

Newton Game Dynamics : les matériaux (Part. 1)

par Gallego Michaël (bakura.developpez.com)

Date de publication : 08/03/2007

Dernière mise à jour : 08/03/2007

Dans les trois premiers tutoriaux, nous nous sommes intéressés à comment créer différentes primitives de collision suivant nos besoins (soit un arbre de collision, soit des enveloppes convexes, ou soit des formes prédéfinies). Dans ce quatrième tutoriel, nous verrons comment rendre tout ça encore plus réaliste grâce au puissant système des matériaux ! Les matériaux tels qu'ils sont implémentés dans Newton Game Dynamics offrent beaucoup de possibilités, c'est pourquoi je vais y consacrer deux tutoriaux, dont voici la première partie !

Version de Newton Game Dynamics utilisée à l'écriture de ce tutoriel : 1.53

- I - Introduction
- II - Une architecture toute nouvelle toute belle !
- III - Tour d'horizon de nos classes Matériel !
 - III-A - Classe NewtMaterialID
 - III-B - Classe NewtMaterialPair
- IV - Programme exemple n°1
- V - Programme exemple n°2
- V - Conclusion
- VI - Remerciements

I - Introduction

Imaginez une balle rebondissante et un ballon de basket. Et un ballon de foot pendant qu'on y est ! Laissez les tomber sur le sol, et vous remarquerez évidemment quelque chose : ils ne réagissent pas pareil au contact du sol. Le ballon de basket rebondira, mais moins que la balle rebondissante, tandis que le ballon de foot, un peu dégonflé, aura tendance à s'écraser sur le sol et à peu rebondir.

Ces phénomènes physiques complexes (il n'est pas question ici de vous expliquer ces phénomènes de manière physique - parce que je ne saurais le faire -, mais juste de savoir COMMENT les implémenter avec Newton Game Dynamics) peuvent être retranscrit via le système des matériaux.

Mais comment fonctionne ce système ? A l'initialisation du moteur, un matériel "par défaut" est créé, avec certaines caractéristiques physiques (friction, élasticité,...) prédéfinies, et assigné à chaque objet, ce qui explique pourquoi tous les objets ont les même comportement lorsqu'ils se rencontrent. Nous allons donc apprendre ici à créer de nouveaux matériaux de votre cru, à leur assigner plusieurs caractéristiques, et les attribuer à des corps. Nous nous arrêterons pour ce tutoriel à ça, dans le prochain, nous verrons que Newton nous propose, avec les matériaux, un système de callback qui nous permet de gérer très finement les collisions, et d'y ajouter notamment diverses effets (un son lorsque deux objets de certains types de matériaux se rencontrent,...).

Plus concrètement, voici comment fonctionne le système des matériaux :

- 1) On crée un nouveau matériel, qui aura son ID (un entier).
- 2) On crée un second matériel, qui aura un autre ID (un entier).
- 3) Puis on définit les caractéristiques physique pour chaque pair. C'est à dire, indiquer à Newton comment réagir lorsque deux objets ayant certains matériaux vont devoir réagir. Avec deux matériaux, ce n'est pas très compliqué : il nous faut définir les caractéristiques pour le couple matériel1-matériel1, et matériel1-matériel2 (les caractéristiques seront partagés également par le couple matériel2-matériel1, il est impossible de définir d'autres caractéristiques). Idéalement, il serait judicieux de définir aussi les comportements pour les couples matérielParDefaut-matériel1 et matérielParDefaut-matériel2.
- 4) Enfin, on assigne les matériaux aux corps.

Je pense que vous avez saisi le concept ! Mais avant de nous lancer comme ça dans le bain, je vais vous parler un peu de la nouvelle architecture utilisée dans ces tutoriaux dès à présent.

II - Une architecture toute nouvelle toute belle !

En écrivant ce tutoriel avec l'ancienne architecture que j'utilisais auparavant, je me suis aperçu que ce n'était pas facile du tout, et que sa simplicité m'empêchait de pouvoir aborder les matériaux sans faire un truc tout foireux. J'ai donc décidé de tout revoir, en m'inspirant d'un excellent wrapper Newton pour Ogre (avec l'autorisation de son auteur, s'il vous plaît), tout en la simplifiant énormément (je n'ai notamment pas utilisé certaines fonctions très utiles de Boost pour les callbacks, même si je le ferai pour mon utilisation personnelle).

