UNIVERSITÀ
DEGLI STUDI
FIRENZE

High Dynamic Range Rendering

Scuola di Ingegneria
Dipartimento di
Ingegneria dell'Informazione

Esaminando : Billi Marco

Esaminatore : Prof. Berretti Stefano

Elaborato Computer Graphics
Anno Accademico 2024/25

Introduzione

Contesto di riferimento e definizioni

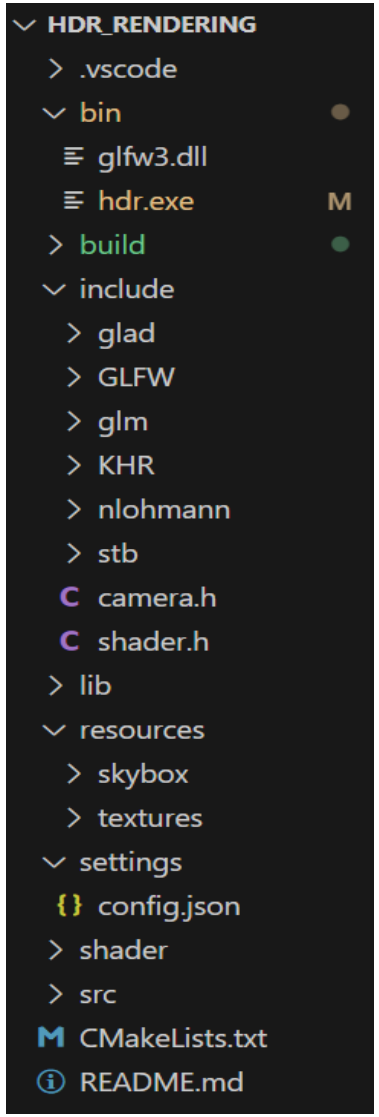
High Dynamic Range Rendering è un modo di fare rendering in CG di una scena con l'utilizzo di calcoli sulla luce utilizzando un intervallo di dinamica più ampio rispetto a quello standard. Il vantaggio di questi calcoli è dato da una maggior preservazione dei dettagli che normalmente sarebbero persi con la limitazione del rapporto di contrasto, oltre ad un miglioramento complessivo della percezione realistica della scena. L'ampiamento della dinamica standard è svolto tramite tecniche di Tone-Mapping che servono per la conversione dei colori da un intervallo di dinamica maggiore verso un intervallo di dinamica inferiore.

Obiettivo

L'obiettivo dell'elaborato è di implementare una scena utile per lo studio dell'HDRR con l'utilizzo di OpenGL e discutere le varie tecniche di Tone-Mapping implementate ed eventuali effetti.

Configurazioni e Strutture implementate

Oltre alla normale rappresentazione della scena avevo bisogno di un modo per gestire le varie configurazioni dell'applicativo e i vari parametri per l'utilizzo dell'HDRR senza andare a toccare il codice ma solo cambiando un semplice file. Per questo alcuni dati come la grandezza della finestra oppure la posizione della camera, per esempio, vengono presi a runtime da un file JSON che può essere modificabile dall'utente prima di far partire l'eseguibile. Inoltre sono state aggiunte delle classi per la gestione degli shader e della camera in modo semplificato e alcuni metodi per la gestione delle texture, delle callback e delle chiamate I/O. Per ulteriori informazioni su come impostare ed utilizzare l'applicativo basta andare a leggere il file README.md. Il codice è scaricabile dalla repository pubblica collegata al Qrcode oppure utilizzando il link https://github.com/billimarco/HDR_rendering.



