# Raytracing simple acelerado sobre GPUs

Pablo Máñez Fernández[1]

Universidad de Alicante

*Email:* pablomanez@hotmail.com

*Github:* https://github.com/pablomanez

**Abstract.** En este documento encontrarás mi implementación personal del algoritmo que Peter Shirley explica en su libro Ray Tracing in One Weekend [2], además de la implementación del mismo algoritmo usando CUDA y un estudio comparativo.

**Keywords:** CUDA · raytracing · raytracer · C++ · glm.

## 1 ¿Qué es el raytracing?

Para aclarar dudas lo antes posible, raytracing [1], o trazado de rayos, es una técnica de renderizado para generar una imagen trazando el camino de la luz como píxeles sobre una imagen plana. Gracias a esta técnica, podemos producir un mayor grado de realismo respecto a los métodos tradicionales. Por consecuente, también necesita de un mayor coste computacional, debido al alto grado de cálculos que debe realizar.

Por ello, en este documento vas a encontrar la documentación necesaria para la implementación de un algoritmo de trazado de rayos por CPU, dado por Peter Shirley, en su libro Ray Tracing in One Weekend [2]. Además, se realizará un estudio del mismo, junto con una implementación en GPU utilizando la tecnología que Nvidia nos proporciona, ya que, como he apuntado antes, necesita de un mayor coste computacional.

## 2 Primeros pasos

Primeramente, voy a escribir un trozo de código muy pequeño, para que veamos alguna imagen, por así decirlo, en pantalla. En este caso, voy a utilizar un formato especial de imagen llamado PPM, que simplemente son números seguidos de saltos de línea y espacios, que marcan un cierto color del píxel, junto al ancho y al alto de la imagen.

```
P3
200 100
255
```

```
0 253 51
1 253 51
2 253 51
3 253 51
5 253 51
6 253 51
7 253 51
8 253 51
10 253 51
11 253 51
12 253 51
14 253 51
15 253 51
16 253 51
17 253 51
19 253 51
```

Por lo tanto, nos dejamos de tonterías y escribimos el siguiente código en C++:

```
float r, g, b;
int ir, ig, ib;
int nx = 200;   // Ancho
int ny = 100;   // Alto

// Inicio del archivo PPM
//      P3  -> El archivo esta en ASCII
//      255 -> Color 'maximo'
std::cout << "P3\n" << nx << "␣" << ny << "\n255\n";
for (int j = ny - 1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
                r = float(i) / float(nx);
                g = float(j) / float(ny);
                b = 0.2;
                ir = int(255.99*r);
                ig = int(255.99*g);
                ib = int(255.99*b);

                // Valores de los pixeles
                std::cout <<ir<< "␣" <<ig<< "␣" <<ib<< "\n";
        }
}

// La finale
getchar();
exit(-1);
```

Como primera observación, además de que si lo ejecutamos se tira un buen rato para obtener el resultado final, se puede ver que este resultado lo obtenemos

gracias a la salida de los cout, por lo tanto, lo primero que voy a hacer es crear un archivo que contenga dicha imagen. Cabe decir que NUNCA se deben dejar cout abiertos en ningún programa, ya que puede acarrear a problemas de caché y, por lo tanto, llegar a que falle.

Por lo tanto, además de incluir la librería fstream, que nos permitirá realizar la tarea, modificamos un poco el código anterior y lo dejamos tal que así:

```
float r, g, b;
int ir, ig, ib;
int nx = 200;   // Ancho
int ny = 100;   // Alto

std::ofstream _out("img.PPM");

// Inicio del archivo PPM
//      P3  -> El archivo esta en ASCII
//      255 -> Color 'maximo'
_out << "P3\n" << nx << "␣" << ny << "\n255\n";
for (int j = ny - 1; j >= 0; j--) {
        for (int i = 0; i < nx; i++) {
                r = float(i) / float(nx);
                g = float(j) / float(ny);
                b = 0.2;
                ir = int(255.99*r);
                ig = int(255.99*g);
                ib = int(255.99*b);

                // Valores de los pixeles
                _out << ir << "␣" << ig << "␣" << ib << "\n";
        }
}

// La finale
getchar();
exit(-1);
```

Ahora habrá un archivo llamado "img.PPM" que contendrá una ristra de 200x100 píxeles con sus respectivos valores en código RGB.

Finalmente, solo hay que ver la imagen que se ha construido, y, para ello, se hace uso de algún visor de PPM. En mi caso, uso un visor online [3] para facilitar la tarea de instalar nada en el ordenador.

**Fig. 1.** El hello world más colorido de todos

Como se ve, con unas cuantas líneas de código podemos ya realizar una imagen, como primer paso. Y esto solo es el Hello World, por así decirlo. De ahora en adelante es cuando empieza a llegar la chicha.

## 3 Reduciendo un poco el código

Por ahora tenemos los colores RGB mediante números flotantes y enteros, y lo que se va a hacer ahora es introducir los vectores para que nos ayuden a la hora de operar. Para ello, y como aparece en el libro de Peter Shirley, podemos implementar una clase llamada vec3, implementando todos los métodos necesarios para el trabajo con vectores de 3 dimensiones. Pero, como eso lleva un tiempo, aunque no mucho, podemos utilizar librerías externas para que ese tiempo de implementación sea igual a cero.

En mi caso, utilizaré la librería matemática GLM (OpenGL Mathematics Library) [4] : Una librería escrita en C++ y que posiblemente nos ayude mucho más adelante, gracias al soporte con CUDA que posee. Entonces solo tenemos que añadir la carpeta de glm al proyecto, que posee todas las cabeceras necesarias para que glm funcione y, por supuesto, que no se nos olvide incluir glm.hpp en nuestro main. Además, en mi caso ya tengo un proyecto de CUDA creado en Visual Studio, por lo tanto, las líneas necesarias para un proyecto de CUDA compile con glm son las siguientes:

```
#include <cuda.h>
#define GLM_FORCE_PURE
#define GLM_FORCE_CUDA
#include <glm/glm.hpp>
```

Una vez tenemos listo todo lo anterior, como bien dice el título de la sección, debemos tener nuestro main lo más limpio posible reduciendo el número de líneas del código. Así que vamos a crear una clase que se encargue de crear dicha imagen.

```
void ppmManagement::createImage(
    int _w,
    int _h,
```

```
    std::ofstream &_out)
{
        glm::vec3 color;
        int ir, ig, ib;

        // Inicio del archivo PPM
        //      P3  -> El archivo esta en ASCII
        //      255 -> Color 'maximo'
        _out << "P3\n" << _w << "␣" << _h << "\n255\n";
        for (int j = _h - 1; j >= 0; j--) {
                for (int i = 0; i < _w; i++) {
                        color = glm::vec3(
                                float(i) / float(_w),
                                float(j) / float(_h),
                                0.2
                        );

                        ir = int(255.99 * color.r);
                        ig = int(255.99 * color.g);
                        ib = int(255.99 * color.b);

                        // Valores de los pixeles
                        _out << ir << "␣";
                        _out << ig << "␣";
                        _out << ib << "\n";
                }
        }
}
```