Elle n'est pas très difficile à comprendre, juste un peu plus longue, je vous invite donc à lire le code en détail. Concrètement, la nouvelle architecture se présente ainsi :

- Une classe `NewtWorld`, utilisant le pattern Singleton (il serait étrange d'avoir plusieurs mondes Newton !). Cette classe englobe plusieurs fonctions de Newton (pas toutes, je les ajouterai au fur et à mesure). Les commentaires sont suffisants pour comprendre l'utilité de chaque fonction !

- Une classe `NewtCollision`, qui, elle aussi, regroupe plusieurs fonctions du moteur ayant trait aux collision. Elle dispose comme variable membre un pointeur vers une structure `NewtonCollision`.

- On a ensuite un fichier nommé `NewtCollisionPrimitives`, qui est en faite une petite "compilation" de plusieurs classes très courtes, qui sont en faite chaque primitive de collision (Box, Sphere, Cone, enveloppe convexe, tableau d'enveloppes convexes,...). Chacune de ces "micro-classe" dérivent de `NewtCollision` et permettent donc de bénéficier de ses méthodes et variables membres.

- Vient ensuite la classe `NewtCollisionTree`, qui hérite aussi de `NewtCollision`. J'aurais pu l'introduire avec le fichier `NewtCollisionPrimitives`, mais les arbres de collision ont également d'autres fonctionnalités que nous verrons ultérieurement, ce qui fait que j'ai préféré en faire un fichier complètement indépendant.

- Enfin, vient la très importante classe `NewtBody`. Elle englobe aussi de nombreuses fonctions relatives à la gestion des corps (il en reste beaucoup à implémenter, mais j'ai préféré ne pas les ajouter pour le moment car nous ne les avons pas encore étudié :)). Elle dispose, outre un pointeur vers la classe `NewtWorld`, un objet `NewtCollision` (puisque chaque corps a une primitive de collision associée), un pointeur vers une structure `NewtonBody`, un entier flottant pour la masse et un objet `CModel` (le modèle 3D).

Dans la pratique, il s'avère que l'utilisation d'une telle architecture est très commode. Voici un petit exemple très simple de comment ça fonctionne :

Exemple d'utilisation

```
CModel * monModele = new CModel (); // On crée un objet CModel
monModel->OuvrirFichier ("nomDuModele.ase"); // On ouvre le fichier 3D

CVector tailleBoite (1.0, 1.0, 1.0); // Le modèle 3D est une boite ayant ces dimensions-ci
NewtCollisionBox maBoite (tailleBoite); // On crée la primitive de collision avec la taille de la
boite

CVector positionBoite (0.0, 0.0, 0.0); // La position du modèle
NewtCollisionBody * monCorps = new NewtCollisionBody (maBoite, monModele, positionBoite); // On
créé le corps
monCorps->NewtBodySetMassMatrix (40); // Le corps pèse 40 kg
monCorps->NewtBodySetForceAndTorqueCallback (); // On spécifie un callback
monCorps->NewtBodySetTransformCallback (); // Le callback de transformation

maBoite.NewtReleaseCollision (); // On n'oublie pas de libérer la mémoire de la primitive de
collision
```

Comme vous pouvez le voir, c'est très facile ! En gros, on peut créer n'importe quel primitive de collision (NewtCollisionBox, NewtCollisionSphere, NewtCollisionConvexHull,...) et assigner cet objet au constructeur de NewtCollisionBody. L'autre avantage est que la primitive de collision peut-être utilisable par plusieurs corps sans la créer plusieurs fois. Vous pouvez donc créer un seul objet NewtCollisionBox et, pour peu que plusieurs corps aient la même dimension, passer cet objet à plusieurs NewtCollisionBody !

Voilà pour ce petit tour d'horizon de notre nouvelle architecture. Nous allons donc enfin pouvoir nous pencher sur la nouveauté : les matériaux !

III - Tour d'horizon de nos classes Matériel !

Comme je l'ai dit en introduction, les matériaux fonctionnent par "pair". Nous avons donc deux classes, NewtMaterialID et NewtMaterialPair. La première est une toute petite classe qui permet de créer un nouveau matériel, tandis que la seconde, qui a comme variables membres deux objets NewtMaterialID (donc deux matériaux), et plusieurs fonctions pour configurer les différents paramètres. Etudions à présent plus en détail ces deux classes.

