Le *ray tracer* – 2^{e} partie

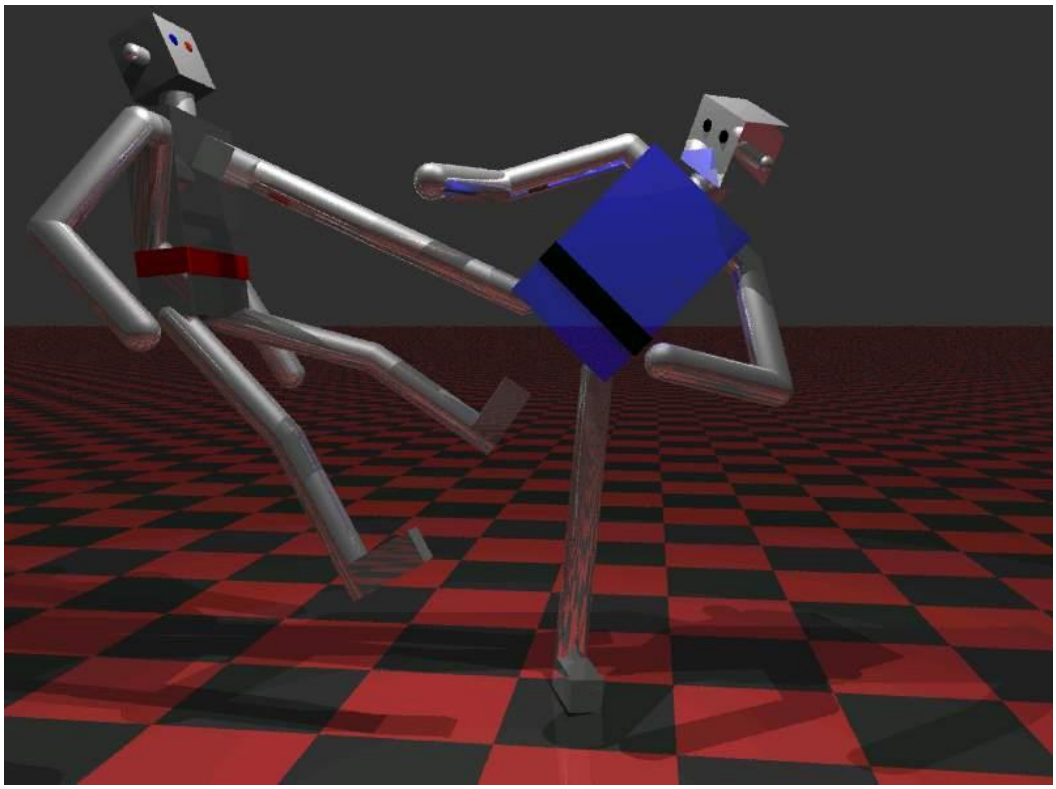


Image générée par Simon Vézina et Yannick Simard

Table des matières

Introduction

À partir de l'application **SIMRenderer** que vous avez développée dans la 1re partie du laboratoire de ***ray tracer***, vous allez ajouter des nouvelles fonctionnalités en lien avec des modèles d'illumination et visualiser des nouvelles images. Vous devrez ainsi télécharger le document de réponse « *Ray_tracer–Feuille_des_donnees-Partie_2.doc* » disponible au lien

http://physique.cmaisonneuve.qc.ca/svezina/projet/ray_tracer/download/Ray_tracer-Feuille_des_donnees-Partie_2.doc

afin d'y introduire vos images. Vous devez également y inscrire vos réponses aux questions posées tout au long de ce laboratoire.

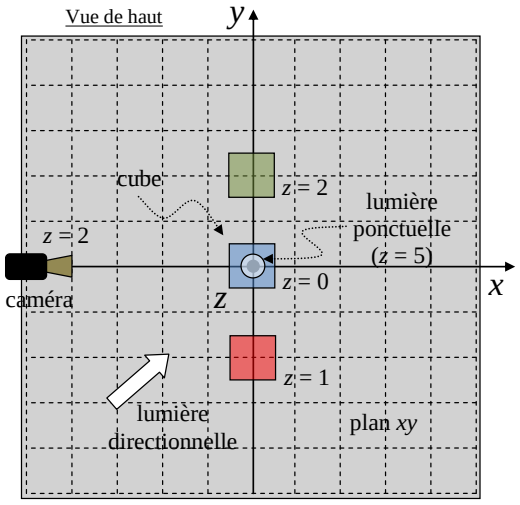
Le modèle d'illumination directe

Comme vous l'avez préalablement remarqué, l'application génère des images en utilisant uniquement la couleur du matériel appliqué sur la géométrie, car il n'y a pas de modèle d'illumination d'implémenté. C'est pour cette raison que vous allez implémenter un modèle d'illumination directe pour simuler la réflexion ambiante et la réflexion diffuse ce qui vous permettra de colorer vos primitives de votre scène en fonction de l'éclairage disponible.

