Rapport du projet de Technologies Objets

Maxime Arthaud Korantin Auguste Martin Carton

11 juin 2013

Table des matières

Table des figures

1 Introduction

Ce projet consiste en la création d'une interface utilisateur permettant de gérer une scène 3D et à la réalisation d'un moteur de rendu par lancé de rayons.

Comme expliqué dans le rapport d'analyse, nous avons décidé de nous découper le travail, de façon à ce qu'une personne fasse l'interface graphique, une autre le parseur de fichier et l'écriture d'images au format PPM, et une autre travaille particulièrement sur le cœur du raytracer.

2 Architecture

Le projet est donc constitué de deux programmes distincts :

L'interface graphique Qui devra permettre de créer simplement des fichiers représentant des scènes, à passer à notre raytracer.

Le raytracer Qui, à partir d'un fichier représentant une scène, devra générer un rendu, dans le format d'image de notre choix.

2.1 Écriture d'images PPM

Pour écrire les images PPM, nous avons écrit un plugin Java qui permet d'enregistrer ce format auprès de `javax.image.ImageIO` qui propose une interface commune pour enregistrer des images dans différents formats. Ainsi notre programme peut générer des images au format PPM, mais aussi par exemple en PNG.

Le choix de ce format est déterminé par le troisième paramètre d'appel du programme (la valeur pas défaut est png, ce format étant plus commun).

Ce plugin consiste en deux classes `imageio.PPMImageWriterSpi` et `imageio.PPMImageWriterSpi`. La première permet d'indiquer à Java les capacité de notre plugin. La deuxième permet d'écrire une image sur un flux de sortie quelconque.

2.2 Format de fichier

Nous avons décidé de mettre à disposition un format de fichier dans lequel on peut décrire une scène afin de pouvoir enregistrer celles-ci, ou de la générer automatiquement (ce qui nous a même permis de faire quelques vidéos).

Le format de fichier ressemble à ceci :

Listing 1 – Exemple de fichier associé à la figure 1

```
//exemple de scene :
Camera(eye=(0, 0, 0), origin=(-0.5, 0, 1.5), abscissa=(1, 0, 0),  \
ordinate=(0, 1, 0), widthPixel=200, heightPixel=200)
AmbientLights(0.2, 0.2, 0.2)
Light(pos=(5, 0, 0), intensity=(0.5, 0.5, 0.5))
Sphere(center=(0, 5, 20), radius=3.14, k_diffuse=(0.6,0.3,0.6))
Plane(p1=(-1,0,30), p2=(1, 0, 30), p3=(0, 1, 30))
```

Il y a un élément par ligne et seule la ligne décrivant au moins une caméra est obligatoire. La casse et l'espacement sont ignorés. La plupart des propriétés ont une valeur par défaut afin d'éviter de surcharger le fichier et de permettre de l'écrire à la main. Des commentaires de fin de ligne commençant par « `//` » peuvent être insérés.

Les éléments suivants peuvent être ajoutés :

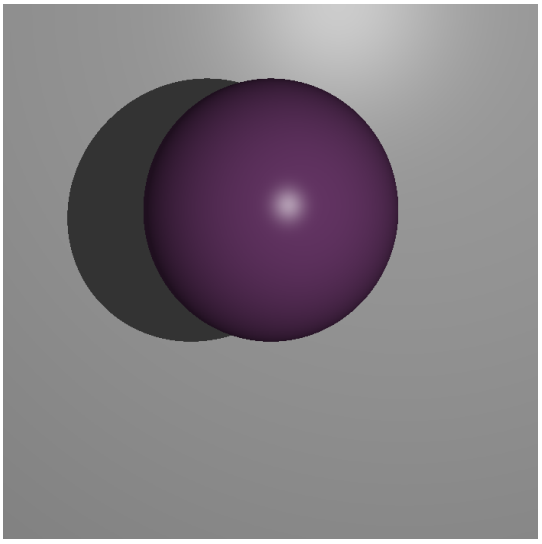


FIGURE 1 – Exemple de rendu simple

Camera décrit une caméra de la scène : elle doit posséder les propriétés suivantes :

- **eye** un point décrivant la position de l'œil ;
- **origin** un point décrivant l'origine du rectangle de l'écran ;
- **abscissa et ordinate** les vecteurs qui avec *origin* définissent l'écran ;
- **widthPixel et heightPixem** deux entiers donnant la taille de l'image à générer.

