# Rapport projet de Ray Tracing

Guillaume LEMONNIER
Harrisson KOBYLT
Antoine LEMAITRE
Logan VIVIEN

L2 info, 2021-2022
Groupe 4A

28 janvier 2023

# Table des matières

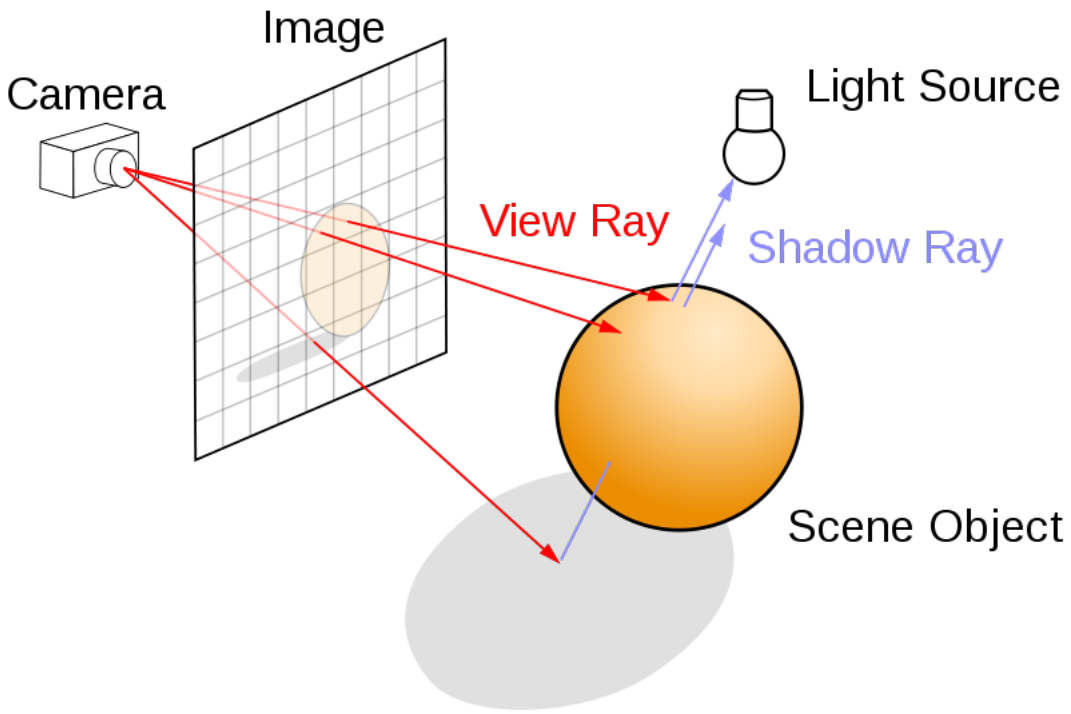
# 1 Introduction

Pour la réalisation du projet de Conception Logiciel Avancée nous avions à choisir un sujet parmi une liste imposée, ainsi nous avons décidé de travailler sur le projet de rendu 3D par lancer de rayons en java. Nous avons choisi de prendre ce sujet car c'était celui pour lequel nous avions une vision plus claire des objectifs à réaliser, ce qui rendrait le développement plus agréable.

## 1.1 Objectifs du projet à réaliser

Le projet que nous avons choisi est donc celui du rendu 3D par lancer de rayons. L'objectif de celui-ci est donc de créer un moteur de rendu 3D permettant à partir d'une scène donnée dans un fichier, de réaliser les différentes étapes afin d'obtenir une image de celle - ci.
Parmi celles-ci, on peut compter le parsing du fichier au format .pov, la création d'une fenêtre d'affichage, le lancer de rayons sur la scène décrite, et enfin la génération de l'image résultant de cela.

Les objectifs étaient donc de réaliser un Parser qui créerait les objets de la scène, la caméra ainsi que la lumière entre autres, à partir du fichier.pov.
De plus, nous devions réaliser un algorithme de lancer de rayons gérant les ombres et les différentes positions, orientations de la caméra.
De même nous avions à implémenter une interface graphique qui afficherait le résultat de l'image et permettrait à l'utilisateur de se déplacer dans la scène et de pouvoir changer l'orientation de son point de vue.
Dans l'idée, nous voulions obtenir un résultat semblable au schéma suivant :




Nous avons réalisé la plupart de ces fonctionnalités qui seront mieux décrites ci-dessous.

## 1.2 Organisation du rapport

Notre plan est organisé de la manière suivante. Tout d'abord nous allons parler de notre organisation lors du projet, c'est à dire comment nous avons divisé le travail afin que nous ne travaillions pas sur les mêmes choses, ainsi que la façon dont on se concertait chaque semaine pour se tenir au courant de l'avancement.
Ensuite nous expliquerons comment notre application est construite, nous permettant d'aborder le thème de l'architecture du projet. Nous expliquerons alors les fonctionnalités proposées, comme le fait de pouvoir se déplacer dans une scène ainsi que la gestion des ombres sur le rendu de l'image et comment le code est organisé, comment il fonctionne. Pour finir, nous conclurons en réalisant un bilan du développement de l'application et en expliquant les améliorations réalisables.

# 2 Organisation du développement

Dans cette partie nous allons décrire comment nous avons travaillé autour de ce projet.

## 2.1 Scénario d'utilisation et découpage du projet

D'abord, nous nous penchâmes sur un scénario d'utilisation lequel est le suivant : dans un premier temps l'utilisateur clic sur notre programme, là s'ouvre une fenêtre avec une image par défaut. Puis l'utilisateur aura un bouton qui lui permettrait d'ouvrir un fichier décrivant une scène, l'image de la scène s'afficherait sur cette fenêtre et l'utilisateur pourrait naviguer autour de l'objet.

De ce scénario, nous avons coupé le projet en quatre parties distinctes : la première, celle qui portera sur le parsing du fichier pov, la seconde sur la création de l'image, la troisième sur l'affichage de cette image et de la navigation dans la scène et la dernière qui sera la création des classes des objets et rayons 3D.

## 2.2 Répartition des tâches

Ainsi, après avoir divisé nous avons donné une partie à réaliser à chacun selon ses préférences. Alors, Guillaume prit la partie de la création des Objets 3D et des rayons, Antoine s'occupa de la création de l'image à partir d'une scène donnée, Logan a fabriqué le Parser et Harrisson a géré l'affichage de la fenêtre avec l'image résultat ainsi que les déplacements dans la scène.
Nous obtenions alors un tableau tel que celui-ci :

| | |
|---|---|
| Guillaume LEMONNIER | Model 3D et Rayons |
| Antoine LEMAITRE | Calcul et création d'images |
| Harrisson KOBYLT | Interface Graphique |
| Logan VIVIEN | Parser |
| Erin LE DREAU | Interface Grahique |

Nous avions alors une partie chacun, ce qui nous paraissait équitable. D'ailleurs si l'un de nous finissait plus tôt sa partie il viendrait en appui à un autre camarade afin que l'on avance ensemble plus efficacement.
A noter que, si Erin et Harrisson se sont retrouvés sur la même partie c'est parce qu' Erin n'avait à l'origine pas de groupe.

