Le *ray tracer* – 3e partie

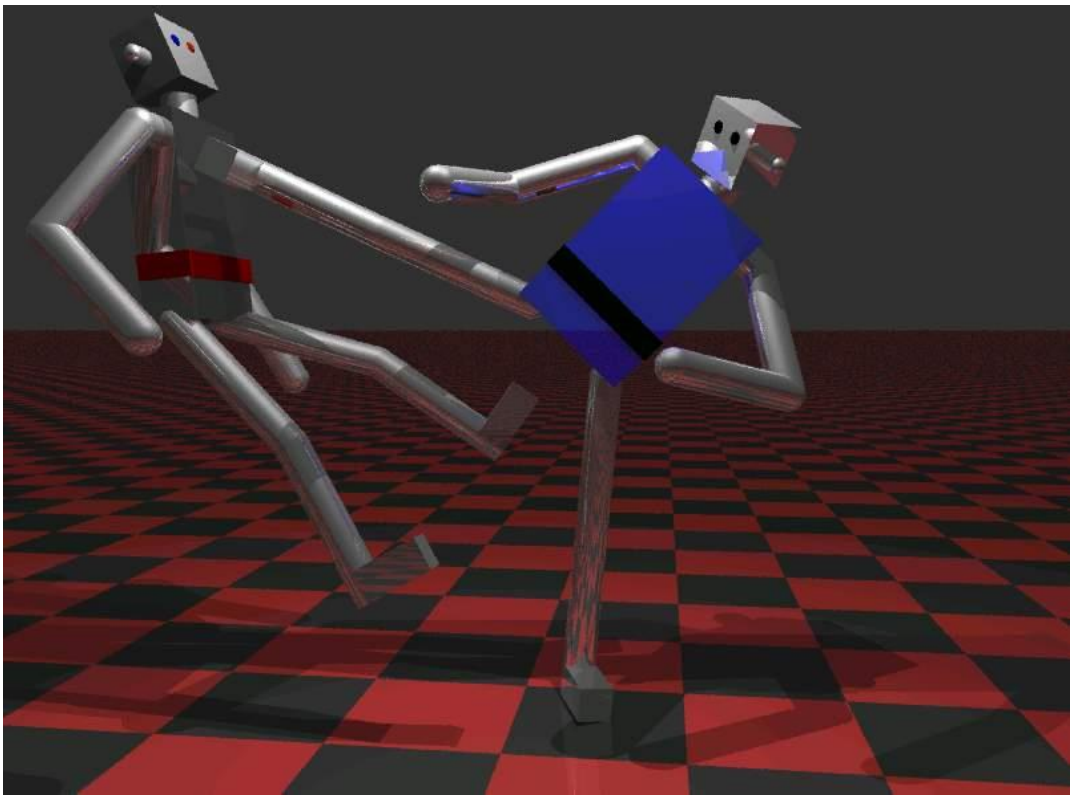


Image générée par Simon Vézina et Yannick Simard

Table des matières

Introduction

À partir de l'application **SIMRenderer** que vous avez développée dans la 1re et 2e partie du laboratoire de ***ray tracer*** , vous allez ajouter des nouvelles fonctionnalités en lien avec les triangles et les transformations affines et visualiser des nouvelles images. Vous devrez ainsi télécharger le document de réponse « *Ray_tracer–Feuille_des_donnees-Partie_3.doc* » disponible au lien

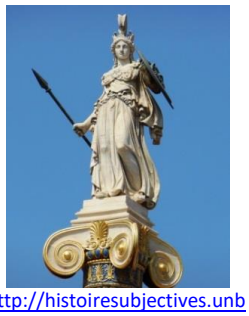
http://physique.cmaisonneuve.qc.ca/svezina/projet/ray_tracer/download/Ray_tracer-Feuille_des_donnees-Partie_3.doc

afin d'y introduire vos images. Vous devez également y inscrire vos réponses aux questions posées tout au long de ce laboratoire. À la fin de cette dernière partie, vous devrez remettre l'intégralité de votre programme afin que son implémentation puisse être complètement évaluée.

Le triangle

Présentement, votre application permet l'intersection d'un rayon avec plusieurs types de géométries, mais ne détermine pas l'intersection avec un triangle qui correspond à la géométrie de prédilection lors de la confection d'un modèle 3d. Vous allez maintenant rédiger des méthodes pour réaliser l'intersection d'un rayon avec un triangle en implémentant deux versions différentes.

Ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène pour *atenea_sans_normale.txt* (*read_data atenea_sans_normale.txt*). Cette scène contient un modèle 3D de format OBJ (Wavefront Technologie) comprenant plus de 1000 triangles représentant la tête de la déesse grecque Athéna.



http://histoiresubjectives.unblog.fr/2014/05/31/athena-la-deesse-adulee/
Statue de la déesse Athéna.

