Épreuve d’admissibilité de modélisation d'un système, d'un procédé ou d'une organisation

A. Présentation de l’épreuve

Arrêté du 19 avril 2013 et arrêté du 19 avril 2016

- Durée totale de l'épreuve : 6 heures
- Coefficient 1

L'épreuve est spécifique à l'option choisie.

À partir d'un dossier technique comportant les éléments nécessaires à l'étude, l'épreuve a pour objectif de vérifier que le candidat est capable de synthétiser ses connaissances pour modéliser un système technique dans le domaine de la spécialité du concours dans l'option choisie en vue de prédire ou de vérifier son comportement et ses performances.

B. Sujet

Le sujet est disponible en téléchargement sur le site du ministère à l’adresse :

http://media.devenirenseignant.gouv.fr/file/agregation_externe/87/9/s2019_agreg_externe_sii_informatique_2_1_1093879.pdf

http://media.devenirenseignant.gouv.fr/file/agregation_externe/88/0/s2019_agreg_externe_sii_informatique_2_2_1093880.pdf

Ce sujet porte sur un simulateur 3D de conduite d’un camion-citerne. Les thématiques principalement abordées sont :

- la modélisation numérique d’un système physique décrit par des équations différentielles ;
- l’analyse des principes de programmation d’un moteur physique 3D ;
- la modélisation d’une scène en 3 dimensions.

C. Éléments de correction

Q1 Système différentiel du modèle simplifié

$$\begin{cases} \dfrac{d\theta}{dt} = \varphi \\ \dfrac{d\varphi}{dt} = \dfrac{-g}{l_2}\sin\theta \end{cases}$$

Q2 Résolution avec le module `odeint` de `Scipy`

```
#pendule simple odeint
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
b=10;N=5000
h=b/float(N-1)
t=numpy.linspace(0,b,N)

def fphitheta(y,t,g,l2):
    theta,phi=y
    dydt=[phi,(-g/l2)*math.sin(theta)]
    return dydt

y0=[0.1,0.0]
#appel odeint avec paramètres
sol=odeint(fphitheta,y0,t,args=(g, l2))
matplotlib.pyplot.plot(t, sol[:, 0], 'b', label='theta(t)')
```

Q3 Schéma d'Euler explicite

$$y_{i+1} = y_i + h \times F(t_i, y_i)$$

Q4 Application au pendule simple

$$\begin{cases} \theta_{i+1} = \theta_i + h \times \varphi_i \\ \varphi_{i+1} = \varphi_i - h \times \dfrac{g}{l_2}\sin\theta_i \end{cases}$$

Q5 Programme `Python` de la méthode d'Euler explicite

```
#pendule simple Euler explicite
import numpy ;import math;import matplotlib.pyplot
#Constantes du problème
g=9.81;l2=0.386
#Pas de temps et intervalle
b=10;N=5000
h=b/float(N-1)
#Tableaux
phi=numpy.zeros(N,float);theta=numpy.zeros(N,float)
t=numpy.linspace(0,b,N)
#Conditions initiales
phi[0]=0.0
theta[0]=0.1
```

```
#Euler 2nd ordre
for i in range(N-1):
    theta[i+1]=theta[i]+h*phi[i]
    phi[i+1]=phi[i]+h*(-g/l2)*math.sin(theta[i])

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

Q6 Comparaison Euler explicite et `odeint`

Pour N=5 000 points, la méthode `odeint` converge vers la solution car l'angle θ oscille entre deux valeurs correspondant à la condition initiale.
Pour N=5 000 points, la méthode d'Euler explicite diverge car l'angle θ oscille entre deux valeurs qui croissent dans le temps.

Conclusion : la stabilité de la méthode d'Euler dépend des valeurs des constantes du problème à nombre de points de calcul imposé.

Autres méthodes : le problème différentiel semble être un problème dit « raide ». Ce genre de problème différentiel est résolu en utilisant des méthodes rétrogrades (« Backward Differential Formula ») comme la méthode d'Euler implicite, Runge Kutta ordre 4 implicite et autres…

Q7 Modèle complet

$$(m_1+m_2)\frac{d^2y_1}{dt^2}+m_2l_2\frac{d^2\theta}{dt^2}+ky_1+\beta_1\frac{dy_1}{dt}=f_1(t)$$
$$m_2l_2\frac{d\varphi}{dt}+m_2\frac{d\omega}{dt}+m_2g\theta+\beta_2\varphi=f_2(t)$$

Q8 Différences finies : coefficients $\boldsymbol{A, B, C, D, E, F}$ **et** $\boldsymbol{M}$

$$\begin{cases} A=m_1+m_2+\beta_1 \mathrm{h} \\ B=m_2l_2 \\ C=-(m_1+m_2) \\ D=-m_2l_2 \\ \mathrm{E}=2(m_1+m_2)+\beta_1\mathrm{h}+\mathrm{kh}^2 \\ F=2m_2l_2 \\ \mathrm{M}=\mathrm{h}^2 \end{cases}$$

Q9 Différences finies : matrices $\boldsymbol{M_1, M_2, M_3}$ **et** $\boldsymbol{M_4}$

$$M_1=\begin{bmatrix} A & B \\ G & H \end{bmatrix}, M_2=\begin{bmatrix} E & F \\ K & L \end{bmatrix}, M_3=\begin{bmatrix} C & D \\ I & J \end{bmatrix}, M_4=\begin{bmatrix} M & 0 \\ 0 & N \end{bmatrix}$$

Q10 Différences finies : programme Python

