PROGETTO COMPUTER GRAPHICS
SIMPLE GPU RAYTRACER
A.A. 2023/2024

Leonardo Temperanza (196836)

Indice dei contenuti

Descrizione generale

È stato realizzato un programma per il rendering di immagini realistiche utilizzando la tecnica del raytracing, ossia tracciando all'inverso il percorso che potrebbe seguire un raggio di luce prima colpire un'immaginaria lente. Il renderer è stato progettato per sfruttare l'architettura parallela delle schede grafiche, ed eseguire in parallelo le computazioni relative a numerosi pixel contemporaneamente. Per fare ciò, è stato scritto un programma (shader) nel linguaggio GLSL che opera sui pixel, ovvero un fragment shader, e dunque il graphics API OpenGL.

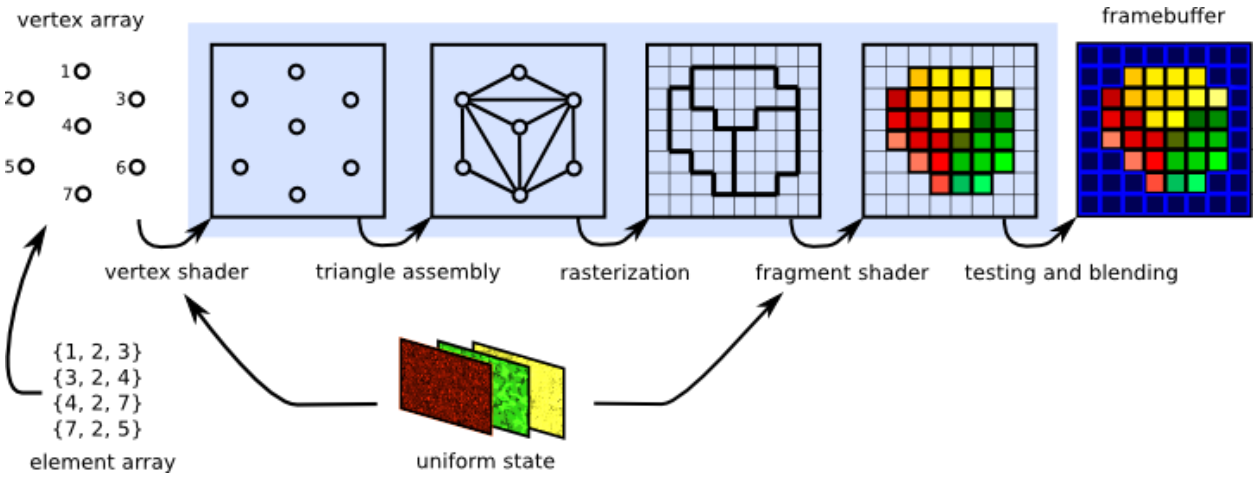

Figura 1 Pipeline Grafica OpenGL. *Tratta da: https://graphicscompendium.com/intro/01-graphics-pipeline*

La tipica pipeline in computer grafica raster comprende vari stadi: vengono inizialmente processati vertici di triangoli, per poi arrivare ad eseguire computazioni per ciascun pixel sullo schermo. In questo corso ci si occupa di grafica realistica, ovvero renderizzata mediante tecniche derivate da raytracing, dunque questa pipeline non viene utilizzata direttamente. Al contrario, si utilizza propriamente solamente lo stadio di fragment shader, fornendo invece come unici vertici quelli di un rettangolo che ricopre l'intero schermo.

Descrizione della scena

Le informazioni della scena, di fatto, sono memorizzate all'interno del programma stesso come: una lista di sfere, una lista di quad, un environment map. La scena è memorizzata nel seguente modo:

```
struct Sphere
{
    vec3 pos;
    float rad;
    Material mat;
};


struct Quad
{
    vec3 p[4];
    vec3 coords[4];
    Material mat;
};
```

Una sfera è specificata con la usa origine nello spazio globale, il suo raggio, e il materiale associato ad essa. Un quad è formato da due triangoli rivolti nella stessa direzione, e vengono specificati i vertici nello spazio globale, le texture coordinate associate a ciascun vertice, e il suo materiale.

I materiali sono specificati nel seguente modo:

```
struct Material
{
    uint matType;
    vec3 emissionScale;
```

```
    vec3 colorScale;
    float roughnessScale;
    uint emission;
    uint color;
    uint roughness;
};
```

È dunque caratterizzato da una determinata tipologia, che può essere una delle seguenti:

- Matte
- Reflective
- Glossy
- Transparent

Inoltre presenta alcune proprietà, sia in forma di moltiplicatore scalare sia in forma di texture: color, roughness, emission. Queste proprietà possono essere utilizzate o meno a seconda della tipologia di materiale.