Como se ve en el código anterior, simplemente se ha creado una clase que se ocupa de la creación de la imagen, un copia-pega del código que estaba antes en el main pero ahora está en una clase. Por lo tanto, desde el main, solamente se realiza la llamada a dicha función, pasando por parámetros el ancho, alto y una referencia a ofstream que sí que creamos en el propio main. También podemos simplificar dicha llamada si, en vez de pasar por parámetro la referencia a un ofstream, pasamos solo una cadena de texto que indique la ubicación y el nombre del archivo final.

```
int main(void) {
        mng.createImage(200,100,"img.PPM");

        std::cout << "Finalizado" << '\n';
}
```

## 4 Rayos!

No podemos hacer un raytracer sin escribir código que nos permita, de alguna forma, trazar rayos desde un sitio a otro. Por lo tanto, voy a realizar la implementación de una cámara, de la que saldrán una serie de rayos y dibujarán una

especie de "cielo" azul. Primeramente, hay que implementar una clase que nos permita, por así decirlo, tener control sobre una línea, vector o, dicho de otra forma, un rayo.

```
class ray{
        public:
                ray();
                ray(const glm::vec3&, const glm::vec3&);
                ~ray();

                void setRayParameters(
                const glm::vec3&,
                const glm::vec3&
        );

                glm::vec3 getOrigin();
                glm::vec3 getDirection();
                glm::vec3 getPointAtParameter(float);

        private:
                glm::vec3 origin;
                glm::vec3 dir;
};
```

```
#include "ray.h"

ray::ray() {};
ray::~ray(){}

ray::ray(const glm::vec3 &A, const glm::vec3 &B){
        origin = A;
        dir = B;
}


void ray::setRayParameters(
    const glm::vec3 &A,
    const glm::vec3 &B)
{
        origin = A;
        dir = B;
}

glm::vec3 ray::getOrigin() {
        return origin;
}

glm::vec3 ray::getDirection() {
        return dir;
}
```

```
glm::vec3 ray::getPointAtParameter(float _time) {
        return origin + (dir*_time);
}
```

Dicha clase, contendrá unas coordenadas con el origen del rayo y la dirección que éste lleva, además de varios métodos auxiliares en caso de necesitarlos. El método más relevante es getPointAtParameter(float): Dicho método devuelve la posición en la que se encontrará el origen del rayo, dado un determinado período de tiempo (t) en la dirección que tiene asignada, tanto si t es positivo (hacia adelante) como si es negativo (hacia atrás).

Ahora mismo, ya teniendo una base medianamente sólida con una clase que nos permite localizar en el espacio los rayos, se puede empezar a implementar una primera versión de un raytracer.

### 4.1 Mi primer raytracer

La función de un raytracer es lanzar rayos desde una cámara y saber que color se está viendo en la dirección que toman dichos rayos. Su núcleo se compone de una simple cámara, para saber desde donde hay que lanzar dichos rayos, y, por supuesto, el ancho y alto de la imagen que queremos sacar.

Por lo tanto, nuestro grandioso bucle se nos queda tal que así:

```
void ppmManagement::createImage(
    int _w,
    int _h,
    const std::string &_name)
    {
        std::ofstream _out(_name);
        int ir, ig, ib;
        ray r;
        float u,v;
        glm::vec3 color, dir;

        glm::vec3 lower_left_corner (-2.0, -1.0, -1.0);
        glm::vec3 horizontal        ( 4.0,  0.0,  0.0);
        glm::vec3 vertical          ( 0.0,  2.0,  0.0);
        glm::vec3 camOrigin         ( 0.0,  0.0,  0.0);

        // Inicio del archivo PPM
        //      P3  -> El archivo esta en ASCII
        //      255 -> Color 'maximo'
        _out << "P3\n" << _w << "␣" << _h << "\n255\n";
        for (int j = _h - 1; j >= 0; j--) {
                for (int i = 0; i < _w; i++) {
                        u = float(i) / float(_w);
                        v = float(j) / float(_h);

                        dir = lower_left_corner
```

```
                        + u * horizontal
                        + v * vertical;

                        r.setRayParameters(camOrigin, dir);
                        color = r.getColor();

                        ir = int(255.99 * color.r);
                        ig = int(255.99 * color.g);
                        ib = int(255.99 * color.b);

                        // Valores de los pixeles
                        _out << ir << "␣";
                        _out << ig << "␣";
                        _out << ib << "\n";
                }
        }
}
```

Si se observan las variables, se puede ver que nuestra cámara se encuentra en (0,0,0), mientras que nuestro "lienzo" está una unidad hacia adelante y tiene una dimensión de 4x2 unidades. Además también indicamos un punto de inicio, como es la esquina inferior izquierda en (-2,-1,-1) y, si afinamos la vista y se es un poco observador, se puede observar una función llamada getColor().

Pues bien, esta función consigue una especie de degradado en tono azulado y es tal que así:

```
glm::vec3 ray::getColor() {
        glm::vec3 unitDirection = glm::normalize(dir);
        float t = 0.5 * (unitDirection.y + 1.0f);
        return ((1.f - t)*glm::vec3(1.f))
              + (t*glm::vec3(0.5, 0.7, 1.0));
}
```

Y, finalmente si ejecutamos nuestro main debe salir algo parecido a esta imagen:

**Fig. 2.** El primer trazado de rayos

## 5 Dando alegría a la escena

Ahora mismo la imagen que se ha producido podría decirse que es una especie de cielo, pero está muy vacía, por lo tanto ahora se va a añadir la primera esfera a la escena. ¿Por qué una esfera? Porque, a parte de que saber si un rayo en concreto está colisionando con una esfera es sencillo, como más adelante se añadirán reflexiones, pues que menos que poner esferas para verlos en su máxima totalidad.

No voy a entrar en detalles sobre como se realizan estos cálculos, aunque sí citar al autor del libro principal que estoy siguiendo, Peter Shirley, que lo explica de maravilla:

> Recall that the equation for a sphere centered at the origin of radius R is x*x + y*y + z*z = R*R.
> The way you can read that equation is "for any (x, y, z), if x*x + y*y + z*z = R*R then (x,y,z) is on the sphere and otherwise it is not". It gets uglier if the sphere center is at (cx, cy, cz):
>
> $$(x - cx) * (x - cx) + (y - cy) * (y - cy) + (z - cz) * (z - cz) = R * R \quad (1)$$
>
> In graphics, you almost always want your formulas to be in terms of vectors so all the x/y/z stuff is under the hood in the vec3 class. You might note that the vector from center C = (cx,cy,cz) to point p = (x,y,z) is (p - C) . And dot((p - C) ,(p - C) ) = (x-cx)*(x-cx) + (y-cy)*(y-cy) + (z-cz)*(z-cz).
> So the equation of the sphere in vector form is:
>
> $$dot((p - c), (p - c)) = R * R \quad (2)$$
>
> We can read this as "any point p that satisfies this equation is on the sphere". We want to know if our ray p(t) = A + t*B ever hits the sphere anywhere. If it does hit the sphere, there is some t for which p(t) satisfies the sphere equation. So we are looking for any t where this is true:
>
> $$dot((p(t) - c), (p(t) - c)) = R * R \quad (3)$$
>
> or expanding the full form of the ray p(t):
>
> $$dot((A + t * B - C), (A + t * B - C)) = R * R \quad (4)$$
>
> The rules of vector algebra are all that we would want here, and if we expand that equation and move all the terms to the left hand side we get:
>
> $$t*t*dot(B, B)+2*t*dot(B, A-C)+dot(A-C, A-C)-R*R = 0 \quad (5)$$

```
#define SPHERE_CENTER glm::vec3(0,0,-1)
#define SPHERE_RADIUS 0.5

bool ray::hitSphere(
    const glm::vec3 &center,
    float radius)
    {
        glm::vec3 oc = origin - center;
        float a = glm::dot(dir, dir);
        float b = 2.0*glm::dot(oc,dir);
        float c = glm::dot(oc,oc) - radius*radius;
        float disc = b * b - 4 * a*c;
        return (disc > 0);
}

glm::vec3 ray::getColor() {
        if (hitSphere(SPHERE_CENTER, SPHERE_RADIUS)) {
                return glm::vec3(1, 0, 0);
        }
        glm::vec3 unitDirection = glm::normalize(dir);
        float t = 0.5 * (unitDirection.y + 1.0f);
        return ((1.f - t) * glm::vec3(1.f))
            + (t * glm::vec3(0.5, 0.7, 1.0));
}
```

Si actualizamos las funciones y ejecutamos debe salir algo parecido a esta imagen:

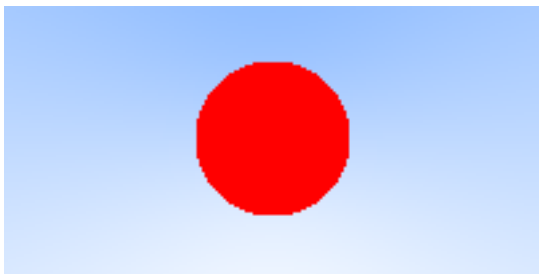

**Fig. 3.** Esfera roja sobre un cielo

## 6 Añadiendo normales y varios objetos en la escena

Una normal simplemente es un vector que siempre está en dirección perpendicular a la superficie. En el caso de las esferas, nos encontramos que el vector siempre tendrá una distancia "offset" que sale desde el centro de ésta.

Si realizamos algunos cambios en nuestro código, podemos conseguir algo así

**Fig. 4.** Esfera con normales

Aunque aquí hemos venido a jugar y vamos a escribir un código que nos permita poner varias esferas en la escena, es decir: Como la imagen anterior, pero todo parametrizado y ordenado en distintas clases.

Primeramente creamos una clase abstracta, de la que heredarán la mayoría de clases que queremos que tengan una colisión visible a nuestra cámara. Para ello se va a implementar un método virtual con el que, a parte de conseguir los datos de la colisión mediante una estructura, sabremos si colisiona con alguno de los objetos de la escena, o no.

```
struct hit_record {
    float t;
    glm::vec3 p;
    glm::vec3 normal;
};

class hitable {
    public:
    virtual bool hit(
        const ray &r,
        float tmin,
        float t_max,
        hit_record& rec) const = 0;
};
```

Y ahora la clase esfera en cuestión, que, obviamente, heredará de hitable.

```
#include "hitable.h"

class sphere : public hitable {
    private:
        glm::vec3 center;
        float radius;

    public:
        sphere();
        ~sphere();
        sphere(glm::vec3, float);
```

```
        virtual bool hit(
            const ray&,
            float,
            float,
            hit_record&) const;
};
```

```
#include "sphere.h"

sphere::sphere() {}
sphere::~sphere() {}

sphere::sphere(glm::vec3 cen, float r)
: center(cen), radius(r){}

bool sphere::hit(
    const ray &r,
    float t_min,
    float t_max,
    hit_record &rec) const
    {

    glm::vec3 const origin = r.getOrigin();
    glm::vec3 const dir = r.getDirection();

    glm::vec3 oc = origin - center;
    float a = glm::dot(dir, dir);
    float b = glm::dot(oc, dir);
    float c = glm::dot(oc, oc) - radius * radius;
    float disc = b*b - a*c;

    if (disc > 0) {
        float temp = (-b - glm::sqrt(b*b - a*c)) / a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.getPointAtParameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            return true;
        }

        temp = (-b + glm::sqrt(b*b - a*c)) / a;
        if (temp < t_max && temp > t_min) {
            rec.t = temp;
            rec.p = r.getPointAtParameter(rec.t);
            rec.normal = (rec.p - center) / radius;
            return true;
        }
    }
    return false;
}
```

Como se ve, este el método sphere::hit sobreescribe el antiguo ray::hitSphere. Finalmente creamos una clase con la que poder añadir los objetos que queramos. Para ello necesitamos almacenar un array de objetos colisionables y su tamaño, para más comodidad.

```
#include "hitable.h"

class hitable_list : public hitable {
    private:
        hitable **list;
        int list_size;

    public:
        hitable_list();
        ~hitable_list();
        hitable_list(hitable**, int);
        virtual bool hit(
            const ray&,
            float,
            float,
            hit_record&) const;
};
```

```
#include "hitable_list.h"

hitable_list::hitable_list() {}
hitable_list::~hitable_list() {}

hitable_list::hitable_list(hitable **l, int n) {
    list = l;
    list_size = n;
}

bool hitable_list::hit(
    const ray &r,
    float t_min,
    float t_max,
    hit_record &rec) const
    {

    hit_record temp_rec;
    bool hit_anything = false;
    double csf = t_max;

    for (int i = 0; i < list_size; i++) {
        if (list[i]->hit(r, t_min, csf, temp_rec)) {
            hit_anything = true;
            closest_so_far = temp_rec.t;
            rec = temp_rec;
        }
```

```
    }
    return hit_anything;
}
```

Dicha clase comprueba la colisión entre todos los objetos que pertenezcan al array de objetos colisionables.