```
import matplotlib.pyplot;import numpy;import math

#Constantes du problème
m1=1500;m2=10250;l2=0.39;k=300000;g=9.81;l=0.33;b1=20000;b2=20000
#Paramètres de simulation
Npts=10000;tmax=10.;h=tmax/(Npts-1)
#Définition des coefficients
```

```
A=m1+m2+h*b1;B=m2*l2;C=-(m1+m2);D=-m2*l2;E=2*(m1+m2)-
k*h*h+h*b1;F=2*m2*l2;M=h*h
G=m2;H=m2*l2+h*b2;I=-m2;J=-m2*l2;K=2*m2;L=m2*(2*l2-h*h*g)+h*b2;N=h*h
#Définition des matrices
M1=numpy.array([[A,B],[G,H]]);M2=numpy.array([[E,F],[K,L]])
M3=numpy.array([[C,D],[I,J]]);M4=numpy.array([[M,0],[0,N]])
#Vecteurs colonne
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(3,Npts-1):
    F[0,i]=F1;F[1,i]=F2
    T[i]=i*h

    X[:,i+1]=numpy.dot(M1inv,numpy.dot(M2,X[:,i])+numpy.dot(M3,X[:,i-
1])+numpy.dot(M4,F[:,i]))

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])
```

Q11 Déplacement de la masse $\boldsymbol{m_2}$

$$y_2(t) = y_1(t) + l_2 \sin\theta(t)$$

Q12 Modèle complet, approximation linéaire : conditionnement pour la méthode `odeint`

$$\frac{dy_1}{dt} = \omega$$
$$\frac{d\theta}{dt} = \varphi$$
$$(m_1 + m_2)\frac{d\omega}{dt} + m_2 l_2 \frac{d\varphi}{dt} + k y_1 + \beta_1 \omega = f_1(t)$$
$$m_2 l_2 \frac{d\varphi}{dt} + m_2 \frac{d\omega}{dt} + m_2 g\theta + \beta_2 \varphi = f_2(t)$$

Q13 Modèle complet, approximation linéaire : expression des matrices $\boldsymbol{M_5, M_6\ et\ M_7}$

$$M_5 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & m_1 + m_2 & m_2 l_2 \\ 0 & 0 & m_2 & m_2 l_2 \end{bmatrix}, M_6 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -k & 0 & -\beta_1 & 0 \\ 0 & 0 & -m_2 g & -\beta_2 \end{bmatrix}, M_7 = \begin{bmatrix} 0 \\ 0 \\ f_1 \\ f_2 \end{bmatrix}$$

Q14 Expression premier élément de `dxdt`

```
dxdt[0]=numpy.dot(M5inv,numpy.dot(M6,X1)+Fo)[0]
```
ou
```
dxdt[0]=numpy.dot(M5inv,numpy.dot(M6,X1)+M7)[0]
```

Q15 Modèle complet, méthode `odeint` : programme `Python`

```
#Constantes du problème
m1=1500;m2=10250;l2=0.39;k=300000;g=9.81;b1=20000;b2=20000
#Paramètres de simulation
Npts=1000;tmax=10.;h=tmax/(Npts-1)
#Définition des matrices
M5=numpy.array([[1,0,0,0],[0,1,0,0],[0,0,m1+m2,m2*l2],[0,0,m2,m2*l2]])
M6=numpy.array([[0,0,1,0],[0,0,0,1],[-k,0,-b1,0],[0,-m2*g,0,-b2]])
M7=numpy.zeros(4)
t=numpy.linspace(0,tmax,Npts)
y2=numpy.zeros(Npts)
F1=60000;F2=400000
M7[2]=F1;M7[3]=F2

