

Newton Game Dynamics : les Collision Tree



par [Gallego Michaël](#)

Date de publication : 26/12/2006

Dernière mise à jour : 26/12/2006

Dans ce second tutoriel consacré au moteur physique Newton Game Dynamics, nous nous attacherons à découvrir les Collision Tree, qui permettent de faire des collisions sur des environnements extrêmement complexes (modèles 3D, heightmap,...).

- I - Introduction
- II - Quelques éléments supplémentaires
- III - Les classes
 - III-1 - La classe CPhysHeightmap
 - III-1-A - CPhysHeightmap.h
 - III-1-B - CPhysHeightmap.cpp
- IV - Faisons marcher le tout !
- V - Conclusion
- VI - Remerciements

I - Introduction

Dans le tutoriel précédent, vous avez découvert comment fonctionne le moteur Newton Game Dynamics pour permettre à des formes géométriques simples (sphères ou boîtes par exemple) d'interagir entre elles de manières réalistes. Toutefois, comment faire dans le cas d'environnements complexes ? Bien sûr, on pourrait diviser un mesh complexe en plusieurs primitives de bases (sphères, boîtes), mais cela s'avérerait extrêmement fastidieux et plutôt compliqué. Pour faire face à ce problème, le développeur de la librairie (car je crois qu'il est tout seul, son travail est à saluer !) a instauré les collision tree. Les collision tree permettent de définir des primitives de collision bien plus complexes en passant au moteur les coordonnées des vertices du modèle.

L'un des gros inconvénients des collision tree est que, compte tenu de leur complexité, ils doivent rester immobiles. C'est à dire qu'on ne peut leur donner ni vitesse, ni force,... Il existe bien sûr des techniques "détournées" qui permettent de donner l'illusion de les déplacer (on peut par exemple effectuer une rotation sur le modèle 3D, et donner translation, forces,... adéquates à tous les autres corps). Toutefois, pour attribuer des primitives de collision complexes à des modèles 3D, il existe un autre moyen - bien que moins précis - , que nous verrons dans un prochain tutoriel : les convex hulls et les compound (qui contiennent plusieurs convex hulls).

Mais revenons à nos collision tree. Je disais donc qu'ils devront rester immobiles, tout du moins dans la version de Newton avec laquelle j'écris cet article. Toutefois, la limitation actuelle des collision tree n'en est pas vraiment une, puisque leur but premier est de définir des primitives de collision complexes pour des niveaux (un terrain,...), des modèles 3D,... et qui ne bougent donc normalement pas.

Par rapport au précédent tutoriel, j'ai revu l'organisation des classes afin que ça soit plus aisément compréhensible et plus facile à gérer. Je ne reviendrai pas sur ce que nous avons vu précédemment, ou que brièvement. Concernant la mise en application de ce concept, j'ai choisi d'utiliser un heightmap, qui représentera notre sol, et des sphères et des boîtes qui tomberont sur le sol.

II - Quelques éléments supplémentaires

Avant de rentrer dans le vif du sujet, et pour ceux ayant un peu la flemme de lire la doc, voici les fonctions pour créer des primitives de collisions pour des formes de base :

1) NewtonCollision * NewtonCreateCone (const NewtonWorld * nWorld, float rayon, float hauteur, const float * offsetMatrix);

Elle permet, comme son nom l'indique, de créer une primitive de collision pour un cône. Le premier paramètre est un pointeur vers un objet NewtonWorld valide, le deuxième élément la valeur du rayon de la base, le troisième paramètre la hauteur à partir de la base et le dernier élément un pointeur vers un tableau de 16 floats contenant le centre de la matrice du corps. Je pense que ça correspond au centre de gravité du corps. En laissant NULL comme je le fais dans les tutoriaux, la matrice sera automatiquement réglée à l'origine du corps.

2) NewtonCollision * NewtonCreateCapsule (const NewtonWorld * nWorld, float rayon, float hauteur, const float * offsetMatrix);

Pas besoin d'explications je pense.