Por último queda modificar nuestra clase principal para incluir todos los cambios que se han realizado hasta ahora.

```
glm::vec3 ppmManagement::getColor(
    const ray& r,
    hitable *WORLD)
    {

    hit_record rec;
    if (WORLD->hit(r, 0.0, FLT_MAX, rec)) {
        return  0.5f
                * glm::vec3(
                    rec.normal.x + 1,
                    rec.normal.y + 1,
                    rec.normal.z + 1);
    }
    else {
        glm::vec3 unitDirection
        unitDirection = glm::normalize(r.getDirection());
        float t = 0.5 * (unitDirection.y + 1.0f);
        return  ((1.f - t)
                * glm::vec3(1.f))
                + (t * glm::vec3(0.5, 0.7, 1.0));
    }
}

void ppmManagement::createImage(
    int _w,
    int _h,
    const std::string &_name)
    {

    std::ofstream _out(_name);
    int ir, ig, ib;
    float u,v;
    ray r;
    glm::vec3 color, dir;

    glm::vec3 lower_left_corner (-2.0, -1.0, -1.0);
    glm::vec3 horizontal                ( 4.0,  0.0,  0.0);
    glm::vec3 vertical                  ( 0.0,  2.0,  0.0);
    glm::vec3 camOrigin                 ( 0.0,  0.0,  0.0);

    int const total_hitables = 2;
```

```
    hitable *list[total_hitables];
    list[0] = new sphere(glm::vec3(0, 0, -1), 0.5);
    list[1] = new sphere(glm::vec3(0, -100.5, -1), 100);

    hitable *WORLD = new hitable_list(list, total_hitables);

    // Inicio del archivo PPM
    //  P3  -> El archivo esta en ASCII
    //  255 -> Color 'maximo'
    _out << "P3\n" << _w << "␣" << _h << "\n255\n";
    for (int j = _h - 1; j >= 0; j--) {
        for (int i = 0; i < _w; i++) {
            u = float(i) / float(_w);
            v = float(j) / float(_h);

            dir = lower_left_corner
                + u*horizontal
                + v*vertical;
            r.setRayParameters(camOrigin, dir);

            color = getColor(r, WORLD);

            ir = int(255.99 * color.r);
            ig = int(255.99 * color.g);
            ib = int(255.99 * color.b);

            // Valores de los pixeles
            _out << ir << "␣";
            _out << ig << "␣";
            _out << ib << "\n";
        }
    }
}
```

Si compilamos y damos al play, la salida debe ser algo así:

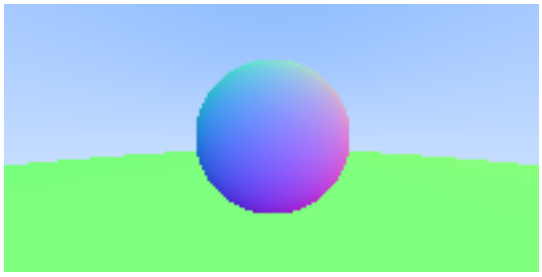

**Fig. 5.** Esfera con normales

## 7 Mejorando la calidad de la imagen incluyendo aliasing

Si vemos la escena que hemos generando anteriormente, podemos observar que los bordes de ambas esferas son algo "perfectos". Esto es generado así ya que estamos comprobando pixel a pixel de la propia imagen, pero en la vida real, esto no es así y siempre hay algo de "difuminado" en los bordes. Esto se llama aliasing.

Para aplicarlo al código, primero vamos a crear una clase camera y así organizamos más nuestro código.

```
#include "ray.h"
#include "Util.h"

class camera {
    private:
        glm::vec3 lower_left_corner;
        glm::vec3 horizontal;
        glm::vec3 vertical;
        glm::vec3 camOrigin;

    public:
        camera();
        ~camera();

        ray getRay(float, float);
};
```

```
#include "camera.h"

camera::camera() {
    lower_left_corner = glm::vec3(-2.0, -1.0, -1.0);
    horizontal = glm::vec3(4.0, 0.0, 0.0);
    vertical = glm::vec3(0.0, 2.0, 0.0);
    camOrigin = glm::vec3(0.0, 0.0, 0.0);
}
camera::~camera() {}

ray camera::getRay(float u, float v) {
    glm::vec3 a_ret =   lower_left_corner +
                        u*horizontal +
                        v*vertical -
                        camOrigin;
    return ray(camOrigin, a_ret);
}
```

Esta clase simplemente es la abstracción de como creábamos la cámara directamente en nuestra clase principal. Así está de una forma mucho más ordenada y limpia. Por último se modifica el bucle principal:

```
void ppmManagement::createImage(
```

```
    int _w,
    int _h,
    int ns,
    const std::string &_name)
    {

    // ns es la precision del aliasing
    // ns = 100 (default)
    std::ofstream _out(_name);
    int ir, ig, ib, ns;
    float u,v;
    ray r;
    glm::vec3 color;

    hitable *list[2];
    list[0] = new sphere(glm::vec3(0, 0, -1), 0.5);
    list[1] = new sphere(glm::vec3(0, -100.5, -1), 100);
    hitable *WORLD = new hitable_list(list, 2);

    camera cam;

    std::default_random_engine gen;
    std::uniform_real_distribution<double> dist(0.0,1.0);
    // Inicio del archivo PPM
    //  P3  -> El archivo esta en ASCII
    //  255 -> Color 'maximo'
    _out << "P3\n" << _w << "␣" << _h << "\n255\n";
    for (int j = _h - 1; j >= 0; j--) {
        for (int i = 0; i < _w; i++) {
            color = glm::vec3(0, 0, 0);
            for (int s = 0; s < ns; s++) {
                u = float(i + dist(gen)) / float(_w);
                v = float(j + dist(gen)) / float(_h);
                r = cam.getRay(u, v);
                color += getColor(r, WORLD);
            }

            color /= float(ns);
            ir = int(255.99 * color.r);
            ig = int(255.99 * color.g);
            ib = int(255.99 * color.b);

            // Valores de los pixeles
            _out << ir << "␣";
            _out << ig << "␣";
            _out << ib << "\n";
        }
    }
}
```

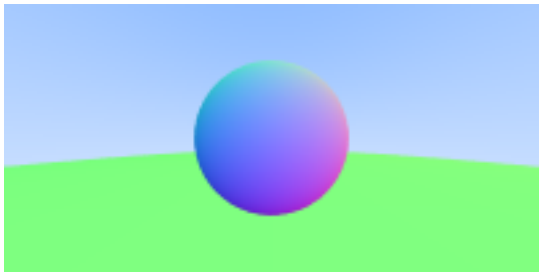

**Fig. 6.** Esfera con aliasing

Comparando las dos imágenes, podemos observar el difuminado de los bordes, haciendo que sea mucho más natural una imagen que la otra, si se compara con la vida real. Además que, si no se realiza este aliasing, los bordes quedan de una forma muy brusca.

