UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUE-INFORMATIQUE APPLIQUÉES

PAR
JULIEN GUILLEM-LESSARD

GESTION D'ÉVÉNEMENTS ET D'INTERACTIONS DANS UN ENVIRONNEMENT 3D

Décembre 2006

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

L'auteur de ce mémoire ou de cette thèse a autorisé l'Université du Québec à Trois-Rivières à diffuser, à des fins non lucratives, une copie de son mémoire ou de sa thèse.

Cette diffusion n'entraîne pas une renonciation de la part de l'auteur à ses droits de propriété intellectuelle, incluant le droit d'auteur, sur ce mémoire ou cette thèse. Notamment, la reproduction ou la publication de la totalité ou d'une partie importante de ce mémoire ou de cette thèse requiert son autorisation.

Résumé

L'industrie du jeu vidéo est en pleine expansion depuis quelques années, totalisant un revenu annuel de 7.3 milliards de dollars américain en 2004. Cette industrie requiert des connaissances avancées en programmation.

Différents algorithmes et optimisations sont nécessaires afin de développer ce type d'applications informatiques. Entre autres, la détection de collisions et la gestion d'événements sont des concepts complexes mais nécessaires au développement d'un jeu vidéo.

Ce document présente plusieurs techniques et plusieurs optimisations pour résoudre ces concepts en un temps d'exécution raisonnable. La détection de collisions est l'algorithme le plus lourd. Toutes les approches présentées utilisent une modélisation polygonale ou la géométrie constructive. Plusieurs algorithmes publics sont également expliqués : I-Collide, Rapid, V-Collide, V-Clip et SOLID.

D'autres concepts sont également nécessaires au développement d'un jeu vidéo, comme le graphisme 3D et l'interface de l'utilisateur, et seront abordés dans ce document.

La seconde partie du document porte sur un logiciel développé afin de mettre en pratique les différents algorithmes et optimisations présentés dans le premier chapitre. La détection de collisions utilisée dans ce logiciel est réalisée à l'aide d'un algorithme hybride innovateur basé sur les différentes techniques expliquées dans la première partie du document. Les méthodes graphiques utilisées sont également présentées.

Finalement, dans le dernier chapitre, nous analysons les résultats du logiciel développé : Arka3D.

Abstract

Video games became an important market generating around 7.3 billions US dollars in 2004. Video games require high level programming.

Many algorithms and optimizations are needed to develop this kind of software. Collisions detection and events handling are complex and necessary concepts that a video game must used.

This paper provides many techniques and optimizations to solve complex algorithms in decent computational time. Collisions detection is the most complex of these algorithms. All approaches are based on polygonal models or constructive geometry. Several public algorithms are also presented: I-Collide, Rapid, V-Collide, V-Clip and SOLID.

Some video game concepts, like computer graphic techniques and gaming interfaces, are also explained to complete the first part of this paper.

In the second part of this paper, we present a video game that we developed to experiment these algorithms. Collisions detection used is a hybrid algorithm based on algorithms explained in the first part of this paper. Graphic techniques are also presented.

Finally, we analyse the results of the developed software: Arka3D.

Table des matières

Résumé **2**
Abstract **3**
Introduction **7**
Chapitre 1 : Revue de littérature **9**
1.1 Introduction 9
1.2 Modélisation : Représentation des objets 10
1.2.1 Introduction 10
1.2.2 Modèles polygonaux 11
1.1.3 Constructive solid geometry (CSG) 13
1.2.4 Conclusion 14
1.3 Type d'environnement 16
1.3.1 Introduction 16
1.3.2 Type de requête 16
1.3.3 Deux ou N objets 17
1.3.4 Mouvement statique ou dynamique 18
1.3.5 Modèle rigide ou déformable 18
1.3.6 Conclusion 19
1.4 Optimisation des détections de collisions 20
1.4.1 Introduction 20
1.4.2 Simplification des objets 21
1.4.3 Limite de collision autour des objets 23
1.4.4 Subdivision des objets 24
1.4.5 Division de l'espace 26
1.4.6 Déterminer la distance entre deux objets 27
1.3.7 Interférence par balayage 27
1.4.8 Interférence multiple 30
1.4.9 Cohérence temporelle 32
1.4.10 Conclusion 33
1.5 Gestion de forces et d'événements 34
1.5.1 Introduction 34
1.5.2 Les champs de forces 34
1.5.3 Attraction et évitement 35
1.5.4 Rebondissements et Déviations 36
1.5.5 Conclusion 38
1.6 Algorithmes existants 38
1.6.1 Introduction 38
1.6.2 I-Collide 39
1.6.3 Rapid 42
1.6.4 V-Collide 45
1.6.5 V-Clip 47
1.5.6 SOLID 49
1.6.7 Comparaisons entre les algorithmes 50
1.6.8 Conclusion 55

