TP MSO Synthèse d'images: Lancé de rayons
CPE

durée - 4h

2013

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

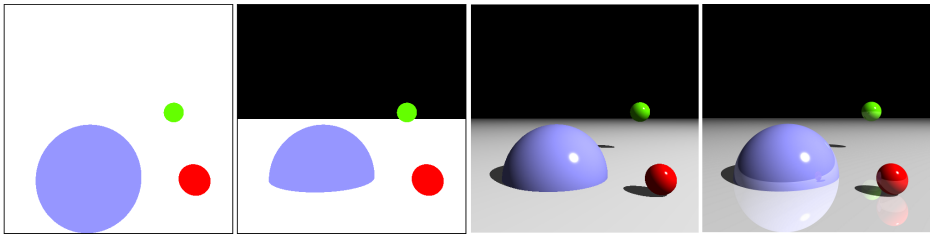

FIGURE 1 – Étapes de l'algorithme de lancé de rayons. De gauche à droite : équation du calcul d'intersection ; image des intersections ; ordonnancement des intersections suivant leur profondeur ; calcul d'illumination et d'ombrage ; réflexions.

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 associé à chaque intersection.
- Enfin, nous pourrons mettre en place différents effets réalisables aisément par lancé de rayons tels que la réflexion, l'anti-aliasing, ...

2 Prise en main de l'environnement

2.1 Les différents répertoires

Les répertoires contiennent différentes librairies qui vous sont fournies soit complètes, soit à compléter au fur et à mesure du TP.

- `libimage/` contient les classes et fonctions de bases permettant de manipuler une image 2D.
- `lib3d/` est le répertoire contenant une librairies entièrement codé pour manipuler des points en 2D, 3D, et 4D, ainsi que des matrices 2x2, 3x3 et 4x4.

- `libcommon/` contient la définition des exceptions de bases qui sont émises par les différentes fonctions et classes en cas de problèmes.
- `libmesh/` contient les classes de manipulation d'import et d'export de maillages complets.
- `libobject3d` contient la définition des objets 3D que l'on peut afficher par la méthode du Ray-Tracing. On complétera les classes d'objets de type plans et sphère dans ce TP.
- `libraytracing` contient les différents algorithmes et structures du *lancer de rayons*. On y trouvera la classe de rayon `ray`, les structures d'intersections et les fonctions de rendu.
- `libscene` contient les différents paramètres d'une scène 3D. Lumière, couleurs d'objets, camera, etc. La classe `scene` rassemble ces objets pour être utilisé pour être compatible avec nos fonctions de *lancer de rayons*.
- `local/` contient la fonction `main()` et ce qui sera nécessaire aux appels globales pour une scène spécifique.

2.2 Programme main

La fonction *main* réalise l'appel et l'affichage d'une scène 3D.

Les appels sont, dans l'ordre d'exécution :
- Création d'un buffer d'image.
- Initialisation et remplissage d'une scène par des objets 3D (spheres+plan) colorés (couleur+type d'illumination).
- Appel à l'algorithme de lancé de rayons dans la scène 3D sur l'image.
- Écriture de l'image dans un format ascii classique *ppm* (non compressé). L'image pouvant être manipulée par d'autres outils annexes : gimp, ...

Les différents appels sont présentés en fig. 2. Une fois l'ensemble des fonctions complétée, l'image de sortie doit représenter une vue de 3 sphères et d'un plan similaire à la fig. 8.

2.3 Classes utilisables

Un ensemble de classes de bases vous est fourni pour faciliter la mise en place de ce TP. L'ensemble des classes reste cependant bas-niveau et elles ne font pas appels à de librairies externes. Elles restent donc modifiables pour votre TP.

