Lecrubier Vincent

Année 2007 - 2008

Olivier Solenne

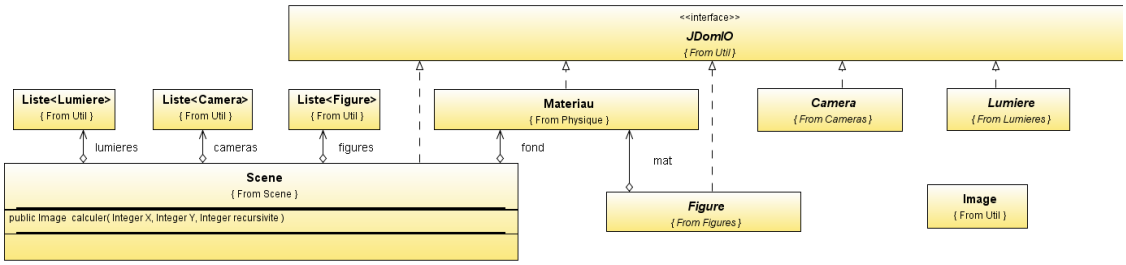
BE Programmation orientée objet

Table des matières

1 Conception

1.1 Architecture générale

1.1.1 Architecture de la partie rendu d'image

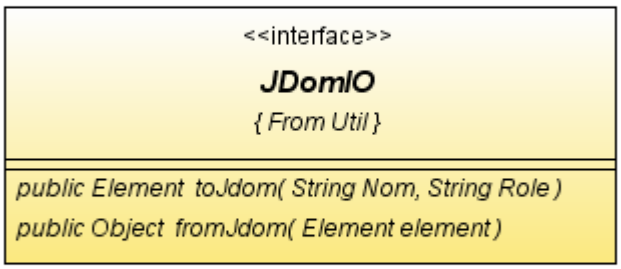


La classe centrale pour le rendu d'une image est la classe Scene qui contient tous les éléments permettant le calcul de l'image : Lumières, cameras et figures. La classe scène a donc été choisie pour l'implantation de « cœur » du logiciel : le code de calcul par lancer de rayons.

Les scènes sont importées depuis le format XML grâce à l' API Jdom.

1.2 Interfaces principales

1.2.1 JdomIO



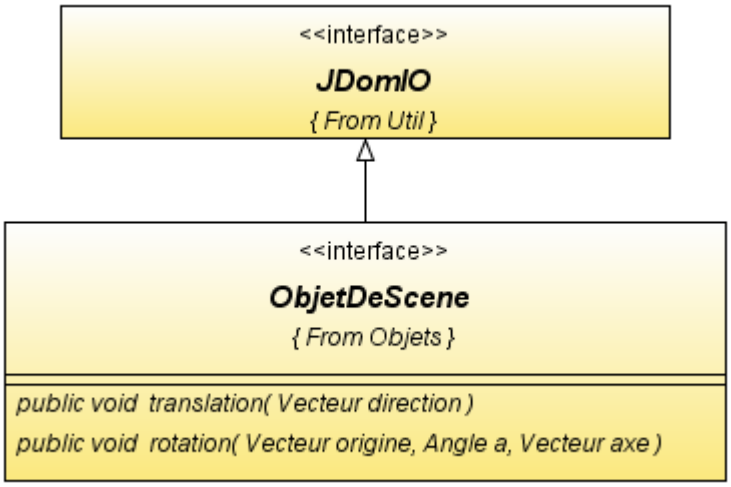
L'interface JdomIO permet d'importer et d'exporter les objets vers un arbre Jdom. Il est ensuite possible d'exporter facilement l'arbre Jdom vers un fichier XML, ou d'importer un fichier XML.

La méthode toJdom permet d'exporter l'objet sur lequel la méthode est appliqué, elle renvoi une instance de org.Jdom.Element, qui correspond aux nœuds de l'arbre Jdom.

La méthode fromJdom permet d'importer un objet depuis une instance de org.Jdom.Element, elle renvoie l'objet, mais l'objet sur lequel la méthode est appliquée n'est pas modifié, son seul rôle est de définir le type de l'objet à importer.

L'implémentation des ces méthodes est assez simple si les attributs de la classe sont des instances d'une classe qui implémente JdomIO, il suffit alors de faire appel à l'interface JdomIO pour chaque attribut. L'arbre Jdom est ainsi construit de manière itérative, à moindre frais.

1.2.2 ObjetDeScene

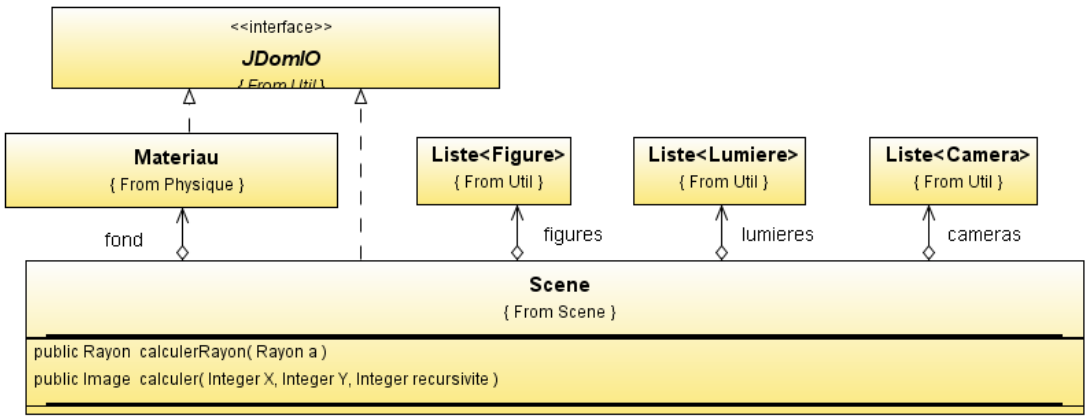


L'interface ObjetDeDcene doit être implémentée par tous les objets qui font partie d'une scène (Lumières, Cameras, Figures). Elle contient les méthodes importantes pour la manipulation de ces objets.

Cette interface étend JdomIO, En effets touts les objets de scene doivent impérativement pouvoir être importés et exportés en XML.

1.3 Descriptions des principales classes

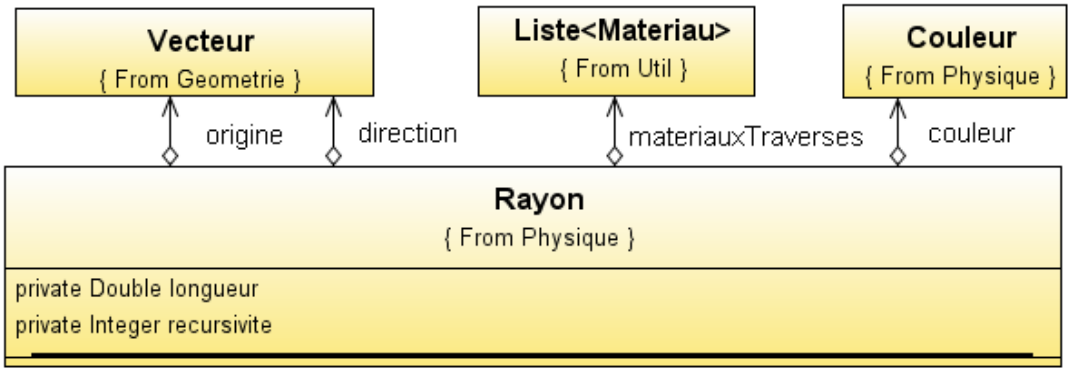
1.3.1 Scene



La classe Scene contient des listes de figures, lumières et caméras. A partir de ces données, la méthode calculer permet de rendre une image. La méthode calculer utilise la méthode calculerRayon pour chaque pixel. La méthode calculerRayon calcule la couleur d'un rayon à partir des objets contenus dans la scène.