III-A - Classe NewtMaterialID

Voici le header de notre classe NewtMaterialID :

```
NewtMaterialID.hpp
#ifdef NEWTMATERIALID_HPP
#define NEWTMATERIALID_HPP

#include "NewtWorld.hpp"

class NewtMaterialID
{
public:
    // Constructeur / Destructeur
    NewtMaterialID (const int defaultID = 0);
    ~NewtMaterialID ();

    // Fonction renvoyant l'ID du matériel
    inline int GetID () const;

private:
    const NewtWorld * _nWorld; // Pointeur vers un objet NewtWorld
    int _matID; // Identifiant du matériel
};

int NewtMaterialID::GetID () const
{
    return _matID;
}

#endif // NEWTMATERIALID_HPP
```

Nous avons donc un constructeur, qui prend en paramètre un entier. Si la variable defaultID vaut 1, le matériel créé est le matériel par défaut, tandis que si aucune valeur n'est spécifiée, defaultID prendra la valeur 0, ce qui signifie qu'un nouveau matériel sera créé. La fonction GetID, déclarée inline, permet de renvoyer l'identifiant du matériel. Concernant les variables membres, on a un pointeur constant vers un objet NewtWorld, ainsi qu'un entier _matID qui est donc le fameux identifiant. Voici donc à présent la définition de cette classe :

```
NewtMaterialID.cpp
#include "NewtMaterialID.hpp"

// Constructeur, se charge de créer un nouveau matériel
NewtMaterialID::NewtMaterialID (const int defaultID)
{
    _nWorld = NewtWorld::GetInstance ();

    // Ce n'est pas le matériel par défaut
    if (defaultID == 0)
        _matID = NewtonMaterialCreateGroupID (_nWorld->GetNewtonWorld());

    // On récupère le matériel par défaut si defaultID == 1
    else if (defaultID == 1)
        _matID = NewtonMaterialGetDefaultGroupID (_nWorld->GetNewtonWorld());
}
```

NewtMaterialID.cpp

```

}

NewtMaterialID::~NewtMaterialID ()
{
}
    
```

Le constructeur se charge, dans un premier temps, de récupérer l'instance unique de NewtWorld (puisque'il s'agit d'un Singleton). Si defaultID vaut 0, on crée donc un nouveau matériel grâce à la fonction NewtonMaterialCreateGroupID, qui prend en paramètre un pointeur vers une structure NewtonWorld (qu'on récupère grâce à notre fonction GetNewtonWorld()). Si defaultID vaut 1, on appelle la fonction NewtonMaterialGetDefaultGroupID, qui prend également en paramètre un pointeur vers une structure NewtonWorld. Notez que cette dernière fonction ne CREE pas un nouveau matériel mais renvoie juste l'identifiant du matériel par défaut.

III-B - Classe NewtMaterialPair

NewtMaterialPair.hpp

```

#ifndef NEWMATERIALPAIR_HPP
#define NEWMATERIALPAIR_HPP

#include <iostream>
#include <string>

#include "NewtMaterialID.hpp"

class NewtMaterialPair
{
public:
    // Constructeur / Destructeur
    NewtMaterialPair (const NewtMaterialID * mat1, const NewtMaterialID * mat2);
    ~NewtMaterialPair ();

    /// -----
    /// FONCTIONS RELATIVES A NEWTON GAME DYNAMICS
    /// -----
    // Définit si deux matériaux rentrent ou non en contact
    void NewtMaterialSetDefaultCollidable (int etat);

    // Définit la "douceur" du matériel
    void NewtMaterialSetDefaultSoftness (const float softnessValue) const;

    // Définit l'elasticité du matériel
    void NewtMaterialSetDefaultElasticity (const float elasticityValue) const;

    // Définit les valeurs de friction du matériel
    void NewtMaterialSetDefaultFriction (const float staticFriction, const float kineticFriction)
const;

private:
    const NewtWorld * _nWorld; // Pointeur vers un objet NewtWorld

    const NewtMaterialID * _mat1; // Pointeur vers le matériel n°1
    const NewtMaterialID * _mat2; // Pointeur vers le matériel n°2
};

#endif // NEWMATERIALPAIR_HPP
    
```

