Titre: Title:	Détection de collisions en temps réel entre surfaces nurbs déformables
Auteur: Author:	Francis Pagé
Date:	2003
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Pagé, F. (2003). Détection de collisions en temps réel entre surfaces nurbs déformables [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7147/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/7147/
Directeurs de recherche: Advisors:	François Guibault
Programme: Program:	Unspecified

UNIVERSITÉ DE MONTRÉAL

DÉTECTION DE COLLISIONS EN TEMPS RÉEL ENTRE SURFACES NURBS DÉFORMABLES

FRANCIS PAGÉ
DÉPARTEMENT DE GÉNIE ÉLECTRIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE ÉLECTRIQUE)
MAI 2003

Your file Votre référence
ISBN: 0-612-86423-5
Our file Notre référence
ISBN: 0-612-86423-5

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé:

DÉTECTION DE COLLISIONS EN TEMPS RÉEL ENTRE SURFACES NURBS DÉFORMABLES

présenté par: PAGÉ Francis

en vue de l'obtention du diplôme de: Maîtrise ès science appliquées

a été dûment accepté par le jury d'examen constitué de:

Me. FARIDA Cheriet, Ph.D., président

M. FRANÇOIS Guibault, Ph.D., membre et directeur de recherche

M. YVES Martel, B.Ing., membre

REMERCIEMENTS

Je tiens à remercier sincèrement mon directeur de recherche, François Guibault. S'il n'avait pas été là, je n'aurais tout simplement pas continué mes études en maîtrise et je serais passé à côté de cette belle expérience. Il a cru en mes capacités dès le départ, avant même que je commence mon projet de recherche. J'ai souvent eu des périodes de démotivation, mais à chaque fois, je n'avais qu'à passer le voir à son bureau pour regagner confiance en moi et retrouver l'enthousiasme pour mon projet. Sans lui, j'aurais probablement abandonné avant même de commencer. Merci donc à toi François! Tu m'as permis d'apprendre énormément, de me discipliner, de croire en mes capacités et plus encore.

Je remercie Céline, ma mère, qui m'a toujours soutenu pendant mes études, qui a toujours cru en moi et qui a eu tant de petites attentions pour moi. Je remercie aussi Alain, mon père, qui depuis que je suis tout petit, a développé mon intérêt pour les sciences et le génie. Il a toujours été là pour répondre à mes questions et me donner de judicieux conseils. Sans mes parents, je ne sais pas trop ce que je serais devenu...

Finalement, je voudrais remercier tous les autres membres de ma famille pour leur présence et leurs encouragements. Je remercie mes amis, qui m'ont toujours encouragé et qui m'ont permis de me distraire de temps en temps!

Merci à vous tous, car sans vous, je n'aurais jamais pu me rendre où je suis maintenant.

RÉSUMÉ

Les jeux vidéo ont atteint un niveau de réalisme jamais vu auparavant. L'ombrage programmable, la simulation physique en 3D de même que les surfaces courbes vont bientôt devenir des caractéristiques communes à plusieurs jeux. La détection de collisions en temps réel, nécessaire à ce genre d'applications, est un problème difficile pour lequel la recherche est encore très active. Ce document présente une revue des algorithmes actuels de détection de collisions pour différents types de représentations d'objets. Par la suite, un nouvel algorithme de détection de collisions entre surfaces NURBS déformables sera décrit. Cet algorithme vise les applications en temps réel, nécessitant une détection rapide de collisions, mais non exacte. Ce qui a motivé le développement de cet algorithme est la quasi-absence de telles solutions adaptées aux surfaces paramétriques dans la littérature spécialisée. Les surfaces paramétriques étant de plus en plus utilisées dans le domaine des jeux, un algorithme rapide de détection de collisions adapté pour ces surfaces semblait une bonne avenue de recherche.

Ce nouvel algorithme fonctionne en créant des hiérarchies de boîtes englobantes orientées à partir des points de contrôle des surfaces. Il effectue ensuite un test d'intersection entre ces boîtes afin de déterminer s'il y a possibilité de collision entre les surfaces. Si c'est le cas, les surfaces sont subdivisées et l'algorithme est appelé récursivement avec ces nouvelles surfaces. La récursion est arrêtée lorsqu'un niveau de précision défini par l'usager est atteint. Les résultats sont les coordonnées des collisions dans l'espace, de même que les coordonnées paramétrique (u, v) de chacune des surfaces.

Le test d'intersection entre boîtes englobantes orientées est rapide et efficace. La construction d'une telle boîte à partir des points de contrôles est simple et mène

à une boîte épousant bien la forme de la surface. Ces caractéristiques sont la clé de cet algorithme. Comme les boîtes sont calculées seulement lorsque nécessaire, l'algorithme utilise peu de mémoire et est utilisable avec des surfaces déformables sans aucune dégradation de vitesse.

Les résultats obtenus ont permis de déterminer que l'algorithme est précis autant pour des surfaces immobiles que des surfaces se déplaçant dans l'espace. L'algorithme a aussi été testé dans un environnement de simulation des corps rigides. Des objets modélisés à l'aide de surfaces NURBS pouvaient être déplacés en appliquant des forces et une impulsion était calculée lors de collision. Les résultats de cette simulation ont été visuellement convaincants, quoiqu'un peu plus lents que souhaités. D'autres tests ont permis de déterminer que la vitesse d'une requête de détection de collisions est de l'ordre de la milliseconde, ce qui est assez rapide pour être utilisé dans certaines applications temps réel.

Cette recherche a permis de conclure qu'il est possible d'effectuer de la détection de collisions entre surfaces NURBS en temps réel. Par contre, l'algorithme étant quand même plus lent que les solutions déjà existantes pour les modèles polygonaux, la prochaine étape de recherche devrait porter sur ses optimisations possibles.

ABSTRACT

Video games have reached a new level of realism. Programmable shading, 3D physics simulation and curved surfaces will soon become standard features. Real time collision detection, needed for this kind of application, is a difficult problem where the research is always active. This document presents a survey of the current collision detection algorithms for different object representations. Then, a new algorithm for interactive collision detection between deformable NURBS surfaces is described. This algorithm is intended to be used in real time applications, which need fast, but not exact, collision detection. There are almost no collision detection algorithms dealing with parametric surfaces in real time. Moreover, parametric surfaces are used in an increasing number of games. These reasons motivated the choice of this research avenue.

This new algorithm works by creating oriented bounding boxes (or OBB) hierarchies with the surface control points. It then tests them for overlapping. If the test detects an overlap, the surfaces are subdivided into smaller NURBS surfaces and the algorithm is called recursively on these new surfaces. It stops when a certain precision level is reached, that is user definable as a function of the application. The results are the world space coordinates of the contact point, and the (u, v) parametric coordinates on both surfaces.

The use of OBB allows for fast and effective overlapping tests. The construction of an OBB with the surface's control points is simple and leads to a tight fitting bounding volume, which is the key of this fast collision detection algorithm. As boxes are only computed when needed, the algorithm uses a small amount of memory and can be used with deformable surfaces without any overhead.

The algorithm gives precise results for moving and static surfaces. It was tested in a rigid body simulation. All objects where modeled using NURBS surfaces and where moved by applying forces on them. When a collision was detected, an impulse was computed to prevent object interpenetration. The results of this simulation were visually convincing, but were a little slower than expected. Other tests have shown that a collision detection request takes about one millisecond, which is fast enough for some real time applications.

This research has shown that it is possible to do real time collision detection between NURBS surfaces. On the other hand, the algorithm is slower than existing ones for polygonal models, so the next step should be a research on the possible optimizations.

TABLE DES MATIÈRES

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT vii

TABLE DES MATIÈRES ix

LISTE DES TABLEAUX xii

LISTE DES FIGURES xiii

LISTE DES NOTATIONS ET DES SYMBOLES xv

LISTE DES ANNEXES xvi

INTRODUCTION 1

CHAPITRE 1 REPRÉSENTATION GRAPHIQUE D'OBJETS ET ALGORITHMES DE DÉTECTION DE COLLISIONS 6

1.1 Modes de Représentation 6

1.1.1 Représentation Polygonale 7

1.1.2 Fonctions Implicites 10

1.1.3 Surfaces Paramétriques 12

1.2 Détection de Collisions 16

1.2.1 Modèles Polygonaux 17

1.2.1.1 Hiérarchie de Volumes Englobants 17

1.2.1.2 Distance Entre les Objets 23

1.2.1.3 Décomposition de l'Espace 26

1.2.1.4 Détection dans le Domaine Image 31
1.2.2 Fonctions Implicites 35
1.2.2.1 Échantillonnage 36
1.2.2.2 Solution Numérique 37
1.2.3 Surfaces Paramétriques 38
1.2.3.1 Méthodes de subdivision 38
1.2.3.2 Arbres de Coquilles Sphériques 39

CHAPITRE 2 ALGORITHME DE DÉTECTION DE COLLISIONS . . 41
2.1 Construction des Boîtes Englobantes 42
2.1.1 Calcul de l'Orientation 44
2.1.2 Création de la Boîte 45
2.2 Intersection de Boîtes Englobantes Orientées 45
2.3 Subdivision d'une surface NURBS 47
2.4 Algorithme Global 47

CHAPITRE 3 RÉSULTATS ET DISCUSSION 55
3.1 Test de l'Influence du Niveau de Récursion 56
3.1.1 Résultats 57
3.2 Test de l'Influence du Taux de Collision 58
3.2.1 Résultats 59
3.3 Test de l'Influence de la Complexité des Surfaces 60
3.3.1 Résultats 62
3.4 Test de Précision 63
3.4.1 Résultats 65
3.5 Test d'Intégration au Moteur Physique 67
3.5.1 Résultats 68
3.6 Discussion 69

CONCLUSION 72

RÉFÉRENCES 77

ANNEXE 83

LISTE DES TABLEAUX

Tableau 3.1 Caractéristiques des surfaces du premier test. 57

Tableau I.1 Fonctions relatives aux variables d'état et constantes d'un corps rigide 93

Tableau I.2 Fonctions relatives aux variables auxiliaires d'un corps rigide 94

Tableau I.3 Fonctions relatives aux accumulateurs de force et moment de force d'un corps rigide 94

Tableau I.4 Autres fonctions d'un corps rigide 95

Tableau I.5 Fonctions relatives aux attributs d'un objet de détection de collisions 97

Tableau I.6 Fonctions du gestionnaire de physique pour l'ajout et la suppression d'objets 100

Tableau I.7 Fonctions du gestionnaire de physique pour définir les variables globales de la simulation 101

Tableau I.8 Caractéristiques des systèmes masses-ressort des différentes simulations. 106

LISTE DES FIGURES

Figure 1.1 Un tétraèdre. 8

Figure 1.2 Une bande et un éventail de triangles. 9

Figure 1.3 Un objet 2D et deux k-DOP correspondant. 22

Figure 1.4 Un polygone et ses régions de Voronoï. Trois sont associées à des sommets et trois autres à des arrêtes. 25

Figure 1.5 Subdivision de l'espace à l'aide de trois plans et arbre BSP correspondant. 29

Figure 1.6 Décalage des plans d'un arbre BSP. 31

Figure 1.7 Valeur du tampon pochoir. 33

Figure 1.8 Une coquille sphérique. 40

Figure 2.1 Une surface NURBS. 41

Figure 2.2 Le système d'axes défini par les vecteurs propres de la matrice de covariance. 43

Figure 2.3 L est un axe séparateur pour les boîtes A et B. 46

Figure 3.1 Surfaces utilisées lors du premier test. 56

Figure 3.2 Temps d'une requête en fonction du niveau de récursion. . . 58

Figure 3.3 Temps d'une requête en fonction du taux de collision. 60

Figure 3.4 Disposition de surfaces menant au pire cas de l'algorithme de détection de collisions. 61

Figure 3.5 Temps d'une requête en fonction de la complexité des surfaces. 62

Figure 3.6 Les six situations du test de precision. 64

Figure 3.7 Erreur en fonction du niveau de récursion pour des surfaces statiques. 65

Figure 3.8 Erreur en fonction du niveau de récursion pour des surfaces mobiles. 66

Figure 3.9 Séquence d'images du test d'intégration au moteur physique. 68

Figure I.1 Les systèmes d'axes du monde et du corps rigide. 84

Figure I.2 Diagramme des classes du simulateur physique. 92

Figure I.3 Séquences d'images pour différentes configurations de systèmes masses-ressort. 107

LISTE DES NOTATIONS ET DES SYMBOLES

NURBS	Non-Uniform Rational B-Spline
BSP	Binary Space Partition
AABB	Axis-Aligned Bounding Box
OBB	Oriented Bounding Box
DOP	Discrete Orientation Polytope
SIMD	Simple Instruction, Multiple Data

LISTE DES ANNEXES

ANNEXE I ENVIRONNEMENT DE VALIDATION 83
I.1 Simulation Physique de Corps Rigides 83
I.1.1 Position et Orientation 85
I.1.2 Vitesses Linéaire et Angulaire 86
I.1.3 Masse et Tenseur d'Inertie 87
I.1.4 Forces et Moments de Force 89
I.1.5 Dynamique des Corps Rigides 90
I.2 Description du Système de Simulation 91
I.2.1 Classe *CRigidBody* 93
I.2.2 Interface *IForce* 95
I.2.3 Interface *ICollider* 96
I.2.4 Interface *IConstraint* 98
I.2.5 Classe *CPhysicManager* 99
I.3 Exemple d'Application 102

INTRODUCTION

Contexte de Recherche

Au cours de la dernière décennie, on a assisté à une évolution marquée du réalisme dans les jeux vidéo. C'est au début des années 90 que l'on a vu apparaître les premiers jeux tridimensionnels, dans lesquels le joueur peut explorer un monde virtuel tout comme s'il y était vraiment. Ces jeux ont eu beaucoup de succès, car ils atteignaient un niveau de réalisme jamais vu auparavant. Par contre, comme le rendu graphique d'une scène tridimensionnelle en temps réel demande beaucoup de puissance de calcul (du moins pour les ordinateurs de l'époque), des astuces et optimisations poussées ont dû être utilisées. Par la suite, comme la popularité des jeux en trois dimensions ne semblait pas vouloir se dissiper, des compagnies spécialisées en matériel graphique ont commencé à offrir des cartes offrant des fonctions 3D accélérées. Cela a permis aux développeurs de jeux d'utiliser plus de puissance de calcul pour autres choses que le rendu graphique. C'est à ce moment que sont apparus des jeux où l'intelligence des personnages devenait plus convaincante, où les décors étaient plus réels et où les lois de la physique semblaient être respectées.

Les jeux actuels utilisent majoritairement des modèles en fils de fer pour représenter les objets et les scènes. Quelques jeux récents commencent à employer des surfaces paramétriques comme alternative aux modèles en fils de fer. Par contre, à l'heure actuelle, généralement seules les surfaces de Bézier de faible degré sont utilisées en pratique. L'usage des surfaces paramétriques dans le domaine des jeux vidéo offre plusieurs avantages (Watt et Policarpo, 2001):

- modification simple de la forme des objets en temps réel (en déplaçant les points de contrôle);
- utilisation du matériel graphique actuel pour le rendu, en convertissant les surfaces en polygones à l'aide d'algorithmes rapides;
- implémentation simple de niveaux de détails;
- représentation compacte d'objets complexes;
- minimisation de la quantité de données si les algorithmes d'affichage de surfaces paramétriques sont intégrés au matériel.
- représentation exacte de surfaces courbes (et non une approximation polygonale).

Tous ces avantages, ainsi que la tendance des compagnies de matériel graphique à offrir des produits permettant d'accélérer le rendu des surfaces paramétriques, permettent d'imaginer que dans les prochaines années, ces types de surfaces deviendront le standard et remplaceront graduellement les modèles en fils de fer.

Un autre aspect des jeux vidéo permettant d'obtenir un plus grand réalisme, et du même coup une plus grande immersion du joueur, est la simulation des lois de la physique. En effet, rien n'est plus frustrant lors de l'utilisation d'un jeu que d'avoir une solution logique à un problème, mais de ne pas pouvoir la réaliser juste parce que cette solution n'a pas été prévue par le concepteur du jeu (par exemple: ne pas pouvoir couper un arbre pour traverser une rivière). Souvent, ces situations se présentent parce que les lois de la physique ne sont pas implémentées, ou le sont, mais avec des simplifications extrêmes. Dans la plupart des jeux, les solutions aux problèmes sont écrites dans des scripts et le joueur doit effectuer une séquence

d'actions précises pour pouvoir continuer. L'implantation des lois de la physique dans un jeu a plusieurs avantages:

- le joueur peut intuitivement prévoir le résultat de ses actions;
- les événements découlant de l'action des lois de la physique n'ont plus besoin d'être décrits dans des scripts;
- la sensation du joueur d'être dans un monde réel est plus grande.

Comme la simulation des lois de la physique apporte un niveau de réalisme très important, et comme les jeux vidéo tendent à devenir de plus en plus réalistes, on peut facilement imaginer que la prochaine génération de jeux sera marquée par l'arrivée d'un nombre important de produits tirant profit de cette technologie.

Objectifs de Recherche

Le développement d'un nouveau moteur de jeux doit apporter une contribution originale pour que les concepteurs de jeux le trouvent intéressant. Il apparaît clairement que deux caractéristiques importantes devraient être disponibles dans un nouveau moteur: la représentation d'objets utilisant des surfaces courbes et un bon modèle physique permettant de simuler de façon réaliste le comportement des objets.

Il existe plusieurs méthodes permettant d'obtenir des surfaces courbes: fonctions implicites, surfaces paramétriques, surfaces de subdivisions, etc. La technique retenue pour ce projet est l'utilisation des surfaces paramétriques, plus particulièrement les surfaces b-splinaires rationnelles non-uniformes (NURBS). Ces surfaces ont plusieurs caractéristiques mathématiques intéressantes, elles sont très

flexibles pour ce qui est de la modélisation d'objets et des algorithmes performants permettant leur discrétisation en polygones existent (Piegl et Tiller, 1996). De plus, les cartes graphiques récentes offrent des méthodes accélérées par le matériel permettant d'afficher des surfaces paramétriques sans faire travailler le processeur central. La primitive de base de ce projet est donc la surface NURBS. Tous les objets seront décrits à l'aide d'un ensemble de NURBS.

L'implémentation d'un moteur physique requiert la résolution d'un problème important: celui de la détection de collisions. En effet, il est important que les objets se déplacent en respectant les lois de la physique, mais il est aussi très important de détecter si deux objets entrent en collision, et au quel cas, simuler la réponse physique de cet impact. En l'absence de détection de collisions, les objets passeraient tout simplement l'un au travers l'autre, comme s'ils étaient complètement immatériels.

L'état actuel de la recherche en détection de collisions entre surfaces paramétriques est relativement peu avancé comparativement à la recherche effectuée pour les modèles polygonaux. Les solutions déjà existantes sont plutôt axées sur le calcul de courbes d'intersection et sont loin d'être utilisables en temps réel. Comme les surfaces paramétriques deviennent de plus en plus utilisées dans des applications interactives, particulièrement les jeux, la détection rapide de collisions entre ces surfaces devient un problème actuel très intéressant. La principale difficulté du problème de détection de collision n'est pas la détection des points de contacts, mais plutôt la détection *rapide* de ces points. L'objectif de ce travail de recherche consiste donc au développement et à l'implémentation d'un algorithme de détection de collisions en temps réel entre surfaces NURBS. Les caractéristiques souhaitées de cet algorithme sont les suivantes:

- utilisable dans une application devant faire des requête de collisions en temps réel;
- permet de travailler avec des surfaces déformables, c'est à dire pouvant changer de forme pendant la simulation;
- nécessite peu de mémoire;
- précision paramétrable: il doit être possible de choisir entre une grande précision ou une exécution rapide;
- peut retourner les coordonnées de la collision autant dans le repère du monde que dans l'espace paramétrique de chaque surface.

Structure du Document

Ce document est divisé en trois chapitres. Le premier contient une revue de la littérature traitant tout d'abord des différents modes de représentation d'objets en trois dimensions. Ensuite, plusieurs algorithmes de détection de collisions reliés à chacun de ces modes de représentation y sont étudiés. Le second chapitre décrit en détail tous les aspects de l'algorithme original de détection de collisions entre surfaces NURBS. Finalement, le troisième chapitre présente les résultats obtenus et les différents tests ayant été effectués pour valider l'algorithme. Le document se termine par une conclusion décrivant la contribution apportée par ce travail de recherche et discutant des différentes avenues de recherche qu'il serait intéressant d'explorer.

CHAPITRE 1

REPRÉSENTATION GRAPHIQUE D'OBJETS ET ALGORITHMES DE DÉTECTION DE COLLISIONS

Ce chapitre présente une revue détaillée des recherches qui ont été effectuées dans les domaines de la représentation des objets en trois dimensions, de la détection de collisions ainsi que de la simulation physique des corps rigides. La première section discute des trois principaux types de représentation d'objets et permet d'apprécier les avantages et inconvénients de chacune. La deuxième section, la plus importante relativement à ce projet, décrit les principales techniques de détection de collisions. Elles sont regroupées en fonction du type de représentation des objets sur laquelle elles sont basées.

1.1 Modes de Représentation

Un écran d'ordinateur ne peut malheureusement afficher que des images en deux dimensions. Pour donner l'illusion à un usager d'être dans un monde tridimensionnel, il est donc nécessaire de recourir à un système permettant de calculer une image en fonction de ce que verrait une personne dans ce monde virtuel. On peut faire l'analogie avec une caméra vidéo qui transforme une vue d'un monde tridimensionnel en une série d'images bidimensionnelles. Pour effectuer le rendu d'une scène sur l'écran d'un ordinateur, il est nécessaire de spécifier les données géométriques des objets qu'elle contient. Ces données peuvent avoir n'importe quelle forme, en autant qu'elles permettent de décrire des objets.

Cette section présente les trois principaux modes de représentation d'objets en trois dimensions pour fin d'affichage, qui sont:

1. les modèles polygonaux;
2. les fonctions implicites;
3. les surfaces paramétriques.

Le premier est le modèle le plus intuitif et le plus simple permettant de représenter des objets, tandis que les deux autres reposent sur des fondements mathématiques plus élaborés qui offrent plus de puissance.

1.1.1 Représentation Polygonale

De tous les modes de représentation d'objets, c'est sans doute celui utilisant des polygones qui est le plus répandu. En effet, comme les cartes graphiques actuelles ne supportent que l'accélération matérielle de scènes composées de polygones, il est normal qu'une application graphique nécessitant un rendu rapide utilise ce type de représentation. Les technologies de rendu graphique à l'aide de polygones sont même si avancées que la plupart du temps, même si un autre mode de représentation des objets est utilisé à la base, ces objets seront convertis en polygones pour être affichés le plus rapidement possible.

Le fonctionnement d'un système basé sur des polygones est relativement simple. La surface des objets est représentée à l'aide de polygones, ces derniers pouvant être définis comme des surfaces planes finies et limitées par des segments linéaires. Dans la plupart des systèmes, seuls les triangles peuvent être utilisés comme polygones. Cela est principalement dû au fait qu'il est beaucoup plus rapide de dessiner un

triangle à l'écran que tout autre polygone. De plus, n'importe quel polygone peut être décomposé en un certain nombre de triangles. D'autre part, comme un triangle est défini par 3 points, ces derniers sont nécessairement coplanaires. Ce n'est pas le cas lorsque le nombre de points dépasse 3, il faut alors s'assurer que les points sont coplanaires pour définir un polygone valide.

