Project SI4 Semestre 2

Lancé de Rayon - Ray tracing

Valentin Michelet, Quentin Grimaud, Michael Gygli

Table des matières

1. Description du projet

1.1 Cahier des charges

Lorsque l'on implémente un ray tracer, il faut d'abord savoir comment vont être représentés, d'un point de vue mathématique, les objets à afficher. L'approche la plus courante consiste à représenter les objets sous forme de maillage de triangles et à trouver des intersections entre un rayon et un triangle. Ici, pour des raisons de précision, nous devions développer un raytracer capable de modéliser des objets définis par leur équation paramétrée, sans approximation. Le maillage est certes plus rapide, mais l'important ici n'était pas le temps d'exécution (par exemple, le temps de rendu d'une image d'un film du studio d'animation Pixar est d'environ 4h en utilisant une ferme de serveurs), il s'agissait surtout d'obtenir un rendu final de très bonne qualité.

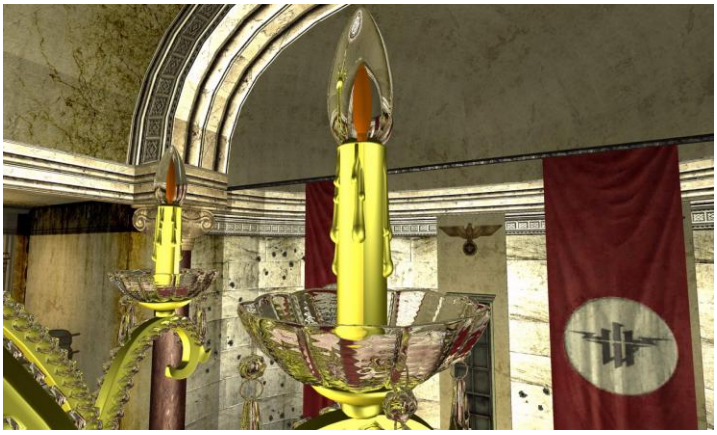

Image 1 Exemple d'image de jeu vidéo obtenu par ray tracing

Pour réaliser ce projet de synthèse d'image nécessitant des compétences à la fois en mathématiques et en informatique, nous étions répartis en deux équipes : une focalisée sur la partie mathématique de résolution d'équations permettant de calculer les intersections entre droite et surface paramétrée, l'autre chargée de réaliser un ray tracer capable de fournir des images à partir d'équation paramétrique d'objet, donc avec une architecture réalisée pour se servir de ces résultats d'intersections avec une surface paramétrée. Nous avons également tenu à ce que notre code soit le plus modulaire possible, afin de pouvoir changer facilement de modèle de lumière, ajouter d'autres fonctions, ou pourquoi pas de gérer également des objets définis par leur équation implicite. Il est également très facile de jouer sur le nombre d'objets, de source de lumières, ou de disposition dans la scène en les écrivant dans le 'main'.

Quant au langage de programmation, nous avons choisi de coder ce projet en C++, et ce pour plusieurs raisons. Tout d'abord, il s'agissait d'un langage que nous connaissions tous pour l'avoir étudié pendant notre parcours scolaire. Ensuite, nous avions à notre disposition un livre très complet sur le sujet qui fournissait des exemples de classes dans ce langage, très utile pour avoir des pistes et avoir nos premiers résultats rapidement. De plus, le C++ nous semblait être un langage rapide, donc approprié puisque le ray tracing est gourmand en calculs. En outre, la bibliothèque Lapack, celle utilisée par l'autre équipe afin d'effectuer les calculs de valeurs propres généralisées et donc d'obtenir les intersections, proposait une version utilisable en C++. En utilisant C++ plutôt que Java, autre candidat possible pour ce projet, nous étions sûrs de trouver un large éventail de bibliotèques

mathématiques. Enfin, l'un de nous connaissait assez bien la SDL, une bibliothèque graphique C, utile pour visualiser nos images de synthèse et permettre une interactivité avec l'utilisateur.

1.2 Principe général du lancer de rayons

La lancé de rayons est un sujet qui a été largement abordé et qui permet de générer des images de synthèse avec un rendu excellent. Dans la nature, les rayons de lumière frappent les objets et rebondissent pour atteindre notre œil. C'est comme cela que nous pouvons les voir, apprécier leur forme, leur texture ou leur brillance.

Si nous avions dû modéliser tous les rayons de lumière qui partent de différentes sources de lumière, nous nous serions vite rendu compte que beaucoup ne parviennent pas jusqu'à notre œil et leur calcul aurait par conséquent été inutile. C'est pourquoi, afin de modéliser les objets et les sources de lumière, nous avons utilisé le fonctionnement inverse : en partant de l'œil, nous avons tracé des rayons pour parcourir tous les pixels de l'écran. À chaque rebond sur un objet, nous calculons la contribution des sources lumineuses présentes dans la scène et nous récupérons les couleurs des objets qu'il a touchés pour obtenir la couleur finale du pixel concerné. S'il n'y a eu aucun rebond, on considère que le pixel est de la couleur du fond.

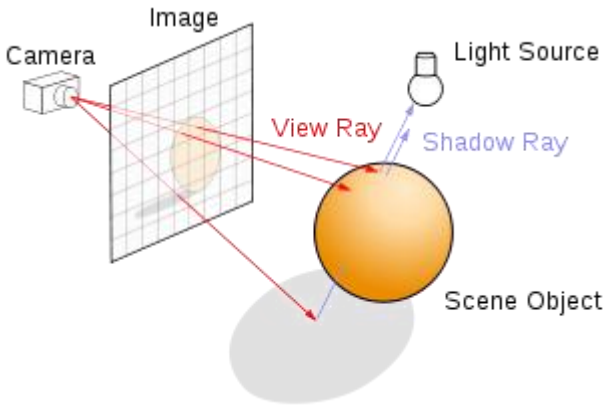

Image 2 Schéma de visualisation des rayons lumineux

Un autre intérêt de cette technique réside dans sa facilité d'introduire les phénomènes optiques classiques tels que la réflexion, la réfraction, la diffusion... Quand un rayon touche un objet, nous connaissons ses propriétés physiques, et il suffit d'appliquer des formules mathématiques pour obtenir rapidement des images réalistes. Le sujet étant étudié depuis plusieurs années, on dispose d'énormément de documentation traitant du développement des différents modèles physiques de la lumière, plus ou moins compliqués, qui ont déjà fait l'objet de recherches.

Image 3 Un exemple de réfraction avec trois milieu transparents différents

2 Le Ray casting

L'idée du ray casting et du ray tracing est assez simple : on calcule le déplacement des rayons de lumière. Pour des raisons d'efficacité, on ne suit pas le chemin de la lumière vers l'oeil, mais on fait plutôt du ray backtracing. C'est-à-dire qu'on ne lance pas un rayon depuis la source lumineuse, comme ça se passe en réalité, mais plutôt à partir du point de l'observateur. Ce lancé de rayon est fait vers chaque pixel d'un écran (image visuelle). Si le rayon intersecte un object, on affiche une couleur pour ce pixel.

Dans la version simple, le ray casting, on affiche simplement la couleur de l'objet de l'intersection. Dans le cas plus élaboré, le ray tracing, on fait rebondir le rayon et on calcule la contribution des sources lumineuses et d'autres objets. Nous décrivons ici le fonctionnement du ray casting, nous nous pencherons ensuite sur le ray tracing.

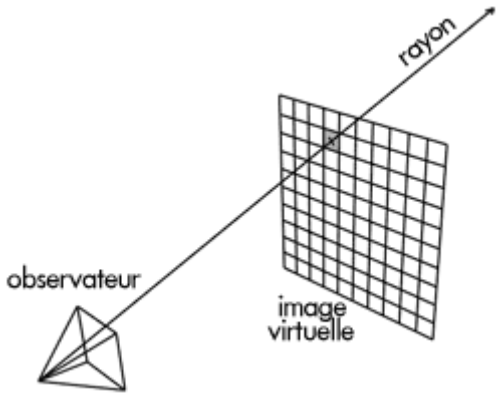

Image 4 arrangement d'observateur et d'écran

2.1 La théorie

Nous avons démarré le projet en écrivant un prototype de ray caster. Cette section explique en détail le fonctionnement d'un ray caster et donne les équations mathématiques nécessaires pour détecter l'intersection avec des sphères.

Comme expliqué au-dessus, le ray casting est très simple. On a besoin de :

- Un point d'observateur
- Un écran
- Un rayon
- Un objet
- Un algorithme pour calculer les intersections entre un rayon et un objet.

