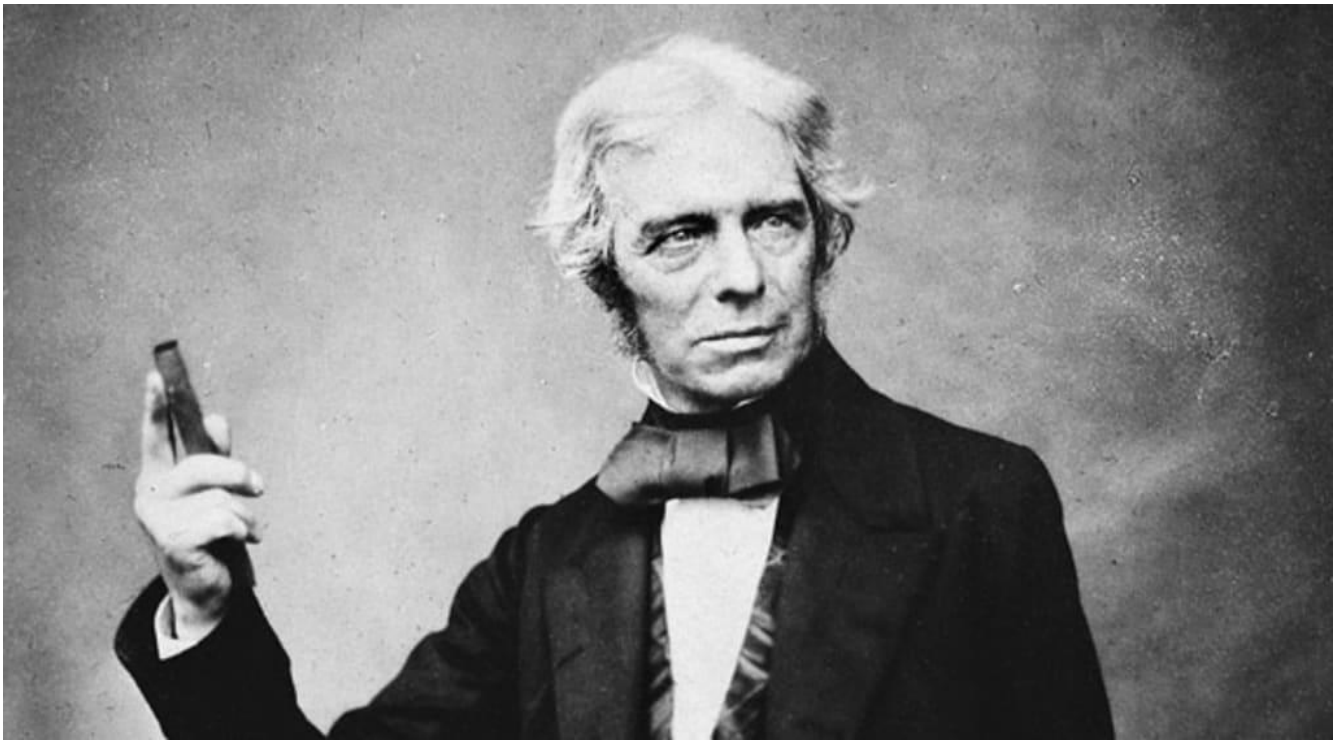
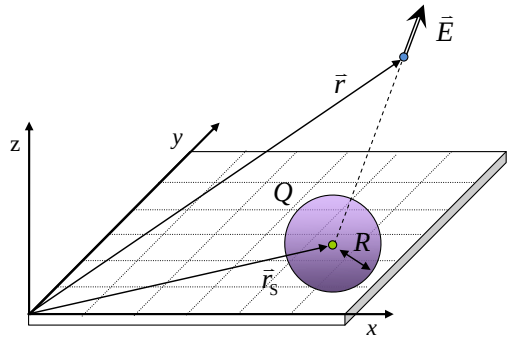
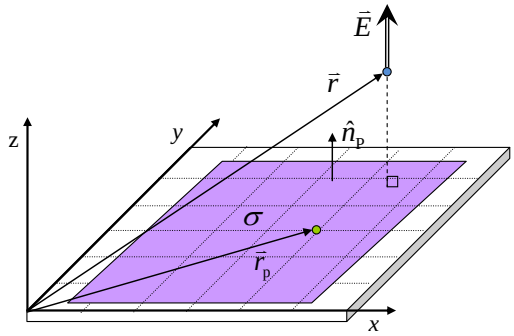
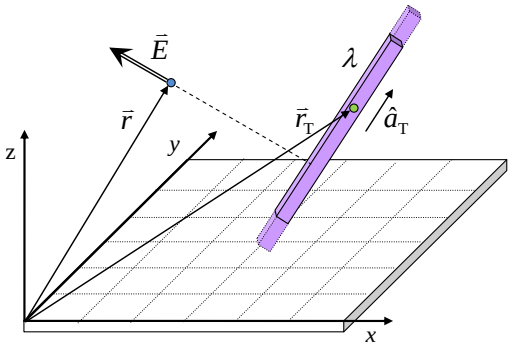
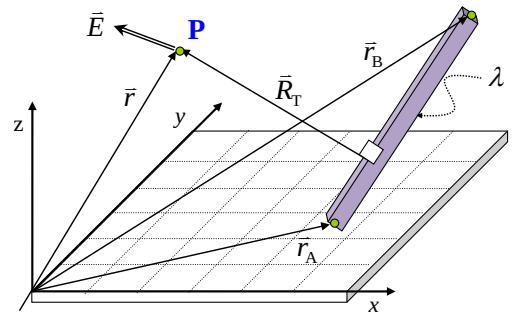
Le champ électrique



https://vocal.media/futurism/michael-faradad

Introduction

Le laboratoire *Le champ électrique* consistera à implémenter des algorithmes de calcul de champs électriques en trois dimensions générés par les distributions de charges suivantes :

Sphère uniformément chargée	Plaque plane infinie uniformément chargée (PPIUC)
	
La tige rectiligne infinie uniformément chargée (TRIUC)	**La tige rectiligne uniformément chargée (TRUC)**
	

Pour ce faire, vous devrez ajouter des fonctionnalités à la classe **SVector3d** afin de réaliser des calculs comme le produit scalaire $C = \vec{A} \cdot \vec{B}$ et le produit vectoriel $\vec{C} = \vec{A} \times \vec{B}$.

Pour visualiser le champ électrique en trois dimensions, vous exploiterez la puissance de la librairie graphique ***OpenGL*** implémentée pour JAVA sous le nom de ***JOGL*** (***Java Binding for the OpenGL API***). Ainsi, si votre ordinateur est munie d'une carte graphique, l'exécution de ce programme sera plus rapide.

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/champ_electrique/download/SIM-ChampElectrique.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-ChampElectrique.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-ChampElectrique.zip* » étant de nom *SIM*. Vous avez maintenant configuré votre environnement de développement.

