Le *ray tracer* – 1re partie

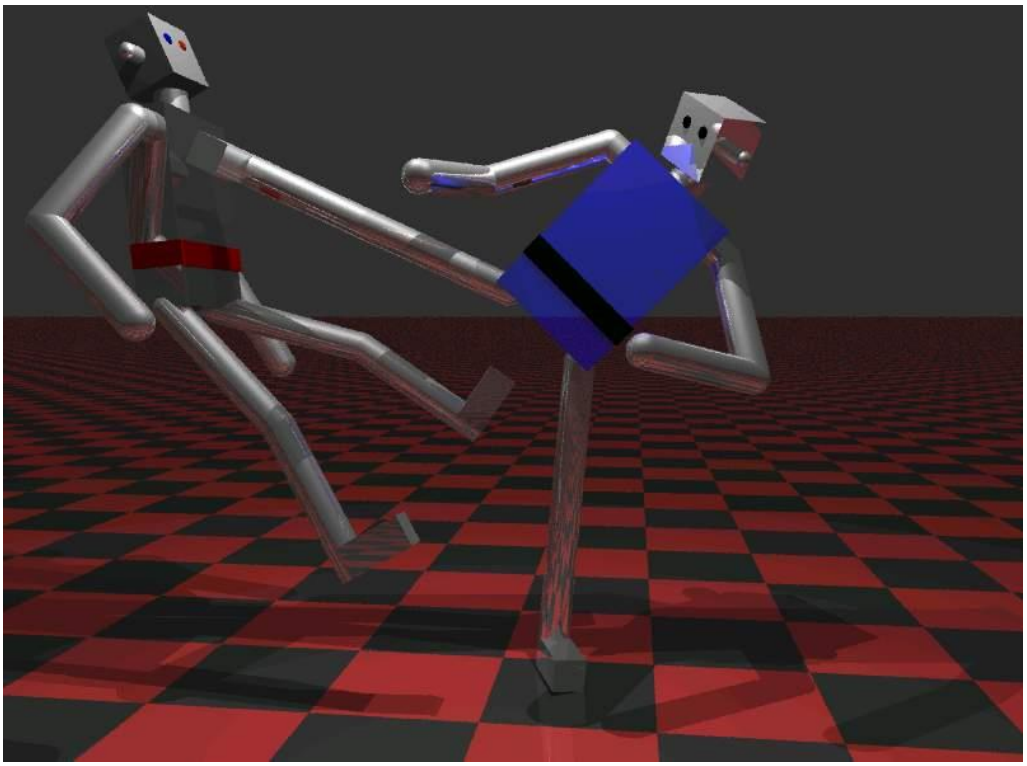


Image générée par Simon Vézina et Yannick Simard

Table des matières

Introduction

Dans ce laboratoire, vous allez étudier différentes fonctionnalités d'un programme de ***ray tracer*** préalablement développé pour vous. Vous allez compléter le programme en rédigeant quelques ***méthodes*** sur des sujets discutés en classe. En quelques mots, vous allez programmer des ***méthodes*** de calcul d'intersection entre un rayon et des géométries, faire des calculs d'illumination, évaluer la déviation de rayon selon les lois de l'optique géométrique, et appliquer des calculs matriciels pour faire la transformation de rayon et de géométrie.

Pour réaliser ce laboratoire, vous avez accès au ***projet Java* SIM**. Vous pouvez télécharger le projet avec lien suivant :

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

Commencez par vous définir un répertoire (exemple « *java* ») qui vous permettra de définir votre ***workspace*** lors de l'ouverture du logiciel ***Eclipse***. Par la suite, décompressez le fichier « *SIM.zip* **»** dans le répertoire de votre ***workspace***. Vous devriez obtenir un répertoire au nom de « *SIM* ».

Ouvrez le logiciel ***Eclipse***. Dans l'onglet ***File***, choisissez l'option ***Switch Workspace*** et identifiez la localisation de votre répertoire de projet (dans l'exemple : « *java* »). Votre ***workspace*** est maintenant configuré.

Dans l'onglet ***File***, choisissez l'option ***Open Projects from File System…*** . Dans la fenêtre ***Import Projects from File System or Archive***, modifiez le champ ***Import source*** à l'aide du bouton ***Directory…*** et sélectionnez le répertoire du projet contenu dans le fichier décompressé « *SIM.zip* » étant de nom *SIM*. Vous avez maintenant configuré votre environnement de développement.