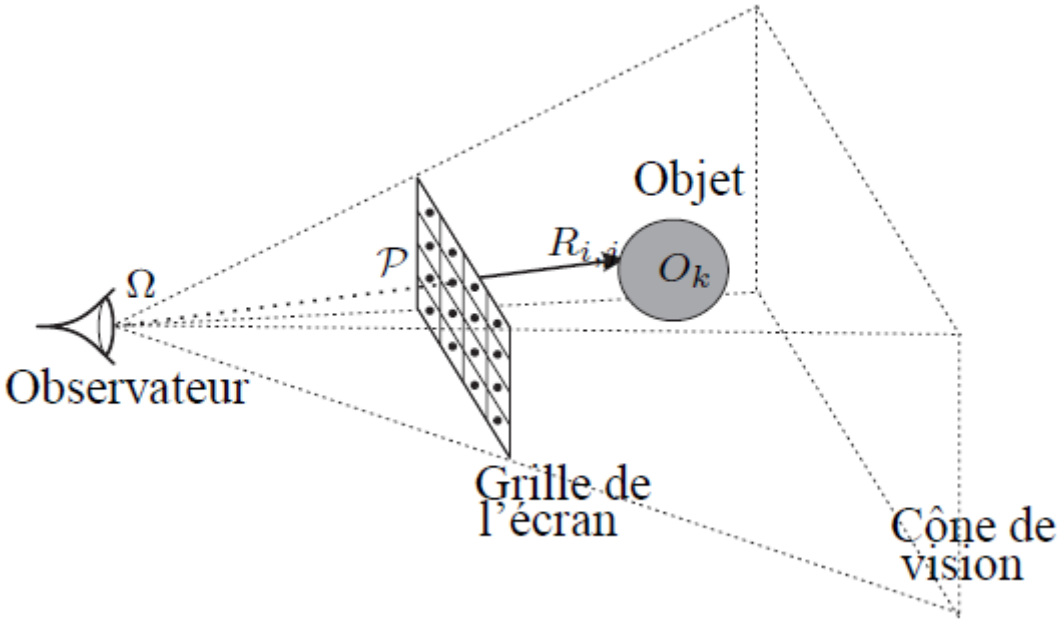

Image 5 Arrangement simple du ray casting

2.1.1 L'observateur

Le point d'observateur est tout simplement un point dans l'espace.

2.1.2 L'écran

L'écran est un rectangle qui est perpendiculaire au vecteur qui va de l'observateur au centre de l'écran.

2.1.3 Le rayon

Un rayon r est défini par un point de départ $\vec{o}$ et une direction $\vec{d}$, alors

$$r(t) = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \vec{o} + \vec{d} * t$$

2.1.4 L'objet

Comme objet n'importe quelle forme est possible. Tout ce dont le ray casting a besoin, c'est d'un algorithme qui peut calculer les intersections avec cet objet. Pour notre prototype nous n'avons implémenté que l'intersection avec des sphères, qui ont cette formule :

$$(x - x_m)^2 + (y - y_m)^2 + (z - z_m)^2 = R^2$$

2.1.5 Algorithme d'intersection

Pour intersecter un rayon avec une sphère, il suffit d'injecter les valeurs de x,y,z du rayon dans l'équation de la sphère. On a alors :

$$(o_x + d_x * t - x_m)^2 + (o_y + d_y * t - y_m)^2 + (o_z + d_z * t - z_m)^2 = R^2$$

Ce qui nous donne une équation du second degré avec une inconnue, qu'on sait résoudre comme suit :

$$a = |\vec{d}|$$
$$b = (d_x(o_x + m_x) + d_y(o_y + m_y) + d_z(o_z + m_z)$$
$$c = |m|^2 + |\vec{o}|^2 - 2(o_x m_x + o_y m_y + o_z m_z) - R^2$$

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

De là, il suffit de prendre le plus petit t positif (positif pour ne pas considérer les objets "derrière" l'observateur) pour calculer la première intersection.

2.2 En pratique

Nous allons maintenant définir la structure de notre ray caster. Tout d'abord, identifions ce dont nous avons besoin :

- un observateur (position de l'œil ou de la caméra)
- un point visé ou une direction
- une représentation dans l'espace 3D de l'écran 2D

- des objets à placer sur la scène

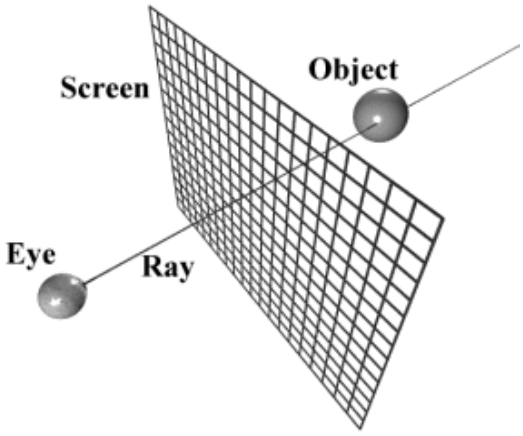

Image 6 Arrangement d'observateur et d'écran

On obtient donc le diagramme UML suivant :

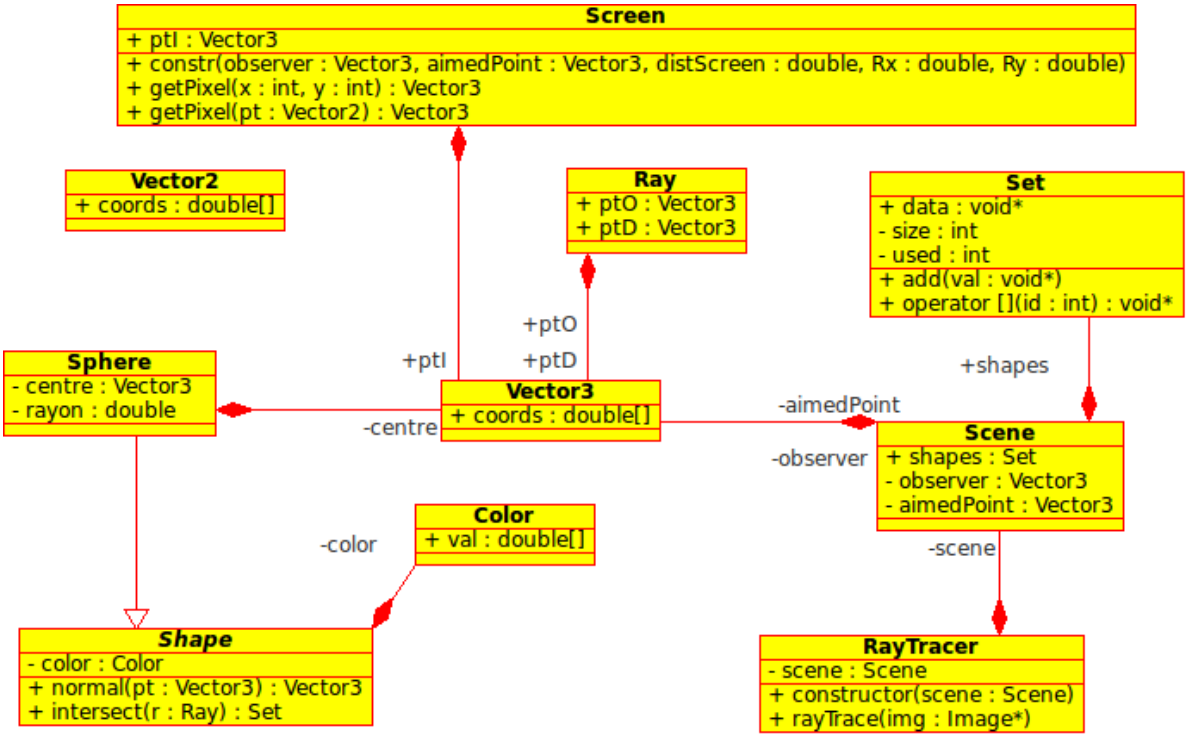

Image 7 diagramme UML pour le Ray Casting

On place donc l'écran entre l'observateur et le point visé (cf Positionnement de l'écran dans Modules intéressants pour la méthode utilisée pour cela). Pour chaque pixel, on lance un rayon depuis l'observateur vers ce pixel, et on appelle la fonction d'intersection de chaque objet de la scène avec ce rayon en paramètre. On garde l'intersection la plus proche de l'observateur, la couleur du pixel courant est alors assignée à la couleur de l'objet intersecté. S'il n'y a pas d'intersection, on assigne le pixel à la couleur de fond.

Le plus gros des calculs se trouve dans le calcul de l'intersection, c'est également l'un des points les plus difficiles algorithmiquement, selon le type de forme. Une forme très facile à implémenter est la

sphère (cf l'explication théorique plus haut), c'est donc la première forme que nous avons implémentée et affichée dans notre ray caster.

