

Newton Game Dynamics : les enveloppes convexes

par [Gallego Michaël](#)

Date de publication : 28/01/2007

Dernière mise à jour : 04/02/2007

Après avoir étudié les arbres de collision avec le moteur physique Newton Game Dynamics, nous nous intéresserons dans ce troisième tutoriel aux enveloppes convexes et les objets composés, qui permettent de définir des primitives de collision pour des modèles 3D complexes et, à la différence des arbres de collision, peuvent être soumis à toutes sortes de forces. Version de Newton Game Dynamics utilisée à l'écriture de ce tutoriel : 1.53

- I - Introduction
- II - Qu'est-ce qu'un modèle 3D ?
 - II-A - Le format ASE
 - II-A-1 - Section MATERIAL_LIST
 - II-A-2 - Section GEOMOBJECT
 - II-B - Conclusion
- III - Les classes
 - III-1 - La classe CPhysique
 - III-1-A - CPhysique.h
 - III-1-B - CPhysique.cpp
 - III-2 - La classe CPhysConvex
 - III-2-A - CPhysConvex.h
 - III-2-B - CPhysConvex.cpp
- IV - Faisons marcher le tout !
- V - Conclusion
- VI - Remerciements

I - Introduction

Le tutoriel précédent s'intéressait aux arbres de collision. Comme je l'avais répété maintes et maintes fois, l'utilisation première de ces arbres de collision est de les utiliser dans le contexte d'un niveau (heightmap,...) ou de modèles 3D fixes, car ils offrent une précision pour la détection collisions inégalables. De ce fait, ils ne sont pas du tout adaptés dans le cas de traitements physiques plus complexes (on ne peut pas leur appliquer des forces, et restent donc immobiles). Pour pallier à cet inconvénient, il existe dans Newton Game Dynamics les enveloppes convexes, ou convex hull en anglais, et les objets composés (compound en anglais), qui ne sont rien d'autres qu'un ensemble de plusieurs enveloppes convexes réunies dans une même primitive de collision.

Ce type de primitive de collision est particulièrement adapté dans le cas de modèles 3D devant être bougés (par exemple en réagissant à d'autres corps lorsqu'ils se touchent). On pourra donc utiliser les enveloppes convexes pour, par exemple, une chaise, une table, une roue, une hâche,... ou n'importe quel objet, pourvu qu'ils ne soit pas trop complexe.

Concrètement, quelle est la différence entre les arbres de collision et les enveloppes convexes ? Dans les arbres de collisions, comme vous avez pu le constater dans l'exemple du tutoriel n°2 en activant l'affichage le maillage de la primitive de collision, qu'elle épouse parfaitement la forme de l'objet, en reliant chaque vertice une à une, ce qui signifie que la géométrie de l'objet sera exactement celle de l'arbre de collision (à moins évidemment de "zapper" une vertice sur deux par exemple, ce qui entraînera un maillage moins précis). Dans tous les cas, la complexité des arbres de collision est importante, et il est aisé de comprendre que gérer plusieurs objets avec une telle précision serait bien trop coûteux en terme de performance ! Pour illustrer les enveloppes convexes, imaginez un objet, une arme par exemple. Prenez un papier cadeau et enveloppez ce papier cadeau tout autour de l'arme. Ainsi, seules les points qui ressortent le plus seront visibles. Et bien ce papier tout autour de l'arme, cela représentera l'enveloppe convexe. Et comme vous pouvez l'imaginer, ce "papier" tout autour de l'arme n'épousera pas parfaitement la géométrie de l'arme, même si elle en respectera le contour général. Ainsi, tout en produisant un résultat assez crédible, la géométrie de la primitive de collision sera bien moins complexe que celle des arbres de collision, ce qui confère donc aux enveloppes convexes la possibilité de leur appliquer des forces, de les bouger,...

Maintenant que vous avez une petite idée de ce que sont les enveloppes convexes, voilà comment j'ai décidé d'appliquer ce concept : tout en gardant l'architecture du précédent tutoriel afin de ne pas vous perdre, nous allons utiliser un loader de modèle 3D au format ASE (format que j'expliquerai brièvement dans la prochaine rubrique) que j'avais créé il y a quelques temps, ainsi qu'une nouvelle classe qui se chargera, à l'instar de la classe CPhysHeightmap du tutoriel précédent, de charger un modèle 3D puis de créer sa primitive de collision. Nous chargerons ainsi des cubes, en utilisant la même classe CPhysBox que d'habitude (que vous devez commencer à bien connaître !!), quelques chaises, des roues, ainsi que des armes, que nous ferons tomber sur un sol tout gris. Vous pouvez évidemment combiner tout ce que nous avons appris, en faisant tomber de tels objets sur un heightmap ! Mais pour garder le tout plus simple, j'ai décidé de ne pas faire ainsi. Pour tout le reste, rien de bien nouveau.

