CG – prática

Esteban Walter Gonzalez Clua
Instituto de Computação – UFF
esteban@ic.uff.br

Implementar um ray tracing...

Implementar um ray tracing...

https://raytracing.github.io/books/RayTracingInOneWeekend.html

Tarefa 1

Criar um arquivo de imagem

Criar uma imagem de xadrez (preto e branco)

(seção 2.2)

Tarefa 2 – Implementar Vec/point

Tarefa 2 – Implementar Vec/ point

```
#ifndef VEC3_H
#define VEC3_H

#include <cmath>
#include <iostream>

class vec3 {
  public:
    double e[3];

    vec3() : e{0,0,0} {}
    vec3(double e0, double e1, double e2) : e{e0, e1, e2} {}

    double x() const { return e[0]; }
    double y() const { return e[1]; }
    double z() const { return e[2]; }

    vec3 operator-() const { return vec3(-e[0], -e[1], -e[2]); }
    double operator[](int i) const { return e[i]; }
    double& operator[](int i) { return e[i]; }

    vec3& operator+=(const vec3& v) {
        e[0] += v.e[0];
        e[1] += v.e[1];
        e[2] += v.e[2];
        return *this;
    }
```

Tarefa 2 – Implementar Vec/ point

```
    vec3& operator*=(double t) {
        e[0] *= t;
        e[1] *= t;
        e[2] *= t;
        return *this;
    }

    vec3& operator/=(double t) {
        return *this *= 1/t;
    }

    double length() const {
        return std::sqrt(length_squared());
    }

    double length_squared() const {
        return e[0]*e[0] + e[1]*e[1] + e[2]*e[2];
    }
};
```

Tarefa 2 – Implementar Vec/ point

```
// point3 is just an alias for vec3, but useful for geometric clarity in the code.
using point3 = vec3;
```

Tarefa 2 – Implementar Vec/ point

```
// Vector Utility Functions

inline std::ostream& operator<<(std::ostream& out, const vec3& v) {
    return out << v.e[0] << ' ' << v.e[1] << ' ' << v.e[2];
}

inline vec3 operator+(const vec3& u, const vec3& v) {
    return vec3(u.e[0] + v.e[0], u.e[1] + v.e[1], u.e[2] + v.e[2]);
}

inline vec3 operator-(const vec3& u, const vec3& v) {
    return vec3(u.e[0] - v.e[0], u.e[1] - v.e[1], u.e[2] - v.e[2]);
}

inline vec3 operator*(const vec3& u, const vec3& v) {
    return vec3(u.e[0] * v.e[0], u.e[1] * v.e[1], u.e[2] * v.e[2]);
}

inline vec3 operator*(double t, const vec3& v) {
    return vec3(t*v.e[0], t*v.e[1], t*v.e[2]);
}

inline vec3 operator*(const vec3& v, double t) {
    return t * v;
}

inline vec3 operator/(const vec3& v, double t) {
    return (1/t) * v;
}
```

Tarefa 2 – Implementar Vec/ point

```
inline double dot(const vec3& u, const vec3& v) {
    return u.e[0] * v.e[0]
         + u.e[1] * v.e[1]
         + u.e[2] * v.e[2];
}

inline vec3 cross(const vec3& u, const vec3& v) {
    return vec3(u.e[1] * v.e[2] - u.e[2] * v.e[1],
                u.e[2] * v.e[0] - u.e[0] * v.e[2],
                u.e[0] * v.e[1] - u.e[1] * v.e[0]);
}

inline vec3 unit_vector(const vec3& v) {
    return v / v.length();
}

#endif
```

Tarefa 3 – Implementar Color

Tarefa 4 – imagem com Vec e color

```cpp
#include "color.h"
#include "vec3.h"

#include <iostream>

int main() {

    // Image

    int image_width = 256;
    int image_height = 256;

    // Render

    std::cout << "P3\n" << image_width << ' ' << image_height << "\n255\n";

    for (int j = 0; j < image_height; j++) {
        std::clog << "\rScanlines remaining: " << (image_height - j) << ' ' << std::flush;
        for (int i = 0; i < image_width; i++) {
            auto pixel_color = color(double(i)/(image_width-1), double(j)/(image_height-1), 0);
            write_color(std::cout, pixel_color);
        }
    }

    std::clog << "\rDone.                 \n";
}
```