M5inv=numpy.linalg.inv(M5)

def fdxdt(X1,t,Fo,a):
    y1,theta,w,phi=X1
    dXdt=[numpy.dot(M5inv,numpy.dot(M6,X1)+Fo)[0],
          numpy.dot(M5inv,numpy.dot(M6,X1)+Fo)[1],
          numpy.dot(M5inv,numpy.dot(M6,X1)+Fo)[2],
          numpy.dot(M5inv,numpy.dot(M6,X1)+Fo)[3]]
    return dXdt

x0=[0.0,0.0,0.0,0.0]
sol=scipy.integrate.odeint(fdxdt,x0,t,args=(M7,a))

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

Q16 Comparaison méthodes numériques

Figure 11, N=250

- RK4 et `odeint` sont superposés ;
- Euler explicite diverge ;
- différences finies proche d'`odeint`.

Figure 12, N=150

- RK4 diverge ;
- `odeint` similaire à la simulation à N=250 ;
- différences finies proche d'`odeint`.

Conclusion :
Si `odeint` existe en C++, choisir la bibliothèque concernée (`odeint` C++ ou `Boost odeint` C++ par exemple),
Si `odeint` n'existe pas en C++, réaliser une interface (*wrapper*) entre le module `Python` et le module C++,
Sinon implémenter les différences finies avec une bibliothèque C++ pour les matrices.
Dernière solution, implémenter en C++ les algorithmes d'`odeint` de Python qui utilisent le sous module `LSODA` du module `ODEPACK` de `FORTRAN` (algorithme d'Adams pour les problèmes non raides et BDF pour les problèmes raides).

Q17 Comparaison relation de composition et d'association

- listing1 → relation de composition ;
- listing2 → relation d'association unidirectionnelle.

- objet1→ instanciation de classe1 ;
- objet2→ instanciation de classe2.

Type de relation	durée de vie	accès membres privés d'objet1 par objet2	accès membres publics d'objet1 par objet2	partage d'objet2 par 2 objets de classe1
composition	objet2 a la même durée de vie qu'objet1. Si objet1 est détruit, objet2 est détruit aussi.	Non permis	Oui	Non, car relation exclusive d'appartenance.
association	objet1 et objet2 ont des durées de vie indépendantes	Non permis	Oui	Oui. Un objet2 peut être accessible par plusieurs objets de classe1.

Q18 Rôle des classes du DT2

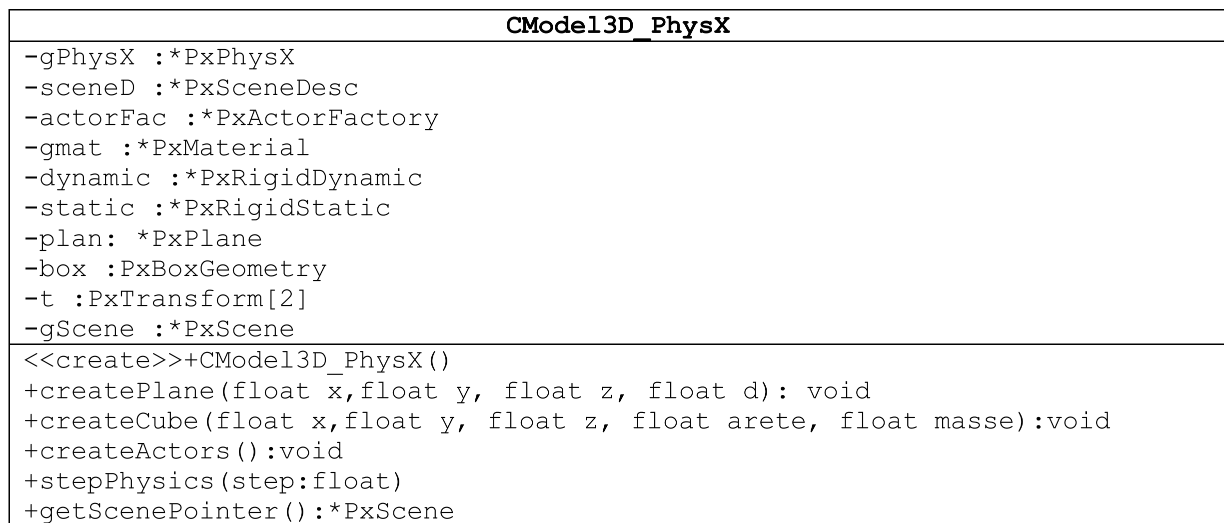
À la création de l'objet de la classe `CRendu_3D_2D_OpenGL`, un objet de classe `CModel3D_PhysX` est créé.
À la création de l'objet de la classe `CModel3D_PhysX`, un objet de classe `PxPhysX` est créé.
L'appel de la méthode `createScene` de la classe `PxPhysX` contenue dans la classe `CModel3D_PhysX` permet de créer la scène.
L'appel de la méthode `createActors` de la classe `CModel3D_PhysX` permet de créer les acteurs de la scène. Cette dernière contient les adresses des acteurs des classes `PxRigidDynamic` et `PxRigidStatic`.
La méthode `rendersActor3D` de la classe `CRendu_3D_2D_OpenGL` assure le rendu 3D.

