Maîtrise Informatique : TER-STAGE 2004

Réalisation d'un Moteur Physique

par Eric Mounhem – eric@skbo.net
et Bruno Paillier – bowno@noos.fr

Site internet du projet :
http://www.skbo.net/virchor/

Responsable de stage : Christian Jacquemin

Présentation

Ce travail a été effectué dans le cadre d'un TER-Stage (Travail d'Étude et de Recherche) au LIMSI (Laboratoire d'Informatique pour la Mécanique et les Sciences de l'Ingénieur), projet d'une durée de six mois imposant une demi-journée hebdomadaire en laboratoire encadré par un chercheur, au cours de notre Maîtrise d'informatique à l'Université Paris-Sud Orsay.
Le sujet proposé était de réaliser un moteur physique pour le logiciel de description de scènes Virtual Choreographer *(`http://virchor.sourceforge.net/index.html`) créé par Christian Jacquemin. Ce logiciel permet de construire des scènes qui pourront ensuite être rendues en temps réel avec* OpenGL *(`http://www.opengl.org`) ou par des programmes externes tels que* PovRay *(`http://www.povray.org`) et* Renderman *(`http://www.renderman.org`).*
Le domaine de la physique étant extrêmement large, nous avons décidé de travailler essentiellement au niveau de la gestion des collisions, élément important d'un moteur physique. Le rendu temps réel de Virtual Choreographer *impose des coûts réduits en temps et par conséquent un algorithme adapté.*
Pour la partie physique pure, nous nous sommes limités à des collisions sans déformations, ni frottements et à la gestion de la gravité.

Table des matières

Chapitre 1

Introduction

1.1 Le problème des collisions

Le but de la gestion de collision est de corriger le comportement des éléments d'une scène afin d'empêcher des comportements impossibles tels que le chevauchement de deux objets, ou un objet qui en traverse un autre, etc. Intuitivement on comprend bien qu'il faut tester chaque objet avec les autres éléments de la scène afin de voir s'il entre en collision avec un autre auquel cas on lui applique une fonction « réaction physique »qui corrigera les trajectoires.

Cependant l'idée naïve de tester chaque objet avec l'ensemble de la scène est irréaliste dans un rendu temps réel d'une scène pouvant contenir des centaines d'objets sachant qu'il est nécessaire, pour améliorer le réalisme, de connaître les triangles des maillages en contact. De tels algorithmes ne conviennent donc pas à un traitement brutal sur l'ensemble de la scène.

De nombreuses recherches ont donc été effectuées afin d'optimiser au mieux ce processus coûteux.

1.2 Principe de traitement

La méthode retenue est un traitement par phases de plus en plus précises.

1. *Détection des collisions* :
   On cherche les paires d'objets qui peuvent être en collision parmi toute la combinatoire possible pour la scène. On se contente de faire des approximations peu précises des objets, l'important étant que le traitement élimine rapidement les éléments qui ne peuvent être en collision parce qu'ils sont trop éloignés les uns des autres.

2. *Détermination de la collision* :
   ou encore *Localisation du contact*,

   Cet algorithme prend en entrée deux objets et indique s'ils sont en situation de collision. Si c'est le cas, il peut renvoyer des informations supplémentaires telles que le point de collision, la normale au point, le ou les triangle(s) de contact, la distance de pénétration, etc.

   Cet algorithme coûteux n'est appelé que sur les objets qui n'ont pas été éliminés à l'étape précédente limitant les calculs inutiles.

3. *Résolution de la collision* :
   On agit sur les objets en réaction à la collision afin de corriger la scène en supprimant les comportements impossibles physiquement. Cette phase n'est pas vraiment complexe du point de vue algorithmique, mais elle l'est du point de vue physique suivant le degré de réalisme que l'on veut atteindre.

Il existe également des détecteurs de collision qui séparent la phase de détermination de la collision en deux avec une première passe cherchant si des triangles sont susceptibles de se chevaucher et une seconde pour les trouver. Toutes ces étapes intermédiaires permettent de gagner du temps dans le cas où les phases suivantes sont inutiles et évitent d'appliquer des algorithmes coûteux pour rien.

1.3 Structure du rapport

Dans le deuxième chapitre, nous verrons quelques méthodes de détection de collisions. Ensuite, nous décrirons les algorithmes de détermination de collisions sur lesquels nous nous sommes documentés. Dans le quatrième chapitre nous expliciterons la physique utilisée avant de conclure sur l'intégration du moteur au sein de *Virtual Choreographer*.

Chapitre 2

Détection de collisions

Comme nous l'avons dit, le but de cette partie est d'éliminer rapidement les objets qui n'ont aucune chance d'être en contact. Il ne s'agit donc pas d'être trop précis et on peut se permettre d'approximer les objets par des volumes englobants . Le choix de ce volume englobant n'en reste pas moins une question délicate dépendante du cadre de l'application. Par exemple, dans le cadre d'un jeu, il est plus intéressant d'approximer un personnage par un cylindre que par une sphère et de se contenter de faire des tests sur le cylindre.

Ici nous allons voir deux volumes englobants classiques que sont la sphère et la boîte englobante. Ensuite nous décrirons rapidement une autre méthode de test de collision par l'utilisation d'arbres BSP.

2.1 Deux premiers volumes englobants

2.1.1 La sphère

La sphère est un volume qui se prête bien à ce rôle de volume englobant (Fig. 2.2). En effet le test de collision entre deux sphères ne coûte rien puisqu'il suffit de voir si la distance entre les centres est inférieure à la somme des rayons. De plus la sphère n'a pas besoin d'être orientée, juste positionnée.

Par contre, le calcul de la sphère englobante n'est pas aussi simple qu'il n'y paraît, car, s'il est facile de trouver une sphère, obtenir la sphère optimale est plus complexe. Il est toutefois possible d'accélérer le calcul en appliquant des heuristiques telles que l'algorithme linéaire de Ritter décrit sur le site de Dan Sunday [12] :

- On parcourt la liste des points de l'objet à la recherche de deux points éloignés (mais pas obligatoirement les plus éloignés de l'ensemble). Ces deux points constituent le diamètre de la sphère initiale.
- On parcourt une nouvelle fois l'ensemble de points en testant si les points sont à l'intérieur de la sphère. À l'étape i, si le point p_{i+1} est inclus dans la sphère S_i, alors $S_{i+1} = S_i$. Si p_{i+1} est en dehors de B_i, il faut alors agrandir B_i jusqu'à avoir p_{i+1}. Pour cela, on trace une ligne partant de p_{i+1}, passant par le centre de B_i, que l'on prolonge jusqu'à atteindre l'autre côté de la sphère. Cette ligne sera le nouveau diamètre de la nouvelle sphère B_{i+1} qui contient p_{i+1} ainsi que B_i, et par conséquent tous les points déjà vus, comme on peut le voir sur le dessin (Fig. 2.1).

Cette méthode approximative permet de calculer rapidement une sphère englobante proche de l'optimale et Ritter estime que l'erreur commise est de moins de 5%.

Néanmoins, dans certains cas, l'approximation peut se révéler un peu trop forte et peu adaptée à l'objet (Fig. 2.3). La sphère est alors essentiellement constituée de vide.

2.1.2 Les *Axis Aligned Bounding Boxes*

Mis à part pour les objets sphériques, les sphères ne constituent pas une très bonne approximation risquant de réduire le nombre d'objets éliminés pendant la phase de détection de collisions. Pour un coût pratiquement équivalent, il est donc plus judicieux de remplacer la sphère par une boîte englobante dont

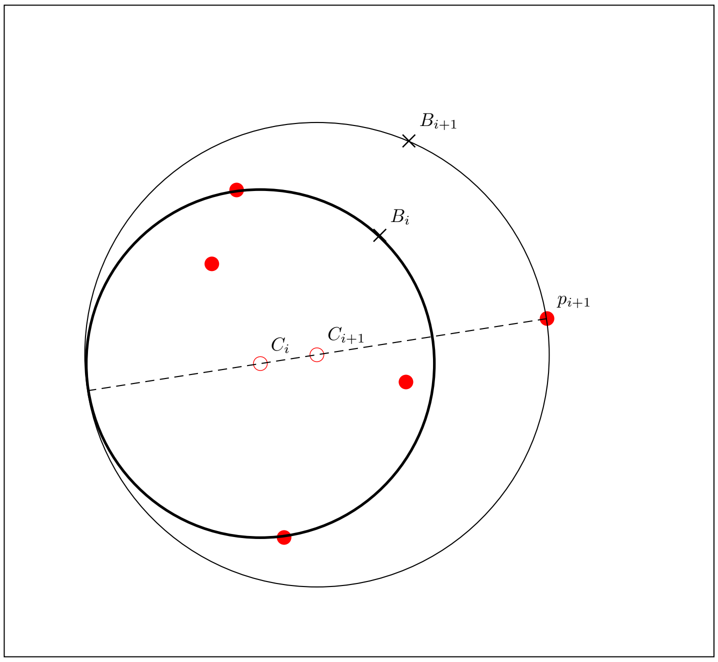

FIG. 2.1 – Construction itérative d'une sphère englobante

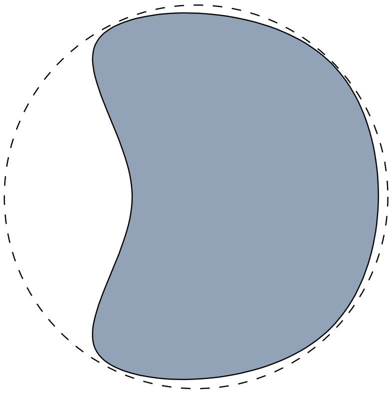

FIG. 2.2 – Sphère englobante bien adaptée à la forme de l'objet

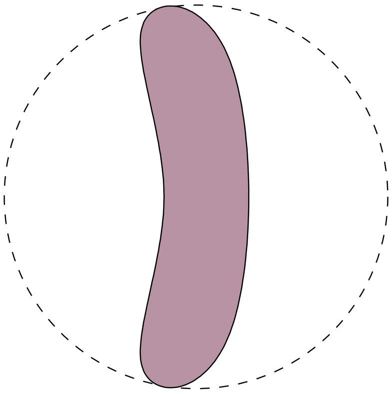

FIG. 2.3 – Sphère englobante trop grande pour l'objet

les côtés sont parallèles aux axes du repère. Ce sont les *AABB (Axis Aligned Bounding Boxes)* (Fig. 2.4 et 2.5).

Pour construire une AABB d'un objet, il suffit de prendre le maximum et minimum des coordonnées suivant les trois axes du repère puis de tracer les plans passant par ces coordonnées. L'intersection de ces plans forme l'AABB (Fig. 2.6).

Cependant certains objets se prêtent mal aux AABB, notamment si l'objet est incliné à 45°(Fig. 2.7 et 2.8).

Après avoir vu ce que sont les AABB, nous allons les utiliser pour la première phase du traitement des collisions.