3.1 - La normale à la surface d'un triangle

Fichier à modifier : **SLinearAlgebra.java**

Prérequis : Aucun

Fichier de scène : Aucun

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

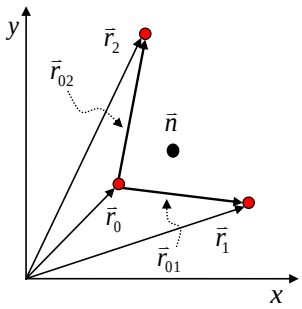
public SVector3d planNormal(SVector3d r0, SVector3d r1, SVector3d r2)

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. Modifiez cette méthode afin qu'elle détermine la normale à un plan formé à l'aide de trois vecteurs position r0, r1 et r2. Puisqu'un plan peut avoir deux orientations admissibles pour sa normale, utilisez l'ordre r0, r1 et r2 avec la règle de la main droite pour effectuer votre calcul. Ainsi, utilisez le produit vectoriel de la façon suivante :

$$\hat{n} = \vec{r}_{01} \times \vec{r}_{02} \quad \text{avec} \quad \vec{r}_{ij} = \vec{r}_j - \vec{r}_i$$

ATTENTION : Il ne faut pas normaliser votre résultat. Ce travail sera géré par une autre méthode.

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



Normale au plan :
$\hat{n} = \vec{r}_{01} \times \vec{r}_{02}$

Question 3.1 :

Dans la classe **STriangleGeometry** représentant un triangle, la classe utilisera lors de son initialisation la méthode

public SVector3d normalizedPlanNormal(SVector3d r0, SVector3d r1, SVector3d r2) **throws** SColinearException

disponible dans la classe **SLinearAlgebra** pour déterminer la normale à la surface du triangle, mais normalisera la normale (mettre le module unitaire). Est-il possible que cette méthode « ne fonctionne pas bien » ? Justifiez votre réponse à l'aide d'un court argument en décrivant le résultat de la méthode (ce qui sera retourné par celle-ci).

3.2 - L'intersection d'un rayon avec un triangle par vecteurs intérieurs

Fichier à modifier : **STriangleGeometry.java**

Prérequis : 2.5 et 3.1

Fichier de scène : atenea_sans_normale.txt

Dans la classe **STriangleGeometry** 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 triangle. À l'aide de vos notes de cours[1], modifiez la méthode afin d'évaluer adéquatement le temps *t* pour réaliser l'intersection entre le rayon et le triangle et vérifiez si l'intersection est à l'intérieur du triangle en utilisant la technique des vecteurs intérieurs.

Pour implémenter votre méthode, utilisez les champs locaux (la normale a déjà été initialisée pour vous)

P0, P1, P2 et normal

et la méthode

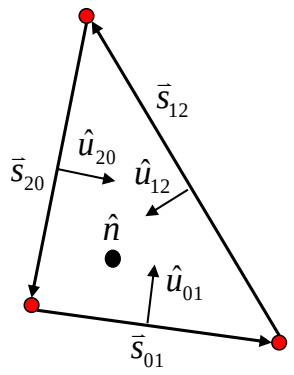
public static double[ ] planeIntersection(SRay ray, SVector3d r_plane, SVector3d n_plane)

disponible dans la classe **SGeometricIntersection** (***package*** **sim.geometry**) pour évaluer le temps *t* de l'intersection. Vérifiez par la suite que l'intersection est à l'intérieur du triangle (s'il y a bien entendu une intersection valide avec le plan du triangle).

Pour ce faire, vous devrez construire[2] des vecteurs représentant les segments $\vec{s}_{01}$, $\vec{s}_{12}$ et $\vec{s}_{20}$ ainsi que les vecteurs intérieurs[3] $\hat{u}_{01}$, $\hat{u}_{12}$ et $\hat{u}_{20}$ afin d'y réaliser les tests de contraintes

$$\left(\vec{r}_{\text{int}} - \vec{P}_0\right)\cdot\hat{u}_{01} > 0 \quad , \quad \left(\vec{r}_{\text{int}} - \vec{P}_1\right)\cdot\hat{u}_{12} > 0 \quad \text{et} \quad \left(\vec{r}_{\text{int}} - \vec{P}_2\right)\cdot\hat{u}_{20} > 0$$

pour vos trois côtés de triangle où $\vec{r}_{\text{int}}$ peut être calculé grâce à la méthode getPosition(double t) appelée par votre objet ray avec le bon temps d'intersection *t*.



Calcule des vecteurs intérieurs à un triangle.

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

[2] Les équations associées aux calculs de ces vecteurs sont disponibles dans la section 6.2b.

[3] La classe **SVector3d** permet de normaliser un vecteur (vecteur de taille 1) avec la méthode normalize( ).

Si l'intersection est à l'intérieur du triangle, 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

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

pour évaluer le bon sens de la normale à la surface du triangle. S'il n'y a pas eu d'intersection adéquate, retournez tout simplement le rayon passé en paramètre à l'aide de l'instruction

```
return ray;
```

Attention : Vous pouvez définir des nouveaux paramètres ***private*** à la classe si vous voulez effectuer des précalculs afin d'accélérer le calcul de l'intersection (ex : évaluer les segments du triangle). Si vous prenez cette direction, assurez-vous de réaliser l'initialisation de vos paramètres dans la méthode

```
private void initialize() throws SInitializationException
```

afin de vous s'assurer que l'initialisation sera réalisée après la lecture et pas uniquement dans le constructeur de la classe.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **STriangleGeometryTest** 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 les changements au programme (scène : *atenea_sans_normale.txt*). Si la représentation du modèle vous semble adéquate, copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 3.2**.

