

Groupe 12  
Anthony Foulfoin  
Julien Badie  
Samuel Kohn  
Benoît Reulier  
Maxime Blaess

## **Synthèse d'image par lancer de rayon**

# Table des matières

1	Introduction .....	5
2	Le lancer de rayon .....	5
2.1	Principe .....	5
2.2	Algorithme du lancer de rayon .....	6
2.3	Calcul des composantes .....	7
2.3.1	Composante diffuse .....	7
2.3.2	Composante réflexive.....	7
2.3.3	Composante réfractée .....	8
2.3.4	Composante ambiante .....	8
2.3.5	Composante spéculaire .....	8
3	Conception .....	9
3.1	Objets modélisables .....	9
3.1.1	La sphère .....	9
3.1.2	L'objet à facettes .....	9
3.1.3	Le cube .....	9
3.1.4	Le plan.....	9
3.1.5	La lumière .....	9
3.2	Les packages du programme.....	10
3.2.1	interfaceGraphique .....	10
3.2.2	interfaceTexte .....	10
3.2.3	lancerRayon.....	10
3.2.4	tests .....	10
3.2.5	util .....	10

3.2.6	xmlParser .....	10
3.3	Diagrammes de classes et description des classes .....	11
3.3.1	Package lancerRayon .....	11
3.3.2	Package util .....	14
3.3.3	Package xmlParser .....	14
3.3.4	Package interfaceGraphique.....	15
3.4	Mécanisme de mise à jour des composants d’affichage .....	16
3.5	MultiThreading de l’application .....	16
3.6	Différences par rapport à la phase d’analyse .....	17
4	Interface utilisateur .....	19
4.1	Interface graphique .....	19
4.2	Interface en ligne de commande.....	22
5	Tests .....	24
6	Organisation du projet .....	25
6.1	Environnement de développement.....	25
6.2	Répartition des tâches.....	25
6.3	Travail réalisé par chaque membre .....	26
6.3.1	Benoît .....	26
6.3.2	Anthony .....	26
6.3.3	Maxime .....	27
6.3.4	Julien.....	27
6.3.5	Samuel .....	28
7	Conclusion et perspectives .....	29
7.1	Conclusions personnelles .....	29
7.1.1	Benoît .....	29

7.1.2	Anthony .....	29
7.1.3	Maxime .....	30
7.1.4	Julien .....	30
7.1.5	Samuel .....	30
7.2	Conclusion générale et perspectives .....	31

# 1 Introduction

Le lancer de rayon est une technique de rendu en synthèse d'image simulant le parcours inverse de la lumière de la scène vers l'œil.

Cette technique simple reproduit les phénomènes physiques que sont la réflexion et la réfraction et permet de résoudre simplement certains problèmes liés à la synthèse d'images tels que l'élimination des parties cachées et le calcul des ombres portées.

Grâce à cet algorithme, il est ainsi possible d'obtenir une projection 2D à partir d'une scène 3D.

L'objectif de ce projet était de réaliser un programme de lancé de rayons, dont le principe est rappelé en tout premier lieu.

## 2 Le lancer de rayon

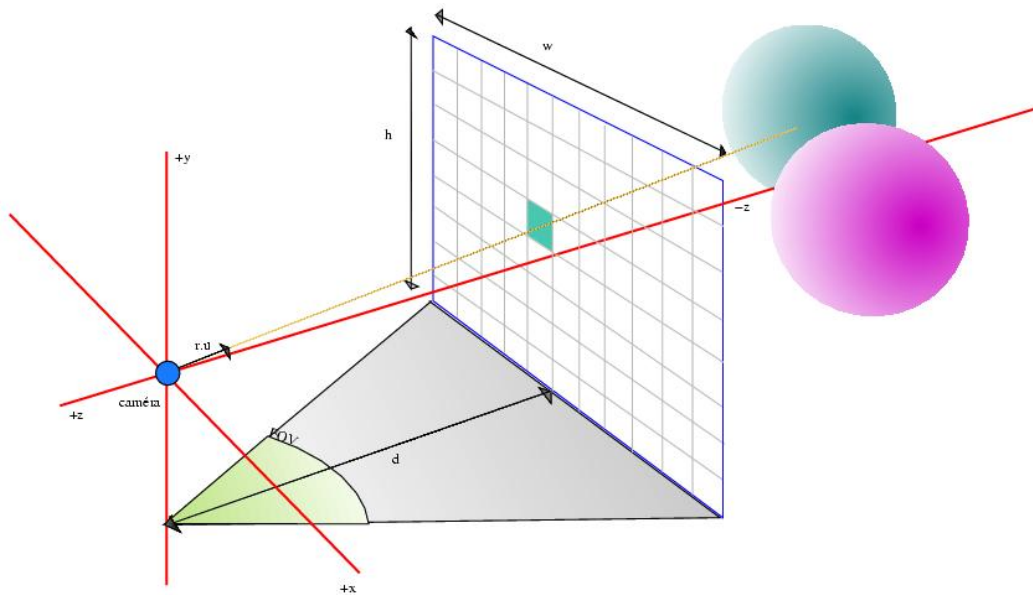
### 2.1 Principe

Le but de l'algorithme du lancer de rayon est de calculer une image, afin d'obtenir une projection d'une scène en 3D.

Ainsi l'image est représentée par un écran sur lequel on projette la scène 3D, cette image étant observée depuis un point de vue (encore appelé caméra) donné.

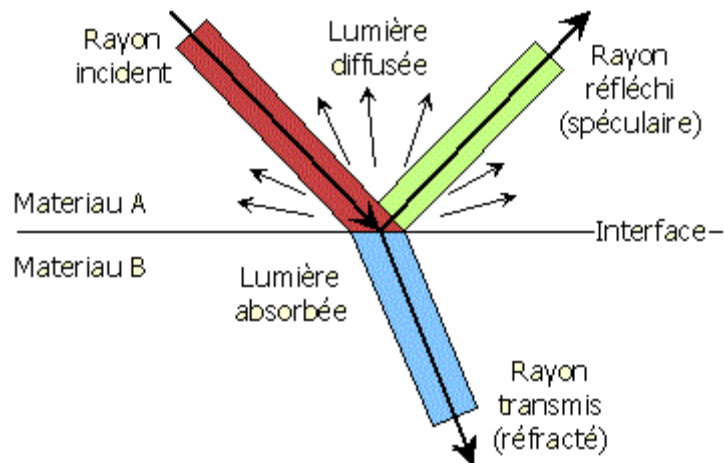
Le lancer de rayon consiste, pour chaque pixel de l'image générée, à lancer un rayon depuis le point de vue vers ce pixel. Le premier point d'impact du rayon sur un objet définit la couleur du pixel correspondant grâce au calcul de plusieurs composantes.

