Lancé de Rayons

Forma3Dev pour CPE Lyon
Sujet original de Damien Rohmer

16 & 18 octobre 2018

$$\begin{cases} \|\mathbf{x} - \mathbf{x_0}\|^2 = r^2 \\ \mathbf{x} = \mathbf{x_s} + t\mathbf{u} \end{cases}$$

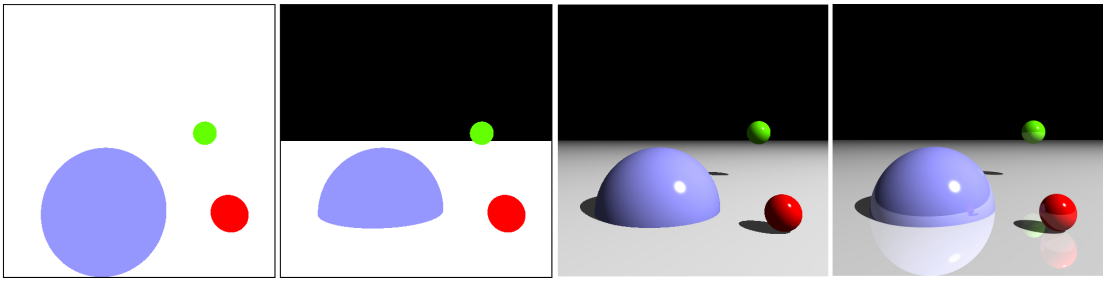

FIGURE 1 – Différentes étapes du lancé de rayons. De gauche à droite : équation du calcul d'intersection ; image des intersections ; ordonnancement des intersections suivant leur profondeur ; application de l'illumination ; réflexion.

1 But

L'objectif de ce TP est de coder un outil de rendu par lancé de rayons (ray-tracing) tel qu'on peut le trouver dans différents outils de rendu off-line (PovRay, Yafray, etc).

Nous mettrons en avant les avantages et inconvénients de cette approche par rapport au rendu projectif basé sur des triangles.

- Dans un premier temps, nous mettrons en place l'intersection entre des rayons (droites) et des primitives géométriques simples.
- Dans un second temps, nous implémenterons le calcul de la couleur associée à chaque intersection.
- Enfin, nous pourrons mettre en place différents effets réalisables aisément par lancé de rayons tels que la réflexion, l'anti-aliasing, etc.

2 Prise en main de l'environnement

2.1 Compilation

Question 1 : *C*ompilez le projet et assurez-vous que celui-ci s'exécute. Assurez-vous que vous puissiez l'éditer depuis l'éditeur de votre choix (QtCreator ou autre). Notez que le programme génère une image sur le disque dur, assurez vous que vous puissiez y accéder. N'oubliez pas que vous pouvez configurer l'emplacement du lancement de votre exécutable dans vos IDE.

2.2 Les différents répertoires des sources

Vous retrouverez les répertoires ayant déja servis dans d'autres projets tels que `lib/` et `image/`. Dans ce TP, nous ajoutons le répertoire `ray_tracing` qui contient les sous répertoires suivants :

- `rimitives` contient la définition des primitives géométriques de la scène 3D (plan, sphère), et en particulier implémente le calcul d'intersection entre la primitive et une demi-droite de l'espace. Le répertoire contient également la structure stockant

les informations relatives à une intersection entre une demi-droite et une surface.
- `render_engine` contient les différentes fonctions du lancé de rayons en tant que tels. Parcours de l'image, calcul des couleurs, des ombres, des réflexions, etc.
- `scene` contient la gestion des éléments de la scène 3D tels que la caméra, les lumières, les matériaux pour l'illumination de la surface des objets, ainsi qu'un conteneur stockant l'ensemble des informations relatives à une scène `scene_parameter`.

2.3 Programme *main*

Question 2 : *O*bservez la fonction `main()` et retrouvez les éléments suivants :
- Création d'un buffer d'image.
- Remplissage d'une scène par des objets 3D (et leurs couleurs).
- Appel à l'algorithme du lancé de rayons.
- Écriture de l'image dans un format ppm Ascii non compressé sur le disque dur.

