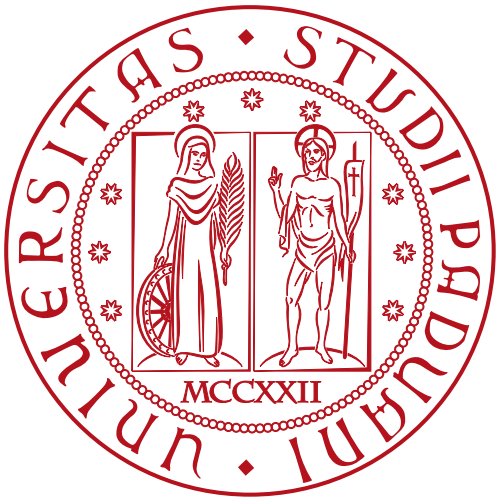
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Ottimizzazione di un renderer 2D mediante integrazione di GPU

Tesi di laurea

Relatore

Prof. Marco Zanella

Laureando

Samuele Esposito

Matricola 2068233

ANNO ACCADEMICO 2024-2025

Samuele Esposito: *Ottimizzazione di un renderer 2D mediante integrazione di GPU*, Tesi di laurea.

When someone says: “I want a programming language in which
I need only say what I wish done”, give him a lollipop.
— Alan J. Perlis

Dedicato ad Arianna.

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di *stage*, della durata di trecentododici ore, dal laureando Samuele Esposito presso l'azienda UNOX S.p.A.

L'obiettivo principale del progetto è stata l'integrazione della *GPU*[G] all'interno di una libreria adibita alla creazione di interfacce grafiche, destinata allo sviluppo del sistema operativo Digital.ID™ installato nei forni di nuova generazione prodotti dall'azienda.

In particolar modo, il documento descrive le nozioni teoriche apprese durante lo svolgimento di suddetto *stage*, introducendo alcuni accenni relativi all'implementazione per fornire al lettore esempi pratici circa le scelte di sviluppo adottate.

Il periodo di *stage* si è articolato in diverse fasi, ognuna della quali ha contribuito al raggiungimento degli obiettivi prefissati:

- **Prima fase**: analizzare e comprendere la struttura del codice sorgente già esistente all'inizio del periodo
- **Seconda fase**: acquisizione delle competenze necessarie per la programmazione di *GPU*[G]
- **Terza fase**: fase di sviluppo destinata all'integrazione della *GPU*[G] all'interno della libreria
- **Quarta fase**: esecuzione di *test* di correttezza per verificare la presenza di problemi, originati dallo sviluppo della medesima integrazione, e analizzato le prestazioni tramite un *frame profiler*[G] in maniera tale da verificare se l'integrazione della *GPU*[G] avesse portato ai miglioramenti previsti.

Indice

Elenco delle Figure

Elenco delle Equazioni

Elenco dei blocchi di codice

Capitolo 1.
Introduzione

1.1. L'azienda

Il progetto di stage si è svolto presso la sede di Cadoneghe dell'azienda UNOX S.p.A. L'azienda si occupa della progettazione e produzione di forni professionali per i settori della ristorazione, del *retail*, della pasticceria, e della panificazione, e vanta 690 dipendenti solo nella sede di Padova.

UNOX si occupa di tutte le fasi del ciclo di vita dei forni, dalla progettazione e produzione alla commercializzazione e vendita. Alcuni dei prodotti presentano un sistema *smart*, denominato *Digital.ID*™, che fornisce funzionalità quali programmazione delle cotture, controllo remoto, o emissione di notifiche in caso di particolari eventi.

1.2. L'idea

Il sistema *Digital.ID*™, attualmente, è installato solo sui forni di fascia più alta che UNOX produce. L'obiettivo dell'azienda è quello di ridurre i costi dell'*hardware* utilizzato da *Digital.ID*™, in maniera tale da poter essere utilizzato anche all'interno dei nuovi forni, garantendo un prezzo di commercializzazione inferiore. Secondo test effettuati internamente, però, l'attuale implementazione adottata dai forni in commercio non è in grado di garantire le prestazioni adeguate, in quanto viene utilizzato *hardware* di fascia più bassa per contenere i costi di produzione, e, di conseguenza, si è reso necessario sostituirla con una che potesse soddisfare le nuove esigenze.

Da alcuni mesi l'azienda sta sviluppando un *renderer*$_{\text{G}}$ 2D$_{\text{G}}$) completamente nuovo, in grado di raggiungere prestazioni migliori rispetto alla piattaforma precedente a parità di *hardware*. Il progetto di stage prevede, quindi, di affiancare il *team* di sviluppo nell'ottimizzazione delle prestazioni, mediante l'integrazione di codice capace di sfruttare la *GPU* del sistema per accelerare le fasi più critiche.

1.3. Organizzazione del testo

Il documento è suddiviso in 2 capitoli:

1. **Introduzione**: viene introdotto il progetto di stage, fornendo una breve spiegazione del contesto che ha portato alla sua creazione. Successivamente, vengono fornite le indicazioni relative alla struttura e alle convenzioni tipografiche adottate dal testo.
2. **Il processo di rendering**: viene esposta l'architettura adottata nello sviluppo del *software*.
3. **Programmazione di GPU**: introduce i concetti fondamentali di come lavorare con le moderne *GPU*.
4. **Composizione in GPU**: presenta l'integrazione della *GPU* all'interno del progetto.

5. **Conclusioni**: espone le considerazioni finali del progetto, portando alla luce i risultati ottenuti, le future possibilità di sviluppo e la valutazione personale relativamente all'esperienza di *stage*.

Nel documento sono state adottate le seguenti convenzioni tipografiche:

- Gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento;
- La prima occorrenza di un termine riportato nel glossario ed individuato all'interno del testo deve presentare la nomenclatura: *parola*$_{G}$
- I termini in lingua straniera o appartenenti al gergo tecnico vengono riportati in *corsivo*.

Capitolo 2.
Il processo di rendering

Prima di iniziare questo progetto, l'azienda era solita sviluppare le proprie interfacce utente utilizzando tecnologie *web* come la libreria React [1]. Usando queste risorse diventa possibile sviluppare delle vere e proprie applicazioni, dotate di grosse quantità di decorazioni e animazioni, ma delegando al *browser*$_G$ dell'utente la visualizzazione a schermo. Dopo aver ponderato a lungo il problema di come implementare un'applicazione simile e dopo numerose consultazioni con il *team* che si occupa di *design*, sono stati identificati i seguenti requisiti di massima:

1. Possibilità di visualizzare **elementi** rettangolari, possibilmente con bordi arrotondati e varie altre decorazioni;
2. Possibilità di inserire **immagini** nell'applicazione;
3. Supporto all'inserimento di **testo**;
4. Consentire lo **scorrimento** delle viste, per esempio quando l'utente utilizza la rotellina del mouse;
5. Deve essere possibile l'inserimento di **effetti**, per esempio sfocatura.
6. **Interattività**, per esempio utilizzando alcuni elementi come bottoni;
7. Il *software* deve essere **multi-piattaforma**, supportando come minimo i forni UNOX e i dispositivi *Android* e *iOS*;
8. Infine, l'utilizzo delle risorse di sistema deve essere relativamente **efficiente**, dato che spesso il *software* verrà eseguito su dispositivi mobili.

A seguito dell'analisi dei requisiti individuati, l'azienda ha rilevato che la maggior parte di essi risulta già implementata nei moderni browser. Di conseguenza, si è deciso di trarre ispirazione dalle architetture sviluppate da questi ultimi nel corso degli ultimi trent'anni. In particolare, è stato adottato come modello Servo [2], un browser sperimentale di nuova generazione.

I *browser engine* moderni sono molto complessi, principalmente a causa di *sandboxing*$_G$ e altre tecniche adottate per proteggere l'utente da possibili siti malevoli. Tuttavia, se ignoriamo queste considerazioni, non necessarie per il progetto, possiamo trovare che l'architettura di *rendering* è organizzata a «stadi» eseguiti in maniera lineare, ossia dove il risultato di uno stadio viene consumato dallo stadio successivo senza che il flusso del programma possa «tornare indietro». Questo ne rende la comprensione più semplice, dato che ogni stadio è indipendente dagli altri se non per le strutture dati utilizzate per trasmettere i risultati. L'architettura adottata dal progetto è mostrata in Figura 1; i nodi rappresentano le strutture dati, mentre gli archi rappresentano i quattro stadi. Seguirà una descrizione di ognuno di essi nelle sezioni successive.