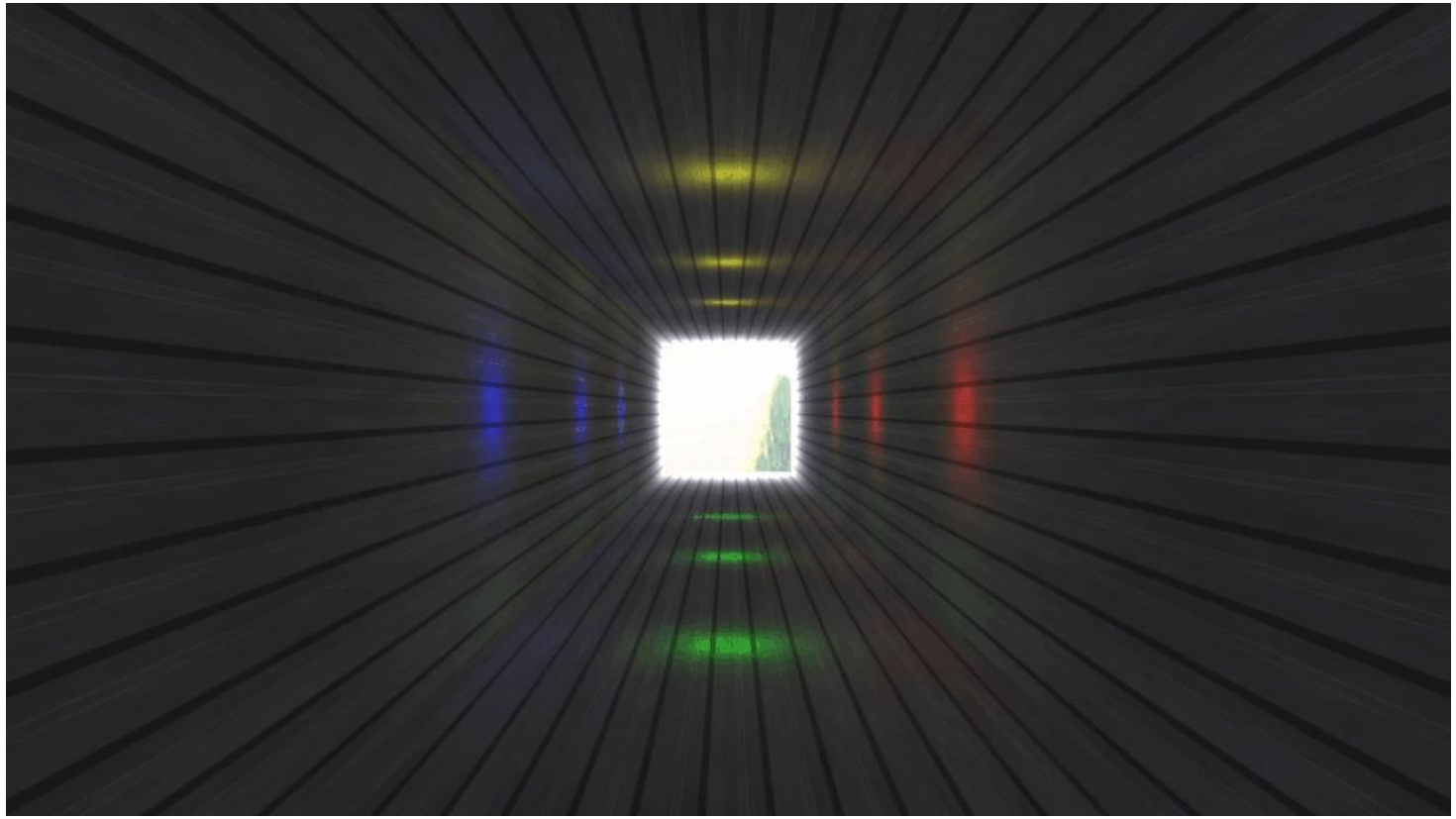
SCAN ME

Rappresentazione della scena

Per implementare una buona scena per lo studio dell'HDRR avevo bisogno principalmente di tre oggetti principali:

- Un tunnel che chiamerò Container per semplicità (ha la texture di un container) chiuso da una parte e aperto dall'altra. Avrà alcune luci interne che non sforano l'intervallo di dinamica standard[0,1].
- Una scena dell'orizzonte che chiamerò SkyBox con un po' di dettagli visivi rilevanti e colori diversificati.
- una luce particolarmente potente fuori dal container che sforerà l'intervallo di dinamica in maniera considerevole (questa luce rappresenta il Sole).

La scena è studiata in modo che ci sia un cambio di colore di frame costante più ci si muove verso l'uscita del container, quindi da uno spazio con poca illuminazione e colori più scuri verso una scena più illuminata e con colori più accesi rappresentati dalla SkyBox.



Luce

Prima di parlare di qualsivoglia applicazione di HDR dobbiamo definire come la luce interagisce con il nostro container. Qui accanto viene rappresentato il Fragment Shader per la renderizzazione del container. Ho aggiunto la possibilità di avere luce ambientale anche se la parte più importante è data dalla luce diffusa, oltre ad aver tenuto conto dell'attenuazione. Non ho aggiunto luce speculare perché non è utile per l'esercizio. Dobbiamo far si che le varie luci presenti interagiscano tra di loro (quindi le luci si sommano se si ha interazione). La parte relativa alla luminosità sottostante calcola un altro buffer di colore che sarà utile per l'utilizzo del bloom. I valori uscenti vengono salvati in un framebuffer che avrà salvato al suo interno due immagini, una per il colore e una per la luminosità, sotto forma di texture su cui verrà svolto del post-processing.

```
#version 330 core
layout (location = 0) out vec4 FragColor;
layout (location = 1) out vec4 BrightColor;

in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoords;

struct Light {
    vec3 Position;
    vec3 Color;
};

uniform Light lights[16];
uniform sampler2D difTex;
uniform vec3 viewPos;

void main()
{
    vec3 color = texture(difTex, TexCoords).rgb;
    vec3 normal = normalize(Normal);
    // ambient
    vec3 ambient = 0.1 * color;
    // lighting
    vec3 lighting = vec3(0.0);
    for(int i = 0; i < 16; i++)
    {
        // diffuse
        vec3 lightDir = normalize(lights[i].Position - FragPos);
        float diff = max(dot(lightDir, normal), 0.0);
        vec3 diffuse = lights[i].Color * diff * color;
        vec3 result = diffuse;
        // attenuation (use quadratic as we have gamma correction)
        float distance = length(FragPos - lights[i].Position);
        result *= 1.0 / (distance * distance);
        lighting += result;
    }
    FragColor = vec4(ambient + lighting, 1.0);

    // check whether fragment output is higher than threshold, if so output as brightness color
    float brightness = dot(FragColor.rgb, vec3(0.2126, 0.7152, 0.0722));
    if(brightness > 1.0)
        BrightColor = vec4(FragColor.rgb, 1.0);
    else
        BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
}
```

