Projet de Rendu 3D par Tracé de Rayons
Rapport

Papa Diadji BOYE et Halimata Ndiaye

Janvier 2025

Table des matières

1 Introduction

Le rendu 3D par tracé de rayons (*ray tracing*) est une technique fondamentale en infographie permettant de produire des images réalistes en simulant le comportement physique de la lumière. Cette méthode repose sur le calcul des interactions entre des rayons lumineux virtuels et les objets d'une scène, prenant en compte des phénomènes tels que la réflexion, la réfraction et les ombres.

Dans le cadre de ce projet, nous avons développé un moteur de rendu minimaliste en C++ sans l'aide de bibliothèques externes, à l'exception de SDL pour l'affichage. Ce moteur a pour objectif de générer des images en utilisant une scène composée d'objets géométriques simples (quadrilatères et sphères), une caméra, et une source lumineuse.

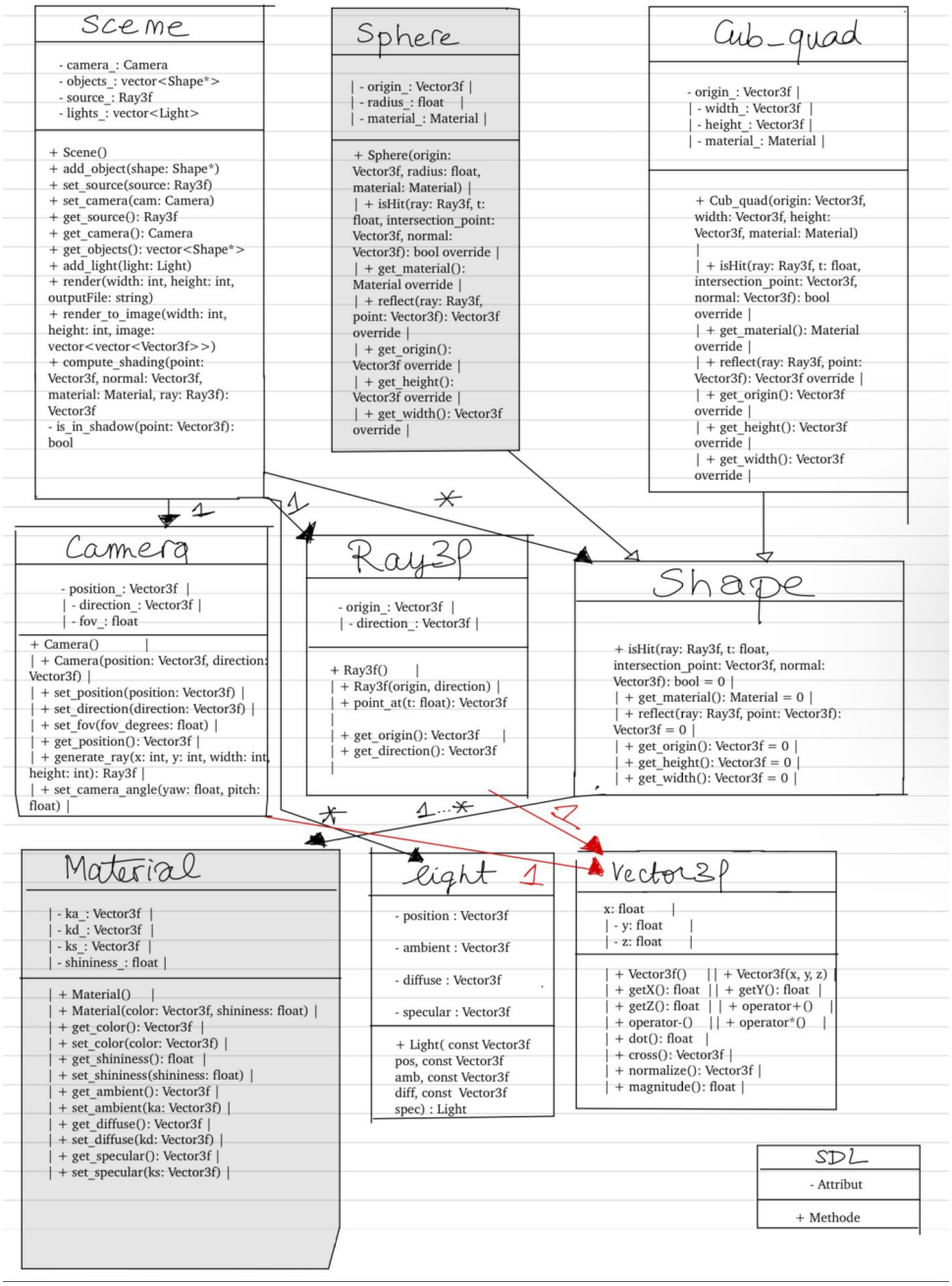
Ce rapport décrit en détail les étapes de conception et d'implémentation, les défis rencontrés, ainsi que les résultats obtenus. Il présente également des perspectives pour enrichir le projet, comme l'ajout d'autres formes géométriques ou la gestion avancée de la lumière.