Vous remarquerez la présence de librairie de référence (Referenced Librairies : Referenced Libraries). Ces librairies ont été préalablement installées dans votre projet afin de faciliter la procédure d'installation. Si vous désirez créer un nouveau projet utilisant les librairies de ***OpenGL***, vous être invités à suivre la procédure suivante :

https://physique.cmaisonneuve.qc.ca/svezina/info/JAVA/installation-jogl-win64.pdf
(procédure à réaliser pour installer JOGL à un nouveau projet JAVA)

Durant le laboratoire, vous devrez répondre à des questions conceptuelles. Vous pouvez télécharger le cahier de réponse avec lien suivant :

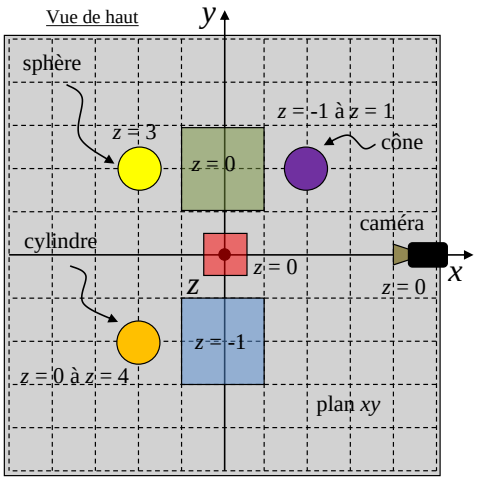
http://physique.cmaisonneuve.qc.ca/svezina/projet/champ_electrique/download/Cahier-ChampElectrique.docx

La description générale du programme

Ce laboratoire comportera deux applications.

L'application **SIMIntroOpenGL** vous permettra de visualiser la scène illustrée sur le schéma ci-contre. Elle validera l'implémentation de nouvelles opérations sur les vecteurs qui seront utilisées pour articuler la caméra.

L'application **SIMElectricFieldViewer** vous permettra de visualiser le champ électrique par la représentation en champ vectoriel. Elle validera l'implémentation de vos calcul liés à la production de champs électriques $\vec{E}$.



Visualition de haut de la scène de l'application SIMIntroOpenGL.

1.1 La première exécution du programme

Fichier à modifier : aucun

Fichier à exécuter : **SIMIntroOpenGL.java**

Prérequis : aucun

Dans la fenêtre ***Package Explorer*** du logiciel ***Eclipse***, ouvrez le répertoire **SIM** puis ouvrez le répertoire **src**. Constatez la présence de quelques ***packages*** nécessaires à l'exécution de ce programme. Ouvrez-le ***package*** **sim.application** et lancez l'exécution de la classe **SIMIntroOpenGL.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***.

Vous êtes présentement plongés dans un environnement 3d où il y a la présence de quelques formes géométriques (3 cubes, une sphère, un cylindre ainsi que d'un cône) que vous visualisez avec un angle d'ouverture de 60°. Vous constaterez que vous ne pouvez ni déplacer la caméra ni la tourner. Ces fonctionnalités seront implémentées prochainement.

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***.

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 **couleur verte** sera affichée (Failures : 0).
- S'il y a des implémentations inadéquates, une **couleur rouge** sera affichée (Failures : « nombre > 0 »).

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 ».

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/math***. Vous remarquerez qu'il est en succès (**couleur verte**), mais qu'il y a des tests non effectués.

Le calcul vectoriel en JAVA

La classe **SVector3d** disponible dans le ***package*** **sim.math** vous permettra, grâce à sa structure ***orientée-objet***, d'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. Plusieurs méthodes ont déjà été implémentées pour vous et vous devrez en ajouter deux autres : le produit scalaire et le produit vectoriel.

Voici quelques exemples afin de vous illustrer comment exploiter cette classe dans votre code :

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	$w = \lVert\vec{A}\rVert$ $\Rightarrow w = \sqrt{29}$	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	$\vec{C} = \vec{A} / \lVert\vec{A}\rVert \Rightarrow$ $\vec{C} = \frac{2}{\sqrt{29}}\vec{i} + \frac{3}{\sqrt{29}}\vec{j} + \frac{4}{\sqrt{29}}\vec{k}$	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 (un peu comme vous devez faire pour sauvegarder un calcul réalisé avec la classe de **String**).

2.1 Le produit scalaire

Fichier à modifier : **SVector3d.java**

Prérequis : aucun

Dans la classe **SVector3d** disponible dans le ***package*** **sim.math**, vous allez programmer la méthode suivante :

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

L'objectif de cette méthode sera de réaliser le calcul du produit scalaire pour un vecteur à 3 dimensions :

$$\vec{A}\cdot\vec{B} = A_x B_x + A_y B_y + A_z B_z$$

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction `throw new SNoImplementationException( );` et complétez l'implémentation en utilisant les champs x, y et z de la classe pour réaliser vos calculs où ceux-ci représentent les composantes de votre vecteur.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SVector3dTest** du ***package*** **sim.math** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Répondez à la question suivante dans le cahier de réponse :

Question 2.1 :

En physique mécanique, le produit scalaire $\vec{A}\cdot\vec{B}$ est présenté à l'aide de l'équation suivante : $\vec{A}\cdot\vec{B} = \|\vec{A}\|\,\|\vec{B}\|\cos(\theta)$. Identifiez dans quel contexte un tel calcul vous a été pertinent.

2.2 Le produit vectoriel

Fichier à modifier : **SVector3d.java**

Prérequis : aucun

Dans la classe **SVector3d** disponible dans le ***package*** **sim.math**, vous allez programmer la méthode suivante :

```
public SVector3d cross(SVector3d v)
```

L'objectif de cette méthode sera de réaliser le calcul du produit vectoriel suivant :

$$\vec{A}\times\vec{B} = \left(A_y B_z - A_z B_y\right)\vec{i} - \left(A_x B_z - A_z B_x\right)\vec{j} + \left(A_x B_y - A_y B_x\right)\vec{k}$$

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction `throw new SNoImplementationException( );` et complétez l'implémentation en utilisant les champs x, y et z de la classe pour réaliser vos calculs où ceux-ci représentent les composantes de votre vecteur.

Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SVector3dTest** du ***package*** **sim.math** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Répondez à la question suivante dans le cahier de réponse :

Question 2.2 :

En physique mécanique, le produit vectoriel $\vec{A}\times\vec{B}$ est présenté à l'aide de l'équation suivante : $\vec{A}\times\vec{B}=\|\vec{A}\|\,\|\vec{B}\|\sin(\theta)\hat{n}$. Identifiez dans quel contexte un tel calcul vous a été pertinent.

La caméra

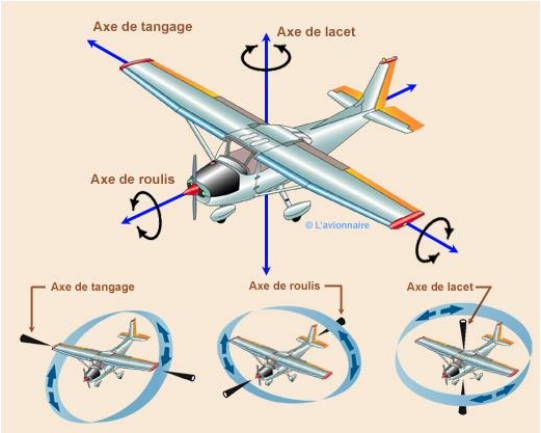
Afin de bien réaliser la projection d'une scène en 3D sur l'écran de vue (*viewport*) en ***OpenGL***, il faut bien définir les caractéristiques d'une caméra que nous allons déplacer en trois dimensions comme un avion. Il existe plusieurs implémentations. L'usage du quaternion est traditionnellement de mise, mais cet outil mathématique ne vous a pas encore été enseigné. C'est pour cette raison que nous exploiterons les opérations mathématiques sur les vecteurs que vous connaissez.

