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)
```

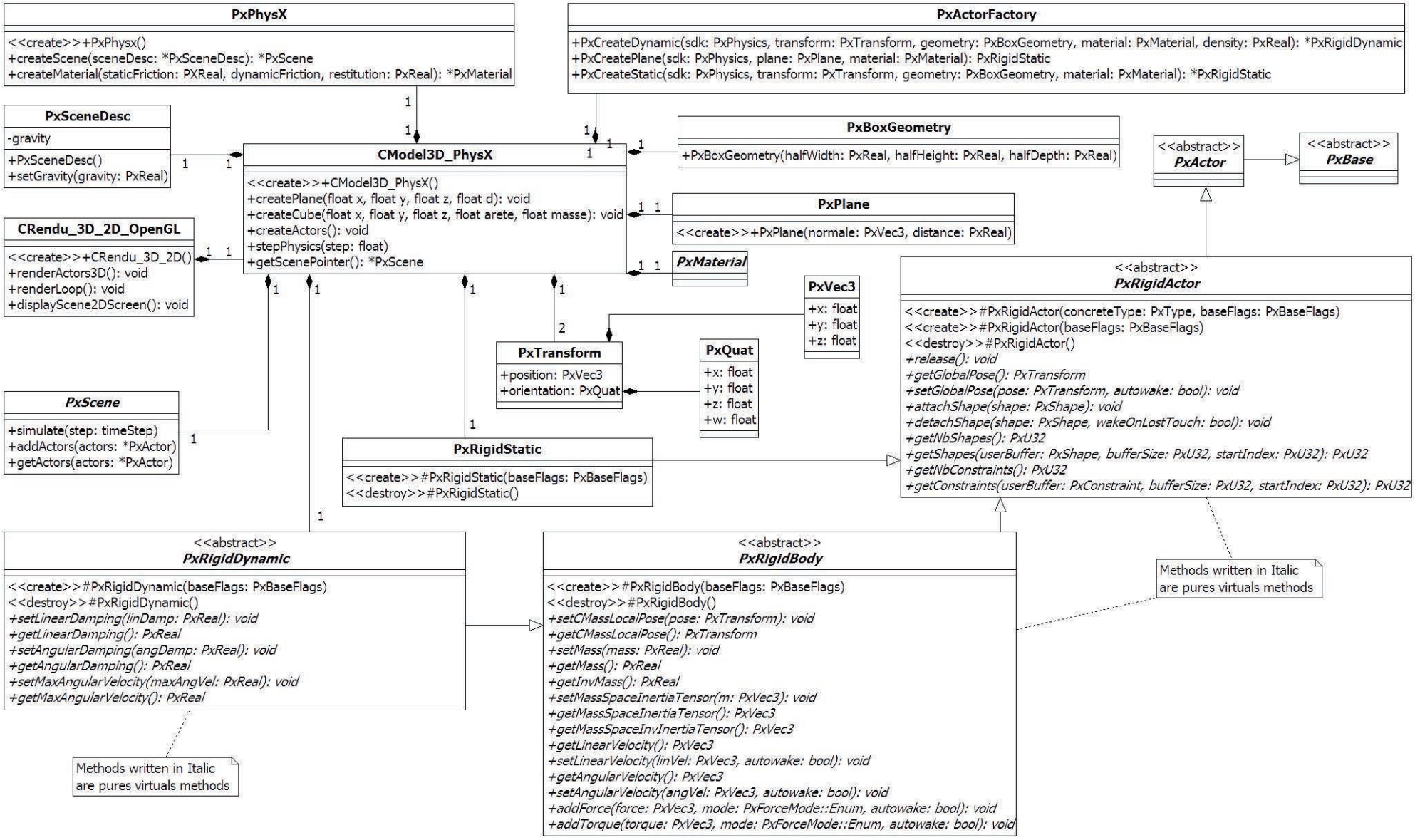
Call `odeint` to generate the solution. To pass the parameters `b` and `c` to the **pend** function; we give them to `odeint` using the `args` argument:

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

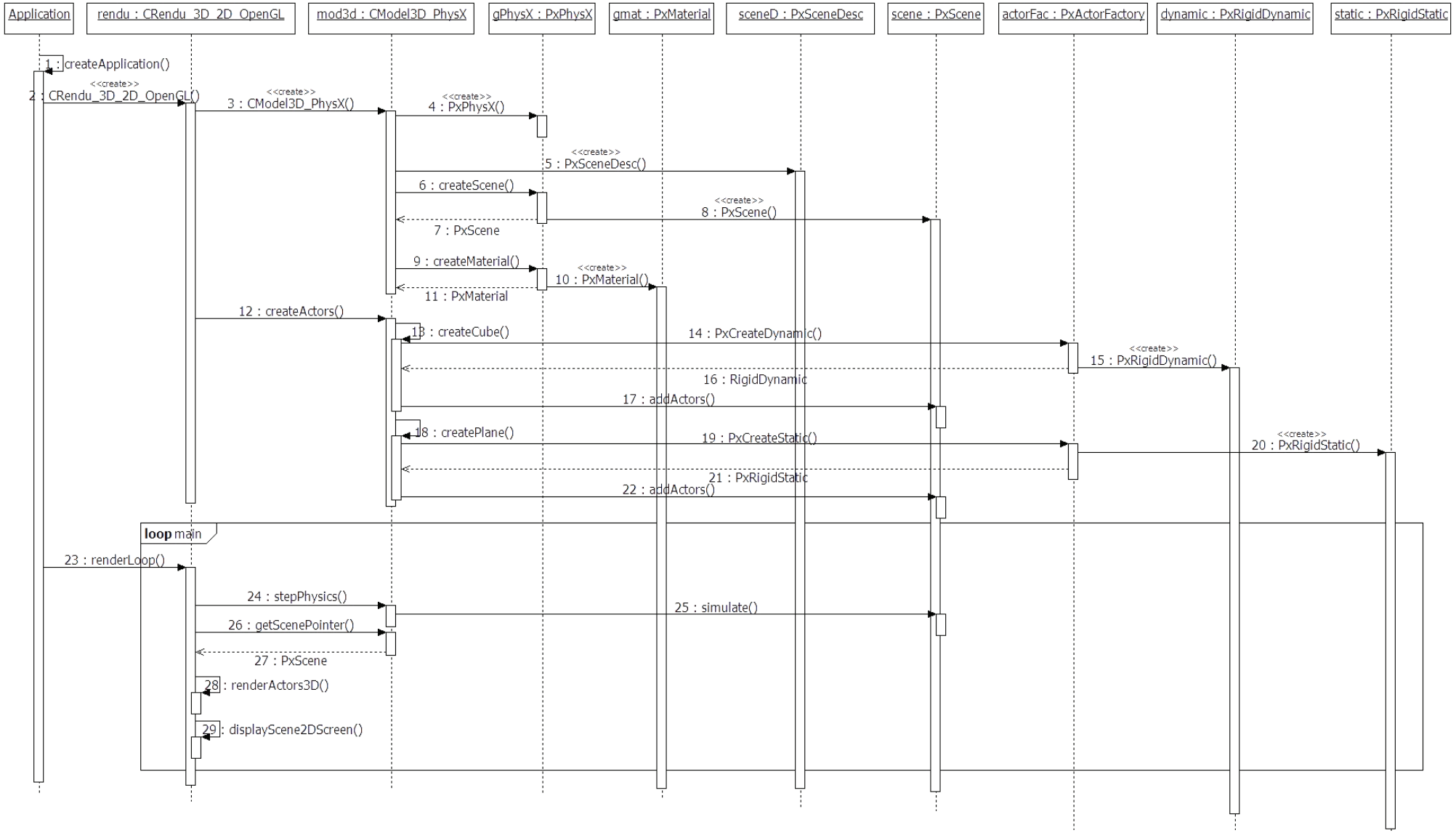

Tournez la page S.V.P.