Ci-dessus, le header de notre classe NewtMaterialPair. Le constructeur prend en paramètre deux pointeurs vers des objets NewtMaterialID, représentant le premier matériel, puis le second, ce qui constitue donc, ... la paire ! Nous avons ensuite 5 fonctions, qui nous permettent de définir les caractéristiques de la paire de matériaux. La première fonction, NewtMaterialSetDefaultCollidable, prend un paramètre un entier. Si l'entier est égal à 0, alors les collisions entre les deux objets seront ignorés totalement. La seconde fonction NewtMaterialSetDefaultSoftness permet de définir une

valeur de "douceur". A vrai dire, j'ai fait beaucoup de tests avec différentes valeurs, et les différences sont très très minimes, même entre deux valeurs extrêmement éloignées. La troisième fonction, `NewtMaterialSetDefaultElasticity`, permet elle de définir une valeur d'"élasticité". Le résultat est par contre, ici, très nette. Plus la valeur est haute, plus l'objet "rebondira", comme une balle rebondissante. A noter que les valeurs pour l'élasticité et la "douceur" doivent être, de préférence, comprises entre 0.0 et 1.0 (vous pouvez bien sûr spécifier plus). Enfin, la dernière fonction, `NewtMaterialSetDefaultFriction`, permet de définir les valeurs de friction statique et friction dynamique.

La friction statique est une force qui, lorsqu'un objet est à l'arrêt, comme un livre posé sur une table, s'oppose au déplacement de l'objet (même lorsque celui-ci est arrêté). La force de friction statique est donc la force minimale pour que l'objet se mette en mouvement ! La friction dynamique, elle, est une force qui, elle, s'oppose au déplacement lorsque l'objet est en mouvement ! Généralement, la friction statique est supérieure à la friction dynamique.

Enfin, notons comme d'habitude un pointeur vers un objet `NewtWorld` et deux pointeurs constants vers des objets `NewtMaterialID` (qui définissent donc la paire des deux matériaux). Voyons à présent la définition de cette classe qui, vous le remarquerez, n'est pas plus compliqué.

NewtMaterialPair.cpp

```
NewtMaterialPair::NewtMaterialPair (const NewtMaterialID * mat1, const NewtMaterialID * mat2)
: _mat1 (mat1), _mat2 (mat2)
{
    _nWorld = NewtWorld::GetInstance ();
}

NewtMaterialPair::~NewtMaterialPair ()
{
}

// Définit si deux matériaux peuvent "entrer en collision". Si le couple de matériaux ont
// comme valeur 0 (non collidable), alors les corps ayant ces matériaux ne rentreront pas
// en contact et se "transperceront"
void NewtMaterialPair::NewtMaterialSetDefaultCollidable (int etat)
{
    if (etat < 0 || etat > 1)
        etat = 1;

    NewtonMaterialSetDefaultCollidable (_nWorld->GetNewtonWorld(), _mat1->GetID(),
        _mat2->GetID(), etat);
}

// Définit un degré de "douceur" du corps. A vrai dire, je n'ai pas vu beaucoup de différence
// en changeant ce paramètre. J'en ferai d'autres plus tard. Cette valeur doit être
// comprise, de préférence, entre 0.0 et 1.0.
void NewtMaterialPair::NewtMaterialSetDefaultSoftness (const float softnessValue) const
{
    NewtonMaterialSetDefaultSoftness (_nWorld->GetNewtonWorld(), _mat1->GetID(),
        _mat2->GetID(), softnessValue);
}

// Définit un degré d'élasticité du matériel. Plus cette valeur sera haute, plus le corps
// ayant ce matériau "rebondira", comme une balle rebondissante. Cette valeur doit être
// comprise, de préférence, entre 0.0 et 1.0.
void NewtMaterialPair::NewtMaterialSetDefaultElasticity (const float elasticityValue) const
{
    NewtonMaterialSetDefaultElasticity (_nWorld->GetNewtonWorld(), _mat1->GetID(),
        _mat2->GetID(), elasticityValue);
}

// Définit les valeurs de friction.
void NewtMaterialPair::NewtMaterialSetDefaultFriction (const float staticFriction,
    const float kineticFriction) const
```

NewtMaterialPair.cpp

```
{  
    NewtonMaterialSetDefaultFriction (_nWorld->GetNewtonWorld(), _mat1->GetID(),  
                                     _mat2->GetID(), staticFriction, kineticFriction);  
}
```