Les caractéristiques géométriques de base d'une caméra (classe **SCamera** du ***package*** **sim.graphics.camera**) sont les suivantes :

- La position (position)
- L'axe du devant (front)
- L'axe du haut (up)
- L'axe de droite (right)

Pour réaliser la rotation de la caméra, nous exploiterons trois types de rotation :

- La rotation lacet (*yaw*) autour de l'axe bas.
- La rotation tangage (*pitch*) autour de l'axe droit.
- La rotation roulis (*roll*) autour de l'axe devant.



https://www.lavionnaire.fr/CelluleGouvernes.php

3.1 Le déplacement de la caméra

Fichier à modifier : **SMovableCamera.java** (héritière de la classe **SCamera**)

Fichier à exécuter : **SIMIntroOpenGL.java**

Prérequis : aucun

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

```
protected void move(SVector3d direction, double distance)
```

L'objectif de cette méthode sera de changer la position de la caméra à l'aide d'un déplacement correspondant à l'équation

$$\vec{r}=\vec{r}_0+\vec{o}\,d$$

où $\vec{r}$ est la position finale de la caméra, $\vec{r}_0$ est la position initiale de la caméra, $\vec{o}$ est l'orientation du déplacement (direction) et d est la distance à parcourir (distance). Utilisez le champ **this**.position de la classe héritière **SCamera** pour obtenir $\vec{r}_0$ et sauvegardez le résultat de votre calcul dans cette même variable.

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

Si vous le désirez, vous pouvez consulter les implémentations des méthodes

```
public void moveFront(double distance)
public void moveRight(double distance)
public void moveUp(double distance)
```

qui vous permettra de comprendre comment la méthode que vous venez tout juste de programmer sera exploitée pour réaliser des déplacements dans les trois directions principales : devant, droite, haut

Afin de constater l'ajout de cette méthode au fonctionnement de la caméra, lancez l'exécution de la classe **SIMIntroOpenGL.java** disponible dans le ***package*** **sim.application**.

Pour déplacer la caméra, vous devez enfoncer le « ***clic-droit*** » de la souris et déplacez celle-ci sur la fenêtre de l'application horizontalement et verticalement. Vous constaterez que la caméra peut se déplacer vers l'avant et l'arrière ainsi que sur la droite et la gauche.

Répondez à la question suivante dans le cahier de réponse :

Question 3.1 :

À quelle équation de la physique mécanique l'équation $\vec{r} = \vec{r}_0 + \vec{o}\,d$ vous fait penser ? Écrivez votre équation physique comparative et comparez brièvement les unités de ces deux formules.

3.2 La rotation *yaw*

Fichier à modifier : **SVectorCamera.java** (héritière de la classe **SMovableCamera**)

Fichier à exécuter : **SIMIntroOpenGL.java**

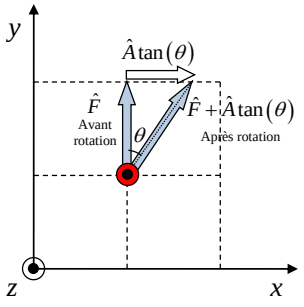
Prérequis : 2.2

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

```
protected void rotationYaw(double degrees) throws IllegalArgumentException
```

L'objectif de cette méthode sera de réaliser la rotation lacet (*yaw*) de la caméra à l'aide de l'algorithme de la rotation par addition de vecteur.

Cet algorithme a pour but d'ajouter au vecteur devant (**this**.front) un vecteur de longueur appropriée pour réorienter le devant dans une nouvelle direction pour correspondre à la bonne rotation (voir schéma ci-contre). Dans le cas de la rotation lacet, il faut ajouter le vecteur droit au vecteur devant pour réaliser la rotation.



À partir des définitions et des équations suivantes, implémentez l'algorithme de la rotation lacet par addition de vecteur dans le but de modifiez adéquatement les champs **this**.front, **this**.right pour réaliser une rotation d'un angle θ (en degrés) dans le sens du vecteur $\hat{A}$ qui sera dans ce cas-ci l'orientation droite. Le champ **this**.up restera inchangé puisque la rotation s'effectue autour de l'axe haut :

$$\hat{A} \leftarrow \hat{R}$$
(définir le vecteur d'addition)

$$\hat{F} \leftarrow \hat{F} + \hat{A}\tan(\theta)$$
(à normaliser)

$$\hat{R} \leftarrow \hat{F} \times \hat{U}$$
(à normaliser)

$\hat{U}$ reste inchangé

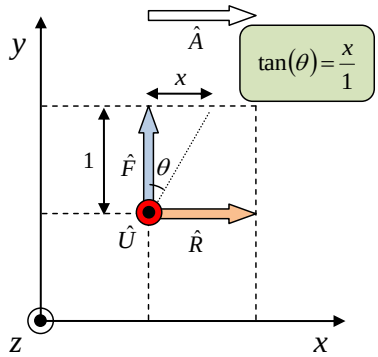
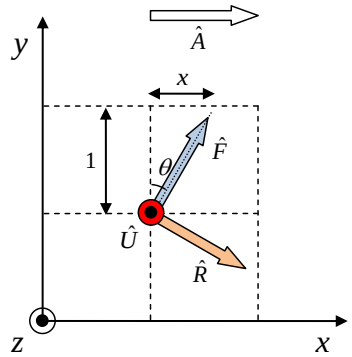


Schéma décrivant le module à ajouter.



Résultat de la rotation lacet (*yaw*).

- $\hat{A}$: Vecteur unitaire désignant le sens pour réaliser la rotation par addition.
- $\hat{F}$: Vecteur unitaire désignant l'avant (*front*). Ce paramètre correspond à **this**.front.
- $\hat{U}$: Vecteur unitaire désignant le haut (*up*). Ce paramètre correspond à **this**.up.
- $\hat{R}$: Vecteur unitaire désignant la droite (*right*). Ce paramètre correspond à **this**.right.
- θ : L'angle de rotation à réaliser (en degrés).

Remarque :

- Pour normaliser un vecteur, utilisez la méthode normalize( ) disponible dans la classe **SVector3d**.
- La méthode Math.tan( ) consomme des radians et non des degrées. Utilisez la classe **Math** pour convertir vos degrés en radians.

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

Afin de constater l'ajout de cette méthode au fonctionnement de la caméra, lancez l'exécution de la classe **SIMIntroOpenGL.java** disponible dans le ***package*** **sim.application**.

Pour changer l'orientation de la caméra en rotation lacet, vous devrez enfoncer le « ***clic-gauche*** » de la souris et effectuer un mouvement horizontal avec celle-ci.

3.3 La rotation *pitch*

Fichier à modifier : **SVectorCamera.java** (héritière de la classe **SMovableCamera**)

Fichier à exécuter : **SIMIntroOpenGL.java**

Prérequis : 2.2

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

```
protected void rotationPitch(double degrees) throws IllegalArgumentException
```

L'objectif de cette méthode sera de réaliser la rotation tangage (*pitch*) de la caméra à l'aide de l'algorithme de la rotation par addition de vecteur. Cet algorithme a pour but d'ajouter au vecteur devant (**this**.front) un vecteur de

longueur appropriée pour réorienter le vecteur devant dans une nouvelle direction pour correspondre à la bonne rotation. Dans le cas de la rotation tangage, il faut ajouter le vecteur haut au vecteur devant.

