Document de conception moteur physique : Ajout d'un système élémentaire de résolution de collisions

Groupe C

Réalisé par :
Antoine de Saint Martin DESA09050103
Victor Guiraud GUIV09029901
Axel Guza GUZA16069906

Sommaire :

1. La conception

Le but de cette phase est d'ajouter au projet un système de collision entre rigidbodies et pour cela nous passons par différentes parties d'implémentation.

Dans un premier temps nous allons implémenter un système de détection de collision (broad phase) via notre structure Octree qui fonctionne efficacement dans un cadre tridimensionnel, son voisin le Quadtree se limite à un usage bidimensionnel. Cet Octree va nous permettre de partitionner l'espace en sous région et répartir l'ensemble de nos rigidbodies dans ces sous régions. Après la répartition effectué, un test sera effectué entre les éléments de la même sous région, ainsi au lieu d'avoir une boucle de détection en $O(n^2)$ avec "n" égal au nombre total d'élément, nous réduisons "n" en le limitant à certains éléments.

Après avoir détecté s'il pouvait y avoir collision entre deux éléments d'une sous région, nous opérons sur ces deux éléments (deux rigidbodies) en comparant leurs caractéristiques spatiales. En fonction de leurs types nous effectuons telle ou telle opération, si nous avons collision (car interpénétration par exemple), nous initialisons une donnée de type "contactData", cette donnée sera exploitée pour résoudre notre collision.

Pour supporter et mettre en image ces différents éléments, nous avons implémenté une démonstration via OpenGL qui permet d'observer plusieurs contacts entre un plan et deux cubes, les données relatives aux contacts sont disponibles via l'affichage console.

Octree :

Notre structure de données Octree est implémentée via un pseudo code du cours "Structures de données avancées et leurs algorithmes 8INF840" et diverses sources trouvées afin d'aider notre compréhension (sources disponibles dans la partie "Sources" du rapport).

```
//Noeud de notre arbre Octree, correspond a
struct OctreeNode
{
    //Centre de la region concernee
    Vector3D m_center;
    //taille de la region concernee
    float m_halfWidth;
    //elements sous cette region, plus bas
    OctreeNode* m_childrens[8];
    //Elements presents dans cette region
    OctreeDetection* m_objectList;
```

Classe OctreeNode : contient les informations d'une sous région de l'arbre. Un positionnement d'un partitionnement de l'espace initial.

```
struct OctreeDetection
{
    //Constructeur
    OctreeDetection(RigidBody* entity);
    //situation dans l'espace tridimensionnel
    Vector3D m_center;
    //taille de l objet
    float m_radius;
    //RigidBody associe a cette donnee tridimer
    RigidBody* m_rb;
    //Objet voisin dans la hierarchie
    OctreeDetection* m_nextObject = nullptr;
};
```

Classe OctreeDetection : contient les informations d'un élément d'une sous région, du type de donnée que nous voulons stocker (un rigidBody).

Cette structure contient diverses méthodes, une méthode de construction d'arbre récursive qui viendra placer les éléments dans les sous régions adéquates par rapport à leurs coordonnées. Nous aurons dans notre arbre un total de 8 sous régions, ces sous régions divisent un cube en 8 sous cubes qui possède chacun leur centre et une taille similaire. Lors de la construction nous assignons à ces cubes des enfants. Après construction nous testons les collisions avec notre méthode "TestAllCollision()", cette méthode teste une collision si plusieurs enfants sont dans la sous région. Si elle détecte deux rigidbodies dans une sous région, elle appelle une seconde méthode, "TestCollision()" qui elle prend en paramètre les rigidbodies et appelle la méthode adéquate entre "CollisionBoiteBoite", "CollisionBoitePlan".... en fonction de la forme des objets (récupérées par l'attribut "shape" de leurs primitives).