Elle nous a rejoint au bout du troisième cours après la répartition des tâches. De ce fait, nous lui avons proposé d'assister Harrisson dans l'interface graphique.
Plus tard elle quitta le groupe pour des raisons professionnelles sa participation est donc très réduite.

# 3 Architecture du projet

Pour ce projet, nous avons cherché à produire une architecture assez simple. Par conséquent, nous avons décidé de mettre le main à la source du projet et de créer des packages en fonction du domaine de la tâche qu'ils effectuent.Ainsi, il existe un package fenêtre qui gère l'affichage graphique et les raccourcis claviers, un package scene qui est utilisé pour définir la classe caméra et celle nécessaire à la génération de l'image de cette dernière et la classe scene, un package object3D qui définis les classes liées aux volumes et aux calculs de rayons, un package parser qui gère la lecture de fichiers et les exceptions qui pourraient arriver et enfin un package shadow qui contient la lumière et les matériaux.
On obtient donc ce diagramme de classe [1] :

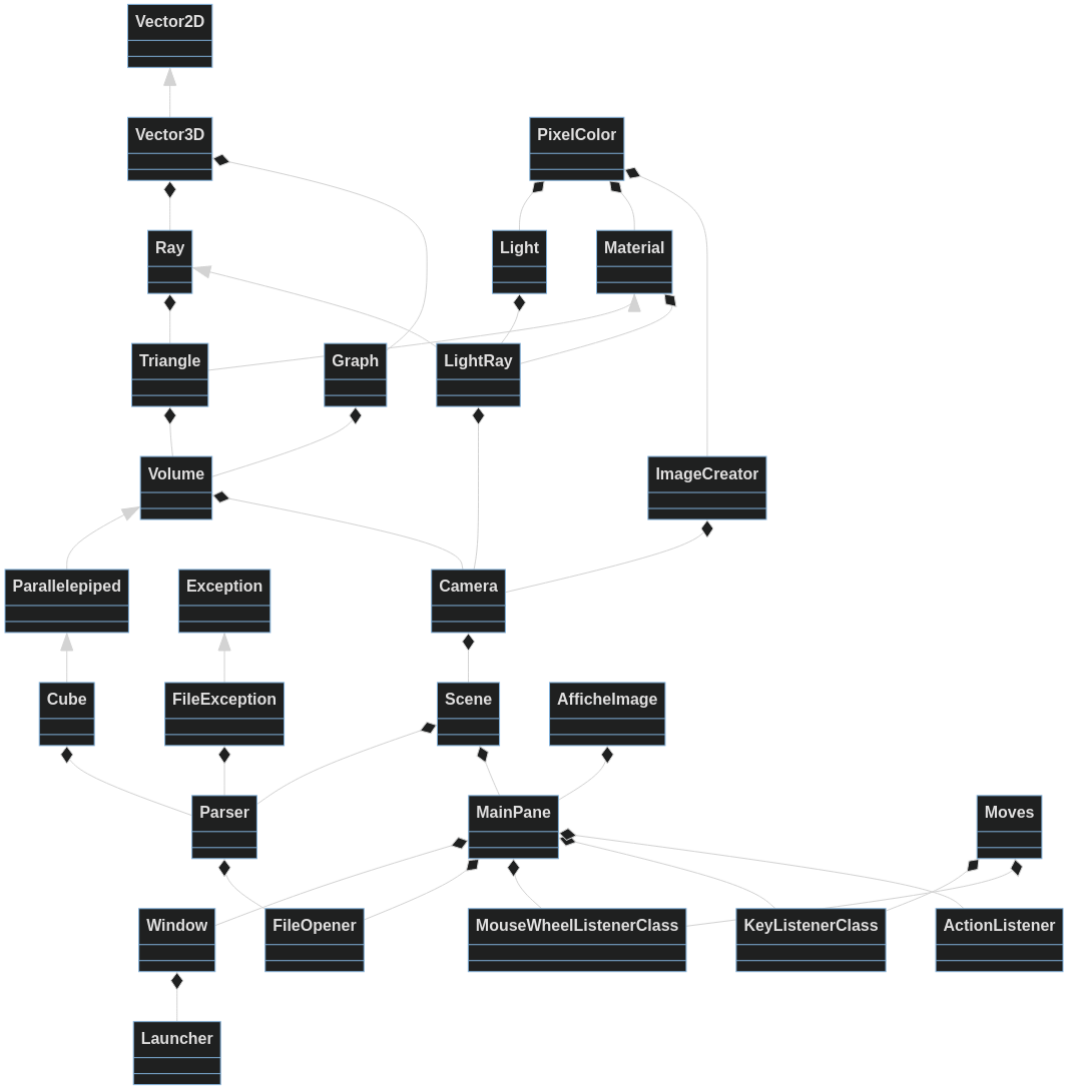



FIGURE 1 – Diagramme Classe

1. Pour plus de précisions, veuillez vous référer à l'annexe A

# 4 Mode d'emploi

Afin de pouvoir lancer l'application, vous allez devoir compiler à l'aide de la commande "make compile". Puis pour exécuter le programme, utiliser la commande " make run ", vous accéderez alors à la fenêtre principale. Pour la fermer il faut simplement cliquer sur la croix en haut à droite.

## 4.1 Normes d'écriture des fichiers Pov

Un fichier pov doit respecter certaines règles afin que le Parser accepte de le lire. Pour commencer, tout le fichier doit être entre accolades, donc la première et la dernière ligne sont les accolades. Ensuite,les paramètres doivent être séparés par des / afin que le ScanFile sache différencier et utiliser les éléments. L'ordre des paramètres ont une importance car les instances sont créées selon un ordre bien précis.

## 4.2 Ouvrir son fichier pov

Pour ouvrir un fichier pov, il suffit de cliquer sur le bouton "Ouvrir" de la fenêtre principale, celui-ci ouvrira un explorateur de fichier qui vous permettra de sélectionner le fichier voulu.
Une fois sélectionné la scène résultant du fichier apparaît sur la fenêtre principale.

## 4.3 Déplacement et rotation

Les raccourcis pour se déplacer sont :

— La molette vers l'avant / vers l'arrière permettant d'incrémenter ou décrémenter l'axe Z de la caméra.
— Les touches Z / S qui permettent d'incrémenter ou de décrémenter l'axe Y de la caméra.
— Les touches Q / D qui permettent d'incrémenter ou de décrémenter l'axe X de la caméra.