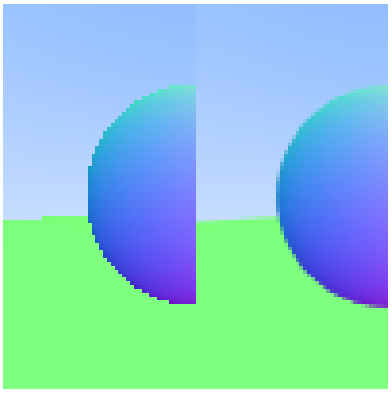

**Fig. 7.** Comparación entre imágenes

## 8 Materiales difusos

Para conseguir una escena mucho más realista y fiel a la realidad, debemos dejar atrás cualquier color sólido que estemos usando para dar color a ésta. Por lo tanto, hay que empezar por los materiales difusos, en este caso. Un material difuso es tipo mate, donde su reflexión no es muy alta y absorben gran parte de la luz.

```
std::default_random_engine gen;
std::uniform_real_distribution<double> d_dist(0.0, 1.0);

glm::vec3 ppmManagement::randomInUnitSphere() {
```

```
    glm::vec3 p;
    do {
        p = 2.0f*glm::vec3(
                        d_dist(gen),
                        d_dist(gen),
                        d_dist(gen)) - glm::vec3(1,1,1);
    } while ( (glm::length(p)*glm::length(p)) >= 1.0);
    return p;
}

glm::vec3 ppmManagement::getColor(
    const ray& r,
    hitable *WORLD)
    {

    hit_record rec;
    if (WORLD->hit(r, 0.0, FLT_MAX, rec)) {
        glm::vec3 target =  rec.p +
                            rec.normal +
                            randomInUnitSphere();
        ray ray_ret(rec.p, target - rec.p);
        return 0.5f*getColor(ray_ret,WORLD);
    }
    else {
        glm::vec3 unitDirection;
        unitDirection = glm::normalize(r.getDirection());
        float t = 0.5 * (unitDirection.y + 1.0f);
        return  ((1.0f - t) *
                glm::vec3(1.f)) +
                (t * glm::vec3(0.5, 0.7, 1.0));
    }
}
```

Mediante esta modificación del código, se logra que los rayos que salen de nuestra cámara tengan unas direcciones "aleatorias" para generar una especie de textura y obtener una escena más realista que solo simples colores.

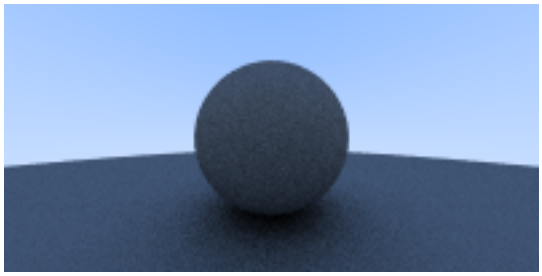

**Fig. 8.** Primera prueba de materiales difusos

Si realizamos unas pequeñas correcciones en el código, logramos que se vea mucho mejor la escena y también quitar un poco el ruido que se genera en ésta.

Primero, modificar esta línea e ignorar los puntos que son casi cero.

```
if (WORLD->hit(r, 0.001, FLT_MAX, rec)) {
```

Después, ya que la escena es un poco oscura de más, se puede aumentar el "gamma" de ésta para que podamos verla más "alegre" o iluminada.

**Listing 1.1.** ppmManagement.cu

```
...
color /= float(ns);
color = glm::vec3(
    glm::sqrt(color.r),
    glm::sqrt(color.g),
    glm::sqrt(color.b));
ir = int(255.99 * color.r);
ig = int(255.99 * color.g);
ib = int(255.99 * color.b);
...
```

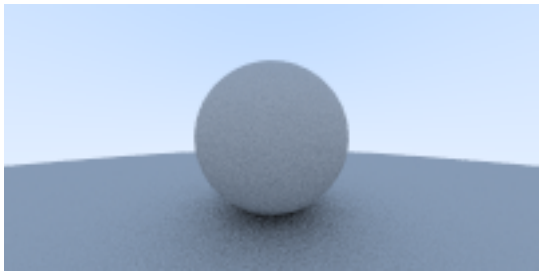

**Fig. 9.** Escena con materiales difusos 2

## 9 Metal!

Si queremos tener distintos tipos de materiales, a parte de uno tipo mate, como el que se va a implementar ahora, de metal, lo que se debe hacer es, primero, la creación de una clase abstracta que encapsule todas aquellas que serán materiales y crear una herencia muy básica.

**Listing 1.2.** material.h

```
#include "hitable.h"
#include "ray.h"

class material {
    public:
```

```
        virtual bool scatter(
            const ray &r_in,
            const hit_record &rec,
            glm::vec3 &atten,
            ray &scattered) const = 0;
};
```

Además, puesto que ahora vamos a tener distintos resultados dependiendo del material que cada objeto de la escena que se utilice, hay que guardar ese material de alguna forma. Por lo tanto, en la estructura que guarda los datos de colisión de cada rayo tenemos que guardar un puntero a éste, además de guardarlo en la propia esfera.

**Listing 1.3.** hitable.h

```
...
class material;

struct hit_record {
    float t;
    glm::vec3 p;
    glm::vec3 normal;
    material *mat_ptr;
};
...
```

**Listing 1.4.** sphere.h

```
...
class sphere : public hitable {
    private:
        glm::vec3 center;
        float radius;
        material *mat_ptr;

    public:
...
```

**Listing 1.5.** sphere.cu

```
...
if (disc > 0) {
    float temp = (-b - glm::sqrt(b*b - a*c)) / a;
    if (temp < t_max && temp > t_min) {
        rec.t = temp;
        rec.p = r.getPointAtParameter(rec.t);
        rec.normal = (rec.p - center) / radius;
        rec.mat_ptr = mat_ptr;
        return true;
    }

    temp = (-b + glm::sqrt(b*b - a*c)) / a;
```

```
    if (temp < t_max && temp > t_min) {
        rec.t = temp;
        rec.p = r.getPointAtParameter(rec.t);
        rec.normal = (rec.p - center) / radius;
        rec.mat_ptr = mat_ptr;
        return true;
    }
}
...
```

A continuación, implementamos ambos materiales, difuso y metal, respectivamente:

**Listing 1.6.** lambertian.h

```
#include "ray.h"
#include "material.h"

class lambertian : public material {
    private:
        glm::vec3 albedo;
    public:
        lambertian(const glm::vec3&);
        virtual bool scatter(
            const ray&,
            const hit_record&,
            glm::vec3&,
            ray&) const;
};
```

**Listing 1.7.** lambertian.cu

```
#include "lambertian.h"

lambertian::lambertian(const glm::vec3 &a) {
    albedo = a;
}

bool lambertian::scatter(
    const ray &r_in,
    const hit_record &rec,
    glm::vec3 &attenuation,
    ray &scattered) const
    {

    glm::vec3 target =  rec.p +
                        rec.normal +
                        randomInUnitSphere();
    scattered = ray(rec.p, target - rec.p);
    attenuation = albedo;
    return true;
}
```

**Listing 1.8.** metal.h

```
#include "ray.h"
#include "material.h"

class metal : public material {
    private:
        glm::vec3 albedo;
    public:
        metal(const glm::vec3&);
        virtual bool scatter(
            const ray&,
            const hit_record&,
            glm::vec3&,
            ray&) const;
};
```

**Listing 1.9.** metal.cu

```
#include "metal.h"

metal::metal(const glm::vec3 &a) {
    albedo = a;
}

bool metal::scatter(
    const ray &r_in,
    const hit_record &rec,
    glm::vec3 &attenuation,
    ray &scattered) const
    {

    glm::vec3 reflected;
    reflected = glm::reflect(
                            glm::normalize(
                                r_in.getDirection()
                            ),
                            rec.normal);
    scattered = ray(rec.p, reflected);
    attenuation = albedo;
    return true;
}
```

Y finalmente que tenemos implementados ambos materiales, solo queda modificar la función que determina el color que aparece finalmente en la escena y, como no, nuestro bucle principal.

