Newton Game Dynamics : premier programme

Version PDF

par Gallego Michaël

Date de publication : 03/11/2006

Dernière mise à jour : 03/11/2006

Dans ce tutoriel consacré au moteur physique Newton Game Dynamics, vous apprendrez à utiliser cette puissante bibliothèque pour créer une simple petite application faisant réagir de manière réaliste des cubes et des sphères.

I - Introduction
II - Pour commencer,...
  II-1 - Installation
  II-2 - Pré-requis
III - Préparons le terrain !
  III-1 - La classe CVector
    III-1-A - CVector.h
    II-1-B - CVector.cpp
  III-2 - La classe Objet
    III-2-A - Objet.h
    III-2-B - Objet.cpp
  III-3 - La classe Boite
    III-3-A - Boite.h
    III-3-B - Boite.cpp
  III-4 - La classe Sphere
  III-5 - Classe Objet, le retour !
IV - Faisons marcher le tout !
V - Conclusion
VI - Remerciements

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.
http://bakura.developpez.com/temp/test/

I - Introduction

La place occupée par la physique dans les jeux récents est de plus en plus importante. En effet, celle-ci ajoute grandement au réalisme (adieu les bidons ou autres boîtes collés au sol !) mais propose également une nouvelle dimension au gameplay d'un jeu. Toutefois, gérer la physique en partant de rien est difficile. En effet, beaucoup d'aspects rentrent en jeu pour avoir un résultat réaliste, et cela nécessite d'être bien calé sur les lois physiques. Et puis cela prendrait tellement de temps que ça empiéterait grandement sur un temps que l'on pourrait consacrer à la création du jeu lui-même. C'est pourquoi que nous voyons fleurir depuis quelques temps de nombreux moteurs physiques. Ces moteurs sont des ensembles de fonctions permettant de gérer la physique de manière réaliste, sans connaissances poussées en physiques, sans trop de difficultés et sont optimisés (bien plus que ce qu'on pourrait créer par nous-même) ! Le plus célèbre d'entre-eux est sans aucun doute le moteur Havok, utilisé par beaucoup de jeux professionnels. Toutefois, ce dernier est cher et réservé aux developpeurs professionnels. Mais ne nous inquiétons pas, il en existe de nombreux autres qui sont gratuits. Parmi ces derniers, on peut citer ODE, Tokamak ou autre Newton Game Dynamics. C'est de ce dernier dont nous allons parler aujourd'hui ! Ses atouts sont qu'il est complètement gratuit, très puissant, simple d'utilisation, et surtout toujours mis à jour et activement supporté par une communauté assez nombreuse. Vous pourrez juger de ses capacités en vous rendant à cette adresse : ici. Dans ce premier article consacré au moteur Newton (il y en aura peut-être d'autres plus tard), je me suis efforcé à créer quelque chose de très facile (le langage utilisé est le C++). Au final, vous obtiendrez un sol statique, avec des cubes et des sphères tombant et réagissant les uns aux autres de manière réaliste. Vous verrez, ce n'est pas très dur !

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

II - Pour commencer,...

II-1 - Installation

L'installation du Newton Game Dynamics est pour le moins facile. Téléchargez le SDK correspondant à votre OS en suivant cette adresse (à l'heure où j'écris, la dernière version est la version 1.53) : ici. Un executable vous guidera pour extraire tout ce qui s'y trouve. Vous devrez donc trouver dans votre dossier fraîchement installé trois dossiers : doc (qui contient des tutoriaux sur des sujets plus complexes ainsi que la doc, INDISPENSABLE pour connaître l'utilité de chaque fonction,...), samples (qui contient les codes-sources des tutoriaux de la doc et quelques autres), et enfin sdk. Copiez le fichier Newton.h dans le dossier include de votre compilateur habituel. Dans le sous-dossier dll, copiez Newton.dll dans C:/WINDOWS/system32 (vous devrez livrer cette dll avec vos programmes pour que les autres puissent le lancer), puis Newton.lib dans votre dossier lib de votre compilateur. Il ne nous reste plus qu'à lier le tout. Chez moi, sur Code::Blocks, voici la procédure à suivre : Settings > Compiler and Debugger > Linker > dans Link Librairies, cliquer sur Add > Ajouter le fichier lib précédemment sauvegardé. Et voilà, vous êtes prêt à utiliser le Newton Game Dynamics.

II-2 - Pré-requis

Vous devez avoir les bases du C++ et de l'OpenGL (j'utiliserai également GLU pour les sphères). Pour le fenêtrage, j'ai utilisé la SDL, mais rien ne vous empêche d'utiliser GLUT ou l'API Win32. Bien sûr, j'ai utilisé OpenGL, mais vous pouvez utiliser DirectX, NGD n'est pas spécifique à une API. J'ai également fait usage de la bibliothèque SDL_gfx pour gérer efficacement le frame-rate. En effet, cela permet de "bloquer" le frame-rate, ainsi le programme marche de la même façon quelque soit l'ordinateur (sauf si évidemment l'ordinateur est trop peu puissant pour atteindre ce frame-rate). Pour plus d'informations sur comment utiliser cette bibliothèque (c'est très facile et très utile !), ou pour passer par une autre méthode, je vous renvoie vers l'article de fearyourself : voir ici

Concernant la physique, vous n'avez pas besoin d'être un crack, mais avoir déjà entendu parler des forces est bien entendu un plus. Pour ceux qui ignoreraient totalement ce concept ou qui auraient oublié leurs cours de seconde, sachez qu'une force est un phénomène physique. Pour rester simple, chaque corps subit plusieurs forces qui attirent ce corps, ce qui provoque une modification de la vitesse de l'objet (par exemple, le simple fait de lâcher un stylo fait intervenir plusieurs forces, la force gravitationnelle de la Terre attire le stylo vers le sol). Pour plus d'informations sur les forces, voici un article très complet et bien fait : http://fr.wikipedia.org/wiki/Force_(physique)

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

III - Préparons le terrain !