Un objet est donc représenté à l'aide d'une liste de triangles représentant sa surface. Ces triangles sont définis par 3 points, mais comme ces points sont souvent partagés par plusieurs triangles, il est possible d'optimiser l'espace mémoire nécessaire au stockage d'un objet. On peut voir à la figure 1.1 un objet composé de 4 faces triangulaires et de 4 sommets. Pour stocker 4 faces, il est normalement nécessaire

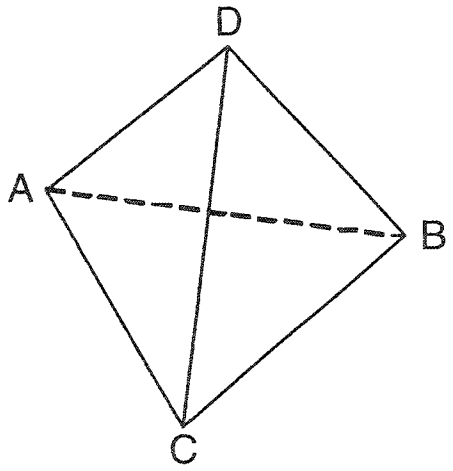

Figure 1.1 Un tétraèdre.

d'avoir 12 points (3 par face). Par contre, si ces points font partie du même objet, il est possible de regrouper ceux qui sont confondus sur un même sommet en un seul point et ainsi construire un tableau des points des sommets. Les faces ne seront plus définie comme un ensemble de 3 points, mais plutôt comme un ensemble de 3 index dans la table des points des sommets. Comme un index prend moins d'espace mémoire que les trois composantes d'un point, moins de mémoire est utilisée. De

plus, des triangles adjacents peuvent être stockés en bandes ou en éventail (voir figure 1.2). On peut voir que pour ce type de stockage des données, le premier

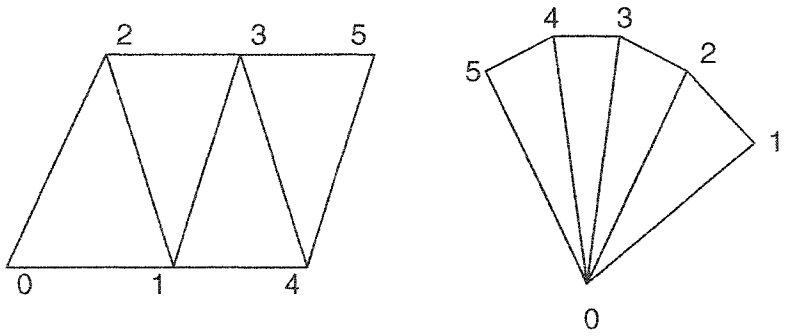

Figure 1.2 Une bande et un éventail de triangles.

triangle nécessite 3 points, alors que tous les suivants n'ajoutent qu'un seul point. Pour le tétraèdre de la figure 1.1, si on utilise une bande de triangles, seulement 6 points sont nécessaires pour le définir (A, B, C, D, A, B) . De plus, si l'on combine cette technique avec l'utilisation d'index, on se retrouve avec seulement 4 points et 6 index. On doit donc retenir que, comme dans toute application informatique, il est primordial de bien structurer les données pour obtenir une application performante.

La simplicité de ce type de représentation entraîne par contre un inconvénient de taille. Comme les polygones composant les objets sont des surfaces planes, on ne peut représenter exactement une surface courbe. Les objets ayant des parties courbes doivent être constitués d'un très grand nombre de polygones pour faire l'approximation de la véritable surface. Cela nécessite donc une grande quantité de mémoire pour stocker ces polygones ce qui détériore les performances d'affichage. Par contre, les cartes graphiques modernes sont optimisées à un point tel qu'elles peuvent afficher plusieurs millions de polygones par seconde. C'est une des principales raisons qui fait que ce mode de représentation est encore le plus populaire. En effet, les deux principales librairies de fonctions graphiques, OpenGL et Direct3D, sont basées sur des modèles polygonaux. Watt et Policarpo, 2001, Eberly, 2000,

Moller et Haines, 2002 ont tous utilisé les modèles polygonaux dans le domaine des jeux vidéo.

1.1.2 Fonctions Implicites

Des objets quelconques peuvent être représentés à l'aide de fonctions implicites. Ces fonctions ont la forme suivante:

$$F(x, y, z) = 0$$

La surface de l'objet correspond aux points de l'espace qui annulent la fonction. L'intérieur de l'objet peut être distingué de l'extérieur à l'aide du signe de la fonction: si l'évaluation de cette fonction donne une valeur négative c'est que le point est à l'intérieur et dans le cas d'une valeur positive, le point est à l'extérieur.

Les principaux avantages de cette représentation sont:

- la possibilité de représenter exactement des surfaces courbes;
- la quantité réduite d'espace mémoire nécessaire pour stocker un objet.

Il est en effet possible de représenter une grande variété de formes à l'aide de fonctions implicites, quoique si cette forme est le moindrement irrégulière, l'équation devient très complexe et augmente rapidement en degré. Pour permettre la construction d'objets complexes sans augmenter inutilement la complexité des équations, Rotgé, 1997 a développé un système dont les primitives sont des quadriques et des quartiques. Les objets sont construits à l'aide d'opérations booléennes (et autres opérations d'ensemble) sur les primitives. De cette manière, il est possible de

construire des formes complexes sans nécessairement augmenter la complexité des équations qui les définissent.

La construction d'objets à l'aide de fonctions implicites a, par contre, plusieurs désavantages:

- il est très difficile de trouver ou d'éditer une fonction implicite qui aura la forme désirée;
- il n'est pas évident d'effectuer des transformations matricielles sur ce genre de fonctions;
- les cartes graphiques actuelles ne supportent pas l'accélération matérielle du rendu de surfaces implicites.

Le problème majeur de ce mode de représentation dans un système devant effectuer de l'affichage en temps réel est qu'il n'existe pas de matériel pouvant afficher ce genre de surface. Deux solutions sont donc envisageables: effectuer la transformation de la fonction implicite en polygones ou développer un logiciel permettant de faire directement le rendu graphique de ces surfaces. En pratique, ces deux solutions ne sont utilisables que dans le cas de scènes simples, car elles demandent toutes les deux beaucoup de temps de calcul. Pour l'instant, il est donc très restrictif de travailler avec ce mode de représentation dans le domaine du jeu ou pour des applications grand-public, où le nombre d'objets distincts à manipuler est très grand.

1.1.3 Surfaces Paramétriques

Les courbes et surfaces paramétriques ont été développées au départ pour le domaine de la conception assistée par ordinateur. Les courbes paramétriques sont construites à l'aide d'une fonction transformant un paramètre en un point de l'espace. Dans le cas des surfaces, deux paramètres sont utilisés pour obtenir un point. Plusieurs fonctions peuvent être utilisées pour définir une surface paramétrique, mais en général ce sont des fonctions polynomiales de faible degré. Une courbe paramétrique polynomiale de degré n est donc représentée par l'équation suivante:

$$\mathbf{C}(u) = \sum_{i=0}^{n} \mathbf{a}_i u^i$$

où les a_i sont les coefficients des puissances de u. De la même manière, une surface paramétrique polynomiale de degré n en u et m en v est représentée par:

$$\mathbf{S}(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} \mathbf{a}_{i,j} u^i v^j$$

Cette forme est la plus simple mathématiquement, mais elle n'est pas pratique, car les coefficients $\mathbf{a}$ n'ont aucune interprétation géométrique. Il est donc très complexe de déterminer ces coefficients pour obtenir une forme souhaitée.

Pour simplifier la manipulation et l'édition de courbes et surfaces paramétriques, Bézier, 1972 a développé des équations dont les coefficients sont des points de l'espace qui déterminent la forme de la courbe ou de la surface. Une surface de

Bézier a une équation de la forme:

$$\mathbf{S}(u,v) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_{i,n}(u)B_{j,m}(v)\mathbf{P}_{i,j}$$

où les $B_{i,n}$ et $B_{i,m}$ sont les polynômes de Bernstein de degré n et m et les $\mathbf{P}_{i,j}$ représentent une grille de $n \times m$ points de contrôles. Les polynômes de Bernstein agissent comme fonctions d'interpolation entre les points de contrôle. On peut donc dire qu'un point sur une surface de Bézier est un mélange pondéré de tous les points de contrôle qui la définissent.

Le principal désavantage des courbes et surfaces de Bézier est qu'elles ne permettent pas de représenter exactement une courbe conique (cercles, ellipses...) ou une surface quadrique (sphères, cylindres, etc.). Ces types de courbes et surfaces jouent un rôle très important pour plusieurs applications liées au domaine de la conception assistée par ordinateur. La solution à ce problème est l'utilisation de surfaces de Bézier rationnelles. Elles sont définies par:

$$\mathbf{S}(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} B_{i,n}(u)B_{j,n}(v)w_{i,j}\mathbf{P}_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} B_{i,n}(u)B_{j,n}(v)w_{i,j}}$$

où $w_{i,j}$ est le poids de chaque point de contrôle. Ces poids permettent de donner plus ou moins d'importance à un point de contrôle, ce qui donne encore plus de liberté lors de l'édition d'une courbe ou d'une surface et permet entre autre d'obtenir des coniques et quadriques.

Les courbes de Bézier rationnelles sont très puissantes, mais elles comportent encore

quelques inconvénients (Piegl et Tiller, 1996):

- un degré élevé est nécessaire pour obtenir une courbe satisfaisant un grand nombre de contraintes, ce qui est inefficace et numériquement instable;
- un degré élevé est aussi nécessaire pour obtenir une forme complexe;
- une courbe complexe, donc de degré élevé, se manipule mal; même si on peut déplacer les points de contrôle individuellement, ils ont une répercussion sur toute la courbe.

C'est pour remédier à ces problèmes que les fonctions de base b-splinaires ont été développées. Elles permettent entre autres, à l'aide d'un vecteur nodal, de définir des courbes et surfaces polynomiales ou rationnelles par morceaux.

Les courbes et surfaces paramétriques obtenues à l'aide des fonctions de base b-splinaires sont appelées b-splines rationnelles non-uniformes et seront identifiées par l'acronyme NURBS (non-uniform rational b-splines) par la suite. L'équation d'une surface NURBS est la suivante:

$$\mathbf{S}(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)N_{j,q}(v)w_{i,j}\mathbf{P}_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)N_{j,q}(v)w_{i,j}}$$

où les $N_{i,p}$ et $N_{i,q}$ sont les fonctions de base b-splinaires définies sur les vecteurs nodaux suivants:

$$U = \{\underbrace{0,\cdots,0}_{p+1}, u_{p+1},\cdots,u_n,\underbrace{1,\cdots,1}_{p+1}\}$$

$$V = \{\underbrace{0, \cdots, 0}_{q+1}, v_{p+1}, \cdots, v_m, \underbrace{1, \cdots, 1}_{q+1}\}$$

Les vecteurs nodaux permettent de définir quelle plage des paramètres u et v est associée à quels points de contrôle et de modifier la paramétrisation de la courbe.

Les surfaces NURBS ont plusieurs caractéristiques, dont voici les plus intéressantes (Piegl et Tiller, 1996):

1. la surface peut passer par les points de contrôle aux coins;
2. une transformation affine sur la surface est appliquée en l'effectuant sur les points de contrôle;
3. l'enveloppe convexe des points de contrôle englobe totalement la surface lorsque les poids sont positifs;
4. modifications locales: si un point $\mathbf{P}_{i,j}$ est déplacé, la surface n'est affectée que dans le rectangle $[u_i, u_{i+p+1}) \times [v_j, v_{j+q+1})$;
5. les surfaces de Bézier rationnelles et non-rationnelles sont des cas spéciaux des surfaces NURBS.

De nombreux algorithmes efficaces ont été développés relativement aux surfaces NURBS. Ils permettent entre autres l'évaluation rapide d'un point, l'évaluation des dérivées, l'interpolation d'une courbe ou surface à partir de points, etc. Piegl et Tiller, 1996 les ont décrites en détail et ont procédé à une étude approfondie des NURBS.

Jusqu'à très récemment, les cartes accélératrices 3D n'avaient aucune fonction permettant d'afficher des objets définis à l'aide de surfaces paramétriques. Il était donc nécessaire de convertir la surface en polygones pour pouvoir l'afficher rapidement.

Cette conversion étant relativement rapide, il était possible d'afficher une scène en temps réel, à condition que celle-ci soit simple et que les surfaces soient de faible degré. C'est pourquoi, à l'heure actuelle, la plupart des jeux utilisant des surfaces paramétriques se limitent à des surfaces de Bézier de degré 2 ou 3. La situation risque de changer dans les prochaines années, grâce à de nouvelles fonctions offertes par les cartes accélératrices. Les programmes de vertex, pouvant être définis par le développeur et chargés dans la carte graphique, permettront d'augmenter de beaucoup la complexité des objets pouvant être affichés, de même que la manière dont ils seront définis.

1.2 Détection de Collisions

Selon le mode de représentation des objets, différentes techniques de détection de collisions existent. D'après Lin, 1993, la détection de collisions se définit comme étant l'identification d'un contact géométrique sur le point de se produire ou qui s'est déjà produit. C'est un problème complexe et pour lequel la recherche est encore très active. En général, il est relativement simple de trouver un algorithme permettant de trouver une collision entre deux objets. La complexité du problème vient plutôt du fait que ces algorithmes sont très coûteux en temps de calcul et doivent être exécutés un très grand nombre de fois (entre chaque paire de primitive des objets). On doit donc développer des techniques permettant d'optimiser les requêtes de détection de collisions.

Cette section discute des recherches qui ont été accomplies dans le but de solutionner efficacement le problème de la détection de collisions. Les prochaines sections décrivent les méthodes de détections de collisions existantes respectivement pour les modèles polygonaux, les fonctions implicites et les surfaces paramétriques.

Une dernières section discute de certaines méthodes récentes qui fonctionnent pour n'importe quel type de représentation géométrique.

1.2.1 Modèles Polygonaux

Les modèles polygonaux étant de loin les plus populaires à l'heure actuelle, il est normal de constater que la plus grande part des travaux de recherche en détection de collisions sont basés sur ce mode de représentation. Malgré tout, aucune solution générale n'a pu être trouvée à ce jour. Certain algorithmes fonctionnent mieux pour certaines applications, des techniques fondamentalement différentes donnent lieu à des résultats semblables, bref la recherche est encore loin d'être terminée sur ce sujet.

Une solution triviale, mais aussi totalement inefficace, consiste à prendre chacune des paires d'objets d'une scène, et pour chacune de ces paires, effectuer un test d'intersection entre tous les polygones qui les composent. On peut facilement imaginer que si les objets sont complexes et nombreux, cette solution est impraticable. Plusieurs techniques permettent de diminuer le temps nécessaire pour déterminer s'il y a collision ou non entre des objets. Cette section décrit les plus couramment utilisées.

1.2.1.1 Hiérarchie de Volumes Englobants

La méthode la plus populaire permettant de diminuer le nombre de tests nécessaire pour déterminer les collisions entre objets est sans doute l'utilisation d'une hiérarchie de volumes englobants. Ces arbres sont conçus de manière à ce que la racine corresponde à un volume qui englobe tout l'objet. Les niveaux suivants contiennent

des volumes plus petits, mais dont l'union englobe la partie de l'objet contenue dans le volume du niveau supérieur. Plusieurs types de volumes englobants ont été utilisés par différents chercheurs, allant de la simple sphère jusqu'à l'enveloppe convexe de l'objet. En fait, un bon volume englobant possède deux caractéristiques importantes:

1. il y a peu d'espace vide entre celui-ci et l'objet qu'il englobe;
2. il est très rapide d'effectuer un test d'intersection entre deux volumes englobants du même type.

Comme ces deux caractéristiques sont quelque peu contradictoires, un compromis doit être trouvé afin d'obtenir des performances acceptables pour une application donnée. Comme les volumes englobants ne sont qu'une approximation de l'objet auquel ils correspondent, ils ne servent que pour un premier test grossier de détection de collisions. Si le test est positif, on doit nécessairement effectuer par la suite un test précis d'intersection entre les objets contenus dans les feuilles de la hiérarchie de volumes englobants qui ont été marquées comme étant potentiellement en collision.

Le type de volume le plus simple permettant d'entourer un objet est sans contredit la sphère. Le test d'intersection entre deux sphères est extrêmement rapide, il suffit de trouver la distance entre les centres des sphères et de comparer avec la somme des deux rayons. Si la distance est plus petite, c'est que les sphères entrent en collision. La deuxième caractéristique est donc satisfaite: le test est rapide. Par contre, les objets d'une scène étant généralement loin d'être sphériques, la plus grande part du volume de la sphère ne correspond pas à l'objet qu'elle contient. La première caractéristique pourrait donc être mieux satisfaite. Les arbres doivent être construits de façon à minimiser le nombre de tests nécessaires pour déterminer les

feuilles qui entrent en collision. Pour ce faire, des algorithmes permettant de générer des arbres de sphères englobantes très performants ont été développés. Certains de ces algorithmes créent des arbres dont la structure ressemble à un octree (Liu *et al.*, 1988, Tzafesta et Coiffet, 1996, Hubbard, 1993), d'autres plus performants placent les sphères à l'aide d'une surface d'axe médian (Hubbard, 1996, Bradshaw et O'Sullivan, 2002).

Un autre type de volume englobant simple pouvant être utilisé est la boîte. Deux techniques existent qui permettent d'utiliser les boîtes comme volume englobant. La première utilise une boîte dont les faces sont alignées sur les axes de coordonnées du monde virtuel et la deuxième permet l'utilisation d'une boîte orientée de manière quelconque. Ces deux méthodes ont chacune des avantages et des inconvénients.

Les boîtes englobantes alignées sur les axes (axis-aligned bounding boxes, AABB) sont probablement à la base du plus grand nombre d'algorithmes de détection de collision. Cela est dû au fait qu'elles ont certains avantages qui les rendent simples à utiliser. Premièrement, elles sont très faciles à construire: pour un objet donné, on trouve ses points extrêmes sur chacun des trois axes de coordonnées. De plus, le test d'intersection de deux AABB est simple et rapide: si les intervalles définissant deux boîtes se chevauchent sur les trois axes, c'est qu'elles entrent en collision. Finalement, elles ne prennent pas beaucoup d'espace mémoire, deux points suffisent pour définir une boîte.

Les AABB ont par contre deux désavantages importants. Le premier est que, les boîtes étant alignées sur les axes, elles ne représentent pas nécessairement la boîte la plus petite englobant l'objet. Le deuxième désavantage est que les boîtes doivent être reconstruites à chaque fois que l'objet change d'orientation. La construction d'un arbre d'AABB est un processus relativement exigeant en temps de calcul. van den Bergen, 1997a propose donc une optimisation permettant de mettre à jour

un arbre en recalculant de manière exacte seulement les feuilles, les boîtes parents étant recalculées en fonction des dimensions de leurs enfants.

D'autres optimisations possibles sont proposées par Cohen *et al.*, 1995 pour des objets dont les feuilles sont convexes. Premièrement, une structure de données contenant les points voisins de tous les points de l'objet est construite au départ. Comme les feuilles sont convexes, on peut trouver rapidement les nouveaux points extrêmes en vérifiant les points extrêmes actuels et leurs voisins. La deuxième amélioration proposée permet d'accélérer les tests d'intersection entre les boîtes dans une scène en mouvement mais cohérente dans le temps. La technique consiste à conserver trois listes en mémoire, une pour chaque axe de coordonnées. Ces listes contiennent les intervalles correspondant à la projection des boîtes sur l'axe associé à cette liste. La seconde étape consiste à trier la liste, ce qui permet par la suite de déterminer facilement quels intervalles se chevauchent. On sait qu'il y a collision s'il y a chevauchement sur les trois axes. Le tri des listes est une opération relativement longue la première fois, mais si on prend en considération que la scène ne change pas énormément entre chaque pas de temps, on peut considérer que la liste est presque déjà triée. Le tri devient donc une opération pouvant s'effectuer très rapidement. La simplicité et l'efficacité des arbres d'AABB rend son utilisation dans le domaine de la détection de collisions très populaire et plusieurs algorithmes les utilisent (Smith *et al.*, 1995, Ponamgi *et al.*, 1997, van den Bergen, 2001, Hudson *et al.*, 1997).

La seconde technique utilisant des boîtes englobantes ne les restreint pas à être alignées sur les axes de coordonnées. Elles peuvent avoir une orientation quelconque, une matrice de transformation leur est donc associée. Gottschalk *et al.*, 1996 expliquent en détail les hiérarchies de OBB (Oriented bounding boxes). Ces boîtes ont deux grands avantages sur les AABB: elles sont plus ajustées à l'objet

qu'elles englobent et elles n'ont pas à être recalculées à chaque fois que l'objet change d'orientation, une simple multiplication de matrices suffit à trouver la nouvelle boîte.

Pour construire une OBB, on doit calculer la matrice de covariance des points composant l'objet. Ensuite, on calcule les vecteurs propres de cette matrice qui serviront à déterminer l'orientation des faces de la boîte. Deux de ces vecteurs sont alignés sur les axes de variance maximale et minimale de l'objet, ils permettent donc d'obtenir une boîte englobant généralement l'objet de très près.

À première vue, on peut croire qu'il est relativement coûteux d'effectuer un test d'intersection entres deux OBB: il y a théoriquement 144 tests de collision entre faces et arrêtes. Par contre, en utilisant le théorème d'axe séparateur (Gottschalk, 1996), il est possible d'obtenir des performances impressionnantes. La projection de deux objets convexes disjoints sur un axe séparateur donne deux intervalles qui ne se chevauchent pas. Le théorème stipule que deux objets convexes sont disjoints s'il existe un axe séparateur orthogonal à une des faces des objets ou orthogonal à une combinaison de deux arrêtes prises sur chaque objet. Dans le cas d'une boîte, il y a 3 orientations de face et 3 orientations d'arrête par objet. En tout, il y a 6 axes possibles provenant des faces et 9 provenant des combinaisons des arrêtes, ce qui donne 15 axes séparateurs au maximum à tester pour déterminer si deux boîtes sont en collision ou non. Les hiérarchies de OBB se comparent généralement avantageusement aux hiérarchies de AABB, principalement parce que les OBB sont mieux ajustées au volume qu'elles englobent. Par contre, elles ont certains désavantages. Le fait qu'une hiérarchie de OBB est assez coûteuse à construire les rendent moins adaptées aux objets déformables, les AABB peuvent devenir plus avantageuses dans ce cas (van den Bergen, 1997a). De plus, une OBB demande plus d'espace mémoire qu'une AABB, car on doit aussi stocker la matrice

de transformation permettant d'orienter la boîte.

Il existe un dernier type de volume englobant relativement populaire. Il s'agit des k-DOP (Discrete Orientation Polytopes), qui sont une généralisation des boîtes alignées sur les axes. Un k-DOP est un polyèdre convexe dont les k faces ont une normale orientée selon les directions positive ou négative de k/2 axes prédéfinis. Une AABB est donc un 6-DOP dont les orientations sont les 6 directions définies par les 3 axes du système de coordonnées. La figure 1.3 illustre des k-DOP en 2 dimensions. On peut voir que pour k=4 (en 2D), le k-DOP correspond à la boîte englobante alignée sur les axes. Pour effectuer un test d'intersection entre deux

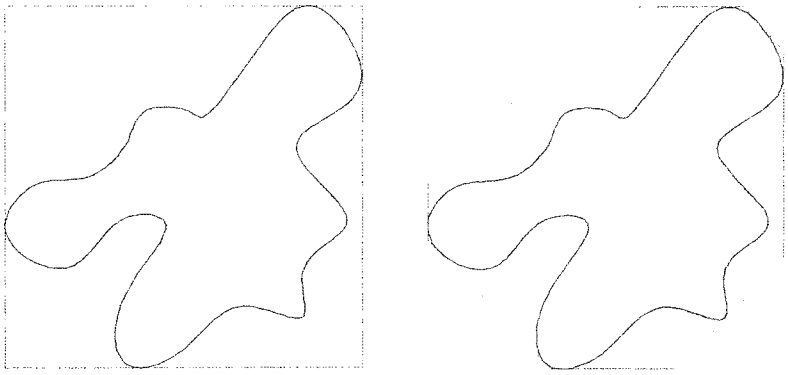

Figure 1.3 Un objet 2D et deux k-DOP correspondant.

k-DOP, il suffit de les projeter sur les k/2 axes. S'il y a chevauchement simultané des intervalles sur les k/2 axes, il y a possiblement une collision. Plus la valeur de k est grande, plus le k-DOP se rapproche de l'objet qu'il englobe, mais plus le test d'intersection devient coûteux. En pratique, des valeurs de k de 14, 18 et 26 sont généralement utilisées, car elles correspondent à des orientations relativement naturelles. Comme les k-DOP ont des caractéristiques semblables aux AABB, les mêmes optimisations peuvent être appliquées, à savoir les listes triées d'intervalles et les listes de points voisins. Klosowski *et al.*, 1998 et Zachmann, 1998 expliquent les détails de l'utilisation des k-DOP dans le domaine de la détection de collisions.

Les principaux types de volumes englobants utilisés en pratique sont les sphères, les boîtes et les k-DOP, mais rien n'empêche d'utiliser d'autre types de volumes. La seule chose que l'on doit garder à l'esprit est que les hiérarchies de volumes englobants servent à accélérer les requêtes de détection de collisions. Il faut donc s'assurer d'avoir un volume englobant simple et un arbre de volumes balancé et bien construit, ce qui n'est pas en soi un problème simple dans le cas général.

1.2.1.2 Distance Entre les Objets

Étant donné qu'on a une collision si la distance entre deux objets devient nulle ou négative, il est possible de se servir de cette distance comme base de détection de collisions. Par contre, le problème de calcul de distance est tout aussi complexe que celui de la détection de collisions. Il faut donc utiliser une méthode rapide et optimisée. Un des moyens permettant d'accélérer le calcul des distances est d'utiliser seulement des objets convexes et de représenter les objets concaves à l'aide d'un ensemble d'objets convexes. Il est en effet beaucoup plus simple de calculer la distance entre deux objets convexes.