Question 3.2 :

La technique du calcul de l'intersection d'un rayon avec un triangle par vecteurs intérieurs peut également s'appliquer à un ensemble de *N* points. Identifiez une contrainte que l'on doit faire respecter à l'ensemble des *N* points afin qu'ils puissent représenter un polygone adéquat pour l'application de la technique.

3.3 - Le partitionnement de l'espace par voxel

Fichier à modifier : aucun

Prérequis : 3.1

Fichier de scène : teapot.txt

Ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène pour *teapot.txt* (*read_data teapot.txt*). Cette scène contient un modèle 3D représentant une théière comprenant 6320 triangles. Exécutez le programme et remarquez le temps requis pour générer l'image. Présentement, l'algorithme effectue

500	x	500	x	1	x	6320	x	2	=	3 160 000 000
(pixels largeur)		(pixels hauteur)		(récursivité)		(nb triangles)		(nb lumières)		

tests d'intersection pour calculer l'image puisque l'on doit tester l'ensemble des géométries afin d'identifier l'intersection de moindre temps.

Afin de réduire le temps de calcul, il faudra réduire le nombre de tests d'intersection. Pour ce faire, une technique de partitionnement de l'espace par voxel[4] a déjà été implémentée pour vous. Pour l'activer, modifiez dans votre fichier de scène le paramètre *space* dans disponible dans le paramètre *raytracer* et choisissez l'option *voxel* (au lieu de *linear*). Exécutez à nouveau le programme pour constater l'amélioration. En consultant le message console du programme, nous remarquons qu'un voxel contient en moyenne 5,9 primitives/voxels.

Si l'on suppose que la visite d'un seul voxel est nécessaire pour identifier l'intersection de moindre temps, le nombre de tests d'intersection requis pour calculer l'image chute en moyenne à

500	x	500	x	1	x	5,9	x	2	=	2 950 000
(pixels largeur)		(pixels hauteur)		(récursivité)		(nb triangles moyens)		(nb lumières)		

Exécutez le programme et copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 3.3a**.

Question 3.3a :

Sur l'image générée par la scène *teapot.txt*, on y aperçoit une ligne de pixels « mal définie » au centre de l'image. Donnez une explication « géométrique » plausible permettant de justifier cette erreur dans l'image.
Remarque :

- L'objet *teapot* est situé à la coordonnée (0,0,0). La camera est située à la coordonnée (5,5,0) et elle regarde à la coordonnée (0,0,0). Le paramètre *pixel_coordinate* est défini à *top_left*.
- L'usage du partitionnement de l'espace en voxel n'y est pour rien !

Une solution simplement pour corriger l'image générée par le fichier de scène *teapot.txt* peut être obtenue en modifiant un des paramètres de l'élément *raytracer* de ce fichier de scène. Si vous ne savez pas quelles valeurs vous pouvez affecter aux différents paramètres, tentez votre chance ! Un message console sera affiché lors de la lecture de la scène vous signalant votre erreur et en vous proposant des choix valides.

Question 3.3b :

Proposez une solution pour régler le problème du fichier *teapot.txt* en précisant le paramètre changé ainsi que sa nouvelle valeur. Vous n'avez pas le droit de déplacer la caméra ni le modèle 3d !

Lorsque vous avez solutionné le problème, copiez une image corrigée sur votre feuille des données numériques dans la **section 3.3b**.

Le triangle en coordonnée barycentrique

Un triangle en coordonnée barycentrique permet d'attribuer des propriétés particulières à chaque coordonnée à l'intérieur du triangle basées sur des valeurs déterminées aux trois sommets du triangle. Par exemple, on peut interpoler une normale à la surface ainsi qu'une coordonnée de texture ce qui permet d'améliorer grandement les détails que l'on peut attribuer à un modèle 3d.

Ainsi, vous aurez à implémenter le calcul des coordonnées barycentriques d'un triangle, l'interpolation en coordonnée barycentrique ainsi que l'intersection du triangle en coordonnée barycentrique.

http://guy-grave.developpez.com/tutoriels/hardware/les-cartes-graphiques/pipeline-non-programmable/rasterization/
Affichage d'un triangle avec interpolation linéaire des couleurs en coordonnée barycentrique.

[4] Ce sujet est abordé dans les notes de cours dans la section 6.X2.

3.4 - Le calcul et l'analyse des coordonnées barycentriques d'un triangle

Fichier à modifier : **SLinearAlgebra.java**

Prérequis : aucun

Fichier de scène : Aucun

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

```
public static double[ ] triangleBarycentricCoordinates(SVector3d P0, SVector3d P1, SVector3d P2, SVector3d r)
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. À l'aide de vos notes de cours[5], complétez l'implémentation de cette méthode afin d'obtenir la projection du vecteur r en coordonnée barycentrique dans le plan du triangle formé des points P0 , P1 et P2. Retournez vos deux coordonnées barycentriques dans un tableau de type **double** dans l'ordre [b_1, b_2].

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

Ensuite, vous allez programmer la méthode suivante :

```
public static boolean isBarycentricCoordinatesInsideTriangle(double[ ] b_coord) throws SRuntimeException
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. Complétez cette méthode en analysant le tableau de type **double** comprenant des coordonnées barycentriques de triangle. La méthode doit retournée **true** si les coordonnées sont à l'intérieur du triangle et **false** sinon.

