MINISTÈRE
DE L'ÉDUCATION
NATIONALE

EAE SIN 2

SESSION 2019

AGREGATION
CONCOURS EXTERNE

Section : SCIENCES INDUSTRIELLES DE L'INGÉNIEUR

Option : SCIENCES INDUSTRIELLES DE L'INGÉNIEUR ET INGÉNIERIE INFORMATIQUE

MODÉLISATION D'UN SYSTÈME, D'UN PROCÉDÉ OU D'UNE ORGANISATION

Durée : 6 heures

Calculatrice électronique de poche - y compris calculatrice programmable, alphanumérique ou à écran graphique – à fonctionnement autonome, non imprimante, autorisée conformément à la circulaire n° 99-186 du 16 novembre 1999.

L'usage de tout ouvrage de référence, de tout dictionnaire et de tout autre matériel électronique est rigoureusement interdit.

Si vous repérez ce qui vous semble être une erreur d'énoncé, vous devez le signaler très lisiblement sur votre copie, en proposer la correction et poursuivre l'épreuve en conséquence. De même, si cela vous conduit à formuler une ou plusieurs hypothèses, vous devez la (ou les) mentionner explicitement.

NB : Conformément au principe d'anonymat, votre copie ne doit comporter aucun signe distinctif, tel que nom, signature, origine, etc. Si le travail qui vous est demandé consiste notamment en la rédaction d'un projet ou d'une note, vous devrez impérativement vous abstenir de la signer ou de l'identifier.

Tournez la page S.V.P.

A

Ce document est composé :

- d'un dossier sujet de 24 pages ;
- d'un dossier technique (DT) de 17 pages ;
- d'un dossier réponse (DR) de 5 pages.

Conseils aux candidats :

Les différentes parties du sujet sont indépendantes.

Un parcours attentif de l'ensemble du document est conseillé avant de composer.

La présentation des programmes doit respecter les mots clés du langage cible ainsi que l'indentation des structures algorithmiques.

Les réponses doivent être présentées avec clarté, rigueur et concision.

INFORMATION AUX CANDIDATS

Vous trouverez ci-après les codes nécessaires vous permettant de compléter les rubriques figurant en en-tête de votre copie.

Ces codes doivent être reportés sur chacune des copies que vous remettrez.

Concours	Section/option	Epreuve	Matière
EAE	1417A	102	2680

Simulateur 3D de camion-citerne

Présentation

Les sociétés TRYDEA (Nancy) et Sym2B (Grenoble) conçoivent, développent et fournissent des simulateurs de conduite de véhicules spécifiques, notamment des simulateurs de bus urbains et des simulateurs de camion-citerne.

Le simulateur de conduite du camion-citerne a été conçu afin de permettre à des utilisateurs de s'entrainer sur un système virtuel avant d'achever la formation sur le véhicule réel. Cela permet de former les conducteurs à des situations qui seraient trop difficiles, trop dangereuses, ou trop coûteuses à reconstituer dans la réalité.

La perception physique de l'utilisateur avec le simulateur est assurée par le matériel suivant :

- un volant à retour d'effort ;
- une boîte de vitesse ;
- des pédales d'accélérateur, d'embrayage et de frein ;
- un levier de commande du ralentisseur ;
- un mouvement à 6 degrés de liberté du siège conducteur (hexapode).

Afin d'assurer la perception visuelle 3D, trois écrans assurent l'immersion de l'utilisateur dans une scène qui représente respectivement :

- la vue du pare-brise avant (figure 1) ;
- la vue de la fenêtre gauche ;
- la vue de la fenêtre droite.

Tournez la page S.V.P.

Le monde extérieur, rendu en images de synthèse, s'affiche sur trois écrans de 55 et 75 pouces de diagonale. Ils sont disposés en triptyque, les 2 écrans latéraux formant un angle de 65° avec l'écran central. Les écrans choisis disposent d'une résolution 4K UHD, c'est-à-dire 3840 x 2160.

Figure 1 : vue utilisateur depuis le pare-brise avant

Pour calculer les images sur les trois écrans (figure 2), un ordinateur est dédié à la génération du flux vidéo de chaque écran. Les trois ordinateurs intègrent tous une carte graphique GeForce GTX 1080 pour générer des images en 4K. Une fréquence de génération de 60 trames par seconde est nécessaire pour procurer une sensation de fluidité dans le mouvement et une lisibilité optimale de la signalisation.

Un quatrième ordinateur est embarqué dans le véhicule. Il centralise les données de tous les périphériques du poste de pilotage et synchronise l'ensemble des calculs avec les autres ordinateurs connectés à un réseau Ethernet.

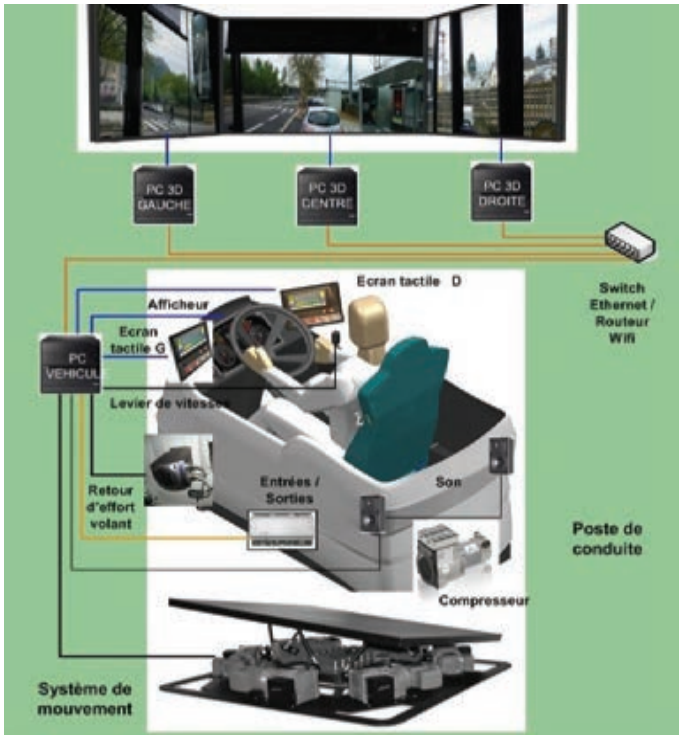


Figure 2 : synoptique du système

Le diagramme des cas d'utilisation suivant (figure 3) présente l'ensemble des fonctionnalités réalisées par le simulateur et les parties grisées précisent celles qui sont étudiées dans ce sujet.

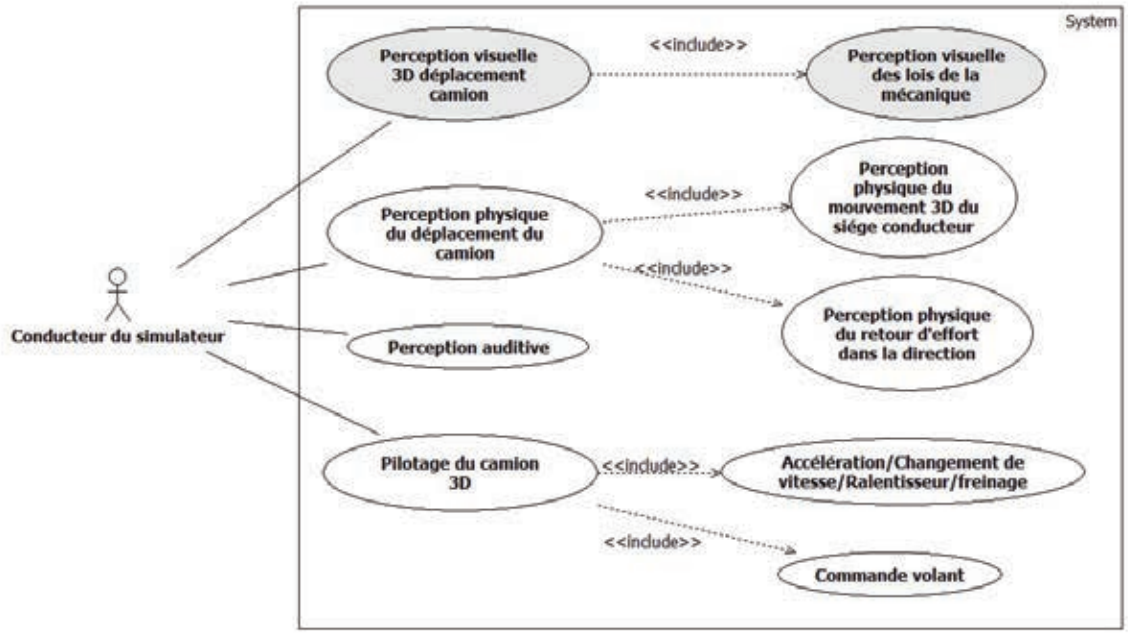


Figure 3 : diagramme des cas d'utilisation

Le diagramme de déploiement (figure 4) présente une vue logique du logiciel. La partie relative à la simulation physique du simulateur s'appuie sur le moteur physique `PhysX`, aujourd'hui propriété de NVIDIA. Ce moteur est une bibliothèque logicielle indépendante appliquée à la résolution de problème de la mécanique classique. Elle propose notamment des modèles dynamiques paramétrables pour les véhicules standard.

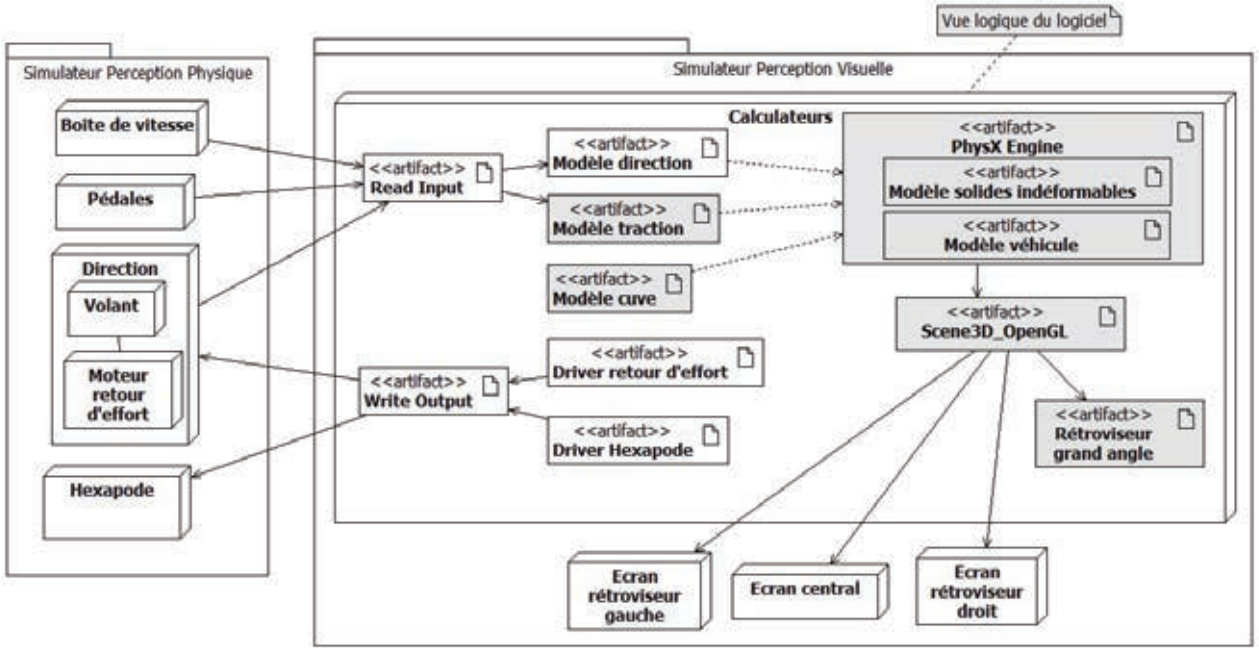


Figure 4 : vue logique du logiciel

Dans le simulateur de camion-citerne, le système utilise des fonctionnalités standard de `PhysX` pour gérer le châssis, les roues, la transmission du couple avec le terrain, les suspensions et le modèle de direction.

Tournez la page S.V.P.

D’autres parties spécifiques comme le modèle de traction, le mouvement de liquide au sein de la citerne et le rétroviseur grand angle, font l’objet de développements spécifiques. Pour chacune de ces parties, une modélisation doit être proposée, validée de manière indépendante avant intégration dans le simulateur.

Ce sujet contient 3 parties :

Partie n°1 : modélisation du liquide de la citerne ;
Partie n°2 : modèles de programmation basés sur la bibliothèque `PhysX` ;
Partie n°3 : modélisation du rendu 3D dans le rétroviseur grand angle.

Partie 1. Modélisation du liquide dans la citerne

L’objectif de cette partie est de déterminer un modèle numérique du comportement du liquide dans la citerne afin d’intégrer ce dernier dans l’application finale.

Le modèle mécanique retenu sépare les mouvements du liquide de surface, sensible aux accélérations latérales et mouvements brefs, des déplacements du reste du liquide sensible à l’orientation de la citerne.

La masse totale du liquide est alors composée de deux masses (figure 5) :
- la première m_1 de centre de masse G_1 associée au liquide de surface ;
- la deuxième m_2 de centre de masse G_2 associée au liquide situé en dessous.

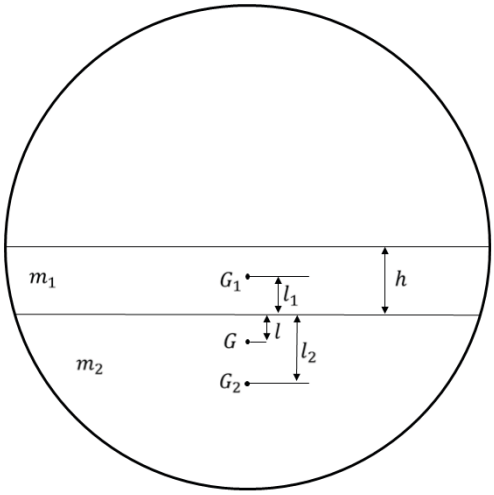


Figure 5 : coupe transversale de la citerne

Ce modèle peut être assimilé à un système pendulaire (figure 6) dont le point d’accroche du pendule se situe sur une masse prise entre 2 ressorts.

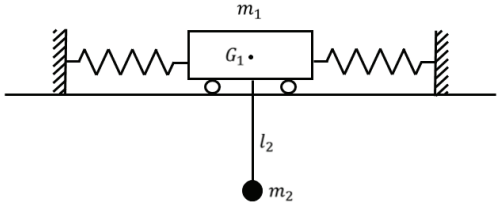


Figure 6 : modèle masse-ressorts-pendule

L'objectif est de définir la dynamique du modèle connaissant les forces auxquelles le système est soumis.

Ainsi, lorsque la citerne est soumise à une accélération $\vec{\gamma}$, le principe fondamental de la dynamique appliqué dans le référentiel non galiléen de la citerne inclut deux forces d'inertie d'entraînement $\vec{f_1} = -m_1\vec{\gamma}$ et $\vec{f_2} = -m_2\vec{\gamma}$.

Le modèle excité est représenté ci-dessous :

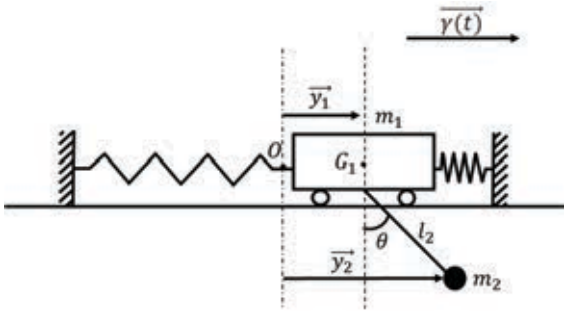


Figure 7 : modèle soumis à une accélération d'entraînement $\overrightarrow{\gamma(t)}$

Les paramètres du modèle sont synthétisés dans le tableau ci-dessous :

Masse m_1 (kg)	1 500
Masse m_2 (kg)	10 250
Longueur l (m)	0,33
Longueur l_1 (m)	0,05
Longueur l_2 (m)	0,386
Raideur des ressorts k (N/m)	289 393
Coefficients de frottement β_1 et β_2	20 000
Accélération de la pesanteur g (m/s²)	9,81

1.1 Modèle simplifié et méthode d'Euler

Dans le cadre de cette modélisation, la partie m_1 va d'abord être considérée fixe. Le système s'apparente à un pendule simple non excité et non amorti.

