Documentation du Code Raytracer

Classe principale Raytracer

1. Initialisation et Configuration :
    - Le constructeur prend en paramètre un fichier contenant la description de la scène à rendre, ainsi qu'un objet pour vérifier et analyser cette description.
    - La classe utilise également des bibliothèques externes pour charger dynamiquement des primitives telles que des sphères, des cylindres, des cônes, etc.
2. Rendu :
    - La méthode run() est responsable de lancer le processus de rendu.
    - Elle crée une fenêtre SFML pour afficher l'image rendue.
    - Elle parcourt les pixels de l'image, trace des rayons à travers chaque pixel et calcule la couleur de chaque rayon en fonction de la scène décrite dans le fichier.
3. Gestion des Primitives :
    - La classe utilise des gestionnaires de primitives (sphères, cylindres, plans, cônes) pour stocker les objets de la scène.
    - Elle charge dynamiquement des bibliothèques contenant des implémentations de ces primitives, ce qui permet une extensibilité du système.
4. Éclairage :
    - La classe gère les lumières directionnelles et l'éclairage ambiant de la scène.
5. Multithreading :
    - Elle peut créer l'image de manière multithreadée pour améliorer les performances de rendu.
6. Exportation :
    - Une fois le rendu terminé, la classe peut sauvegarder l'image résultante dans un fichier PPM, un format d'image portable.

Extrait de la classe :

```
class Raytracer {
   public:
       Raytracer(std::string file, check_and_parse &check_parse);
       ~Raytracer() = default;

       void run();
   private:
```

```
        PrimitiveManager<Sphere> sphereManager;
        PrimitiveManager<Cylinder> cylinderManager;
        PrimitiveManager<Plane> planeManager;
        PrimitiveManager<Cone> coneManager;

        std::vector<Point> directional_lights;
        int lightIntensity = 0;
        int ambientLight = 0;

        void create_map_multithreaded();

        void save_ppm();

        //// libsphere /////
        void load_sphere_library(const std::string& libraryPath);
        //// libcylinder /////
        void load_cylinder_library(const std::string& libraryPath);
        //// libplane /////
        void load_plane_library(const std::string& libraryPath);
        //// libcone /////
        void load_cone_library(const std::string& libraryPath);


        void primitivesClear();
};
```

Classe d’Erreur Raytracer

La classe RaytracerException définit une exception personnalisée spécifique au projet du raytracer. Voici une explication de chaque élément de la classe :

- Constructeur :
    - Le constructeur prend deux paramètres : name et type, qui sont des chaînes de caractères. name représente le message d'erreur associé à l'exception, tandis que type représente le type d'erreur, permettant une distinction précise entre différents types d'exceptions.
- Méthodes :
    - getName(): Cette méthode retourne le message d'erreur associé à l'exception. Elle est utilisée pour obtenir des informations sur l'erreur survenue.
    - getType(): Cette méthode retourne le type d'erreur. Elle permet d'identifier le type spécifique d'exception qui s'est produite.

        - what(): Cette méthode est héritée de std::exception et est redéfinie ici. Elle retourne une chaîne de caractères C-style représentant une description de l'exception. Dans ce cas, elle renvoie simplement le message d'erreur associé à l'exception.
- Attributs :
    - _message : C'est le message d'erreur associé à l'exception.
    - _type : C'est le type d'erreur associé à l'exception.

En résumé, cette classe encapsule une exception personnalisée pour le projet du raytracer, fournissant des informations sur l'erreur survenue via le message d'erreur et le type d'erreur. Elle est conçue pour faciliter la gestion des erreurs spécifiques au raytracer et améliorer la lisibilité et la précision des messages d'erreur.