3. Ray tracing

3.1 Théorie

Le ray tracing est basé sur le ray casting. Mais tandis que le ray caster s'arrête après une intersection, le ray tracing continue en gérant les rebonds. Tout d'abord, on calcule la lumière ou l'ombre en ce point d'intersection (modèle de lumière et d'ombres). Pour ça on a besoin de la normale au point d'intersection et du rayon réfléchi. Le rayon réfléchi $\vec{r}$ est donné par (voir [3], page 65):

$$\vec{r} = -\left(2\vec{n}\left(\vec{n}.\vec{d}\right) - \vec{d}\right)$$

Avec la normale $\vec{n}$ et la direction du rayon qui intersecte $\vec{d}$.

Après avoir calculé la lumière en ce point, on relance le rayon réfléchi en ce point d'intersection et on calcule les prochaines intersections et leur lumière/couleur, qui contribuent à la couleur du point de la première intersection. Le degré de contribution est défini par le coefficient kreflex qui est dépend du matériel. Si kreflex est égal à 0, la surface n'a pas de réflexion. S'il est égal à 1, la surface a une réflexion parfaite. Dans le ray tracing, il est également assez facile de simuler la réfraction, c'est-à-dire des objets transparents ou semi-transparents. Dans notre ray tracer par contre nous n'avons pas implémenté ce phénomène.

Pour l'implémentation de la réflection, nous avons utilisé un algorithme récursif qui est appelé autant de fois que le nombre de rebonds passés en paramètre. Nous avons basé notre algorithme sur la théorie donnée en [3].

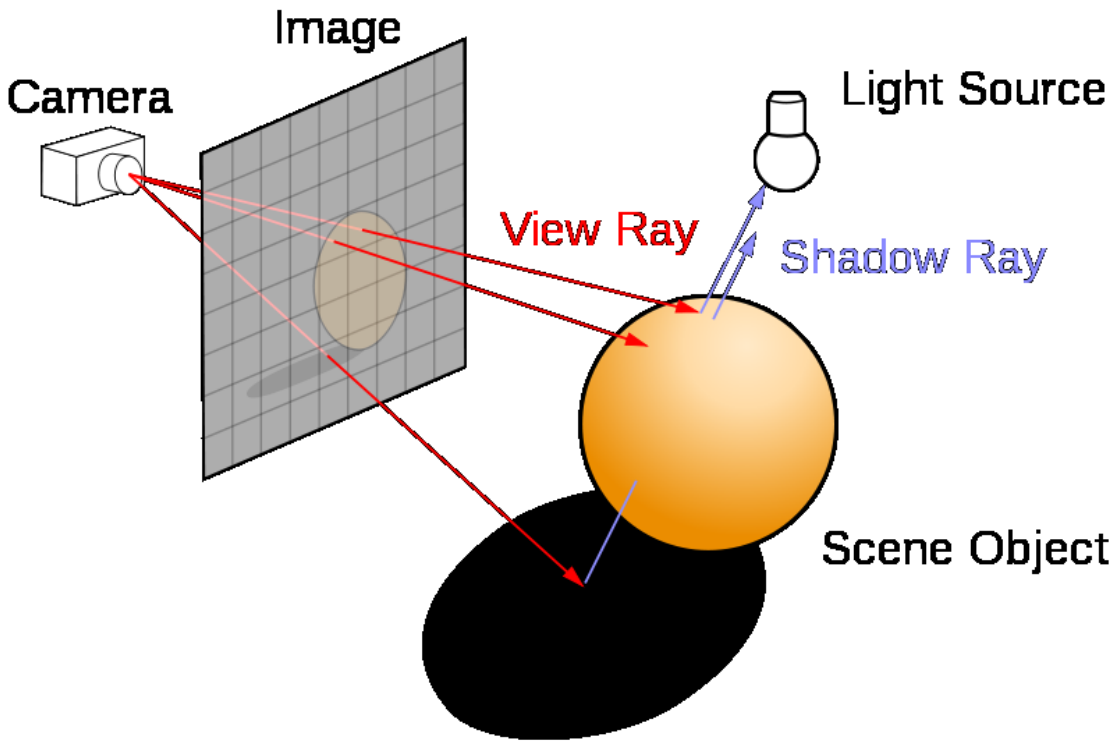

Image 8 Ray tracing modèle

3.2 Modèle de lumière

Comme modèle de lumière plusieurs choix sont possibles. Pour des raisons de simplicité, nous nous sommes limités au le Model de Phong, un modèle simple, qui permet d'avoir des résultats assez réalistes. Une liste non exhaustive de plusieurs modèles utilisés dans le ray tracing est donnée en [2].

Le modèle de Phong est composé de 3 lumières différentes : La lumière ambiante, la lumière diffuse et la lumière spéculaire. La lumière totale est la somme de ces trois :

$$\mathrm{I}_{\mathrm{totale}} = \mathrm{I}_{\mathrm{ambiente}} + \mathrm{I}_{\mathrm{diffuse}} + \mathrm{I}_{\mathrm{speculaire}}$$

Un exemple d'utilisation du modèle de Phong :

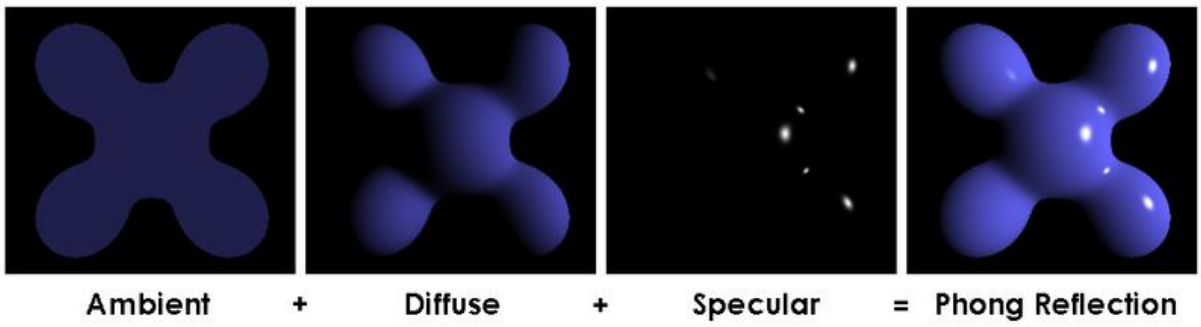

Image 9 Lumières différentes du Phong Model

Les lumières différentes se calculent comme ainsi :

3.2.1 La lumière ambiante

Cette lumière se calcule simplement avec la lumière ambiante dans une scène, multipliée par un coefficient $k_{ambiante}$ qui est défini par le matériau de l'objet.

$$I_{ambiante} = k_{ambiante} * \mathrm{I}_{scène}$$

3.2.2 La lumière diffuse

La lumière diffuse en un point p est la somme des lumières diffuses apportées par chaque source lumineuse qui contribue à la lumière en ce point. La lumière pour une seule source est dépendante de l'intensité de la source lumineuse, du coefficient $k_{diffuse}$ du matériau et de l'angle α_i qui est l'angle entre la normale au point p et le vecteur du point p à la lumière L_i.

$$I_{diffuse} = k_{diffuse} * \sum_{i=0}^{n} \mathrm{I}_i * \cos \alpha_i$$

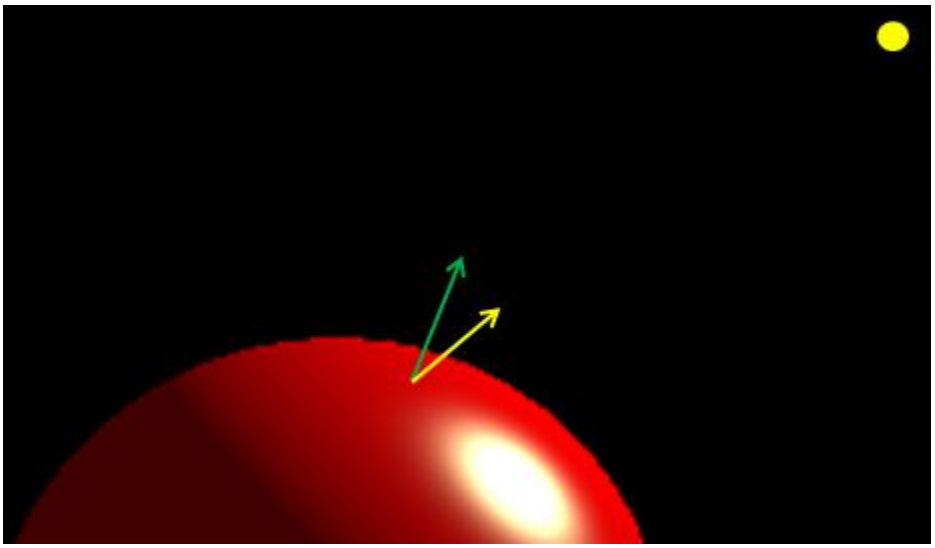

Image 10 En verte: La normale; En jaune: Le vecteur vers la source lumineuse

3.2.3 La lumière speculaire

La lumière spéculaire en un point p est la somme des lumières spéculaires apportées par chaque source lumineuse qui contribue à la lumière en ce point. Cette lumière est dépendante de l'angle β_i qui est l'angle entre le rayon réfléchi et le vecteur du point p à la lumière L_i. Le coefficient $\mathrm{k}_{\mathrm{speculaire}}$ définit l'intensité de la lumière spéculaire. Le coefficient ns définit le radius de réflexion spéculaire. Si $n_s \to \infty$ on a une réflexion parfaite. Si ns est proche de 1 on a une surface rugueuse, si $n_s > 20$ la surface est brillante (voir Image 12).