2.2 Détection de collision avec les AABB

Comme il a été dit précédemment, un des problèmes majeurs de la détection de collisions est que l'algorithme doit être en mesure d'indiquer rapidement les objets qui ne peuvent pas être en situation de

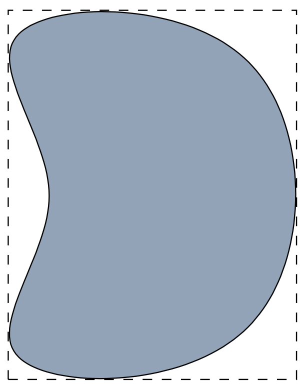

Fig. 2.4 – AABB

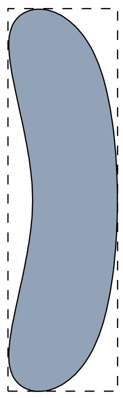

Fig. 2.5 – AABB réduite

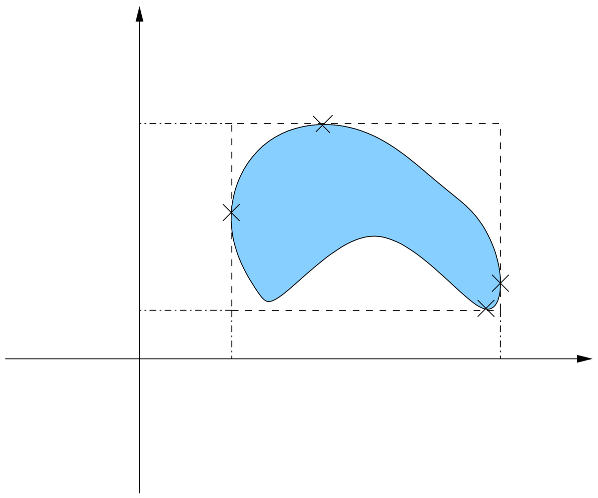

Fig. 2.6 – Construction d'une boîte englobante axée sur les axes (*AABB*)

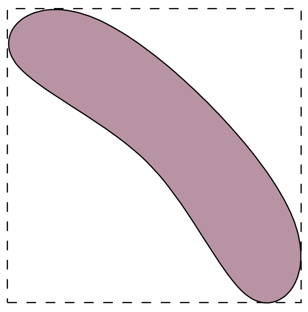

Fig. 2.7 – AABB moins optimale

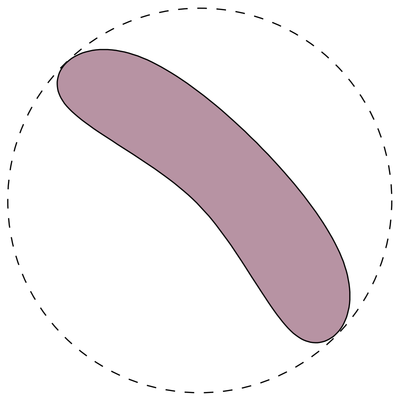

Fig. 2.8 – Sphère englobante identique

collision. Or, nous avons vu que tester tous les objets entre eux est une solution beaucoup trop coûteuse, même en utilisant les AABB. Heureusement, il est possible de simplifier le problème grâce à la *cohérence temporelle*.

2.2.1 La cohéerence temporelle

La cohérence temporelle est une hypothèse simplificatrice qui suppose qu'entre deux frames il ne peut pas y avoir de grands changements au niveau des objets. Plus précisément cela revient à dire que d'une frame à l'autre les objets ne se déplacent pas de grands intervalles (et de même pour les déformations etc.).

2.2.2 Utilisation de la cohérence temporelle

Dans un premier temps, on projette les boîtes englobantes sur les trois axes du repère. Ceci nous donne des intervalles sur chaque axe pour chaque objet. Deux boîtes se chevaucheront si et seulement si les intervalles se chevauchent sur les trois axes.

Pour détecter le chevauchement d'un intervalle sur un axe, on procède comme suit :
- Pour chaque objet i, on appelle s_i le début de l'intervalle, e_i la fin de l'intervalle ;
- On trie les intervalles par s_i croissant ;
- On parcourt la liste obtenue. Pour chaque s_i rencontré, on le place dans un ensemble E. Lorsque l'on rencontre e_i, on retire s_i de E. Si on rencontre un s_i alors que E n'est pas vide, c'est que les intervalles se chevauchent.

On procède de même sur les trois axes (exemple en dimension 2, Fig. 2.9 et 2.10).

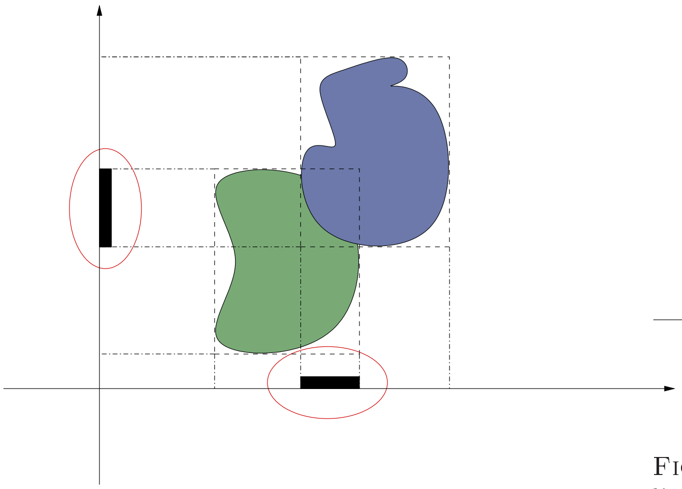

FIG. 2.9 – Chevauchement des boîtes

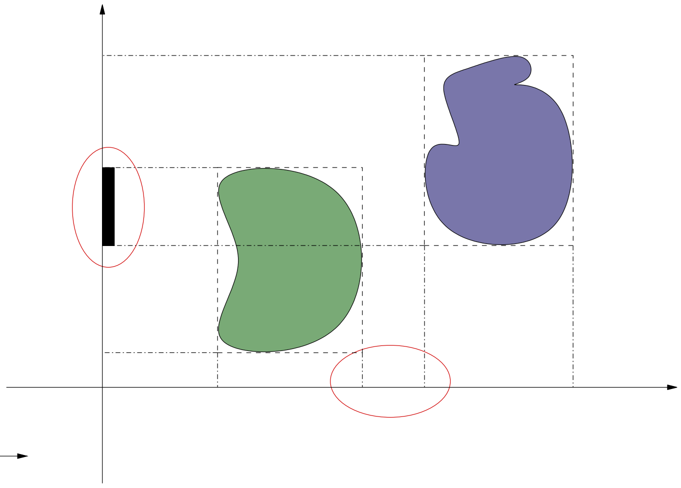

FIG. 2.10 – Les intervalles se chevauchent sur l'axe des ordonnées, mais pas sur l'axe des abscisses, les boites ne sont pas en contact.

L'algorithme complet est le suivant :

1. À la première frame, on construit les trois listes d'intervalles, puis on les trie sur les débuts d'intervalle en appliquant un algorithme complexe tel que le *quicksort* ;
2. On parcourt les listes triées en notant les paires d'intervalles qui se chevauchent ;
3. Lorsque les intervalles se chevauchent sur les trois axes, on ajoute la paire d'objets correspondants à la liste des objets pour la suite du traitement ;
4. À la frame suivante, la cohérence temporelle nous assure qu'il n'y a pas eu de grands déplacements d'objets et par conséquent les listes d'intervalles sont pratiquement encore triées. On peut donc les trier en appliquant un tri simple comme le *tri par insertion* ou le *tri bulle* qui sont efficaces sur les listes triées ($O(n)$). Ce tri se contentera de parcourir les listes en échangeant simplement quelques débuts d'intervalles.
5. On revient à l'étape 2.

Dans le cas de *Virtual Choreographer*, les AABB ne sont pas fixes : elles peuvent non seulement être modifiées par un déplacement de l'objet, mais également par une déformation. Ceci n'est cependant pas une difficulté puisqu'on peut également appliquer la cohérence temporelle pour les déformations.

2.3 Détection de collisions utilisant les arbres BSP

Cet algorithme permet de tester la collision entre un objet et une scène décrite par un arbre BSP.

Un arbre BSP est un arbre binaire représentant une scène. Il est construit en choisissant un axe placé à un nœud de l'arbre et en séparant les polygones de la scène en un ensemble à gauche et un à droite de l'axe, formant les deux fils du nœud. On utilise également deux types de feuilles pour savoir si on est à l'intérieur (feuille « pleine ») ou à l'extérieur d'un objet (feuille « vide »).

La détection de collision est ensuite effectuée en testant si lors du déplacement dans la scène, et par conséquent dans l'arbre, on est passé par une feuille « pleine ». Si ce n'est pas le cas, le déplacement est valide, sinon il y a eu collision.

Cette méthode est efficace et très utilisée dans les jeux, notamment les « doom-like », pour représenter les décors statiques. Dans le cadre de *Virtual Choreographer*, où nous testons les collisions entre un nombre indéterminé d'objets en mouvement, elle devient inefficace puisqu'il faudrait reconstruire l'arbre BSP à chaque frame.

2.4 Conclusion

Étant donné que dans *Virtual Choreographer* l'AABB de chaque objet est fournie, nous nous sommes naturellement tournés vers l'algorithme de détection de collisions basé sur ce type de volume englobant. Les AABB sont un bon compromis en terme de simplicité de calcul et d'utilisation et sont suffisament proches des contours des objets pour cette première passe.

Chapitre 3

Détermination de collisions

La détermination des collisions fait suite à la détection des collisions vue à la section précédente et consiste à déterminer plus précisément l'endroit du contact entre les objets, s'il se trouve qu'il y a effectivement contact. En effet, la détection seule est insuffisante pour modéliser un comportement réaliste des objets. Il est nécessaire, d'une part, de confirmer le contact potentiel relevé à la détection, et d'autre part d'avoir des informations plus précises sur l'endroit du contact, ce qui joue un rôle important dans la réponse physique. Par exemple, lors de rotations des objets à la suite d'un choc, il est intéressant de connaître l'emplacement du contact sur l'objet afin d'avoir un angle de rotation proportionnel à la distance de la projection du point de contact sur la perpendiculaire à la normale passant par le centre de l'objet au centre de gravité de l'objet (voir section 4.1.4).

Malgré sa précision, la localisation doit pouvoir être effectuée dans des temps raisonnables, particulièrement dans le cas d'applications temps réel.

Une méthode classique pour atteindre cet objectif est d'utiliser des hiérarchies de volumes englobants. Aussi nous allons commencer par voir ce que sont ces hiérarchies, comment les construire et les utiliser efficacement. Puis nous présenterons succintement trois algorithmes de détection de collision avec la méthode utilisée.

3.1 Hiérarchie de volumes englobants

Une hiérarchie de volumes englobants est une représentation d'un objet par un arbre, où les nœuds internes sont des volumes englobants qui entourent tous leurs fils à l'intérieur de leur volume et les feuilles un ensemble non vide de primitives (Fig. 3.1). En général ce sont des arbres binaires qui sont utilisés, étant plus simples et rapides à construire, tout en permettant un traitement efficace.

Nous allons nous attarder sur ces arbres en abordant leur construction et leur utilisation.

3.1.1 Construction d'une hiérarchie

On note trois manières de construire une hiérarchie : les méthodes *bottom-up*, *incremental tree-insertion* et *top-down*.

La méthode *bottom-up*

On commence par regrouper des primitives proches par un volume englobant, puis on recommence avec d'autres primitives, ou en combinant les volumes englobants, jusqu'à obtenir un volume unique représentant l'ensemble de la scène qui devient la racine de l'arbre.

La difficulté est bien évidemment de trouver les primitives à regrouper.

La méthode *incremental tree-insertion*

On part avec un arbre vide, puis on ajoute les primitives et leur volume englobant une par une. Étant donné que le but est de minimiser le volume de l'arbre total, on insère chaque primitive au niveau de l'arbre où l'augmentation du volume sera le plus petit.

Cette méthode est essentiellement utilisée dans les algorithmes de *raytracing*.