L'algorithme le plus connus effectuant cette tâche est celui de Gilbert *et al.*, 1988, appelé algorithme GJK. Il permet de trouver rapidement les points les plus rapprochés de deux objets convexes. Cet algorithme a été utilisé avec succès par van den Bergen, 1997b, en apportant trois améliorations, qui sont: l'utilisation d'un cache pour les données de l'algorithme GJK, la sortie prématurée de l'algorithme si un axe séparateur est trouvé et l'utilisation d'un cache contenant le dernier axe séparateur trouvé. Sato *et al.*, 1996 ont aussi utilisé l'algorithme GJK dans leur méthode de détection de collisions pour des objets concaves. La première phase de leur algorithme consiste à conserver la distance entre les enveloppes convexes des objets à l'aide de GJK. Quand cette distance diminue jusqu'à un certain seuil, la

distance est calculée à l'aide d'un algorithme plus lent, mais supportant les objets concaves (Quinlan, 1994). Joukhadar *et al.*, 1999 proposent une autre méthode utilisant l'algorithme GJK pour effectuer la détection de collision entre objets concaves et pouvant se déformer. Leur méthode consiste en un premier lieu à déterminer un ensemble de faces potentiellement en collision et en second lieu à utiliser GJK pour déterminer si ces faces entrent réellement en collision.

Un autre algorithme permettant de déterminer les points les plus rapprochés de deux objets convexes est celui de Lin et Canny, 1991. Il est relativement simple comparativement à l'algorithme GJK et il offre des performances semblables. Son fonctionnement prévoit une phase de pré-calcul permettant d'obtenir une structure contenant des plans qui définissent les régions de Voronoï associées à chaque élément (sommets, arrêtes et faces) de l'objet. Une région de Voronoï correspond à l'ensemble de tous les points de l'espace les plus rapprochés d'un élément que de tous les autres. La figure 1.4 montre un exemple des régions de Voronoï pour un polygone en 2D. Chaque plan définissant une de ces régions est aussi marqué de l'élément correspondant à la région adjacente à ce plan, permettant ainsi de déterminer rapidement quel est le voisin. La première étape consiste à choisir un élément sur chaque objet, ils doivent idéalement être le plus près possible de l'autre objet. On détermine ensuite si le point le plus près de l'autre objet est situé sur cet élément à l'aide des régions de Voronoï. Si oui, il sera compris dans cette région et on arrête, sinon on continue en effectuant le même test, mais pour l'élément correspondant au plan qui a permis de déterminer que le test a échoué. Quand on a finalement trouvé les deux éléments les plus rapprochés, on trouve algébriquement les points les plus près. Si la scène est cohérente dans le temps et l'espace, alors les éléments les plus rapprochés ne changent pas beaucoup d'un pas de temps à un autre. On prend donc les éléments qui étaient les plus près au pas de temps précédent comme valeur de départ, c'est la clé permettant à l'algorithme

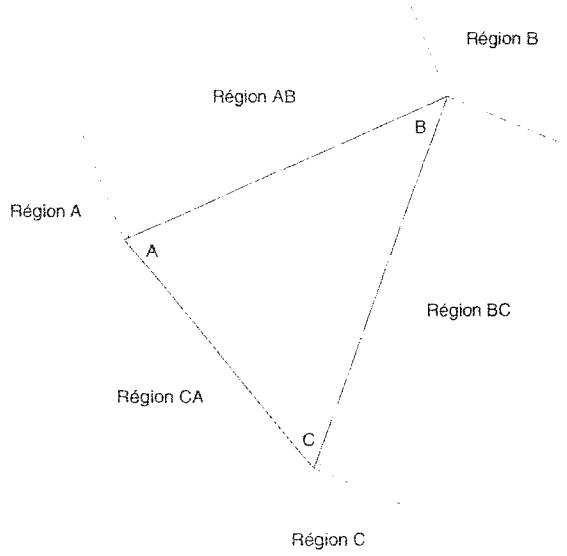

Figure 1.4 Un polygone et ses régions de Voronoï. Trois sont associées à des sommets et trois autres à des arrêtes.

Lin-Canny d'être très performant et de s'exécuter en temps presque linéaire. Il a été utilisé comme première phase de détection de collisions entre polyèdres convexes dans le package I-Collide (Cohen *et al.*, 1995). Mirtich, 1998 a proposé certaines améliorations à l'algorithme Lin-Canny, le rendant plus robuste, plus simple et surtout capable de gérer l'interpénétration de deux objets.

Une autre technique de détection de collisions utilisant la distance entres des objets est décrite par Ehmann et Lin, 2000. Ils utilisent l'algorithme de Lin-Canny pour conserver la distance entre les objets, mais en utilisant une hiérarchie de niveaux de détails de l'objet. La distance est calculée au départ sur le niveau de détails le plus bas et, si cette distance devient nulle, on descend d'un niveau dans la hiérarchie (donc vers un niveau de détails plus élevé). La distance est à nouveau calculée et le processus se répète jusqu'à ce qu'on détermine qu'il n'y a pas de collision ou qu'on ait atteint le niveau de détails le plus élevé.

Les deux algorithmes les plus utilisés pour déterminer la distance entre des objets

sont donc GJK et Lin-Canny. Depuis leur développement, plusieurs modifications ont été apportées par différentes personnes, les rendant ainsi plus performants et plus robustes.

1.2.1.3 Décomposition de l'Espace

La principale cause ralentissant un algorithme naïf de détection de collisions est le fait qu'un trop grand nombre de tests de collision entre des objets très distants sont effectués. L'utilisation d'un algorithme de partition de l'espace permet de réduire ces tests inutiles en permettant de retrouver rapidement les objets rapprochés. Il existe plusieurs techniques permettant de décomposer l'espace en sous-régions. Les trois technique qui sont les plus utilisées sont basées respectivement sur les voxels (de "volume element", par analogie aux pixels), les octrees et les arbres binaires de partition de l'espace (binary space partition ou BSP). Ils sont tous les trois utilisables pour accélérer la détection de collisions.

Le type de décomposition de l'espace le plus simple est la division en voxels et Turk, 1990 fut le premier à les utiliser dans le domaine de la détection de collisions. L'idée consiste à diviser l'espace en une grille 3D régulière, dont chaque élément est appelé voxel. Un point de référence représentant l'origine du système de voxels dans l'espace est spécifié, de même que les dimensions d'un voxel (ils doivent tous avoir la même taille). De cette manière, on peut identifier n'importe quel voxel par trois indices dans la grille: un pour chaque dimension. À première vue, il peut sembler nécessiter une très grande quantité de mémoire pour stocker tous les voxels, mais en pratique, seul les voxels non-vides sont conservés en mémoire à l'aide d'une fonction de hachage sur ses indices.

Le principe de base de la détection de collisions avec des voxels est le suivant.

Premièrement, tous les voxels qui sont en intersection avec un sommet, une arrête ou une face sont marqués avec une référence vers cet élément et une autre vers l'objet auquel appartient cet élément. Ensuite, tous les voxels contenant une référence vers plus d'un objet sont ajoutés dans une liste. Finalement, un test de collision exact est effectué entre tous les éléments d'objets différents référencés par chaque voxel de la liste. De cette manière, on arrive à réduire de beaucoup le nombre de tests d'intersection exacts devant être effectués. Il est important de noter que la taille des voxels doit être bien sélectionnée. En effet, si les voxels sont trop petits, le temps nécessaire pour décomposer les objets en voxels sera trop long et si les voxels sont trop grand, un nombre élevé d'éléments d'objet seront référencés par chaque voxel, allongeant ainsi le temps nécessaire au test exact. Pour avoir un système efficace, la taille des voxels doit donc être à peu près la même que la taille moyenne des polygones composant la scène.

Les principaux désavantages de cette technique sont qu'elle ne fonctionne pas bien avec des objets de tailles très différentes et que la mise à jour des voxels est coûteuse en temps de calcul, il faut donc avoir peu d'objets mobiles. Elle a par contre été utilisée avec succès par Zhang et Yuen, 2000 dans une simulation de tissus, car la structure régulière du modèle du tissus rendait la décomposition en voxel rapide et efficace. De plus, Lawlor et Kalé, 1988 ont mis à profit le fait que la détection de collisions avec des voxels peut facilement être parallélisée, car les tests d'intersection exacts sont indépendants pour chaque voxel.

La technique de partition de l'espace la plus répandue est probablement l'utilisation d'octrees. Les octrees sont une extension en trois dimensions des quadtrees, ces derniers étant décrits en détail par Samet, 1984. Un octree est un arbre de degré 8 dont chaque noeud comporte nécessairement 8 enfants (un pour chaque octant), ou aucun si c'est une feuille.

La première technique permettant d'effectuer la détection de collision avec des octrees consiste à les utiliser comme approximation des objets. La racine de l'arbre correspond au plus petit cube englobant l'objet. Elle est subdivisée en 8 octants, chacun étant marqué noir si son volume est complètement à l'intérieur de l'objet, blanc s'il est à l'extérieur ou gris dans les autres cas. Le même processus est effectué récursivement pour tous les enfants marqués gris, jusqu'à ce qu'il n'y en ait plus ou qu'un certain niveau de précision soit atteint. Le test de collision entre deux objets consiste à trouver l'intersection entre leur octree: si elle n'est pas vide, c'est qu'il y a collision. Cette méthode a quelques désavantages importants. Premièrement, les octrees des objets se déplaçant doivent être recalculés, ce qui demande du temps de calcul. De plus, si l'octree ne représente pas exactement l'objet auquel il correspond, la précision du point de collision se limite au niveau de raffinement de l'octree. Pour contrer ces inconvénients, Garcia et Corre, 1989 ont proposé des algorithmes utilisant des techniques permettant de mettre à jour un octree en fonction d'une translation et d'une rotation.

Il existe une autre méthode permettant d'utiliser les octrees pour faire de la détection de collision. Cette fois-ci, au lieu d'utiliser l'arbre pour représenter les objets, on l'utilise pour décomposer l'espace de travail. Le principe est le suivant. On divise la racine en ses 8 octants. Tous les octants contenant plus de N objets sont eux même divisés en 8 et le processus se répète récursivement avec les enfants. Le résultat est un arbre ayant des feuilles contenant au maximum N objets, N valant habituellement 2. L'étape suivante consiste à effectuer un test précis de détection de collisions entre les objets contenus dans les feuilles d'au moins deux objets. À ce stade, rien n'empêche d'utiliser une autre méthode d'accélération, comme par exemple une hiérarchie de volumes englobants. Swan II, 1993 décrit comment utiliser les octrees de cette manière, il explique aussi comment gérer efficacement la mise à jour de l'octree pour les objets qui se déplacent. Le principal inconvénient de cette

méthode est le fait que si les objets sont complexes et très rapprochés, le temps nécessaire à la mise à jour de l'arbre sera très long. Il peut donc être nécessaire d'insérer dans l'octree un volume englobant au lieu de l'objet lui-même.

La méthode de décomposition de l'espace la plus utilisée dans le domaine des jeux vidéo est l'utilisation d'un arbre de partition binaire. Les arbres BSP sont une généralisation des octrees: il n'y a pas de contrainte sur la position des plans de séparation. Ils ont été décris en premier par Fuchs *et al.*, 1980 qui les ont développés dans le but de résoudre le problème des faces cachées. Le principe est le suivant. Premièrement, on choisit un plan quelconque qui sera utilisé comme racine de l'arbre. Ensuite, on détermine les polygones de la scène qui sont du côté positif du plan et on subdivise récursivement cette partie de l'espace à l'aide d'un autre plan qui deviendra le premier fils du plan parent. On fait de même avec le côté négatif, ce qui nous donne un deuxième plan fils. La subdivision récursive se poursuit jusqu'à ce que chaque région définie par les feuilles ne contiennent qu'un très petit nombre de polygones (généralement un). La figure 1.5 montre un arbre BSP contenant trois plans (P1 à P3) et définissant quatre régions (R1 à R4). En pratique, les

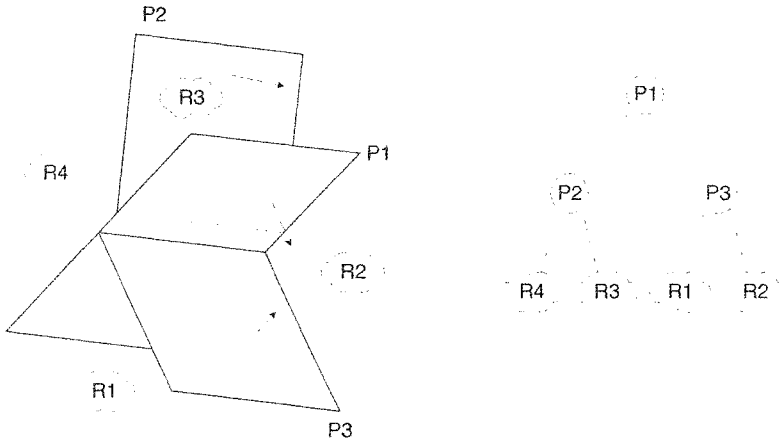

Figure 1.5 Subdivision de l'espace à l'aide de trois plans et arbre BSP correspondant.

plans utilisés pour construire l'arbre sont ceux que définissent les polygones de la scène. Il sont choisi de manière à ce qu'ils divisent le plus également possible les polygones, permettant ainsi d'obtenir un arbre balancé.

La popularité des arbres BSP dans le domaine des jeux vidéo s'explique par le fait qu'ils permettent de résoudre plusieurs problèmes classiques. Ils permettent de trier les polygones par ordre de distance à l'observateur, d'obtenir la liste des polygones possiblement dans le volume de visualisation et aussi d'accélérer la détection de collisions, ce qui nous intéresse particulièrement. Plusieurs techniques peuvent être utilisées pour accélérer la détection de collisions à l'aide d'un arbre BSP. Les deux plus utilisées seront décrites dans les paragraphes suivants.

La première méthode prévoit que tous les objets de la scène aient chacun leur arbre BSP. À chaque pas de temps, l'intersection de ces arbres est faite. Si elle n'est pas vide, on sait qu'il y a possiblement une collision dans la région de l'intersection (dans le cas d'objets concaves) ou assurément une collision (si les objets sont convexes). Naylor *et al.*, 1990 ont montré comment exécuter des opérations d'ensemble à l'aide de BSP, entre autres comment effectuer une intersection.

La seconde technique de détection de collision, la plus utilisée, ne fait appel qu'à un seul arbre BSP, correspondant aux objets statiques de la scène (ceux qui composent le décor). Les objets mobiles, quant à eux, sont réduits à une sphère englobante. Pendant la construction de l'arbre BSP, qui est effectuée une seule fois au départ, les plans sont décalés vers l'extérieur d'une distance correspondant au rayon de la sphère englobante. Cette opération est illustrée sur la figure 1.6. De cette manière, il suffit d'insérer le centre de la sphère englobante d'un objet dans l'arbre pour déterminer s'il y a une collision. Si c'est le cas, il est alors possible d'effectuer un test de collision plus précis si nécessaire. On peut évidement voir que cette solution a des inconvénients. Premièrement, tous les objets mobiles doivent avoir une sphère

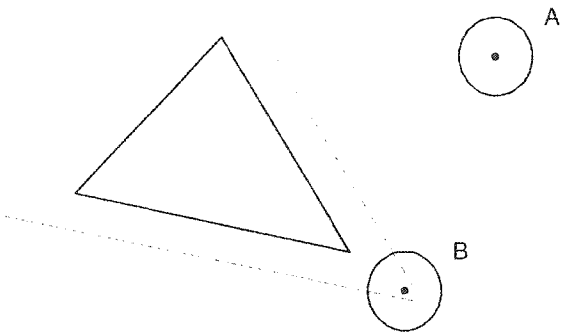

Figure 1.6 Décalage des plans d'un arbre BSP.

englobante de la même taille, ou alors différents arbres BSP doivent être construits pour différentes tailles de sphères englobantes. De plus, les coins sont surévalués et causent la détection de collision inexistantes (voir objet B sur la figure 1.6). Ces problèmes sont difficilement acceptables, Melax, 2001 a donc développé un algorithme permettant de décaler les plans dynamiquement en fonction de la taille d'un objet à mesure qu'un arbre BSP est visité. Il a aussi proposé l'ajout de plans dans les coins, permettant de limiter leur surévaluation.

1.2.1.4 Détection dans le Domaine Image

Les cartes graphiques actuelles permettent d'afficher des millions de polygones par seconde, en libérant ainsi le processeur de cette tâche. Il est donc tentant de développer une technique permettant de mettre à profit cette puissance de calcul dans le but d'effectuer de la détection de collisions. C'est ce qu'ont fait Lombardo *et al.*, 1999 en utilisant la librairie OpenGL. L'application visée par leur algorithme est la détection de contact entre un outil chirurgical et des organes. Ils mettent donc à profit le fait qu'un seul objet ayant une forme simple est susceptible d'entrer en collision avec les autres. Le fonctionnement de leur algorithme est le suivant:

1. ajustement du volume de visualisation pour qu'il corresponde au volume de l'outil;

2. rendu des objets en mode de sélection OpenGL;

3. analyse du tampon de sélection pour déterminer si un objet a été touché par l'outil.

Le mode de sélection d'OpenGL permet de donner un nom symbolique aux objets de la scène et d'obtenir la liste des noms des objets qui auraient été affichés si la scène avait été rendue normalement. Cet algorithme permet donc d'obtenir la liste des objets qui touchent à l'outil, car le volume de visualisation correspond au volume de l'outil. Comme la mise à jour de la scène à afficher se fait à une fréquence finie, il est possible qu'une collision soit manquée si l'outil bouge assez vite pour qu'il passe complètement au travers d'un objet entre deux pas de temps. Pour remédier à ce problème, le volume de visualisation doit correspondre au volume balayé par l'outil entre ces deux pas de temps. De cette manière, aucune collision ne peut être manquée. Si on compare cette méthode de détection de collision avec les précédentes, les performances sont très impressionnantes. Elle a par contre deux désavantages qui la rendent inutilisable dans la plupart des applications: un seul objet peut entrer en collision avec les autres et cet objet doit avoir un volume très simple, soit un prisme rectangulaire ou cylindrique.

Un autre algorithme, pouvant être utilisé avec des objets plus généraux, a été proposé par Baciu *et al.*, 1999. Il fonctionne en prenant les objets deux à deux pour déterminer s'ils se touchent. Tous les objets doivent avoir une boîte englobante alignée sur les axes. La première étape de l'algorithme consiste à ajuster le volume de visualisation pour qu'il corresponde au volume de l'intersection des deux boîtes englobantes. Évidement, si l'intersection est vide, le test s'arrête là et on peut

conclure qu'il n'y a pas de collision. Sinon, on rend le premier objet en ajustant le tampon Z de manière à ce qu'il contienne les valeurs de l'objet les plus éloignées de la direction de visualisation (l'inverse de l'utilisation classique du tampon Z). Ensuite, l'écriture dans le tampon Z est désactivée et le second objet est rendu dans le tampon pochoir, lequel est incrémenté lorsque le test suivant est vérifié: $Z_2 <= Z_1$ (à l'aide du tampon Z). Pour des objets convexes, les valeurs possibles du tampon pochoir sont $\{0, 1, 2\}$. La figure 1.7 montre les cas où ces valeurs apparaissent. Dans le cas où la valeur est 0, il n'y a pas de collision à ce point.

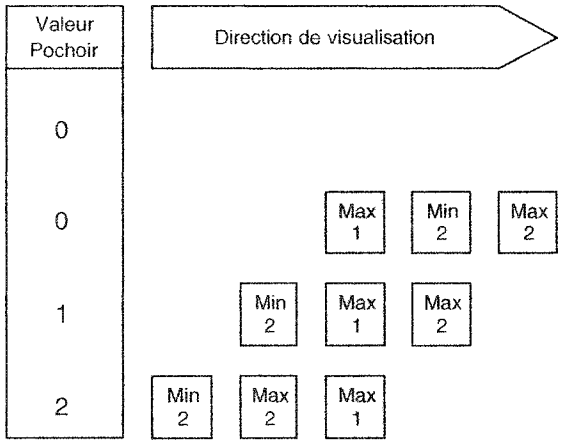

Figure 1.7 Valeur du tampon pochoir.

Lorsque la valeur est 1, il y a assurément une collision à ce point. Par contre, lorsque la valeur est 2, on ne peut déterminer s'il y a collision ou non. On doit donc refaire les opérations, mais en inversant les rôles des deux objets. De cette manière, on peut déterminer s'il y a effectivement une collision ou non. Les performances obtenues par cet algorithme sont relativement bonnes, mais le principal désavantage est qu'il ne fonctionne uniquement pour des objets convexes.

Un autre algorithme utilisant le matériel graphique a été récemment proposé par Dinerstein et Egbert, 2002. Ce dernier permet que les objets soient aussi bien convexes que concaves. L'idée derrière cet algorithme est l'utilisation de la couleur

pour encoder la troisième dimension. Une image contient à la base deux dimensions, mais la valeur de la couleur peut être utilisée astucieusement pour représenter une profondeur. L'algorithme se compose de quatre étapes:

1. test d'intersection entre les boîtes englobantes alignées sur les axes des deux objets;
2. ajustement du volume de visualisation au volume correspondant à l'intersection des deux boîtes englobantes;
3. rendu des objets en utilisant une texture 1D permettant de discrétiser la profondeur;
4. analyse de l'image produite pour déterminer si deux objets partagent la même tranche de profondeur à un même pixel.

Les deux première étapes sont identiques à l'algorithme présenté au paragraphe précédent. La troisième étape effectue le rendu des objets. Le premier est rendu en utilisant une texture 1D de 16 éléments. Cette texture se compose des 16 premières puissances de 2, ses éléments ont donc seulement un bit à 1 et tiennent dans les 16 bits les moins significatifs de la couleur. Les coordonnées de texture sont assignées en fonction de la distance du point à l'observateur (cette fonction est disponible dans OpenGL). Le second objet est rendu en assignant les coordonnées de texture de la même manière, mais les 16 éléments de la texture sont cette fois-ci les 16^e à 31^e puissances de deux. De plus, le rendu s'effectue en utilisant une opération OU logique sur les couleurs (cette fonction est disponible dans une extension OpenGL). L'image résultante est donc un ensemble de pixels dont les valeurs des couleurs indiquent la profondeur respective des deux objets. La profondeur du premier est indiquée par les 16 bits les moins significatifs tandis que celle du second, par les

16 bits les plus significatifs. La quatrième étape consiste donc à vérifier si les deux objets partagent une même tranche de profondeur à un même pixel, ce qui veut dire qu'il y a possibilité de collision à cet endroit (les objets étant discrétisés, il est impossible de savoir avec certitude s'il y a une collision). Pour améliorer la précision, il est possible de procéder à une nouvelle itération du même algorithme, mais en prenant comme volume de visualisation le volume du pixel où la collision a été détectée. La force de cet algorithme est qu'il fonctionne directement avec n'importe quel type d'objet pouvant être affiché avec OpenGL. Il a par contre deux inconvénients: la précision est limitée et il est impossible de déterminer quelles faces des objets sont entrées en collision.

Ces nouvelles méthodes de détection de collisions utilisant le matériel graphique sont très prometteuses, car la technologie des cartes graphiques s'améliore à un rythme très rapide. Le développement d'extensions aux fonctions offertes par le matériel graphique permettra sans doute de développer des algorithmes de détection de collisions rapides et robustes.

1.2.2 Fonctions Implicites

Les objets représentés à l'aide d'une fonction implicite ne contiennent pas directement d'information sur la position des points constituant leur surface. Des méthodes de détection de collision différentes de celles décrites pour les modèles polygonaux doivent être utilisées. Certaines idées vues précédemment peuvent quand même être utilisées, comme par exemple les volumes englobants, mais seulement pour une première phase de tri des objets. Comme les objets sont définis par une fonction mathématique, il est tentant de solutionner l'équation de l'intersection. Malheureusement, cette technique est impraticable, car ces équations sont complexes à solutionner et mènent à des fonctions de degré très élevé. Il est donc

nécessaire de recourir à des méthodes d'approximation si l'on vise à obtenir une application en temps réel. Deux méthodes sont présentées dans cette section.