$$I_{speculaire} = k_{speculaire} * \sum_{i=0}^{n} \mathrm{I}_i * \cos\beta^{n_s}$$

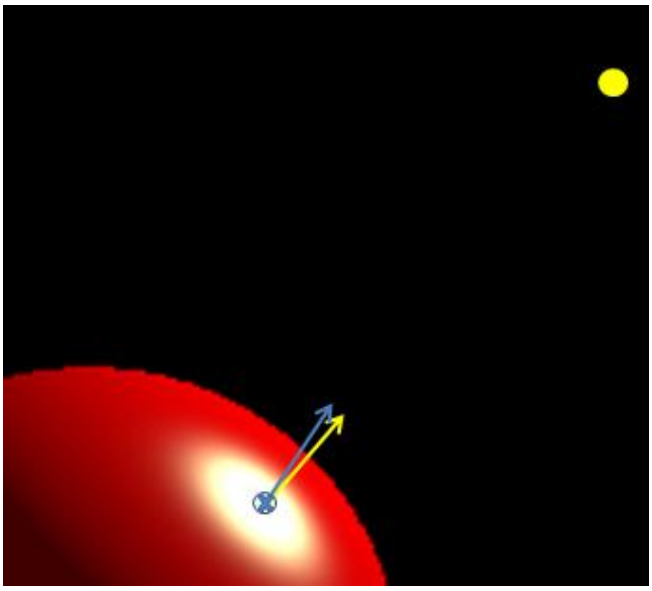

Image 11 En bleu: Le rayon réfléchi; En jaune: Le vecteur vers la source lumineuse

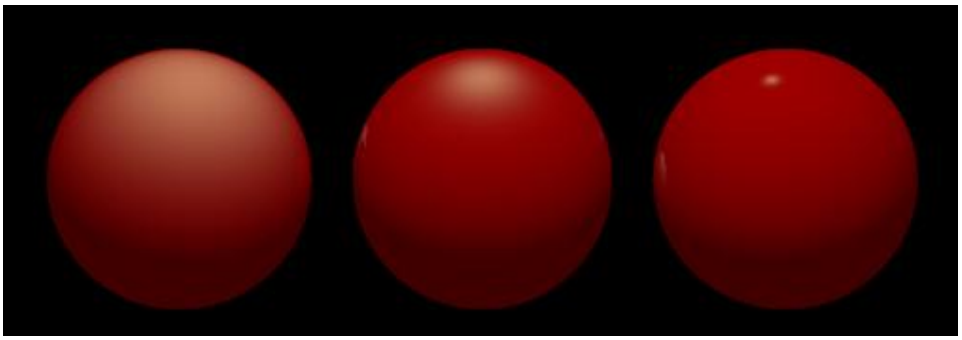

Image 12 à gauche: n_s=5, au millieu n_s=30, à droite: n_s=1000

3.3 Ombres

Un objet O est à l'ombre (par rapport à une source lumineuse donnée), s'il y a un objet entre la source lumineuse L_i et notre objet O. Donc pour déterminer si le point p est dans l'ombre, il suffit de calculer le rayon de p à la source lumineuse et de l'intersecter avec tous les objets dans la scène. S'il y a une intersection, la source lumineuse n'est pas visible du point p, alors cette source lumineuse ne contribue pas à l'éclairage en ce point (on dit donc que ce point est à l'ombre pour cette source lumineuse). Les luminosités diffuse et spéculaire valent donc 0 pour cette source lumineuse.

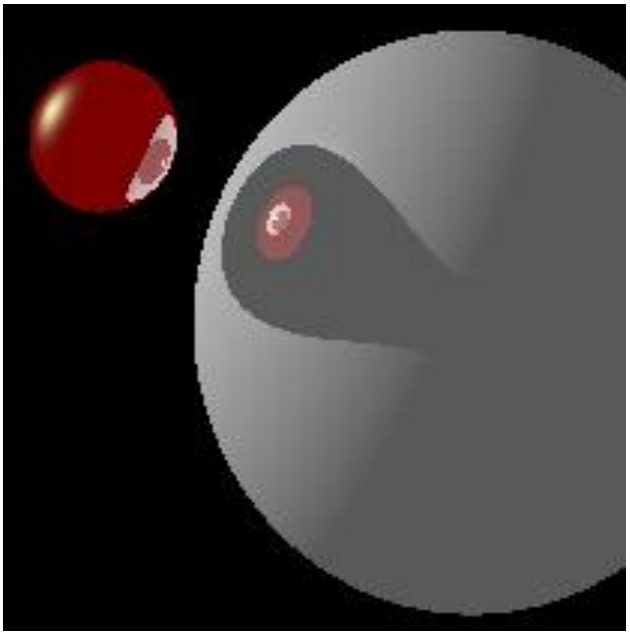

Image 13 Exemple d'un ombre

3.4 En pratique

Le ray tracing est une évolution du ray casting. Nous ajoutons les notions suivantes :

- les lumières
  - ambiante, de diffusion, spéculaire
  - les ombres
  - la réflexion
  - C'est ici que l'évaluation de la couleur en un pixel devient récursive : on calcule un nouveau rayon qui est le rayon réfléchi par rapport à la normale au point d'intersection, et on calcule à nouveau ses intersections
- les propriétés des matériaux
  - coefficients pour la lumière diffuse, ambiante et spéculaire
- les modèles de lumière
  - plusieurs modèles existent, nous avons principalement choisi le modèle de Phong, qui est celui utilisé en OpenGL

On obtient donc le diagramme UML suivant :

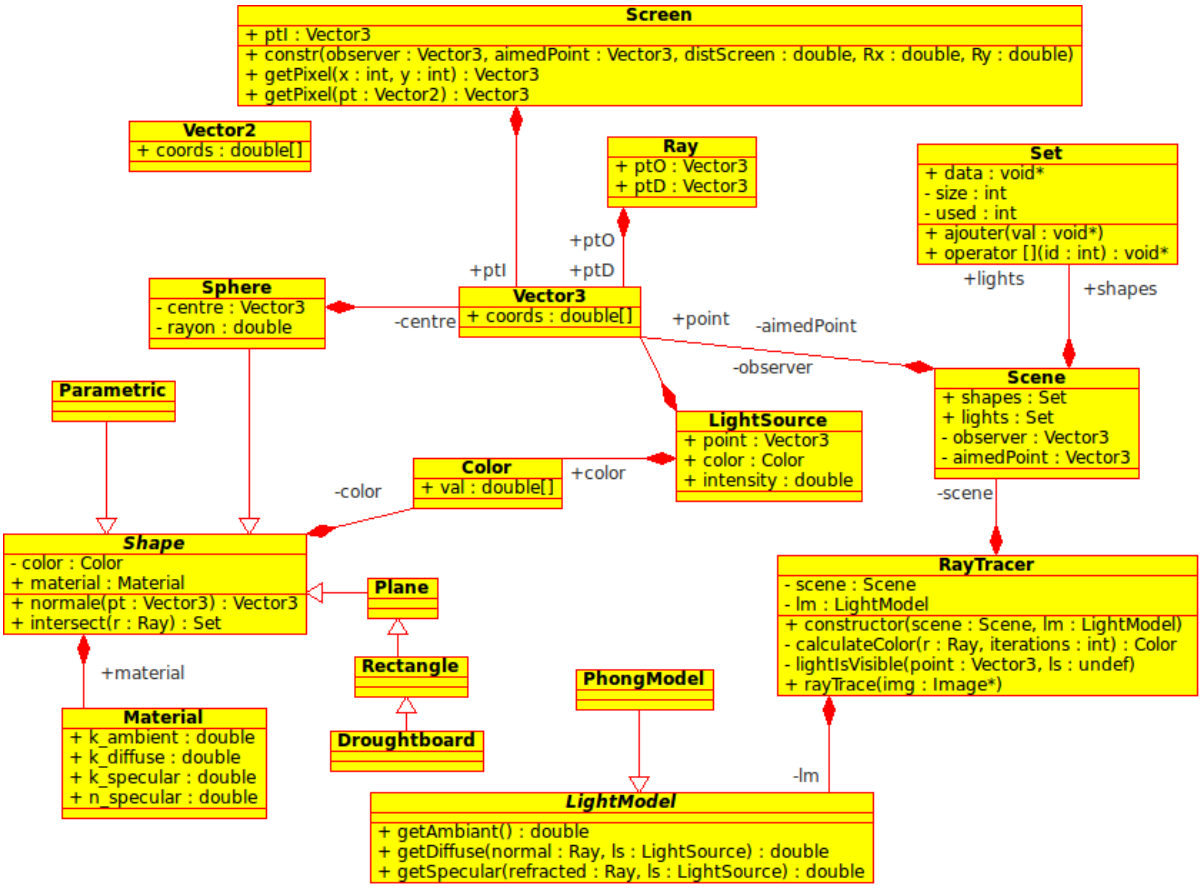

Image 14 UML pour le Ray tracing

3.5 Modules intéressants

3.5.1 Positionnement de l'écran

Pour pouvoir correctement déterminer la position de l'écran à partir des positions de l'observateur et du point visé, nous avons besoin de plusieurs paramètres.