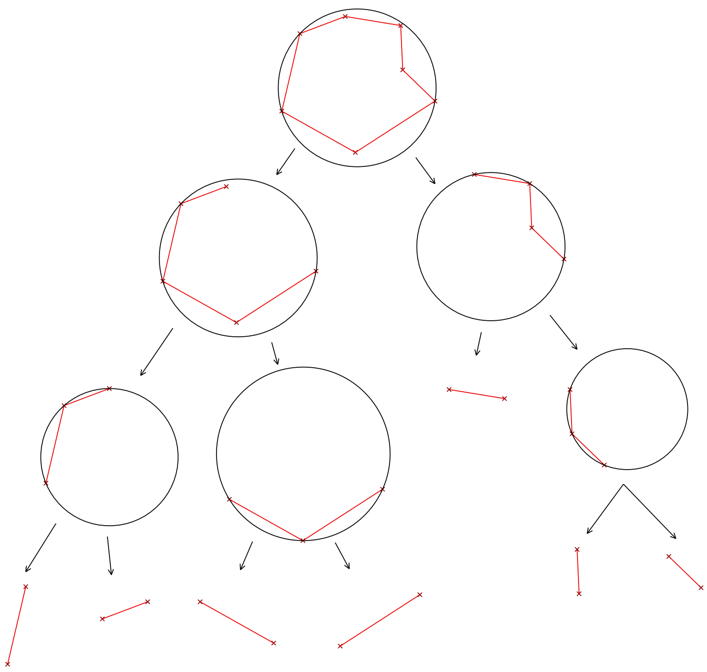

Fig. 3.1 – Représention d'une hiérarchie de sphères englobantes. Chaque nœud contient une sphère englobant un objet complexe. Cet objet est coupé en deux dans les fils du nœud jusqu'à atteindre les feuilles qui ne contiennent, elles, que des primitives de base.

La méthode *top-down*

C'est la méthode la plus courante dans les algorithmes de construction. Le principe est simple : on commence par trouver un volume englobant la scène dans son ensemble, puis on le divise, on cherche pour chaque partie les primitives incluses et on trouve un nouveau volume englobant ces primitives. Par récursion, on obtient la hiérarchie (Fig. 3.2).

Elle est assez simple à mettre en place, mais coûteuse pour une application temps-réel s'il faut la construire pendant le rendu.

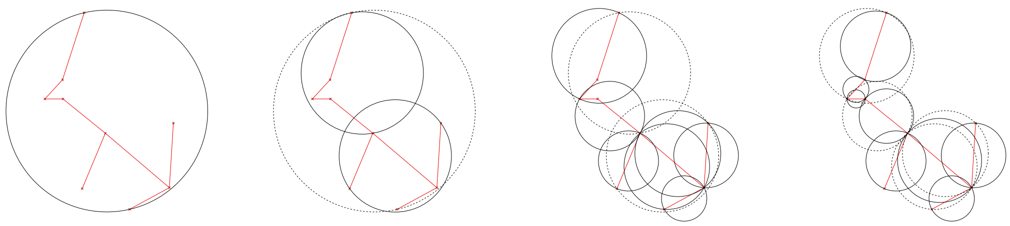

Fig. 3.2 – Construction récursive d'une hiérarchie de volumes englobants

D'une manière générale c'est cette dernière méthode qui est la plus utilisée.

Dans une application temps réel aucune de ces méthodes ne peut être appliquée à chaque frame, c'est pourquoi la construction des hiérarchies est faite lors d'une étape de pré-traitement et n'a donc pas besoin d'être réalisée en temps réel où on se contente d'effectuer de petites mises à jours sur les hiérarchies après des transformations de l'objet (translations, rotations, déformations). Cela permet d'effectuer des optimisations qui se traduiront par une amélioration des performances pendant l'usage en temps réel des hiérarchies au moment de la détermination des collisions.

Un autre élément à prendre en compte dans la construction des hiérarchies est la structure de données qui peut être statique ou dynamique.

Une structure statique, une fois créée, ne peut pas être facilement modifiée pour ajouter ou enlever des objets de la scène et doit alors être reconstruite entièrement. Elle est en générale construite en appliquant une approche *top-down*.

Au contraire, une structure dynamique permet de faire des mises à jour sur la scène sans recalculer l'ensemble des hiérarchies, ce qui peut être très long. Cependant ces mises à jour doivent être effectuées en temps réel nécessitant des opérations très optimisées. En général les structures dynamiques sont construites par la méthode *bottom-up*.

Conclusion : Dans *Virtual Choreographer*, chaque scène est décrite avec tous les objets qu'elle comporte. Nous avons donc choisi pour *Virtual Choreographer* une structure statique. Cette approche est également celle de la plupart des packages de détection de collision disponibles, partant du principe qu'une fois que l'environnement a été modélisé, il y a peu de chances qu'il soit modifié.

3.1.2 Utilisation de hiérarchies

Le but des hiérarchies de volumes englobants est de faciliter les tests de collisions entre deux objets. Pour cela, l'algorithme de calcul de collision appliqué est le même quel que soit le choix du volume englobant (algorithm 1).

L'algorithme est appelé sur A et B qui sont des nœuds des deux hiérarchies testées, initialement les racines des hiérarchies.

L'algorithme parcourt ensuite récursivement les hiérarchies jusqu'à trouver des triangles des maillages qui se chevauchent ou épuiser une des hiérarchies.

Algorithm 1 FindFirstHitCD(A,B)

```
Ensure: returns ({TRUE, FALSE})
  if isLeaf(A) and isLeaf(B) then
    for all triangle pair T_A ∈ A_c and T_B ∈ B_c do
      if overlap(T_A,T_B) then
        return TRUE
      end if
    end for
  else if isNotLeaf(A) and isNotLeaf(B) then
    if Volume(A) > Volume(B) then
      for all child C_A ∈ A_c do
        FindFirstHitCD(C_A,B)
      end for
    else
      for all child C_B ∈ B_c do
        FindFirstHitCD(A,C_B)
      end for
    end if
  else if isLeaf(A) and isNotLeaf(B) then
    for all child C_B ∈ B_c do
      FindFirstHitCD(A,C_B)
    end for
  else
    for all child C_A ∈ A_c do
      FindFirstHitCD(C_A,B)
    end for
  end if
  return FALSE
```

Le test sur les volumes de A et B permet de descendre systématiquement le nœud ayant le plus grand volume ce qui tend à améliorer les performances.

Cependant cet algorithme générique n'utilise pas les propriétés de facilité de test d'intersection des volumes englobants. Aussi on utilise un autre algorithme de parcours (algorithm 2) utilisant cette fois les volumes englobants des nœuds pour descendre dans l'arbre. Dans notre cas, c'est cet algorithme que nous avons également choisi.

Algorithm 2 TraverseTrees(A,B)

```
Ensure: returns ({TRUE, FALSE})
  if b(A) ∩ b(B) ≠ 0 then
    if isLeaf(A) then
      if isLeaf(B) then
        for all triangle pair T_A ∈ A_c and T_B ∈ B_c do
          if overlap(T_A,T_B) then
            return TRUE
          end if
        end for
      else
        for all child C_A ∈ A_c do
          TraverseTrees(C_A,B)
        end for
      end if
    else
      for all child C_B ∈ B_c do
        TraverseTrees(A,C_B)
      end for
    end if
  end if
  return FALSE
```

L'étape suivante est le choix du volume englobant utilisé dans la hiérarchie ce que nous allons voir dans les algorithmes présentés qui utilisent respectivement les arbres d'OBB, les arbres d'AABB et les arbres de k-dop.

Ce choix est déterminant et guidé par des contraintes naturelles :

- Le volume englobant doit « coller »au plus près à l'objet afin d'en minimiser le volume global ;
- Le test entre deux objets doit être le plus rapide possible ;
- Comme le volume doit suivre l'évolution de l'objet, la mise à jour doit être rapide.

3.2 Volumes englobants et packages existant

Sur internet, chaque type de hiérachie est lié à un – ou plusieurs – packages de traitement de collisions. C'est pourquoi nous donnerons pour chaque volume englobant un package les implémentant. Ces packages nous ont également permis de bénéficier de *benchmarks* afin de comparer les performances respectives des volumes.

Pour informations, les packages de détection de collision peuvent être caractérisés par plusieurs aspects en plus de l'efficacité :

- *Nbody ou 2Body* : un package est NBody s'il peut être appliqué sur n objets et 2Body si on ne peut l'appliquer que sur deux objets à la fois. On peut toutefois noter qu'un package 2Body peut facilement être rendu NBody en lui ajoutant une première passe de détection de collision comme celle vue dans le chapitre 2.2.
- Les types d'objets acceptés : certaines méthodes imposent d'avoir des objets convexes en entrée, ou bien prennent des soupes de polygones (objets sans contrainte sur leur forme telle que la convexité...). Cependant, il existe des méthodes pour se ramener à des objets convexes afin de permettre de traiter les soupes de polygones.

3.2.1 Arbres OBB – *RAPID* et *VCollide*

Les OBB sont des boîtes orientées pour mieux convenir aux contours de l'objet (Fig. 3.3). Elles sont décrites dans l'article de Gottschalk, Lin et Manocha [4].

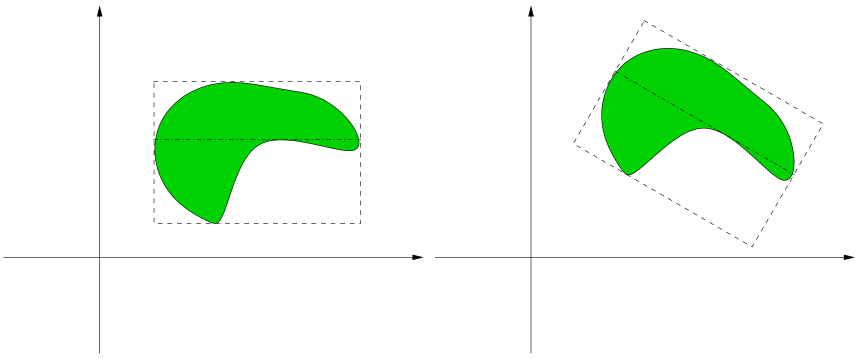

FIG. 3.3 – OBB offrant peu d'espace vide quelle que soit l'orientation de l'objet

Construction d'une OBB

L'étape cruciale de la construction d'une OBB est la détermination de la meilleure orientation de la boîte. Pour cela, on utilise un traitement statistique nommé *ACP (Analyse en Composantes Principales)* ([15] et [16]).

L'analyse en composantes principales est une technique mathématique permettant de réduire un système complexe de corrélations en un plus petit nombre de dimensions. Elle est beaucoup utilisée dans le domaine de l'analyse de données ou de la compression.

Grossièrement, considérant un ensemble d'informations, elle consiste à placer des axes « expliquant » au mieux ces informations. Dans notre cas, nos informations sont des points et les résultats de l'application de l'ACP sont les axes principaux de la répartition des points. Ce sont ces axes suivant lesquels les points sont répartis qui orienteront la boîte (Fig. 3.4).

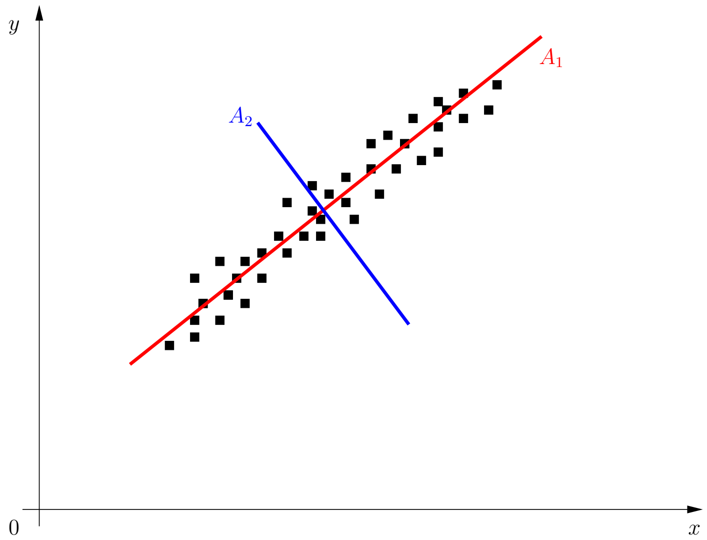

FIG. 3.4 – Sur cette distribution de points, l'axe A_1 est l'axe principal décrivant la répartition des points et l'axe A_2, orthogonal à A_1, le second axe de répartition des points.