Rien de compliqué ici. Le constructeur se charge d'initialiser nos deux variables `_mat1` et `mat2`, ainsi que de récupérer l'instance unique de notre classe `NewtWorld`. Le destructeur ne fait rien. Vient ensuite la définition des quatre fonction. `NewtMaterialSetDefaultCollidable` vérifie d'abord que la variable nommée `etat` soit une valeur valide (soit 0 ou 1). Si ce n'est pas le cas, on l'a règle à 1 (les collisions des deux matériaux ne seront pas ignorées). Puis on appelle la fonction `NewtonMaterialSetDefaultCollidable` correspondante. Le premier paramètre est le pointeur vers une structure `NewtonWorld`, le second l'identifiant du premier matériel, le troisième l'identifiant du second paramètre, puis le dernier paramètre la variable `etat`. Idem pour les deux fonctions suivantes, `NewtMaterialSetDefaultSoftness` et `NewtMaterialSetDefaultElasticity` qui fonctionnent exactement de la même façon, si ce n'est que les fonctions correspondantes sont respectivement `NewtonMaterialSetDefaultSoftness` et `NewtonMaterialSetDefaultElasticity`. Enfin, la dernière fonction, `NewtMaterialSetDefaultFriction`, qui appelle `NewtonMaterialSetDefaultFriction`. La seule différence avec les trois précédentes est qu'elle prend 5 paramètres au lieu de 4 ; le quatrième étant la valeur de la friction statique et le cinquième la valeur de la friction dynamique. Voilà voilà, étudions maintenant tout ça en situation ^^ !

IV - Programme exemple n°1

Comme le titre le laisse entendre, nous allons créer deux programmes : le premier pour illustrer différentes valeurs d'élasticité, et le second pour montrer la différence entre plusieurs valeurs de friction. Je n'en ferai pas pour la valeur de "douceur" (ou softness), car comme je vous l'ai dit, les différences sont tellement minimes que ça ne servirait à rien !

D'un point de vue pratique, et contrairement aux tutoriaux précédents, j'ai préféré ici créer une classe indépendante, que j'ai appelé sobrement Affichage, qui se chargera de créer la scène et d'afficher le tout, plutôt que de tout écrire dans le main. Je ne vais pas expliquer cette classe puisqu'elle est très facile à comprendre, je vais juste vous parler de la fonction qui va nous intéresser : InitScene.

Nous allons, dans un premier lieu, charger nos deux modèles 3D que nous allons utiliser : un sol tout simple, non texturé, et une boîte, texturée :

Affichage.cpp

```
// SOL
CModel * sol = new CModel();
sol->OuvrirFichier ("sol.ase");

/// BOITE
CModel * boite = new CModel();
boite->OuvrirFichier ("box.ase");
```

On crée ensuite les deux primitives de collision associées aux deux objets :

Affichage.cpp

```
CVector tailleSol (20.0, 20.0, 1.10);
NewtCollisionBox colSol (tailleSol);

CVector tailleBoite (1.0, 1.0, 1.0);
NewtCollisionBox colBoite (tailleBoite);
```

Vous remarquerez que j'ai ici utilisé la classe NewtCollisionBox tout en connaissant les vraies dimensions de l'objet. Si vous ne les connaissez pas, j'ai créé une fonction incluse dans la classe CModel qui vous permet de calculer les dimensions d'un objet. Sinon, vous pouvez utiliser une enveloppe convexe via la classe NewtCollisionConvexHull en passant en paramètre au constructeur un pointeur vers le modèle 3D. Toutefois, lorsque les objets ont des primitives de collision parfaites (les boîtes, les sphères, les cônes,...), il est largement conseillé de les utiliser plutôt que d'utiliser des enveloppes convexes qui vont créer des sommets inutiles.

Passons à présent à la création de nos corps :

Affichage.cpp

```
CVector positionSol (5.0, 6.0, -6.0); // La position du sol
NewtBody * bodySol = new NewtBody (colSol, sol, positionSol); // Création de l'objet bodySol
bodySol->RotationX (90); // On effectue une rotation de 90° du sol, pour qu'il soit posé
// horizontalement
_vBody.push_back (bodySol); // On ajoute le corps à notre vector _vBody
colSol.NewtReleaseCollision (); // On n'oublie pas de libérer la primitive de collision
// puisqu'elle n'est plus utilisée

CVector positionBoite; // Position des boîtes
NewtBody * bodyBox [5]; // On crée cinq pointeurs vers des objets NewtBody
```

Affichage.cpp

```
for (int i = 0 ; i < 5 ; ++i)
{
    positionBoite.SetCoordonnees (-2 * i + 4, 20, -8.0); // On définit la position de la boite n°i
    bodyBox[i] = new NewtBody (colBoite, boite, positionBoite); // On crée l'objet
    bodyBox[i]->NewtBodySetMassMatrix (40); // On définit une masse de 40 kg
    bodyBox[i]->NewtBodySetForceAndTorqueCallback (); // On définit un callback pour appliquer les
    forces
    bodyBox[i]->NewtBodySetTransformCallback (); // On définit un callback de transformation
    bodyBox[i]->RotationY (-40); // On effectue une rotation de -40° sur l'axe des Y

    _vBody.push_back (bodyBox[i]); // On l'ajoute à notre vector _vBody
}

colBoite.NewtReleaseCollision (); // On libère la primitive de collision
```