L'application du ***ray tracer*** que vous allez étudier est disponible dans la ***classe* SIMRenderer** disponible dans le ***package* sim.application**. Tout au long du laboratoire, vous aurez à ***exécuter*** la classe **SIMRenderer** et vous pourrez constater l'évolution de l'application au fur et à mesure que vous allez ajouter des fonctionnalités. Vous devrez sauvegarder les images générées en les copiant aux endroits appropriés dans un fichier portant le nom de « *Ray_tracer–Feuille_des_donnees-Partie_1.doc* » jouant le rôle de feuille des données numériques. Vous pouvez télécharger ce fichier avec le lien suivant :

http://physique.cmaisonneuve.qc.ca/svezina/projet/ray_tracer/download/Ray_tracer-Feuille_des_donnees-Partie_1.doc

Vous aurez également à répondre tout au long de ce laboratoire à des questions théoriques à l'aide de courts textes. Vous devrez écrire vos réponses dans le document numérique précédent.

Fichier de configuration

Le programme **SIMRenderer** nécessite la lecture d'un fichier de configuration nommé « *configuration.cfg* ». Ouvrez le fichier *configuration.cfg* avec un éditeur de texte (ex : WORDPAD) ou à partir du logiciel ***Eclipse*** afin d'y voir son contenu. Ce fichier est disponible dans le répertoire ***SIM*** de votre ***workspace***.

Dans la configuration, vous pouvez choisir la **scène** qui sera visualisée par l'exécution du programme en affectant à la variable *read_data* le nom du fichier de scène désiré. Nous allons débuter l'analyse du programme avec la scène *intro_plan.txt* (*read_data intro_plan.txt*). Le résultat du programme sera une image générée en format « .png » qui sera disponible dans le répertoire ***SIM*** de votre ***workspace*** sous un nom défini dans le fichier de **scène**. Un numéro différent sera affecté à tous les images (ex : *image_007.png*) afin d'éviter de réécrire sur des images déjà existantes. On peut configurer ce numéro à l'aide de la variable *viewport_image_count* (ex : *viewport_image_count 1*). Vous pouvez voir en temps réel la création de l'image générée par l'application par l'entremise d'un affichage dans un ***JFrame*** de java. Pour choisir cette option, attribuez à la variable *application* la valeur *frame* (*application frame*). Pour avoir une exécution un peu plus rapide mais sans ***JFrame***, attribuez à la variable *application* la valeur *console* (*application console*).

1.1 – La première exécution du programme

Fichier à modifier : aucun

Prérequis : aucun

Fichier de scène : intro_plan.txt

Dans la fenêtre ***Package Explorer*** du logiciel ***Eclipse***, ouvrez le répertoire **SIM** puis ouvrez le répertoire **src**. Constatez la présence des nombreux ***packages*** nécessaires à l'exécution de ce programme. Ouvrez le ***package*** **sim.application** et lancez l'exécution de la classe **SIMRenderer.java** à l'aide d'un « clic droit » sur le fichier. Sélectionnez dans le ***pop-pop menu*** l'option ***Run As*** et l'autre option ***Java Application***.

Présentement, l'application génère une « image noire ». Vous pouvez confirmer cette affirmation en consultant le fichier image *intro_xxx.png* où l'expression *xxx* représente le numéro de l'image générée par l'application (paramètre défini dans le fichier *configuration.cfg* avec la variable *viewport_image_count*). L'image est disponible dans le répertoire ***SIM*** de votre ***workspace*** (là où vous avez décompressé le fichier *SIM.zip*). Récupérez l'image générée et copiez-là sur votre feuille des données numériques dans la **section 1.1**.

Question 1.1 :

Dans un algorithme de *ray tracer*, décrivez un scénario permettant d'expliquer pourquoi la couleur calculée d'un pixel pourrait être noir ?

1.2 – L'exécution des JUnit Test

Fichier à modifier : aucun

Prérequis : aucun

Pour s'assurer de la qualité d'un programme, il est important de tester les fonctionnalités des différentes méthodes implémentées. L'environnement de développement ***Eclipse*** permet l'exécution de batterie de tests unitaires avec une gestion des succès (***succes***) et des échecs (***fail***).

