Fonctionnement global du moteur de Art

Ce document décrit le fonctionnement du logiciel de lancer de rayon tel qu'il est développé en date de février 2013.

Sans entrer dans des détails de conception, ce court rapport présente les classes et les concepts clefs utilisées dans le programme.

1. **Initialisation du lancer de rayon**
2. **Structure de partitionnement et volumes englobants**
    1. **Structures de partitionnement**
    2. **Volumes englobants**
    3. **Options de toricité**
3. **Primitives géométriques utilisées dans ART**
4. **Modèles radiatif utilisés dans ART**
    1. **Modèles de rediffusion**
    2. **Modèles d'interception**
5. **Interaction rayon/objets de la scène**
6. **Interaction rayon/capteur**
7. **Exemples d'implémentation**
    1. **Simulation Lidar**
    2. **Bilan Radiatif**

1. Initialisation du lancer de rayon

Le moteur de lancer de rayons de Art a besoin d'être initialisé en suivant plusieurs étapes.
Cette initialisation consiste *in fine* à instancier un objet qui prend en charge les calculs de propagation optique, `RayManager` (`jeeb.lib.archimed2.raytracing.ray.RayManager`).

Le développeur peut étudier le code source du constructeur de cette classe `RayManager`, mais la documentation suivante permet rapidement de cerner les techniques mises en œuvres lors de l'initialisation du moteur. L'étape d'initialisation consiste en ces étapes :

1. Importation des données. Cette première étape consiste à créer un objet `Scene` (`jeeb.lib.archimed2.scene.Scene`) à partir d'entités existantes (depuis Simeo par exemple). A ce stade, les objets de la scène possèdent seulement une géométrie et un modèle radiatif, un identifiant de plante et d'organe. *Dans une version future il serait approprié d'ajouter des informations, comme par exemple le type d'objet*. Ces objets sont une collection d' `ArtNode`

2. La `Scene` est ensuite analysée par le `voxelManager` (`jeeb.lib.archimed2.raytracing.voxel.VoxelManager`) :
    - La répartition spatiale des objets dans la scène : Une indexation des entités selon leur voxel d'appartenance est réalisée (cf. « partitionnement et volumes englobants ») .
    - La réduction/augmentation de la scène est calculée lorsque l'utilisateur spécifie une plot box (cela inclue la duplication/suppression d'entités de la scène)
    - Le `voxelManager` permet aussi d'associer aux objets de la scène un volume englobant lors de l'analyse.

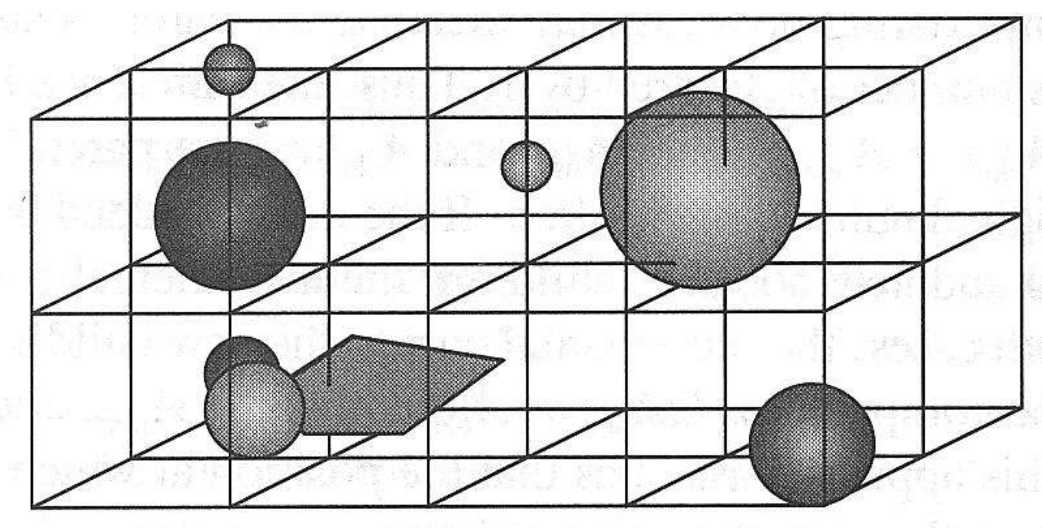
3. Lorsque le `voxelManager` est instancié, une structure d'accélération est créée à partir de l'analyse de la scène et des paramètres spécifiés. (cf. « partitionnement et volumes englobants ») . Cette structure permet d'effectuer de façon plus rapide les calculs de propagation des rayons dans la scène. `RayManager` est l'objet dédié à ce calcul, et utilise le `voxelManager` pour bénéficier de cette accélération.

2. Structure de partitionnement et volumes englobants utilisés pour le lancer de rayon dans ART

Dans ART est mis en œuvre une structure d'accélération reposant sur deux stratégies: La subdivision de l'espace, et l'utilisation de volumes englobants pour chaque objet de la scène.

2.1.Partitionnement de l'espace

Une subdivision régulière de la boite englobante de la scène est réalisée. L'utilisateur spécifie la subdivision souhaitée (trois valeurs entières – pour chaque direction xyz, ou un nombre approximatif de cellules). Chaque cellule, ou voxel, contient une liste des objets intersectant. Le parcours des voxels s'effectue de proche en proche, de l'origine vers l'arrière, et est stoppé lorsqu'une intersection est trouvée.



Les classes utilisées pour la gestion du parcours des voxels se trouvent dans le package `jeeb.lib.archimed2.raytracing.voxel` et plus particulièrement la classe VoxelManager.

Dans cette classe, plusieurs fonctions clef :

