Ray-Tracer

Baptiste CLOCHARD
Tom CLABAULT
Thibaut LETOURNEUR
Willie TANZIL

Table des matières

1 Introduction

1.1 Présentation du sujet

Le sujet que nous avons choisi pour ce projet est "Rendu 3D par lancer de rayon". Le lancer de rayon (raytracing en anglais) est la simulation du comportement des rayons de lumière visant à créer des images 3D réalistes. Ce sujet nous demandait d'implémenter :

— Le parsing d'un fichier .pov
— La réflexion
— La réfraction
— L'ombrage de Phong

Nous avons réussi à implémenter tout ce qui était demandé pour le sujet.

1.2 Forme finale du projet

Tout en respectant le sujet, nous avons décidé d'implémenter un certain nombre de fonctionnalités facultatives, mais qui nous semblaient intéressantes.

Nous avons donc réalisé, en plus de ce qui était exigé par le sujet :

— La gestion du rendu sur plusieurs threads.
— Le déplacement dans la scène 3d à l'aide de touches du clavier (avec actualisation de l'affichage en conséquence).
— le choix de la taille du rendu ainsi que son étirement possible pour conserver de la fluidité (mode automatique)
— L'usage d'une skybox.
— Les objets 3d possédant de la rugosité (raughness).
— L'ouverture et l'enregistrement du rendu 3D à l'aide de l'interface graphique.
— Le changement des paramètres du rendu à la volée (avec actualisation du rendu en conséquence).
— Un mode automatique permettant l'étirement propre de la fenêtre de rendu.
— L'ajout d'une texture en damier générée mathématiquement.

1.3 Répartition du travail

Assez tôt dans le projet le travail a été réparti et est resté pratiquement inchangé. Au final, voici les contributions de chacun :

Tom s'est chargé des structures mathématiques, des matériaux, des calculs du raytracing et de la gestion des threads

Baptiste s'est occupé des réfractions, de la gestion et de l'affichage du rendu, du déplacement de la caméra et des possibilités offertes par l'interface graphique.

Thibaut a codé le parser de fichier pov à l'aide d'un automate ainsi que les scripts de "déploiement".

Willie a réalisé les différents objets 3D (formes géométriques, lumière).

Chaque personne étant responsable de la javadoc des packages qu'il code.

Pour le rapport, chaque personne a détaillé les parties de son code et Baptiste s'est chargé des parties communes (introduction ...)

2 Structuration du projet

2.1 Les packages

2.1.1 Le parser

Pour parser notre fichier, nous avons besoin d'une classe servant en quelque sorte de chef d'orchestre. C'est la classe Automat. Elle possède un état courant, qui correspond à une figure dans le fichier pov, de type State. State est l'énumération des différentes figures possibles. Cette classe et cette énumération sont contenues dans le package povParser.

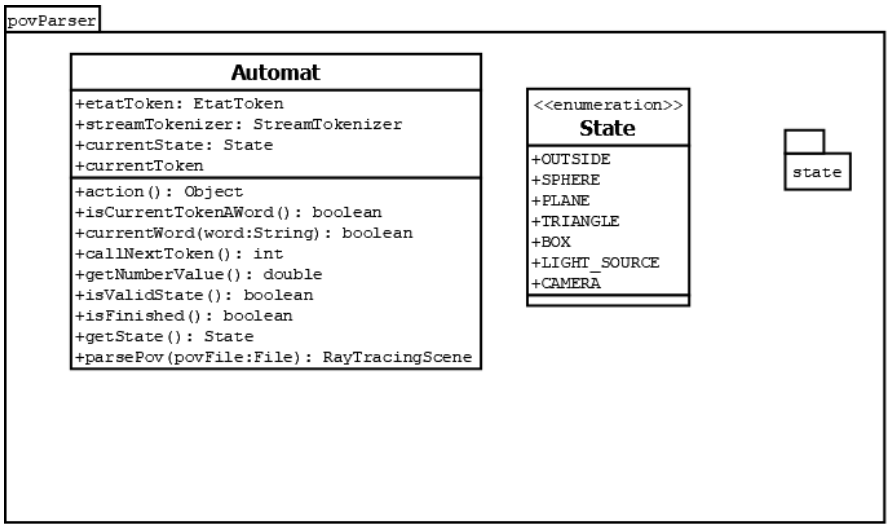


FIGURE 1 – Diagramme du package povParser

Le package state, quant à lui contient toutes les classes représentant des états, soit de figures (sphère, triangle, ...), la source de lumière, la caméra etc. Toutes les classes de ce package implémentent l'interface EtatToken. Chaque classe d'état possède sa propre énumération qui définit des contantes représentant des éléments de syntaxe.

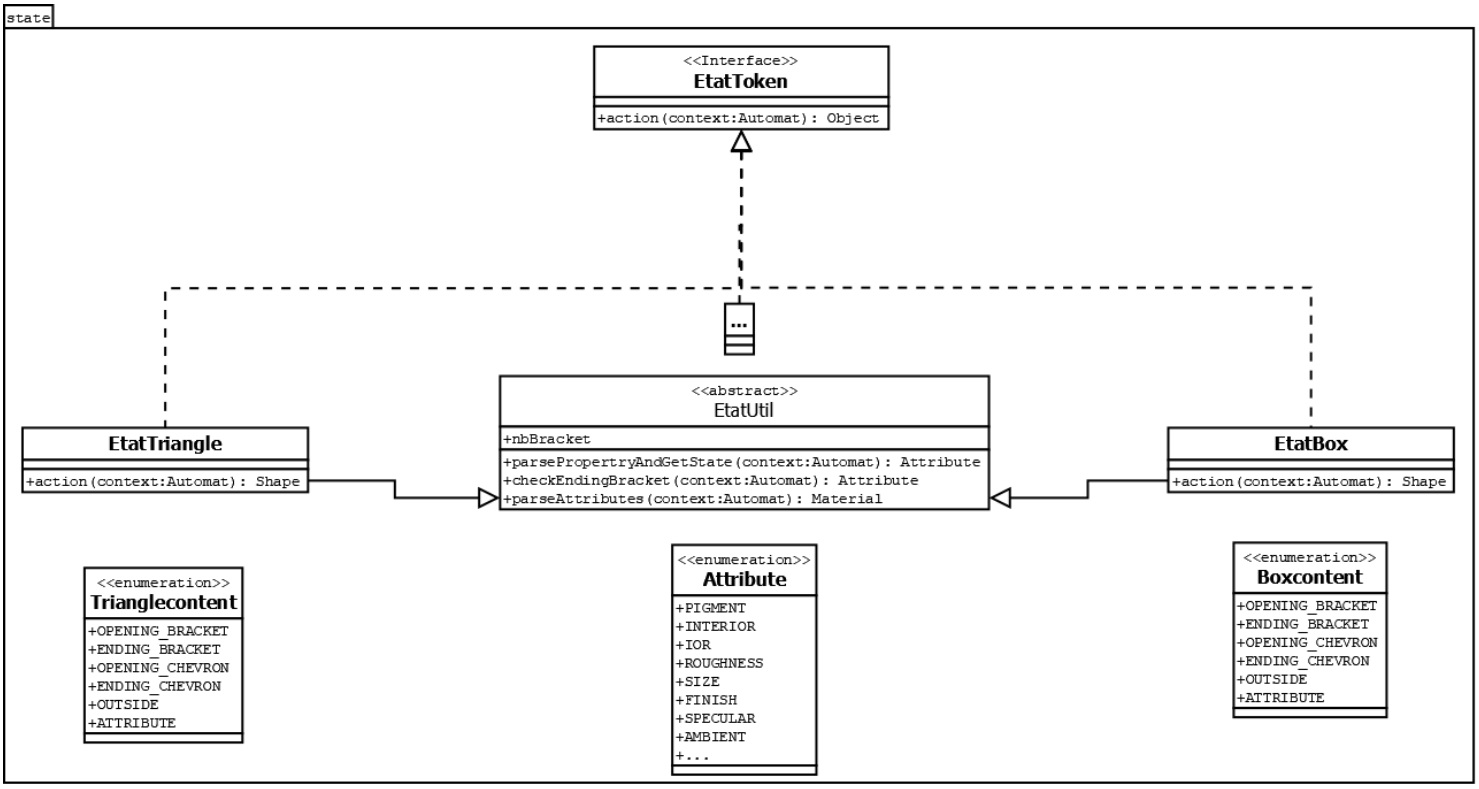


FIGURE 2 – Diagramme du package state

2.1.2 Liés au Ray Tracer

Pour calculer un rendu, le RayTracer a besoin de :

— Une résolution de rendu
— Une scène contenant les objets à rendre
— Une ou plusieures sources de lumière
— Une caméra
— Un nombre de thread sur lequel le rendu sera effectué

La résolution de rendu est fixe et est donnée au constructeur du RayTracer. Les autres réglages, se trouvant dans la classe *RayTracerSettings* sont quant à eux dynamiques. S'ils sont modifiés, le ray tracer adaptera alors son rendu en conséquence et en temps réel.

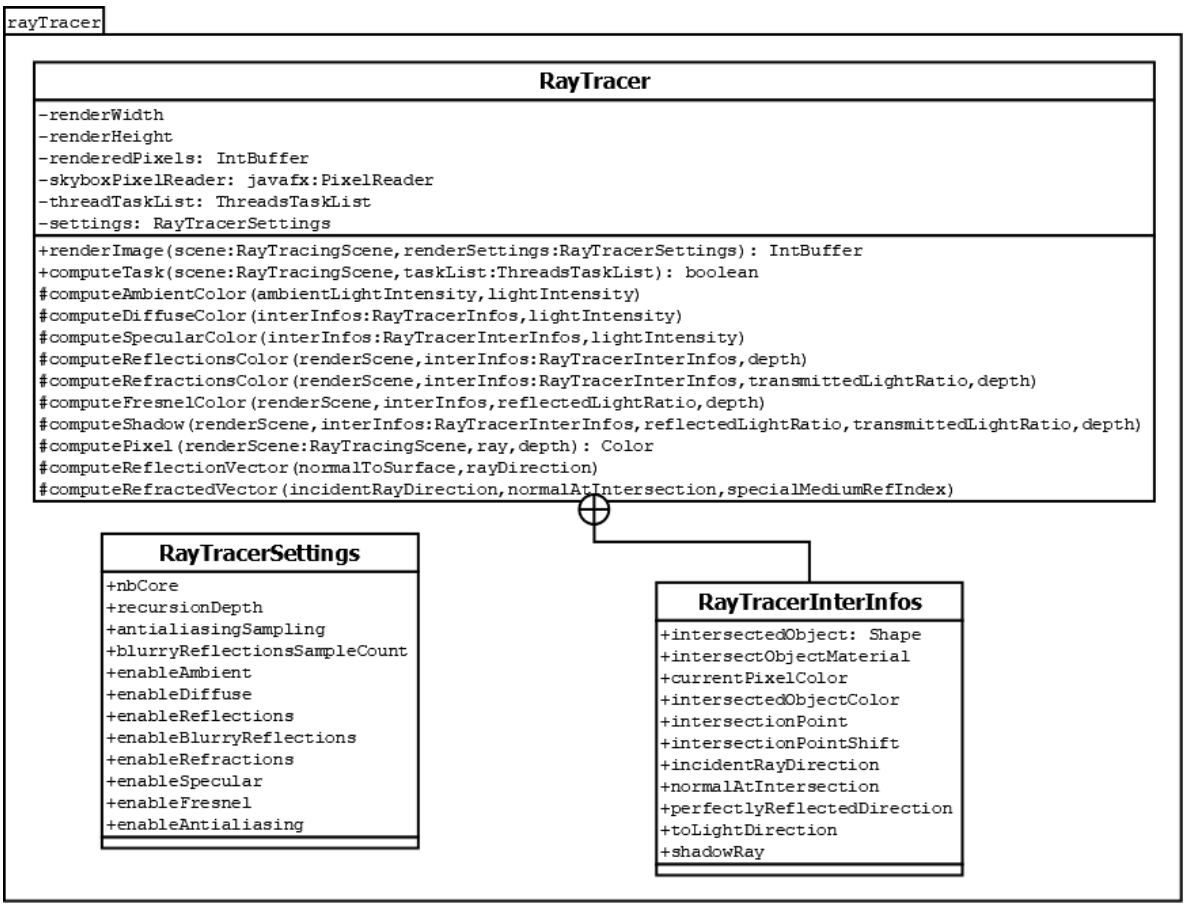


FIGURE 3 – Diagramme du package RayTracer

La classe *RayTracerInterInfos* est une classe interne à RayTracer. Elle récolte toutes les informations concernant l'intersection d'un rayon et d'un objet de la scène. Cette classe permet, en interne, une gestion plus facile du ray tracer et du calcul des réflexions, réfractions, etc...