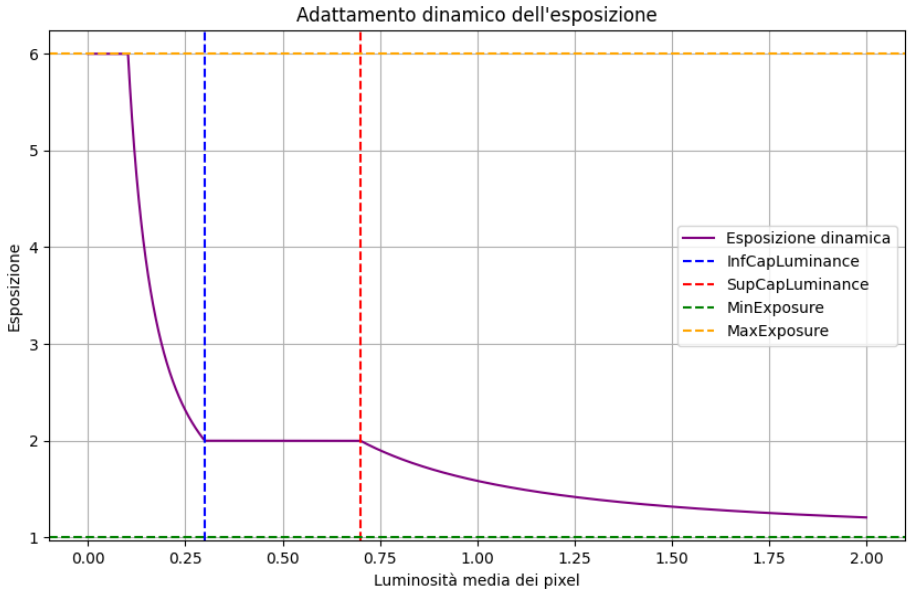
Bloom

Il bloom produce frange di luce che si estendono dai bordi delle aree luminose di un'immagine, contribuendo all'illusione di una luce estremamente brillante che sovrasta la fotocamera o l'occhio che cattura la scena. Per produrre questo effetto prendiamo l'immagine che rappresenta la luminosità prodotta precedentemente e gli applichiamo un filtro gaussiano a due passaggi (prima lo applico in orizzontale, poi in verticale e così via) per un numero limitato di volte e poi aggiungiamo il risultato finale alla immagine che rappresenta il colore.


```glsl
#version 330 core
#define PI 3.14159265358979323846264338327795

float gaussian(float offset);

out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D brightFrame;

uniform bool blurDirection;
uniform float stdDev;
uniform int kernelSize;

void main()
{
    vec2 tex_offset = 1.0 / textureSize(brightFrame, 0);
    float actual_offset = 0.0;
    vec3 result = texture(brightFrame, TexCoords).rgb * gaussian(actual_offset);
    if(blurDirection)
    {
        for(int i = 1; i < kernelSize; i++)
        {
           actual_offset += tex_offset.x;
           result += texture(brightFrame, TexCoords + vec2(actual_offset, 0.0)).rgb * gaussian(actual_offset);
           result += texture(brightFrame, TexCoords - vec2(actual_offset, 0.0)).rgb * gaussian(actual_offset);
        }
    }
    else
    {
        for(int i = 1; i < kernelSize; i++)
        {
           actual_offset += tex_offset.y;
           result += texture(brightFrame, TexCoords + vec2(0.0, actual_offset)).rgb * gaussian(actual_offset);
           result += texture(brightFrame, TexCoords - vec2(0.0, actual_offset)).rgb * gaussian(actual_offset);
        }
    }
    FragColor = vec4(result, 1.0);
}

float gaussian(float offset) {
    if(blurDirection)
        return exp(-pow(TexCoords.x+offset-TexCoords.x,2)/(2*pow(stdDev + 1e-6,2)))/sqrt(2*PI*stdDev + 1e-6);
    else
        return exp(-pow(TexCoords.y+offset-TexCoords.y,2)/(2*pow(stdDev + 1e-6,2)))/sqrt(2*PI*stdDev + 1e-6);
}
```

Esposizione dinamica

L'esposizione dinamica si riferisce a tecniche in cui l'esposizione (il controllo della quantità di luce che raggiunge il sensore o la pellicola) viene regolata in modo adattivo durante una scena. Quando ci si sposta da una scena scura a una luminosa (o viceversa), il sistema regola dinamicamente la "luminosità" per replicare l'adattamento naturale dell'occhio. Questo serve per rendere tecniche di tone-mapping globali più simili ad un approccio locale.




```
void updateExposure(Illumination* illum){
    // Target of ideal value of exposure
    float targetExposure =  (*illum).avgExposure;
    // If average pixel luminance is lower than an inferior cap (dark scene) then increase exposure
    // in a non-linear way
    if ((*illum).avgPixelScreenLuminance < (*illum).infCapLuminance) {
        targetExposure = pow((*illum).infCapLuminance / (*illum).avgPixelScreenLuminance, 1.5f);
    }
    // If average pixel luminance is greater than a superior cap (light scene) then decrease exposure
    // in a non-linear way
    else if ((*illum).avgPixelScreenLuminance > (*illum).supCapLuminance) {
        targetExposure = pow((*illum).supCapLuminance / (*illum).avgPixelScreenLuminance, 1.5f);
    }

    // This formula describes exposure change based on frame by frame difference with a learning rate
    float exposureChange = (targetExposure - (*illum).exposure) * (*illum).adaptationSpeed * deltaTimeFrame;
    // Limit exposure change into a range [-maxChange,maxChange] for limiting simil-instatanous frame
    // by frame average pixel luminance change
    exposureChange = std::clamp(exposureChange, -(*illum).maxChange, (*illum).maxChange);
    // Adding exposure change to image exposure
    (*illum).exposure += exposureChange;
    // Limit exposure into a range [minExposure,maxExposure] for infinite exposure in dark scene
    // behaviour and viceversa
    (*illum).exposure = std::clamp((*illum).exposure, (*illum).minExposure, (*illum).maxExposure);
}
```