3) NewtonCollision * NewtonCreateCylinder (const NewtonWorld * nWorld, float rayon, float hauteur, const float * offsetMatrix);

Pareil, mais pour créer un cylindre !

4) NewtonCollision * NewtonCreateChamferCylinder (const NewtonWorld * nWorld, float rayon, float hauteur, const float * offsetMatrix);

Pour créer un cylindre chamfer.

III - Les classes

Comme je l'ai dit en introduction, j'ai remanié légèrement l'architecture des classes par rapport au premier tutoriel. On retrouve donc la classe CVector, qui elle n'a pas bougé d'un iota, la classe CPhysique, CPhysSphere, CPhysBox et la petite nouveauté, CPhysHeightmap.

La classe CPhysique est donc la classe de base des trois autres. Elle possède comme donnée membre un pointeur vers un objet NewtonWorld. Concernant ses fonctions membres, on retrouve un constructeur et destructeur, une fonction SetPosition qui modifie la matrice de l'objet, et une fonction Render déclarée pure, qui devra donc être définie dans chacune de ses classes de base. A noter comme pour le premier tutoriel une structure matrice simplifiée, qui comprend juste en plus une fonction LoadIdentity, afin de l'initialiser à la matrice identité plus facilement.

La classe CPhysBox dérive de la classe CPhysique et a donc accès à ses variables et fonctions membres. Contrairement au premier tutoriel, j'ai décidé de diviser la phase d'initialisation en plusieurs phases. Ainsi, la fonction CreerBoite, qui prend en argument un pointeur vers un objet NewtonWorld et une longueur, se charge seulement de créer la primitive de collision, sans lui affecter de callback. Pour ceci, il faut appeler la fonction SetMasse en lui passant en argument une masse, afin de lui affecter un callback. De cette manière, si vous souhaitez utiliser les capacités du moteur juste pour les collisions, il vous suffit d'appeler la fonction CreerBoite. Si en plus vous voulez lui affecter des forces, vous le pouvez en appelant la fonction SetMasse. A part ça, rien de bien nouveau.

La classe CPhysSphere est l'équivalent de la classe CPhysBox, mais pour gérer les sphères. Là aussi, à part le changement d'organisation de la classe, rien de neuf.

III-1 - La classe CPhysHeightmap

III-1-A - CPhysHeightmap.h

CPhysHeightmap.h

```
#ifndef CPHYSHEIGHTMAP_H
#define CPHYSHEIGHTMAP_H

#include <string>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/corona.h>
#include "cphysique.h"

// Quelques données relatives au Heightmap
const GLuint TAILLE_MAP = 257; // La largeur et la longueur de la map
const GLuint iPrecision = 5; // Précision de la map

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

    // Fonction qui va charger le Heightmap et créer le collision tree
    bool CreerHeightmap (NewtonWorld * nWorld, std::string szNom);
    GLfloat LireHauteur (GLuint x, GLuint z); // Fonction pour lire la hauteur

    void Render (); // Fonction Render
};
```

CPhysHeightmap.h

```

protected:
    GLubyte m_ubHeightmap [TAILLE_MAP * TAILLE_MAP]; // Tableau contenant les données du Heightmap
    GLuint m_uiTexture;
};

#endif // CPHYSHEIGHTMAP_H
};

```

Voici ci-dessus l'header de notre classe CPhysHeightmap. Dans les headers inclus, seul GL/corona.h peut vous surprendre. En fait, il s'agit tout simplement d'une librairie très simple d'utilisation pour charger des images de plusieurs formats (j'ai en effet appliqué une texture à notre heightmap). Vous pouvez télécharger cette librairie à [cette adresse](#).

Un peu plus bas, deux variables globales définissant la taille de notre map et là précision du rendu. L'objectif de ce tutoriel n'est pas de vous expliquer comment fonctionne un heightmap (ce n'est pas bien compliqué de toute façon), je ne m'y attarderai donc pas trop. Sachez juste que le heightmap est une image en niveau de gris, chaque point représentant une hauteur. Plus le point est blanc, plus la hauteur sera grande, au contraire plus le point est noir, plus la hauteur sera basse. Les hauteurs seront donc stockées dans un tableau (ici m_ubHeightmap) de valeurs comprises entre 0 (noir complet), et 255 (blanc complet). Pour dessiner le heightmap, il nous suffira donc à itérer dans le tableau, et de récupérer la hauteur relative aux valeurs x et z. Si ceci vous semble confus, lisez attentivement le code, vous comprendrez sans aucun doute !