À partir des définitions précédentes et des équations suivantes, implémentez l'algorithme de la rotation tangage par addition de vecteur dans le but de modifiez adéquatement les champs **this**.front, **this**.up pour réaliser une rotation d'un angle θ (en degrés) dans le sens du vecteur $\hat{A}$ qui sera dans ce cas-ci l'orientation haut. Le champ **this**.right restera inchangé puisque la rotation s'effectue autour de l'axe droit :

$$\hat{A} \leftarrow \hat{U}$$
(définir le vecteur d'addition)

$$\hat{F} \leftarrow \hat{F} + \hat{A}\tan(\theta)$$
(à normaliser)

$$\hat{U} \leftarrow \hat{R} \times \hat{F}$$
(à normaliser)

$\hat{R}$ reste inchangé

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

Afin de constater l'ajout de cette méthode au fonctionnement de la caméra, lancez l'exécution de la classe **SIMIntroOpenGL.java** disponible dans le ***package*** **sim.application**.

Pour changer l'orientation de la caméra en rotation tangage, vous devrez enfoncer le « ***clic-gauche*** » de la souris et effectuer un mouvement vertical avec celle-ci.

3.4 La rotation *roll*

Fichier à modifier : **SVectorCamera.java** (héritière de la classe **SMovableCamera**)

Fichier à exécuter : **SIMIntroOpenGL.java**

Prérequis : 2.2

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

```
protected void rotationRoll(double degrees) throws IllegalArgumentException
```

L'objectif de cette méthode sera de réaliser la rotation roulis (*roll*) de la caméra à l'aide de l'algorithme de la rotation par addition de vecteur. Cette fois-ci, vous devrez construire votre propre algorithme afin de réaliser la rotation roulis adéquate tout en vous inspirant de l'approche par addition de vecteur.

Modifiez les champ **this**.up et **this**.right adéquatement tout en maintenant le champ **this**.front inchangé puisque la rotation sera effectuée autour de l'axe devant. N'oubliez pas de normaliser vos vecteurs !

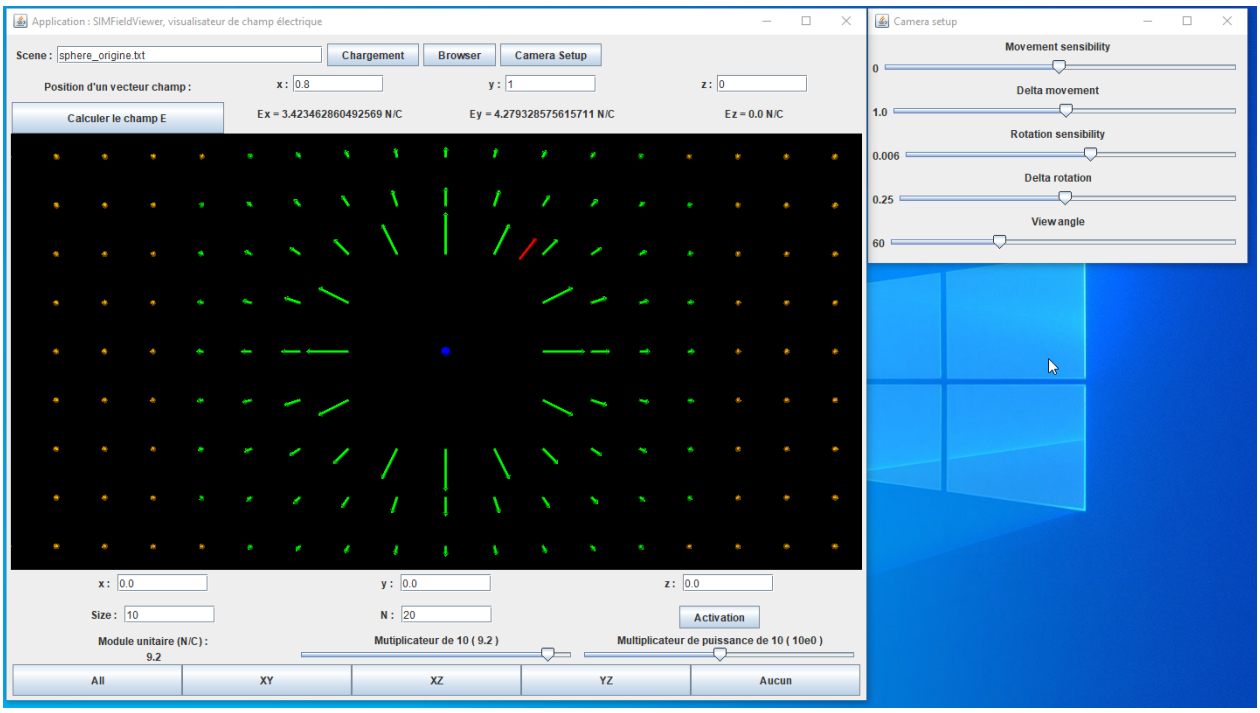
Pour vérifier votre implémentation, exécutez la batterie de tests unitaires de java ***JUnit Test*** disponible dans la classe **SVectorCameraTest** du ***package*** **sim.graphics.camera** située dans le répertoire **test/src**. Corrigez au besoin votre implémentation afin de satisfaire les tests.

Afin de constater l'ajout de cette méthode au fonctionnement de la caméra, lancez l'exécution de la classe **SIMIntroOpenGL.java** disponible dans le ***package*** **sim.application**.

Pour changer l'orientation de la caméra en rotation roulis, vous devrez enfoncer le « ***clic-gauche*** » et le « ***clic-droit*** » de la souris et effectuer un mouvement horizontal avec celle-ci.

Le champ électrique

Afin de visualiser le champ électrique en champ vectoriel dans ce laboratoire, vous aurez l'occasion d'utiliser l'application **SIMElectricViewer** disponible dans le ***package*** **sim.application**. Cette application vous permettra de :



1) Sélectionner un fichier de scène (bouton *Browser*) en cherchant dans le répertoire « *field_viewer* ».
2) Charger une scène (bouton *Chargement*) qui va définir le champ électrique.
3) Calculer le champ électrique $\vec{E} = \left(E_x, E_y, E_z\right)$ à une coordonnée $\vec{r} = \left(x, y, z\right)$ (bouton *Calculer le champ E*). Ce champ sera illustré à l'aide d'une flèche <u>rouge</u>.
4) Déplacer une caméra en 3d (usage de la *souris*).
5) Ajuster la sensibilité les mouvements de la caméra (bouton *Camera Setup*).
6) Positionner le centre de visualisation du champ électrique $\vec{E}$ (***TextField*** **x**, **y** et **z** sous la fenêtre de dessin).
7) Définir la taille de la zone d'affichage (*Size*) du champ électrique et le nombre d'élément fini (*N*) calculé dans la zone.
8) Dessiner le champ électrique dans la zone définie à l'aide de flèche. Il y aura trois types de flèche :
    - Flèche <u>verte</u> : La longueur de la flèche est proportionnelle à l'échelle d'affichage.
    - Flèche <u>orange</u> : La longueur de la flèche est plus petite que l'échelle d'affichage.
    - Flèche <u>jaune</u> : La longueur de la flèche est plus grande que l'échelle d'affichage.
9) Visualiser l'origine du système de coordonnée *xyz* à l'aide d'un point <u>bleu</u>.
10) Réguler la longueur des flèches de champ électrique à l'aide de deux barres glissières (***Slider***) « *Multiplicateur de 10* » et *Multiplicateur de puissance de 10* ».
11) Cinq boutons (*All*, *XY*, *XZ*, *YZ*, *Aucun*) limittant les plans d'affichage du champ électrique (pour réduire le nombre de flèche dessiné).