Les raccourcis pour effectuer des rotations de regard sont :

— Les touches A / E qui permettent d'incrémenter ou de décrémenter la rotation de la caméra autour de l'axe Y.

# 5 Fonctionnalités de l'application

Ici nous expliquerons, les fonctionnalités implémentées par chacun d'entre nous dans les différentes parties. Nous suivrons l'ordre suivant : de la partie la plus éloignée de l'utilisateur à la plus proche.

## 5.1 Modèle 3D et Rayon

### 5.1.1 Ray

Guillaume, ayant déjà une bonne idée de ce qu'il devait faire, a commencé les Vector2D, Vector3D et Ray qu'il a fini pendant la première séance sans trop de difficultés. Vector2D et Vector3D sont uniquement des classes pour contenir des adresses X, Y, Z, afin de créer des Points qui serviront de bases à une grande partie du programme. En effet, ils serviront pour créer les Volumes (créer l'adresse de leurs sommets et leur position dans l'espace), les rayons (qui sont une combinaison de deux Vector3D pour créer une droite avec une direction).

### 5.1.2 Models 3D

Lors de la deuxième séance il a commencé à créer un cube en 3D avec juste la position de ses sommets comme référence. Un cube peut être réduit à une liste de ses sommets et donc avoir uniquement une liste de Vector3D. Cependant avec cette façon il était possible d'avoir une mauvaise représentation du cube. En effet il était possible de connecter des sommets entre eux sans que cela soit possible pour un cube dans la réalité. Guillaume s'est alors posé une question, "Comment obliger les sommets à se connecter de manière logique entre eux ?". Ce à quoi il s'est dit que créer des Triangles pour représenter les faces allait aider Antoine pour les collisions et allait obliger les sommets à se lier entre eux logiquement. Il entreprit donc de créer la classe Triangle. Avec la classe Triangle, quand il créait un Volume il se retrouvait avec des triangles qui se superposaient. Il chercha donc un algorithme pour sélectionner les triangles à utiliser pour éviter ceux qui se superposent. Il créa pour cela une classe Graph servant à référencer la référence des triangles voisins du sommet sélectionné. Avec tout cela il pouvait alors créer l'algorithme suivant :

**Algorithme 1 :** MAKETRIANGLE Créer une liste de Triangle sans dupplication de Triangle

**Entrées :**
**Output :**

```
1  graphs ← Liste_Graph
2  triangles ← Liste_Triangle
3  centres ← Liste_Vector3D
4  exist ← {0, 1}
5  pour chaque g1 ← this.graphs faire
6      exist = 0
7      pour chaque g2 ← graphs faire
8          si g1.superpose(g2) alors
9              exist = 1
10             break
11         fin
12     fin
13     si n' exist pas alors
14         graphs.add(g1)
15         centres.add(g1.getCenter())
16         pour j ← 0; j < g1.getNeightbourSize(); j++ faire
17             this.listTriangle.add(newTriangle(
18             g.getCenter(),
19             g.getNeightbour(j%g.getNeightbourSize()),
20             g.getNeightbour((j + 1)%g.getNeightbourSize());
21             ))
22         fin
23     fin
24 fin
```

Grâce à cet algorithme nous nous retrouvons dans le cas où juste deux triangles sont gardés pour faire une face d'un cube. Ainsi si nous prenons l'exemple d'un cube nous nous retrouvons avec juste 12 Triangles à afficher au lieu de 24 avant l'algorithme. Chaque autres Volumes spécifiques comme Parallelogramme ou Cube hériterons de cette classe afin de pouvoir faire une liste de tous les volumes existant.

## 5.2 Parsing

Le Parser est le programme qui, à partir d'un fichier texte, renvoie une liste de volumes ainsi que les paramètres de la caméra. L'objectif est donc de pouvoir extraire d'un fichier sous format .pov les paramètres voulus et créer des volumes avec ces derniers.

### 5.2.1 Les constructeurs

Les constructeurs ne sont là que pour attribuer le fichier souhaité à une variable qui sera utilisé dans les méthodes.
Le premier constructeur prend donc en paramètre le nom du fichier, le second quant à lui est un constructeur vide qui met un fichier par défaut préalablement créé dans le Parser.

### 5.2.2 ScanFile

Cette méthode est celle qui permet d'obtenir les paramètres contenus dans les lignes de texte. L'extraction des éléments a été réalisée grâce à la classe utilitaire java.util.scanner. Cette dernière est simple d'utilisation est m'a permis de pouvoir définir mon Parser.
Afin de vérifier que le fichier et son contenu soient valides, le code a été inclus dans un try, qui envoie une exception si il rencontre une erreur lors de l'exécution grâce à un catch situé juste après le try, cela permet d'éviter que n'importe quel fichier soit mis dans le Parser.
La variable donnée par le constructeur permet donc à Scanfile d'ouvrir le fichier localisé dans un dossier nommé pov, où sont situés tout les fichiers pov.
Une fois le fichier ouvert, chacune de ses lignes est ajouté dans une ArrayList de String, puis une fois que la liste est dépourvue de son premier et dernier élément, qui n'étaient que des accolades vérifiant la conformité du fichier, chacune des lignes sont à leur tour scannées et délimitées cette fois-ci par le signe "/".
Nous nous retrouvons donc avec une liste de tableau de String nommé listParam, qui contient chaque élément singulier du fichier. Cette liste de tableau sera lue par la seconde méthode afin de retourner la scène à afficher.

