Un moteur de physique
Article 3 - La détection de collisions

Gabriel Peyré
nikopol0@altern.org
www.orion3d.fr.st

Le 19 septembre 2001

Table des matières

1. En français dans le texte : volume englobants
2. En français dans le texte : arbre de boites englobantes alignées sur les axes

Introduction

Bon, inutile d'insister sur l'importance de la détection de collisions, que ce soit dans un jeu vidéo, mais aussi dans le milieu industriel (principalement en robotique, test de tolérance ...). Par contre, il convient de préciser brièvement comment la détection de collisions s'inscrit dans un moteur de gestion de physique, partie essentielle de tout bon moteur 3D qui se respecte.

Globalement, le *moteur de physique* se divise en deux parties bien distinctes :

- La détection de collisions : on veut savoir quels objets entrent en contact, et, le cas échéant, à quel endroit la collision a eu lieu, et la normale au point de contact.
- La résolution des équations de la physique (que j'appellerais le *solveur de mouvement*).

Deux principales "missions" sont remplies par cette partie importante du moteur de physique :

. La réponse aux collisions : à chaque impact, le moteur de collisions demande au *solveur de mouvement* d'apporter une réponse adéquate au choc.

. Le déplacement des objets : à chaque rafraîchissement d'image, il faut animer les objets, ceux-ci étant soumis à des forces extérieures. Il convient donc de résoudre les équations de la physique pour calculer les variables du mouvement (vitesse, position, rotation, vitesse de rotation).

Chapitre 1

Intégration dans Orion3D

Avant de se lancer dans l'explication détaillée des algorithmes de détection de collision, je tiens à expliquer comment cela est mis en ordre dans **Orion3D**.

Orion3D n'implémente pas directement ces algorithme, puisqu'il utilise **Opcode**[1], une librairie de détection de collision à base d'arbre hiérarchique d'AABB bounding box. Cependant, il est important de créer une interface orientée objet efficace entre le moteur de physique et le reste de l'application (ici un moteur 3D).

1.1 Les colliders

1.2 Le manager de collision

1. http://www.codercorner.com/Opcode.htm

Chapitre 2

Détection grossière

Je vais maintenant décrire en détail comment fonctionne l'algorithme de détection de collisions. Tout d'abord un bref aperçu (on note n le nombre de corps) des trois phases que nous allons détailler par la suite :

- Selection parmi l'ensemble des paires d'objets de celles susceptibles d'entrer en contact.
- Choix des 2 *entités* [1] (= face ou arrête ou vertex)les plus proches dans l'*enveloppe convexe* [2].
- Si collision, et si la face ne fait pas partie de l'objet, détection exacte de la face.

2.1 Problème à N corps

Commençons donc par la première phase, qui permet de sélectionner de façon "*heuristique*" parmi l'ensemble des objets, ceux qui nous intéressent. En effet, l'algorithme présenté précédemment ne permet de s'occuper que de l'interaction de deux corps. Il s'agit donc de ramener ce "problème à N corps" (en référence au fameux problème d'astronomique) à un problème à deux corps, que l'on sait résoudre. En considérant une scène contenant N objets en mouvement, et M objets statique, nous avons donc à prendre en compte $C_N^2+M^2$ couples d' objets (où C_N^2 est le nombre de combinaisons), ce qui peut représenter une masse de calculs énorme. Il convient donc d'établir un heuristique nous permettant de réduire ce nombre.

2.2 Utilisation de bounding box

Ming C. Lin propose dans sa thèse deux *heuristiques*, j'ai choisi de me concentrer plus particulièrement sur l'un d'entre eux, qui semble être celui retenu dans la pratique. Pour parvenir à nos fins, on va effectuer ce qu'il convient d'appeler une "*détection grossière*", et que dans la littérature on appelle plus souvent "*sweep and prune algorithm*" [3].