2 Conception et Architecture

article graphicx

Diagramme UML

Voici une image insérée dans un document LaTeX :



La strucure de notre projet est donnée par le diagramme UML suivant (les fleches pleines représentant une dépendanse et les flèches vides un heritage) :

3 Implémentation

Les fichiers `.cpp` des classes présentées dans le diagrammes uml ont été implémentés pour modéliser et gérer les différentes composantes nécessaires à la réalisation d'un moteur de lancer de rayons. Chaque fichier joue un rôle spécifique dans l'architecture du programme, allant de la gestion des structures géométriques jusqu'aux calculs d'éclairage et de rendu. Voici dans cette section une description de l'implementation de chaque classe.

3.1 Classe `Sdl`

Rôle La classe `Sdl` gère l'affichage des images générées à l'aide de la bibliothèque SDL. Elle crée la fenêtre et un renderer SDL.

Méthodes principales

- `SDL_CreateWindow` : Crée une fenêtre SDL.
- `SDL_CreateRenderer` : Crée un renderer pour dessiner dans cette fenêtre.
- `SDL_Init` : vérifie que SDL est correctemnt initialisé.
- En cas d'échec, des messages d'erreur explicites sont affichés, et une exception est levée pour interrompre le programme proprement. Si une étape échoue (initialisation, création de la fenêtre ou du renderer), les ressources déjà allouées sont libérées avant de quitter.

3.2 Classe `Vector3f`

Rôle La classe Vector3f représente un vecteur en trois dimensions, essentiel pour les calculs géométriques dans le contexte du lancer de rayons. Elle encapsule trois coordonnées flottantes (x, y, z) et fournit des outils pour manipuler ces vecteurs de manière intuitive. Les constructeurs permettent d'initialiser un vecteur à des valeurs spécifiques ou par défaut à (0, 0, 0).

Les opérations fondamentales comme l'addition (operator+), la soustraction (operator-), et la multiplication scalaire ou vectorielle (operator*) sont définies pour faciliter plus tard le calcul de rayon et de distance(par exempe das la fonction isHit). La classe implémente également des fonctionnalités essentielles comme le produit scalaire (dot), le produit vectoriel (cross), la normalisation (normalize) pour obtenir un vecteur unitaire, et le calcul de la norme (magnitude). La méthode clamp permet de limiter chaque composante à un intervalle donné. //

Ces fonctions sont implémentées avec des opérations mathématiques directes sur les coordonnées.

Constructeurs

- **Constructeur par défaut :** Initialise le vecteur à $(0, 0, 0)$.
- **Constructeur avec paramètres :** Initialise le vecteur avec des coordonnées spécifiques (x, y, z).

Méthodes principales

- `operator+`, `operator-`, `operator*` : Permettent d'effectuer des opérations vectorielles (addition, soustraction, multiplication scalaire ou vectorielle).
- `dot` : Calcule le produit scalaire.
- `cross` : Calcule le produit vectoriel.
- `normalize` : Normalise le vecteur pour obtenir un vecteur unitaire.
- `magnitude` : Calcule la norme (longueur) du vecteur.
- `clamp` : Contraint chaque composante du vecteur à un intervalle donné.

3.3 Classe `Ray3f`

Rôle La classe `Ray3f` modélise un rayon dans l'espace tridimensionnel, défini par une origine et une direction normalisée. Les rayons sont utilisés pour projeter des pixels depuis la caméra, détecter les intersections avec les objets, et déterminer la couleur des pixels dans l'image finale.