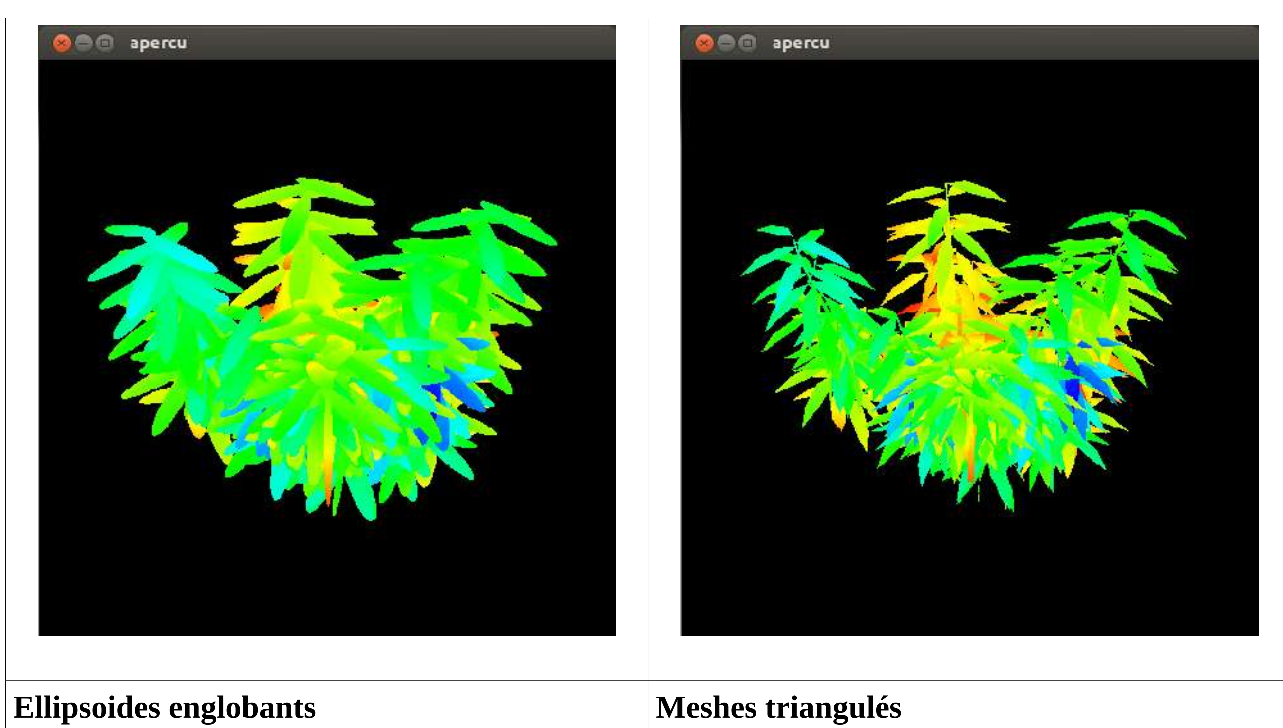
- `buildArtNodesIndexList` construit la liste des objets intersectant, pour chaque voxel
- `getFirstVoxel` trouve le premier voxel rencontré par un rayon (le rayon peut être à l'intérieur de la boite englobante de la scène, comme à l'extérieur)
- `crossVoxel` trouve les indices du prochain voxel à être visité par le rayon, dans le cas où celui-ci ne rencontre pas d'intersection. Cette fonction permet également de calculer les translations à effectuer au rayon lorsque, dans le cas d'une scène torique, il sort d'un côté de la boite englobante de la scène.

2.2.Volumes englobants

Le principe est d'effectuer un calcul simple d'intersection sur le volume englobant, de ne calculer l'intersection sur l'objet que si le rayon intersecte son volume englobant.
Les volume englobants utilisés dans ART sont des sphères ou des ellipses, primitives dont les calculs d'intersection sont très rapides. Les volumes englobants sont intéressants à utiliser sur des objets dont le calcul d'intersection est lent (par exemple, des meshes triangulés avec un grand nombre de facettes).

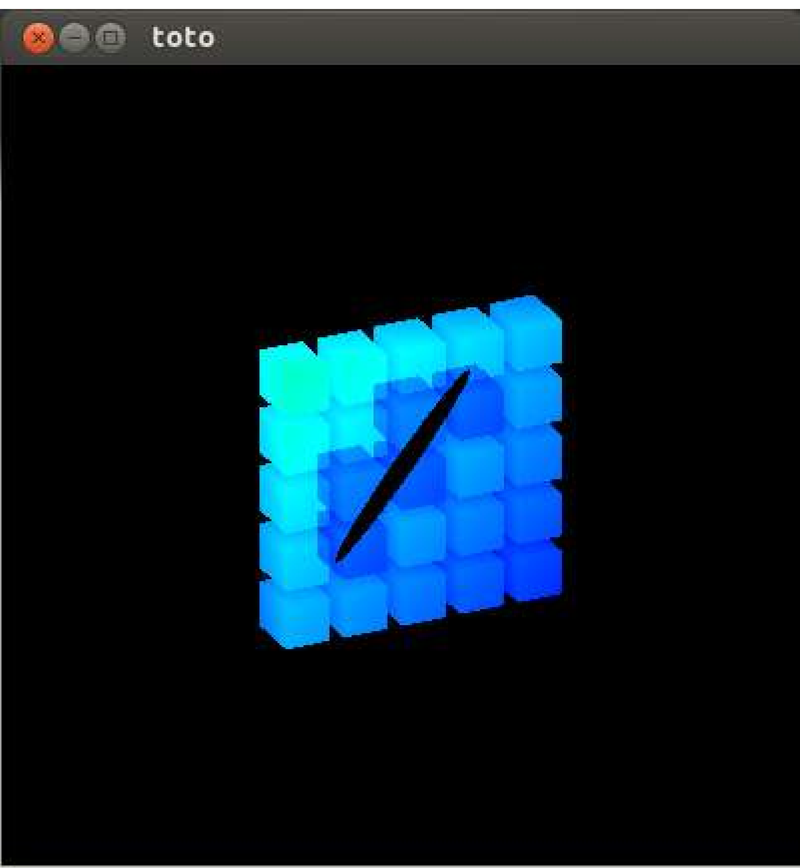
Les volumes englobants sont de type VolumicShape (`jeeb.lib.archimed2.geometry.shapes.VolumicShape`) et sont associés aux shapes de chaque ArtNode dans la fonction `buildArtNodesIndexList` de VoxelManager : De fait il y a autant de bounding shapes dans un ArtNode que de shapes (il peut y avoir une duplication de shape lorsque l'utilisateur spécifie une PlotBox et qu'un shape est « à cheval » sur une limite de la PlotBox). Le calcul du volume englobant des shapes est effectué par la classe `jeeb.lib.archimed2.geometry.shapes.BoundingShapeComputing`

Cette classe est développée autour de l'algorithme de la sphère minimale de Welzl et du calcul de convex hull. L'algèbre linéaire permet d'effectuer les transformations nécessaires à l'obtention d'ellipsoïdes englobants. ATTENTION : Ces ellipsoïdes sont optimaux au sens du volume, cependant le temps nécessaire au calcul de l'intersection avec le rayon est légèrement supérieur à celui de la sphère. L'utilisation de sphère ou d'ellipsoïde pour la description du volume englobant peut peut être plus ou moins rapide selon la scène. Dans BoundingShapeComputing une méthode empirique permet de choisir une sphère ou une ellipse selon le gain de volume englobant. (Choix d'un ellipsoïde pour des objets allongés, d'une sphère pour des objets compacts. Mais cette méthode peut être améliorée en prenant en compte des considérations statistiques sur le coût de calcul de l'objet à englober/son volume englobant dans la scène)