C

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



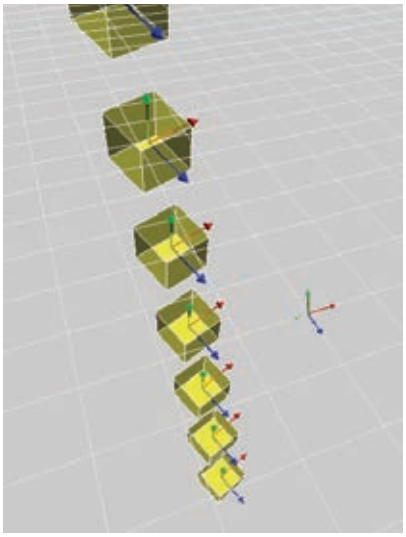
Document technique DT3 : Diagramme de séquence de l'application de test.



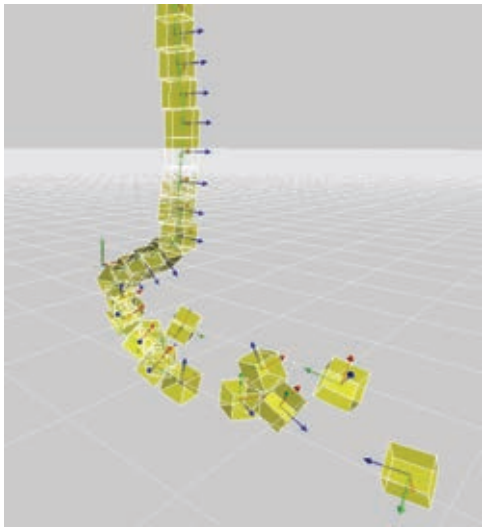
Tournez la page S.V.P.

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 $\vec{z}$.