1.2.2.1 Échantillonnage

Pentland et Williams, 1989 ont décrit brièvement une méthode permettant d'effectuer de la détection de collisions entre objets définis par des surfaces implicites. L'idée est de calculer une série de points d'échantillonnage situés sur les surfaces. Ensuite, les points d'un objet sont évalués dans la fonction définissant l'autre objet. Si un de ces points donne une évaluation inférieur ou égale à zéro de la fonction, c'est qu'il y a collision aux alentours de ce point. Si aucun point n'a été trouvé comme étant à l'intérieur de l'autre objet, on recommence l'opération, mais en inversant les rôles de deux objets. La principale difficulté de cette manière de procéder est le fait que l'échantillonnage de la surface doit être très uniforme et assez fin pour ne pas rater de collisions. L'utilisation de l'algorithme des "marching cubes" (Lorensen et Cline, 1987) permet de transformer une surface implicite en polygones et donc de réaliser efficacement la tâche de l'échantillonnage. Cani-Gascuel et Desbrun, 1997 ont utilisé des fonctions implicites pour simuler des objets déformables. Leurs objets étant définis à l'aide d'un squelette générant des fonctions implicites, ils utilisent une technique d'échantillonnage basée sur ce squelette. Chaque partie du squelette émet des points d'échantillonnage dans des directions fixes du système de repère. Cette technique de détection de collisions est relativement rapide, mais demeure une approximation.

1.2.2.2 Solution Numérique

Un algorithme permettant d'obtenir une précision supérieure a été proposé par Baraff, 1990. Cet algorithme fonctionne en déterminant la distance entre deux surfaces implicites. Comme la plupart des algorithme de calcul de distance, il ne fonctionne qu'avec des objet convexes. Pour trouver la distance, on commence par déterminer les points les plus rapprochés des deux surfaces. Pour ce faire, on solutionne le système d'équations non-linéaires suivant:

$$\begin{cases} \nabla F(p_a, t) + \lambda_2 \nabla G(p_b, t) = \vec{0} \\ F(p_a, t) = 0 \\ G(p_b, t) = 0 \\ (p_b - p_a) + \lambda_1 \nabla G(p_b, t) = \vec{0} \end{cases}$$

où λ_1 et λ_2 sont des scalaires quelconques et p_a et p_b sont les points à trouver. Comme on peut le voir, les surfaces sont aussi fonction du temps, mais ce dernier est fixé à la valeur du temps actuel, ce n'est donc pas une inconnue. La première équation contraint les normales à être colinéaires, les deux suivantes garantissent que les points soient sur leur surface respective et la dernière force le déplacement de p_a vers p_b à être parallèle aux normales des surfaces. La solution donne donc les points les plus rapprochés de deux surfaces, si ces dernières sont convexes. La résolution numérique du système nécessite une approximation convenable des points pour converger rapidement vers la solution. Pour y arriver, on calcule l'intersection de la droite reliant les centres de masse des deux surfaces avec ces dernières, ce qui donne un point sur chacune d'elle. Ces points sont utilisés au départ de la simulation, mais par la suite, les points trouvés au pas de temps précédents sont utilisés. Cela permet d'augmenter de beaucoup les performances

de l'algorithme lorsque la scène est cohérente entre chaque pas de temps.

1.2.3 Surfaces Paramétriques

Les surfaces paramétriques peuvent facilement être converties en polygones, il est donc possible d'utiliser les méthodes de détection de collisions relatives aux polygones. Par contre, comme elles sont définies à l'aide de fonctions mathématiques ayant des propriétés particulières, il est possible de mettre à profit ces propriétés pour améliorer les performances et augmenter la précision.

1.2.3.1 Méthodes de subdivision

Snyder, 1992 a présenté deux algorithmes basés sur l'arithmétique d'intervalle: SOLVE et MINIMIZE. Le premier permet de calculer la solution d'un système de contraintes et le second détermine le minimum global d'une fonction, pouvant être soumise à des contraintes. Les solutions obtenues ne sont par contre pas exactes, ce sont des intervalles permettant d'obtenir une borne supérieure et inférieure de la solution réelle. Une des applications qu'il propose à ces algorithmes est le calcul de la distance minimale entre deux surfaces paramétriques. Le problème est posé de la manière suivante:

$$\text{MINIMUM}\|\mathbf{S}(u,v) - \mathbf{T}(r,s)\|$$

où $\mathbf{S}(u,v)$ et $\mathbf{T}(r,s)$ sont les équations des surfaces paramétriques. L'algorithme de minimisation fonctionne en prenant au départ un certain domaine sur les variables (ici u, v, r et s) et en le subdivisant pour converger vers la plus petite borne

inférieure de la fonction à minimiser. La solution obtenue n'est bien sûr pas exacte, mais permet généralement d'obtenir une bonne approximation de la plus petite distance entre deux surfaces et des points les plus rapprochés. Cet algorithme a par la suite été optimisé pour la détection de collisions entre surfaces courbes en ajoutant des contraintes sur les points à trouver (Snyder *et al.*, 1993). Ces contraintes ont permis de diminuer la dimension du problème et ainsi d'augmenter les performances.

1.2.3.2 Arbres de Coquilles Sphériques

Les surfaces paramétriques peuvent facilement être subdivisées en sous-surfaces. Il est possible d'utiliser cette propriété pour construire une hiérarchie de volumes englobants permettant d'effectuer la détection de collisions entre surfaces paramétriques. Krishnan *et al.*, 1998 ont présenté un nouveau type de volume englobant: les coquilles sphériques. Une coquille sphérique est déterminée par deux sphères concentriques et un cône dont la pointe coïncide avec le centre des sphères. Le volume de la coquille correspond à l'intersection du volume du cône avec le volume de la grande sphère, moins le volume de la petite sphère (figure 1.8). L'algorithme qu'ils ont proposé pour effectuer la détection de collisions comporte deux phases. La première phase est le pré-calcul (avant la simulation) d'une boîte englobante orientée ainsi que d'un certain nombre de niveaux de la hiérarchie de coquilles sphériques pour chaque surface. La deuxième phase a lieu pendant la simulation et consiste à déterminer, pour chaque paire de surfaces, si les boîtes orientées se chevauchent. Si c'est le cas, on effectue une descente dans la hiérarchie de coquilles pour déterminer s'il y a réellement collision. Il est possible de calculer sur demande des niveaux plus raffinés de coquilles pour augmenter la précision. Le principal avantage de ce type de volume englobant est qu'il permet de représenter

Figure 1.8 Une coquille sphérique.

exactement des surfaces d'ordre deux. Par contre, la construction de ces coquilles sphériques est relativement complexe et le test d'intersection est beaucoup plus coûteux que pour d'autres types de volumes englobants.

CHAPITRE 2

ALGORITHME DE DÉTECTION DE COLLISIONS

Ce chapitre présente l'algorithme de détection de collisions entre surfaces paramétriques NURBS. Une surface NURBS est définie par une grille de points de contrôle permettant de spécifier la forme de la surface. La figure 2.1 montre une surface NURBS et son polygone de contrôle déterminé par ses points de contrôle. Une propriété

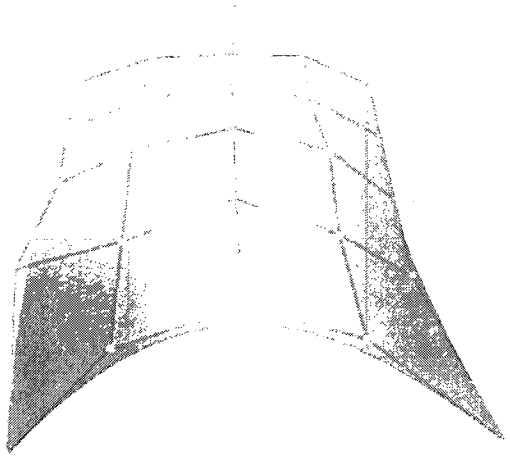

Figure 2.1 Une surface NURBS.

intéressante d'une surface NURBS est le fait qu'elle soit contenue dans l'enveloppe convexe définie par les points de contrôle. Cette propriété peut être exploitée dans le but d'effectuer de la détection de collision.

En effet, si l'on construit une boîte englobante contenant tous les points de contrôle, on peut être assuré que la surface est, elle aussi, contenue dans cette boîte. Par la suite, il est très rapide de tester si ces boîtes entrent en collision. Comme la boîte englobante est une approximation très grossière de la surface, plusieurs

collisions inexistantes seront détectées. Pour remédier à ce problème, il faut raffiner la précision des boîtes englobantes lorsqu'une collision est probable. Pour y arriver, les surfaces NURBS sont subdivisées en sous-surfaces, lesquelles sont utilisées pour construire de nouvelles boîtes englobantes.

L'algorithme de détection de collisions est donc de nature récursive : à chaque fois qu'une collision est détectée entre les boîtes englobantes, on subdivise les surfaces et l'on recommence le test avec les nouvelles surfaces. Le niveau de précision peut donc être déterminé à l'avance, en spécifiant le niveau maximal de récursion.

Les prochaines sections décrivent plus en détails les différentes étapes de cet algorithme. Premièrement, il sera question de la construction des boîtes englobantes. Ensuite, le test d'intersection entre les boîtes englobantes sera décrit, de même que la méthode utilisée pour subdiviser les surfaces. Finalement, toutes ces étapes seront mises ensemble pour bâtir l'algorithme complet de détection de collisions.

2.1 Construction des Boîtes Englobantes

Un bon volume englobant a une taille la plus petite possible, tout en permettant un test d'intersection rapide. Comme il en a été discuté au chapitre 1, plusieurs types de volumes englobants sont utilisables, ayant chacun leurs avantages et leurs inconvénients. Il a été décidé d'utiliser les boîtes englobantes orientées, car même si elles sont un peu plus longues à construire que les boîtes alignées sur les axes, elles épousent beaucoup mieux la forme de l'objet qu'elles englobent. Cette caractéristique augmente la vitesse de convergence de l'algorithme et sa précision.

La construction d'une boîte englobante orientée nécessite la détermination d'un système d'axes permettant de construire une boîte qui englobe le plus étroitement

possible la surface. Ce problème n'est pas trivial et plusieurs travaux ont été faits dans le but de trouver une solution optimale (Gottschalk *et al.*, 1996, Iones *et al.*, 1998, Barequet et Har-Peled, 2001). Ces algorithmes ont généralement une complexité trop grande pour pouvoir calculer les boîtes en temps réel. C'est pourquoi une solution plus simple, mais donnant une boîte orientée acceptable, a été retenue.

La détermination du système d'axe définissant l'orientation de la boîte est effectuée en calculant la matrice de covariance des points de contrôle. Les vecteurs propres de cette matrice définissent un système d'axe orthogonal dont un axe correspond à la plus grande variance et un autre à la plus petite variance. La figure 2.2 illustre le résultat pour un nuage de points en deux dimensions. L'orientation ainsi trouvée

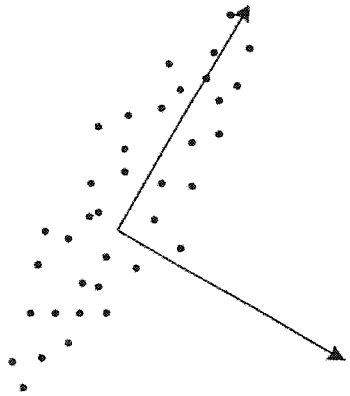

Figure 2.2 Le système d'axes défini par les vecteurs propres de la matrice de covariance.

ne garantit pas l'obtention d'une boîte orientée optimale, mais cette boîte permet tout de même d'obtenir de meilleures performances qu'une boîte qui aurait été alignée sur les axes du repère initial.

2.1.1 Calcul de l'Orientation

Soit une surface NURBS définie par :

$$\mathbf{S}(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)N_{j,q}(v)w_{i,j}\mathbf{P}_{i,j}}{\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u)N_{j,q}(v)w_{i,j}}$$

où les $\mathbf{P}_{i,j}$ sont les points de contrôles. La moyenne des points de contrôle est donnée par :

$$\mu = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} \mathbf{P}_{i,j}}{nm}$$

Les éléments de la matrice de covariance sont données par :

$$C_{k,l} = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} (\mathbf{P}_{i,j} - \mu)_k (\mathbf{P}_{i,j} - \mu)_l}{nm - 1}$$

où $k, l = 0, 1, 2$. Comme les vecteurs propres de deux matrices identiques à un facteur d'échelle près sont les mêmes, il est possible de simplifier les calculs en enlevant la division par $nm - 1$. On obtient donc la matrice $\mathbf{C}'$ dont les éléments sont donnés par :

$$C'_{k,l} = \sum_{i=0}^{n}\sum_{j=0}^{m} (\mathbf{P}_{i,j} - \mu)_k (\mathbf{P}_{i,j} - \mu)_l$$

Le calcul de cette matrice est très rapide, il a une complexité $O(n)$ en fonction du nombre de points de contrôle.

Ensuite, il faut calculer les vecteurs propres normalisés de cette matrice pour obtenir l'orientation de la boîte. Comme la matrice de covariance est symétrique, ce calcul est lui aussi très rapide. L'orientation de la boîte est spécifiée à l'aide d'une matrice de rotation 3×3 :

$$\mathbf{R} = \begin{bmatrix} v1_0 & v2_0 & v3_0 \\ v1_1 & v2_1 & v3_1 \\ v1_2 & v2_2 & v3_2 \end{bmatrix}$$

où $v1, v2, v3$ sont les vecteurs propres normalisés de la matrice de covariance.

2.1.2 Création de la Boîte

Il faut maintenant trouver les dimensions de la boîte. Il suffit de trouver les minimums et maximums sur les trois axes de coordonnées des points de contrôle dans le système d'axes trouvé précédemment ($\mathbf{P'}_{i,j} = \mathbf{R}^T\mathbf{P}_{i,j}$). Ces valeurs permettent de construire deux vecteurs, $\mathbf{V}_{min}$ et $\mathbf{V}_{max}$.

Une boîte orientée est définie par trois variables : la position de son centre, ses dimensions et son orientation. La position de son centre est donnée par $(\mathbf{V}_{min} + \mathbf{V}_{max})/2$, ses dimensions sont $(\mathbf{V}_{max} - \mathbf{V}_{min})/2$ et son orientation est donnée directement par la matrice $\mathbf{R}$. La construction de la boîte est maintenant terminée.

2.2 Intersection de Boîtes Englobantes Orientées

Comme il a été dit précédemment, un bon volume englobant doit avoir un test d'intersection rapide. À première vue, il peut sembler coûteux en temps de calcul

d'effectuer ce test entre deux boîtes ayant une orientation différente. Par contre, Gottschalk *et al.*, 1996 ont montré qu'il peut être effectué très efficacement. Cette section explique plus en détail leur manière de procéder.

Deux boîtes ayant la même orientation peuvent être rapidement testées pour savoir si elles se chevauchent. Il suffit de projeter ces deux boîtes sur les axes de leur système de référence et de vérifier que leurs projections se chevauchent sur les trois axes simultanément. Si c'est le cas, c'est qu'elles entrent en collision. Ce principe peut être étendu pour des boîtes ayant une orientation quelconque : si on trouve un axe sur lequel les projections des deux boîtes sont disjointes, c'est que les boîtes le sont aussi. Un tel axe est appelé axe séparateur (figure 2.3). Gottschalk, 1996 a

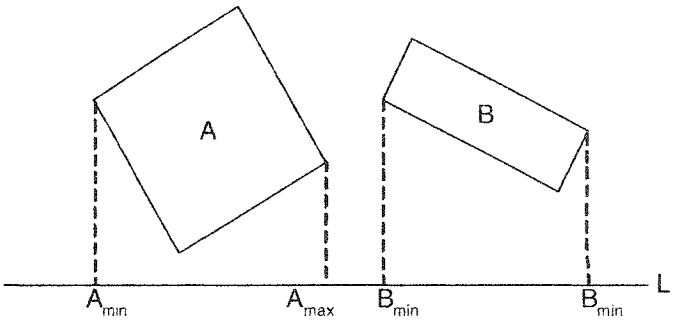

Figure 2.3 L est un axe séparateur pour les boîtes A et B.

montré que pour deux boîtes orientées, un maximum de 15 axes est nécessaire pour déterminer si ces boîtes entrent en collision. Ces axes sont définis par les normales des faces des deux boîtes, de même que par le produit vectoriel de ces normales prises deux à deux. Il y a 3 orientations de normale par boîte, donc 9 combinaisons possibles, ce qui donne bien 15 axes au total.

Ce test est rapide, car dès que l'on trouve un axe séparateur, il est certain que les deux boîtes sont disjointes. Il y a donc au maximum 15 tests à effectuer, mais très souvent, ce nombre est plus petit.

2.3 Subdivision d'une surface NURBS

Pour raffiner la solution et faire converger l'algorithme de détection de collision, il est nécessaire de subdiviser les surfaces lorsqu'il y a possibilité de collision. Pour ce faire, l'algorithme d'insertion de noeud est utilisé (Piegl et Tiller, 1996). Cet algorithme permet d'ajouter des noeuds dans un des vecteurs nodaux d'une surface NURBS sans modifier sa forme. Comme on ajoute des noeuds, il est nécessaire d'ajouter aussi des points de contrôle. L'algorithme s'occupe de calculer la position de ces points pour que la surface conserve sa forme initiale.

Lorsqu'un noeud est présent p fois dans un vecteur nodal, où p est le degré de la surface dans cette direction, il y a création d'une discontinuité à cette valeur paramétrique. Cette discontinuité permet de séparer la surface en deux nouvelles surfaces indépendantes. C'est cette propriété qui permet d'utiliser l'algorithme d'insertion de noeud pour subdiviser la surface.

2.4 Algorithme Global

Les sections précédentes ont présenté les différentes parties de l'algorithme de détection de collisions. Il est maintenant temps de les rassembler.

L'idée principale de l'algorithme consiste à construire des boîtes englobantes orientées à partir des points de contrôle des surfaces et de tester s'il y a collision entre ces boîtes. Si c'est le cas, on subdivise ces surfaces et on recommence l'étape précédente avec les nouvelles surfaces tant que la précision voulue n'est pas atteinte. L'algorithme 2.1 illustre ce processus. Il assume l'existence des fonctions *BuildOBB* et *OBBIntersect* qui, respectivement, construit une boîte englobante orientée à partir d'une surface NURBS et effectue un test d'intersection entre deux

boîtes englobantes orientées.

Algorithme 2.1 NURBSCollideBase(Surface1, Surface2)

```
Box1 ← BuildOBB( Surface1 )
Box2 ← BuildOBB( Surface2 )
if OBBIntersect( Box1, Box2 ) then
  if niveau maximal de récursion atteint then
    Collision détectée
  else
    Subdivision et appel récursif à
    NURBSCollideBase
  end if
else
  Pas de collision
end if
```

L'exécution de cet algorithme produit un arbre de recherche pouvant avoir différentes allures selon la manière dont les surfaces sont subdivisées à chaque appel de la fonction. En effet, il est possible de les subdiviser en deux dans une des deux directions paramétriques ou de les subdiviser en quatre, en faisant une division dans chaque direction. Il est aussi possible de subdiviser une seule surface ou les deux à la fois. Par exemple, l'algorithme 2.2 effectue une subdivision des deux surfaces en quatre. Il assume l'existence de la fonction *Subdivide4* qui subdivise une surface NURBS en quatre.

On peut estimer le temps de calcul d'un algorithme de cette nature avec l'équation suivante :

$$T = NC_cC_tC_s$$

Où N est le nombre de fois que la fonction sera exécutée, C_c est le coût de la construction d'une boîte englobante orientée, C_t est le coût du test et C_s est le coût de la subdivision.

Pour obtenir un algorithme rapide, il faudrait pouvoir diminuer ces quatre quan-

Algorithme 2.2 NURBSCollide(Surface1, Surface2, Level)

```
Box1 ⇐ BuildOBB( Surface1 )
Box2 ⇐ BuildOBB( Surface2 )
if OBBIntersect( Box1, Box2 ) then
  if Level = 0 then
    Collision détectée
  else
    SubSurfaces1 ⇐ Sudivide4( Surface1 )
    SubSurfaces2 ⇐ Sudivide4( Surface2 )
    for i = 1 → 4 do
      for j = 1 → 4 do
        NURBSCollide( SubSurfaces1_i, SubSurfaces2_j, Level −1 )
      end for
    end for
  end if
else
  Pas de collision
end if
```

tités. Comme on travaille avec des boîtes orientées, on ne peut jouer sur les valeurs de C_c et C_t. Par contre, en fonction de la stratégie de subdivision utilisée, on peut facilement jouer sur les termes C_s et N. Évidemment, il est plus rapide de subdiviser une surface en deux que de la subdiviser en quatre. Pour diminuer la valeur de C_s, on doit donc diviser les surfaces en deux. De plus, si l'on subdivise seulement une seule des deux surfaces à chaque appel de la fonction, en alternant entre les deux, il est possible de diminuer ce terme de moitié. Évidemment, en subdivisant moins les surfaces à chaque fois, plus de tests seront nécessaires et N augmentera. Par contre, comme C_s est le terme le plus important dans l'équation, les performances globales sont améliorées si une seule subdivision a lieu à chaque appel de la fonction.

Ce petit calcul nous pousse donc à modifier l'algorithme 2.2 pour obtenir de meilleures performances. Il faut le modifier pour ne subdiviser qu'une seule surface à chaque fois. Comme le nouvel algorithme ne subdivise qu'une seule surface à la

fois, et ce dans une seule direction paramétrique, on doit s'assurer que l'on alterne entre les deux surfaces et entre les deux directions paramétriques. L'alternance entre les deux surfaces est facile à réaliser : il suffit d'appeler récursivement la fonction avec la seconde surface devenue première surface. Pour la subdivision, on choisit toujours la première surface. Il faut par contre déterminer dans quelle direction on subdivise. Si un des deux vecteurs nodaux contient plus de noeuds que l'autre, on subdivise dans cette direction. Par contre, si les deux ont la même taille, il faut trouver un moyen d'alterner entre les deux directions. Pour ce faire, on vérifie la valeur du second bit du niveau de récursion actuel. Ce bit change de valeur à tous les deux niveaux, ce qui le rend très approprié comme indicateur de direction de subdivision. On obtient finalement l'algorithme 2.3, qui implémente ces nouvelles caractéristiques. Comme cet algorithme ne subdivise qu'une seule des deux sur-

Algorithme 2.3 NURBSCollide(Surface1, Box1, Surface2, Level)

```
Box2 ⇐ BuildOBB( Surface2 )
if OBBIntersect( Box1, Box2 ) then
  if Level = 0 then
    Collision détectée
  else
    if Surface1.NbUKnots > Surface1.NbVKnots then
      SubSurfaces ⇐ SubdivideU( Surface1 )
    else if Surface1.NbUKnots ≠ Surface1.NbVKnots then
      SubSurfaces ⇐ SubdivideV( Surface1 )
    else if Level AND 2 then
      SubSurfaces ⇐ SubdivideU( Surface1 )
    else
      SubSurfaces ⇐ SubdivideV( Surface1 )
    end if
    NURBSCollide( Surface2, Box2, SubSurfaces_0, Level −1 )
    NURBSCollide( Surface2, Box2, SubSurfaces_1, Level −1 )
  end if
else
  Pas de collision
end if
```

faces à la fois, il prend comme paramètre la boîte englobante de la surface 1. Il aurait été possible de faire calculer cette boîte dans la fonction, mais cela aurait été redondant, car elle a déjà été calculée au niveau de récursion précédent. Cela implique que lorsqu'on utilise cet algorithme, il faut calculer la boite englobante de la surface 1 avant de l'appeler (car le premier appel n'est pas récursif).

Cet algorithme est facilement paramétrable pour obtenir une précision ou une vitesse d'exécution voulue. La variable *Level* est utilisée pour limiter le niveau de récursion, et ainsi limiter le temps d'exécution à une borne supérieure. Il est possible de modifier cette condition sur *Level* pour une condition sur la taille des boîtes englobantes, ce qui permet de définir un niveau de précision en terme de dimensions spatiales.

Il est aussi possible de paramétrer le nombre de points de collision que l'algorithme retourne. Si l'information nécessaire est simplement de savoir s'il y a collision ou non, alors dès qu'un point de collision est trouvé, on sort de la fonction en retournant ce point. Par contre, si tous les points de collision sont nécessaires, il est possible de les ajouter dans une liste à mesure qu'ils sont détectés et de continuer la recherche à chaque fois.

La position de la collision peut être déterminée de plusieurs manières. Une approximation de la position dans l'espace peut être calculée en faisant la moyenne du centre de masse des deux boîtes englobantes qui ont permis de détecter la collision. Cette solution est rapide, mais n'est pas très précise. Il est possible de modifier l'algorithme pour permettre de déterminer approximativement les paramètres (u, v) de chaque surface où la collision a eu lieu. Pour y arriver, on doit passer en paramètre à l'algorithme les coordonnées (u, v) où se situe le milieu des sous-surfaces par rapport aux surfaces initiales. L'algorithme 2.4 illustre la manière de procéder.