Q19 Relation de composition et programme principal de l'application de base

L'objet « rendu » de la classe `CRendu_3D_2D_OpenGL` contient l'objet `mod3d` de la classe `CModel3D_PhysX`.

```
void main()
{
	CRendu_3D_2D_OpenGL rendu ;
}
```

Q20 Diagramme UML de la classe `CModel3D_PhysX`



```
CModel3D_PhysX
-gPhysX :*PxPhysX
-sceneD :*PxSceneDesc
-actorFac :*PxActorFactory
-gmat :*PxMaterial
-dynamic :*PxRigidDynamic
-static :*PxRigidStatic
-plan: *PxPlane
-box :PxBoxGeometry
-t :PxTransform[2]
-gScene :*PxScene
<<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
```

Q21 Deuxième objet créé dans l'application de base

a) le deuxième objet créé découle de l'instanciation de la classe `CModel3D_PhysX`.

b) les objets créés directement dans son constructeur sont issus des classes :

- `PxPhysX` ;
- `PxSceneDesc`.

c) les méthodes appelées de cet objet sont :

- `createActors` qui permet de créer le cube (instanciation de la classe `PxRigidDynamic`) via la méthode « createCube » et le plan (instanciation de la classe `PxRigidStatic`) via la méthode `createPlane` ;
- `stepPhysX` qui permet de lancer la simulation à chaque pas de temps (step) ;
- `getScenePointer` qui permet de récupérer l'adresse de la scène pour le rendu 3D effectué par l'objet `CRendu_3D_2D_OpenGL`.

Q22 Hiérarchie de classes

a) la classe au sommet de la hiérarchie est la classe `PxBase` qui est dérivée pour obtenir la classe `PxActor` qui à son tour est dérivée pour obtenir la classe `PxRigidActor`.
`PxBase←PxActor←PxRigidActor`

La dérivation de la classe `PxRigigActor` produit deux classes filles : `PxRigidStatic` et `PxRigidBody`.
`PxRigidActor←PxRigidStatic`
`PxRigidActor←PxRigidBody`

La classe `PxRigidBody` est dérivée pour produire la classe fille `PxRigidDynamic`.
`PxRigidBody← PxRigidDynamic`

b) l'application de base utilise les classes `PxRigidStatic` et `PxRigidDynamic`.

Q23 Principe de construction des objets 3D

a) l'instanciation des classes `PxRigidBody` et `PxRigidDynamic` est impossible car ce sont des classes abstraites. Pour `PxRigidStatic`, le constructeur est protégé, on ne peut pas l'instancier.
b) la dérivation de ces classes est impossible car les constructeurs sont protégés.
c) les instanciations sont réalisées par la classe `PxActorFactory` et ses méthodes associées à chaque type d'acteur.
L'appel de sa méthode `PxCreateDynamic` retourne l'adresse d'un objet du type `PxRigidDynamic`.
L'appel de sa méthode `PxCreateStatic` retourne l'adresse d'un objet du type `PxRigidStatic`.
d) l'intérêt est de faciliter la construction des objets partiellement paramétrés par le constructeur d'objet (Factory). C'est l'un des `design pattern` utilisé en programmation orientée objet.

Q24 Méthode `addActors`

Le prototype de cette méthode s'écrit :

`addActors(PxActors *actor)` : en vertu du polymorphisme en programmation orientée objet, toutes les classes dérivées de la classe `PxActors` peuvent être passées en paramètre de cette méthode.
La variable `*actor` est un alors considérée comme un pointeur polymorphe.
Les deux acteurs sont le cube (PxRigidDynamic) et le plan (PxRigidStatic).

Q25 Méthode createCube et createActors

En conservant la méthode `createCube`, la méthode `createActors` devient :

```
void CModel3D_PhysX::createActors()
{
	this->createPlane(0, 1, 0, 0);
	float hmax = 24, hmin = 2;
	unsigned char i,N=50;
	for (i=0;i<N; i++)
	{
		this->createCube(-2.,(float)((hmax-hmin)*i)/float(N-
1)+hmin,3.,0.10);
	}
}
```

Q26 Méthodes définissant la position et l'orientation d'un objet 3D

a) à la création des objets avec les méthodes :

`PxCreateDynamic(…) et PxCreateStatic(…)`

Avec le mutateur de la classe mère `PxRigidBody` :

`setGlobalPose(…)`

b) un objet de la classe `PxTransform` est composée :
- d'un objet de la classe `PxVec3` permettant de définir la position dans l'espace d'un objet 3D ;
- d'un objet de la classe `PxQuat` permettant de définir l'orientation dans l'espace d'un objet 3D.