La classe Scene contient aussi un attribut fond, qui est une instance de Materiau. Ce matériau défini le matériau dans lequel baigne la scène. Le plus souvent le matériau fond est l'air, le logiciel permet cependant de personnaliser ce matériau pour rendre des images qui peuvent êtres prises sous l'eau par exemple. Si le matériau fond a une atténuation trop forte, l'image sera bien entendu sans intérêt, puisqu'aucun objet ne sera visible. Le matériau fond généralise la notion de lumiere ambiante, par exemple.

1.3.2 Rayon



La classe Rayon ne contient aucune méthode importante, mais contient toutes les informations relatives à un rayon, On retrouve ainsi son origine, sa direction (Vecteur de normalisé), sa couleur, sa longueur.

On trouve l'attribut récursivité qui est un entier représentant le nombre de générations de rayons que ce rayon peut engendrer. Ainsi lors d'une collision du rayon avec la scène, le rayon engendre plusieurs fils : les rayons réfléchis et réfractés. Ces rayon auront une récursivité décrémentée de 1 par rapport au rayon parent. Ceci permet de limiter la propagation des rayons, et réduire le temps de calcul d'une scène à une valeur finie.

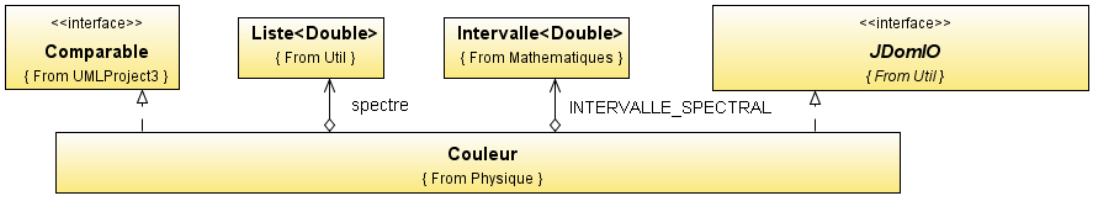
On trouve aussi l'attribut materiauxTraversés, qui est une liste de matériaux. Cet attribut permet de déterminer quel matériau est traversé par le rayon. Un exemple permettra de comprendre l'utilisation de cet attribut :

1. Le rayon est lancé depuis une caméra de la scène : la liste des matériaux traversés par le rayon est alors : {fond}
2. Le rayon entre en collision avec un objet translucide : le rayon réfracté, qui sera situé à l'intérieur de l'objet translucide, aura alors l'attribut MateriauxTraversés égal à {fond, MateriauTranslucide}
3. Le rayon refracté ressort de l'objet translucide. On supprime alors l'élément « MateriauTranslucide » de la liste des matériaux traversés par le rayon ressortant de l'objet, qui aura donc son attribut MateriauTraversés égal à {fond}.

Cette approche permet d'éviter les incohérences dans la description du materiau traversé par le rayon, et permet de gérer sans problème le cas ou deux objets se superposent, ou sont inclus l'un dans l'autre.

Dans tous les cas, seul le dernier élément de la liste de MatériauxTraversé est utile pour le calcul d'un rayon, il représente le matériau dans lequel le rayon se déplace réellement. Les autres éléments permettent de déterminer les matériaux dans lesquels se propageront les rayons fils du rayon à calculer.

1.3.3 Couleur



La classe Couleur est l'une des originalités de ce logiciel de lancer de rayons. En effet, nous avons choisi de décrire les couleurs non pas par 3 composante RGB, ou TSL par exemple. Afin d'obtenir un calcul plus proche de la réalité physique, et plus précis, nous avons choisi de décrire les couleurs par un spectre de précision arbitraire. Il existe ainsi plus de subtilité dans le processus de calcul des réflexions et refraction par exemple.

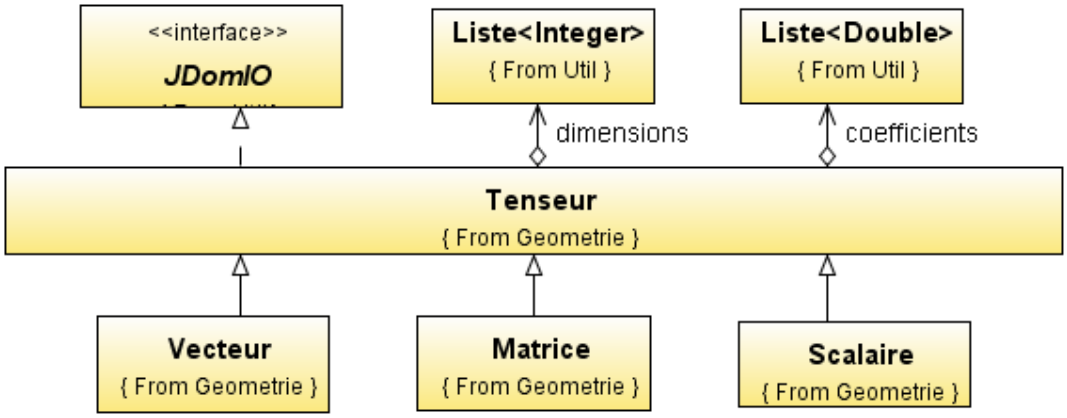
La classe couleur implémente bien entendu l'interface JdomIO, ce qui permet à l'utilisateur de définir des couleurs personnalisées.

Concrètement, la classe couleur comporte un attribut spectre qui est une liste de réels correspondant à l'amplitude de l'onde électromagnétique se déplaçant selon la direction du rayon. La constante INTERVALLE_SPECTRAL représente l'intervalle de longueurs d'onde pour lequel le spectre est défini. Pour rendre des images en lumière visible, cet intervalle est fixé à [400nm, 800nm]. Il est ainsi aisé de modifier cette constante afin de rendre des images selon des longueurs d'onde différentes, ou un spectre plus étendu.

La précision des couleurs, et donc la longueur de la liste représentant le spectre est fixe. Nous l'avons fixé à 12, ce qui permet d'être beaucoup plus précis qu'avec seulement 3 composantes, mais ne ralenti quasiment pas le calcul. Les test ont montré que le temps de calcul est aux erreurs de mesure près identique avec 36, 12 ou 3 composantes.

Ce choix nous a semble intéressant, car il permet d'envisager des évolutions intéressantes pour le logiciel. Il nous a cependant demandé un temps considérable. Ainsi, afin de pouvoir afficher les couleurs à l'écran, nous avons du concevoir des méthodes permettant de transformer les instances de la classe Couleur en instances de la classe java.awt.Color. Nous avons aussi dû développer la classe util.Distributions qui permet de définir des couleurs à l'aide de distributions spectrales, par exemple la méthode CouleurGauss permet de créer une couleur dont la répartition spectrale suit une loi normale centrée sur une longueur d'onde, avec une largeur de bande donnée etc...