Algorithme 2.4 NURBSCollide(Surface1, Box1, Surface2, Level, u1, v1, u2, v2)

```
Box2 ← BuildOBB( Surface2 )
if OBBIntersect( Box1, Box2 ) then
  if Level = 0 then
    Collision détectée à (u1, v1) and (u2, v2)
  else
    λ ← InitialLevel − Level +2
    if Surface1.NbUKnots > Surface1.NbVKnots then
      SubSurfaces ← SubdivideU( Surface1 )
      Δ_u ← 1/2^λ
      Δ_v ← 0
    else if Surface1.NbUKnots ≠ Surface1.NbVKnots then
      SubSurfaces ← SubdivideV( Surface1 )
      Δ_u ← 0
      Δ_v ← 1/2^λ
    else if Level AND 2 then
      SubSurfaces ← SubdivideU( Surface1 )
      Δ_u ← 1/2^λ
      Δ_v ← 0
    else
      SubSurfaces ⇐ SubdivideV( Surface1 )
      Δ_u ⇐ 0
      Δ_v ⇐ 1/2^λ
    end if
    NURBSCollide( Surface2, Box2, SubSurfaces_0, Level −1, u1 + Δ_u, v1 + Δ_v
    )
    NURBSCollide( Surface2, Box2, SubSurfaces_1, Level −1, u1 − Δ_u, v1 − Δ_v
    )
  end if
else
  Pas de collision
end if
```

Cette solution permet de trouver les coordonnées paramétriques de la collision. Si ces coordonnées ne sont pas obligatoires, mais qu'une bonne approximation de la courbe d'intersection dans l'espace est nécessaire, il est possible de faire une nouvelle modification pour l'obtenir. Au lieu d'arrêter l'algorithme en fonction du niveau de récursion ou de la taille des boîtes englobantes, il est possible de l'arrêter lorsque l'on considère que les sous-surfaces peuvent être représentée par un plan. Il suffit de vérifier si une des trois dimensions des boîtes englobantes est proche de 0. Dans ce cas, on considère que la sous-surface est plane et l'on construit quatre triangles à partir des points de contrôle aux coins. Il ne reste plus qu'à effectuer le calcul des droites d'intersection entre ces triangles pour obtenir une approximation de la courbe d'intersection réelle entre ces deux sous-surfaces. En effectuant ce calcul pour toutes les collisions détectées entre deux surfaces, on obtient un ensemble de segments de droite qui approximent la courbe d'intersection réelle.

Une autre caractéristique intéressante de l'algorithme est le fait qu'il soit possible de calculer une approximation de la normale de la surface aux points de collision. En effet, lorsque le niveau de récursion est assez grand, les boîtes englobantes deviennent très minces dans le sens de la surfaces. Elles finissent par devenir pratiquement des polygones rectangulaires. Il suffit donc de trouver la normale de ce polygone pour obtenir une bonne approximation de la normale de la surface. C'est une caractéristique intéressante, car dans plusieurs applications, cette donnée est très importante à la gestion de la collision.

Ce nouvel algorithme de détection de collisions entre surfaces NURBS est donc simple et facilement paramétrable en fonction de l'application visée. Comme les boîtes englobantes sont calculées seulement lorsque nécessaires et qu'aucun calcul ne doit être effectué avant la simulation, il utilise peu de mémoire et il permet de

travailler avec des surfaces déformables. En effet, la modification des points de contrôle ou même des noeuds des vecteurs nodaux entre deux requêtes de collision n'entraîne aucun coût de traitement supplémentaire. Les performances obtenues avec cet algorithme sont aussi très intéressantes. Le chapitre 3 discute des résultats obtenus pour diverses applications.

CHAPITRE 3

RÉSULTATS ET DISCUSSION

Ce chapitre présente les résultats obtenus par l'algorithme de détection de collisions entre surfaces NURBS et discute de ses forces et faiblesses. Comme il existe très peu d'algorithmes permettant d'effectuer de la détection de collisions entre surfaces paramétriques, il sera très difficile de faire une comparaison avec des solutions existantes. La majorité des recherches en détection de collisions a été faite pour des modèles polygonaux, les surfaces paramétriques sont donc généralement discrétisées en polygones et le test de collision est effectué sur ces polygones.

L'algorithme 2.3 a été implémenté en C++ de manière à retourner la première collision trouvée. Afin de déterminer ses performances, des tests ont été effectués et le temps moyen d'une requête de détection de collisions a été calculé. Ce temps de réponse peut varier significativement en fonction de différentes variables, telles que la distance et l'orientation des surfaces, du niveau maximal de récursion ainsi que de la complexité des surfaces. Pour visualiser l'impact qu'ont ces variables sur la performance de l'algorithme, différents tests ont été effectués.

En plus de ces tests de performance, d'autres tests ont été réalisés afin de déterminer la précision de l'algorithme. Afin de comparer les résultats théoriques et expérimentaux, différentes configurations de surfaces, dont les points de contact pouvaient être déduits théoriquement, ont été modélisés. Deux types de ces tests ont été exécutés: le premier utilisait des surfaces statiques et le second, des surfaces mobiles. Le mouvement des surfaces mobiles était géré par le simulateur physique présenté au chapitre précédent.

Finalement, l'algorithme a été testé en effectuant une simulation d'objets représentés à l'aide de surfaces NURBS et dont le mouvement était géré par le simulateur physique. Cette simulation n'étant pas analysable théoriquement, le but de ce test était de vérifier si l'algorithme donne des résultats assez précis pour être utilisé dans une simulation physique convaincante.

À la suite des tests effectués, une discussion sera présentée, portant sur les résultats obtenus. Il y sera entre autre question des forces ainsi que des faiblesses de l'algorithme.

3.1 Test de l'Influence du Niveau de Récursion

Ce test a été réalisé dans le but de déterminer la vitesse d'exécution de l'algorithme en fonction du niveau maximal de récursion spécifié. Les surfaces utilisées dans ce test sont illustrées à la figure 3.1 et leurs caractéristiques sont données dans le tableau 3.1.

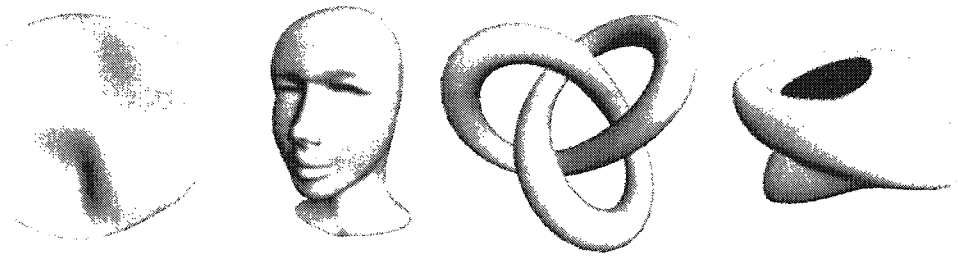

Figure 3.1 Surfaces utilisées lors du premier test.

Des séries de 500 tests ont été effectuées entre les 4 surfaces, ce qui donne en tout $500 \times 6 = 3000$ tests surface-surface par série, car il existe 6 paires de surfaces distinctes pour un ensemble de 4 surfaces. Au début de chacun de ces 500 tests, les surfaces ont été disposées à une position et orientation aléatoire. Une valeur

Tableau 3.1 Caractéristiques des surfaces du premier test.

Surface	Points de contrôle	Degré
1	5×5	3
2	30×25	3
3	70×21	3
4	15×14	3

moyenne fiable du temps d'une requête a donc pu être calculée après chaque série de test.

Pour évaluer l'influence du niveau de récursion sur le temps de réponse de l'algorithme, des séries de tests ont été faites avec différentes valeurs de niveau de récursion. Les surfaces ont été placées de manière à obtenir en moyenne un taux de 50% de collision, c'est-à-dire que chaque requête retourne une collision une fois sur deux.

3.1.1 Résultats

La figure figure 3.2 illustre la relation qui existe entre le temps moyen d'une requête et le niveau de récursion utilisé. Comme on peut s'y attendre, plus le niveau de récursion est élevé, plus le temps d'une requête augmente. Par contre, il est intéressant de noter qu'à partir du niveau 30, la courbe atteint un plateau jusqu'au niveau 55. Cela veut donc dire qu'il est possible d'obtenir des résultats d'une plus grande précision, sans que le temps de la requête n'augmente significativement. Par contre, à partir du niveau 55, le temps d'une requête augmente très rapidement en fonction du niveau de récursion. Ce comportement est dû au fait que l'algorithme a été implémenté avec des nombres à virgule flottante simple précision et qu'à ce niveau, on dépasse la précision de ces nombres. Bien entendu, ces résultats sont très liés aux surfaces utilisées. Si leur taille avait été plus ou moins grande, le niveau

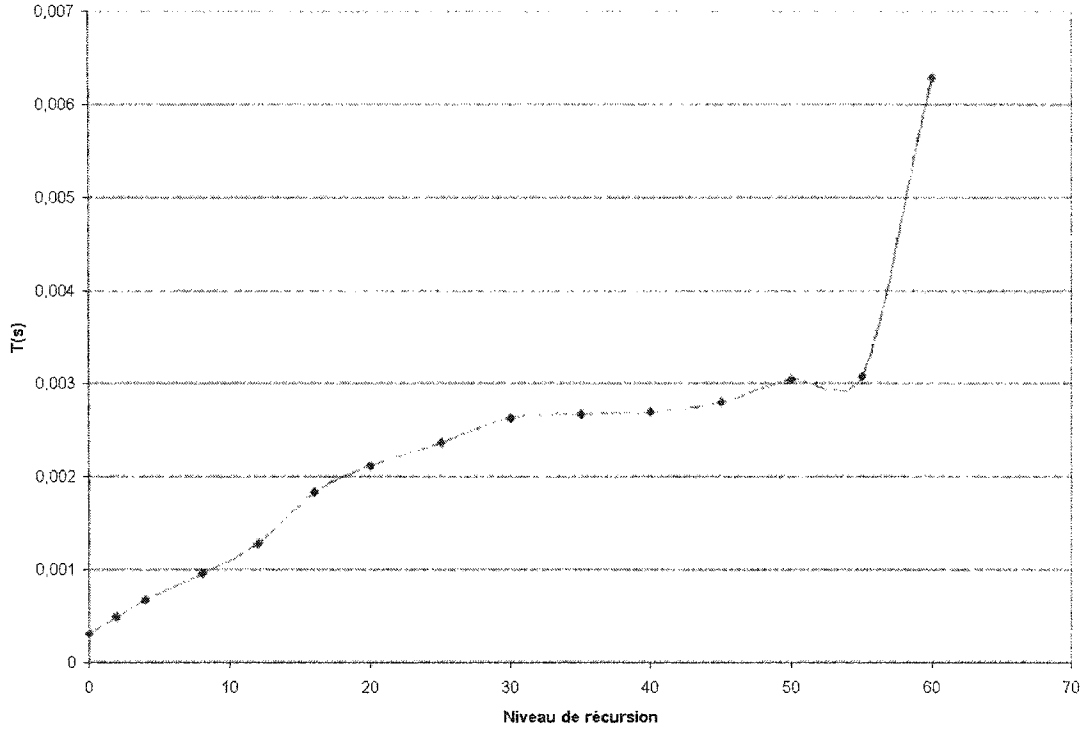

Figure 3.2 Temps d'une requête en fonction du niveau de récursion.

maximal de récursion possible sans dépasser la précision des nombres auraient été différent.

3.2 Test de l'Influence du Taux de Collision

Ce test a été réalisé dans le but de déterminer la vitesse d'exécution de l'algorithme en fonction de la probabilité qu'une collision se produise. Les surfaces utilisées pour ce test sont celle illustrées à la figure 3.1 et dont les caractéristiques sont données dans le tableau 3.1.

Encore une fois, des séries de 500 tests ont été effectuées entre les 4 surfaces, donc 3000 tests surface-surface par série. Au début de chacun de ces 500 tests,

les surfaces ont été disposées à une position et orientation aléatoire. Une valeur moyenne fiable du temps d'une requête a donc pu être calculée après chaque série de test.

Pour évaluer l'influence de la probabilité de collision sur le temps de réponse de l'algorithme, des séries de tests ont été faites de manière à obtenir en moyenne un certain taux de collision. La méthode utilisée pour contrôler le taux de collision consiste à placer aléatoirement les surfaces à l'intérieur d'un cube de volume variable. Plus le cube est petit, plus le taux de collision est élevé et plus le cube est grand, plus le taux est petit. À la fin de chaque série de tests, le nombre total de collisions détectées a été divisé par le nombre de tests, ce qui donnait une bonne idée du taux de collision. Pour une même taille de cube, les taux étaient très similaires d'une série à l'autre.

3.2.1 Résultats

La figure figure 3.3 illustre la relation qui existe entre le temps moyen d'une requête et le taux de collision. On voit très bien que plus la probabilité qu'une collision se produise est élevée, plus le temps d'une requête est grand. Entre un taux de collision de 0% et de 100%, il y a un ordre de grandeur de différence. Ce phénomène s'explique par la nature de l'algorithme. L'utilisation de volumes englobants est très efficace lorsque les surfaces sont éloignées les unes des autres. Par contre, si elles sont près, les surfaces devront être subdivisées un très grand nombre de fois et à plusieurs endroits avant de pouvoir déterminer si elles se touchent ou non. La figure 3.4 illustre une disposition de surfaces menant au pire cas. Comme les surfaces sont semblables et disposées de manière parallèle sans se toucher, elles devront être subdivisées uniformément sur toute leur superficie avant que l'algorithme puisse déterminer qu'elles ne se touchent pas.

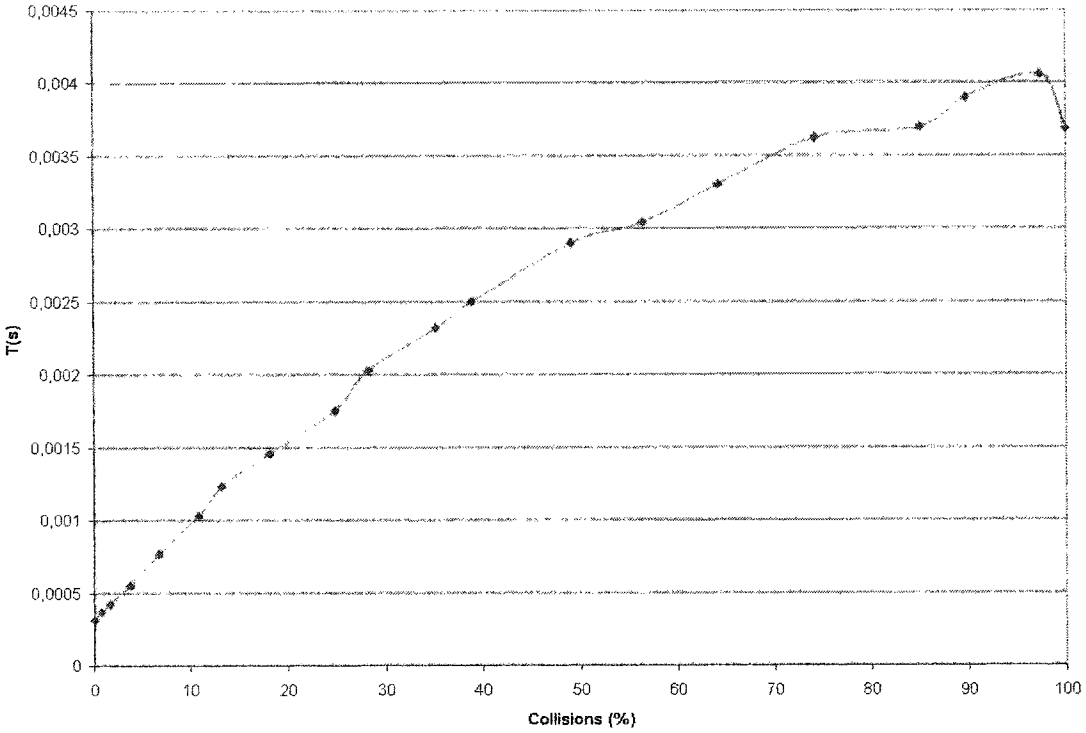

Figure 3.3 Temps d'une requête en fonction du taux de collision.

On peut aussi remarquer une petite diminution du temps d'une requête lorsque la probabilité de collision atteint 100%. Cela est du au fait que l'algorithme a été implémenté pour ne trouver que la première collision. Il arrête dès qu'il en trouve une. L'algorithme est donc légèrement plus rapide s'il y a collision que s'il n'y en a pas dans un pire cas, car il n'a pas à subdiviser complètement les deux surfaces pour s'assurer qu'elles ne se touchent pas.

3.3 Test de l'Influence de la Complexité des Surfaces

Ce test a été réalisé afin de déterminer quel est l'impact de la complexité des surfaces sur le temps d'exécution de l'algorithme. Une seule surface a été utilisée

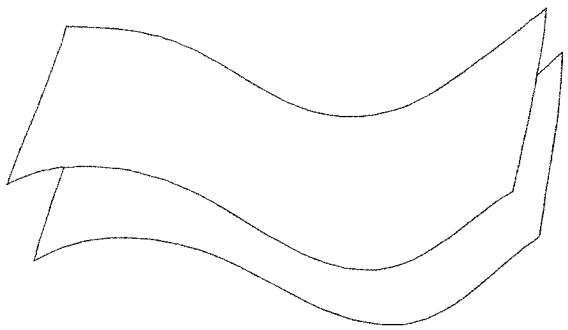

Figure 3.4 Disposition de surfaces menant au pire cas de l'algorithme de détection de collisions.

pour ce test, mais en deux exemplaires. Cette surface était originellement une grille de 4×4 points de contrôles avec un certain niveau de bruit. Comme la complexité d'une surface est directement reliée au nombre de points de contrôle qu'elle contient, cette surface a été raffinée pour obtenir d'autres surfaces identiques, mais contenant plus de points de contrôle. Trois autres surfaces ont ainsi été créées: une de 10×10, une autre de 15×15 et finalement une de 20×20.

Le fait que les surfaces soient identiques, mais avec un nombre de points de contrôle différent, permet de réaliser des séries de tests identiques, mais avec des surfaces de complexités différentes. Il est alors possible de mesurer l'influence de cette complexité sur le temps de réponse de l'algorithme.

Pour chacune des quatre surfaces, une série de 5000 tests a été effectuée. Chacun de ces tests consistait à placer les deux surfaces à des positions et orientations aléatoires et ensuite vérifier s'il y a collision. Les surfaces ont été placées de manière à obtenir en moyenne un taux de 50% de collision, c'est-à-dire que chaque requête retourne une collision une fois sur deux.

3.3.1 Résultats

La figure 3.5 illustre la relation qui existe entre le temps moyen d'une requête et la complexité des surfaces. Ces résultats montrent que, comme prévu théoriquement,

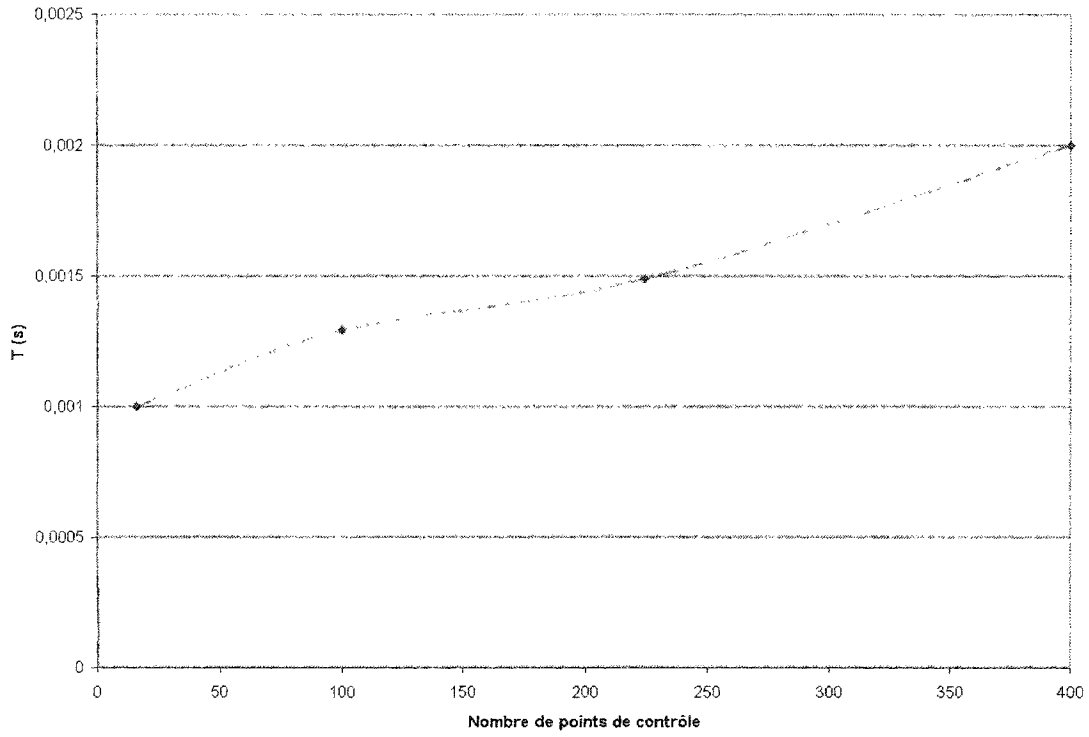

Figure 3.5 Temps d'une requête en fonction de la complexité des surfaces.

la complexité de l'algorithme est $O(n)$ en fonction du nombre de points de contrôle. Le nombre de points de contrôle affecte le temps de calcul de la matrice de covariance permettant de déterminer l'orientation des boîtes englobantes. Ce calcul est linéaire en fonction du nombre de points, ce qui est confirmé par ces résultats.

3.4 Test de Précision

Plusieurs tests ont été effectués afin de déterminer si l'algorithme de détection de collisions retourne des points de contact valides et précis. Des situations pouvant être solutionnées analytiquement ont été reproduites et les résultats expérimentaux ont été comparés avec les résultats théoriques.

Deux séries de tests ont été effectués. La première utilisait des paires d'objets immobiles en contact. Six cas ont été étudiés:

1. Deux sphères une par dessus l'autre, le point de contact étant à l'origine;
2. Une sphère à l'origine et une autre située en diagonale sur le plan xy;
3. Une sphère à l'origine et une autre située en diagonale selon les trois axes de coordonnées;
4. Une sphère au dessus du plan xz;
5. Une sphère au dessus du plan xz tourné de 30° autour de l'axe des z;
6. Une sphère au dessus du plan xz tourné de 30° autour de l'axe des z et de 30° autour de l'axe des y.

Ces situations sont illustrées sur la figure 3.6. La sphère est modélisée à l'aide de huit surfaces NURBS de degré 2, une pour chaque octant. Le plan est une surface NURBS de degré 3 de 4×4 points de contrôle. Ces différents cas visaient à tester la précision de l'algorithme lorsque le point de contact est situé à différents endroits des surfaces, comme sur un coin, sur un côté ou ailleurs sur la surface. Les cas 1 et 4 mènent à un point de contact sur un pôle d'une sphère, là où les coins de quatre surfaces se joignent. Les cas 2 et 5 mènent à un point de contact sur un joint entre

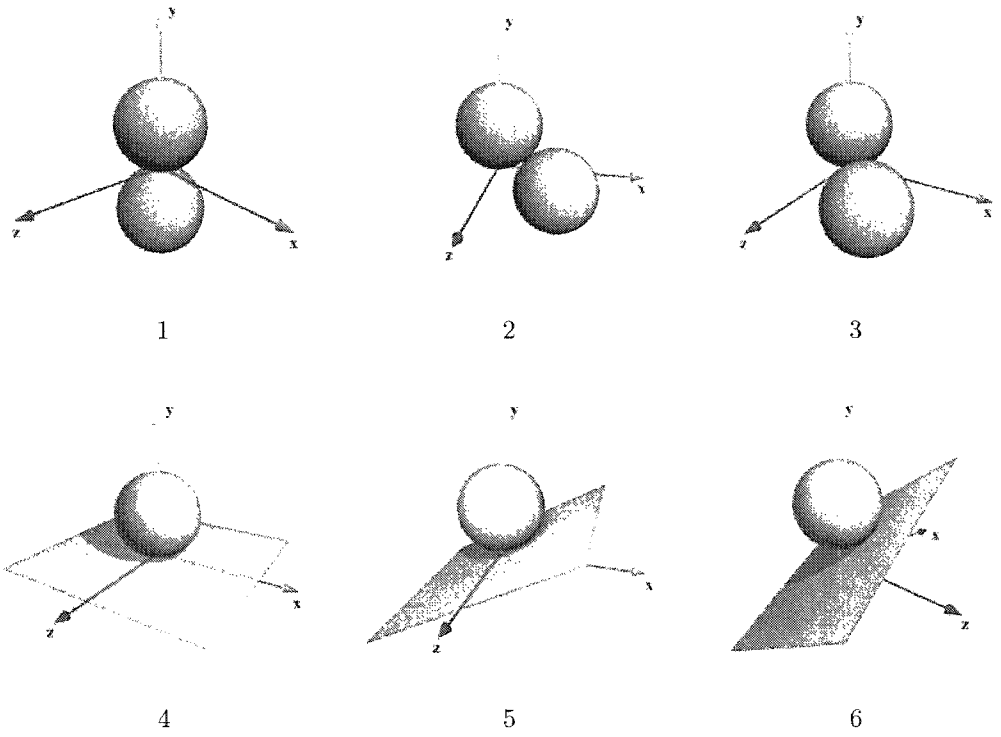

Figure 3.6 Les six situations du test de precision.

deux surfaces. Finalement, les cas 3 et 6 mènent à un point de contact situé à un endroit quelconque à l'intérieur des surfaces.