Afin de vous permettre de visualiser l'implémentation de votre modèle d'illumination directe, nous avons ajouté des sources de lumière à la scène *intro_plan.txt* et nous avons changé le mode d'illumination sans lumière (*no_light*) pour un autre mode d'illumination (ex : *ambient*, *diffuse*, *blinn_specular*, *no_specular*, *blinn*).

La première scène analysée sera à l'aide du fichier *intro_ambiant.txt* où il y a la présence uniquement d'une source de lumière ambiante blanche.

La seconde scène analysée sera à l'aide du fichier *intro_diffuse.txt* où il y a la présence d'une source de lumière ambiante blanche, une source directionnelle blanche orientée selon la direction (1.0, 1.0, -1.0) et une source ponctuelle blanche située à la coordonnée (0.0, 0.0, 5.0).



Vue de haut de la scène *intro_diffuse.txt*.

2.1 - La luminosité par réflexion ambiante

Fichier à modifier : **SIIllumination.java**

Prérequis : 1.7

Fichier de scène : intro_ambiant.txt

Dans la classe **SIIllumination** disponible dans le ***package*** **sim.graphics.shader**, vous allez programmer la méthode suivante :

```
public static SColor ambientReflexion(SColor La, SColor Sa)
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Modifiez la méthode afin qu'elle retourne adéquatement la bonne couleur par **réflexion ambiante** en vous inspirant des notes de cours présentées en classe[1]. Consultez la classe

[1] Cette information est disponible dans la section 6.4.

SColor dans le ***package*** **sim.graphics** pour connaître les caractéristiques de la classe et ses méthodes ***public*** disponibles.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SIlluminationTest** du ***package*** **sim.graphics.shader** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène pour *intro_ambiant.txt* (*read_data intro_ambiant.txt*). Exécutez le programme afin de constater le changement au programme (scène : *intro_ambiant.txt*). Si l'illumination de la scène sous une **réflexion ambiante** vous semble adéquate, copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 2.1**.

Question 2.1 :

Si l'application pouvait accepter des couleurs négatives, identifiez un effet indésirable que cela pourrait engendrer lors de la superposition des couleurs (l'addition des couleurs) ?

2.2 - La luminosité par réflexion diffuse

Fichier à modifier : **SIllumination.java**

Prérequis : 2.1

Fichier de scène : intro_diffuse.txt

Dans la classe **SIllumination** disponible dans le ***package*** **sim.graphics.shader**, vous allez programmer la méthode suivante :

```
public static SColor lambertianReflexion(SColor Ld, SColor Sd, SVector3d N, SVector3d d)
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Modifiez la méthode afin qu'elle retourne adéquatement la bonne couleur par **réflexion diffuse** en vous inspirant des notes de cours présentées en classe. Si la couleur calculée est « noire », vous n'avez qu'à retourner la valeur

```
return NO_ILLUMINATION;
```
qui est équivalente à
```
return new SColor(0.0,0.0,0.0);
```

Attention : Assurez-vous que votre méthode ne retourne pas une couleur négative. Si le cas se présente, une exception de type **SConstructorException** sera lancée ce qui vous permettra de mieux localiser votre erreur d'implémentation.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SIlluminationTest** du ***package*** **sim.graphics.shader** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène pour *intro_diffuse.txt* (*read_data intro_diffuse.txt*). Exécutez le programme afin de constater le changement au programme (scène : *intro_diffuse.txt*). Si l'illumination du plan *xy* sous une **réflexion ambiante** et **diffuse** vous semble adéquate, copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 2.2**.

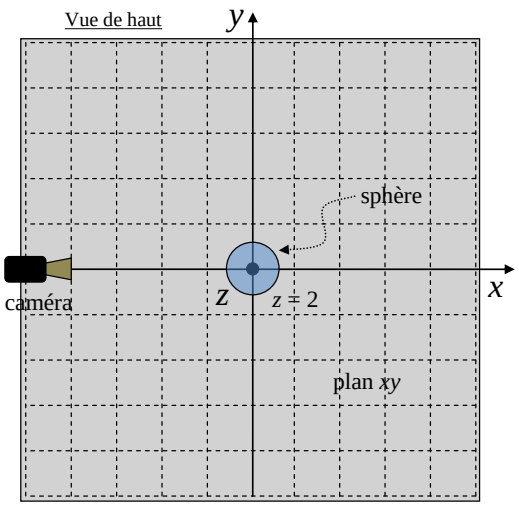
Question 2.2 :

Vous avez probablement remarqué la présence d'ombrage dans l'image générée par la scène *intro_diffuse.txt*. Pourquoi une source de lumière strictement ambiante ne permet pas de générer des ombrages ?