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

Question 3.4 :

Si les coordonnées barycentriques satisfaisaient les contraintes $b_1 \geq 0$, $b_2 \geq 0$ et $b_1 + b_2 = 1$, où étaient-elles situées par rapport à leur triangle de référence ?

3.5 - Le calcul de l'interpolation linéaire en coordonnée barycentrique

Fichiers à modifier : **SVector.java**

Prérequis : aucun

Fichier de scène : aucun

Puisque l'action d'interpoler est applicable à tout type de vecteur, vous aurez à implémenter l'interpolation linéaire en coordonnée barycentrique dans l'interface **SVector.**

Les vecteurs qui implémenteront cette interface comme les classes **SVector3d** (pour la normale du triangle) et **SVectorUV** (pour les coordonnées de texture du triangle) pourront ainsi profiter de cette implémentation générique.

[5] Cette information est disponible dans la section 6.2b.

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

public static SVector linearBarycentricInterpolation(SVector v0, SVector v1, SVector v2, **double** t1, **double** t2)

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Utilisez la ***javadoc*** pour vous guider dans votre implémentation.

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

Question 3.5 :

Dans l'interface **SVector**, on propose à toutes les classes désirant implémenter cette interface d'implémenter la méthode add(…) , multiply(…) et dot(…). Cependant, vous remarquerez que la méthode cross(…) réalisant le produit vectoriel entre deux vecteurs ne fait pas partie de cette interface.

Justifiez pourquoi il en est ainsi en basant votre réponse sur le fait que les classes **SVectorUV** et **SVector4d** implémentent l'interface **SVector** (c'est deux classes sont disponibles dans le ***package* sim.math**)

3.6 - L'intersection d'un rayon avec un triangle barycentrique

Fichiers à modifier : **SBTriangleGeometry.java**

Prérequis : 2.5 , 3.4 et 3.5

Fichier de scène : atenea.txt et deer.txt

Avant de passer à la prochaine étape, ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène pour *atenea.txt* (*read_data atenea.txt*). Cette scène contient le même modèle 3D que le précédent, mais avec une définition de normale pour l'ensemble des points des triangles du modèle 3D. Cela vous permettra de construire des triangles en coordonnées barycentriques dont la méthode d'intersection devra être programmée. Le rendu de ce modèle 3D sera ainsi amélioré.

Voici la structure de hiérarchique de la classe **SBTriangleGeometry** :

SGeometry (interface)

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

↦ **STriangleGeometry** (contenant P0, P1, P2 et normal)

↦ **SBTriangleGeometry** (contenant N0, N1, N2, UV0, UV1, UV2)

Dans la classe **SBTriangleGeometry** 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 triangle. Modifiez la méthode afin d'évaluer adéquatement le temps *t* pour réaliser l'intersection entre le rayon et le triangle et vérifiez si l'intersection est à l'intérieur du triangle en utilisant la technique des coordonnées barycentriques.

Pour évaluer le temps *t* de l'intersection dans le plan du triangle, utilisez la méthode :

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

disponible dans la classe **SGeometricIntersection** (***package* sim.geometry**).

Utilisez les méthodes

public static double[ ] triangleBarycentricCoordinates(SVector3d r0, SVector3d r1, SVector3d r2, SVector3d r)

public static boolean isBarycentricCoordinatesInsideTriangle(**double**[ ] b_coord) **throws** SRuntimeException

de la classe **SLinearAlgebra** pour obtenir les coordonnées barycentriques de l'intersection ainsi que pour vérifier si l'intersection est réalisée à l'intérieur du triangle.

Pour évaluer votre normale à la surface de l'intersection, utilisez la méthode d'interpolation linéaire

public static SVector linearBarycentricInterpolation(SVector v0, SVector v1, SVector v2, **double** t1, **double** t2)

de la classe **SVector** que vous avez déjà implémentée où t1 et t2 sont les coordonnées barycentriques b_1 et b_2.

S'il y a eu intersection, retournez un nouveau rayon en lançant l'appel de la méthode

public SRay intersection(SGeometry geometry, SVector3d normal, SVectorUV uv, **double** t) **throws** SAlreadyIntersectedRayException

à l'objet ray (avec l'instruction **return** ray.intersection(**this** , …);). Remarquez que l'on intègre à cet appel un nouveau paramètre : uv. Cette coordonnée de texture[6] doit également être interpolée afin de pouvoir appliquer des textures de couleurs à votre triangle. Utilisez la méthode d'interpolation de la classe **SVector** pour calculer la coordonnée de texture interpolée puisqu'un vecteur **SVectorUV** implémente l'interface **SVector**. Ne pour oublier de « ***caster*** » vos **SVector** après interpolation dans le format approprié (soit **SVector3d** pour la normale et **SVectorUV** pour uv).

Attention : Bien que la méthode pour évaluer la normale au triangle soit nécessaire pour calculer le temps *t* pour réaliser l'intersection, cette normale ne peut par être utilisée pour définir la normale à l'endroit de l'intersection, car elle n'est pas le résultat de la normale interpolée.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SBTriangleGeometryTest** 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 les changements au programme (scène : *atenea.txt*). Si la représentation du modèle vous semble adéquate, copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 3.6a**.