Rien à signaler ici, les commentaires sont assez explicites ! Passons à présent à la création des matériaux :

Affichage.cpp

```
const NewtMaterialID * mat1 = new NewtMaterialID ();
const NewtMaterialID * mat2 = new NewtMaterialID ();
const NewtMaterialID * mat3 = new NewtMaterialID ();
const NewtMaterialID * mat4 = new NewtMaterialID ();
const NewtMaterialID * mat5 = new NewtMaterialID ();

const NewtMaterialID * matDefault = new NewtMaterialID (1);
```

En premier lieu, on crée quatre matériaux, plus un qui est le matériel par défaut (qu'on récupère en spécifiant 1 au constructeur de NewtMaterialID). Comme les matériaux fonctionnent par paire, on aura donc 14 paires différentes à définir (1-1, 1-2, 1-3, 1-4, 1-5, 2-2, 2-3, 2-4, 2-5, 3-3, 3-4, 4-4, 4-5 et enfin 5-5), plus une autre paire qui sera la paire "par défaut", c'est à dire matDefault-matDefault.

Affichage.cpp

```
NewtMaterialPair * pair1 = new NewtMaterialPair (mat1, mat1);
_vMaterials.push_back (pair1);
NewtMaterialPair * pair2 = new NewtMaterialPair (mat1, mat2);
_vMaterials.push_back (pair2);
NewtMaterialPair * pair3 = new NewtMaterialPair (mat1, mat3);
_vMaterials.push_back (pair3);
NewtMaterialPair * pair4 = new NewtMaterialPair (mat1, mat4);
_vMaterials.push_back (pair4);
NewtMaterialPair * pair5 = new NewtMaterialPair (mat1, mat5);
_vMaterials.push_back (pair5);
NewtMaterialPair * pair6 = new NewtMaterialPair (mat2, mat2);
_vMaterials.push_back (pair6);
NewtMaterialPair * pair7 = new NewtMaterialPair (mat2, mat3);
_vMaterials.push_back (pair7);
NewtMaterialPair * pair8 = new NewtMaterialPair (mat2, mat4);
_vMaterials.push_back (pair8);
NewtMaterialPair * pair9 = new NewtMaterialPair (mat2, mat5);
_vMaterials.push_back (pair9);
NewtMaterialPair * pair10 = new NewtMaterialPair (mat3, mat3);
_vMaterials.push_back (pair10);
NewtMaterialPair * pair11 = new NewtMaterialPair (mat3, mat4);
_vMaterials.push_back (pair11);
NewtMaterialPair * pair12 = new NewtMaterialPair (mat4, mat4);
_vMaterials.push_back (pair12);
NewtMaterialPair * pair13 = new NewtMaterialPair (mat4, mat5);
_vMaterials.push_back (pair13);
NewtMaterialPair * pair14 = new NewtMaterialPair (mat5, mat5);
_vMaterials.push_back (pair14);

NewtMaterialPair * pairDefault = new NewtMaterialPair (matDefault, matDefault);
_vMaterials.push_back (pairDefault);
```

C'est un peu long, mais pas compliqué du tout :). On crée juste différentes paires, en passant au constructeur les différents objets `NewtMaterialID` pour créer toutes les paires possibles. Il ne reste plus qu'à définir les caractéristiques. Comme je l'ai dit, ce programme exemple n°1 vous permettra de voir différentes valeurs d'élasticité, nous allons donc définir seulement des valeurs d'élasticité. Vu que le principe est le même pour toutes les paires, mais juste avec des valeurs différentes, je ne vais pas tout recopier ici, mais juste vous en montrer un petit exemple :

Affichage.cpp

```
// 1-1
pair1->NewtMaterialSetDefaultElasticity (0.3f);

// 1-2
pair2->NewtMaterialSetDefaultElasticity (0.9f);

// 1-3
pair3->NewtMaterialSetDefaultElasticity (0.05f);
```