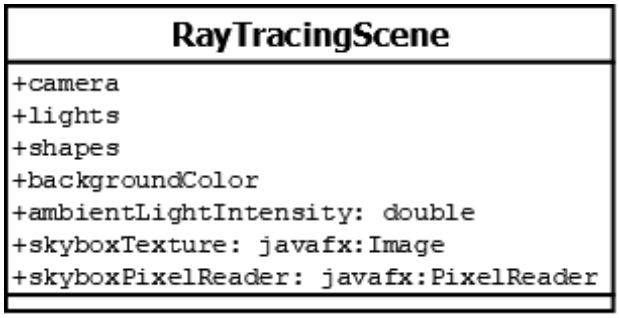


FIGURE 4 – Diagramme de la classe RayTracingScene contenant les informations de la scène

La classe Camera, l'interface Light et la classe RayTracingScene s'organisent toutes dans le package scene :

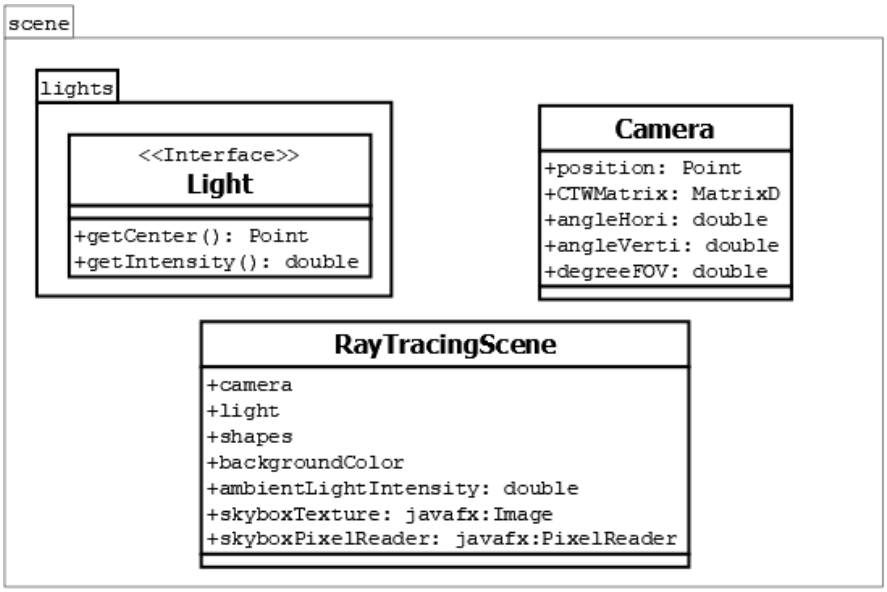


FIGURE 5 – Diagramme du package scene

2.2 Le package multithreading

Le package multithreading contient les trois classes qui sont utilisées pour le multithreading de notre ray tracer :

ThreadsTaskList Maintient la liste des tâches à calculer par les threads

TileTask Représente une tâche à calculer par un thread

TileThread Classe implémentant l'interface Runnable permettant de créer les threads

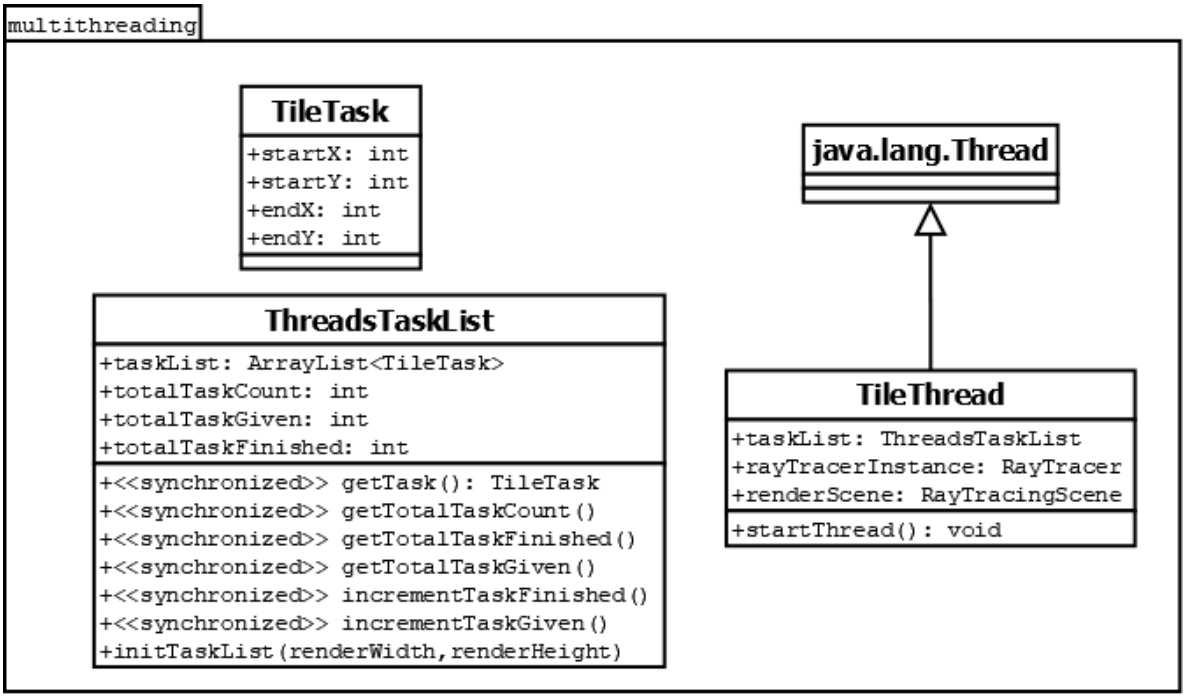


FIGURE 6 – Diagramme du package multithreading

2.3 Le package geometry

Notre projet dispose de plusieurs formes :
— Sphère
— Plan
— Triangle
— Icosphere
— ...
Les formes faites avec des triangles héritent de la classe ShapeTriangleUtil qui factorise les méthodes communes. Les formes définies avec des relations mathématiques héritent de ShapeUtil. Toutes les formes implémentent l'interface Shape qui est l'interface définissant une forme.

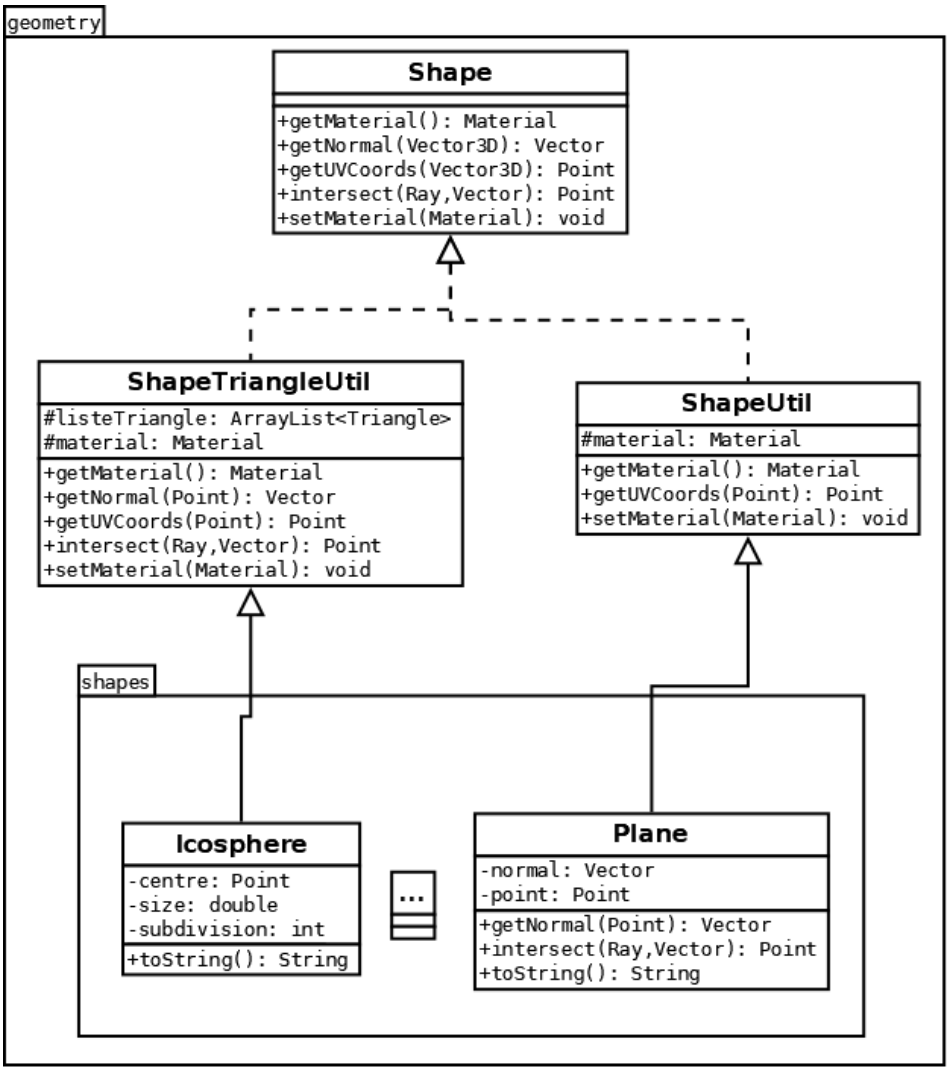


FIGURE 7 – Diagramme du package geometry

2.4 Le package materials

Afin de colorer notre scène, nous devons donner une couleur aux formes des objets qui composent la scène. De plus, nous devons indiquer au RayTracer si l'objet est réfléchissant ou

si au contraire il est mat, s'il est spéculaire et si oui à quel point, s'il est diffus... Nous allons donc organiser toutes ces caractéristiques dans ce que l'on va appeler des matériaux.