1.3.4 Tenseur



La classe Tenseur est elle aussi une spécificité de ce logiciel de lancer de rayons. En effet, nous avons choisi de généraliser la notion de scalaire, matrice et vecteur. Nous avons pour cela développé une classe Tenseur très puissante, puisqu'elle permet d'utiliser l'algèbre tensorielle sans limitation. Ainsi il est possible de calculer le produit triplement contracté d'un tenseur d'ordre 27 et d'un autre d'ordre 13, en dimension 42 sans que cela ne pose de problème autre que le temps de calcul et la mémoire.

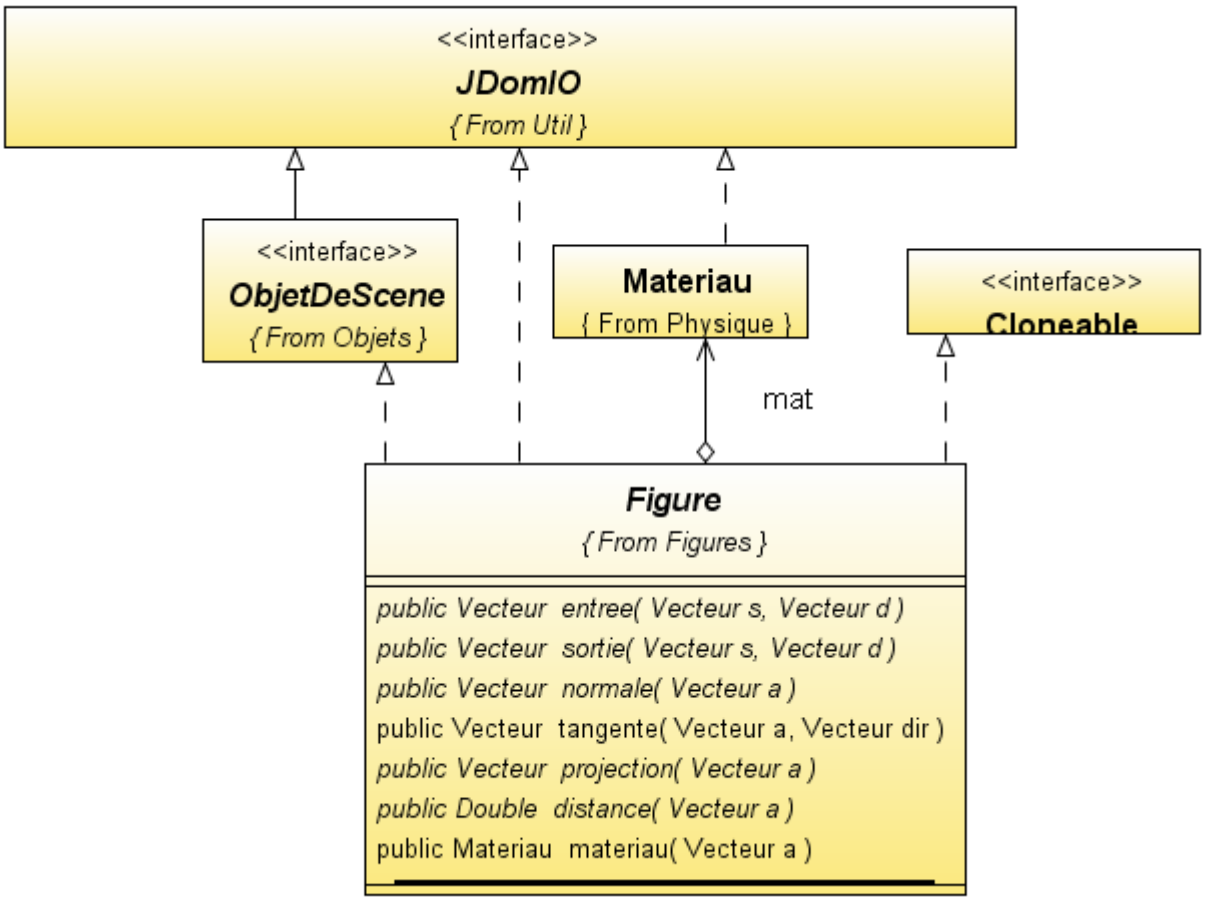
Ce choix, bien qu'intéressant, est plus discutable que le choix effectué pour la représentation des couleurs, car il s'avère que c'est le principal frein à la rapidité de calcul des images. En effet, un tenseur se compose d'une liste de coefficients et d'un liste de dimensions. Ainsi pour représenter un vecteur, au lieu de trois réels, il faudra ici une liste d'entiers : {3} et une liste de réels {x,y,z}. Nous avons donc en quelque sorte choisi d'utiliser un marteau pour écraser une mouche.

Ce choix laisse cependant la possibilité séduisante d'amélioration du programme, afin de permettre le rendu de scène de dimensions supérieures à 3. Il serait par exemple possible de calculer la projection en 2D d'un hypercube de dimension 5 sans avoir besoin de modifier profondément le programme.

Nous avons spécialisé la classe Tenseur en Vecteur et Matrice qui représentent des matrices et vecteur tridimensionnels, permettant de rendre des scènes « classiques ». Il est important de noter que les calculs effectués sur les vecteurs et matrice n'utilisent que les méthodes de Tenseur, ainsi, nous n'avons pas eu besoin de coder le produit d'une matrice par un vecteur, puisque cette opération est généralisée par le produit tensoriel contracté.

Le problème de la lenteur du calcul pourrait donc être assez facilement résolu, comme nous le verrons au §4.2.1 : Optimisations (p.23).

1.3.5 Figure



La classe abstraite figure représente un figure de la scène, quel que soit son type (Plan, sphère, cylindre etc...) Elle contient les méthodes qui sont indispensables au calcul de la représentation graphique de la figure. Bien entendu, elle implémente JdomIO et Objet de Scene. Les méthodes sont les suivantes :

- Entrée : renvoie le premier point d'entrée d'un rayon issu du point s, dirigé selon d, dans la figure, si ce point existe.
- Sortie : donne le premier point de sortie de ce même rayon, s'il existe.
- Normale : donne le vecteur normal à la surface de la figure, au niveau de la projection du point a sur la surface de la figure. La normale est dirigée de l'intérieur vers l'extérieur de la figure.
- Tangente : donne le vecteur tangent à la surface de la figure, au niveau de la projection du point a sur la surface de la figure. Le résultat vérifie la propriété : la normale, la tangente et dir sont coplanaires.
- Projection : donne la projection du point a sur la surface de la figure, c'est le point appartenant a la figure le plus proche du point a.
- Distance : calcule la distance entre le point a et sa projection sur la figure, négatif si le point a est intérieur à la figure, nul s'il est à sa surface, positif s'il est extérieur.
- Materiau : renvoie le materiau situé au point a sur la figure. (Methode quasi

inutile dans la version actuelle, mais permettant l'évolutivité du logiciel, car facilite la gestion de textures).

Ces méthodes doivent être codées pour chaque nouveau type de figure.