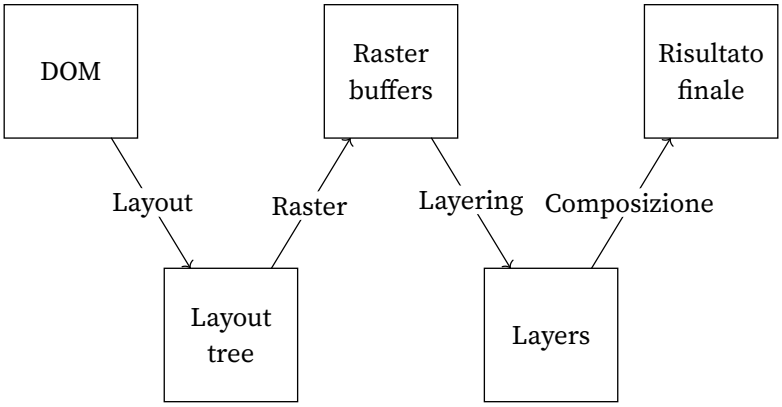


Figura 1: Stadi del processo di *rendering*, e strutture dati di supporto

2.1. Lo stadio di Layout

Il primo stadio del processo di *rendering* è lo stadio di *layout*; esso si occupa di decidere dove posizionare gli elementi dell'interfaccia grafica. L'*input* di questa fase è una descrizione di ogni elemento della pagina e del suo stile, rappresentato mediante un'istanza di uno `struct` (in modo da avere sia più *type-safety* sia maggiori prestazioni rispetto all'uso di una rappresentazione testuale).

Creare un sistema di *layout* efficace è estremamente complicato: oltre a essere corretto ed efficiente, deve anche essere capace di rappresentare ogni possibile desiderio dei *designer*. Fortunatamente, vista sia la presenza di Servo, sia la familiarità del *team* con CSS, è venuto naturale adottare la libreria `taffy` [3] (utilizzata da Servo come implementazione degli algoritmi di CSS) come metodo di *layout*.

L'utilizzo di `taffy` richiede all'utente di fornire un tipo che rappresenti un albero di elementi. Successivamente, per esso devono essere implementati dei *trait* (l'equivalente in Rust di quelle che in altri linguaggi si chiamano interfacce) che sostanzialmente forniscono un'astrazione tale da consentire a `taffy` di visitare tutti i nodi dell'albero in maniera *depth-first*. A partire da esse `taffy` implementa i principali algoritmi di *layout* di CSS, ossia `flexbox`, `grid` e `block`, oltre a un sistema di *caching* che rimuove il più possibile il lavoro ripetuto.

2.2. Lo stadio di Raster

Lo stadio di *raster* è con tutta probabilità il più all'interno dell'applicazione. Esso, dato un componente, la descrizione del suo stile, e l'*output* della fase di *layout* (da cui si ricava, per esempio, la sua dimensione finale), si occupa di produrre un insieme di *pixel* che lo rappresentino.

Per effettuare la rasterizzazione di oggetti anche relativamente complessi (pensiamo per esempio alla rasterizzazione del testo), si è utilizzata una libreria chiamata `tiny-skia`[4]. Come dice il nome, è una versione più piccola di `skia`, una libreria di rendering 2D sviluppata da Google.

2.3. Lo stadio di Layer

La stadio di *layer* si occupa di comporre, a partire da una lista di componenti già rasterizzati, un'unica immagine. Questo approccio consente, quando il componente viene ridisegnato, di ricreare solo il *layer* corrispondente e non tutto lo schermo.

Questo stadio non è strettamente necessario. Tuttavia l'aumento di efficienza che ha portato si è rivelato essere fondamentale per i dispositivi meno potenti.

Lo stadio di *raster*, è utile ricordare, produce in *output* un *array* monodimensionale di *pixel*, che se accoppiato alla larghezza del componente può essere interpretato come immagine bidimensionale. Per unire più immagini all'interno del *buffer* di *layer*, però, è fondamentale considerare che non è possibile effettuare direttamente una copia. Invece, per ogni riga dell'immagine sorgente è necessario copiare i *pixel* all'interno della posizione corretta nell'immagine destinazione, e, successivamente, spostarsi «in basso» di una riga. Per fare ciò, è fondamentale conoscere quella che in gergo tecnico viene chiamata *stride*, ossia la quantità di memoria occupata da una riga dell'immagine. Essa può essere diversa dalla larghezza dell'immagine, in quanto:

1. Un *pixel* può potenzialmente occupare più di un *byte*.
2. È possibile che sia presente spazio di «*padding*» non considerato parte dell'immagine, ma inserito solitamente perché l'*hardware* richiede che ogni riga sia allineata a un certo numero di *byte*.

2.4. Lo stadio di Composizione

Lo stadio di Composizione si occupa di unire i risultati dello stadio di *layer* per produrre un'immagine finale da mostrare a schermo. La distinzione rispetto ad esso sta nel fatto che è lo stadio di Composizione ad occuparsi di applicare i *transform* (ossia scala, rotazione, traslazione).

Questa distinzione è stata esplicitamente inserita per consentire l'implementazione di elementi come *slider* in maniera efficiente. A tal proposito, quando uno *slider* viene spostato dall'utente, l'unica cosa che cambia è la sua posizione; questo significa che, se la posizione viene controllata mediante traslazione, non è necessario rieseguire gli stadi di *raster* e di *layer*. Ovviamente, questa ottimizzazione funziona nel caso in cui gli elementi con trasformazioni appartengono a *layer* separati da quelli degli elementi sottostanti.

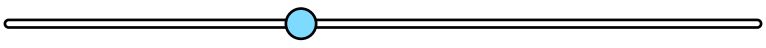


Figura 2: Esempio di *slider*. Il cursore azzurro può essere spostato dall'utente.

È necessario evidenziare che le trasformazioni tridimensionali di CSS non erano ancora implementate, insieme con la trasformazione di rotazione. Questo ha semplificato di molto la fase di composizione, dato che ogni oggetto mantiene una forma rettangolare.

2.4.1. Trasformazione di scala

L'implementazione della traslazione è estremamente semplice (è sufficiente modificare la posizione in cui il *layer* verrebbe inserito); al contrario, invece, l'implementazione della scala è più complicata. In particolare, un'importante consi-

derazione da fare prima di applicarla è: se un oggetto viene scalato, dovrà venire disegnato con il numero di *pixel* finali o con il numero di *pixel* che avrebbe avuto senza scala?

All'interno del progetto è stata adottata la seconda opzione, optando per l'uso di un algoritmo di *resampling* per adattare il numero di *pixel* in modo che fosse corretto. Esistono diversi algoritmi di *resampling* [5], ma i più comuni sono:

- **Nearest-Neighbor**: per ogni *pixel* di destinazione viene copiato il *pixel* sorgente a lui più vicino [6];
- **Bilinear**: ogni *pixel* finale viene calcolato come l'interpolazione tra i quattro *pixel* sorgenti più vicini [7];
- **Bicubic**: ogni *pixel* finale viene calcolato come l'interpolazione tra i nove *pixel* sorgenti più vicini [8];
- **Sinc**: un algoritmo di *resampling* avanzato, che teoricamente fornisce la migliore ricostruzione possibile di un immagine [9];
- **Lanczos**: un'approssimazione di *Sinc*, che nella pratica spesso fornisce risultati migliori [10].

Per il progetto è stato scelto come metodo di *resampling* il *Nearest-Neighbor*, in quanto gli altri metodi risultavano troppo lenti in assenza di supporto dedicato da parte dell'*hardware*, e nella pratica si è notato che il *Nearest-Neighbor* forniva risultati con qualità sufficiente per gli scopi del progetto.

2.5. Architettura del software

Il *software* è relativamente complicato, presentando numerosi moduli e strutture dati. Gli *struct* principali sono:

- `Runtime`: è responsabile dell'interazione tra applicazione e *renderer*. Inoltre, incapsula l'*event loop*$_G$ dell'applicazione.
- `Gui`: si occupa di tutto ciò che concerne l'interazione con il sistema operativo, come la gestione della finestra, la ricezione degli eventi, e la visualizzazione dei risultati del *rendering*.
- `AppUI`: si occupa di eseguire il processo di *rendering*, inviando i risultati a `Gui`.
- `Dom`: si occupa di salvare la lista di componenti; inoltre, incapsula parte della logica di *rendering*.
- `BaseComponent`: rappresenta un componente dell'interfaccia utente, e fornisce metodi per accedere alle sue proprietà. Inoltre, contiene un metodo che esegue il *raster*.