II - Qu'est-ce qu'un modèle 3D ?

Même si le but de ces tutoriaux n'est pas de vous apprendre à charger des modèles 3D, je pense qu'il est utile d'écrire quelques lignes sur le sujet, afin de mieux comprendre la suite. Donc qu'est-ce qu'un modèle 3D ? Un modèle 3D est constitué de plusieurs meshes, un mesh étant un objet en 3D composé de triangles ou de polygones. Un modèle 3D peut être constitué d'un seul mesh (pour un objet peu complexe par exemple), ou de plusieurs meshes, pour les modèles plus complexes. Par exemple, imaginez une voiture. Le graphiste peut la créer en un seul mesh, donc d'un seul bloc, ou bien, pour des raisons pratiques, en plusieurs meshes (un mesh représentera une roue, puis sera dupliqué en quatre, un autre mesh représentera la carrosserie,...). Le loader du format ASE que j'ai créé permet donc de charger un objet 3D composé d'un ou plusieurs meshes. Pour savoir ce que contient un fichier ASE, direction la rubrique suivante !

II-A - Le format ASE

Le format ASE est un format en texte, c'est à dire non binaire comme le format 3DS par exemple, et peut donc être lu avec le bloc-notes de Windows par exemple. Voici, pour que vous vous fassiez une idée, un exemple d'un modèle 3D d'une arme composé d'un seul mesh : [ici](#). Je précise que ce modèle est à la base un modèle 3DS, converti en ASE grâce à un plug-in sur Blender.

II-A-1 - Section MATERIAL_LIST

La première partie intéressante du fichier ASE est la section MATERIAL_LIST. Comme nous l'indique la ligne MATERIAL_COUNT, l'objet n'est composé que d'un mesh. S'il aurait été composé de deux meshes, vous auriez eu... deux (bravo !). Puis vient la ligne MATERIAL 0, qui est donc la liste de matériaux du mesh n°0 (le premier de la liste). De cette liste, je ne tire que les informations suivantes : MATERIAL_AMBIANT, MATERIAL_DIFFUSE, MATERIAL_SPECULAR et, un peu plus bas BITMAP.

Les trois premiers paramètres nous servent pour l'éclairage, et définissent comment la lumière va se comporter au contact de l'objet. Pour plus d'informations sur ceci, la FAQ vous sera d'une [grande aide](#). Quant à l'information BITMAP, elle définit tout simplement le nom de la texture à appliquer le mesh en question. Dans certains modèles, ce paramètre sera absent, cela voudra dire que ce mesh n'aura pas de texture.

II-A-2 - Section GEOMOBJECT

Et voici la section la plus importante, celle qui contient toutes les informations concernant la géométrie de l'objet. Le bloc se termine à la fin du fichier. Dans le cas d'un objet composé de plusieurs meshes, vous aurez donc plusieurs blocs GEOMOBJECT, mais ici, il n'y en a qu'un. Les premières informations ne nous intéressent pas, descendons directement au bloc MESH. Les premières informations contenues dans ce bloc vous permettent de savoir combien de sommets il y a, ainsi que le nombre de faces. Puis vient l'énumération de chaque vertice. Elles sont identifiées par un numéro (0, 1, 2, 3, 4,...) puis par trois valeurs, qui correspondent aux valeurs x, y, z. Rien de bien compliqué en somme. Descendez donc jusqu'au bloc MESH_FACE_LIST.

Prenons la première ligne : MESH_FACE 0: A:0 B:1 C:7 (la fin de la ligne ne nous intéresse guère). Ceci signifie que la face contient trois sommets (il s'agit donc... d'un triangle !). La première vertice est composée de la vertice n°0, la seconde vertice la vertice identifiée par le chiffre 1, et la troisième par le n°7. Bref, rien de bien sorcier là-dedans.

Vient ensuite la partie MESH_TVERTLIST, puis le bloc MESH_TFACELIST. Cela fonctionne sensiblement de la

même façon qu'avec les sommets, sauf qu'il s'agit des coordonnées de texture. A noter que les fichiers ne disposant pas de textures n'auront pas ces blocs (en théorie,...).