Il est intéressant de remarquer que cette définition d'un figure permet de dégager le concept d'intérieur et d'extérieur d'une figure. Ainsi, il nous a été très simple de créer les figures Union et Intersection, auxquelles nous n'avions pas pensé au départ, mais qui sont très intéressantes puisqu'elles permettent de créer des figures complexes à partir de figures simples.

Illustration 1: Union de deux cylindres infinis

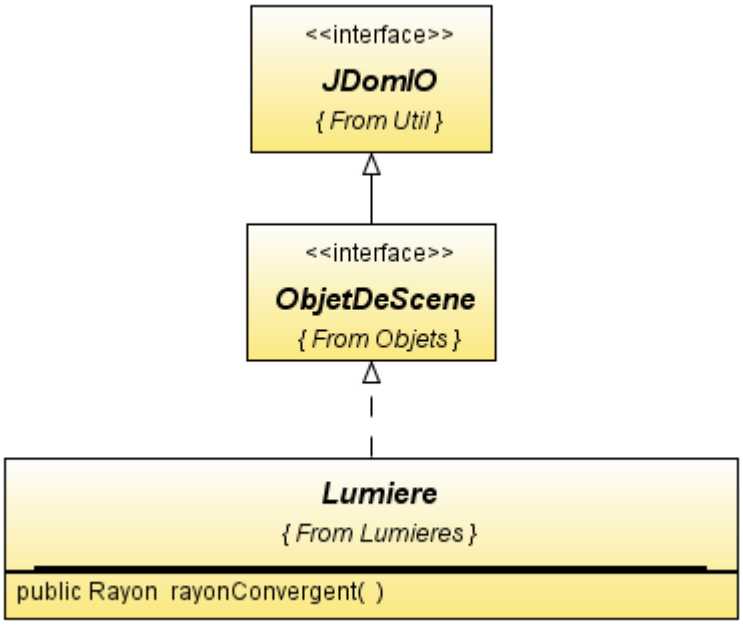
Les différents types de figures présents dans la version actuelle du programme sont :

- Plan : un plan infini défini par un point et une normale, représentant un demi espace plein.
- Plan3Points : idem, mais défini avec 3 points.
- Sphere : une sphère pleine.
- Cylindre : un cylindre infini plein.
- Union : l'union de plusieurs figures.
- Intersection : l'intersection de plusieurs figures.
- Négation : la négation d'une figure, l'intérieur et l'extérieur sont échangés.

Cette approche évite elle aussi les incohérences, ainsi chaque figure est dotée d'un intérieur et d'un extérieur, seuls des solides peuvent être représentés. En effet le concept de Facette, bien qu'intéressant pour le rendu rapide, n'est pas physique, et peut causer des incohérences, l'intérieur et l'extérieur d'un objet n'étant plus définis.

Ce concept est ici naturellement éliminé car quiconque voudrait créer une classe Facette réalisant Figure serait bien embêté pour coder les classes entrée et sortie de manière cohérente.

1.3.6 Lumiere

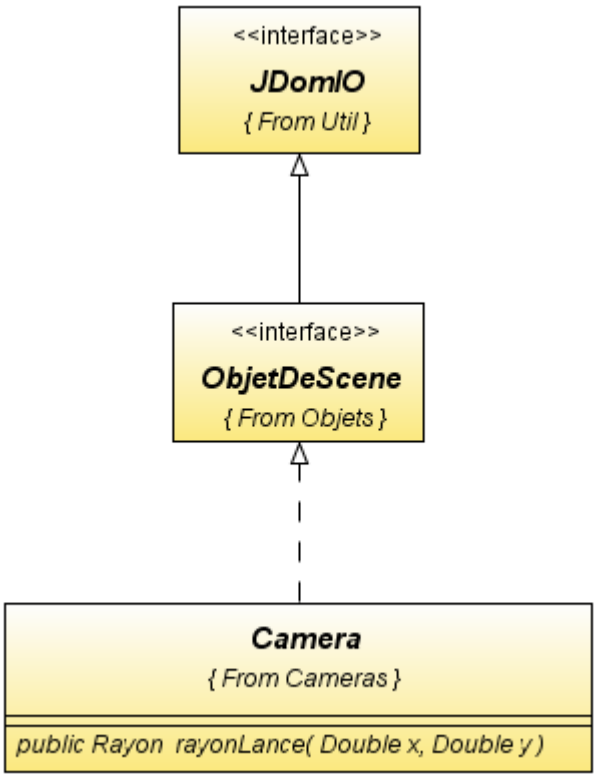


La classe abstraite Lumière représente une Lumière. Elle a un attribut Couleur qui représente la couleur émise par la lumière.

La méthode abstraite rayonConvergent renvoie le rayon convergent vers la lumière depuis le point a. Cette méthode doit être codée pour chaque réalisation de Lumière, ainsi, nous avons réalisé la classe LumiereSimple qui est une lumière ponctuelle, le rayon convergent sera donc dirigé vers le point ou se situe la lumière. On pourrait imaginer une lumière directionnelle, ou linéaire, simplement en modifiant la méthode rayonConvergent.

Les lumières implémentent l'interface JdomIO afin de pouvoir être décrites par l'utilisateur.

1.3.7 Camera

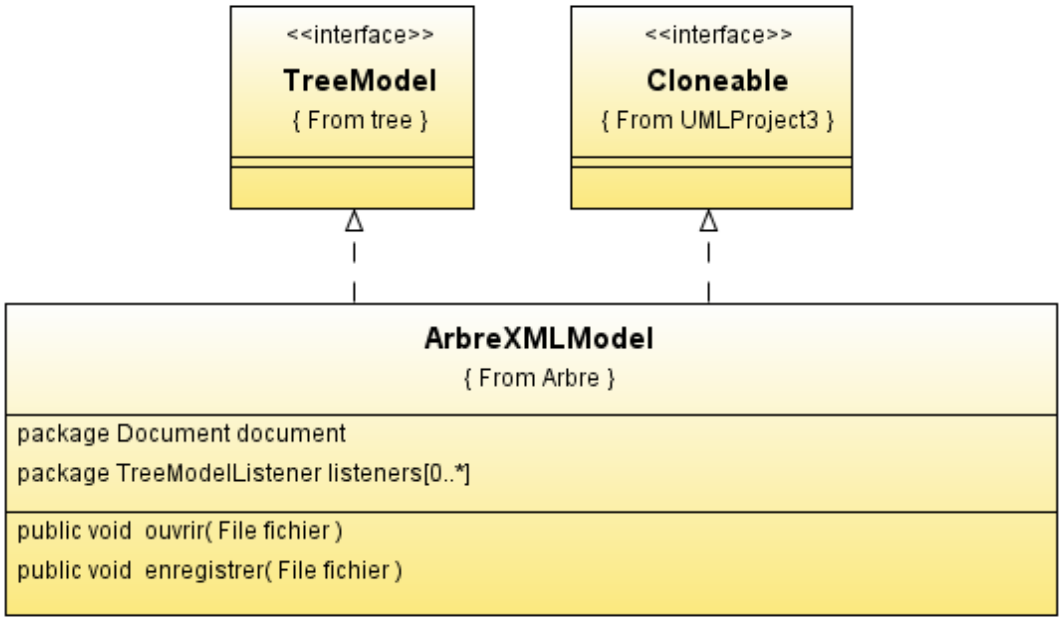


La classe abstraite Camera représente une caméra, permettant un rendu sous forme d'image. Sa seule méthode est rayonLance(x,y), qui retourne le rayon issu de la caméra, en fonction des coordonnées du pixel de l'image.