La fonction CreerHeightmap prend en paramètre un pointeur vers un NewtonWorld ainsi qu'une string contenant le nom du heightmap à charger. La fonction LireHauteur a pour but de récupérer la hauteur du heightmap de deux points, et la fonction Render se charge de dessiner le heightmap.

Concernant les données membres, on retrouve un tableau pour stocker les valeurs du heightmap, et un unsigned int pour l'identifiant de la texture OpenGL.

III-1-B - CPhysHeightmap.cpp

Passons à présent à la définition de cette classe. Le constructeur se charge d'appeler le constructeur de la classe de base, le destructeur pour sa part détruit la texture. Je vous laisse regarder par vous même la fonction LireHauteur, et intéressons nous à la fonction CreerHeightmap.

CPhysHeightmap.cpp

```

// Création du heightmap et du collision tree
bool CPhysHeightmap::CreerHeightmap (NewtonWorld * nWorld, std::string szNom)
{
    FILE * pFile = NULL; // On crée un pointeur vers nouveau fichier

    pFile = fopen (szNom.c_str(), "rb"); // On ouvre le fichier .RAW

    if (!pFile)
    {
        std::cerr << "Erreur dans l'ouverture du fichier heightmap.";
        return EXIT_FAILURE;
    }

    fread (&m_ubHeightmap, TAILLE_MAP * TAILLE_MAP, 1, pFile); // On copie dans le tableau un nombre
nTaille de 1 byte // à partir du fichier pFile

    fclose (pFile); // On n'a plus besoin de ce fichier, on le ferme

```

On crée un pointeur pFile, on ouvre le fichier RAW, et on le lit grâce à la fonction fread, et enfin on ferme le fichier avec la fonction fclose.

CPhysHeightmap.cpp

```
// On créé le collision tree
NewtonCollision * nCollision = NULL;

// Pour plus d'explications sur la création du collision tree, voir le tuto
// sur le site
nCollision = NewtonCreateTreeCollision (nWorld, NULL);

NewtonTreeCollisionBeginBuild (nCollision);

GLfloat fVert[9];
GLint strideInBytes = sizeof(GLfloat) * 3;

for (GLuint x = 0 ; x < TAILLE_MAP - 1 ; x += iPrecision)
{
    for (GLuint z = 0 ; z < TAILLE_MAP - 1 ; z += iPrecision)
    {
        fVert [0] = x;
        fVert [1] = LireHauteur (x, z) / 10.0f;
        fVert [2] = z;

        fVert [3] = x;
        fVert [4] = LireHauteur (x, z + iPrecision) / 10.0f;
        fVert [5] = z + iPrecision;

        fVert [6] = x + iPrecision;
        fVert [7] = LireHauteur (x + iPrecision, z) / 10.0f;
        fVert [8] = z;

        NewtonTreeCollisionAddFace (nCollision, 3, &fVert [0], strideInBytes, 0);
    }
}

NewtonTreeCollisionEndBuild (nCollision, 1); // 1 = optimisation

m_pBody = NewtonCreateBody (nWorld, nCollision);

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

NewtonReleaseCollision (nWorld, nCollision); // On libère le collision tree
```

Voici la création du collision tree à proprement parler. Respirez profondément, et on y va ! Comme d'habitude, on crée un pointeur vers un objet NewtonCollision, qu'on initialise à NULL. Dans la ligne suivante, on initialise notre pointeur avec un appel à la fonction NewtonCreateTreeCollision. Le premier paramètre est le pointeur vers un monde Newton valide, et le deuxième est un pointeur de fonction vers un callback pour le tree collision. Laissez le à NULL.