Extrait de la classe :

```
class RaytracerException : public std::exception {
public:
   RaytracerException(const std::string& name, const std::string& type)
: _message(name), _type(type) {}

   const std::string& getName() const {
       return _message;
   }

   const std::string& getType() const { return _type; }
   const char *what() const noexcept override { return
_message.c_str(); }

private:
   std::string _message;
   std::string _type;
};
```

Classe Des Lights

La première classe, `ILight`, représente une interface abstraite pour un type de lumière dans le projet de raytracer. Voici une explication de chaque élément de la classe :

- **Méthodes :**
  - *virtual ~Light() {}:* Un destructeur virtuel, ce qui signifie que cette classe est destinée à être utilisée comme une interface polymorphique et qu'elle doit être dérivée pour pouvoir être instanciée.
  - *// virtual Color intensityAt(const Point& point) const = 0;*: Cette méthode virtuelle pure (commentée dans le code) définit une fonction membre qui doit être implémentée par les classes dérivées. Elle est destinée à calculer l'intensité lumineuse au niveau d'un point donné dans la scène.

La deuxième classe, `DirectionalLight`, hérite de l'interface *ILight*. Voici une explication de cette classe :

- **Constructeur :**
  - *DirectionalLight(/* paramètres de construction */) = default;*: Le constructeur par défaut de la classe.

- **Méthodes :**
  - *// Color intensityAt(const Point& point) const override { return Color(0, 0, 0); };*: Cette méthode, commentée dans le code, est censée être une implémentation de la méthode virtuelle pure `intensityAt` de la classe de base `ILight`. Elle renvoie une couleur noire pour le moment. Cette méthode devrait être implémentée pour calculer l'intensité lumineuse d'une lumière directionnelle à un point donné dans la

```
class Light {
public:
   virtual ~Light() {}
};

class DirectionalLight : public Light {
public:
   DirectionalLight(/* paramètres de construction */) = default;

   void setX(double x) { _x = x; }
   void setY(double y) { _y = y; }
   void setZ(double z) { _z = z; }
   double getX() { return _x; }
   double getY() { return _y; }
   double getZ() { return _z; }

private:
   double _x = 0;
   double _y = 0;
   double _z = 0;
};
```

La classe Primitive Manager

La classe `PrimitiveManager` est une classe générique qui gère un ensemble de primitives géométriques dans le projet de raytracer. Voici une explication de ses fonctionnalités principales :

- **Constructeur** :
  - `PrimitiveManager() {}`: Un constructeur par défaut qui initialise la classe.

- **Destructeur** :
  - `~PrimitiveManager()`: Un destructeur qui libère la mémoire allouée pour les primitives.

- **Méthodes** :
  - `void clear()`: Cette méthode supprime toutes les primitives du gestionnaire.
  - `void addPrimitive(std::unique_ptr<Primitive>&& primitive)`: Cette méthode ajoute une primitive au gestionnaire en transférant la propriété de la primitive passée en argument.
  - `bool findClosestIntersection(const Ray& ray, Intersection& intersection) const`: Cette méthode cherche l'intersection la plus proche entre un rayon donné et les primitives dans le gestionnaire. Elle retourne `true` si une intersection est trouvée et met à jour l'intersection passée en paramètre, sinon elle retourne `false`.

- **Attributs privés** :
  - `std::vector<std::unique_ptr<Primitive>> primitives_`: Un vecteur de pointeurs uniques vers des primitives, utilisé pour stocker les primitives géométriques.

En résumé, la classe `PrimitiveManager` fournit une interface pour gérer un ensemble de primitives géométriques dans le projet de raytracer. Elle permet d'ajouter des primitives, de rechercher des intersections avec les rayons, et de libérer la mémoire associée aux primitives lors de la destruction de l'objet.

```
template<typename T>
class PrimitiveManager {
public:
   PrimitiveManager() {}
   ~PrimitiveManager() {
       for (size_t i = 0; i < primitives_.size(); ++i)
           if (primitives_[i])
               primitives_[i].release();
   void clear();

   void addPrimitive(std::unique_ptr<Primitive>&& primitive);
```

```
    bool findClosestIntersection(const Ray& ray, Intersection&
intersection) const {};

private:
    std::vector<std::unique_ptr<Primitive>> primitives_;
};



template<typename... Managers>
bool findClosestIntersectionAmong(const Ray& ray, Intersection&
intersection, const Managers&... managers) {
    auto checkIntersection = [&](const auto& manager) {};

    (checkIntersection(managers), ...);
}
```