Pour rappel, n'oubliez pas que vous devez toujours convertir une image ppm vers un format compressé (tel que png ou jpg) avant de stocker ou inclure vos images dans vos rapports. Les logiciels tels que ImageMagick ou Gimp permettent de faire ces conversions.

2.4 Lancé de rayons

Question 3 : *O*bservez la fonction `render()` dans le fichier `render_engine/render_engine.cpp` et observez la création des rayons.

3 Intersection avec les primitives 3D

3.1 Primitive générique d'une forme 3D

Dans la méthode de lancé de rayon, un objet 3D est défini uniquement par ses intersections avec une demi-droite D. La classe `primitive_basic` prévoit l'interface commune à toutes les formes que l'on peut afficher par un lancé de rayons. Chaque forme doit être en mesure de renvoyer l'intersection entre lui-même et une demi-droite stockée dans une structure `ray`. Il s'agit du rôle de la fonction `intersect()`.

Question 4 : *O*bservez la structure d'une classe `ray`.

Lors de l'appel à la méthode `intersect()`, il existe deux possibilités :
1. Il n'existe aucune intersection entre la demi-droite et l'objet, et la fonction renvoie `false`.
2. Il existe une intersection entre la demi-droite et l'objet, et la fonction renvoie `true` tout en complétant la structure `intersection_data`.

Question 5 : *O*bservez la structure d'une classe `intersection_data`. Pourquoi stocke-on la normale au point d'intersection ?

Question 6 : *E*st-ce que la variable `relative` peut prendre une valeur négative ? Pourquoi ?

3.2 Cas du plan

Soit le plan $\mathcal{P}$ donné par l'équation $< \mathbf{x} - \mathbf{x}_p, \mathbf{n}_p >= 0$, avec $\mathbf{x}_p$ un point quelconque donné du plan, et $\mathbf{n}_p$ la normale du plan.

L'intersection de ce plan avec une droite passant par $\mathbf{x}_s$ et de vecteur directeur $\mathbf{u}$ est donnée par

$$\begin{cases} < \mathbf{x} - \mathbf{x}_p, \mathbf{n}_p >= 0 \\ \mathbf{x} = \mathbf{x}_s + t\,\mathbf{u} \end{cases}$$

Question 7 : *D*éterminez le paramètre t_{inter}, solution de ce système d'équations avec son domaine de validité. Donnez la valeur de la position et de la normale associée.

La classe `plane` contient les paramètres définissant le plan.

Question 8 : *C*omplétez la méthode `intersect()` de la classe `plane` qui reçoit en paramètre le rayon incident actuel.

À ce stade, le plan gris devrait apparaître horizontale sur votre image de résultat.

3.3 Cas de la sphère

L'intersection entre une sphère de centre $\mathbf{x}_0$ et de rayon r avec une droite passant par x_s et de vecteur directeur u est donnée par la solution du système

$$\begin{cases} \|\mathbf{x} - \mathbf{x}_0\|^2 = r^2 \\ \mathbf{x} = \mathbf{x_s} + t\,\mathbf{u} \end{cases}$$

Question 9 : *D*éterminer le paramètre t_{inter} solution du système d'équations, et donnez la position $\mathbf{x}_{\text{inter}}$, ainsi que la normale associée.

Question 10 : *E*n vous inspirant du calcul de l'intersection pour le plan, écrivez la méthode `intersect` de la classe `sphere`.

À cette étape, on devra obtenir une image telle que celle montrée en figure 2.