Tout d'abord un vecteur « vers le haut » pour l'observateur, c'est-à-dire l'inclinaison de la tête sur le côté. Ce vecteur peut-être n'importe quel vecteur perpendiculaire au vecteur qui va de l'oeil au point visé. Pour simplifier dans la première version du programme, nous avions posé simplement que ce vecteur vers le haut est le vecteur unité sur l'axe de notre repère qui représente l'élévation. L'observateur ne pouvait donc regarder que dans la direction d'un point qui est à sa hauteur. Nous avions aussi simplifié le calcul des quatre points des extrémités de l'écran, en ne traitant que le cas où le point visé est dans l'un des axes du repère. Ceci dans le but d'obtenir rapidement un résultat graphique correct pour valider le reste de notre code. Dans une deuxième version de cet algorithme, nous avons évidemment supprimé ces limitations et traité le cas général qui décrit dans un instant.

Le centre de l'écran est clairement sur le segment allant de l'observateur au point visé. On se base sur une information d'angle de vue donnée par l'utilisateur, c'est-à-dire l'angle qui délimite le champ de vision.

Pour déterminer la taille de l'écran dans l'espace, plusieurs solutions s'offrent à nous, sachant que la taille de l'écran 2D est connue (taille de l'image) :

- l'utilisateur (ou une constante du programme) nous donne une distance souhaitée de l'observateur à l'écran définie par l'utilisateur pour arrêter ce point (il faut donc s'assurer qu'aucun objet de la scène ne soit à une distance de l'observateur plus petite que cette donnée). On peut alors calculer la largeur de l'écran 3D en utilisant les relations trigonométriques du rectangle rectangle formé par l'observateur, le centre de l'écran, et l'un de ses bords. On déduit la valeur de la hauteur via le rapport largeur/hauteur de l'écran 2D.
- l'utilisateur (ou une constante du programme) nous donne la largeur de l'écran 3D. On en déduit la valeur de la hauteur en utilisant le rapport largeur/hauteur de l'écran 2D. La valeur de la distance entre l'observateur et l'écran peut alors être calculée via les mêmes relations trigonométriques que précédemment.

Une fois que l'on connaît la taille de l'écran 3D, il nous faut placer les quatre points du rectangle formant l'écran. On part pour cela du point central dont on a calculé les coordonnées au début. La direction des ordonnées dans le repère de l'écran est la direction « vers le haut » de l'observateur. La direction des abscisses s'obtient en effectuant le produit vectoriel entre la direction de l'oeil au point visé et cette direction vers le haut (le produit vectoriel a la propriété de rendre un vecteur formant un repère direct avec les deux premiers). Pour obtenir la position de ces quatre pointrs, il nous suffit donc de normaliser ces deux vecteurs et de se déplacer à partir du point central de plus ou moins la moitié de la largeur dans la direction des abscisses, et de plus ou moins la moitié de la hauteur dans la direction des ordonnées.

Le pixel en (x,y) de l'écran 2D a donc comme coordonnées 3D :

$$p(x,y) = pointHautGauche + (\frac{x}{width2D}) * (pointHautDroite - pointHautGauche) + (\frac{y}{height2D}) * (pointBasGauche - pointHautGauche)$$

3.5.2 Création d'une animation

Pour mieux détecter les éventuels défauts de notre implémentation, nous avons pensé que plutôt que d'essayer de comparer les résultats avec de deux scènes légèrement différentes (en ayant déplacé un petit peu la caméra ou une lumière par exemple), ce serait une bonne idée de voir comment le rendu évoluait dans le temps lors d'une transition entre ces deux états.

Créer une animation consiste à faire dépendre d'une variable t une partie des paramètres de la scène, et de lancer le rendu de la scène dans une boucle qui augmente t, en exportant à chaque fois l'image résultat dans un fichier différent. Nous avons donc choisi une animation simple de déplacement de la caméra en cercle : il est facile de donner une paramétrisation d'un cercle de rayon donné et centré en le point visé (ici l'une des sphères de la scène). L'équation est la suivante :

$$posObserver_x = posSphere_x + \cos(t) * circleRadius$$

$$posObserver_z = posSphere_z + \sin(t) * circleRadius$$

Avec cette paramétrisation, une boucle sur t effectuant un rendu dans une nouvelle image à chaque fois donnera une séquence d'images qui montrera le déplacement en cercle de la caméra autour de la sphère (ici la caméra reste à élévation constante). Nous avons pu utiliser le même principe pour déplacer les lumières (ou tout objet). Il est à noter cependant, que pour le cas de déplacement de la caméra, s'il modifie l'inclinaison de la caméra il faut également répercuter cette modification d'inclinaison au vecteur « vers le haut » associé à la caméra, par exemple en utilisant une transformation de matrices.

Il nous a suffi ensuite d'écrire un script Bash (le fichier tools/createvideo.sh) qui convertit cette séquence d'images exportées en vidéo. Étant donné que l'animation en cercle revient au point de départ lorsque t atteint 2 Pi, il a été facile de sélectionner le bon nombre d'images ((2 Pi) / pas_de_t) pour obtenir une vidéo qui boucle parfaitement.

3.5.3 Autres points d'intérêt

- La classe RayTracer. C'est le coeur du programme. C'est elle qui utilise quasiment tous les autres modules. Elle contient l'algorithme principal qui calcule la couleur du pixel courant, à partir du modèle de lumière. Elle s'occupe également de déterminer si un point de l'espace se trouve à l'ombre ou pas pour une lumière donnée.
- L'utilisation de l'héritage pour le calcul des intersections avec n'importe quel type d'objet. La scène contient un ensemble d'objets. Le type des objets contenus dans cet ensemble est

Shape (ou donc, grâce à l'héritage, une classe fille de Shape). Cette classe abstraite force toutes ses classes filles à définir des méthodes intersect() et normal(). De cette manière, le ray tracer est certain de pouvoir évaluer la présence d'intersections avec n'importe lequel des objets de la scène. Il est donc facile de rajouter une nouvelle sorte d'objet au programme, par exemple pour faire le lien avec l'autre groupe de ce projet, le type ParamSurf (pour « surface paramétrée »). Il leur « suffit » alors d'implémenter ces deux méthodes pour que le ray tracer soit immédiatement opérationnel avec ce type de forme (mais l'écriture du contenu de ces méthodes pour gérer n'importe quelle surface paramétrée est en réalité très difficile en termes de mathématiques, c'était le but de leur projet).

- Le damier. Nous avons défini une classe Plane, qui calcule l'intersection entre un rayon et un plan. Nous en avons dérivé une classe Rectangle, qui est un plan délimité, et de cette classe Rectangle nous avons pu dériver une classe Draughtboard, dont la méthode getColor() en un point renvoie blanc ou gris selon sur quelle case on se trouve (ceci est déterminé à l'aide de de modulos sur les coordonnées du point considéré et sur la taille des cases).

4. Scénarios d'utilisation

Notre application est surtout axée sur le déplacement de la caméra pour visualiser les différents éléments de la scène. Nous n'avons pas écrit de module permettant de modifier l'organisation de la scène ou le nombre d'objets qu'elle contient via l'interface utilisateur (c'est toutefois facilement faisable). Pour modifier la position, le nombre ou la forme des objets et des sources de lumière, il faut éditer soi-même le code du programme principal.