Nous avons créé la classe CameraSimple qui représente une caméra simple, dont l'angle d'ouverture est modificable.

Les caméras implémentent bien entendu JdomIO.

1.3.8 ArbreXMLModel

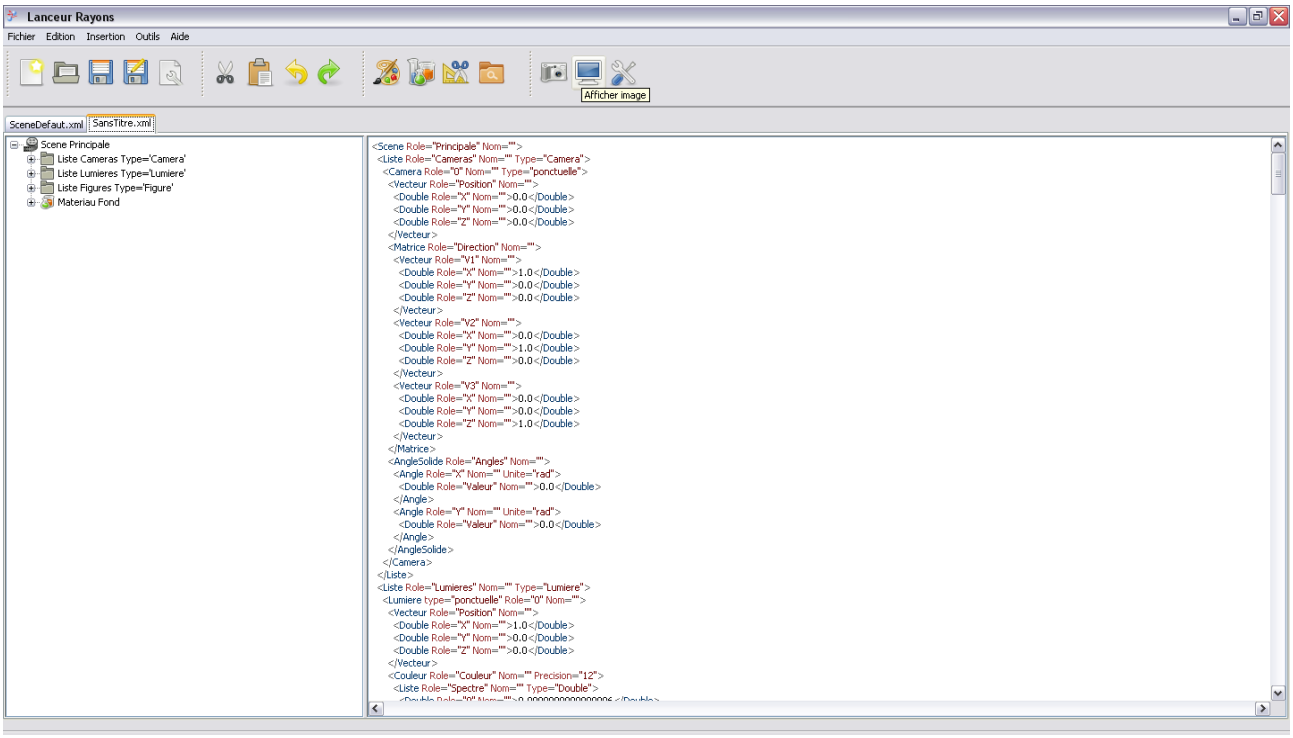


La classe ArbreXMLModel est importante puisqu'elle est au centre de la relation entre l'utilisateur et l'algorithme de calcul de scène. En effet, cette classe implémente l'interface TreeModel, qui est utilisée par le composant Swing Jtree afin d'afficher un arbre. L'attribut Document représente un document XML interprété lors de l'appel de la méthode ouvrir.

ArbreXMLModel réalise donc la transition entre les fichiers XML et les instances des classes implémentant JdomIO.

1.4 Interface homme-machine

1.4.1 Fenêtre principale



L'interface homme machine se compose d'une fenêtre principale comportant plusieurs zones :

- La barre de menus : présente toutes les actions possible pour l'utilisateur sous la forme de menus déroulants classiques.
- La zone d'outils : comporte plusieurs barres d'outils permettant a l'utilisateur d'acceder rapidement aux fonctionnalités les plus courantes.
- La barre d'état comporte une zone d'aide indiquant le rôle de l'élément survolé par le curseur (ToolTipText), une zone de texte indiquant la tache en cours en arrière plan, ainsi que les avertissements et informations utiles, et une barre de progression indiquant la progression de la tache en cours en arrière plan.
- La zone de travail comportant autant d'onglets qu'il y a de documents ouverts. Chaque document est présenté sous la forme d'un arbre représentant l'instance de ArbreXMLModel correspondant au document actuel, et d'une zone de texte représentant le fichier XML en cours de modification.

1.4.2 Fenêtres annexes

Les fenêtres annexes ont été conçues dans la continuité de la fenêtre principale, afin de ne pas déstabiliser l'utilisateur. Ces fenêtres sont :

- La fenêtre de visualisation d'images.
- La fenêtre de la bibliothèque d'objets.
- Les boites de dialogues basées sur les composants Swing JoptionPane et JfileChooser.