1.7 Conclusion 56
Chapitre 2 : Modélisation 58
2.1 Introduction 58
2.2 Application développée (Explication du jeu) 59
2.2.1 Introduction 59
2.2.2 Fonctionnement du jeu 60
2.2.3 Capacité et contrainte 62
2.2.4 Technique de développement 63
2.2.5 Conclusion 65
2.3 Modélisation des objets 66
2.3.1 Introduction 66
2.3.2 L'objet Arka3D 66
2.3.3 L'objet Murs 69
2.3.4 L'objet Balles 70
2.3.5 L'objet Disque 72
2.3.6 L'objet Cubes 74
2.3.7 L'objet Débris 77
2.3.8 L'objet Explosions 78
2.3.9 Conclusion 80
2.4 Détection de collisions 81
2.4.1 Introduction 81
2.4.2 Détection de collision entre 2 sphères 81
2.4.3 Collision sur les murs 83
2.4.4 Collision sur le disque 85
2.4.5 Collision avec les cubes 87
2.4.6 Conclusion 90
2.5 Gestion d'événements 91
2.5.1 Introduction 91
2.5.2 Rebondissement 92
2.5.3 Déviation 96
2.5.4 Destruction 97
2.5.5 Objets parentés 97
2.5.6 Conclusion 98
2.6 Effets visuels 99
2.6.1 Introduction 99
2.6.2 Murs Translucides 99
2.6.3 Effets de lumière sur les murs 100
2.6.4 Séquence de textures 101
2.6.5 Explosions et débris 102
2.6.6 Disque opaque ou translucide 102
2.6.7 Points de vues multiples 103
2.6.8 Conclusion 106
2.7 Erreurs de détection de collisions et de gestion d'événements. 107
2.8 Conclusion 108
Chapitre 3 : Expérimentation et discussion 110
3.1 Introduction 110

3.2 Expérimentation 110
3.2.1 Introduction 110
3.2.2 Scénario 1 : Niveau simple 111
3.2.3 Scénario 2 : Niveau intermédiaire 113
3.2.4 Scénario 3 : Niveau difficile 115
3.2.5 Scénario 4 : Test sur la division d'espace 118
3.2.6 Scénario 5 : Test sur la cohérence temporelle 119
3.2.7 Scénario 6 : Test sur l'élimination de facettes à l'aide de la distance 121
3.2.8 Conclusion 122
3.3 Discussion 123
3.3.1 Introduction 123
3.3.2 Comparaison entre les environnements 123
3.3.3 Comparaison entre les routines 125
3.3.4 Répartition des temps pour la détection de collisions 126
3.3.5 Comparaison des facettes présélectionnées et traitées 128
3.3.6 Comparaison des erreurs 131
3.3.7 Conclusion 132
3.4 Conclusion 133
4 Conclusion 134
Annexes 135
Annexe A : Résumé de Direct X 135
Qu'est-ce que DirectX? 135
DirectX Graphics 136
DirectInput 137
DirectSound et DirectMusic 138
DirectPlay 138
DirectShow 139
Conclusion 141
Annexe B : Résumé des versions de Arka3D 142
Références 146
Liste des figures 149
Liste des tableaux 154
Liste des équations 155

Introduction

L'informatique offre une panoplie d'applications dans un grand nombre de domaines. La simulation d'environnement est une de ces applications qui est en pleine expansion.

La simulation d'environnement permet de représenter logiquement une approximation d'un environnement réel afin de déduire des événements futurs. Dans ces environnements, plusieurs objets sont présents et interagissent ensemble. Plusieurs événements découlent de ces interactions et devront être déduits logiquement. C'est principalement sur la déduction de ces différentes interactions et de ces événements que porte ce document.

Trois types de champs d'application sont reliés à cette simulation :

- Le domaine scientifique, où l'on tente de déduire avec la plus grande précision possible les événements futurs afin d'en tirer des conclusions certaines. Ici, la précision est très importante et le temps de réponse requis peut souvent être très grand.

- Le domaine de l'animation, où l'on tente de faire ressembler l'environnement virtuel à l'environnement réel sans qu'il soit parfaitement conforme aux lois de la physique. C'est le cas des films d'animation (comme ceux de Disney ou Pixar). Cette fois-ci, la précision est un peu moins importante et le temps requis au calcul logique de l'environnement doit être plus raisonnable.

- Le domaine du temps réel, où l'on tente de créer un environnement virtuel le plus réaliste possible avec une contrainte de temps qui est souvent de l'ordre de la fraction de seconde. C'est le cas des jeux vidéo et de certains logiciels d'entraînement utilisés par l'armée ou la Nasa (des simulateurs de vol par exemple). Ici, on tente de simuler un environnement réaliste dans un temps très minime.

Ce mémoire se penchera surtout sur le domaine du temps réel où l'on tentera d'optimiser au maximum le temps d'exécution d'un jeu vidéo conçu dans le cadre de cette maîtrise. La principale problématique de ce domaine est la complexité des algorithmes de détection de collisions entre les objets graphiques de la scène. Ceux-ci doivent être optimisés au maximum afin de permettre un temps d'exécution et une précision acceptable.