Q27 Quaternions dans `PhysX`

a) produit d'un quaternion avec son conjugué
D'après le produit de deux quaternions :

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

En considérant :
$q = (\omega + \vec{v})$ et $\bar{q} = (\omega - \vec{v})$

Le produit devient :
$q\bar{q} = (\omega^2 + \vec{v} \cdot \vec{v}, -\omega\vec{v} + \omega\vec{v} - \vec{v} \times \vec{v})$

Or
$-\omega\vec{v} + \omega\vec{v} - \vec{v} \times \vec{v} = \vec{0}$ et $\vec{v} \cdot \vec{v} = x^2 + y^2 + z^2$

Par conséquent :
$q\bar{q} = \omega^2 + x^2 + y^2 + z^2 = |q|^2 = 1$

b) calcul du quaternion pour $\vec{n} = \frac{1}{\sqrt{2}}\vec{\imath} + \frac{1}{\sqrt{2}}\vec{\jmath}$ et $\theta = \frac{\pi}{4}$

$$q = \cos\frac{\theta}{2} + \left(\sin\frac{\theta}{2}\right)\left(\frac{1}{\sqrt{2}}\vec{\imath} + \frac{1}{\sqrt{2}}\vec{\jmath}\right) = 0.928 + 0.27\vec{\imath} + 0.27\vec{\jmath} + 0$$

Q28 Utilisation des quaternions et des matrices de rotation

Les quaternions sont utilisés pour les calculs d'orientation des objets dans `PhysX` car plus intuitif pour le développeur.
`OpenGL` utilise les matrices de rotation pour réaliser le rendu 3D.

Q29 Conversion quaternion vers matrice

La classe `PxMat44` avec son constructeur `PxMat44(PxTransform)`.

Q30 Relation physique-programmation
a) relations entre variables et méthodes C++

m	`setMass, getMass`
$\vec{v}$	`setLinearVelocity, getLinearVelocity`
$\bar{\bar{J}}$	`setMassSpaceInertia, getMassSpaceInertia`
$\vec{T}$	`addTorque`
$\vec{\omega}$	`setAngularVelocity, getAngularVelocity`
$\vec{F}$	`addForce`

b) frottement fluide linéaire
La méthode `setLinearDamping` permet d'ajouter une force de frottement proportionnel à la vitesse linéaire. Le modèle du corps rigide indéformable contient nativement cette force de frottement.

c) frottement fluide quadratique
Cette force n'est pas implémentée dans le modèle du corps rigide indéformable.
Grâce à la méthode `getLinearVelocity`, la vitesse à chaque pas de temps est accessible en valeur.
Il faut utiliser la méthode `addForce` et passer en paramètre une expression proportionnelle à la vitesse linéaire au carré.
Par exemple, en considérant un objet `rD` de la classe `PxRigidDynamic` :

```
double coef=0.05;
rD.addForce(coef*rD.getAngularVelocity*rD.getAngularVelocity);
```

Q31 Modèle de traction

a) relations

Moteur (engine)	`PxVehicleEngineData`
Embrayage (clutch)	`PxVehicleClutchData`
Boîte de vitesse (gearbox)	`PxVehicleGearsData`
Différentiel	`PxVehicleDifferential4WData`

Roues	`PxVehicleWheelsSimData`
	`PxVehicleWheelData`
	`PxVehicleTireData`

b) le constructeur de la classe `PxVehicleDrive4W` est protégé. Il n'est pas possible d'instancier cette classe. La solution est d'utiliser la méthode statique `create` de cette classe. L'appel s'effectue avec l'opérateur de résolution de portée (`::`) :
`PxVehicleDrive4W::create(…)`

Les classes nécessaires sont :

- `PxPhysics ;`
- `PxRigidDynamic ;`
- `PxVehicleWheelsSimData ;`
- `PxVehicleDriveSimData4W.`

Q32 Relation d'amitié

L'instruction ci-dessous est suffisante pour définir l'amitié entre deux classes. Elle doit être écrite dans le fichier de définition de la classe acceptant l'amitié (`classe2`).

```
friend class Classe1;
```

Q33 Propriétés de l'amitié

Cette relation n'est pas bidirectionnelle : si `classe1` est amie avec `classe2`, la réciproque n'est pas vraie.
Cette relation n'est pas transitive : si `classe1` est amie avec `classe2` et `classe2` est amie `classe3`, `classe1` n'est pas amie avec `classe3`.
En cas d'héritage, l'amitié n'est pas héritée. En effet, l'amitié est déclarée explicitement dans la classe qui reçoit cette amitié (`classe2`) en précisant le nom de la classe, i.e., `classe1`. Si une classe `classe4` hérite de `classe1`, la définition de `classe2` reste inchangée, `classe4` ne devient pas amie avec `classe2`.