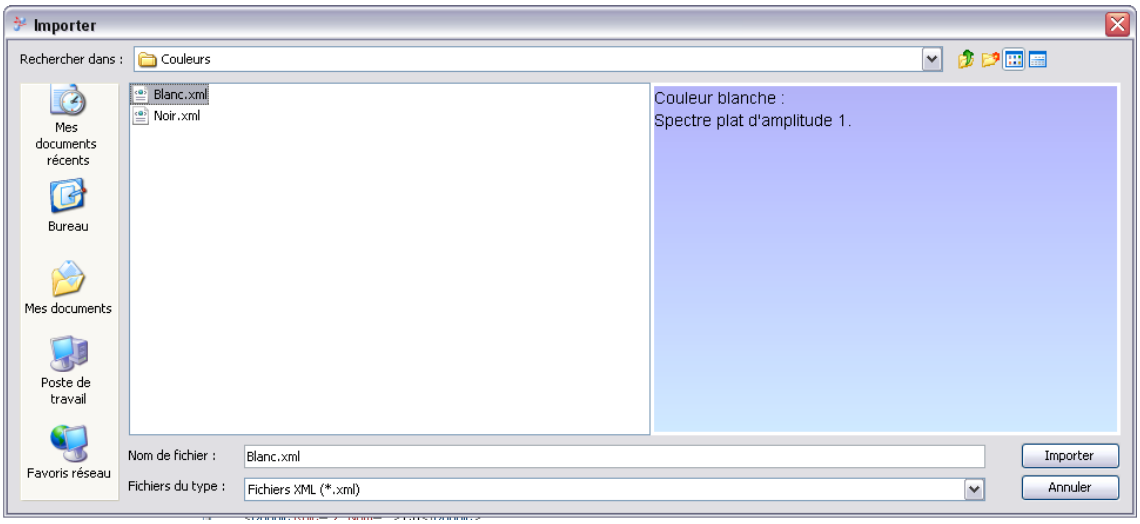


Illustration 2: La fenêtre de la bibliothèque d'objets

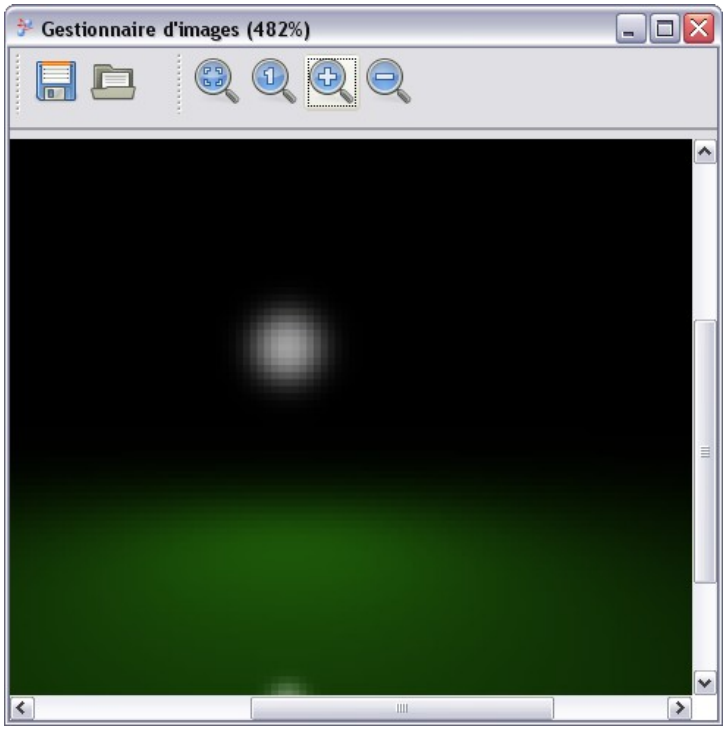


Illustration 3: La fenêtre de visualisation d'images

2 Manuel utilisateur

2.1 Utilisation de l'interface graphique

2.1.1 Utilisation générale

L'interface graphique a été conçue dans un soucis de clarté et de fonctionnalité. Son fonctionnement est classique et demande un temps d'apprentissage assez réduit.

Les principaux éléments à comprendre sont les suivants :

- Dans la version actuelle, un document n'est éditable que via sa représentation sous forme d'arbre, l'édition de texte est désactivée, seul l'affichage fonctionne.
- La modification de l'arbre fonctionne, mais est encore limitée à l'édition des composants de base de chaque objet, c'est à dire les éléments qui sont des feuilles dans l'arbre de représentation d'une scène.
- Dans ce contexte, la bibliothèque joue un rôle primordial, c'est pourquoi nous allons la décrire plus en détail.

2.1.2 Bibliothèque

La bibliothèque est une fonctionnalité intéressante de l'IHM, qui s'appuie sur la flexibilité offerte par l'orientation objet. Elle permet d'insérer des objets à partir de modèles prédéfinis. Il est ensuite possible de modifier ces objets, qui servent de support à la création d'objets personnalisés.

2.2 Modèles

2.2.1 Documents XML

La bibliothèque permet de se faire une idée de la structure des documents XML.

3 Tests

3.1 Tests unitaires

3.1.1 Tenseurs

Les tests sur les tenseurs nous ont permis de verifier le bon fonctionnement de cette classe comme dans cet exemple :

```
////////////////Test vecteurs//////////////////////
        Vecteur a = new Vecteur(10.,1230.,540.);
        System.out.println("Vecteur a                  : " +
a);
        System.out.println("Vecteur a normalisé        : " +
a.normaliser());
        System.out.println("Norme du vecteur normalisé : " +
a.normaliser().norme());
        System.out.println("Vecteur a après rotation   : " +
a.rotation(Vecteur.zero, Angle.Deg(90.),Vecteur.e1));
////////////////////////////////////
```

Résultat :

```
Vecteur a                  : [ 10.0 1230.0 540.0 ]
Vecteur a normalisé        : [ 0.007444054147293959
0.9156186601171569 0.40197892395387375 ]
Norme du vecteur normalisé : 0.9999999999999998
Vecteur a après rotation   : [ 540.0 1230.0
-9.999999999999966 ]
```

3.1.2 Couleurs

Afin de tester les couleurs, nous avons ajouté une méthode afficherSpectre à la classe Couleur. Voici par exemple le résultat d'un test sur l'addition et la multiplication de deux Spectres de couleurs gaussiennes. Le résultat parle de lui même : ces opérations fonctionnent correctement.

```
////////TestCouleurs/////////////////////////////////////
        Couleur.PRECISION = 12;
        Couleur test1 = new Couleur().CouleurGauss(500.,
                                        100., 1.);
        Couleur test2 = new Couleur().CouleurGauss(700., 50.,
                                        1.);
        Couleur test3 = test1.somme(test2);
        Couleur test4 = test1.multiplication(test2);
        Double k = .9/test3.normeInf();
        Ecran a = new Ecran("Test", 800,600,400);
        test1.afficherSpectre(a,k);
        test2.afficherSpectre(a,k);
        test3.afficherSpectre(a,k);
        test4.afficherSpectre(a,k);

////////////////////////////////////////////////////
```

Résultat :

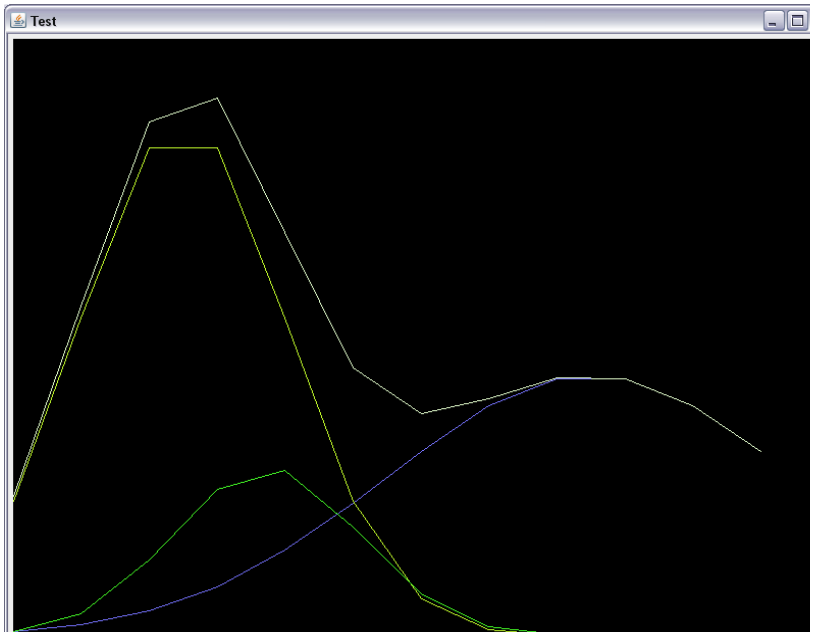


Illustration 4: Résultat d'un test sur les couleurs

3.1.3 Figures

Nous avons effectué des tests unitaires sur le figures, comme ci dessous :

```
//////////////Test figures////////////////
        Sphere a = new Sphere(Vecteur.zero,1.);
        System.out.println(a.entree(new
Vecteur(10.,0.,0.),Vecteur.ne0));
        System.out.println(a.sortie(new
Vecteur(10.,0.,0.),Vecteur.ne0));
////////////////////////////////////
```

Résultat :

```
Entrée : [ 1.0 0.0 0.0 ]
Sortie : [ -1.0 0.0 0.0 ]
```

3.2 Tests de rendu

3.2.1 Modèles d'éclairement

De nombreux tests ont été effectués sur les modèles d'éclairement, afin d'atteindre un rendu réaliste. Ainsi les trois illustration suivante montrent une même scène, composée d'un plan vert, d'aspect « lustré », surplombé par une lumière ponctuelle blanche, rendue par trois modèles d'éclairement successifs, les deux premiers contenant des erreurs, corrigées dans le troisième.