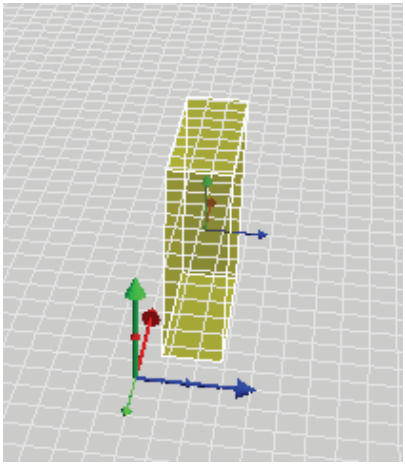


Scène à t=0 s

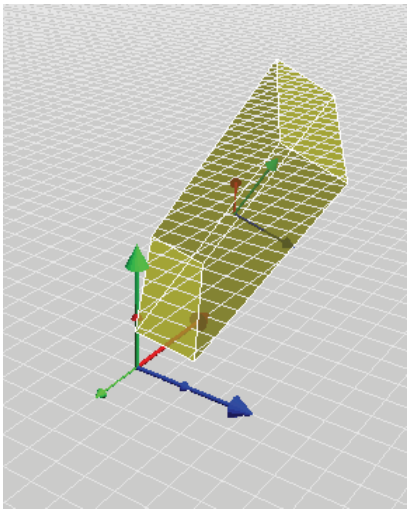


Scène à t=3 s

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°

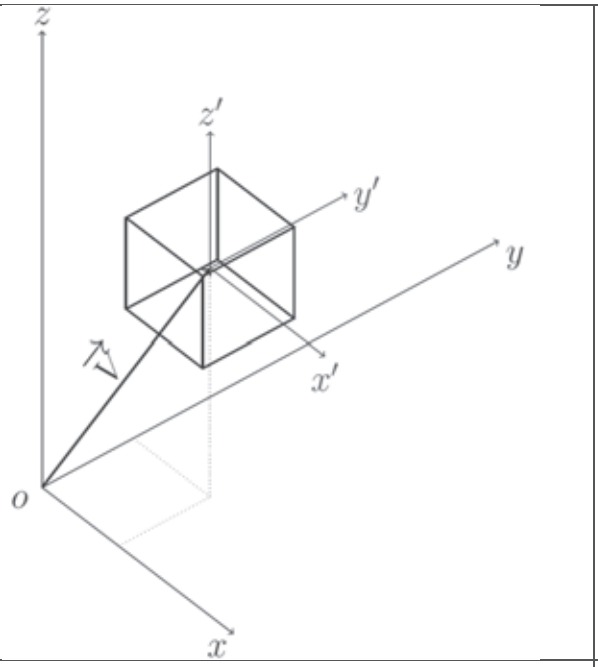
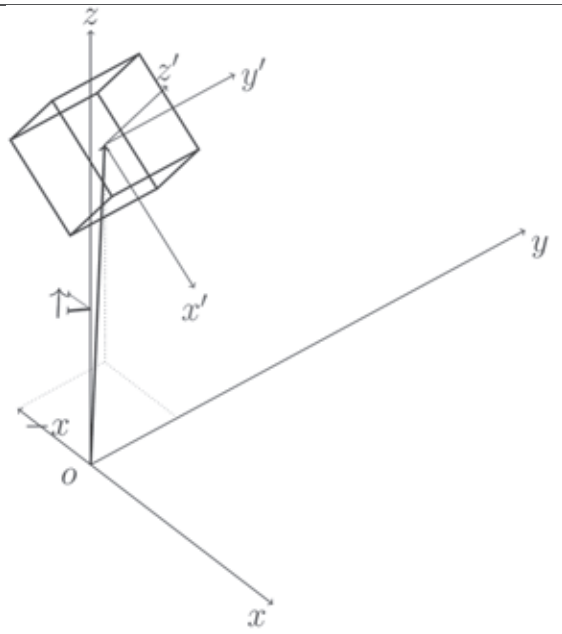
C:\WINDOWS\system32\cmd.exe

```
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 :