Constructeurs

— **Par défaut :** Initialise un rayon avec une origine $(0, 0, 0)$ et une direction $(1, 0, 0)$.
— **Avec paramètres :** Permet de spécifier l'origine et la direction.

Méthodes principales

— `point_at` : Calcule un point à une distance paramétrique t le long du rayon. Elle est utilisé dans la fonction `isHit` de la classe `Cub_quad` pour calculer la distance entre le point d'intersection rayon-quadrilatère et la caméra.

3.4 Classe `Material`

Rôle La classe `Material` modélise les propriétés d'un matériau, déterminant comment un objet interagit avec la lumière. Ces propriétés influencent l'apparence visuelle de l'objet, notamment sa couleur, son éclat, et ses reflets. Cette classe est utilisée dans le contexte du lancer de rayons pour simuler des effets d'éclairage réalistes, comme la réflexion diffuse, la réflexion spéculaire, et la brillance en utilisant le modèle d'éclairage de Phong.

3.4.1 Modèle d'éclairage de Phong

Le modèle d'éclairage de Phong se compose de trois contributions lumineuses : la lumière ambiante, la lumière diffuse et la lumière spéculaire. Ces composantes permettent de simuler des interactions réalistes entre la lumière et les surfaces dans une scène 3D.

— **Lumière ambiante (L_{ambiant})**
La lumière ambiante ajoute un éclairage constant et uniforme, simulant une lumière environnementale globale qui éclaire toutes les surfaces de manière égale.

$$L_{\text{ambiant}} = k_a \cdot I_{\text{ambiant}} \tag{1}$$

Où :
— k_a est le coefficient de réflexion ambiante du matériau.
— I_{ambiant} est l'intensité de la lumière ambiante.

— **Lumière diffuse (L_{diffus})**
La lumière diffuse dépend de l'orientation de la surface par rapport à la source lumineuse. Elle est calculée selon le modèle de Lambert, où l'intensité lumineuse est proportionnelle au cosinus de l'angle entre la normale de la surface (N) et la direction de la lumière (L).

$$L_{\text{diffus}} = k_d \cdot I_{\text{diffus}} \cdot \max(0, N \cdot L) \tag{2}$$

Où :
— k_d est le coefficient de réflexion diffuse du matériau.
— I_{diffus} est l'intensité de la lumière diffuse.

— $N \cdot L$ est le produit scalaire entre la normale (N) et la direction de la lumière (L).

— **Lumière spéculaire ($L_{\text{speculaire}}$)**

La lumière spéculaire simule les reflets brillants, qui dépendent de la direction de la réflexion (R) et de la position de l'observateur (V). Elle est contrôlée par un facteur de brillance α, qui détermine la concentration du reflet.

$$L_{\text{speculaire}} = k_s \cdot I_{\text{speculaire}} \cdot (\max(0, R \cdot V))^{\alpha} \tag{3}$$

Où :

— k_s est le coefficient de réflexion spéculaire du matériau.

— $I_{\text{speculaire}}$ est l'intensité de la lumière spéculaire.

— R est le vecteur réfléchi, calculé par :

$$R = 2(N \cdot L)N - L$$

— V est la direction vers l'observateur.

— α (ou shininess dans le code) contrôle la taille et l'intensité du reflet.

Ces calculs sont réalisés dans la fonction render_to_image (cf 3.9 Classe Scene)

Attributs principaux

— `ka_`, `kd_`, `ks_` : Coefficients de réflexion ambiante, diffuse et spéculaire.

— `shininess_` : Facteur de brillance, entre 0 et 1.

Méthodes principales

— Accesseurs et mutateurs pour les coefficients de réflexion et la brillance.

— `set_shininess` : Contraint la brillance à l'intervalle $[0, 1]$.

3.5 Classe `Light`

La structure `Light` représente une source lumineuse.

Attributs principaux

— `position` : Position de la lumière.

— `ambient`, `diffuse`, `specular` : Composantes lumineuses.

Rôle Cette structure est essentielle pour calculer l'éclairage et les ombres dans la scène.

3.6 Classe `Camera`

La classe `Camera` modélise la caméra et génère des rayons pour chaque pixel. Grâce à la fonction generate_ray chaque rayon est projeté depuis la position de la caméra à travers un point spécifique sur un plan d'image virtuel. Les paramètres fov (champ de vision) et direction permettent de paramétrer dynamiquement notre caméra pour simuler différentes perspectives.