Contact RigidBody :

La classe ContactRigidBody permet de de récupérer toutes les informations de collision entre 2 rigidbodies.

```
class ContactRigidBody
{

public:

    //restitution du contact
    float m_restitution = 1;

    //interpenetration entre nos deux rigidbodies
    float m_penetration = 0;

    //normal au point de contact
    Vector3D m_contactNormal;

    //point de contact
    Vector3D m_contactPoint;

    //nous gardons une reference vers nos deux rigidbodies
    RigidBody* m_rigidbodies[2] = { new  RigidBody() };
```

Voici les informations contenues, m_restitution qui jouera sur les forces résultantes du contact, m_penetration qui gérera l'aspect visuel de la collision pour décaler les objets en conséquence et ainsi de suite. Cette structure contient l'ensemble des éléments obligatoires de gestion de contact et de résolution.

C'est dans cette classe que nous allons résoudre les contacts et y appliquer les modifications sur chacun des rigidbodies. Nous avons donc une méthode "resolve()" qui sera appelée à chaque tour de boucle dès lors qu'un contact est détecté. Cette méthode viendra gérer l'interpénétration et gérer la velocité résultante du contact, c'est à dire appliquer à nos éléments les forces d'après coup (selon leurs vitesses initiales, la normale du contact…).

Boite, Plan et primitive :

Boite, plan et primitive sont des classes pour définir les formes qui vont être contenues dans notre projet. Une forme sera assignée à un rigidbody lors de sa création, cette forme impactera le comportement de l'objet lors de la détection/résolution du collision. Nous avons donc une classe mère primitive qui contient les éléments communs tel que l'attribut forme ou encore l'offset… Nos classes filles quant à elles font la différenciation entre ces éléments.

Un plan aura forcément une normal qui caractérise son orientation, un cube aura une taille et des sommets.

Dans notre rendu, le plan utilise également le rendu d'un cube car nous n'avons pas voulu perdre trop de temps sur le rendu graphique, le grand cube de la démo correspond à un plan de normale (0,1,0) et de position (0,0,0). Nous n'avons pas implémenté la forme sphérique car elle n'était pas demandée dans le rendu final, nous avions commencé son implémentation puis nous l'avons laissée de côté.

Collision Data :

```
struct CollisionData
{
    //On recupere l'ensemble des contacts generes
    std::vector<ContactRigidBody*> contact = std::vector<ContactRigidBody*>();

    //Combien de contact il nous reste a resoudre
    int contactLeft = 0;

    void addContact(int count)
    {
        contactLeft -= count;
    }
};
```

Collision data est la structure répertoriant et gardant l'ensemble des contacts qui ont eu lieu durant le tour de boucle courant. Notre vecteur "contact" contient l'ensemble des caractéristiques d'un contact et nous partons donc de cette structure afin d'itérer sur son

vecteur “contact” et les résoudre. ContactLeft correspond au nombre de contact total à résoudre.

Nous créons cette structure au début d’un tour de boucle, elle passe par de nombreuses méthodes comme “TestAllCollision()” de l’Octree puis par “TestCollision()” pour arriver finalement dans une des méthodes de type “CollisionBoitePlan()” ou nous ajouterons notre contact à son vecteur.

```
//Pour les contacts detectes, nous les resolvons.
if (data->contactLeft < 50)//contact left correspond
{
    for (int i = 0; i < 50 - data->contactLeft; i++)
    {
        data->contact[i]->resolve(newDuration);//on re
    }
}
```

Cette boucle est située à la fin de notre UpdateVariousFrameRate() de notre script de démonstration, c’est donc ici, en fin de tour de boucle que nous itérons sur le vecteur contact de notre structure “ContactData()” afin de résoudre l’ensemble.

Contact Resolution :

Dans le contact résolution nous pouvons y retrouver tout ce qui est en lien avec les collisions entre un type de rigidbody et un autre. Nous n’avons implémenté que la collision boite->plan, c’est pour cela que seule la méthode correspondante est présente dans le script.