Des rayons sont alors lancés depuis le point d'impact en direction de chaque source de lumière pour déterminer sa luminosité (est-il éclairé ou à l'ombre d'autres objets ?). Cette luminosité combinée avec la couleur de l'objet ainsi que d'autres informations éventuelles (angles entre la normale à l'objet et les sources de lumières, réflexions, transparence, etc.) déterminent la couleur finale du pixel.



## 2.2 Algorithme du lancer de rayon

1. On commence par lancer un rayon depuis le point de vue vers un pixel de l'écran
2. Deux cas sont alors possibles :
  - a. Aucun objet n'intercepte le rayon. Dans ce cas, le pixel prend la couleur de la lumière ambiante.
  - b. Des objets interceptent le rayon. Dans ce cas :
    - On trouve l'objet le plus proche de du point de vue
    - En fonction des caractéristiques de cet objet vis à vis de la lumière, la teinte du pixel est calculée à partir des composantes de lumière diffusée, ambiante, spéculaire, réfléchie et transmise par l'objet au point d'intersection créé.



## 2.3 Calcul des composantes

La couleur d'un pixel est le résultat de la somme de plusieurs composantes.

### 2.3.1 Composante diffuse

Pour calcul la composante diffuse, on devra :

- déterminer le point d'intersection entre l'objet et le rayon incident.
- lancer un rayon de ce point vers chacune des sources lumineuses présentes dans la scène pour déterminer si ce rayon est intercepté ou non par un objet (à l'ombre de celui-ci ou non).
- pour chaque rayon non intercepté, appliquer la formule correspondante (loi de Lambert) au calcul de la lumière diffuse

### 2.3.2 Composante réflexive

La valeur obtenue par réflexion est calculée :

- en relançant récursivement l'algorithme de lancer de rayons sur un rayon secondaire ayant pour origine le point d'intersection et pour direction la direction de réflexion du rayon initial.
- en multipliant la valeur obtenue par le coefficient de réflexion de l'objet.

### 2.3.3 Composante réfractée

La valeur obtenue par réfraction est calculée :

- en relançant récursivement l'algorithme de lancer de rayons sur un rayon secondaire ayant pour origine le point d'intersection et pour direction la direction de réfraction du rayon initial.
- en multipliant la valeur obtenue par le coefficient de réfraction de l'objet.

### 2.3.4 Composante ambiante

La valeur obtenue est calculée en multipliant le coefficient d'absorption de l'objet par la lumière ambiante de la scène.

### 2.3.5 Composante spéculaire

Pour calcul la composante spéculaire, on devra :

- déterminer le point d'intersection entre l'objet et le rayon incident.
- lancer un rayon de ce point vers chacune des sources lumineuses présentes dans la scène pour déterminer si ce rayon est intercepté ou non par un objet.
- pour chaque rayon non intercepté, appliquer la formule correspondante au calcul de la lumière spéculaire.



## **3 Conception**

### **3.1 Objets modélisables**

5 différents objets sont modélisables dans ce programme de lancer de rayon.

#### **3.1.1 La sphère**

La sphère est représentée par son centre et son rayon

#### **3.1.2 L'objet à facettes**

L'objet à facettes est représenté par une liste de facettes. Une facette est un objet triangulaire représenté par 3 points.

#### **3.1.3 Le cube**

Le cube est considéré comme un cas particulier d'objet à facettes. En effet, chacune de ses faces est considéré comme l'assemblage de deux facettes triangulaires.

#### **3.1.4 Le plan**

Le plan est modélisé par 3 points non colinéaires.

#### **3.1.5 La lumière**

Une source lumineuse est caractérisée par sa position dans l'espace et sa couleur.

## 3.2 Les packages du programme

### 3.2.1 interfaceGraphique

L'interface graphique se compose d'une classe principale qui peut être exécutée directement si l'utilisateur veut créer dynamiquement une scène, l'éditer, la visualiser et la sauvegarder.

### 3.2.2 interfaceTexte

Ce package contient les classes nécessaires à l'exécution du programme en lignes de commande.

### 3.2.3 lancerRayon

Il s'agit du package principal du programme. C'est dans ce package que figurent les classes permettant de modéliser les objets, les vues, les scènes ainsi que le calcul du rendu.

### 3.2.4 tests

Ce package rassemble tous les tests JUnit réalisés pour le projet. Chaque classe a été testée individuellement et possède donc ses propres tests JUnits.

### 3.2.5 util

Ce package contient toutes les classes considérées comme des outils pour la réalisation du projet. Ainsi elle contient des classes permettant de modéliser des points, des vecteurs, ou d'enregistrer une image au format PPM.