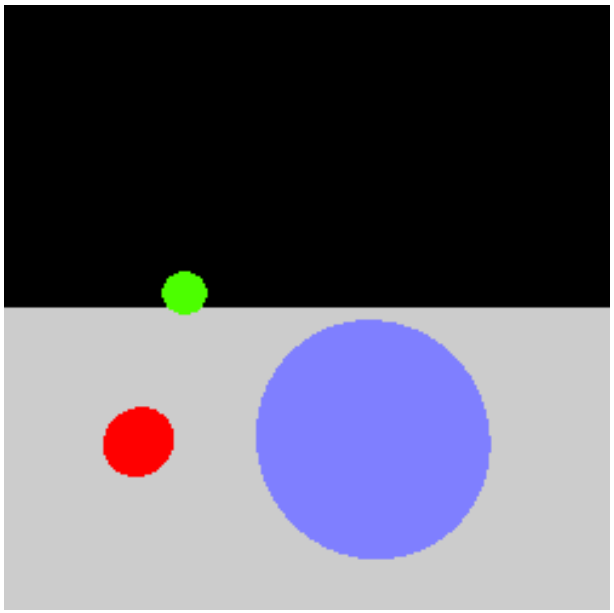

FIGURE 2 – Figure résultant de l'intersection des rayons par les 3 sphères et le plan. Notons que les couleurs des objets sont directement affectées aux pixels, et que l'ordre d'intersection par rapport à la caméra n'est pas pris en compte.

4 Lancé de rayons

4.1 Stockage des primitives

Question 11 : *O*bservez la classe de stockage des primitives d'une scène dans la classe `scene\_parameter`. Pourquoi les primitives

sont stockées en tant que pointeurs et non en tant que copie d'objet comme les matériaux et les lumières ?

4.2 Recherche de l'intersection la plus proche

Dans la fonction `ray_trace()` du fichier `render_engine`, notez l'utilisation de la fonction `compute_intersection()`. Le rôle de cette fonction est de trouver, si elle existe, l'intersection la plus proche avec une primitive de la scène le long de la demi-droite correspondant au rayon courant.

Question 12 : *M*odifiez la fonction `compute_intersection()` de manière à satisfaire cette contrainte.

À présent, votre résultat devrait être cohérent avec celui présenté en figure 3.

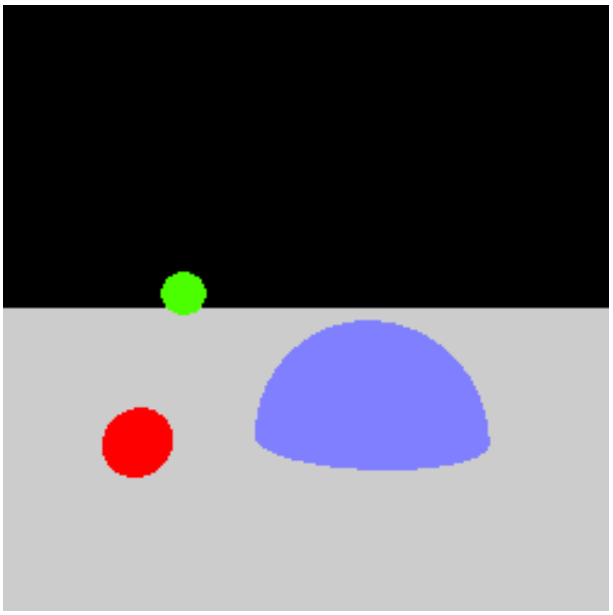

FIGURE 3 – Figure résultant de l'intersection des rayons par les 3 sphères et le plan. Les couleurs sont toujours directement affectées aux pixels, mais l'ordre des intersections est cette fois correctement prise en compte. Notons que le plan coupe bien la sphère bleue en deux, et que seule la partie du plan située à l'avant de la caméra est affichée.

4.3 Ombrage et illumination

Question 13 : *O*bservez l'enchaînement des appels de fonctions à partir de `ray_trace()`, `compute_illumination()`, et `is_in_shadow()`. Retrouvez la structure algorithmique suivante :

```
1  Recherche de l'intersection la plus proche
2  Si une intersection est trouvee,
3    Alors calcul de la couleur a appliquer sur le pixel
4  Sinon
5    Appliquer couleur noire
```

Ainsi que la fonction de calcul de la couleur :

```
1    Pour chaque lumiere L de la scene
2       Si le point courant est dans l'ombre de L
3          Appliquer la couleur d'ombre.
4       Sinon
5          Caluler l'illumination au point courant.
```

Question 14 : Complétez la fonction de calcul d'ombrage `is_in_shadow()` afin de déterminer si le point courant est dans l'ombre d'un autre objet par rapport à la lumière donnée.