Le calcul de l'intersection d'une sphère

Présentement, votre application vous permet de visualiser l'intersection d'un rayon avec un plan infini, un disque et un cube. Puisqu'il y a plusieurs autres géométries disponibles (sphères, tube, cylindre, triangles, …), vous aurez à rédiger les méthodes d'intersection de ces géométries.

Pour visualiser le calcul de l'intersection de la sphère, vous allez visualiser la scène *plan_et_sphere.txt*. Ouvrez le fichier *configuration.cfg* est modifiez le fichier de scène pour *plan_et_sphere.txt* (*read_data plan_et_sphere.txt*). Cette scène comprend un plan infini *xy* (z = 0) et une sphère située à la coordonnée (0.0, 0.0, 2.0) de 1 m de rayon. Une source de lumière ambiante est présente ainsi qu'une source de lumière ponctuelle située à la coordonnée (0.0, 0.0, 5.0) et une source de lumière directionnelle ayant l'orientation (1.0, 1.0, -1.0) correspondant à un angle de 45° dans le plan *xy* et un angle de à 45° vers le bas dans le plan *xz*. La caméra est située à la coordonnée (-5.0, 0.0, 2.0) et regarde vers l'origine (0.0, 0.0, 0.0) ce qui lui donnera une petite inclinaison en -*z*. Vue de haut, la scène ressemble à l'image ci-contre.



Vue de haut de la scène *plan_et_sphere.txt*.

2.3 - Le calcul de l'intersection d'un rayon avec une sphère

Fichier à modifier : **SGeometricIntersection.java**

Prérequis : 1.5

Fichier de scène : aucun

Dans la classe **SGeometricIntersection** disponible dans le ***package* sim.geometry**, vous allez programmer la méthode suivante :

```
public static double[ ] sphereIntersection(SRay ray, SVector3d r_sphere, double R)
```

Cette méthode permet d'évaluer l'ensemble des intersections entre un rayon et une sphère. Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. À l'aide de vos notes de cours[2], modifiez la méthode afin qu'elle retourne adéquatement les temps *t* pour réaliser les multiples intersections possibles entre un rayon et une sphère. N'oubliez pas que vous pouvez utiliser la méthode

```
public static double[ ] quadricRealRoot(double a, double b, double c)
```

de la classe **SMath** développée précédemment.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SGeometricIntersectionTest** du ***package* sim.geometry** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

[2] Cette information est disponible dans la section 6.2a.

Question 2.3 :

Soit un rayon qui effectue une intersection avec une sphère. Les temps d'intersection *t* sont égaux à

$t = \{ -0.5 , 0.0 \}$.

A) De quel endroit le rayon était-il lancé ?	B) Dans quelle direction le rayon était-il lancé ?
a. De l'extérieur de la sphère. b. De la surface de la sphère. c. De l'intérieur de la sphère.	a. Vers l'extérieur de la sphère. b. Vers l'intérieur de la sphère.
Réponse :	Réponse :

2.4 - L'intersection d'un rayon avec une sphère

Fichier à modifier : **SSphereGeometry.java**

Prérequis : 2.3

Fichier de scène : plan_et_sphere.txt et deux_spheres_superposees.txt

Dans la classe **SSphereGeometry** disponible dans le ***package*** **sim.geometry**, vous allez programmer la méthode suivante :

```
public SRay intersection(SRay ray) throws SAlreadyIntersectedRayException
```

Présentement, cette méthode retourne le rayon passé en paramètre signifiant selon la documentation ***javadoc*** de la méthode qu'il n'y a pas d'intersection entre le rayon et la sphère. À l'aide de la méthode que vous venez tout juste d'implémentez, modifiez la méthode afin d'évaluer la 1ière intersection entre un rayon et une sphère (consultez votre prélaboratoire au besoin). S'il y a intersection entre le rayon et la sphère, retournez un nouveau rayon intersecté en lançant l'appel de la méthode

```
public SRay intersection(SGeometry geometry, SVector3d normal, double t) throws SAlreadyIntersectedRayException
```

à l'objet ray (avec l'instruction **return** ray.intersection(…);) en précisant les bons paramètres définissant l'intersection (consultez la ***javadoc*** pour plus de détail sur les paramètres). Puisque la géométrie intersectée est l'objet dans lequel la méthode sera exécutée, le paramètre geometry sera égal à **this**. Utilisez la méthode locale

```
protected SVector3d evaluateIntersectionNormal(SRay ray, double intersection_t)
```

pour calculer le paramètre normal.

S'il n'y a pas eu d'intersection, retournez tout simplement le rayon passé en paramètre à l'aide de l'instruction

```
return ray;
```

Attention : Pour utiliser la méthode

```
public static double[ ] sphereIntersection(SRay ray, SVector3d r_sphere, double R) ,
```

utilisez les champs ***private*** de la sphère et les méthodes ***public*** du rayon ray pour obtenir vos informations.