III-1 - La classe CVector

III-1-A - CVector.h

CVector.h
```
#ifndef CVECTOR_H
#define CVECTOR_H

#include <GL/GL.h> // Pour les GLfloat ; vous pouvez vous passer de cet inclusion de fichier en
                   // remplacant les GLfloat par des float

class CVector
{
   public:
      CVector ();
      CVector (const GLfloat fX, const GLfloat fY, const GLfloat fZ);
      virtual ~CVector();

      void ReglerCoordonnees (const GLfloat fX, const GLfloat fY, const GLfloat fZ);

      GLfloat x, y, z;
};

#endif // CVECTOR_H
```

Voici le header pour notre classe CVector. Ceci est une classe vecteur très largement simplifiée. Il est évident que dans une application plus conséquente, il vous faudra quelque chose de plus robuste. Toutefois, elle nous suffira pour notre usage.

Nous définissons donc un constructeur CVector ne prenant aucun paramètre et un second prenant en paramètre les trois coordonnées pour initialiser notre vecteur. La fonction ReglerCoordonnnes a ce même usage. Il permet de réutiliser plusieurs fois le même objet CVector sans en recréer plusieurs.

Enfin, les trois variables sont déclarées ici public, pour un soucis de simplicité.

II-1-B - CVector.cpp

CVector.cpp
```
#include "CVector.h"

CVector::CVector ()
   : x (0.0f), y (0.0f), z (0.0f)
{
   // Ne fait rien
}

CVector::CVector (const GLfloat fX, const GLfloat fY, const GLfloat fZ)
   : x (fX), y (fY), z (fZ)
{
   // Ne fait rien
}

CVector::~CVector()
{
   // Ne fait rien
}

void CVector::ReglerCoordonnees (const GLfloat fX, const GLfloat fY, const GLfloat fZ)
```


Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

CVector.cpp
```
    {
        x = fX;
        y = fY;
        z = fZ;
    }
```

Je ne pense pas que le reste de cette classe CVector ne vous pose de problème. On se charge juste d'initialiser toutes les valeurs.

III-2 - La classe Objet

III-2-A - Objet.h

Après la classe CVector qui était pour le moins très simple, voici à présent la classe Objet. Cette classe est une classe abstraite. Les classes suivantes en hériteront.

Nous incluons en premier lieu nos fichiers à utiliser :

Objet.h
```
#include <iostream>

#include <GL/GL.h>
#include <GL/glu.h>
#include <Newton/Newton.h> // Header pour utiliser le Newton Engine
#include "CVector.h"
```

L'en-tête pour iostream est obligatoire sans quoi vous aurez pleins de messages d'erreurs. L'inclusion de l'en-tête fichier Newton.h sera peut-être différent chez vous. Moi, dans mon dossier include, j'y ai créé un dossier Newton ou j'y ai collé le fichier header. Si vous l'avez mis dans un dossier différent du mien, vous devrez changer cette ligne. Enfin le dernier en-tête à inclure nous permet d'utiliser notre classe vecteur créée précédemment.

Nous allons également avoir besoin d'une petite structure matrice, que nous définissons juste après les différents include :

Objet.h
```
// Structure matrix
struct matrix
{
GLfloat matrice [4][4];
};
```

Nous utilisons des matrices OpenGL, donc de 4*4. Celle-ci nous sera obligatoire pour gérer la physique. En effet, les translations, rotations et autre scaling sont gérées grâce à la matrice. Ainsi, chaque objet aura sa propre matrice, qui sera modifiée par la bibliothèque Newton à chaque fois que la position de l'objet sera changée.

Nous définissons ensuite notre classe abstraite Objet, qui se présente comme-ceci :

Objet.h
```
class Objet
{
// Fonction amie qui se chargera d'appliquer les forces à l'objet.
```


Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

Objet.h
```
// Elle sera utilisée en fonction Callback par le Newton Engine
friend void ApplyForceAndTorqueCallback (const NewtonBody * nBody);

   public:
      Objet (); // Constructeur
      virtual ~Objet (); // Destructeur

      virtual void Initialiser () {};
      virtual void Dessiner () = 0; // Classe pure
      virtual void SetColor (const CVector &) = 0;

   protected:
      NewtonBody * m_pBody; // Pointeur vers un NewtonBody
      GLfloat m_masse; // Masse de l'objet
      CVector m_couleur; // Couleur de l'objet
};
```

Celle-ci n'a rien de très compliquée, toutefois elle a de quoi vous surprendre peut-être un peu. Nous utilisons en effet une fonction ami nommé ApplyForceAndTorqueCallback, et prenant en paramètre un pointeur vers un objet NewtonBody. En effet, la bibliothèque Newton Game Dynamics utilise des callbacks. Il s'agit en fait d'une fonction qui sera systématiquement appelée pour chaque corps à chaque boucle de l'application, sauf si l'objet est inactif ou a atteint un état d'équilibre (dixit la doc). Le paramètre est justement le corps en question.