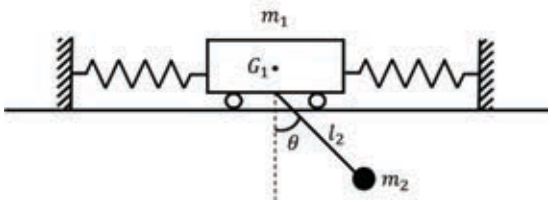


Figure 8 : cas où la masse m_1 est fixe

Le système est régi par l'équation différentielle suivante :

$$\frac{d^2\theta}{dt^2} + \frac{g}{l_2}\sin\theta = 0 \qquad (1)$$

associée aux conditions initiales :

$$\frac{d\theta}{dt}(t=0) = 0\,,\ \theta(0) = \theta_0$$

Tournez la page S.V.P.

Q 1. En posant $\varphi = \frac{d\theta}{dt}$, écrire l'équation précédente sous la forme d'un système de deux équations différentielles du premier ordre comme ci-dessous :

$$\begin{cases} \frac{d\theta}{dt} = F(\varphi) \\ \frac{d\varphi}{dt} = Q(\theta) \end{cases} \quad (2)$$

Q 2. En utilisant la méthode `odeint` du sous-package `scipy.integrate` (document technique DT1), compléter le programme Python (document réponse DR1) permettant de calculer $\theta(t)$ sur l'intervalle $[0, t_{max}]$ avec N points. L'algorithme sera testé avec un état initial : $\theta_0 = 0{,}1$ rad et $\frac{d\theta}{dt}(t=0) = 0\ \text{rad.s}^{-1}$.

Afin d'analyser convenablement le problème, la méthode d'Euler explicite est utilisée comme première approche. Cette méthode consiste à déterminer une solution approchée d'une équation différentielle du type $\frac{dy}{dt}(t) = F(t, y(t))$, sous forme discrétisée $y_i = y(t_i)$.

Les variables de la discrétisation du temps sont définies ci-dessous :

$t_i = i \times h$	le temps à l'itération i
N	le nombre de points de calcul
t_{max}	la durée de la simulation
$h = \frac{t_{max}}{N-1}$	le pas de temps de calcul

Q 3. Rappeler le schéma numérique de la méthode d'Euler qui associe aux grandeurs définies par l'équation différentielle $\frac{dy}{dt}(t) = F(t, y(t))$, les grandeurs discrètes définies par l'équation de récurrence $y_{i+1} = G(t_i, y_i, h)$.

Q 4. En appliquant la méthode d'Euler au système de la figure 8, écrire les équations de récurrence associées respectivement à θ_{i+1} et φ_{i+1}.

Q 5. Compléter le programme Python (DR1) en utilisant la méthode d'Euler.

Les figures suivantes montrent les résultats de simulation concernant les deux méthodes numériques (**`odeint`** et Euler) pour deux valeurs du nombre d'itérations N :

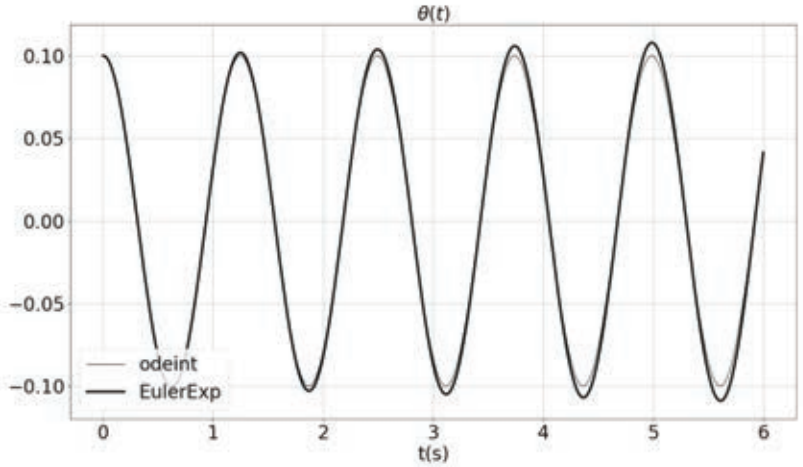


Figure 9 : simulations pour $N = 5000$

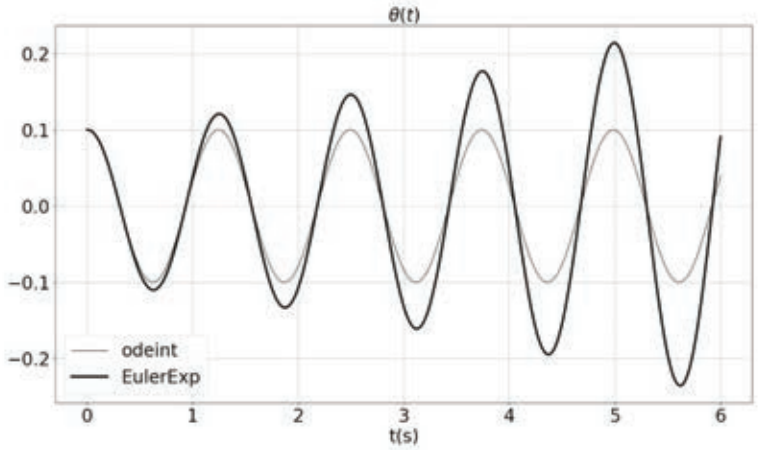


Figure 10 : simulations pour $N = 500$

Q 6. Comparer les résultats de simulation des deux méthodes numériques. Conclure sur l'inconvénient majeur de la méthode d'Euler. Citer des méthodes numériques plus efficaces.

1.2 Modèle complet et autres méthodes numériques

Conformément à la figure 7, le point d'accroche du pendule est fixé sur une masse m_1 en mouvement. La variable y_1 représente la distance projetée sur l'axe horizontal entre le centre de gravité de la masse m_1 et l'origine 0 du repère. La variable y_2 représente la distance projetée sur l'axe horizontal entre la masse ponctuelle m_2 et l'origine 0 du repère.

L'équation du mouvement peut se mettre sous la forme :

$$\begin{cases} (m_1+m_2)\dfrac{d^2y_1}{dt^2} + m_2 l_2 \dfrac{d^2\theta}{dt^2}\cos\theta - m_2 l_2 \left(\dfrac{d\theta}{dt}\right)^2 \sin\theta + k y_1 + \beta_1 \dfrac{dy_1}{dt} = f_1(t) \\ m_2 l_2 \dfrac{d^2\theta}{dt^2} + m_2 \dfrac{d^2y_1}{dt^2}\cos\theta + m_2 g \sin\theta + \beta_2 \dfrac{d\theta}{dt} = f_2(t) \end{cases} \qquad (3)$$

Les conditions initiales sont les suivantes :

$$y_1(0) = 0,\ \theta(0) = 0,\ \ \frac{dy_1}{dt}(t=0) = 0,\ \ \frac{d\theta}{dt}(t=0) = 0$$

Q 7. Linéariser le système précédent en considérant les angles petits devant 1 et les termes quadratiques négligeables.

La méthode des différences finies est choisie dans un premier temps. Ci-dessous sont rappelées les dérivées numériques première et seconde concernant une fonction y :

$$\frac{dy}{dt} \approx \frac{y_{i+1} - y_i}{h} \quad \text{et} \quad \frac{d^2y}{dt^2} \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}$$

Q 8. À partir des relations précédentes, écrire le système sous la forme ci-dessous et exprimer A, B, C, D, E, F *et* M en fonction des constantes du problème.

$$\begin{cases} A y_{1_{i+1}} + B\theta_{i+1} = C y_{1_{i-1}} + D\theta_{i-1} + E y_{1_i} + F\theta_i + M f_{1_i}(t) \\ G y_{1_{i+1}} + H\theta_{i+1} = I y_{1_{i-1}} + J\theta_{i-1} + K y_{1_i} + L\theta_i + N f_{2_i}(t) \end{cases}$$

Tournez la page S.V.P.

En posant X_i et F_i, respectivement le vecteur des inconnues et le vecteur des forces définis à l'itération i :

$$X_i = \begin{bmatrix} y_i \\ \theta_i \end{bmatrix} \text{ et } F_i = \begin{bmatrix} f_{1_i} \\ f_{2_i} \end{bmatrix}$$

Q 9. Écrire le système sous la forme matricielle comme ci-dessous en définissant chaque matrice en fonction des coefficients $A, B, C, D, E, F, G, H, I, J, K, L, M$ et N.

$$M_1 X_{i+1} = M_2 X_i + M_3 X_{i-1} + M_4 F_i$$

Q 10. En utilisant la fonction `numpy.dot(A,B)` qui réalise le produit matriciel $A \times B$, compléter le programme Python du document réponse DR1 permettant de finaliser l'implémentation de la relation précédente.

Q 11. Extraire du programme précédent (DR1) la relation permettant de calculer le déplacement de la masse m_2.

Afin de confirmer les résultats de la méthode des différences finies, il est souhaitable de comparer cette dernière avec le résultat retourné par la fonction `odeint`.

En considérant :

$$\omega = \frac{dy_1}{dt} \; ; \; \varphi = \frac{d\theta}{dt} \; ; \; X = \begin{bmatrix} y_1 \\ \theta \\ \omega \\ \varphi \end{bmatrix} \text{ et } \frac{dX}{dt} = \begin{bmatrix} \frac{dy_1}{dt} \\ \frac{d\theta}{dt} \\ \frac{d\omega}{dt} \\ \frac{d\varphi}{dt} \end{bmatrix}$$

Q 12. À partir du système différentiel (3), et ce toujours dans l'approximation linéaire ($\theta \ll 1$ et termes quadratiques négligeables), exprimer le nouveau système composé de 4 équations différentielles associées respectivement à y_1, θ, ω et φ.

Q 13. Exprimer le nouveau système comme ci-dessous en définissant chaque matrice.

$$\frac{dX}{dt} = M_5^{-1} \cdot (M_6 \cdot X + M_7)$$

Q 14. En associant à la matrice colonne $\frac{dX}{dt}$ une liste notée `dXdt` (voir DR1), exprimer le premier élément de `dXdt` par une instruction Python utilisant les méthodes `numpy.linalg.inv(A)` et `numpy.dot(A,B)`.

Q 15. Compléter le programme Python (DR1) permettant d'appliquer la méthode `odeint` au système précédent.

Ci-après (figure 11 et figure 12) sont présentés les résultats de la simulation utilisant la méthode des différences finies, la méthode d'Euler explicite, la méthode de Runge-Kutta d'ordre 4 (RK4) et la fonction `odeint` pour deux valeurs du nombre d'itérations N.

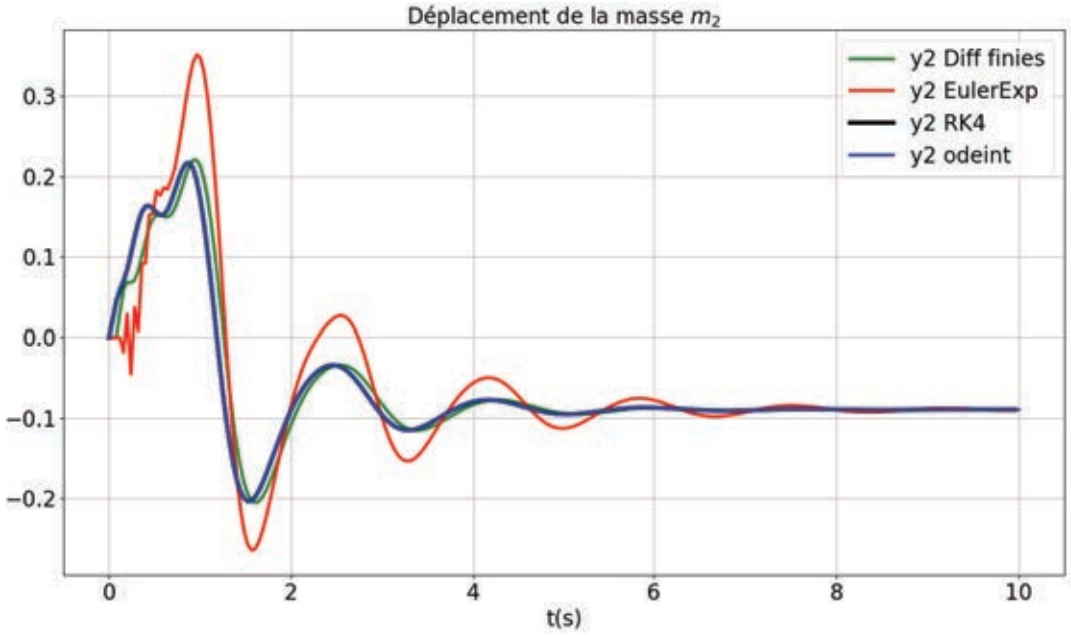


Figure 11 : simulations pour $N = 250$

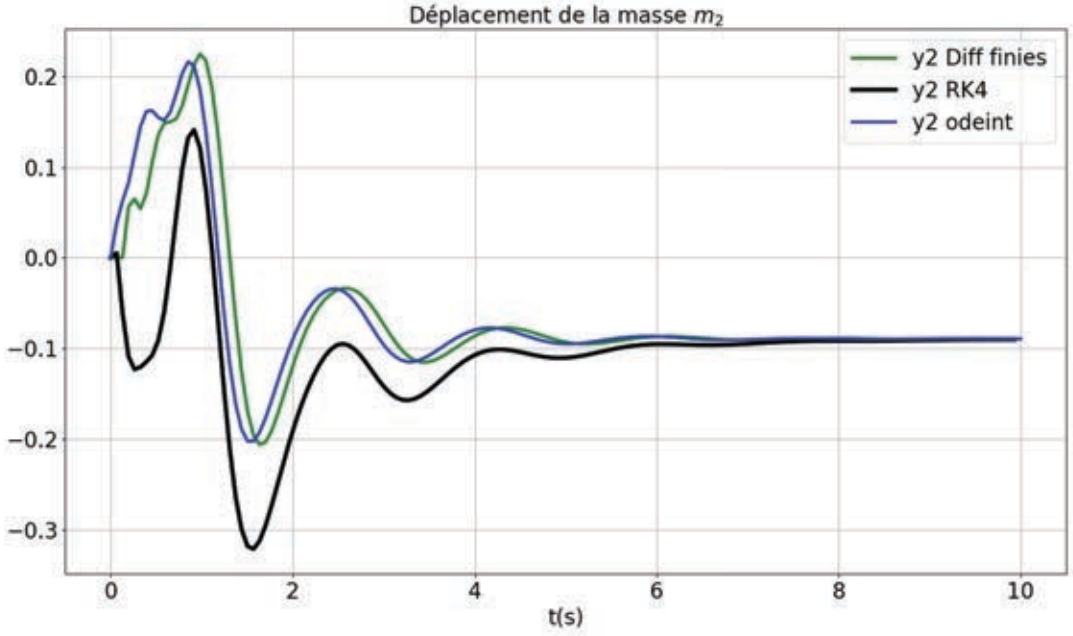


Figure 12 : simulations pour $N = 150$

Q 16. Le modèle final doit être intégré en C++. Critiquer chaque méthode en termes d'intégration logicielle et justifier le choix de la méthode numérique la mieux adaptée dans ce contexte.

Tournez la page S.V.P.

Partie 2. Modélisation logicielle

La programmation graphique 3D (3 dimensions) permet de modéliser les objets du monde réel, perçus en 3 dimensions, dans un ordinateur et ceci dans un espace virtuel également à 3 dimensions à l'aide de structures de données dédiées qui optimisent les transformations géométriques.

Afin de rendre la perception visuelle proche de la réalité, un moteur physique permet de calculer l'évolution dans le temps de la position des objets lorsque ces derniers sont soumis à des actions mécaniques.

Ce moteur utilise les lois de la physique (mécaniques du solide non déformable ou déformable, des fluides, etc..) et des modèles propres comme les modèles de collision entre objets.
L'affichage 2D final sur un écran d'ordinateur depuis l'espace virtuel 3D concerne les concepts de rendu 3D et 2D.