La seconde série de test consistait à reproduire les même situations, mais en utilisant le simulateur physique présenté à l'annexe I pour déplacer les objets. Dans chaque cas, un des deux objets était immobile et le second tombait sous la force de la gravité. L'objet tombant n'était pas initialement en contact avec l'objet immobile. Le but de ces tests était de valider la précision de l'algorithme lorsqu'utilisé dans un système dynamique. La précision de ces tests dépend du pas de temps de l'intégrateur du simulateur physique, c'est pourquoi les tests de précision avec des objets statiques et des objets mobiles ont été effectués séparément. Le pas de temps utilisé pour les tests dynamiques est de 0.0001.

Pour chacune des deux séries de tests, la norme utilisée pour mesurer l'erreur est la distance entre le point de contact retourné par l'algorithme et le point de contact théorique. Comme la précision est directement liée au niveau maximal de récursion utilisé, l'erreur de position des points de contact a été mesurée pour différents niveaux de récursion.

3.4.1 Résultats

Les résultats des tests effectués avec les surfaces statiques sont présentés sur la figure 3.7. Comme on peut le constater, les courbes sont linéaires en fonction du

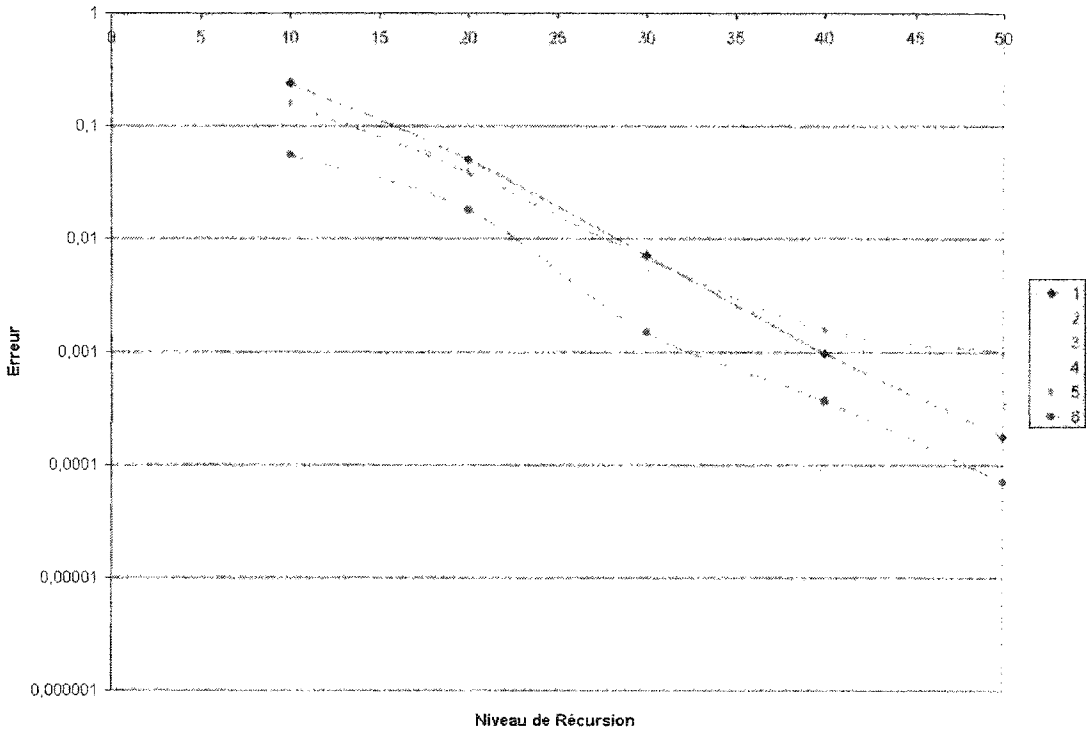

Figure 3.7 Erreur en fonction du niveau de récursion pour des surfaces statiques.

niveau de récursion. Comme l'axe des erreurs a une échelle logarithmique, on peut conclure que l'erreur diminue exponentiellement en fonction du niveau de récursion.

Les résultats des tests effectués avec les surfaces mobiles sont présentés sur la figure 3.8. Les résultats obtenus ici sont semblables à ceux observés avec des surfaces

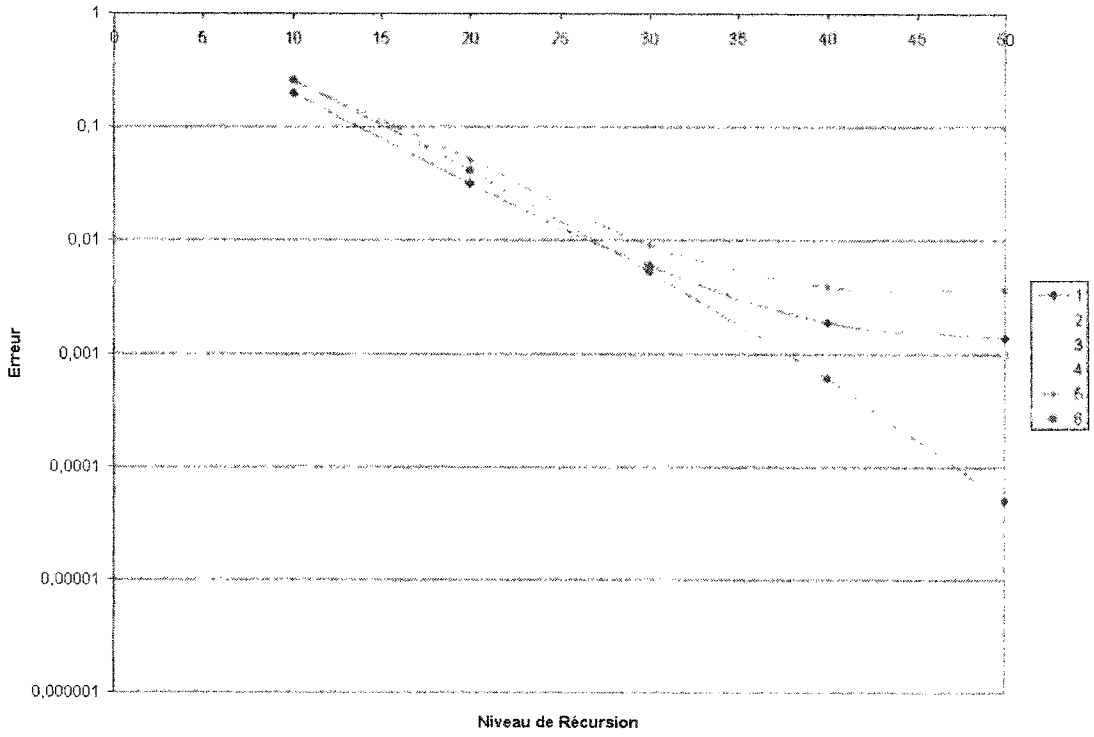

Figure 3.8 Erreur en fonction du niveau de récursion pour des surfaces mobiles.

statiques, mais environ un ordre de grandeur moins précis. Cette diminution de précision s'explique par le fait que la détection de collisions est effectuée seulement entre des intervalles discrets de temps, en fonction du pas de temps de la simulation. Le premier contact entre deux surfaces a lieu à un temps précis, mais ce temps peut être n'importe quand à l'intérieur d'un pas de temps. La collision est donc toujours détectée un peu après le temps du premier contact, les objets s'interpénètrent donc un peu, de là la diminution de précision.

Dans les deux cas, l'erreur diminue exponentiellement en fonction du niveau de récursion. Ce comportement est dû à la nature de l'algorithme: à chaque niveau de

récursion, une des surfaces est subdivisée en deux selon une direction paramétrique. La précision selon cette direction augmente ainsi de deux. À tous les quatre niveaux, les deux surfaces ont été subdivisées en quatre sous-surfaces, on peut donc déduire que la précision dans le domaine paramétrique est proportionnelle à $2^{\frac{n}{2}}$, où n est le niveau maximal de récursion.

On peut remarquer que la courbe du 3^e cas montre une erreur très faible pour le niveau 20 de récursion. Ce résultat étrange est probablement causé par symétrie un peu trop parfaite des cas de test. En effet, la collision pour ce cas est exactement au centre de deux surfaces parmi celles formant les sphères. Il est probable qu'au niveau 20 de récursion, la subdivision tombe presque qu'exactement au point théorique de collision. Ce point ne devrait donc pas être pris en compte pour déterminer la précision de l'algorithme dans le cas général.

3.5 Test d'Intégration au Moteur Physique

Le dernier test ayant été effectué consistait à réaliser une simulation physique comportant trois corps rigides définis à l'aide de surfaces NURBS. Cette simulation ne s'analyse pas théoriquement dû à sa complexité. Le but de ce test n'est donc pas de comparer des résultats théoriques avec des résultats expérimentaux, mais bien de voir si l'algorithme de détection de collisions est suffisamment stable et précis pour être utilisé dans une simulation.

La simulation consiste à laisser tomber deux objets, un tore et un vase, sur une surface rectangulaire attachée par quatre ressorts, un à chaque coin. À chaque fois qu'une collision est détectée, une impulsion est appliquée sur les objets au point de contact, de manière à simuler une réponse physiquement correcte à la collision. De cette manière, les objets ne s'interpénètrent pas et la simulation est convaincante.

Comme la simulation nécessite un très grand nombre de requêtes de détection de collisions par seconde (entre 1000 et 10000), la simulation n'a pas été calculée en temps réel.

3.5.1 Résultats

La figure 3.9 illustre les résultats obtenus pour un intervalle de neuf secondes. La séquence d'images montre l'état de la scène à chaque seconde. Lorsque visualisés

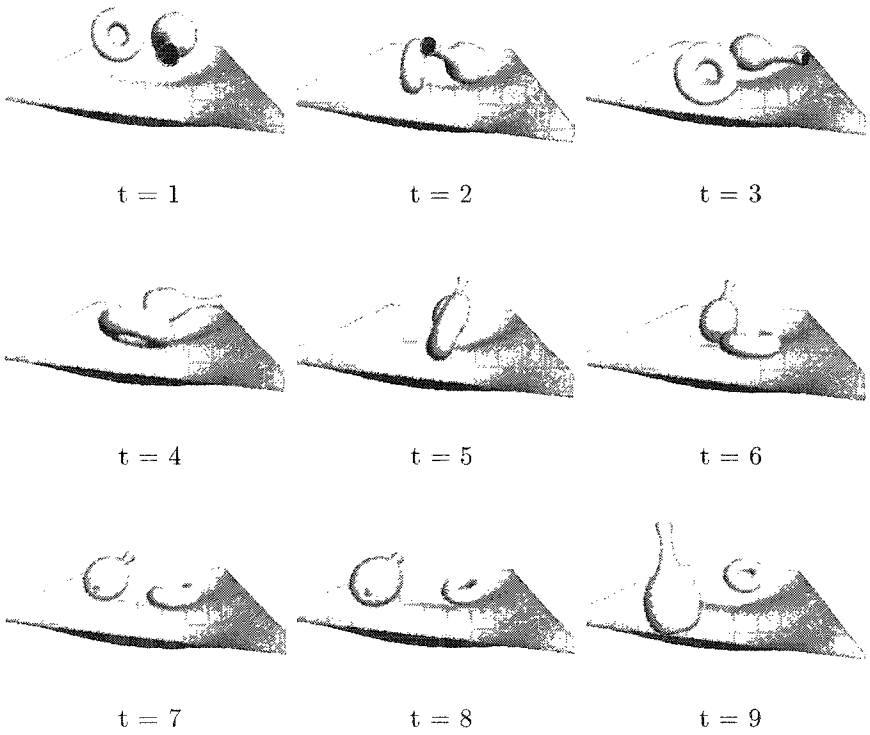

Figure 3.9 Séquence d'images du test d'intégration au moteur physique.

sous forme d'animation, les résultats sont très convaincants et la simulation semble physiquement correcte. L'algorithme de détection de collisions est donc utilisable dans un environnement de simulation, par contre, il n'est pas assez rapide pour l'être en temps réel.

3.6 Discussion

À la lumière des résultats obtenus, plusieurs conclusions peuvent être tirées. Premièrement, comme les temps de requêtes sont généralement de quelques millisecondes, on peut dire que l'algorithme est assez rapide pour être utilisé dans une application en temps réel demandant environ une centaine de requêtes par seconde.

Ensuite, la courbe de la figure 3.2 montre un plateau intéressant à partir du niveau 30 environ. Cette caractéristique permet d'augmenter le niveau de précision souhaité sans devoir augmenter énormément le temps de calcul.

Il a aussi été constaté que la proximité des surfaces augmente le temps nécessaire à la détermination des collisions. C'est normal, car les surfaces doivent être subdivisées un plus grand nombre de fois. L'algorithme est donc plus performant lorsque les surfaces sont généralement éloignées et occasionnellement en contact. En fait, l'utilisation d'une structure de décomposition de l'espace, comme un octree, BSP ou autre, pourrait être avantageusement utilisé comme phase de tri avant d'appeler l'algorithme de détection de collisions. De cette manière, plusieurs tests inutiles pourraient être évités.

Dans le cas des tests de précision, il a été constaté que l'erreur diminue exponentiellement en fonction du niveau de récursion. C'était prévisible théoriquement, mais le fait de le constater en pratique rend l'algorithme très intéressant. Combinée avec le plateau observé sur la figure 3.2, cette propriété permet de conclure que l'erreur diminue exponentiellement alors que le temps de la requête n'augmente presque pas.

Encore une fois, on peut remarquer que certaines courbes ne suivent pas la tendance générale. En effet, les cas 2 et 3 montrent des convergences plus rapides

pour les niveaux 20 et 40. Un phénomène similaire à celui décrit auparavant est probablement la cause de ce comportement.

D'autres qualités intéressantes de l'algorithme sont:

- la possibilité de travailler avec des surfaces NURBS déformables sans coût additionnel;
- la simplicité de l'algorithme;
- le peu de mémoire nécessaire à l'exécution (quelques centaines de Ko au plus pour une précision maximale);
- la flexibilité des résultats.

Par contre, le point faible de l'algorithme est son temps d'exécution. Il est assez rapide pour certaines applications temps réel, mais pas assez pour effectuer des simulations. Si l'on compare les résultats avec des algorithmes de détection de collisions entre modèles polygonaux, on se rend compte que le temps d'une requête est beaucoup plus petit que pour l'algorithme avec des surfaces NURBS. Dépendamment des cas, les résultats peuvent être de un à trois ordres de grandeur plus rapides. Ces algorithmes ont par contre des inconvénients majeurs pour certaines applications:

- ils ne peuvent être utilisés avec des surfaces mobiles car ils requièrent une phase de pré calcul;
- ils nécessitent une très grande quantité de mémoire pour stocker les résultats du pré calcul;
- ils n'utilisent pas la définition mathématique de la surface, mais bien une approximation polygonale.

Si l'un de ces points est nécessaire à une application, l'algorithme de détection de collisions entre surfaces NURBS est donc une alternative viable, tant que ses performances restent acceptables.

CONCLUSION

Ce mémoire a tout d'abord présenté au premier chapitre une revue détaillée de la littérature scientifique traitant des modes de représentation d'objets en trois dimensions, de même que des techniques actuelles de détection de collisions reliées à chacun de ces modes. Les modèles de représentation étudiés furent les polygones, les fonctions implicites et les surfaces paramétriques.

Dans le cas des modèles polygonaux, une des méthodes de détections de collisions décrites a été les hiérarchies de volumes englobants, permettant d'accélérer la détection de collisions en construisant un arbre de volumes simples autour des objets. Il a aussi été question des algorithmes permettant de déterminer la distance entre des objets. Par contre, ces algorithmes fonctionnent généralement seulement avec des objets convexes. Les algorithmes de décomposition de l'espace, comme les octrees et les voxels ont aussi été décrits. Finalement, des méthodes récentes permettant de mettre à profit le hardware graphique pour effectuer de la détection de collisions ont été exposées.

Pour ce qui est des objets représentés à l'aide de fonctions implicites, deux algorithmes de détection de collisions ont été étudiés. Le premier est basé sur un échantillonnage de la surface et le second détermine une solution numérique au problème. Finalement, dans le cas des surfaces paramétriques, il a été question des algorithmes permettant de détecter des collisions en effectuant une subdivision des surfaces. Une autre solution implique la création d'un arbre de coquilles sphériques englobant la surface.

Le second chapitre a présenté un nouvel algorithme de détection de collisions entre surfaces NURBS. Cet algorithme a la caractéristique intéressante d'être assez rapide

pour être utilisé en temps réel et avec des surfaces pouvant se déformer dans le temps. De plus, le niveau de précision et le nombre de points de contact retournés peut être fixé par l'utilisateur, en fonction de l'application visée. En fonction de deux surfaces NURBS, l'algorithme est en mesure de retourner les positions spatiale et paramétriques d'un ou de tous les points de contact à un certain niveau de précision.

Finalement, le troisième chapitre a présenté les différents résultats obtenus en utilisant l'algorithme de détection de collisions. Afin de vérifier les performances, trois tests ont été effectués pour déterminer l'influence du niveau de récursion, du taux de collisions ainsi que de la complexité des surfaces. Pour ces trois paramètres, l'influence sur le temps de réponse a été pratiquement linéaire. Il n'y a que la courbe en fonction du niveau de récursion qui atteint un plateau, ce qui est intéressant, car il est possible d'obtenir des résultats beaucoup plus précis sans augmenter énormément le temps d'une requête.

Deux tests ont été effectués afin de mesurer le niveau de précision pouvant être obtenu par l'algorithme. Le premier test comparait les résultats théoriques de la position d'un point de contact avec les résultats expérimentaux pour deux surfaces immobiles. Le second test effectuait les même cas de test que le premier, mais avec des surfaces mobiles dont le mouvement était déterminé par le module de simulation physique présenté à l'annexe I. Dans les deux cas, il a été conclu que la précision augmentait exponentiellement en fonction du niveau de récursion, avec des résultats environ 10 fois plus précis pour les surfaces immobiles.

Un dernier test a été réalisé, cette fois-ci avec des surfaces assez complexes pour ne pas être analysables théoriquement. Une animation a été réalisée en effectuant une simulation physique de ces objets et les résultats ont permis de constater que l'algorithme semble fonctionner parfaitement dans le cas général. Il a par contre

été décevant de constater qu'il était impossible de réaliser une simulation physique en temps réel. En effet, le nombre de requêtes de détection de collisions est si élevé lors de la simulation qu'il est impossible d'afficher les résultats en temps réel.

Retour sur les Objectifs

Le but de ce projet de recherche était le développement d'un nouvel algorithme permettant de détecter des collisions entre surfaces NURBS. Les objectifs visés quant aux caractéristiques de l'algorithme étaient les suivants:

- utilisable dans une application devant faire des requête de collisions en temps réel;
- permet de travailler avec des surfaces déformables, c'est à dire pouvant changer de forme pendant la simulation;
- nécessite peu de mémoire;
- paramétrable en terme de précision: il doit être possible de choisir entre une grande précision ou une exécution rapide;
- peut retourner les coordonnées de la collision autant dans le repère du monde que dans l'espace paramétrique de chaque surface.

Les quatre derniers objectifs ont été atteint à un niveau très satisfaisant. L'algorithme ne souffre d'aucune pénalité lorsqu'utilisé avec des surfaces déformables. Il utilise une quantité très raisonnable de mémoire. Il est possible de fixer le niveau maximal de récursion, ce qui détermine directement la précision et la vitesse d'exécution. Finalement, il a été montré qu'il est possible d'obtenir les coordonnées des points de contact autant dans le repère du monde que dans l'espace paramétrique.

Pour ce qui est de l'objectif premier, celui d'être utilisable dans des applications en temps réel, son atteinte est un peu moins statisfaisante. Certes, il est utilisable pour des applications nécessitant moins d'une centaine de requêtes par seconde, mais dans plusieurs domaines, le nombre de requêtes par seconde dépasse cette quantité. Ce résultat est un peu décevant, car il aurait été intéressant de pouvoir utiliser cet algorithme dans une application de simulation de corps rigides telle que présentée en annexe. Les prochaines recherches permettront peut-être d'améliorer l'algorithme afin d'atteindre ce but.

Travaux Futurs

Les résultats obtenus sont encourageants et permettent d'imaginer plusieurs directions de recherche pouvant mener à des améliorations substantielles de l'algorithme.

Tout d'abord, il serait intéressant d'étudier d'autres types de volumes englobants au lieu des boîtes orientées présentement utilisées. Le principal avantage de ce volume est qu'il épouse bien la forme d'une surface, mais il est relativement long à construire. Il serait intéressant de comparer avec, par exemple, des k-DOP. Ces volumes sont beaucoup plus rapides à construire, la vitesse du test d'intersection est semblable, et si le niveau du k-DOP est assez élevé, ils peuvent entourer la surface de très près. Il pourrait aussi être intéressant de vérifier les résultats avec des sphères ou des boîtes alignées sur les axes. En fait, l'idéal serait de modifier l'algorithme en lui ajoutant un paramètre permettant de choisir le type de volume englobant à utiliser.

Les surfaces ont besoin d'être subdivisées avec l'algorithme d'insertion de noeuds pour faire converger l'algorithme vers une solution plus précise. Il serait donc intéressant d'optimiser cet algorithme de subdivision, car c'est un des goulots

d'étranglement de l'algorithme global. Lors de la subdivision, plusieurs opérations doivent se faire simultanément sur des vecteurs de quatre composants, il serait donc envisageable d'utiliser les nouvelles instructions SIMD des processeurs modernes. Un gain de performance important pourrait probablement être obtenu.

Le calcul de l'orientation des boîtes englobantes orientées s'effectue en trouvant simplement les vecteurs propres de la matrice de covariance des points de contrôle de la surface. Ces vecteurs permettent d'obtenir une orientation passablement bonne, mais non idéale. Il serait donc intéressant de tester si l'utilisation d'une méthode de calcul d'orientation plus précise, mais plus coûteuse en temps de calcul, en vaut la peine.

La subdivision des surfaces est toujours faite au milieu des coordonnées paramétriques. S'il était possible de déterminer une valeur paramétrique où effectuer une subdivision plus efficace, il serait possible d'augmenter de beaucoup la vitesse de convergence de l'algorithme. C'est un problème très complexe, mais qui vaudrait la peine d'être étudié plus en profondeur.

Les avenues de recherche sont donc très vastes dans ce domaine. Plusieurs autres points pourraient d'ailleurs encore être cités. La détection de collisions est un grand problème auquel de nouvelles solutions sont constamment proposées. La recherche dans ce domaine est loin d'être terminée...

RÉFÉRENCES

BACIU, G., WONG, S. et SUN, H. (1999). Recode: An image-based collision detection algorithm. *Journal of Visualization and Computer Animation, 10*, 181–192.

BARAFF, D. (1990). Curved surfaces and coherence for non-penetrating rigid body simulation. *Proceedings of SIGGRAPH*, 19–28.

BAREQUET, G. et HAR-PELED, S. (2001). Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. *Journal of Algorithms, 38*, 91–109.

BRADSHAW, G. et O'SULLIVAN, C. (2002). Sphere-tree construction using dynamic medial axis approximation. *Proceedings of the ACM SIGGRAPH Symposium on Computer animation*, 33–40.

BÉZIER, P. (1972). *Numerical Control, Mathematics and Applications.* Wiley.

CANI-GASCUEL, M. et DESBRUN, M. (1997). Animation of deformable models using implicit surfaces. *IEEE Transactions on Visualization and Computer Graphics, 3*, 39–50.

COHEN, J., LIN, M., MANOCHA, D. et PONAMGI, M. (1995). I-collide: An interactive and exact collision detection system for large-scale environments. *Proceedings of ACM International 3D Graphics Conference*, 189–196.

DINERSTEIN, J. et EGBERT, P. (2002). Practical collision detection in rendering hardware for two complex 3d polygon objects. *Proceedings of IEEE Eurographics UK Conference.*

EBERLY, D. (2000). *3D Game Engine Design.* Morgan Kaufmann.

EHMANN, S. et LIN, M. (2000). Swift: Accelerated distance computation between convex polyhedra by multi-level voronoi marching. *Proceedings of IROS.*

FUCHS, H., KEDEM, Z. et NAYLOR, B. (1980). On visible surface generation by a priori tree structures. *Proceedings of the Conference on Computer Graphics and Interactive Techniques*, 124–133.