La première partie de ce document présentera une revue de littérature sur les modélisations numériques des objets, sur les environnements et les nécessités de ceux-ci, sur la gestion d'événements tels les rebondissement et

aussi principalement sur les optimisations des algorithmes de détection de collisions. De plus, certains algorithmes de détection de collisions complets seront examinés.

La seconde partie présente le logiciel Arka3D contenant un algorithme de détection de collisions adapté et innovateur permettant de résoudre de manière linéaire la détection de collisions et la gestion d'événements. La technique de programmation et les différentes optimisations y seront décrites.

Finalement, au chapitre 3, nous présenterons les résultats et examineront l'efficacité des différentes optimisations. Ces résultats montrent la grande efficacité des algorithmes dans des environnements possédant un très grand nombre d'objets graphiques.

Chapitre 1 : Revue de littérature

1.1 Introduction

Ce chapitre permet de faire un survol de la littérature au niveau des techniques de détection de collisions dans des environnements 3D. Pour bien comprendre tous les aspects théoriques et techniques des approches de détection de collisions, il faut au préalable exposer les méthodes de modélisation d'objets 3D puisqu'elles conditionnent grandement les algorithmes de détection de collisions.

Cette modélisation est directement liée au type d'environnement 3D dans lequel elle évolue. Ces environnements peuvent, par exemple, être en temps réel. Les caractéristiques des environnements sont primordiales parce quelles influencent grandement l'efficacité des différents algorithmes de détection de collisions. Le nombre d'objets 3D, leur vitesse de déplacement et de rotation ainsi que la densité de ces objets dans l'environnement sont des facteurs qui doivent influencer le choix d'un algorithme de détection de collisions.

Dans un premier temps, nous présenterons les différents types de modélisation des objets graphiques utilisés en détection de collisions. Ensuite, des types d'environnement 3D seront présentés. Ces deux premières sections sont essentielles pour le choix de bonnes techniques de détection de collisions.

Par la suite, nous décrirons différentes techniques permettant l'optimisation des détections de collisions. Ces techniques permettront d'améliorer grandement le temps d'exécution en éliminant plusieurs calculs. En combinant ces techniques, il sera possible de créer des algorithmes de détection de collisions très efficaces.

Suite à une détection de collisions positive, l'environnement devra réagir à cette collision afin de simuler les effets de celle-ci. Les forces de gravité, d'attraction et de rebondissement sont donc traitées dans la cinquième section de ce chapitre.

Dans la dernière section, nous examinerons différents algorithmes de détection de collisions créés par différents groupes de recherche qui utilisent les techniques présentées dans les sections précédentes. Ces algorithmes seront ensuite comparés dans un contexte de planification de trajectoire.

Cette revue de littérature survole donc les différents concepts liés à la gestion d'événements dans un environnement 3D.

1.2 Modélisation : Représentation des objets

1.2.1 Introduction

La première phase du développement d'applications 3D consiste à modéliser numériquement les objets et les environnements. La modélisation permet de représenter la forme 3D de chaque objet ainsi que les interactions physiques entre ces objets en mouvement dans l'environnement. Les objets sont rarement des cubes parfaits ou des sphères, ils possèdent des formes très variées et des imperfections. Les modèles seront une approximation des objets réels puisqu'il est impossible de traiter toutes les caractéristiques physiques de ces objets rapidement.

Les modèles sont également utilisés au niveau de l'affichage graphique. Ceux-ci peuvent être plus complexes car ils sont uniquement utilisés pour le rendu à l'écran. Les cartes graphiques sont conçues de façon efficaces pour l'affichage des modèles très complexes afin d'obtenir le meilleur rendu d'images possible. Cet affichage est également bien optimisé à la fois par les cartes graphiques et par les langages de programmation comme DirectX (Voir Annexe A). Les cartes graphiques fonctionnent, actuellement, uniquement avec les modèles polygonaux.

Les modèles complexes sont, par contre, à éviter quand à la détection de collisions puisque la lourdeur des calculs de celui-ci est beaucoup plus grande. Il est donc préférable d'utiliser une modélisation distincte pour accomplir la détection de collision qui sera plus adaptée à celle-ci. Un modèle 3D des objets servira alors à la qualité graphique de l'affichage en étant utilisé par la carte graphique [1] et un autre modèle du même objet sera utilisé pour la schématisation des détections de collisions. L'utilisation d'un modèle simplifié pour optimiser l'algorithme de collisions n'aura donc aucun impacte sur la qualité visuelle de l'image.

Il existe donc plusieurs techniques de modélisation des objets afin d'améliorer le temps de calcul de la détection de collisions. Les modèles sont divisés en deux catégories soient les modèles polygonaux et ceux non polygonaux (Figure 1.1).