Tarefa 5 – Ray

```
#ifndef RAY_H
#define RAY_H

#include "vec3.h"

class ray {
  public:
    ray() {}

    ray(const point3& origin, const vec3& direction) : orig(origin), dir(direction) {}

    const point3& origin() const  { return orig; }
    const vec3& direction() const { return dir; }

    point3 at(double t) const {
        return orig + t*dir;
    }

  private:
    point3 orig;
    vec3 dir;
};

#endif
```

Tarefa 6 – Camera I

```
auto aspect_ratio = 16.0 / 9.0;
int image_width = 400;

// Calculate the image height, and ensure that it's at least 1.
int image_height = int(image_width / aspect_ratio);
image_height = (image_height < 1) ? 1 : image_height;

// Viewport widths less than one are ok since they are real valued.
auto viewport_height = 2.0;
auto viewport_width = viewport_height * (double(image_width)/image_height);
```

Tarefa 7 – Varredura da imagem/camera

```
#include "color.h"
#include "ray.h"
#include "vec3.h"

#include <iostream>

color ray_color(const ray& r) {
    return color(0,0,0);
}
```

Tarefa 7 – Varredura da imagem/camera

```
int main() {

    // Image

    auto aspect_ratio = 16.0 / 9.0;
    int image_width = 400;

    // Calculate the image height, and ensure that it's at least 1.
    int image_height = int(image_width / aspect_ratio);
    image_height = (image_height < 1) ? 1 : image_height;

    // Camera

    auto focal_length = 1.0;
    auto viewport_height = 2.0;
    auto viewport_width = viewport_height * (double(image_width)/image_height);
    auto camera_center = point3(0, 0, 0);

    // Calculate the vectors across the horizontal and down the vertical viewport edges.
    auto viewport_u = vec3(viewport_width, 0, 0);
    auto viewport_v = vec3(0, -viewport_height, 0);

    // Calculate the horizontal and vertical delta vectors from pixel to pixel.
    auto pixel_delta_u = viewport_u / image_width;
    auto pixel_delta_v = viewport_v / image_height;

    // Calculate the location of the upper left pixel.
    auto viewport_upper_left = camera_center
                             - vec3(0, 0, focal_length) - viewport_u/2 - viewport_v/2;
    auto pixel00_loc = viewport_upper_left + 0.5 * (pixel_delta_u + pixel_delta_v);
```

Tarefa 7 – Varredura da imagem/camera

```
    // Render

    std::cout << "P3\n" << image_width << " " << image_height << "\n255\n";

    for (int j = 0; j < image_height; j++) {
        std::clog << "\rScanlines remaining: " << (image_height - j) << ' ' << std::flush;
        for (int i = 0; i < image_width; i++) {
            auto pixel_center = pixel00_loc + (i * pixel_delta_u) + (j * pixel_delta_v);
            auto ray_direction = pixel_center - camera_center;
            ray r(camera_center, ray_direction);

            color pixel_color = ray_color(r);
            write_color(std::cout, pixel_color);
        }
    }

    std::clog << "\rDone.                 \n";
}
```

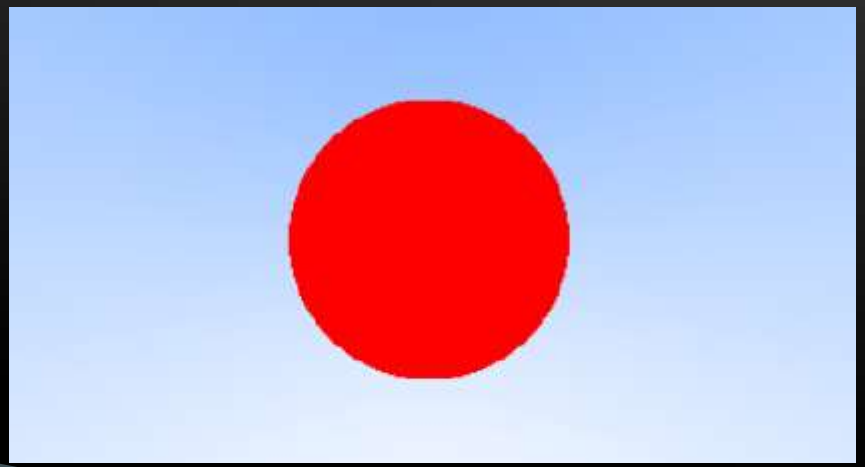
Tarefa 8 – Teste da câmera e raios

```
#include "color.h"
#include "ray.h"
#include "vec3.h"

#include <iostream>


color ray_color(const ray& r) {
    vec3 unit_direction = unit_vector(r.direction());
    auto a = 0.5*(unit_direction.y() + 1.0);
    return (1.0-a)*color(1.0, 1.0, 1.0) + a*color(0.5, 0.7, 1.0);
}
```