Q34 Création du véhicule

a) en utilisant la méthode statique `create` de la classe `PxVehicleDrive4W` :

```
PxVehicleDrive4W
*vehDrive4W=PxVehicleDrive4W::create(physics,veh4WActor,wheelsSimData,driveSi
mData,nbNonDrivenWheels);
```

b) assignation vitesse maximale du moteur
Les données membres de la classe `PxVehicleEngineData` sont publiques. La modification s'effectue par affectation directe :

```
engine.mMaxOmega=628;
```

Le membre `mDriveSimData` de la classe `PxVehicleDrive4W` est public. L'accès en lecture/écriture est autorisé.
Cet objet est du type `PxVehicleDriveSimData4W`.
Par héritage, cet objet possède la méthode `setEngineData` de la classe mère `PxVehicleDriveSimData` qui prend comme paramètre un objet du type `PxVehicleEngineData`.
L'instruction permettant l'assignation de la nouvelle vitesse maximale s'exprime selon l'expression suivante :

```
vehicleDrive4W→mDriveSimData.setEngineData(engine) ;
```

c) héritage de l'amitié

- déclaration du membre `mEngine` (`protected`) dans la classe mère du type de la classe acceptant l'amitié (`PxVehicleEngineData`). Ce qui permet à la classe fille de contenir ce membre également. Les méthodes de la classe fille peuvent modifier ce membre ;

- définition d'une méthode publique `setEngineData` de la classe mère permettant de modifier l'objet de la classe amie (`PxVehicleEngineData`). La classe fille appelle cette méthode pour modifier le membre du type de la classe amie avec la classe mère ;

- la relation d'amitié permet de mettre en relation des objets dont les durées de vie peuvent être différentes. Il suffit simplement de déclarer la classe amie dans la définition de la classe acceptant l'amitié.

Cependant, en cas d'héritage, il sera nécessaire de créer par des relations de composition les membres acceptant l'amitié dans la classe mère. Ce qui entraînera une architecture logicielle complexe et très peu modulable.

Q35 Analyse de la fenêtre de debug

a) hiérarchies de classes

- la première : `PxBase`, `PxVehicleWheels`, `PxVehicleDrive` et `PxVehicleDrive4W` ;
- la seconde : `PxVehicleDriveSimData`, `PxVehicleDriveSimData4W` ;
- la troisième (non visible mais implicite) : `PxActor`, `PxRigidActor`, `PxRigidBody`, `PxRigidDynamic`.

b) cohérence des résultats
La fenêtre de debug montre le contenu de l'objet `gVehicle4W` de la classe `PxVehicleDrive4W`.

Par héritage, cet objet contient les membres `mWheelsSimData`, `mWheelsDynData`, `mActor`, `mDriveDynData` respectivement associés aux classes `PxVehicleWheelsSimData`, `PxVehicleWheelsDynData`, `PxRigidDynamic` et `PxVehicleDriveDynData`.

Le membre `mDriveSimData` apparaît conformément au diagramme de classes.
Ce dernier contient les objets des classes `PxVehicleEngineData`, `PxVehicleGearsData` et `PxVehicleClutchData` par héritage.

Par ailleurs, il contient par amitié les objets des classes `PxVehicleDifferential4WData` et `PxVehicleAckermannGeometryData`.

c) différences avec le diagramme de classes

- la classe `PxVehicleAutoBoxData` n'est pas représentée dans le diagramme UML ;
- la classe `PxVehicleDriveDynData` amie de la classe `PxVehicleDrive` n'est pas représentée dans le diagramme UML ;
- la classe `PxVehicleWheelsDynData` amie de la classe `PxVehicleWheels` n'est pas représentée dans le diagramme UML ;
- les objets des classes `PxVehicleAntiRollBarData`, `PxVehicleTireData`, `PxVehicleSuspensionData` et `PxVehicleWheelData` n'apparaissent pas dans la fenêtre de debug car elles sont contenues dans le membre `mWheelsSimData` ;
- l'amitié entre `PxVehicleDrive4W` et `PxVehicleWheelsSimData` n'apparaît pas dans la fenêtre de debug.