Un test unitaire réalise l'exécution d'une méthode (ou quelques méthodes) d'une classe ou de plusieurs classes dans un scénario particulier. Le résultat de l'exécution (« ***calculated*** ») est alors comparé avec un résultat attendu (« ***expected*** »). Si le résultat calculé est identique au résultat attendu, le test est un succès. Dans le cas contraire, le test est alors un échec.

https://www.javacodegeeks.com/2013/12/parameterized-junit-tests-with-junitparams.html

Afin de vous familiariser avec l'exécution des ***JUnit Test***, vous allez réaliser l'exécution de l'ensemble des tests préalablement écrit pour vous afin de vous guider dans la qualité de vos implémentations.

Pour ce faire, vous allez :

- Dans la fenêtre ***Package Explorer***, ouvrez le répertoire **SIM** contenant l'ensemble des éléments du projet.
- Sur le répertoire **test/src**, effectuez un « clic droit » et sélectionnez dans le pop-pop menu l'option ***Run As*** et réalisez l'exécution de type ***JUnit Test***.
- (**Si les étapes précédentes ne fonctionnent pas**) Établir le lien avec la librairie ***JUnit 4***. Faire un clic droit sur le répertoire **SIM** dans la fenêtre ***Package Explorer*** et choisir l'option ***Properties***. Dans la propriété ***Java Build Path***, choisir l'onglet ***Libraries*** et activer le bouton ***Add Library***. Choisir ***JUnit*** et activer le bouton ***next***. Choisir ***JUnit 4*** et activer le bouton ***Finish***. Reprenez les étapes précédentes.

Dans la fenêtre ***JUnit***, vous pouvez visualiser l'ensemble des tests effectués pour valider certaines fonctionnalités du projet :

- Si toutes les implémentations sont adéquates, une barre de **couleur verte** sera affichée.
- S'il y a des interruptions d'exécution (Errors) ou des erreurs d'implémentations (Failures) retournant de mauvais résultat, une barre de **couleur rouge** sera affichée.

Dans la fenêtre ***Console***, vous remarquerez la présente de messages indiquant que certains tests n'ont pas été effectués, mais qu'ils sont considérés comme des succès. Ce sont des tests reliés à des méthodes que vous devrez implémenter. Ces méthodes retournent présentement une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Lorsque l'implémentation sera réalisée, le test sera effectué « officiellement ».

Présentement, vous remarquerez que sur 257 tests, il y a quelques tests en échec (Errors : 0 et Failures : 24). Ceci est tout à fait normal puisqu'il y a des méthodes calculant des intersections entre des rayons et des géométries qui ne sont pas implémentés adéquatement (elles retournent toujours « pas d'intersection »). Ce choix permet à l'application de fonctionner, mais ne permet pas de visualiser les géométries. Vous aurez à modifier ces méthodes dans ce laboratoire.

Pour lancer l'exécution d'un nombre restreint de tests, vous n'avez qu'à effectuer le « clic droit » sur le répertoire ou le fichier désiré. Par exemple, vous pouvez lancer le test de la classe **SMathTest** du ***package*** **sim.math** situé dans le répertoire **test/src**. Vous remarquerez qu'il est en succès (**couleur verte**), mais qu'il y a plusieurs tests non effectués.

Question 1.2a : (Identifiez la bonne réponse)

Vous exécutez un ***JUnit Test*** (adéquatement écrit) ciblant uniquement une méthode d'une classe et le résultat du test est un succès, alors

a) Vous avez la certitude que l'implémentation de la méthode est adéquate.
b) Vous avez une certaine confiance sur l'exactitude de l'implémentation de la méthode.
c) Vous avez la certitude que l'implémentation est inadéquate.

Question 1.2b : (Identifiez la bonne réponse)

Vous exécutez un ***JUnit Test*** (adéquatement écrit) ciblant uniquement une méthode d'une classe et le résultat du test est un échec, alors

a) Vous avez la certitude que l'implémentation de la méthode est adéquate.
b) Vous avez une certaine confiance sur l'exactitude de l'implémentation de la méthode.
c) Vous avez la certitude que l'implémentation est inadéquate.

Le calcul vectoriel

Le programme **SIMRenderer** possède quelques implémentations de calculs mathématiques comme le calcul vectoriel et matriciel. En utilisant la classe **SVector3d** disponible dans le ***package*** **sim.math**, vous pourrez écrire sous forme mathématique des équations et les objets de type **SVector3d** employés dans vos équations exécuteront les opérations mathématiques comme vous l'avez appris en mathématique.