Attention : Pour des raisons numériques, tout temps *t* pour définir une intersection doit être supérieur à un terme **EPSILON** (ε) déterminé par la classe **SRay** à l'aide de la méthode statique

public static double getEpsilon( )

Cette condition est nécessaire pour empêcher qu'un rayon initialement lancé depuis la surface d'une géométrie puisse intersecter à nouveau cette géométrie à la même place. Ainsi, lorsque vous obtiendrez votre ensemble solution $t = \{t_1, t_2\}$, assurez-vous de bien choisir le **plus petit temps** t, mais **plus grand que** SRay.getEpsilon( ). Un oubli de votre part forcera la classe **SRay** à lancer des exceptions de type **SConstructorException** si un rayon intersecté est construit sans respecter cette condition.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SSphereGeometryTest** du ***package*** **sim.geometry** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Exécutez le programme afin de constater le changement au programme (scène : *plan_et_sphere.txt*). Si l'illumination du plan et de la sphère sous une **réflexion ambiante et diffuse** vous semble adéquate, copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 2.4a**. Modifiez le fichier *plan_et_sphere.txt* afin de définir le rayon de la sphère à 1,5 m (cherchez les mots clé *sphere* et *ray*) qui avait initialement une valeur de 1,0 m. Exécutez le programme afin de constater le changement et copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 2.4b**.

Question 2.4 :

Si une scène contient une sphère rouge et une sphère bleue de même rayon et qu'ils sont positionnées au même endroit, quelle(s) sphère(s) peut-on voir et justifiez pourquoi ? Utilisez le fichier de scène *deux_spheres_superposees.txt* (*read_data deux_spheres_superposees.txt*) et modifiez-le afin d'orienter la rédaction de votre réponse.

La suite du modèle d'illumination directe

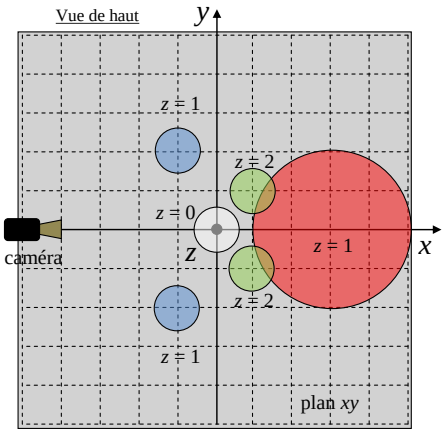
Présentement, le modèle d'illumination implémenté ne permet pas de calculer une illumination par réflexion spéculaire. Ce type d'illumination permet de visualiser la position et l'orientation des sources de lumière par l'entremise des reflets dirigés vers la caméra.

Vous allez donc améliorer le modèle d'illumination directe en ajoutant la luminosité par réflexion spéculaire.

http://www.simonpow.com/blog/howtos/light-painting-a-car/
Visualisation d'une ligne de réflexion spéculaire sur une voiture étant produite par un anneau de lumière autour de la voiture.

Pour visualiser l'amélioration de votre modèle d'illumination directe, vous aller visualiser une scène en comparant trois modes d'illumination. La scène *spheres_ambiante.txt* calculera uniquement l'illumination ambiante, la scène *spheres_diffuse.txt* calculera uniquement l'illumination diffuse, la scène *spheres_blinn.txt* calculera uniquement l'illumination spéculaire selon le modèle de Blinn et finalement la scène *spheres.txt* calculera toutes ces modes d'illumination.

La scène comprend 1 plan *xy*, 6 sphères et 2 sources de lumières comme celle de la scène *plan_et_sphere.txt*.



Vue de haut de la scène *spheres.txt*.

2.5 - La luminosité par réflexion spéculaire selon Blinn

Fichier à modifier : **SIllumination.java**

Prérequis : 2.4

Fichier de scène : spheres_ambiante.txt , spheres_diffuse.txt , spheres_blinn.txt et spheres.txt

Dans la classe **SIllumination** disponible dans le ***package*** **sim.graphics.shader**, vous allez programmer la méthode permettant d'évaluer la luminosité par réflexion spéculaire à l'aide du modèle de **Blinn** :