Per associare le texture a ciascun materiale, si utilizza un identificatore numerico in modo tale che, nell'esecuzione dell'algoritmo di raytracing, si possa dinamicamente effettuare il sampling della texture appropriata. In particolare, è stata utilizzata una funzionalità di OpenGL denominata "Texture Arrays", che sostanzialmente permette di effettuare il sampling di una texture tridimensionale. In questo caso la terza dimensione viene puramente utilizzata per le diverse texture presenti all'interno della scena. Vi sono due texture array: una per le environment-map, e una per le texture utilizzate per color, emission e roughness. Le texture all'interno di una texture array devono essere uniformi.

Il comportamento dei materiali è tratto dal capitolo 5 del libro LittleCG.

Funzionalità implementate

Come già menzionato, il programma è in grado di renderizzare realisticamente scene descritte come una sequenza di sfere e di quad dotati di materiali, utilizzando l'algoritmo di raytracing montecarlo.

In questo paragrafo si descriveranno le funzionalità più interessanti tra quelle sviluppate.

Anti-aliasing

L'aliasing (o "stair stepping") è un fenomeno in computer grafica che consiste nell'apparenza "a gradini" di linee diagonali. Gli algoritmi che permettono di ridurre o eliminare questo fenomeno vengono detti algoritmi di anti-aliasing. In un raytracer montecarlo si fa affidamento alla generazione di numeri casuali per riprodurre i fenomeni legati ai raggi luce in maniera realistica, pertanto l'anti-aliasing è facilmente implementabile, perturbando leggermente la direzione dei raggi che vengono lanciati a partire da ciascun pixel. L'immagine di seguito dovrebbe mostrare la differenza dell'utilizzo di anti-aliasing (a parte eventuale compressione delle immagini).

Figura 2 Dimostrazione Anti-aliasing in una scena di testing

Progressive rendering

Per renderizzare immagini realistiche spesso vi è la necessità di mandare numerosi raggi dalla telecamera, ad esempio 300 raggi per pixel. A seconda della scena e dei materiali utilizzati, i raggi possono rimbalzare numerose volte prima di raggiungere l'environment-map (o prima di arrivare alla soglia di numero massimo di rimbalzi). Per questi motivi il processo di convergenza verso un'immagine accettabile potrebbe essere molto lento e oneroso.

Questo va in contrasto con la volontà di poter apportare modifiche alla scena in maniera iterativa, e possibilmente in maniera veloce. Tipicamente per conciliare questi due aspetti si fornisce la possibilità di visualizzare immediatamente un render grezzo della scena, che viene poi rifinito con il passare del tempo.

Questo viene realizzato nel progetto facendo uso di due immagini che vengono utilizzate

interscambiabilmente (ping pong buffers). Al termine di ciascun render, il risultato finale viene ottenuto calcolando una media pesata tra le due immagini. Il calcolo viene fatto nel seguente modo:

```
vec4 curColor = vec4(finalColor, 1.0f);

if(frameAccum != 0)

{

    Float weight = 1.0f / float(frameAccum);

    Vec4 prevColor = texture(previousFrame, texCoords);

    fragColor = prevColor * (1.0f – weight) + curColor * weight;

}

else

    fragColor = curColor;
```

Depth of field

Inizialmente il renderer non teneva conto di una proprietà delle telecamere e dell'occhio umano, la messa a fuoco.

Per implementare questo effetto è necessario definire due parametri: una distanza di fuoco e un raggio di apertura. Il punto focale di un pixel si può determinare prendendo l'origine usuale del raggio, e percorrendo la distanza focale nella direzione usuale del raggio. La distanza focale è la distanza per cui la telecamera riesce a vedere in maniera perfettamente nitida. Dopodichè si può generare un punto casuale attorno all'origine usuale del raggio, e il raggio finale ha origine nel punto appena calcolato e direzione verso il punto focale.

Il risultato è un blur sempre crescente per oggetti che si allontanano sempre di più (in entrambe le direzioni) dal punto focale.