Tone Mapping

Dopo aver calcolato il bloom ed averlo aggiunto all'immagine principale, dobbiamo ricalcolare l'intervallo di dinamica in modo da passare da un intervallo a dinamica elevata ad uno a dinamica bassa, tendenzialmente quella standard [0,1]. Questa operazione viene svolta da una funzione di Tone Mapping che preso un valore in ingresso anche elevato, restituisce un valore compreso nell'intervallo ammissibile standard. Ci sono due tipi di approcci principalmente:

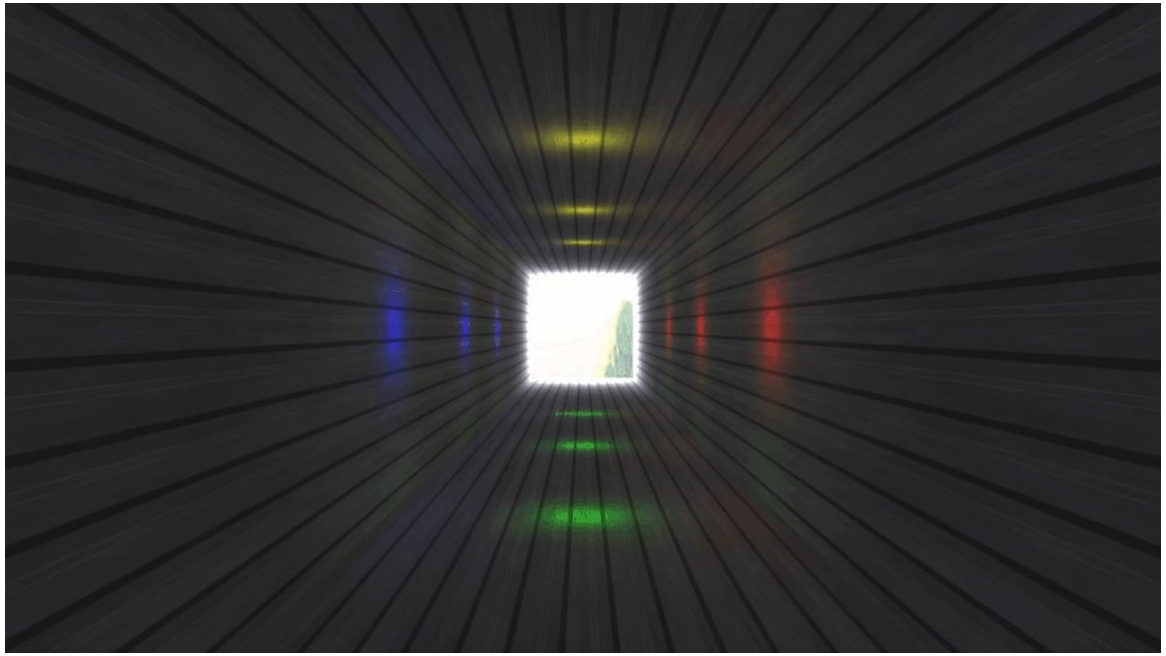
- Calcolo globale: questo tipo di calcolo avviene senza valutare la scena che stiamo guardando. In questi casi possiamo utilizzare l'esposizione dinamica se vogliamo un approccio più locale.
- Calcolo locale: questo tipo di calcolo avviene valutando la scena che stiamo guardando.

```glsl
#version 330 core

float log10(float x);
out vec4 FragColor;

in vec2 TexCoords;

uniform sampler2D hdrBuffer;
uniform sampler2D bloomBuffer;
uniform int hdr;
uniform bool bloom;
uniform float exposure;
uniform float maxPixelScreenLuminance;
uniform float avgPixelScreenLuminance;

void main()
{
    const float gamma = 2.2;
    vec3 hdrColor = texture(hdrBuffer, TexCoords).rgb;
    vec3 bloomColor = texture(bloomBuffer, TexCoords).rgb;
    if(bloom)
        hdrColor += bloomColor; // additive blending
    vec3 result;
    switch(hdr){
        case 0:
            result = pow(hdrColor, vec3(1.0 / gamma));
            break;
        case 1://Reinhard Tone-Mapping
            hdrColor *= exposure;
            result = hdrColor / (hdrColor + vec3(1.0));
            result = pow(result, vec3(1.0 / gamma));
            break;
        case 2://Exponential Tone-Mapping
            result = vec3(1.0) - exp(-hdrColor * exposure);
            result = pow(result, vec3(1.0 / gamma));
            break;
        case 3://Drago Tone-Mapping
            float bias = 0.85;
            float L_dmax = 100; //Max Luminance of the screen
            float L_wmax = maxPixelScreenLuminance*exposure/avgPixelScreenLuminance; //Max Luminance of the scene scaled by World Adaptation Luminance
            float L_w = dot(hdrColor*exposure, vec3(0.2126, 0.7152, 0.0722)); // Luminance of the pixel before Tone-Mapping (Y compenent of XYZ color map)
            float L_d = (L_dmax * 0.01) * log(1.0 + L_w) / (log10(L_wmax+1) * log(2.0 + 8.0 * pow(L_w/ L_wmax, log(bias)/log(0.5)))); // Luminance of the pixel after Tone-Mapping
            result = hdrColor * (L_d / (L_w + 1e-6));  // Avoid division by zero
            result = pow(result, vec3(1.0 / gamma));
            result = clamp(result, 0.0, 1.0);
            break;
    }
    FragColor = vec4(result, 1.0);
}

float log10(float x) {
    return log(x) / log(10.0);
}
```