On commence la création du collision tree avec la fonction NewtonTreeCollisionBeginBuild, avec comme paramètre le pointeur vers l'objet NewtonCollision. Il ne nous reste plus qu'à remplir notre collision tree. Pour ce faire, il nous faudra appeler la fonction NewtonTreeCollisionAddFace, qui prend comme premier paramètre notre pointeur vers l'objet NewtonCollision, en second paramètre le nombre de vertices contenues dans le troisième paramètre, qui est un pointeur vers un tableau de vertices. Chaque vertice doit contenir au moins trois valeurs flottantes (x, y, z par exemple). Le quatrième paramètre de cette fonction est la taille, en byte, de chaque vertice. D'après la documentation, cette valeur doit être égale à une valeur de 12 ou supérieure. Le dernier et cinquième paramètre est un identifiant du polygone. Dans cet exemple, cette valeur est de 0.

Avant de remplir notre collision tree et conformément aux arguments que prend la fonction, on crée un tableau de

9 floats, appelé fVert. En effet, nous allons remplir notre collision tree avec des triangles, et chaque triangle étant composé de 3 vertice de 3 valeurs x, y, z chacun, on se retrouve bien avec 9 floats. La variable strideInBytes est de trois fois la taille d'un float, puisque chaque vertice contient trois float.

On passe ensuite à la partie plus délicate : le remplissage du collision tree. On parcourt le heightmap avec deux boucles, qui sont incrémentés à chaque fois de la valeur iPrecision, afin de ne pas trop perdre en performance compte tenu de la complexité du heightmap. A noter que plus la valeur iPrecision sera basse, plus le collision tree sera précis, et plus les collisions seront précises, au détriment de la performance. Essayez donc d'augmenter la valeur d'iPrecision à 16 ou 32 : la majorité des sphères passeront au travers du collision tree ; tandis qu'en réglant à 1, la valeur la plus précise, les collisions seront extrêmement précises, mais le fps en pâtira pas mal ! J'ai choisi une valeur de 5 pour cet exemple, ce qui permet un bon compromis précision/performance.

Le tableau fVert est ensuite rempli, les trois premières valeurs correspondent à la première vertice du triangle, les trois suivantes à la seconde et les trois dernières à la dernière vertice. A noter que la valeur renvoyée par LireHauteur est divisé par 10. En effet, comme je l'ai dit plus haut, la hauteur du heightmap est en faite la couleur représentée dans le fichier RAW, et celle-ci peut s'étalonner de 0 à 255, et une hauteur de 255 serait bien trop !

Une fois le tableau rempli, on appelle la fonction NewtonTreeCollisionAddFace. Notre collision tree créé, il faut le finaliser en appelant la fonction NewtonTreeCollisionEndBuild. Le deuxième paramètre est une valeur, soit 0 ou 1. 1 laisse Newton optimiser la géométrie du collision tree en supprimant notamment les vertices redondantes. Même si ici l'optimisation n'a que peu d'effets, sur certains modèles 3D, la différence que j'ai remarquée était impressionnante !

La ligne suivante créé notre corps avec la fonction NewtonCreateBody, puis on libère la primitive de collision.

Je ne détaillerai pas ici la suite de cette fonction puisqu'il s'agit juste d'ouvrir une image et de créer la texture. Reste la fonction Render que je n'expliquerai pas non plus puisqu'il ne s'agit pas du moteur Newton. Je précise juste que vous je n'ai pas appliqué de callback à notre collision tree, car comme je l'ai dit au début du tutoriel, les collisions tree DOIVENT rester immobile (c'est une des limitations actuelles du collision tree).

IV - Faisons marcher le tout !

Il ne nous reste plus que la fonction main, qui n'a finalement que peu changer par rapport à la dernière fois, si ce n'est les appels pour créer les sphères et les boîtes qui sont un peu différentes compte tenu du petit changement d'organisation des classes.