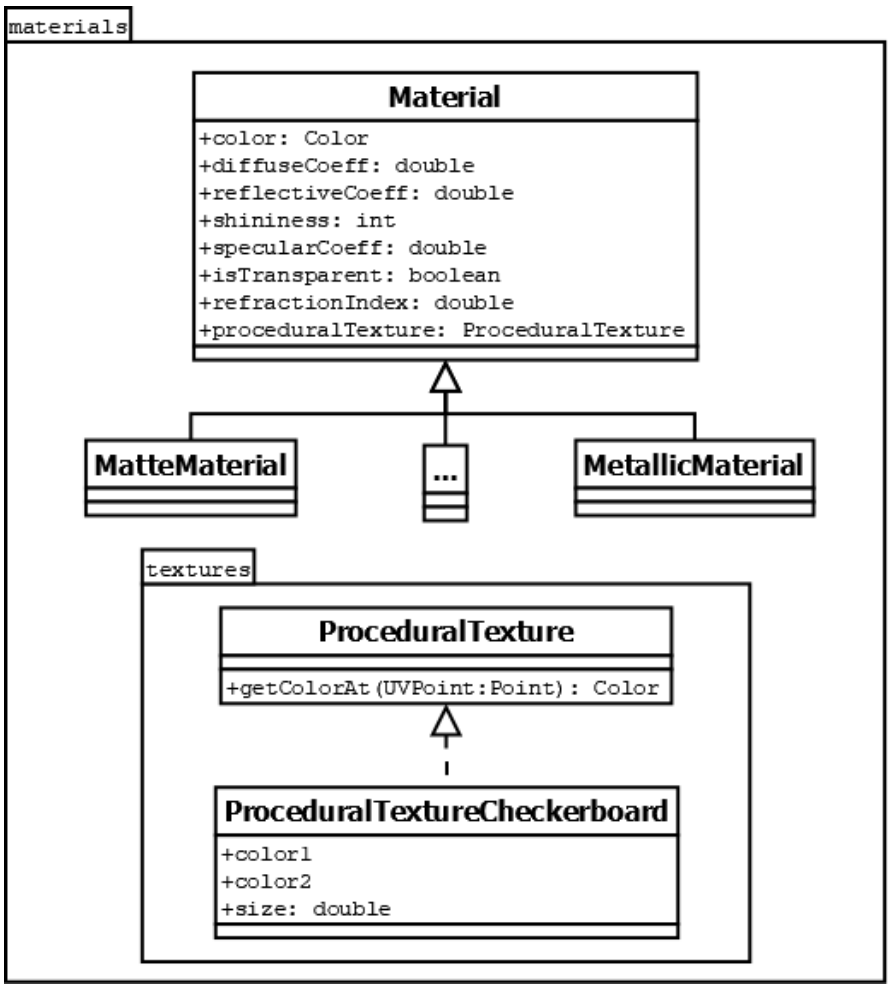


FIGURE 8 – Diagramme du package materials

Une classe mère Material permet de définir un matériau complètement arbitrairement. Toutes les caractéristiques peuvent être choisies comme on le souhaite. C'est cette classe qu'utilise le parser de fichier POV lorsqu'il doit construire un matériau à partir de ce qu'il rencontre dans le fichier POV donné en entrée.
Différentes classes héritent ensuite de cette classe mère afin de définir des catégories de matériau avec des propriétés fixées. Ces classes sont cependant devenues obsolètes après la finition du parser qui ne peut constuire que des matériaux arbitraires.

2.5 Le package render

Le package render 9 contient le code de l'interface graphique, c'est-à-dire celui permettant l'affichage du rendu et des fenêtres.

MainApp contient le main de l'application et lance toutes les fenêtres.

SetSizeWindow est la classe responsable de la fenêtre du choix de taille du rendu.

RenderWindow est la classe responsable de l'affichage de la fenêtre du rendu.

Toolbox est la classe responsable de la fenêtre de changement lors de l'affichage du rendu.

CameraTimer est la classe responsable du déplacement de la caméra lors de l'appui de touches.

DoImageTask est la task contenant les calculs du rendu. Voir L'interface graphique pour l'usage des taches.

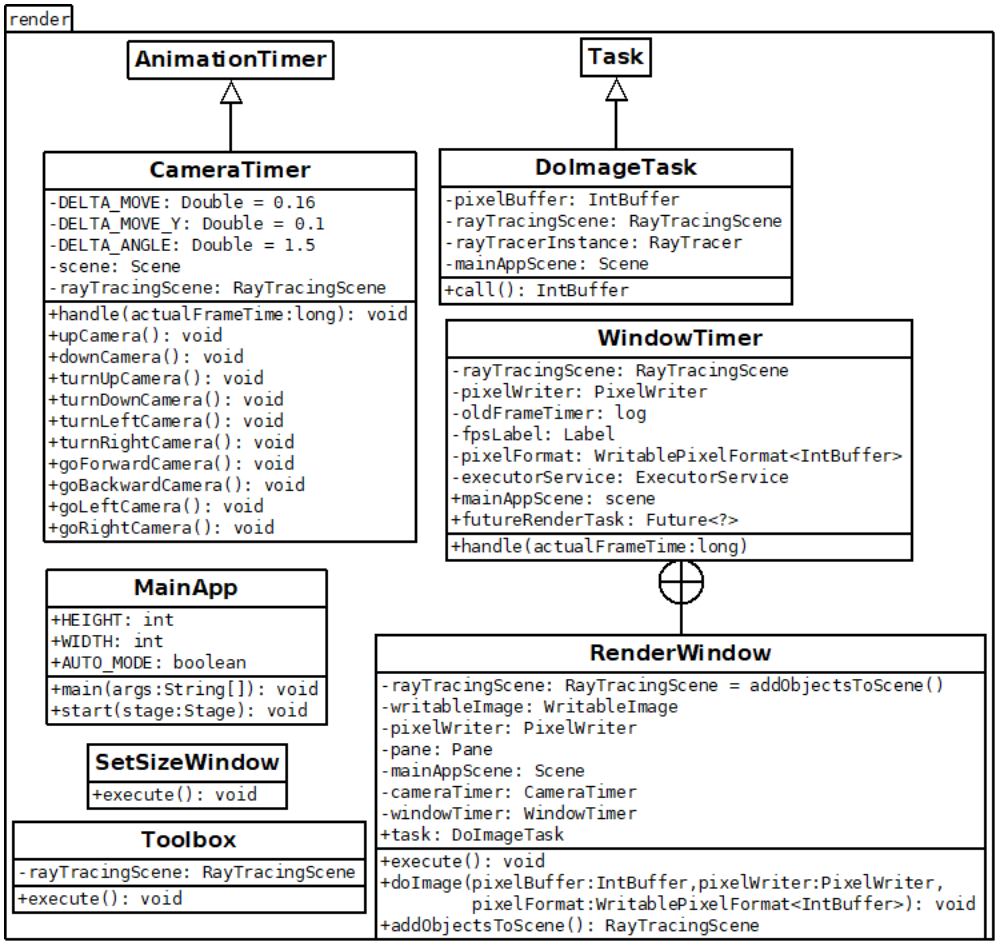


FIGURE 9 – Diagramme du package render

3 Détail des parties techniques

3.1 Le Ray Tracer

3.1.1 L'algorithme de base

Les algorithmes de ray tracing permettent de générer des images fortes de réalisme grâce à la simulation de lois physiques appliquées à la lumière. Pour rendre une image par lancer de rayon, nous avons besoin de :

— Une caméra
— Une source de lumière
— Une scène contenant différents objets

Le principe pour effectuer le rendu est ensuite assez simple. Nous lançons un rayon, partant de la caméra, à travers chaque pixel de l'image à rendre (on appelle cette image fictive par

laquelle passent les rayons le "plan de la caméra"). Si ce rayon intersecte un objet de la scène, alors nous renvoyons la couleur de cet objet pour le pixel traversé par le rayon (voir annexe A.1). Sinon, nous renvoyons la couleur du fond de la scène (ou de la skybox, section 3.1.5).

3.1.2 L'ombrage de Phong

En appliquant le principe de la section 3.1.1, nous obtenons une image plate, sans aucun relief. Pour palier à cela, un modèle d'ombrage reste à implémenter. Ce sont en effet les ombres et les effets de lumière qui donnent aux images leur réalisme et leur relief. Un ombrage de Phong (imposé par le sujet) a donc été implémenté. L'ombrage de Phong consiste en l'addition de trois composantes distinctes :

— La composante ambiante. Elle permet de donner une luminosité minimale à la scène.
— La composante diffuse. Illumine d'autant plus l'objet que les rayons de lumière le frappe perpendiculairement.
— La composante spéculaire. Rend l'objet brillant en tenant compte de l'angle avec lequel les rayons de lumière rebondissent vers la caméra.

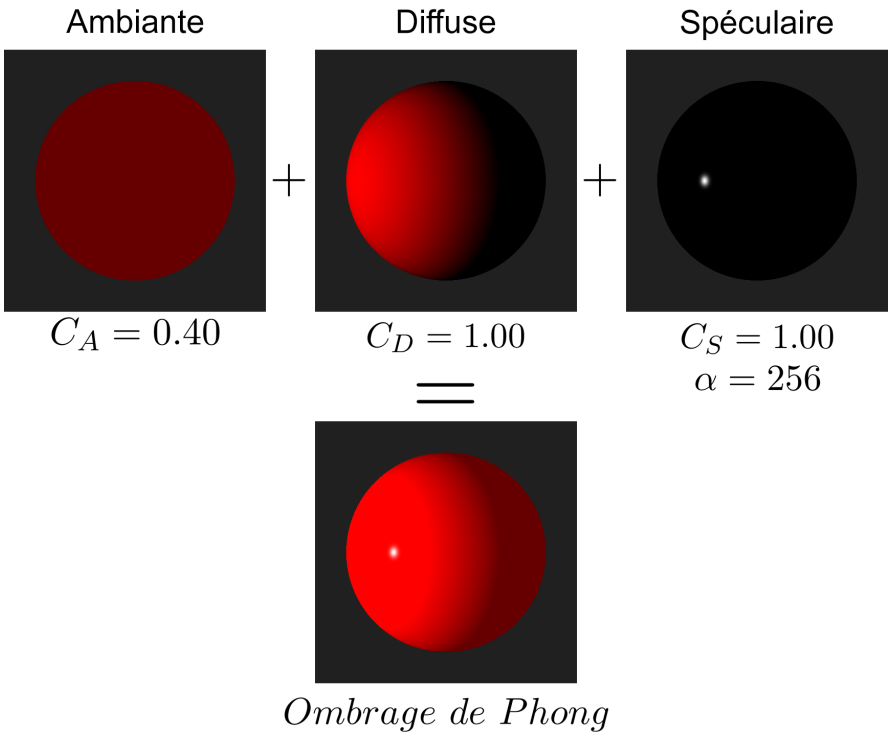


FIGURE 10 – Séparation des composantes de l'ombrage de Phong d'une sphère

3.1.3 Les réflexions

Une autre exigence du sujet (en plus de l'ombrage de Phong section 3.1.2) était également l'implémentation de réflexions. Nous avons donc implémenté la réflexion des matériaux à la façon des miroirs. Autrement dit, nous supposons que les matériaux réflechissants de notre ray tracer sont parfaitement lisses et renvoient les rayons de lumière dans une seule et unique direction. Nous avons pour cela utilisé un algorithme récursif. En effet, si notre rayon frappe, un objet réfléchissant, il sera renvoyé dans une autre direction. De nouveau, si ce nouveau

rayon réfléchi rencontre un autre objet réfléchissant, il sera une fois de plus renvoyé et ainsi de suite. Un algorithme récursif s'applique alors très bien ici.