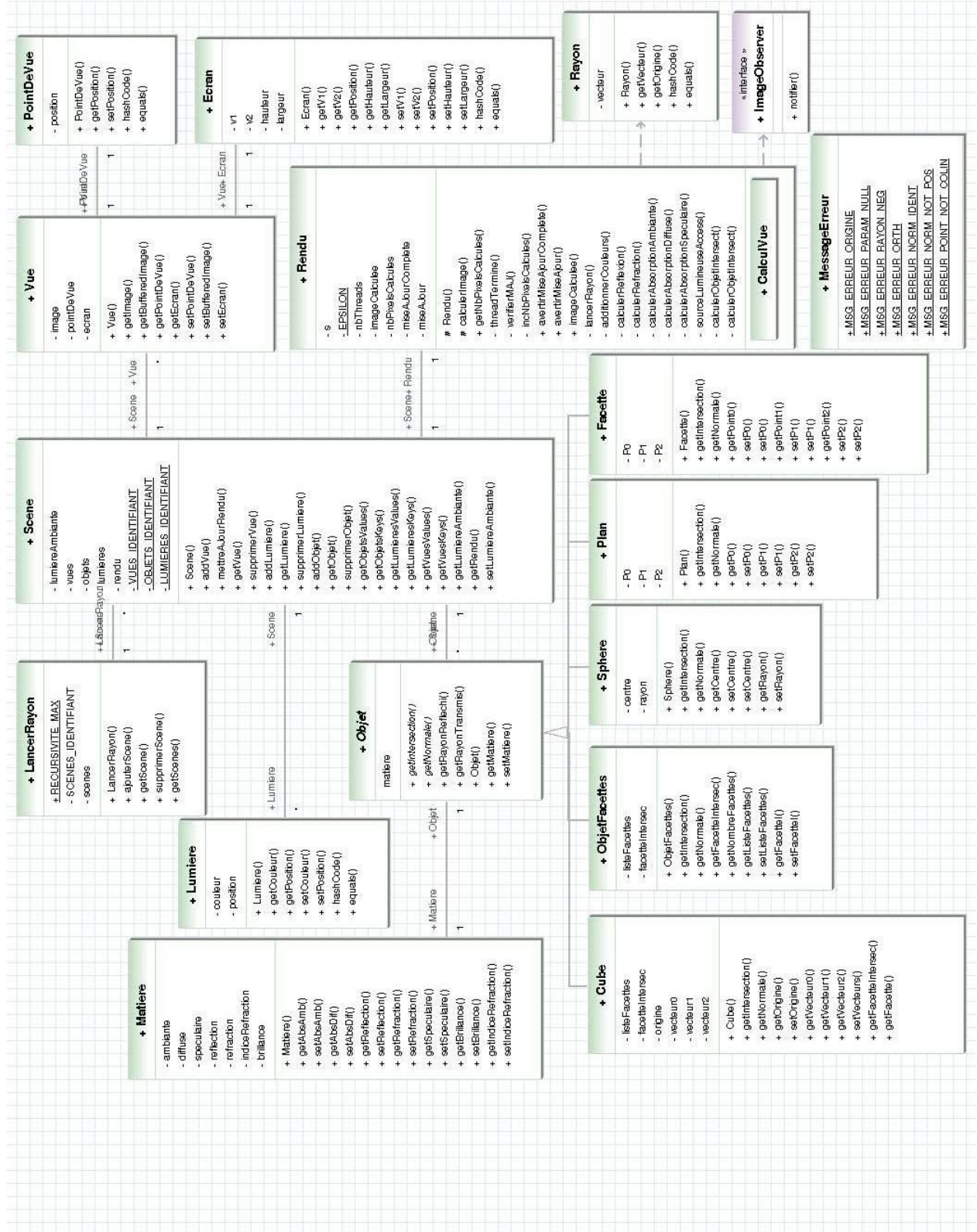
### 3.2.6 xmlParser

Ce package contient les deux classes qui gèrent la lecture et l'enregistrement de fichiers au format XML.

## 3.3 Diagrammes de classes et description des classes

### 3.3.1 Package lancerRayon

Diagramme de classes :



Description textuelle des classes du package :

**Objet** : Classe abstraite permettant de créer un objet 3D. Contient les méthodes permettant de calculer les rayons réfléchis et réfractés par l'objet.

**Plan** : Classe permettant de créer un plan infini à partir de 3 points de l'espace. Contient la méthode permettant de calculer le point d'intersection entre ce plan et un rayon et la méthode permettant de calculer la normale au plan en un point de celui-ci.

**Facette** : Classe permettant de créer une facette triangulaire à partir de 3 points de l'espace, c'est une restriction du plan infini à la surface située entre les points. Contient la méthode permettant de calculer le point d'intersection entre cette facette et un rayon et la méthode permettant de calculer la normale à la facette en un point de celle-ci.

**ObjetFacettes** : Classe permettant de créer un objet à partir d'une liste de facettes. Contient la méthode permettant de calculer le point d'intersection entre cet objet à facettes et un rayon et la méthode permettant de calculer la normale à cet objet en un point de celui-ci.

**Cube** : Classe permettant de créer un cube à partir d'un point de l'espace et de 3 vecteurs deux à deux orthogonaux et de même norme, c'est un objet à facettes dont les 12 facettes sont générées à partir du point et des vecteurs. Contient la méthode permettant de calculer le point d'intersection entre ce cube et un rayon et la méthode permettant de calculer la normale au cube en un point de celui-ci.

**Sphere** : Classe permettant de créer une sphère à partir d'un point de l'espace et d'un entier. Contient la méthode permettant de calculer le point d'intersection entre cette sphère et un rayon et la méthode permettant de calculer la normale à la sphère en un point de celui-ci.

**Ecran** : Classe permettant de modéliser un écran qui est caractérisé par deux vecteurs orthogonaux et une position définissant un des coins de l'écran, ainsi que sa largeur et sa hauteur.

**LancerRayon** : Principale classe du programme. Celle-ci contient la liste des scènes gérées par le programme ainsi qu'un indice de récursivité définissant le nombre de sous-rayons qu'il est possible de lancer pour un pixel donné (récursivité due à la réflexion et la réfraction). Cet indice personnalisable permet de modifier le niveau de détail de l'image final. Cependant, le temps de calcul augmente proportionnellement avec cet indice.

**Lumière** : Classe permettant de modéliser une lumière qui est définie par sa position dans l'espace et une couleur.

**Matiere** : Classe qui caractérise la matière de chaque objet, c'est-à-dire l'ensemble des indices (réfraction, réflexion, absorption...) propres à l'objet.

**PointDeVue** : Classe définissant un point de vue dans l'espace. Un point de vue est la position à partir de laquelle on observe la scène à travers l'écran. Celui-ci est défini par une position dans l'espace.

**Rayon** : Classe permettant de caractériser les rayons lancés. Un rayon est modélisé par un vecteur et une origine. Le vecteur correspondant est automatiquement normalisé.

**Rendu** : Classe de calcul du programme. C'est elle qui calcul chaque pixel de l'image en lançant un rayon à travers chaque pixel de l'écran. Elle calcule toutes les composantes lumineuses, les somme, et associe la couleur correspondant au pixel en question. Elle rafraichie également le composant graphique affichant l'image, et les observateurs souhaitant être avertis de l'avancé du calcul.

**Scene** : Classe qui modélise une scène. Une scène est composée d'un nombre quelconque d'objets, de vues et de lumières. Elle est également caractérisée par une lumière ambiante qui est la couleur par défaut de la scène.

**Vue** : Cette classe permet de modéliser une vue. Chaque vue correspond à une image. Une vue est caractérisée par un écran sur lequel est projetée la scène, et un point de vue depuis lequel la scène est observée.

**ImageObserver** : Cette classe définit un observateur permettant aux classes qui l'implémentent de recevoir des notifications d'avancée du calcul depuis la classe de rendu.

### 3.3.2 Package util

Description textuelle des classes du package :

**Point3D** : Cette classe permet de modéliser un point dans un espace en 3 dimensions. Un point est caractérisé par 3 coordonnées x, y et z.

**Vecteur3D** : Cette classe caractérise un vecteur dans un espace en 3 dimensions. Un vecteur peut être construit soit à partir de 2 Point3D soit à partir de 3 coordonnées. La classe permet d'effectuer un certain nombre de traitement sur les vecteurs, elle peut par exemple calculer le produit scalaire de 2 vecteurs, la norme d'un vecteur, le cosinus de l'angle entre 2 vecteurs, ou encore normaliser un vecteur.

**PPM** : Cette classe permet de traiter le format d'image PPM. Elle possède notamment une méthode permettant de générer un fichier PPM à partir d'une image.