Pour calculer les axes à partir des sommets de l'objet, on commence par calculer la moyenne μ (3.1), puis la matrice de covariance C (3.2). Si les sommets du i-ème triangle sont p^i, q^i et r^i, on calcule pour n triangles :

$$\mu = \Sigma_{i=0}^{n}(p^i + q^i + r^i) \tag{3.1}$$

$$C_{jk} = \frac{1}{3n}\Sigma_{i=0}^{n}(\overline{p_j^i p_k^i} + \overline{q_j^i q_k^i} + \overline{r_j^i r_k^i}), \quad 1 \leq j,k \leq 3 \tag{3.2}$$

où $\overline{p^i} = p^i - \mu, \overline{q^i} = q^i - \mu, \overline{r^i} = r^i - \mu$. Chacun d'eux est un vecteur à trois composantes représentant les coordonnées sur les axes repère, par exemple, $\overline{p^i} = (\overline{p_1^i}, \overline{p_2^i}, \overline{p_3^i})$ et les C_{jk} sont les éléments de la matrice de covariance de dimension 3x3.

Ensuite, on calcule les valeurs propres de la matrice, puis on obtient les vecteurs propres qui forment une base après les avoir normalisés (étant deux à deux orthogonaux). Ils représentent également les axes de plus grandes variances et permettent ainsi d'orienter la boîte. Une fois que nous avons ces vecteurs, il suffit de projeter les points suivant chacun des axes pour obtenir la boîte.

Malheureusement, il peut y avoir des erreurs d'orientation pour certaines distributions spatiales. En effet lors du calcul de la matrice de covariance, on tient également compte des points à l'intérieur de l'objet et un regroupement très dense de points à l'intérieur de l'objet peut amener la boîte englobante à s'aligner dessus. Une solution est de ne considérer que l'enveloppe convexe de l'objet pour calculer la matrice de covariance.

L'arborescence de OBB est construite par la méthode *top-down*, en coupant en deux la boîte suivant l'axe principal (l'axe A_1 de la figure 3.4) et en répartissant les polygones suivant le côté où ils se trouvent.

Mise à jour

Dans le cas d'objets rigides, la mise à jour des OBB est très aisée, aussi bien pour les translations que pour les rotations, puisqu'il suffit d'appliquer la transformation sur la boîte. Par conséquent, la mise à jour d'une hiérarchie est également très simple puisqu'il suffit de parcourir l'arbre OBB de la racine aux feuilles en mettant à jour les boîtes de chaque nœud.

Par contre, si l'objet se déforme, il faut recalculer les axes ce qui est coûteux. Les OBB s'appliquent donc assez mal à des objets déformables.

Test de chevauchement

La deuxième difficulté est de tester le chevauchement de deux OBB. En effet, l'orientation de ces boîtes pouvant varier, il faut en tenir compte dans le test ce qui amène à effectuer 144 tests faces-côtés ce qui est impensable. Pour cela on utilise une méthode de test rapide : *SAT (Separating Axis Test).*

Cette méthode est basée sur le théorème de *l'axe séparateur* : deux polygones sont disjoints s'il existe au moins un axe séparateur. Un axe est dit séparateur pour deux polygones s'il existe un plan dans l'espace orthogonal à cet axe qui sépare les deux polygones, et que cet axe soit orthogonal à la face d'un des deux polygones ou orthogonal à deux côtés de chaque polygone.

Tester si un axe est séparateur est très simple en projetant les centres des boîtes ainsi que leur radius (distance la plus grande du centre aux coins de la boîte) et de tester si la somme des radii est plus petite que la projection de la distance entre les centres. Si c'est le cas, les intervalles sont disjoints et par conséquent les boîtes aussi (Fig. 3.5).

En utilisant ce test rapide, on arrive à 15 tests dans le cas le pire puisque chaque boîte a 3 uniques orientations des faces et 3 uniques orientations des côtés (soit donc 3 faces pour la première boîte, 3 faces pour la seconde et 9 combinaisons deux à deux des côtés).

Cependant il est possible de se ramener à 6 tests comme le suggère Van den Bergen [5] en ne faisant pas les 9 derniers tests qui, d'après des résultats pratiques, servent rarement. Cela revient alors à transformer une des OBB et AABB orientée dans le repère de l'autre, faire un test de chevauchement des boîtes, puis recommencer pour la seconde boîte (Fig. 3.6 et 3.7).

Cette réduction du nombre de tests peut parfois conduire à croire, à tort, que deux boîtes se chevauchent alors que ce n'est pas le cas, cependant il n'y a pas de risque de manquer une intersection par cette optimisation.

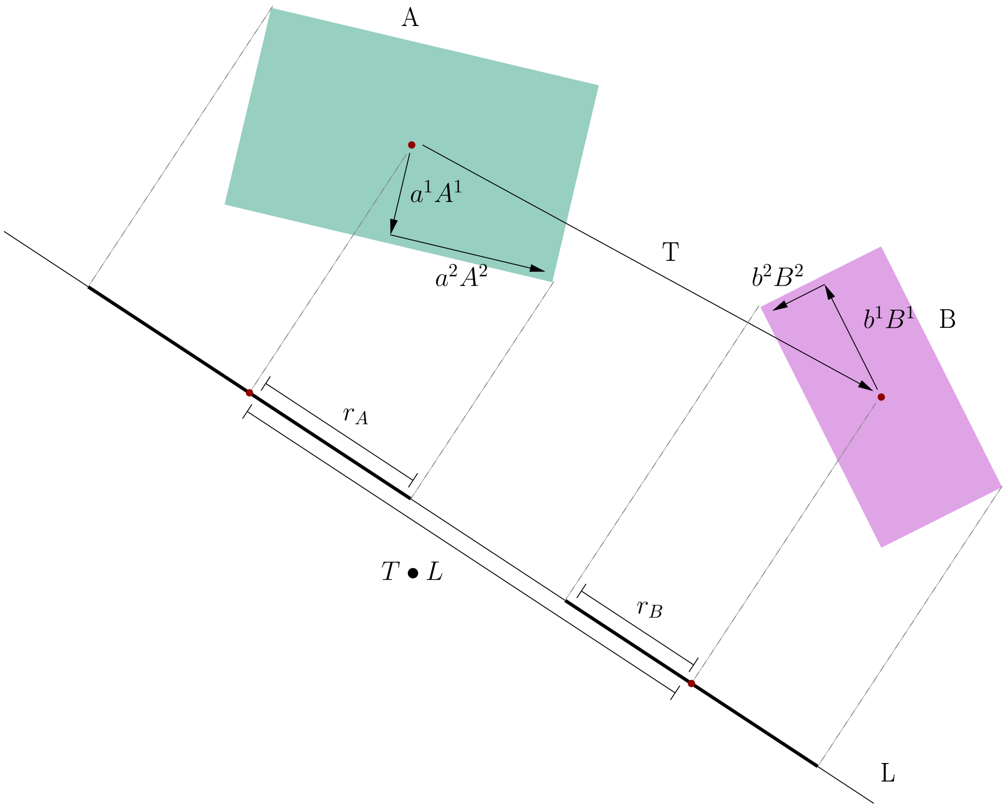

FIG. 3.5 – $\vec{L}$ est un axe de séparation pour les OBB A et B car A et B deviennent des intervalles disjoints une fois projetés sur $\vec{L}$; les demi-dimensions (ou *radii*) de A et B sont a^i et b^i avec $i = 1, 2, 3$; les axes de A et B sont les vecteurs unitaires $\vec{A}^i$ et $\vec{B}^i$ avec $i = 1, 2, 3$.

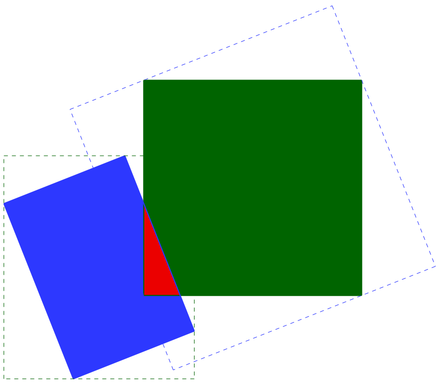

FIG. 3.6 – Le test indique un chevauchement correct.

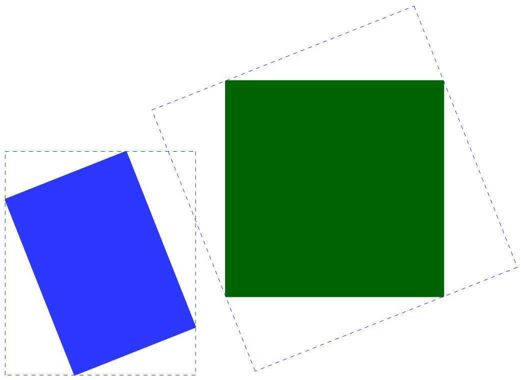

FIG. 3.7 – Le test SAT ne trouve pas de chevauchement des boîtes englobantes ce qui correspond bien à une situation de non chevauchement.

Caractéristiques des packages

Les hiérarchies OBB sont implémentées dans le package *RAPID* [10], qui gère uniquement la détermination de collisions, lui même utilisé dans le package *VCollide* [7]. VCollide est NBody et permet de traiter des soupes de polygones sans conditions particulières. Cependant, il est difficile de construire la hiérarchie de boîtes OBB et il peut parfois croire que des objets se chevauchent alors que ce n'est pas le cas à cause de la transformation des OBB en AABB.

3.2.2 Arbre d'AABB – *SOLID*

Les hiérarchies d'AABB ont été développées et implémentées par Van den Bergen [5]. Cette méthode est moins efficace que celle des boîtes englobantes orientées, mais présente tout de même quelques avantages.

D'une part, elle est moins gourmande en mémoire. D'autre part la construction et la mise à jour est plus simple sans avoir des performances catastrophiques. De plus elle permet de gérer les déformations d'objets.

La construction de l'arbre se fait par une approche *top-down* en partitionnant la boîte suivant l'axe le plus long. La construction de l'arbre est plus rapide que pour les OBB pour lesquelles la recherche de la bonne orientation implique des calculs supplémentaires.

Le test de chevauchement est réalisé en appliquant la méthode SAT comme les arbres OBB.

Mise à jour de la hiérarchie

Les arbres AABB se prêtent bien à la gestion d'objets déformables (ensemble de primitives dont les coordonnées évoluent au cours du temps). La gestion des déformations est un problème délicat car reconstruire l'ensemble de l'arbre après une déformation occasionnerait un coût trop élevé. Par contre, il est bien plus judicieux de mettre à jour l'arbre en agissant sur les boîtes englobantes aux nœuds de l'arbre. L'intérêt des AABB vient alors de la propriété suivante : soit S un ensemble de primitives, S^+ et S^- deux sous-ensembles de S tels que $S = S^+ \cup S^-$, B^+ et B^- les deux plus petites AABB de S^+ et S^-, et B la plus petite AABB englobant $B^+ \cup B^-$, alors, B est la plus petite AABB de S.