Afin de s'assurer que vous avez adéquatement intégré les coordonnées de texture, téléchargez le fichier de scène *deer.zip* à l'adresse suivante :

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

Décompressez le fichier dans le répertoire de votre projet, ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène pour *deer.txt* (*read_data deer.txt*). Cette scène contient un modèle 3D représentant un chevreuil. Si la représentation du modèle vous semble adéquate, copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 3.6b**.

http://unisci24.com/218592.html
Un chevreuil photographie en plein nature.

Par la suite, ouvrez le fichier de scène et remplacez le paramètre *uv_format* disponible dans la construction du modèle 3d (cherchez le mot clé *model*) par la valeur *origin_uv_top_left* (initialement égal *origin_uv_bottom_left*). Générez la nouvelle image et copiez-la sur votre feuille des données numériques dans la **section 3.6c**.

Question 3.6 :

Selon vous, pourquoi la 1re image du chevreuil (section 3.6b) est adéquate et la seconde image (section 3.6c) est erronée ?

[6] La notion de texture est présentée dans les notes de cours dans la section 6.7.

La transformation par calcul matriciel

Présentement, votre application ne permet pas de transformer des modèle 3D afin de les reproduire et les représenter différemment dans une scène. Vous allez maintenant ajouter des fonctionnalités de transformation en utilisant le calcul matriciel.

Ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène pour *teapot_multiple.txt* (*read_data teapot_multiple.txt*). Cette scène contient le modèle 3D du fichier de scène *teapot.txt*, mais également quatre autres copies de ce même modèle où des matrices de transformation leur ont été appliquées afin de les représenter différemment.

Voici les transformations apportées aux quatre modèles :

Modèle 1 :

$Sc = (1.0, 1.0, 1.0),\ R = (0.0, 0.0, 0.0),\ Tr = (0.0, 0.0, 5.0)$

Modèle 2 :

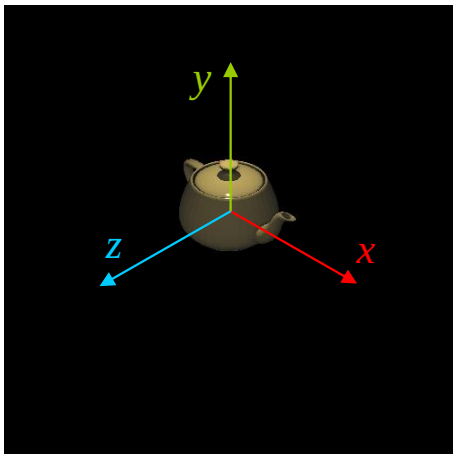
$Sc = (0.5, 0.5, 0.5),\ R = (0.0, 90.0, 0.0),\ Tr = (5.0, 0.0, 0.0)$

Modèle 3 :

$Sc = (0.5, 1.0, 0.5),\ R = (180.0, 0.0, 0.0),\ Tr = (-5.0, 5.0, 0.0)$

Modèle 4 :

$Sc = (1.5, 1.5, 1.5),\ R = (90.0, 90.0, 90.0),\ Tr = (0.0, 0.0, -8.0)$



Modèle sans transformation. Les flèches indiquent le sens des axes, mais ont été ajoutées artistiquement au modèle 3D pour faciliter l'interprétation de l'image.

3.7 - La matrice de transformation objet vers monde

Fichiers à modifier : **SMatrix4x4.java**

Prérequis : aucun

Fichier de scène : aucun

Puisque la transformation d'une géométrie de l'espace objet vers l'espace monde sera nécessaire dans l'implémentation de plusieurs méthodes de ce programme, construire une méthode effectuant ce calcul sera un très bon investissement. C'est pour cette raison que vous allez programmer une méthode calculant la matrice[7] $M_{o\to m}$ qui permettra prochainement d'effectuer l'opération matricielle

$$\vec{r}_m = M_{o\to m}\ \vec{r}_o$$

correspondant à la transformation d'un vecteur position.

Dans la classe **SMatrix4x4** disponible dans le ***package*** **sim.math**, vous devrez programmer la méthode suivante :

public static SMatrix4x4 TrRzyxSc(SVector3d translation, SVector3d rotation, SVector3d scale)

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. Utilisez les méthodes statiques disponibles dans la classe **SMatrix4x4** pour construire vos matrices de transformation (translation, rotation, homothétie) et appliquer des opérations matricielles de multiplication afin d'évaluer la matrice de transformation désirée.

[7] Cette matrice est décrite dans les notes de cours dans la section 6.6. La référence à « o » correspond à « objet » et la référence à « m » correspond à « monde » d'où l'appellation « matrice objet vers monde ».

Pour vérifier votre implémentation, vous allez exécuter la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SMatrix4x4Test** du ***package*** **sim.math** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Question 3.7

Pourquoi est-il impératif de respecter un ordre dans la multiplication de matrice de transformation lors de la construction de la matrice de transformation de l'espace objet vers l'espace monde ?