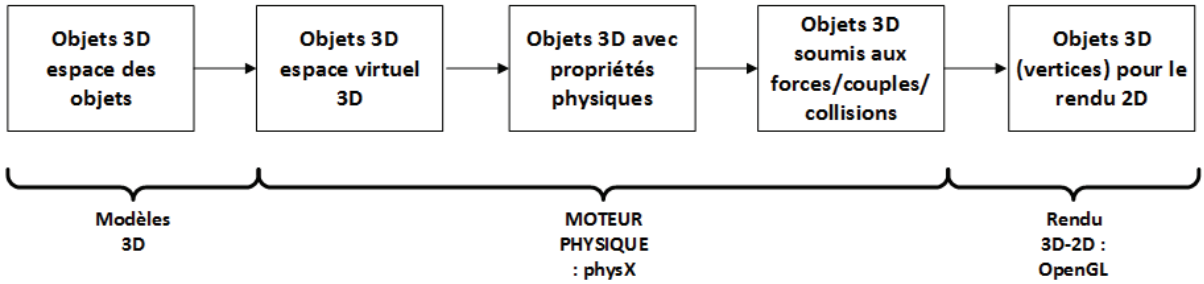


Figure 13 : principe d'un moteur 3D

Le simulateur de conduite du camion-citerne est une application écrite en C++ qui utilise la bibliothèque `OpenGL` pour le rendu 3D/2D et la bibliothèque `PhysX` de Nvidia pour la simulation physique du déplacement des objets et des collisions entre ces derniers.

Cette partie propose d'étudier les fonctionnalités de la bibliothèque `PhysX` intégrée dans le simulateur de conduite.

Plus précisément, la démarche propose d'analyser à l'aide de modèles physiques et logiciels l'architecture logicielle.

Les différentes étapes de cette étude sont listées ci-dessous :

- compréhension du modèle de programmation graphique 3D avec la bibliothèque `PhysX` et `OpenGL` ;
- modèle de programmation du véhicule de `PhysX`.

2.1 Modèle de programmation 3D avec `PhysX`

L'objectif de cette sous-partie est de comprendre le rôle des différentes classes de `PhysX` dédiées à la simulation 3D.

Concernant ses aspects graphiques, un simulateur 3D repose sur les principes cités ci-dessous (figure 14) :

- un simulateur 3D est un monde virtuel 3D visible dans un écran d'ordinateur ;
- ce monde virtuel est composé d'une ou plusieurs scènes ;
- une scène est composée d'acteurs (objets mobiles, immobiles, solides, mous, liquides, gazeux) possédant chacun des propriétés.

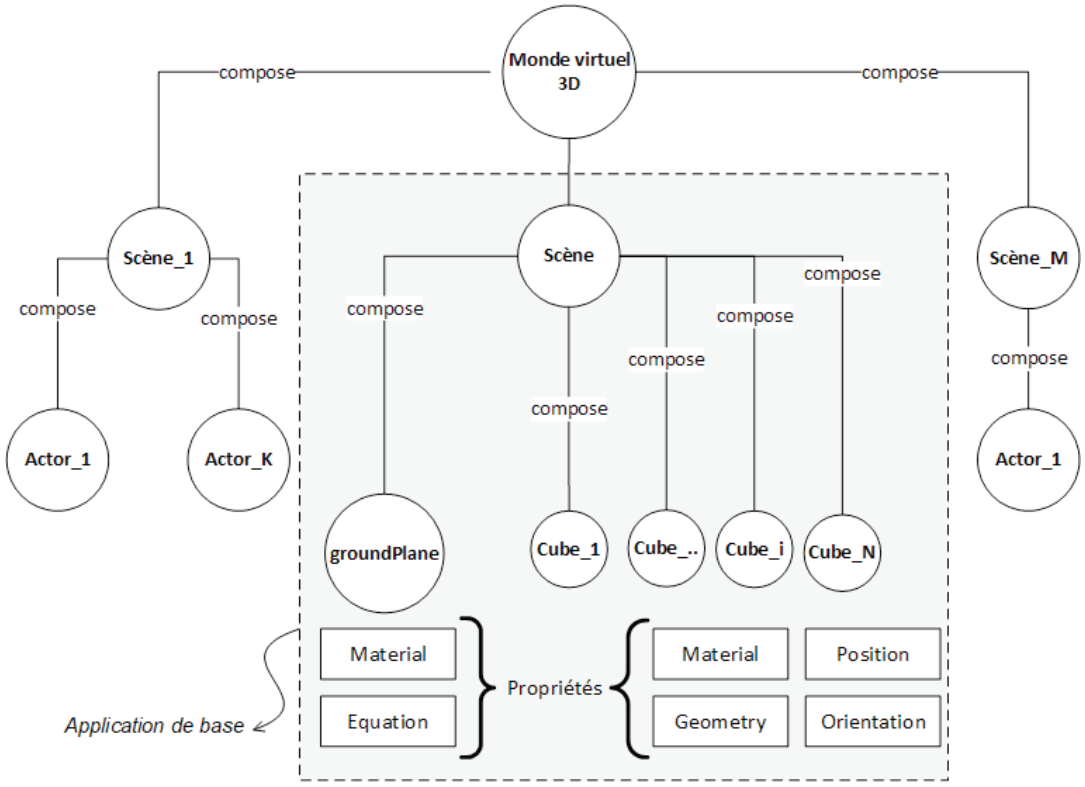


Figure 14 : monde virtuel de `PhysX`

La bibliothèque `PhysX` propose plusieurs types d'acteurs ayant chacun un modèle de structure et un modèle de comportement selon les fonctionnalités souhaitées :

- les corps rigides statiques (*Static Rigid Bodies*) ;
- les corps rigides dynamiques (*Dynamic Rigid Bodies*) ;
- les corps déformables (*Soft Bodies*) ;
- les particules (*Particles*).

Le diagramme de classes du document technique DT2 modélise une application de test simulant la chute d'un cube sur un plan (Ozx) dans une scène munie d'une accélération verticale de valeur $-g$.

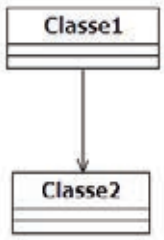
Le diagramme de séquence du document technique DT3 illustre l'interaction entre les différents objets permettant d'obtenir cette simulation. Le document technique DT4 présente les tests associés à cette application.

Tournez la page S.V.P.

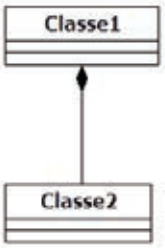
Dans ce diagramme de classes, les relations d'héritage, de composition et d'association unidirectionnelle sont utilisées.

Comme étude préalable, deux exemples de base parmi les relations citées précédemment sont représentés ci-dessous :

Relation n°1



Relation n°2



Avec l'implémentation C++ suivante :

Listing 1

```
//Implémentation Classe2
class Classe2
{
    public:
       Classe2();
};
Classe2::Classe2()
{}
//Implémentation Classe1
class Classe1
{
    private:
        Classe2 c2;
    public:
        Classe1();
};
Classe1::Classe1()
{}
```

```
#include"classe1.h"
int main()
{
    Classe1 *c1=new Classe1;
    return(0);
}
```

Listing 2

```
//Implémentation Classe2
class Classe2
{
    public:
        Classe2();
};
Classe2::Classe2()
{}
//Implémentation Classe1
class Classe1
{
    public:
        Classe2 *c2;
    public:
        Classe1();
        void setClasse2(Classe2 *c2X);
};
Classe1::Classe1()
{}
void Classe1::setClasse2(Classe2 *c2X)
{
    c2=c2X;
}
```

```
#include"classe1.h"
#include"classe2.h"
int main()
{
    Classe2 *c2=new Classe2;
    Classe1 *c1=new Classe1;
    c1->setClasse2(c2);
    return(0);
}
```

Q 17. Donner le nom de chacune des relations. Identifier le programme C++ associé à chacune d'elle. Caractériser ces relations en termes :

- de durée de vie d'un objet de `Classe2` vis-à-vis de la durée de vie d'un objet de `Classe1` ;
- de l'accès des membres privés de `Classe2` depuis les méthodes de `Classe1` ;
- de l'accès de membres publics de `Classe2` depuis les méthodes de `Classe1` ;
- du partage d'un objet de `Classe2` avec deux objets de `Classe1`.

À partir du diagramme de classes du document technique DT2 et du diagramme de séquence du document technique DT3 :

Q 18. Identifier la classe qui contient l'ensemble constitué de la scène et des acteurs ainsi que la classe permettant de réaliser le rendu 3D. Justifier votre réponse.

Q 19. Quel objet est contenu dans l'objet `CRendu_3D_2D_OpenGL` ? Dans un programme principal, instancier le ou les classes nécessaires à l'exécution de l'application de base.

Q 20. Compléter sur votre copie la représentation UML de la classe `CModel3D_PhysX` proposée ci-dessous en faisant apparaître les différents attributs associés aux différentes relations entre les classes et en précisant leur visibilité, leur nom et leur type.

CModel3D_PhysX
<<create>>+CModel3D_PhysX() +createPlane(float x,float y, float z, float d): void +createCube(float x,float y, float z, float arete, float masse):void +createActors():void +stepPhysics(step:float) +getScenePointer():*PxScene

Q 21. Quel est le deuxième objet créé chronologiquement dans cette application ? Quels sont les objets créés dans le constructeur de ce dernier ? Préciser les méthodes appelées appartenant au deuxième objet créé, leur rôle et leur enchaînement chronologique.

Q 22. Identifier les classes qui constituent l'unique hiérarchie de ce diagramme. Préciser les classes de cette hiérarchie utilisées dans l'application de base (DT2).

Q 23. Est-il possible d'instancier ces classes ? Est-il possible de dériver ces classes ? Justifier vos réponses. Expliquer alors le principe de leur instanciation et les classes nécessaires permettant de créer les deux acteurs de l'application de base. Expliquer l'intérêt de laisser à une classe tiers le soin de créer les objets.

Q 24. La méthode `addActors` de la classe `PxScene` permet d'ajouter un nouvel acteur à la scène. Expliquer la raison pour laquelle ce prototype de méthode est correct pour l'ensemble de la hiérarchie. Quels sont les deux acteurs ajoutés à la scène ?

Tournez la page S.V.P.

Le programme ci-après propose l'implémentation C++ des méthodes `createCube` et `createActors` du diagramme de séquence du document technique DT3.

```
void CModel3D_PhysX::createCube(float x, float y, float z, float masse)
{
	//Position initiale du centre du cube
	PxTransform boxPos(PxVec3(x, y, z));
	//Dimensions du cube : demi-arête
	PxBoxGeometry boxGeometry(PxVec3(0.1, 0.1, 0.1));
	//Création de l'acteur
	gBox = PxCreateDynamic(*gPhysics->getPhysics(), boxPos, boxGeometry,
*gMaterial, 1.0);
	//Assignation de la masse
	gBox->setMass(masse);
	//Ajout dans la scène
	gScene->addActor(*gBox);
}
void CModel3D_PhysX::createActors()
{
	this->createPlane(0, 1, 0, 0);
	this->createCube(-2, 1, 3, 0.1);
}
```

Q 25. Proposer une modification de ce programme afin d'obtenir le rendu graphique du document technique DT4 dans lequel 50 cubes de demi-arête $\frac{a}{2} = 0{,}1$ et de masse $0{,}1$ kg sont générés de manière uniforme entre l'altitude 2 et 24.

Le cube de l'application de base est assimilé au modèle du solide indéformable (DT5) dont la position et l'orientation dans l'espace sont connues à chaque pas de la simulation.

Dans le monde 3D de `PhysX` qui précède le rendu 3D par `OpenGL`, les rotations sont calculées dans l'espace des quaternions (DT6). Pour le rendu 3D final, une conversion du quaternion associé vers la matrice de rotation équivalente est effectuée.

Q 26. Quelles sont les méthodes du document technique DT2 utilisant la classe `PxTransform` qui permettent de définir la position et l'orientation d'un corps rigide dynamique dans la scène ? Quels sont les objets qui composent une instance de cette classe (DT7) ?

Q 27. En se référant au document technique DT6, vérifier la relation du produit d'un quaternion unitaire avec son conjugué. D'après le test (DT4), vérifier les valeurs affichées du quaternion pour une rotation de 45° autour du vecteur unitaire $\vec{n} = \frac{1}{\sqrt{2}}\vec{\imath} + \frac{1}{\sqrt{2}}\vec{\jmath}$ appartenant au plan (Oxy).

Q 28. Expliquer pourquoi `PhysX` utilise une représentation des rotations par des quaternions et `OpenGL` par des matrices de rotation.

Q 29. D'après le diagramme de classes du document technique DT7, quelle est la méthode C++ utile pour convertir la position d'un objet défini par sa position et son quaternion (`PxTransform`) vers sa matrice de rotation en coordonnées homogènes (DT10).

Le comportement dynamique du cube ainsi initialisé en position et en orientation est régi par le principe fondamental de la dynamique rappelé dans le document technique DT5.

Le moteur `PhysX` permet de calculer la position et l'orientation de chaque acteur à chaque instant.

Q 30. Dresser un tableau qui met en relation les variables des relations du document technique DT5 et les méthodes des classes `PxRigidDynamic` et `PxRigidBody`. Quelle méthode permet d'ajouter un frottement fluide linéaire lors de la chute du cube ? Dans le cas d'un frottement fluide quadratique, quelle solution est envisageable ?

2.2 Modèle de programmation du véhicule

L'objectif de cette sous-partie est de comprendre le rôle des différentes classes de `PhysX` dédiée à la simulation d'un véhicule et de comparer le modèle proposé avec les résultats de test.

Conformément au principe de réutilisation, le projet final s'appuie sur le modèle natif de véhicule de la bibliothèque `PhysX`.

Ce véhicule est constitué d'un châssis, de quatre roues, d'une suspension, d'une direction (modèle d'Ackermann pour information) et d'une chaîne de transmission décrite à la figure 15 :

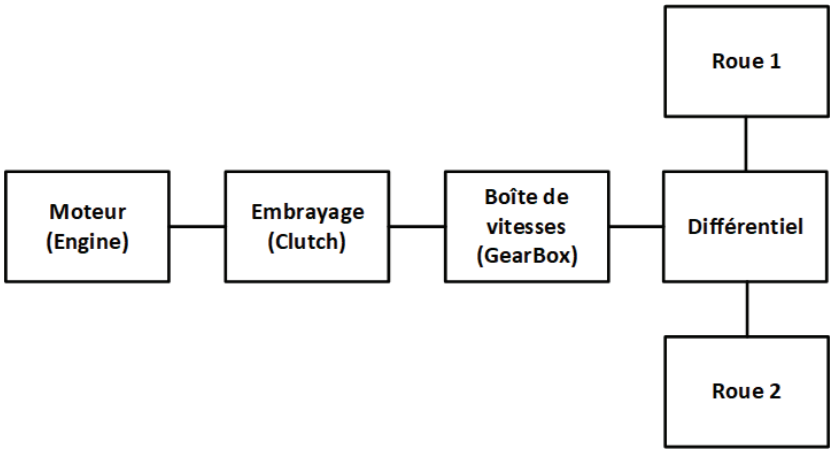


Figure 15 : modèle natif de traction

Le diagramme de classes du document technique DT8 illustre le principe de programmation d'un véhicule constitué entre autres des éléments décrits à la figure précédente.

Q 31. Dresser un tableau mettant en relation les blocs du modèle de traction et les classes du document technique DT8 associées. Est-il possible d'instancier la classe `PxVehicleDrive4W` ? Justifier votre réponse et proposer une solution pour créer une instance de cette classe. Il sera utile de préciser les classes nécessaires lors de cette création.

La programmation d'un véhicule repose en grande partie sur le principe des classes amies.
Le diagramme de classes ci-après (figure 16), représente un exemple de relation d'amitié entre deux classes qui permet à la classe « amie » d'accéder aux membres privés d'une classe acceptant cette amitié.

Tournez la page S.V.P.