LERP (Linear Interpolation)
blendedValue=(1−a)·startValue+a·endValue,

Tarefa 9 – Desenho de esfera

```
bool hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = center - r.origin();
    auto a = dot(r.direction(), r.direction());
    auto b = -2.0 * dot(r.direction(), oc);
    auto c = dot(oc, oc) - radius*radius;
    auto discriminant = b*b - 4*a*c;
    return (discriminant >= 0);
}

color ray_color(const ray& r) {
    if (hit_sphere(point3(0,0,-1), 0.5, r))
        return color(1, 0, 0);

    vec3 unit_direction = unit_vector(r.direction());
    auto a = 0.5*(unit_direction.y() + 1.0);
    return (1.0-a)*color(1.0, 1.0, 1.0) + a*color(0.5, 0.7, 1.0);
}
```



Tarefa 10 – Normal Map esfera

```
double hit_sphere(const point3& center, double radius, const ray& r) {
    vec3 oc = center - r.origin();
    auto a = dot(r.direction(), r.direction());
    auto b = -2.0 * dot(r.direction(), oc);
    auto c = dot(oc, oc) - radius*radius;
    auto discriminant = b*b - 4*a*c;

    if (discriminant < 0) {
        return -1.0;
    } else {
        return (-b - std::sqrt(discriminant) ) / (2.0*a);
    }
}

color ray_color(const ray& r) {
    auto t = hit_sphere(point3(0,0,-1), 0.5, r);
    if (t > 0.0) {
        vec3 N = unit_vector(r.at(t) - vec3(0,0,-1));
        return 0.5*color(N.x()+1, N.y()+1, N.z()+1);
    }

    vec3 unit_direction = unit_vector(r.direction());
    auto a = 0.5*(unit_direction.y() + 1.0);
    return (1.0-a)*color(1.0, 1.0, 1.0) + a*color(0.5, 0.7, 1.0);
}
```

Tarefa 11 – Classe Hittable

```
#ifndef HITTABLE_H
#define HITTABLE_H

#include "ray.h"

class hit_record {
  public:
    point3 p;
    vec3 normal;
    double t;
};

class hittable {
  public:
    virtual ~hittable() = default;

    virtual bool hit(const ray& r, double ray_tmin, double ray_tmax, hit_record& rec) const = 0;
};

#endif
```

Tarefa 11 – Classe Hittable

```
#ifndef SPHERE_H
#define SPHERE_H

#include "hittable.h"
#include "vec3.h"

class sphere : public hittable {
  public:
    sphere(const point3& center, double radius) : center(center), radius(std::fmax(0,radius)) {}

    bool hit(const ray& r, double ray_tmin, double ray_tmax, hit_record& rec) const override {
        vec3 oc = center - r.origin();
        auto a = r.direction().length_squared();
        auto h = dot(r.direction(), oc);
        auto c = oc.length_squared() - radius*radius;

        auto discriminant = h*h - a*c;
        if (discriminant < 0)
            return false;

        auto sqrtd = std::sqrt(discriminant);

        // Find the nearest root that lies in the acceptable range.
        auto root = (h - sqrtd) / a;
        if (root <= ray_tmin || ray_tmax <= root) {
            root = (h + sqrtd) / a;
            if (root <= ray_tmin || ray_tmax <= root)
                return false;
        }
```

Tarefa 11 – Classe Hittable

```
        rec.t = root;
        rec.p = r.at(rec.t);
        rec.normal = (rec.p - center) / radius;

        return true;
    }

  private:
    point3 center;
    double radius;
};

#endif
```

Tarefa 12 – Lista de objetos

```
#ifndef HITTABLE_LIST_H
#define HITTABLE_LIST_H

#include "hittable.h"

#include <memory>
#include <vector>

using std::make_shared;
using std::shared_ptr;

class hittable_list : public hittable {
  public:
    std::vector<shared_ptr<hittable>> objects;

    hittable_list() {}
    hittable_list(shared_ptr<hittable> object) { add(object); }

    void clear() { objects.clear(); }

    void add(shared_ptr<hittable> object) {
        objects.push_back(object);
    }
```