Voici donc ci-dessus la définition de trois paires. Ainsi, plus la valeur d'élasticité est haute, plus les deux objets rebondiront. Lorsqu'un objet de matériel n°1 et un autre de matériel n°2 rentreront en contact, les objets rebondiront fortement, tandis qu'au contraire, lorsqu'un objet de matériel n°1 et un de matériel n°3 rentreront au contact, il n'y aura quasiment aucun rebond, puisque la valeur d'élasticité est très faible. Mais comment trouver une valeur correcte et réaliste ? Contrairement aux valeurs de friction statique et dynamique où il existe de nombreuses tables pour trouver des valeurs corrects, je n'en ait pas trouvé pour l'élasticité. Mais il suffit de faire des tests et de définir une valeur qui semble en adéquation avec l'effet que vous cherchez à rendre. Il est évident que si vous souhaitez simuler un ballon de basket, sa valeur d'élasticité devra être plus élevée que si vous voulez simuler une caisse en bois qui tomberait. Seule paire un peu différente, la paire 1-4 :

Affichage.cpp

```
// 1-4
// Les collisions entre les objets de matériaux 1 et 4 ne se feront pas
pair4->NewtMaterialSetDefaultCollidable (0);
```

Dernière étape : assigner les matériaux aux objets. Ceci ce fait grâce à la fonction `NewtBodySetMaterialGroupID` qui prend comme unique paramètre l'identifiant du matériel :

Affichage.cpp

```
bodySol->NewtBodySetMaterialGroupID (mat1->GetID());
bodyBox [0]->NewtBodySetMaterialGroupID (mat2->GetID());
bodyBox [1]->NewtBodySetMaterialGroupID (mat3->GetID());
bodyBox [2]->NewtBodySetMaterialGroupID (mat4->GetID());
bodyBox [3]->NewtBodySetMaterialGroupID (mat5->GetID());
```

Reste enfin à détruire les objets `NewtMaterialID` en faisant plusieurs `delete` :

Affichage.cpp

```
delete mat1, delete mat2, delete mat3, delete mat4, delete mat5;
```

A noter enfin que les objets `NewtMaterialPair` seront détruits à la fin du programme. En effet, dans le prochain tutoriel, nous verrons comment utiliser différents callbacks avec les matériaux, ce qui nous empêchera donc de détruire directement les objets `NewtMaterialPair`.

On clot enfin l'explication de ce programme n°1 avec la fonction `Render` qui se charge tout simplement de dessiner tous les objets :

Affichage.cpp

```
// On active le mode debug
if (_debug)
    _nWorld->DebugShowCollision ();

// On dessine tous les objets
std::for_each (_vBody.begin(), _vBody.end(), std::mem_fun(&NewtBody::DrawModel));
```

V - Programme exemple n°2

Voici à présent notre deuxième programme exemple, censé montrer la différence à l'écran entre plusieurs valeurs de friction. Tout d'abord, comme je l'ai dit plus haut, il existe sur internet de nombreuses tables de friction (tapez static friction values, dynamic friction values,... sur Google !), ce qui confèrera à vos matériaux des réactions très réalistes. Toutefois, comme pour les valeurs d'élasticité, il est parfois plus judicieux d'inventer les siennes pour arriver au résultat souhaité, plutôt qu'à tout prix vouloir des réactions extrêmement réalistes !

Dans le code lui-même, peu de choses changent. J'ai juste placé la caméra différemment et effectué une rotation du sol afin d'avoir un plan incliné, puis j'ai placé les boîtes de manière à ce qu'elles se trouvent en haut du plan incliné. La seule chose vraiment différente est la définition des matériaux. Au lieu d'appeler `NewtMaterialSetDefaultElasticity`, j'appelle ici `NewtMaterialSetDefaultFriction`. Bien sûr, dans votre application personnelle, il vous faudra mieux définir toutes les valeurs (élasticité ET friction). Voici un petit extrait de code :

```
CPhysConvex.h
// 1-1
pair1->NewtMaterialSetDefaultFriction (0.3f, 0.25f);

// 1-2
pair2->NewtMaterialSetDefaultFriction (1.1f, 0.82f);

// 1-3
pair3->NewtMaterialSetDefaultFriction (0.025f, 0.01f);
```

A noter qu'ici, puisque les boîtes auront déjà un mouvement au début du programme, ce sera la friction dynamique qui interviendra. Ainsi, plus la valeur sera faible, ce qui signifie que la force qui s'opposera au déplacement de l'objet sera faible, plus l'objet ne sera pas "freiné" et dévalera la pente. Au contraire, l'objet doté du matériel n°2, et donc d'une forte friction dynamique, sera très vite freiné dans le déplacement jusqu'à l'arrêt, comme vous le verrez en lançant l'exécutable.

Rien d'autre à signaler, le code restant identique au programme exemple n°1 pour le reste.

