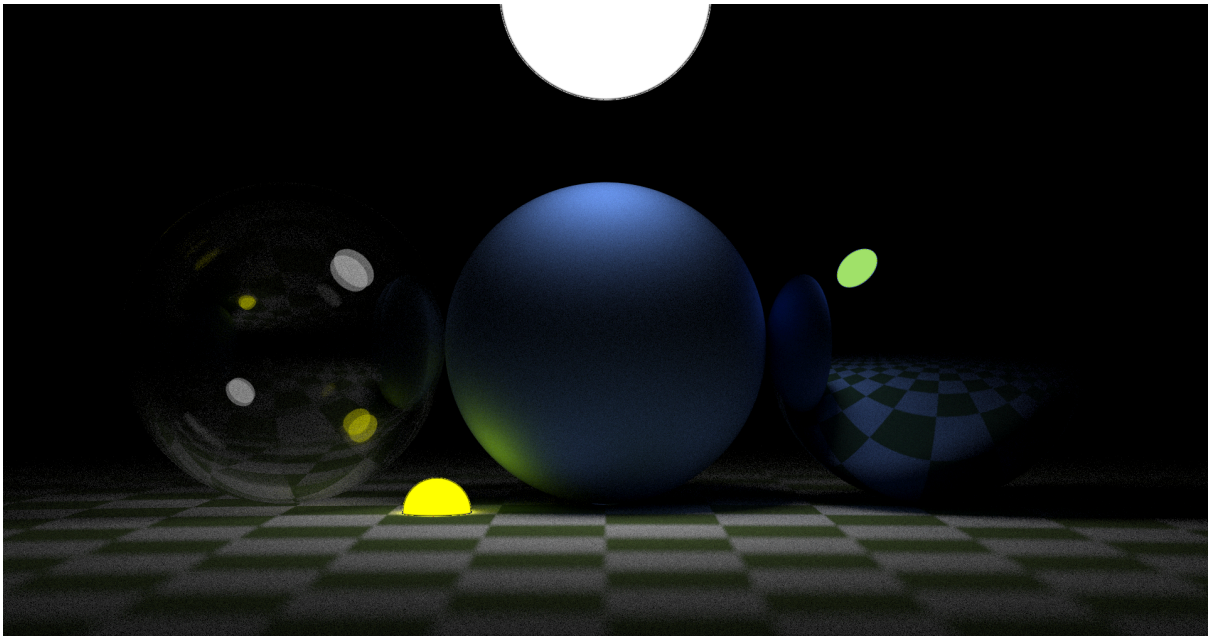
Raytracer

Lucka Valtriani, Steven Deffontaine, Arthur Guyetand, Mickael Frigo



Factory :

```
Hitable *Factory::createLight(Vec3 center, float intensity, Vec3 color)
```

Elle est utilisée pour créer les lights de la scène avec les informations récupérées du parsing.

Observer :

```
void setFilename(std::string filename);
void firstObserver();
bool isModified();
char *getFilename() { return (char *)_filename.c_str(); };
```

Il est utilisée pour recharger le fichier config que l'on parse en pleins rendu

Builder :

```
Builder();
~Builder() = default;
Builder& setType(std::string type);
Builder& setCenter(Vec3 center);
Builder& setRadius(float radius);
Builder& setHeight(float height);
Builder& setLength(float length);
Builder& setWidth(float width);
Builder& setColor(Vec3 color);
Builder& setPoint1(Vec3 point1);
Builder& setPoint2(Vec3 point2);
Builder& setPoint3(Vec3 point3);
Builder& setTexturePatch(std::string type);
Builder& setTexture(std::string type);
Builder& setMaterial(std::string type);
Hitable *buildObject();
```

Il est utilisé pour construire un objet, chaque méthode “set” définit une propriété de l’objet à construire et renvoie le Builder. Une fois les propriétés définies, la méthode “buildObject” est appelée pour créer l’objet.

Parsing :

```
void help() override;
void parseFile(const std::string &file) override;
void checkArgs(int ac, char **av) override;
void loadData() override;
void dumpData() override;
libconfig::Setting &lookUp(const std::string &path) override;
void lookUpValue(const std::string &path, int &value) override;
void lookUpValue(const std::string &path, std::string &value) override;
void lookUpValue(const std::string &path, float &value) override;
camera getCamera() const { return _camera; }
light getLight() const { return _light; }
Hitable_list *getWorld() const { return _world; }
```

Il parle tout simplement le fichier config dans lequel il y a toutes les informations nécessaires pour la créations de la scène et des objets. Le parsing utilise la librairie LibConfig pour récupérer les informations du fichier.

Render :

```
void RenderImage(Parsing &parsing, std::string filename);
Vec3 RayColor(const Ray &r, Hitable *world, int depth);

Vec3 Background;
```

Le Render va tout simplement détecter tous les objets de la scène avec des rayons, va calculer la lumière émise par les objets et pour les matériaux. Le Render permet de tout généré et faire un rendu dans un fichier PPM (il peut aussi se faire en fenêtre SFML)

SFML :

```
void createWindow(int width, int height);
void setPixelColor(int x, int y, sf::Color color);
void destroyWindow();
void loop();
void display_window();
```

La SFML est utilisée pour afficher le rendu dans une fenêtre SFML, les fonctions créer simplement une fenêtre, set les pixels et ensuite display le rendu.