4.1 Bibliothèques et outils utilisés

Nous avons utilisé plusieurs bibliothèques dans notre projet pour répondre à des besoins spécifiques. Pour compiler et lancer l'application, il faudra que la machine possède les bibliothèques suivantes :

- SDL et SDL_image pour l'affichage du programme sous forme d'une fenêtre (GUI) répondant aux interactions de l'utilisateur et gérant l'affichage d'une image. Sous ubuntu, installer les paquets *libsdl1.2debian libsdl1.2-dev libsdl-image1.2 libsdl-image1.2-dev* .

- It++ pour les calculs mathématiques (basés sur les matrices) liés aux surfaces paramétrées. Sous ubuntu, installer les paquets *libitpp6gf* et *libitpp-dev*

- (pour linux) En plus de l'installation basique d'It++, il faut également télécharger l'archive de la dernière version de cette bibliothèque sur http://itpp.sourceforge.net, et coller dans */usr/include/itpp/base* les classes *MatFunc.h* et *MatFunc.cpp*.

La chaîne de compilation du programme est gérée par l'outil Cmake (paquet *cmake* sous linux), qu'il faut également avoir installé pour pouvoir créer le *Makefile* à partir des *CmakeLists.txt* (compiler ensuite de la manière classique en tapant *make*).

4.2 Lancement de l'application

Tout le code source se trouve sur :

http://code.google.com/p/polytechraytracing

Récupérer les classes MatFunc.h et MatFunc.cpp de la librairie itpp sur :

http://itpp.sourceforge.net/devel/mathfunc_8cpp_source.html

Ouvrir un terminal, se placer dans le répertoire du projet et exécuter la commande suivante pour créer le Makefile à partir des fichiers de configuration de CMake :

cmake .

Taper la commande suivante pour compiler le projet :

make

Exécuter l'application en tapant :

bin/appGui ou ***./bin/appGui*** selon le cas.

4.3 Utilisation du programme

Une fois l'application lancée, on peut se déplacer en utilisant les boutons suivants :

- flèche de droite pour se déplacer horizontalement vers la droite
- flèche de gauche pour se déplacer horizontalement vers la gauche
- flèche du haut pour se déplacer verticalement vers le haut
- flèche du bas pour se déplacer verticalement vers le bas
- 'q' et 'd' pour se déplacer circulairement autour de la sphère bleue
- 'w' et 'x' pour tourner sur soi-même
- 'f' pour enregistrer une séquence d'images prises automatiquement le long d'un cercle
- 'l' pour enregistrer une séquence d'images prises automatiquement avec déplacement de la lumière

Après avoir appuyé sur 'f' ou 'l', on peut construire une vidéo à partir des images prises automatiquement grâce à la commande :

tools/createvideo.sh

en se plaçant dans le répertoire contenant les images, et en appelant le createvideo au bon endroit.