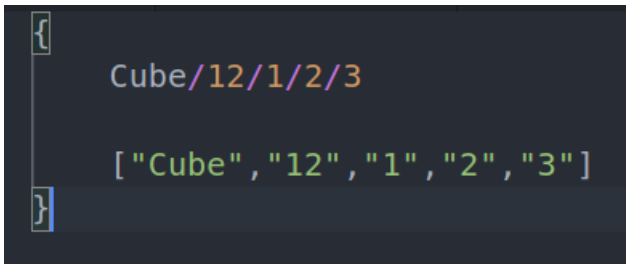



FIGURE 2 – Exemple de la transformation d'une ligne en ArrayList de String

### 5.2.3 listeString2Scene

Vu que le but du Parser est de renvoyer une liste de volumes ainsi qu'une instance de caméra, il a fallu créer une classe abstraite qui permette de contenir les objets de classe volumes ainsi que l'objet gérant la caméra dans un même ArrayList. Étant parti de base sur l'idée de créer une classe abstraite, une solution plus convenante a été trouvée : La classe Scene.
Une Scene est une classe composé d'un ArrayList de volumes, d'une instance de Camera ainsi que d'une instance de lumière. C'est donc une instance de cette classe que la méthode ScanFile renvoie. Par défaut, une instance de Scene possède une caméra et une lumière égales à null et sa liste de volumes est vide, l'utilité de la méthode listeString2Scene est donc de créer et paramétrer une instance de Scene selon les éléments de listParam.
Pour cela nous allons utiliser la même méthode que pour ScanFile c'est-à-dire deux boucles for afin de parcourir chaque ligne de listParam et chaque élément par ligne.
Peu importe la ligne, l'élément le plus important est le premier (c'est-à-dire lineParam.get[i](0) où i est l'indice de la ligne) car c'est dans cet élément qu'est marqué le nom du "paramètre". C'est en lisant cet élément que l'on prendra connaissance de la classe à instancier pour que le programme sache ce à quoi il a à faire, par exemple un cube ou bien une caméra.
Pour que le programme sache quoi faire Logan a eu recours à un switch utilisant le type comme variable des cases. Grâce à ce switch, tant qu'il existe un cas qui a pour nom la variable type, le programme pourra créer une instance de ce même type et paramétrer la Scene avec les instances créées. Au contraire si le switch ne reconnaît pas la variable type, une erreur sera lancée grâce au try mentionné précédemment.
Une fois que toutes les lignes ont été parcourues et que toutes les instances ont été utilisées pour définir la Scene, la méthode retourne la Scene afin qu'elle puisse être affichée.

## 5.3 Création de l'image à partir d'une Scène

Dans cette section nous traiterons des fonctionnalités implémentées dans la création de l'image réalisées par Antoine.

### 5.3.1 Création des Rayons

Tout d'abord, Antoine commença par rechercher comment générer les rayons sortant de la caméra. Ceux-ci devaient prendre en compte la position de la caméra ainsi que de son orientation.
Dans un premier temps, il a recherché à représenter les rayons selon une orientation par défaut. Il a choisi pour orientation principale le vecteur (0,0,20) et pour position par défaut l'origine de l'espace.
Pour définir les autres rayons, il décida que chaque rayon représenterait un pixel sur l'image et que donc la position en (x,y) du pixel influerait la coordonnée x et y de l'orientation du rayon qui lui est associé.
Pour faire cela, il a défini que le rayon du pixel (x,y) serait égal au vecteur de base, (0,0,20), en modifiant la coordonnée x pour qu'elle soit égale à x moins la moitié de la largeur de l'image demandé et en modifiant la coordonnée y de la même façon par rapport à la hauteur de l'image.
De cette manière, il a obtenu un lot de rayons qu'il a vérifié à l'aide de Geogebra [2], voir si visuellement les rayons étaient corrects. Il a obtenu le résultat suivant :

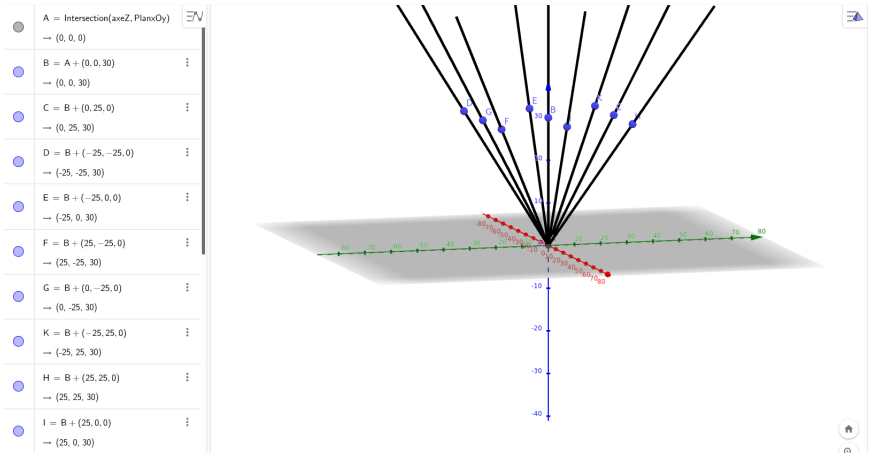



FIGURE 3 – Aperçu de la caméra dans son état de base

Une fois ceci réalisé il suffisait de gérer le déplacement de la caméra, ce qui était simple puisque à chaque direction de rayon il suffisait d'ajouter la position de la caméra.
Néanmoins, il restait encore à chercher comment pouvoir changer l'orientation des rayons ce qui fut difficile. Après quelques essais avec une méthode qui ne fonctionnait pas, il a établi le principe qu'il fallait partir des rayons dans la base canonique pour ensuite les changer en rayons corrects d'une autre base, c'est ici qu'intervient les premiers calculs matriciels.

2. En utilisant Geogebra classique 3D

Il est donc parti du principe que l'axe orientation de base (0,0,20) était la référence et l'orientation donnée par l'utilisateur, l'équivalent de la référence dans la nouvelle base. Antoine a donc fait des recherches pour trouver comment faire une rotation d'un vecteur dans l'espace. Il a trouvé sur wikipédia qu'il avait besoin d'avoir un angle de rotation ainsi que d'un axe autour duquel effectuer cette rotation.

Pour l'angle, il a appris sur wikihow comment calculer l'angle de rotation entre deux vecteurs u et v. Ensuite, il fallait trouver l'axe de rotation. Cependant, il ne savait pas lequel prendre et en allant sur un forum quelqu'un avait proposé une solution à mon problème en expliquant qu'il fallait prendre le vecteur perpendiculaire au plan deux vecteurs. Il suffisait donc de calculer la norme avec le produit vectoriel de $\vec{u}$ et $\vec{v}$. De cette façon, il a obtenu l'axe de rotation ainsi que l'angle pour établir sa matrice de rotation. A l'aide de la formule suivante pour une rotation autour de l'axe $\vec{u}$(x,y,z) et selon un angle $\theta$ où c = cos $\theta$ et s = sin $\theta$ :

$$\begin{vmatrix} x^2 + (1 - x^2)c & xy(1 - c) - zs & xz(1 - c) + ys \\ xy(1 - c) + zs & y^2 + (1 - y^2)c & yz(1 - c) - xs \\ xz(1 - c) - ys & yz(1 - c) + xs & z^2 + (1 - z^2)c \end{vmatrix}$$

De cette manière, on obtient la fonction CreateCanvasRay dont voici l'algorithme :

**Algorithme 2 :** CreateCanvasRay

**Entrées :** Int x, Int y
**Données :** Int width, Int height,Double [][] matriceRotation, Vector3D origineCamera
**Output :** Ray r

1 $point \leftarrow (x - width/2, y - height/2, 20)$
2 $point \leftarrow matriceRotation \wedge point$
3 **retourner** *Ray(origineCamera,point)*

Une fois les rayons de la caméra correctement générés, Antoine s'est penché sur le lancer de ceux -ci et de la détection de collisions.

### 5.3.2 Vérification des collisions des rayons

Ne sachant pas comment fonctionnait un algorithme de Ray tracing, Antoine a donc observé comment fonctionnait les algorithmes présentés dans les liens associés au projet plus particulièrement celui du site scratchapixel . Ainsi, il a décidé de reprendre en grande partie l'algorithme, en modifiant les algorithmes de collisions puisque nous considérions tout objet3D comme une liste de triangle et de créations des rayons.