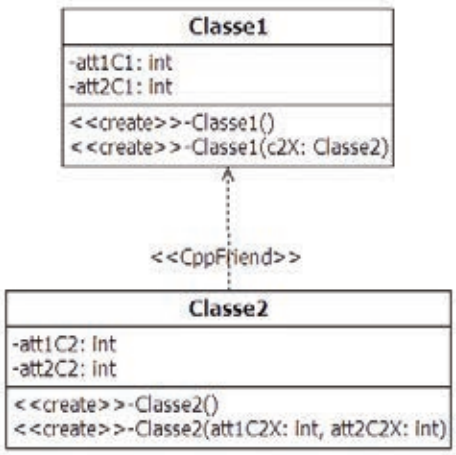


Figure 16 : relation d'amitié ; `classe1` est amie avec `classe2`

L'implémentation C++ et les tests associés à la figure 16 sont donnés ci-dessous :

Fichier header

```
#include<stdio.h>
namespace package1
{
class Classe2
{
    private:
        int att1C2;
        int att2C2;
    public:
        Classe2();
        Classe2(int att1C2X,int
att2C2X);
        friend class Classe1;
};
class Classe1
{
    private:
        int att1C1;
        int att2C1;

        Classe2 c2;

    public:

        Classe1();
        void setClasse2(Classe2
c2X);
};
}
```

Fichier source

```
#include "classesAmies.h"

namespace package1
{
Classe1::Classe1(){}
void Classe1::setClasse2(Classe2 c2X)
{
    c2.att1C2=c2X.att1C2;
    c2.att2C2=c2X.att2C2;
    printf("private member
c2.att1C2=%d\n",c2.att1C2);
    printf("private member
c2.att2C2=%d\n",c2.att2C2);
}
Classe2::Classe2(){}
Classe2::Classe2(int att1C2X,int
att2C2X)
{
    att1C2=att1C2X;
    att2C2=att2C2X;
}
}
```

Programme principal

```
#include "classesAmies.h"
using namespace package1;
int main()
{
    Classe2 c2(210,220);
    Classe1 c1;
    c1.setClasse2(c2);
    return 0;
}
```

Test console

```
C:\Qt\Qt5.11.1\Tools\QtCreator\bin\qtcreatc
private member c2.att1C2=210
private member c2.att2C2=220
```

Q 32. D'après le résultat de la console ci-avant et en relation avec le programme C++, déterminer le(s) instruction(s) permettant de mettre en œuvre cette amitié.

Q 33. Dans le cas où `Classe2` est amie avec une nouvelle classe `Classe3`, la classe `Classe1` devient-elle à son tour amie avec la classe `Classe3` ? Que devient cette relation d'amitié en cas d'héritage de la classe `Classe1` ? Justifier vos réponses.

La fonction C++ `createVehicle4W` ci-après permet de créer et de configurer un véhicule `PhysX`. L'initialisation des membres de chaque objet est effectuée dans leur constructeur respectif.

```
PxVehicleDrive4W *createVehicle4W(PxPhysics* physics, PxU32 nbNonDrivenWheels)
{
      PxRigidDynamic *veh4WActor=NULL;
      PxVehicleChassisData rigidBodyData;
      PxVehicleWheelsSimData *wheelsSimData=
PxVehicleWheelsSimData::allocate(numWheels);
      PxVehicleDriveSimData4W driveSimData;
      PxVehicleDifferential4WData diff;
      PxVehicleEngineData engine;
      PxVehicleGearsData gears;
      PxVehicleClutchData clutch;
      PxVehicleAckermannGeometryData ackermann;
      driveSimData.setDiffData(diff);
      driveSimData.setEngineData(engine);
      driveSimData.setGearsData(gears);
      driveSimData.setClutchData(clutch);
      driveSimData.setAckermannGeometryData(ackermann);

      veh4WActor=createVehicleActor(rigidBodyData,#,#,#,#,#,#,#,#,#,#);

      PxVehicleDrive4W *vehDrive4W=...


      return vehDrive4W;
}
```

Q 34. Donner les instructions permettant de créer le véhicule et d'assigner la vitesse de rotation maximale du moteur à une valeur égale à 628 rad/s. Critiquer le choix de `PhysX` pour résoudre la difficulté de l'héritage de l'amitié.

La fenêtre de `debug` de l'instance `gVehicle4W` de la classe `PxVehicleDrive4W` est donnée dans le document technique DT9.

Q 35. Identifier les hiérarchies du diagramme de classes du DT9 et vérifier la cohérence en précisant si nécessaire, les différences entre ces résultats de test et le modèle UML du document technique DT8.

Tournez la page S.V.P.

Partie 3. Modélisation du rétroviseur grand angle

L'objectif de cette partie est de modéliser le rendu du rétroviseur grand angle dans une scène 3D.

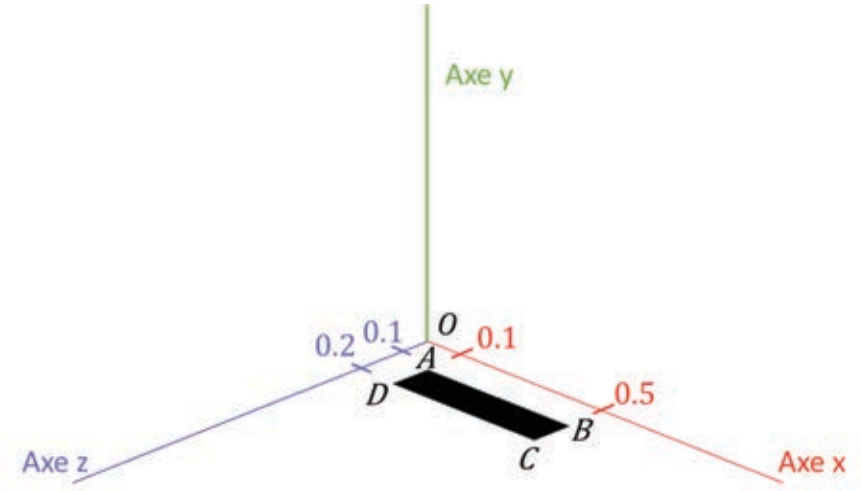
Afin de remplir sa fonction pédagogique, le simulateur doit permettre au conducteur de regarder dans les rétroviseurs de son camion. Les rétroviseurs du système sont simulés sur les écrans latéraux par le rendu d'une image déformée et plaquée sur un objet de la scène 3D.

Figure 17 : le rétroviseur grand angle (entouré en bleu) permet au conducteur de contrôler le positionnement du flanc de sa cuve dans son environnement

Pour le rendu dynamique de l'image dans le rétroviseur, cette dernière est recalculée à chaque pas de temps à partir d'une caméra virtuelle située à la place du rétroviseur, et qui pointe vers l'arrière du camion avec un angle d'ouverture spécifique.

3.1 Positionnement des objets dans la scène

Dans cette partie, le camion est d'abord représenté dans un plan (x, z) par un rectangle dans l'espace à 4 dimensions $(x, y, z, 1)$ des coordonnées homogènes. Les 4 sommets A, B, C et D de ce rectangle ont des coordonnées normalisées entre -1 et 1. Les valeurs de ces sommets sont :



A: (0,1; 0; 0,1; 1)
B: (0,5; 0; 0,1; 1)
C: (0,5; 0; 0,2; 1)
D: (0,1; 0; 0,2; 1)

Figure 18 : représentation 2D simplifiée du camion

Pour les besoins de la représentation de la scène, il est proposé d'appliquer une série de déplacements au camion pour que ses sommets se retrouvent dans la position ci-dessous :

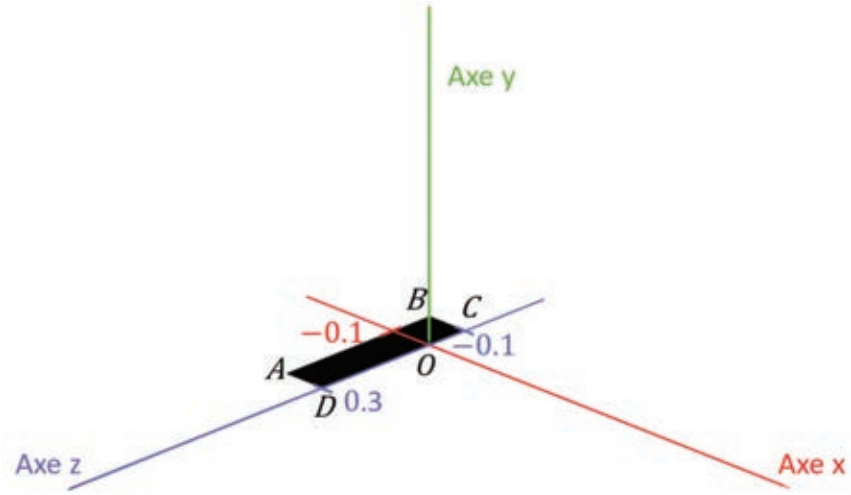


Figure 19 : représentation du camion après transformations

$$
\begin{array}{lllll}
A: & (-0{,}1; & 0; & 0{,}3; & 1) \\
B: & (-0{,}1; & 0; & -0{,}1; & 1) \\
C: & (0; & 0; & -0{,}1; & 1) \\
D: & (0; & 0; & 0{,}3; & 1)
\end{array}
$$

Q 36. Proposer une combinaison d'une rotation et d'une translation permettant cette transformation de coordonnées. Pour chacune des 2 transformations proposées, écrire en dimension 4 à partir du DT10, les matrices de translation $[T]$ de rotation $[R]$ correspondantes avec leurs valeurs numériques associées.

Q 37. Exprimer la matrice de transformation géométrique $[G]$ comme combinaison de $[T]$ et $[R]$. Proposer des valeurs numériques pour $[G]$ et détailler le calcul matriciel de la transformation avec le point A.

Dans le cadre du rendu d'objets 3D, les sommets des objets ne sont généralement pas définis à l'endroit où ils doivent se trouver dans le monde virtuel : ils sont souvent définis « centrés sur l'origine ». C'est pourquoi une matrice de transformation doit leur être appliquée (la même matrice pour tous les sommets d'un même objet).

Q 38. Pourquoi ne pas définir directement la position finale des sommets lors de la définition d'un objet 3D ?

3.2 Positionnement de la caméra dans la scène

Dans cette partie le camion est représenté par un pavé droit (figure 20). Ce pavé a pour base le rectangle $ABCD$ et pour épaisseur normalisée la valeur 0,1 selon l'axe y.

Initialement, lors d'un rendu de scène `OpenGL`, la caméra est placée à l'origine et regarde selon l'axe z dans le sens des z négatifs.
La méthode `gluLookAt()`, dont la documentation est donnée dans le DT11, permet de définir un changement de point de vue de la scène.

Tournez la page S.V.P.

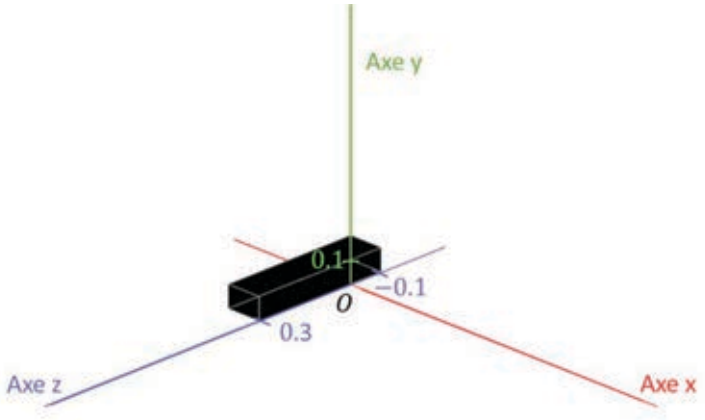
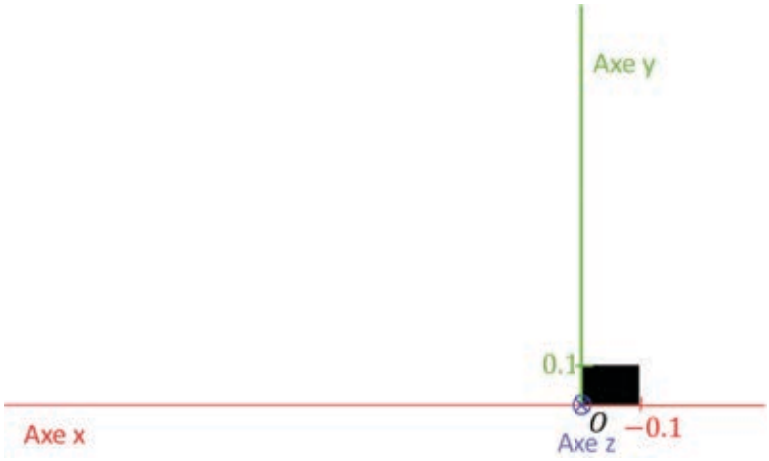


Figure 20 : représentation du camion par un pavé droit

Q 39. Écrire l'instruction C++ appelant la méthode `gluLookAt()` qui permettrait de rendre la scène, à partir d'une caméra placée sur l'origine O, regardant selon l'axe z le point de coordonnée $(0; 0; 1)$ et ayant comme référence verticale l'axe y. Le rendu de la scène pourrait être le suivant avec une projection orthographique.



Q 40. Ce changement de point de vue est réalisé en multipliant les coordonnées de tous les sommets de la scène par une matrice (`ModelView Matrix`). À l'aide du document technique DT11 et du rappel sur le produit vectoriel présenté dans le document technique DT10, exprimer successivement les matrices $F, f, UP'',$ et u.

Q 41. En déduire la matrice M de changement de point de vue.

Q 42. Appliquer ce changement de point de vue sur le point A de coordonnées $(-0{,}1;\ 0;\ 0{,}3;\ 1)$.

3.3 Mesure de l'angle d'ouverture de la caméra

Dans cette partie, le camion est représenté par un modèle 2D vu de dessus.

Pour plus de réalisme, le rendu de la scène doit se faire en perspective. Afin d'estimer un angle d'ouverture de la caméra qui pourrait simuler une image vue du rétroviseur, une mesure est effectuée manuellement, sur le système réel, en plaçant deux cônes à l'arrière du camion.

L'un est placé sous le coin arrière droit du camion, l'autre est placé à l'extrémité du champ de vision du conducteur dans le rétroviseur.

Ce second cône est placé dans l'alignement de l'arrière du camion de manière à ce qu'il soit à la limite de la visibilité droite du rétroviseur. Une fois le cône posé, les distances par rapport au rétroviseur et la distance entre les cônes sont mesurées.

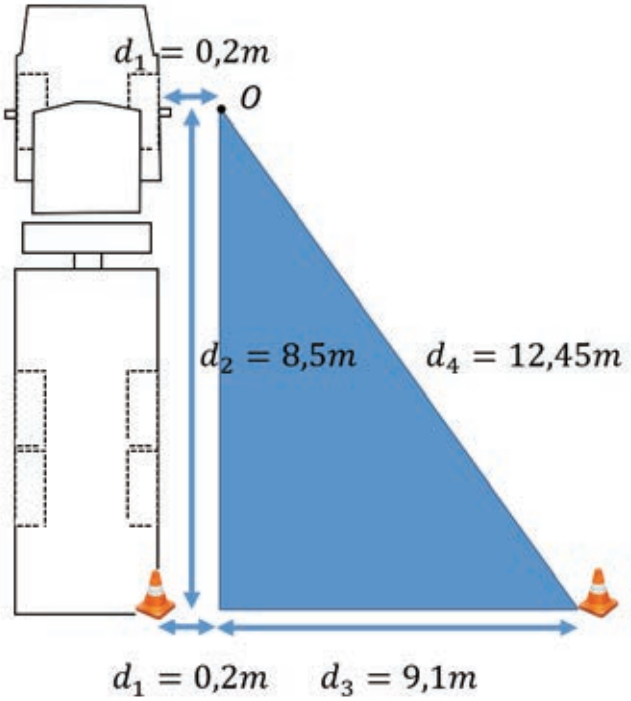


Figure 21 : mesures effectuées sur le système réel

Le centre du rétroviseur, représenté par le point O sur la figure est situé à une distance $d_1 = 0{,}2$ m par rapport au flanc du camion. La distance entre le rétroviseur et la ligne de l'arrière du camion, parallèle au flanc du camion, est $d_2 = 8{,}5$ m. Les deux plots sont posés dans l'alignement de l'arrière du camion, c'est-à-dire orthogonalement au flanc. La distance entre les deux plots est de $d_1 + d_2 = 9{,}3$ m.