Pour détecter une collision entre une boite et un plan, nous projetons chacun des sommets de cette boite (de ce cube) sur le vecteur normal du plan, si cette projection est inférieur ou égale à 0, alors nous avons contact. La projection se fait via un produit vectoriel.

Vu que nous itérons sur la totalité des sommets de notre cube (8), nous pouvons récupérer le nombre de contacts, 4 si c’est une face, 2 si c’est une arête, 1 si c’est un coin. Ensuite, nous remplissons les données du contact, de la classe “ContactRigidBody.h”.

Conclusion de conception :

-Résultat final :

Nous avions commencé à créer les primitives et résolutions de contacts pour des objets de type sphère, donc “SphereBoite”, “SpherePlan” ainsi que “BoiteBoite”... Or lors du dernier cours vous nous avez notifié que seul les informations et le visuel d’un contact “BoitePlan” était nécessaire, nous nous sommes donc concentré sur ce rendu.

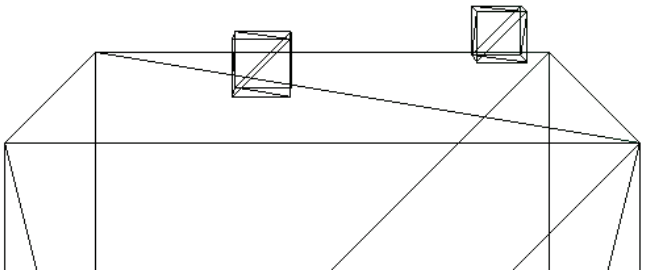
-Problèmes et incohérences :

Normalement, pour la broadphase et l'utilisation de l'Octree, nous aurions dû utiliser des volumes englobants, nous étions parti sur des objets englobants de type sphériques car facile à décrire et il est facile de détecter s'ils sont dans une sous région lors de la création de l'Octree. Au final nous nous sommes résolus à n'utiliser que les primitives des objets et donc directement les formes cubiques car nous étions sur des formes très simples. Le volume englobant est utiles lorsque nous sommes face à des formes complexes de type polygonales car il évite des tests de collisions qui prennent de nombreuses ressources.

Même si nous sommes dans la possibilités de détecter le nombre de sommets d'un cube qui entrent en collision avec notre plan (4 face, 2 arête,1 coin…), nous sommes resté sur une résolution de contact face-face, ainsi même dans l'exemple avec rotation du cube, le contact résultant réagira avec un contact face-face.

2. Démonstration

Afin de démontrer les propriétés de collision que nous avons mis en place, nous avons créer des cubes et un plan (représenté par un cube, la face supérieure est notre plan) dans la même fenêtre qui seront affectées par les forces et rotations que nous avons mis en place durant les phases de développement précédentes.



Collision cube - Projet de session

Dès la compilation effectuée, le lancement de notre démonstration est possible sans besoin d'apporter des changements. Vous pouvez observer deux cubes se déplaçant sur écran blanc rentrant en collision avec un troisème cube fixe (notre plan), ces cubes sont affectés par une force simulant la gravité. Un point de contact est affiché dans le terminal, lors d'un contact cube-plan nous avons 4 affichages de contact correspondant à face contre face. Le cube 0 est le plus proche de l'utilisateur.

```
        Contact detecte entre :
RigidBody : RigidBodyCube : 0
et RigidBody : RigidBodyPlan : 0
La vitesse les separant est : -30.955
Le point de contact est : -24.1755 -0.015258 1.5
La normale au point de contact est : 0 1 0
Le contact a ete resolu apres une penetration de : 0.015258
Le coefficient de restitution du contact est de  : 1

        Contact detecte entre :
RigidBody : RigidBodyCube : 0
et RigidBody : RigidBodyPlan : 0
La vitesse les separant est : -30.955
Le point de contact est : -24.1755 -0.015258 2.5
La normale au point de contact est : 0 1 0
Le contact a ete resolu apres une penetration de : 0.015258
Le coefficient de restitution du contact est de  : 1
```