Qu'arrive-t-il à votre image si votre rayon de calcul d'ombrage part exactement du même point que l'endroit de l'intersection ? Expliquez pourquoi et ce qu'il faut faire pour remédier au problème. Notez que vous disposez de la méthode `offset()` permettant de décaler légèrement le point d'origine d'un rayon dans la direction de celui-ci.

Une fois les ombres mises en place, votre image devrait être cohérente avec la figure 4.

Question 15 : Complétez la fonction d'application de la couleur `compute_illumination()` ainsi que la fonction de calcul d'illumination (on considérera une illumination classique de type phong) `compute_shading()` dans le fichier `shading_parameter.cpp`. Observez que cette fois, les objets ne sont plus de couleurs uniformes comme illustré en figure 5.

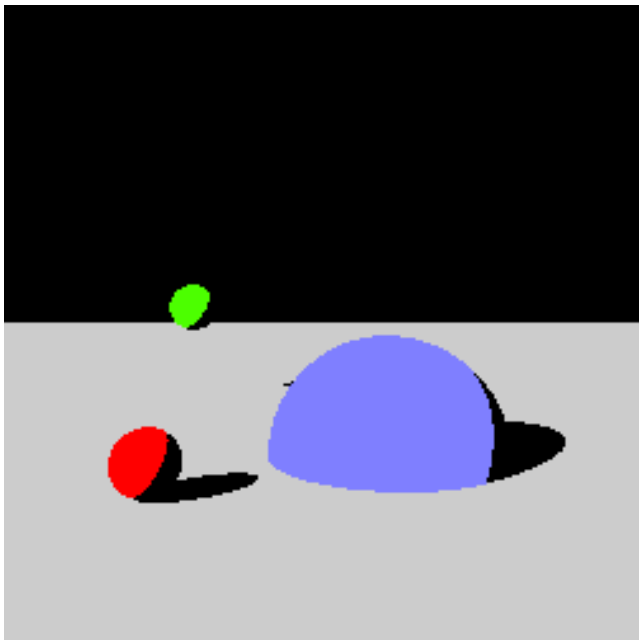

FIGURE 4 – Résultat obtenu après prise en compte des ombres (couleur mise à (0,0,0) lorsqu'une ombre est détectée.

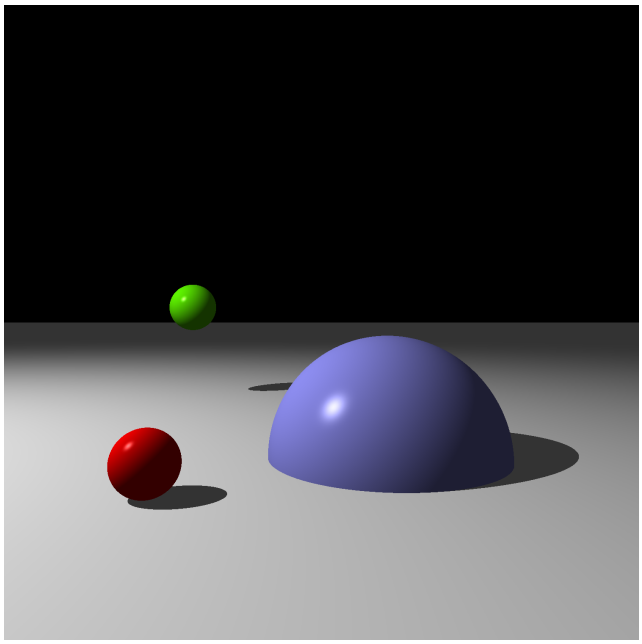

FIGURE 5 – Résultat obtenu après application d'une illumination de Phong sur les objets de la scène.