Enfin, dernier bloc et non le moindre, MESH_NORMALS. A l'intérieur, le bloc commence par MESH_FACENORMAL, suivi d'un identifiant puis de trois valeurs. A l'intérieur de ce bloc se trouve trois autres lignes, MESH_VERTEXNORMAL suivi d'un nombre puis de trois valeurs. Nous avons vu plus haut que la première face est constitué des sommets 0, 1, et 7. Ici, nous avons donc les valeurs des normales des sommets 0, 1 et 7. Rien de compliqué n'est-ce pas ? A noter que les trois valeurs qui suivent MESH_FACENORMAL permettent d'avoir un éclairage type flatshading, donc bien moins précis. C'est à dire que chaque vertice composant le triangle auront la même normal, au lieu d'avoir une normale différente.

Enfin, finissons ce tour d'horizon en descendant tout en bas, et, juste avant la fin du bloc GEOMOBJECT, remarquez la ligne MATERIAL_REF 0, ce qui signifie que ce mesh est affecté du matériel n°0.

Récapitulons. Ce fichier ASE exemple contient un modèle 3D composé d'un seul mesh, en témoignent un seul bloc GEOMOBJECT, et un seul matériel. Dans le cas de deux meshes, nous aurions donc eu deux matériaux, et deux blocs GEOMOBJECT. C'est clair ?

II-B - Conclusion

Bien sûr, ce petit tour d'horizon du format ASE n'est pas indispensable pour la suite, mais vu que j'utilise le format ASE dans le cadre de ce tutoriel, j'ai jugé bon d'en faire une petite explication. Bien sûr, le but n'est pas d'expliquer à fond le loader, et je vous propose d'aller y jeter un petit coup d'oeil. Le code est un peu documenté, et je l'ai fait de manière la plus simple possible. Seul point un petit peu délicat, j'utilise des vertex buffer objects (VBO) pour dessiner les modèles, mais une recherche rapide sur Google devrait vous renseigner sur les VBO.

III - Les classes

III-1 - La classe CPhysique

III-1-A - CPhysique.h

CPhysique.h

```

class CPhysique
{
public:
    // Constructeur / Destructeur
    CPhysique ();
    virtual ~CPhysique ();

    virtual void SetPosition (CVector & vPosition) const; // Pour régler la position de l'objet
    virtual void SetRotationX (const GLint rotatX) const; // Pour faire tourner un corps sur l'axe
X
    virtual void SetRotationY (const GLint rotatY) const; // Pour faire tourner un corps sur l'axe
Y
    virtual void SetRotationZ (const GLint rotatZ) const; // Pour faire tourner un corps sur l'axe
Z
    virtual void Render () = 0; // Fonction pure qui devra obligatoirement être définie dans
    // toutes les classes dérivées

protected:
    NewtonBody * m_pBody; // Un corps Newton
};

```

Vous devez déjà connaître cette classe puisque je l'ai utilisée pour le tutoriel précédent sur les arbres de collision. Il s'agit d'une classe abstraite, dont héritera toutes les autres classes liées à la physique. En effet, chaque objet Newton contient un pointeur vers un objet NewtonBody, a besoin d'une fonction Render (qui est déclarée pure et devra être obligatoirement redéfinie dans les classes dérivées, puisque chaque type objet se "dessine" différemment). Chaque objet Newton a également besoin d'une fonction pour régler sa position. En fait, la seule petite différence par rapport à la dernière fois, c'est qu'elle dispose de trois nouvelles fonctions : SetRotationX, SetRotationY, et SetRotationZ, chacune prenant en paramètre un angle de rotation.

L'utilité de ces fonctions est évidente : imaginez un modèle 3D. Vous pourrez très bien lui effectuer des rotations grâce à la fonction glRotate, mais la primitive de collision qui "englobe" le corps, elle, ne sera pas modifiée, et donc les collisions seront fausses. Pour effectuer la rotation sur la primitive de la collision, nous devons modifier la matrice du corps. Nous utiliserons ces fonctions de rotation pour que les objets soient lancés dans des positions différentes.