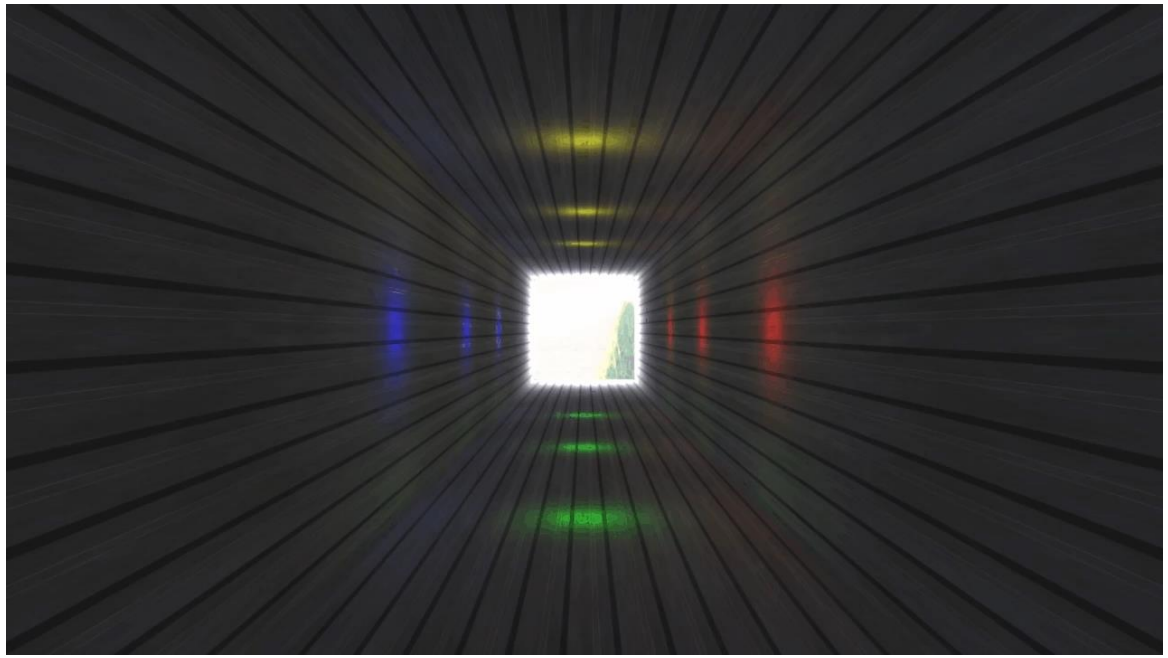
Senza HDR

Premendo il tasto **0** non si applica nessun filtro HDR, però si applica la gamma correction. In questa versione si possono notare alcuni effetti indesiderati come la perdita dei dettagli in zone altamente luminose e zone d'ombra troppo accentuate. Possiamo utilizzare l'esposizione dinamica per risolvere quest'ultime, ma otterremo comunque un risultato pessimo.

```
case 0:
    result = pow(hdrColor, vec3(1.0 / gamma));
    break;
```