4.1 La sphère uniformément chargée

Fichier à modifier : **SElectrostatics.java**

Fichier à exécuter : **SIMElectricFieldViewer.java**

Fichier de scène : *Chap1.4-SA.txt* et *Chap1.4-SC.txt*

Prérequis : aucun

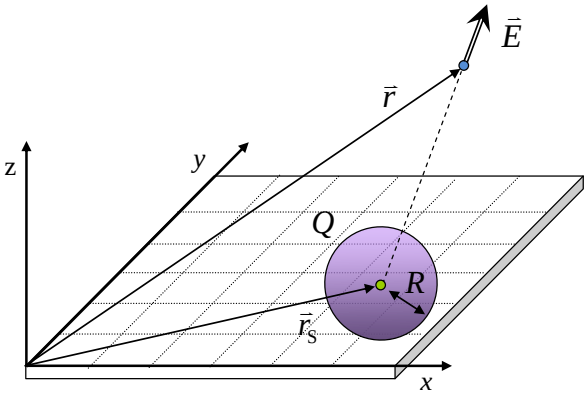
Dans la classe **SElectrostatics** disponible dans le ***package*** **sim.physics**, vous allez programmer la méthode suivante :

```
public static SVector3d sphereElectricField(SVector3d r_S, double R, double Q, SVector3d r)
```

L'objectif de cette méthode sera de calculer le champ électrique générée par une sphère uniformément chargée :

$$\vec{E} = \begin{cases} kQ\dfrac{\vec{R}_S}{\left\|\vec{R}_S\right\|^3} & \text{si } \left\|\vec{R}_S\right\| > R \\ 0 & \text{si } \left\|\vec{R}_S\right\| \le R \end{cases}$$

avec $\vec{R}_S = \vec{r} - \vec{r}_S$



Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction `throw new SNoImplementationException( );` et complétez l'implémentation en exploitant la position de la sphère r_S, le rayon de la sphère R, la charge de la sphère Q ainsi que la coordonnée r où l'on désire évaluer le champ électrique. N'oubliez pas d'utiliser la constante k déjà définie dans la classe **SElectrostatics** pour évaluer votre constante de Coulomb k.

Remarquez que si r est à l'intérieur de la sphère, vous devrez retourner le vecteur zéro (`return NO_ELECTRIC_FIELD;`).

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

Afin de constater la fonctionnalité de cette méthode, lancez l'exécution de la classe **SIMElectricViewer.java** disponible dans le ***package*** **sim.application** où la scène *Chap1.4-SA.txt* aura été déterminé par défaut. Cette scène vous permettra de visualiser le champ électique associé à la **Situation A :** ***Le champ électrique vectoriel d'une particule*** de la section 1.4 de vos notes de cours. Appuyez sur le bouton « Chargement » pour charger la scène.

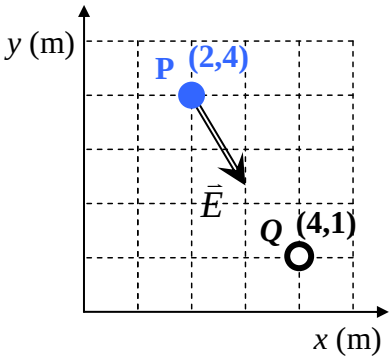


Schéma de la situation A de la section 1.4.

Visualisez l'ensemble du champ électrique afin de confirmer qu'il est conforme avec le champ généré par une sphère uniformément chargée et vérifiez que le champ à la coordonnée $\vec{r} = (2,4,0)\,\text{m}$ est bel et bien égal à $\vec{E} = (1{,}54\vec{i} - 2{,}30\,\vec{j}) \times 10^3\ \text{N/C}$.

Pour répondre à la prochaine question conceptuelle, vous devrez ouvrir le fichier de scène *Chap1.4-SC.txt*. Pour ce faire, vous pouvez à votre choix :

1) Utiliser le bouton « browser » de l'application et sélectionner le fichier *Chap1.4-SC.txt* qui sera localisé dans le répertoire « field_viewer ». Appuyez sur le bouton « Chargement » pour activer la scène.
2) Ouvrez le fichier *field_configuration.cfg* dans le répertoire de votre projet avec un éditeur de texte et modifiez le fichier de scène pour *Chap1.4-SC.txt* (*read_data Chap1.4-SC.txt*).

Répondez à la question suivante dans le cahier de réponse :

Question 4.1 :

Dans le fichier de scène *Chap1.4-SC.txt* (**Situation C : *Deux sphères chargées*, section 1.4**), on peut y observer le champ électrique généré par deux sphères uniformément chargées (distribution en surface).

Décrivez un problème observable avec cette implémentation du champ électrique si l'on désirait à la place illustrez le champ électrique généré par deux sphères conductrices uniformément chargée ?

4.2 La plaque plane uniformément chargée (PPIUC)

Fichier à modifier : **SElectrostatics.java**

Fichier à exécuter : **SIMElectricFieldViewer.java**

Fichier de scène : *Chap1.9-SA.txt* et *plaques_et_spheres.txt*

Prérequis : 2.1

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

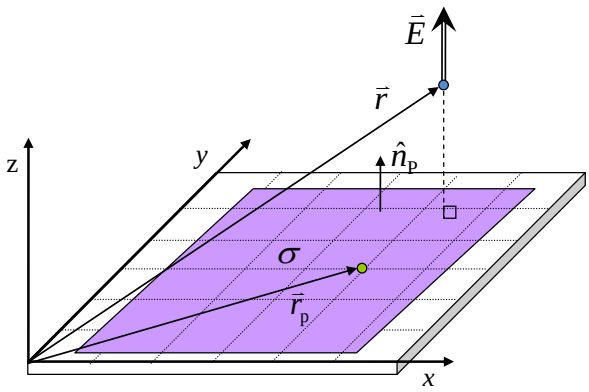
```
public static SVector3d infinitePlateElectricField(SVector3d r_P, SVector3d n_P, double sigma, SVector3d r)
throws SImpossibleNormalizationException
```

L'objectif de cette méthode sera de calculer le champ électrique générée par une plaque infinie uniformément chargée (PPIUC) :

$$\vec{E} = s\frac{\sigma}{2\varepsilon_0}\hat{n}_P$$

avec

$$s = \begin{cases} 1 & \text{si} \quad \hat{n}_P \cdot \left(\vec{r} - \vec{r}_P\right) > 0 \\ -1 & \text{si} \quad \hat{n}_P \cdot \left(\vec{r} - \vec{r}_P\right) < 0 \\ 0 & \text{si} \quad \hat{n}_P \cdot \left(\vec{r} - \vec{r}_P\right) = 0 \end{cases}$$



Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction **throw new** SNoImplementationException( ); et complétez l'implémentation en exploitant la position de la plaque r_P, la normale à la surface de la plaque n_P, la densité de charge surfacique sigma ainsi que la coordonnée r où l'on désire évaluer le champ électrique. N'oubliez pas d'utiliser la constante EPSILON_ZERO déjà définie dans la classe **SElectrostatics** pour évaluer votre constante constante électrique ε_0.