4.4 Réflexion

Considérons un rayon incident de direction unitaire u intersectant un objet dont la normale unitaire au point d'intersection est n. Il est possible de lancer un second rayon dans la direction u' symétrique de u par rapport à n. L'expression de u' en fonction de u et n est donnée par

$$u' = u - 2 < u, n > \ n \ ,$$

notez également que la fonction `reflected()` dans la classe `ray` permet de réaliser ce calcul.

Le lancement de rayon secondaire peut s'itérer un nombre arbitraire de fois permettant ainsi de modéliser des réflexions en chaîne. La couleur finale du pixel est alors obtenue par la somme pondérée des contributions de chaque rayon réfléchi. Pour chaque rayon réfléchi, on pourra considérer un facteur d'atténuation faisant diminuer l'intensité de la contribution.

Question 16 : Implémentez le mécanisme de réflexion dans la fonction `ray_trace()`. Le facteur d'atténuation pour chaque réflexion sera donné par la composante `reflection` de la classe `material`. Implémentez le cas de N réflexions et vérifiez votre résultat. Notez que le cas d'exemple est illustré en figure 6.

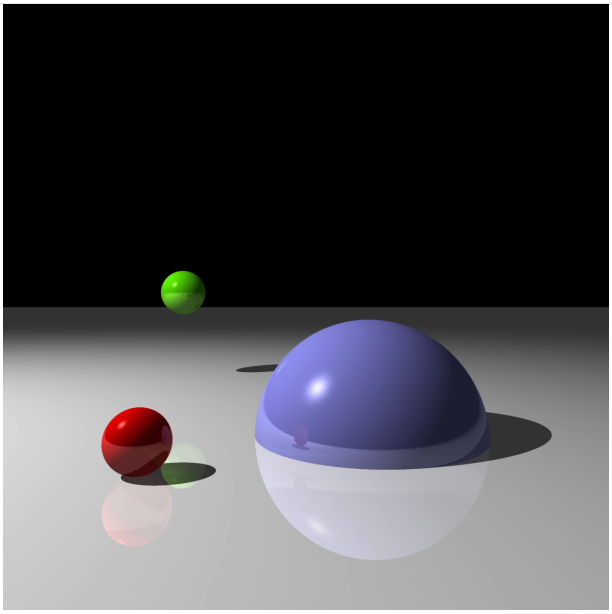

FIGURE 6 – Mise en place des réflexions sur les sphères et le plan. Ici 5 niveaux de réflexions sont attribués. Chaque couleur est atténuée par un facteur de 0.2 pour chaque niveau de réflexion.

5 Anti aliasing

Jusqu'à présent, un seul rayon est lancé pour chaque pixel ce qui aboutit à un crénelage parfois visible sur les contours des objets. Pour limiter cet effet de créneau, il est possible de lancer plusieurs rayons par pixel venant échantillonner différentes directions autour de la direction principale.

La couleur du pixel est alors donnée par la moyenne pondérée des couleurs de chaque échantillon. En contrepartie, cette approche rend le calcul du lancé de rayon plus coûteux.

Question 17 : *I*mplémentez le mécanisme d'anti-aliasing dans la fonction `render()` en sur-échantillonnage plusieurs rayons pour chaque pixel. Notez qu'il est possible de s'aider de la classe `anti_aliasing_table` pour la mise en place d'un noyau de pondération de type Gaussien comme illustré en figure 7.

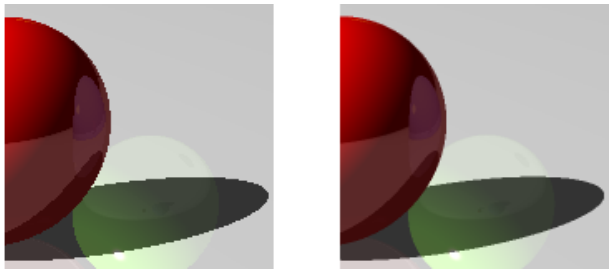

FIGURE 7 – Comparaison avant/après mise en place d'un sur-échantillonnage permettant l'anti-aliasing. La figure de gauche représente un zoom sur la figure avant le sur-échantillonnage, alors que la figure de droite montre le résultat après sa mise en place. Notez les transitions plus douces.