III-1-B - CPhysique.cpp

Voici à présent l'explication de ces fonctions :

CPhysique.cpp

```

// Fonction pour faire tourner un corps sur l'axe X
void CPhysique::SetRotationX (const GLint rotatX) const
{
    matrice maMatrice; // La matrice du corps
    matrice matriceModifiee; // La matrice à laquelle on va effectuer les rotations
    matriceModifiee.LoadIdentity ();
    NewtonBodyGetMatrix (m_pBody, &maMatrice.matrice [0][0]);

    GLfloat sinel = sin(rotatX * 3.1416 / 180);
    GLfloat cosinel = cos(rotatX * 3.1416 / 180);

```

CPhysique.cpp

```

// On applique les rotations à la matrice
matriceModifiee.matrice [1][1] = cosinel;
matriceModifiee.matrice [2][1] = -sinel;
matriceModifiee.matrice [1][2] = sinel;
matriceModifiee.matrice [2][2] = cosinel;

// Enfin on crée une dernière matrice, qui est la multiplication des deux précédentes
matrice newMat = matriceModifiee * maMatrice;

NewtonBodySetMatrix (m_pBody, &newMat.matrice [0][0]);
}

```

Dans un premier temps, on récupère la matrice du corps, on crée une nouvelle matrice, puis on calcul les cosinus et sinus des angles passés en paramètre. La multiplication par PI puis la division par 180 permettent de convertir en radians. Ensuite, on modifie juste la matrice de manière à effectuer une rotation sur celle-ci. Enfin, on crée une dernière matrice qui sera le résultat de la multiplication des deux premières. Pour plus de détails sur comment effectuer une rotation sur une matrice sur un axe X, Y et Z, je vous renvoie vers la FAQ qui vous expliquera ça tout bien : [ici](#). Pour terminer, on applique la matrice modifiée au corps via la fonction NewtonBodySetMatrix.

III-2 - La classe CPhysConvex

III-2-A - CPhysConvex.h

CPhysConvex.h

```

#ifndef CPHYS CONVEX_H
#define CPHYS CONVEX_H

#include "cphysique.h"
#include "cmodel.h"

class CPhysConvex : public CPhysique
{
public:
    CPhysConvex ();
    virtual ~CPhysConvex ();

    bool CreerObjetConvexe (NewtonWorld * nWorld, CModel * pModel); // Fonction pour créer la
boîte
    void SetMasse (GLfloat fMasse); // Pour régler la masse

    void Render (); // Fonction Render, pour dessiner

protected:
    CVector m_vLongueur; // Longueurs du cube
    CVector m_vCouleur; // Sa couleur

    CModel * m_pModel; // Pointeur vers un modèle 3D
};

#endif // CPHYS CONVEX_H

```

Je pense que vous commencez dorénavant à être habituer à ma façon de procéder. La classe CPhysConvex hérite de CPhysique et dispose donc de ses fonctions et variables membres. La fonction permettant de créer la primitive de collision se nomme CreerObjetConvexe. Le premier paramètre est le fameux pointeur vers le monde Newton et, petite touche d'exotisme, le deuxième paramètre est un pointeur vers un objet CModel, qui est donc un pointeur vers mon loader de fichier ASE (pour information, ma classe CModel possède des pointeurs vers la classe CMesh, car, comme je l'ai expliqué dans la section II, un modèle 3D peut-être constitué d'un ou plusieurs meshes). L'habituelle fonction SetMasse est toujours de la partie, tout comme la fonction Render ().

Au niveau des variables membres, notons le pointeur vers l'objet CModel, nommé m_pModel.

III-2-B - CPhysConvex.cpp

CPhysConvex.cpp

```
// Fonction pour créer la boîte de collision de la boîte
bool CPhysConvex::CreerObjetConvexe (NewtonWorld * nWorld, CModel * pModel)
{
    // On commence par charger notre modèle 3D grâce au loader ASE
    m_pModel = pModel;

    // Comme dans le tutoriel n°1, nous créons une primitive de collision
    NewtonCollision * nCollision = NULL;
```

Voici le début de la fonction CreerObjetConvexe. Pour l'instant, rien de nouveau, si ce n'est que nous initialisons notre variable m_pModel avec la variable pModel passée en paramètre. Ceci nous permet de charger le modèle dans le main une seule fois et de le passer comme paramètre à tous les objets qui utilisent ce modèle 3D, plutôt que de le créer à chaque fois de nouveau. Puis nous créons une variable nCollision que nous initialisons à NULL, comme toujours.