Voici quelques exemples :

Définition en mathématique	En informatique
$\vec{A} = 2\vec{i} + 3\vec{j} + 4\vec{k}$ $\vec{B} = \vec{i} - 2\vec{j} - \vec{k}$ $a = 8.0$ $\vec{C}$: un vecteur étant la réponse à un calcul w : un scalaire étant la réponse à un calcul	SVector3d A = **new** SVector3d(2.0, 3.0, 4.0); SVector3d B = **new** SVector3d(1.0, -2.0, -1.0); **double** a = 8.0; SVector3d C; **double** w;

Opération mathématique	En mathématique	En informatique
L'addition de vecteur	$\vec{C} = \vec{A} + \vec{B}$ $\Rightarrow \vec{C} = 3\vec{i} + \vec{j} + 3\vec{k}$	C = A.add(B);
La soustraction de vecteur	$\vec{C} = \vec{A} - \vec{B}$ $\Rightarrow \vec{C} = \vec{i} + 5\vec{j} + 4\vec{k}$	C = A.substract(B);
La multiplication par un scalaire	$\vec{C} = a\vec{A}$ $\Rightarrow \vec{C} = 16\vec{i} + 24\vec{j} + 32\vec{k}$	C = A.multiply(a);
Le module d'un vecteur	$w = \|\vec{A}\|$ $\Rightarrow w = \sqrt{29}$	w = A.modulus();
Le produit scalaire	$w = \vec{A} \cdot \vec{B}$ $\Rightarrow w = -8$	w = A.dot(B);
Le produit vectoriel	$\vec{C} = \vec{A} \times \vec{B}$ $\Rightarrow \vec{C} = 5\vec{i} + 6\vec{j} - 7\vec{k}$	C = A.cross(B);
La normalisation	$\vec{C} = \vec{A} / \|\vec{A}\|$ $\Rightarrow \vec{C} = \frac{2}{\sqrt{29}}\vec{i} + \frac{3}{\sqrt{29}}\vec{j} + \frac{4}{\sqrt{29}}\vec{k}$	C = A.normalize();

En analysant les champs x, y et z la classe **SVector3d**, vous remarquerez qu'ils sont tous ***final***. Ainsi, un vecteur est un objet immutable (ne pouvant pas changer sous l'action d'une méthode). Tout calcul mathématique avec un objet **SVector3d** va générer un nouveau vecteur qui devra être affecté à une variable si vous désirez conserver le fruit du calcul.

1.3 – L'équation d'un rayon

Fichier à modifier : **SRay.java**

Prérequis : aucun

Fichier de scène : aucun

Afin de vous familiariser avec les opérations mathématiques de la classe **SVector3d**, vous allez programmer l'équation du rayon

$$\vec{r}_{\text{ray}} = \vec{r}_0 + \vec{v}t$$

où $\vec{r}_{\text{ray}}$ représente un point le long de la droite d'une rayon à un temps *t* lorsque $\vec{r}_0$ représente l'origine du rayon et $\vec{v}$ l'orientation (vitesse) du rayon.

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

public SVector3d getPosition(**double** t)

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Complétez la méthode afin qu'elle retourne la position $\vec{r}_{\text{ray}}$ du rayon au temps t. Utilisez les champs locaux origin ($\vec{r}_0$) et direction ($\vec{v}$) de la classe **SRay** pour effectuer votre calcul.

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

Question 1.3 :

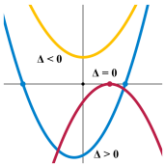
Sachant qu'un rayon est initialement lancé depuis une caméra, qu'elle serait la signification d'utiliser la méthode

public SVector3d getPosition(**double** t)

de la classe **SRay** avec un paramètre t négatif ?

Les polynômes

Dans un programme de *ray tracer*, le calcul des racines réelles d'un polynôme est une opération mathématique très récurant. Ainsi, il est important d'en avoir une implémentation adéquate et efficace.



https://fr.wikipedia.org/wiki/Discriminant
Illustration des racines d'un polynôme du 2e degré.