GARCIA, G. et CORRE, J. L. (1989). A new collision detection algorithm using octree models. *IEEE/RSJ International Workgroup on Intelligent Robots and Systems*, 93–98.

GILBERT, E., JOHNSON, D. et KEERTHI, S. (1988). A fast procedure for computing the distance between complex objects in three-dimensinal space. *IEEE Journal of Robotics and Automation, 4*, 193–203.

GOTTSCHALK, S. (1996). Separating axis theorem. Rapport technique TR96-024, UNC Chapel Hill.

GOTTSCHALK, S., LIN, M. et MANOCHA, D. (1996). Obbtree: A hierarchical structure for rapid interference detection. *Proceedings of SIGGRAPH*, 171–180.

HUBBARD, P. (1993). Interactive collision detection. *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, 24–31.

HUBBARD, P. (1996). Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics, 15*, 179–210.

HUDSON, T., LIN, M., COHEN, J., GOTTSCHALK, S. et MANOCHA, D. (1997). V-collide: Accelerated collision detection for vrml. *Proceedings of VRML.*

IONES, A., ZHUKOV, S. et KRUPKIN, A. (1998). On optimality of obbs for visibility tests for frustum culling, ray shooting and collision detection. *Proceedings of Computer Graphics International*, 256–263.

JOUKHADAR, A., SCHEUER, A. et LAUGIER, C. (1999). Fast contact detection between moving deformable polyhedra. *Proceedings IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1810–1815.

KLOSOWSKI, J., HELD, M., MITCHELL, J., SOWIZRAL, H. et ZIKAN, K. (1998). Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, *4*, 21–36.

KRISHNAN, S., GOPI, M., LIN, M. et PATTEKAR, A. (1998). Rapid and accurate contact determination between spline models using shelltrees. *Proceedings of Eurographics.*

LAWLOR, O. et KALÉ, L. (1988). A voxel-based parallel collision detection algorithm. *Proceedings of the International Conference in Supercomputing*, 285–293.

LIN, M. (1993). *Efficient Collision Detection for Animation and Robotics.* Thèse de doctorat, University of California, Berkeley.

LIN, M. et CANNY, J. (1991). A fast algorithm for incremental distance calculation. *Proceedings of IEEE Conference on Robotics and Automation*, 1008–1014.

LIU, Y., NOBORIO, H. et ARIMOTO, S. (1988). Hierarchical sphere model (hsm) and its application for checking an interference between moving robots. *IEEE International Workshop on Intelligent Robots*, 801–806.

LOMBARDO, J., CANI, M. et NEYRET, F. (1999). Real-time collision detection for virtual surgery. *Computer Animation Conference.*

LORENSEN, W. et CLINE, H. (1987). Marching cubes, a high resolution 3d surface construction algorithm. *Proceedings of SIGGRAPH*, 163–169.

MELAX, S. (2001). Bsp collision detection as used in mdk2 and neverwinter nights. *Proceedings of the Game Developers Conference.*

MIRTICH, B. (1998). V-clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics, 17*, 177–208.

MOLLER, T. et HAINES, E. (2002). *Real-Time Rendering.* A.K. Peters.

NAYLOR, B., AMANATIDES, J. et THIBAULT, W. (1990). Merging bsp trees yields polyhedral set operations. *Proceedings of the Conference on Computer Graphics and Interactive Techniques*, 115–124.

PENTLAND, A. et WILLIAMS, J. (1989). Good vibrations: Modal dynamics for graphics and animation. *Proceedings of SIGGRAPH*, 215–222.

PIEGL, L. et TILLER, W. (1996). *The NURBS Book.* Springer.

PONAMGI, M., MANOCHA, D. et LIN, M. (1997). Incremental algorithms for collision detection between polygonal models. *IEEE Transactions on Visualization and Computer Graphics, 3*, 51–64.

QUINLAN, S. (1994). Efficient distance computation between non-convex objects. *Proceedings of IEEE Conference on Robotics and Automation, 4*, 3324–3329.

ROTGÉ, J. (1997). *L'arithmérique des Formes: une introduction à la logique de l'espace.* Thèse de doctorat, Université de Montréal.

SAMET, H. (1984). The quadtree and related hierarchical data structures. *Computing Surveys, 16*, 187–260.

SATO, Y., HIRATA, M., MARUYAMA, T. et ARITA, Y. (1996). Efficient collision detection using fast distance calculation algorithms for convex and non-convex objects. *Proceedings of IEEE International Conference on Robotics et Automation*, 771–778.

SHOEMAKE, K. (1994). Quaternions.

SMITH, A., KITAMURA, Y., TAKEMURA, H. et KISHINO, F. (1995). A simple and efficient method for accurate collision detection among deformable polyhedral objects in arbitrary motion. *Proceedings of IEEE Virtual Reality Annual International Symposium*, 136–145.

SNYDER, J. (1992). Interval analysis for computer graphics. *Computer Graphics*, 121–130.

SNYDER, J., WOODBURY, A., FLEISHER, K., CURRIN, B. et BARR, A. (1993). Interval methods for multi-point collisions between time-dependent curved surfaces. *Proceedings of SIGGRAPH*, 321–334.

SWAN II, J. (1993). Fast collision detection with an n-objects octree. Rapport technique OSU-ACCAD-12/93-TR7, Ohio State University.

TURK, G. (1990). Interactive collision detection for molecular graphics. Rapport technique TR90-014, UNC Chapel Hill.

TZAFESTA, C. et COIFFET, P. (1996). Real-time collision detection using spherical octrees: Virtual reality application. *IEEE International Workshop on Robot and Human Communication*, 500–506.

VAN DEN BERGEN, G. (1997a). Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, *2*, 1–13.

VAN DEN BERGEN, G. (1997b). A fast and robust gjk implementation for collision detection of convex objects. *Journal of Graphics Tools*, *2*, 1–13.

VAN DEN BERGEN, G. (2001). Proximity queries and penetration depth computation on 3d game objects. *Proceedings of Game Developers Conference.*

WATT, A. et POLICARPO, F. (2001). *3D Games, Real-time Rendering and Software Technology.* ACM Press.

WITKIN, A. et BARAFF, D. (1997). Physically based modeling: Principles and practice.

ZACHMANN, G. (1998). Rapid collision detection by dynamically aligned dop-tree. *Proceedings IEEE Virtual Reality Annual International Symposium*, 90–97.

ZHANG, D. et YUEN, M. (2000). Collision detection for clothed human animation. *Proceedings of IEEE Pacific Graphics*, 328–337.

ANNEXE I

Environnement de Validation

Pour valider la précision des résultats de l'algorithme de détection de collisions, un environnement de simulation de corps rigides a été développé. Il s'agit d'un ensemble de classes permettant de créer des corps rigides et de simuler leur déplacement en fonction des forces qui leur sont appliquées. Ce chapitre décrit le fonctionnement de cet environnement. La théorie derrière la simulation sera d'abord expliquée, puis les classes du système seront décrites. Finalement un exemple simple d'utilisation sera présenté.

I.1 Simulation Physique de Corps Rigides

Cette section présente le modèle théorique utilisé pour simuler les corps rigides. Witkin et Baraff, 1997 ont développé un modèle de simulation de corps rigides. Ce qui est présenté ici est grandement inspiré de leurs travaux.

Un corps rigide est un objet indéformable. Les lois de la physique étant plus simples lorsque les objets ne peuvent se déformer, le modèle présenté ici ne fonctionne qu'avec ce type d'objet. Pour définir complètement un corps rigide à un instant donné, il faut quelques constantes et variables d'état. Les constantes sont la masse du corps et le tenseur d'inertie. Les variables d'état sont la position, l'orientation, la vitesse linéaire et la vitesse angulaire. Les sections suivantes vont décrire plus en détails ces différentes notions, de même que les équations de la dynamique des corps rigides.

Avant de décrire ces concepts, il est nécessaire de définir deux systèmes de coordonnées. Le premier est le système du monde, toutes les variables d'état des corps rigides sont définies dans ce système, il est donc fixe pour tous les corps rigides. Le second système de coordonnées est différent pour chaque corps rigide. Son origine est toujours située au centre de masse du corps et chaque point de ce dernier a une position constante dans ce système.

Le centre de masse d'un objet peut être déterminé à l'aide de l'intégrale suivante:

$$\mathbf{C} = \frac{1}{M} \int_V \rho(\mathbf{r})\, \mathbf{r}\, d^3\mathbf{r}$$

avec $\mathbf{r} = (x, y, z)$, V le volume de l'objet et ρ sa densité.

La figure I.1 illustre ces deux systèmes d'axes. Le système du monde est représenté par les axes X_m, Y_m et Z_m, tandis que le système local du corps est représenté par les axes X_c, Y_c et Z_c.

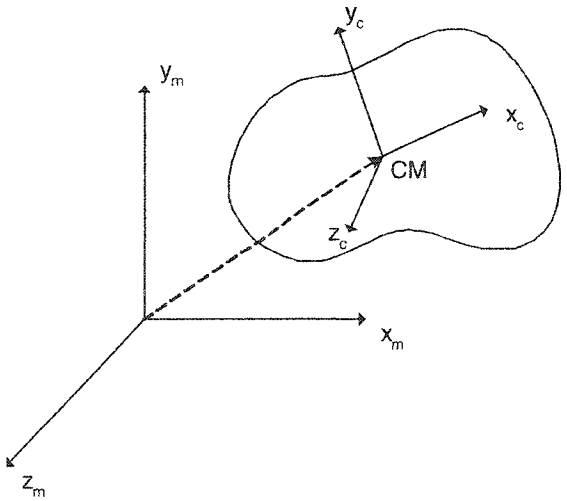

Figure I.1 Les systèmes d'axes du monde et du corps rigide.

I.1.1 Position et Orientation

La position d'un corps rigide est tout simplement la position du centre de masse de l'objet à un instant donné. C'est un point de l'espace dans le repère du monde. La position est notée avec le symbole $\mathbf{x}(t)$.

Comme un corps rigide n'est pas ponctuel, il peut avoir une certaine orientation. Il est donc nécessaire d'avoir une variable d'état permettant de la définir. L'orientation d'un objet peut être décrite de différentes manières, les plus populaires étant les angles d'Euler, les matrices de rotation et les quaternions. Le système présenté ici utilise les quaternions car ils ont plusieurs avantages. Premièrement, ils ne sont pas sujets à la perte d'un degré de liberté (Gimbal lock), comme les angles d'Euler et les matrices. De plus, ils sont représentés à l'aide de seulement 4 scalaires (x, y, z, w), contrairement aux matrices qui en nécessitent 9. Finalement, les calculs sont simplifiés en utilisant les quaternions et il est plus efficace de transformer un vecteur par une multiplication de quaternions que par une multiplication matricielle. L'orientation est notée avec le symbole $\mathbf{q}(t)$. On peut déterminer le quaternion correspondant à une rotation d'un angle θ autour de l'axe $\mathbf{v}$ à l'aide du calcul suivant:

$$\mathbf{q} = (s\mathbf{v}_x, s\mathbf{v}_y, s\mathbf{v}_z, c) \tag{I.1}$$

$$s = \sin(\theta/2) \tag{I.2}$$

$$c = \cos(\theta/2) \tag{I.3}$$

Shoemake, 1994 a décrit en détails les quaternions ainsi que les opérations que l'on peut effectuer sur ces entités.

Connaissant ces deux variables d'état d'un corps rigide, il est possible de déterminer

la position actuelle de n'importe quel point de ce corps, exprimée dans le système du monde. Si $\mathbf{p}_c$ représente la position d'un point du corps dans le système de référence lié au corps, sa position dans le système du monde est donnée par:

$$\mathbf{p}(t)_m = \mathbf{q}(t)\,\mathbf{p}_c + \mathbf{x}(t)$$

Ces deux variables permettent donc de déterminer complètement de quelle manière le corps rigide est positionné et orienté dans l'espace.

I.1.2 Vitesses Linéaire et Angulaire

En plus de la position et de l'orientation du corps, il est nécessaire de savoir à quel rythme ces variables changent dans le temps pour simuler leur mouvement. Deux nouvelles variables doivent être définies: la vitesse linéaire et la vitesse angulaire.

La vitesse linéaire est un nombre d'unités de distance par seconde. Il indique la distance que parcourrait le centre de masse du corps en une seconde si la vitesse demeurait constante pendant cette seconde. C'est un vecteur à trois composantes, une pour chaque dimension de l'espace. La vitesse est notée avec le symbole $\mathbf{v}(t)$.

La signification de la vitesse angulaire est un peu plus difficile à imaginer. C'est un vecteur à trois composantes, dont la direction indique l'axe de rotation et la longueur indique la vitesse de rotation, en radians par seconde. La vitesse angulaire est nécessaire pour décrire le mouvement des corps rigides, car si l'on prend l'exemple d'une toupie tournant sur une table sans se déplacer, on sait que sa vitesse linéaire est nulle. La seule chose pouvant expliquer son mouvement est donc la présence d'une vitesse angulaire. La vitesse angulaire est notée $\omega(t)$.

Connaissant les vitesses linéaire et angulaire d'un corps rigide, il est possible de déterminer la vitesse actuelle de n'importe quel point de ce corps, exprimée dans le système du monde. Soit $\mathbf{p}_c$, la position d'un point dans le système de référence du corps, sa vitesse dans le système du monde est donnée par:

$$\mathbf{v_p}(t) = \omega(t) \times (\, \mathbf{q}(t)\, \mathbf{p}_c \,) + \mathbf{v}(t) \tag{I.4}$$

I.1.3 Masse et Tenseur d'Inertie

La masse d'un corps rigide est la quantité de matière qui le compose. Elle peut être vue comme un indicateur de la grandeur de la force qu'il faut appliquer sur ce corps pour modifier sa vitesse d'une certaine valeur. Plus un objet a une masse importante, plus il faudra une grande force pour modifier sa vitesse. Le changement de vitesse est en fait une accélération: c'est le taux de changement de la vitesse par seconde. La deuxième loi de Newton exprime cela mathématiquement:

$$\mathbf{a}(t) = \frac{\mathbf{F}(t)}{M}$$

où $\mathbf{a}(t)$ est l'accélération du corps, $\mathbf{F}(t)$ est la force appliquée sur ce dernier et M est sa masse. On voit clairement que pour une force de grandeur fixe, l'accélération résultante sera plus grande si cette force est appliquée sur un objet de petite masse que sur un objet de grande masse. La masse est un scalaire noté avec le symbole M.

Le tenseur d'inertie est l'équivalent de la masse, mais pour les mouvements de rotation. Comme un objet n'est pas nécessairement symétrique selon les trois

axes de coordonnées, il est nécessaire de spécifier son inertie autrement que par un simple scalaire. C'est pourquoi on utilise le tenseur d'inertie: une matrice carrée de dimension 3. Lorsqu'on calcule le tenseur d'inertie, on le fait toujours par rapport à un système d'axes spécifique. Si ce système d'axes a son origine au centre de masse du corps rigide, le tenseur d'inertie est une matrice diagonale. Le tenseur d'inertie du corps est donc toujours spécifié par rapport au système de référence du corps rigide. Il est noté avec le symbole $\mathbf{I}_0$, et se calcule comme suit:

$$\mathbf{I}_0 = \begin{bmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{bmatrix}$$

où

$$\begin{aligned} I_x &= \int\int\int_V \rho(x,y,z)\,(y^2+z^2)\,dx\,dy\,dz \\ I_y &= \int\int\int_V \rho(x,y,z)\,(x^2+z^2)\,dx\,dy\,dz \\ I_z &= \int\int\int_V \rho(x,y,z)\,(x^2+y^2)\,dx\,dy\,dz \end{aligned}$$

Comme les variables d'état d'un corps rigide sont exprimées par rapport au système d'axes du monde, on doit pouvoir transformer le tenseur d'inertie exprimé dans le système local du corps pour qu'il soit valide selon l'orientation des axes du monde. Le calcul est simplement:

$$\mathbf{I}(t) = \mathbf{R}(t)\mathbf{I}_0\mathbf{R}(t)^T$$

où $\mathbf{R}(t)$ est la matrice d'orientation dérivée du quaternion $\mathbf{q}(t)$.

I.1.4 Forces et Moments de Force

Tel que mentionné précédemment, une force induit sur un corps rigide une accélération d'une grandeur inversement proportionnelle à sa masse. C'est la deuxième loi de Newton. Une force appliquée le long d'un axe passant par le centre de masse d'un objet induit une accélération linéaire. Par contre, une force appliquée le long d'un axe ne passant pas par le centre de masse du corps induit, en plus d'une accélération linéaire, une accélération angulaire. La deuxième loi de Newton pour les mouvements de rotation est la suivante:

$$\alpha(t) = \mathbf{I}(t)^{-1}\tau(t)$$

où $\alpha(t)$ est l'accélération angulaire, $\mathbf{I}(t)^{-1}$ est le tenseur d'inertie inverse et $\tau(t)$ est le moment de force. Le moment de force est défini comme suit:

$$\tau_i(t) = (\mathbf{p}_i - \mathbf{x}(t)) \times \mathbf{F}_i(t)$$

pour une force $\mathbf{F}_i(t)$ appliquée sur le corps rigide au point $\mathbf{p}_i$ dans le référentiel du monde.

Pour résoudre les équations de la dynamique des corps rigides, il est nécessaire de déterminer la force totale agissant sur le corps, de même que le moment de force total. Pour chaque corps rigide, on effectue les sommations suivantes:

$$\mathbf{F}(t) = \sum \mathbf{F}_i(t) \tag{I.5}$$

et

$$\tau(t) = \sum \tau_i(t) = \sum (\, \mathbf{p}_i - \mathbf{x}(t)\,) \times \mathbf{F}_i(t) \tag{I.6}$$

permettant ainsi d'obtenir la force résultante et le moment de force résultant.

I.1.5 Dynamique des Corps Rigides

Maintenant que les différentes variables ont été définies, il est possible de décrire les équations de la dynamique des corps rigides. Sachant que:

$$\frac{d}{dt}\mathbf{x}(t) = \mathbf{v}(t) \tag{I.7}$$

$$\frac{d}{dt}\mathbf{v}(t) = \mathbf{a}(t) \tag{I.8}$$

$$\mathbf{a}(t) = \frac{\mathbf{F}(t)}{M} \tag{I.9}$$

on détermine:

$$\mathbf{x}(t_2) = \int_{t_1}^{t_2} \mathbf{v}(t)\, dt + \mathbf{x}(t_1) \tag{I.10}$$

$$\mathbf{v}(t_2) = \frac{1}{M} \int_{t_1}^{t_2} \mathbf{F}(t)\, dt + \mathbf{v}(t_1) \tag{I.11}$$

De manière analogue, pour les mouvements de rotation, on a:

$$\frac{d}{dt}\mathbf{q}(t) = \frac{1}{2}\omega(t)\mathbf{q}(t) \tag{I.12}$$

$$\frac{d}{dt}\omega(t) = \alpha(t) \tag{I.13}$$

$$\alpha(t) = \mathbf{I}^{-1}(t)\tau(t) \tag{I.14}$$

et on peut déterminer:

$$\mathbf{q}(t_2) = \frac{1}{2}\int_{t_1}^{t_2} \omega(t)\mathbf{q}(t)\,dt + \mathbf{q}(t_1) \tag{I.15}$$

$$\omega(t_2) = \mathbf{I}^{-1}(t)\int_{t_1}^{t_2} \tau(t)\,dt + \omega(t_1) \tag{I.16}$$

(Voir Witkin et Baraff, 1997 pour le développement de l'équation I.12).

Il est donc nécessaire d'intégrer plusieurs équations afin de déterminer le nouvel état des corps rigides. Pour simuler le déplacement de ces corps par l'ordinateur, une intégration numérique de ces équations est effectuée à chaque pas de temps. Pour simplifier les calculs, l'intégrateur utilisé est de type Euler. La stabilité de cet intégrateur est limitée, il faut donc choisir soigneusement le pas de temps qui sera utilisé en fonction de la simulation.

I.2 Description du Système de Simulation

Le système de simulation physique est accessible sous la forme d'une bibliothèque de classes C++. Ces classes représentent les différentes entités nécessaires à la simulation de corps rigides. Les relations entre les différentes classes sont illustrées à la figure I.2.

Comme on peut le constater, il y a une classe représentant un corps rigide (*CRigidBody*), une interface pour les forces (*IForce*), une interface pour les objets de

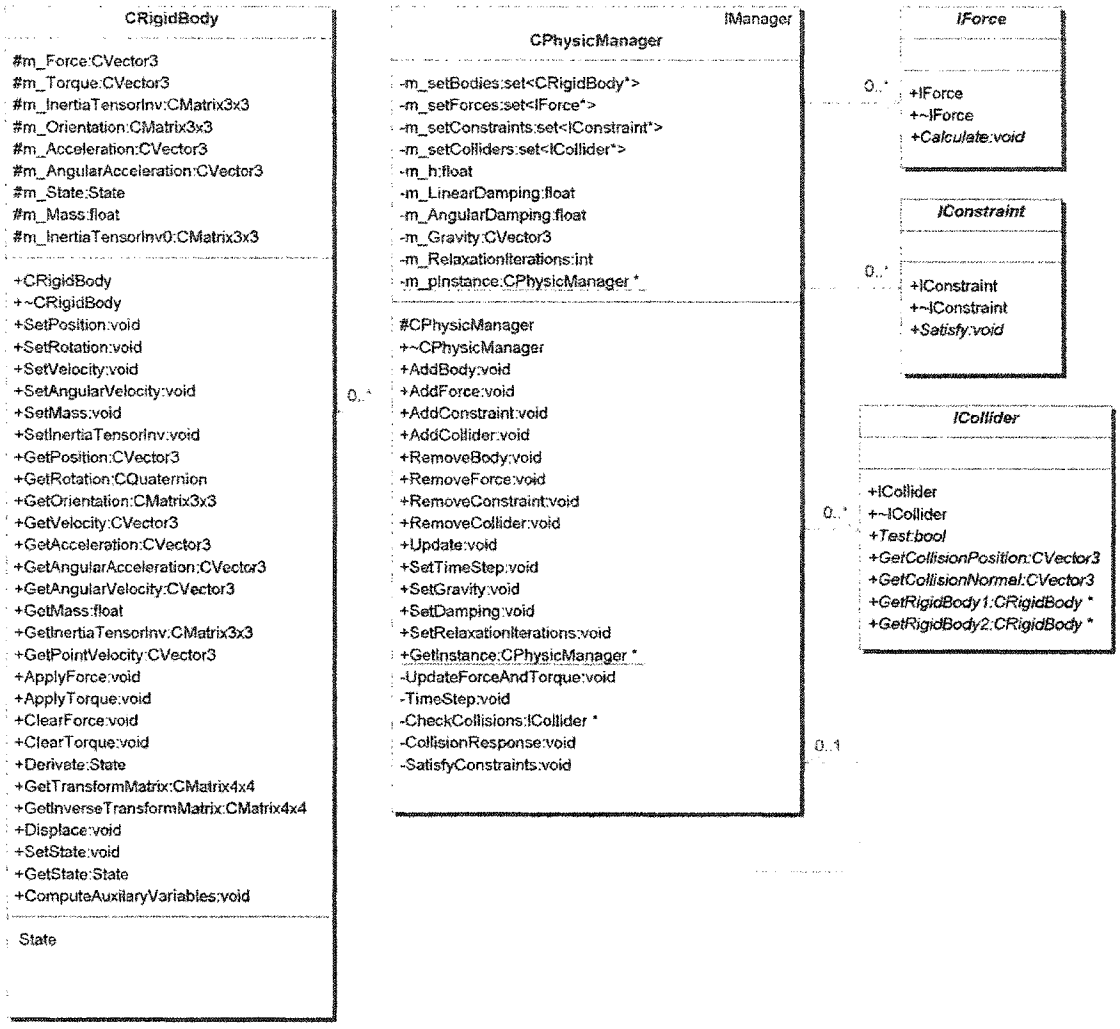

Figure I.2 Diagramme des classes du simulateur physique.

détection de collisions (*ICollider*) et une interface pour les contraintes (*IConstraint*). Il existe aussi une classe centrale (*CPhysicManager*) qui permet de gérer ces différents objets et d'effectuer la simulation en tant que telle. Les prochaines sections vont décrire plus en détail chacune de ces classes.

I.2.1 Classe *CRigidBody*

La classe *CRigidBody* encapsule les données relatives à un corps rigide. Ses attributs sont les variables d'état et les constantes dont il a été question dans la section précédente. Le tableau I.1 résume les fonctions de la classe relatives à ces variables.