Notons que dans les moteurs actuels de ray-tracing, il est possible de mettre en place un échantillonnage adaptatif se réalisant uniquement lorsque cela est nécessaire, c'est à dire lorsque les couleurs varient localement.

6 Travaux supplémentaires

D'autres améliorations peuvent être mises en place dans votre code. Voici quelques possibilités :

6.1 Généralisation à d'autres primitives

Le ray tracing peut se généraliser à d'autres types de primitives. Il faut pour cela déterminer l'intersection d'une droite avec cette primitive.

Question 18 : *G*énéralisez le calcul du ray-tracing pour d'autres primitives. En particulier, le cas du triangle qui permettra de réaliser le rendu d'une surface maillée quelconque.
On pourra également s'intéresser au cas d'une primitive cylindrique.

6.2 Parallélisation

La méthode de ray-tracing possède l'avantage de pouvoir se paralléliser trivialement. Il est possible de prendre avantage des multi-processeurs afin d'appeler des threads permettant de calculer la couleur de différents pixels en parallèle.

Question 19 : *I*mplémentez un tel calcul en parallèle et comparez le temps de rendu pour un calcul séquentiel total et un calcul en parallèle. Assurez-vous que les parties en parallèle soient bien indépendantes.

6.3 Ombrage doux

L'utilisation de lumières ponctuelles viennent générer des ombres aux transitions franches que l'on ne retrouve pas dans une scène réelle. Une manière de générer des transitions d'ombres plus douces consiste à considérer des lumières non ponctuelles (que l'on peut par exemple modéliser en échantillonnant plusieurs lumières ponctuelles l'un à côté de l'autre).

Question 20 : *I*mplémentez la mise en place d'ombrages doux. Commentez l'évolution du temps de calcul en fonction de l'amélioration de la qualité de l'image.

6.4 Réfraction

Nous avons pu mettre en place l'utilisation de rayons réfléchis par la surface. Il est également possible de considérer les réfractions. Pour cela, à chaque intersection, un rayon peut être tracé suivant les lois de Snell-Descartes tel que

$$n_1 \sin(\theta_1) = n_2 \sin(\theta_2) \, ,$$

avec n_1 et n_2 les indices optiques des milieux incidents et réfléchis, et θ_1 et θ_2 les angles incidents et réfléchis des rayons par rapport à la normale $\mathbf{n}$ à la surface.

De même que dans le cas de la réflexion, l'intensité finale est obtenue par moyenne des couleurs issues de l'ensemble des rayons.

Question 21 : *I*mplémentez la mise en place de rayons réfractés dans votre code. Chaque objet volumique possédant alors une propriété d'indice optique. Notez que cette amélioration peut demander des modifications majeures dans le code.

6.5 Implémentation GPU

La méthode de lancé de rayons sur des objets tels que des sphères et plans se prête particulièrement bien au calcul sur GPU ce qui permet d'obtenir un rendu avancé en temps réel. L'ensemble du code peut être entièrement codé dans le fragment shader qui revient alors à lancer un rayon par pixel. Notez que cette approche est largement utilisée pour les scène 3D vues dans les exemples de Shader Toy.

6.6 Extensions aux méthodes dites physiques

Les moteurs de lancé de rayons modernes réalisent des images sur des approches dites physiques (physically based rendering). Des moteurs de rendus tels que Blender Cycles, LuxRender, YafaRay, Maxwell Render, etc, sont capables de réaliser ce type de rendus.

Question 22 : *E*xpliquez et analysez les différences entre les méthodes dites physiques et votre implémentation actuelle. Quels sont les avantages et inconvénients de chaque méthode. Mettez en place une partie de votre choix d'une méthode dite physique.