In Appendice A è mostrato il *control flow* del *renderer*, mediante un grafico che mostra il flusso delle chiamate che portano alla generazione di un *frame*.

Capitolo 3.
Programmazione di GPU

Le *GPU* sono dispositivi estremamente potenti e flessibili. Esse sono coprocessori che agiscono in maniera asincrona rispetto al processore principale, il quale si occupa solamente di inviargli comandi e successivamente recuperare il risultato. La natura asincrona è estremamente importante, dato che le *GPU* presentano una memoria separata da quella principale, e i trasferimenti da una all'altra sono relativamente lenti. Inoltre, consente alla CPU_G di eseguire altre attività in attesa del completamento di quelle della *GPU*.

Inizialmente, le *GPU* contenevano solamente una *pipeline* esplicitamente progettata per il *rendering* di grafica $3D_G$. Con il tempo, però, si è vista una vera e propria trasformazione delle *GPU* in processori generici altamente paralleli, consentendo vastissime applicazioni in ambiti scientifici non strettamente legati alla *grafica*. Queste capacità, utilizzabili dall'utente mediante scrittura di *compute shader*, non verranno però discusse in questa tesi, in quanto non sono state utilizzate all'interno del progetto di *stage*. Il lettore interessato può trovarne una spiegazione in [11].

Nel resto di questo capitolo, viene trattata la struttura e l'utilizzo della *pipeline* grafica. È importante ricordare che i dettagli specifici del funzionamento sono differenti in base a quale *API* (Vulkan, OpenGL, ecc.) viene utilizzata; in questa tesi viene discussa la *pipeline* di *WebGPU*, l'*API* adottata dal progetto, che può risultare differente da altre in merito ai dettagli specifici.

3.1. Vertex buffer

La prima fase del *rendering* consiste nella creazione e il popolamento di un *vertex buffer*. Esso è un *array* di strutture definite dal programmatore, solitamente contenenti almeno le coordinate del vertice da disegnare. Ogni vertice che si vuole disegnare sarà rappresentato da un elemento del *vertex buffer*.

Affinché le *vertex shader* (trattate in §3.2) riescano a interpretare questo *buffer*, tuttavia, l'applicazione deve informare la *GPU* di come le strutture al suo interno sono organizzate.

Supponiamo, per esempio, di voler definire una *vertex shader* capace di creare dei rettangoli colorati a schermo. Essa avrà bisogno dei seguenti *input*:

- Il **colore** con cui vogliamo rappresentare il rettangolo. Decidiamo di passarlo in rappresentazione RGBA (ossia con i quattro canali di rosso, verde, blu e trasparenza, ognuno salvato con un *byte*).
- La **posizione** nello spazio tridimensionale, rappresentata con tre interi con segno, ognuno da 1 *byte*;
- La **dimensione** del rettangolo, rappresentata con due interi con segno, sempre di 1 *byte* ciascuno.

Potremmo decidere, quindi, di organizzare il *buffer* come in Figura 3, dove le celle viola contengono il colore del vertice (*RGBA* sta per *red*, *green*, *blue*, *alpha*), le celle

rosse contengono la sua posizione (i tre assi *X*, *Y*, *Z*), e le celle verdi la sua dimensione (*W* sta per *width*, mentre *H* sta per *height*). Si può notare che sono state inserite delle celle vuote (di «*padding*»), per scopi di allineamento. In particolare, è stato deciso di allineare il campo **dimensione** a 2 *byte* per facilitarne la scrittura alla *CPU*. È inoltre presente una dimensione di ogni vertice che è un multiplo di quattro *byte* (in quanto richiesto dalla *GPU*).

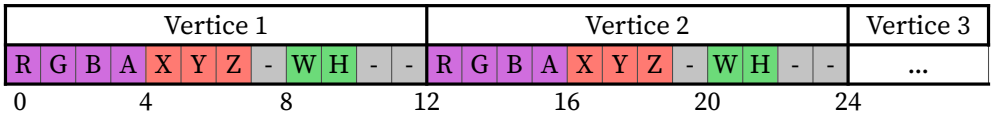


Figura 3: Possibile organizzazione di un *vertex buffer*

Per rendere questo *buffer* comprensibile alla *GPU*, è necesario fornire un oggetto simile a quello rappresentato in Codice 1, per descriverne il *layout*. Si elencano di seguito i campi:

- `attributes`: un *array* di oggetti che descrivono gli attributi del *vertex buffer*. Ogni attributo viene descritto da:
  - `format`, che indica il tipo di dato (per esempio `sint8` o `uint32`), opzionalmente insieme a un numero di canali;
  - `offset`, ossia la distanza tra il primo *byte* del vertice e l'inizio dell'attributo;
  - `shaderLocation`, ossia quale «slot» la *shader* potrà utilizzare per accedere al valore.
- `arrayStride`: indica la dimensione di un singolo vertice. Notare che non è obbligatoriamente uguale alla somma delle dimensioni dei singoli attributi, per esempio se si decide di inserire *byte* di *padding*.
- `stepMode`: può avere come valori `"vertex"` (il default) oppure `"instance"`. Definisce se l'indice del *buffer* deve essere incrementato una volta per ogni vertice, oppure una volta per ogni oggetto nella scena (mantenendo quindi lo stesso indice per tutti i vertici dello stesso oggetto). Dato che noi dobbiamo disegnare quattro vertici a partire dallo stesso *input*, imposteremo `'instance'` come `stepMode`; il default è `'vertex'`,

```
{
  attributes: [
    { format: 'unorm8x4', offset: 0, shaderLocation: 0 }, // colore
    { format: 'snorm8x4', offset: 4, shaderLocation: 1 }, // posizione
    { format: 'snorm8x2', offset: 8, shaderLocation: 2 }, // dimensione
  ];
  arrayStride: 12;
  stepMode: 'instance';
}
```

Codice 1: Esempio di un descrittore del *layout* di un *vertex buffer*

3.2. Vertex shader

Una volta definito il *vertex buffer*, è necessario scrivere una *vertex shader* capace di lavorare con questo *buffer*. Una *shader* è un piccolo programma, solitamente definito in un linguaggio apposito, che la *GPU* esegue tante volte durante durante l'esecuzione

della sua *pipeline* grafica. Le *vertex shader* sono uno dei due tipi principali di *shader*, insieme alle *fragment shader* (di cui discuteremo in §3.4).

Le *shader*, in *WebGPU*, sono definite in un linguaggio apposito, chiamato *WGSL* (*WebGPU Shading Language*). Per definire una *vertex shader* è sufficiente passare il Codice 2 a *WebGPU*: esso definisce una semplice funzione, chiamata `vs_main`, che ritorna un valore di tipo `vec4<f32>`. La funzione è annotata con l'attributo `@vertex`, mentre il suo valore di ritorno è annotato con `@builtin(position)`. All'interno del corpo della funzione, prima viene creata una costante (`let` definisce una variabile non modificabile) con nome `out` e tipo `vec4<f32>`, a cui viene assegnato un vettore contenente quattro volte 1. Successivamente, il valore della variabile `out` viene ritornato.

```
@vertex
fn vs_main() -> @builtin(position) vec4<f32> {
  let out: vec4<f32> = vec4<f32>(1, 1, 1, 1);
  return out;
}
```

Codice 2: La più piccola *vertex shader* valida

Le variabili, in *WGSL*, possono avere i seguenti tipi:

- `i32`, `u32`: interi da 32 bit (4 *byte*), rispettivamente con e senza segno;
- `f16`, `f32`: numeri *floating-point*, rispettivamente da 16 bit (2 *byte*) o 32 bit (4 *byte*);
- `bool`: un valore booleano, `true` oppure `false`.
- `vecN<T>`: un vettore, con `N` componenti di tipo scalare `T`.
- `matCxR<T>`: una matrice di `C` colonne e `R` righe, con componenti di tipo *floating-point* `T`;
- `array<E>`: un *array* con dimensione determinata a runtime, ed elementi di tipo `E`;
- `array<E, N>`: un *array* con dimensione `N`, ed elementi di tipo `E`;
- `struct Nome { field1: type1, field2: type2, ... }`, una struttura.