Quelques petites choses sont toutefois à éclaircir, comme ces quelques lignes juste après la création du monde Newton :

main.cpp

```
float min[] = {-400, -400, -400};
float max[] = {400, 400, 400};
NewtonSetWorldSize (nWorld, min, max);
NewtonSetMinimumFrameRate (nWorld, 60.0f);
```

En effet, de base, la taille du monde Newton est limitée à une boîte dont le minimum est au point -100;-100;-100 et le maximum au point 100;100;100. Tous les objets en dehors de cette limite sont considérés par Newton comme inactif, et les callback n'agissent donc pas sur eux. Il nous faut donc augmenter cette taille, puisque le heightmap est plutôt assez grand. Attention toutefois à ne pas donner des dimensions trop extravagantes sous peine de faire ramer fortement le moteur.

Dernière chose à vous expliquer, les trois fonctions pour le moins barbares au début du fichier :

main.cpp

```
/// FONCTIONS DEBUG
CVector debugFace [1024][2];

void DebugShowGeometryCollision (const NewtonBody* body, int vertexCount, const dFloat* faceVertec,
int id);
void DebugShowBodyCollision (const NewtonBody* body);

int debugCount = 0;

// show all collision geometry is debug mode
void DebugShowCollision ()
{
    int i;

    //glDisable (GL_LIGHTING);
    glDisable(GL_TEXTURE_2D);

    glBegin(GL_LINES);
    glColor3f(1.0f, 1.0f, 0.0f);
    NewtonWorldForEachBodyDo (nWorld, DebugShowBodyCollision);

    glColor3f(0.0f, 0.0f, 1.0f);
    for (i = 0; i < debugCount; i++) {
        glVertex3f (debugFace[i][0].x, debugFace[i][0].y, debugFace[i][0].z);
        glVertex3f (debugFace[i][1].x, debugFace[i][1].y, debugFace[i][1].z);
    }
    glEnd();
    glEnable (GL_TEXTURE_2D);
}

// show collision geometry in debug mode
void DebugShowGeometryCollision (const NewtonBody* body, int vertexCount, const dFloat* faceVertec,
int id)
{
    int i;

    i = vertexCount - 1;
    CVector p0 (faceVertec[i * 3 + 0], faceVertec[i * 3 + 1], faceVertec[i * 3 + 2]);
    for (i = 0; i < vertexCount; i++) {
```

main.cpp

```
        CVector p1 (faceVertec[i * 3 + 0], faceVertec[i * 3 + 1], faceVertec[i * 3 + 2]);
        glVertex3f (p0.x, p0.y, p0.z);
        glVertex3f (p1.x, p1.y, p1.z);
        p0 = p1;
    }

// show rigid body collision geometry
void DebugShowBodyCollision (const NewtonBody* body)
{
    NewtonBodyForEachPolygonDo (body, DebugShowGeometryCollision);
}
/// FIN DES FONCTIONS DEBUG
```

Ces fonctions, dites Debug, nous permettent de dessiner ce que Newton "voit", c'est-à-dire les lignes de collision. Elles peuvent s'avérer extrêmement utiles, notamment pour vérifier que les primitives de collision que l'on a créées soient bien en accord avec ce que l'on a souhaité faire.

Pour pouvoir les utiliser, un simple appel à la fonction DebugShowCollision () avant le dessin des objets suffit. Attention toutefois, ces fonctions ne sont là qu'à un but de debug car elles ralentissent extrêmement le framerate (vous pourrez en juger par vous-même).

J'ai ajouté dans la boucle des événements de la SDL un switch qui permet de détecter dès que la touche F1 est enfoncé, afin d'activer et de désactiver à volonté l'affichage de ces lignes de debug.