Attributs principaux

- `position_`, `direction_` : Position et direction de la caméra.
- `fov_` : Champ de vision. Contrôle l'angle d'ouverture de la caméra, affectant la manière dont la scène est vue (plus large ou plus étroite).

Méthodes principales

- `set_camera_angle` : Définit la direction à l'aide de rotations autour des axes X (horizontal) et Y(vertical).
- `generate_ray` : La méthode `generate_ray` joue un rôle crucial dans le processus de lancer de rayons. Elle génère un rayon partant de la position de la caméra et traversant un pixel spécifique de l'image. Ce rayon est ensuite utilisé pour déterminer les intersections avec les objets de la scène.

  Étapes : La méthode suit plusieurs étapes pour calculer la direction correcte du rayon :

  1. **Normalisation des coordonnées du pixel :** Les coordonnées (x, y) du pixel sont transformées pour être normalisées entre -1 et 1, où :

  $$u = \left(\frac{2(x + 0.5)}{\text{width}}\right) - 1, \quad v = \left(\frac{2(y + 0.5)}{\text{height}}\right) - 1$$

  où width et height sont respectivement la largeur et la hauteur de l'image.

  2. **Correction en fonction du rapport d'aspect (rapport entre la largeur et la hauteur de l'image) et du champ de vision (FOV) :** Les coordonnées sont ajustées pour tenir compte du rapport d'aspect (aspect_ratio) et de l'angle de champ de vision (FOV) :

  $$u = u \cdot \text{aspect\_ratio} \cdot \tan\left(\frac{\text{FOV}}{2}\right), \quad v = v \cdot \tan\left(\frac{\text{FOV}}{2}\right)$$

  3. **Calcul de la direction du rayon :** La direction du rayon est calculée en combinant les axes locaux de la caméra :

  $$\text{ray\_direction} = \text{direction\_} + (\text{right} \cdot u) + (\text{up} \cdot v)$$

  où :
  - direction_ est le vecteur direction de la caméra.
  - right est l'axe horizontal de la caméra.
  - up est l'axe vertical de la caméra.

  4. **Normalisation :** La direction du rayon est normalisée pour obtenir un vecteur unitaire :

  $$\text{ray\_direction} = \frac{\text{ray\_direction}}{\|\text{ray\_direction}\|}$$

  Retourne : La méthode retourne un objet `Ray3f`, représentant un rayon partant de la position de la caméra (position_) dans la direction calculée (ray_direction).

Rôle Elle détermine le point de vue et projette les rayons dans la scène.

3.7 Classe `Sphere`

Rôle La classe `Sphere` représente une sphère tridimensionnelle dans la scène, avec des propriétés géométriques et matérielles. Elle permet de :

- **détecter les intersections entre rayons et sphères :** La méthode `isHit` identifie les collisions entre un rayon et la sphère, déterminant ainsi les pixels à dessiner.
- **utilisé toute type de sphère :** La gestion du matériau permet de représenter des sphères avec des propriétés variées (miroir, mat, brillant).

Constructeur

- **Rôle :** Initialise une sphère avec un centre (`origin_`), un rayon (`radius_`), et un matériau (`material_`).
- **Détails :** Le constructeur prend des paramètres pour définir les caractéristiques géométriques et visuelles de la sphère.

Méthode `isHit`

- **Rôle :**
  - Détermine si un rayon intersecte la sphère.
  - Calcule également le point d'intersection et la normale au point d'impact.
- **Détails :** La méthode utilise l'équation quadratique dérivée de la géométrie sphérique :

$$\|\mathbf{R}(t) - \mathbf{C}\|^2 = r^2$$

où :
  - $\mathbf{R}(t) = \text{origine} + t \times \text{direction}$,
  - $\mathbf{C}$ est le centre de la sphère,
  - r est le rayon de la sphère.

Les coefficients de l'équation quadratique sont :

$$a = \text{direction} \cdot \text{direction}, \quad b = 2 \times \mathbf{oc} \cdot \text{direction}, \quad c = \mathbf{oc} \cdot \mathbf{oc} - r^2$$

où $\mathbf{oc} = \text{origine} - \text{centre}$.
- **Discriminant :** Le discriminant ($b^2 - 4ac$) détermine les solutions :
  - < 0 : Pas d'intersection.
  - > 0 : Deux intersections possibles (t_1 et t_2).
- **Choix :** La méthode choisit la plus proche intersection positive (t_1 ou t_2).