Bloom

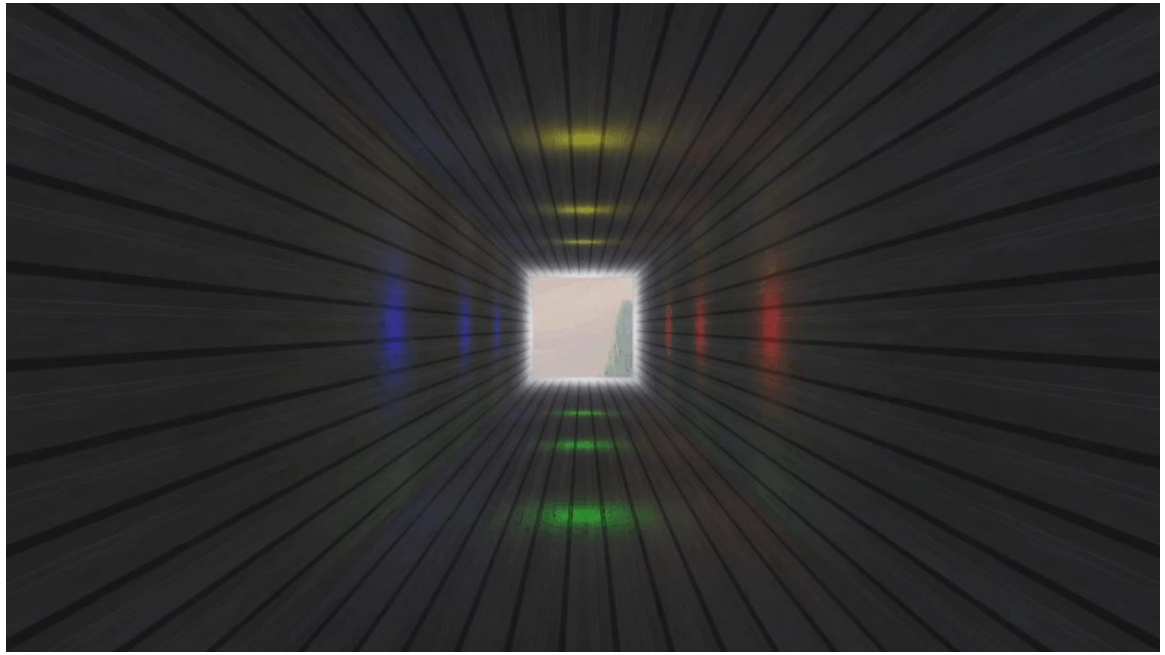


Bloom + Esposizione Dinamica

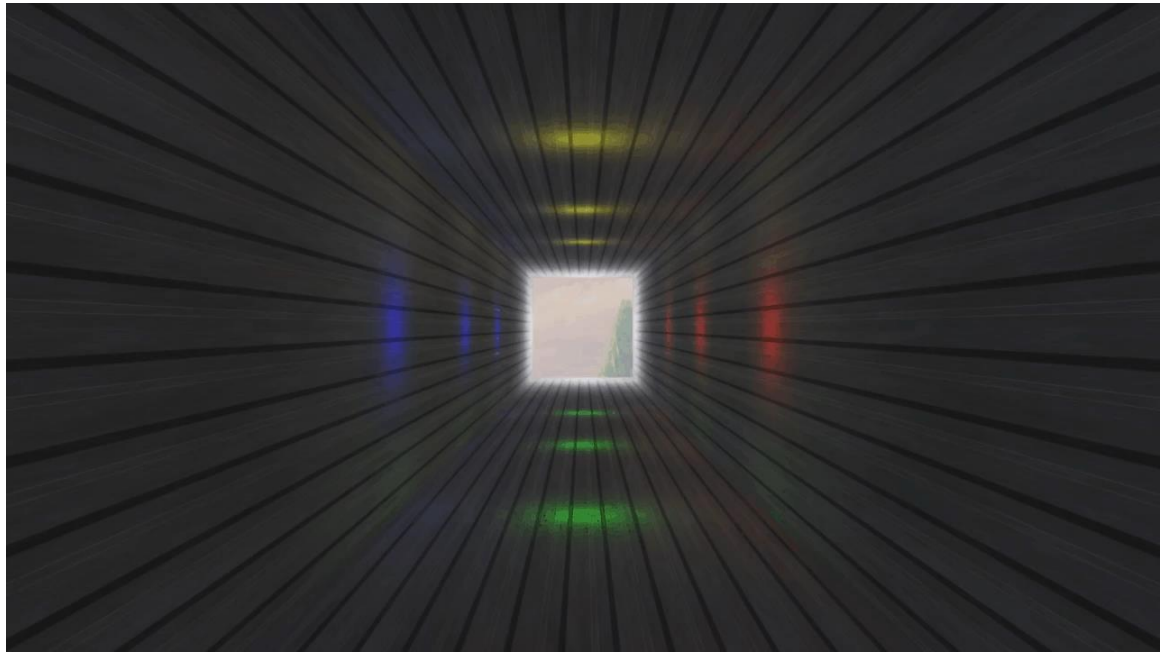
Reinhard HDR

Premendo il tasto **1** si applica il filtro HDR di Reinhard. É uno dei tone-mapping più semplici da implementare e garantisce un'immagine bilanciata. In zone più luminose o più scure si tende a perdere comunque dei dettagli (comunque sempre meglio della versione senza HDR). Possiamo implementare un approccio locale con l'utilizzo dell'esposizione dinamica.

```
case 1://Reinhard Tone-Mapping
    hdrColor *= exposure;
    result = hdrColor / (hdrColor + vec3(1.0));
    result = pow(result, vec3(1.0 / gamma));
    break;
```