La partie de recherche de collisions a été celle qui a pris le plus de temps à Antoine puisqu'il était difficile de trouver le point d'intersection entre un triangle et un rayon. Dans un premier temps, Antoine a cherché à faire son propre algorithme que voici.
La méthode du triangle pour savoir si un rayon le traversait prenait en paramètre un rayon. Cette méthode calculait alors la coordonnée x la plus petite et la plus grande des trois points du triangle et faisait de même pour y et z. Puis pour une variable i de la plus petite coordonnée x à la plus grande avec un pas réglable, le programme testait si le point ayant i pour valeur de x et appartenant au rayon. Si le point appartenait au triangle la méthode retournait le point sinon elle continuait à tester pour les axes y et z. Cette méthode produisait comme un quadrillage du triangle avec des mailles de plus ou moins grande taille afin de capter toutes les intersections.

**Algorithme 3 :** Premier algorithme de collision

**Entrées :** Triangle ABC, Rayon r, Vecteur3D vecteurDirecteurR
**Output :** Vector3D pointImpact

```
1  pas ← 0.1 minX ← min(A.getX(), B.getX(), C.getX()
2  minY ← min(A.getY(), B.getY(), C.getY()
3  minZ ← min(A.getZ(), B.getZ(), C.getZ()
4  maxX ← max(A.getX(), B.getX(), C.getX()
5  maxY ← max(A.getY(), B.getY(), C.getY()
6  maxZ ← max(A.getZ(), B.getZ(), C.getZ()
7  pour j ← minX; j < maxX; j+ = pas faire
8  |   coeff = vecteurDirecteurR.getX() / ( j - r.getOrigin().getX() )
9  |   point = r.getOrigin() + (j, vecteurDirecteurR.getY * coeff,vecteurDirecteurR.getZ * coeff)
10 |   si point ∈ Triangle alors
11 |   |   retourner point
12 |   fin
13 fin
14 pour i ← minY; i < maxY; i+ = pas faire
15 |   coeff = vecteurDirecteurR.getY() / ( i - r.getOrigin().getY() )
16 |   point = r.getOrigin() + (vecteurDirecteurR.getX * coeff, i,vecteurDirecteurR.getZ * coeff)
17 |   si point ∈ Triangle alors
18 |   |   retourner point
19 |   fin
20 fin
21 pour k ← minZ; k < maxZ; k+ = pas faire
22 |   coeff = vecteurDirecteurR.getZ() / ( k - r.getOrigin().getZ() )
23 |   point = r.getOrigin() + (vecteurDirecteurR.getX * coeff, vecteurDirecteurR.getY *
   |     coeff,k)
24 |   si point ∈ Triangle alors
25 |   |   retourner point
26 |   fin
27 fin
28 retourner null
```

Cependant, même en faisant un maillage de plus en plus précis l'algorithme ne remplissait pas totalement son objectif, même si Antoine pensait que cet algorithme allait suffire. Il a donc du chercher un autre algorithme qui fonctionnait totalement.

Guillaume trouva un algorithme sur le wikipédia du Ray tracing pour savoir si un triangle est touché par un rayon et en quel point. Antoine a donc implémenté ces algorithmes et le résultat fut assez satisfaisant. Présenté ici, vous trouverez l'algorithme beaucoup plus simple et efficace. De plus, cet algorithme n'est pas très coûteux il ne nécessite que deux produits matriciels et cinq produit scalaires par rayon testé.

**Algorithme 4 :** Second algorithme de collision

**Entrées :** Triangle ABC, Rayon r, Vecteur3D vecteurDirecteurR
**Output :** Vector3D pointImpact

$norme = ABC.getNorm()$
$diviseur = norme \cdot vecteurDirecteurR$
$\vec{AO} = Vecteur3D(r.getOrigin() - (Ax, Ay, Az))$
$Ir = \frac{normale \cdot \vec{AO}}{diviseur}$
$Iu = \frac{(\vec{AO} \wedge \vec{AC}) \cdot vecteurDirecteurR}{diviseur}$
$Iv = \frac{(\vec{AB} \wedge \vec{AO}) \cdot vecteurDirecteurR}{diviseur}$
**si** $Ir \geq 0 \ \& \ 0 \leq Iu \leq 1 \ \& \ 0 \leq Iv \leq 1 \ \& \ Iu + Iv \leq 1$ **alors**
  **retourner** *point*
**fin**
**retourner** *null*

Néanmoins, les coordonnées du point d'intersection retournés étaient dans une base différente de la base canonique, Antoine programma alors une méthode qui mettait à jour une matrice de passage afin de pouvoir retransformer les coordonnées dans la base canonique. Antoine n'a cependant pas réussi à faire fonctionner cette partie. Ainsi dans le moteur de rendu, les traits bleus étant normalement les arêtes du cube ne sont pas corrects.

### 5.3.3 Gestion des Ombres

En parallèle de ceci, Antoine a travaillé sur la lumière et les ombres. Il a d'abord étudié l'ombrage de Phong et comment stocker la lumière selon des intensités et comment les retranscrire en RGB.
Chaque intensité des lumières diffuse ambiante ou encore spéculaire nous permettent d'obtenir une couleur plus ou moins forte en multipliant chaque champs RGB de la couleur de notre lumière par l'intensité associée. De cette façon, on obtient trois couleur RGB, qu'il suffit de faire la somme pour obtenir la couleur réfléchie.

Le programme doit alors appliquer les formules de l'ombrage de phong pour obtenir les intensités correspondantes.

### 5.3.4 Passage à l'image

Pour implémenter le changement d'un tableau de Pixels à une image, Antoine voulu créer une classe spécialement dédié à ça qu'il nomma ImageCreator.

Pour stocker les pixels Antoine a choisi d'utiliser un tableau à deux dimensions puisque le nombre de pixels est défini dans le fichier pov et ne changeait pas au cours du programme.
Ensuite il fallait transformer ce tableau en une image, comme il n'avait jamais fait cela il regarda sur un forum un exemple de code voir quel package était utilisé [3] . Ici le programme est assez simple, il faut créer un fichier à l'aide d'un objet File et stocker les données de l'image dans un objet BufferedImage et d'utiliser la méthode write de la classe ImageIO.

3. Exemple de code trouvé sur geeksforgeeks

## 5.4 Affichage de la Fenêtre

Dans cette partie du rapport nous expliquerons pourquoi et comment à été créée l'interface graphique et de quelle manière elle a été liée avec les autres morceaux de code afin de donner une application fonctionnelle.

### 5.4.1 Objectifs de l'interface graphique

Le but principal de l'interface graphique de notre application est de permettre à l'utilisateur de choisir un fichier à lire afin d'en afficher le traitement à l'écran et de pouvoir modifier l'angle, et plus généralement la position, dans lequel l'objet est observé.

