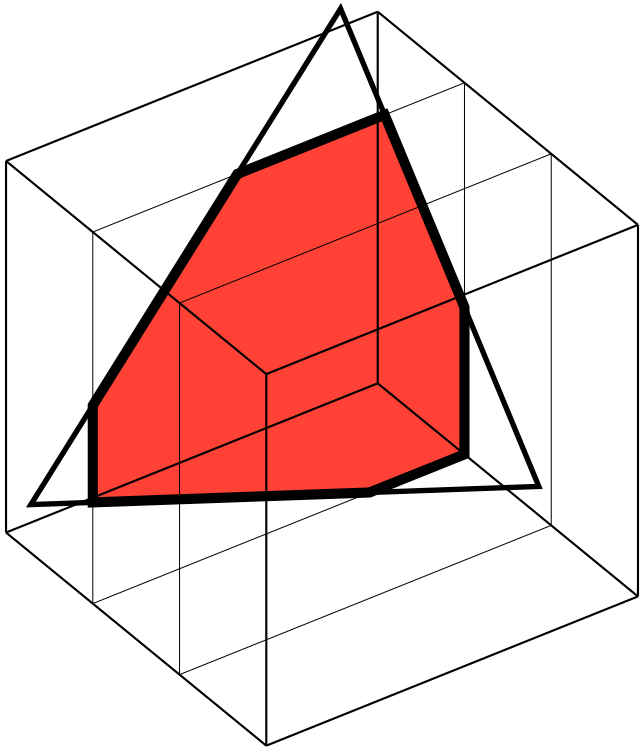


Successivamente, viene eseguita una serie di fasi *fixed function*.

La prima è il Clipping, in cui le primitive non visibili vengono rimosse.

Successivamente avviene la rasterizzazione, in cui viene generato un *fragment* per ogni pixel coperto dalla primitiva.

Il Clip Space



Il clipping viene effettuato coordinate omogenee, un concetto derivato dalla geometria proiettiva.

Tutte le coordinate (tridimensionali) vengono estese con un quarto componente, w. Viene definito un cubo, chiamato “clip volume”, la cui dimensione dipende da w. Tutte le coordinate all’esterno di esso vengono considerate “invisibili”.

Fragment shader

Si occupano di calcolare il colore di un *fragment* che ricevono in ingresso.

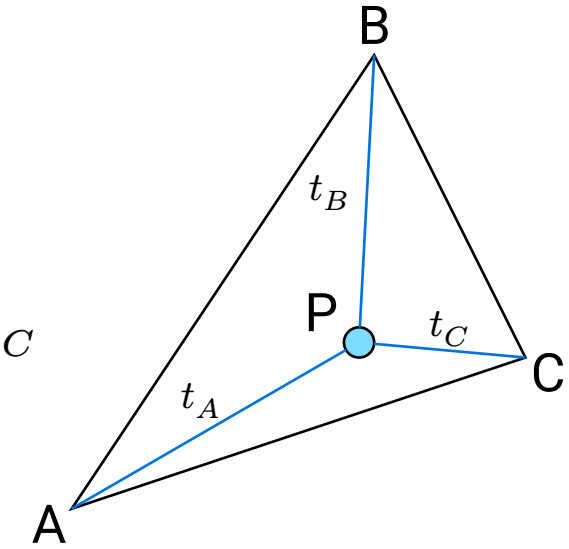
Fragment shader

Si occupano di calcolare il colore di un *fragment* che ricevono in ingresso.

I valori di un *fragment* sono calcolati mediante interpolazione, usando le coordinate baricentriche del punto come coefficienti.

Posizione: $P = A \cdot t_A + B \cdot t_B + C \cdot t_C$

Colore: $\text{color}(P) = \text{color}(A) \cdot t_A + \text{color}(B) \cdot t_B + \text{color}(C) \cdot t_C$



Utilizzo di texture nelle fragment shader

Come si possono creare i *vertex buffer*,
si possono anche creare *texture buffer*.

Utilizzo di texture nelle fragment shader

Come si possono creare i *vertex buffer*, si possono anche creare *texture buffer*.

Esse però possono solo essere lette mediante un *sampler* il quale applica interpolazione tra i pixel più vicini.

```
var ourSampler: sampler;
var ourTexture: texture_2d<f32>;

@fragment fn fs_main(
  @location(0) texcoord: vec2<f32>,
) -> @location(0) vec4f
{
  return textureSample(
    ourTexture,
    ourSampler,
    texcoord,
  );
}
```