Q36 Matrices de rotation et de translation

Une solution possible peut être une rotation de 90° dans le sens trigonométrique autour de l'axe y suivie d'une translation de (-0,2 ; 0 ; 0,4).

La matrice de rotation s'exprime :

$$[R] = R_y(90) = \begin{bmatrix} \cos 90 & 0 & \sin 90 & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 90 & 0 & \cos 90 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

La matrice de translation s'exprime :

$$[T] = \begin{bmatrix} 1 & 0 & 0 & -0{,}2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0{,}4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Q37 Calcul de la transformation du point A

$$[G] = [T][R] = \begin{bmatrix} 1 & 0 & 0 & -0{,}2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0{,}4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & -0{,}2 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0{,}4 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$[G]A = \begin{bmatrix} 0 & 0 & 1 & -0{,}2 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0{,}4 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0{,}1 \\ 0 \\ 0{,}1 \\ 1 \end{bmatrix} = \begin{bmatrix} -0{,}1 \\ 0 \\ 0{,}3 \\ 1 \end{bmatrix}$$

La transformation du point A fonctionne et est correcte.

Q38 Objets centrés sur l'origine

De manière générale, les objets sont créés individuellement, centrés à l'origine, et sont ensuite correctement placés (translation / rotation / taille) par la matrice de transformation, ce qui permet par exemple de créer plusieurs instances du même objet à des positions différentes, en utilisant un seul et même objet 3D.

Q39 Positionnement de la caméra

gluLookAt(0,0,0,0,0,1,0,1,0);

Q40 Coefficients de la matrice de point de vue

$$F = \begin{bmatrix} \text{centerX} - \text{eyeX} \\ \text{centerY} - \text{eyeY} \\ \text{centerZ} - \text{eyeZ} \end{bmatrix} = \begin{bmatrix} 0-0 \\ 0-0 \\ 1-0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$f = \frac{F}{||F||} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$UP = \begin{bmatrix} \text{upX} \\ \text{upY} \\ \text{upZ} \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$UP'' = \frac{UP}{||UP||} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

$$s = f \ \times UP'' = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$$

$$u = f \times \frac{s}{||s||} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix}$$

Q41 Matrice de point de vue

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

Q42 Point A et matrice de point de vue

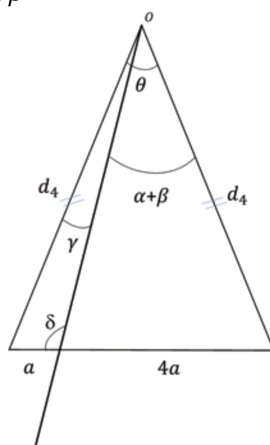
$$A' = [M]A = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -0{,}1 \\ 0 \\ 0{,}3 \\ 1 \end{bmatrix} = \begin{bmatrix} 0{,}1 \\ 0 \\ -0{,}3 \\ 1 \end{bmatrix}$$

Q43 Angles $\boldsymbol{\alpha}$ et $\boldsymbol{\beta}$

$$\alpha = arctan\frac{d_3}{d_2} = arctan\frac{9{,}1}{8{,}5} = 46{,}95\ °$$

$$\beta = \ arctan\frac{d_1}{d_2} = arctan\frac{0{,}2}{8{,}5} = 1{,}35\ °$$

Q44 Expression de γ en fonction de α et β



Nous nous intéressons au triangle isocèle ou α+β est connu et fixé à $46{,}95 + 1{,}35 = 48{,}3$°.

Par la loi des sinus, on obtient

$$\frac{d_4}{\sin(\delta)} = \frac{a}{\sin(\gamma)}$$

Dans l'autre triangle, par cette même loi :

$$\frac{4a}{\sin(\alpha+\beta)} = \frac{d_4}{\sin(\pi-\delta)}$$

Par composition, substitution de d_4:

$$d_4 = \frac{a\sin(\delta)}{\sin(\gamma)} = \frac{4a\sin(\pi-\delta)}{\sin(\alpha+\beta)}$$

En simplifiant par$\boldsymbol{a}$, il reste :

$$\frac{\sin(\delta)}{\sin(\gamma)} = \frac{4\sin(\pi-\delta)}{\sin(\alpha+\beta)}$$

D'où

$$\sin(\gamma) = \frac{\sin(\alpha+\beta)\sin(\delta)}{4\sin(\pi-\delta)} = \frac{\sin(\alpha+\beta)}{4}$$

Il vient :

$$\gamma = \arcsin\left(\frac{\sin(\alpha+\beta)}{4}\right)$$

Q45 Valeur de $\boldsymbol{\gamma}$ et $\boldsymbol{\theta}$

$$\gamma = \arcsin\left(\frac{\sin(\alpha+\beta)}{4}\right) = 10{,}76°$$

$\theta = \alpha + \beta + \gamma = 46{,}95 + 1{,}35 + 10{,}76 = 59{,}06°$

Q46 Rôle des distances proche et lointaine

Le fait d'imposer une distance proche et une distance lointaine facilite la transformation mathématique de la perspective car les 2 plans parallèles formés délimitent un solide fini (tronc géométrique). Ce solide de 8 sommets avec 2 plans parallèles est beaucoup plus simple à transformer en un cube normalisé qu'un volume non borné.