Avant de déterminer le signe du paramètre s de l'équation, vous devez vérifier si r est sur la plaque. Dans ce contexte, le paramètre s sera une valeur très près de « zéro ». Utilisez la méthode

public static boolean nearlyZero(**double** value)

de la classe **SMath** du ***package*** **sim.math**. Si s est près de zéro, alors vous devrez retourner le vecteur zéro avec l'instruction **return** NO_ELECTRIC_FIELD; signifiant que r est sur la plaque et que le vecteur champ électrique est nul.

Par la suite, pour déterminer le signe du paramètre calculé s, vous pouvez utiliser la méthode Math.signum(...) avec votre variable s. Cette méthode retournera :

- 1.0 si s est positif
- -1.0 si s est négatif
- 0.0 si s est nul (scénario non plausible, car scénario préalablement vérifié !)

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

Ouvrez le fichier *field_configuration.cfg* dans le répertoire de votre projet avec un éditeur de texte et modifiez le fichier de scène pour *Chap1.9-S2.txt* (*read_data Chap1.9-S2.txt*). Cette scène contient deux PPIUC tel que vous l'avez étudié dans la **Situation 2 :** ***Le principe de superposition appliqué aux PPIUC*** de la section 1.9 de vos notes de cours.

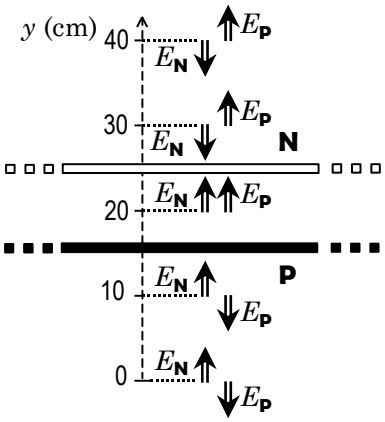


Schéma de la situation 2 de la section 1.9.

Afin de constater la fonctionnalité de cette méthode, lancez l'exécution de la classe **SIMElectricViewer.java** disponible dans le ***package*** **sim.application**.et appuyez sur le bouton « Chargement ». Visualisez l'ensemble du champ électrique afin de confirmer qu'il est conforme avec le champ généré par deux PPIUC et vérifiez que le champ à la coordonnée $\vec{r} = (0, 0.2, 0)\,\mathrm{m}$ est bel et bien égal à $\vec{E} = 282\,\vec{j}$ N/C et que les champ électrique est nul au-dessus et sous les deux plaques.

Pour répondre à la prochaine question conceptuelle, vous devrez ouvrir le fichier de scène *plaques_et_sphere.txt*.

Répondez à la question suivante dans le cahier de réponse :

Question 4.2 :

À partir du fichier de scène *plaques_et_sphere.txt* contenant deux plaques uniformément chargées ainsi qu'une sphère uniformément chargée, vous allez analyser le champ électrique dans le plan *xy* afin de répondre aux deux questions suivantes :

a) Identifiez approximativement une coordonnée *xyz* où le champ électrique est nul ($\vec{E} = 0$).
b) Dessinez qualitativement la forme du champ électrique (en champ de vecteur) autour de la position où le champ électrique est nul.

4.3 La tige rectiligne infinie uniformément chargée (TRIUC)

Fichier à modifier : **SElectrostatics.java**

Fichier à exécuter : **SIMElectricFieldViewer.java**

Fichier de scène : *Chap1.7-SA.txt*

Prérequis : 2.2

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

```
public static SVector3d infiniteRodElectricField(SVector3d r_R, SVector3d axis, double lambda, SVector3d r)
throws SImpossibleNormalizationException
```

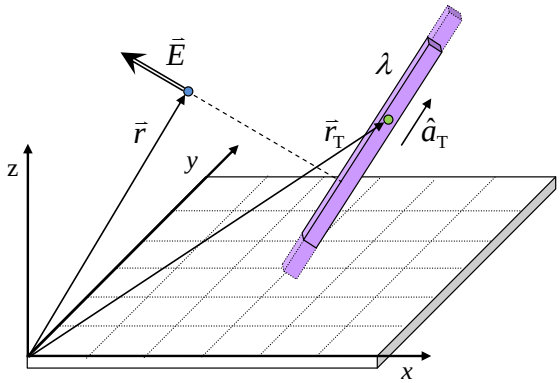
L'objectif de cette méthode sera de calculer le champ électrique générée par une tige rectiligne infinie uniformément chargée (TRIUC) :

$$\vec{E} = 2k\lambda \frac{\vec{R}_{\mathrm{T}}}{\left\|\vec{R}_{\mathrm{T}}\right\|^2}$$

avec $\vec{R}_{\mathrm{T}} = \hat{a}_{\mathrm{T}} \times \left(\vec{r} - \vec{r}_{\mathrm{T}}\right) \times \hat{a}_{\mathrm{T}}$

et

$\vec{E} = 0$ si $\vec{R}_{\mathrm{T}} = 0$



Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction `throw new SNoImplementationException( );` et complétez l'implémentation en exploitant la position de la tige r_R, l'axe de la tige axis, la densité de charge linéaire lambda ainsi que la coordonnée r où l'on désire évaluer le champ électrique.

Remarquez que si r est sur l'axe de la tige, alors $R_{\mathrm{T}} = \left\|\vec{R}_{\mathrm{T}}\right\| = 0$. Puisque cette valeur sera calculée, elle sera difficile à comparer à « zéro » (problème de précision avec le type ***double***). Pour affronter ce problème, utilisez la méthode

```
public static boolean nearlyZero(double value)
```

disponible dans la classe **SMath** du ***package*** **sim.math** pour réaliser votre comparaison. Si cette méthode s'avère être vraie, vous devrez retourner le vecteur zéro (`return NO_ELECTRIC_FIELD;`), car la coordonnée r sera située sur la tige signifiant que le champ électrique est nul.

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

Ouvrez le fichier *field_configuration.cfg* dans le répertoire de votre projet avec un éditeur de texte et modifiez le fichier de scène pour *Chap1.7-SA.txt* (*read_data Chap1.7-SA.txt*). Cette scène contient une TRIUC tel que vous l'avez étudié dans la **Situation A : *La tige chargée vue de haut*** de la section 1.7 de vos notes de cours.

Afin de constater la fonctionnalité de cette méthode, lancez l'exécution de la classe **SIMElectricViewer.java** disponible dans le ***package*** **sim.application** et appuyez sur le bouton « Chargement ». Visualisez l'ensemble du champ électrique afin de confirmer qu'il est conforme avec le champ généré par la TRIUC et vérifiez que le champ à la coordonnée $\vec{r} = (4,3,0)\,\mathrm{m}$ est bel et bien égal à $\vec{E} = (2{,}16\,\vec{i} + 0{,}72\,\vec{j}) \times 10^4\,\mathrm{N/C}$.