AmbientLights donne les trois composantes primaire de la lumière ambiante.

Light décrit une source de lumière, elle doit avoir les propriétés suivantes :

- **pos** la position de cette source ;
- **intensity** l'intensité lumineuse de chaque couleur primaire.

Sphere décrit une sphère, doit avoir les propriétés suivantes :

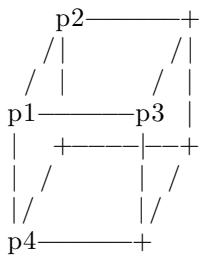
- **center** un point représentant le centre de la sphère ;
- **radius** un flottant représentant le rayon de la sphère.

Plane décrit un plan, doit avoir trois points nommés *P1*, *P2* et *P3*.

Triangle décrit un triangle, doit avoir trois points nommés *P1*, *P2* et *P3*.

Parallelogram décrit un parallélogramme, doit avoir trois points nommés *P1*, *P2* et *P3*.

Cube décrit un cube [1], doit avoir quatre points nommés *P1*, *P2*, *P3* et *P4* répartis comme ceci :



1. En fait les points ne sont pas testés pour former un cube, cet objet peut représenter n'importe quel parallélépipède.

Les objets *Sphere*, *Plane*, *Triangle*, *Parallelogram* et *Cube* peuvent en plus avoir les propriétés optionnelles suivantes :

brightness flottant, par défaut à 30. Plus ce nombre est grand, plus la tâche provoquée par la composante spéculaire sera localisée.

k_specular flottant, par défaut à 1. Coefficient par lequel on multiplie la composante spéculaire.

k_diffuse triplet, par défaut à *(1, 1, 1)*. Coefficients par lesquels on multiplie les composantes diffuse pour chaque couleur. Donne donc la couleur de l'objet.

k_reflection flottant, par défaut à 0. Coefficient par lequel on multiplie la composante réfléchie.

k_refraction triplet, par défaut à *(0, 0, 0)*. Coefficients par lesquels on multiplie les composantes réfractées pour chaque couleur.

refractive_index flottant, par défaut à 1 (la lumière réfractée ne sera alors pas déviée). Indice du milieu intérieur à l'objet.

Un objet de type `Scene` est construit à partir d'un tel fichier à l'aide la méthode statique `raytracer.FileReader.read`.

2.3 Interface graphique

Maxime ?

2.4 Raytracer

Pour créer le raytracer, nous avons convenu d'architecturer notre programme selon le diagramme de classes suivant présenté en figure 4.

2.4.1 Changements de conception

Par rapport à ce que nous avions prévu lors de notre rapport d'analyse, nous avons fait quelques modifications : nous avons rajouté une classe pour représenter une lumière, que nous avions oubliée, nous n'utilisons pas la classe `java.awt.color` car il s'est avéré plus pratique de manipuler des `double[]` et nous n'utilisons pas `javax.vectmath`, qui n'est pas installé à l'ENSEEIHT, mais nos propres classes.

Ensuite, une scène peut posséder plusieurs caméras, ce qui permet d'avoir plusieurs images de la même scène de plusieurs points de vues différents. Nous avons ajouté les méthodes dont nous avons eu besoin à la classe `Object`. Les classes `Triangle` et `Plane` héritent de `TriPointed` car ces objets ont beaucoup de points en commun : ils sont représentables par trois points et les formules qu'il y a derrière ces classes sont les mêmes ; nous en avons profité pour ajouter une classe `Parallelogram` pour les mêmes raisons.

La classe `Mesh` qui devait représenter un objet composé d'un ensemble de facettes triangulaires a été changée pour représenter un ensemble d'objets basiques, puisque nous n'utilisions rien de spécifique aux triangles la restriction n'était pas nécessaire, bien sûr un objet à facettes triangulaires peut être représenté par cette classe.

Enfin nous avons ajouté une classe `Texture` pour alléger notre conception.