Gli argomenti di una funzione marcata `@vertex` possono essere di tipo scalare o di tipo struttura. Se scalari, ognuno deve essere marcato con un attributo `@builtin` oppure con un attributo `@location`; se strutture, lo stesso requisito si applica a ognuno dei campi della stessa. Questi due attributi specificano da dove la *GPU* deve recuperare i valori. `@builtin` indica uno tra vari significati «predefiniti» che un attributo può avere. Per gli attributi specificati dall'utente, `@location` indica l'ordine in cui essi vengono passati alla *shader*.

3.2.1. Clip space coordinates

Si può notare che il tipo di ritorno della *shader* in Codice 2 è un vettore con quattro componenti, nonostante venga fatto un *rendering* in tre dimensioni. Ciò avviene in quanto le *vertex shader* ritornino vertici con coordinate espresse in **clip-space**. Ciò significa che esse sono un vettore a quattro componenti:

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix}$$

dove x_c, y_c e z_c definiscono una posizione nello spazio, mentre w_c definisce la dimensione di un cubo, chiamato *clip volume*, in cui tutti i vertici devono essere contenuti per evitare che la *GPU* li scarti.

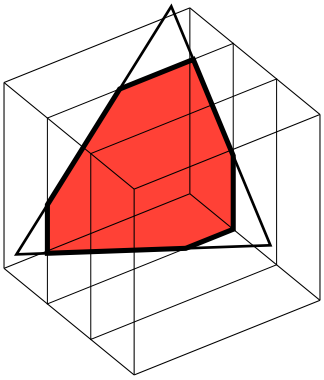


Figura 4: Esempio di *clip space*, con un triangolo parzialmente al suo interno. La parte evidenziata in rosso è quella completamente inscritta nel *clip volume*

L'utilità di questo *clip-space* è semplicemente di consentire alla *GPU* di implementare il *clipping*$_G$ in maniera efficiente, dato che tutti i vertici visibili a schermo rispettano le condizioni

$$\begin{cases} -w_c \leq x_c \leq w_c \\ -w_c \leq y_c \leq w_c \\ 0 \leq z_c \leq w_c \end{cases}$$

dove le prime due condizioni asseriscono che il vertice sia all'interno dell'area dello schermo, mentre l'ultima asserisce che il vertice non sia «troppo vicino» (ossia abbia $z_c < 0$), oppure «troppo lontano» (ossia abbia $z_c > w_c$). In Figura 4 si trova un esempio di *clip-space*. Il triangolo al suo interno è parzialmente occluso, e si può notare che l'area colorata, ossia quella visibile, è quella all'interno del *clip volume*.

Se l'utente volesse implementare proiezioni prospettiche, sistemi di telecamere, o altri sistemi particolari, sarà sua premura applicare all'interno della *vertex shader* una trasformazione, che porti le coordinate dalla forma da lui scelta, a coordinate in *clip-space*.

3.3. Clipping e rasterizzazione

Dopo aver eseguito la *vertex shader* come da §3.2, vi sono una serie di fasi non programmabili (in gergo vengono anche chiamate fasi «*fixed-function*», riferendosi al fatto che la loro funzione è fissa e non scelta dall'utente) che si occupano di prendere il risultato della *vertex shader* (la quale, ricordiamo, opera su vertici presi singolarmente) e di preparare l'*input* per la *fragment shader*.

La prima fase è la fase di **primitive assembly**, che prende i vertici e ne crea una lista di forme geometriche «primitive» (solitamente triangoli, ma esistono *GPU* che ne supportano anche altre).

Successivamente, vengono eliminate le primitive che risultano essere esterne al *clip space*. Qualora una di esse fosse esterna solo parzialmente, allora verrebbe sostituita con un poligono tale da esserne completamente inscritto.

Infine, avviene la fase di **rasterizzazione**. Questa è la fase più complicata di tutta la pipeline di *rendering*, che si occupa di creare, per ogni primitiva che ha passato le fasi precedenti, una lista di *fragment*, in quantità di (almeno) uno per *pixel* dello schermo coperto dalla primitiva. Ognuno contiene una posizione in *device coordinates* (coordinate comprese tra 0 e la dimensione dello schermo), una profondità espressa in numero tra 0 e 1, e altri attributi utili. Inoltre, per ogni attributo *user-defined* (ossia annotato con `@location(N)`) specificato dalla *vertex shader*, il *fragment* conterrà l'interpolazione tra i valori definiti per i vertici della primitiva corrispondente.

3.4. Fragment shader

Le *fragment shader* sono dei piccoli programmi definiti dall'utente con lo scopo di calcolare il colore di un particolare *pixel* delle primitive. Come dice il nome, il loro ingresso è un *fragment*, una struttura dati che rappresenta un «possibile» *pixel*, e contiene vari valori ottenuti interpolando tra i corrispondenti valori nei vertici.

In Codice 3 è presente un esempio di una semplice *vertex shader*, che ritorna sempre un colore rosso completamente opaco (per una introduzione alla sintassi con cui essa è scritta, fare riferimento a §3.2). Rispetto alle *vertex shader*, possiamo far notare alcune differenze:

- L'ingresso della *fragment shader* è marcato come `@builtin(position)`; tuttavia ha un significato molto diverso da quello del valore di ritorno della *vertex shader*. Nelle *vertex shader*, `@builtin(position)` viene interpretato come una coordinata nel *clip-space*, mentre nelle *fragment shader* sono *framebuffer coordinates*, ossia coordinate relative allo schermo.
- L'uscita non è marcata come `@builtin`, ma è un *user-defined output*. Questo perché in fase di configurazione della *GPU* è possibile definire una lista di *color attachment*, ossia descrizioni delle risorse su cui disegnare. Nel caso di Codice 3, è stato definito un solo *render attachment* (quindi con indice 0) dove i *pixel* sono stati rappresentati in formato *RGBA*.

```
@fragment
fn fs_main(@builtin(position) coord_in: vec4<f32>) -> @location(0)
vec4<f32> {
  return vec4<f32>(1.0, 0.0, 0.0, 1.0);
}
```

Codice 3: La più piccola *fragment shader* valida

Può essere desiderabile, inoltre, applicare anche delle immagini alla superficie dei nostri triangoli; fortunatamente, è un desiderio così comune che le *GPU* presentano supporto specifico per esse. In particolare, è possibile creare dei *buffer*, diversi dai *vertex buffer* o dai *color attachment*, a cui la *shader* può accedere come se fossero variabili globali. Questi *buffer* prendono il nome di ***texture***. Possiamo quindi modificare la nostra *fragment shader* per inserire due diverse variabili globali:

- `texBuffer` di tipo `texture_2d<f32>`, che conterrà i nostri *pixel*;
- `texSampler` di tipo `sampler`, utilizzata per interpretare i dati della *texture*.

L'utilizzo di una *texture* è relativamente semplice, come mostrato in Codice 4. Per ogni *fragment*, è sufficiente chiamare la funzione *built-in* `textureSample`, passando

una *texture*, un *sampler*, e delle coordinate bidimensionali che indichino da quale punto della *texture* recuperare il *pixel* (o, qualora il punto non fosse esattamente all'interno di un *pixel*, la media tra i *pixel* adiacenti) Queste coordinate saranno fornite dalla *vertex shader* come un secondo attributo, `texcoord`, da aggiungere a `VertexOutput`.

Ovviamente, però, la *vertex shader* verrà eseguita per ogni vertice, e non per ogni *fragment* all'interno della primitiva. Di conseguenza, la *GPU* eseguirà per ogni *fragment* una interpolazione tra i valori calcolati per ogni vertice, e utilizzerà i valori calcolati come *input* della *fragment shader*.

```
struct VertexOutput {
  @builtin(position) position: vec4f,
  @location(0) texcoord: vec2f,
};
@group(0) @binding(0) var ourSampler: sampler;
@group(0) @binding(1) var ourTexture: texture_2d<f32>;

@vertex fn vert_main(
  @builtin(vertex_index) vertexIndex : u32
) -> VertexOutput {
  let pos = array(
    vec2f(0.0,  0.0),
    vec2f(1.0,  0.0),
    vec2f(0.0,  1.0),
    vec2f(0.0,  1.0),
    vec2f(1.0,  0.0),
    vec2f(1.0,  1.0),
  );

  var output: VertexOutput;
  let xy = pos[vertexIndex];
  output.position = vec4f(xy, 0.0, 1.0);
  output.texcoord = xy;
  return output;
}

@fragment fn fs_main(input: VertexOutput) -> @location(0) vec4f {
  return textureSample(ourTexture, ourSampler, input.texcoord);
}
```

Codice 4: Esempio di *shader* che applicano una *texture* ad un rettangolo.