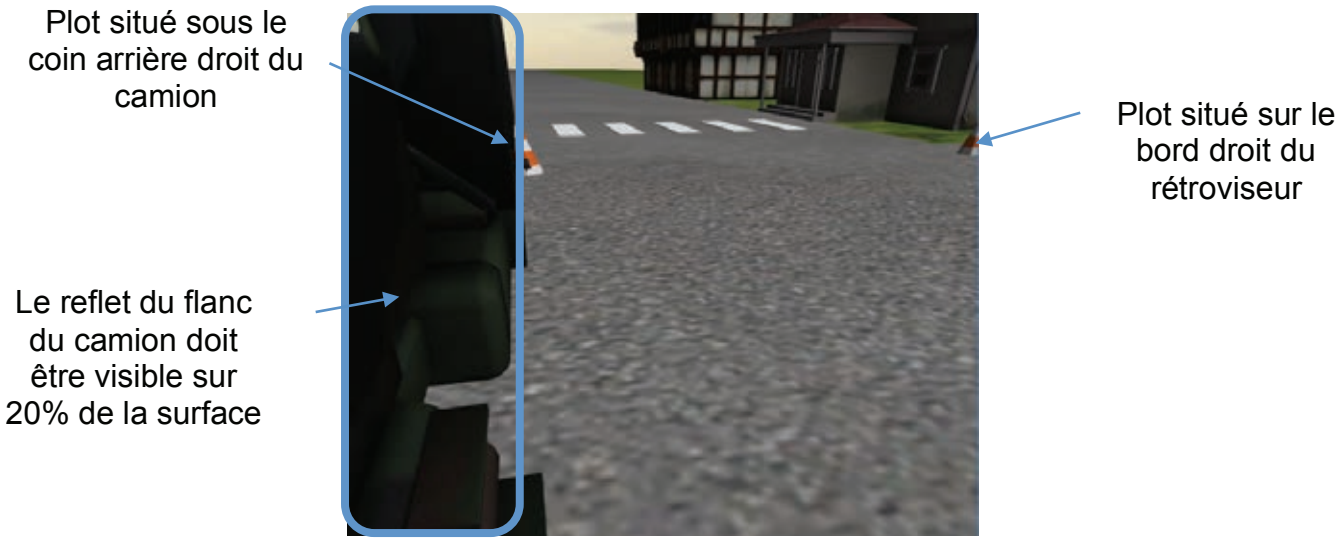


Figure 22 : scène vue du chauffeur dans le rétroviseur grand angle dans le système simulé

Tournez la page S.V.P.

Par ailleurs, lorsque le rétroviseur est regardé depuis la place du conducteur, il est réglé latéralement de manière à ce que le flanc du véhicule soit visible sur 20% de la surface du miroir (figure 22). Pour le réglage en hauteur, il est approximativement réglé afin que la ligne d'horizon soit visible en haut du rétroviseur.

Le point O de la figure 23 est considéré comme le centre du rétroviseur. Le champ de vision reflété sur le rétroviseur est borné par deux droites formant un angle θ qui représente l'angle d'ouverture de la caméra.

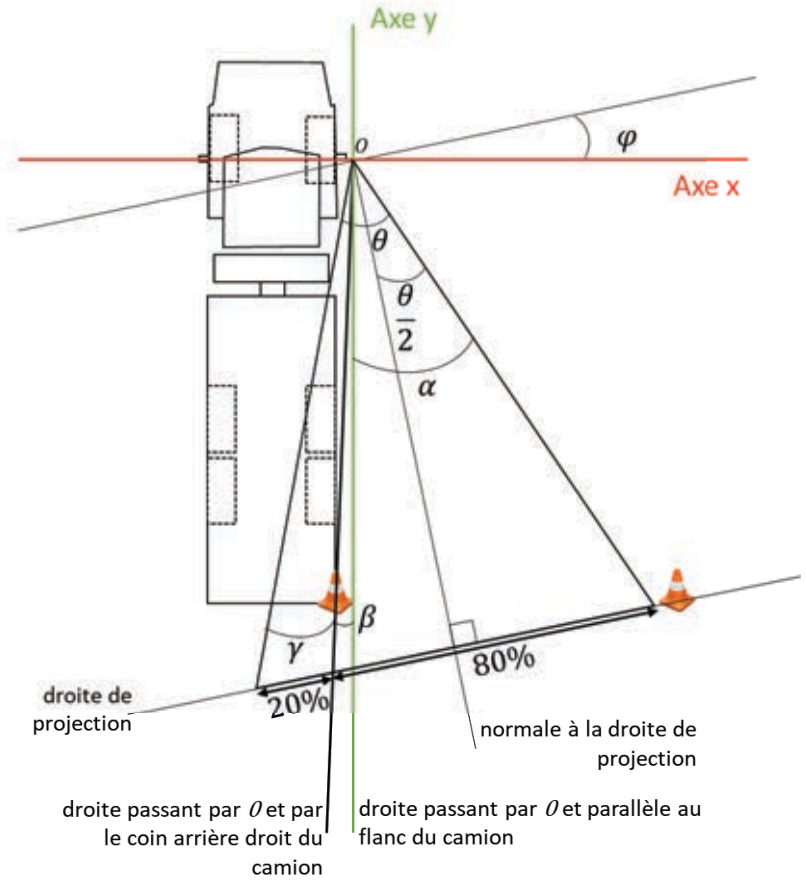


Figure 23 : calcul de l'angle d'ouverture de la caméra à partir des mesures

Q 43. Exprimer la valeur des angles α et β en fonction des distances d_1, d_2 et d_3. Donner en degrés les valeurs numériques de α et β.

Q 44. Sachant que $\theta = \alpha + \beta + \gamma$ et que la partie permettant de voir le camion représente 20% de la longueur totale vue du rétroviseur, exprimer γ en fonction de α et β sans faire l'approximation des petits angles.

Q 45. Donner la valeur numérique en degrés de γ et celle de l'angle d'ouverture θ.

Pour la suite du sujet, la valeur $\theta = 60°$ sera retenue.

3.4 Rendu en perspective de la scène

Q 46. À partir du document technique DT11 relatif à la fonction `gluPerspective()`, expliquer la nécessité de définir, dans le cadre général d'un rendu en perspective, une distance proche `zNear` et une distance lointaine `zFar` au-delà desquelles la scène ne peut pas être représentée.

Le document réponse DR2 présente le programme qui permet de mettre en place la vue ainsi que la perspective grâce à des multiplications successives de matrices. Pour les besoins du sujet, les méthodes `gluPerspective()` et `gluLookat()` n'ont volontairement pas été utilisées.

Q 47. En prenant l'hypothèse que le rétroviseur soit carré et qu'il réfléchisse avec le même angle horizontalement que verticalement, compléter le document réponse DR2 pour obtenir la vue souhaitée.

3.5 Utilisation des `shaders` pour l'effet miroir

Dans cette partie le camion est représenté par un modèle 3D dans son environnement.

L'image obtenue par le rendu de la scène via la caméra configurée précédemment ne peut pas être rendue directement à l'écran, mais est sauvegardée dans un `FrameBufferObject` (équivalent d'une texture en mémoire).

Ainsi, elle peut être réutilisée en tant que texture. Cette texture est appliquée sur un objet 3D plat, aux bords arrondis, ayant la forme du rétroviseur du camion. Ce miroir étant un grand-angle, l'image devra subir aussi une déformation pour donner un effet « bombé ». Le résultat attendu est le suivant :

Figure 24 : à gauche, l'image rendue par la caméra, au milieu la même image après retournement et déformation, à droite l'image transformée et plaquée sur le rétroviseur

Un objet 3D plat, dont le maillage est représenté en orange ci-dessous, est utilisé pour plaquer l'image calculée. Ses coordonnées de texture sont définies entre 0 et 1 selon les axes x et y.

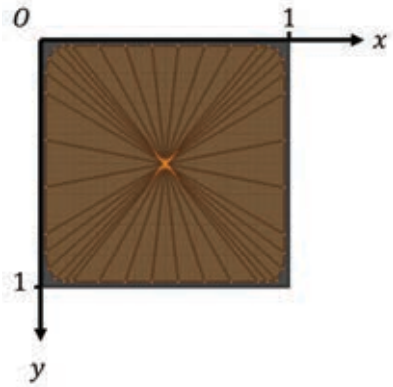


Figure 25 : maillage de l'objet sur lequel sera plaquée l'image calculée par la caméra

Tournez la page S.V.P.

Q 48. Expliquer, avec schéma à l'appui si nécessaire, le principe général de l'application des textures sur des objets 3D.

Par une interpolation linéaire de pixels, l'application de la texture sur le maillage ne permet pas de réaliser complètement l'effet miroir bombé. L'utilisation de `shaders` est nécessaire. Il s'agit de programmes compilés et exécutés par la carte graphique permettant de calculer la couleur de chaque pixel indépendamment.

Q 49. De manière générale, en supposant que les coordonnées d'un point sur une image soient (x, y), avec $x = [0, 1]$ et $y = [0, 1]$, quelles seraient les coordonnées de ce même point sur une image inversée ? (inversion au sens d'un effet miroir)

Le pipeline de rendu d'un `shader` exécuté sur une carte graphique est présenté dans le document technique DT12. Le `shader` utilisé dans le simulateur et présenté dans le document technique DT13, est codé en GLSL (*OpenGL Shading Language*). Ce langage permet un contrôle avancé du pipeline de la carte graphique.

Q 50. Expliquer la stratégie utilisée pour réaliser l'effet bombé.

Q 51. Quel est l'intérêt de réaliser ces opérations sur le processeur d'une carte graphique plutôt que sur le processeur d'un ordinateur ?

Q 52. En utilisant le document technique DT14, identifier dans quel composant de la carte graphique sont réalisées les étapes de *vertex processing*, de *tesselation* ainsi que les calculs de projection liés au changement de point de vue. Combien y a-t-il d'instances de ce composant par cluster ?

Le document technique DT14 indique que la carte graphique utilisée dans le simulateur contient 2560 cœurs CUDA. Chacun d'eux contient une unité arithmétique et logique (ALU, Arithmetic and Logic Unit) ainsi qu'une unité de calcul en virgule flottante (FPU, Floating Point Unit).

Q 53. Expliquer succinctement la différence entre ces 2 unités.

Q 54. Chacun des cœurs possède 48 ko de mémoire cache de niveau L1. Qu'est-ce que la mémoire cache d'un microprocesseur ?

Document technique DT1 : Fonction ODEINT de Scipy

Description

`sol=scipy.integrate.odeint(func, y0, t, args=())`

Integrate a system of ordinary differential equations. Solve a system of ordinary differential equations using `lsoda` from the FORTRAN library `odepack`. Solves the initial value problem for stiff or non-stiff systems of first order ode-s:

`dy/dt = func(y, t0, ...)`, where `y` can be a vector.

Parameters

`func` : callable(y, t0, ...), computes the derivative of `y` at `t0`.
`y0` : array, initial condition on `y` (can be a vector).
`t` : array, a sequence of time points for which to solve for `y`. The initial value point should be the first element of this sequence.
`args` : tuple, optional, extra arguments to pass to function.

Returns

`sol` : array, `shape (len(t), len(y0))`, array containing the value of `y` for each desired time in `t`, with the initial value `y0` in the first row.

Example

The second order differential equation for the angle theta of a pendulum acted on by gravity with friction can be written:

$$\frac{d^2\theta}{dt^2} + b\frac{d\theta}{dt} + c.\sin(\theta) = 0$$

Where `b` and `c` are positive constants. To solve this equation with **`odeint`**, we must first convert it to a system of first order equations. By defining the angular velocity `Ω(t)=θ'(t)`, we obtain the system:

$$\begin{cases} \dfrac{d\theta}{dt} = \Omega(\mathrm{t}) \\ \dfrac{d\Omega}{dt} = -b\dfrac{d\theta}{dt} - c.\sin(\theta) \end{cases}$$

Let `y` be the vector [theta, omega]. We implement this system in Python as:

```
def pend(y,t,b,c):
    theta,omega=y
    dydt=[omega,-b*omega-c*math.sin(theta)]
    return dydt
```

For initial conditions, we assume the pendulum is nearly vertical with `theta(0)=pi-0.1`, and it initially at rest, so `omega(0)=0`. Then the vector of initial conditions is:

```
y0=[math.pi-0.1,0.0]
```

We generate a solution 101 evenly spaced samples in the interval 0 <= t <= 10. So our array of times is:

```
t=numpy.linspace(0,10,101)
```

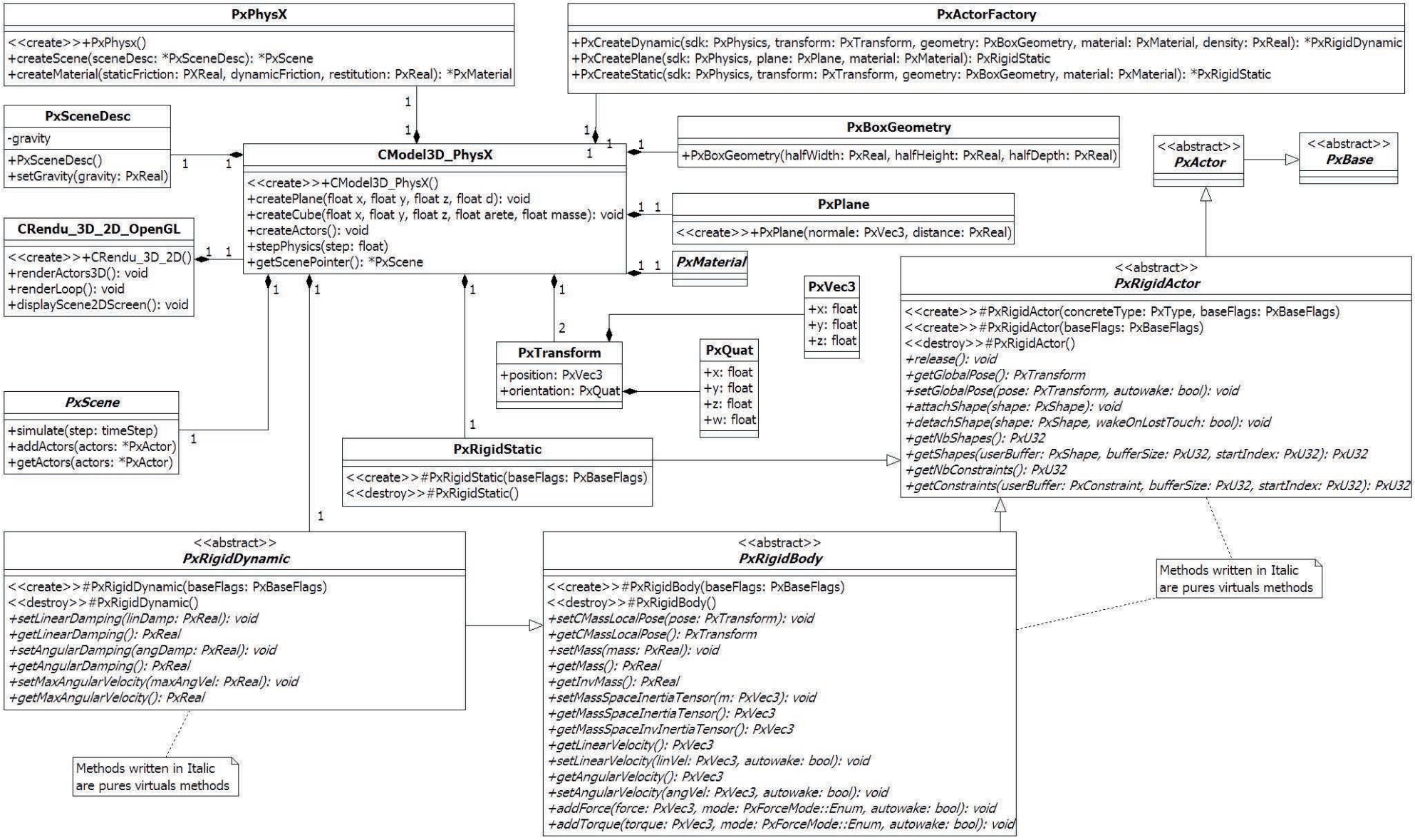
Call `odeint` to generate the solution. To pass the parameters `b` and `c` to the **pend** function; we give them to `odeint` using the `args` argument:

```
from scipy.integrate import odeint
sol=odeint(pend,y0,t,args=(b,c))
```