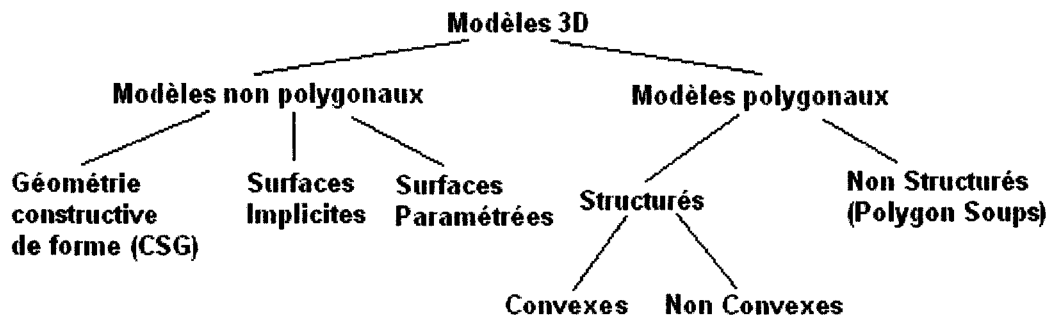


Figure 1.1. Classification des modèles 3D.

Les modèles polygonaux sont largement utilisés puisqu'ils sont plus flexibles et ils sont conformes aux normes d'affichage graphique. Dans ces modèles, les objets 3D sont représentés par un ensemble de polygones. Ces polygones sont toujours des triangles représentés par trois vecteurs 3D.

Même si les modèles non polygonaux sont peu utilisés pour la détection de collisions, ils le sont malgré tout uniquement dans ce domaine. Les modèles non polygonaux sont encore très peu développés au niveau de la détection de collisions et ils ne sont pas supportés par les cartes graphiques pour l'affichage. Par contre, ces modèles (non polygonaux) pourraient être beaucoup plus efficaces dans la détection les collisions puisqu'ils supportent les surfaces courbes.

Voici les modèles qui seront examinés en détail dans ce document :

- Modèles polygonaux
- Constructive solid geometry (CSG)

Dans les sections suivantes une description sommaire des méthodes de représentation d'objets 3D est présentée et ce dans un contexte de détection de collisions.

1.2.2 Modèles polygonaux

Les modèles polygonaux sont largement utilisés dans l'affichage graphique. Ils sont versatiles et grandement optimisés par les cartes graphiques.

Étant donné que les modèles de base représentant les objets sont souvent sous la forme polygonale, la majorité des algorithmes de détection de collisions utilisent ces modèles.

La structure la plus utilisée est l'ensemble de polygones ou « polygon soup ». Cette structure contient une collection de polygones non reliés entre eux et pouvant former un ou plusieurs objets. Il est également possible de structurer les polygones de manière à former des objets fermés qui ont une surface extérieure et une surface intérieure.

Les objets 3D étant formés par des ensembles de polygones sont soit complètement convexes ou non convexes. Les modèles convexes possèdent des propriétés qui permettront de simplifier grandement les calculs de distance entre deux objets et également la pénétration entre paires d'objets. Les modèles non convexes sont beaucoup plus problématiques à cause principalement de leur forme irrégulière. De plus, les algorithmes de détection de collisions ne fonctionnent pas tous sur ce type de modèle.

La figure 1.2 présente un exemple d'un modèle convexe et d'un modèle non convexe.

Figure 1.2. Modèle convexe et modèle non convexe.

Dans le cas d'un modèle non convexe, il est souvent possible de diviser l'objet en parties convexes afin d'éliminer les parties non convexes. C'est une stratégie largement utilisée [2]. Par contre, certains modèles requièrent un temps de calcul très long ou sont impossible à diviser ce qui à pour effet de ralentir considérablement les algorithmes de détection de collisions. La Figure 1.3 présente l'exemple de la division du beigne (« Torus ») qui est impossible à diviser en parties convexes.

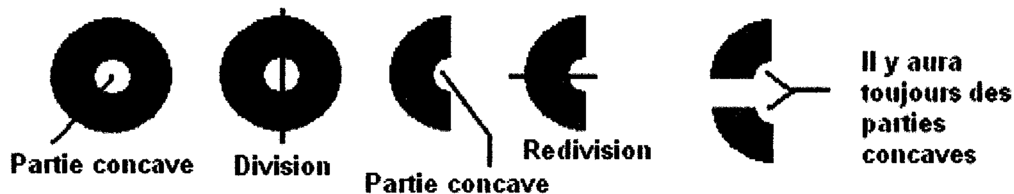


Figure 1.3. Division d'un beigne.

La décomposition des modèles devra également être optimisée afin d'avoir le moins de parties possibles [3]. Des algorithmes permettent même de décomposer les modèles en parties ayant le même nombre de polygones [4]. Ce dernier algorithme à l'avantage de pouvoir décomposer les objets en parties simples ce qui aide certaines méthodes de détection de collisions à être plus

efficaces. Les parties simples sont des morceaux de l'objet d'origine qui possèdent moins de polygones et aucune ou peu de parties concaves.