2.3.1 Classe pos2

Une classe de conteneur de base de 2 entiers. Elle sert principalement à indexer un pixel de coordonnées (kx,ky) dans une matrice.

```
pos2 u(5,4);        //u=(5,4)
u.x()=8;            //u=(8,4)
u=2*u-pos2(1,0)     //u=(15,8)
```

2.3.2 Classe vec3

Classe de conteneur de point 3D quelconque. En interne (x,y,z) étant stockés sur 3 flottants. Contiens de nombreuses fonctions vectorielles.

```
vec3 p(1.5,1.0,-2);              //p=(1.5,1.0,-2.0)
vec3 o=p.dot(vec3(1,1,0))*p;     //o=<p,(1,1,0)> p
o+=p;                            //o=o+p
```

```
//creation d'une image vide
image im(250);

//parametres de la scene 3D
scene scene_parameter;
//placement de la camera dans la scene 3D
scene_parameter.set_camera(camera(vec3(0,0,-2),vec3(0,0,1),vec3(0,1,0),2.0,1.2));

//parametres d'illumination par defaut
shading shading_parameter;

//ajout d'objets et de leur couleurs dans la scene 3D
scene_parameter.add(new sphere(vec3(-0.5,1,2),0.9),
                    material(color(0.5,0.5,1.0),shading_parameter,0.5));

scene_parameter.add(new sphere(vec3(1,1-0.2,1),0.2),
                    material(color(1,0,0),shading_parameter,0.9));

scene_parameter.add(new sphere(vec3(+1.7,-0.2,5),0.3),
                    material(color(0.3,1,0),shading_parameter,0.9));

scene_parameter.add(new plane(vec3(0,+1,0),vec3(0,-1,0)),
                    material(color(0.8,0.8,0.8),shading_parameter,0.1));


//ajout d'une lumiere dans la scene 3D
scene_parameter.add(light(vec3(15,-10,-10)));

//Rendu de la scene par ray-tracing dans l'image
render(im,scene_parameter);

//liberation memoire des objets de la scene alloues dynamiquement
scene_parameter.clean_memory();

//ecriture de l'image au format ppm
im.export_file("my_pic.ppm");
```

FIGURE 2 – Fonctions appelées depuis la fonction *main()* originale.

```
vec3 e=o.normalized();          //e=o / ||o||
e.z()=4;                        //e=(e.x,e.y,4)
```

2.3.3 Classe color

Classe de conteneur d'une couleur (r,g,b) où chaque canal est encodé sur un flottant $\in [0,1]$.

```
color c(1.0,1.0,0.0) // c <- jaune
c.b()=1.0;           // c <- blanc
color bleu=(0,0,1.0);

//magenta = (1-0.4)*rouge + 0.4*bleu
color magenta=(1-0.4)*color(1,0,0)+0.4*color(0,0,1);
```

2.3.4 Classe image

Classe de gestion d'une image (r,g,b). L'image est stockée en interne sous forme de vecteur concaténé de `color`. La classe implémente l'initialisation, l'accès protégé aux données de couleurs et l'export d'une image dans un fichier *ppm*.

```
image im(500); // créé une image 500x500
im.fill(color(1,0,0)); // colore l'image en rouge

// colore pixel(10,15) en vert
im(pos2(10,15))=color(0,1,0);

//exporte l'image dans fichier mon_pimage.ppm
im.export_file(''mon_image.ppm'');
```

2.3.5 Classe ray

Classe définissant un rayon (droite infinie orientée). La droite est définie par une position origine x_0 et un vecteur directeur unitaire $\mathbf{u}$.

```
//création d'un rayon orienté suivant l'axe x
ray r(vec3(0,1,0),vec3(1,0,0));

//evalue x0+5.5*u
v3 y=r(5.5);//y=(5.5,1,0)
```

2.3.6 Classe object3d

Classe virtuelle pure d'un objet 3D générique. L'objet étant défini par l'intersection entre une droite infinie (définit par la classe *ray*) et l'objet lui même. Tout objet d'une scène 3D doit dériver de cette classe générique.