Bloom

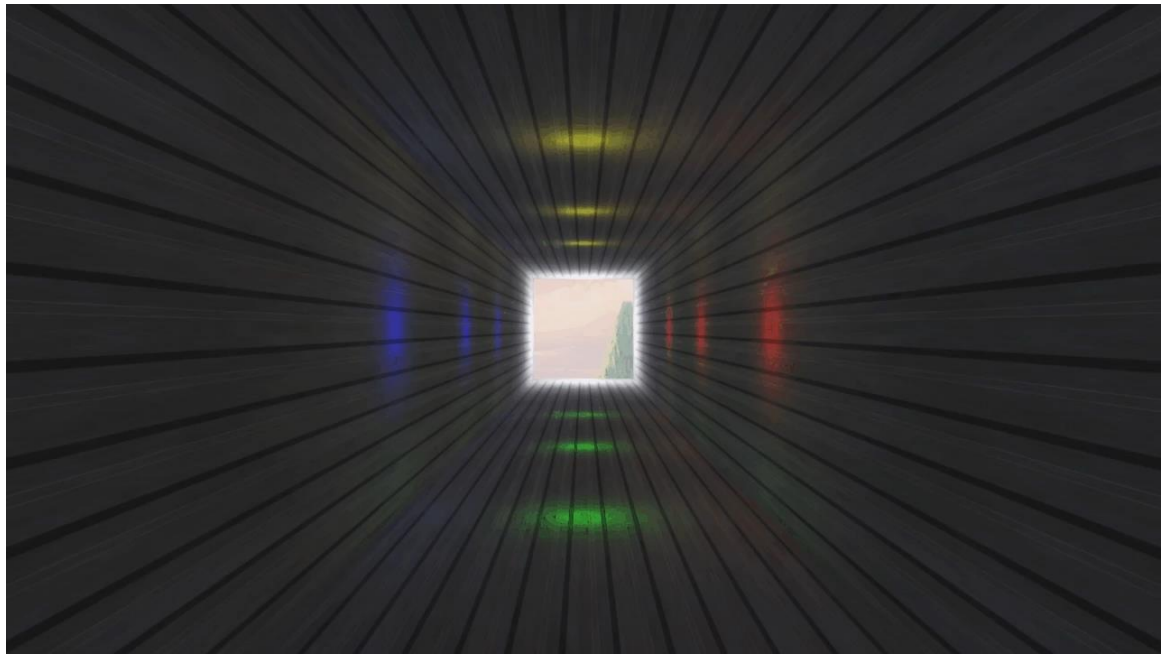


Bloom + Esposizione Dinamica

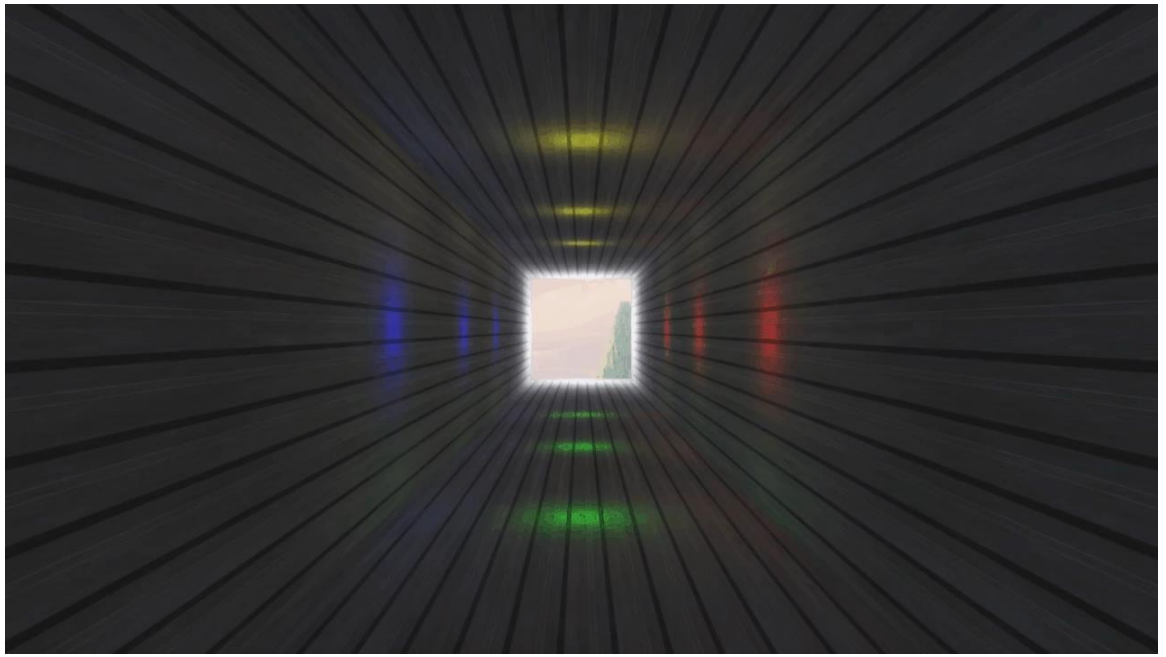
HDR esponenziale

Premendo il tasto **2** si applica il filtro HDR esponenziale. È semplice da implementare come quello di Reinhard, anche se l'immagine ottenuta è generalmente più accesa. Questa ultima peculiarità rende difficile l'utilizzo dell'esposizione dinamica per l'approccio locale, la quale va controllata più attentamente modificando i parametri. In compenso, si ottiene un effetto più realistico di uscita dal tunnel e conseguente esposizione ad una luce più intensa.

```
case 2://Exponential Tone-Mapping
    result = vec3(1.0) - exp(-hdrColor * exposure);
    result = pow(result, vec3(1.0 / gamma));
    break;
```