Capitolo 4.
Composizione in GPU

L'obiettivo dello *stage* è stato incrementare le prestazioni del *renderer* mediante l'utilizzo della *GPU*. Nelle fasi iniziali si è preferito limitare l'ambito del progetto alla sola realizzazione dello stadio di Composizione, lasciando potenzialmente *performance* non sfruttate, ma riducendo anche il numero di cambiamenti da fare a codice preesistente.

Inizialmente, si è pianificato di applicare un approccio ispirato a quelli adottati per il *rendering* 3D, con l'obiettivo di migliorare l'efficienza; successivamente, però, è stato possibile notare che questo approccio non era realizzabile. Di conseguenza, è stato necessario cambiare l'approccio stesso, passando a una implementazione più semplice.

4.1. Approccio iniziale: grafica 3D

Le *GPU* sono state progettate per la grafica 3D, e utilizzarle per produrre grafica 2D non è, strettamente parlando, ciò per cui sono state inizialmente realizzate. Inizialmente, quindi, si è pensato di produrre una scena tridimensionale che rappresentasse una specie di «pila» di *layer*, in maniera tale da implementare successivamente una «telecamera» che riprendesse la scena dall'alto in proiezione ortografica, producendo il risultato desiderato. Viene illustrato un sistema simile a quello implementato in Figura 6.

Per fare ciò, è stato scritto del codice che calcolasse per ogni *layer*, la dimensione e la posizione; questa informazione è stata inserita all'interno di un *vertex buffer*, e una *vertex shader* estremamente semplice produceva successivamente i quattro vertici di ogni rettangolo. Ogni rettangolo viene rappresentato con una coppia di triangoli rettangoli, disposti in modo tale da avere le ipotenuse coincidenti, e i cateti paralleli a uno dei due assi X o Y. Inoltre, si è optato di organizzare i vertici in maniera antioraria, dato che *WebGPU* usa il senso di rotazione per determinare quale tra le due facce di un triangolo è visibile. Un esempio di questa rappresentazione si può trovare in Figura 5.

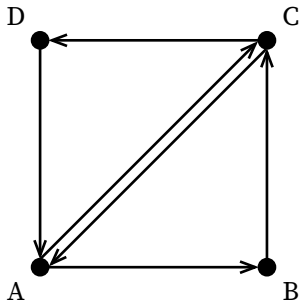


Figura 5: Triangolizzazione di un rettangolo, con vertici in senso antiorario

Lo svantaggio di questo approccio è emerso, però, nel momento in cui si è tentato di visualizzare delle immagini all'interno di questi *layer*. Infatti, in *WebGPU*, non è

al momento possibile associare un *array* di *texture* come variabile di ingresso per una *shader*. Nel caso ciò fosse stato possibile, sarebbe stato possibile disegnare tutti i *layer* nello stesso comando per la *GPU*, utilizzando l'asse Z per codificare l'ordine degli elementi.

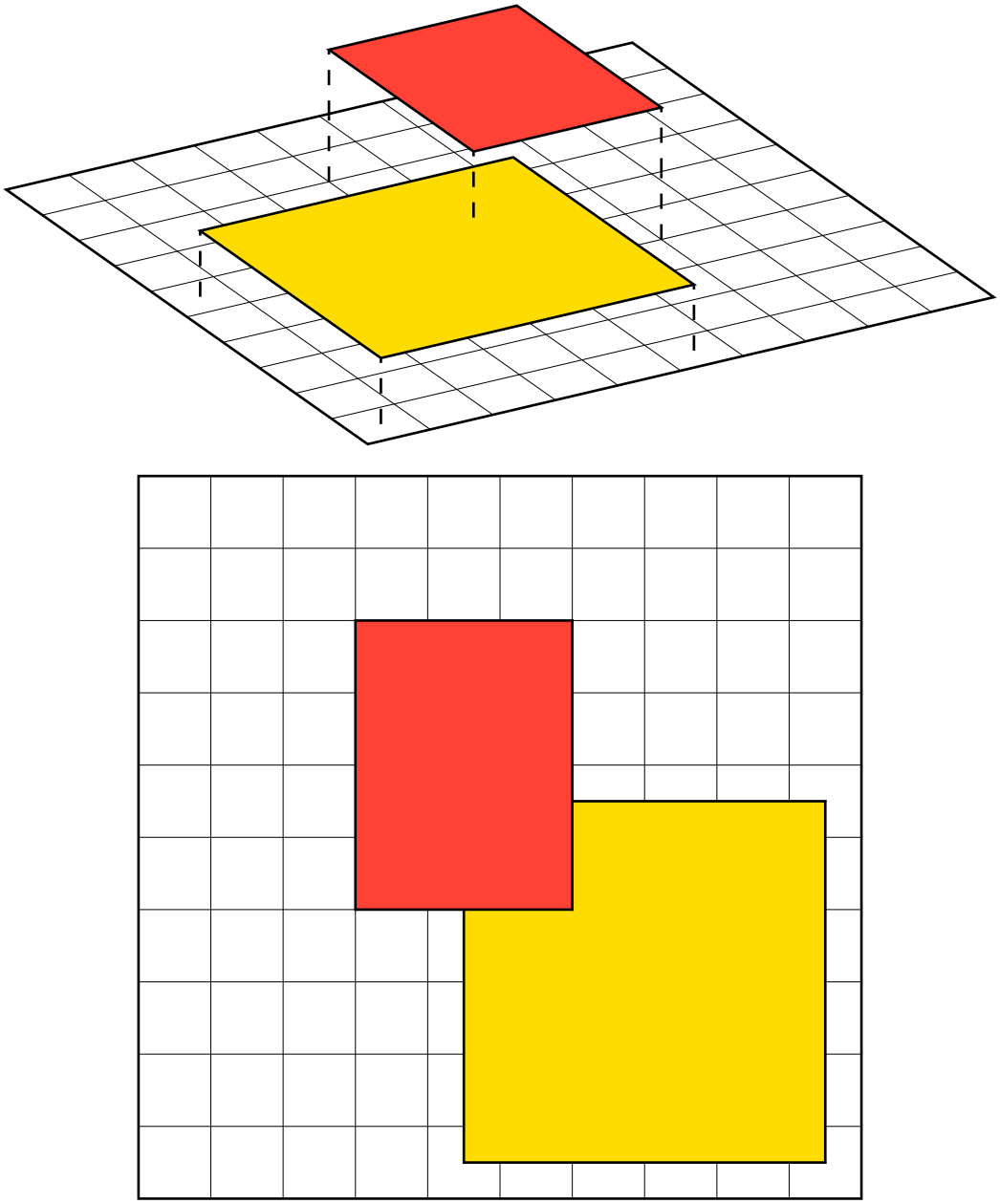


Figura 6: Esempio di composizione di layer (colorati) sullo schermo

4.2. Approccio semplificato

Nella sezione precedente si è discusso di come risultasse impossibile fornire un numero dinamico di *texture* alla stessa *draw call*; di conseguenza, è risultato necessario eseguire una *draw call* per ogni *layer*.

Questo si è rivelato essere notevolmente più lento, in quanto per ogni *layer* è necessario, prima di effettuare ogni *draw call*, attendere che quella precedente sia terminata.

Il vantaggio, ovviamente, è che non è più necessario calcolare la coordinata Z, ma risulta sufficiente effettuare le *draw call* in maniera ordinata.

4.3. Trasformazioni degli elementi mediante matrici

Come discusso precedentemente, i requisiti del progetto includevano la possibilità di applicare trasformazioni agli elementi senza dover eseguire nuovamente le fasi di *layout*, rasterizzazione e *layering*, in quanto relativamente costose.

Le trasformazioni richieste sono:

- Traslazione, ovvero spostamento di un elemento in una posizione diversa;
- Rotazione, ovvero rotazione di un elemento attorno a un asse (nel nostro caso solamente l'asse Z, essendo il nostro spazio «utile» bidimensionale);
- Scala, ovvero ingrandimento o rimpicciolimento di un elemento.

Le trasformazioni di rotazione e scala sono trasformazioni lineari, ossia trasformazioni che possono essere rappresentate mediante una matrice.