On utilise des $AABB$ [4], et on regarde sur chaque axe si elles s'intersectent. Il y a 2 options :

- soit les box sont statiques : elles ne sont pas redimentionnées (il faut donc les prévoir assez grandes).
- soit on les re-dimentionne a chaque frame, en classant les coordonnées des vertex sur chaque axe, pour determiner le min / le max.

1. En anglais *features*
2. Voir plus loin pour une définition
3. Intraduisible ...
4. **A**xis **A**ligned **B**ounding **B**ox

Chapitre 3

Marching Voronoy

Nous attaquons maintenant la deuxième phase de l'algorithme, celle qui nous permet de suivre au cours du temps l'évolution de la distance minimale entre deux objets. C'est cette partie de l'algorithme qui a été introduit par **M. C . Lin** dans son PhD ([LIN1]), et qui présente pas mal d'innovations.

3.1 Présentation de l'algorithme

Tout d'abord un bref aperçu des intérêts et caractéristiques de cet algorithme :

- Il permet de prendre en compte l'évolution des objets au cours du temps : il utilise donc avantageusement la *cohérence temporelle* du modèle, et dans le cas où les objets ne bougent pas de façon excessive, on peut estimer que cet algorithme s'exécute en temps *constant* relativement au nombre de faces : quel que soit la complexité des objets, cet algorithme tournera toujours aussi vite !
- Ce qui permet à cet algorithme d'être aussi efficace, c'est que la plus part des calculs sont effectués une fois pour toute au moment du chargement des objets. Ainsi, on peut utiliser des structures de données adéquates pour stocker les informations *géométriques* (position des points), ainsi que *topologiques* (place des faces, arrêtes et points les uns par rapport aux autres). De plus, il calcule des informations supplémentaires sur nos objets, les fameux *diagrammes de Voronoy*, dont nous reparlerons par la suite.
- Enfin, il est important de noter que cet algorithme ne marche que sur les objets convexes. Pour les objets de formes quelconques, deux alternatives sont possibles :
  . Découpage de l'objet en un arbre d'objets convexes : c'est ce que préconise **Ming C. Lin** dans son rapport ([LIN1]), et ceci implique de faire des tests de collision récursifs entre les arbres. Ceci peut être trés lourd dans le cas d'objets très irréguliers ("*polygons soup*" [1]), d'où l'idée d'implementer un algorithme plus général, dans une troisième passe.
  . Faire une phase de détection exacte suite à la détection de collision sur les *enveloppes convexes* des objets : c'est cette technique que nous allons présenter dans la troisième phase de présentation de l'algorithme.

3.2 Enveloppe convexe

Tout d'abord, précisions que l'on travaille maintenant sur l'*enveloppe convexe* de nos objets, qui est, par définition, le plus petit convexe contenant l'objet. C'est en fait l'ensemble des *bary-*

1. Littéralement "soupe de polygones"

centres de nos vertex, affectés de coefficients positifs. Par exemple, l'enveloppe convexe de 3 points est un triangle ... On peut montrer que l'on peut se ramener à des barycentres de n+1 points, où n est la dimension de l'espace (théorème de **Caratheodory**, utile pour montrer que l'enveloppe convexe d'un compact est compacte ... oups, je m'égare). En tout cas, il existe quantité d'algorithmes dans le domaine public pour déterminer cette enveloppe en $O(n * log(n))$. Par exemple, quantité de librairies de collision utilisent la librairie **QHull**, qui semble très efficace.

3.3 Diagramme de Voronoy

Définissons maintenant ce qu'est un *diagramme de Voronoy*. Un *diagramme de Voronoy* est constitué de l'ensemble des régions de Voronoy de l'objet. Une région de Voronoy est un ensemble de points plus proche d'une entité que de aucune autre. Ainsi, pour un objet convexe (c'est ici que la convexité joue un role !), le *diagramme de Voronoy* forme une partition de l'espace externe de l'objet, et nous permet donc de dire instantanément quel est l'entité (face, point ou arrête) qui est la plus proche d'une autre entité donnée.