### 3.3.3 Package xmlParser

Description textuelle des classes du package :

**RayonParserReader** : Cette classe définit deux méthodes qui permettent :

- De lire un fichier XML de rendu et de renvoyer la liste des *Vue* définies dans le fichier.
- De lire un fichier XML de scène et de renvoyer un objet *Scene* la modélisant.

**RayonParserWriter** : Cette classe définit deux méthodes qui permettent :

- De créer un fichier XML de rendu à partir d'une liste de *Vue*.
- De créer un fichier XML de scène à partir d'une *Scene*.

### 3.3.4 Package interfaceGraphique

Ce package contient les classes suivantes :

- InterfacePrincipale qui contient une méthode main ainsi que trois boutons permettant de créer une nouvelle scène, de charger une scène et de quitter l'application;
- ChargerScene est un gestionnaire de fichier doté d'un filtre sur les fichier XML qui permet de charger un fichier XML contenant soit un fichier soit une scène;
- InterfaceImage est la fenêtre qui permet l'édition de la scène. Elle se compose d'un arbre récapitulant les différents éléments de la scène regroupés par catégorie (Objet, Source lumineuse, Vue, Lumière ambiante) ainsi que d'une aire de texte où sont décrits les caractéristiques de l'objet sélectionné dans l'arbre. Cette fenêtre comporte cinq boutons et un menu. Les boutons permettent d'ajouter un objet à la scène, de le modifier, de le supprimer, de visualiser la scène et de revenir au menu principal. Le menu permet de d'ajouter des vues grâce à un fichier XML et de sauvegarder soit la scène, soit les vues;
- ChargerVues, SauvegarderScene et SauvegarderVues sont des gestionnaires de fichier permettant d'effectuer les trois actions décrites dans le menu ci-dessus;
- InterfaceObjet est une fenêtre qui permet à l'utilisateur d'ajouter des objets à la scène en rentrant toutes les données physiques nécessaires à sa bonne définition. La fenêtre est divisée en cinq onglets qui permettent d'ajouter respectivement une sphère, un plan infini, un cube, une source lumineuse et une vue. Chaque onglet dispose de deux boutons, un pour ajouter l'objet et un pour annuler. Les erreurs de création d'objet (champ manquant, objet invalide...) sont rapportées à l'utilisateur. Cet interface sert aussi à l'édition des objets en isolant, dans ce cas, l'onglet dans lequel l'objet doit être édité;
- LumiereAmbiante sert à régler la lumière ambiante de la scène en réglant les trois paramètres rouge, vert et bleu. Par défaut, la lumière ambiante est initialisée à (255,255,255);
- l'interface ChoixVues apparaît lorsque l'utilisateur appuie sur le bouton Visualiser. Cette fenêtre n'apparaît seulement si la scène comporte au moins une vue et permet à l'utilisateur de choisir la vue à partir de laquelle il désire voir la scène. Dans le cas où une vue est sélectionnée, la fenêtre affichant l'image apparaît avec un menu permettant de sauvegarder l'image sous format PPM.

### 3.4 Mécanisme de mise à jour des composants d'affichage

Le programme doit afficher les pixels de l'image au fur et à mesure de leur calcul. Pour cela, le composant graphique affichant l'image calculée vient s'enregistrer auprès du rendu associé à la vue qui l'intéresse. A chaque nouveau pixel calculé, le rendu commandera alors le rafraichissement du composant graphique afin que ce dernier affiche le nouveau pixel calculé.

De plus, un mécanisme plus complet de mise à jour à été mis au point. Ainsi toute classe implémentant l'interface *ImageObserver* peut être notifiée de l'avancement du calcul. Pour cela, il lui suffit de venir s'enregistrer auprès du rendu en précisant au bout de combien de pixels le rendu devra lui envoyer une notification.

Ce mécanisme permet par exemple à l'interface en ligne de commandes d'afficher un message à chaque fois que 5% de pixels supplémentaires ont été calculés, afin de faire connaître l'état d'avancement du calcul à l'utilisateur.

### 3.5 MultiThreading de l'application

L'algorithme du lancer de rayon calcule chaque pixel de l'écran de manière totalement indépendante des autres pixels. D'où ce constat : pourquoi ne pas calculer plusieurs pixels en même temps ? La très grande majorité des ordinateurs actuellement commercialisés possèdent effectivement plusieurs cœurs et se prêtent donc parfaitement au traitement de différentes tâches en parallèle.

Nous avons donc choisi de paralléliser les calculs effectués par le moteur de rendu. Voici son fonctionnement :

Le programme commence par obtenir le nombre d'unités de calcul disponibles auprès de l'API java. Soit  $N$  ce nombre, celui-ci déterminera le nombre de threads créés. Soit  $H$  la hauteur de l'image à calculer et  $L$  sa largeur.

Le calcul de l'image est alors réparti entre ces divers threads. Les  $N-1$  premiers threads devront calculer chacun une image de hauteur  $H/N$  et de



largeur L. La division n'étant pas forcément juste, le dernier thread N devra calculer les pixels restants, soit une image de hauteur  $H/N + H\%N$ .

Ainsi chaque thread se voit affecté le calcul d'une zone différente de l'image.

L'utilisation des threads fait cependant apparaitre plusieurs problèmes :

- Plusieurs threads peuvent accéder en même temps à une zone critique du logiciel, notamment celle qui vérifie le nombre de pixels calculés pour envoyer éventuellement des notifications aux observateurs. Ce problème est réglé en rendant les méthodes critiques *synchronized*.
- Le thread courant poursuit son exécution et n'attend pas que les N threads de calcul aient terminés leur tâche, ce qui peut poser problème dans la suite du programme. Ce problème est réglé en mettant en pause le thread courant grâce à la méthode *wait()* de l'API java. Une fois que tous les threads de calcul ont terminés leur tâche, le dernier thread réveille alors le thread courant grâce la méthode *notify()* de l'API java. Le thread courant peut alors poursuivre son exécution.

### 3.6 Différences par rapport à la phase d'analyse

Très peu de différences séparent le logiciel actuel de ce qui avait été prévu lors de la phase d'analyse. Le diagramme de classe d'analyse a en effet été entièrement respecté, ce qui prouve que celui-ci était fiable et particulièrement réaliste.

Quelques modifications mineures ont cependant été apportées :

- Par commodité, nous avons créé une classe *Matiere* qui permet d'alléger la classe *Objet* en stockant l'ensemble des indices liés à un objet.
- Nous avons ajouté une classe d'observateur afin que les classes qui le souhaitent puissent être notifiées de l'avancement du calcul de l'image, afin par exemple d'afficher une barre de progression ou l'état en pourcentage du calcul de l'image.