Méthode `get_material`

- **Rôle :** Retourne le matériau associé à la sphère.
- **Détails :** Permet d'accéder aux propriétés visuelles comme la réflexion, la diffusion, ou la couleur.

Méthode `reflect`

- **Rôle :** Calcule le rayon réfléchi à partir de la loi de réflexion :

$$\mathbf{R} = \mathbf{D} - 2(\mathbf{D} \cdot \mathbf{N})\mathbf{N}$$

où :
  - $\mathbf{D}$ est la direction du rayon incident,

— **N** est la normale à la surface au point d'intersection. La normale est calculée comme le vecteur entre le point d'intersection (P) et le centre de la sphère (C), puis normalisée :

$$\mathbf{N} = \frac{\mathbf{P} - \mathbf{C}}{\|\mathbf{P} - \mathbf{C}\|}$$

— Cette fonction permet de faire des calculs avancés pour l'éclairage.

Méthodes `get_origin`, `get_height`, `get_width`

— **Rôle :**
  — `get_origin` retourne le centre de la sphère.
  — `get_height` et `get_width` retournent des valeurs nulles, car une sphère n'a pas de dimensions rectangulaires.

3.8 Classe `Cub_quad`

Rôle La classe `Cub_quad` représente un quadrilatère dans l'espace tridimensionnel. Elle est utilisée dans **main.cpp** pour modéliser les surfaces planes de notre scène 3D. Cette classe permet de détecter les intersections entre un rayon et le quadrilatère et de gérer les propriétés matérielles nécessaires au rendu.

Constructeur Le constructeur de `Cub_quad` initialise un quadrilatère défini par un point d'origine (`origin_`), deux vecteurs (`width_` et `height_`) qui déterminent les dimensions et l'orientation, ainsi qu'un matériau (`material_`). Ces paramètres permettent de définir des rectangles alignés ou inclinés selon différents axes.

Méthode `isHit` La méthode `isHit` détermine si un rayon intersecte le plan du quadrilatère et vérifie si le point d'intersection se trouve à l'intérieur de ses limites. Elle suit les étapes suivantes :

— **Calcul de la normale :** La normale au plan est obtenue en utilisant le produit vectoriel des vecteurs `AB` (largeur) et `AD` (hauteur).

— **Intersection avec le plan :** L'intersection est calculée en résolvant l'équation du plan :

$$t = \frac{(\mathbf{A} - \text{origine du rayon}) \cdot \text{normale}}{\text{direction du rayon} \cdot \text{normale}}$$

Si le dénominateur est proche de zéro, le rayon est parallèle au plan.

— **Validation du point :** Le point d'intersection P est vérifié pour s'assurer qu'il se situe à l'intérieur des limites du quadrilatère en utilisant les projections sur les axes définis par `AB` et `AD`.

Méthode `reflect` Comme dans la classe sphere la méthode `reflect` calcule la direction d'un rayon réfléchi en utilisant la normale de la surface et la direction du rayon incident selon la loi de réflexion :

$$\mathbf{R} = \mathbf{D} - 2(\mathbf{D} \cdot \mathbf{N})\mathbf{N}$$

où **D** est la direction du rayon incident et **N** est la normale à la surface. Pour un quadrilatère la normale estt calculée comme le produit vectoriel des vecteurs définissant les dimensions du quadrilatère (width_ et height_), puis normalisé.

Propriétés matérielles La méthode `get_material` retourne le matériau associé au quadrilatère. Ces propriétés influencent directement l'apparence visuelle du quadrilatère dans le rendu.

3.9 class Scene :

Rôle : La classe `Scene` est l'élément centrale de du système 3D . elle organise et gère l'ensemble des objets qui composent la scène , tels que les sphères, cubes . Elle contient les sources lumineuses ,essentiels pour l'éclairage des objets .

Constructeur : Le constructeur de la classe initialise une scène vide en créant les en créant les éléments essentiels nécessaires à un rendu 3D . Il configure une `Caméra` par défaut une `Source lumieuse` , mais ne contient initialement aucun objet 3D .