Ellipsoides englobants	**Meshes triangulés**

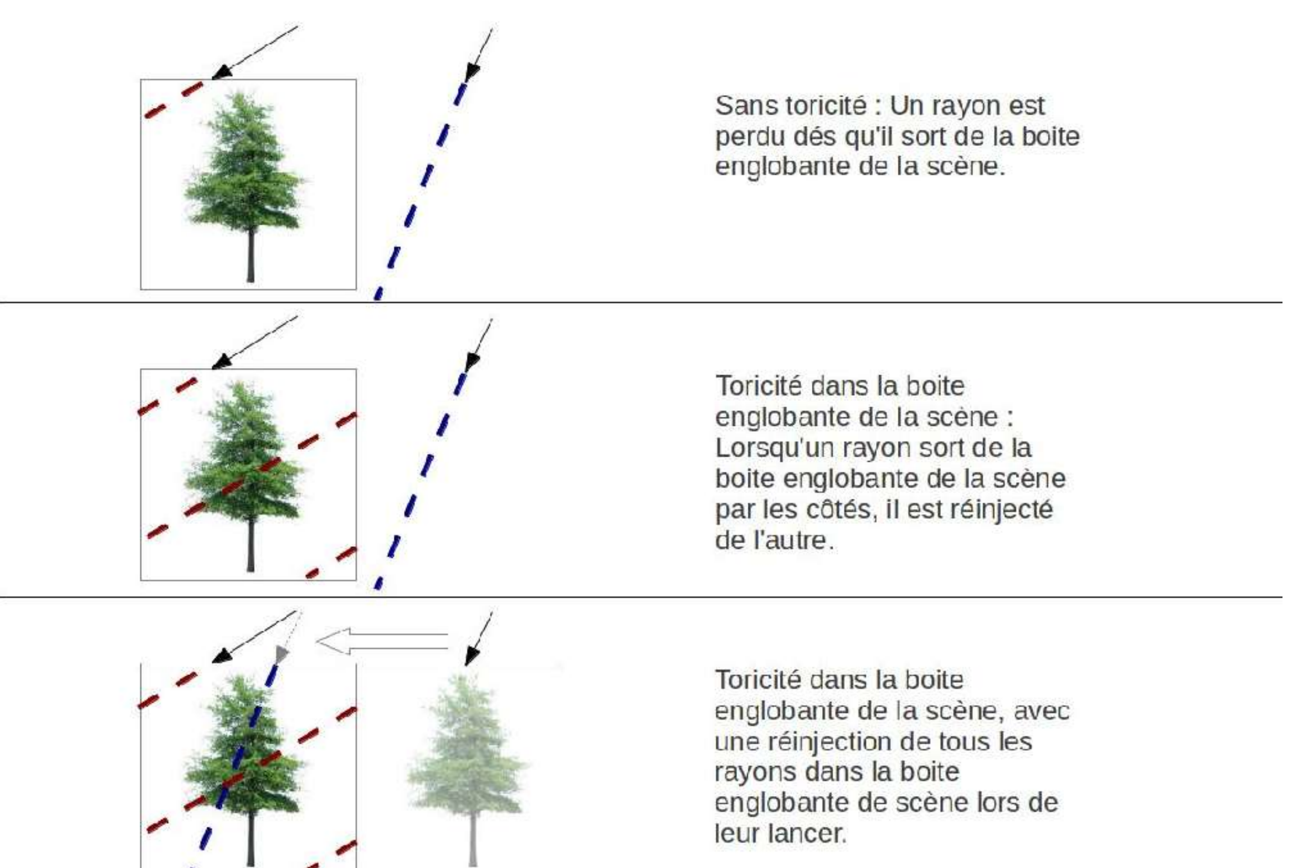
Les ellipsoïdes englobants permettent également de trouver rapidement les voxels utilisés par un shape comme l'illustre la figure suivante (utilisé dans la méthode `buildArtNodesIndexList`).



3.Options de toricité

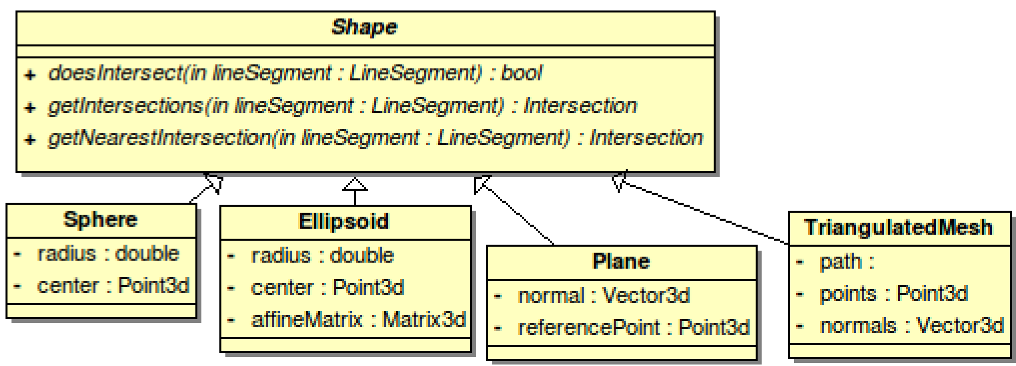
La toricité de l'espace permet de réinjecter un rayon qui sort de la scène par les côtés (c'est à dire par les plans normaux aux axes X et Y). Cette option peut servir à créer une scène infinie à partir de quelques plantes virtuelles, par exemple pour simuler une forêt d'individus.
Trois options de toricité sont possibles :

- Pas de toricité : Lorsque un rayon sort de la boite englobante de la scène, il est perdu c'est à dire que le calcul de sa propagation est interrompu.
- Toricité « finie » : Dés lors qu'un rayon entre dans la boite englobante de la scène, son trajet est géré de façon torique. Par contre, si le rayon n'atteint jamais la boite englobante de la scène, il est perdu.
- Toricité « infinie » : Tout rayon entrant dans « la boite englobante infinie » (volume compris entre deux plans normaux à Z et passants respectivement par le Zmax et le Zmin de la boite englobante de la scène) est géré de façon torique.