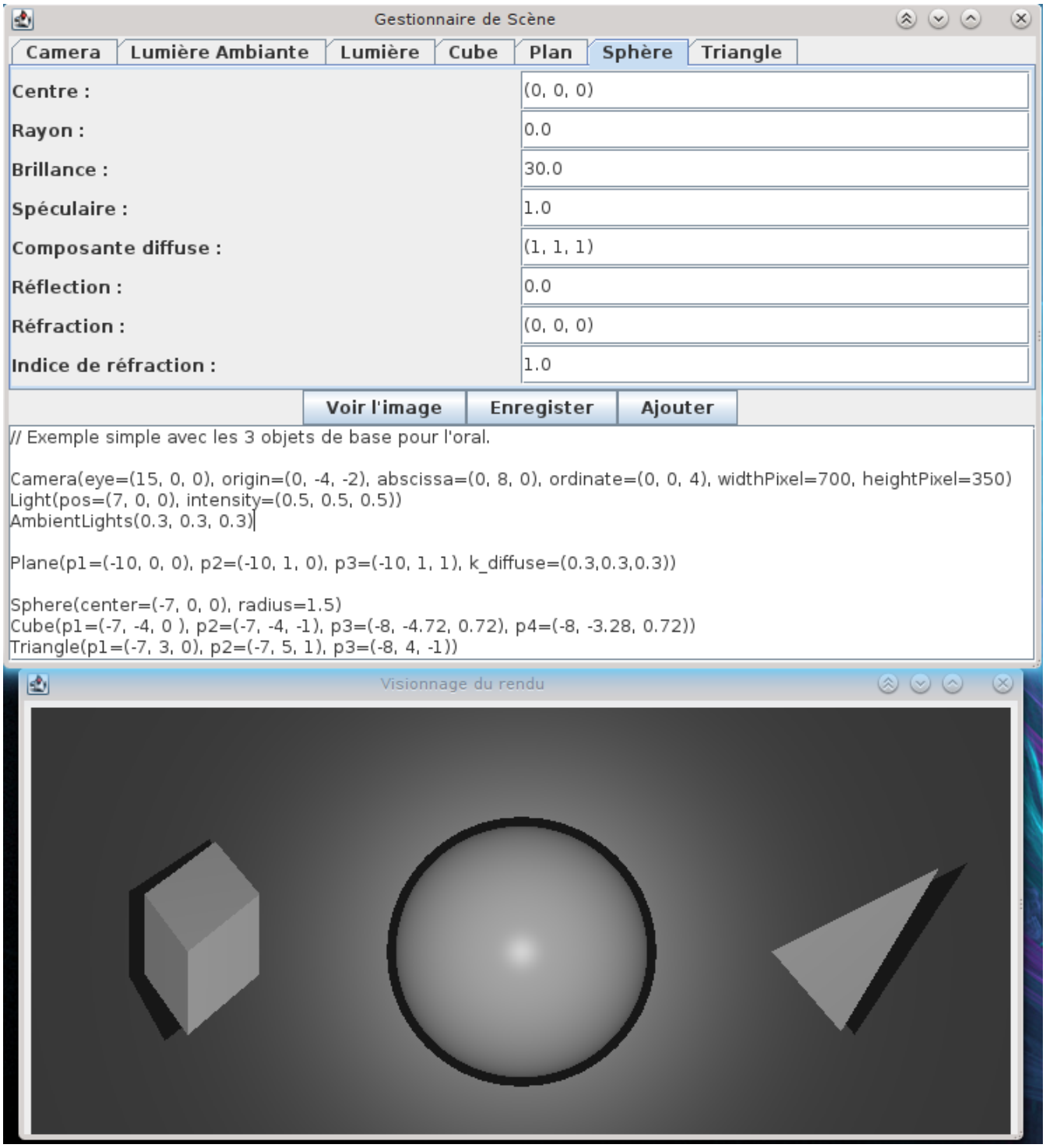


FIGURE 2 – Interface graphique

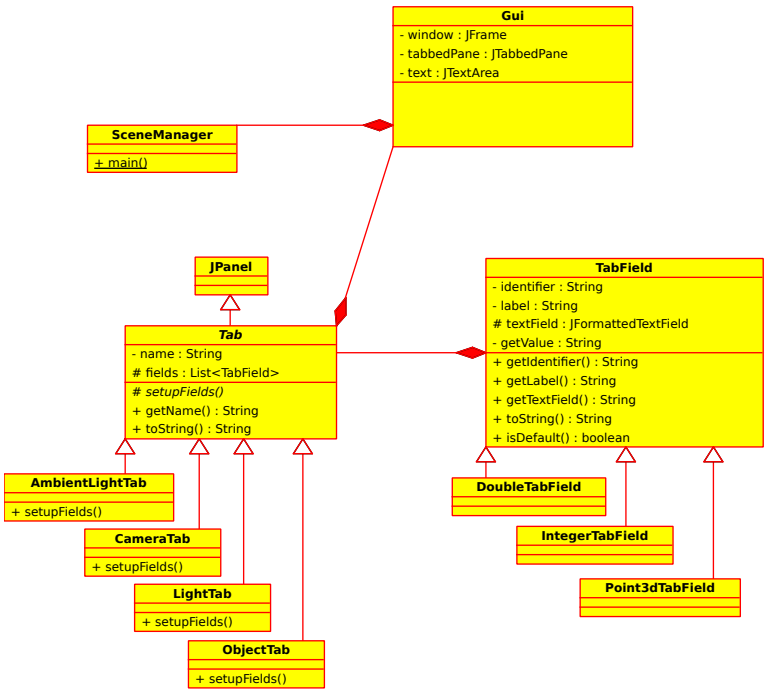


FIGURE 3 – Diagramme UML de la GUI

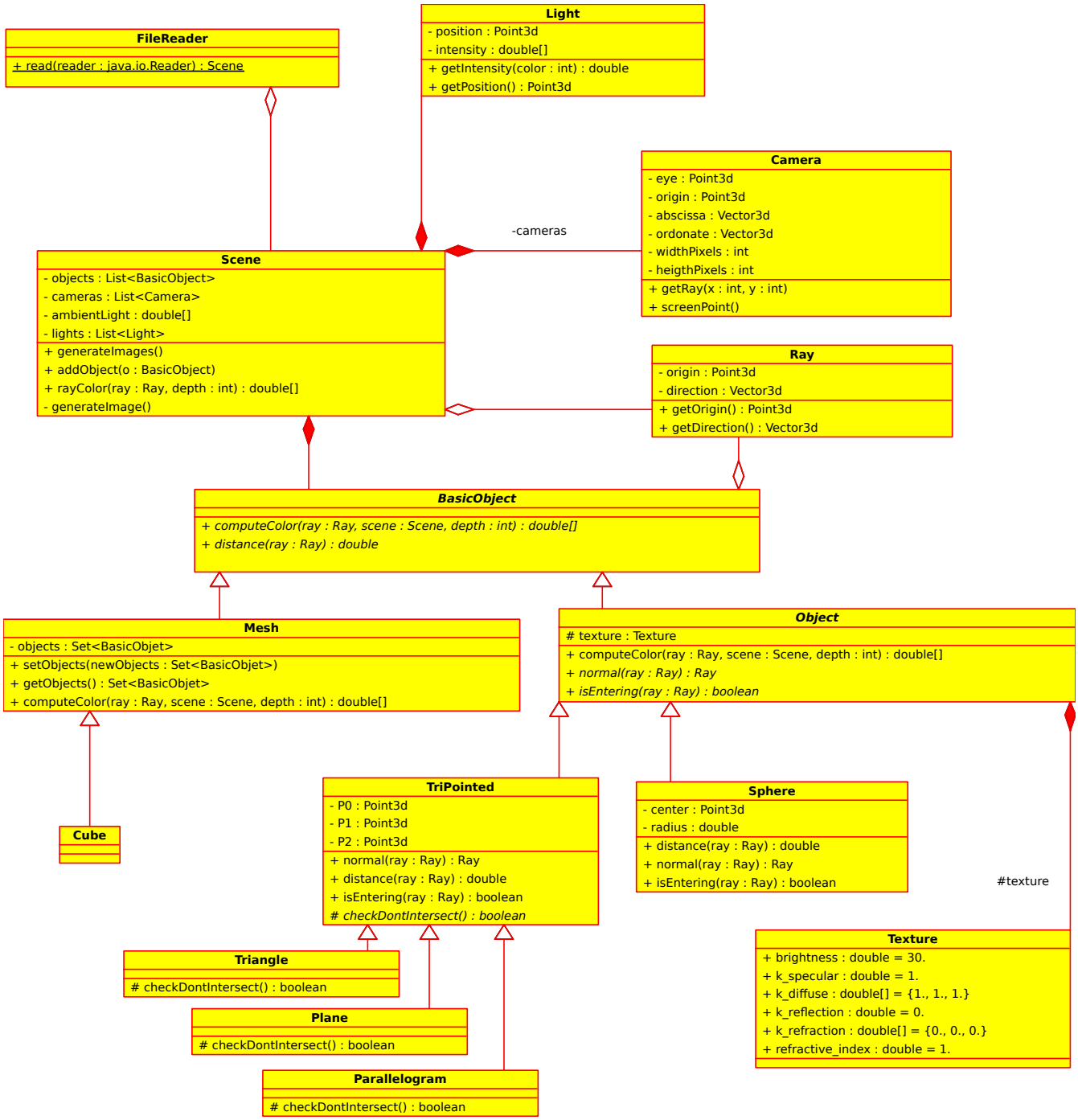


FIGURE 4 – Diagramme UML

2.4.2 Fonctionnement

L'objet `Scene` dispose d'une méthode pour calculer la couleur d'un rayon passé en paramètre. Pour le faire, il va regarder quel objet va intersecter avec le rayon en premier, et appeler la méthode `computeColor` de l'objet en question.

Cette méthode, définie dans la classe `Object`, va faire tous les calculs nécessaires pour calculer les différentes composantes. Pour cela, elle peut même appeler à nouveau la méthode `rayColor` de la classe scène, sur les rayons réfléchis ou réfractés. Dans ces calculs, elle va appeler la méthode `normal`, qui va donner la normale au point d'intersection du rayon avec l'objet, méthode qui sera définie dans des sous-classes, en fonction de l'objet.

2.4.3 Problèmes

Un problème qui se pose est que, en lançant le rayon avec la méthode `rayColor` d'une scène, à partir d'un point d'intersection avec un objet, l'objet le plus proche qui va intersecter de nouveau sera l'objet lui même, à ce même point d'intersection et avec une distance de zéro. Pour cela, on aurait pu définir