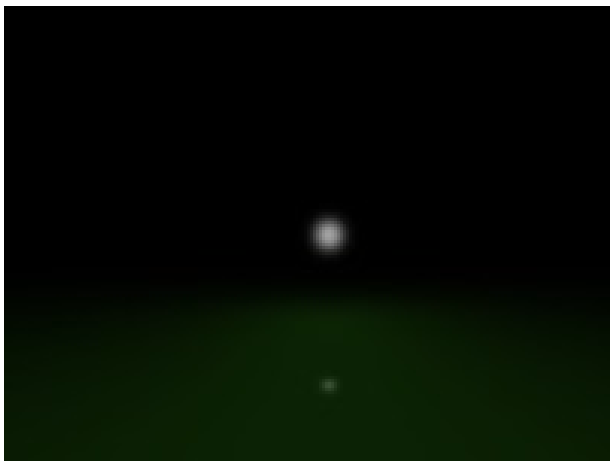


Illustration 5: Un modèle d'éclairement erroné

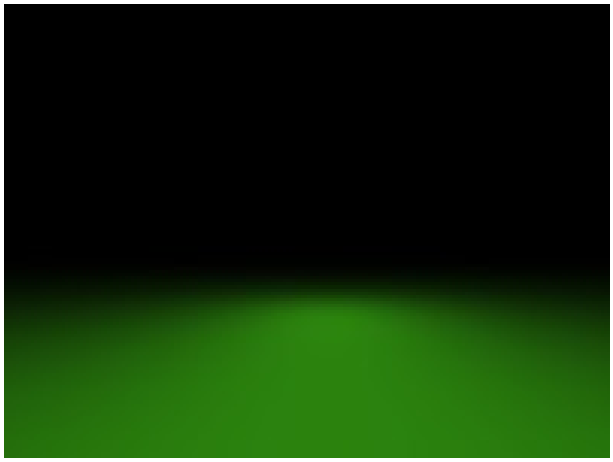


Illustration 6: Un autre modèle d'éclairement erroné

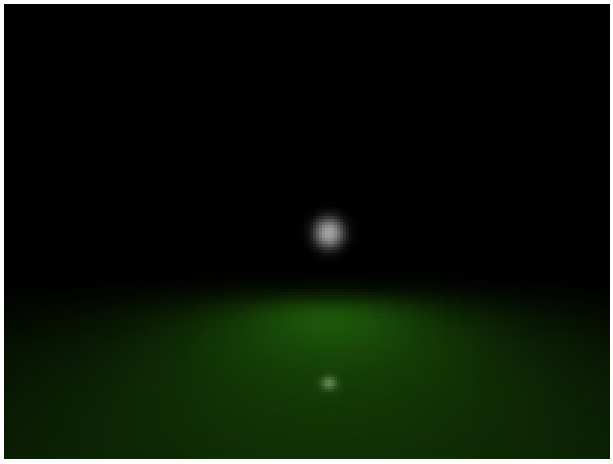


Illustration 7: Le modèle d'éclairement correct

3.2.2 Reflexion et refraction

Les test visuels de réfraction et réflexions nous ont permis de résoudre certains problèmes qui auraient pu passer inaperçus. Il faut toujours être vigilant face aux bogues, même lorsque tout le travail semble être accompli sans problème, ils peuvent toujours exister.

Une galerie présentant différents essais est fournie en annexe.

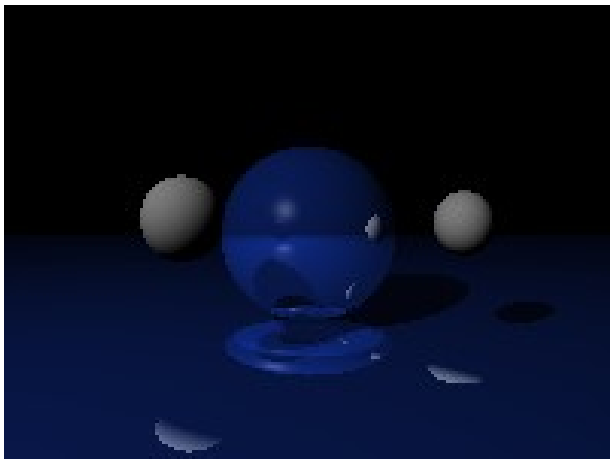


Illustration 8: Reflets incorrects

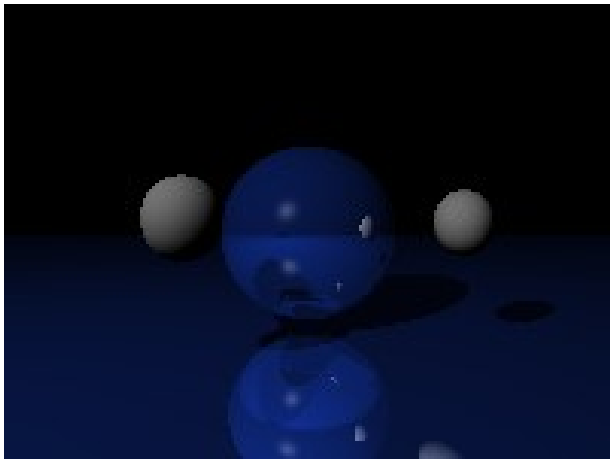


Illustration 9: Reflets corrects

3.3 Tests de lecture/écriture

3.3.1 Fichiers XML et JDom

Les tests de lectures et écriture de scènes on été effectués en même temps que les tests de rendu, et se sont avérés concluants. La modification d'un fichier entraine bien la modification de la scène rendue. Tout a fonctionné correctement dès les premiers essais, grâce à l'interface Jdom.

3.3.2 Jdom et objets de scène

L'implémentation de l'interface JdomIO sur les objets a parfois été difficile au début. Il a été nécessaire de faire de nombreux essais afin de valider l'équivalence entre le contenu de l'arbre DOM et le contenu de l'objet à rendre. Nous avons pour cela procédé à des tests sur certains objets, afin d'obtenir un code efficace et dépourvu de bogues. Nous avons ensuite extrapolé ce code pour les objets semblables, en conservant sa structure.

4 Limites du logiciel

4.1 Limites

4.1.1 Vitesse

Lors de la conception du programme, nous avons choisi d'opter pour un calcul proche de la réalité, et une architecture souple et extensible. Ainsi, l'utilisation de couleurs définies par leur spectre (plusieurs composantes au lieu de 3 pour les couleurs RGB classiques), et de tenseurs afin de généraliser la notion de vecteur et de matrice, sont deux choix qui, s'ils ont des avantages que nous avons expliqués, ralentissent considérablement la vitesse de calcul.