1.4 - La résolution d'un polynôme du 1er degré

Fichier à modifier : **SMath.java**

Prérequis : aucun

Fichier de scène : aucun

Vous allez maintenant implémenter une méthode permettant d'évaluer la racine réelle d'un polynôme du 1er degré

$$Ax + B = 0 \quad .$$

Dans la classe **SMath** disponible dans le ***package*** **sim.math**, vous allez compléter la méthode suivante :

public static double[ ] linearRealRoot(**double** a, **double** b)

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Bien qu'un polynôme du 1er degré ne possède qu'au plus une racine réelle, nous allons quand même utiliser un tableau pour retourner la solution au calcul. Cette signature de méthode permettra d'harmoniser les signatures des autres méthodes calculant les racines à des polynômes de degré supérieur à 1.

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

Question 1.4 :

Sous quelle condition un polynôme du 1er degré $Ax+B=0$ ne possède pas de racine ?

1.5 - La résolution d'un polynôme du 2e degré

Fichier à modifier : **SMath.java**

Prérequis : 1.3

Fichier de scène : aucun

Vous allez maintenant implémenter une méthode permettant d'évaluer les racines réelles d'un polynôme du 2e degré

$$Ax^2+Bx+C=0 \quad .$$

Dans la classe **SMath** disponible dans le ***package*** **sim.math**, vous allez compléter la méthode suivante :

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

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Lorsque vous allez générer votre tableau des solutions à votre polynôme du 2e degré, assurez-vous de les insérer dans le tableau en ordre croissant (consultez votre prélaboratoire au besoin). Cet effort vous permettra de simplifier grandement la rédaction de plusieurs autres méthodes nécessitant l'usage de celle-ci tout au long de ce laboratoire.

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

Question 1.5 :

Si le temps *t* pour réaliser l'intersection entre un rayon et une géométrie correspond à une racine réelle d'un polynôme du 2e degré de la forme $At^2+Bt+C=0$, que signifie alors une solution sans racine réelle ?

Fichier de scène

Le fichier de scène contient les informations permettant de décrire la **scène** comme les sources de **lumières**, les **primitives**, les **géométries** et les **matériaux** et de décrire les outils de rendu comme le **ray tracer**, la **caméra** et le **viewport**. Ouvrez le fichier *intro_plan.txt* situé dans le répertoire *SIM/scene* de votre ***workspace*** afin de voir la structure du fichier de description de scène.

Sans trop rentrer dans les détails, la structure du fichier est décrite selon la forme suivante :

Explication de la structure	Exemple
« mot clé de l'objet dans de scène » « propriété objet 1 » « valeur » « propriété objet 2 » « valeur » … « mot clé d'objet interne à l'objet courant » « propriété objet interne 1 » « valeurs » « propriété objet interne 2 » « valeur » … « mot clé de fin (end) de l'objet interne » « propriété objet 3 » « valeur » … « mot clé de fin (end) de l'objet de scène »	*primitive* *material_name sphere* *sphere* *position 2.0 4.0 0.0* *ray 0.2* *end* *end*

Le programme permet la lecteur d'un fichier de scène même si certaines propriétés n'ont pas été précisées (exemple : Il n'y a pas de caméra). Des valeurs par défaut ont été choisies lors de l'exécution du programme afin de vous permettre d'écrire une scène sans avoir à tout spécifier. S'il y a une erreur d'écriture dans votre fichier de scène, le logiciel vous indiquera vos erreurs via un « message console » ou lors de la lecture du fichier *log.txt* (écriture du message console dans un fichier texte). Les paramètres erronés lors de la lecture de la scène seront remplacés par les paramètres valides par défaut ce qui permettra l'exécution du programme. Cependant, les résultats ne seront probablement pas ceux désirés.

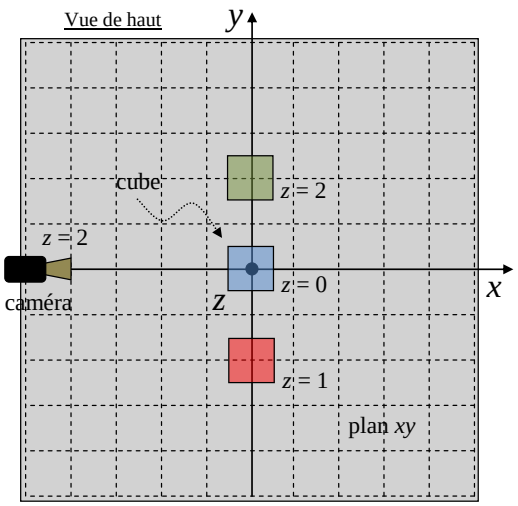
Si vous désirer voir le fichier de votre scène avec l'ensemble des paramètres définis, vous devez exécuter le programme (avec la classe **SIMRenderer**) et consulter le fichier *out.txt* qui est l'écriture complète de votre scène en fichier texte interprété par l'application.