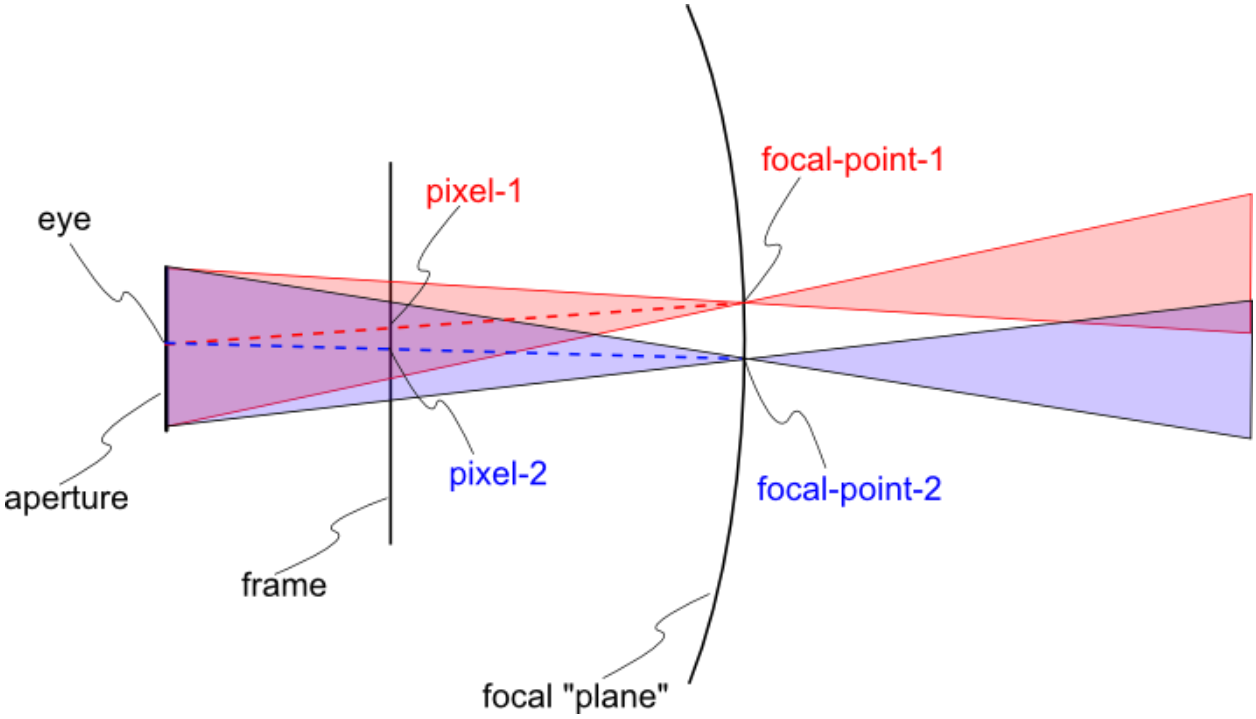

Figura 3 Illustrazione del funzionamento del depth of field. *Tratta da: https://stackoverflow.com/questions/13532947/references-for-depth-of-field-implementation-in-a-raytracer*

Figura 4 Depth of field molto esagerato in azione.

HDR, Tonemapping e Gamma Correction

L'environment-map viene utilizzata per dare illuminazione alla scena; pertanto, si utilizza un'immagine High Dynamic Range (HDR) per descrivere un'elevata ricchezza e dettaglio della scena. Queste immagini devono essere convertite in Low Dynamic Range in un processo denominato tonemapping, utilizzando un parametro di esposizione della telecamera. Inoltre, il render prodotto è inizialmente nello spazio lineare dei colori, ed è opportuno convertirlo nello spazio gamma, in modo tale da riflettere la linearità percepita dall'occhio umano che differisce da quella fisica.

In seguito al render, viene eseguito un passo di tonemapping e gamma correction per mostrare l'immagine finale. Questo è indipendente dal render in sé, perciò aggiustare l'esposizione non necessita di far ripartire il render da capo. È stato utilizzato un filmic tonemapper.

La trasformazione risultante è dunque la seguente:

```
vec3 filmic(vec3 c)
{
    return (0.9f*c*c + 0.02*c)/(0.87f*c*c + 0.35f * c + 0.14f);\n"
}
void main()
{
    vec3 color = vec3(texture(tex, texCoords));
    color = filmic(pow(2.0f, exposure)*color);
    color.x = pow(color.x, 1.0f/2.2f);
    color.y = pow(color.y, 1.0f/2.2f);
    color.z = pow(color.z, 1.0f/2.2f);
    fragColor = vec4(color, 1.0f);
}
```

Figura 5 Illustrazione di diversi livelli di esposizione

Esempi illustrativi

Si mostrano di seguito alcuni esempi atti a mostrare le capacità dello shader sviluppato. Questi esempi sono direttamente disponibili nell'eseguibile (occorre premere i tasti numerici per cambiare la scena attuale).

Figura 6 Scena con texture per legno, pelle, e metallo

Figura 7 Scena con materiali emissivo, riflettente e trasparente

Figura 8 Materiale glossy e coverage mask

Figura 9 Sfere riflettenti con livello di roughness crescente

Fonti e codice utilizzato

Durante lo sviluppo si è fatto ricorso ad alcune fonti, per verificare la correttezza o per ispirazione. Queste fonti si possono trovare all'interno del codice sotto forma di commenti, tuttavia si ritiene opportuno, per completezza, riportarle qui di seguito:

- LittleCG: Verifica della correttezza dell'implementazione di Fresnel Schlick, e per l'implementazione del sampling delle microfacce, in particolare per la trasformazione da spazio locale a globale.
- www.pcg-random.org e www.shadertoy.com/view/XlGcRh, per l'implementazione di Permuted Congruential Generator (PCG), un generatore di numeri casuali.
- https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution.html, per l'implementazione di intersezione raggio-triangolo.
- https://ceng2.ktu.edu.tr/~cakir/files/grafikler/Texture_Mapping.pdf, per ottenere le coordinate baricentriche a partire dalle coordinate cartesiane.
- https://stackoverflow.com/questions/13532947/references-for-depth-of-field-implementation-in-a-raytracer, per ispirazione per l'implementazione di depth of field.
- stb_image.h, una libreria per il caricamento di immagini.
- glfw, una windowing library.