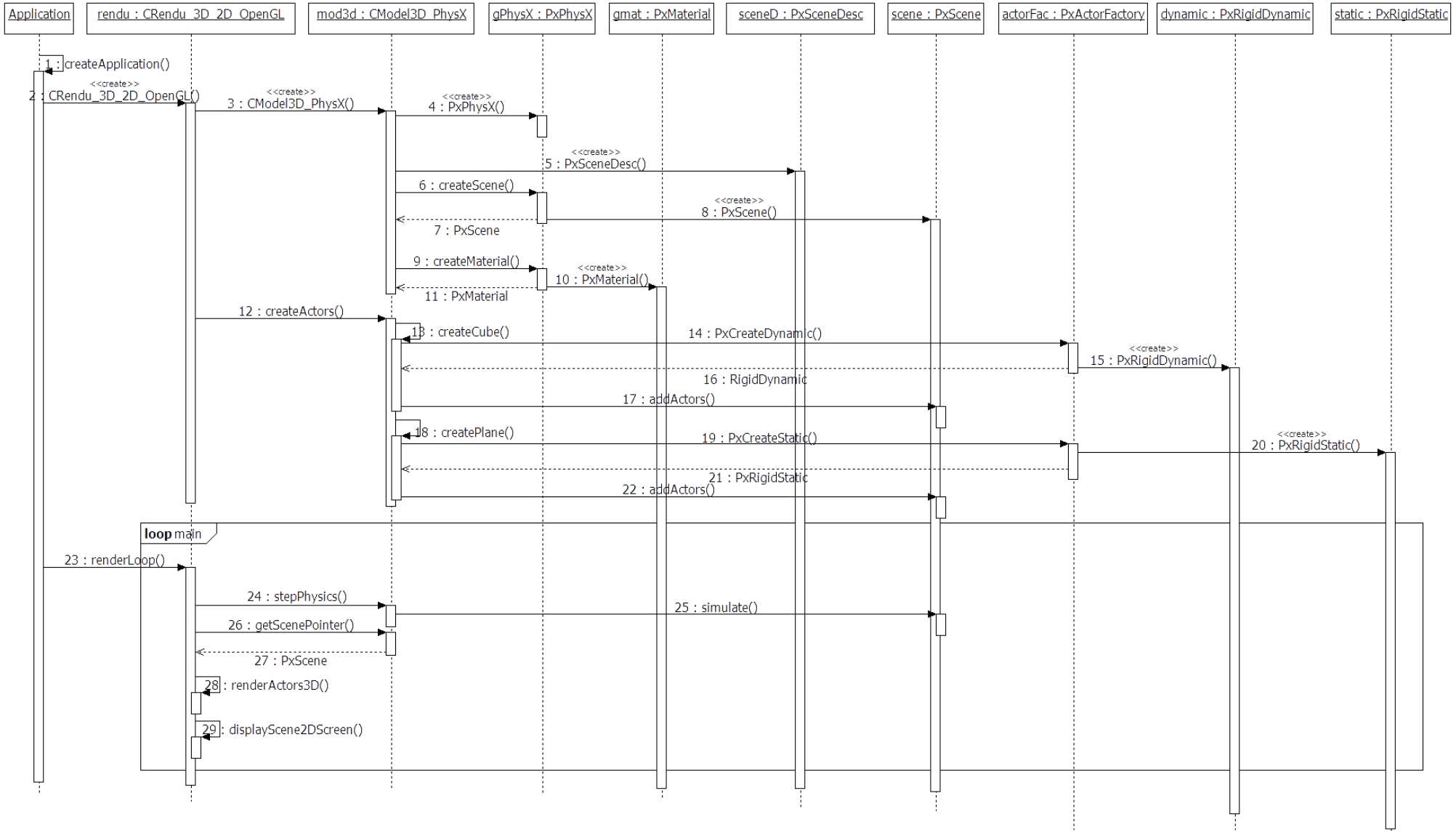

Tournez la page S.V.P.

C

Document technique DT2 : Diagramme de classes de l'application de test utilisant la bibliothèque `PhysX`.



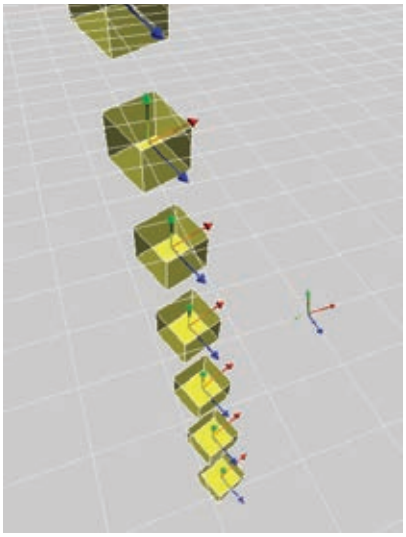
Document technique DT3 : Diagramme de séquence de l'application de test.



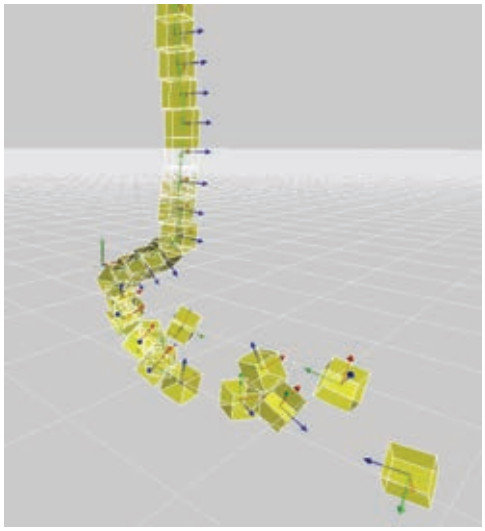
Tournez la page S.V.P.

Document technique DT4 : Tests de l'application de base

Rendu graphique de l'application de base.
En rouge l'axe $\vec{x}$, en vert l'axe $\vec{y}$ et en bleu l'axe $\vec{z}$.

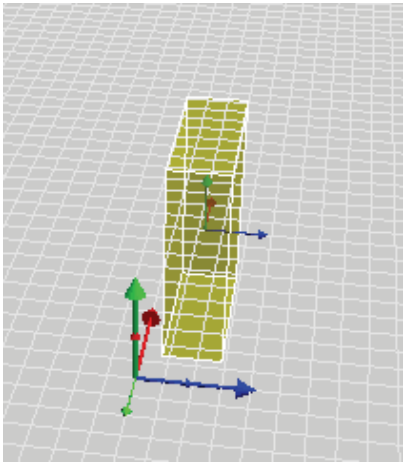


Scène à t=0 s

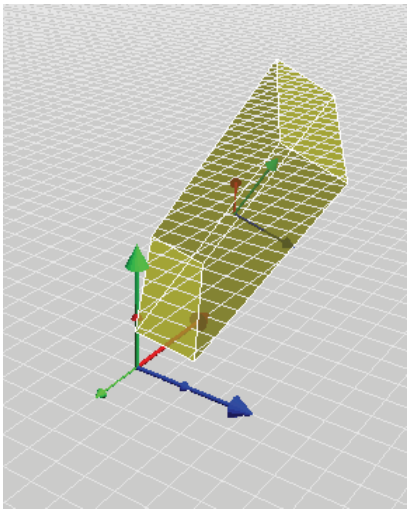


Scène à t=3 s

Rotation de 45° avec un quaternion
En rouge l'axe $\vec{x}$, en vert l'axe $\vec{y}$ et en bleu l'axe $\vec{z}$.



Pose initiale



Rotation de 45°

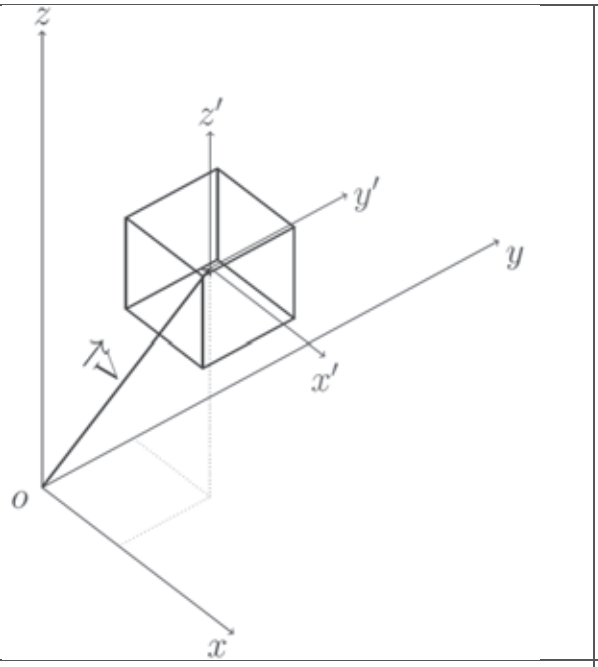
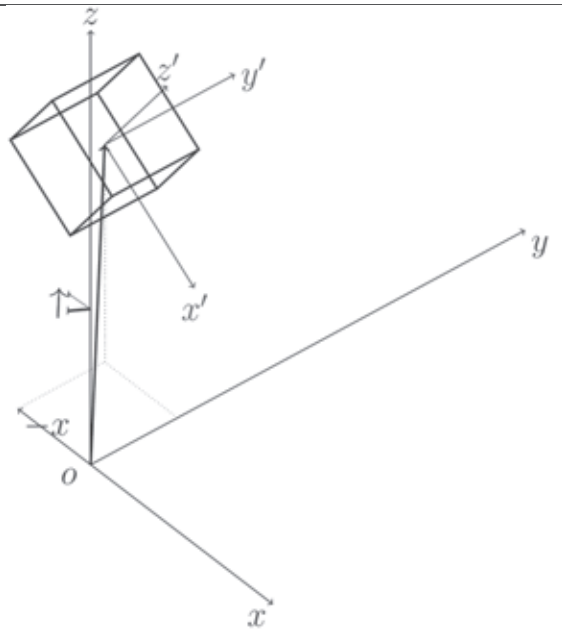
C:\WINDOWS\system32\cmd.exe

```
connexion ok
initialisation PhysX ok
pose initiale
position x:0.5
position y:0.5
position z:0.5
quaternion q, w:1
quaternion q, x:0
quaternion q, y:0
quaternion q, z:0
rotation de 45 degres
position x:0.5
position y:0.5
position z:0.5
quaternion q, w:0.923976
quaternion q, x:0.270433
quaternion q, y:0.270433
quaternion q, z:0
```

Valeurs position et quaternion avant et après la rotation

Document technique DT5 : Solide indéformable dans `PhysX`

Repère d'un objet et repère du Monde :
Un objet 3D est défini dans son repère $(\overrightarrow{x'}, \overrightarrow{y'}, \overrightarrow{z'})$. Le repère est centré sur son centre de masse. L'objet est positionné dans le repère du Monde $(\vec{x}, \vec{y}, \vec{z})$ par l'intermédiaire de son vecteur position $\vec{R}$ et d'une orientation angulaire. Dans le repère de l'objet, il est possible de modifier l'orientation et la taille de l'objet. Et dans le repère du Monde, il est possible d'effectuer des rotations, des translations et des changements d'échelle.

	
Translations selon x, y, z de l'objet dans le repère du Monde.	Rotation de l'objet dans son repère (selon y') et dans le repère du Monde (selon y).

Mouvement linéaire :
Pour un mouvement linéaire, le solide indéformable est modélisé par une masse. La somme des forces s'applique au centre de masse. Conformément à l'équation de la résultante du principe fondamental de la dynamique, l'accélération est donnée par la relation suivante où $\vec{R}$ est le vecteur position :

$$m\vec{\gamma} = m\left(\frac{d^2\vec{R}}{dt^2}\right)_{/(\vec{x},\vec{y},\vec{z})} = \sum \vec{F}$$

Mouvement de rotation :
Conformément à l'équation du moment du principe fondamental de la dynamique :

$$\left(\frac{d\vec{L}}{dt}\right)_{/(\vec{x},\vec{y},\vec{z})} = \bar{\bar{J}} \times \left(\frac{d\vec{\omega}}{dt}\right)_{/(\vec{x},\vec{y},\vec{z})} = \sum \vec{T}$$

Collision :
Le modèle de collision n'est pas étudié.

Tournez la page S.V.P.

Document technique DT6 : Les quaternions

Les quaternions sont avantageusement utilisés pour calculer les rotations de vecteurs en remplacement des matrices de rotation.
Un quaternion q est une paire ordonnée constituée d'un scalaire ω associé à l'angle de rotation et d'un vecteur à 3 dimensions $\vec{v}$ associé à l'axe de rotation :

$$q = (\omega, \vec{v})$$

Semblablement aux nombres complexes, un quaternion peut être considéré comme la somme d'une partie réelle (ω) et d'une partie imaginaire ($\vec{v}$) :

$$q = \omega + \vec{v}$$

La norme du quaternion unitaire est définie par :

$$|q|^2 = \omega^2 + x^2 + y^2 + z^2 = 1$$

Le conjugué de q est défini par :

$$\bar{q} = q^{-1} = \omega - \vec{v}$$

Le corps des quaternions contient l'addition :

$$q_1 + q_2 = \omega_1 + \omega_2 + \overrightarrow{v_1} + \overrightarrow{v_2}$$

et la multiplication non commutative :

$$q_1 q_2 \neq q_2 q_1$$

$$q_1 q_2 = (\omega_1 + \overrightarrow{v_1})(\omega_2 + \overrightarrow{v_2}) = (\omega_1\omega_2 - \overrightarrow{v_1} \cdot \overrightarrow{v_2},\ \omega_1\overrightarrow{v_2} + \omega_2\overrightarrow{v_1} + \overrightarrow{v_1} \times \overrightarrow{v_2})$$

Pour un quaternion unitaire :

$$q\bar{q} = 1$$

La rotation d'un vecteur $\vec{v}$ d'un angle θ suivant l'axe porté par le vecteur $\vec{n}$, consiste à considérer ce dernier comme un quaternion imaginaire $p = (0, \vec{v})$ et l'opération suivante permet de calculer le quaternion $p' = \left(0, \overrightarrow{v'}\right)$:

$$p' = qpq^{-1}$$

où le vecteur $\overrightarrow{v'}$ représente la rotation du vecteur $\vec{v}$ et où le quaternion q s'exprime selon l'expression suivante :

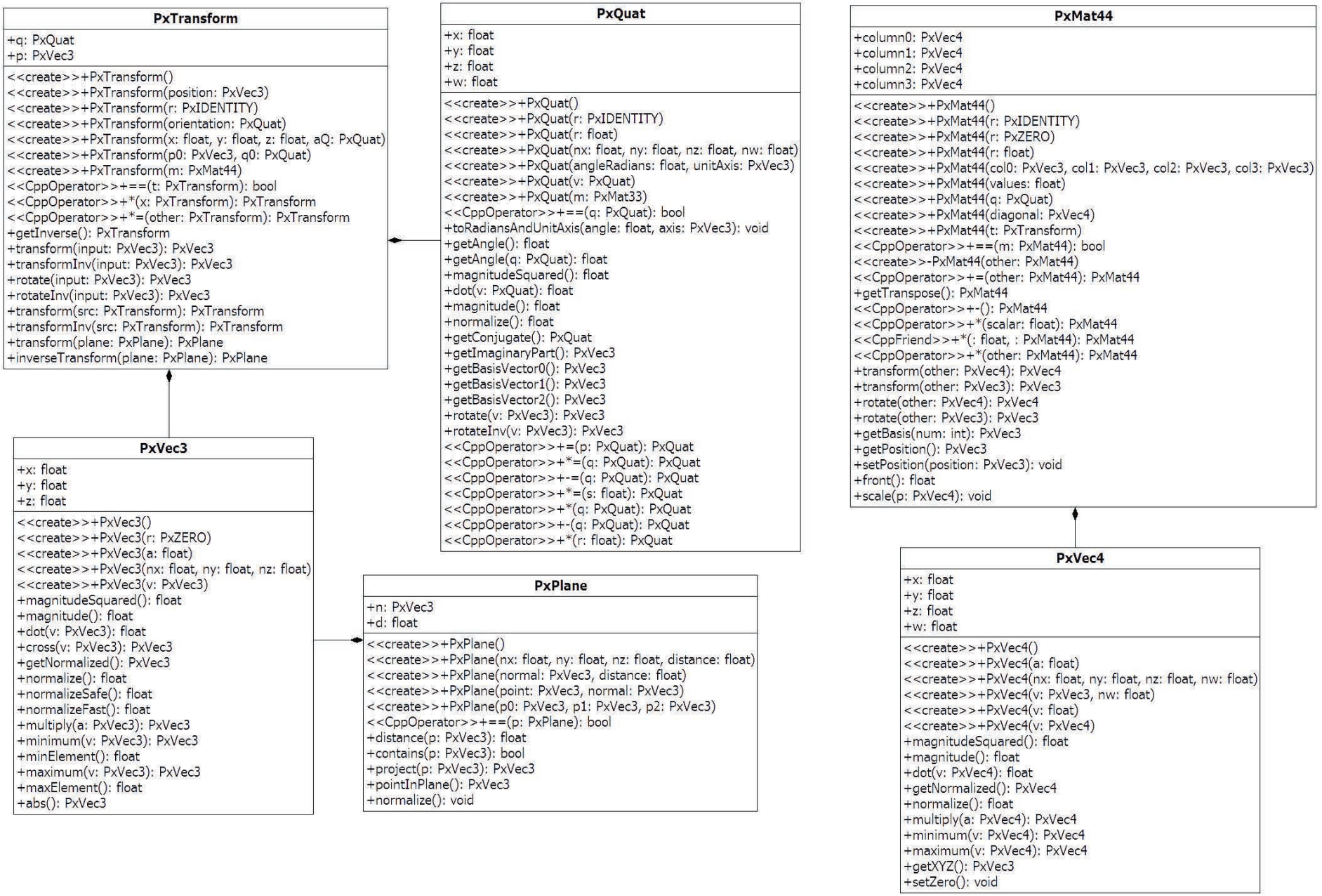
$$q = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\vec{n}$$