un ε et vérifier que la distance y soit supérieure, mais cela entraînait des problèmes de pixels complètement différent des autres, et il aurait fallu trouver une valeur adéquate pour ε. Nous avons donc décidé de ne pas utiliser ce système, et tout simplement d'ignorer l'objet qui intersecte, lors du lancer de rayon. La méthode `rayColor` a donc un paramètre (pouvant être nul), qui indique si elle doit ignorer un objet. Ce système pose toutefois un problème : un rayon entrant dans une sphère coupera toutefois la sphère en un autre endroit. Ce cas est actuellement ignoré, et fait que notre réfraction dans une sphère est peu représentative de la réalité.

Un autre problème assez important, mais lié à la façon dont le raytracer fonctionne, est qu'un objet transparent ou qui va seulement dévier légèrement la lumière, sera capable de masquer les sources lumineuses. Cela veut dire qu'il laissera une ombre, alors qu'il est pourtant transparent.

On peut évaluer la complexité de notre algorithme à du $O(x \times y \times n \times m)$, où :

- x est la largeur de l'image,
- y est la hauteur de l'image,
- n est le nombre d'objets dans l'image,
- m est le nombre de sources lumineuses.

On voit donc que même si l'algorithme est rapide sur des scènes simples, plus la scène contiendra d'objets et de lumières, plus le temps de rendu sera élevé, même pour une scène simple, il n'est ainsi pas possible d'afficher le rendu en direct dans l'interface graphique.

2.4.4 Améliorations

En changeant la façon dont le raytracer fonctionne, il serait possible d'éviter ce genre de problème : une solution simple est d'utiliser des simulations de Monte-Carlo : pour obtenir l'éclairement d'un point, on lancera des rayons à partir de ce point, dans des directions choisies au hasard (mais réparties de manière non uniforme, pour tenir compte que plus de lumière provient des rayons dans la direction de la normale), et on regardera la lumière diffuse émanera de ces points. Ainsi, le problème d'ombre précédemment développé pourra être éliminé. Toutefois, cette méthode est très coûteuse en calculs.

Concernant le temps de rendu sur de grosses scènes, plusieurs choses sont possibles :

- On pourrait paralléliser la génération des images sur tous les cœurs disponibles sur l'ordinateur. C'est typiquement quelque chose de trivial à paralléliser puisque chaque pixel est généré indépendamment des autres (il faudrait cependant modifier nos objets qui conservent des données calculés entre les différents appels de méthodes).