Tarefa 12 – Lista de objetos

```
    bool hit(const ray& r, double ray_tmin, double ray_tmax, hit_record& rec) const override {
        hit_record temp_rec;
        bool hit_anything = false;
        auto closest_so_far = ray_tmax;

        for (const auto& object : objects) {
            if (object->hit(r, ray_tmin, closest_so_far, temp_rec)) {
                hit_anything = true;
                closest_so_far = temp_rec.t;
                rec = temp_rec;
            }
        }

        return hit_anything;
    }
};

#endif
```

Tarefa 13 – Refatorando a Camera

```
#ifndef CAMERA_H
#define CAMERA_H

#include "hittable.h"

class camera {
  public:
    /* Public Camera Parameters Here */

    void render(const hittable& world) {
        ...
    }

  private:
    /* Private Camera Variables Here */

    void initialize() {
        ...
    }

    color ray_color(const ray& r, const hittable& world) const {
        ...
    }
};

#endif
```

Tarefa 13 – Refatorando a Camera

```
class camera {
  ...

  private:
    ...


    color ray_color(const ray& r, const hittable& world) const {
        hit_record rec;

        if (world.hit(r, interval(0, infinity), rec)) {
            return 0.5 * (rec.normal + color(1,1,1));
        }

        vec3 unit_direction = unit_vector(r.direction());
        auto a = 0.5*(unit_direction.y() + 1.0);
        return (1.0-a)*color(1.0, 1.0, 1.0) + a*color(0.5, 0.7, 1.0);
    }
};

#endif
```

Tarefa 13 – Refatorando a Camera

```
class camera {
  public:
    double aspect_ratio = 1.0;  // Ratio of image width over height
    int    image_width  = 100;  // Rendered image width in pixel count

    void render(const hittable& world) {
        initialize();

        std::cout << "P3\n" << image_width << ' ' << image_height << "\n255\n";

        for (int j = 0; j < image_height; j++) {
            std::clog << "\rScanlines remaining: " << (image_height - j) << ' ' << std::flush;
            for (int i = 0; i < image_width; i++) {
                auto pixel_center = pixel00_loc + (i * pixel_delta_u) + (j * pixel_delta_v);
                auto ray_direction = pixel_center - center;
                ray r(center, ray_direction);

                color pixel_color = ray_color(r, world);
                write_color(std::cout, pixel_color);
            }
        }

        std::clog << "\rDone.                 \n";
    }
```

Tarefa 13 – Refatorando a Camera

```
private:
  int    image_height;   // Rendered image height
  point3 center;         // Camera center
  point3 pixel00_loc;    // Location of pixel 0, 0
  vec3   pixel_delta_u;  // Offset to pixel to the right
  vec3   pixel_delta_v;  // Offset to pixel below

  void initialize() {
      image_height = int(image_width / aspect_ratio);
      image_height = (image_height < 1) ? 1 : image_height;

      center = point3(0, 0, 0);

      // Determine viewport dimensions.
      auto focal_length = 1.0;
      auto viewport_height = 2.0;
      auto viewport_width = viewport_height * (double(image_width)/image_height);

      // Calculate the vectors across the horizontal and down the vertical viewport edges.
      auto viewport_u = vec3(viewport_width, 0, 0);
      auto viewport_v = vec3(0, -viewport_height, 0);

      // Calculate the horizontal and vertical delta vectors from pixel to pixel.
      pixel_delta_u = viewport_u / image_width;
      pixel_delta_v = viewport_v / image_height;

      // Calculate the location of the upper left pixel.
      auto viewport_upper_left =
          center - vec3(0, 0, focal_length) - viewport_u/2 - viewport_v/2;
      pixel00_loc = viewport_upper_left + 0.5 * (pixel_delta_u + pixel_delta_v);
  }
```

Tarefa 13 – Refatorando a Camera
(como ficou a main, depois de refatorar)

```
#include "rtweekend.h"

#include "camera.h"
#include "hittable.h"
#include "hittable_list.h"
#include "sphere.h"

color ray_color(const ray& r, const hittable& world) {
    ...
}

int main() {
    hittable_list world;

    world.add(make_shared<sphere>(point3(0,0,-1), 0.5));
    world.add(make_shared<sphere>(point3(0,-100.5,-1), 100));

    camera cam;

    cam.aspect_ratio = 16.0 / 9.0;
    cam.image_width  = 400;

    cam.render(world);
}
```