- Une modification du sujet nous a obligé à ajouter une classe *ObjetFacette* permettant de définir un objet quelconque composé de facettes. Par la même occasion, nous avons dû faire hériter la classe *Facette* de la classe *Objet*.

## 4 Interface utilisateur

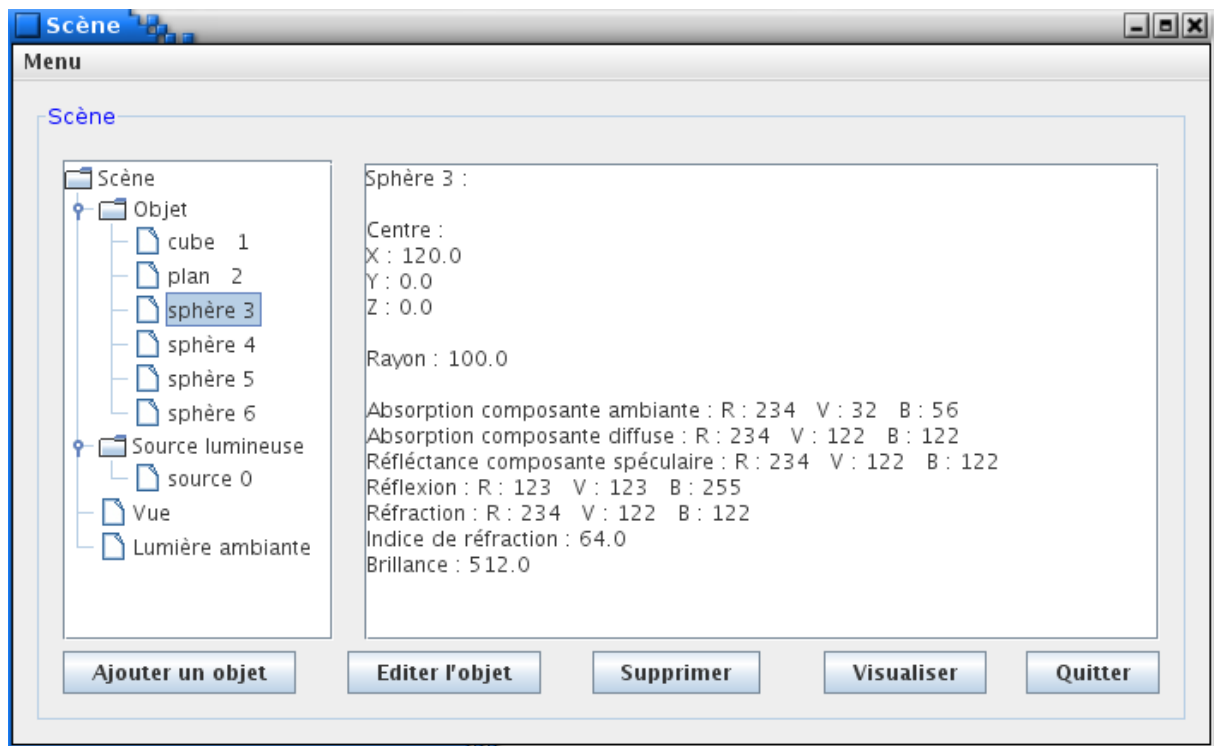
### 4.1 Interface graphique

L'interface graphique est composée de plusieurs fenêtré qui interagissent entre elles en essayant de garantir un maximum de stabilité du point de vue de l'utilisateur. Toutefois, certains détails n'ont pas pu être traités. Ils auraient pu être traités si nous disposions de plus de temps.

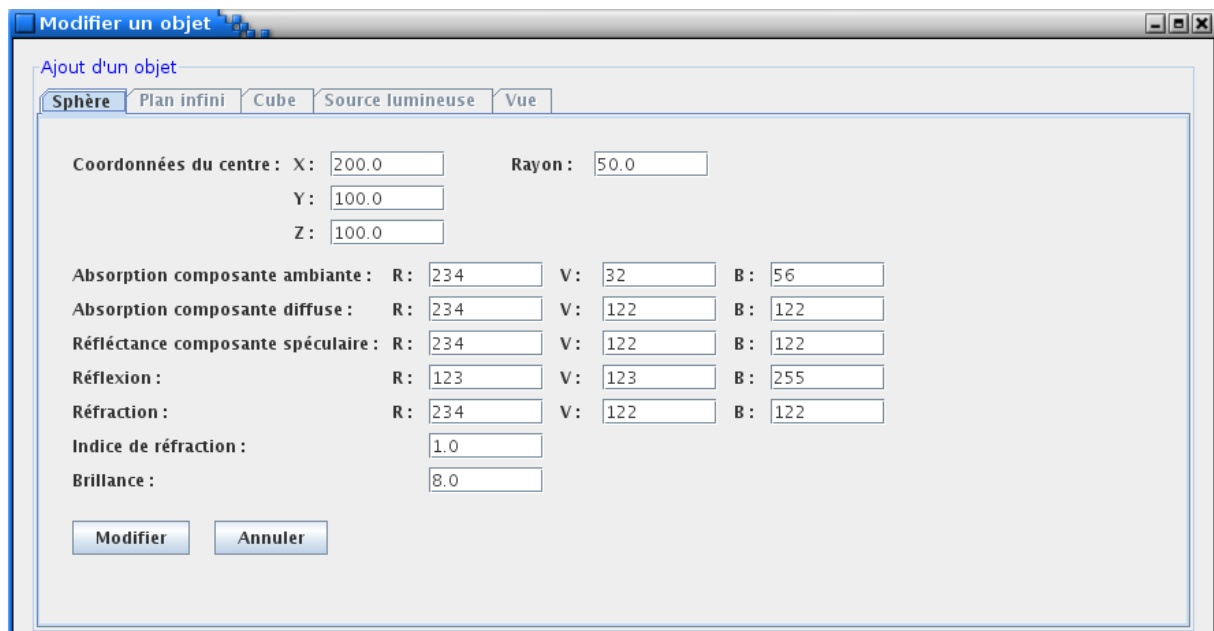
Plaçons-nous du point de vue de l'utilisateur. La première fenêtré à apparaître est l'interface principale. A partir d'ici, l'utilisateur peut choisir de créer une nouvelle scène, de charger une scène existante ou de quitter l'application. Aucun de ces choix ne produit d'exception ou d'erreur. Cependant le code peut être utilisé de façon à générer plusieurs scènes en même temps via la classe LancerRayon qui gère une liste de scène. Cette option n'a toutefois pas pu être implantée dans l'interface graphique.