Rendu de collision - Projet de session

```
int main(int argc, char** argv)
{

    InitOpenGL();
    demoRigidbodiesCollision = new DemoRigidBodiesCollision(window, camera, false);/
```

Si vous insérez “true” en tant que 3ème paramètre dans le fichier “mainRendu4.cpp”, nos cubes seront impactés par une force de rotation à un point local, visuellement cela permet de mettre en avant simultanément le travail du rendu 3 et celui du rendu 4. Cependant le comportement ne sera pas impacté car nous utilisons dans tous les cas un contact face-face et donc la rotation n’est pas impactée.

3. Les difficultés rencontrées

Pour cela, il nous a fallu du temps pour manipuler les paramètres de VBO/VAO et buffer opengl et utiliser différentes méthodes d’affichage pour parvenir à un résultat satisfaisant.

De même nous avions compris initialement qu’en plus de devoir effectuer des collisions et résolutions entre des plans et cubes, nous devions créer une sphère et gérer les rigidbodies et collision propre à cette dernière.

Nous avons donc recherché des moyens de dessiner une sphère avec opengl. Le plus dur était qu’une fois nous avons eu cette sphère et de réussir à appliquer toutes nos forces et rotations à cette dernière, lesquelles étaient uniquement appliquées sur des formes cubiques et rectangulaires. Après en avoir discuté avec notre professeur nous avons conclu qu’il n'était pas nécessaire d’implémenter une sphère et nous nous sommes donc focalisés sur la mise en place de cubes et plans.

<u>Antoine :</u>

-L’Octree n’était pas une structure facile à mettre en place malgré des pseudos codes à disposition, il m’a fallu pas mal de tests afin de réussir à obtenir un comportement satisfaisant et cohérent.

-Ce rendu nécessitait plusieurs scripts mis en relation les uns avec les autres, de nombreuses méthodes sont implémentées et les sources d’erreurs sont nombreuses, il a fallu bien comprendre le principe global et que celui-ci soit clair afin de réussir le débogage de manière optimale.

-J’aurai aimé lors de ce rendu implémenter les objets de type “ForceGenerator” et “ContactGenerator” pour les rigidbodies, je n’ai pas eu le temps de les implémenter,

ils n'étaient pas obligatoires mais j'aurai aimé pour accroître ma satisfaction personnelle.

-Je reste cependant satisfait du rendu global, le résultat final est fonctionnel, je suis satisfait de chacun des rendus, séparément comme dans l'ensemble. Ce cours m'a énormément apporté, étant friand de l'utilisation des moteurs de jeux, j'ai aimé approfondir ma compréhension de certains éléments et de la logique que j'utilise de façon quotidienne sans m'en rendre vraiment compte. L'élément le plus intéressant à mes yeux reste le sujet du dernier rendu et l'optimisation du système de détection avec volumes englobants et utilisation de la structure des Octree. Ça me permet de mettre d'associer un cas pratique à la structure vue dans le cours 8IN840 de l'UQAC.

4. Source :

Création d'une sphère opengl :
-http://www.songho.ca/opengl/gl_sphere.html (non utilisé au final)

Aide à l'affichage de multiple objets :
-VBO: http://www.songho.ca/opengl/gl_vbo.html
-Buffer : https://www.khronos.org/opengl/wiki/Buffer_Object

Structure octree implémentation :
-https://www.geeksforgeeks.org/octree-insertion-and-searching/
-https://github.com/PointCloudLibrary/pcl/tree/master/octree
-https://moodle.uqac.ca/course/view.php?id=25031&section=7 (cours 8INF840)
-https://nomis80.org/code/octree.html

-Cours de moteur physique et les PDF associés au rendu de cette phase. (8INF935)

Livre de référence du cours, C.Ericson "Real time collision detection" pour la compréhension de l'ensemble du système :
-https://uqac.on.worldcat.org/oclc/437177069