Cette propriété, partagée par les k-dops mais pas par les OBB, permet de mettre à jour l'arbre en adoptant une approche *bottom-up* : on descend aux feuilles, on recalcule leur AABB, puis on remonte l'arbre, en recalculant les AABB des nœuds à partir des AABB des fils. Cela revient à un parcours postfixe de l'arbre AABB.

Le défaut de cette méthode est que lors de fortes déformations, les boîtes recalculées risquent de plus se chevaucher que les boîtes d'un arbre que l'on aurait entièrement reconstruit. Cependant, on observe peu de baisse de performances tant que les relations d'adjacences entre les triangles du maillage sont conservées.

Caractéristiques du package

Le package SOLID utilisant les arbres AABB permet de gérer des soupes de polygones. Des tests pratiques montrent que les arbres AABB restent nettement moins performant que les arbres OBB pour les collisions sans déformations.

3.2.3 Conclusion intermédiaire : arbres OBB – arbres AABB

Au vue des performances de ces deux types de hiérarchies, les arbres OBB restent les plus intéressants. La diminution du coût de construction des arbres AABB entraînent également une perte importante en précision pour des performances inférieures aux arbres OBB dans la majorité des cas de collisions. L'intérêt des arbres AABB vient essentiellement du traitement simple des déformations. Cependant pour que ce traitement soit efficace, il est impératif que les déformations ne soient pas trop importantes. Notamment les relations d'adjacence entre les faces du maillage ne doivent pas être modifiées et la déformation ne doit pas radicalement changer les proportions de l'objet. Or les déformations possibles dans *Virtual Choreographer* dépassent le cadre des déformations à la suite de chocs (notamment lors d'interpolations d'objets), et il n'est pas assuré que l'arbre AABB permette de gérer correctement ces déformations.

3.2.4 Arbres k-dops – *QuickCD*

Décrits dans la thèse de J. T. Klosowski [3], les *k-dops (discrete orientation polytopes)* sont des polygones convexes dont les faces sont déterminées par des demi-plans dont les normales proviennent d'un *ensemble fixé de k orientations* (Fig. 3.8 et 3.9). Par exemple, les boîtes AABB sont des 6-dops dont les vecteurs orientations sont les vecteurs du repère.

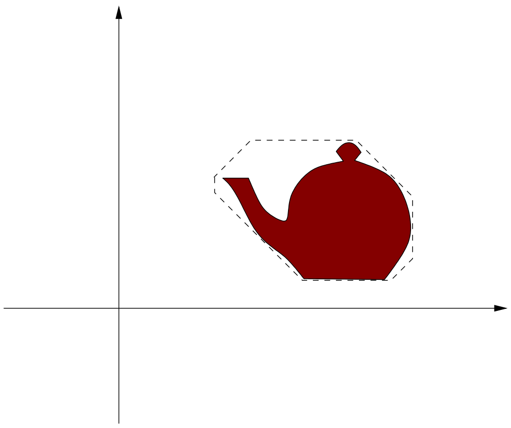

FIG. 3.8 – La boîte englobante est optimale initialement (8-dop)

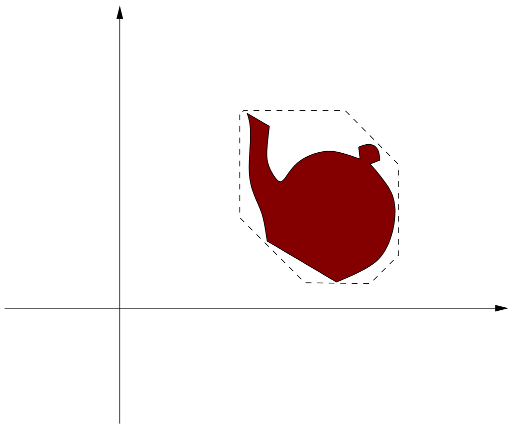

FIG. 3.9 – Approximation après rotation de l'objet (8-dop)

Construction des k-dops

Les k-dops permettent d'avoir un contrôle sur le degré de complexité du polygone englobant. Plus la valeur de k est élevée, plus il sera « proche »de l'objet.

Par ailleurs, il est très simple de calculer l'enveloppe d'un objet. En effet, il faut voir les k-dops comme des intervalles basés sur $k/2$ directions. Par exemple dans le cas d'un 8-dop, on se fixe quatre vecteurs de base (par exemple en 2 dimensions $(1,0)(0,1)(1,1)(1,-1)$) puis on calcule le produit scalaire de ces vecteurs avec les sommets de l'objet considéré et on garde le maximum et le minimum sur chaque direction. La construction d'un k-dop est obtenue par le calcul de produits scalaires entre les vertexes de l'objet et les k vecteurs. Dans le cas d'un objet contenant n vertexes, la construction en une complexité en temps de $O(kn)$ contre $O(n)$ pour la construction d'un AABB ce qui reste dans le même ordre de grandeur à une constante près.

De plus, il est judicieux d'utiliser des vecteurs composés de 0 et de 1 pour les axes (il n'est pas nécessaire qu'ils soient normalisés) afin de simplifier le calcul des produits scalaires.

Sur la figure 3.10, on voit graphiquement la construction d'un 8-dop d'axes $(0,1),(1,0),(1,1),(1,-1)$ pour le triangle $(1,2),(5,1),(3,5)$. L'intersection des projections des sommets extrêmes sur chacun des axes forme les contours du 8-dop.

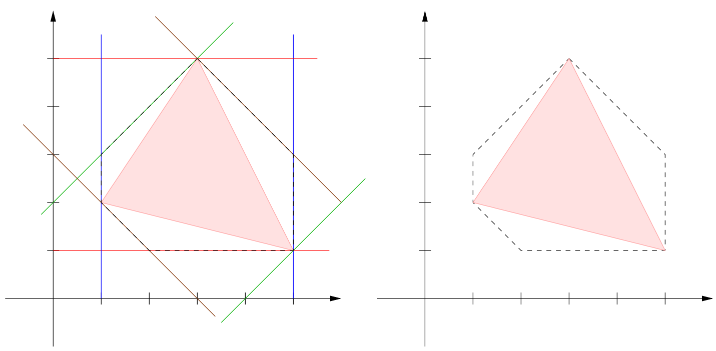

FIG. 3.10 – Construction d'un 8-dop sur un triangle. Le k-dop est obtenu en prenant l'intersection des intervalles projetés sur chacun des quatre axes fixés.

Pour construire la hiérarchie de k-dops, on applique l'approche *top-down* en subdivisant chaque ensemble en deux sous-ensembles par la méthode *splatter*. Cette méthode consiste à projeter le centre de gravité de chaque triangle du maillage sur chacun des axes du k-dop, puis de choisir l'axe sur lequel on observe la plus grande variation (Fig. 3.11). Une fois connu l'axe suivant lequel on va séparer l'ensemble, il reste à déterminer le point de subdivision. Pour cela, on prend la moyenne des coordonnées des projections des centres de gravité sur l'axe de subdivision. Les résultats expérimentaux montrent que prendre la moyenne plutôt que simplement le milieu de l'axe minimise le volume global de l'arbre.

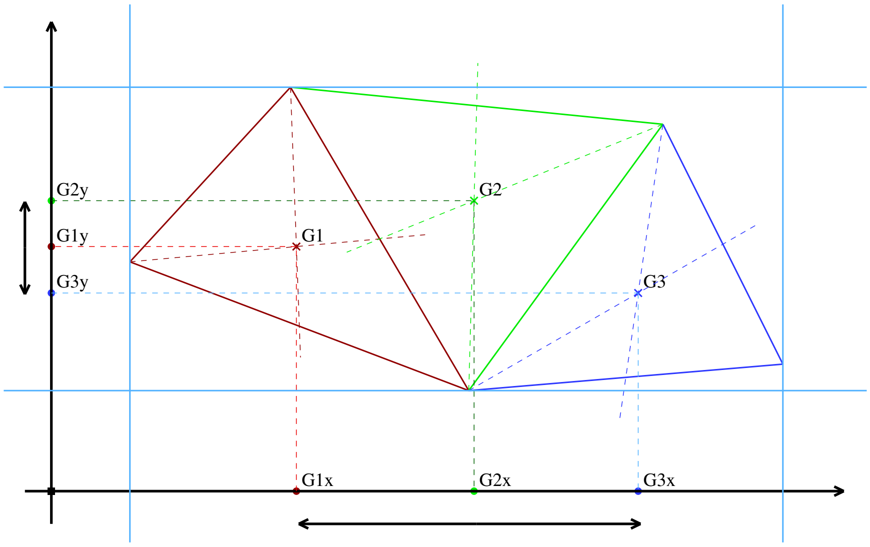

FIG. 3.11 – Chaque centre de gravité des triangles est projeté sur sur les deux axes. Sur l'axe Ox, on observe une plus grande variation entre les projections que sur l'axe Oy, on choisit donc de séparer l'objet orthogonalement à cet axe (4-dop).

Mise à jour

Étant donné que les axes du k-dop sont fixés initialement, après une rotation de l'objet, on ne peut pas calculer le k-dop de l'objet transformé en faisant une simple rotation du k-dop. Cependant, il est tout de même possible de traiter les rotations sans recalculer une nouvelle hiérarchie de k-dop et en se contentant de mettre à jour chaque nœud interne au prix d'une approximation.

On calcule en effet le k-dop résultant en appliquant la transformation sur les sommets au coin du k-dop, puis en construisant un nouveau k-dop uniquement sur les sommets transformés. Ainsi, le travail est réduit tant que l'objet contient moins de sommets que le k-dop. Le défaut de cette approximation est qu'elle augmente nécessairement la taille du k-dop (Fig. 3.12).

Test de chevauchement

Vu que l'on considère les k-dops comme des intervalles alignés sur $k/2$ directions, il suffit donc de réaliser $k/2$ tests de chevauchement d'intervalles ce qui est beaucoup plus simple que les tests que nous avons vu sur les OBB.

Caractéristiques du package

Le package QuickCD de Klosowski [3] utilise les k-dops ; il est NBody, mais ne gère pas les déformations. Il obtient des performances proches, voire supérieures à RAPID pour des coûts mémoires bien moindre.

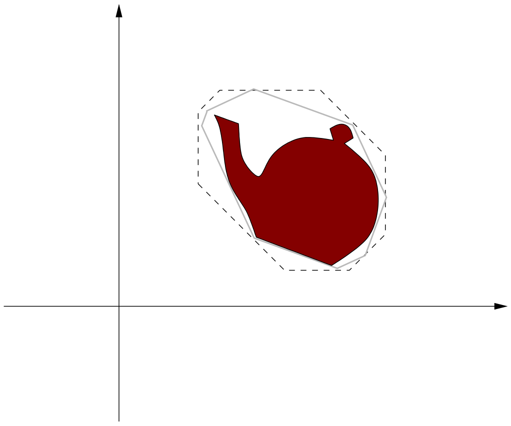

FIG. 3.12 – Cette figure montre le résultat de l'appoximation sur le k-dop à la suite d'une rotation : le tracé gris correspond au k-dop original après rotation, le tracé noir le nouveau k-dop obtenu à partir du tracé gris. On observe une augmentation du volume du k-dop par rapport au k-dop optimal (Fig. 3.9)

3.2.5 Conclusion

Notre objectif dans *Virtual Choreographer* est de réaliser des collisions simples entre des objets sur lesquels peuvent s'appliquer des transformations telles que des rotations ou des déformations. Les k-dops, par leur faible coût mémoire ($k/2$ axes et k valeurs par rapport aux stockage matrices de transformation des OBB), leur simplicité de construction et de test de chevauchement, sont les volumes englobants que nous avons choisis. Ils sont un peu moins performants lors des rotations, mais peuvent permettre de gérer les déformations sans reconstruire les hiérarchies en tentant de transposer la méthode des arbres AABB aux arbres k-dops.