3.4 Suivie de la distance minimale

Tout d'abord, intéressons nous à la suivie de la distance minimale. Cet partie de l'algorithme est souvent appelée "*Voronoy marching*", car il s'agit de parcourir les diagrammes de Voronoy de nos objets. Entrons dans le vif du sujet...

Grace à l'algorithme présenté dans le chapitre précédent, on sélectionne les paires d'objets qui nous intéressent. On va utiliser la coherence temporelle pour le suivi de la distance minimale : on va considérer que les mouvements / rotations des objets ne sont pas trop grands. Le principe consiste a détecter une entité dans chaque objet, pour réaliser le minimum de la distance. Pour cela, on part des derniers résultats obtenu (cohérence temporelle ...), et on s'aide des régions de Voronoy pour sélectionner les nouvelles entités.

Pour rendre l'ensemble des structures de données plus cohérent, **M. C. Lin** préconise de subdiviser les faces avec trop de cotés (typiquement plus que 6) en faces plus petites, pour que les régions de Voronoy soient toujours délimitées par un petit nombre de plans (pour les régions associées à des faces, bien sûr !).

Grosso-modo, l'algorithme consiste a parcourir de proche en proche les regions de Voronoy jusqu'à arriver à une region qui contienne le point de l'autre objets sélectionné. En se débrouillant bien, on arrive a diminuer la distance (en tout cas si l'objet est convexe, ce qui est le cas, vu qu'on travaille sur les enveloppe convexes !), et donc l'algorithme converge plutôt vite (en tout cas, la coherence temporelle joue pleinement son role !). Pour décrire plus précisément l'algorithme, il faut noter que plusieurs cas sont à distinguer, selon que l'on envisage de tester la proximité de deux faces, d'une face et un point, de deux points, etc. Par exemple, dans le cas de deux arrêtes, il faudra tout d'abord déterminer quels sont les points les plus proche, pour se ramener au cas du test de proximité de deux points.

On part donc avec les deux *entités* qui sont a priori les plus proche (c.a.d. les deux *entités* qui ont marché pour le coup précédent), si elles sont dans les bonnes *régions de Voronoy*, on arrête, sinon, on passe à une région voisine qui est plus proche que la région testée. Dans son PhD, **Ming C. Lin** ([LIN1]) prouve que dans des cas de mouvements restreints (pas trop rapides), cet algorithme tourne bien en temps constant par rapport au nombre d'entités. La figure 3.1 montre comment trouver le vertex de l'objet O le plus proche du vertex V de l'objet O'.

3.5 Interpénétration

Enfin, la grosse astuce, pour déterminer si il y a collision, est de ne pas se limiter aux régions externes de l'objet, mais aussi de créer des *"pseudo" régions de Voronoy* à l'intérieur de l'objet, et de

FIG. 3.1 – *E_1, E_2, E_3, E_4 sont les régions de Voronoy des 4 vertex de O (V_1, V_2, V_3 et V_4). E_a, E_b, E_c et E_d sont les régions de Voronoy des quatres arrêtes de O (a, b, c, d). On cherche l'entité de O la plus proche de V, un vertex de O'. C'est V_1 car V se trouve dans E_1, la région de Voronoy de V_1.*

FIG. 3.2 – *E_a, E_b, E_c et E_d sont les pseudo-régions de Voronoy des quatres arrêtes de O (a, b, c, d)*

poursuivre la recherche de notre entité à l'intérieur de l'objet, ce qui est synonyme de collision. On ne calcule pas exactement les régions de Voronoy internes, tout simplement car on considère que l'on arrête de rechercher la distance minimale dés lors que les objets s'interpénètrent. Pour finir, précision comment calculer ces "*pseudo-régions*" : il suffit pour cela de les délimiter par des plans joignant le centre de l'objet a chacune des arrêtes, ce qui constitue bien une bonne approximation, dés lors que les objets utilisés sont convexes. La figure 3.2 montre sur un exemple 2D (un quadrilatère) le construction de ces régions de Voronoy.

Chapitre 4

Détection éxacte

Enfin, voici la dernière partie de l'algorithme, qui permet de faire une détection *exacte* des faces où ont eu lieu la collision.