Nous avons ensuite un constructeur, et un destructeur virtuel. Nous avons ensuite une fonction virtuelle Initialiser, une fonction virtuelle Dessiner qui est déclarée pure (qui devra donc obligatoirement être substituée dans les classes filles), ainsi qu'une autre fonction pure prenant en paramètre une référence vers un objet CVector et qui nous permettra, à travers les trois valeurs du vecteur, de spécifier la couleur de l'objet (ici, xyz = RGB).

Concernant les variables membres (attention à bien les déclarer protected et non private pour que les autres classes puissent y accéder), nous avons un pointeur m_pBody de type NewtonBody. Voici la première nouveautée ! Chaque corps devra donc posséder sa propre variable NewtonBody. Nous avons ensuite la masse de l'objet spécifiée sous la type d'un entier flottant, et enfin un objet m_couleur de type CVector pour stocker la couleur de l'objet.

III-2-B - Objet.cpp

En voici à présent la définition :

Objet.cpp
```
Objet::Objet ()
: m_pBody (NULL)
{
   // Ne fait rien
}

Objet::~Objet ()
{
   NewtonDestroyBody (NewtonBodyGetWorld (m_pBody), m_pBody);
}
```

Rien de bien compliquer... Nous initialisons à NULL l'objet m_pBody, tandis que le destructeur se charge de détruire le corps. A noter comment nous détruisons le corps : la fonction NewtonDestroyBody prend en paramètre un objet de type NewtonWorld * (nous le verrons plus tard), et un pointeur vers le corps en question. Le monde auquel est rattaché le corps peut-être obtenu par la fonction NewtonBodyGetWorld. C'est ce que nous faisons ici.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

Concernant la fonction ApplyForceAndTorqueCallback, je vous l'expliquerai un peu plus tard...

III-3 - La classe Boite

III-3-A - Boite.h

Après nous être mis en jambe, nous allons passer à quelque chose d'un peu plus conséquent, à savoir la classe que j'ai nommé Boite. Cette classe hérite de la classe Objet par un héritage public. Nous devons donc y inclure l'header de cette dernière. Mais voici sans attendre cette classe Boite :

Boite.cpp
```
class Boite : public Objet
{
   public:
      Boite (); // Constructeur
      virtual ~Boite (); // Destructeur

          // Fonction se chargeant de l'initialisation de la boîte
      virtual void Initialiser (NewtonWorld * nWorld, const CVector &, const CVector &,
          GLboolean mobile = false, GLfloat masse = 1.0f);
          virtual void Dessiner (); // Dessine l'objet
          virtual void SetColor (const CVector &); // Règle les couleurs

   private:
          CVector * m_taille; // Dimensions de la boîte
};
```

Toujours un constructeur, un destructeur. Puis la fonction Initialiser qui est un peu plus complexe. Le premier paramètre est un pointeur vers un objet de type NewtonWorld. En effet , pour initialiser un corps, nous devons la première fois le "lier" à un monde. Puis nous disposons de deux références constantes vers des objets CVector (respectivement la taille et la position). Pas de nouveauté pour le reste, si ce n'est une nouvelle variable membre, m_taille, pour spécifier la taille du cube.

III-3-B - Boite.cpp

Boite.cpp
```
Boite::Boite ()
: Objet ()
{
   // Ne fais rien
}

Boite::~Boite()
{
   // Ne fait rien, l'objet est détruit par la classe de base
}
```

Les constructeurs et destructeurs ne font rien de spécial, si ce n'est que le constructeur appel celui de la classe de base. Sinon, le corps n'est pas à détruire puisqu'il est détruit par le destructeur de la classe de base.

Boite.cpp
```
void Boite::Initialiser (NewtonWorld * nWorld, const CVector & taille, const CVector & position,
                         GLboolean mobile, GLfloat masse)
{
   // On initialise le vecteur de dimensions
   m_taille.x = taille.x;
   m_taille.y = taille.y;
   m_taille.z = taille.z;

   // On définit la masse de l'objet
   m_masse = masse;
```


Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

Boite.cpp

```
	matrix matrice; // On créé une matrice

	// On initialise la matrice, qu'on définit comme matrice identité
	for (int x = 0 ; x < 4 ; ++x)
	   for (int y = 0 ; y < 4 ; y++)
	   {
	      if (x == y)
	         matrice.matrice [x][y] = 1.0f;
	      else
	         matrice.matrice [x][y] = 0.0f;
	    }

	// On définit la matrice de manière à ce que l'objet soit placé aux positions
	// spécifiées en utilisant la dernière colonne de la matrice
	matrice.matrice [3][0] = position.x;
	matrice.matrice [3][1] = position.y;
	matrice.matrice [3][2] = position.z;

	// On initialise la boîte de collision
	NewtonCollision * collision = NULL;

	// On créé la boite de collision aux dimensions de l'objet
	collision = NewtonCreateBox (nWorld, m_taille.x, m_taille.y, m_taille.z, NULL);

	// On initialise le corps avec la boite de collision
	m_pBody = NewtonCreateBody (nWorld, collision);

	if (m_pBody == NULL)
	   std::cerr << "Impossible d'initialiser le corps.";

	// On détruit la boite de collision, on n'en a plus besoin
	NewtonReleaseCollision (nWorld, collision);

	// Enfin, on affecte notre matrice (qui représente donc sa position dans l'espace)
	// à notre corps grâce à la fonction NewtonBodySetMatrix
	NewtonBodySetMatrix (m_pBody, &matrice.matrice [0][0]);

	// On initialise à présent les propriétés physiques de l'objet. Toutefois, donner
	// à un objet qui ne bougera pas une masse, lui associer un callback,... n'a aucun
	// intêret, on vérifie donc si l'objet sera mobile ou immobile
	if (mobile == true)
	{
	   // On calcul l'inertie du corps, en passant par une petite formule
	   CVector inertie;

	   inertie.x = 0.7f * m_masse * (m_taille.y * m_taille.y + m_taille.z * m_taille.z) / 12;
	   inertie.y = 0.7f * m_masse * (m_taille.x * m_taille.x + m_taille.z * m_taille.z) / 12;
	   inertie.z = 0.7f * m_masse * (m_taille.x * m_taille.x + m_taille.y * m_taille.y) / 12;

	   // On définit ensuite la masse et l'inertie pour ce corps
	   NewtonBodySetMassMatrix (m_pBody, m_masse, inertie.x, inertie.y, inertie.z);

	   // On règle enfin le Callback, qui sera nécessaire pour que le corps bouge
	   NewtonBodySetForceAndTorqueCallback (m_pBody, ApplyForceAndTorqueCallback);
	}

	m_taille.x *= 0.5f;
	m_taille.y *= 0.5f;
	m_taille.z *= 0.5f;
}
```