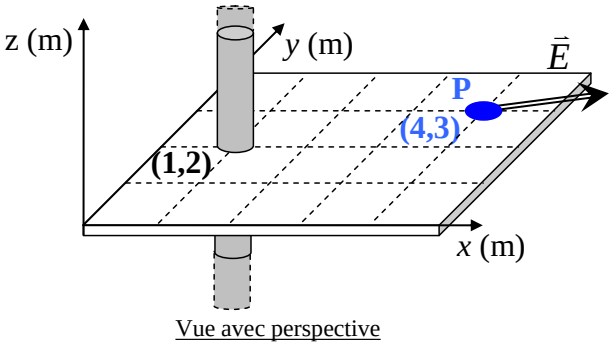


Schéma de la situation A de la section 1.7.

Répondez à la question suivante dans le cahier de réponse :

Question 4.3 :

Selon la théorie présentée en classe, l'équation de la TRIUC $E = 2k\lambda / R$ propose que le module du champ électrique soit inversement proportionnel à la distance du point P avec la tige. Étant donné que l'équation à programmer est divisée par $R_{\mathrm{T}}^{\,2}$, expliquez pourquoi elle reste valable avec l'équation « classique » de la TRIUC.

4.4 La tige rectiligne uniformément chargée : hors axe (TRUC)

Fichier à modifier : **SElectrostatics.java**

Prérequis : 2.1 et 2.2

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

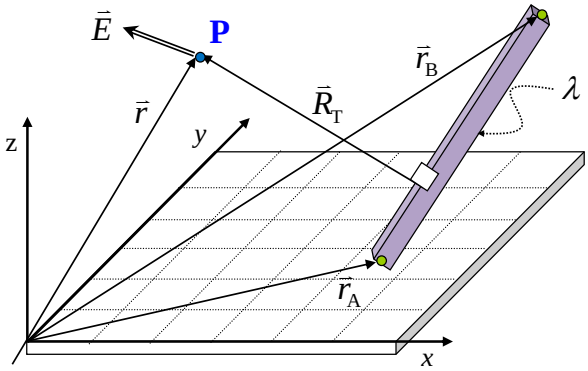
```
public static SVector3d finiteRodElectricFieldOutsideAxis(SVector3d r_A, SVector3d r_B, double lambda, SVector3d r)
    throws SImpossibleNormalizationException, SUndefinedFieldException
```

L'objectif de cette méthode sera de calculer le champ électrique générée par une tige rectiligne uniformément chargée (TRUC) hors axe :

$$\vec{E} = E\sin\left(\frac{\alpha_A + \alpha_B}{2}\right)\hat{a}_T + E\cos\left(\frac{\alpha_A + \alpha_B}{2}\right)\hat{R}_T$$

où

$$E = \sqrt{2}\,\frac{k\lambda}{R_T}\sqrt{1-\cos(\alpha_A - \alpha_B)}$$



avec les termes suivants :

$$\hat{a}_T = \frac{\vec{r}_B - \vec{r}_A}{\|\vec{r}_B - \vec{r}_A\|}$$

$$\vec{r}_{pA} = \vec{r}_A - \vec{r}$$

(point **P** vers **A**)

$$\vec{r}_{pB} = \vec{r}_B - \vec{r}$$

(point **P** vers **B**)

$$\hat{r}_{pA} = \frac{\vec{r}_{pA}}{\|\vec{r}_{pA}\|} \qquad \hat{r}_{pB} = \frac{\vec{r}_{pB}}{\|\vec{r}_{pB}\|}$$

$$\vec{n} = \vec{r}_{pA} \times \hat{a}_T$$

Attention au symbole de flèche !!!

$$\hat{n} = \frac{\vec{n}}{\|\vec{n}\|} \qquad \vec{R}_T = \vec{n} \times \hat{a}_T \qquad R_T = \|\vec{R}_T\| \qquad \hat{R}_T = \frac{\vec{R}_T}{\|\vec{R}_T\|}$$

$$\alpha_A = \arcsin\left(\hat{R}_T \times \hat{r}_{pA} \cdot \hat{n}\right)$$

(angle positif ou négatif)

$$\alpha_B = \arcsin\left(\hat{R}_T \times \hat{r}_{pB} \cdot \hat{n}\right)$$

(angle positif ou négatif)

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction `throw new SNoImplementationException( );` et complétez l'implémentation en exploitant les positions r_A et r_B désignant les extrémités de la tige, la densité de charge linéaire lambda ainsi que la coordonnée r où l'on désire évaluer le champ électrique.

Remarquez que si r est sur l'axe de la tige, alors le module $R_T = \|\vec{R}_T\| = 0$. Si ce scénario se produit, vous devrez retourner une exception de type **SUndefinedFieldException** à l'aide de l'instruction

```
throw new SUndefinedFieldException();
```

signifiant que cette méthode ne peut pas calculer le champ électrique à la coordonnée r demandée. Assurez-vous d'utiliser la bonne instruction pour réaliser votre comparaison avec « zéro ».

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

Répondez à la question suivante dans le cahier de réponse :

Question 4.4 :

Dans l'implémentation de l'équation de la TRIUC, nous avons imposé que le champ électrique était nul ($\vec{E}=0$) lorsque $R_T=0$. Pourquoi dans l'implémentation actuelle du champ électrique de la TRUC *hors axe*, nous avons décidé de retourner une exception lorsque $R_T=0$? Justifiez votre réponses à l'aide d'un argument physique.

4.5 La tige rectiligne uniformément chargée (TRUC)

Fichier à modifier : **SElectrostatics.java**

Fichier à exécuter : **SIMElectricFieldViewer.java**

Fichier de scène : *Chap1.8a-SA.txt* et *Chap1.8b-SA.txt*

Prérequis : 2.1, 2.2, 4.1 et 4.4

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

```
public static SVector3d finiteRodElectricField(SVector3d r_A, SVector3d r_B, double Q, SVector3d r)
```

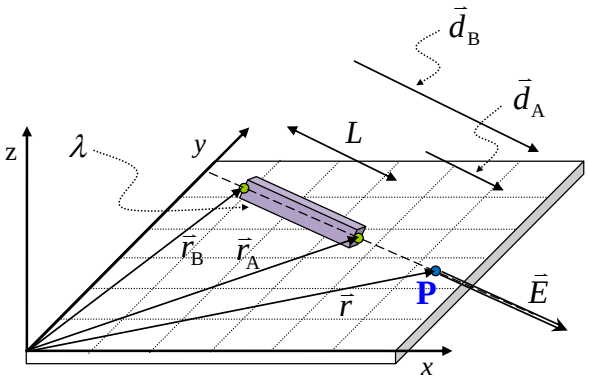
L'objectif de cette méthode sera de calculer le champ électrique générée par une tige rectiligne uniformément chargée (TRUC) le long de son axe et d'intégrer les équations de 4.1 et 4.4 sous certaines conditions :

$$\vec{E}=\frac{kQ}{\|\vec{d}_A\|\,\|\vec{d}_B\|}\hat{d}$$

(équation de la TRUC sur axe)

avec

$$\vec{d}_A=\vec{r}-\vec{r}_A \quad , \quad \vec{d}_B=\vec{r}-\vec{r}_B \quad \text{et} \quad \hat{d}=\frac{\vec{d}_A}{\|\vec{d}_A\|}=\frac{\vec{d}_B}{\|\vec{d}_B\|}$$



selon les conditions suivantes :