Algorithme 1 : Algorithme de calcul des réflexions pour des objets non colorés - computeReflection($R, depth$)

Entrées : $\overrightarrow{R}$ le rayon incident (partant de la caméra)
depth la profondeur actuelle de récursion
Output : La couleur du pixel du plan de l'image par lequel est passé le rayon

```
1  si depth == 0 alors // La profondeur de récursion maximale a été atteinte
2  |  retourner Color.BLACK
3  fin
4
5  si R.intersects(objetsScene) alors // On vérifie si on a intersecté quelque
     chose
6  |  intersectedObject ← objet intersecté par R
7  |
8  |  si intersectedObject.isReflexive() alors // Si l'objet intersecté est
   |   réfléchissant
9  |  |  R_reflect ← computeReflectedDirection(R, normale)
10 |  |
11 |  |  retourner computeReflection(R_reflect, depth - 1)// On effectue un appel
   |  |   récursif pour relancer un rayon dans la direction du rayon
   |  |   réfléchi
12 |  fin
13 |  sinon // L'objet n'est pas réfléchissant, on va simplement retourner
   |   son ombrage de Phong
14 |  |  retourner phongShading()
15 |  fin
16 fin
17 sinon // On a rien intersecté
18 |  retourner backgroundColor
19 fin
```

L'algorithme lance un rayon et vérifie si une intersection avec un objet a été trouvée ou non. Si oui, on s'assure que l'objet est réfléchissant. Dans ce cas, nous calculons la direction du rayon réfléchi et faisons un appel récursif à computeReflection. Nous avons également défini une profondeur maximale de récursion (ou nombre de réflexion maximal) afin d'éviter les récursions infinies (un rayon coincé dans une boîte dont les faces seraient toutes réfléchissantes par exemple). Si l'objet n'est pas réfléchissant, nous renvoyons simplement son ombrage de Phong comme vu dans la section 3.1.2. Une représentation graphique de ce principe de réflexion est donné en annexe A.2.
Avec une scène contenant des sphères réfléchissantes, des sphères mattes et un plan pour le sol, nous obtenons le rendu suivant :

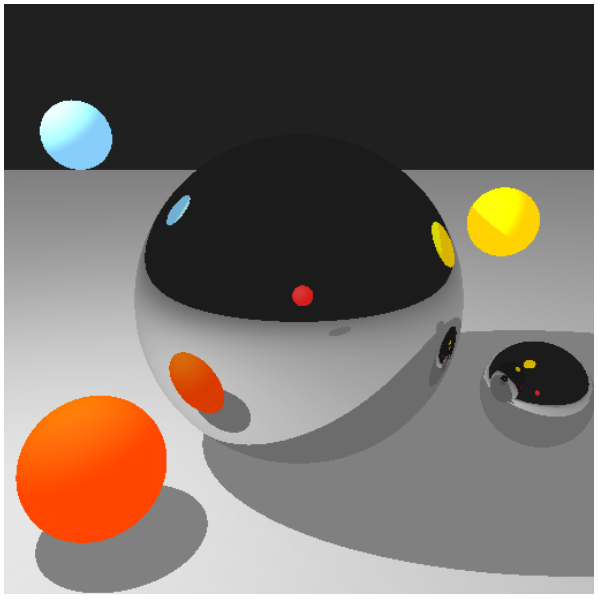


FIGURE 11 – On peut voir une sphère rouge derrière la caméra grâce à la reflexion de la sphère centrale.

3.1.4 Les réfractions

Le procédé des réfractions est le plus compliqué, mathématiquement parlant, à mettre en oeuvre. Cette difficulté est cependant récompensée d'une part par le réalisme de l'objet obtenu sur le rendu et d'autre part, par la complexité visuelle de l'objet observé.

Pour les formules mathématiques de réfractions, nous avons utilisé celle sur le site scratchapixel. Les matériaux utilisant de la transparence ont un paramètre supplémentaire : l'indice de réfraction. Pour nos tests, nous avons pris celui donné par Wikipedia pour le pyrex(1,474). Nous avions au départ mis un paramètre pour savoir à quel point le materiau était réfractif ou non, mais nous l'avons remplacé par l'équation de fresnel qui calcul la proportion de lumière qui est réfléchie / réfractée. Cette proportion change en fonction de l'angle de pénétration de l'objet, d'où la nécessité de l'emploi de la formule.

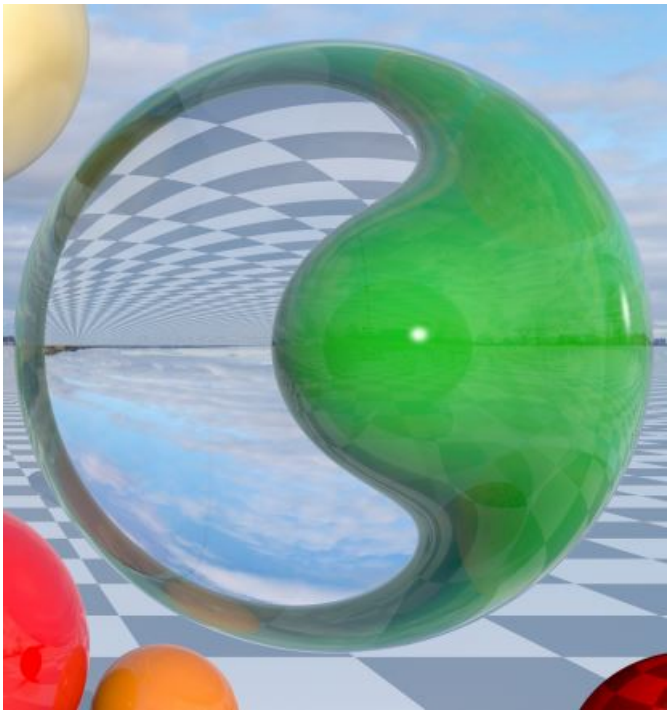
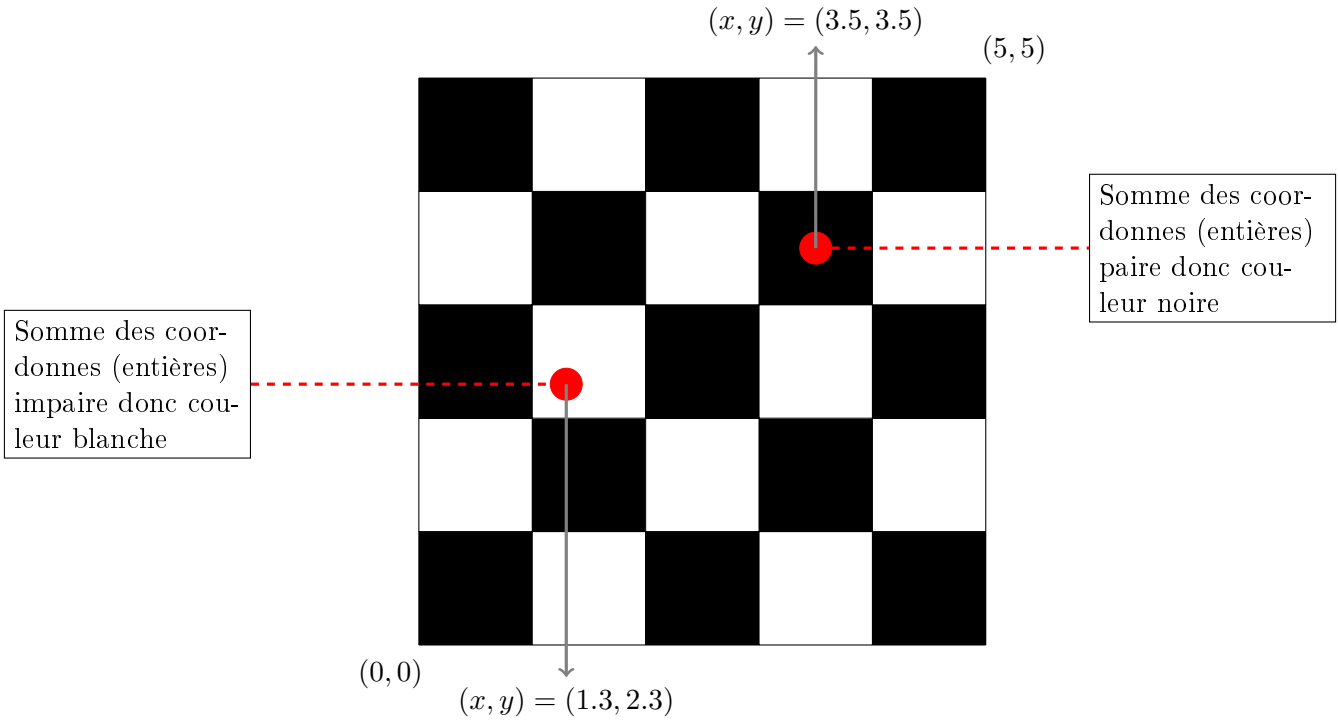


FIGURE 12 – Nous pouvons observer une boule verte à travers la boule transparente ainsi que le paysage qui est inversé à cause de la réfraction

3.1.5 Le damier et le ciel (skybox)

Afin de rendre les réflexions de la scène plus remarquables et cette dernière moins monotone, nous avons implémenté un damier au sol au moyen d'UV mapping. Ainsi, pour chaque point d'intersection avec le plan représentant le sol, nous regardons si la somme des coordonnées X et Y du point d'intersection est paire ou non. Si la somme est paire, nous renverrons la première couleur du damier. Si elle est impaire, nous renverrons alors la deuxième couleur du damier.



Bien que simple, cet algorithme d'UV mapping nous a permis d'embrayer sur l'implémentation d'une skybox. Notre skybox n'est autre qu'une image équirectangulaire haute résolution d'un espace à ciel ouvert. Pour l'afficher correctement comme le ciel de notre scène, nous allons, là encore, utiliser un mécanisme d'UV mapping. C'est en effet un moyen simple de convertir les coordonnées d'un point sur une sphère hypothétique de rayon 1 (notre ciel) en des coordoonées 2D pour une image (notre image haute résolution).

3.2 Les formes

Notre projet dispose de plusieurs formes qui sont utilisées pour le rendu des scènes. Les deux formes les plus utilisées sont le plan et la sphère. Nous avons aussi implémenté les parallélépipèdes, les prismes, les pyramides, ... Une forme plus technique que les autres que nous avons implémenté est l'icosphère.

3.2.1 Définition d'une icosphère

L'Icosphère est une sphère construite avec des triangles et un nombre de subdivisions donné. La Figure13 montre des icosphères avec des nombres de subdivisions différents.

3.2.2 Construction d'une icosphère

En partant de 20 triangles positionnés d'une certaine façon, nous pouvons calculer des subdivisions. Chaque subdivision multiplie le nombre de triangle de l'icosphère par 4 en divisant chaque triangle existant en 4 nouveaux. L'icosphère se rapproche alors, après chaque subdivision, d'une sphère parfaite. L'image 13 ci-dessous montre la différence entre des icosphères de subdivision 1, 3 et 5.