**Listing 1.10.** ppmManagement.cu

```
glm::vec3 ppmManagement::getColor(
    const ray& r,
    hitable *WORLD,
```

```
    int depth)
    {

    hit_record rec;
    if (WORLD->hit(r, 0.001, FLT_MAX, rec)) {
        ray scattered;
        glm::vec3 attenuation;
        if (depth < 50 &&
            rec.mat_ptr->scatter(
                                r,
                                rec,
                                attenuation,
                                scattered
                                ))
        {
            return  attenuation *
                    getColor(scattered, WORLD, depth + 1);
        }
        else {
            return glm::vec3(0, 0, 0);
        }
    }
    else {
        glm::vec3 unitDirection;
        unitDirection = glm::normalize(r.getDirection());
        float t = 0.5 * (unitDirection.y + 1.0f);
        return ((1.0f - t) * glm::vec3(1.f)) +
                (t * glm::vec3(0.5, 0.7, 1.0));
    }
}

void ppmManagement::createImage(
    int _w,
    int _h,
    int ns,
    const std::string &_name)
    {

    // ns es la precision del aliasing
    // ns = 100 (default)
    std::ofstream _out(_name);
    int ir, ig, ib;
    float u,v;
    ray r;
    glm::vec3 color;

    int const total_hitables = 4;
    hitable *list[total_hitables];
    list[0] =   new sphere(glm::vec3(0, 0, -1),
                0.5,
```

```
                new lambertian(glm::vec3(0.8, 0.3, 0.3)));
    list[1] =   new sphere(glm::vec3(0, -100.5, -1),
                100,
                new lambertian(glm::vec3(0.8, 0.8, 0.0)));
    list[2] =   new sphere(glm::vec3(1,0,-1),
                0.5,
                new metal(glm::vec3(0.8, 0.6, 0.2)));
    list[3] =   new sphere(glm::vec3(-1,0,-1),
                0.5,
                new metal(glm::vec3(0.8, 0.8, 0.8)));

    hitable *WORLD;
    WORLD = new hitable_list(list, total_hitables);

    camera cam;

    // Inicio del archivo PPM
    //  P3  -> El archivo esta en ASCII
    //  255 -> Color 'maximo'
    _out << "P3\n" << _w << "␣" << _h << "\n255\n";
    for (int j = _h - 1; j >= 0; j--) {
        for (int i = 0; i < _w; i++) {
            color = glm::vec3(0, 0, 0);
            for (int s = 0; s < ns; s++) {
                u = float(i + UTIL_rand_d()) / float(_w);
                v = float(j + UTIL_rand_d()) / float(_h);
                r = cam.getRay(u, v);
                color += getColor(r, WORLD,0);
            }

            color /= float(ns);
            color = glm::vec3(
                glm::sqrt(color.r),
                glm::sqrt(color.g),
                glm::sqrt(color.b)
            );
            ir = int(255.99 * color.r);
            ig = int(255.99 * color.g);
            ib = int(255.99 * color.b);

            // Valores de los pixeles
            _out << ir << "␣";
            _out << ig << "␣";
            _out << ib << "\n";
        }
    }
}
```

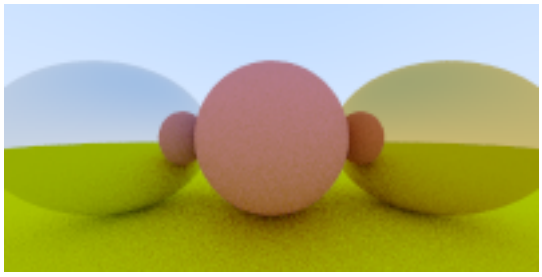

**Fig. 10.** Varios materiales en la escena

También se puede añadir algo de borrosidad a los objetos de la escena, simplemente añadiendo un parámetro más, en este caso, al material de metal.

```
metal::metal(const glm::vec3 &a, float f = 0) {
    albedo = a;

    (f < 1) ? fuzz = f : fuzz = 1.0;
}

...
    scattered = ray(
                    rec.p,
                    reflected + fuzz*randomInUnitSphere()
                );
...
```

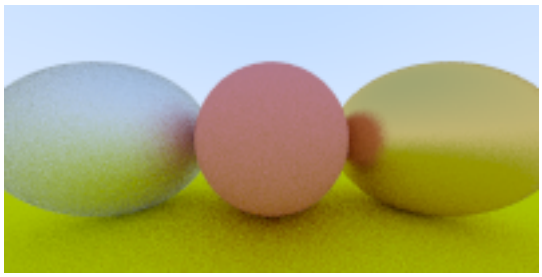

**Fig. 11.** Escena con metal borroso

### 9.1 Un poco de orden

Puesto que ahora tenemos unas cuantas clases y algo de código que empieza a remezclarse, el orden es muy importante. Lo que yo utilizo para que no haya que hacer include de mil cosas en cada archivo es crear un .h en el que añado

todos los include comunes de todos los archivos, como puede ser iostream, vector, etc. Además, hay algunas funciones que en el código son utilizadas por distintas clases no relacionadas en distintos trozos de código. Por ello también las incluyo en este archivo para evitar problemas y duplicidad de código.

Además, debo puntualizar el cambio de generador de números aleatorios a la librería random, algo más estandarizado y, además, más sencillo.

**Listing 1.11.** Util.h

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <iostream>
#include <fstream>
#include <string>
#include <random>

#include <cuda.h>
#define GLM_FORCE_PURE
#define GLM_FORCE_CUDA
#include <glm/glm.hpp>