Bloom

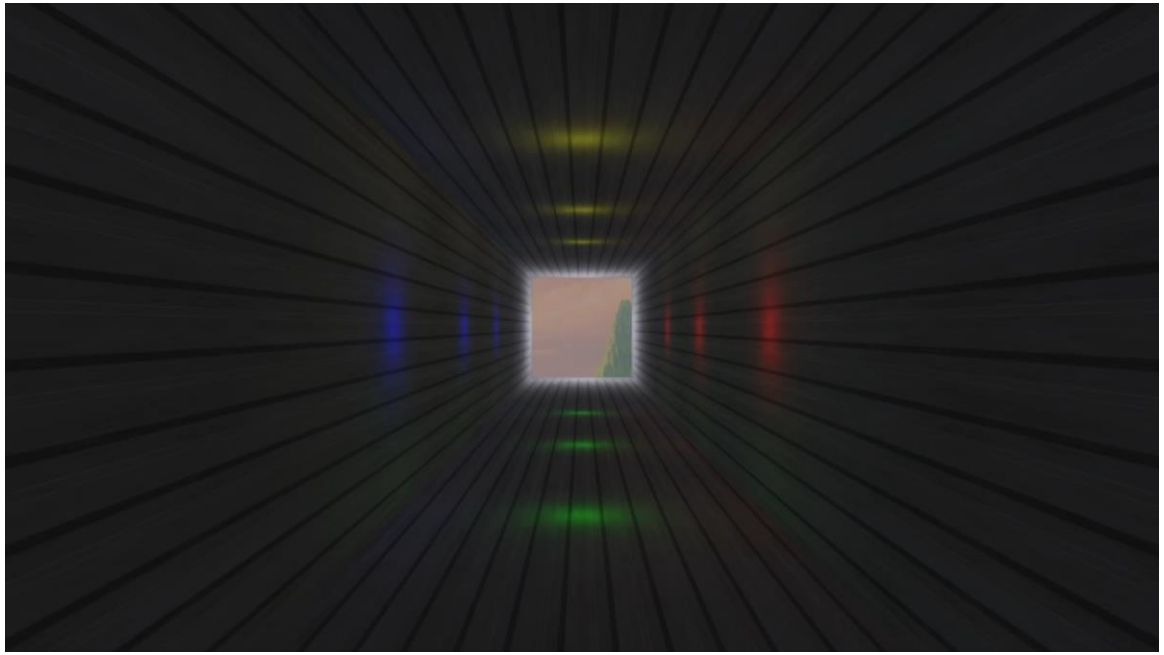


Bloom + Esposizione Dinamica

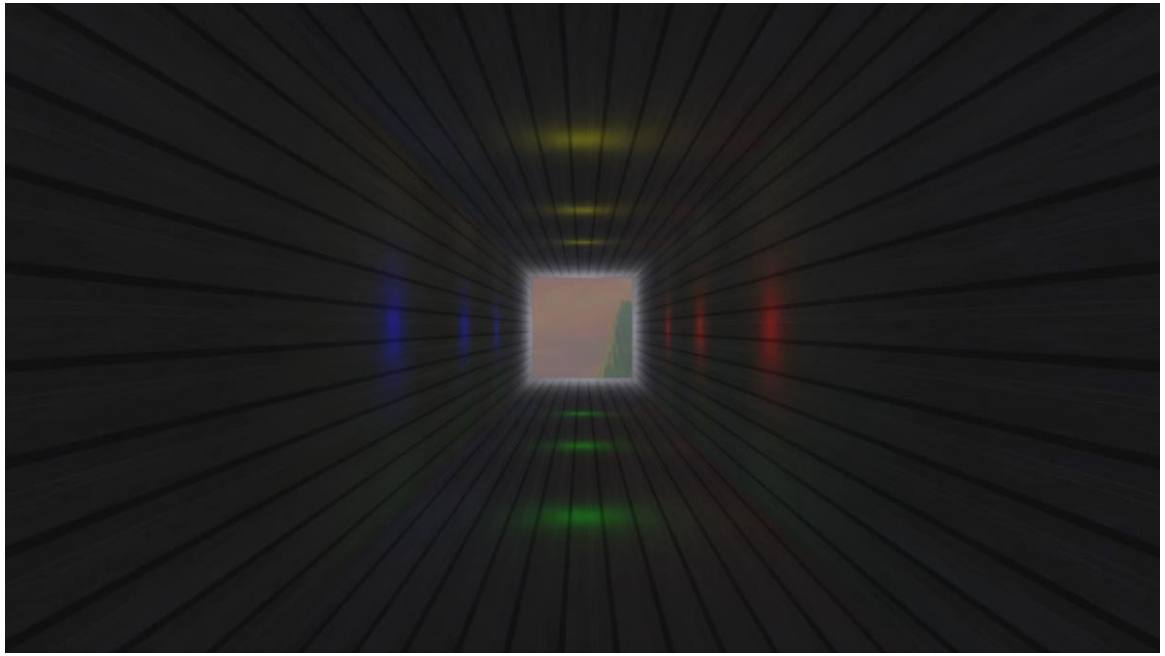
Drago HDR

Premendo il tasto **3** si applica il filtro HDR di Drago. Rispetto ai precedenti questo filtro è pensato già come un approccio locale, ma i parametri abbiamo dei parametri da definire. Questo tone-mapping si basa sul concetto di luminanza rispetto agli altri che lavorano solo sul colore. L'immagine ottenuta preserva meglio i dettagli ed è più realistica in immagini con elevato contrasto rispetto agli altri filtri, anche se il suo calcolo è generalmente più complesso.

```
case 3://Drago Tone-Mapping
    float bias = 0.85;
    float L_dmax = 100; //Max Luminance of the screen
    float L_wmax = maxPixelScreenLuminance*exposure/avgPixelScreenLuminance; //Max Luminance of the scene scaled by World Adaptation Luminance
    float L_w = dot(hdrColor*exposure, vec3(0.2126, 0.7152, 0.0722)); // Luminance of the pixel before Tone-Mapping (Y compenent of XYZ color map)
    float L_d = (L_dmax * 0.01) * log(1.0 + L_w) / (log10(L_wmax+1) * log(2.0 + 8.0 * pow(L_w/ L_wmax, log(bias)/log(0.5)))); // Luminance of the pixel after Tone-Mapping
    result = hdrColor * (L_d / (L_w + 1e-6));  // Avoid division by zero
    result = pow(result, vec3(1.0 / gamma));
    result = clamp(result, 0.0, 1.0);
    break;
```



Bloom



Bloom + Esposizione Dinamica

Conclusione

Risultati

Dopo un'attenta analisi possiamo notare che:

- A livello puramente visivo, il tone-mapping di Drago è quello che rende l'immagine più realista e preserva meglio i dettagli. Inoltre l'adattamento alla transizione tra zone scure verso quelle chiare è estremamente adatto a contesti videoludici, anche se non troppo realistico per la sua velocità.
- Un approccio con esposizione dinamica si adatta abbastanza bene al tentativo di simulare l'adattamento a contesti luminosi diversi della visione umana.
- L'utilizzo dell'HDR rende tendenzialmente migliore l'immagine ottenuta migliorando il contrasto.
- Non esiste un unico approccio giusto per ogni situazione possibile. Nei fatti un tone-mapping di Reinard o esponenziale può essere migliore di quello di Drago in alcuni contesti applicativi.

Cosa ho imparato

Lavorare a questo progetto mi ha permesso di comprendere meglio il funzionamento della pipeline OpenGL. Inoltre ho potuto approfondire alcuni concetti di modellazione della luce non coperta dal programma dell'esame.

Domande