Nous souhaitons tester deux k-dops :

- 14-dops : Axes $(1,0,0), (0,1,0), (0,0,1), (1,1,1), (1,-1,1), (1,1,-1), (1,-1,-1)$;
- 18-dops : Axes $(1,0,0), (0,1,0), (0,0,1), (1,1,0), (1,0,1), (0,1,1), (1,-1,0), (1,0,-1), (0,1,-1)$.

Chapitre 4

La physique

4.1 Réponse à la collision

La réponse à la collision est la dernière étape dans le traitement de la collision. Elle est beaucoup moins complexe au niveau algorithmique, mais nécessite une recherche théorique si on souhaite obtenir des effets réalistes.

En général, on modélise le contact entre deux corps par un point et un plan de contact tangent aux deux corps. On doit alors déterminer les efforts tangentiels et normaux (Fig. 4.4). Les effort tangentiels correspondent aux forces de frottement que l'on va, pour le moment, négliger. Il existe deux grandes méthodes pour calculer les efforts normaux : la méthode basée sur la notion d'impulsion et la méthode de pénalité.

4.1.1 Méthode d'impulsion

Les méthodes sont basées sur la notion de force *d'impulsion* : l'impact est une collision quasi-instantanée (soit de durée négligeable) pendant laquelle les deux objets (parfaitement rigides) exercent une force (dite impulsion) très importante sur eux-mêmes. Au moment de l'impact, la composante normale de la vitesse du point peut être modifiée en réponse à la collision. Une grandeur scalaire, le coefficient de restitution e, située entre 0 et 1, peut être appliquée à la vitesse résultante pour modéliser le degré d'élasticité de la collision (si $e = 1$, la norme de la vitesse résultante est égale à la norme de la vitesse initiale, il n'y a aucune perte d'énergie, la collision est parfaitement élastique). Le calcul du vecteur vitesse résultant peut être fait en appliquant la méthode décrite dans la section 4.1.4.

Le principe est simple : quand les deux objets sont suffisament proches, l'impulsion est calculée de sorte qu'elle soit capable de les séparer en une seule itération.

Cependant il est possible dans cette méthode de permettre un traitement plus précis de l'instant de la collision en « remontant »le temps jusqu'au point d'impact (en assimilant les objets à des points) à partir de l'instant où on a détecté un chevauchement des objets. Pour cela, on condidère la trajectoire de l'objet aux instants de la collision.

Une première méthode consiste à appliquer une recherche dichotomique sur les intervalles de temps. Si on sait qu'à l'instant t_{i-1} les objets étaient séparés alors qu'ils se chevauchent à l'instant t_i, on initialise l'intervalle de recherche à $[t_{i-1}, t_i]$ (Fig. 4.1).

Une seconde méthode, plus rapide, consiste à supposer la vitesse constante et le trajet linéaire sur l'intervalle de temps. Les positions du point de la trajectoire avant le chevauchement et à l'instant de la détection du chevauchement peuvent alors être utilisées pour déterminer un trajet linéaire et ainsi calculer analytiquement le point de contact avec une surface plane (Fig. 4.2).

Cette approche, bien que plus précise, peut devenir lourde dans un environnement complexe où de nombreuses collisions très rapprochées se produisent.

4.1.2 Méthode de pénalité

Dans les méthodes dites de *pénalité*, en général, les objets sont déformables, et la période de la collision ne peut être considérée comme négligeable. Ces méthode sont alors mieux adaptées pour traiter

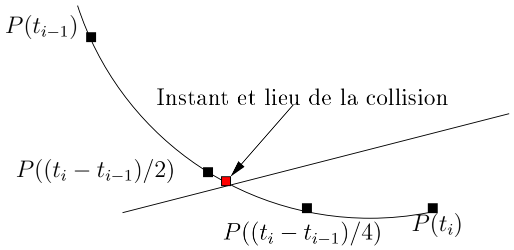

FIG. 4.1 – L'instant et lieu de collision est calculé précisément par dichotomie : dans un premier temps l'intervalle vaut $[t_{i-1}, t_i]$, puis est ramené à $[(t_i - t_{i-1})/2, t_i]$, et ainsi de suite.

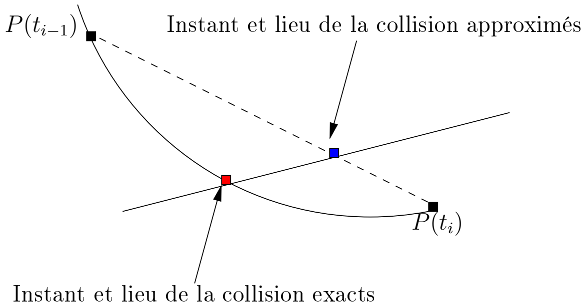

FIG. 4.2 – La trajectoire réelle du point est approximée par une trajectoire linéaire joignant $P(t_{i-1})$ à $P(t_i)$. Le point de contact approximé peut être alors calculé comme l'intersection d'une droite et d'un plan.

la collision. Le principe consiste à générer dynamiquement une force répulsive qui dépend de la valeur de l'interpénétration fictive entre les deux objets. La force engendrée par cette méthode nécessite plusieurs itérations pour séparer les deux objets, contrairement aux méthodes d'impact qui séparent les deux objets en une seule itération. Le plus souvent la force répulsive est générée par un mécanisme de type « ressort/amortisseur »(Fig. 4.3), et elle peut être donnée dans ce cas par la formule suivante :

$$\vec{F_c} = \begin{cases} (-\lambda x - \mu \dot{x})\vec{k} & \text{si x} < 0 \\ \vec{0} & \text{sinon} \end{cases}$$

où λ est la rigidité de la collision, μ la viscosité de la collision (qui représente la dissipation de l'énergie), x est la distance entre les deux points en contact, $\dot{x}$ est la dérivée de x, $\vec{k}$ est la direction de la collision (orientée d'un point vers l'autre).

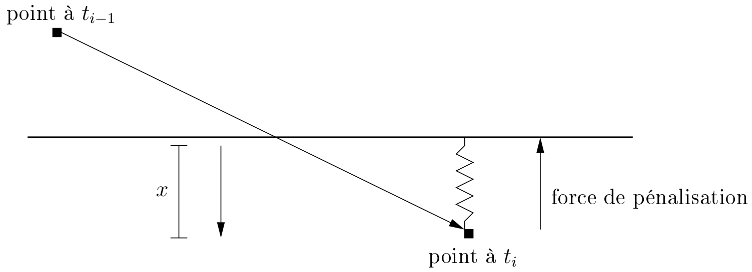

FIG. 4.3 – Après pénétration du plan par la trajectoire de l'objet, on applique un ressort de pénalisation à partir du point pénétré pour séparer les objets en plusieurs itérations.

4.1.3 Informations géométriques pour une collision

Pour utiliser ces méthodes, il faut en général avoir des informations précises telles que les normales aux faces en contact. Ces normales ne sont généralement pas fournies par les algorithmes de détermination de collision. Cela impose donc un traitement qui peut être coûteux.

Cependant, il faut noter que les expériences de O'Sullivan et Dingliana [14] ont montré qu'il est difficile pour un être humain de juger si la réponse à une collision en 3D est correcte. Il est même possible d'utiliser une réponse aléatoire qui sera malgré tout perçue comme crédible.

On voit donc qu'on peut se permettre de simplifier énormément le modèle physique de gestion de la collision.

4.1.4 Réponse physique aux chocs

Dans le cadre du stage, nous allons uniquement considérer des chocs élastiques (c'est à dire sans déformation des objets), dans des référentiels fixes.

Vecteur vitesse résultant

Nous utilisons une réponse simple par impulsion, sans recherche du moment précis du contact. Une fois qu'on a obtenu le point de contact et la normale par l'algorithme de détermination de collision, on projette le vecteur vitesse de chaque objet en collision sur la normale et sur le plan orthogonal à la normale. On prend ensuite l'opposé de la composante normale du vecteur vitesse et on garde la composante tangentielle (ou on la multiplie par un coefficient $k \leq 1$ représentant la perte d'énergie lors du contact) ce qui donne le nouveau vecteur vitesse de l'objet (Fig. 4.4).

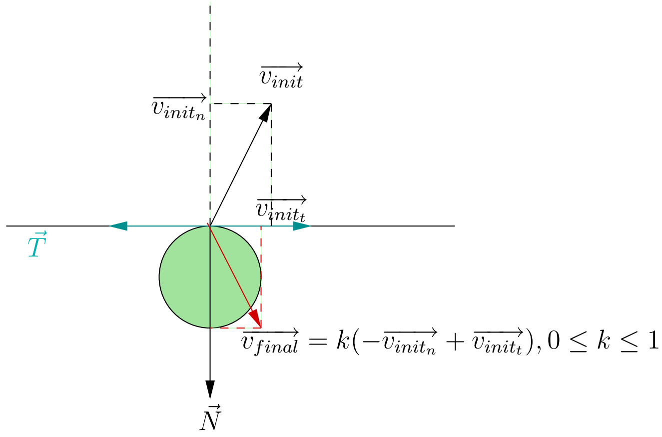

FIG. 4.4 – La composante normale du vecteur vitesse est l'opposée de la composante normale du vecteur initial. De plus, le vecteur vitesse résultant est multiplié par un coefficient $k \in [0, 1]$.

Pour obtenir la normale, on applique le principe suivant : lorsque l'on approxime les objets par des sphères, la direction de la normale est portée par la droite passant par les centres des deux sphères (Fig. 4.5). Le plan tangent à la collision est le plan orthogonal au vecteur normal obtenu précédemment.

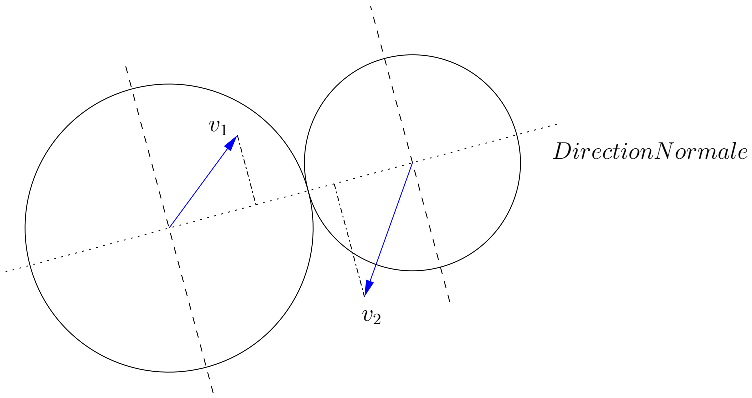

FIG. 4.5 – La normale est portée par la droite passant par les centres des sphères approximant les objets.

À l'implémentation, il est tout à fait possible d'utiliser le centre du volume englobant sans avoir à se ramener nécessairement à une sphère.

Il est à noter que cette approximation s'applique bien dans le cas d'objets de volumes comparables, mais fonctionnera mal dans le cas d'une collision point–plan où la normale obtenue ne correspondra pas à la normale du plan (Fig. 4.6).

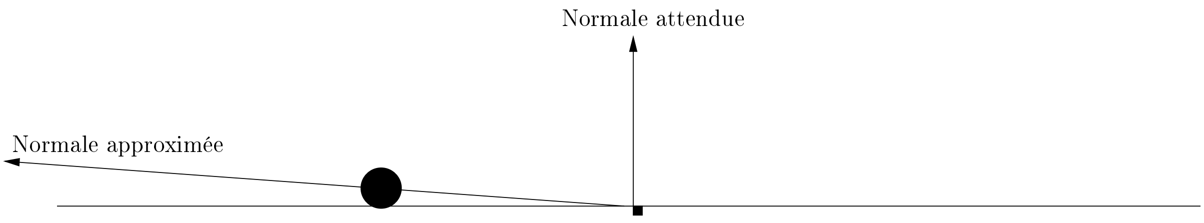

Fig. 4.6 – La normale, portée par la droite passant par le centre de la sphère et le centre du plan, est très éloignée de la normale utilisée dans une collision point–plan.