La fonction Initialiser est déjà plus conséquente ! Les commentaires devraient être suffisant, mais voici quelques compléments pour mieux saisir : la variable m_taille est initialisée avec les valeurs passées en paramètres et divisées par deux.

Nous définissons ensuite la masse avec la valeur passée en paramètre (qui est par défaut à 0.0, soit un corps fixe). On crée ensuite une matrice, que nous initialisons à la matrice identité (où m[0][0] = m[1][1] = m[2][2] = m[3][3] = 1.0). Cette matrice permettra de gérer tout ce qui concerne les transformations. Il est inutile de définir la matrice comme une variable membre car Newton en dispose une en interne, qu'il nous suffira tout simplement de récuperer. Dans une application plus sérieuse, il serait évidemment mieux de créer une jolie classe et une fonction

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

pour directement mettre à la matrice identité.

Ensuite, nous positionnons le cube avec les valeurs passés en paramètres. En effet, la dernière colonne de la matrice est dédiée aux transformations. Ainsi la case [3][0] définit la position x, [3][1] la position y et [3][2] la position z. Nous créons ensuite un pointeur vers un objet de type NewtonCollision. Ceci nous permet de créer le rectangle (ou sphère, ou d'autres formes, voir la doc de Newton) de collision qui englobera l'objet et permettra les collisions réalistes. On l'initialise avec la valeur de retour de la fonction NewtonCreateBox. Celle-ci prend comme premier paramètre un pointeur vers un objet NewtonWorld (passé en paramètre à la fonction Initialiser), puis la taille x, y, z de l'objet. Le dernier paramètre sera spécifié NULL afin de spécifier que la boîte de collision sera centrée à l'origine de la boîte.

Puis on initialise le corps (la variable m_pBody) avec la fonction NewtonCreateBody qui prend comme premier paramètre le monde associé et le rectangle de collision associé que nous avons créé précédemment. On vérifie ensuite que l'allocation se soit bien réalisée en testant le pointeur vers le corps. Si le pointeur vaut NULL, alors on écrit dans un fichier d'erreur. Il ne faut ensuite pas oublier de libérer l'objet de collision avec la fonction NewtonReleaseCollision. A noter qu'un même rectangle de collision peut-être utilisé plusieurs fois pour un même d'objet s'il possède la même taille.

Nous avons notre corps, nous lui assignons à présent sa propre matrice (qui définit donc sa position dans l'espace) avec la fonction NewtonBodySetMatrix, qui prend en premier paramètre le corps en question et en second la matrice.

On vérifie ensuite le booléen "mobile". Les commentaires sont je pense assez explicites. Ceci permet d'éviter quelques appels de fonctions inutiles. Calculer ses valeurs d'inértie, lui affecter un callback,... n'est d'aucune utilité si l'objet ne bouge pas. En revanche, s'il bouge, c'est très important ! Pour cette tache, nous créons le vecteur d'inertie. Puis nous l'initialisons avec quelques calculs. Spécifier de bonnes valeurs pour l'inertie est important pour que le corps réagisse de manière réaliste. D'après la définition de Wikipedia : "L'inertie d'un corps découle de la nécessité d'exercer une force sur celui-ci pour modifier sa vitesse (vectorielle). Ainsi, un corps immobile ou en mouvement rectiligne uniforme (se déplaçant sur une droite à vitesse constante) n'est soumis à aucune force, et réciproquement." Concernant le calcul, je l'ai retiré des exemples des tutoriaux livrés avec le SDK. Contentez-vous de les appliquer.

Bref, une fois que nous avons tout ça, nous pouvons enfin dire à notre corps sa masse et son inertie (nul doute qu'il est très heureux !) grâce à la fonction NewtonBodySetMassMatrix. La fonction prend un pointeur vers le corps, puis la masse et les trois valeurs de l'inertie. Pour finir, nous appliquons à notre corps le fameux callback. Rappelons le, la fonction spécifiée en callback sera appelée à chaque boucle pour effectuer les mises à jour sur le corps, jusqu'à ce qu'il soit immobile. On utilise pour cela la fonction NewtonBodySetForceAndTorqueCallback (rien que ça !). Celle-ci prend en premier paramètre le pointeur vers le corps et un pointeur de fonction vers notre fonction ApplyForceAndTorqueCallback (je l'ai appelé comme ça, mais vous pouvez l'appeler Force ou n'importe quel nom...).