Certains algorithmes fonctionnent seulement avec des modèles convexes fermés, par conséquent, les parties devront également être fermées [5]. Dans d'autres cas, seulement des ensembles non fermés seront suffisants [6]. La division de modèles d'objets 3D occasionne un temps de traitement qui n'est pas négligeable, mais pour les modèles non déformables, elle est calculée seulement une fois en début de traitement.

Chaque entité découlant de la division d'objet est utilisée individuellement pour effectuer les tests de collision avec les parties des autres objets. De plus, il faudra mettre en relation les parties d'un même objet puisque la détection de collisions entre ces parties n'est pas requise.

Les modèles polygonaux sont donc les plus répandus et les plus documentés. La plupart des algorithmes de détection de collisions utilisent ce type de modèle. Par contre, il est impossible de simuler avec précision des surfaces courbes avec ce type de modèle ce qui altère grandement la qualité de la représentation des objets 3D.

1.1.3 Constructive solid geometry (CSG)

L'approche de modélisation Géométrie constructive de forme (Constuctive solid geometry ou CSG [7]) permet de construire des modèles à partir de formes simples préétablies. Cette technique permet d'approximer des collisions et elle est utilisée surtout dans le domaine du jeu vidéo. Le jeu Unreal, par exemple, utilise cette technique.

Les formes utilisées pour construire les modèles d'objets 3D sont les sphères, les cubes, les cylindres, les cônes, les prismes et les pyramides. La construction des formes plus complexes se fait à l'aide de ces formes simples et d'opérations d'union, de différence et d'intersection sur ces formes de base telles que présentées dans la figure 1.4.

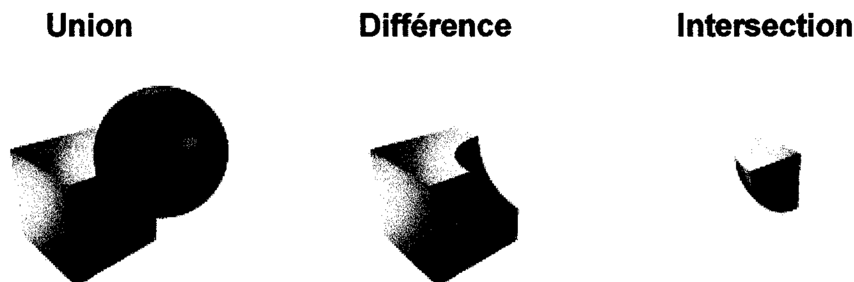


Figure 1.4. Opérations d'union, de différence et d'intersection dans CSG.(1)

(1) Wikipedia, the free encyclopedia : http://en.wikipedia.org

Avec une succession d'opérations sur différentes formes, comme le cube et la sphère, il est possible d'approximer les objets 3D plus complexes. La détection de collisions est toujours basée sur les modèles simples qui ont servi à construire les modèles d'objets 3D plus complexes ce qui simplifie grandement cette opération. Les modèles ainsi créés sont toujours fermés. L'intérieur et l'extérieur sont également bien définis. L'aire de pénétration est aussi facile à calculer.

Par contre, la technique CSG provoque souvent des erreurs de collisions au niveau des intersections entre les formes. Par exemple, il existe à l'intersection entre un cube et une sphère deux normales d'orientation différentes ce qui rend problématique d'application d'un effet de rebondissement comme décrit à la figure 1.5.

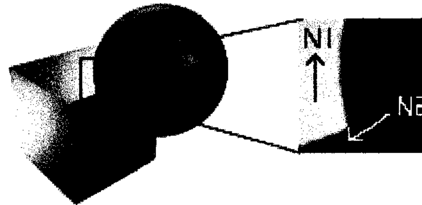


Figure 1.5. Représentation des deux normales formées par une intersection.

Dans la figure 1.5, la ligne verte correspond aux points d'intersection entre les deux formes. Il existe donc deux normales en ces points, soit N1, qui est la normale du cube et N2, qui est la normale de la sphère.

Plus le nombre de formes simples est grand dans un modèle, plus la détection de collisions avec ces modèles est longue. Il faut alors trouver le juste milieu entre la précision et le temps d'exécution. Un grand nombre de formes simples et d'opérations causent plus d'intersections, donc, plus de possibilités d'erreurs. Puisqu'il y a une bonne différence entre le modèle CSG d'un objet 3D et sa forme d'origine, il est préférable d'afficher la représentation polygonale d'un objet 3D par le système d'affichage.

En résumé, l'approche CSG est une méthode très utile pour détecter les collisions entre des modèles statiques. Elle supporte également les surfaces courbes contrairement à la méthode polygonale. Il est par contre difficile de créer un modèle exact de l'objet d'origine. CSG est une méthode très rapide pour approximer les collisions.