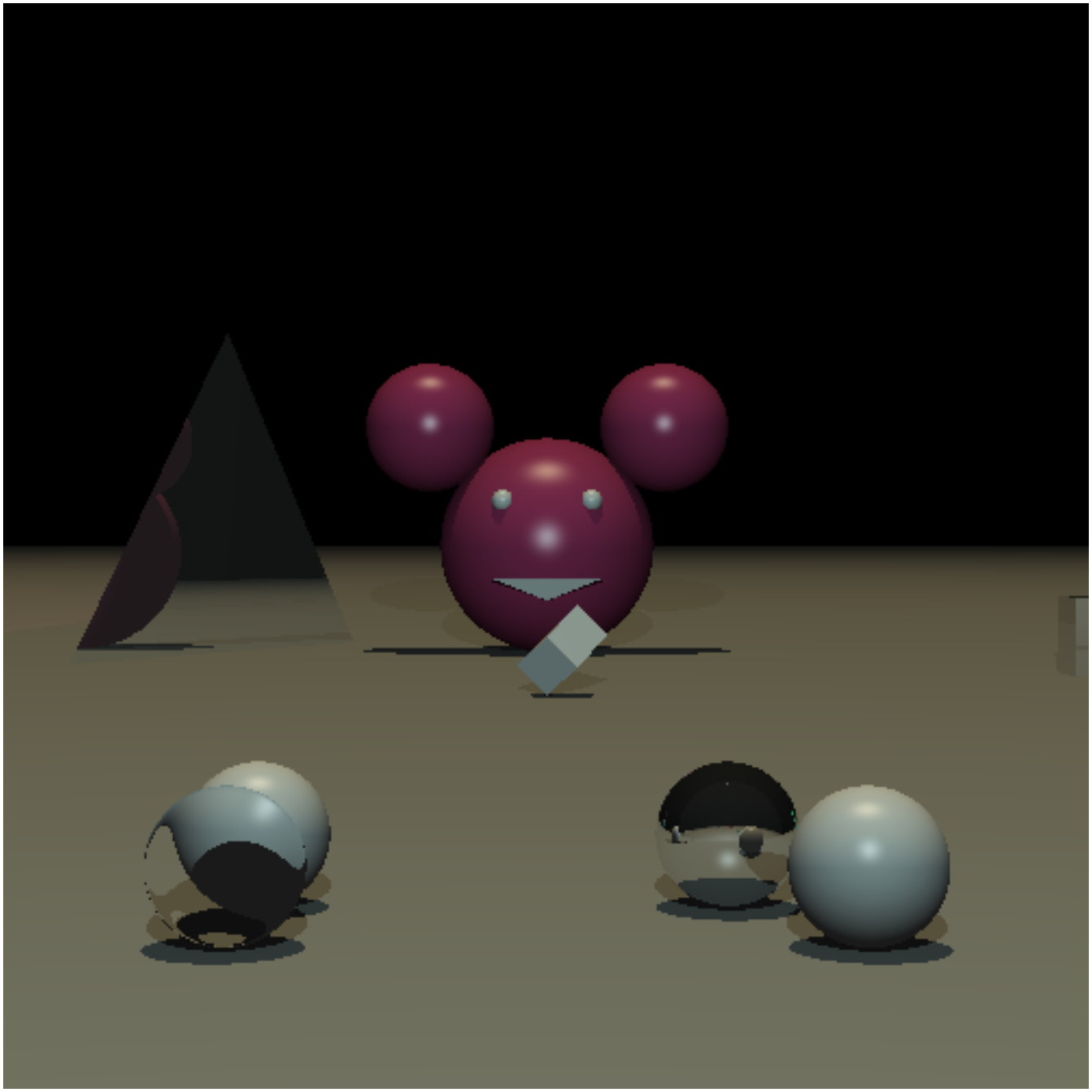


FIGURE 5 – Exemple de rendu complexe

– Pour chaque source de lumière, plutôt que de parcourir tous les objets pour voir si ils interceptent, on pourrait parcourir seulement qui sont sur le parcours du rayon. Pour cela, il nous faudrait diviser notre espace 3D, par exemples à l'aide d'une partition binaire de l'espace (BSP), ou d'un *octree*.
– On pourrait aussi profiler le code, et essayer de voir si certaines méthodes ne gagneraient pas à être optimisées.