4.1 Bounding volume[1]

La précédente phase nous permet de détecter dans l'*enveloppe convexe* des objets, quelles faces sont entrées en collision. Deux cas peuvent se presenter :

- Les deux faces appartiennent effectivement aux 2 objets : on est content.
- Une des faces n'appartient pas a l'objet : ouille, si on veut faire de la détection exacte, il nous reste du boulot !

4.2 OOB tree[2]

C'est là qu'entre en jeu la 3e phase de la détection, la plus complexe. Il s'agit, pour déterminer efficacement la face, d'utiliser une structure de données spéciale, en l'occurrence un arbre hiérarchique de **OBB**[3], qui sont simplement des boites avec 3 axes les orientant.

La chose la plus complexe est la création de cet arbre. Pour ce faire, on subdivise récursivement la boite initiale en deux parties. La difficulté réside dans un choix efficace des 3 axes, pour qu'ils s'alignent le mieux possible sur la topologie de l'objet. Pour se faire, on étudie la *matrice de covariance* des positions des vertex. En effet, cette matrice est symétrique, elle diagonalise en base orthonormée, ce qui signifie que ses 3 vecteurs propres sont orthogonaux, et ce sont ces vecteurs qui vont constituer les 3 axes de notre boite.

4.3 Rappels sur la matrice de covariance

Juste pour rappel : une fois calculée l'espérance des positions des vertex, notons la (m_x,m_y,m_z) (m comme moyenne ...), on calcule la matrice de covariance par les formules :

$$C = \begin{pmatrix} C_{xx} & C_{xy} & C_{xz} \\ C_{xy} & C_{yy} & C_{yz} \\ C_{xz} & C_{yz} & C_{zz} \end{pmatrix} \tag{4.1}$$

avec :

$$\boxed{\begin{array}{c} C_{xy} = \sum_{i=0}^{N} (x_i - m_x).(y_i - m_y) \\ C_{yz} = \sum_{i=0}^{N} (z_i - m_z).(z_i - m_z) \\ C_{xz} = \sum_{i=0}^{N} (x_i - m_x).(z_i - m_z) \\ C_{xx} = \sum_{i=0}^{N} (x_i - m_x)^2 \\ C_{yy} = \sum_{i=0}^{N} (y_i - m_y)^2 \\ C_{zz} = \sum_{i=0}^{N} (z_i - m_z)^2 \end{array}}$$

où (x_i,y_i,z_i) représente la position du vertex i :

1. En français dans le texte : volume englobants
2. En français dans le texte : arbre de boites englobantes alignées sur les axes
3. **O**riented **B**ounding **B**ox

Pour subdiviser la boite en deux boites égales, on choisit le coté le plus grand, et on le divise selon la valeur de l'espérance sur cet axe.

TODO : décrire le parcours de l'arbre.

4.4 Quelques remarques

Enfin, notons qu'il existe des améliorations à cet algorithme, notamment pour éviter que de trop grandes concentrations de points viennent fausser l'algorithme (on passe par l'enveloppe convexe).

Tous ces calculs étant effectués une fois pour toute au chargement de l'objet, leur complexité (ouch, ça fait beaucoup de calculs, tout ça !) n'a que peu d'importance. Par contre, tout ceci risque de poser des problèmes en ce qui concerne les mesh dynamiques, tels les skins des personnages ...

Chapitre 5

Implémentation dans un jeu

Cette partie se veut un peu à part dans cet article, puisqu'elle fait en quelque sorte la jonction avec un autre article, celui qui s'intéresse à la construction d'un moteur de physique. Ces deux composantes essentielles d'un moteur 3D sont intimement liées, puisque c'est au moteur de physique que revient de le rôle de réagir aux solicitations du moteur de détection de collisions. L'implémentation de la résolution des équation sous contraintes sont en parties tirés de [JAK].