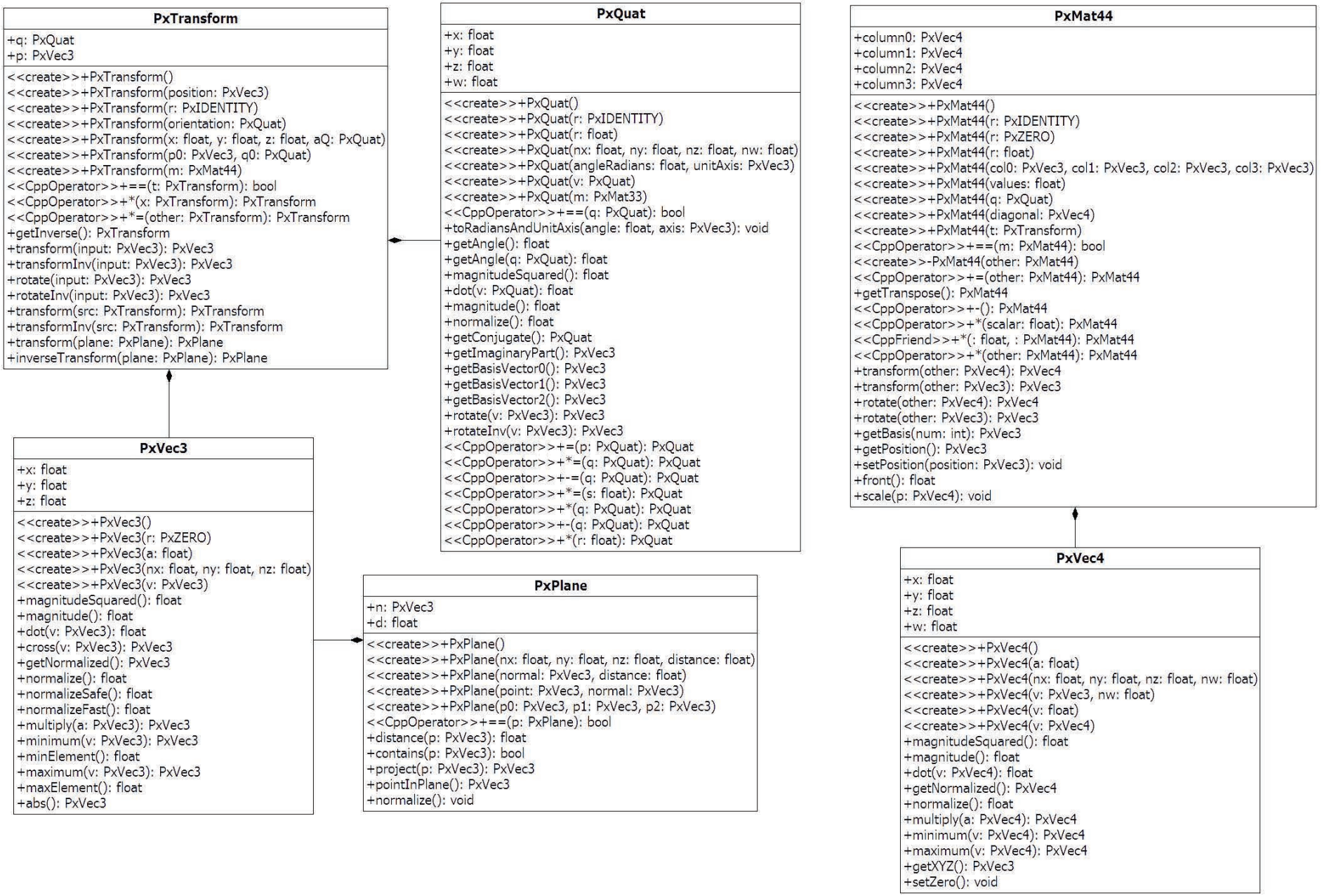
$$q = \cos\frac{\theta}{2} + \sin\frac{\theta}{2}\vec{n}$$

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 :

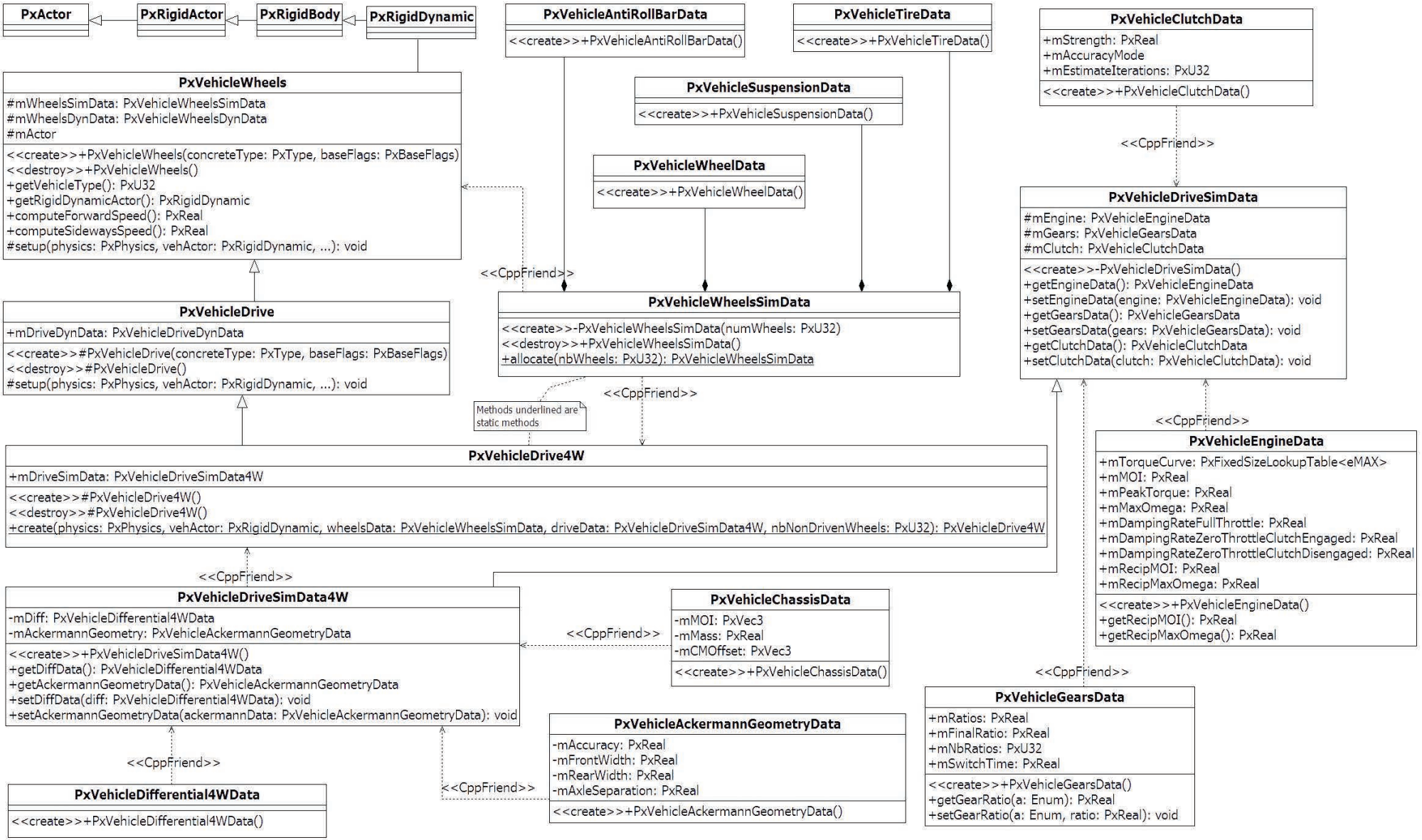
$$M = \begin{bmatrix} a^2 + b^2 + c^2 + d^2 & 2(bc - ad) & 2(bd - ac) \\ 2(bc + ad) & a^2 + b^2 + c^2 - d^2 & 2(cd - ab) \\ 2(bd - ac) & 2(cd - ab) & a^2 - b^2 - c^2 + d^2 \end{bmatrix}$$

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 ;

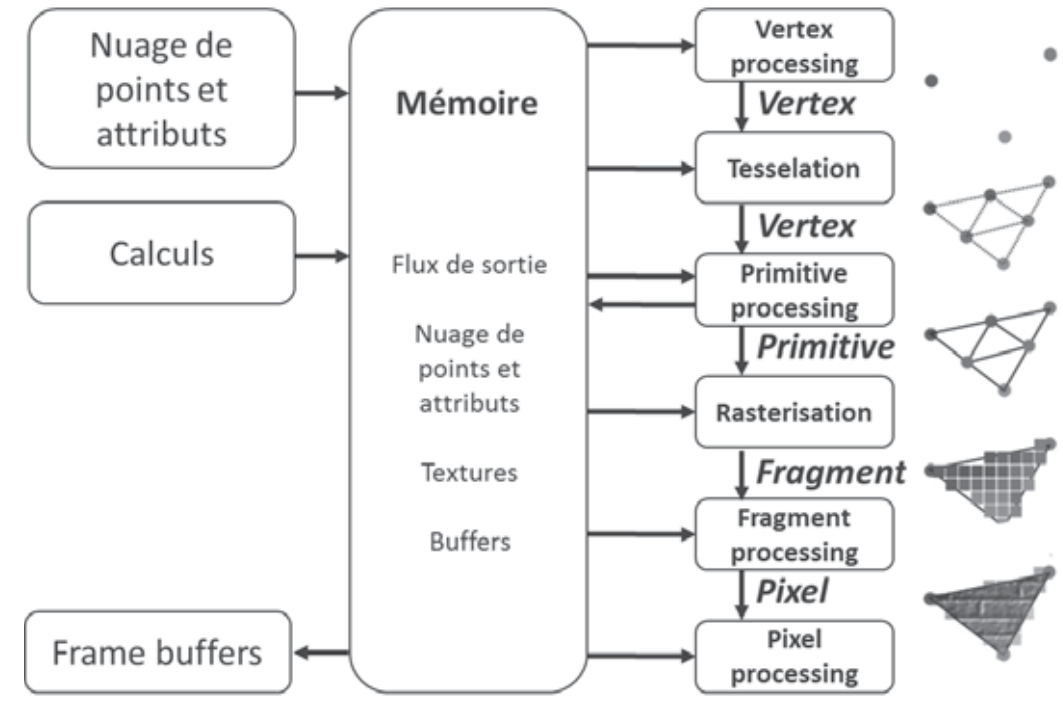
4. Compilation et déclaration d'utilisation des `shaders` à utiliser ;

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 )
```