Tarefa 14 – Anti-aliasing with random

```
#include <cmath>
#include <cstdlib>
#include <iostream>
#include <limits>
#include <memory>
...

// Utility Functions

inline double degrees_to_radians(double degrees) {
    return degrees * pi / 180.0;
}

inline double random_double() {
    // Returns a random real in [0,1).
    return std::rand() / (RAND_MAX + 1.0);
}

inline double random_double(double min, double max) {
    // Returns a random real in [min,max).
    return min + (max-min)*random_double();
}
```

Tarefa 14 – or std random

```
...

#include <random>

...

inline double random_double() {
    static std::uniform_real_distribution<double> distribution(0.0, 1.0);
    static std::mt19937 generator;
    return distribution(generator);
}

inline double random_double(double min, double max) {
    // Returns a random real in [min,max).
    return min + (max-min)*random_double();
}

...
```

Tarefa 14 – or std random

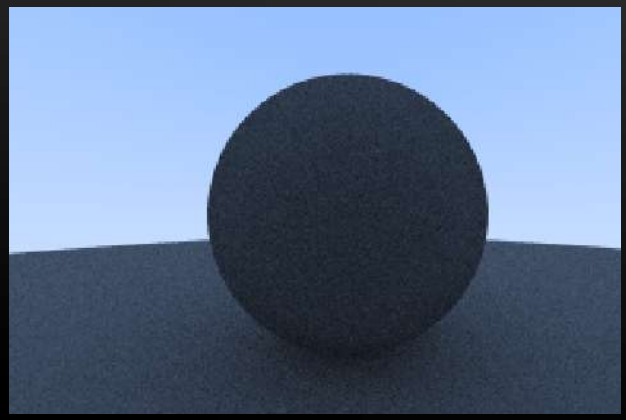
```
class camera {
  public:
    double aspect_ratio      = 1.0;  // Ratio of image width over height
    int    image_width       = 100;  // Rendered image width in pixel count
    int    samples_per_pixel = 10;   // Count of random samples for each pixel

    void render(const hittable& world) {
        initialize();

        std::cout << "P3\n" << image_width << ' ' << image_height << "\n255\n";

        for (int j = 0; j < image_height; j++) {
            std::clog << "\rScanlines remaining: " << (image_height - j) << ' ' << std::flush;
            for (int i = 0; i < image_width; i++) {
                color pixel_color(0,0,0);
                for (int sample = 0; sample < samples_per_pixel; sample++) {
                    ray r = get_ray(i, j);
                    pixel_color += ray_color(r, world);
                }
                write_color(std::cout, pixel_samples_scale * pixel_color);
            }
        }

        std::clog << "\rDone.                 \n";
    }
    ...
```

Tarefa 14 – or std random

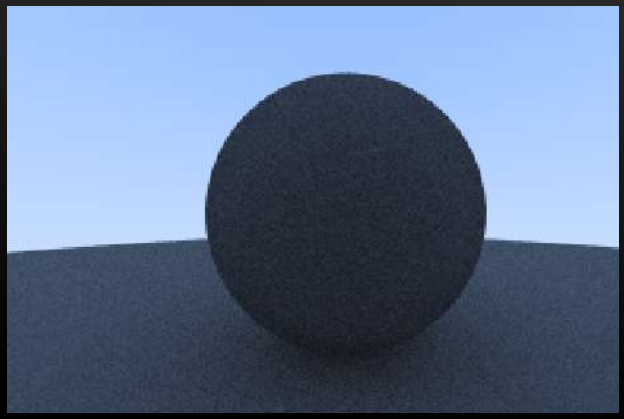
```
  private:
    int    image_height;         // Rendered image height
    double pixel_samples_scale;  // Color scale factor for a sum of pixel samples
    point3 center;               // Camera center
    point3 pixel00_loc;          // Location of pixel 0, 0
    vec3   pixel_delta_u;        // Offset to pixel to the right
    vec3   pixel_delta_v;        // Offset to pixel below

    void initialize() {
        image_height = int(image_width / aspect_ratio);
        image_height = (image_height < 1) ? 1 : image_height;

        pixel_samples_scale = 1.0 / samples_per_pixel;

        center = point3(0, 0, 0);
        ...
    }

    ray get_ray(int i, int j) const {
        // Construct a camera ray originating from the origin and directed at randomly sampled
        // point around the pixel location i, j.

        auto offset = sample_square();
        auto pixel_sample = pixel00_loc
                          + ((i + offset.x()) * pixel_delta_u)
                          + ((j + offset.y()) * pixel_delta_v);

        auto ray_origin = center;
        auto ray_direction = pixel_sample - ray_origin;

        return ray(ray_origin, ray_direction);
    }

    vec3 sample_square() const {
        // Returns the vector to a random point in the [-.5,-.5]-[+.5,+.5] unit square.
        return vec3(random_double() - 0.5, random_double() - 0.5, 0);
    }

    color ray_color(const ray& r, const hittable& world) const {
        ...
    }
};

#endif
```