Sans toricité : Un rayon est perdu dés qu'il sort de la boite englobante de la scène.
Toricité dans la boite englobante de la scène : Lorsqu'un rayon sort de la boite englobante de la scène par les côtés, il est réinjecté de l'autre.
Toricité dans la boite englobante de la scène, avec une réinjection de tous les rayons dans la boite englobante de scène lors de leur lancer.

3. Primitives géométriques utilisées dans ART pour le calcul de lancer de rayon

Les primitives dérivent d'une même classe mère, `Shape` (`jeeb.lib.archimed2.geometry.shapes.Shape`).



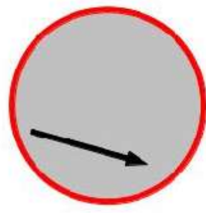
Les primitives ont des méthodes communes pour le calcul d'intersection avec le rayon (argument d'entrée : `LineElement`, demi-droite support du rayon) :

- `isIntersectedBy` booléen renvoyant true si la primitive est intersectée (utilisé uniquement pour les calculs d'intersection avec les volumes englobants)
- `getNearestIntersection` de type Intersection (`jeeb.lib.archimed2.geometry.Intersection`) cette méthode renvoie la plus proche intersection entre la primitive et le rayon.
- `getIntersections` Renvoie la liste des intersections entre la primitive et le rayon (utilisé uniquement dans le calcul d'interception avec des modèles d'interception volumique)

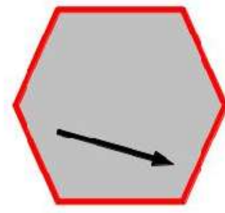
Les primitives volumiques (`Sphere, Ellipsoid, ConvexMesh, VolumicTriangulatedMesh`) implémentent une interface, `VolumicShape`, qui possède une méthode pour tester si un point est à l'intérieur du volume ou non.

L'utilisateur peut ajouter d'autres `Shape` en implémentant les méthodes ci-dessus. Si le `Shape` est infini (exemples : plans, nurbs, fractales) la variable « finite » doit être fixée à « false » (« true » est défini par défaut à la création d'un Shape)

ATTENTION : `Sphere` et `Ellipsoid` implémentent de façon particulière la méthode `isIntersectedBy` de façon à ce qu'une intersection soit détectée de façon volumique, par opposition à surfacique. Illustration par opposition avec un ConvexMesh :



Sphere :
isInstersected(ray)=TRUE
Volumic intesection



ConvexMesh :
isInstersected(ray)=FALSE
Surfacic intersection

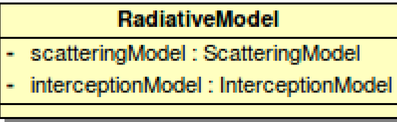
La raison de cette différence d'implémentation est due au fait que pour les éléments englobants,

seule la primitive sphère est utilisée. En effet le besoin de détecter si un rayon intersecte une sphère englobante est aussi valable si le rayon est déjà à l'intérieur.. Avec l’ellipse englobant, le problème est ramené à celui la sphère car il s'agit simplement d'une transformation affine du rayon dans l'espace, puis l'intersection est calculée sur une sphère.

Lorque l'utilisateur décide de créer un nouveau shape, une autre chose importante est d'implémenter la méthode qui donne les voxels occupés par ce shape, dans la classe VoxelSpace (`jeeb.lib.archimed2.raytracing.voxel.VoxelSpace`), fonction `getShapeVoxelIndices(Shape)` qui fait appel à des fonctions de ShapeUtils (`jeeb.lib.archimed2.geometry.shapes.ShapeUtils`)

4. Modèles radiatifs utilisés dans ART

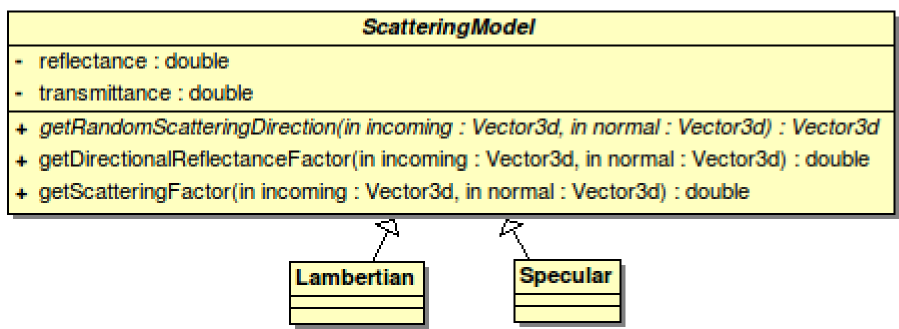
Chaque entité de la scène (`ArtNode`) possède un modèle radiatif (`RadiativeModel`). Le modèle radiatif gère l'interaction entre un rayon et un objet de la scène.
Le modèle radiatif comporte un modèle de scattering (`ScatteringModel`) et un modèle d'interception (`InterceptionModel`).



Le modèle de scattering est uniquement surfacique et concerne la rediffusion du rayon lumineux, tandis que le modèle d'interception peut être volumique et concerne l'interception du rayon lumineux.

4.1.Modèle de scattering

Ce modèle décrit une interaction de rediffusion (*scattering*) : le rayon peut être transmis, réfléchi, ou absorbé par une surface dans le cas d'un lancer de photon. (lois de Fresnel)



Divers modèles de rediffusion peuvent être implémentés. Le modèle déclaré doit hériter de la classe `jeeb.lib.archimed2.raytracing.scatteringmodel`

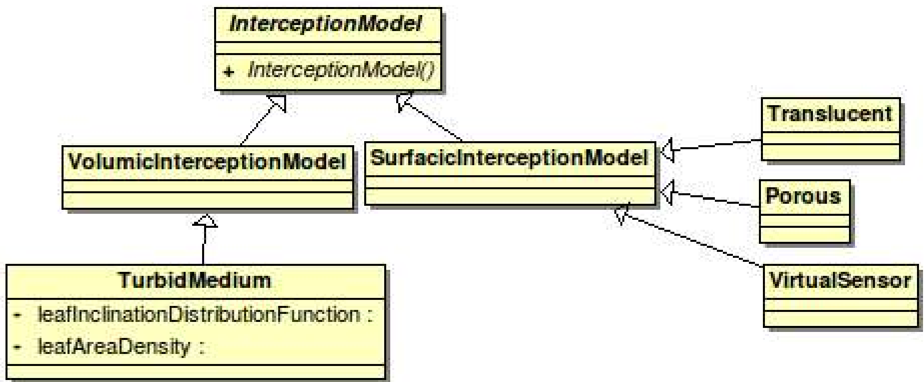
L'utilisateur doit y implémenter les méthodes suivantes :

1. `getRandomScatteringDirection` donnant la nouvelle direction prise par le rayon
2. `getDirectionalReflectanceFactor` donnant la réflectance directionnelle de l'impact lumineux (permet de calculer la quantité d'énergie reçue par un capteur)
3. `getScatteringFactor` donnant la fraction de l'intensité transmise par le rayon lorsque celui-ci n'est pas un photon

Les modèles de rediffusion ont comme paramètre de base une réflectance et une transmittance qui décrivent les lois de fresnel. La réflectance représente la probabilité que le rayon soit réfléchi, la transmittance représente la probabilité que le rayon soit transmis à travers a surface. Ainsi la somme de la réflectance et de la transmittance doivent être inférieures à 1. La probabilité qu'un rayon soit absorbé est 1 – (reflectance+transmttance) dans le cas d'un lancer de photon.