4.3.1. Trasformazioni lineari

Consideriamo il segmento bidimensionale $\overline{\mathrm{AB}}$, presente in Figura 7. Se esso viene ruotato di 90 gradi attorno all'origine in senso orario, allora il punto A si sposterà in A', e il punto B si sposterà in B'. La trasformazione che ha portato da A a A' e da B a B' può essere espressa mediante una matrice, che chiameremo R_θ, che ha come effetto quello di ruotare i punti di θ gradi attorno all'origine.

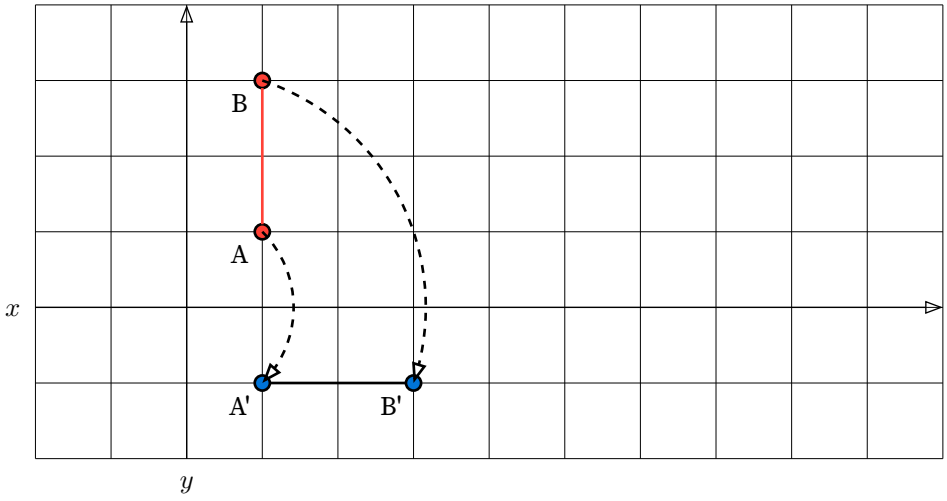


Figura 7: Esempio di rotazione di un segmento attorno all'origine

La matrice di rotazione in senso antiorario R_θ è definita come segue, dove θ è l'angolo di rotazione:

$$R_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Equazione 1: Matrice di rotazione di θ gradi in senso antiorario

Se esprimiamo i punti A e B come vettori colonna, possiamo allora effettuare una moltiplicazione matrice-vettore per ottenere i punti trasformati A' e B':

$$A = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad B = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

$$A' = R_{-90°} \cdot A = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$$

$$B' = R_{-90°} \cdot B = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 3 \end{bmatrix} = \begin{bmatrix} 3 \\ -1 \end{bmatrix}$$

Questo tipo di trasformazione, ottenibile mediante un prodotto matrice-vettore, è chiamata trasformazione lineare.

4.3.2. Trasformazioni affini

Mediante trasformazione lineare non è possibile effettuare traslazioni, ossia spostamenti di un elemento in una posizione diversa. Questo è dovuto al fatto che le trasformazioni lineari sono, in sintesi, delle moltiplicazioni; se applichiamo una trasformazione lineare al vettore nullo che rappresenta l'origine, otterremo ancora l'origine, dato che ogni numero moltiplicato per zero è zero.

Per poter effettuare anche le traslazioni, è necessario estendere il concetto di trasformazione lineare a quello di trasformazione affine. Una trasformazione affine è una trasformazione ottenuta sommando un vettore al risultato di una trasformazione lineare. In Figura 8 è presente un esempio di una trasformazione affine, la rototraslazione.

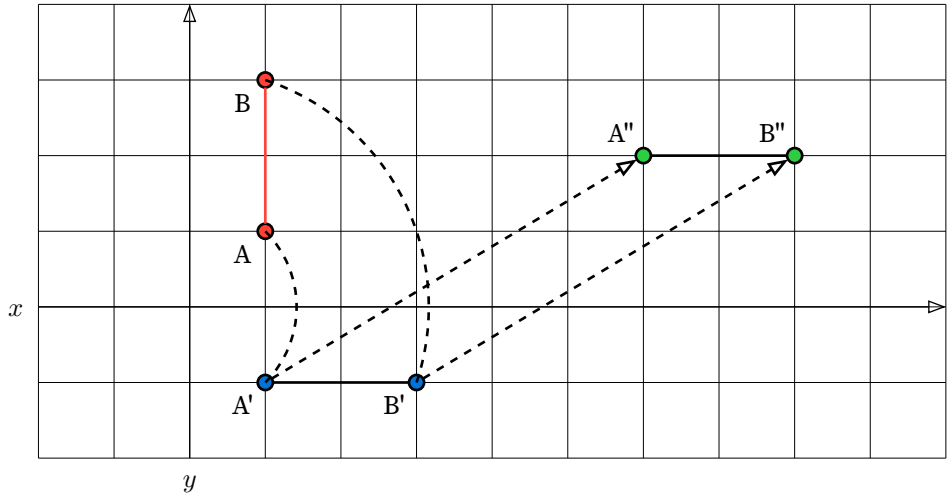


Figura 8: Esempio di rototraslazione, ossia rotazione seguita da traslazione

La rototraslazione in Figura 8 è rappresentata dalla matrice di rotazione definita precedentemente, seguita da una traslazione, come si vede in Equazione 2.

$$R_{-90^\circ} = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \qquad t = \begin{bmatrix} 5 \\ 3 \end{bmatrix}$$

$$A'' = A' + t = A \cdot R_{-90^\circ} + t = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 5 \\ 3 \end{bmatrix} = \begin{bmatrix} 6 \\ 2 \end{bmatrix}$$

$$B'' = B' + t = B \cdot R_{-90^\circ} + t = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 3 \end{bmatrix} + \begin{bmatrix} 5 \\ 3 \end{bmatrix} = \begin{bmatrix} 8 \\ 2 \end{bmatrix}$$

Equazione 2: Esempio di applicazione della rototraslazione

4.3.3. Combinazione di trasformazioni

È possibile combinare più trasformazioni lineari insieme per ottenerne una più complessa. Questo è possibile effettuando un prodotto matrice-matrice, dove la matrice destra è la rappresentazione della prima trasformazione, e la matrice sinistra è la rappresentazione della seconda trasformazione.

Supponiamo, per esempio, di voler effettuare una trasformazione di rotazione, seguita da una trasformazione di scala. Oltre a poter applicare le due trasformazioni in sequenza (quindi moltiplicare la coordinata prima con la matrice di rotazione, e poi con quella di scala), è possibile moltiplicare le due matrici assieme, ottenendo così una singola matrice che rappresenta entrambe le trasformazioni in un passo soltanto.

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \quad v = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$$

$$BAv = (BA)v = \left(\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} \cdot \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \right) \cdot v$$

Equazione 3: Combinazione di trasformazioni mediante proprietà associativa. Applicare la trasformazione C è equivalente ad applicare in sequenza A e B.

Quanto rappresentato in Equazione 3 è possibile perché il prodotto tra matrici gode della proprietà associativa, che permette di raggruppare le operazioni in modo arbitrario senza che il risultato cambi.

4.3.4. Trasformazioni nello spazio tridimensionale

Desideriamo ora estendere le trasformazioni elencate precedentemente in modo da poterle applicare anche nello spazio tridimensionale. È sufficiente, per questo, aggiungere un terzo componente ai vettori che rappresentano le coordinate, e definire nuove matrici che abbiano dimensione 3×3, al posto di 2×2.

Come discusso in §3.2.1, però, le *GPU* utilizzano uno spazio quadridimensionale per rappresentare le coordinate dei vertici. Questo risulta utile in quanto è possibile sfruttare un espediente per poter trasformare la traslazione tridimensionale (che ricordiamo essere non lineare), in una trasformazione lineare quadridimensionale con effetto simile.

4.3.5. Matrici di trasformazione

Segue una lista delle più importanti matrici di trasformazione.

4.3.5.1. Trasformazione identità

La trasformazione di identità mappa ogni punto dello spazio a sé stesso. È rappresentata dalla matrice identità, mostrata in Equazione 4 [12].

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equazione 4: Matrice per la trasformazione identità

4.3.5.2. Trasformazione di scala

La trasformazione di scala ingrandisce o rimpicciolisce gli elementi, avvicinando o allontanando ogni punto all'origine (come può essere visto in Figura 9). Può essere implementata con la matrice in Equazione 5 [12].

$$S = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equazione 5: Matrice per la trasformazione di scala

Il quarto componente della diagonale della matrice è volutamente lasciato ad 1, in quanto le trasformazioni che applichiamo non devono occuparsi della coordinata w.