La fonction d'intersection retourne *true* ou *false* en fonction de l'existance d'une interesection valide. Les données de l'intersection (position, normale, coordonnée relative) sont mis à jour via le paramètre `intersection`).

2.3.7 Classe plane

Classe implémentant un *object3d*. Un plan est défini par une position x_0 et une normale $\mathbf{n}$. Le calcul de l'intersection est à compléter.

2.3.8 Classe sphere

Classe implémentant un *object3d*. Une sphère est définie par un centre x_0 et un rayon r. Le calcul de l'intersection est à compléter.

2.3.9 Classe material

Un *matériaux* est un ensemble de propriété caractéristique de l'apparence visuelle d'un objet. Ce sont des propriétées liées à la structure interne de l'objet. On peut y retrouver entre autres

– La couleur intrinsèque de l'objet.

– Les paramètres d'illuminations (coefficient ambiant, diffus, spéculaires).
– L'amplitude des reflections.
– L'indice optique et le coefficient de réfraction.

2.3.10 Classe scene

Conteneur d'un ensemble d'objets 3D. La classe stocke un vecteur d'objets 3D (sous forme de pointeurs afin de profiter du polymorphisme), ainsi que leur *matériaux* (couleur+illumination) associés (il doit y avoir autant d'objets que de *matériaux*). Les lumières sont stockées dans un autre vecteur.

Si tous les objets 3D ont été alloués dynamiquement, la libération mémoire peut se réaliser directement par la classe suite à l'appel explicite à *clean memory*.

```
//cree scène vide
scene ma_scene;

//ajoute une sphere dans la scene
scene.add(new sphere(...),material(...));

//ajoute une lumière dans la scène
scene.add(light(...));

//utilisation du polymorphisme pour le calcul d'intersection
//(l'intersection appelée est celui de la sphère)
const object3d *obj=scene.get_object(0);
intersection mon_intersection;
bool is_intersect=obj->intersect(ray(...),scene,mon_intersection);
if(is_intersect==true)
  std::cout<<mon_intersection<<std::endl;

//libère la mémoire allouée pour la sphère
scene.clean_memory();
```

2.3.11 Fichier ray tracer

La classe *ray tracer* est un ensemble de fonctions mettant en place l'algorithme du *lancer de rayons*. Les différentes fonctions sont à compléter dans ce TP.

2.4 Diagramme de classes

Un diagramme des classes principales est proposé en fig. 3. Vous pourrez vous y reporter au fur et à mesure de l'avancement du TP pour vous aider à comprendre la communication entre les différents modules et fonctions proposées.

Notez que d'autres classes sont présentes (en dépassant le cadre de ce TP), la librairie étant réutilisée pour des TP ultérieurs ainsi qu'en 5ETI. L'ensemble peut être utile pour vos projets scolaire ou professionnels futurs.