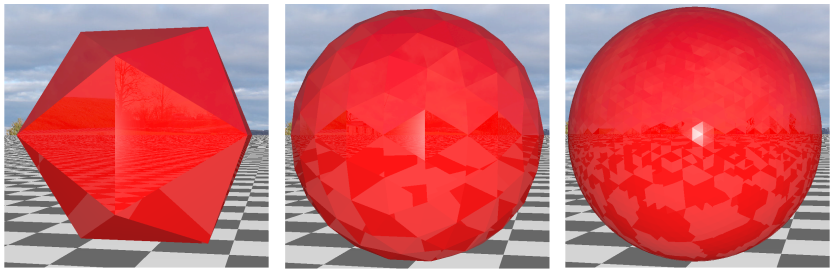


FIGURE 13 – Plusieurs icosphères de subdivisions différentes

3.3 Le multithreading

Les algorithmes de ray tracing sont réputés pour être lents à exécuter. Cependant, chaque pixel pouvant être calculé totalement indépendemment des autres, ces algorithmes sont des cibles parfaites pour la parallélisation de calcul. Nous avons donc mis en place, afin d'accélérer les temps de rendu, un multithreading de l'application grâce aux threads de Java. Pour ce faire, nous découpons l'image à rendre en plusieures "tuiles". Nous construisons ensuite une tâche à partir de chaque tuile. Une tâche consistant en :

— Un pixel de départ X et Y sur l'image à rendre
— Un pixel d'arrivée X et Y sur l'image à rendre

A partir de toutes ces tâches isolées est ensuite construite une liste de tâche. Cette dernière servira aux threads. Ils viendront en effet piocher dans la liste afin de récupérer les informations sur la zone de l'image qu'ils ont à rendre. Un diagramme présente le package multithreading dans la section 2.2.

Afin d'éviter les situations de concurrences entre les threads, la plupart des méthodes de la classe *ThreadsTaskList* sont ***synchronized***. Grâce à ce mot clé, seulement un thread du programme peut accéder à la méthode à un instant t et nous évitons ainsi les problèmes de concurrence.

3.4 Les mouvements de caméra

Afin de pouvoir se déplacer dans la scène, nous nous sommes attelé à l'implémentation de mouvements de caméra grâce aux touches du clavier. Les mouvements possible sont : les translations de la caméra ainsi que les rotations de cette dernière autour des axes X et Y. Translater la caméra est assez simple. Il suffit de choisir un vecteur par lequel translater la caméra et appliquer cette même translation à l'origine des rayons (car leur origine doit rester la caméra) ainsi qu'à leur point de direction. Pour ce qui est des rotations, le principe est le même. En effet, faire une rotation de la caméra de 90° autour de l'axe Y revient à appliquer la même rotation aux points de direction des rayons. Afin de simplifier l'utilisation des translations et des rotations, nous avons utilisé une matrice appelée la CTWMatrix (Camera To World Matrix). Cette matrice rassemble les transformations de translation et de rotation appliquées à la caméra. Il n'y a donc plus qu'à multiplier le point que l'on veut transformer par cette matrice et le tour est joué.
Les matrices de rotation dans un espace de dimension 3 se présentent sous la forme suivante :

$$R_{Y_\alpha} = \begin{pmatrix} cos(\alpha) & 0 & sin(\alpha) \\ 0 & 1 & 0 \\ -sin(\alpha) & 0 & cos(\alpha) \end{pmatrix} ; R_{X_\beta} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos(\beta) & -sin(\beta) \\ 0 & sin(\beta) & cos(\beta) \end{pmatrix}$$

Source : wikipedia [1]

R_{Y_α} représente la matrice de rotation d'angle α autour de l'axe Y et R_{X_β} son équivalent d'angle β autour de l'axe X.

Une matrice de translation pour un vecteur $\overrightarrow{u}(a, b, c)$ se présente comme suit :

$$T_{\overrightarrow{u}} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix}$$

On sait que la multiplication de deux matrices de rotation donne aussi une matrice de rotation combinant les deux rotations : ${R_{Y_\alpha}}^* R_{X_\beta} = R_{Y_\alpha X_\beta}$. On peut ensuite combiner la matrice de rotation obtenue avec notre de matrice de translation et on obtient :

$$CTWMatrix = \begin{pmatrix} R_{YX_{0,0}} & R_{YX_{0,1}} & R_{YX_{0,2}} \\ R_{YX_{1,0}} & R_{YX_{1,1}} & R_{YX_{1,2}} \\ R_{YX_{2,0}} & R_{YX_{2,1}} & R_{YX_{2,2}} \\ a & b & c \end{pmatrix}$$

Multiplier un point par cette matrice revient alors à effectuer les rotations des deux matrices R_{Y_α} et R_{X_β} et ensuite à translater le point résultant par le vecteur $\overrightarrow{u}(a, b, c)$.Tout cela en une seule opération de multiplication matricielle.

Des schémas représentant les rotations de la caméra autour des axes X et Y sont disponibles en annexe A.3

3.5 L'interface graphique

3.5.1 Pourquoi JavaFX ?

Une question que nous nous sommes posée est : Swing ou JavaFX ?

Nous avons choisi d'utiliser JavaFX pour une principale raison. JavaFX est désormais la librairie officielle de Java. Swing n'étant plus maintenu, nous ne voulions pas développer avec une librairie qui commence à être dépréciée.

De plus, le sujet de Bataille Navale (le projet de Complément de POO que nous devons réaliser en parallèle) étant à faire obligatoirement en swing, nous voulions essayer les deux librairies.

3.5.2 L'interface graphique

Le développement de l'interface graphique a progressé au rythme du reste du projet, implémentant ce qui a été codé en parallèle. Plusieurs optimisations ont cependant été réalisées pour que l'affichage à l'écran impacte le moins possible les performances générales.

L'optimisation la plus impactant est l'usage d'un IntBuffer et de PixelFormat<IntBuffer>. Ils ont permis d'utiliser la méthode setPixels de PixelWriter, permettant ainsi de définir les pixel de la fenêtre de rendu d'un coup et non pas pixel par pixel comme fait précédememnt.

Un des effets secondaires engendrés par les calculs lourds du rendu est la fenêtre qui n'était plus réactive. Au départ, nous utilisions CameraTimer et WindowTimer, deux classes héritant de Animation Timer. WindowTimer lançait le rendu et CameraTimer s'occupait de gérer un déplacement de caméra. La méthode handle de AnimationTimer s'exécute à chaque frame, ce qui nous permettait de gérer l'affichage des FPS, la caméra et le rendu à l'aide de la méthode handle et tout était donc synchronisé.

Le problème posé par ce choix était que JavaFX gérait directement les calculs de rendus. La conséquence était une interface graphique réactive à la vitesse du rendu (quelques images par seconde).

Pour palier à ce problème, nous avons utilisé des classes proposées par le package concurrent de JavaFX. Le problème a été résolu grâce à la classe tasks de ce package, permettant de faire des calculs lourds sans impacter la réactivité de l'interface graphique. Comme javafx et le calcul de rendu n'étaient plus liés il a fallu synchroniser les mouvements de caméra et le calcul de l'image d'après.

3.5.3 La Toolbox

La toolbox est un élément essentiel du raytracing. Elle permet à l'utilisateur d'utiliser le logiciel avec simplicité, sans devoir passer par des lignes de commandes ou des scripts d'exécution. Ensuite, le logiciel de rendu utilise différents procédés qui peuvent fonctionner en parallèle ou les uns au-dessus des autres. La toolbox permet de choisir quels éléments l'utilisateur veux conserver pour pouvoir ainsi mieux comprendre les différentes étapes jusqu'au rendu final. Pour finir, cette interface permet à l'utilisateur d'avoir un maximum de liberté sur l'utilisation du logiciel, lui permettant ainsi de faire un compromis entre fidélité et rapidité.

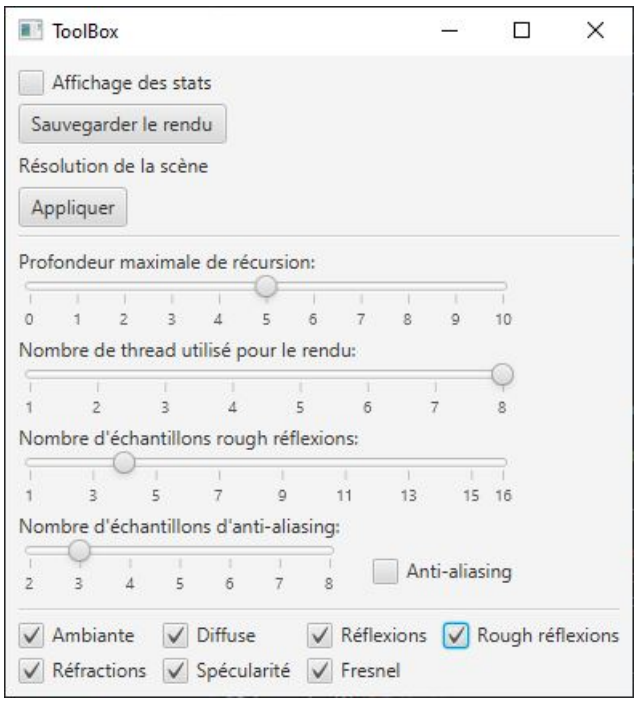


FIGURE 14 – La toolbox et les possibilités qu'elle offre.

3.5.4 Le CSS

En quelques mots, JavaFX permet de lier le style de ces fenêtres à des fichiers CSS. Nous en avons profité pour aérer les fenêtres de choix de taille et la "toolbox".

3.5.5 Le mode automatique

Le mode automatique récupère la taille de l'écran principal, maximise la fenêtre et étire le rendu de manière à ce qu'il soit pixélisé, mais qu'il occupe tout l'écran.

Dans une version plus ancienne, la taille du rendu était également fixée par la taille de la fenêtre. Suite à l'ajout de la réfraction et de l'uvsphère qui sont assez coûteuses en ressource nous avons décidé de supprimer cette fonctionnalité.

3.5.6 Le conteur de fps

Le conteur de FPS tire parti des avantages offerts par la classe AnimationTimer et utilise l'horodatage fournit par celle-ci pour calculer la vitesse de rendue.