Méthodes principales : La méthode `render_to_image()` génère une image 2D en effectuant un rendu de la scène 3D avec des objets, des lumières et une caméra. Voici les étapes principales, accompagnées des formules correspondantes :

- **Génération des rayons :** Pour chaque pixel (x, y), un rayon est généré par la caméra selon la fonction `generate_ray(x, y, width, height)`.
- **Initialisation de la couleur de base :** La couleur initiale de chaque pixel est noire :
$$\text{color} = (0.0f, 0.0f, 0.0f)$$
- **Test d'intersection :** Pour chaque objet dans la scène, la méthode `isHit()` est utilisée pour tester l'intersection entre le rayon et l'objet. Si une intersection est trouvée, la distance t est utilisée pour déterminer l'objet le plus proche de la caméra. L'objet intersecté est celui qui minimise t.
- **Calcul de l'éclairage :** Si un objet est intersecté, les contributions lumineuses sont calculées en utilisant trois composantes :
  - **Composante ambiante :** L'éclairage ambiant est calculé par :
  $$\text{ambient\_color} = \text{ambient\_color} + \text{light.ambient} \times \text{material.get\_ambient}()$$
  - **Composante diffuse (Lambert) :** La lumière diffuse dépend de l'angle entre la direction de la lumière L et la normale à la surface $\mathbf{N}$. L'intensité lumineuse diffuse est donnée par :
  $$I_{\text{diffuse}} = \max(\mathbf{N} \cdot \mathbf{L}, 0.0)$$
  et la couleur diffuse :
  $$\text{diffuse\_color} = \text{diffuse\_color} + \text{material.get\_diffuse}() \times \text{light.diffuse} \times I_{\text{diffuse}}$$
  - **Composante spéculaire (Phong) :** L'éclairage spéculaire dépend de l'angle entre la direction de la réflexion $\mathbf{R}$ et la direction vers la caméra $\mathbf{V}$. L'intensité spéculaire est donnée par :
  $$I_{\text{specular}} = \max(\mathbf{R} \cdot \mathbf{V}, 0.0)^{\text{shininess}}$$
  et la couleur spéculaire :
  $$\text{specular\_color} = \text{specular\_color} + \text{material.get\_specular}() \times \text{light.specular} \times I_{\text{specular}}$$

— **Combinaison des composantes :** La couleur finale du pixel est la somme des trois composantes :

$$\text{color} = \text{ambient\_color} + \text{diffuse\_color} + \text{specular\_color}$$

— **Clamping des valeurs :** La couleur est clampée pour s'assurer que les composantes RGB sont comprises entre 0 et 1 :

$$\text{color} = \text{color.clamp}(0.0f, 1.0f)$$

— **Mise à jour du pixel :** La couleur finale est affectée au pixel dans l'image 2D à la position correspondante $[y][x]$.

Ainsi, la méthode `render_to_image()` applique les modèles d'éclairage ambiant, diffus et spéculaire pour chaque pixel en fonction des objets, des lumières et de la caméra, produisant une image réaliste de la scène.

4 Résultats

5 Défis et Solutions

Durant la réalisation de ce projet , plusieurs défis techniques ont été rencontrés , tant au niveau de la conception des classes que l'implémentation des fonctionnalités essentilles pour le tracé de rayon et ces objets . Cette section détaille les principaux problèmes rencontrés et les solutions apportées .

5.1 Représentation des Objets 3D

Un premier défi a été de définir des objets géométriques simples tels que les sphéres les quadrilatères de la scene . La méthode isHit , essentielle pour détecter les intersections entre un rayon et ces objets , a nécessité une implémentation mathématique rigoureuse .
Solution : La classe `Sphere` a été conçue pour gérer les équations d'intersection entre un rayon et une sphère. La classe `Cub_quad`, représentant un quadrilatère dans l'espace, a utilisé les produits vectoriels pour calculer les normales et vérifier si le point d'intersection était à l'intérieur des limites.

5.2 Gestion de la Caméra

un autre défi était de positionner la caméra correctement , afin qu'elle regarde vers l'intérieur de la boite et à une distance adapté . Les vecteurs position et directions devaient etre définis pour que le rayon générés poientent vers les objets
Solution : La classe `Camera` a été enrichie avec des méthodes pour calculer des rayons à partir des pixels de l'image. Le positionnement de la caméra a été ajusté en fonction de boite et des objets , en utilisant des calculs vectoriels pour garantir une vue centré et aussi une méthode **set-angle-camera** qui permet de la pivoter pour adapter l'angle de vue .