CPhysConvex.cpp

```
// Pour stocker chaque enveloppe convexe dans une primitive de collision finale
// les englobant toutes (notre variable nCollision), on crée un vector composé
// de plusieurs primitives de collision
std::vector <NewtonCollision *> enveloppeConvexe;

// On parcourt chaque mesh du modèle. Pour plus de détails, voir tutoriel sur
// developpez.com
for (GLint i = 0 ; i < m_pModel->GetNumMeshs () ; ++i)
{
    CMesh * tmpMesh = m_pModel->GetMesh (i); // On récupère un pointeur vers le mesh n°i
    GLint nbsommets = tmpMesh->GetNumsommets (); // On récupère le nombre de sommets du ledit mesh
    std::vector <aseVertex> sTmpVert = tmpMesh->Getsommets (); // Enfin, on récupère tous les
    sommets de ce mesh

    // Puis on crée notre primitive de collision avec les coordonnées passées à la
    // fonction NewtonCreateConvexHull
    nCollision = NewtonCreateConvexHull (nWorld, nbsommets, &sTmpVert[0].x, sizeof (GLfloat) * 3,
    NULL);

    // On remplit ensuite notre vector de NewtonCollision avec la primitive de collision
    // précédemment remplie
    enveloppeConvexe.push_back (nCollision);
}

// Puis on attribue à notre variable nCollision la primitive de collision finale,
// constituée de plusieurs enveloppes convexes
nCollision = NewtonCreateCompoundCollision (nWorld, m_pModel->GetNumMeshs (),
&enveloppeConvexe[0]);
```

Et voici ce que vous attendez tous, la nouveauté de ce tutoriel ! Il y a donc un peu plus à dire. Pour créer des primitives d'objets complexes à l'aide des enveloppes convexes, la bibliothèque nous donne accès à deux fonctions principales, dont voici les prototypes :

1) NewtonCollision * NewtonCreateConvexHull (const NewtonWorld * newtonWorld, int nbsommets, const float * tableauxommets, int strideInBytes, const float * offsetMatrix)

2) NewtonCollision * NewtonCreateCompoundCollision (const NewtonWorld * newtonWorld, int nbCollisions, NewtonCollision * const tableauCollisions[])

En fait, les deux sont très liés. `NewtonCreateCompoundCollision` n'est qu'une "extension" de `NewtonCreateConvexHull`. Étudions la première fonction : le premier paramètre est un pointeur vers le monde Newton, le second paramètre est un entier représentant le nombre total de vertice, le troisième un tableau contenant chaque sommet, le quatrième la taille en octets d'une vertice, tandis que le dernier paramètre est un pointeur vers un tableau de 16 floats contenant le volume de collision de l'objet. Comme d'habitude, laissez le sur `NULL` pour que le volume de collision soit centré au milieu.

Pour faire simple, cette fonction permet de créer la primitive de collision pour un mesh. Or, un modèle 3D peut-être composé de plus d'un mesh. C'est là qu'intervient la seconde fonction. Le premier paramètre est un pointeur vers un monde Newton, le deuxième le nombre de pointeurs d'objets `NewtonCollision` à stocker, et le troisième un tableau contenant chaque `NewtonCollision`. La primitive de collision renvoyée est donc un objet `NewtonCollision` "global". Ceci permet donc de créer facilement des primitives de collision pour des modèles composés de plusieurs meshes ! Bref, quand vous êtes sûr que le modèle 3D n'est composé que d'un seul mesh, il est plus simple d'utiliser directement la première fonction. Quand vous ne savez pas, vous devez utiliser la seconde.

Maintenant que vous comprenez bien ces fonctions, étudions notre code ! On commence donc à créer un vector de `NewtonCollision` *. Puis nous créons une boucle `for` pour parcourir chaque mesh. Dans la boucle, nous créons un pointeur vers un objet `CMesh` (je rappelle qu'un objet 3D `CModel` est constitué de plusieurs meshes, donc de plusieurs objets `CMesh`), qu'on initialise avec la valeur de retour de la fonction `m_pModel->GetMesh (i)`. Cette fonction se charge tout simplement de renvoyer le pointeur vers le mesh identifié par la valeur `i` de la boucle. On récupère ensuite le nombre de sommets du mesh grâce à la fonction `GetNumsommets ()`. Nous créons ensuite un vector de `aseVertex` (il s'agit d'une structure très simple de mon loader qui en fait contient trois entiers flottants représentant les coordonnées `x, y, z`) que nous initialisons avec ce que renvoie la fonction `Getsommets`. Le vector `sTmpVert` contiendra donc toutes les sommets du mesh.