Une petite précision : comme nous l'avons vu au dessus, certains corps ne bénéficieront donc pas de callbacks (les objets immobiles). En effet, Newton peut également être utilisé comme un simple moteur de collision, sans la gestion de la force. Il est donc tout à fait possible de ne spécifier qu'un simple rectangle de collision sans spécifier de callback.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

C'en est fini avec notre if, toutefois il nous reste une dernière chose à régler. Nous allons diviser par deux toutes les tailles (donc x, y, z), de notre cube. En effet, lorsque nous dessinerons notre cube, nous ferons par exemple : glVertex3f (-m_taille.x, m_taille.y, m_taille.z). Nous dessinerons "des deux côtés", ainsi sur un carré de un de côté, nous dessinerons 0.5 à gauche et 0.5 à droite, plutôt que partir d'un point 0 et tracer jusqu'à 1.0. Si nous faisons ceci, les collisions ne marcheront pas (pour revenir sur un côté de 1.0, Newton délimite le rectangle de collision de manière à ce qu'il y ait 0.5 à gauche et 0.5 à droite, ce qui nous donne bien 1, plutôt que de tracer de 0.0 à 1.0 d'un coup). J'espère que je suis assez clair. Vous aurez qu'à faire quelques modifs sur ces lignes, vous comprendrez.

Boite.cpp

```
void Boite::SetColor (const CVector & couleur)
{
   m_couleur.x = couleur.x / 255;
   m_couleur.y = couleur.y / 255;
   m_couleur.z = couleur.z / 255;
}
```

La fonction SetColor n'a pas besoin de trop d'explication je pense. Si ce n'est que nous divisons les valeurs par 255 pour se retrouver avec les valeurs en tant que float plutôt qu'en unsigned byte lors de l'appel à glColor.

Boite.cpp

```
void Boite::Dessiner ()
{
   // Toutes les modifications effectuées dans le Callback modifient la matrice de
   // l'objet, ce qui permet à l'objet de "bouger"
   matrix matrice;
   NewtonBodyGetMatrix (m_pBody, &matrice.matrice [0][0]);

   glPushMatrix (); // On sauvegarde la matrice actuelle
   {
      glMultMatrixf (&matrice.matrice [0][0]); // On multiplie la matrice actuelle
                                               // par la matrice du corps, ainsi
                                               // le corps sera dessiné au bon endroit
      glColorMaterial (GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE);
      glColor3f (m_couleur.x, m_couleur.y, m_couleur.z);
      glColorMaterial (GL_FRONT_AND_BACK, GL_SPECULAR);
      glColor3f (1.0f, 1.0f, 1.0f);
      glColorMaterial (GL_FRONT_AND_BACK, GL_EMISSION);
      glColor3f (0.0f, 0.0f, 0.0f);
      glMaterialf (GL_FRONT_AND_BACK, GL_SHININESS, 128);

      glBegin(GL_QUADS);
         // Devant
         glNormal3f (0.0f, 0.0f, 1.0f);
         glVertex3f (-m_taille.x, -m_taille.y, m_taille.z);
         glVertex3f (m_taille.x, -m_taille.y, m_taille.z);
         glVertex3f (m_taille.x, m_taille.y, m_taille.z);
         glVertex3f (-m_taille.x, m_taille.y, m_taille.z);

         // Derrière
         glNormal3f (0.0f, 0.0f, -1.0f);
         glVertex3f (-m_taille.x, -m_taille.y, -m_taille.z);
         glVertex3f (-m_taille.x, m_taille.y, -m_taille.z);
         glVertex3f (m_taille.x, m_taille.y, -m_taille.z);
         glVertex3f (m_taille.x, -m_taille.y, -m_taille.z);

         // Haut
         glNormal3f (0.0f, 1.0f, 0.0f);
         glVertex3f (-m_taille.x, m_taille.y, -m_taille.z);
         glVertex3f (-m_taille.x, m_taille.y, m_taille.z);
         glVertex3f (m_taille.x, m_taille.y, m_taille.z);
         glVertex3f (m_taille.x, m_taille.y, -m_taille.z);

         // Bas
         glNormal3f (0.0f, -1.0f, 0.0f);
         glVertex3f (-m_taille.x, -m_taille.y, -m_taille.z);
         glVertex3f (m_taille.x, -m_taille.y, -m_taille.z);
         glVertex3f (m_taille.x, -m_taille.y, m_taille.z);
         glVertex3f (-m_taille.x, -m_taille.y, m_taille.z);

         // Droite
         glNormal3f (1.0f, 0.0f, 0.0f);
```


Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

Boite.cpp
```
            glVertex3f (m_taille.x, -m_taille.y, -m_taille.z);
            glVertex3f (m_taille.x, m_taille.y, -m_taille.z);
            glVertex3f (m_taille.x, m_taille.y, m_taille.z);
            glVertex3f (m_taille.x, -m_taille.y, m_taille.z);

            // Gauche
            glNormal3f (-1.0f, 0.0f, 0.0f);
            glVertex3f (-m_taille.x, -m_taille.y, -m_taille.z);
            glVertex3f (-m_taille.x, -m_taille.y, m_taille.z);
            glVertex3f (-m_taille.x, m_taille.y, m_taille.z);
            glVertex3f (-m_taille.x, m_taille.y, -m_taille.z);
         glEnd ();
      }
      glPopMatrix (); // On rétablit la matrice
   }
```

Voilà enfin la fonction qui sert à dessiner tout ce beau monde. Nous commençons en premier lieu par récupérer la nouvelle matrice de l'objet (et par conséquent, sa position,...) avec la fonction NewtonBodyGetMatrix. Nous lui passons en second paramètre notre matrice afin qu'elle la mette à jour. Cette étape est importante. En effet, le callback a directement modifié la matrice de l'objet, et il nous faut la récupérer afin de dessiner l'objet au bon endroit.