4.2.Modèle d’interception

Le modèle d'interception décrit une interaction qui peut être surfacique ou volumique.



Ce type de modèle d'interception doit hériter de la classe `jeeb.lib.archimed2.raytracing.interceptionmodel` et l'utilisateur doit implémenter la méthode `interception` : Cette méthode a comme entrées l'objet rayon et l'objet shape du ArtNode associé au modèle. L'objet renvoyé par cette méthode est un objet de type `jeeb.lib.archimed2.geometry.Intersection`
Lorsque le `getNormal()` de cet objet est null, le rayon est considéré comme non intercepté et continue son chemin. Sinon, le `getNormal()` de cet objet renvoie la normale au point d'interception. Ce fonctionnement est commun aux modèles d'interception surfacique et volumique.

1. Dans le cas d'un modèle d'interception surfacique, le rayon peut subir une interception, ou ne rien subir du tout : dans ce cas il traverse simplement la surface (modèles translucides ou poreux). Une interception signifie :
   - Une rediffusion : Si le `ScatteringModel` est instancié, le comportement du rayon sera une transmission, une réflexion, ou une absorption dans le cas d'un photon.
   - Une absorption : si le `ScatteringModel` est null, l'interception est l'arrêt de la propagation du rayon (absorption dans le cas d'un photon)

   La méthode `interception` du `InterceptionModel` doit toujours renvoyer `intersection.getNormal()`=null lorsque le rayon n'est pas intercepté. Lorsque le rayon est intercepté, la méthode doit retourner la normale de l'interception, ainsi que la distance.
2. Idem dans le cas d'un modèle d'interception volumique. L'utilisateur peut développer sa propre méthode en prenant en compte le fait que le `shape` de `artNode` associé au modèle est nécessairement volumique. En effet l'utilisateur peut caster le `Shape` en `VolumicShape` (`jeeb.lib.archimed2.geometry.shapes.VolumicShape`) dans la méthode `interception`. Implicitement un modèle d'interception volumique contient un nuage de particules (ex. pour une couronne d'arbre un nuage d'éléments de surfaces foliaires). Si une interception est simulée, la rediffusion est traitée en se ramenant au cas d'un objet surfacique. Si ce dernier n'est pas défini la surface interceptrice est considérée comme noire (pas de rediffusion). Sinon on applique le modèle spécifié.

ATTENTION : L'utilisateur peut choisir d'intégrer un modèle de rediffusion au modèle radiatif. Dans ce cas, lorsque le modèle d'interception intercepte un rayon, celui-ci va interagir avec le modèle de rediffusion précédemment défini. La méthode `interception` doit alors nécessairement renvoyer une intersection dont la méthode `getNormal()` ne doit pas être null, de façon à ce que le modèle de rediffusion puisse fonctionner. Il faut donc

implémenter le modèle en conséquence, c'est à dire définir une normale à l'intersection. Si aucun modèle de rediffusion n'est défini, le modèle d'interception volumique aura un fonctionnement similaire à celui du modèle d'interception surfacique : le rayon peut être soit transmis, soit absorbé.

Remarques : L'implémentation du mois de février 2013 est incomplète, car lorsque le rayon entre dans un milieu volumique, la scène n'est plus prise en compte. Les calculs d'intersection avec les objets de la scène sont effectués de nouveau lorsque le rayon sort du milieu volumique. En effet, le modèle d'interception est prioritaire sur les entités de la scène. Il faudrait calculer à chaque fois l'intersection probabiliste dans le milieu, et simultanément l'intersection avec les entités de la scène. Ainsi, au mois de février 2013, il est impossible de simuler une plante virtuelle placée dans un milieu volumique, ou de simuler des milieux volumiques inclus dans d'autres milieux volumiques par exemple.

Pour cela il faut adapter la méthode `computeOnePropagation` de rayManager de façon à ce que le calcul des intersections les plus proches avec les objets de la scène soit effectué, et que la proximité de l'intersection soit comparée avec celle de l'intersection probabiliste donnée par le modèle d'interception.

Récapitulatif des différentes configurations possibles d'un modèle radiatif

Interception Model	*null*	surfacic	volumic	surfacic	volumic
Scattering Model	lambertian	*null*	*null*	lambertian	lambertian
Illustration					
Description	*The ray is scattered on the surface.*	*The ray can pass through the surface, or not.*	*No rediffusion when there is an interception. Else, the ray (or the photon) pass through the medium. Remark : When the ray pass through the medium, its intensity must not be change.*	*The ray can pass through the surface, or can be scattered. Its intensity can be attenuated.*	*The ray can pass through the volume, or can do multiple (number>0) scattering in the volume.*

5. Interaction rayon/objets de la scène

L'implémentation d'un lancer de rayon s'articule autour de l'objet `RayManager`. Cette classe gère les intersections du rayon avec les entités de la scène, et son comportement optique.

1.Propagation

Exemple simple de calcul de la propagation d'un rayon (ray) avec un RayManager (rayManager).

```
while (ray.getIntensity () > lowIntensityTreshold)
      RayShot rayShot = rayManager.computeRayPropagation (ray);
```

Dans cet exemple, l'itération est effectuée tant que le rayon est porteur d'intensité. Le résultat d'un seul calcul de propagation (une itération, correspondant à une rediffusion simple) est décrit par un objet de type RayShot (`jeeb.lib.archimed2.raytracing.ray.RayShot`).
L'objet RayShot permet de décrire l'interaction physique entre le rayon et son environnement (Lieu de l'impact, nature de l'impact, entité impactée, intensité, etc). La fonction `computeRayPropagation` du RayManager renvoie un objet de type RayShot et modifie le rayon (Origine, direction, intensité, longueur totale parcourue, ordre de la rediffusion courante) à chaque calcul de propagation.