Description de la scène d'introduction

La scène *intro_plan.txt* comprend plusieurs éléments. On y retrouve un plan infini *xy* (*z* = 0) gris, trois cubes de taille égal à 1 de couleur bleu, vert et rouge situés aux coordonnées (0.0, 0.0, 0.0), (0.0, 2.0, 2.0) et (0.0, -2.0, 1.0).

La caméra est située à la coordonnée (-5.0, 0.0, 2.0) et regarde à l'origine.

La scène ne comprend pas de source de lumière (illumination en mode *no_light*). Pour l'instant, l'algorithme de *ray tracer* ne permettra que de déterminer la visibilité des géométries sans effectuer de calcul d'illumination. Ainsi, la couleur qui sera attribuée à chaque pixel sera uniquement la couleur du matériel appliquée sur la géométrie visible au rayon lancé dans le pixel.



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

L'intersection d'un rayon avec une géométrie planaire

Le calcul de l'intersection d'un rayon avec un plan est un calcul très important, car il est utilisé lors de l'intersection d'un rayon avec les géométries comme le plan infini, le disque et le triangle étant des géométrie planaire ainsi que le cube étant composé de six plans. Pour réaliser le calcul de cette intersection, il faut mathématiquement obtenir les racines d'un polynôme du 1er degré.

1.6 - Le calcul de l'intersection d'un rayon avec un plan

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[ ] planeIntersection(SRay ray, SVector3d r_plane, SVector3d n_plane)
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Vous devrez modifier le contenu de cette méthode afin de calculer le temps *t* requis afin qu'un rayon ray puisse réaliser une intersection avec un plan positionné en r_plane dont la normale à la surface est n_plane.

Selon la théorie[1], le temps *t* qui réalise l'intersection s'obtient par la résolution de l'équation

$$At + B = 0$$

où $A = \vec{n} \cdot \vec{v}$ et $B = \vec{n} \cdot (\vec{r}_0 - \vec{r}_P)$ tel que $\vec{n}$ est la normale à la surface du plan, $\vec{r}_P$ est la position du plan, $\vec{v}$ est l'orientation du rayon et $\vec{r}_0$ est l'origine du rayon.

Pour calculer les variables A et B, utilisez les deux méthodes

```
public SVector3d substract(SVector3d v)
public double dot(SVector3d v)
```

de la classe **SVector3d** pour réaliser vos calculs vectoriels. Pour obtenir la racine de votre polynôme, utilisez la méthode

```
public static double[ ] linearRealRoot(double a, double b)
```

de la classe **SMath** que vous avez préalablement implémentée.

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 1.6 :

Lors du calcul de l'intersection entre un rayon et un plan, sous quelle condition le paramètre A de l'équation $At + B = 0$ sera-t-il égal à zéro ? Précisez votre réponse en incluant une description géométrique du plan et du rayon.

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

1.7 - L'intersection d'un rayon avec un plan

Fichier à modifier : **SPlaneGeometry.java**

Prérequis : 1.3 et 1.6

Fichier de scène : intro_plan.txt

Dans la classe **SPlaneGeometry** 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 plan.

À l'aide de la méthode

```
public static double[ ] planeIntersection(SRay ray, SVector3d r_plane, SVector3d n_plane)
```

que vous venez tout juste d'implémenter dans la classe **SGeometricIntersection.java**, commencez par évaluer le temps *t* (qui sera dans un tableau) de l'intersection entre un rayon et le plan. Pour obtenir la position du plan r_plane ainsi que sa normale à la surface n_plane, utilisez les champs locaux de la classe **SPlaneGeometry**. Par la suite, analysez votre tableau des temps *t*.

S'il n'y a pas de temps *t* dans votre tableau, c'est qu'il n'y a pas d'intersection ! Retournez tout simplement le rayon passé en paramètre (rayon non intersecté) à l'aide de l'instruction

```
return ray;
```

S'il y a un temps *t* positif, c'est qu'il y a une intersection entre le rayon et le plan ! Retournez un nouveau rayon intersecté en lançant l'appel de la méthode

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

depuis 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 paramètres normal (afin d'obtenir la normale du bon côté).