Les gestionnaires de fichiers permettent de sauvegarder ou de charger tous les fichiers nécessaires au projet. Les opérations simples de chargement et de sauvegarde fonctionnent. La touche annuler peut créer une erreur dans le sens où le fait de sélectionner un fichier et appuyer sur annuler a le même effet que d'appuyer sur ouvrir. Cela est du au fait que la classe JFileChooser n'est pas facile d'accès surtout dans le cas où l'on veut faire un ActionPerformed qui ne peut pas être ciblé sur chaque bouton.

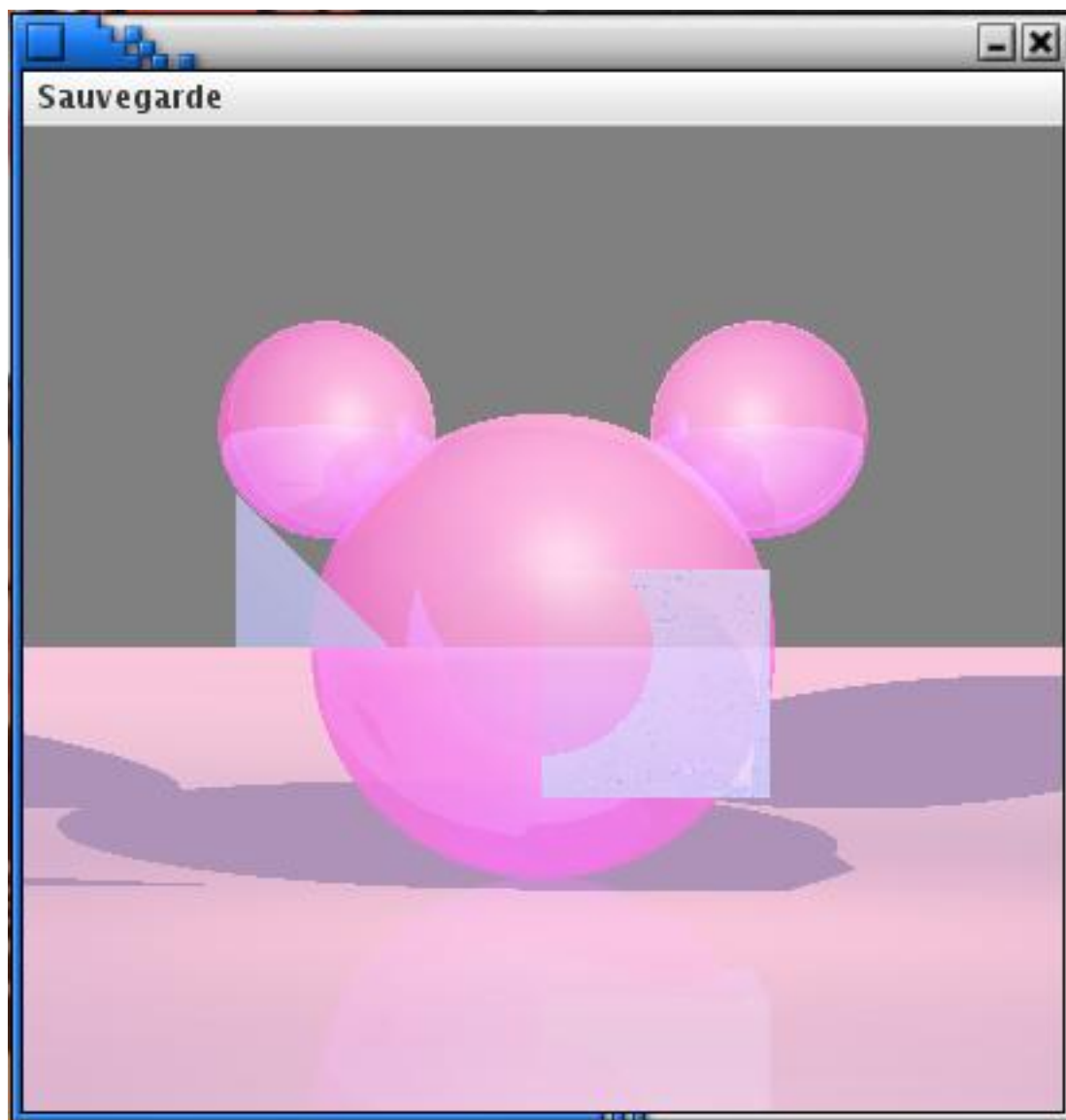
L'interface d'édition des scènes permet d'effectuer toutes les actions nécessaires sur les scènes au travers de ses objets, c'est-à-dire l'ajout, l'édition et la suppression. Cette fenêtré ne pose aucun problème connu.



L'interface d'édition des objets permet de gérer la création et l'édition des objets proprement dit. Cette fenêtre ne génère également pas d'erreurs.



Fenêtre du rendu final :



## 4.2 Interface en ligne de commande

Une interface en ligne de commande non interactive à été réalisée. Celle-ci prend en paramètre deux arguments obligatoires et un argument optionnel.

Utilisation :

```
java InterfaceTexte nomFichierScene nomFichierRendu [nomFichierSortie]
```

- *nomFichierScene* : l'adresse relative ou absolue du fichier contenant la description de la scène.

- *nomFichierRendu* : l'adresse relative ou absolue du fichier contenant la description des vues.

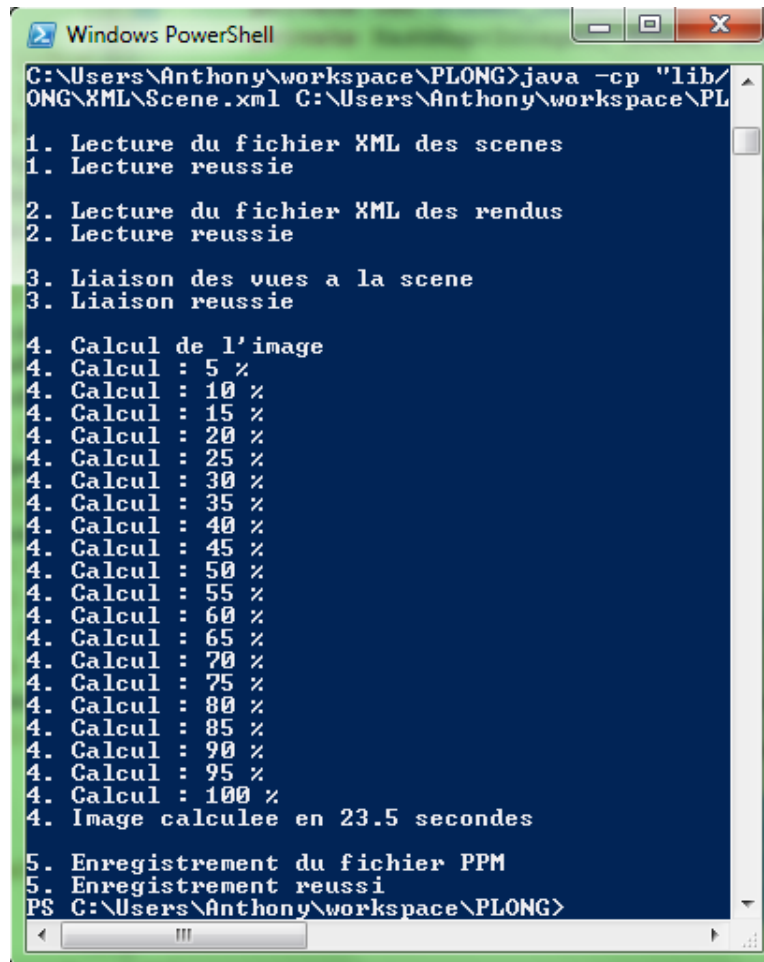
- *nomFichierSortie* : l'adresse relative ou absolue du fichier dans lequel enregistrer l'image calculée. Si cet argument n'est pas indiqué, le programme enregistre automatiquement l'image dans un fichier output.ppm.