3.8 - La transformation d'un vecteur position

Fichiers à modifier : **SAffineTransformation.java**

Prérequis : aucun

Fichier de scène : aucun

Dans la classe **SAffineTransformation** disponible dans le ***package*** **sim.math**, vous devrez programmer la méthode suivante :

```
public static SVector3d transformPosition(SMatrix4x4 transformation, SVector3d v)
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. Modifiez cette méthode afin qu'elle effectue le calcul de la transformation linéaire d'une vecteur position à l'aide d'une matrice de transformation.

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

Question 3.8

Sachant que le vecteur origine en 4D correspond à (0, 0, 0, 1), est-ce que celui-ci peut être influencé par une matrice de transformation ? Si oui, décrivez qualitativement l'effet que cette matrice peut avoir sur sa valeur.

3.9 - La transformation d'un triangle par matrice de transformation

Fichiers à modifier : **SModelReader.java**

Prérequis : 3.1, 3.7 et 3.8

Fichier de scène : teapot_multiple.txt

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

```
private SGeometry transformTriangleGeometry(STriangleGeometry triangle)
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. Votre tâche sera de retourner un nouveau triangle transformé de l'espace objet vers l'espace monde à l'aide des vecteurs

scale, rotation et translation

contenus dans la classe **SModelReader** représentent les informations des transformations à apporter aux triangles de ce modèle 3D. Utilisez la classe **SMatrix4x4** disponible dans le ***package*** **sim.math** pour construire vos matrices de transformation permettant de réaliser vos calculs.

Pour arriver à transformer votre triangle, vous avez deux choix possibles :

1) Réaliser la transformation des trois points du triangle avec la matrice de transformation correspondant à votre modèle directement dans la classe **SModelReader** et retourner un nouveau triangle transformé.

2) Implémenter la méthode

   public STriangleGeometry transform(SMatrix4x4 transformation)

   de la classe **STriangleGeometry** disponible dans le ***package*** **sim.geometry** pour fractionner la tâche de la transformation du triangle et utiliser cette méthode sur le triangle en lui donnant comme attribut la matrice de transformation correspondant à votre modèle. Vous pourrez vérifier votre implémentation en exécutant la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **STriangleGeometry** du ***package*** **sim.geometry** située dans le répertoire **test/src**.

Peu importe l'approche que vous prendrez, vous devrez transformer trois points à l'aide des fonctionnalités de la classe **SAffineTransformation** que vous avez programmé précédemment. Puisque le calcul de la normale à la surface se fait de façon interne lors de la construction du triangle de type **STriangleGeometry**, vous n'avez pas à vous soucier de cette transformation.

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

Exécutez le programme afin de constater les changements au programme (scène : *teapot_multiple.txt*). 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 3.9**.

Question 3.9

Pour un **STriangle**, l'ordre des points définit l'orientation de la normale à la surface selon la règle de la main droite. Soit un **STriangle** formé de 3 points dans le plan *XZ*, est-ce que la normale à la surface sera modifiée si

a) le triangle subit une translation (-1, -1, -1) ?	OUI	NON
b) le triangle subit une rotation (0, 180°, 0) ?	OUI	NON
c) le triangle subit une homothétie (-1, -1, -1) ?	OUI	NON

3.10 - La transformation du vecteur orientation et normale

Fichiers à modifier : **SAffineTransformation.java**

Prérequis : 3.8

Fichier de scène : aucun

Dans la classe **SAffineTransformation** disponible dans le ***package*** **sim.math**, vous devrez programmer les deux méthodes suivantes :

public static SVector3d transformOrientation(SMatrix4x4 transformation, SVector3d v)

public static SVector3d transformNormal(SMatrix4x4 transformation, SVector3d v)

Présentement, ces deux méthodes retournent une exception de type **SNoImplementationException**. Modifiez ces deux méthodes afin qu'elles effectuent le calcul de la transformation linéaire d'une vecteur orientation et d'un vecteur normale. Assurez-vous que votre vecteur normal transformé soit unitaire !

Remarque : Vous pouvez utiliser la méthode

public static SVector3d transformOrigin(SMatrix4x4 transformation)

pour réaliser vos calculs. Cependant, vous devrez compléter son implémentation (avec ***JUnit Test*** à l'appuis).

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

Question 3.10

Pourquoi faut-il calculer la transformation d'un vecteur position et la transformation d'un vecteur orientation différemment ?

3.11 - La transformation d'un triangle barycentrique par matrice de transformation

Fichiers à modifier : **SModelReader.java**

Prérequis : 3.7, 3.8 et 3.10

Fichier de scène : mickey_mousse.txt et star_wars.txt

Avant de passer à la prochaine étape, ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène pour *mickey_mouse.txt* (*read_data mickey_mouse.txt*). Cette scène contient exactement **6 copies** d'un modèle 3D représentant le célèbre *Mickey Mouse* où plusieurs transformations lui ont été apportées.