Conversion d'un quaternion en matrice de rotation :

La matrice de rotation associée au quaternion $p = a + b\vec{\imath} + c\vec{\jmath} + d\vec{k}$ s'exprime suivant la relation ci-dessous :

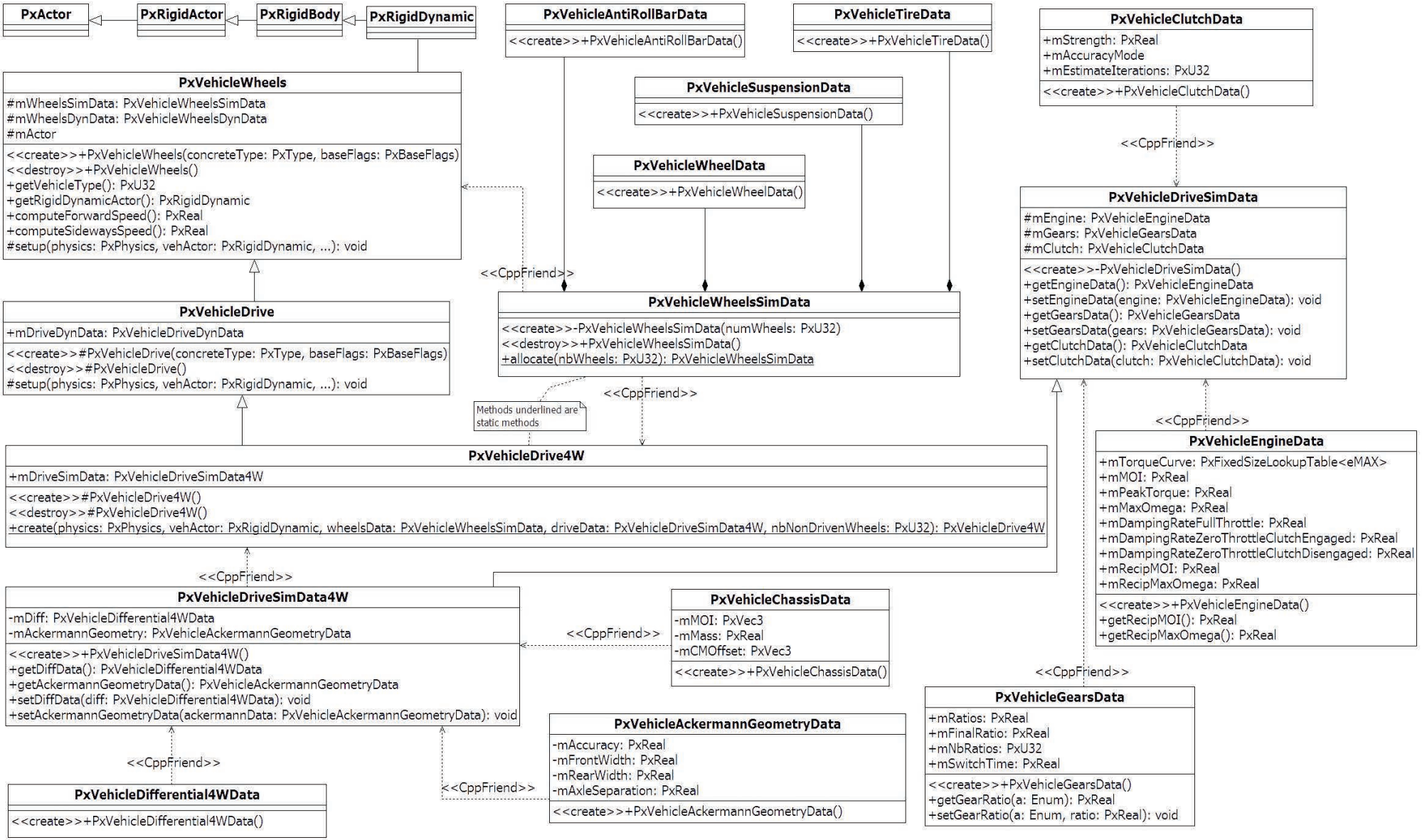
$$M = \begin{bmatrix} a^2 + b^2 + c^2 + d^2 & 2(bc - ad) & 2(bd - ac) \\ 2(bc + ad) & a^2 + b^2 + c^2 - d^2 & 2(cd - ab) \\ 2(bd - ac) & 2(cd - ab) & a^2 - b^2 - c^2 + d^2 \end{bmatrix}$$

Document technique DT7 : Diagramme de classes pour les transformations géométriques



Tournez la page S.V.P.

Document technique DT8 : Diagramme de classes d'un véhicule dans PhysX



Document technique DT9 : Fenêtre de debug de l'application « véhicule »

Watch 1

Name	Value	Type
gVehicle4W	0x000000000483cdd0 {mDriveSimData={mDiff={mFrontRearSplit=0.449999988 mFrontLeftRightSplit=0.500000000 ...} ...} }	physx::PxVehicleDrive4W *
physx::PxVehicleDrive	{mDriveDynData={mControlAnalogVals=0x000000000483ce90 {0.000000000, 0.000000000, 0.000000000, 0.000000000, ...} ...} }	physx::PxVehicleDrive
physx::PxVehicleWheels	{mWheelsSimData={mNormalisedLoadFilter={mMinNormalisedLoad=0.000000000 mMinFilteredNormalisedLoad=0.230800003 ...} ...} ...}	physx::PxVehicleWheels
physx::PxBase	{mConcreteType=513 mBaseFlags={mBits=3 } }	physx::PxBase
mWheelsSimData	{mNormalisedLoadFilter={mMinNormalisedLoad=0.000000000 mMinFilteredNormalisedLoad=0.230800003 mMaxNormalisedLoad=...} ...}	physx::PxVehicleWheelsSimData
mWheelsDynData	{mWheels4DynData=0x000000000483d8d0 {mWheelSpeeds=0x000000000483d8d0 {0.000000000, 0.000000000, 0.000000000, ...} ...} ...}	physx::PxVehicleWheelsDynData
mActor	0x0000000004849d00 {...}	physx::PxRigidDynamic * {PhysX3DEBUG_x64.dll!physx::NpRigidDynamic}
mNbNonDrivenWheels	4	unsigned int
mOnConstraintReleaseCounter	2 '\x2'	unsigned char
mType	0 '\0'	unsigned char
mPad	0x000000000483ce7e "ffffffffffffffff..."	unsigned char[14]
mDriveDynData	{mControlAnalogVals=0x000000000483ce90 {0.000000000, 0.000000000, 0.000000000, 0.000000000, 0.000000000, ...} ...}	physx::PxVehicleDriveDynData
mControlAnalogVals	0x000000000483ce90 {0.000000000, 0.000000000, 0.000000000, 0.000000000, 0.000000000, 0.000000000, 0.000000000, ...}	float[16]
mUseAutoGears	true	bool
mGearUpPressed	false	bool
mGearDownPressed	false	bool
mCurrentGear	2	unsigned int
mTargetGear	2	unsigned int
mEnginespeed	0.000000000	float
mGearSwitchTime	0.000000000	float
mAutoBoxSwitchTime	0.000000000	float
mPad	0x000000000483cee8 {3452816845, 3452816845}	unsigned int[2]
mDriveSimData	{mDiff={mFrontRearSplit=0.449999988 mFrontLeftRightSplit=0.500000000 mRearLeftRightSplit=0.500000000 ...} ...}	physx::PxVehicleDriveSimData4W
physx::PxVehicleDriveSimData	{mEngine={mTorqueCurve={mDataPairs=0x000000000483cef0 {0.000000000, 0.800000012, 0.330000013, 1.00000000, ...} ...} ...} ...}	physx::PxVehicleDriveSimData
mEngine	{mTorqueCurve={mDataPairs=0x000000000483cef0 {0.000000000, 0.800000012, 0.330000013, 1.00000000, 1.00000000, ...} ...} ...}	physx::PxVehicleEngineData
mTorqueCurve	{mDataPairs=0x000000000483cef0 {0.000000000, 0.800000012, 0.330000013, 1.00000000, 1.00000000, 0.800000012, ...} ...}	physx::PxFixedSizeLookupTable<8>
mMOI	1.00000000	float
mPeakTorque	500.000000	float
mMaxOmega	600.000000	float
mDampingRateFullThrottle	0.150000006	float
mDampingRateZeroThrottleClutchEngaged	2.00000000	float
mDampingRateZeroThrottleClutchDisengaged	0.349999994	float
mRecipMOI	1.00000000	float
mRecipMaxOmega	0.00166666671	float
mGears	{mRatios=0x000000000483cf60 {-4.00000000, 0.000000000, 4.00000000, 2.00000000, 1.50000000, 1.10000002, ...} ...}	physx::PxVehicleGearsData
mRatios	0x000000000483cf60 {-4.00000000, 0.000000000, 4.00000000, 2.00000000, 1.50000000, 1.10000002, 1.00000000, ...}	float[32]
mFinalRatio	4.00000000	float
mNbRatios	7	unsigned int
mSwitchTime	0.500000000	float
mPad	0.000000000	float
mClutch	{mStrength=10.0000000 mAccuracyMode=eBEST_POSSIBLE (1) mEstimateIterations=5 ...}	physx::PxVehicleClutchData
mStrength	10.0000000	float
mAccuracyMode	eBEST_POSSIBLE (1)	physx::PxVehicleClutchAccuracyMode::Enum
mEstimateIterations	5	unsigned int
mPad	0x000000000483cffc ""	unsigned char[4]
mAutoBox	{mUpRatios=0x000000000483d000 {0.649999976, 0.150000006, 0.649999976, 0.649999976, 0.649999976, 0.649999976, ...} ...}	physx::PxVehicleAutoBoxData
mUpRatios	0x000000000483d000 {0.649999976, 0.150000006, 0.649999976, 0.649999976, 0.649999976, 0.649999976, ...}	float[32]
mDownRatios	0x000000000483d080 {2.00000000, 0.500000000, 0.500000000, 0.500000000, 0.500000000, 0.500000000, 0.500000000, ...}	float[32]
mDiff	{mFrontRearSplit=0.449999988 mFrontLeftRightSplit=0.500000000 mRearLeftRightSplit=0.500000000 ...}	physx::PxVehicleDifferential4WData
mAckermannGeometry	{mAccuracy=1.00000000 mFrontWidth=2.09999990 mRearWidth=2.09999990 ...}	physx::PxVehicleAckermannGeometryData

Tournez la page S.V.P.

Document technique DT10 : Transformations géométriques en coordonnées homogènes

Soit un vecteur colonne $\vec{V}$ dans l'espace à 4 dimensions et $(x, y, z, 1)$ ses coordonnées homogènes. La transformation géométrique de ce vecteur est donnée par le produit matriciel suivant :

$$\overrightarrow{V'} = [G]\vec{V}$$

où G est la matrice de transformation géométrique de dimension 4 et $\overrightarrow{V'}$ le vecteur obtenu après transformation.
Les transformations géométriques élémentaires sont la rotation, la translation et le changement d'échelle. La matrice G représente une transformation élémentaire ou une combinaison des ces dernières.

Matrice de translation :
En considérant les translations dx, dy et dz respectivement le long des axes $\vec{x}$, $\vec{y}$ et $\vec{z}$:

$$T = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrice de changement d'échelle :
En considérant les facteurs d'échelle sx, sy et sz respectivement associés aux axes $\vec{x}, \vec{y}$ et $\vec{z}$:

$$S = \begin{bmatrix} sx & 0 & 0 & 1 \\ 0 & sy & 0 & 1 \\ 0 & 0 & sz & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Matrices de rotation :
La rotation du vecteur $\vec{V}$ est donnée par le produit matriciel :

$$\overrightarrow{V'} = [R]\vec{V}$$

où R est la matrice de rotation de dimension 3 et $\overrightarrow{V'}$ le vecteur obtenu après rotation.

Matrices de rotation autour des axes principaux :

Autour de l'axe $\vec{x}$ selon un angle θ :

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Autour de l'axe $\vec{y}$ selon un angle α :

$$R_y(\alpha) = \begin{bmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Autour de l'axe $\vec{z}$ selon un angle φ :

$$R_z(\varphi) = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 & 0 \\ \sin\varphi & \cos\varphi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation autour d'un axe quelconque :

Elle s'effectue selon les étapes suivantes :

1- déplacement du repère au centre de l'objet,
2- rotation du vecteur,
3- transformation inverse de l'étape 1.

Rappel du produit vectoriel entre 2 vecteurs :

Notons $\vec{U}$ et $\vec{V}$ deux vecteurs colonnes de dimensions 3, de coordonnées respectives (u_1, u_2, u_3) et (v_1, v_2, v_3), leur produit vectoriel est donné par :

$$\vec{U} \wedge \vec{V} = \begin{pmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{pmatrix}$$

Dans la documentation OpenGL, le produit vectoriel est souvent noté $\vec{U} \times \vec{V}$

Tournez la page S.V.P.

Document technique DT11 : Librairie OpenGL

Function name : **gluLookAt**

Define a viewing transformation

C specifications

```
void gluLookAt(GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ,
GLdouble centerX, GLdouble centerY, GLdouble centerZ, GLdouble
upX, GLdouble upY,GLdouble upZ);
```

Parameters

`eyeX, eyeY, eyeZ` : **specifies the position of the eye point.**
`centerX, centerY, centerZ` : **specifies the position of the reference point.**
`upX, upY, upZ` : specifies the direction of the up vector.

Description

`gluLookAt` creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an `UP` vector.

The matrix maps the reference point to the negative $\vec{z}$ axis and the eye point to the origin. When a typical projection matrix is used, the center of the scene therefore maps to the center of the viewport. Similarly, the direction described by the `UP` vector projected onto the viewing plane is mapped to the positive $\vec{y}$ axis so that it points upward in the viewport. The `UP` vector must not be parallel to the line of sight from the eye point to the reference point.

Let $F = \begin{bmatrix} \text{centerX} - \text{eyeX} \\ \text{centerY} - \text{eyeY} \\ \text{centerZ} - \text{eyeZ} \end{bmatrix}$ and $UP = \begin{bmatrix} \text{upX} \\ \text{upY} \\ \text{upZ} \end{bmatrix}$

Then normalize as follows: $f = \frac{F}{\|F\|}$ and $UP'' = \frac{UP}{\|UP\|}$.

Finally, let $s = f \times UP''$ and $u = f \times \frac{s}{\|s\|}$

`M` is then constructed as follow:

$$M = \begin{bmatrix} \text{s}[0] & \text{s}[1] & \text{s}[2] & 0 \\ \text{u}[0] & \text{u}[1] & \text{u}[2] & 0 \\ -\text{f}[0] & -\text{f}[1] & -\text{f}[2] & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and `gluLookAt` is equivalent to :

```
glMultMatrixf(M);
glTranslated(-eyex, -eyey, -eyez);
```

Function name : **gluPerspective**

Set up a perspective projection matrix

C specifications

```
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar);
```

Parameters

`fovy` : specifies the field of view angle, in degrees, in the y direction.

`aspect` : specifies the aspect ratio that determines the field of view in the $\vec{x}$ direction. The aspect ratio is the ratio of x (width) to y (height).

`zNear` : specifies the distance from the viewer to the near clipping plane (always positive).

`zFar` **:** specifies the distance from the viewer to the far clipping plane (always positive).

Description

`gluPerspective` specifies a viewing frustum into the world coordinate system. In general, the aspect ratio in `gluPerspective` should match the aspect ratio of the associated viewport. For example, aspect = 2.0 means the viewer's angle of view is twice as wide in x as it is in y. If the viewport is twice as wide as it is tall, it displays the image without distortion.

The matrix generated by `gluPerspective` is multipled by the current matrix, just as if `glMultMatrix` were called with the generated matrix. To load the perspective matrix onto the current matrix stack instead, precede the call to `gluPerspective` with a call to `glLoadIdentity`.