On sauvegarde ensuite la matrice actuelle, puis on la multiplie avec la matrice du corps. On spécifie les couleurs avec la fonction glColorMaterial puis on dessine l'objet. Enfin, on rétablit la matrice. Alors, c'est pas si compliqué ?

III-4 - La classe Sphere

La classe Sphère fonctionne sensiblement de la même façon que la classe Boite, si ce n'est que le vecteur pour la taille est remplacé par une simple variable pour stocker le rayon, et que nous utilisons GLU pour créer la sphère. Il ne s'agit pas de l'objectif de ce tutoriel, mais voici en quelques lignes comment vous servir des quadrics de GLU pour créer diverses formes, comme des sphères.

1) Vous devez en premier lieu créer un pointeur vers un objet GLUquadric, de cette façon : GLUquadric * notreQuadric;.

2) Il vous faut ensuite l'initialiser avec un appel à la fonction gluNewQuadric(), qui renvoie un quadric disponible : notreQuadric = gluNewQuadric().

3) Pour dessiner une sphère, un simple appel à gluSphere suffit. Cette fonction prend comme premier paramètre un quadric initialisé, en deuxième paramètre le rayon du quadric, et en troisième et quatrième paramètre des entiers stacks et slices. Pour rester simple, plus ces valeurs sont élevées, plus la sphère sera "ronde" et jolie, au détriment des performances évidemment. Par exemple : gluSphere (notreQuadric, notreRayon, 35, 35);

4) Enfin, il ne faut pas oublier de détruire le quadric une fois qu'il n'est plus utilisé afin d'éviter une fuite de mémoire : gluDeleteQuadric (notreQuadric);.

A noter que créer et afficher les sphères de cette façon est assez gourmand.

III-5 - Classe Objet, le retour !

Il manque en effet une chose très importante, notre callback ApplyForceAndTorqueCallback. Pour rappel, celle-ci a

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

été déclarée fonction amie de la classe abstraite Objet. Ainsi, celle-ci pour être appelée aussi bien par la classe Sphere que par la classe Boite, tout en ayant une syntaxe plus claire. Sans plus attendre, voici la définition de cette fonction :

Objet.cpp

```
void ApplyForceAndTorqueCallback (const NewtonBody * nBody)
{
    // Cette fonction est une fonction Callback. Elle sera appelée à chaque fois
    // qu'une modification aura lieu sur le corps.

    // On récupère en premier lieu la masse ainsi que l'inertie
    GLfloat masse; // Contiendra la masse de l'objet pris en paramètre par la fonction
    CVector inertie; // Contiendra l'inertie du corps
    CVector force; // Spécifiera la force appliquée sur le corps

    NewtonBodyGetMassMatrix (nBody, &masse, &inertie.x, &inertie.y, &inertie.z);

    force.x = 0.0f;
    force.y = -masse * 9.81; // 9.81 est l'attraction gravitationnelle de la Terre
    force.z = 0.0f;

    NewtonBodyAddForce (nBody, &force.x); // On ajoute la force au corps
}
```

Nous allons devoir les récupérer grâce à la fonction NewtonBodyGetMassMatrix, dont voici le prototype : void NewtonBodyGetMassMatrix (const NewtonBody * bodyPtr, float * mass, float * inertieX, float * inertieY, float * inertieZ);

Une fois toutes ces valeurs récupérées, nous pouvons y ajouter la force. x et z restent à 0, tandis que y est initialisée avec -masse * 9.81 où 9.81 est l'attraction gravitationnelle de la Terre. Vous pouvez facilement changer les comportements des objets avec ce chiffre (pour simuler par exemple le comportement physique sur d'autres planètes !). Et pour finir, nous ajoutons la force au corps passé en argument à la fonction avec NewtonBodyAddForce.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.
http://bakura.developpez.com/temp/test/

IV - Faisons marcher le tout !

Voilà, nous avons fini d'expliquer toutes nos classes, il nous reste maintenant à les faire fonctionner ensemble. Je passerai sur le passage concernant l'initialisation de la fenêtre avec la SDL et l'OpenGL, pour me concentrer sur les choses spécifiques à Newton. Je vous conseille d'ouvrir le fichier main.cpp afin de mieux suivre.

Nous allons commencer par créer quatre variables globales ainsi que deux valeurs constantes :

Main.cpp
```
const GLint NOMBRE_BOITES = 30;
const GLint NOMBRE_SPHERES = 30;

// Variables globales pour Newton
NewtonWorld * nWorld = NULL; // Monde
Sphere * spheres [NOMBRE_SPHERES];
Boite * sol = NULL; // Pointeur vers un objet sol représentant,... le sol !
Boite * boites [NOMBRE_BOITES]; // Tableau de pointeurs vers des objets Boîte

void InitNewton ();
```

Les deux premières variables spécifient le nombre de boites et de sphères affichés à l'écran. Les sphères créées avec GLU sont particulièrement gourmandes, si votre ordinateur est un peu vieux, je vous conseille de baisser ce chiffre à 10 pour obtenir un résultat fluide. Pour le reste, nous avons un pointeur vers un objet NewtonWorld, que nous initialisons à NULL, plusieurs pointeurs vers des objets de notre classe Sphere, un pointeur vers un objet Boite pour représenter le sol et plusieurs pointeurs vers des objets Boites (qui auront tous la même dimension). Notez également le prototype de la fonction InitNewton ();