Vecteur rotation résultant

Lors du choc, les objets peuvent pivoter sur eux-même. Cette rotation est modélisée de la façon suivante :

- l'axe de rotation A_1 passe par le centre O_1 du solide et sa direction est obtenue en calculant le produit vectoriel du vecteur vitesse $\vec{v_1}$ par le vecteur joignant les centres des objets en contact $\overrightarrow{O_1O_2}$ (Fig. 4.7) ;
- la vitesse angulaire w_1 est proportionnelle à la distance d_1 du projeté du point de contact sur l'axe principal de l'objet au centre de l'objet. Ainsi, plus le choc se produit près du centre de gravité de l'objet et moins la vitesse angulaire est importante.

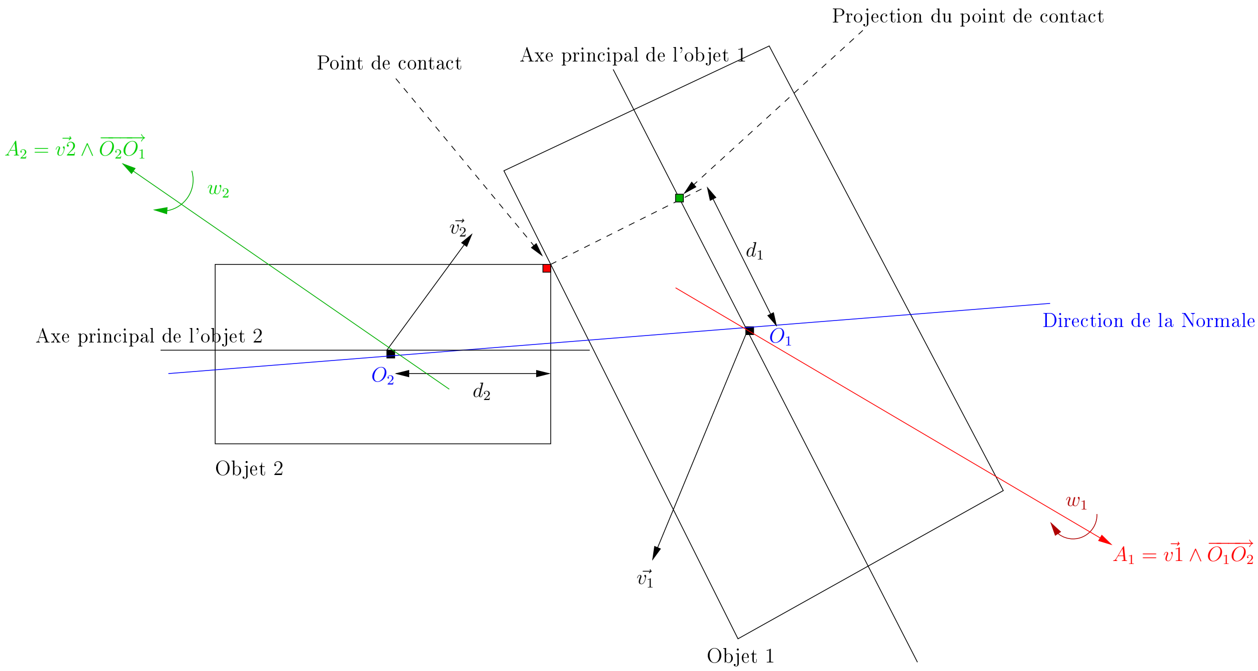

Fig. 4.7 – Les axes de rotations sont obtenus par produit vectoriel entre les vecteurs vitesses de chaque objet et le vecteur joignant les centres des objets. Les vitesses angulaires sont proportionnelles aux distances aux centres des projection du point de contact sur les axes principaux des objets.

4.2 Gravité

Nous souhaitons gérer la gravité dans le moteur afin de permettre la chute libre avec vitesse initiale. Il faut alors calculer la trajectoire de l'objet en fonction du temps.

Soit v_0 $(v_{0_x}, v_{0_y}, v_{0_z})$ le vecteur vitesse initial, après projection sur les axes du repère $Oxyz$. À l'instant

t, la variation de vitesse dv vaut :

$$\begin{cases} dv_x = 0 \\ dv_y = -dg \\ dv_z = 0 \end{cases}$$

où $\vec{g} = (0, 0, -g)$ est le vecteur gravité (Fig. 4.8).

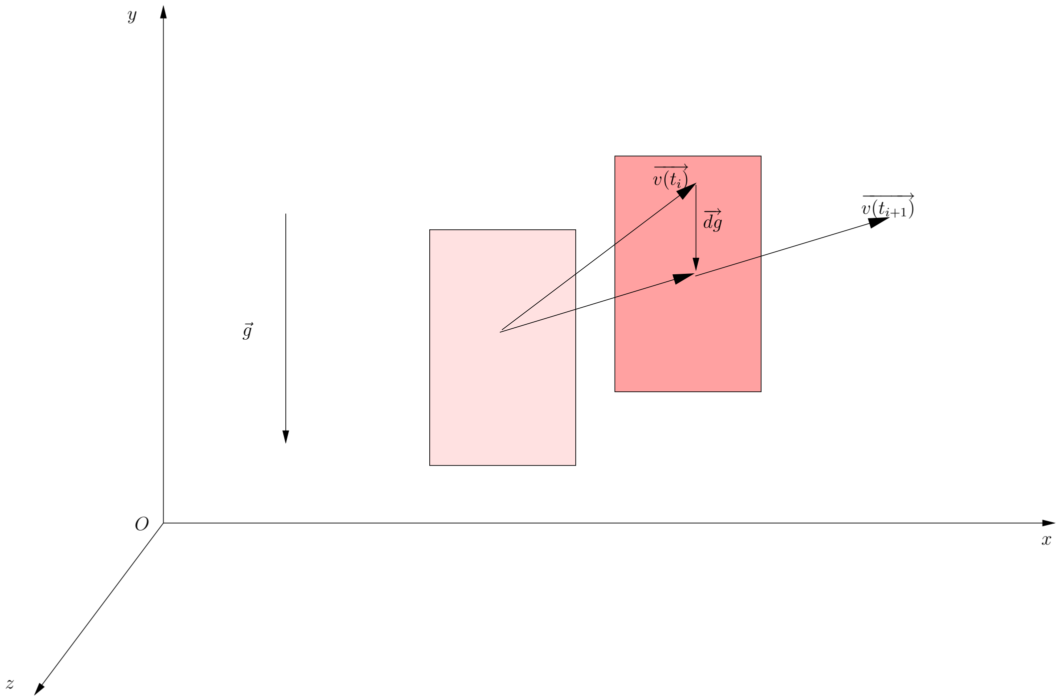

FIG. 4.8 – Pour une variation de temps dt, on ajoute le vecteur $\vec{dg}$ au vecteur vitesse $\vec{v}$ initial pour obtenir le nouveau vecteur vitesse.

4.3 Améliorations

Comme nous nous sommes essentiellement concentrés sur la partie détection de collisions, il est possible de faire de nombreuses améliorations au modèle physique que nous utilisons.

Par exemple, il pourrait être intéressant d'attribuer une masse aux objets géométriques afin d'en tenir compte lors du calcul de la réponse au choc. En effet, le comportement des objets n'est pas le même s'ils ont tous deux la même masse et si l'un est beaucoup plus lourd que l'autre. De même, on pourrait également gérer les matériaux constituants les objets, certains pouvant se fissurer, se briser d'une manière qui leur est propre. La gestion des frottements pourrait également être ajoutée.

Un autre point intéressant serait la résolution des contacts permanents entre un ou plusieurs objets (appelés contacts de repos ou encore d'équilibre) qui est l'un des problèmes de dynamique les plus compliqués de l'animation informatique. Pour se donner une idée de la difficulté, le principe est le suivant : pour chaque point de contact, il existe une force normale à la surface de contact comme dans le cas du contact par collision. L'objectif est de trouver la grandeur de cette force pour une configuration donnée des objets. Ces forces doivent :

1. être suffisament puissantes pour éviter une interpénétration ;
2. repousser uniquement les objets les uns des autres ;
3. devenir nulles au point de contact au moment où les objets commencent à se séparer.

La résolution de ce problème requiert l'utilisation de travaux complexes dépassant le cadre de ce TER–Stage.

Chapitre 5

Implémentation dans *Virtual Choreographer*

5.1 Objectifs

Le but que nous nous somme fixé pour incorporer la gestion des collisions dans *Virtual Choreographer* est de rester aussi modulaire que possible. Nous souhaitons essentiellement fournir une architecture de moteur physique avec l'application des forces « simples »et la gestion de collisions qui sera simple à améliorer.

De plus nous insistons particulièrement sur la partie concernant gestion des collisions qui est plus complexe au niveau algorithmique et structure de données que la partie physique pure.

5.2 Préparation du programme existant

Afin de faciliter la mise en place du moteur physique un certains nombres de modifications ont été nécessaires :

- un vecteur gravité a été ajouté dans le fichier de configuration de la scène afin de pouvoir spécifier une unique force de gravité par scène ;
- des classes permettant de manipuler le maillage des objets (accéder aux triangles du maillage, etc.) ont été codées, permettant de construire les hiérarchies de volumes englobants ;
- une classe d'attributs physiques des objets a été mise en place ;
- des transformations locales spéciales ont été ajoutées avant chaque objet dans le graphe de scène afin que les modifications des positions des objets dues à la physique se fassent par leur intermédiaire.

5.3 Organisation générale

5.3.1 Application de la physique

Chaque objet possède un attribut physique permettant de définir ses caractéristiques. Notamment, il possède un vecteur *impulsion* qui indique le vecteur vitesse courant de l'objet. Ensuite, c'est en agissant sur ce vecteur que l'on peut modifier le comportement de l'objet (par exemple à la suite de collisions), de même pour la vitesse angulaire. Ces modifications sont ensuite répercutées à chaque frame par l'appel d'une méthode de mise à jour des objets.

5.3.2 Gestion des collisions

Le traitement des collisions suit l'organisation décrite dans le rapport.

Dans un premier temps, on limite la combinatoire en appliquant l'algorithme *sweep and prune*. Pour cela, on construit trois tableaux d'intervalles (un par axe du repère) (Fig. 5.2) que l'on trie en appliquant le tri *qsort* de la librairie standard du C++. Par la suite, les mises à jours modifient les valeurs des éléments des intervalles, puis les tableaux sont triés par un tri à bulle que nous implémentons. À chaque frame, il suffit de parcourir les tableaux et construire trois listes de couples (une par axe) des *id* des objets correspondant aux intervalles qui se chevauchent. Les couples qui apparaissent dans les trois listes indiquent des objets qui se chevauchent et sont ajoutés dans la liste des couples d'objets susceptibles de se chevaucher.

Cette liste de couples est ensuite utilisée dans la phase de détermination de collisions. À l'initialisation de la scène, on construit les arbres de k-dop pour chaque objet (Fig. 5.1). Les arbres sont placés dans la classe *PhysicalProperties* ce qui évite de les construire pour les objets sur lesquels on ne souhaiterait pas appliquer de la physique. À chaque frame, la fonction de détermination de collision est appelée sur les couples d'objets en contact potentiel repérés lors de la phase précédente. Pour chaque couple, elle applique la fonction de parcours des arbres de k-dop des deux objets afin de déterminer s'ils sont réellement en contact. Si ce n'est pas le cas, on retire les couples de la liste.