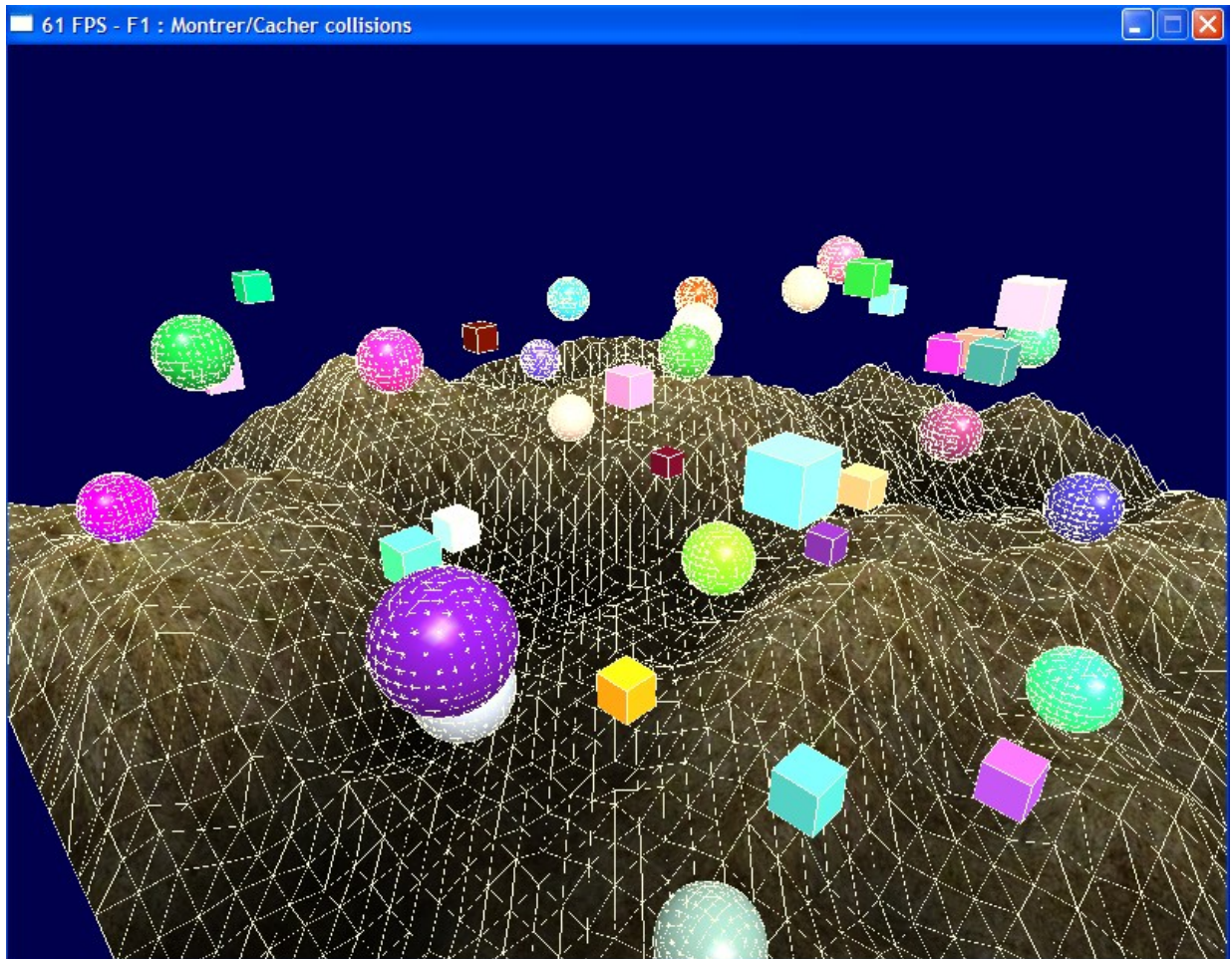
V - Conclusion

C'est tout pour ce tutoriel consacré aux collision tree. J'espère avoir été assez clair, et si vous rencontrez un quelconque problème, vous pouvez m'envoyer un message privé, je tâcherai d'y répondre. Pour la prochaine fois, je compte vous montrer comment utiliser les convex hulls qui, comme les collision tree, permettent de créer des primitives de collisions complexes (moins précises que les collision tree toutefois), mais qui ont l'avantage de pouvoir être soumis à diverses forces et donc bouger de manière réaliste. En attendant, voici le résultat de ce tutoriel :



Sans mode debug

Sans mode debug



Avec mode debug

Avec mode debug

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

VI - Remerciements