4.4 Images obtenues

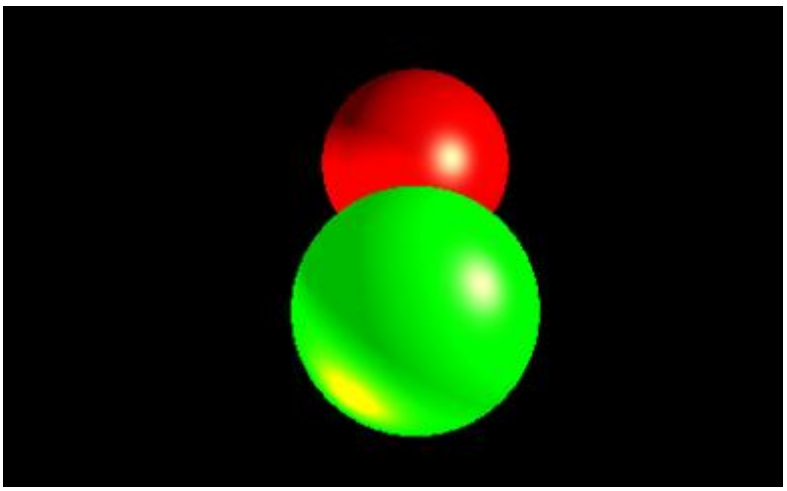

Image 15 Sans ombre

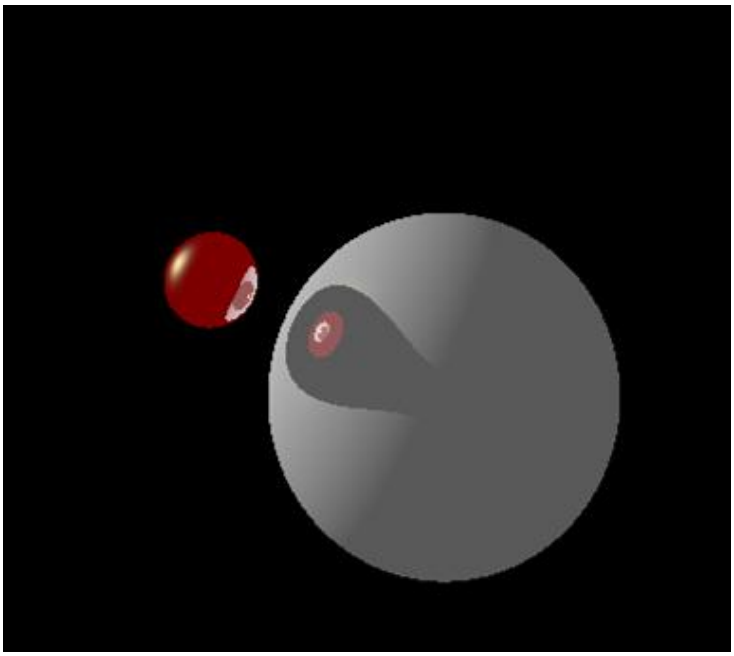

Image 16 Avec ombre

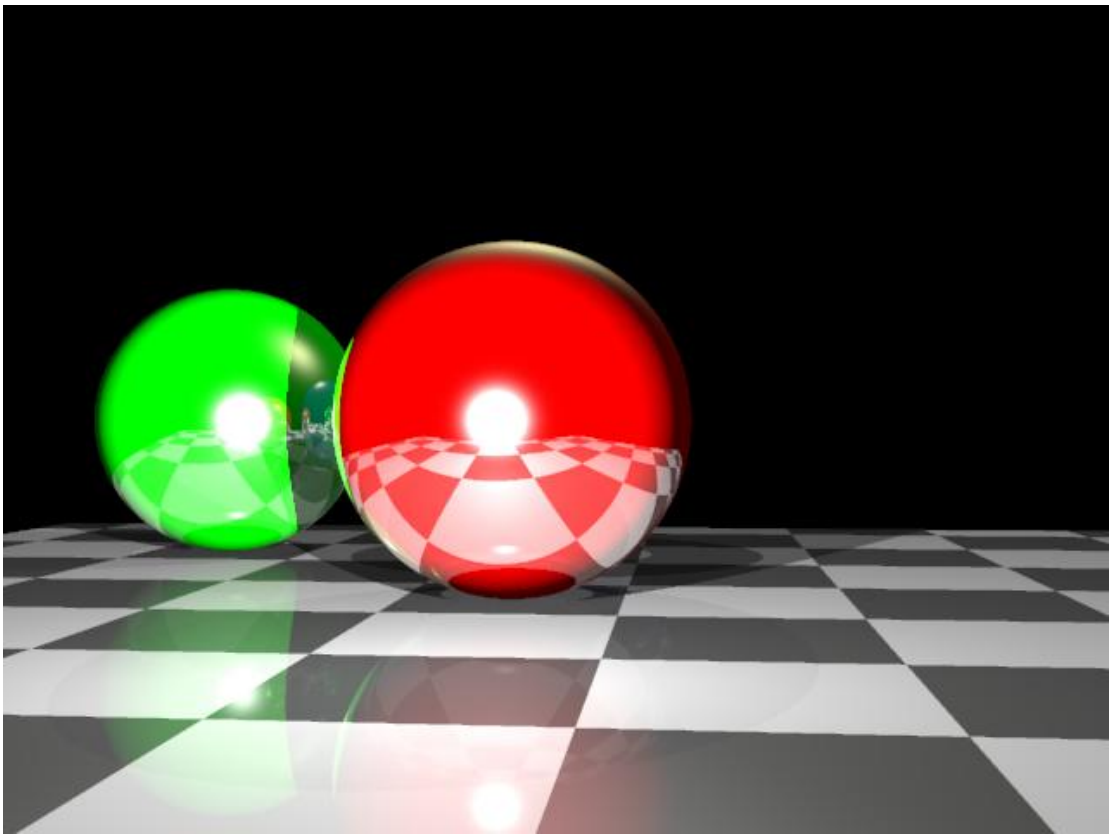

Image 17 Le damier

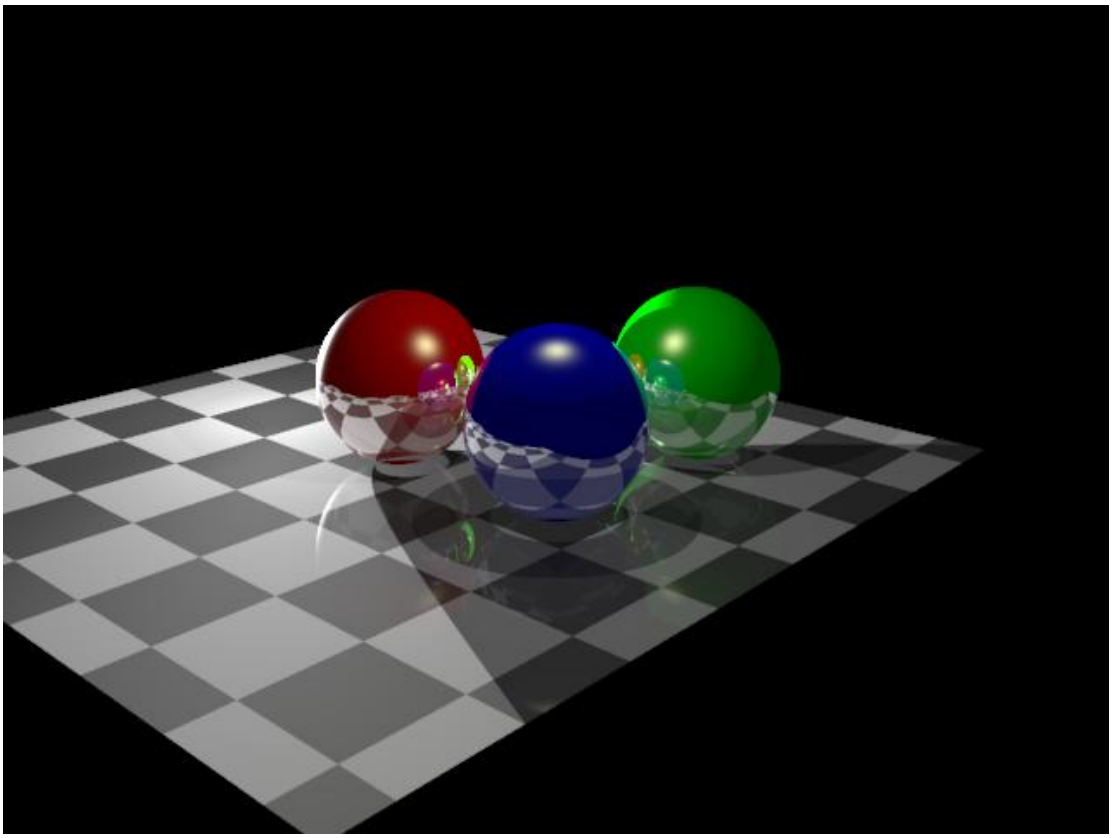

Image 18 Rendu finale

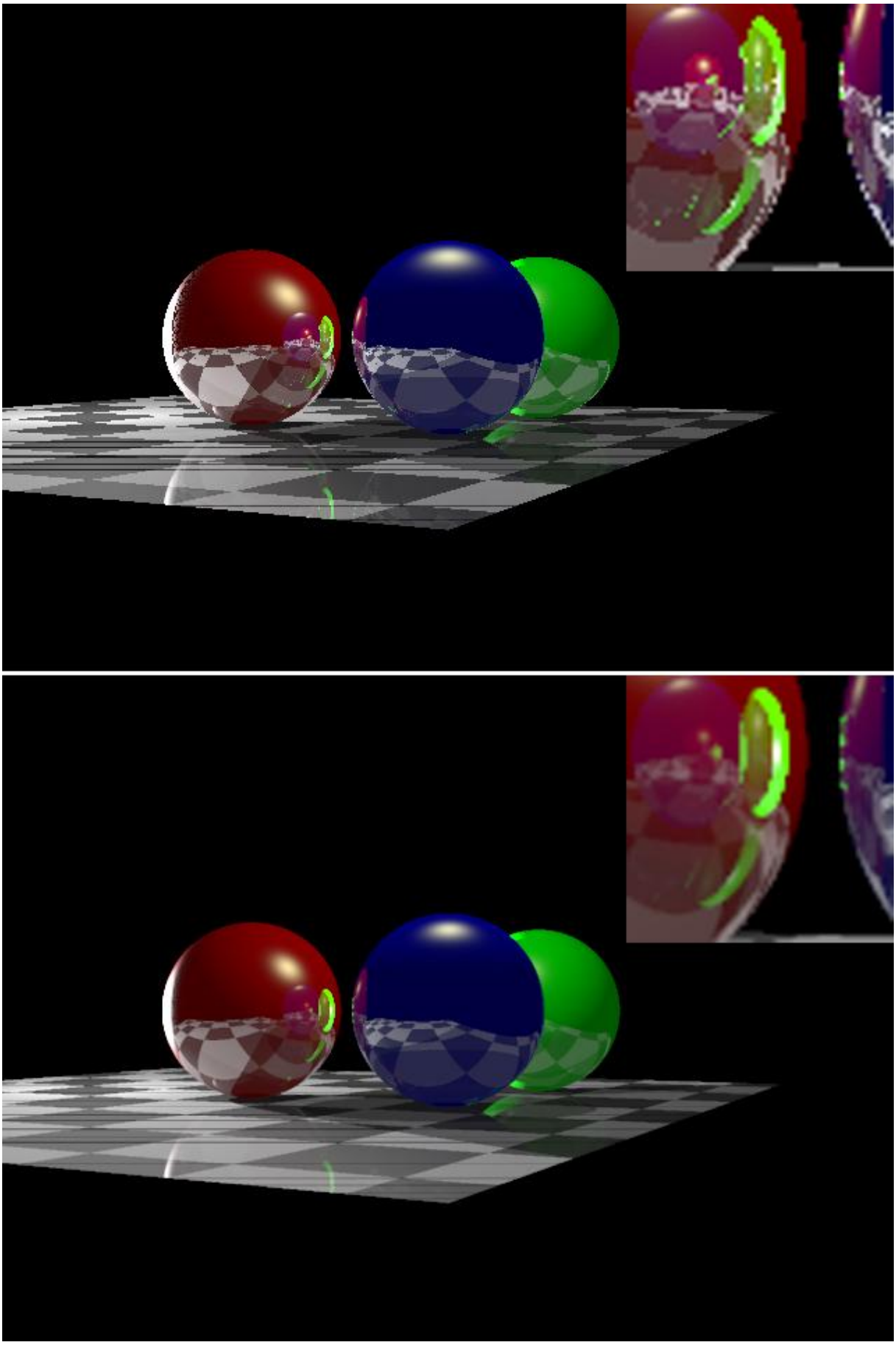

Image 19 Effects d'antialiasing (Image petite: à 300%)

5. Conclusion

5.1 Difficultés rencontrées

Nous avons connu une première difficulté en voulant connecter la bibliothèque Lapack, utilisée pour faire les calculs d'intersections, avec notre code C++. En effet, cette bibliothèque est principalement basée sur du Fortran et sa liaison avec un projet est vraiment difficile à mettre en place. Heureusement, Jean-Christophe Lombardo, un chercheur à l'espace immersif dans l'équipe DREAM à l'INRIA, s'est déplacé pour résoudre notre problème.

Laisser la compilation de notre projet à l'IDE Eclipse n'était pas nécessairement un bon choix. En effet, après les mises à jour, chacun ayant son propre programme principal de test utilisant les autres classes du projet, nous nous retrouvions facilement avec plusieurs fichiers contenant une fonction "main", ce qui empêchait de compiler le projet correctement. De plus, la connexion avec la bibliothèque Lapack n'a pas été possible sous le logiciel. La configuration des options de compilation et des chemins menant aux bibliothèques externes a été pour nous un procédé assez lourd à gérer. Pour pallier à ce problème, nous avons finalement opté pour l'utilisation de CMake, facile à configurer et à personnaliser.

Sans compter que les erreurs de C++ sont généralement loin d'être les plus explicites.

De plus, l'implémentation de la classe écran permettant de voir les objets de la scène nous a posé quelques difficultés. Tout d'abord, il a fallu savoir où le positionner pour voir nos objets. Ensuite, nous avons dû nous mettre d'accord sur une convention d'orientation des axes. Et la gestion de son déplacement n'est toujours pas fait de manière rigoureuse : le vecteur "vers le haut" associé à l'observateur n'est pas mis à jour automatiquement lorsqu'une modification de la position de l'observateur ou du point visé est appliquée, c'est pour l'instant au programmeur de répercuter manuellement sur ce vecteur la transformation qu'il effectue. Pour l'instant, selon l'angle de vue, on peut observer une distorsion de l'image, principalement lorsque l'on déplace la caméra vers le haut.