Hot Spots - Method	Self time [%]	Self time	Invocations
LanceurRayon.Util.Geometrie.Tenseur.**getPosition** (LanceurRayon.Util.Liste)		1735 ms (13,2 %)	1506718
LanceurRayon.Util.Geometrie.Tenseur.**produit** (LanceurRayon.Util.Geometrie.Tenseur)		1209 ms (9,2 %)	29917
LanceurRayon.Util.Geometrie.Tenseur.**longueur** ()		991 ms (7,5 %)	2240490
LanceurRayon.Util.Liste.**getPart** (Integer, Integer)		974 ms (7,4 %)	741964
LanceurRayon.Util.Geometrie.Tenseur.**getDouble** (LanceurRayon.Util.Liste)		901 ms (6,8 %)	1135687
LanceurRayon.Util.Geometrie.Tenseur.**ordre** ()		831 ms (6,3 %)	2380312
LanceurRayon.Util.Liste.**lire** (Integer)		601 ms (4,6 %)	1410066
LanceurRayon.Util.Liste.**<init>** ()		552 ms (4,2 %)	1383644
LanceurRayon.Util.Liste.**getFrom** (Integer)		447 ms (3,4 %)	370982
LanceurRayon.Util.Geometrie.Tenseur.**getIndice** (Integer)		333 ms (2,5 %)	370982
LanceurRayon.Util.Geometrie.Tenseur.**contraction** (Integer)		326 ms (2,5 %)	29916
LanceurRayon.Util.Liste.**getTo** (Integer)		322 ms (2,4 %)	370982

Illustration 10: Répartition du temps de calcul entre les différentes méthodes (obtenu avec NetBeans)

4.1.2 Utilisation mémoire

Pour les mêmes raisons que celles citées pour la vitesse de calcul, l'utilisation mémoire est parfois trop importante. Il peut arriver que l'erreur Java heap space soit levée.

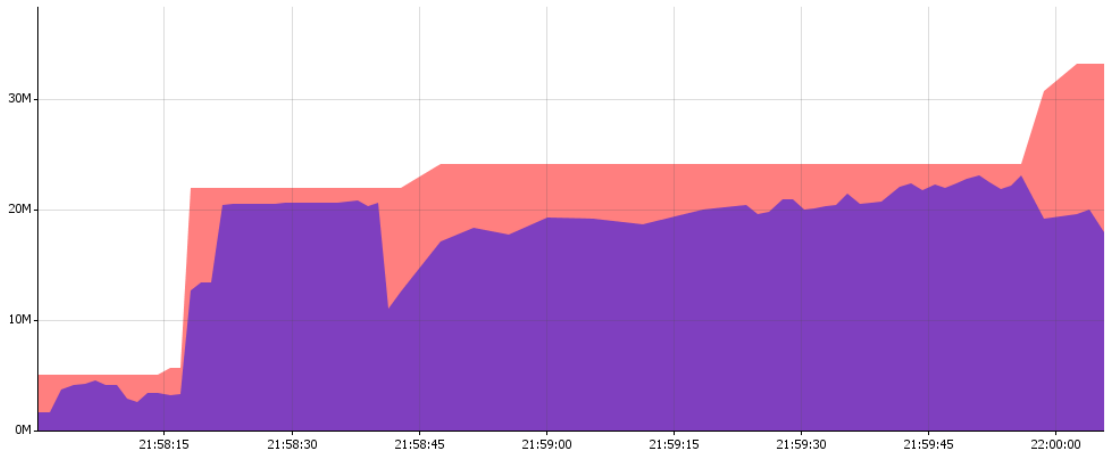


Illustration 11: Heap size, used heap lors du calcul d'une scène (obtenu avec NetBeans)

4.1.3 Rendu

Les limites du logiciel en matière de rendu sont dues au choix d'un algorithme de lancer de rayon, qui, par nature ne permet pas de rendre compte de la totalités des phénomènes lumineux. Afin d'obtenir un rendu plus proche de la réalité physique, il faudrait ajouter des fonctionnalités, telles que la radiosité, ou le photon mapping.

De plus, la version actuelle du logiciel ne gère pas les milieux dispersifs, c'est à dire les milieux dont l'indice de réfraction dépend de la longueur d'onde. Ainsi la proprieté IndiceRefraction, qui est représentée par un spectre, n'est en fait interprétée que comme un réel égal à la valeur moyenne du spectre. Le problème des milieux dispersifs en lancer de rayon est qu'ils nécessitent un rayon différent par composante du spectre. Ainsi, afin de conserver un temps de calcul correct, nous avons volontairement bridé cette possibilité, qui engendrerait rapidement des milliers de rayons différents.

4.2 Ameliorations possibles

4.2.1 Optimisations

Les problèmes de temps de calcul et d'utilisation mémoires peuvent être réduits dans des proportions importantes en optimisant le logiciel. On constate sur l'Illustration 10 que la majeure partie du temps de calcul est utilisé a calculer des produits tensoriels, en particulier des produits scalaires de vecteurs et des produits de matrices par des vecteurs. Il serait par exemple possible de redéfinir la méthodes produitScalaire entre deux vecteurs afin de ne pas avoir à utiliser la définition tensorielle d'un vecteur, ce qui permettrait un calcul plus rapide.

4.2.2 Precision du rendu

Plusieurs améliorations pourraient être apportées au logiciel afin d'obtenir un rendu plus réaliste. L'utilisation du photonMapping semble intéressante, car elle se rapproche de la réalité d'un point de vue énergétique, alors que le lancer de rayon seul engendre souvent des aberrations au niveau des énergies mises en jeu.

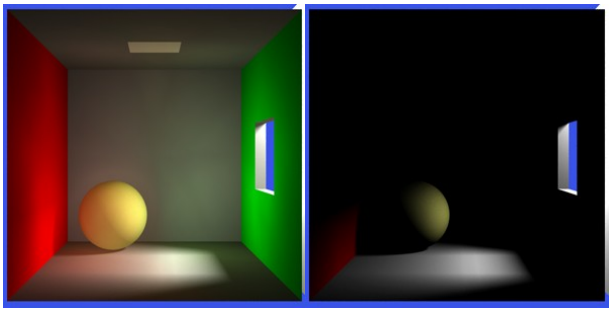


Illustration 12: Intérêt du photon mapping : avec et sans.

L'utilisation de textures pour les matériaux peut elle aussi être envisagée, d'autant plus qu'elle a déjà été préparée dans l'architecture du programme, et demande un effort d'implémentation assez faible en comparaison du réalisme qu'elle procure. Le seul souci que nous pouvons soulever ici est les format des textures, qui est problématique à cause de la gestion des couleurs sur plus de 3 canaux utilisée ici.

Au prix d'un temps de calcul plus long, il est possible de gérer les milieux dispersifs. Une amélioration possible serait d'optimiser le calcul des rayons réfractés à l'entrée d'un

milieu dispersif, par exemple en regroupant les rayons semblables.

5 Conclusion

Le développement de ce logiciel nous a permis de mieux comprendre les mécanismes de la conception orientée objet. Nous avons découvert l'importance d'une bonne méthode, qui permet d'éviter les pièges dès le départ.