La fonction que j'ai appelé InitOGL se charge d'initialiser les réglages spécifiques à OpenGL. Pour tout ce qui concerne éclairage et autre, regardez les sources. A la fin de cette fonction, nous appelons notre fonction InitNewton (). En voici la définition :

Main.cpp
```
void InitNewton ()
{
   nWorld = NewtonCreate (NULL, NULL); // On initialise nWorld

   // On définit le sol
   CVector taille (10.0f, 0.5f, 10.0f);
   CVector position (-5.0f, 0.0f, 0.0f);
   CVector couleur (100, 100, 100);

   // On alloue la mémoire. Le troisième argument est false pour signifier qu'il sera
   // immobile et donc ne bougera pas
   sol = new Boite ();
   sol->Initialiser (nWorld, taille, position, false, 0.0f);
   sol->SetColor (couleur);

   srand (time (NULL));

   // A présent le cube
   for (int i = 0 ; i < NOMBRE_SPHERES ; i++)
   {
      GLfloat x, y, z;

      x = -10 + rand()%15;
      y = 1 + rand()%10;
      z = -10 + rand()%15;

      position.ReglerCoordonnees (x, y, z);

      // On sélectionne les trois couleurs
      couleur.x = rand() % 256;
```


Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

Main.cpp

```
        couleur.y = rand() % 256;
        couleur.z = rand() % 256;

        spheres [i] = new Sphere ();
        spheres [i]->Initialiser (nWorld, 1.0f, position, true, 15.0f);
        spheres [i]->SetColor (couleur);
     }

     // A présent le cube
     for (int i = 0 ; i < NOMBRE_BOITES ; i++)
     {
        GLfloat x, y, z;

        x = -10 + rand()%15;
        y = 1 + rand()%10;
        z = -10 + rand()%15;

        taille.ReglerCoordonnees (1.0f, 1.0f, 1.0f);
        position.ReglerCoordonnees (x, y, z);

        // On sélectionne les trois couleurs
        couleur.x = rand() % 256;
        couleur.y = rand() % 256;
        couleur.z = rand() % 256;

        boites [i] = new Boite ();
        boites [i]->Initialiser (nWorld, taille, position, true, 10.0f);
        boites [i]->SetColor (couleur);
     }
  }
```

Cette fonction est moins complexe qu'elle n'en a l'air. Le monde de Newton est d'abord initialisé avec la fonction NewtonCreate. Elle prend en paramètre deux pointeurs de fonction callback pour allouer et désallouer de la mémoire, respectivement. En spécifiant les deux paramètres NULL, on laisse Newton utiliser les deux fonctions malloc et free. C'est ce que nous faisons ici. Attention à ne pas oublier cette appel, il est très important ! Un monde non initialisé provoquera un méchant plantage.

Après ça, on crée trois objets CVector : un pour la taille, un pour la position et un pour la couleur, que nous initialisons via le constructeur. Ces valeurs sont pour le sol, que nous initialisons juste après. Nous l'initialisons puis nous spécifions sa couleur. A noter l'avant-dernier paramètre, false, pour spécifier que l'objet sera immobile (et nous évitera donc les calculs non nécessaires).

La petite ligne srand (time (NULL)) permet de réinitialiser la graîne de la génération aléatoire. Puis nous initialisons un nombre NOMBRE_SPHERES d'objets sphères. Rien de bien compliqué. Trois variables x, y, z pour les positions. Nous les initialisons avec des valeurs comprises entre certaines bornes de façon qu'elles soient dans l'écran mais surtout au dessus du sol (le sol a une hauteur de 0.5, la position y de la sphère sera dans une intervalle [1;10]. On change les valeurs du vecteur position créé précédemment avec les nouvelles valeurs, puis on prend trois valeurs entre 0 et 255 pour les couleurs. Il ne nous reste plus qu'à créer la sphère avec new, l'initialiser avec 1.0f en rayon et en spécifiant true pour dire que la sphère sera en mouvement, puis on règle la couleur.

On fait de même pour le cube. Seul différence, on règle la taille de manière à ce que chaque cube face 1.0 de largeur de longueur et de profondeur.

Analysons à présent la fonction Prepare. Pour que la physique soit réaliste, il ne faut pas négliger ce passage. Il faut préciser exactement le temps qu'il s'est écoulé entre chaque appel de la fonction pour mettre à jour le monde et les objets de Newton.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.
http://bakura.developpez.com/temp/test/

Main.cpp

```
void Prepare ()
{
   NewtonUpdate (nWorld, 1.0f/80.0f);
}
```

On appelle la fonction NewtonUpdate pour mettre à jour tous les éléments. Le premier paramètre est un pointeur vers un monde Newton, tandis que le second paramètre est une valeur fixe qui doit être comprise entre 1.0/60.0 et 1.0/600.0. Ici, notre FPS est bloqué à 80 fps, donc nous effectuons le calcul 1.0/80.0.

Main.cpp

```
void Render ()
{
   glClearColor(0.0f, 0.0, 0.3, 0.0);
   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity ();

   gluLookAt (0.0f, 0.0f, 20.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);

   // Puis quelques rotations afin de bien voir le sol
   glRotatef(15, 1, 0, 0);
   glRotatef(45, 0, 1, 0);

   // On va d'abord dessiner le sol
   sol->Dessiner ();

   // Puis la boîte
   for (int i = 0 ; i < NOMBRE_BOITES ; i++)
   {
      boites [i]->Dessiner ();
   }

   for (int i = 0 ; i < NOMBRE_SPHERES ; i++)
   {
      spheres [i]->Dessiner ();
   }

   SDL_GL_SwapBuffers();
}
```

La fonction Render se charge de dessiner le sol, toutes les boîtes, et enfin toutes les sphères.