Une fois toutes les valeurs nécessaires à la création de l'enveloppe convexe récupérées, on initialise la variable `nCollision` avec la fonction `NewtonCreateConvexHull`. Enfin, nous ajoutons la primitive de collision à notre vector, puis on réitère jusqu'à ce que toutes les primitives de collisions de chaque mesh du modèle aient été insérées dans notre vector.

Une fois la boucle `for` terminée et donc notre tableau rempli, nous créons notre primitive de collision finale grâce à la fonction `NewtonCreateCompoundCollision`.

CPhysConvex.cpp

```
m_pBody = NewtonCreateBody (nWorld, nCollision);

if (m_pBody == NULL)
{
    std::cerr << "Impossible de créer le corps Newton";
    return EXIT_FAILURE;
}

// On libère chaque primitive de collision contenue dans le vector
std::for_each (enveloppeConvexe.begin (), enveloppeConvexe.end (),
bind1st(ptr_fun(&NewtonReleaseCollision), nWorld));

NewtonReleaseCollision (nWorld, nCollision); // On libère la primitive de collision

// On assigne la matrice identité pour la position du corps. Pour la changer, il
// faut passer par la fonction SetPosition
matrice maMatrice;
maMatrice.LoadIdentity ();

NewtonBodySetMatrix (m_pBody, &maMatrice.matrice [0][0]);

return EXIT_SUCCESS; // Tout s'est bien déroulé
}
```

Une fois notre primitive de collision finale créée, on initialise comme d'habitude notre corps NewtonBody avec la fonction NewtonCreateBody. Enfin, il ne faut pas oublier de libérer la mémoire pour ne pas créer de fuites. On libère tout d'abord tous les NewtonCollision * contenus dans le vector enveloppeConvexe grâce à la fonction for_each, puis la collision finale nCollision. Et pour finir, on règle la matrice de transformation de l'objet à la matrice identité.

CPhysConvex.cpp

```
// Fonction pour régler la masse
void CPhysConvex::SetMasse (GLfloat fMasse)
{
    CVector origine;
    CVector inertie;
    GLfloat Ixx;
    GLfloat Iyy;
    GLfloat Izz;

    NewtonCollision * nCollision = NewtonBodyGetCollision (m_pBody);

    // On calcul l'inertie du corps, ainsi que le centre de la masse
    NewtonConvexCollisionCalculateInertialMatrix (nCollision, &inertie.x, &origine.x);
    NewtonBodySetCentreOfMass (m_pBody, &origine.x);

    // On multiplie les valeurs d'inertie par la masse passée en paramètre
    Ixx = fMasse * inertie.x;
    Iyy = fMasse * inertie.y;
    Izz = fMasse * inertie.z;

    NewtonBodySetMassMatrix (m_pBody, fMasse, Ixx, Iyy, Izz);

    // On spécifie le callback
    NewtonBodySetForceAndTorqueCallback (m_pBody, ForceAndTorqueCallback);

    NewtonBodySetLinearDamping (m_pBody, 0.5f);
}
```

Voici ici la fonction SetMasse, qui diffère un peu de celle que nous avons écrite les fois dernières. En effet, les enveloppes convexes représentant des objets 3D assez complexes, le centre d'inertie de l'objet ne sera pas le centre (comme ce sera le cas sur une sphère ou une boîte). Pour avoir un résultat plus réaliste, la fonction NewtonConvexCollisionCalculateInertialMatrix nous vient en aide ! Le premier paramètre est un pointeur vers une primitive de collision (que l'on récupère juste avant grâce à la fonction NewtonBodyGetCollision), le deuxième paramètre est un pointeur de 3 entiers flottants représentant les valeurs de l'inertie, tandis que le dernier paramètre est lui aussi un pointeur de 3 entiers flottants représentant le centre d'inertie. On récupère donc tout ce joli monde, puis on appelle la fonction NewtonBodySetCentreOfMass, qui permet de spécifier le centre d'inertie, avec les valeurs stockées ici dans notre objet origine, récupérées auparavant. Enfin, on multiplie les trois valeurs de l'inertie avec la masse passée en argument. Comme d'habitude, on appelle la fonction NewtonBodySetMassMatrix en lui spécifiant la masse et les valeurs d'inertie. On donne ensuite au corps un callback via NewtonBodySetForceAndTorqueCallback (vous savez, le callback pour ajouter des forces, comme la force gravitationnelle, et donc faire "tomber" les objets !). Et pour finir, on spécifie une force de frottement via la fonction NewtonBodySetLinearDamping. Cette force est ajoutée aux forces extérieures appliquées au corps. Cette force est proportionnelle au carré de la magnitude de la vitesse du corps dans la direction opposée à la vitesse du corps (ouf !). En clair, cette force permet de faire en sorte que le corps perde de la vitesse (imaginez une sphère que vous faites rouler sur une table. Sa vitesse diminuera jusqu'à l'arrêt, et ceci grâce à une force de frottement. Pour revenir à la fonction NewtonBodySetLinearDamping, ses valeurs peuvent être comprises entre 0.0 et 1.0. Pour être complet, la fonction NewtonBodySetAngularDamping fait de même mais pour ajouter une force angulaire, ce qui aura pour effet de diminuer petit à petit la vitesse de rotation.