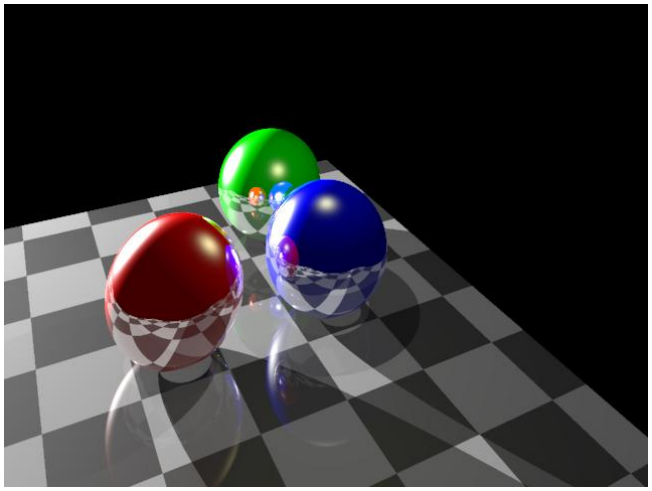

Image 20 Sur cette image, la boule rouge semble plus ellipsoïdale que sphérique

Enfin, nous avons parfois rencontré des erreurs étranges, qu'il était difficile de corriger en nous basant uniquement sur les images obtenues. Ainsi, nous avions des rayures sombres qui apparaissaient sur le damier sans aucune explication. Elles étaient localisées, et ne dépendaient pas

vraiment de l'orientation par rapport aux axes. En normalisant nos vecteurs, nous avons réussi à passer d'un problème de rayures à un problème de points sombres.

En réfléchissant un peu et en effectuant différents tests pour faire varier l'angle de vue et en isoler les différents modules, nous avons fini par comprendre le problème : quand nous regardions la contribution des sources de lumière sur les points du damier, les points incriminés n'appartenaient tout simplement pas à la surface. En effet, avec les approximations de calcul dues au format double, ces points étaient considérés comme étant un peu en-dessous du damier. Par conséquent, tout se passait comme si le rayon damier/source de lumière le traversait : ces points étaient donc dans l'ombre. Nous avons simplement pris un rayon de même direction qui part d'un point légèrement au dessus du damier, pour être sûr que le nouveau rayon ne trouve pas immédiatement une intersection (avec l'objet lui-même d'où part ce rayon). C'est une technique que nous avions appliquée pour les sphères et qui fonctionnait bien, mais que nous avions oublié de reproduire sur le damier.

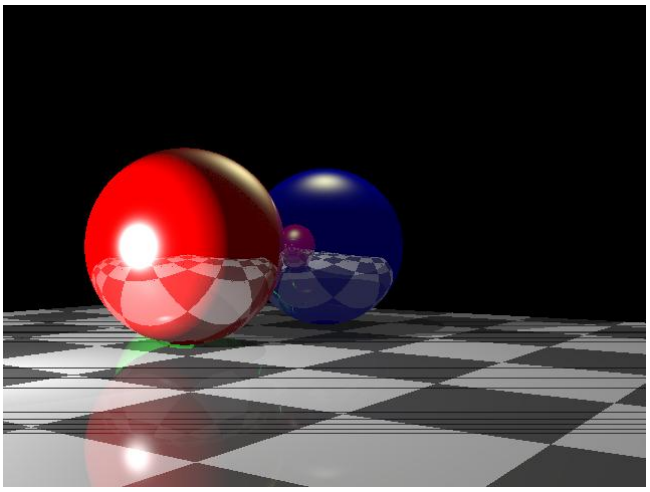

Image 21 Les bandes noires visibles sur l'image ont été assez difficiles à supprimer

5.2 Réalisation

Nous avons rempli les exigences du cahier des charges et avons ajouté des modules supplémentaires (déplacement de caméra, de lumières, export d'une animation, ...). En effet, le but était que notre raytracer puisse calculer les intersections avec n'importe quel type d'objet, pour pouvoir facilement brancher un nouveau type d'objet défini par une surface paramétrée (c'est ici que se trouve la liaison avec l'autre partie de ce projet, réalisée par les trois étudiants en MAM). Nous avons donc déterminé qu'un objet affichable devait obligatoirement disposer d'une fonction renvoyant ses intersections avec un rayon, ainsi qu'une fonction qui nous renvoie la normale de la surface en un point. Le tout en utilisant la représentation paramétrée d'un rayon (o + td). Si l'autre équipe arrive à nous fournir la normale et le paramètre t en chaque point de l'écran, nous pourrons visualiser n'importe quelle surface paramétrée.

Ensuite, nous avons également implémenté d'autres fonctionnalités. Le ray tracer est améliorable à l'infini, et nous avons choisi quelques modules à ajouter pour obtenir un meilleur rendu.

- Nous avons choisi pour modèle de la lumière celui de Phong, classique mais efficace, puisqu'avec quelques constantes et une formule physique, il permet de jouer assez bien sur le rendu des matériaux.
- Ensuite, nous avons estimé que l'ajout d'anti-aliasing était un choix pertinent. En effet, il est rapide à programmer et donne des résultats visibles, même avec neuf rayons par pixel.
- Nous avons également ajouté le déplacement de la caméra. Même si le ray tracing n'est pas fait pour une visualisation en temps réel, il nous a permis de voir le comportement du modèle de Phong sous différents points de vue, ce qui nous a aussi aidé à voir si tout fonctionnait correctement.
- De plus, comme dans beaucoup de ray tracer, le damier était une étape importante : grâce à lui, on peut voir les différentes cases se refléter dans les sphères, ce qui appuie la validité du modèle utilisé. Le damier étant un rectangle avec des cases et le rectangle étant un plan fini, nous avions aussi la possibilité d'obtenir trois nouvelles formes usuelles assez simplement.
- Enfin, l'enregistrement de vidéos à partir de clichés pris à intervalles réguliers est un petit plus qui fait toujours son effet : la vidéo obtenue montre assez bien le jeu des ombres et lumières en mettant en scène des images de bonnes qualités. C'est une sorte d'aboutissement de notre projet : les images s'animent, un peu comme dans un studio d'animation.

Pour terminer, notre code est suffisamment souple et modulaire pour changer de modèle de la lumière, régler le nombre de sources lumineuses et d'objets dans la scène, ajouter des textures ou encore travailler avec des équations implicites.

5.3 Améliorations possibles

Comme expliqué dans la partie précédente, le raytracer peut être amélioré à indéfiniment. Les travaux dans ce domaine ont été largement exploré, et parmi les modules à ajouter, on peut citer :

- la gestion de la réfraction, semblable à la réflexion et qui permet de gérer des différences de milieux transparents tel que la modélisation d'un verre et d'un liquide à l'intérieur;
- l'ajout de textures et de bruits, pour créer quelques effets amusants;
- la possibilité de plaquer une image sur une surface. On pourrait ainsi s'affranchir du calcul de modulo dans la création du damier, juste en appliquant un motif de carreaux sur une rectangle, ou même de créer des damier plus complexes, à motif. Une autre possibilité amusante consisterait à obtenir des boules de billards en appliquant des images se résumant à un carré de couleur et un chiffre en gras sur fond blanc;
- Changer le modèle de Phong pour un plus complexe mais générant des images plus réalistes;
- afin de gagner en rapidité dans la génération d'images, nous pourrions programmer une classe BoundingVolume, capable de délimiter les objets de manières grossières et de réduire le nombre de calculs;
- nos rayons lumineux agissent comme dans le vide : ils ne perdent jamais en intensité ni en puissance, l'ajout de calculs de radiométrie donnerait un rendu plus réaliste
- enfin, en ce qui concerne le rendue, la technique de "mapping photon" consistant à envoyer des photon dans toutes les directions de manière aléatoire simulerait correctement ce qui se passe dans la nature. C'est un procédé évidemment très long puisque beaucoup de ces photons n'atteignent pas l'œil, mais il donne des résultats assez impressionnants.

6. Remerciements

Nous tenons à remercier notre tuteur Mr Laurant Busé pour sa disponibilité et ses précieux conseils.

Nous remercions également Mr Jean-Christophe Lombardo qui nous a permis de lier la bibliothèque Lapack à notre projet.

7. Table des Images

8. Références

[1] http://medias.3dvf.com/publish/raytracing/figure1.png

[2] http://mathinfo.univ-reims.fr/image/siRendu/Documents/2004-Chap5-BRDF.pdf

[3] http://mathinfo.univ-reims.fr/image/siRendu/Documents/2004-Chap6-RayTracing.pdf

[4] http://en.wikipedia.org/wiki/Phong_reflection_model

[5] **Shirley P., Morley R.: Realistic Ray Tracing**