5.1 Objets libres et sous contraintes

En réalité, il s'agit ici d'expliquer comment impléménter de façon pratique la communication entre ces deux entités. Pour ce faire, je propose de distinguer deux types d'objets, chacun étant traité d'une manière différente :

- Les objets indépendants : ils sont traités comme des solides indéformables, et à chaque collision, on applique les règles classiques de la détection de collision pour déterminer la réaction adéquat. Se reporter à la partie "Mécanique classique" de l'article sur les moteurs de physique pour plus de détails.
- Les objets soumis à des contraintes internes ou externes. Les contraintes sont modélisées par des liaisons ("bones" en anglais), les contraintes pouvant être relatives à la taille du bones, ou à un angle entre deux bones. Se reporter à la partie "Mécanique sous contrainte" de l'article sur les moteurs de physique pour plus de détails.

5.2 Implémentation pour les deux types d'objets

Sans empiéter sur l'article suivant, voici les remarques quand aux techniques employées dans les deux cas.

5.2.1 Mécanique classique

Dans le cas de la mécanique classique, il s'agit de faire des calculs exacts sur le mouvement des solides. Attention, ces calculs peuvent être très complexes, allergiques à la physiques du solide s'abstenir !

5.2.2 Mécanique sous contraintes

Dans le cas de la mécanique sous contraintes, il s'agit de faire bon nombre d'approximations, pour permettre de résoudre de façon pragmatique (et rapide !) les différentes équations issues des contraintes et de la physique du solide. En bref, voici ces approximations :

- On ne considère plus une masse répartie de façon continue sur le solide, mais plutôt

concentrée aux extrémités des *bones* (analogie avec des *particules pesantes*).
- On fait une detection de collision simplifiée au niveau des bones (par ex. avec des *bounding cylindres*), entre autre pour ne pas se reposer sur le mesh polygonal en lui même (possibilités de déformations).
- On "fait semblant" de résoudre les systèmes de contraintes en employant une méthode dite de "*relaxations successives*".
- On utilise un schéma de résolution des équations du mouvement le plus stable possible. Typiquement, le schéma dit "*Verlet*" (du nom de son inventeur), communément employé en modélisation moléculaire, est bien adapté. En effet, sa grande stabilité le rend peu sensible aux erreurs commises par la méthode de relaxation.

Conclusion

Pour finir, il convient de noter que l'algorithme ne prend en compte que les collisions entre les objets en mouvements. Il faudra bien évidemment ajouter les tests de collisions avec l'environnement, ie. le BSP tree, le moteur de terrain, etc !

Enfin, un mots sur les objets non polygonaux, qui ont aussi fait l'objets de recherches intensives. Je pense particulièrement aux ensembles algébriques, ie. les ensembles définis par des équation polynomiales homogènes, comme par exemple les *Bezier Patch* ou les *Nurbs*. Le PhD de **Ming C. Lin** parle abondamment des algorithmes utilisables pour effectuer des tests de collisions avec de tels objets, assez techniques, ces remarques sont néanmoins très intéressantes.

J'attend vos commentaires avec impatience, sachant que pour l'instant, j'en suis encore au stade expérimental !

Annexe A

Références

Voici un ensemble de resource on-line, avec une brève description.

- Le groupe de recherche de **University of North Carolina**
  http://www.cs.unc.edu/%7Egeom/collide/index.shtml
- La page personnelle de **Ming C. Lin**
  http://www.cs.unc.edu/~lin/
- La librairie **Opcode**
  http://www.codercorner.com/Opcode.htm

Bibliographie

[LIN1] Ming C. Lin, Efficient Collision Detection for Animation and Robotics, http://www.cs.unc.edu/~lin/papers.html, 1997

[LIN2] M. Lin and S. Gottschalk, Collision Detection between Geometric Models : A Survey, ,1998

[JAK] Thomas Jakobsen, Advanced Character Physics, http://www.ioi.dk/Homepages/tj/publications/gdc2001.htm,1998

[ICOL] - J. Cohen, M. Lin, D. Manocha and K. Ponamgi, I-COLLIDE : An Interactive and Exact Collision Detection System for Large-Scaled Environments , ,1998