5.3 Calcul de l'Éclairage

L'ajout d'une éclairage réaliste constituait une étape importante . Il a fallu intégrer les composantes ambiantes , diffuse et spéculaire dans les calculs d'ombrage ,tout en prenant compte de la direction de la lumière et les normales de objets .
Solution : Une méthode `compute_shading` a été ajoutée pour gérer ces calculs. Des structures de données comme `Light` ont permis de représenter les sources lumineuses, et des vérifications d'ombres ont été implémentées pour gérer les obstructions. // Des structure de données comme `Light` ont permis de représenter les sources lumineuses, et des vérifications d'ombres ont été implémentées pour gérer les obstructions.

5.4 Affichage et Débogage

Un dernier défi consistait à afficher correctement l'image générée dans une fenêtre et à comprendre les erreurs liées au tracé des rayons, notamment lorsque les intersections ne renvoyaient aucun objet.
Solution : La bibliothèque SDL a été utilisée pour afficher les résultats en temps réel. Des messages de débogage ont été ajoutés pour suivre les calculs des intersections et ajuster les paramètres de la caméra ou des objets si nécessaire

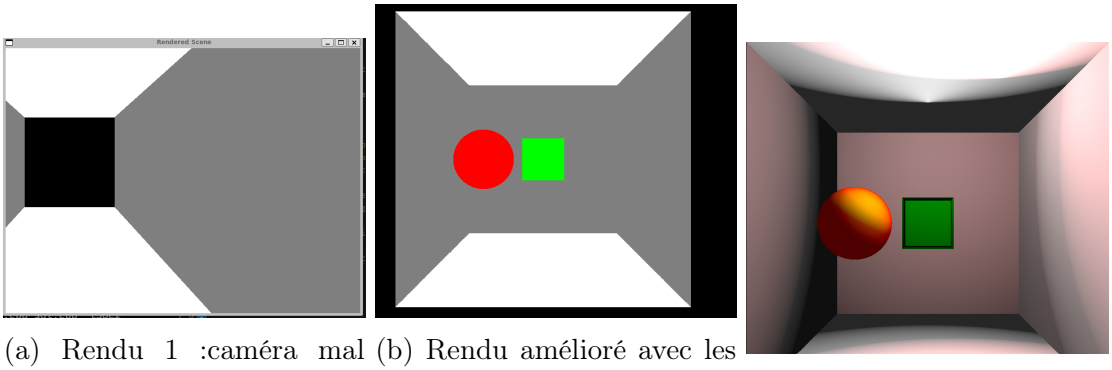
5.5 Gestion des Réflexions

Pour rendre la scène plus réaliste, il a fallu gérer les réflexions des objets brillants. Cela impliquait de calculer les rayons réfléchis en fonction des normales au point d'intersection.

5.6 Gestion des Réflexions

Pour rendre la scène plus réaliste, il a fallu gérer les réflexions des objets brillants. Cela impliquait de calculer les rayons réfléchis en fonction des normales au point d'intersection.
Solution : Les objets, via leur classe mère `Shape`, ont été dotés d'une méthode `reflect` permettant de calculer la direction d'un rayon réfléchi. Cette méthode a été utilisée lors du tracé de rayons secondaires pour simuler les réflexions.

5.7 Rendu final



(a) Rendu 1 :caméra mal placé (b) Rendu amélioré avec les deux objets (c) Rendu final

FIGURE 1 – évolution de la scene

La figure 1 montre la première scene obtenue que nous avons obtenue mais nous avions du mal à bien positionner la caméra qui semable ici dirigé dans la mauvaise direction Nous avons eu donc l'idée d'intégrer une fonction setAngleCamera qui nous permet de pivoter la caméra . Cette idée fut la bonne puisqu'elle nous a permis d'avoir le rendu 2 plus axé et dirigé vers l'intérieur de la boite .

On a ensuite voulu rendre la scene plus réaliste en intégrant une nouvelle classe Light qui permet d'ajouter des source de lumière sur la scene car la source de base ne semblait pas etre adaptée à non attentes . Nous avons aussi amélioré la classe matérial en intégrant de nouveaux coéfficients de réflexions .

6 Conclusion et Perspectives

A Annexes