3.6 Le parser

3.6.1 Structure d'un fichier POV

Dans le sujet, il nous été demandé de parser un fichier au format pov (Persistence Of Vision). La syntaxe d'un fichier pov ressemble assez à celle du langage C. Pour définir une

figure il suffit de préciser son nom. C'est dans le bloc d'un objet, délimité par des accolades, que se trouvent les coordonnées de l'objet, sa couleur, etc. Voici une sphère définit en pov :

```
sphere
{
    <-2, -0.65, -5>, 0.35
    pigment
    {
        color rgb <1, 1, 1>
    }
    finish
    {
        ambient 1
        diffuse 0.75
        specular 0.05
        phong_size 1
    }
    interior
    {
        ior 0.5
    }
}
```

On retrouve tout d'abord les propriétés intrinsèques de l'objet, donc le centre et le rayon de la sphère dans l'exemple ci-dessus. On peut ensuite rajouter des éléments de couleur dans un bloc pigment et des attributs liés à la lumière, aux ombres dans un bloc finish. On peut aussi rencontrer un bloc interior dans lequel est définit un indice de réfraction. Le bloc pigment peut aussi contenir un élément checker qui définit un damier avec deux couleurs que l'on donne.
De plus, nous avons fait le choix de rajouter un élément de syntaxe à des fins pratiques. C'est l'attribut size du checker qui permet de choisir plus simplement une taille pour le damier. En effet, la manipulation équivalente en POV est bien plus laborieuse et demande d'initialiser d'autres attributs qui ne sont pas implémentés dans notre lanceur de rayons.
Afin de parser notre fichier, nous avons choisi la classe StreamTokenizer de Java comme moteur pour notre parser. Elle fournit des outils simples mais efficaces pour parser des fichiers ressemblant au C/C++. Son principe est simple, on parcourt le fichier à l'aide d'un jeton, qui est de type entier. Ce jeton prend des valeurs différentes selon les objets qu'il rencontre. S'il se trouve sur un mot, dans ce cas-là, il sera égal à une constante TT_WORD, idem pour un nombre avec TT_NUMBER. Lorsque le jeton est égal à la constante TT_EOF, cela signifie que nous avons atteint la fin du fichier et que l'on peut arrêter le parsing.

3.6.2 Automate à état finit

La méthode retenu pour effectuer le parsing d'un fichier POV est celle de l'automate à états finis. C'est donc pour cela que l'on a crée la classe Automat. Le principe est simple, on associe à chaque figure une classe d'état. Une classe EtatSphere, une classe EtatTriangle, etc. Chaque classe d'état s'occupe de parser son objet associé. La classe Automat s'occupe juste d'appeler au bon moment les états, c'est à dire quand le jeton de son StreamTokenizer rencontre un mot connu, soit sphère, plane, box... Les différents états possibles de l'automate sont définis dans l'énumération State. Voici le diagramme d'états des figures :

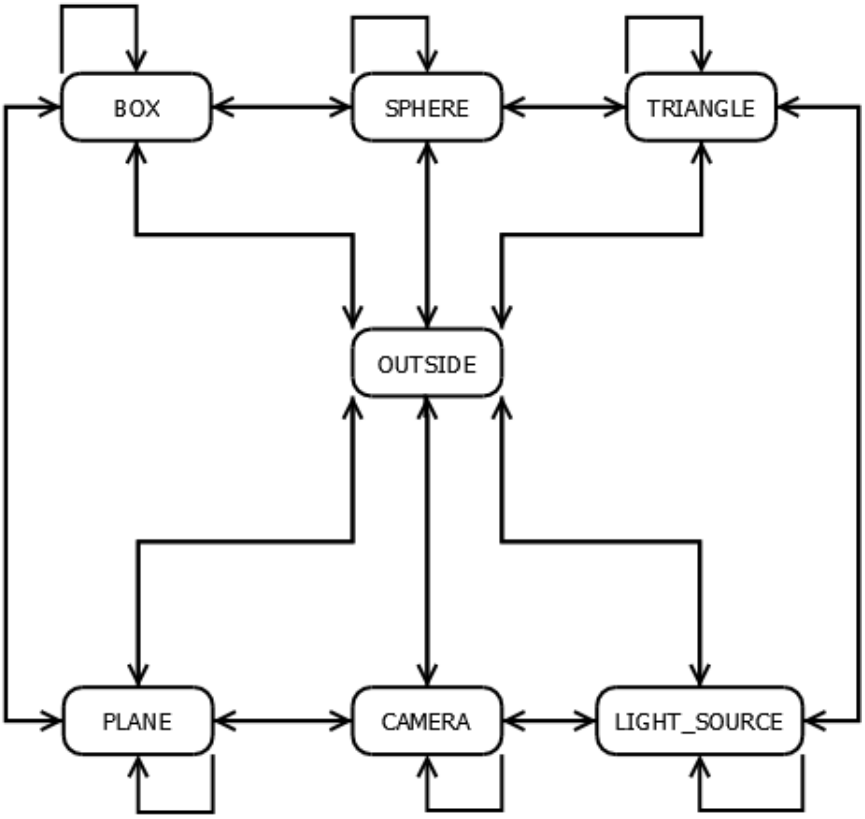


FIGURE 15 – Diagramme d'étas des figures

Concrètement, dans la classe Automat, on traverse le fichier jusqu'à ce qu'on rencontre une figure connue. A ce moment là, on fixe l'état sur cette figure et on appelle la classe d'état correspondante. Dès lors que la parsing de la figure est fini, on regarde le prochain mot et on fixe l'état correspondant. Tout ces traitements dont effectués dans une boucle qui a pour condition d'arrêt la fin du fichier. Ensuite on switch sur l'état courant et on définit autant de case qu'il y a d'états. On a aussi définit un état vide, l'état OUTSIDE qui correspond à aucun élément de syntaxe. Dès lors que l'on se trouve dans un état OUTSIDE, on avance juste notre jeton dans le fichier pour tomber sur la prochaine figure, s'il y en a une.

```
while(! automat.isFinished())
            {
                if(automat.isValidState())
                {
                    State currentState = automat.getState();

                    switch (currentState) {

                        case LIGHT_SOURCE: {
                            automat.setState(new EtatLightSource());
                            PositionnalLight light = (PositionnalLight) automat.action();
                            scene.addLight(light);
                            break;
                        }

                        case SPHERE: {
                            automat.setState(new EtatSphere());
                            Shape sphere = (Shape) automat.action();
                            scene.addShape(sphere);
                            break;
                        }

                        case OUTSIDE: {
                            automat.setState(new EtatOutside());
                            automat.action();
                            break;
                        }

                    ...
```

De plus nous avons implémenté des sous états qui se trouvent dans chaque classe. Ils sont contenus dans des énumérations présentes dans chaque classe d'états. Ces états correspondent en fait à des éléments de syntaxe que l'on trouve dans un bloc de figure. C'est à dire qu'on a un état pour l'accolade fermante, ouvrante, le chevron ouvrant , le chevron fermant etc. Le principe est le même que pour les états de figures. Même s'ils se ressemblent fortement, nous avons besoin d'une énumération par classe car la syntaxe n'est pas toujours la même pour chaque figure.

Par ailleurs, pour des raisons de factorisation, nous avons ajouté un état ATTRIBUTE. En effet, quelque soit la figure sur laquelle on se trouve, si elle possède un bloc finish (lumière) ou un bloc pigment (couleurs), la syntaxe sera la même peu importe la figure. C'est donc pour cela que l'on a crée un classe EtatUtil qui s'occupe de parser ces autres blocs de code. En fait chaque classe d'état parse les coordonées de sa figure et appelle la classe mère EtatUtil en lui donnant le contexte courant pour parser ses attributs s'il y en a. Toutes les classes d'état héritent donc de la classe mère EtatUtil

3.6.3 Implémentation du pattern state

Afin d'organiser tous les états entre eux, la méthode qui a été retenu est celle du pattern State. Ce pattern s'est révélé être le plus adapté pour implémenter un automate à états finis. Voici, ci-dessous, le diagramme de classe de notre pattern state avec les classes d'état qui l'implémentent :

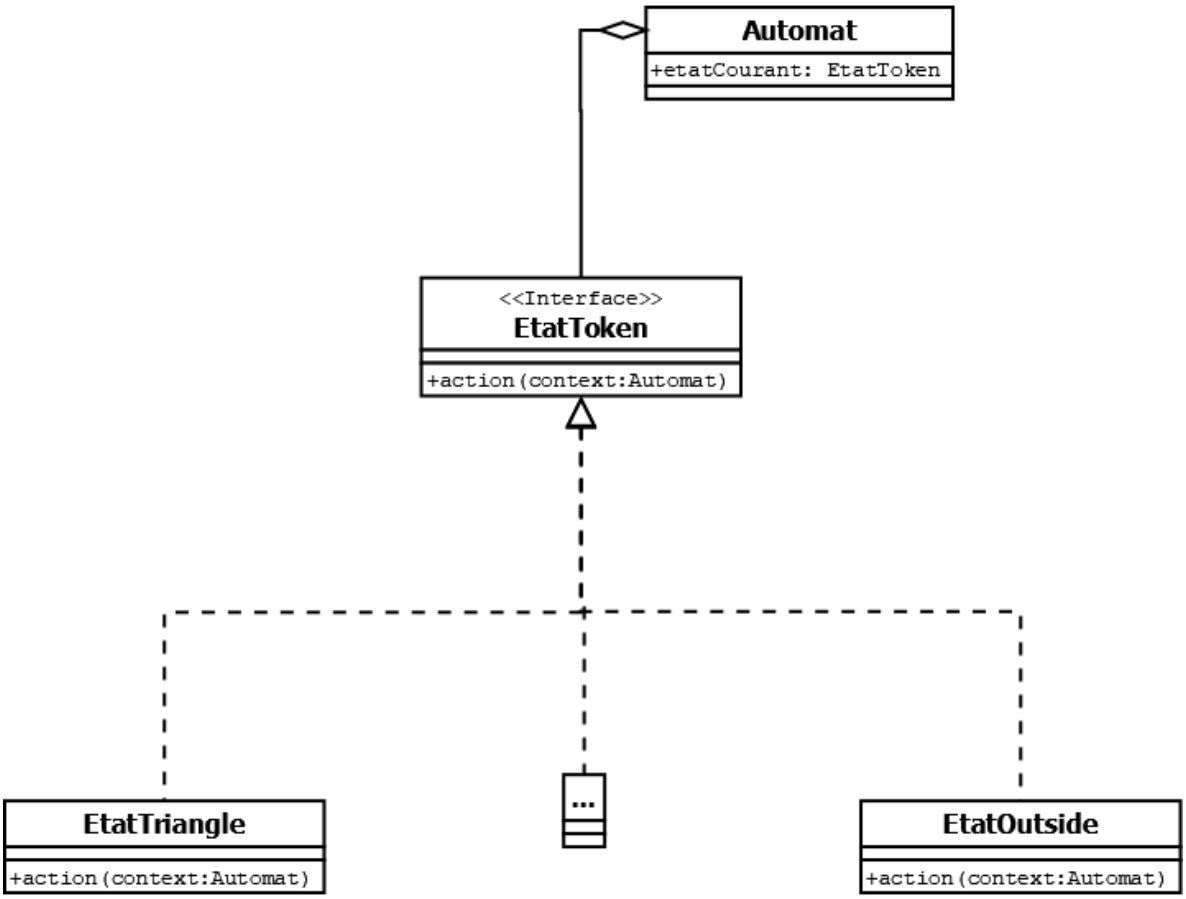


FIGURE 16 – Diagramme du pattern state appliqué à notre parser

Ce pattern est composé d'une interface EtatToken qui définit une méthode action qui servira à parser le bon objet. Ensuite, on crée ce que l'on appelle des classes d'état qui implémentent l'interface et donc redéfinissent la méthode action. Ce sont elles qui s'occupent de tout le traitement de la figure. Ces classes correspondent à un seul objet/figure à la fois, à l'exception de la classe EtatSpherePlane dont on parlera plus tard. Cette interface est donc en quelque sorte le modèle d'un état. On a ensuite notre classe principale, la classe Automat qui possède un attribut de type EtatToken. Le grand avantage de ce pattern se trouve dans cette classe. Au lieu d'avoir chaque état dans la classe Automat, on a un seul état type, celui de l'interface. Dès que l'on se trouve sur une certaine figure on change l'état courant et on appelle sa méthode action. La méthode action possède en argument le contexte d'exécution de l'automate, c'est important notamment pour récupérer l'instance courante du streamTokenizer et donc sa dernière position dans le fichier. Voici un exemple de fonctionnement d'un appel d'état

Algorithme 2 : exemple d'appel d'état

```
1 automat.setState(nouveau EtatTriangle()) // on change l'état
2 triangle ← automat.action() // on appel la méthode action du nouvel état (on parse
    la figure) et on stocke son résultat
```

Et voici la méthode action de la classe Automat (qui n'a rien à voir avec celle des états) :

Algorithme 3 : méthode action de la classe automate

```
1 retourner etatToken.action(this)
```

Elle nous sert simplement à appeler la méthode action de l'interface.