Given f defined as follows:

$$f = \text{cotangent}\left(\frac{fovy}{2}\right)$$

The generated matrix is :

$$\begin{bmatrix} \frac{\text{f}}{\text{aspect}} & 0 & 0 & 0 \\ 0 & \text{f} & 0 & 0 \\ 0 & \frac{\text{zFar} + \text{zNear}}{\text{zNear} - \text{zFar}} & \frac{2 * \text{zFar} * \text{zNear}}{\text{zNear} - \text{zFar}} & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Notes

Depth buffer precision is affected by the values specified for `zNear` and `zFar`. The greater the ratio of `zFar` to `zNear` is, the less effective the depth buffer will be at distinguishing between surfaces that are near each other.

If $r = \frac{zFar}{zNear}$ roughly $\log_2(r)$ bits of depth buffer precision are lost. Because r approaches infinity as `zNear` approaches 0, `zNear` must never be set to 0.

Tournez la page S.V.P.

Document technique DT12 : Pipeline de rendu d'un `shader` executé sur les cartes graphiques

1. Mise en mémoire du buffer définissant les différents sommets de l'objet 3D avec leurs propriétés nécessaires (normales, couleurs, coordonnées de texture…) ;

2. Mise en mémoire du buffer définissant les indices des sommets permettant de créer des triangles (soit 3 x nb de triangles indices) ;

3. Transfert des buffers définis précédemment à la carte graphique ;

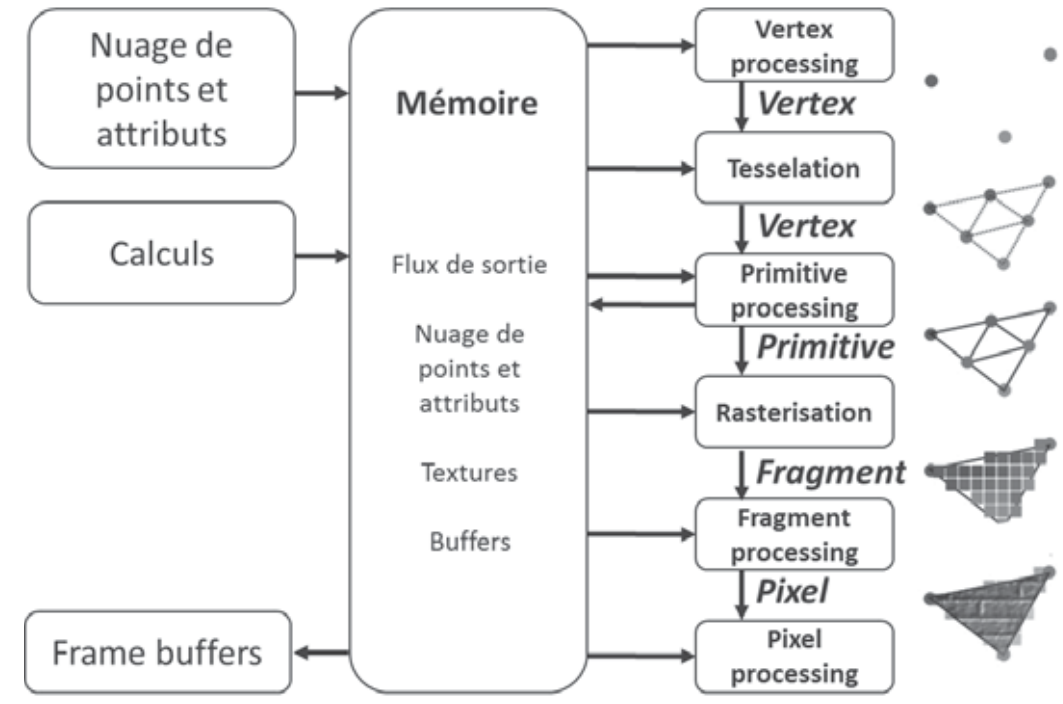
4. Compilation et déclaration d'utilisation des `shaders` à utiliser ;

5. Définition des paramètres des `shaders` (position des propriétés des sommets dans les buffers, textures à utiliser…) ;

6. Appel du lancement du rendu par la carte graphique :

a. Pour chaque sommet, utilisation du `vertex shader` : celui-ci peut récupérer des paramètres communs (variables `uniform`) à tous les sommets, et peut également récupérer les différents paramètres propres à chaque sommet définis précédemment (variables `attribute`). Son but est la projection de ce sommet sur la surface de rendu (ici, l'écran), la définition de valeurs qui seront interpolées pour chaque pixel remplissant le triangle projeté par ses 3 sommets (variables `varying` à définir dans le `shader`) ;

b. Pour chaque pixel, utilisation d'un `fragment shader` : celui-ci peut récupérer des paramètres globaux ainsi que les valeurs interpolées qui ont été définies précédemment. Son objectif est de définir la couleur à définir pour chaque pixel.



Document technique DT13 : `Fragment shader` permettant de réaliser l'effet bombé du rétroviseur grand angle

```
uniform sampler2D uDiffuse ;  // Texture à utiliser (la texture qui a été créée
par le rendu du rétroviseur)
varying vec2 vTexCoords;            // Coordonnée de texture au niveau du
fragment


void main (void)
{
    // Calcul de la distance de la coordonnée de texture par rapport au centre
de la texture
    vec2 diff=gl_TexCoord[0].xy-0.5;
    float D=diff.x*diff.x+diff.y*diff.y;

    // Calcul du ratio de déformation
    float R=1.0-exp(-D/0.05);

    // Calcul de la nouvelle coordonnée de texture
    vec2 t=(vTexCoords -0.5)*R+0.5;
    // Inversion de l'image selon l'axe horizontal
    t.x= 1.0-t.x;
    // Obtention de la couleur au niveau de la coordonnée de texture calculée
    vec4 Tex=texture2D(uDiffuse,t) ;
    // Application de cette couleur (utilisation des composantes RGB, on met la
valeur A toujours à 1 car le miroir est opaque)
    gl_FragColor=vec4(Tex.xyz,1) ;
}
```


Tournez la page S.V.P.

Document technique DT14 : Carte graphique Nvidia GeForce

Pascal GPUs are composed of different configurations of Graphics Processing Clusters (GPCs), Streaming Multiprocessors (SMs), and memory controllers. Each SM is paired with a PolyMorph Engine that handles vertex fetch, tessellation, viewport transformation, vertex attribute setup, and perspective correction. The GP104 PolyMorph Engine also includes a new Simultaneous Multi-Projection unit that will be described below. The combination of one SM plus one Polymorph Engine is referred to as a TPC.

Block Diagram of the GP104 GPU

The GeForce GTX 1080 and its GP104 GPU consist of four GPCs, twenty Pascal Streaming Multiprocessors, and eight memory controllers. In the GeForce GTX 1080, each GPC ships with a dedicated raster engine and five SMs. Each SM contains128 CUDA cores, 256KBof register file capacity, a 96KB shared memory unit, 48KB of total L1 cache storage, and eight texture units. The SM is a highly parallel multiprocessor that schedules warps (groups of 32 threads) to CUDA cores and other execution units within the SM. The SM is one of the most important hardware units within the GPU; almost all operations flow through the SM at some point in the rendering pipeline. With 20 S Ms, the GeForce GTX 1080 ships with a total of 2560 CUDA cores and 160 texture units.

The GeForce GTX 1080 features height 32-bit memory controllers (256-bit total). Tied to each 32-bit memory controller are eight ROP units and 256 KB of L2 cache. The full GP104 chip used in GTX 1080 ships with a total of 64 ROPs and 2048 KB of L2 cache.The following table provides a high-level comparison of GeForce GTX 1080 versusthe previous-generation GeForce GTX 980 GPU:

GPU	GeForce GTX 980 (Maxwell)	GeForce GTX 1080 (Pascal)
SMs	16	20
CUDA Cores	2048	2560
Base Clock	1126 MHz	1607 MHz
GPU Boost Clock	1216 MHz	1733 MHz
GFLOPs	4981[1]	8873[1]
Texture Units	128	160
Texel fill-rate	155.6 Gigatexels/sec	277.3 Gigatexels/sec
Memory Clock (Data Rate)	7,000 MHz	10,000 MHz
Memory Bandwidth	224 GB/sec	320 GB/sec
ROPs	64	64
L2 Cache Size	2048 KB	2048 KB
TDP	165 Watts	180 Watts
Transistors	5.2 billion	7.2 billion
Die Size	398 mm^2	314 mm^2
Manufacturing Process	28 nm	16 nm

Modèle CMEN-DOC v2 ©NEOPTEC

Nom de famille :
(Suivi, s'il y a lieu, du nom d'usage)

Prénom(s) :

Numéro Inscription :

(Le numéro est celui qui figure sur la convocation ou la feuille d'émargement)

Né(e) le : / /

(Remplir cette partie à l'aide de la notice)

Concours / Examen : …….. **Section/Spécialité/Série :** ……

Epreuve : …….. **Matière :** …….. **Session :** ……..

CONSIGNES

- *Remplir soigneusement, sur CHAQUE feuille officielle, la zone d'identification en MAJUSCULES.*
- *Ne pas signer la composition et ne pas y apporter de signe distinctif pouvant indiquer sa provenance.*
- *Numéroter chaque PAGE (cadre en bas à droite de la page) et placer les feuilles dans le bon sens et dans l'ordre.*
- *Rédiger avec un stylo à encre foncée (bleue ou noire) et ne pas utiliser de stylo plume à encre claire.*
- *N'effectuer aucun collage ou découpage de sujets ou de feuille officielle. Ne joindre aucun brouillon.*

DR1

Tous les documents réponses sont à rendre, même non complétés.

NE RIEN ECRIRE DANS CE CADRE

Document réponse DR1 : Simulation du comportement de la citerne

Utilisation de `scipy.integrate.odeint` (Q2)

```
#Pendule simple
import numpy
import math
import matplotlib.pyplot
from scipy.integrate import odeint
#Constantes du problème
g=9.81
l2=0.386
#Pas de temps et intervalle
tmax=10;N=5000
h=tmax/float(N-1)

t=...

def fphitheta(y,t,g,l2):
    ...




y0=...

sol=...


matplotlib.pyplot.plot(t, sol[:, 0], 'b', label='theta(t)')
```

Méthode d'Euler (Q5)

```
#Pendule simple
import numpy;import math;import matplotlib.pyplot
#Constantes du problème
g=9.81 ;l2=0.386
#Pas de temps et intervalle
tmax=10;N=5000;h=tmax/float(N-1)
#Tableaux
phi=numpy.zeros(N,float);theta=numpy.zeros(N,float)

t=...

#Conditions initiales

phi[0]= ...

theta[0]= ...
#Euler 2nd ordre

for i in range(...              ):
    ...




matplotlib.pyplot.plot(t,theta,label="angle theta")
```

Différences finies (Q10-Q11)

```
import matplotlib.pyplot;import numpy;import math
#Constantes du problème
m1=1500;m2=10250;l2=0.386;k=289393;g=9.81;b1=20000;b2=20000
#Paramètres de simulation
Npts=250;tmax=10.;h=tmax/(Npts-1)
#Définition des coefficients
A=###;B=###;C=###;D=###;E=###;F=###;M=###
G=###;H=###;I=###;J=###;K=;L=###;N=###
#Définition des matrices
M1=numpy.array([[###],[###]]);M2=numpy.array([[###],[###]])
M3=numpy.array([[###],[###]]);M4=numpy.array([[###],[###]])
#Tableaux
X=numpy.zeros((2,Npts));F=numpy.zeros((2,Npts));T=numpy.zeros(Npts)
y2=numpy.zeros(Npts)
F1=60000;F2=400000
F[0,0]=F1;F[1,0]=F2;F[0,1]=F1;F[1,1]=F2;F[0,2]=F1;F[1,2]=F2
M1inv=numpy.linalg.inv(M1)
#Boucle de calcul

for i in range(...

    F[0,i]=F1;F[1,i]=F2
    T[i]=i*h

    X[:,i+1]= ...


for i in range(Npts):
    y2[i]=X[0,i]+l2*math.sin(X[1,i])

matplotlib.pyplot.plot(T[:Npts-1],X[0,:Npts-1])
matplotlib.pyplot.plot(T[:Npts-1],X[1,:Npts-1])
matplotlib.pyplot.plot(T[:Npts-1],y2[:Npts-1])
```


Tournez la page S.V.P.

Utilisation de `scipy.integrate.odeint` (Q14-Q15)

```
import matplotlib.pyplot
import numpy
import math
import scipy.integrate

#Constantes du problème
m1=1500;m2=10250;l2=0.386;k=289393;g=9.81;b1=20000;b2=20000
#Paramètres de simulation
Npts=250;tmax=10.;h=tmax/(Npts-1)
#Définition des matrices

M5=numpy.array([...

M6=numpy.array([...

M7=numpy.zeros(4)
M7[2]=F1;M7[3]=F2

t=...

M5inv=...

def fdxdt(X1,t,Fo):

    ......................=X1

    dXdt=…

    return dXdt


x0=[...

sol=scipy.integrate.odeint(...


matplotlib.pyplot.figure(0)
matplotlib.pyplot.plot(t,sol[:,0])
matplotlib.pyplot.figure(1)
matplotlib.pyplot.plot(t,sol[:,1])
matplotlib.pyplot.figure(2)
matplotlib.pyplot.plot(t,y2)
```

Modèle CMEN-DOC v2 ©NEOPTEC

Nom de famille :
(Suivi, s'il y a lieu, du nom d'usage)

Prénom(s) :

Numéro Inscription :

(Le numéro est celui qui figure sur la convocation ou la feuille d'émargement)

Né(e) le : / /

(Remplir cette partie à l'aide de la notice)

Concours / Examen : …….. **Section/Spécialité/Série :** ……

Epreuve : …….. **Matière :** …….. **Session :** ……..

CONSIGNES

- *Remplir soigneusement, sur CHAQUE feuille officielle, la zone d'identification en MAJUSCULES.*
- *Ne pas signer la composition et ne pas y apporter de signe distinctif pouvant indiquer sa provenance.*
- *Numéroter chaque PAGE (cadre en bas à droite de la page) et placer les feuilles dans le bon sens et dans l'ordre.*
- *Rédiger avec un stylo à encre foncée (bleue ou noire) et ne pas utiliser de stylo plume à encre claire.*
- *N'effectuer aucun collage ou découpage de sujets ou de feuille officielle. Ne joindre aucun brouillon.*

DR2

Tous les documents réponses sont à rendre, même non complétés.

Tournez la page S.V.P.

NE RIEN ECRIRE DANS CE CADRE

Document réponse DR2 : Mise en place de la perspective et de la vue

(Seules les lignes marquées d'une flèche sont à compléter)

```
// Création de la matrice de perspective
double zFar =  100;     // Plan éloigné (100m)
double zNear = 0.1;     // Plan  proche (10 cm)
double pRes[16];
// Définition de l'angle d'ouverture en radian

double theta =

double cotantheta2 = 1.0/tan(theta/2);
// Remplissage de la matrice de projection qui doit être remplie colonne par
colonne, et non pas ligne par ligne (norme OpenGL)
pRes[0] =

pRes[1] =

pRes[2] =

pRes[3] =

pRes[4] =

pRes[5] =

pRes[6] =

pRes[7] =

pRes[8] =

pRes[9] =

pRes[10] =

pRes[11] =

pRes[12] =

pRes[13] =

pRes[14] =

pRes[15] =
```

```
// Utilisation de la matrice de projection
glMatrixMode( GL_PROJECTION ) ;
glLoadIdentity();
glMultMatrixd(pRes);

// Angle de rotation en degré de la caméra autour de l'axe Y vertical au sol
double angleY = 162.11;
// Angle de rotation autour à l'axe X pour orienter la caméra vers le sol
double angleX = 25;

// Coordonnée de translation en mètres de l'origine à la position de du
rétroviseur
double X = 0.2; // décalage du rétroviseur par rapport au flanc: 20cm
double Y = 2.15; // hauteur du rétroviseur sur le camion : 2m15
double Z = 0;
// Les angles de rotations et distances de translations doivent être négatifs
(car cela concerne une translation des sommets vers la caméra, et non une
transformation de la caméra elle-même)
double* pTransfo1 = GetTranslationGLMatrix(-X, -Y, -Z);
double* pTransfo2 = GetRotationGLMatrix(AXIS_X, - angleX);
double* pTransfo3 = GetRotationGLMatrix(AXIS_Y, - angleY);

// Multiplication de la matrice de projection par les matrices de mise en place
de la vue

glMultMatrixd(


glMultMatrixd(pTransfo2);


glMultMatrixd(


// Utilisation de la matrice de transformation du modèle pour les opérations
suivantes de positionnement des objets 3D
glMatrixMode( GL_MODELVIEW )
```