inline double UTIL_rand_d(
    double min = 0.0,
    double max = 1.0)
    {

    return  ((double)rand() * (max - min)) /
            (double)RAND_MAX + min;
}

inline glm::vec3 randomInUnitSphere() {
    glm::vec3 p;
    do {
        p = 2.0f*glm::vec3(
                            UTIL_rand_d(),
                            UTIL_rand_d(),
                            UTIL_rand_d()
                        ) - glm::vec3(1, 1, 1);
    } while ((glm::length(p)*glm::length(p)) >= 1.0);
    return p;
}
```

## 10 Comenzando la optimización en CUDA

Aunque el libro de Peter Shirley sigue con algunos pasos más para crear un raytracer un poco más avanzado, el proyecto se encuentra en un momento donde ya puede verse la implementación de una mejora bastante alta de rendimiento y cabe decir que todas las imágenes que se han generado son de dimensión 200x100

píxeles. A continuación, se va a realizar un estudio del rendimiento en la CPU de unos cuantos perfiles elegidos al azar y su consiguiente mejora y optimización con el uso de CUDA.

**Table 1.** Tabla con los tiempos de ejecución sobre la CPU del raytracer implementado

| Ancho | Alto | Precisión aliasing | Tiempo |
|---|---|---|---|
| 200 | 100 | 100 | 00:00:18 |
| 200 | 100 | 1000 | 00:13:13 |
| 1280 | 720 | 100 | 00:02:47 |
| 1280 | 720 | 1000 | mayor que 01:30:13 |

Como se ve, con esta simple prueba, observamos que cuantas más iteraciones carguemos en el bucle principal, mucho mayor será el tiempo que la CPU va a estar trabajando. Por lo tanto, se observa una necesidad clarísima de optimización del código.

Para ello, y como ya he adelantado varias veces anteriormente, voy a usar CUDA.

### 10.1 Conociendo mi tarjeta gráfica

En mi caso tengo una Nvida GTX 660 OC, que pertenece a la arquitectura Kepler. Eso significa que tiene una compute capability 3.0 [5].

Según la tabla de información sobre las especificaciones técnicas de cada compute capability, mi gráfica sería capaz de usar hasta 16 bloques y un máximo de 2048 hilos por cada multiprocesador. Ahora, sabiendo que límites tiene mi tarjeta gráfica, voy a empezar con la implementación.

### 10.2 Cambio de la estructura de clases

Puesto que este es mi primer trabajo "grande", por así decirlo, de CUDA, primero debo documentarme un poco sobre como trabajar con distintas clases, con distintos métodos y, lo que me he dado cuenta, es que he estado usando una estructura errónea de clases durante todo el proyecto.

Como ya tengo un conocimiento previo de C++, di por hecho que todas la clases tenían la estructura de:

1. Declaración de los métodos y variables en un fichero .h o similar.
2. Implementación de los métodos de cada clase en un .cu (.cpp en el caso de C++).

Entonces una clase tenía esta pinta:

**Listing 1.12.** metal.h

```
#include "Util.h"
```

```
#include "ray.h"
#include "material.h"

class metal : public material {
    private:
        glm::vec3 albedo;
        float fuzz;
    public:
        __device__ metal(const glm::vec3&, float);
        __device__ virtual bool scatter(
                            const ray&,
                            const hit_record&,
                            glm::vec3&, ray&) const;
};
```

**Listing 1.13.** metal.cu

```
#include "metal.h"

__device__ metal::metal(const glm::vec3 &a, float f = 0) {
    albedo = a;
    (f < 1) ? fuzz = f : fuzz = 1.0;
}

__device__ bool metal::scatter(
    const ray &r_in,
    const hit_record &rec,
    glm::vec3 &attenuation,
    ray &scattered) const
    {

    glm::vec3 reflected;
    reflected = glm::reflect(glm::normalize
                                        (
                                        r_in.getDirection()),
                                        rec.normal
                                        );
    scattered = ray(
                    rec.p,
                    reflected + fuzz * randomInUnitSphere()
                   );
    attenuation = albedo;
    return true;
}
```

Después de la gran transformación de cada clase, por supuesto añadiendo las correspondientes etiquetas device, se eliminaron todos los .cu, excepto el main principal kernel.cu, quedando tal que así:

**Listing 1.14.** metal.h

```
#include "Util.h"
```

```
#include "ray.h"
#include "material.h"

class metal : public material {
    private:
        glm::vec3 albedo;
        float fuzz;
    public:
        __device__ metal(const glm::vec3&, float);
        __device__ virtual bool scatter(
                            const ray&,
                            const hit_record&,
                            glm::vec3&, ray&) const;
};

__device__ metal::metal(const glm::vec3 &a, float f = 0) {
    albedo = a;
    (f < 1) ? fuzz = f : fuzz = 1.0;
}

__device__ bool metal::scatter(
    const ray &r_in,
    const hit_record &rec,
    glm::vec3 &attenuation,
    ray &scattered) const
    {

    glm::vec3 reflected;
    reflected = glm::reflect(glm::normalize
                                        (
                                        r_in.getDirection()),
                                        rec.normal
                                        );
    scattered = ray(
                    rec.p,
                    reflected + fuzz * randomInUnitSphere()
                   );
    attenuation = albedo;
    return true;
}
```

Cabe destacar que ahora el bucle principal ya no se encuentra en ppmManagement.cu, sino en kernel.cu, por razones obvias que he explicado antes. Por lo tanto solo se hace un copia-pega de lo que había en ppmManagement a kernel.cu y todo debería funcionar tal y como funcionaba antes.

Una vez cambiada toda la estructura hay que pensar como hacer las funciones. Si se hace una traza de lo que la implementación en CPU realiza, se ve que hay un bucle principal que, recorriendo cada pixel de la imagen, saca el color

correspondiente que después se saca al archivo.PPM. Esto puede dar una idea de como se puede llegar a la implementación en CUDA.

**Listing 1.15.** Bucle principal en CPU

```
for:
    for:
        // Determina color
        // Salida a archivo
    end for;
end for;
```

### 10.3 Reservar memoria de las variables que se van a usar

Lo primero de todo es reservar memoria en la GPU de las variables que se van a usar en el cálculo:

**Listing 1.16.** Inicialización de las variables principales para el kernel

```
...
//Establezco la GPU que voy a usar
cudaSetDevice(0);

// DATA
// ns es la precision del aliasing
const std::string filename = "img.PPM";
int const TOTAL_HITABLES = 4;
int _w, _h, ns;
cudaError_t err_;

std::ofstream _out(filename);
_w = 200;
_h = 100;
ns = 100;
int imgSize = _w * _h;

// Definir variables que voy a usar en la GPU
// Array de salida de cada color
glm::vec3 *d_arr = NULL;
glm::vec3 *h_arr = new glm::vec3[_w*_h];

for (int i = 0; i < imgSize; i++) {
    h_arr[i] = glm::vec3(0,0,0);
}

hitable **list = NULL;
hitable **WORLD = NULL;

// Reservo memoria de la GPU
int kMemBytes = sizeof(glm::vec3)*_w*_h;
```

```
int kMemList = sizeof(hitable*)*TOTAL_HITABLES;

cudaMalloc((void**)&d_arr,  kMemBytes);
cudaMalloc((void**)&list,   kMemList);
cudaMalloc((void**)&WORLD,  sizeof(hitable*));

// Solo se lanza 1 nucleo ya que solo se va a hacer 1 vez
initVariables <<< 1,1 >>> (list, TOTAL_HITABLES,WORLD);
...
```

La salida va a ser escrita en un array de glm::vec3 y los objetos a usar para la salida, hitables, se crearán dentro de la GPU. Además, por simple precaución, inicializo el array, que se situa en el host y que después guardará la salida que se ha obtenido en la GPU, a glm::vec3(0,0,0).

Dicha función que inicia las variables realiza lo mismo que hacía en la CPU, pero ahora en la GPU y, MUY IMPORTANTE, solo 1 vez, puesto que sino provocaría fallos de acceso a la memoria que tenemos reservada. Entonces se lanza solo 1 hilo y 1 bloque y, para más precaución, también se comprueba dentro de la función.

**Listing 1.17.** Función initVariables()