3.6.4 Implémentation du pattern template méthod

En POV, la définition d'une sphère et d'un plan est exactement la même d'un point de vue syntaxique. Il suffit de donner des coordonnées, définissant le centre pour la sphère, le vecteur normal pour le plan et un nombre, représentant le centre pour la sphère et la distance pour le plan. En temps normal, cela signifie un état pour la sphère et un pour le plan. Cependant le code à l'intérieur reste le même. Afin de gagner en lisibilité et pour factoriser au mieux le code, nous avons donc eu recours à un autre pattern : le pattern Template Method. L'idée de ce pattern est de crée une classe mère abstraite, la classe EtatSpherePlane. Cette classe définit deux types de méthodes :

— abstraites et qui dépendent donc de la figure
— concrètes et qui ne dépendent donc pas du type de figure

Dans le cas de notre parser, la seule méthode abstraite est celle qui retourne l'objet parsé, la méthode createInstance. C'est normal car on a besoin de savoir si l'on doit renvoyer une sphère ou un plan.

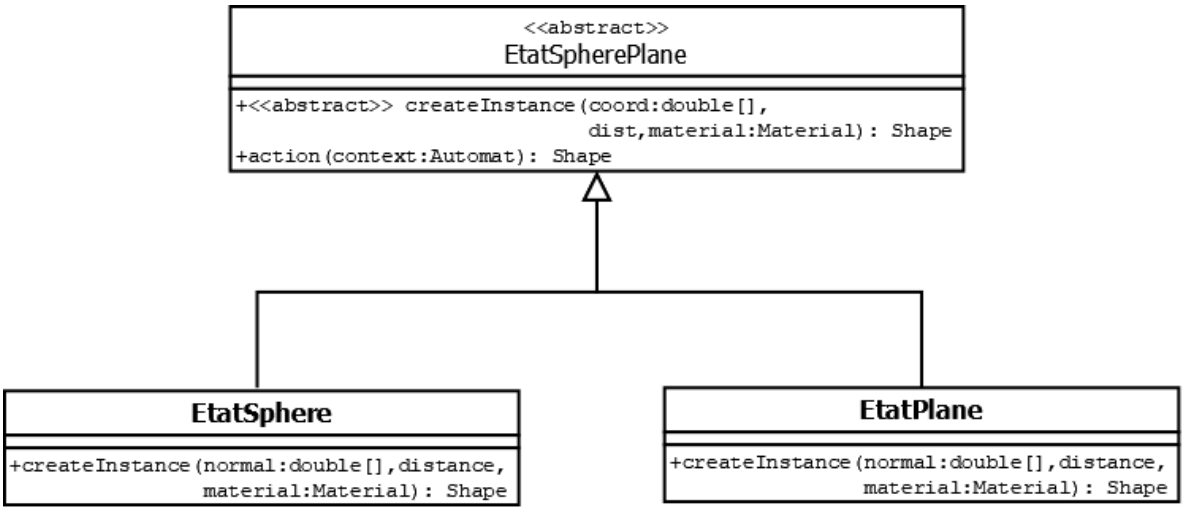


FIGURE 17 – Diagramme du pattern template method appliqué à notre parser

La méthode qui effectue le parsing, au contraire, n'a pas besoin de savoir si c'est une sphère ou plan car la syntaxe est exactement la même et donc l'analyse du bloc de figure est aussi la même.

3.7 Les formes géométriques

4 Analyse des résultats

4.1 Mesure des performances

Comme présenté dans la section 3.3, nous application tire partie de toute la puissnce du processeur de la machine. Le but étant de réduire le temps de calcul, nous présentons dans cette section les gains perçus en terme de performances suite au multithreading de notre application.
Nous comparerons les temps de rendu par image pour différentes résolutions et différents nombre de threads. L'image sera toujours découpée en $nbThread^2$ tuiles.

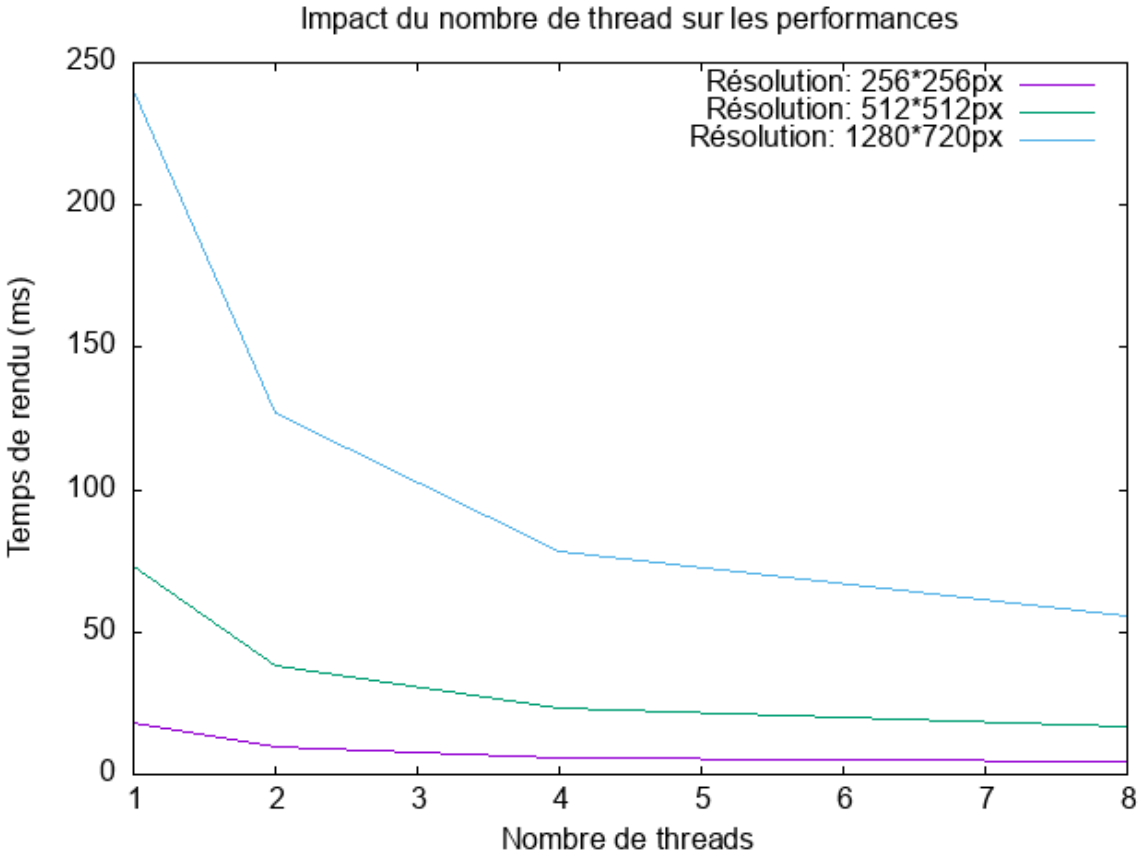


FIGURE 18 – Graph de l'impact du nombre de thread sur le temps de rendu par image

Toutes les mesures ont été faites sur un CPU disposant de 8 processeurs. La scène rendue est la même que celle de la Figure 11.
On observe une réduction significative des temps de rendu quand le nombre de threads utilisé augmente. On passe en effet, pour une résolution de 512*512, de 72.87ms en moyenne à 16.72ms de temps de calcul par image. Cela se traduit par un affichage fluide d'un peu moins de 15 images par seconde avec 1 thread. Avec 8 threads, nous passons à environ 60 images par seconde. Les performances ont donc été multipliées par 4.

4.2 Comparaison avec povRay

5 Idées d'améliorations

La question des performances est au coeur des débats. Bien que notre application soit multithreadée, il n'est pas rare, lors de rendu de scène gourmande, que le temps de calcul d'une image dépasse plusieurs secondes. C'est un comportement potentiellement attendu d'un ray tracer, ces algorithmes étant coûteux en temps de calcul. Nous trouvons toutefois cela dommage car les mouvements de notre caméra ne peuvent alors plus s'effectuer convenablement. Une idée serait alors d'optimiser d'une façon ou d'une autre les temps de calcul de notre algorithme.

6 Conclusion

Ce projet de rendu 3D par lancée de rayon a été très intéressant à réaliser. Grâce à ce travail, nous avons mis en pratique ce que nous avions appris au cours de l'année, mais aussi des éléments que nous n'avions pas étudiés et que nous avons découverts durant le projet. De plus, nous avons observé comment les mathématiques et l'informatique peuvent être mises en commun dans un projet, les cours d'algèbres linéaires de L1 nous ayant été d'une grande aide dans plusieurs aspects du logiciel. Même s'il restera toujours des perfections à apporter, nous sommes plus que satisfaits du résultat obtenu.