Le point **P** est sur $\vec{r}_A$ ou $\vec{r}_B$	La tige possède une longueur nulle. ($L=0$)	Le point **P** est sur l'axe de la tige.	Le point **P** est situé sur la tige.	Le point **P** est hors axe de la tige.
$\|\vec{d}_A\|=0$ ou $\|\vec{d}_B\|=0$	$L=\|\vec{r}_A-\vec{r}_B\|=0$	$\frac{\vec{d}_A\cdot\vec{d}_B}{\|\vec{d}_A\|\,\|\vec{d}_B\|}=1$	$\frac{\vec{d}_A\cdot\vec{d}_B}{\|\vec{d}_A\|\,\|\vec{d}_B\|}=-1$	$\frac{\vec{d}_A\cdot\vec{d}_B}{\|\vec{d}_A\|\,\|\vec{d}_B\|}\neq 1$
$\vec{E}=0$	Champ d'une sphère de rayon *R* = 0.0 (voir 4.1)	$\vec{E}=\frac{kQ}{\|\vec{d}_A\|\,\|\vec{d}_B\|}\hat{d}$	$\vec{E}=0$	Champ d'une tige hors axe (voir 4.4)

Présentement, cette méthode retourne une exception de type **SNoImplementationException** spécifiant que la méthode n'a pas été implémentée. Effacez l'instruction **throw new** SNoImplementationException( ); et complétez l'implémentation en exploitant les positions r_A et r_B désignant les extrémités de la tige, la charge Q de la tige ainsi que la coordonnée r où l'on désire évaluer le champ électrique.

Remarquez qu'il y aura 5 scénarios de calcul de champ électrique à gérer qui nécessiteront la position r où l'on désire évaluer le champ électrique ainsi que les positions r_A et r_B des extrémités de la tige. Assurez-vous de retourner la bonne expression du champ électrique !

1) Pour évaluer un champ électrique nulle ($\vec{E} = 0$), effectuez tout simplement l'instruction

```
return NO_ELECTRIC_FIELD;
```

2) Pour évaluer le champ électrique dans le cas où la tige serait une charge ponctuelle, exploitez la méthode

```
public static SVector3d sphereElectricField(SVector3d r_S, double R, double Q, SVector3d r)
```

que vous avez programmé à l'étape 3.1. Assurez-vous de mettre la bonne valeur dans le paramètre R !

3) Pour évaluer le champ électrique dans le cas d'une tige hors axe, exploitez la méthode

```
public static SVector3d finiteRodElectricFieldOutsideAxis(SVector3d r_A, SVector3d r_B, double lambda, SVector3d r)
throws SImpossibleNormalizationException, SUndefinedFieldException
```

que vous avez programmé à l'étape 4.4. Assurez-vous de mettre la bonne valeur dans le paramètre lambda !

4) Pour évaluer le champ électrique dans le cas d'une tige sur l'axe, implémentez l'équation

$$\vec{E} = \frac{k\,Q}{\left\|\vec{d}_{\mathrm{A}}\right\| \left\|\vec{d}_{\mathrm{B}}\right\|}\hat{d}$$

adéquatement.

Pour réaliser vos comparaisons avec les valeurs numériques 0.0, 1.0 et -1.0 obtenues par les différentes équations, vous devrez utiliser les méthodes

```
public static boolean nearlyZero(double value)
public static boolean nearlyEquals(double a, double b)
```

disponible dans la classe **SMath** du ***package*** **sim.math** pour réaliser vos comparaisons.

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

Ouvrez le fichier *field_configuration.cfg* dans le répertoire de votre projet avec un éditeur de texte et modifiez le fichier de scène pour *Chap1.8a-SA.txt* (*read_data Chap1.8a-SA.txt*). Cette scène contient une TRUC tel que vous l'avez étudié dans la **Situation A : *Attirer avec une tige chargée*** de la section 1.8a de vos notes de cours.

Afin de constater la fonctionnalité de cette méthode, lancez l'exécution de la classe **SIMElectricViewer.java** disponible dans le ***package*** **sim.application** et appuyez sur le bouton « Chargement ». Visualisez l'ensemble du champ électrique afin de confirmer qu'il est conforme avec le champ généré par la TRUC (axe) et vérifiez que le champ à la coordonnée $\vec{r} = (0,0,0)\,\mathrm{m}$ est bel et bien égal à $\vec{E} = 4{,}5\times10^{6}\,\vec{j}$ N/C .

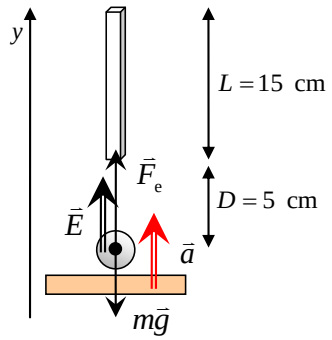


Schéma de la situation A de la section 1.8a.

À l'aide du bouton de recherche de fichier « browser », sélectionnez le fichier *Chap1.8b-SA.txt* situé dans le répertoire « field_viewer ». Validez ette scène avec le bouton « *chargement* ». Cette scène contient une TRUC tel que vous l'avez étudié dans la **Situation A : *La tige finie sur l'axe y.*** de la section 1.8b de vos notes de cours.

Visualisez l'ensemble du champ électrique afin de confirmer qu'il est conforme avec le champ généré par la TRUC (hors axe) et vérifiez que le champ à la coordonnée $\vec{r} = (1,1,0)\,\mathrm{m}$ est bel et bien égal à $\vec{E} = \left(-5{,}730\ \vec{i} - 2{,}215\ \vec{j}\right)\times 10^4\,\mathrm{N/C}$.

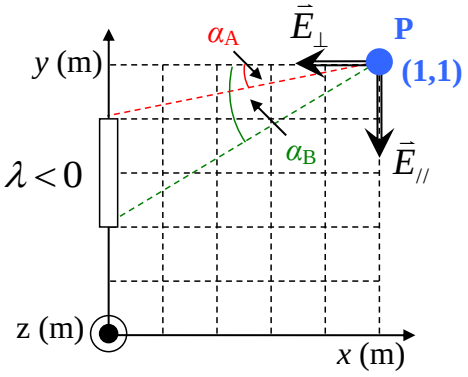


Schéma de la situation A de la section 1.8b.

Répondez à la question suivante dans le cahier de réponse :

Question 4.5 :

Pour programmer le champ électrique généré par une TRUC, nous utilisons la charge de la tige *Q* et non pas la densité de charge *λ* (comme dans l'équation implémentée en 4.**4**).

En vous basant sur <u>un cas particulier</u> de l'équation du champ généré par une TRUC le long de l'axe *x*, expliquez pourquoi ce choix a été judicieux. Dans votre réponse, vous devez expliquer un problème qui aurait été engendré si l'équation avait été implémentée avec un densité de charge *λ*.

Conclusion

Félicitations ! Vous avez complété l'implémentation de votre champ électrique et vous êtes apte à illustrer différentes combinaisons de distribution de charge en trois dimensions en exploitant la librairie graphique de ***OpenGL***.

Remise du programme

Pour effectuer la remise électronique de votre programme, envoyer le fichier ci-dessous dans <u>l'espace de dépôt</u> exigé par votre enseignant (exemple : OMNIVOX/LÉA) :

- Le **répertoire SIM** de votre projet dans un **format compressé** « zip » sous le nom *SIM-ChampElectric.zip* comme vous l'avez initialement téléchargé.