Main.cpp

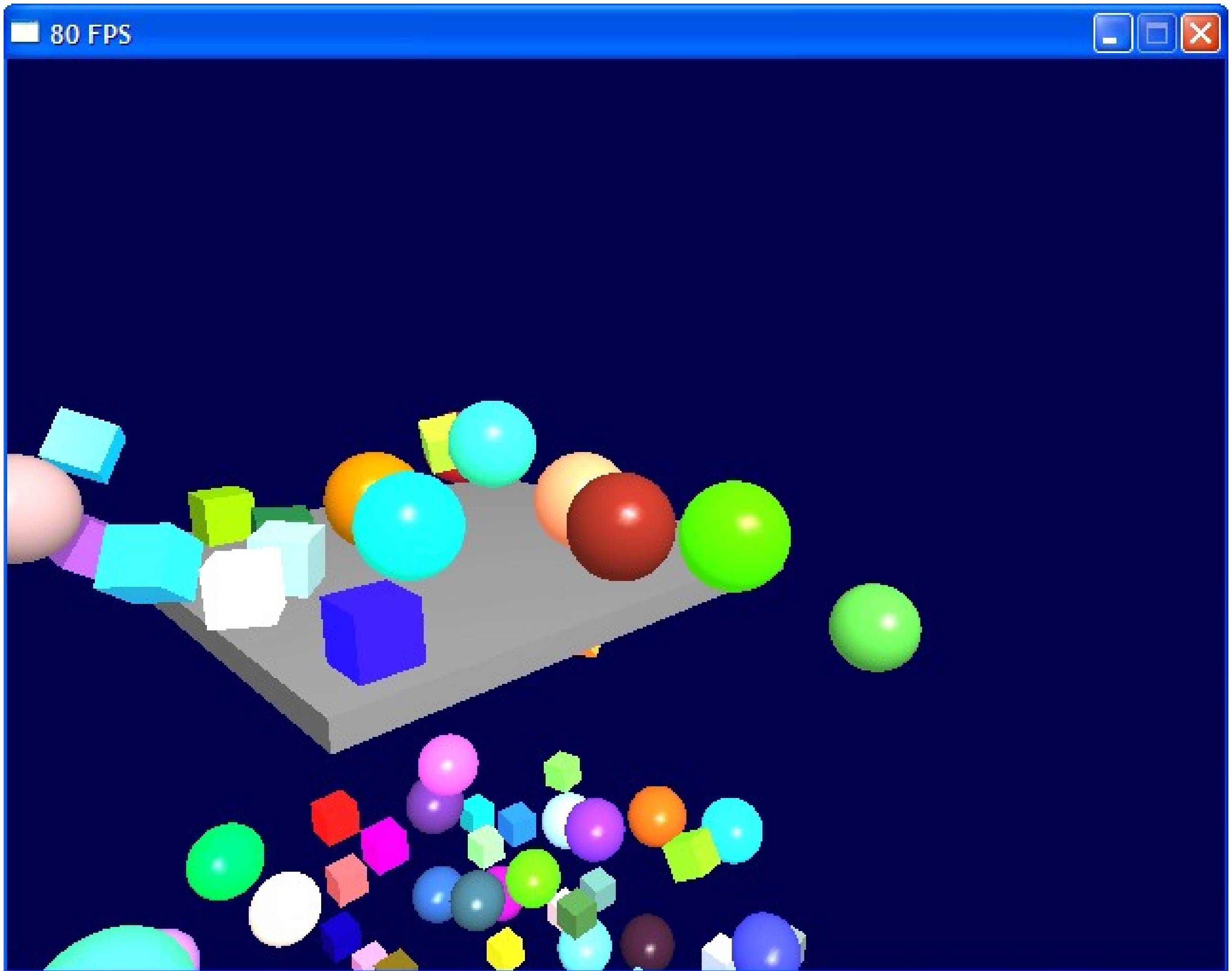
```
void Shutdown ()
{
   delete sol, sol = NULL;

   for (int i = 0 ; i < NOMBRE_BOITES ; i++)
      delete boites [i], boites [i] = NULL;

   for (int i = 0 ; i < NOMBRE_SPHERES ; i++)
      delete spheres [i], spheres [i] = NULL;

   NewtonDestroy (nWorld);
}
```

Reste la fonction Shutdown, à ne pas négliger ! On supprime d'abord le sol, puis toutes les boîtes, toutes les sphères, et enfin on détruit le monde avec la fonction NewtonDestroy.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

V - Conclusion

Et voilà le résultat de notre première application Newton :



Alors, c'était pas si dur, hein ? Maintenant à vous de jouer, cette bibliothèque propose bien d'autres fonctionnalités encore plus puissantes et impressionnantes.

Version PDF : ici (miroir)

Sources du programme : ici (miroir)

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.

VI - Remerciements

Merci à fearyourself pour ses nombreuses corrections, à Loulou pour sa relecture acharnée, ainsi qu'à Miles pour sa correction orthographique.

Ce document est issu de http://www.developpez.com et reste la propriété exclusive de son auteur. La copie, modification et/ou distribution par quelque moyen que ce soit est soumise à l'obtention préalable de l'autorisation de l'auteur.