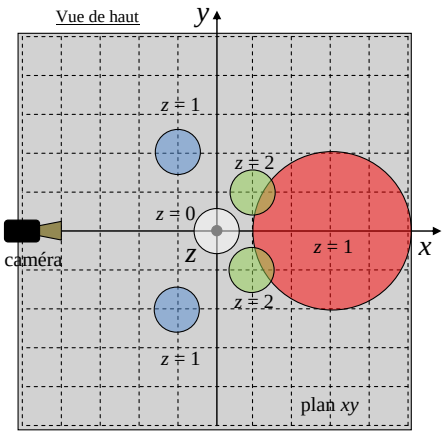
```
public static SColor blinnSpecularReflexion(SColor Ls, SColor Ss, SVector3d N, SVector3d v,
SVector3d d, double n)
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Modifiez la méthode afin qu'elle retourne adéquatement la bonne couleur par **réflexion spéculaire selon Blinn** en vous inspirant des notes de cours présentées en classe.

Attention : Assurez-vous que votre méthode ne retourne pas une couleur négative !

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SIlluminationTest** du ***package*** **sim.graphics.shader** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Pour visualiser l'effet de l'illumination ambiante, diffuse et spéculaire explicitement, vous allez générer les images des fichiers de scène *spheres_ambiante.txt* , *spheres_diffuse.txt*, *spheres_blinn.txt*, et *spheres.txt* (modifiez le fichier *configuration.cfg* et son champ *read_data* adéquatement). Si l'illumination des scènes et plus spécifiquement l'illumination par **réflexion spéculaire selon Blinn** vous semble adéquate, copiez les fichiers images générés par l'application sur votre feuille des données numériques dans la **section 2.5a**, **section 2.5b**, **section 2.5c** et **section 2.5d**.



Vue de haut de la scène *spheres.txt*.

Question 2.5 :

Pourquoi les **highlights** causés par les deux sources de lumière blanche sont « blanc » sur la sphère bleue de gauche et « bleu » sur la sphère bleue de droite ?

Suggestion : Consultez le fichier de scène *spheres.txt* pour connaître les propriétés de vos sphères et de vos matériaux. Vous pouvez également consulter la méthode

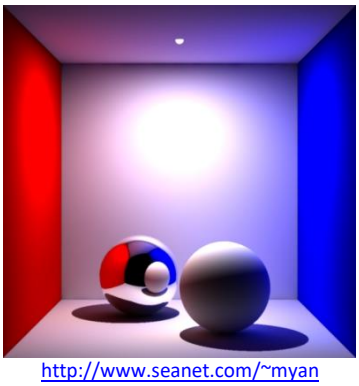
public SColor specularColor()

de la classe **SBlinnMaterial** située dans le package **sim.graphics.material** si vous cherchez toujours une explication.

Le modèle d'illumination indirecte

Présentement, l'application **SIMRenderer** ne permet pas d'évaluer de l'illumination indirecte permettant de générer de la luminosité par réflexion sur des matériaux réfléchissant et de la luminosité par réfraction sur des matériaux transparent. L'application implémente déjà un algorithme de **lancer de rayon récursif** dans la classe **SRecursiveShader** (dans le ***package* sim.graphics.shaders**) mais les calculs de réflexion et de réfraction ne sont pas implémentés.

Si un matériel possède présentement un **coefficient de réflexion** kr ≠ 0 et un **coefficient de transmission** kt ≠ 0, une exception de type **SNoImplementationException** (dans le ***package* sim.exception**) sera automatiquement lancée.



http://www.seanet.com/~myandper/abstract/sig03c09.htm
Illumination indirect indirecte par réflexion.

2.6 - La loi de la réflexion

Fichier à modifier : **SGeometricalOptics.java**

Prérequis : 2.5

Fichier de scène : spheres_reflexion.txt

Dans la classe **SGeometricalOptics** disponible dans le ***package* sim.physics**, vous allez programmer la méthode permettant d'appliquer la loi de la réflexion aux rayons lancés par le *ray tracer* :

public static SVector3d reflexion(SVector3d v, SVector3d N)

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. Modifiez la méthode afin qu'elle retourne l'orientation du vecteur réfléchi en vous inspirant des notes de cours présentées en classe.

Pour vérifier votre implémentation, vous pouvez exécuter la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SGeometricalOpticsTest** du ***package* sim.physics** située dans le répertoire **test/src**. Corrigez au besoin vos implémentations afin de satisfaire les tests.

Ouvrez le fichier *configuration.cfg* est modifiez le fichier de scène pour *spheres_reflexion.txt* (*read_data spheres_reflexion.txt*). Cette scène est identique à la scène *spheres.txt* sauf que certains des matériaux auront un **coefficient de réflexion** kr ≠ 0. Exécutez le programme afin de constater les changements (scène : *sphere_reflexion.txt*). Si l'illumination des sphères sous une **réflexion indirecte** vous semble adéquate (comparez-là visuellement avec l'image de la **section 2.5d**), vous allez générer trois images de votre scène en

modifiant le paramètre de récursivité (*recursive_level*) de 1 à 3 dans votre fichier de scène (cherchez les mots clé *raytracer* et *recursive_level*) afin de comparer les effets qu'apporte une illumination indirecte par réflexions multiples entre vos objets de votre scène. Copiez vos trois fichiers images générés par l'application sur votre feuille des données numériques dans la **section 2.6a**, **section 2.6b** et **section 2.6c**.

Question 2.6a :

Que peut-on observer sur l'image de niveau de récursivité 3 que l'on ne peut pas voir sur l'image de récursivité 2 ?

Question 2.6b :

Dans la scène *spheres_reflexion.txt*, jusqu'à quel niveau de récursivité arrivez-vous à distinguer à l'œil des changements dans les images générées ? Pour répondre à cette question, vous aurez à générer plusieurs autres images avec différents niveaux de récursivité. Utilisez un logiciel de visualisation (ex : Visionneuse de photos Windows) pour mieux comparer vos images.

2.7 - La loi de la réfraction

Fichier à modifier : **SGeometricalOptics.java**

Prérequis : 2.6

Fichier de scène : refraction.txt

Dans la classe **SGeometricalOptics** disponible dans le ***package*** **sim.physics**, vous allez programmer en premier temps la méthode suivante :