Tarefa 15 – difuso basico

```
class camera {
  ...
  private:
    ...
    color ray_color(const ray& r, const hittable& world) const {
        hit_record rec;

        if (world.hit(r, interval(0, infinity), rec)) {
            vec3 direction = random_on_hemisphere(rec.normal);
            return 0.5 * ray_color(ray(rec.p, direction), world);
        }

        vec3 unit_direction = unit_vector(r.direction());
        auto a = 0.5*(unit_direction.y() + 1.0);
        return (1.0-a)*color(1.0, 1.0, 1.0) + a*color(0.5, 0.7, 1.0);
    }
};
```



Tarefa 15 – difuso básico, limitar recursão, fix Acne problem, increase lambertian distribution

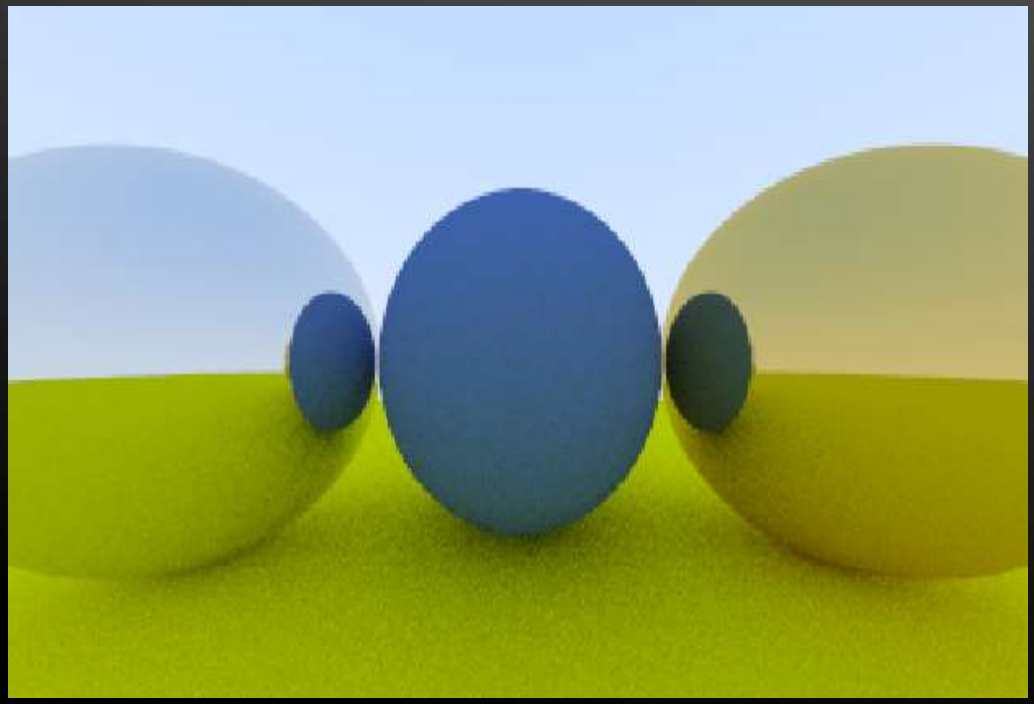


Tarefa 16 – Classe Material, Albedo (10.3)

Tarefa 17 – Reflexo

```
...

class lambertian : public material {
    ...
};

class metal : public material {
  public:
    metal(const color& albedo) : albedo(albedo) {}

    bool scatter(const ray& r_in, const hit_record& rec, color& attenuation, ray& scattered)
    const override {
        vec3 reflected = reflect(r_in.direction(), rec.normal);
        scattered = ray(rec.p, reflected);
        attenuation = albedo;
        return true;
    }

  private:
    color albedo;
};
```

Tarefa 17 – Testar com varias esferas



Tarefa 17* (opcional) – Implementar Refracao/Transparência

Tarefa 18 – Free Camera