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

```
private SGeometry transformBTriangleGeometry(SBTriangleGeometry triangle)
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. Votre tâche sera retourner un nouveau triangle barycentrique transformée de l'espace objet vers l'espace monde à l'aide des vecteurs

scale, rotation et translation

contenus dans la classe **SModelReader** représentent les informations des transformations à apporter aux triangles de ce modèle 3D. Pour ce faire, vous devrez transformer les trois points du triangle ainsi que les trois normales attitrées à ces points à l'aide des fonctionnalités de la classe **SAffineTransformation** afin de retourner la construction d'un nouveau triangle barycentrique adéquat. Puisque les coordonnées de textures *uv* ne requièrent pas de transformation, vous n'avez qu'à les réaffecter à la construction de votre nouveau triangle.

Si vous désirez fractionner votre tâche et la tester, vous pouvez programmer la méthode

```
public SBTriangleGeometry transform(SMatrix4x4 transformation)
```

de la classe **SBTriangleGeometry** disponible dans le ***package*** **sim.geometry** et la tester avec la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SBTriangleGeometry** du ***package*** **sim.geometry** située dans le répertoire **test/src**.

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

Exécutez le programme afin de constater les changements au programme (scène *mickey_mouse.txt*). Si la représentation de la scène vous semble adéquate (vous devriez voir exactement **6 personnages** sans compte les reflets sur le plancher), copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 3.11a**.

Pour répondre à la question 3.7, téléchargez le fichier de scène *star_wars.zip* à l'adresse suivante :

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

Décompressez le fichier dans le répertoire de votre projet, ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène pour *star_wars.txt* (*read_data star_wars.txt*). Exécutez le programme et copiez le fichier image généré par l'application sur votre feuille des données numériques dans la **section 3.11 b**.

Question 3.11

Le *ray tracer* **SIMRenderer** permet la visualisation d'une géométrie uniquement si celle-ci réfléchie la lumière provenant d'une source. Aucun matériel n'émet de lumière. Identifiez un problème que cela peut engendrer en vous inspirant d'une anomalie visuelle que l'on retrouve dans l'image générée par la scène *star_wars.txt*.

3.12 - La matrice de transformation monde vers objet

Fichiers à modifier : **SMatrix4x4.java**

Prérequis : aucun

Fichier de scène : aucun

Puisque la transformation d'une géométrie de l'espace monde vers l'espace objet sera nécessaire dans le calcul de la transformation d'un rayon de l'espace monde vers l'espace objet pour réaliser un test d'intersection dans l'espace de la géométrie, construire une méthode effectuant ce calcul sera un très bon investissement.

C'est pour cette raison que vous allez programmer une méthode calculant la matrice[8] $M_{\mathrm{m}\to\mathrm{o}}$ qui permettra d'effectuer l'opération matricielle

$$\vec{r}_{\mathrm{o}} = M_{\mathrm{m}\to\mathrm{o}}\ \vec{r}_{\mathrm{m}} \quad .$$

Dans la classe **SMatrix4x4** disponible dans le ***package*** **sim.math**, vous devrez programmer la méthode suivante :

```
public static SMatrix4x4 ScRxyzTr(SVector3d scale, SVector3d rotation, SVector3d translation)
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. Modifiez son contenu afin d'y calculer la matrice appropriée.

Pour vérifier votre implémentation, vous allez exécuter la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SMatrix4x4Test** du ***package*** **sim.math** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Question 3.12 :

Dans la classe **SMatrix4x4Test** disponible dans le répertoire **test/scr/sim/math**, pourquoi le test unitaire java ***JUnit Test*** correspondant à la méthode

```
public void TrRzyxScAndScRxyzTrIdentityTest( )
```

est-il si ingénieux à réaliser pour certifier la qualité de l'implémentation ? Justifiez votre réponse à l'aide de propriétés en lien avec vos matrices.

[8] Cette matrice est décrite dans les notes de cours dans la section 6.6. La référence à « m » correspond à « monde » et la référence à « o » correspond à « objet » et d'où l'appellation « matrice monde vers objet ».

3.13 - La transformation d'un rayon

Fichiers à modifier : **SRay.java**

Prérequis : 3.10

Fichier de scène : aucun

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

```
public SRay transformNotIntersectedRay(SMatrix4x4 transformation) throws SAlreadyIntersectedRayException
```

Présentement, cette méthode retourne une exception de type **SNoImplementationException**. Modifiez cette méthode afin de retourner un nouveau rayon transformé par la matrice de transformation. Puisque le rayon n'est pas intersecté, vous n'avez qu'à transformer l'origine du rayon origin et l'orientation du rayon direction. Retournez le résultat du constructeur

```
private SRay(SVector3d origin, SVector3d direction, double refractive_index, SRay previous_ray) throws
SConstructorException
```

en utilisant les champs locaux pour définir refractive_index et previous_ray.

Pour vérifier votre implémentation, vous allez exécuter 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 3.13 :

Pourquoi est-il important de ne pas normaliser l'orientation (paramètre direction) du rayon après lui avoir appliqué une transformation affine (par matrice de transformation).

3.14 - L'intersection d'un rayon avec une géométrie transformable

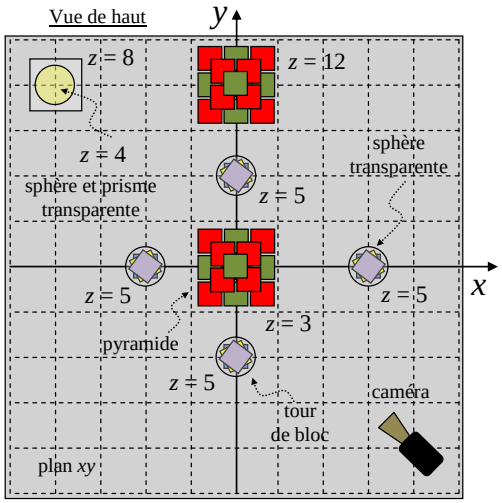
Fichier à modifier : **STransformableGeometry.java**

Prérequis : 2.7, 3.7, 3.12 et 3.13

Fichier de scène : modele_msr.txt

Avant de passer à l'étape suivante, ouvrez le fichier *configuration.cfg* et modifiez le fichier de scène *modele_msr.txt* (*read_data modele_msr.txt*). Cette scène contient plusieurs géométries comme des sphères, des cubes et un plan dont leurs propriétés ont été transformées à l'aide de matrice de transformation et regroupés sous la forme de pyramides et de tours dans des modèles 3D (fichiers *pyramide-9-4-1.msr* et *tour_de_bloc.mrs*).