```
public static boolean isTotalInternalReflection(SVector3d v, SVector3d N, double n1, double n2)
```

Cette méthode a pour but de déterminer s'il y aura une réflexion totale interne lors d'une réfraction d'un vecteur d'orientation v sur une normale à la surface N. Complétez l'implémentation en vous inspirant des notes de cours présentées en classe et déterminez s'il y a une réflexion totale interne (true) ou non (false).

Pour vérifier cette implémentation, vous pouvez exécuter la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SGeometricalOpticsTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin vos implémentations afin de satisfaire les tests.

Par la suite, vous aller programmer la loi de la réfraction :

```
public static SVector3d refraction(SVector3d v, SVector3d N, double n1, double n2)
    throws STotalInternalReflectionException
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. Modifiez cette méthode afin qu'elle calcul l'orientation du vecteur réfracté adéquatement. Cependant, n'oubliez pas qu'une réfraction n'est pas toujours possible! S'il y a réflexion totale interne lors de l'exécution de la méthode refraction(…), celle-ci devra retournera une exception de type STotalInternalReflexionException. Ajoutez la ligne de code

```
throw new STotalInternalReflectionException();
```

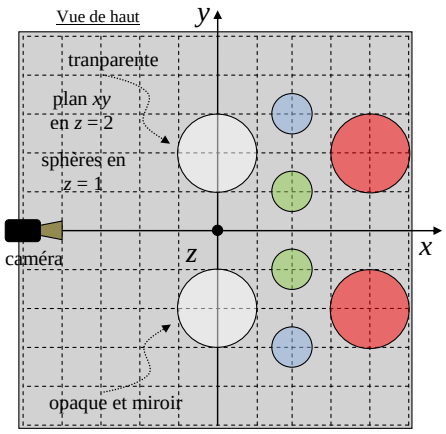
à l'endroit approprié dans votre code si la méthode détecte une réflexion totale interne.

Pour vérifier cette implémentation, vous pouvez exécuter la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SGeometricalOpticsTest** du ***package*** **sim.physics** située dans le répertoire **test/src**. Corrigez au besoin vos implémentations afin de satisfaire les tests.

Ouvrez le fichier *configuration.cfg* est modifiez le fichier de scène pour *refraction.txt* (*read_data refraction.txt*). Cette scène comprend une sphère transparente en (0.0, 2.0, 1.0) et une sphère opaque réfléchissante en (0.0, -2.0, 1.0). Les autres sphères sont opaques et elles sont situées en z = 1 et aux coordonnées xy tel qu'illustré sur le schéma ci-contre. L'éclairage est la même que dans les scènes précédentes[3].

Exécutez le programme afin de constater les changements au programme (scène : *refraction.txt*). Si l'illumination des sphères sous une **réfraction** vous semble adéquate, copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 2.7a**.

Par la suite, modifiez la scène en changeant l'indice de réfraction de la sphère transparente initialement de 1.55 pour 2.8 (cherchez les mots clé *rmaterial* et *n*). Exécutez le programme et copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 2.7b**.



Vue de haut de la scène *refraction.txt*.

Question 2.7 :

Combien de niveau(x) de récursivité faut-il au minimum afin d'observer au travers d'un seul objet transparent ? Expliquez en mots le rôle de chacun des rayons récursifs.

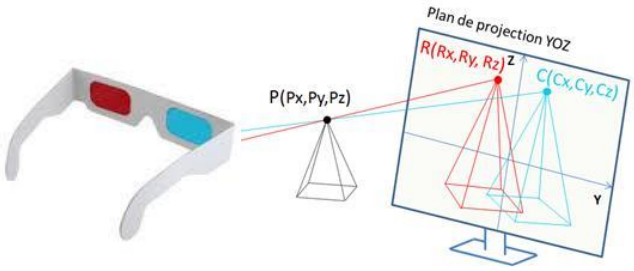
La visualisation d'un modèle

Un autre projet en lien avec le *ray tracer* est la visualisation d'anaglyphe. Ce projet est disponible à l'adresse suivante :

http://projetsmathematiquests.com/projetdetailsmodalsanshead.php?id=22

Ce projet consiste à faire la projection de deux points formant une droite dans la définition d'un modèle 3D à partir de deux points de vue différents (les deux yeux) sur un écran. Cette projection génère deux images que l'on peut regarder à l'aide de lunette à filtre rouge-bleu afin de visualiser le modèle en trois dimensions.