Textures :

```
virtual Vec3 Value(float u, float v, const Vec3 &p) const = 0;
```

```
class Solid_color : public Textures {
    public:
        Solid_color(const Vec3 &albedo) : _albedo(albedo) {}
        Solid_color(double red, double green, double blue) : Solid_color(Vec3(red, green, blue)) {}
        Codeium: Refactor | Explain | Generate Function Comment | X
        Vec3 Value(float u, float v, const Vec3 &p) const override {
```

```
class Checker_texture : public Textures {
    public:
        Codeium: Refactor | Explain | Generate Function Comment | X
        Checker_texture(double scale, std::shared_ptr<Textures> even, std::shared_ptr<Textures> odd)
        : inv_scale(1.0 / scale), even(even), odd(odd) {}
        Codeium: Refactor | Explain | Generate Function Comment | X
        Checker_texture(double scale, const Vec3 &c1, const Vec3 &c2)
        : inv_scale(1.0 / scale), even(std::make_shared<Solid_color>(c1)), odd(std::make_shared<Solid_color>(c2))
        Vec3 Value(float u, float v, const Vec3 &p) const override;
```

```
class Image_texture : public Textures {
    public:
        Image_texture(const char *filename) : _image(filename) {};
        Vec3 Value(float u, float v, const Vec3 &p) const override;
```

Il y a plusieurs texture, la texture avec une couleur uni (Solid_color) la texture a damier (Checker_texture) et la texture via fichier d'image (Image_texture)

Materials :

```
class Materials {
    public:
        Materials() = default;
        ~Materials() = default;
        virtual bool scatter(const Ray &r_in, const hit_record &rec, Vec3 &attenuation, Ray &scattered) const = 0;
        virtual Vec3 emitted(double u, double v, const Vec3 &p) const = 0;
```

```
class Lambertian : public Materials {
    public:
        Lambertian(const Vec3 &a) : Tex(std::make_shared<Solid_color>(albedo)) { (void)a; }
        Lambertian(std::shared_ptr<Textures> Tex) : Tex(Tex) {}
        Codeium: Refactor | Explain | Generate Function Comment | X
        virtual bool scatter(const Ray &r_in, const hit_record &rec, Vec3 &attenuation, Ray &scattered) const {
```

```
class Metal : public Materials {
    public:
        Metal(const Vec3 &a, float f) : albedo(a) {if(f<1) fuzz=f; else fuzz=1;}
        Codeium: Refactor | Explain | Generate Function Comment | X
        virtual bool scatter(const Ray &r_in, const hit_record &rec, Vec3 &attenuation, Ray &scattered) const {
```

```
class Dielectric : public Materials {
    public:
        Dielectric(float ri) : refect_index(ri) {};
        Codeium: Refactor | Explain | Generate Function Comment | X
        virtual bool scatter(const Ray& r_in, const hit_record& rec, Vec3& attenuation, Ray& scattered) const  {
```

Il y a différents matériaux :
Lambertian diffuse la lumière uniformément dans toutes les directions
Métal réfléchit la lumière dans une direction spécifique
Dielectric réfracte la lumière

Lights :

```
class Diffuse_Light : public Materials {
    public:
        Diffuse_Light(std::shared_ptr<Textures> tex) : emit(tex) {}
        Diffuse_Light(const Vec3 &c) : emit(std::make_shared<Solid_color>(c)) {}

        Codeium: Refactor | Explain | Generate Function Comment | X
        virtual bool scatter(const Ray &r_in, const hit_record &rec, Vec3 &attenuation, Ray &scattered) const {
```

Elle représente tout simplement une source de lumière qui est émise dans toutes les directions autour d'elle

CreatePPM :

```
void createPPM() override;
void setPixel(int x, int y, int r, int g, int b) override;
void setResolution(int x, int y) override;
void setFilename(std::string filename) override;
```

CreatePPM permet de créer le fichier .ppm de sortie qui contient le rendu de la scène et des objets, on met chaque pixel de couleur a chaque position

Caméra :

```
Camera(Parsing &parsing);
void CameraCalcul(Vec3 lookfrom, Vec3 lookat, Vec3 vup, float fov, float aspect , float aperture, float focus_dist);
~Camera() = default;

Vec3 randomInUnitDisk();

//getters
Codeium: Refactor | Explain | X
Ray getRay(float u, float v);
```

La caméra est utilisée dans le scène de ray tracing, elle est définie par sa position, sa rotation, son champ de vision (fov), sa profondeur de champ (dof) et son ouverture. Toutes ces informations sont récupère via le fichier config.

Hitable / Hitable_list

```cpp
class Hitable_list : public Hitable {
    public:
        Hitable_list() {};
        ~Hitable_list() = default;

        Hitable_list(Hitable **l, int n);
        virtual bool hit(const Ray &r, float t_min, float t_max, hit_record &rec) const;
        void dumpData() const;
        protected:
            Hitable **_list;
            int _list_size;
        private:
};
```

```cpp
struct hit_record {
    float t;
    Vec3 p;
    Vec3 normal;
    Materials *mat_ptr;
    double u;
    double v;
};

Codeium: Refactor | Explain
class Hitable {
    public:
        Hitable() = default;
        ~Hitable() = default;

        virtual bool hit(const Ray &r, float t_min, float t_max, hit_record &rec) const = 0;
    protected:
    private:
};
```

Hitable_list contient tous les objets de la scène, elle est définit par “world” dans le code, ce qui correspond à tous les objets de la scène, la caméra etc