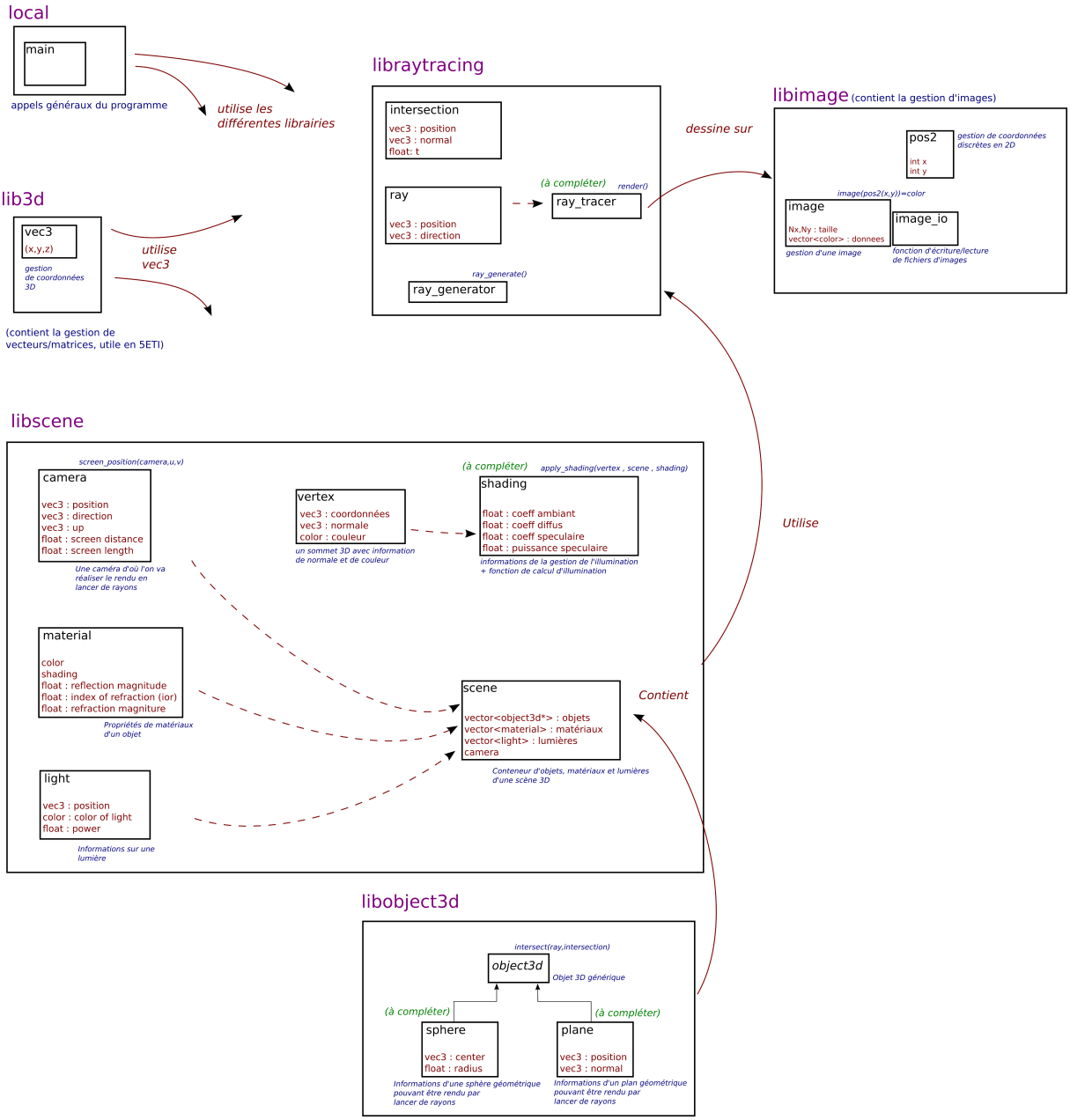

FIGURE 3 – Diagramme des classes et fonctions proposées.

3 Définition d'un objet 3D

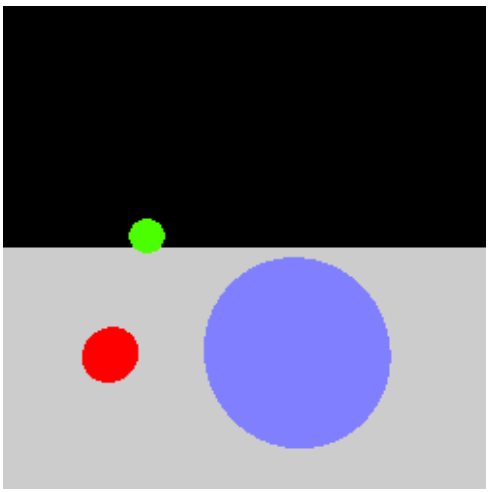

FIGURE 4 – Figure résultant de l'intersection des rayons par les 3 sphères et le plan. Notons que les couleurs des objets sont directement affectées aux pixels, et que l'ordre d'intersection par rapport à la caméra n'est pas pris en compte.