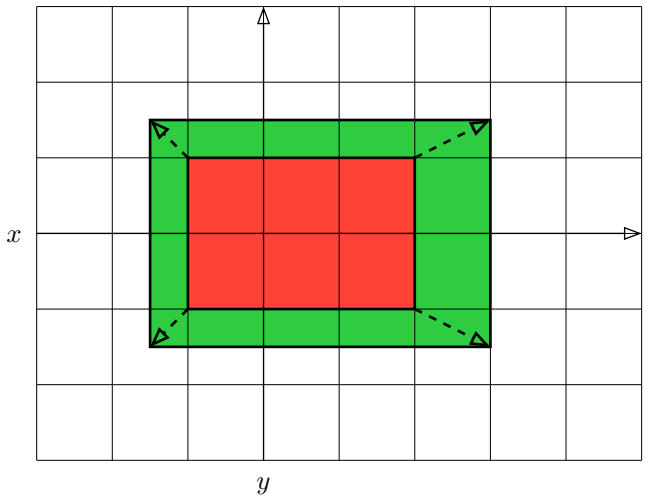


Figura 9: Esempio di trasformazione di scala

4.3.5.3. Trasformazione di rotazione attorno all'asse Z

La trasformazione di rotazione attorno all'asse Z può essere implementata con la matrice mostrata in Equazione 6, dove θ rappresenta l'angolo di rotazione espresso in radianti [12]. Un esempio di rotazione può essere trovato in Figura 7.

$$R = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equazione 6: Matrice per la trasformazione di rotazione

4.3.5.4. Trasformazione di traslazione

La trasformazione di traslazione può essere implementata con la matrice in Equazione 7, dove t_x, t_y, t_z rappresentano la quantità di cui traslare su ogni asse [12].

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Equazione 7: Matrice per la trasformazione di traslazione

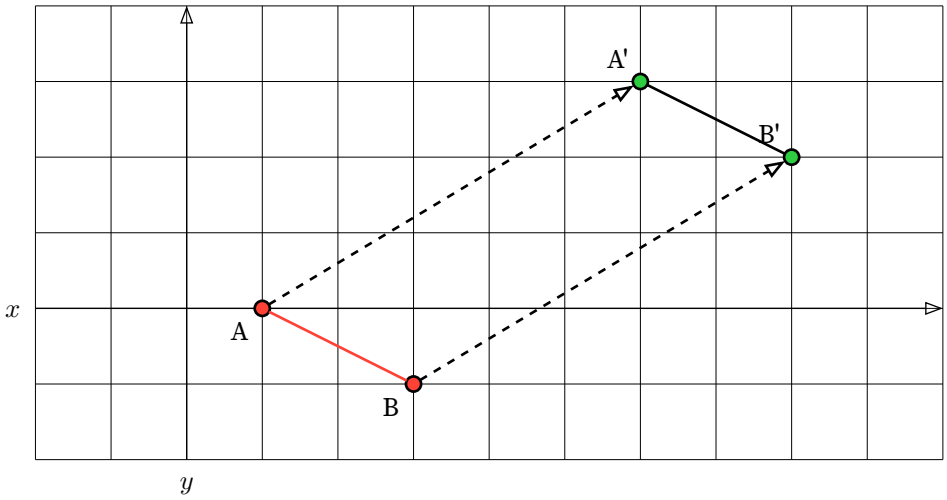


Figura 10: Esempio di traslazione di un segmento

4.4. Scorrimento all'interno di un elemento

Oltre ad avere trasformazioni di tutto l'elemento, è necessario fornire a un elemento la possibilità di «scorrere». Pensiamo, per esempio, a un componente che rappresenti una lista potenzialmente molto grande di elementi. Sarà necessario che questo componente abbia una dimensione fissa, in modo da poterlo inserire a schermo in una posizione determinata, però gli elementi al suo interno devono scorrere con esso.

Questa trasformazione, non è combinabile con le trasformazioni elencate precedentemente, in quanto non desideriamo spostare i vertici di un elemento ma la *texture* all'interno dello stesso, né è implementabile mediante traslazione, dato che ciò potrebbe occludere parte del resto della vista. Il principio generale resta comunque lo stesso, ossia fornire alla *vertex shader* una matrice che lei applicherà alle coordinate della *texture*. In questo caso, abbiamo optato di utilizzare una trasformazione affine, dato che le coordinate della *texture* sono bidimensionali.

Un esempio di una *vertex shader* che implementa questa trasformazione è mostrato in Codice 5.

```
@vertex
fn vs_main(
    model: VertexInput,
    instance: InstanceInput,
) -> VertexOutput {
    let model_matrix = mat4x4<f32>(
        instance.model_matrix_0,
        instance.model_matrix_1,
        instance.model_matrix_2,
        instance.model_matrix_3,
    );
    var out: VertexOutput;
    out.tex_coords = model.tex_coords + instance.view_origin;
    out.clip_position = model_matrix * vec4<f32>(model.position, 1.0);
    return out;
}
```

Codice 5: *vertex shader* che applica uno scorrimento alle coordinate della *texture*

4.5. Codice finale

In Appendice B è presente il codice finale della *shader* che implementa la composizione degli elementi. Per integrare questa *shader*, abbiamo inserito, all'interno del progetto, il metodo `Dom::compose_gpu()` che andasse a sostituire il metodo `Dom::compose_pixels()` che si può vedere in Appendice A. La scelta tra i due metodi viene effettuata a tempo di compilazione, a seconda alla presenza o assenza dell'opzione di compilazione `compose_gpu`.

Se questa opzione è presente, allora il costruttore di `Dom` viene modificato per includere l'inizializzazione della *GPU*, e viene emesso il metodo `Dom::compose_gpu()`. Se, invece, l'opzione non è presente, allora il metodo `Dom::compose_pixels()` viene emesso, a prendere il posto del metodo `Dom::compose_gpu()`.

Capitolo 5.
Conclusioni

Il progetto di *stage* svolto presso l'azienda UNOX S.p.A. ha prodotto risultati interessanti per quanto riguarda le prestazioni del *rendering* di interfacce grafiche.

È stato possibile implementare un sistema di composizione che permette di ottenere prestazioni migliori rispetto al sistema precedente, in quanto il *rendering* avviene in modo più efficiente sfruttando le capacità della *GPU*.

A discapito delle previsioni, i tentativi iniziali di sviluppo dell'integrazione non hanno portato a risultati concreti: l'approccio inizialmente adottato per la composizione in *GPU* si è rilevato essere non implementabile a causa delle limitazioni dello standard di *WebGPU*. Nonostante ciò, la problematica è stata risolta optando per una composizione progressiva degli elementi, gestiti singolarmente tramite il sistema di composizione.

5.1. Possibili sviluppi futuri

Come discusso in §4.1, l'approccio attualmente adattato è relativamente lento, in quanto effettua una *draw call* separata per ogni *layer*. Una possibile soluzione sarebbe quella di implementare «batching»: ogni *draw call* potrebbe comporre fino a un certo numero (fisso, non variabile a *runtime*) di *layer* nello stesso passaggio, ognuno con la propria *texture*. Nella pratica, però, questo è sembrato non essere un grande collo di bottiglia; al contrario, l'impatto maggiore sulle prestazioni sembrava essere il caricamento sulla *GPU* delle *texture* dei *layer* (in quanto un *layer* può essere potenzialmente anche molto grande).

Una soluzione che avevamo proposto era quella di effettuare anche la fase di *layering* sulla *GPU*; così facendo, sarebbe stato necessario caricare solamente i risultati della fase di *raster*, con la conseguenza che, qualora uno di essi venisse modificato (pensiamo, per esempio, al cambio di colore di un bottone quando viene cliccato), il suo *buffer* (potenzialmente molto più piccolo del *buffer* di *layer*) sarebbe l'unico da caricare.

Un altro potenzialmente miglioramento sarebbe quello di effettuare anche la fase di *raster* in *GPU*, sfruttando librerie come Vello [13] create appositamente per questo scopo.

Tuttavia, non è stato possibile implementare nessuno di questi miglioramenti, poiché avrebbero richiesto importanti *refactor* del codice esistente, non sostenibili da effettuare nei tempi imposti per la consegna del *Proof of Concept* imposto dagli *stakeholder* al *team* di sviluppo *software*.

5.2. Valutazione personale

Personalmente, ritengo estremamente interessante il progetto di *stage*: in Italia individuare un progetto che combini la programmazione di *GPU* o utilizzare linguaggi avanzati come *Rust* è estremamente raro. Nonostante le conoscenze di base da me

possedute relative a queste due tecnologie, questo progetto ha sicuramente contribuito a consolidarle.