CPhysConvex.cpp

```
void CPhysConvex::Render ()
{
    // Toutes les modifications effectuées dans le Callback modifient la matrice de
    // l'objet, ce qui permet à l'objet de "bouger"
    matrice maMatrice;

    NewtonBodyGetMatrix (m_pBody, &maMatrice.matrice [0][0]);
}
```

CPhysConvex.cpp

```
glPushMatrix (); // On sauvegarde la matrice actuelle
glMultMatrixf (&maMatrice.matrice [0][0]); // On multiplie la matrice actuelle
// par la matrice du corps, ainsi
// le corps sera dessiné au bon endroit
m_pModel->Dessiner (); // On dessine le modele
glPopMatrix (); // On rétablit la matrice
}
```

La fonction Render ne propose rien de nouveau, à part peut-être l'appel à la fonction Dessiner de notre objet m_pModel.

IV - Faisons marcher le tout !

Comme à mon habitude, voici à présent une petite description de la fonction main, même si elle ne propose pas de grands bouleversements.

main.cpp

```
// On crée un modèle 3D, qu'on passera à la fonction, plutôt que de créer plusieurs
// fois le même objet
CModel * pModeleRoue = new CModel ();
CModel * pModeleArme = new CModel ();
CModel * pModeleChaise = new CModel ();

if (!pModeleRoue->OuvrirFichier ("roue2.ase"))
    std::cerr << "Impossible d'ouvrir roue.ase";

if (!pModeleArme->OuvrirFichier ("arme.ase"))
    std::cerr << "Impossible d'ouvrir arme.ase";

if (!pModeleChaise->OuvrirFichier ("chaise.ase"))
    std::cerr << "Impossible d'ouvrir chaise.ase";

for (GLuint i = 0 ; i < NOMBRE_ROUES ; ++i)
{
    GLfloat x, y, z;
    x = -20 + rand()%50;
    y = 5 + rand()%20;
    z = -30 + rand()%50;

    vPosition.SetCoordonnees (x, y, z);

    pConvexesRoues [i] = new CPhysConvex ();
    pConvexesRoues [i]->CreerObjetConvexe (nWorld, pModeleRoue);
    pConvexesRoues [i]->SetMasse (10.0);
    pConvexesRoues [i]->SetPosition (vPosition);
    pConvexesRoues [i]->SetRotationX ((GLuint)x);
    pConvexesRoues [i]->SetRotationY ((GLuint)y);
    pConvexesRoues [i]->SetRotationZ ((GLuint)z);
}
```

Puisque le programme charge trois modèles 3D différents (une arme, une roue et une chaise), on commence par les charger en passant à la fonction OuvrirFichier le nom du fichier ASE en question. Puis on rentre dans une boucle for pour créer le nombre adéquats d'objets. La seule nouveauté sont les trois lignes SetRotationX, SetRotationY, SetRotationZ. On leur passe la même valeur x, y et z que celles pour la position, ce qui entraînera que les objets démarreront leurs chutes avec une orientation différente.

Pour les autres modèles, cela fonctionne exactement de la même façon.