De plus, la définition d'une distance lointaine permet d'éliminer instantanément tous les objets trop lointains de la scène, ce qui permet d'accélérer les calculs.
Enfin, la distance proche permet d'éviter des problèmes de précision de perspective qui pourraient intervenir en divisant par des coordonnées trop proches de l'origine.

Q47 Programmation opengl : matrice de perspective

```
// 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 = 60*3.14156/180;
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] = cotantheta2 ;
pRes[1] = 0;
pRes[2] = 0;
pRes[3] = 0;
pRes[4] = 0;
pRes[5] = cotantheta2 ;
pRes[6] = 0;
pRes[7] = 0;
pRes[8] = 0;
pRes[9] = 0;
pRes[10] = (zFar +zNear)/ ( zNear - zFar) ;
pRes[11] = -1;
pRes[12] = 0;
pRes[13] = 0;
pRes[14] = (2 * zFar * zNear)/ (zNear - zFar) ;
pRes[15] = 0 ;
// 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 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(pTransfo1);
glMultMatrixd(pTransfo2);
glMultMatrixd(pTransfo3);
// Utilisation de la matrice de transformation du modèle pour les opérations suivantes de positionnement des objets 3D
glMatrixMode( GL_MODELVIEW )
```

Q48 Principe général des textures

Les textures sont des images que l'on va plaquer sur la surface d'un objet. Ces images sont composées de "pixels".
Pour pouvoir les plaquer sur les objets 3D, chaque sommet du maillage 3D de l'objet contient des coordonnées de texture, qui indiquent quel pixel appliquer sur l'image.
Ainsi, lors du rendu de l'objet 3D à l'écran, chacun des sommets de l'objet se voit attribuer un pixel qui indique comment le sommet doit être colorié. Tous les autres pixels sont calculés par interpolation lors de la rasterization.

Q49 Coordonnées d'un point dans une image inversée (miroir)

$$x = 1 - x$$
$$y = y$$

Q50 Effet « bombé »

Pour réaliser un effet bombé, il faut rapprocher les points proches du centre de l'image vers le centre de l'image. Pour réaliser cet effet, il faut que les coordonnées de texture du centre du rétroviseur soient resserrées de manière non linéaire sur l'image à plaquer.

La formule $R = 1.0 - exp^{(-D/0.05)}$ apporte un ratio nul au centre et un ratio de 1 pour les bordures.

Les coordonnées de texture du centre de l'image *(x=0,5 y=0.5)* conduisent à une distance *D=0* donc à un ratio nul. L'opération *(coord-0,5)*R+0,5* ne modifie pas les coordonnées de texture *(x'=0,5 y'=0.5)*

Les coordonnées de texture des bordures de l'image (par exemple *x=1, y=0.5*) distants de *0.5* conduisent à un ratio R d'environ 1. Par l'opération *(coord-0,5)*R+0,5*, la coordonnée de texture reste en bordure l'image : *(x'=1 et y'=0.5)*

En revanche pour une coordonnée de texture d'un point proche de *0.1* unités du centre de l'image (par exemple *x=0.6 y=0.5*), cela conduit à un ratio $R \approx 0,18$. Par l'opération *((coord-0,5)*R+0,5* la coordonnée de texture devient *(x'=0.518, y'=0.5).* La coordonnée de texture a bien été rapprochée du centre.

Q51 Carte graphique

Les cartes graphiques ont une architecture dédiée au calcul parallèle permettant une exécution de rendus d'images 3D beaucoup plus rapide.

Q52 Rôle des composants d'une carte graphique

Toutes ces étapes sont réalisées dans le PolyMorph Engine, il y en a 5 par cluster.

Q53 FPU

La FPU va gérer de manière native l'opération fused multiply add (multiplication suivie d'une addition avec arrondi). Cette opération est fréquemment appelée dans le calcul matriciel.

Q54 Mémoire cache

Le cache est une mémoire qui enregistre temporairement des copies de données provenant d'une source, afin de diminuer ultérieurement le temps d'accès en lecture du processeur à ces données.