1.2.4 Conclusion

La modélisation des objets sert pour l'affichage des objets et la détection de collisions. Les modèles d'affichage sont souvent plus complexes et plus

précis afin d'avoir une meilleure qualité graphique. Les modèles qui servent à la détection de collisions ne sont pas toujours les mêmes que ceux qui ont servis à l'affichage. Ils sont souvent plus simple et adapter à l'algorithme de détection de collisions.

Les modèles polygonaux sont largement utilisés dans l'affichage des objets. Ils sont simples étant donné qu'ils sont représentés par un ensemble de triangles. Il existe une différence dans la gestion des modèles polygonaux convexes et non convexes en détection de collisions, les modèles non convexes sont plus problématiques puisqu'ils ne sont pas supportés par tous les algorithmes et ils doivent alors être subdivisés en parties convexes.

L'approche de modélisation CSG permet de construire des objets à partir des formes simples. Les opérations d'union, de différence et d'intersection sont utilisées afin de créer des modèles approximant les objets. Les formes simples se calculent rapidement ce qui améliore grandement le temps d'exécution.

CSG et les surfaces implicites permettent de simuler des surfaces courbes. Les objets réels sont rarement parfaitement droit alors pour des simulations précises de surfaces courbes, il est préférable de les utiliser.

Les modèles polygonaux servent souvent d'objets de base dans l'affichage des objets. Pour avoir une détection parfaite de ces modèles, il faut les utiliser dans la détection de collisions. Par contre, s'ils possèdent un très grands nombre de polygones (triangles), les calculs deviennent extrêmement lourds voir impossible.

Dans l'expérimentation (Chapitre 2) présentée dans ce mémoire, la modélisation graphique et la modélisation de détection de collisions sont étroitement liées. Les modèles optimisent à la fois les deux systèmes. Les cubes sont représentés par un ensemble de plans 2D pour faire la détection de collisions. Cet ensemble est ensuite divisé en polygones afin d'en faire l'affichage. Des objets utilisent les formes proposées par CSG afin de gérer parfaitement les surfaces courbes.

La meilleure solution consiste souvent à utiliser des méthodes adaptées au objets afin de les approximer le mieux possible et d'en optimiser le temps de calcul. C'est pourquoi une modélisation hybride est utilisée dans l'expérimentation.

1.3 Type d'environnement

1.3.1 Introduction

Les environnements 3D comportent diverses problématiques pouvant être solutionnées par la détection de collisions. Ces problématiques découlent des environnements caractérisés principalement par les entrées de données et les résultats demandés. Ces environnements possèdent donc des propriétés qu'il est possible d'exploiter afin d'optimiser l'algorithme de détection de collisions.

Des algorithmes sont mieux adaptés à certains types d'environnement ou à certaines caractéristiques de ceux-ci. La connaissance de ces caractéristiques est alors primordiale afin de choisir un algorithme mieux adapté à la problématique.

Le type de requêtes, le nombre d'objets, le type de mouvements et les modèles déformables ou non sont les principales propriétés qui caractérisent les environnements. Ce sont ces propriétés qui seront examinées en détail dans cette section.

1.3.2 Type de requête

Une requête correspond au résultat de la détection de collisions. Les détails du résultat peuvent varier d'un environnement à l'autre selon les besoins. Il est préférable de satisfaire ces besoins au minimum pour éviter de faire des calculs inutiles.

Dans certains cas, seulement une valeur booléenne déterminant si des paires d'objets sont en collision ou non est suffisante. Il n'est pas nécessaire dans ce cas de déterminer la distance entre les deux objets ou l'aire en intersection. Il faut les calculer seulement si la requête l'exige.

D'autres applications auront à estimer un temps avant impact afin de déterminer la procédure à suivre après l'occurrence de la collision. Il faut donc prévoir le mouvement des objets afin de déterminer ce temps avant impact ce qui est beaucoup plus lourd qu'une détection de collision sans estimation du mouvement. Certains algorithmes utilisent implicitement le temps avant impact ce qui pourrait être très avantageux pour traiter cette problématique [12].

Lorsque des objets 3D sont en collision et en pénétration, certaines applications devront résoudre ces pénétrations ce qui peux impliquer l'estimation de l'aire d'intersection. Il faut de plus déterminer une translation minimum des

objets afin de pouvoir résoudre la collision et aussi déterminer la direction du rebondissement [13].

Les besoins peuvent donc varier d'une application à l'autre. Examiner les requêtes nécessaires est très important afin de minimiser les calculs requis. Cela permet de diminuer le temps d'exécution et de trouver ou créer un algorithme de détection de collisions adapté aux besoins de l'environnement.

1.3.3 Deux ou N objets

Les applications utilisent un certain nombre d'objets qui peut varier grandement de l'une à l'autre. Pour la détection de collisions, les environnements sont divisés en deux catégories soient : les environnements à 2 objets et les environnements à N objets.