Annexes

Annexe A Le ray tracer

A.1 Configuration d'une caméra, d'une scène et de rayons

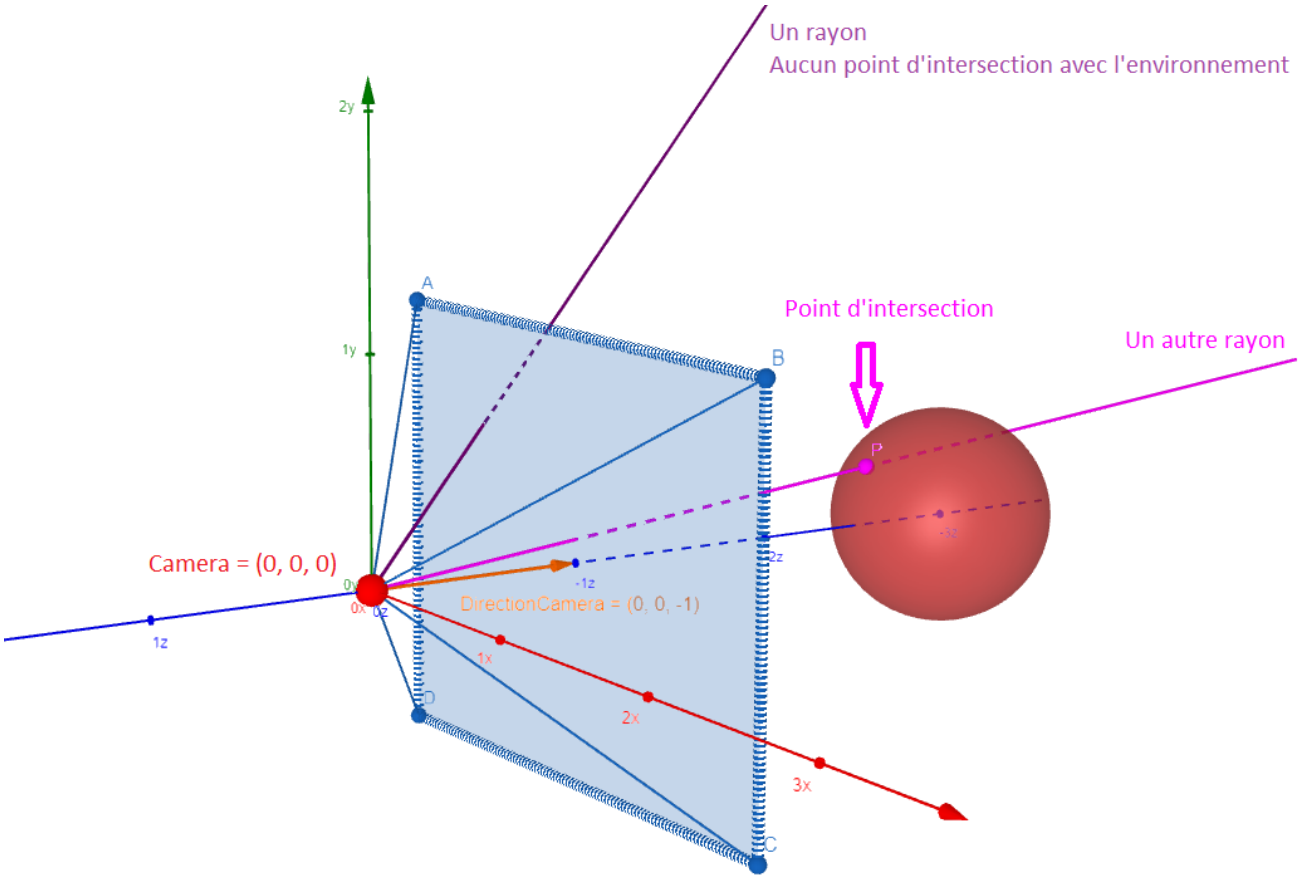


FIGURE 19 – Des rayons sont tirés depuis la caméra dans sa direction de regard. Nous cherchons les points d'intersection avec les objets de la scène.

A.2 Le principe récursif des réflexions

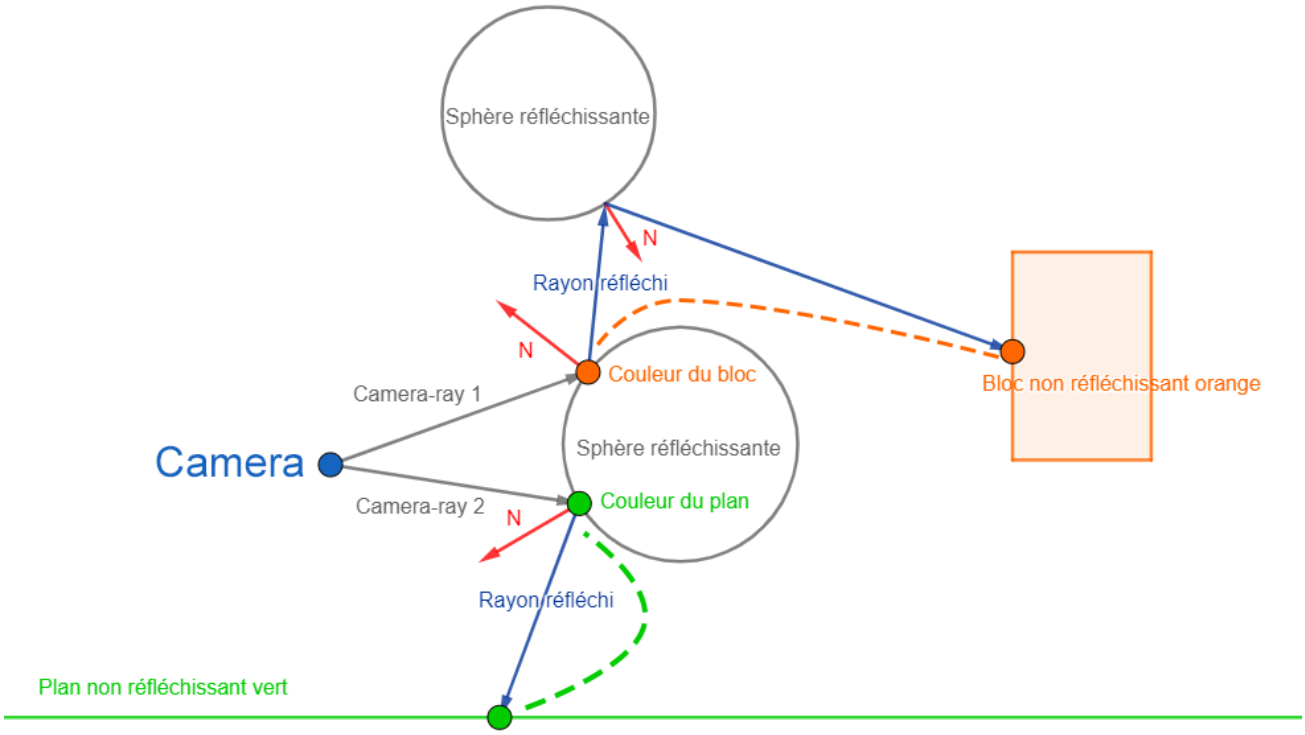


FIGURE 20 – Exemple de rebonds successifs des rayons jusqu'à un objet non réfléchissant

A.3 Rotations de la caméra

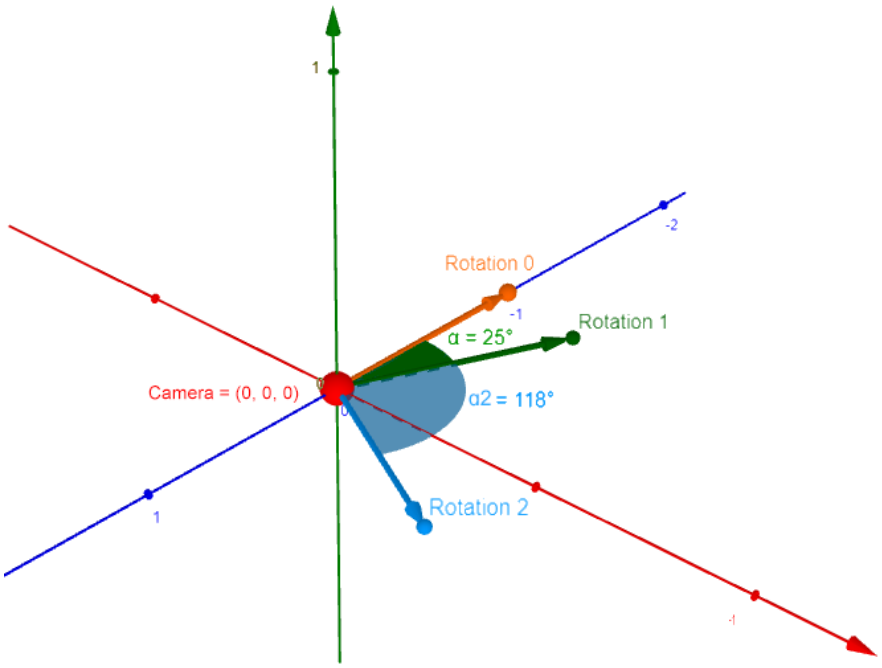


FIGURE 21 – Exemples de rotations de la caméra autour de l'axe Y. 'Rotation 0' est la direction de regard de la caméra avant toute rotation.

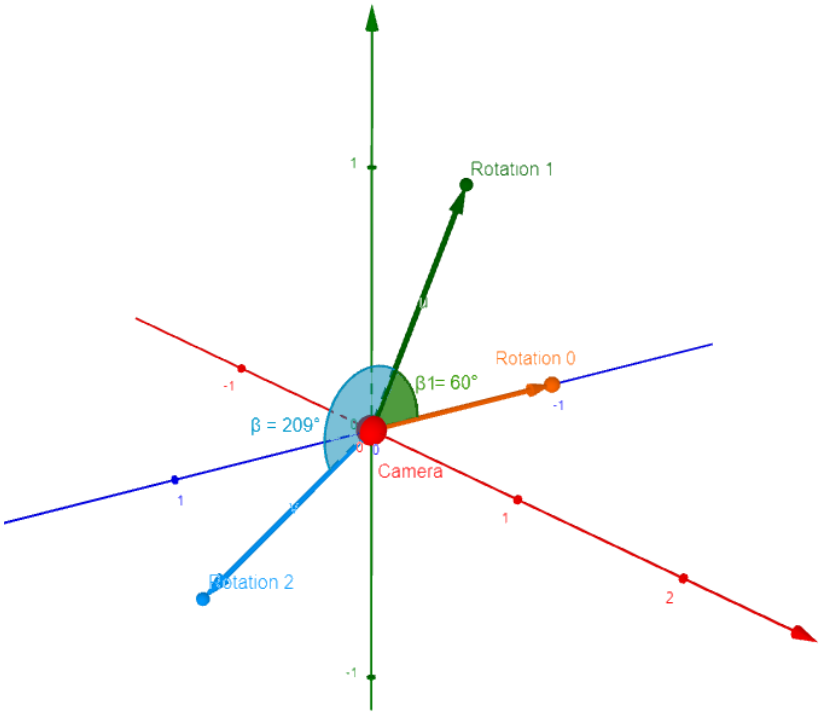


FIGURE 22 – Exemples de rotations de la caméra autour de l'axe X. 'Rotation 0' est la direction de regard de la caméra avant toute rotation.

Références

[1] Matrices de rotation, wikipedia. https://fr.wikipedia.org/wiki/Matrice_de_rotation#En_dimension_trois.

[2] Scratchapixel.com. https://www.scratchapixel.com/.

[3] Wikipedia Phong Reflection Model. https://en.wikipedia.org/wiki/Phong_reflection_model.

[4] MollyRocket - Handmade Ray 01 - Multithreading. https://www.youtube.com/c/MollyRocket. Implémentation de multithreading pour un ray-tracer.

[5] Steve Rotenberg. Fresnel Surfaces. https://cseweb.ucsd.edu/classes/sp17/cse168-a/CSE168_03_Fresnel.pdf, Winter 2017. Publié par University of California San Diego.

[6] demofox. Reflect Refract TIR Fresnel RayT. https://www.shadertoy.com/view/4tyXDR. Référence de rendu visuel.

[7] Prof. Dr. Thorsten Thormählen. http://www.cs.toronto.edu/~jacobson/phong-demo/. Publié par University of Marburg. Référence de rendu visuel.

[8] Position des points de l'icosphere de subdivision 1. http://blog.andreaskahler.com/2009/06/creating-icosphere-mesh-in-code.html.

[9] Design pattern State. https://www.codingame.com/playgrounds/10542/design-pattern-state/introduction.

[10] Patron de méthode, wikipedia. https://fr.wikipedia.org/wiki/Patron_de_m%C3%A9thode.

[11] État (patron de conception), wikipedia. https://fr.wikipedia.org/wiki/%C3%89tat_(patron_de_conception).