Tableau I.1 Fonctions relatives aux variables d'état et constantes d'un corps rigide

Fonction	**Description**
GetPosition()	Retourne la position
SetPosition()	Définit la position
GetRotation()	Retourne l'orientation
SetRotation()	Définit l'orientation
GetVelocity()	Retourne la vitesse linéaire
SetVelocity()	Définit la vitesse linéaire
GetAngularVelocity()	Retourne la vitesse angulaire
SetAngularVelocity()	Définit la vitesse angulaire
GetState()	Retourne toutes les variables d'état
SetState()	Définit toutes les variables d'état
GetMass()	Retourne la masse
SetMass()	Definit la masse
GetInertiaTensorInv()	Retourne la matrice inverse du tenseur d'inertie
SetInertiaTensorInv()	Définit la matrice inverse du tenseur d'inertie

La classe *CRigidBody* contient aussi d'autres attributs souvent nécessaires, mais pouvant être déduits des variables d'état. Pour éviter des calculs inutiles, leur valeur est calculée une seule fois à chaque pas de temps et est sauvegardée dans une variable de la classe. Ces variables sont appelées variables auxiliaires et sont: l'orientation du corps sous forme de matrice, la matrice inverse du tenseur d'inertie selon l'orientation actuelle, l'accélération linéaire et l'accélération angulaire. Le tableau I.2 résume les fonctions relatives à ces variables.

La classe contient aussi deux accumulateurs, un pour les forces et un pour les

Tableau I.2 Fonctions relatives aux variables auxiliaires d'un corps rigide

Fonction	**Description**
GetOrientation()	Retourne la matrice 3x3 définissant l'orientation
GetInertiaTensorInv()	Retourne la matrice inverse du tenseur d'inertie selon l'orientation actuelle
GetAcceleration()	Retourne l'accélération linéaire
GetAngularAcceleration()	Retourne l'accélération angulaire
ComputeAuxilaryVariables()	Calcule les variables auxiliaires en fonctions des variables d'états (appelée par le gestionnaire de physique une seule fois par pas de temps)

moments de forces. Le tableau I.3 résume les fonctions permettant de travailler avec ces accumulateurs.

Tableau I.3 Fonctions relatives aux accumulateurs de force et moment de force d'un corps rigide

Fonction	**Description**
ApplyForce()	Ajoute une force à l'accumulateur (appelée par un objet dérivant de l'interface *IForce*)
ApplyTorque()	Ajoute un moment de force à l'accumulateur (appelée par un objet dérivant de l'interface *IForce*)
ClearForce()	Met l'accumulateur de force à **0** (appelée par le gestionnaire à chaque pas de temps avant de calculer les forces)
ClearTorque()	Met l'accumulateur de moment de force à **0** (appelée par le gestionnaire à chaque pas de temps avant de calculer les moments de forces)

Finalement, quelques fonctions utilitaires complètent la classe et sont décrites au le tableau I.4

L'utilisation d'un objet de la classe *CRigidBody* est donc très simple. Il suffit de

Tableau I.4 Autres fonctions d'un corps rigide

Fonction	Description
Derivate()	Retourne les dérivées des variables d'état
GetTransformMatrix()	Retourne la matrice de transformation permettant de transformer un point du repère du corps rigide vers le repère du monde
GetInverseTransformMatrix()	Retourne la matrice de transformation permettant de transformer un point du repère du monde vers le repère du corps rigide
GetPointVelocity()	Retourne la vitesse d'un point du corps rigide en utilisant la l'équation I.4

le créer et de définir ses constantes et son état initial en appelant les fonctions *Set()* correspondantes aux variables à définir. Il faut ensuite l'enregistrer auprès du gestionnaire de physique. Par la suite, son état peut toujours être obtenu en appelant les fonctions *Get()* voulues.

I.2.2 Interface *IForce*

L'interface *IForce* permet de définir n'importe quel type de force. Elle ne contient qu'une seule fonction: *Calculate(Time)*. Cette fonction est appelée par le gestionnaire de physique à chaque pas de temps, avant de procéder à l'intégration numérique. Une classe implémentant cette interface doit donc calculer la valeur de la force et/ou moment de force et l'appliquer sur le ou les corps rigides affectés.

Le constructeur ou une autre fonction d'une classe héritant de cette interface doit permettre de définir les paramètres de la force. Il faut que la force conserve une référence vers les corps rigides sur lesquels elle agit. Le client est libre de choisir le

mécanisme qu'il souhaite pour fournir ces références à la force. Il en est de même pour les autres paramètres de la force, s'il y a lieu.

La fonction *Calculate(Time)* prend en argument une valeur de temps. Comme c'est le gestionnaire qui appelle cette fonction, il envoie le temps actuel de la simulation. De cette manière, une force peut varier en fonction du temps. Cela peut être utile lorsqu'il est nécessaire d'implémenter une force comme du vent changeant de direction et d'amplitude avec le temps.

I.2.3 Interface *ICollider*

L'interface *ICollider* est utilisée pour effectuer de la détection de collisions. Afin d'empêcher les objets de passer les uns au travers des autres, il est nécessaire de définir un moyen permettant de détecter s'il y a une collision entre les objets. Comme la classe *CRigidBody* ne contient aucune information reliée à la forme, le moyen utilisé pour déterminer s'il y a collision ou non entre deux corps est par l'utilisation d'objets implémentant l'interface *ICollider*.

Le constructeur ou une autre fonction d'une classe héritant de cette interface doit permettre de définir les paramètres relatifs à la détection de collisions. Entre autre, elle doit conserver une référence vers le ou les corps rigides pour lesquels elle doit déterminer s'il y a une collision ou non. Elle doit aussi conserver des données permettant de connaître la forme du corps, par exemple un modèle en fils de fer ou encore des surfaces paramétriques. Le client est libre de choisir le mécanisme qu'il souhaite pour fournir ces paramètres.

La fonction principale de l'interface est la fonction *Test()*. Cette fonction retourne une valeur booléenne vraie s'il y a une collision. Elle est appelée par le gestionnaire une fois par pas de temps.

Les informations nécessaires au gestionnaire de physique lui permettant de simuler correctement le rebondissement dû à la collision sont:

- la position du point de contact;
- la normale de la surface au point de contact;
- une référence vers les deux corps rigides en contact (ou un seul s'il est entré en contact avec un objet statique).

Une classe implémentant l'interface *ICollider* est responsable de la détermination de ces informations. Les valeurs retournées doivent être valides lorsque le dernier appel à la fonction *Test()* a retourné une valeur vraie. Évidemment, les valeurs retournées peuvent être indéterminées si le dernier appel à *Test()* n'a pas détecté de collision. Les fonctions relatives à ces attributs sont résumées dans le tableau I.5.

Tableau I.5 Fonctions relatives aux attributs d'un objet de détection de collisions

Fonction	**Description**
GetCollisionPoint()	Retourne la position du point de contact, dans le référentiel du monde
GetCollisionNormal()	Retourne la normale de la surface au point de contact, dans le référentiel du monde
GetRigidBody1()	Retourne un pointeur vers le premier corps rigide entrant en collision
GetRigidBody2()	Retourne un pointeur vers le second corps rigide entrant en collision avec le premier (si le premier est entré en contact avec un objet statique, cette fonction retourne un pointeur NULL)

I.2.4 Interface *IConstraint*

L'interface *IConstraint* permet de spécifier des contraintes sur le déplacement des objets. Par exemple, il est possible de contraindre un corps pour qu'il ait toujours un point à la même position qu'un point d'un autre corps, créant ainsi un joint sphérique. Il est possible d'imaginer d'autres types de contraintes, comme obliger un corps à rester sur un plan, sur une droite ou autre.

La technique utilisée pour satisfaire les contraintes est très simple. Il aurait été possible de créer un système d'équations linéaires en fonction des équations des contraintes et de le résoudre à chaque pas de temps, mais une autre solution a été retenue. L'idée est que chaque contrainte est responsable de déplacer un corps violant une contrainte vers l'endroit valide le plus près. Évidemment, si un corps viole deux contraintes au même pas de temps, les déplacements engendrés pour satisfaire ces deux contraintes peuvent être contradictoires. C'est pourquoi une boucle de relaxation est utilisée, qui essaie de satisfaire toutes les contraintes à chaque itération. Cette boucle a tendance à converger rapidement vers une solution du système de contraintes, si bien que seulement 5 à 10 itérations sont généralement nécessaires pour obtenir une solution acceptable. Cette méthode est moins précise que la résolution exacte du système, mais elle est beaucoup plus rapide.

L'interface *IConstraint* sert à définir des classes contraignant le mouvement des corps rigides. La seule fonction de cette interface est *Satisfy(Time)*. Cette fonction est appelée par le gestionnaire de physique à chaque itération de la boucle de relaxation. Un objet implémentant cette interface est donc responsable de déplacer les corps rigides qu'il affecte et qui violent la contrainte vers la position/orientation la plus près qui respecte la contrainte.

Le constructeur ou une autre fonction d'une classe héritant de cette interface doit

permettre de définir les paramètres de la contrainte. Il faut que la contrainte conserve une référence vers les corps rigides sur lesquels elle agit. Le client est libre de choisir le mécanisme qu'il souhaite pour fournir ces paramètres.

Tout comme la fonction *Calculate(Time)* de l'interface *IForce*, la fonction *Satisfy(Time)* prend en argument une valeur de temps. Le gestionnaire appelle cette fonction en lui passant le temps actuel de la simulation. Une contrainte peut donc varier en fonction du temps, comme par exemple une porte qui s'ouvre.

I.2.5 Classe *CPhysicManager*

La classe *CPhysicManager* est le gestionnaire de physique. C'est une classe singleton, c'est à dire qu'elle ne peut exister qu'en une seule instance dans un même programme. Elle a la responsabilité de gérer les objets dont il a été question dans les sections précédentes. C'est cette classe qui effectue la simulation physique en tant que telle: elle calcule et intègre les forces pour mettre à jour l'état des corps rigides, elle vérifie s'il y a des collisions, et si oui, fait rebondir les corps affectés et finalement, elle effectue la boucle de relaxation pour satisfaire les contraintes.

La classe *CPhysicManager* contient des fonctions permettant d'ajouter et d'enlever les différents objets du système de simulation physique. Ces fonctions sont résumée dans le tableau I.6.

Il existe quelques variables globales à la simulation. La première est l'accélération gravitationnelle. C'est une accélération qui affecte tous les corps rigides d'une simulation. Il aurait été possible de créer un objet implémentant l'interface *IForce* pour simuler l'effet de la gravité, mais comme c'est une force présente dans la plupart des simulations et qui affecte tous les corps, elle est plutôt définie comme un paramètre global.

Tableau I.6 Fonctions du gestionnaire de physique pour l'ajout et la suppression d'objets

Fonction	Description
AddBody()	Ajoute un corps rigide au gestionnaire
RemoveBody()	Enlève un corps rigide du gestionnaire
AddForce()	Ajoute une force au gestionnaire
RemoveForce()	Enlève une force du gestionnaire
AddCollider()	Ajoute un objet de détection de collisions au gestionnaire
RemoveCollider()	Enlève un objet de détection de collisions du gestionnaire
AddConstraint()	Ajoute une contrainte au gestionnaire
RemoveConstraint()	Enlève une contrainte du gestionnaire

La seconde variable est l'amortissement. C'est une force qui tend à immobiliser les corps. Cette force est toujours appliquée dans une direction opposée à la vitesse actuelle de l'objet et avec une grandeur proportionnelle à cette vitesse. La variable indique quel est le facteur utilisé pour multiplier la vitesse et ainsi obtenir la force d'amortissement. Plus ce facteur est grand, plus la force d'amortissement sera grande et plus les objets auront tendance à s'immobiliser rapidement lorsqu'aucune autre force n'agit sur eux. Il est possible de spécifier un facteur d'amortissement différent pour les vitesses linéaire et angulaire. Il est généralement conseillé d'avoir un amortissement non nul, car cela augmente la stabilité de la simulation.

Une autre variable du gestionnaire pouvant être spécifiée est la valeur du pas de temps maximal utilisé pour l'intégration numérique. Plus cette valeur est petite, plus la précision de la simulation sera grande, mais plus le temps nécessaire pour effectuer les calculs sera grand. Selon le type de simulation, différentes valeurs peuvent être spécifiées.

Finalement, la dernière variable est le nombre d'itérations de la boucle de relax-

ation. Plus sa valeur est élevée, plus les contraintes seront satisfaites correctement, mais plus la simulation sera lente, et vice-versa. Pour plus de détails sur la boucle de relaxation, voir la section I.2.4.

Le tableau I.7 résume les fonctions permettant de définir les variables globales de la simulation.

Tableau I.7 Fonctions du gestionnaire de physique pour définir les variables globales de la simulation

Fonction	**Description**
SetGravity()	Définit le vecteur d'accélération gravitationnelle
SetDamping()	Définit les facteurs d'amortissement linéraire et angulaire
SetTimeStep()	Définit la valeur du pas de temps maximal de la simulation
SetRelaxationIterations()	Définit le nombre d'itérations de la boucle de relaxation

Finalement, la fonction *Update(Time1, Time2)* est le coeur du gestionnaire de physique. Elle prend en argument deux valeurs de temps, la première est le temps auquel débute la simulation et la seconde est le temps de fin de la simulation. La première chose que fait cette fonction, c'est de calculer la différence (Δt) entre les deux valeurs de temps. Si ce Δt est plus petit que le pas de temps maximal spécifié, on utilise Δt comme pas de temps. Sinon, on divise l'intervalle de temps en plusieurs morceaux ne dépassant pas le pas de temps maximal.

Ensuite, pour chaque pas de temps, le gestionnaire met à **0** les accumulateurs de force et de moment de force de tous les corps rigides. Il appelle par la suite la fonction *Calculate()* de toutes les forces du système. Les accumulateurs des corps rigides contiennent à cet instant la force et le moment de force résultant. C'est en

fait l'implémentation des équations I.5 et I.6.

L'étape suivante d'un pas de temps est l'intégration. Les états de tous les corps rigides sont mis à jour en effectuant une intégration numérique avec la méthode d'Euler. Les position, orientation, vitesse linéaire et vitesse angulaire de chaque corps sont mises à jour en résolvant les équations I.10, I.11, I.15 et I.16.

Après l'intégration, le gestionnaire effectue la boucle de relaxation. La fonction *Satisfy()* de toutes les contraintes est appelée à chaque itération de la boucle.

Finalement, la dernière étape d'un pas de temps est la détection de collisions. Le gestionnaire appelle la fonction *Test()* de tous les objets de détection de collisions. Si un test est positif, le gestionnaire effectue une recherche binaire afin de déterminer plus précisément le temps de la collision. Cette recherche est assez coûteuse en temps de calcul, car il faut diviser le pas de temps et refaire l'intégration et le test de collision plusieurs fois. Lorsque que le temps de contact plus précis a été déterminé, les attributs de la collision sont utilisés pour simuler le rebondissement de l'objet. L'algorithme I.1 décrit l'ensemble de la fonction *Update()*.

Cela conclut la description du module de simulation physique. Les diverses classes du système permettent une grande souplesse d'utilisation en laissant l'utilisateur libre d'implémenter les types de forces, contraintes et modèles de collision qu'il désire. Lorsque ces implémentations sont rapides, ce simulateur peut facilement être utilisé dans des applications en temps réel.

I.3 Exemple d'Application

Afin de montrer comment utiliser le module physique, cette section décrit un exemple simple d'application. Le but est de simuler le mouvement de deux corps

Algorithme I.1 Update(Time1, Time2)

```
Δt ← = Time2 − Time1
if Δt < MaxTimeStep then
  Iterations ← 1
  h ← Δt
else
  Iterations ← ⌈Δt/ MaxTimeStep ⌉
  h ← Δt/ Iterations
end if
for i = 1..Iterations do
  for all Forces do
    Calcul de la force et du moment de force
  end for
  for all Corps do
    Intégration des forces et moments de force
  end for
  for j = 1..Relaxation do
    for all Contraintes do
      Calcul de la contrainte
    end for
  end for
  for all Objets de collision do
    Détection de collision et calcul des impulsions
  end for
end for
```

rigides reliés par un ressort. Pour y arriver, il est nécessaire de définir une classe de force simulant un ressort et de créer deux corps rigides.

La classe de la force de ressort implémente évidemment l'interface *IForce*. Les attributs de la force seront spécifiés en les passant au constructeur de la force. Ils sont:

- un pointeur vers les deux corps rigides affectés;
- la longueur du ressort (ℓ);
- la constante de rappel du ressort (k_r);
- la constante d'amortissement du ressort (k_a);
- le point où est attaché le ressort sur chaque corps, dans le système de référence de chaque corps ($\mathbf{p}_{c1}$ et $\mathbf{p}_{c2}$).

Comme on peut le voir, le ressort ne sera pas nécessairement attaché au centre de masse des corps. Il y aura donc création non seulement d'une force sur chacun, mais aussi d'un moment de force. Il faut donc implémenter la fonction *Calculate()* de cette force en fonction de tous ces paramètres.

Premièrement, l'équation permettant de trouver la force générée par un ressort amorti est la suivante:

$$\mathbf{F} = \left(k_r(|\Delta\mathbf{p}| - \ell) - k_a \frac{\Delta\mathbf{v} \cdot \Delta\mathbf{p}}{|\Delta\mathbf{p}|} \right) \frac{\Delta\mathbf{p}}{|\Delta\mathbf{p}|}$$

où $\Delta\mathbf{p} = \mathbf{p}_{m2} - \mathbf{p}_{m1}$, ces points étant les points $\mathbf{p}_{c1}$ et $\mathbf{p}_{c2}$ transformés du système de références des corps rigides vers le système du monde en utilisant la fonction

GetTransformMatrix() des corps rigides. Quant à $\Delta\mathbf{v}$, c'est la différence entre les vitesses des points $\mathbf{p}_{c2}$ et $\mathbf{p}_{c1}$, pouvant être obtenues avec la fonction *PointVelocity()* des corps rigides. Cette force est valide pour le premier corps, la force appliquée sur le second est simplement de signe inverse.

Cette force doit être appliquée au point où est attaché le ressort. La fonction *Calculate()* de la force doit donc aussi calculer un moment de force. Ce moment est différent pour chaque corps rigide et dépend du point où est attaché le ressort. On peut les calculer comme suit:

$$\begin{aligned} \tau_1 &= (\mathbf{p}_{m1} - \mathbf{x_1}) \times \mathbf{F} \\ \tau_2 &= (\mathbf{p}_{m2} - \mathbf{x_2}) \times (-\mathbf{F}) \end{aligned}$$

où $\mathbf{x_1}$ et $\mathbf{x_2}$ sont les positions du centre de masse des corps rigides respectifs. Il ne reste plus qu'à appeler les fonctions *ApplyForce()* et *ApplyTorque()* des corps rigides pour appliquer les forces et moments de force.

Le programme principal effectuant cette simulation doit créer au départ deux corps rigides. Il doit ensuite définir leur masse et tenseur d'inertie, ainsi que leur état initial. L'étape suivante est la création d'une force de ressort avec les paramètres voulus, en lui passant comme pointeurs de corps rigide l'adresse de ceux qui viennent d'être créés. Il faut finalement ajouter ces deux corps et cette force au gestionnaire de physique et définir la valeur des variables globales de la simulation (pas de temps, accélération gravitationnelle, etc.)

La simulation peut maintenant commencer. Il suffit d'appeler la fonction *Update()* du gestionnaire de physique en lui passant l'intervalle de temps d'une étape de

simulation. Après chacune de ces mises à jour, on peut obtenir l'état des corps rigides en appelant les différentes fonctions de la classe *CRigidBody*.

Il est très simple d'afficher la simulation à chaque mise à jour avec une librairie de fonctions graphiques. Il suffit de spécifier la matrice de transformation du modèle comme étant la matrice retournée par la fonction *GetTransformMatrix()* et d'afficher un modèle représentant le corps rigide. Bien sûr, pour que le tout fonctionne correctement, il faut que le modèle soit défini dans un système d'axes dont l'origine est située au centre de masse et il faut aussi calculer le tenseur d'inertie selon ce système.

Tableau I.8 Caractéristiques des systèmes masses-ressort des différentes simulations.

Système	$\mathbf{k_r}$	$\mathbf{k_a}$	ℓ	**Points d'attache du ressort**
1	5.0	0.0	1.2	$(0.5, 0.0, 0.0), (-0.5, 0.0, 0.0)$
2	0.5	0.0	0.8	$(0.5, 0.0, 0.0), (-0.5, 0.0, 0.0)$
3	5.0	0.5	0.8	$(0.5, 0.0, 0.0), (-0.5, 0.0, 0.0)$
4	5.0	0.0	0.8	$(0.5, 0.1, 0.0), (-0.5, 0.3, 0.0)$

La figure I.3 montre des séquences d'images pour quatre simulations de systèmes masses-ressort. Pour ces quatre simulations, les masses sont des cubes de dimension 1, ayant initialement leur centre de gravité aux coordonnées $(-1, 0, 0)$ et $(1, 0, 0)$. La masse de chaque ressort est de un et les tenseurs d'inertie sont des matrices identités. Les autres paramètres des simulations sont énumérés dans le tableau I.8.

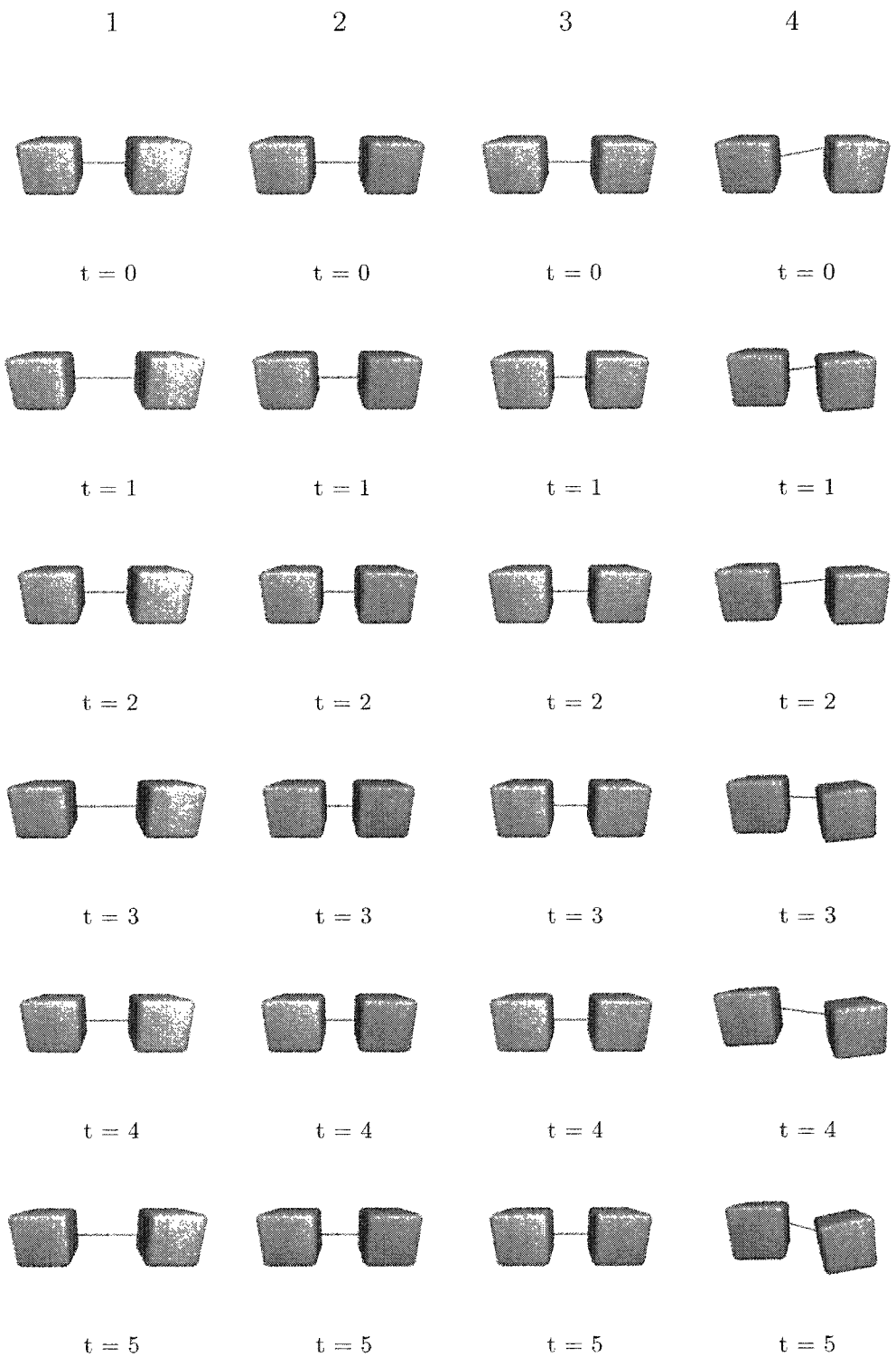

Figure I.3 Séquences d'images pour différentes configurations de systèmes masses-ressort.