Les environnements à 2 objets consistent seulement en deux objets qui sont susceptibles d'entrer en collisions. L'environnement a alors une seule paire d'objets. Ce cas est très simple, où l'algorithme de détection de collisions peut être plus lourd puisque l'algorithme est exécuté une seule fois par itération.

La complexité des environnements à N objets est beaucoup plus grande. Le nombre de paires d'objets augmente de manière exponentielle étant donné que chacun des objets doit faire un test de collisions avec tous les autres. Ce nombre de paires d'objets est déterminé par la formule ***N * (N-1) / 2***. La figure 1.6 permet de visualiser le graphique mettant en relation le nombre de paires d'objets 3D pouvant entrer en collision selon le nombre d'objets composant un environnement 3D.

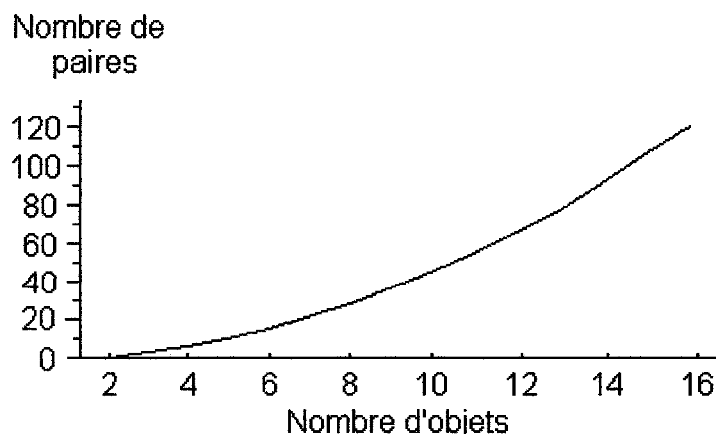


Figure 1.6. Nombre de paires selon le nombre d'objets

L'algorithme doit tester l'ensemble des paires d'objets afin de vérifier si ceux-ci sont en collision. Dans le logiciel présenté dans ce document, il y a plus de milles objets (voir chapitre 2). Des optimisations sont nécessaires afin d'éliminer un maximum de paires d'objets rapidement afin de tester seulement celles à risque.

Le nombre d'objets est alors important puisqu'il fait augmenter le nombre de paires d'objets rapidement et que chacune des paires doit faire appel à l'algorithme de détection de collisions.

1.3.4 Mouvement statique ou dynamique

Le mouvement est également très important dans les algorithmes de détection de collisions. S'il est connu et bien défini, un nombre important de calculs pourra être simplifiés.

Dans certains environnements, le mouvement des objets est inconnu et peu changer drastiquement d'une itération à l'autre. Ce type de mouvement est dynamique. Dans ce cas, il n'y a pas de cohérence dans les mouvements et il est difficile d'optimiser la détection de collisions sur ce type de mouvement. C'est le cas de la réalité virtuelle, où l'utilisateur pour décider à n'importe quel moment de changer de direction. À chaque itération, il faut inévitablement recalculer l'ensemble des objets qui ont bougé par rapport aux autres.

Le mouvement statique est un mouvement connu. La direction et la vitesse du mouvement sont bien définies. L'objet peut être en accélération ou en décélération mais le changement de vitesse est connu. Les environnements entièrement simulés ou les environnement où l'utilisateur a peu d'influence présentent souvent des mouvements statiques. Dans ce cas, il sera possible d'appliquer la cohérence temporelle [14]. La cohérence temporelle est expliquée en détail dans la section *1.4.9 Optimisations des détections de collisions*.

Le type de mouvement est important et il peut être exploité afin d'améliorer l'algorithme de détection de collisions. Le mouvement statique est particulièrement intéressant puisqu'il a des propriétés qui permettent de diminuer le temps d'exécution de la détection. Par contre, dans certains environnements, le mouvement est dynamique et ces optimisations ne sont pas applicables.

1.3.5 Modèle rigide ou déformable

Les modèles rigides sont les plus utilisés puisqu'ils sont les plus simples. Ils sont néanmoins limités par rapport au modèle déformable.

Les modèles rigides ne changent pas de forme dans le temps. Les modèles 3D qui les représentent sont toujours les mêmes d'une itération temporelle à l'autre. Souvent, lorsque l'objet doit changer de forme dans le temps, le modèle de l'objet à un temps t_i est remplacé à un temps t_{i+1} par un autre modèle du même objet qui représente l'état de la déformation.

Cela cause un problème lorsque l'objet change de forme à chaque itération. La majorité des algorithmes détection de collisions sont conçus pour des modèles statiques. C'est pourquoi des travaux sur les modèles déformables ont débutés [15]. La cohérence temporelle sur les déformations est également applicable si les paramètres de déformations sont bien connus.