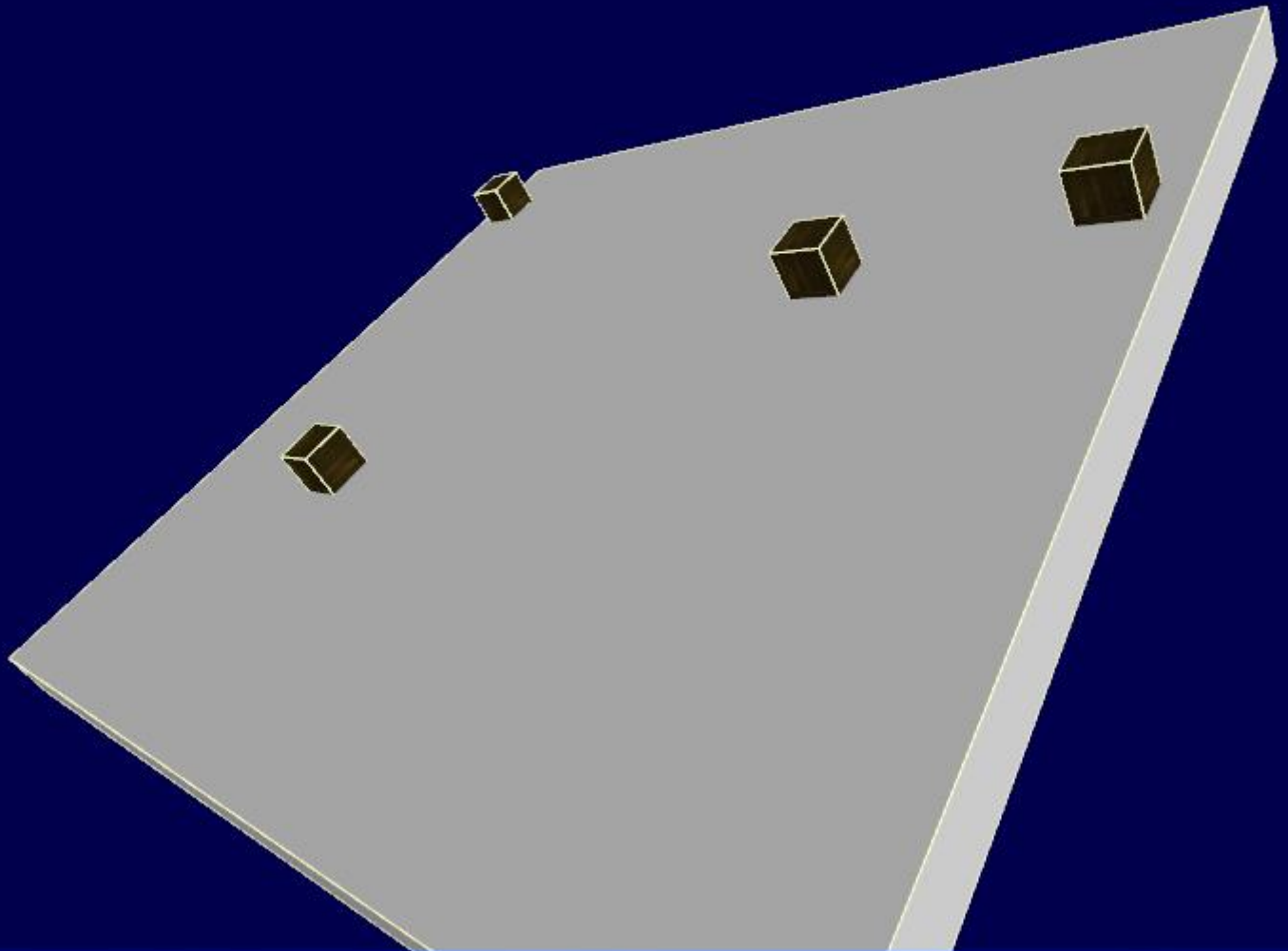
V - Conclusion

Dans ce tutoriel, vous avez appris, grâce à une utilisation basique des matériaux, à définir des réactions plus réalistes suivant le matériel duquel est composé l'objet. N'hésitez pas à expérimenter plusieurs valeurs pour choisir celles qui correspondent à ce que vous recherchez. Dans la prochaine partie, nous étudierons plus en profondeur le système de matériaux offert par Newton ! En attendant, voici les habituels screenshots des deux applications :



Application exemple n°1

80 FPS - F1 : Activation/Desactivation mode debug



Application exemple n°2

Source du programme :  [ici \(miroir\)](#)

VI - Remerciements