Avec les matrices de transformation, il est possible de déformer (scale), de faire tourner (rotation) et de déplacer (translation) une géométrie de base (sphère, cube). Ces modifications apportées aux géométries sont décrites par un objet de type **STransformableGeometry** qui a le mandat de contenir une géométrie de base et d'y appliquer les matrices de transformation afin d'y réaliser des tests d'intersection dans l'espace propre à la géométrie.



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

Dans la classe **STransformableGeometry** 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 géométrie transformable. En utilisant vos notes de cours[9], réalisez les calculs nécessaires afin d'y retourner un rayon avec les propriétés adéquates décrivant l'état de l'intersection. Pour ce faire, vous devrez

1- Transformer le rayon de l'espace monde vers l'espace objet de la géométrie en utilisant les champs scale, rotation et translation de la classe représentant les paramètres pour réaliser la transformation objet-monde. Vous aurez à modifier adéquatement ces valeurs pour réaliser votre transformation monde-objet.

2- Réaliser le test de l'intersection avec la géométrie interne geometry (**this**.geometry.intersection(…)) de la classe. S'il y a intersection, vous devrez obtenir le temps *t* de l'intersection et la normale à la surface dans l'espace objet.

3- S'il y a intersection, vous devrez transformer la normale à la surface de la géométrie interne de l'espace objet vers l'espace monde. Utilisez l'appel de la méthode

public SVector3d getOutsideNormal() **throws** SNotIntersectedRayException

à votre rayon intersecté en espace objet pour obtenir la normale à la surface dans l'espace objet. Cette normale est adéquatement orientée dans le sens extérieur à la géométrie interne.

4- S'il y a intersection, 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 (**return** ray.intersection(**this**, …)) afin que le rayon retourné par la méthode soit intersecté avec la géométrie transformable **STransformableGeometry** (**this**) et non la géométrie interne **SGeometry** (geometry) puisque c'est la géométrie transformable qui possède un accès au matériel de type **SMaterial** et non la géométrie interne à cet objet.

5- S'il n'y a pas d'intersection, retournez tout simplement le rayon ray (**return** ray) passé en paramètre à la méthode.

De plus, si l'on veut que la géométrie transformable puisse être transparente (pour la réfraction) avec un comportement adéquat, vous devrez implémenter la méthode suivante :

public boolean isInside(SVector3d v)

Dans votre implémentation, vous devrez transformer adéquatement le vecteur v de l'espace monde vers l'espace objet de la géométrie interne. Utilisez la méthode isInside(…) de l'objet geometry (**this**.geometry.isInside(…)) pour vérifier si le vecteur en espace objet est à l'intérieur de la géométrie interne.

Attention : Vous pouvez définir des nouveaux paramètres ***private*** à la classe si vous voulez effectuer des précalculs afin d'accélérer le calcul de l'intersection, mais réalisez l'initialisation de ces paramètres dans la méthode suivante :

private void initialize() **throws** SInitializationException

[9] La transformation d'un rayon lors d'un test d'intersection a été présentée dans la section 6.6.

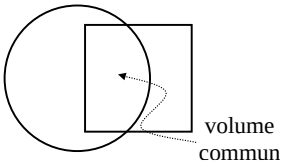
Pour vérifier l'implémentation de la méthode intersection(…) et isInside(…), vous allez exécuter la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **STransformableGeometryTest** du ***package*** **sim.geometry** située dans le répertoire **test/src**. Corrigez au besoin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

Exécutez le programme afin de constater les changements au programme (scène : *modele_msr.txt*). 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 3.14**.

Question 3.14 :

Identifiez une situation problématique pouvant survenir si deux géométries volumiques (pas nécessairement identique) se chevauchent (partage un volume commun).

Exemple :



P.S. Vous n'avez pas à formuler de solution pour résoudre le problème que vous venez d'identifier.

Conclusion

Félicitations ! Vous avez complété la 3e partie de ce laboratoire ! Vous avez maintenant un programme de *ray tracer* avec plusieurs fonctionnalités d'implémentées. L'infrastructure du programme vous permettra également d'intégrer plusieurs autres fonctionnalités si le cœur vous en dit.

Remise

Avant de procéder à la remise, vous allez effacer les informations suivantes de votre projet afin de réduire la taille de celui-ci :

- Le répertoire *deer* contenant les informations du fichier *deer.zip*.
- Le répertoire *star_wars* contenant les informations du fichier *star_wars.zip*.
- Toutes les images générées. Celles importantes ont déjà été copiées dans votre feuille des données numériques.

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_3.doc* où vous avez répondu aux questions conceptuelles.