A chaque étape de l'exécution, le programme rend compte à l'utilisateur de l'avancée du traitement.

De plus, le calcul de l'image pouvant nécessiter un certain temps, le programme affiche la progression du calcul de l'image tous les 5 % afin que l'utilisateur puisse connaître approximativement le temps de calcul restant, et puisse être assuré que le programme n'est pas crashé.

Une fois le calcul de l'image terminé, le programme indique en secondes le temps qui à été nécessaire au calcul de l'image (temps indiqué à  $10^{-3}$  secondes près).

## Exemple d'exécution en ligne de commande



```
C:\Users\Anthony\workspace\PLONG>java -cp "lib/ONG\XML\Scene.xml C:\Users\Anthony\workspace\PLONG>

1. Lecture du fichier XML des scenes
1. Lecture reussie

2. Lecture du fichier XML des rendus
2. Lecture reussie

3. Liaison des vues a la scene
3. Liaison reussie

4. Calcul de l'image
4. Calcul : 5 %
4. Calcul : 10 %
4. Calcul : 15 %
4. Calcul : 20 %
4. Calcul : 25 %
4. Calcul : 30 %
4. Calcul : 35 %
4. Calcul : 40 %
4. Calcul : 45 %
4. Calcul : 50 %
4. Calcul : 55 %
4. Calcul : 60 %
4. Calcul : 65 %
4. Calcul : 70 %
4. Calcul : 75 %
4. Calcul : 80 %
4. Calcul : 85 %
4. Calcul : 90 %
4. Calcul : 95 %
4. Calcul : 100 %
4. Image calculee en 23.5 secondes

5. Enregistrement du fichier PPM
5. Enregistrement reussi
PS C:\Users\Anthony\workspace\PLONG>
```

## 5 Tests

Tous les tests ont été réalisés en Junit. Chaque classe des packages LancerRayon et util ont été testées de manière unitaires

Le but des tests est de vérifier que les classes fonctionnent correctement. Au cours du développement chaque ajout de méthode engendrait l'obligation de créer une méthode de test correspondante.

Ces différents tests nous ont été utiles pour détecter des erreurs et trouver des nouveaux cas « problématiques » nécessitant de lever une exception.

Réalisés au fur et à mesure de l'avancement du projet ils nous ont permis d'avoir un contrôle sur le développement : les personnes codant les classes étant différentes de celles réalisant les tests, nous nous imposons ainsi un « double contrôle » sur notre code.

Il est à noter :

- que certains tests ont nécessité des « rébarbatifs » calculs à la main

- les « setUp » nécessitaient parfois l'utilisation de 4 ou 5 autres classes, d'où la nécessité de les avoir testées auparavant



## 6 Organisation du projet

### 6.1 Environnement de développement

Afin de mener à bien ce projet qui s'effectue en groupe, nous avons choisi de mettre en place un environnement de développement commun, et des outils nous permettant d'organiser et de synchroniser au mieux notre travail.

Cet environnement se compose de :

- Eclipse version 3.4
- plugin subclipse permettant de synchroniser les sources du programme vers un repository SVN via eclipse.
- un serveur SVN privé permettant de synchroniser les sources du groupe.
- un système de gestion de projet en ligne : trac
- NetBeans 6.5.1 pour mettre au point l'interface graphique
- JUnit pour la réalisation des tests unitaires

### 6.2 Répartition des tâches

Nous avons accordé une importance toute particulière aux tests de l'application. Ainsi, tout au long du projet, deux membres de l'équipe ont eu pour tâche de créer des classes de test JUnit pour chacune des classes des packages *lancerRayon* et *util*. Grâce à leur travail, les bugs présents ont pu être corrigés, et le fonctionnement des classes validé.

Un autre membre de l'équipe c'est quand à lui consacré à la réalisation de l'interface graphique tout au long du projet.

Les deux membres restants ont quand à eux principalement développé les classes métiers. Ainsi un membre c'est occupé de créer les classes permettant de modéliser les objets, l'autre membre à créé les autres classes du programme, notamment le rendu.

Grâce à cette répartition des tâches, chacun c'est spécialisé dans un domaine particulier du programme (tests, interface graphique, classes métiers), devenant ainsi plus efficace dans son travail. Ainsi, pour chaque problème rencontré, une notification était envoyée à la personne spécialisée dans le domaine en cause.

## 6.3 Travail réalisé par chaque membre

### 6.3.1 Benoît

Mon travail pour le projet JAVA à été essentiellement de réaliser les tests Junit des différentes classes. J'ai ainsi eu à réaliser les tests des classes indiquées ci-dessous :

- Ecran
- Point3D
- Rayon
- Lancer Rayon
- Lumière
- Point de Vue
- Vue
- Facette

Suite à ces tests, j'ai parfois dû modifier la classe testée, ou notifier la personne qui en avait la charge afin de corriger les erreurs éventuelles.

J'ai également réalisé le `ParserWriter` qui permet d'écrire une Scène ou un Rendu sous forme d'un arbre XML.

### 6.3.2 Anthony

Mon travail pour ce projet aura été relativement transversal.

Ainsi, au début du développement, j'ai d'abord eu pour tâche la réalisation de l'ensemble des classes des package *lancerRayon* et *util*, hormis les classes « objets » (Objet, Cube, Facette ...) qui étaient à la charge de Samuel. J'ai notamment consacré un temps important pour la réalisation et le débogage de la classe de rendu.

Une fois les diverses classes conçues, j'ai « assemblé les briques » afin de réaliser le logiciel final, et j'ai alors entamé une longue phase de débogage afin de m'assurer que le cœur du logiciel fonctionne correctement.