Dans le cadre du projet de *ray tracer*, nous allons visualiser des modèles 3D réalisés lors du projet d'anaglyphe (format AGP) ce qui permettra de comparer deux façons de visualiser un même modèle 3D.



http://projetsmathematiquests.com/projetdetailsmodalsanshead.php?id=22

[3] Une source de lumière ambiante est présente ainsi qu'une source de lumière ponctuelle située à la coordonnée (0.0, 0.0, 5.0) et une source directionnelle ayant l'orientation (1.0, 1.0, -1.0) correspondant à un angle de 45° dans le plan xy et un angle de à 45° vers le bas dans le plan xz.

Un fichier AGP est constitué des informations suivantes et respectant cette convention d'écriture :

#Première séquence des points *x*,*y* et *z* qui sont reliés par des segments de droite :

x_1 x_2 x_3 x_4 ...
y_1 y_2 y_3 x_4 ...
z_1 z_2 z_3 x_4 ...

#Deuxième séquence des points *x*,*y* et *z* qui sont reliés par des segments de droite :

x_1 x_2 x_3 x_4 ...
y_1 y_2 y_3 x_4 ...
z_1 z_2 z_3 x_4 ...

#Troisième séquence des points *x*,*y* et *z* qui sont reliés par des segments de droite :

…

Pour visualiser ce type de modèle dans ce *ray tracer*, nous allons associer chaque segment à la forme d'un tube. Présentement, l'application vous permet de visualiser l'intersection d'un rayon avec plusieurs géométries, mais n'est toujours pas capable de visualiser des tubes. C'est pour cette raison que vous allez ajouter cette géométrie à la collection. La structure de ce programme exploite l'héritage sous la forme suivante :

SGeometry (interface)

↦ **SAbstractGeometry** (propriété de base des géométries)

↦ **STubeGeometry** (contenant P1, P2, R et l'axe du tube S12)

2.8 - Le calcul de l'intersection d'un rayon avec un tube infini

Fichier à modifier : **SGeometricIntersection.java**

Prérequis : 1.4

Fichier de scène : aucun

Dans la classe **SGeometricIntersection** disponible dans le ***package*** **sim.geometry**, vous allez programmer la méthode suivante :

```
public static double[ ] infiniteTubeIntersection(SRay ray, SVector3d r_tube, SVector3d axis, double R)
```

Cette méthode permet d'évaluer l'ensemble des intersections entre un rayon et un tube infini. Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. À l'aide de vos notes de cours[4], modifiez la méthode afin qu'elle retourne adéquatement les temps *t* pour réaliser les multiples intersections possibles entre un rayon et un tube infini.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SGeometricIntersectionTest** du ***package*** **sim.geometry** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Question 2.8 :

Selon vous, pourquoi la méthode précédente a été implémentée avec des paramètres comme la position du tube r_tube et l'orientation de son axe axis et non pas avec la position P1 et P2 correspondant aux deux extrémités d'un tube ?

[4] Cette information est disponible dans la section 6.2c.

2.9 - L'intersection d'un rayon avec un tube

Fichier à modifier : **STubeGeometry.java**

Prérequis : 2.8

Fichier de scène : tubes.txt

Dans la classe **STubeGeometry** disponible dans le ***package* sim.geometry**, vous allez programmer la méthode suivante :

```
public SRay intersection(SRay ray) throws SAlreadyIntersectedRayException
```

Présentement, cette méthode retourne le rayon passé en paramètre signifiant selon la documentation ***javadoc*** de la méthode qu'il n'y a pas d'intersection entre le rayon et le tube. Modifiez cette méthode afin de déterminer adéquatement le temps *t* requis afin qu'un rayon puisse intersecter le tube. N'oubliez pas d'utiliser la méthode qui a été implémentée précédemment.

Utilisez le champ

```
protected SVector3d S12;
```

représentant l'axe du tube partant du point P1 vers le point P2 pour réaliser vos calculs. Il est très important de préciser que cet axe est normalisé.

Puisque le tube est délimité par le point P1 et le point P2, vous devrez imposer une contrainte supplémentaire à votre calcul d'intersection. Implémentez et utilisez la méthode

```
protected boolean isInsideExtremity(SVector3d v)
```

pour appliquer votre contrainte. Utilisez la méthode getPosition(double t) à votre objet ray pour calculer v.