### 5.4.2 Création de l'interface graphique et des classes liées aux options de l'application

La première étape a évidemment été de se renseigner sur les différentes façons de créer une interface graphique à l'aide de Java. Assez vite, et un peu par défaut, car c'est la bibliothèque qui avait été vue en cours, Harrisson s'est dirigé vers Java Swing.

Cependant, il existe un second outil Java pour ce genre d'application ( "application de bureau" ). Il s'agit de JavaFX. Swing est pour l'instant bien plus utilisé dans l'industrie, possède plus de "block de construction" ( UI Components ) et un meilleur support. Cependant, JavaFX propose un meilleur support pour le MVC ( Model-View-Controller ) et a tendance à rattraper Swing sur le reste au fil des années. Harrisson pense finalement, que pour une première application, Swing était sûrement le meilleur outil à utiliser car plus "mature" dans l'ensemble, ce qui peut faciliter l'apprentissage du codage d'une interface graphique. Cependant, FX ayant tendance à gratter du terrain au fil des années, il serait peut-être plus judicieux d'apprendre à utiliser FX également car il risque d'être d'avantage démocratisé que Swing dans les années futures.

Pour commencer, il a créé une forme basique d'interface, une simple fenêtre avec une barre de menu. Ce simple affichage est alors géré par deux classes, Window et MainPane, celle qui nous intéressera surtout est MainPane. En effet il s'agit un peu du centre névralgique de l'interface car la plupart des classes de notre fenêtre sont liées au MainPane.
Un peu en difficulté pour se lancer dans la suite du projet, Harrisson a été aidé par Erin.
Le but alors, a été de comprendre comment faire entendre un clic de souris ou la pression d'une touche de clavier à la fenêtre, et voir si cela fonctionnait.
Il a donc été nécessaire de se renseigner sur les Listener, des interfaces liées à Java Swing et qui permettent, comme leur nom l'indique, d'écouter ce qu'il se passe dans la fenêtre, ou dans un panel. Un panel est un conteneur destiné a contenir d'autres composants et dans lequel il est possible de gérer les positions de ces différents composants à l'aide d'un gestionnaire de placements.
Pour en revenir aux Listener, on peut donner comme exemple la classe MouseWheelListener qui permet de détecter si la roulette de la souris à été utilisée, et dans quel sens, ou encore KeyListener permettant de savoir si une touche du clavier a été pressée, maintenue ou relâchée etc. Toutes ces informations on notamment été trouvées dans la documentation Oracle, sur WayToLearnX ou encore jmdoudou.fr.
Dans notre cas nous avions besoin de savoir si nous pouvions ouvrir un fichier, se déplacer dans l'espace créé à l'aide des touches du clavier et zoomer/dézoomer à l'aide de la roulette de la souris. Nous avons donc créé une classe pour chacune de ces fonctionnalités. Chacune de ces classes devait donc implémenter l'interface de listening liée à l'objet que l'on devait écouter. Lorsque le panel ou

le menu avait bien détecté notre action, une fenêtre supplémentaire s'affichait afin de nous alerter du bon fonctionnement de notre code. Sur le même fonctionnement, il a également été créé une section "Aide" dans le menu, qui, lorsque l'on clique sur "Raccourcis" ouvre une fenêtre détaillant les touches dédiées au déplacement. Ces classes ont été placées dans le package "Listener".

Erin ayant ensuite quitté le groupe, Harrisson s'est occupé de la suite.

Après cela, il a fallu créer la classe qui change les coordonnées de la caméra lorsqu'une touche du clavier correspondant à un déplacement est pressée. Il a alors simplement fallu créer une méthode différente pour chaque touche qui permet un déplacement. Dans chacune de ces méthodes, on récupère donc la position de la caméra, et l'une de ses coordonnées, et on la modifie en fonction de nos besoins. On se retrouve donc avec 8 méthodes chacune liées à une touche du clavier ou un bouton de la souris. Cette classe est ensuite restée "en sommeil" car pour la faire fonctionner il nous fallait afficher l'image créée par Antoine et Logan.

Et pour cela il nous fallait pouvoir choisir un fichier à ouvrir - ou dans notre cas, pouvoir récupérer son "chemin" - et afficher l'image traitée par les autres parties du programme. Ces deux classes sont les dernières liées à l'interface graphique. La partie principale de ces classes a été empruntées sur des codes déjà existant trouvés sur internet, puis par la suite Harrisson a modifié ces classes selon nos besoins.
Commençons par FileOpener, la classe permettant de choisir un fichier et de retourner son chemin afin que le parser puisse récupérer le fichier et le traiter. Le code a été trouvé sur le site WayToLearn et légèrement modifié afin de faire le lien avec le parser et afin d'afficher l'image, mais nous en parlerons un peu plus dans la partie suivante.

Enfin, la dernière classe, AfficheImage provient du site delfstack et a été modifiée afin de pouvoir afficher l'image créée par le programme sur un panel et afin de d'afficher une image par défaut ( une image d'accueil non choisie par l'utilisateur ) qui sera remplacée par les images générées par l'algorithme.

### 5.4.3 Liaison avec les autres parties de code

Évidemment, une interface graphique ne fait pas tout, il a fallu la lier avec l'algorithme principal de l'application afin que tout cela fonctionne.
Afin de clarifier les choses, voici les classes faisant le lien avec celles du reste l'algorithme : Moves, FileOpener, MouseWheelListenerClass et KeyListenerClass.

Moves a pour mission de changer les coordonnées de la caméra afin que l'utilisateur puisse observer l'objet sous différents angles. Elle va donc tout simplement récupérer ces coordonnées et les remplacer par les nouvelles, qui dépendent du déplacement demandé l'utilisateur. On se sert donc ici des getters et setters de la classe Camera.
Dans FileOpener, le lien est fait avec le Parser, que l'on instancie afin que celui-ci reçoive le chemin de l'image que l'on souhaite traiter et crée une scène qui sera ajoutée à notre interface afin de l'afficher. Puis, FileOpener rafraîchit l'affichage de la scène et du panel afin de remplacer l'image par défaut par celle que l'on a choisie.
Quand à elles, MouseWheelListenerClass et KeyListenerClass fonctionnent de la même manière, car, lorsque l'action à laquelle elles sont liées est détectée, elles appellent la méthode de la classe Moves correspondant au déplacement détecté. Puis, elles mettent à jour le panel et la scène afin d'afficher la nouvelle position de la caméra.

### 5.4.4 Affichage final de l'interface

Finalement, nous obtenons le résultat suivant :

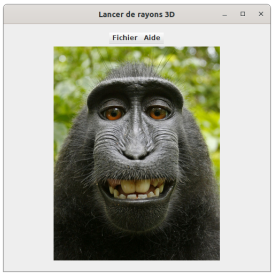



FIGURE 4 – La fenêtre lors de son lancement.