```
__global__ void initVariables(
    hitable **list,
    int list_length,
    hitable **WORLD)
    {

    if (threadIdx.x == 0 && blockIdx.x == 0) {
        list[0] = new sphere(
                    glm::vec3( 0,0,-1),
                    0.5,
                    new lambertian(glm::vec3(0.8, 0.3, 0.3))
                    );
        list[1] = new sphere(
                    glm::vec3( 0,-100.5,-1),
                    100,
                    new lambertian(glm::vec3(0.8, 0.8,0.0)));
        list[2] = new sphere(
                    glm::vec3( 1,0,-1),
                    0.5,
                    new metal(glm::vec3(0.8, 0.6, 0.2),0.3));
        list[3] = new sphere(
                    glm::vec3(-1,0,-1),
                    0.5,
                    new metal(glm::vec3(0.8, 0.8, 0.8),1.0));
        *WORLD = new hitable_list(list, list_length);
    }
}
```

Una vez que se han inicializado estas variables, solo queda copiar el contenido del array que hemos creado antes al que se va a usar en la GPU y lanzar la función principal. Dependiendo del tamaño de bloque que se elija va a haber un tamaño de malla totalmente distinto, por eso utilizo ese pequeño trozo de código que utilizo en el código, además utilizando una notación que me parece muy elegante y limpia:

$$condition?IFTrueDO() : IFFalseDO(); \tag{6}$$

```
...
// Defino el tamanyo de bloque y malla
int K_T_W       = 32;
int K_T_H       = 32;
int K_B_W       =   ((float)_w / K_T_W) > (_w / K_T_W) ?
                    (_w / K_T_W) + 1
                    :
                    (_w / K_T_W);
int K_B_H       =   ((float)_h / K_T_H) > (_h / K_T_H) ?
                    (_h / K_T_H) + 1
                    :
                    (_h / K_T_H);

std::cout << "BLOQUE:␣" << K_T_W << "␣x␣" << K_T_H << "\n";
std::cout << "MALLA:␣" << K_B_W << "␣x␣" << K_B_H << "\n";

// Hilos por bloque             (THREADS PER BLOCK)
dim3 tpb(K_T_W, K_T_H, 1);
// Bloques por malla    (BLOCKS PER GRID)
dim3 bpg(K_B_W, K_B_H, 1);

// LINK STARTO!
cudaMemcpy(d_arr, h_arr, kMemBytes, cudaMemcpyHostToDevice);

createImage <<< bpg, tpb >>> (d_arr, WORLD, _w, _h, ns);
...
```

### 10.4 Función principal

```
...
__global__ void createImage(
    glm::vec3 *d_arr,
    hitable **WORLD,
    int _w,
    int _h,
    int ns)
    {
```

```
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

    if (i >= _w || j >= _h) return;

    glm::vec3 color(0,0,0);
    camera cam;
    ray r;

    for (int s = 0; s < ns; s++) {
        float u = float(i + UTIL_rand_d()) / float(_w);
        float v = float(j + UTIL_rand_d()) / float(_h);
        r = cam.getRay(u, v);
        color += getColor(r, WORLD, 50);
    }

    color /= float(ns);

    color.r = int(255.99 * color.r);
    color.g = int(255.99 * color.g);
    color.b = int(255.99 * color.b);

    // Llevar los resultados al array
    int pos = j * _w + i;
    pos = abs(_w*_h - pos);
    d_arr[pos] = color;
}
...
```

Esta función lanza por cada hilo lanzado, calcula su posición, como si fuera un píxel de la imagen y calcula el color de dicho píxel individualmente. Es lo mismo que se realizaba en la implementación de la CPU, pero sin un bucle principal que recorre todos los píxeles uno a uno, la magia de CUDA.

Finalmente, en la función se ve que el color es llevado al array que hemos creado y reservado en el main para después llevar una copia al host. Además, como se ve, los píxeles son calculados al revés, por lo tanto, si no queremos que la imagen salga invertida, solo hay que hacer el simple cálculo de la posición que se muestra en las últimas líneas.

### 10.5 De la GPU a la CPU

Una vez que el procesamiento ha acabado en la GPU, hay que llevar los datos a la CPU. Para ello usamos el array auxiliar que nos hemos creado al principio de todo y volcamos los datos desde la GPU usando la función cudaMemCpy().

Finalmente volcamos los datos de la CPU al archivo img.PPM y, muy importante, liberar toda la memoria que se ha reservado.

```
...
cudaMemcpy(h_arr, d_arr, kMemBytes, cudaMemcpyDeviceToHost);
```

```
// Inicio del archivo PPM
_out << "P3\n" << _w << "␣" << _h << "\n255\n";

// ESTO DEBE IR EN UNA FUNCION GLOBAL
for (int i = 0; i < imgSize; i++) {
    _out << h_arr[i].r << "␣";
    _out << h_arr[i].g << "␣";
    _out << h_arr[i].b << "\n";
}

// Libero la GPU
delete h_arr;

cudaFree(d_arr);
cudaFree(list);
cudaFree(WORLD);

cudaDeviceReset();
```

## 11 Problemas

El código está acabado, pero, ¿funciona?, la respuesta, en mi caso es simple: NO. El código ejecutado entero no, por problemas que debido a mi escaso conocimiento en CUDA no llego a comprender. También he intentado compilar el código ya implementado sobre el algoritmo de Peter y no funciona tampoco. Llego a creer que el problema es algo relacionado con mi tarjeta gráfica. Por lo tanto, no puedo realizar un estudio de rendimiento de la generación de la imagen, para mi desgracia.

## 12 Conclusión general del trabajo

Podría decir que ha sido uno de los trabajos que me ha costado más, en tema de horas trabajadas, pero también, de los que más me ha gustado realizar. La potencia que CUDA posee y la forma de trabajar con ésta, es algo que nunca había experimentado y en algo tan simple como la generación del algoritmo sal y pimienta de las prácticas me deja sombrado. Una pena no poder haberlo visto en este trabajo después de muchas horas de trabajo.

En cuanto al tiempo dedicado, dividido en las dos partes principales, CPU y GPU, ha sido casi 50-50.

1. En la CPU habían muchas cosas que no entendía y me costó un tiempo aprender como funcionaban, además de que el trabajo era, en líneas de código, inmensamente mayor.
2. En la GPU el mayor reto ha sido entender de verdad como trabaja la tarjeta gráfica. Para ello escribí una simple función que utilizaba los vectores de glm

para tener una implementación propia y, de paso, probar que las funciones de glm funcionaban correctamente.
También me harté de leer documentación oficial e información de foros, aunque no hay que confiar del todo en estos últimos, y agradezco haber hecho eso.

Aunque sé que me he dejado muchas cosas en el tintero, como por ejemplo la implementación del generador de números random propio de CUDA, me llevo una impresión muy buena, aún con el gran fallo que ha habido, y con ganas de más, en un futuro y con una tarjeta gráfica distinta, por si acaso, además que ya es hora de cambiarla.

## References


1. https://en.wikipedia.org/wiki/Ray_tracing_(graphics)
2. https://github.com/petershirley/raytracinginoneweekend
3. http://paulcuth.me.uk/netpbm-viewer/
4. https://glm.g-truc.net/0.9.2/api/index.html
5. https://en.wikipedia.org/wiki/CUDA#Version_features_and_specifications
6. https://devblogs.nvidia.com/accelerated-ray-tracing-cuda/