Puis je me suis attelé à la réalisation d'autres classes : la classe permettant de charger des fichiers XML, le débogage de la classe réalisée par Benoît enregistreur des fichiers XML, réalisation de la classe qui convertit une image en fichier PPM, réalisation de l'interface en ligne de commandes.

Une fois ces étapes réalisées, je me suis alors fixé comme objectif d'améliorer le programme. J'ai ainsi mis en place un calcul parallèle de l'image grâce aux Threads, et un mécanisme d'événement permettant par exemple à l'interface texte de recevoir une notification à chaque fois que 5% d'image supplémentaires ont été calculés. Enfin, j'ai apporté quelques optimisations au rendu afin d'accélérer son exécution.

Puis, au final, j'ai rédigé des scripts .sh permettant de compiler l'ensemble du projet.

### 6.3.3 Maxime

Après la 1ère phase d'analyse du sujet réalisée tous ensemble, j'ai pu participer à la fois à la création des premières classes des objets, mais surtout à une grande partie des tests JUnit (cube, ecran, matière, plan, vecteur3D, vue...) qui ont permis de détecter certaines erreurs. Sans participer à la réalisation des autres parties plus délicates, confiée aux plus expérimentés du groupe, (interface graphique, partie xml...), je m'y suis quand même régulièrement intéressé et constaté l'avancement du projet.

### 6.3.4 Julien

Mon travail sur ce projet a été très spécialisé puisque je me suis occupé pratiquement entièrement de la réalisation de l'interface graphique.

La première partie du développement du projet fut assez indépendante du reste du groupe puisqu'elle consistait à créer un environnement graphique composé de plusieurs fenêtres sachant communiquer entre elles. Cette étape a pu être réalisée grâce à l'aide de la bibliothèque Java Swing et au logiciel Netbeans. La deuxième partie fut plus compliquée puisqu'elle a consisté à raccrocher l'interface graphique aux autres classes. Malgré quelques problèmes de synchronisation, cette étape a permis à tous les membres du groupe de corriger leurs erreurs, de mon côté pour pouvoir

bien gérer tous les paramètres des scènes, de l'autre côté du groupe pour pouvoir s'adapter à n'importe quel utilisateur. Bien que je sois conscient qu'il reste quelques erreurs et beaucoup de programmation « maladroite ». j'espère avoir fourni une interface acceptable pour ce premier essai en condition de projet.

### 6.3.5 Samuel

Mon travail pour ce projet a été de réaliser les classes définissant les différents objets affichables dans la scène, qui sont les classes :

- Objet
- Cube
- Facette
- ObjetFacettes
- Plan
- Sphère

J'ai également réalisé les tests jUnit de ces classes.

Ces classes ont du être modifiées plusieurs fois afin de suivre les évolutions de l'application causés par des erreurs d'interprétation du sujet ou le changement de certains choix d'implémentation.

## **7 Conclusion et perspectives**

### **7.1 Conclusions personnelles**

#### **7.1.1 Benoît**

J'ai été très satisfait de ce travail en équipe. Il est vrai que je ne suis pas celui qui en aura réalisé le plus mais je savais cela depuis de début étant donné le niveau de certains de mes collègues qui était bien supérieur au mien en Java.

Cependant le travail que j'ai eu à produire m'a permis d'avoir pleinement connaissance de l'ensemble du projet. En effet les tests que j'ai réalisés portaient sur une grande partie des classes, puis ParserWriter m'a permis d'aborder le reste du projet (sauf l'interface). La répartition du travail a donc de ce point de vue été correctement réalisée. Je remercie Anthony d'avoir dirigé notre groupe au cours de ces semaines, les différentes étapes du travail ayant toutes abouties dans les timings fixés.

Ce fut donc pour moi une expérience de travail en groupe enrichissante, sur la manière de développer à plusieurs mais aussi sur l'aide que l'on peut se donner et le gain de productivité qui en résulte.

#### **7.1.2 Anthony**

Premier travail en équipe sur un projet de programmation à l'ENSEEIH. Certains auront beaucoup travaillé, d'autres moins, mais tous auront participé à la réalisation de ce projet.

Au final ce projet aura été une bonne occasion de découvrir le travail en équipe et les projets de programmation pour ceux qui n'en n'avaient jamais réalisés.

Le travail aura finalement été accompli avec succès, même si un peu de temps supplémentaire n'aurait pas été de refus pour terminer correctement certaines tâches.

### 7.1.3 Maxime

Travailler en groupe a été bien plus efficace qu'un projet seul. Cela a également été source de motivation pour régulièrement avancer le travail. Cependant, le fait de ne pas forcément toucher à l'ensemble des parties du projet n'est pas la meilleure manière d'approfondir nos connaissances sur le langage utilisé.

Malgré tout, cela a été une bonne expérience de travail en groupe, et le fait d'être à peu près parvenu à achever le projet est un bon résultat.

### 7.1.4 Julien

Ce projet a été une expérience très enrichissante. Cette première immersion dans la programmation technologie m'a beaucoup appris d'autant plus que je me suis occupé des relations entre l'utilisateur et le programme. J'ai tenté d'être le plus exigeant possible car toute erreur de l'interface graphique peut immédiatement se voir à l'écran.

Malgré tout, j'ai rencontré de multiples difficultés, notamment avec l'utilisation des JFileChooser. De plus, garder un contrôle constant sur les différentes fenêtres fut assez compliqué.

### 7.1.5 Samuel

Ce travail en équipe a été très agréable avec une bonne ambiance dans l'équipe car tout le monde a fait preuve de bonne volonté. De plus le sujet ayant une finalité concrète, l'avancement du projet était visible tout au long du développement.

Cette première expérience de programmation en groupe a été très bénéfique car elle m'a permis de découvrir les méthodes de travail en équipe et d'approfondir ma connaissance de java et des test jUnit.

## 7.2 Conclusion générale et perspectives

Au final, le projet aura été réalisé dans son intégralité, le programme conçu est fonctionnel et répond au cahier des charges. Malgré tout, certaines tâches comme l'interface graphique par exemple n'ont pas pu être terminées à temps. Celle-ci est donc fonctionnelle, mais n'intègre pas toutes les fonctionnalités prévues initialement et n'exploite pas toutes les possibilités offertes par le noyau du logiciel.

Du temps supplémentaire nous aurait permis de finaliser l'interface graphique avec toutes les fonctionnalités prévues, de tester davantage l'application finale, ou de modifier le multithreading en mettant par exemple en œuvre une file d'attente permettant d'obtenir les pixels à calculer, plutôt que d'attribuer une zone entière de l'image à chaque thread.

Enfin, il serait également possible d'inclure davantage de formes géométriques par défaut, ou d'introduire des textures pour la surface des objets.