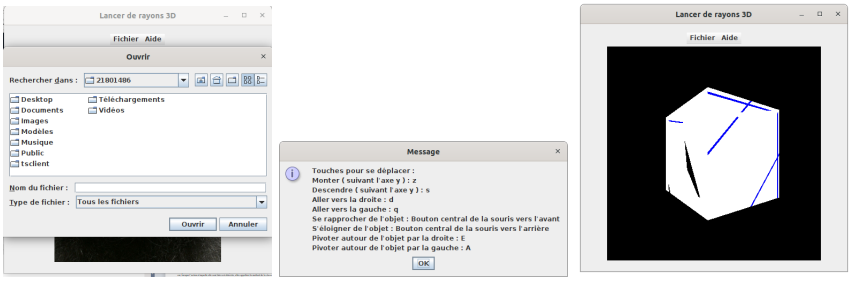



FIGURE 5 – Les options liées à l'interface et ce qu'elle affiche un fois le fichier ouvert ( image 3 ).

# 6 Difficultés rencontrées

## 6.1 Collision entre Triangle et Rayon

La recherche de la collision entre un triangle et un rayon a pris une grande partie de notre temps. En effet, nous avons passé beaucoup de temps à chercher des solutions qui ne fonctionnaient pas ou qui nécessitaient de résoudre un système d'équation, ce qui est, assez difficile à implémenter.
De plus, lorsqu'Antoine trouva la solution pour obtenir le point d'intersection entre le triangle et le rayon, et après avoir réalisé des tests afin de colorier les arêtes des objets en bleu il se rendit compte que les points d'intersection étaient dans certains cas incorrects. Or, il n'a trouvé aucune explication sur internet.
Ainsi, il réalisa différents tests et constata que l'ordre des points du triangle avait une importance. Pour un même triangle, si l'on inverse le point A et le point B, alors le résultat retourné par la méthode de recherche de point d'intersection est différent. Ceci a donc posé un énorme problème quand au développement de l'ombrage de phong et même de l'ombrage simple.

Enfin, Antoine a eut quelques difficultés au début pour implémenter tout ceci puisque il n'avait jamais fait de géométrie dans l'espace auparavant, il avait seulement les bases d'algèbres linéaires de l'année précédente. Cependant, une fois le vocabulaire et les méthodes de calcul compris, par exemple la différence entre un produit vectoriel et un produit scalaire, le développement fut plus agréable.

### 6.2 Liaisons avec l'interface graphique

Faire le lien entre les différentes classes ou méthodes à été à certains moments une difficultés pour Harrisson surtout lorsqu'il a fallu afficher l'image dans le panel. Cependant cela était surtout dû à de petites erreurs qui ont été corrigés à l'aide d'Antoine, après avoir fait une refonte totale de la façon dont été codé le Main, tout est devenu plus clair et Harrisson a réussi à réunir les différentes parties de l'interface graphique.
Une autre difficulté a été rencontrée lorsqu'il a fallu relier l'interface graphique avec les parties du code qui gère le traitement de l'image et les calculs mathématiques. Cependant, une fois qu'Antoine a terminé sa partie, il a pu aider Harrisson a lier les différentes parties. En effet cela est beaucoup plus simple lorsque les personnes qui ont créées leur morceaux de code individuellement travails ensemble.

## 7 Conclusion

Afin de conclure, nous pouvons dire que le projet était plus difficile qu'on le croyait, cependant nous sommes assez fiers de ce que nous avons produit. En effet, malgré les difficultés comme la présence d'un peu de physique et de mathématiques qui sont intervenus dans la gestion des collisions de rayons, nous avons su faire face et ne pas lâcher. Même si le résultat n'est pas à la hauteur de nos espérances, nous fûmes heureux de voir ce que nous avions accomplis.

## 8 Pour aller plus loin

Le moteur de rendu 3D que nous avons réalisé répond à la demande d'un rendu en image d'une scène décrite dans un fichier.Cependant, à cause du problème de l'inexactitude des points d'intersection il ne permet pas d'obtenir une lumière plus réaliste.
En effet, notre rendu 3D ne prend pas en compte des effets plus avancés tel que les caustiques, l'illumination globale ou alors la dispersion lumineuse. Pour tenir compte de ces effets, il faudrait avoir une approche probabiliste avec les techniques de Monte-Carlo, Metropolis ou la radiosité.

De plus il serait possible de créer un GUI pour le parser, la modification des fichiers de scène étant utilisable uniquement par écriture dans un éditeur de texte. Faire un GUI permettrait de contrôler totalement les entrées du parser avec une plus grande facilité, et permettrait à l'utilisateur de créer la scène plus simplement.

### 8.1 Effets avancés

La technique d'illumination globale consiste à prendre en compte la réflexion de la lumière sur les objets. En effet dans notre projet nous avons uniquement pris en compte la lumière venant d'une unique source de lumière. Alors que en réalité tout objet recevant de la lumière reflète à son tour la lumière qu'il n'a pas absorbé. Et donc ce principe prend en compte la réflexion lumineuse des objets.