3.1 Méthode d'intersection

Pour la méthode de lancé de rayon, un objet 3D O est défini uniquement par ses intersections avec une droite D. Notez qu'aucune autre expression de l'objet n'est indispensable (ex. expression paramétrique, implicite, ...).

La classe `object3d` prévoit l'interface commune à tout objet que l'on peut afficher à l'aide du lancer de rayons. Chaque objet 3D de la scène doit être en mesure de retourner l'intersection entre lui même et une ligne droite. La méthode `intersect()` commune à tous les objets 3D réalise ce calcul. Elle possède la signatude suivante :

```
bool intersect(const ray& ray_data,
               intersection& intersection_data) const;
```

Lors de l'appel à la méthode `intersect()`, il existe deux possibilités.
- Soit il existe une intersection, et la méthode renvoit *true*.
- Soit l'objet n'intersecte pas la ligne droite et la méthode renvoit *false*.

La méthode prend en paramètre d'entrée un rayon incident. Le second paramètre (passé en tant que référence non constante) recoit les informations du premier point d'intersection valide si il celui-ci existe. Notez que ce paramètre n'a pas à être mis à jour si il n'existe pas d'intersection.

Soit $\mathbf{x}_s$ la position initiale du rayon, et $\mathbf{u}$ son vecteur directeur directeur unitaire. Soit $\mathbf{x}_{\text{inter}}$ la position d'une intersection valide avec le rayon. On peut donc écrire

$$\mathbf{x}_{\text{inter}} = \mathbf{x}_s + t_{\text{inter}}\,\mathbf{u}\,,$$

où t_{inter} représente la coordonnée locale de l'intersection dans le référentiel 1D du rayon.

Il peut exister plusieurs intersection entre le rayon est l'objet. L'intersection valide retournée par la fonction `intersect()` doit correspondre à la première rencontrée avec $t_{\text{inter}} > 0$.

La classe `intersection` contenu dans la librairie `libraytracing` contient les informations nécessaires pour caractériser localement l'intersection.

```
struct intersection
{
  vec3 x; //position de l'intersection
  vec3 n; //normale sortante en x
  double t; //coordonnee relative le long du rayon
};
```

3.2 Intersection avec un 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 donné par

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

Question 1 *Déterminez 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` de la librairie `libobject3d` contient les paramètres définissant le plan.

Question 2 *Complétez la méthode* `plane::intersect()` *qui recoit en paramètre le rayon incident actuel.*

À ce stade, le plan gris devrait apparaitre horizontale sur l'image de résultat.

3.3 Intersection avec une 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 donné 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 3 *Déterminer le paramètre t_{inter} solution du système d'équations, et donnez la position $\mathbf{x}_{inter}$, ainsi que la normale associée.*

Question 4 *En vous inspirant du calcul de l'intersection pour le plan, écrivez la méthode* sphere : :intersect.

À cette étape, on devra obtenir une image telle que celle montrée en figure 4.

4 Stockage d'une scène 3D

Question 5 *Observez la classe* `scene` *de la librairie* `libscene`. *Comment sont stockés les objets de la scène 3D ? Expliquez pourquoi le* `std::vector` *stocke des pointeurs et non pas des copies d'objets.*

Question 6 *Comment sont stockés les propriétés de couleurs de chaque objet ? Comment récupérer l'information de couleur du 3ème objet dans la fonction* `main()` *?*

5 Ray-tracing

On s'intéresse maintenant à l'algorithme du ray-tracing proprement dit.