Attention :

Pour des raisons numériques, tout temps *t* pour définir une intersection avec un rayon 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. 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 **SPlaneGeometryTest** 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 : *intro_plan.txt*). Si l'image générée vous semble adéquate, copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 1.7**.

Question 1.7 :

Puisqu'il y a un plan *xy* étendu à l'infini, pourquoi y a-t-il du « noir » dans le haut de l'image générée et non la couleur du plan infini ?

1.8 - L'intersection d'un rayon avec un disque

Fichier à modifier : **SDiskGeometry.java**

Prérequis : 1.7

Fichier de scène : intro_disque.txt

Afin d'implémenter l'intersection d'un rayon avec un disque, vous allez modifier la classe **SDiskGeometry** dont la structure hiérarchique est la suivante :

SGeometry (interface)

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

↦ **SPlaneGeometry** (contenant position et surface_normal)

↦ **SDiskGeometry** (permet de restreinte l'intersection du plan)

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

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

Présentement, cette méthode utilise cette même méthode (**super**.intersection(ray);) dont l'implémentation provient de la classe **SPlaneGeometry** héritée par la classe **SDiskGeometry**. Ceci permet d'évaluer l'intersection entre un rayon et un plan infini.

À l'aide du calcul vectoriel disponible dans la classe **SVector3d**, de la variable R de la classe **SDiskGeometry** et de la variable position héritée par la classe **SPlaneGeometry**, établissez un critère permettant d'exclure une intersection à l'extérieur d'un disque de rayon *R* (consultez votre prélaboratoire au besoin).

Si votre test confirme qu'il y a intersection dans le disque, retournez la valeur

```
return ray_plane_intersection;
```

correspondant à un rayon ayant réalisé l'intersection avec le disque et retournez la valeur

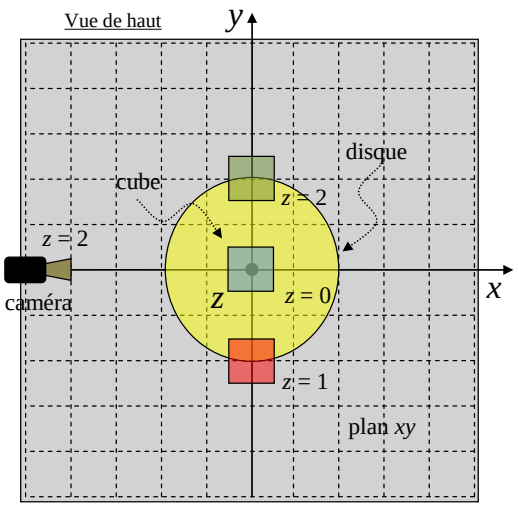
```
return ray;
```

si l'intersection est à l'extérieur du disque.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SDiskGeometryTest** du ***package* sim.geometry** 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_disque.txt* (*read_data intro_disque.txt*). Cette scène est identique à la précédente sauf que l'on a ajouté un disque jaune de rayon égal à 2 à la position (0.0, 0.0, 0.001) tout juste au-dessus du plan xy. La normale du disque est selon l'axe z.

Exécutez le programme afin de constater le changement au programme (scène : *intro_disque.txt*). Si l'image générée vous semble adéquate, copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 1.8a**.



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

Modifiez le fichier *intro_disque.txt* afin de définir le rayon du disque à 3 m (cherchez les mots clé *disk* et *ray*). 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 #1.8b**.

Question 1.8 :

Si l'on voulait générer une géométrie qui aurait la forme d'un « beigne dans un plan », quelles informations auriez-vous de besoin pour former cette géométrie et quelle serait le critère à utiliser permettant d'exclure une intersection dans le « plan du beigne ». Inspirez-vous de la classe **SDiskGeometry** pour formuler votre réponse.

Conclusion

Félicitations ! Vous avez complété la 1re partie de ce laboratoire. Vous avez maintenant un programme de *ray tracer* avec quelques fonctionnalités d'implémentées permettant de visualiser des plans et des disques sans modèle d'illumination. Pour obtenir plus de fonctionnalité, il vous faudra compléter la 2e 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-Partie1.doc* où vous avez répondu aux questions conceptuelles.