2.Bilan radiatif

Il est possible d'effectuer un bilan radiatif des objets de la scène en appelant une méthode de la scène : `scene.addAShot(ray, rayShot)`
Cette méthode va sommer la contribution lumineuse (décrite par « rayShot ») reçue par l'organe (identifié par l'environnement du rayon « ray »).
Les données seront stockées par l'objet scene.

6. Interaction Rayon/Sensor

Lors de l'utilisation d'un (ou de plusieurs) objet(s) sensor(s), un nouvel objet permet de décrire l'interaction entre le capteur et la rediffusion qu'il observe : RayReturn (`jeeb.lib.archimed2.raytracing.ray.RayReturn`).
En effet lors d'une rediffusion, une partie de l'intensité du rayon est susceptible d'être transmise au capteur. Cela est géré par la fonction `computeReturningToSensorFromShot` du rayManager, qui renvoie un objet de type « RayReturn » pour décrire ce possible retour de lumière vers le capteur.
La valeur de l'intensité retournant au capteur est déterminée par le modèle de rediffusion associé pour une direction de rediffusion et une normale donnée (méthode `getDirectionalReflectanceFactor` du `ScatteringModel`).

Cet exemple montre comment récupérer les informations décrivant l'interaction shot/capteur :

```
while (ray.getIntensity () > lowIntensityTreshold) {

      // Computes the propagation of the ray
      RayShot rayShot = rayManager.computeRayPropagation (ray);

      // Computes the ray path to the sensor
      RayReturn lightReturn = rayManager
                  .computeReturningToSensorFromShot (ray, rayShot,
                                    sensor, distanceOffset);

}
```

Ici l'objet « lightReturn » permet de savoir si le shot est dans le FOV du capteur, si le retour vers le capteur s'effectue (si il n'y a pas d'interception entre le shot et le capteur par un objet de la scène), permet également de connaître l'entité de la scène éclairée par le rayon, l'intensité revenant au capteur, la distance totale parcourue par le rayon entre l'émetteur et le capteur, etc.
L'objet « sensor » (cf. « instruments ») passé en argument de la fonction `computeReturningToSensorFromShot` est modifié lors de con appel. En particulier, ses variables privées de type SensorMeasure (`jeeb.lib.archimed2.instrument.sensors.SensorMeasure`) sont actualisées pour prendre en compte les nouvelles informations captées.

Exemple : si le rayon ne subit pas de rediffusion mais est absorbé (cas d'un photon), le capteur ne recoit aucune intensité lumineuse. Au contraire, si une rediffusion lambertienne a lieu, le rayManager va analyser si le sensor « voit » ou non le shot. Si le sensor « voit » le shot, l'objet RayReturn renvoyé va contenir des informations sur le shot, et le sensor aura été actualisé (ajout d'une SensorMeasure).

De cette façon, à la fin d'une simulation, l'objet sensor contient toutes les données utiles, d'un point de vue simulation instrumentale. Il suffit d’accéder à ces « mesures virtuelles » par la méthode `getMeasure()` du sensor.

7. Exemples d'implémentations

7.1. Simulation Lidar

Une implémentation de simulation lidar se trouve dans `jeeb.lib.archimed2.plugin.LidarLauncher`. La méthode main permet de lancer la simulation avec un faible de nombre de paramètres pour une démonstration simple.

La simulation lidar peut être aussi utilisée depuis Simeo via le plugin ART-Lidar (voir le site web de AMAP).

La simulation repose sur le fonctionnement suivant : Les rayons sont envoyés depuis une source, appelée « émetteur ». Les rayons sont ensuite propagés dans la scène et subissent de la rediffusion. A chaque rediffusion , une intensité lumineuse retournant vers le « capteur » est calculée en fonction des paramètres de celui-ci . Le capteur mémorise ensuite les intensités retournées et leur associe la valeur de la longueur totale du trajet lumineux (depuis l 'émetteur jusqu'à la rediffusion final vers le capteur). Cet ensemble de couples (intensité retournée, distance totale du trajet lumineux) forme la forme d'onde simulée du lidar.

L'instrument lidar

Il est modélisé par un objet « Emitter » (`package jeeb.lib.archimed2.instrument.emitters`) et un objet « Sensor » (`jeeb.lib.archimed2.instrument.sensors.Sensor`)

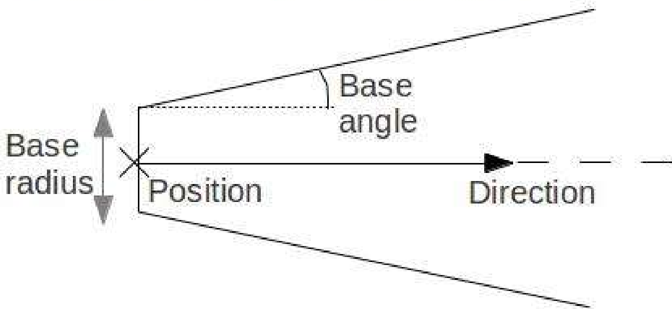
- **L'objet Emitter** est utilisé dans un premier temps pour instancier des objets de type « Ray » (`jeeb.lib.archimed2.raytracing.ray.Ray`) qui vont ensuite être propagés dans la scène.
- **L'objet Sensor** détecte ensuite (ou non) une intensité lumineuse résultant du scattering du rayon propagé.

L'instrument lidar peut être positionné partout dans l'espace, et il est possible de désolidariser l'émetteur du capteur en leur fixant des positions différentes.

L'objet Emitter

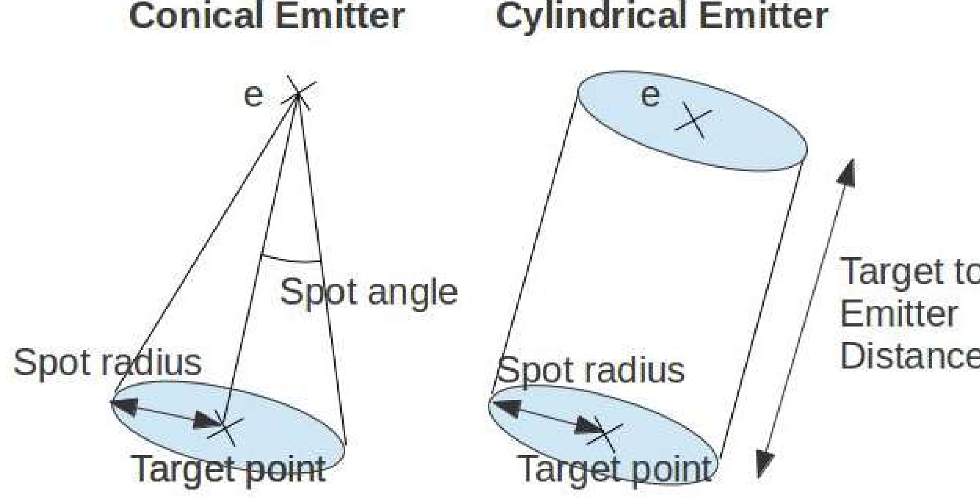
L'objet de type Emitter pour le lidar est un « ConicalEmitter » (Sous classe de Emitter, paramétré par un cône) et se définit comme suit:

1. un point d'origine : centre de la matrice des rayons
2. un angle de base : angle d'ouverture de l'émetteur
3. un rayon de base : rayon de base de la matrice (peut être nul, dans le cas la matrice est ponctuelle)
4. une intensité de base : les rayons instanciés porterons cette valeur d'intensité initiale
5. un nombre de rayons : cela fixe le nombre de rayon de l'émetteur, ainsi il est possible de récupérer les rayons tant que l'émetteur en contient (méthode getRay())



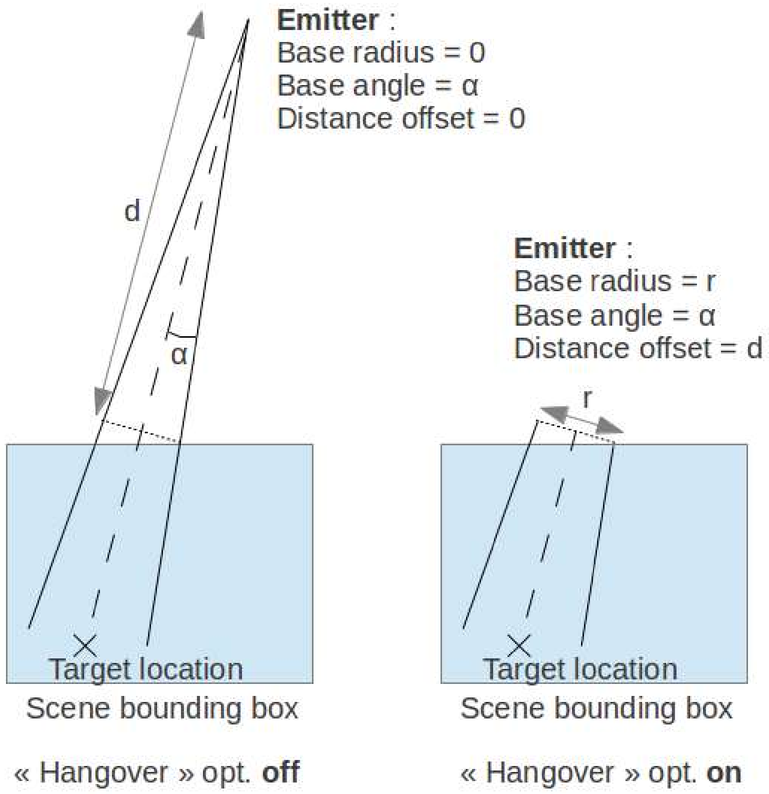
Dans la classe `jeeb.lib.archimed2.plugin.LidarLauncher` l'utilisateur ne peut pas directement fixer le base radius et le base angle, mais a deux options possibles pour paramétrer l'objet Emitter :

- Soit l'objet est définit par un cône (l'utilisateur spécifie un base angle et un base radius)
- Soit l'objet est définit par un cylindre (l'utilisateur spécifie juste un base radius)



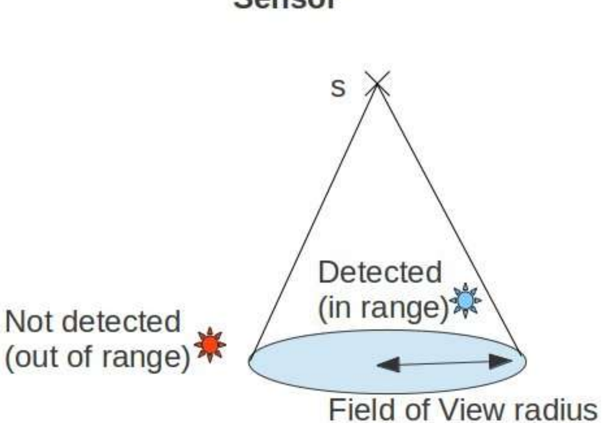
Cependant, il peut être problématique que l'utilisateur définisse un Emitter qui est très éloigné de la cible : Le moteur de ART travaille à l'échelle du cm, en float, et utilise des tolérances géométriques très fines (quelques dixièmes de mm) pour calculer les rebonds du rayon lors des scattering, ou encore les marges des bounding shapes par exemple. Or, le fait de comparer de très grandes valeurs à ces marges très petites peut causer des problèmes de stabilité numérique. En effet l'utilisateur peut très bien demander de positionner un capteur à plusieurs centaines de kilomètres de la cible, comme pour simuler un capteur Satellitaire.

Une transformation de l'émetteur permet de remédier à ce problème numérique, il s'agit de l'option « hangover » qui permet en quelque sorte de le modifier et de le rapprocher de la cible, ce qui permet d'éviter ainsi les désagréments précédemment évoqués. Cette figure explique la façon dont l'Emitter est modifié : Son base radius et sa position sont changés.



L'objet Sensor

L'objet de type Sensor est définit de façon très simple : Il s'agit juste d'un objet qui permet de transformer une information du type (position 3D, intensité) en intensité lumineuse reçue par le capteur. Dans notre cas, l'objet Sensor est définit par un cône de visée (qui délimite la zone « vue » de la zone « non vue », cela est effectué par la méthode `canSee`) et une fonction de pondération de l'intensité reçue en fonction de la position du shot détecté (`getReturningFactor`).