Question 7 *Observez la fonction* `render()` *du fichier* `ray_tracer`*. Expliquez par un schéma ce que réalise la boucle de cet appel.*

5.1 Calcul du premier objet intersecté

La fonction `compute_first_intersection()` du fichier `ray_tracer` retourne la première intersection entre un rayon orienté et les différents objets de la scène 3D. La signature de cette fonction est la suivante :

```
bool compute_first_intersection(const ray& r,
                                const scene& scene_parameter,
                                intersection& inter,
                                int& index);
```

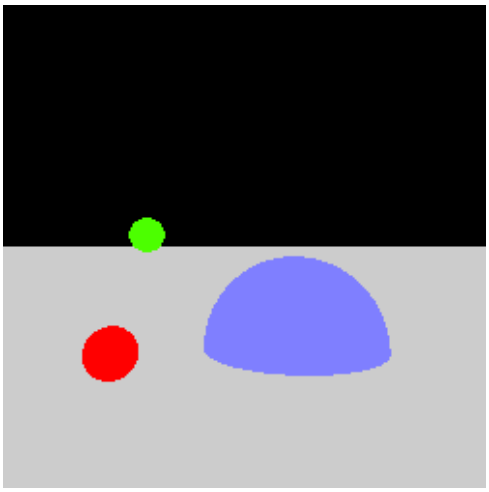

FIGURE 5 – Figure résultant de l'intersection des rayons par les 3 sphères et le plan. Les couleurs sont toujours directement affectées aux pixels, mais l'ordre des intersections est cette fois correctement prise en compte. Notons que le plan coupe bien la sphère bleue en deux, et que seule la partie du plan situé à l'avant de la caméra est affichée.

La fonction retourne *true* si une intersection est trouvée, et *false* si aucune intersection n'est trouvée. Les deux derniers paramètres *inter* et *index* sont passés en tant que références non constantes, et sont modifiés par la fonction.

- `inter` est mis à jour avec les données de la première intersection si celle-ci est trouvée.
- `index` est mis à jour avec l'indice de l'objet rencontré. L'indice faisant référence à l'indice de l'objet dans le `std::vector` stocké dans la classe `scene`.

Question 8 *Complétez la fonction* `compute_first_intersection()` *afin que cette fonction puisse retourner les informations de la première intersection trouvée parmi l'ensemble des objets de la scène.*

Question 9 *Complétez la fonction* `ray_trace()` *en affectant au pixel courant la couleur brute du premier objet rencontré.*

On pourra vérifier que le résultat obtenu est cohérent avec celui présenté en fig. 5.

5.2 Calcul de la couleur de l'objet

La couleur appliquée sur un pixel est calculée par la méthode *ray tracer : :find intersection color*. Sont comportement est le suivant :

- Si il existe au moins une intersection, calculer la couleur en fonction de l'objet et des lumières.
  i. Si une lumière est directement visible, on applique un calcul d'illumination de type Phong.
  ii. Si la lumière est cachée par un autre objet (lancé de rayon du point d'intersection vers la lumière), celui-ci est dans l'ombre, et seule la couleur ambiante de l'objet est attribuée.
- Si il n'existe aucune intersection avec le rayon courant, renvoyer la couleur de fond.

Jusqu'à présent lorsqu'une intersection existe, la couleur du pixel est directement issue de la couleur de base de l'objet. Pour obtenir une illumination de type Phong ainsi que les ombres, il est nécessaire de prendre en compte la normale de la surface au point d'intersection. Il est alors possible d'appliquer le calcul d'illumination vue précédemment ainsi que la position de la lumière.
Pour s'aider, on pourra utiliser la classe *shading* qui aura été complétée au TP précédent, ou que vous compléterez lors de ce TP.

5.2.1 Ombres

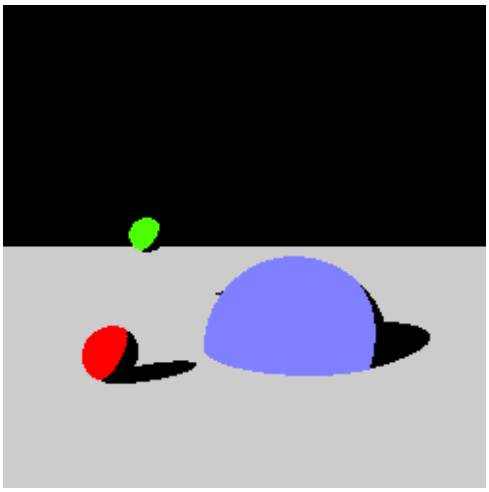

FIGURE 6 – Résultat obtenue après prise en compte des ombres (couleur mise à (0,0,0) lorsqu'une ombre est détectée).