main.cpp

```
// Fonction Render
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, 40.0f, 0.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f);

    // Puis quelques rotations afin de bien voir le sol
    glRotatef(35, 1, 0, 0);
    glRotatef(45, 0, 1, 0);

    if (bDebugMode)
        DebugShowCollision ();

    pSol->Render (); // On dessine le sol

    for (GLuint i = 0 ; i < NOMBRE_BOITES ; ++i)
```

main.cpp

```
pBoites [i]->Render ();

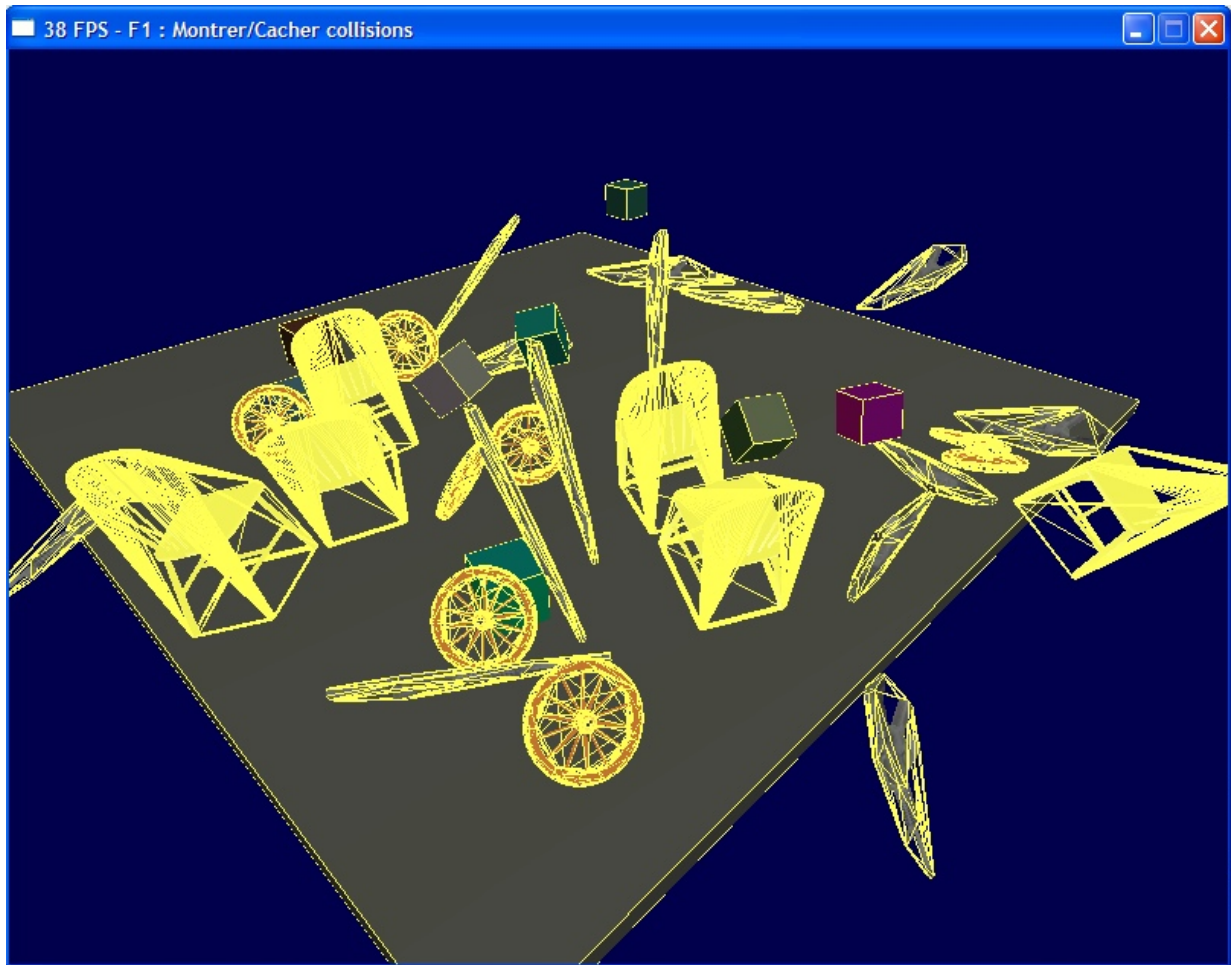
for (GLuint i = 0 ; i < NOMBRE_PISTOLETS ; ++i)
    pConvexesPistolets [i]->Render ();

for (GLuint i = 0 ; i < NOMBRE_ROUES ; ++i)
    pConvexesRoues [i]->Render ();

for (GLuint i = 0 ; i < NOMBRE_CHAISES ; ++i)
    pConvexesChaises [i]->Render ();

    SDL_GL_SwapBuffers();
}
```

Et pour finir ce main en beauté, la fonction Render qui dessine chaque objet, avec toujours la possibilité de dessiner la géométrie des primitives de collision en appuyant sur F1.



Avec mode debug

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

VI - Remerciements

Je remercie Laurent Gomila et fearyourself pour leurs corrections, as usual ^^.