Classe Des Primitives

Cette classe Primitive définit une interface abstraite pour les primitives géométriques dans le projet de raytracer. Voici une explication des différents éléments de la classe :

- Destructeur :
 - virtual ~Primitive() {}: Un destructeur virtuel qui permet à la classe d'être utilisée comme une interface polymorphique. Les classes dérivées doivent implémenter leur propre destructeur.

- Méthodes virtuelles pures :
 - virtual bool intersect(const Ray& ray, Intersection& intersection) = 0;: Cette méthode virtuelle pure doit être implémentée par les classes dérivées pour calculer l'intersection entre un rayon et la primitive.
 - virtual void rotate(double angle, const Vector& axis) = 0;: Cette méthode virtuelle pure est utilisée pour effectuer une rotation de la primitive autour d'un axe donné et d'un certain angle.

En résumé, cette classe fournit une interface commune pour les primitives géométriques dans le raytracer, avec des méthodes pour calculer les intersections avec des rayons, effectuer des rotations, et manipuler les propriétés de position et de couleur des primitives. Les classes dérivées, telles que Sphere, Cylinder, Plane, etc., implémenteront ces méthodes pour représenter leurs spécificités géométriques.

```
class Primitive {
   public:
        virtual ~Primitive() {}
        virtual bool intersect(const Ray& ray, Intersection&
intersection) = 0;
        virtual void rotate(double angle, const Vector& axis) = 0;
};
```

Exemple d’une primitives Cone :

La classe Cone représente une primitive géométrique spécifique, un cône, dans le projet de raytracer. Voici une explication des différents éléments de la classe :

- Constructeurs :
 - Cone() = default;: Un constructeur par défaut qui est utilisé par le parseur.
 - Cone(const Point& apex, const Vector& axis_direction, double angle, double height, const Color& color, const Vector& rotation_axis, double rotation_angle_deg);: Un constructeur qui initialise un cône avec un sommet, une direction d'axe, un angle, une hauteur, une couleur et des informations de rotation.

- Méthodes :
 - bool intersect(const Ray& ray, Intersection& intersection) override;: Une méthode qui calcule l'intersection entre un rayon et le cône.
 - void rotate(double angle, const Vector& axis) override;: Une méthode pour effectuer une rotation du cône autour d'un axe donné et d'un certain angle.

- Attributs protégés :
 - Coordonnées du sommet, direction de l'axe, angle, hauteur et couleur du cône.

- Fonction externe :
 - extern "C" std::unique_ptr<Cone> create_cone(const Point& apex, const Vector& axis_direction, double angle, double height, const Color& color, const Vector& rotation_axis, double rotation_angle_deg);: Une fonction externe qui crée et retourne un pointeur unique vers un objet Cone. Cette fonction est utilisée pour charger dynamiquement des instances de Cone à partir d'une bibliothèque partagée.

```
class Cone : public Primitive {
   public:
        Cone() = default; // Parser constructor
        Cone(const Point& apex, const Vector& axis_direction, double
angle, double height, const Color& color, const Vector& rotation_axis,
double rotation_angle_deg);
```

```
        bool intersect(const Ray& ray, Intersection& intersection)
override;
        void rotate(double angle, const Vector& axis) override;


        void setRotationAxis(std::string axis) { _rotation_axis = axis;
}
        std::string getRotationAxis() { return _rotation_axis; }
        void setRotationDegree(double rD) { _rotation_degree = rD; };
        double getRotationDegree() { return _rotation_degree; }

};

extern "C" std::unique_ptr<Cone> create_cone(const Point& apex, const
Vector& axis_direction, double angle, double height, const Color&
color, const Vector& rotation_axis, double rotation_angle_deg);
```