La fonction `is_in_shadow()` recoit en paramètre une position courante et la position d'une source lumineuse considérée comme ponctuelle. La fonction retourne *true* si cette position est dans l'ombre d'un autre objet par rapport à la source lumineuse, et *false* dans le cas contraire.

Question 10 *En réutilisant la fonction* `compute_first_intersection()`*, complétez la fonction* `is_in_shadow()`*.*

Question 11 *Modifiez la fonction* `ray_trace()` *afin d'affecter une couleur noire au pixel lorsqu'une intersection trouvée est dans l'ombre d'un autre objet.*

À ce stade, vous devriez obtenir une figure similaire à la fig. 6.

5.2.2 Illumination

Étant donné les paramètres d'illuminations, la normale à la surface, une illumination de type Phong peut être calculée par la classe `shading`.

Question 12 *Complétez cette classe si cela n'a pas été fait au TP précédent.*

Question 13 *Modifiez l'implémentation de la fonction* `ray_trace` *de manière à retourner une couleur issue d'une illumination de type Phong. Le résultat obtenu devant s'approcher de l'illustration en fig. 7.*

Question 14 *Modifiez encore cette fonction pour que l'ombre ne soit pas de couleur noire, mais reprenne en partie la couleur de l'objet visualisé.*

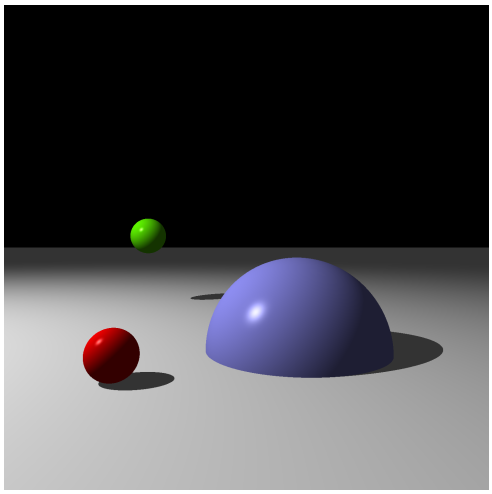

FIGURE 7 – Prise en compte d'une illumination de type Phong.

5.3 Rayons réfléchis

La méthode du ray tracing permet de prendre en compte aisément la réflexion des rayons sur des objets.

Considérons un rayon incident de direction unitaire $\mathbf{u}$ intersectant un objet dont la normale unitaire au point d'intersection est donnée par $\mathbf{n}$. Il est alors possible de lancer un second rayon réfléchi dans la direction $\mathbf{u}_2$ symétrique de $\mathbf{u}$ par rapport à $\mathbf{n}$. C'est à dire

$$\mathbf{u}_2 = \mathbf{u} - 2 < \mathbf{u}, \mathbf{n} > \mathbf{n} \, .$$

Ce nouveau rayon venant alors intersecter potentiellement un nouvel objet et apporter une couleur donnée.
L'algorithme de ray-tracing vu précédemment s'itère pour un nombre quelconque de réflexions. La couleur finale d'un pixel est alors donnée par la somme pondérée des couleurs de chaque rayon à chaque niveau de réflexion. Une amplitude de réflexion < 1 faisant diminuer l'*énergie* de chaque rayon. Pour cela, chaque classe `material` contient l'information d'amplitude de reflection pour un objet donné.

Question 15 *Modifiez la méthode* `ray_trace` *afin de considérer un nombre N fixé de réflexions.*

L'application des réflexions sur l'image standard pourra être comparée à la fig. 8.