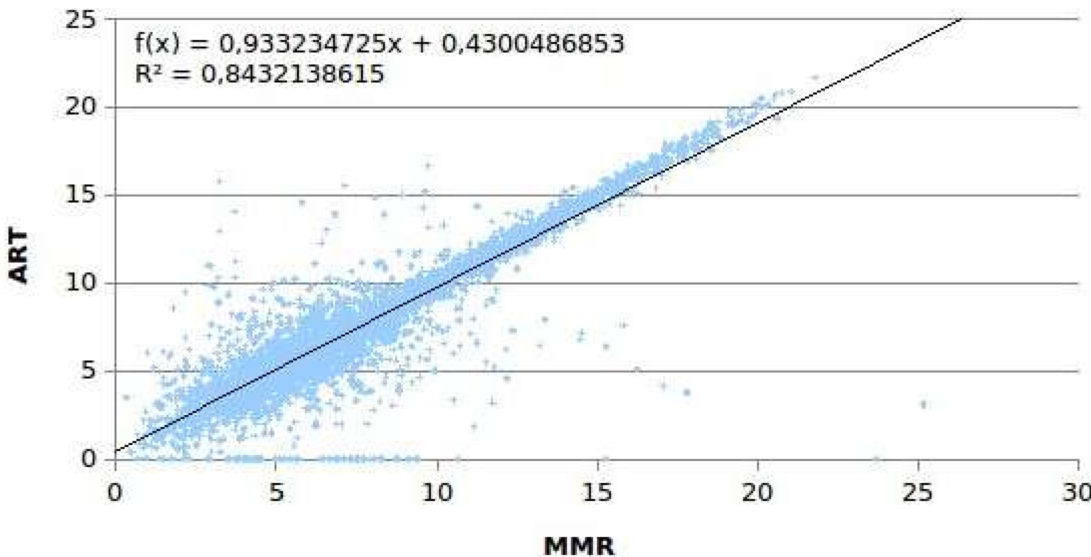
7.2. Bilan Radiatif

Le bilan radiatif est effectué sur une scène donnée avec des conditions d'éclairement spécifiées par l'utilisateur. Une classe indépendante de ART donne le flux lumineux reçu par la scène (`jeeb.lib.archimed2.light.IncidentRadiation`) en fonction d'une plage temporelle d'éclairement solaire que l'utilisateur précise. Cet objet délivre, pour 46 directions (icosahedron modélisant les composantes directionnelles de la voûte céleste), les valeurs du flux lumineux à simuler.

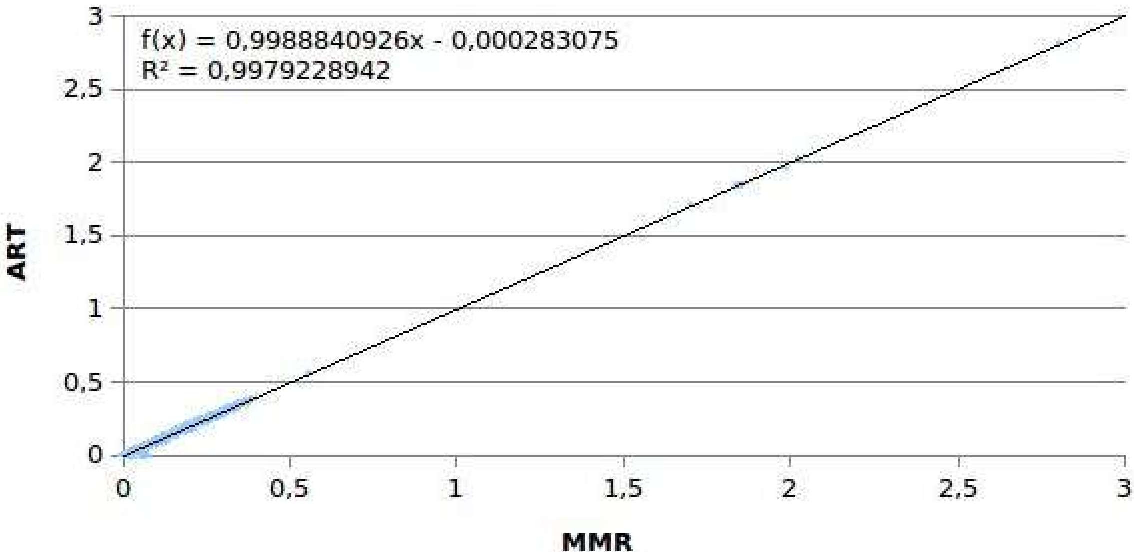
Les 46 lancers de rayons sont effectués, avec 46 directions et intensités différentes (calculées précédemment avec la classe `IncidentRadiation`). On utilise un émetteur parallèle rectangulaire horizontal (`jeeb.lib.archimed2.instrument.emitters.HorizontalParallelEmitter`), positionné au dessus de la boite englobante de la scène et couvrant toute sa surface (de façon à ce que chaque zone de la scène reçoive le même éclairement) pour chaque direction afin d'éclairer l'ensemble de la scène.

Dès qu'un rayon est intercepté par un organe on affecte à ce dernier l'énergie portée par le rayon, ce qui permet le calcul du bilan radiatif d'un plot.

Ces figures comparent le résultat d'un bilan radiatif effectué sur une plante virtuelle avec ART et avec MMR



Le léger écart entre les valeurs de MMR et d'ART provient du calcul de la surface des organes, ART effectuant une triangulation systématique des géométries de type « mesh », alors que MMR utilise des algorithmes de calcul de surface sur des polygones.



Intercepted radiation
f(x) = 0,9988840926x - 0,000283075
R² = 0,9979228942
ART
MMR
3
2,5
2
1,5
1
0,5
0
0
0,5
1
1,5
2
2,5
3

7.3. Autres implémentations

Des exemples des simulations suivantes se trouvent dans une classe test
(`jeeb.lib.archimed2.raytracing.tests.demo2`)

BRDF (Boucle de capteurs)

Un ensemble de 46 capteurs est répartit autour de la scène et permet d'étudier la BRDF.
(`jeeb.lib.archimed2.simulation.BRDFComputingSA`)

BRDF (Methode rayon sortant)

Chaque rayon sortant de la boite englobante de la scène contribue au calcul de la BRDF. La direction du rayon est échantillonnée dans 46 directions (Utilisation d'un KD-Tree).
(`jeeb.lib.archimed2.simulation.BRDFComputingRL`)

Au mois de février 2013, les calculs de BRDF sont effectués pour une source lumineuse de rayons parallèles. Il est tout à fait possible d'adapter le code pour que la source modélise une lumière solaire (plusieurs directions) en utilisant la classe `jeeb.lib.archimed2.light.IncidentRadiation` *de la même façon que cela a été réalisé pour le bilan radiatif, c'est à dire en effectuant autant de simulations qu'il y a de directions différentes.*

Les résultats des simulations sont exportés dans un script MATLAB (compatible OCTAVE) pour la visualisation en 3D de la BRDF. Il est possible d'accéder directement aux valeurs de la distribution avec la classe `jeeb.lib.archimed2.analysis.VectorialDistribution`