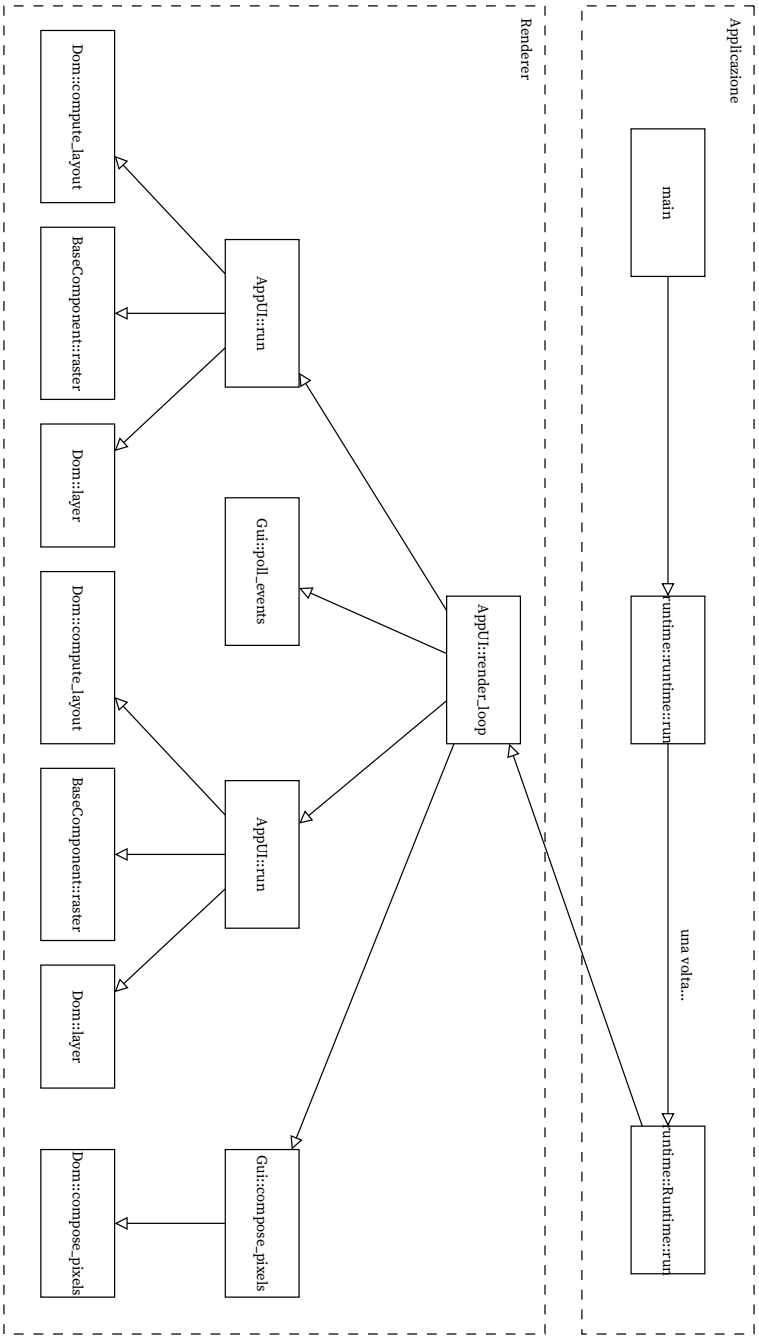
Tuttavia, il codice su cui è stato sviluppata la funzionalità oggetto della tesi risultava inadatto a tale scopo, in quanto lo sviluppo dello stesso non era stato progettato per l'integrazione della *GPU*.

Ritengo inoltre che, nonostante la natura del progetto, ovvero un prototipo non pienamente stabile, siano state prese delle decisioni a mio parere poco adatte e che, se evitate, avrebbero potuto portare valore aggiunto al prodotto finale al termine dello *stage*.

In conclusione, nonostante le problematiche riscontrate durante il periodo di tirocinio, ritengo che quest'ultimo sia risultato interessante e formativo, permettendomi di consolidare le conoscenze in merito alle tecnologie utilizzate.

Appendice A.

Architettura del software



Appendice B.
Codice finale della shader

```
struct VertexInput {
    @location(0) position: vec3<f32>,
    @location(1) tex_coords: vec2<f32>,
}

struct InstanceInput {
    @location(2) view_origin: vec2<f32>,
    @location(3) view_size: vec2<f32>,
    @location(4) model_matrix_0: vec4<f32>,
    @location(5) model_matrix_1: vec4<f32>,
    @location(6) model_matrix_2: vec4<f32>,
    @location(7) model_matrix_3: vec4<f32>,
}

struct VertexOutput {
    @builtin(position) clip_position: vec4<f32>,
    @location(0) tex_coords: vec2<f32>,
}

@vertex
fn vs_main(vert: VertexInput, inst: InstanceInput) -> VertexOutput {
    let model_matrix = mat4x4<f32>(
        inst.model_matrix_0,
        inst.model_matrix_1,
        inst.model_matrix_2,
        inst.model_matrix_3,
    );
    var out: VertexOutput;
    out.tex_coords = vert.tex_coords * inst.view_size + inst.view_origin;
    out.clip_position = model_matrix * vec4<f32>(vert.position, 1.0);
    return out;
}

@group(0) @binding(0) var texture: texture_2d<f32>;
@group(0) @binding(1) var texture_sampler: sampler;

@fragment
fn fs_main(in: VertexOutput) -> @location(0) vec4<f32> {
  return textureSample(
    texture,
```

```
    texture_sampler,
    in.tex_coords
  );
}
```

Glossario

2D di qualsiasi cosa che sia bidimensionale; solitamente usato in ambito grafico.
3D di qualsiasi cosa che sia tridimensionale; solitamente usato in ambito grafico.
Browser programma utilizzato per visualizzare siti *web*.
Clipping il processo di eliminare o ignorare gli elementi non visibili a schermo.
CPU *Central Processing Unit*, il processore principale di ogni computer.
Event loop il ciclo che si occupa di ricevere gli eventi dal sistema operativo, e di inviarli alle parti del *software* che li devono gestire.
GPU *Graphical Processing Unit*, un coprocessore utilizzato per accelerare le operazioni grafiche, riducendo il carico di lavoro del processore principale.
Renderer *software* che, data una descrizione di una scena, realizza un'immagine comprensibile da un umano.
Sandboxing tecnica che permette di isolare un programma dal resto del sistema.

Bibliografia

[1] «React: a library for _web_ and native user interfaces». [Online]. Disponibile su: https://react.dev/

[2] «Servo: an experimental browser engine». [Online]. Disponibile su: https://servo.org/

[3] «Taffy: a flexible, high-performance library for UI layout». [Online]. Disponibile su: https://github.com/DioxusLabs/taffy

[4] «Tiny-skia: a tiny Skia subset ported to Rust». [Online]. Disponibile su: https://github.com/linebender/tiny-skia

[5] «Comparison gallery of image scaling algorithms». [Online]. Disponibile su: https://en.wikipedia.org/wiki/Comparison_gallery_of_image_scaling_algorithms

[6] «Nearest-neighbor interpolation». [Online]. Disponibile su: https://en.wikipedia.org/wiki/Nearest-neighbor_interpolation

[7] «Bilinear interpolation». [Online]. Disponibile su: https://en.wikipedia.org/wiki/Bilinear_interpolation

[8] «Bicubic interpolation». [Online]. Disponibile su: https://en.wikipedia.org/wiki/Bicubic_interpolation

[9] «Whittaker-Shannon interpolation formula». [Online]. Disponibile su: https://en.wikipedia.org/wiki/Whittaker%E2%80%93Shannon_interpolation_formula

[10] «Lanczos resampling». [Online]. Disponibile su: https://en.wikipedia.org/wiki/Lanczos_resampling

[11] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, e J. Phillips, «GPU computing», *Proceedings of the IEEE*, vol. 96, pp. 879–899, 2008, doi: 10.1109/JPROC.2008.917757.

[12] M. A. e Chiara de Fabritiis, *Geometria analitica con elementi di algebra lineare, 2a edizione*. Milano: McGraw-Hill, 2010.

[13] «Vello: a GPU compute-centric 2D renderer». [Online]. Disponibile su: https://github.com/linebender/vello