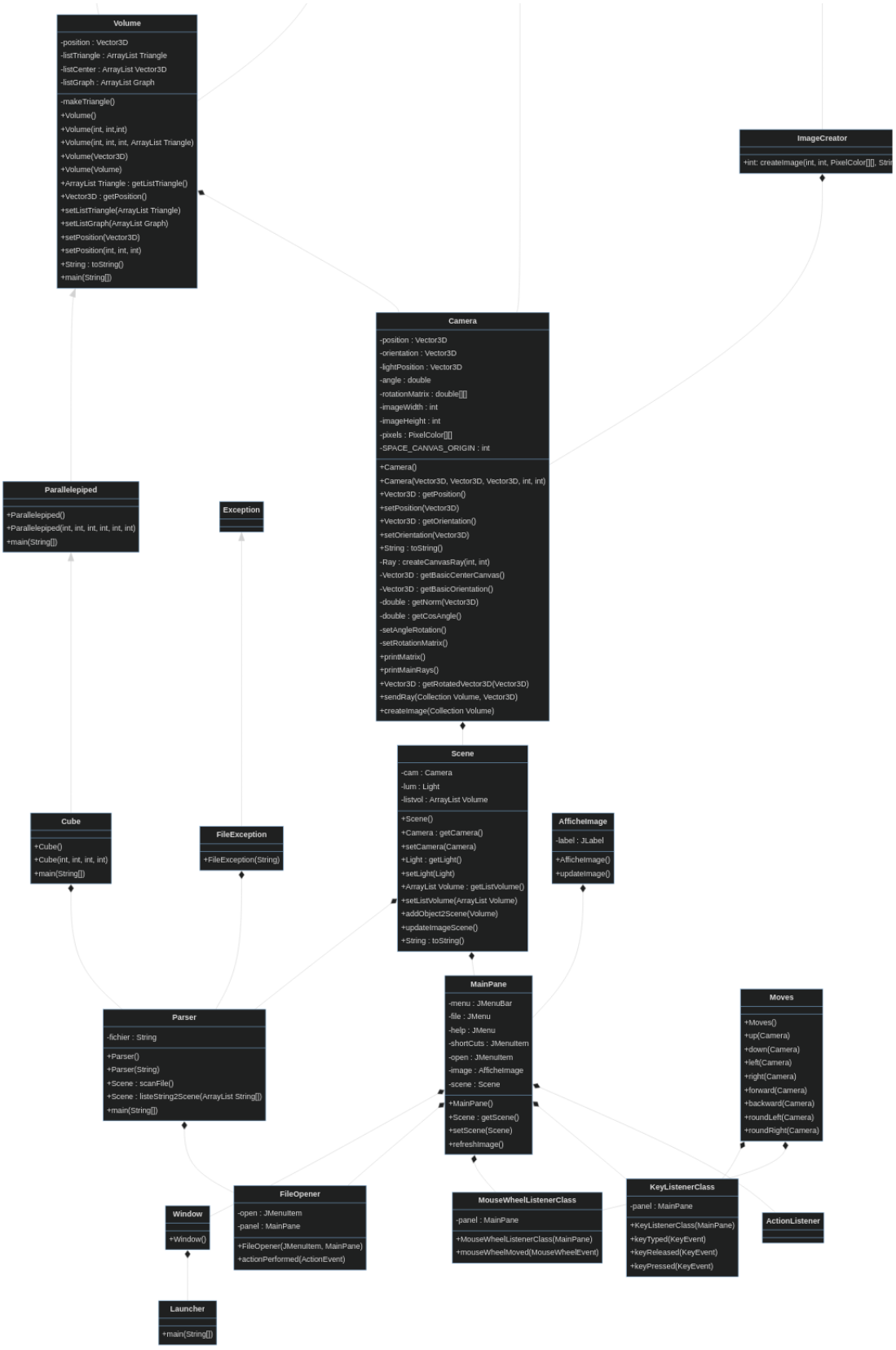
## 8.2 Technique probabiliste

La technique probabiliste consiste à faire en parallèle le calcul du rendu du rayon lancer tout en vérifiant si ce dernier touche en effet un objet. Si le rayon touche bien l'objet alors on ajoute son rendu au rendu final, dans le cas contraire on ne l'ajoute pas bien qu'on l'ait calculé. Cette technique est légèrement plus rapide car le calcul de collision est plus rapide que créer le rendu ce qui fait que le rendu ne sera pas entièrement calculé s'il n'y a pas de collisions.

# 9 Références

1. StackOverflow https://stackoverflow.com/
2. Wikipedia https://fr.wikipedia.org/wiki/Ray_tracing
3. WayToLearnX https://waytolearnx.com/tutoriels-java
4. Delfstack https://www.delftstack.com/fr/
5. WikiHow https://fr.wikihow.com/calculer-l'angle-entre-deux-vecteurs
6. ScratchPixel https://www.scratchapixel.com/lessons/3d-basic-rendering/introduction-to-ray-tracing/how-does-it-work
7. Geogebra https://www.geogebra.org

# A Arboréssance projet


Volume
-position : Vector3D
-listTriangle : ArrayList Triangle
-listCenter : ArrayList Vector3D
-listGraph : ArrayList Graph
-makeTriangle()
+Volume()
+Volume(int, int,int)
+Volume(int, int, int, ArrayList Triangle)
+Volume(Vector3D)
+Volume(Volume)
+ArrayList Triangle : getListTriangle()
+Vector3D : getPosition()
+setListTriangle(ArrayList Triangle)
+setListGraph(ArrayList Graph)
+setPosition(Vector3D)
+setPosition(int, int, int)
+String : toString()
+main(String[])
ImageCreator
+int: createImage(int, int, PixelColor[][], Stri
Camera
-position : Vector3D
-orientation : Vector3D
-lightPosition : Vector3D
-angle : double
-rotationMatrix : double[][]
-imageWidth : int
-imageHeight : int
-pixels : PixelColor[][]
-SPACE_CANVAS_ORIGIN : int
+Camera()
+Camera(Vector3D, Vector3D, Vector3D, int, int)
+Vector3D : getPosition()
+setPosition(Vector3D)
+Vector3D : getOrientation()
+setOrientation(Vector3D)
+String : toString()
-Ray : createCanvasRay(int, int)
-Vector3D : getBasicCenterCanvas()
-Vector3D : getBasicOrientation()
-double : getNorm(Vector3D)
-double : getCosAngle()
-setAngleRotation()
-setRotationMatrix()
+printMatrix()
+printMainRays()
+Vector3D : getRotatedVector3D(Vector3D)
+sendRay(Collection Volume, Vector3D)
+createImage(Collection Volume)
Parallelepiped
+Parallelepiped()
+Parallelepiped(int, int, int, int, int, int)
+main(String[])
Exception
Scene
-cam : Camera
-lum : Light
-listvol : ArrayList Volume
+Scene()
+Camera : getCamera()
+setCamera(Camera)
+Light : getLight()
+setLight(Light)
+ArrayList Volume : getListVolume()
+setListVolume(ArrayList Volume)
+addObject2Scene(Volume)
+updateImageScene()
+String : toString()
Cube
+Cube()
+Cube(int, int, int, int)
+main(String[])
FileException
+FileException(String)
AfficheImage
-label : JLabel
+AfficheImage()
+updateImage()
MainPane
-menu : JMenuBar
-file : JMenu
-help : JMenu
-shortCuts : JMenuItem
-open : JMenuItem
-image : AfficheImage
-scene : Scene
+MainPane()
+Scene : getScene()
+setScene(Scene)
+refreshImage()
Moves
+Moves()
+up(Camera)
+down(Camera)
+left(Camera)
+right(Camera)
+forward(Camera)
+backward(Camera)
+roundLeft(Camera)
+roundRight(Camera)
Parser
-fichier : String
+Parser()
+Parser(String)
+Scene : scanFile()
+Scene : listeString2Scene(ArrayList String[])
+main(String[])
Window
+Window()
FileOpener
-open : JMenuItem
-panel : MainPane
+FileOpener(JMenuItem, MainPane)
+actionPerformed(ActionEvent)
MouseWheelListenerClass
-panel : MainPane
+MouseWheelListenerClass(MainPane)
+mouseWheelMoved(MouseWheelEvent)
KeyListenerClass
-panel : MainPane
+KeyListenerClass(MainPane)
+keyTyped(KeyEvent)
+keyReleased(KeyEvent)
+keyPressed(KeyEvent)
ActionListener
Launcher
+main(String[])