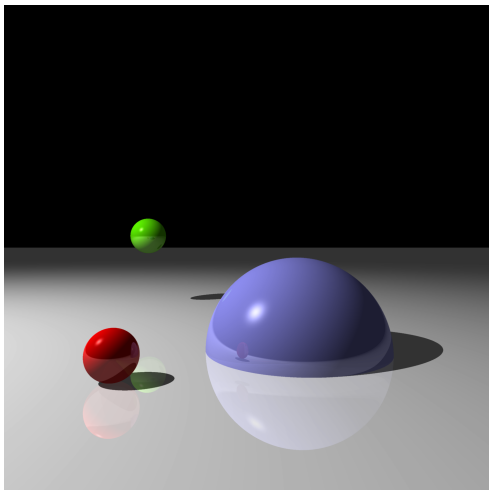

FIGURE 8 – Mise en place des reflexions sur les sphères et le plan. Ici 5 niveaux de réflexions sont attribués. Chaque couleur est atténuée par un facteur de 0.2 pour chaque niveau de réflexion.

6 Extensions possibles

6.1 Anti-aliasing

Chaque rayon lancé est ici associé à une unique évaluation de couleur par pixel. Cet échantillonnage simple pour chaque pixel peut présenter des inconvénients visibles dans les cas suivants :

- On visualise une surface ou une texture oscillant rapidement (typiquement une texture vue de loin). Le non-respect des critères d'échantillonnages introduit un effet de recouvrement gênant.
- On s'intéresse au bords des objets qui présentent des coupes pixelisées franches.

Pour améliorer l'échantillonnage, il est possible de lancer plusieurs rayons pour chaque pixel, et de moyenner la couleur résultante en fonction de ces échantillons. Un exemple de résultat obtenu est illustré en fig. 9.

Question 16 *Modifiez la fonction* `render` *afin de lancer plusieurs rayons pour chaque pixels, et moyenner la couleur résultante. On pourra utiliser la classe* `anti_aliasing_table` *afin d'obtenir une pondération Gaussienne des poids.*

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.

6.2 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 17 *Généralisez 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'interesser au cas d'une primitive cylindrique.

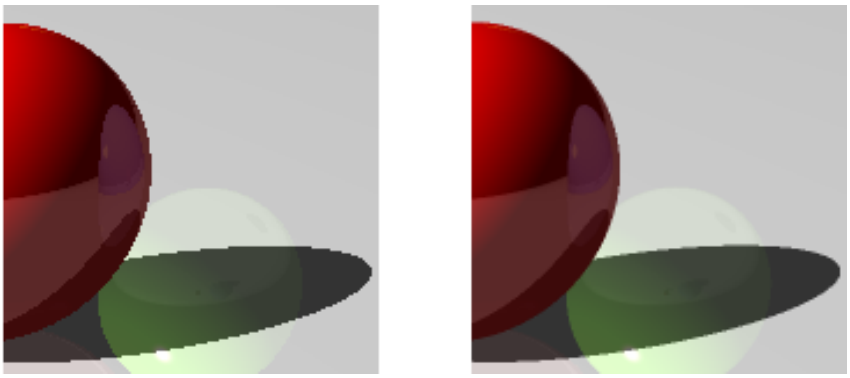

FIGURE 9 – Comparaison avant/après mise en place d'un sur-échantillannage permettant l'anti-alisasing. La figure de gauche représente un zoom sur la figure avant le sur-échantillannage, alors que la figure de droite montre le résultat après sa mise en place. Notez les transitions plus douces.

6.3 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 18 *Implémentez un tel calcul en parallèle et comparez le temps de rendu pour un calcul séquentiel total et un calcul en parallèle.*

6.4 Refraction

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 issue de l'ensemble des rayons.

Question 19 *Implémentez la mise en place de rayons réfractés dans votre code. Chaque objet volumique possédant alors une propriété d'indice optique.*