Au final, on a la liste des couples d'objets qui sont réellement en situation de collision. Pour chaque couple, on résout la collision en appliquant la fonction physique qui se charge de séparer les objets et modifier leur trajectoire en agissant sur leurs attributs de la classe *PhysicalProperties* comme il a été vu à la section 5.3.1.

Virtual Choreographer permet de faire des rendus temps réels, mais également offline. Pour profiter pleinement de ces deux modes de rendu, nous modifions le comportement de la détermination de collision en fonction du mode demandé par l'utilisateur : dans un rendu temps réel, les arbres k-dop sont mis à jour avec les méthodes d'approximations pour les rotations et les déformations ; par contre, dans un rendu différé, on reconstruit les hiérarchies des objets modifiés, permettant d'avoir au final une gestion plus fine des collisions.

5.4 Organisation du code

5.4.1 Structure de l'implémentation

L'affichage au sein de *Virtual Choreographer* est découpé en plusieurs phases où viennent s'insérer le moteur physique. La fonction de mise à jour de la scène commence par mettre à jour les positions des objets en appliquant les transformations définies dans le script *.xml* décrivant la scène s'il y en a. Ensuite, les forces globales (gravité, etc.) sont appliquées modifiant les positions des objets. On cherche alors les éventuelles collisions et on les résout en modifiant de nouveau les positions, qui étaient incohérentes, des objets. Enfin, le rendu graphique est calculé et la scène s'affiche.

La difficulté de l'intégration vient du fait que la scène est décrite sous forme d'un graphe de scène et que les objets sont décrits dans leur propre système de coordonnées. Or pour gérer la physique, il faut avoir une vue de la scène dans le repère canonique afin de pouvoir comparer les positions des objets. Il faut donc en permanence faire attention aux système de coordonnées et appliquer les changements de repères adéquats. La figure 5.3 montre les regroupements que nous avons fait afin de pouvoir centraliser le moteur physique et le rendre le plus indépendant possible de *Virtual Choreographer*. Nous avons donc une classe *PhysicalOverview* qui s'occupe d'appliquer correctement les différentes passes du moteur physique. Il suffit donc de faire appel à la fonction de mise à jour de cette classe dans la fonction de rendu de *Virtual Choreographer* pour que la physique s'exécute.

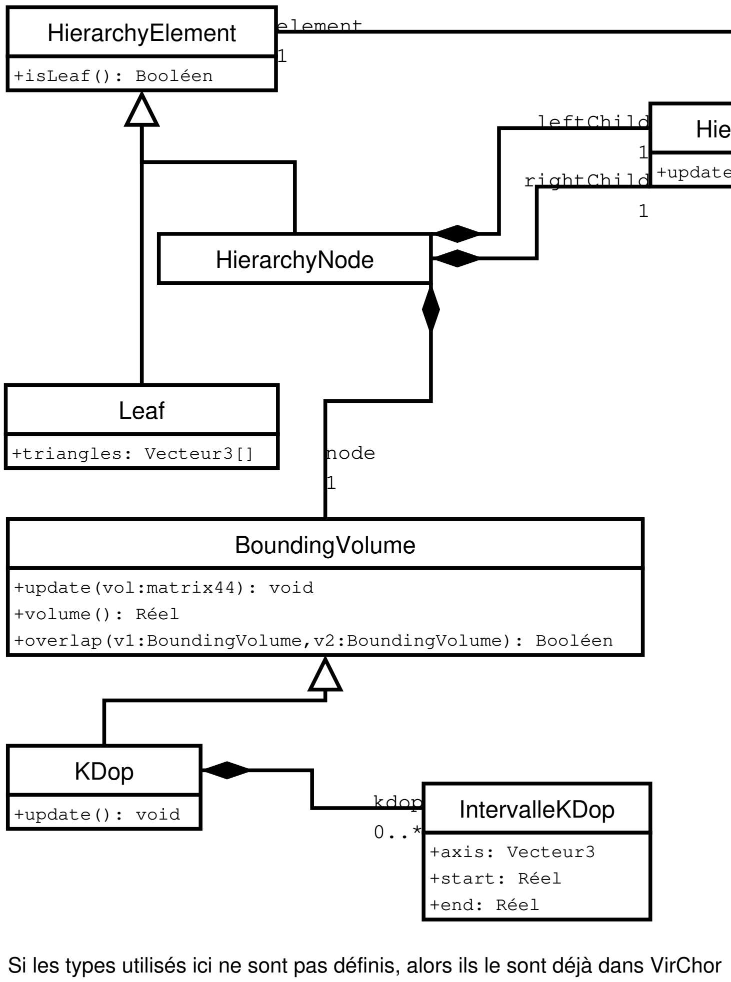

Fig. 5.1 – Diagramme de classes de la hiérarchie de k-dop. Les noeuds de l'arbre contiennent ici un k-dop, mais la définition de nos classes permet de changer aisément de volume englobant du moment que les fonctions de test d'intersection et de mise à jour sont implémentées.

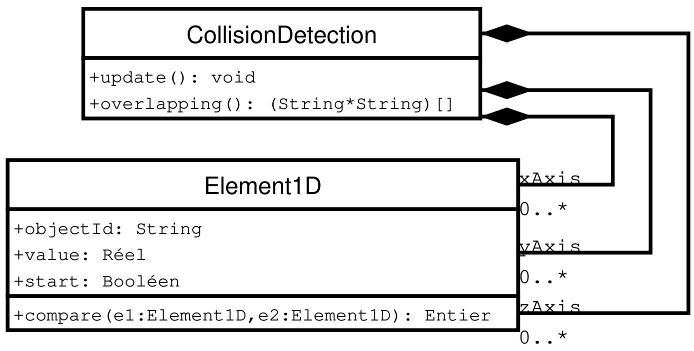

Fig. 5.2 – Diagrame de classes de la classe CollisionDetermination manipulant trois tableaux d'Element1D. Un Element1D est un élément d'intervalle (début ou fin d'intervalle signalée par le booléen *start*). Il contient en plus un champ *id* qui permet d'accéder à l'objet lié à l'intervalle.

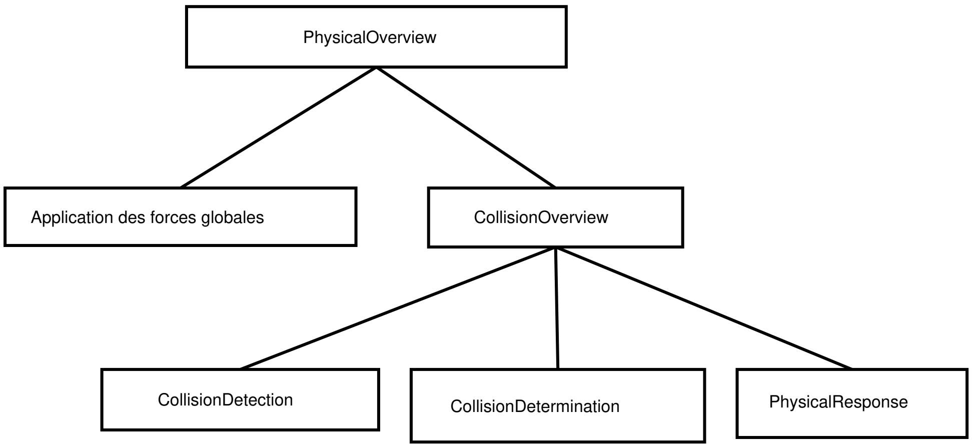

Fig. 5.3 – Structure du code utilisé dans *Virtual Choreographer*.

Chapitre 6

Conclusion

Nous avons pris un grand plaisir à suivre ce TER–Stage qui nous a été bénéfique sur bien des points.

Il nous a, en premier lieu, permis de découvrir un problème classique, mais complexe, de l'animation graphique que l'on retrouve dans un vaste champ d'applications (jeux, simulations réalistes, etc.). De plus, il nous a amené à effectuer un travail de recherche poussé passant par la lecture d'extraits de thèses et d'articles de recherches en plus des traditionnelles recherches de sites internet.

Mais plus encore, nous avons eu une grande liberté dans le choix du sujet. Le domaine de la physique étant extrêmement vaste, nous avons du nous auto-limiter et privilégier certains sujets. Nous avons donc travaillé sur les sujets qui nous motivaient le plus ce qui nous a soutenu pour réaliser les recherches.

Cependant nous regrettons de n'avoir pas pu consacrer plus de temps à l'implémentation de ces techniques, notamment à la phase de détermination des collisions.

Malgré tout, nous restons satisfaits de ce que nous avons fait et nous espérons que d'autres auront l'occasion de poursuivre notre travail puisqu'il reste de nombreux problèmes à couvrir dans ce domaine.

Bibliographie

[1] Tomas Akenine-Möller and Eric Haines. Real-Time Rendering A.K. Peters Ltd., 2nd edition.

[2] Tangi Meyer. Retour d'effort et réalité virtuelle : Proposition d'une boîte à outils pour l'intégration logicielle générique de la détection de collisions et de la simulation physique. 1er octobre 2003.
http://www.irisa.fr/bibli/publi/theses/2003/meyer/meyer.html

[3] James Thomas Kloswski. Efficient Collision Detection for Interactive 3D Graphics and Virtual Environments. 1998
http://www.ams.sunysb.edu/~jklosow/quickcd/QCD_publications.html

[4] S. Gottschalk, M. C. Lin, et D. Manocha. OBBTree : A Hierarchical Structure for Rapid Interference Detection. Sggraph'96.
ftp://ftp.cs.unc.edu/pub/users/manocha/PAPERS/COLLISION/obb.ps.gz

[5] Gino Van Den Bergen. Efficient Collision Detectin of Complex Deformable Models using AABB Trees. 6 novembre 1998.

[6] Kenneth E. Hoff III, Tim Culver, John Keyser, Ming Lin, et Dinesh Manocha. Fast Computation of Generalized Voronoi Diagrams Using Graphics Hardware.

[7] VCollide. Site de l'algorithme de Hudson
http://cs.unc.edu/~geom/V_COLLIDE/

[8] ICollide. Site de l'algorithme de Lin et Canny
http://www.cs.unc.edu/~geom/I_COLLIDE/index.html

[9] SOLID. Site de l'algorithme de Gino van den Bergen.
http://www.win.tue.nl/~gino/solid/index.html#papers

[10] RAPID. Site de l'algorithme de Gottschalk
http://cs.unc.edu/~geom/OBB/OBBT.html

[11] Mustapha Bismi. COLLISIONS : Oriented Bounding Box et SphereTree
http://membres.lycos.fr/javamus/articles/obb.pdf

[12] Dan Sunday. Bounding Containers for Polygons, Polyhedra, and Point Sets (2D and 3D)
http://softsurfer.com/Archive/algorithm_0107/algorithm_0107.htm

[13] Kenneth Sundaraj. Détection et traitement de collision en simulation dynamique
http://www.inrialpes.fr/sharp/people/sundaraj/papers/dea00.pdf

[14] O'Sullivan, C. Dingliana. Collisions and Perception
J. ACM Transactions on Graphics. Vol. 20, No. 3. Juillet 2001.
http://isg.cs.tcd.ie/cosulliv/publications.htm

[15] Principal Component Analysis.
J. Hollmen.
http://www.cis.hut.fi/~jhollmen/dippa/node30.html

[16] Analyse en composantes principales (ACP).
Cours de l'université de Liège.
http://www.ulg.ac.be/pedaexpe/cours/glosaire/acp.htm

Index