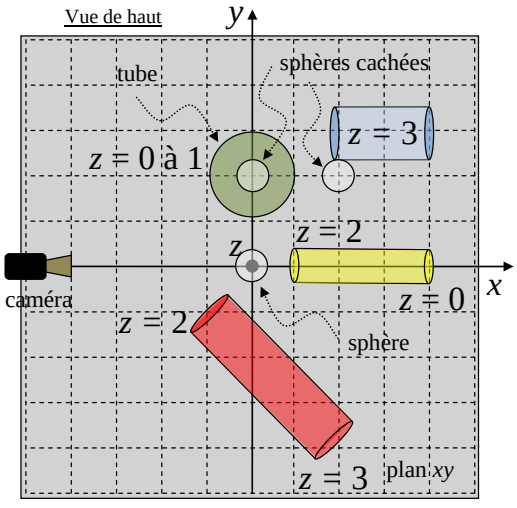
S'il y a une intersection valide, retournez la construction d'un nouveau rayon avec les bons paramètres à l'aide de l'appel de la méthode

```
public SRay intersection(SGeometry geometry, SVector3d normal, double t) throws SAlreadyIntersectedRayException
```

sur l'objet ray et retournez tout simplement le rayon passé en paramètre s'il n'y a pas d'intersection valide.

Pour vérifier l'implémentation de la méthode intersection(...) et isInsideExtremity(...), exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **STubeGeometryTest** du ***package* sim.geometry** située dans le répertoire **test/src**. Corrigez au besoin vos implémentations afin de satisfaire les tests.

Ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène pour *tubes.txt* (*read_data tubes.txt*). Cette scène comprend un plan *xy*, une petite sphère à l'origine, deux sphères cachées ainsi que quatre tubes disposés tel qu'illustré sur le schéma ci-contre. Les surfaces sont réfléchissantes sous un éclairage ponctuelle située à la coordonnée (0, 0, 5) et un éclairage directionnelle d'orientation (1, 1, -1). La caméra est située à la coordonnée (-5, 0, 2) et regarde vers l'origine. Exécutez le programme afin de constater les changements au programme. Si la représentation de la scène vous semble adéquate, copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 2.9**.



Vue de haut de la scène *tubes.txt*.

Question 2.9 :

Selon vous, est-il raisonnable de pouvoir faire réfracter un rayon avec une géométrie comme un tube ? Justifiez votre réponse.

2.10 - Le modèle AGP

Si vous avez déjà réalisé le projet d'anaglyphe, puisque ce projet est réalisé à l'aide du logiciel Excel, vous allez devoir créer votre propre fichier « agp » en respectant la convention décrite précédemment. Vous pouvez tout simplement utiliser des options de « copier-coller » pour transférer vos données du fichier « xls » vers votre fichier « agp ». Utilisez un éditeur de texte comme « notepad » pour écrire votre fichier.

Si vous n'avez pas réalisé le projet d'anaglyphe, vous allez télécharger des modèles 3D de format « agp » qui ont été réalisés par d'autres étudiants à l'adresse suivante :

http://physique.cmaisonneuve.qc.ca/svezina/projet/ray_tracer/download/modele_agp.zip

Décompresser le fichier et rediriger son contenu dans le répertoire de votre projet.

Ouvrez le fichier *modele_agp.txt* qui est localisé dans le répertoire **/SIM/scene**. Modifiez le contenu de ce fichier de scène afin d'y définir le modèle 3D que vous désirez visualiser. Pour ce faire, vous allez modifier le nom du fichier en référence à la construction du modèle 3D par le votre (cherchez les mots clé *model* et *file*). Pour l'instant, vous ne pourrai pas modifier les paramètres d'homothétie (*scale*), de rotation et de translation de votre modèle 3d, car ces fonctionnalités n'ont pas encore été implémentées.

Ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène pour *modele_agp.txt* (*read_data modele_agp.txt*). Exécutez le programme et copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 2.10**.

Question 2.10 :

À partir de la géométrie d'un tube, pouvez-vous formuler une stratégie qui vous permettrait de construire une classe représentant un cylindre en expliquant en quelques phrases comment vous pourriez réaliser le test de l'intersection entre un rayon et le cylindre.

Conclusion

Félicitations ! Vous avez complété la 2e partie de ce laboratoire ! Vous avez maintenant un programme de *ray tracer* avec quelques fonctionnalités d'implémentées permettant de visualiser des sphères et des tubes avec un modèle d'illumination direct et indirect de base. Pour obtenir plus de fonctionnalité, il vous faudra compléter la 3e partie.

Remise

Pour effectuer la remise électronique des fichiers exigés par votre enseignant, transférez vos fichiers dans l'espace de dépôt exigé par votre enseignant (exemple : OMNIVOX/LÉA) :

- Le **répertoire SIM** de votre projet dans un **format compressé** « zip » sous le nom *SIM.zip* comme vous l'avez initialement téléchargé.
- Le fichier WORD *Ray_tracer–Feuille_des_donnees-Partie_2.doc* où vous avez répondu aux questions conceptuelles.