Outline

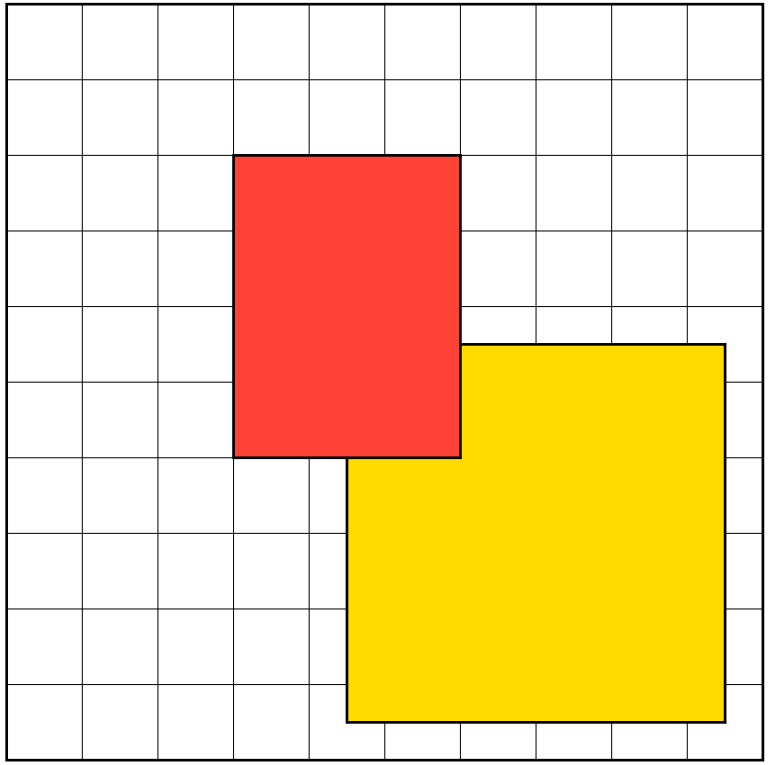
Introduzione

Programmazione di GPU

Lavoro svolto

Composizione in GPU

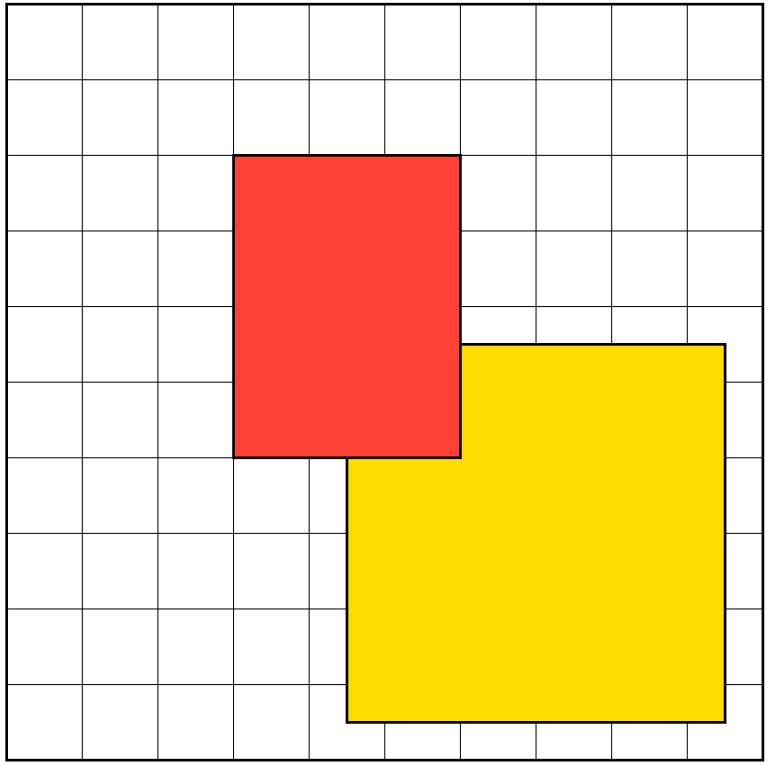
L’idea del progetto era effettuare solamente la composizione in GPU.



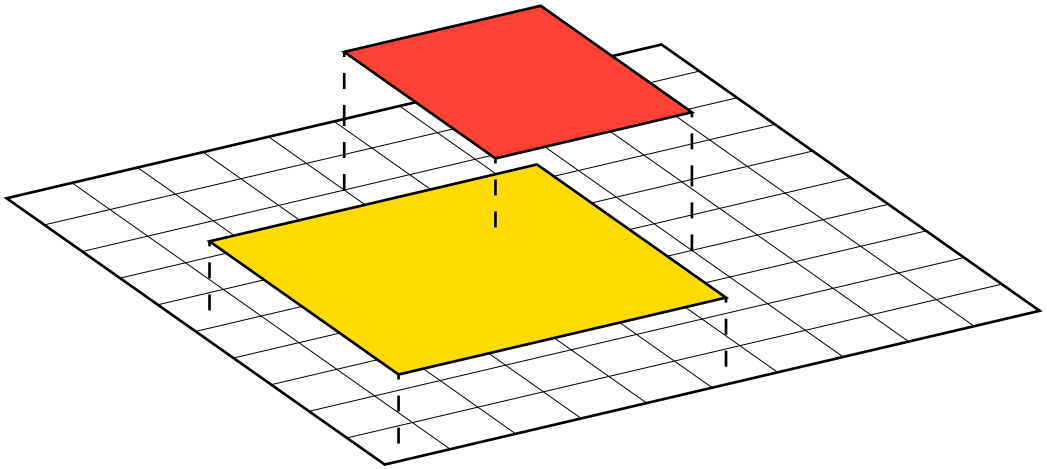
Composizione in GPU

L’idea del progetto era effettuare solamente la composizione in GPU.

Purtroppo, fare così ha lasciato potenziali prestazioni non sfruttate, riducendo però la quantità di codice da modificare.



L'approccio



Abbiamo scelto un approccio di grafica 3D:

- Ogni layer è un rettangolo;
- Una telecamera osserva la scena dall'alto;
- L'altezza dei rettangoli, quindi, determina l'ordine in cui vengono mostrati a schermo.

L'approccio semplificato

Questo approccio è stato abbandonato quando abbiamo scoperto che non è attualmente possibile passare un numero dinamico di *texture* alla *fragment shader*.

Siamo passati ad eseguire una *draw call* per *layer*, rimuovendo la tridimensionalità in quanto non era più necessario sfruttare l'altezza dei *layer* per l'ordine di disegno.