La recherche sur les modèles déformables est encore au stade embryonnaire. Pour l'instant, les modèles déformables sont traités comme une succession de modèles rigides. Cela peut avoir un impact considérable selon l'algorithme de détection de collisions choisi. L'utilisation d'algorithmes où le changement de modèle a peu d'influence sur l'efficacité de la détection de collisions est donc à considérer en attendant le développement d'algorithmes qui supportent efficacement les modèles déformables.

1.3.6 Conclusion

Bien connaître l'environnement dans lequel évolue l'algorithme de détection de collisions est donc essentiel. Cela permet d'optimiser au maximum les calculs de l'algorithme de détection de collisions en appliquant des propriétés relatives à certains environnements.

Les requêtes représentent les besoins de l'environnement que la détection de collisions doit satisfaire. Ces besoins peuvent être très variés et certains demandent plus de précision que d'autres. La satisfaction de ces besoins est essentielle mais surenchérir ces besoins est inutile et peut nuire au temps d'exécution.

Le nombre d'objets qu'utilise l'environnement doit être connu. Étant donné que le nombre de paires d'objets augmente très rapidement avec le nombre d'objets, les environnements ayant un grand nombre d'objets doivent être gérés différemment. Des processus d'élimination rapide de paires sont applicables.

Le mouvement a également un impact important sur l'algorithme de détections de collisions. Ce mouvement peut être statique (connu) ou dynamique (inconnu). Dans les environnements qui génèrent des mouvements statiques, la cohérence temporelle peut être exploitée.

Les modèles déformables causent certains problèmes pour certains algorithmes de détection de collisions. La succession de modèles différents d'un

même objet générés dans un environnement peut avoir un grand impact sur l'efficacité des calculs de détection de collisions. Une modélisation différente pour les modèles déformables est en développement.

Le logiciel développé dans le contexte de ce mémoire présenté au chapitre 3 possède un environnement qui lui est propre. Il requiert une très grande précision de la détection des collisions. Les requêtes doivent donc être les plus exactes possibles afin que les rebondissements soient les plus réalistes possibles. La scène est constituée de plus de milles objets dont plusieurs sont en mouvement. Une optimisation sur les paires d'objets est donc nécessaire. Le mouvement est souvent statique, ce qui permettra d'optimiser une majorité des calculs de détection de collisions. Par contre, un objet déplacé par l'utilisateur doit être géré différemment puisqu'il présente un mouvement dynamique.

Bien connaître l'environnement et ces besoins est donc primordial afin d'appliquer les meilleurs algorithmes et de satisfaire les requêtes. Cette connaissance permet d'utiliser les bonnes optimisations qui seront décrites dans la section suivante.

1.4 Optimisation des détections de collisions

1.4.1 Introduction

L'optimisation est l'aspect le plus important des algorithmes de détection de collisions. Cette optimisation consiste à exploiter certaines caractéristiques des objets et des environnements afin de diminuer de temps exécution de la détection de collisions.

Mathématiquement, les collisions sont souvent bien définies. Les fonctions vectorielles et les courbes dans l'espace permettent de déterminer précisément les points de contact [16].

Bien que ces théorèmes soient justes et établis depuis plusieurs années, ils sont néanmoins excessivement lourds en temps de calculs, même pour un ordinateur d'aujourd'hui. De plus, ces calculs seront souvent répétés plusieurs fois afin de simuler le temps et l'interaction entre plusieurs objets.

Par exemple, dans le cas des modèles polygonaux qui sont constitués d'un ensemble de triangles chacun formé de trois vecteurs 3D, déterminer si deux triangles sont en intersection, donc en collision, est simple et peu gourmant en temps d'exécution. Par contre, un objet normal peut être constitué de milliers de polygones. Lorsque la détection de collisions est effectuée sur deux objets de

mille polygones, cela fait déjà près d'un million de tests de paires de triangles. Chacune des paires de triangles (un triangle de chaque objet) est testée pour déterminer l'intersection entre ces triangles.

Souvent, les environnements simulés fonctionnent avec une interface visuelle qui utilise des fonctions graphiques très lourdes en temps de calcul. Si tel est le cas, l'algorithme de détection de collisions devra prendre une fraction du temps de d'affichage graphique afin que le rendu graphique ne soit pas trop affecté.

De là découle l'importance de l'optimisation, qui devra permettre d'éliminer en début de traitement l'ensemble des objets 3D dont la collision est improbable. Certaines optimisations affecteront la précision du résultat de la détection de collisions et d'autres, n'auront aucune incidence selon le type d'algorithme de détection de collisions employé.

Cette section survolera un ensemble de techniques dont plusieurs sont utilisées dans le développement du logiciel Arka 3D qui fait l'objet du présent document. Ces techniques exploitent diverses notions mathématiques qui sont entre autres :

- La simplification des objets
- La distance entre les objets
- Le mouvement et la prévision de celui-ci
- Les bornes autour des objets

Ces ensembles de techniques utilisées conjointement permettent de réduire considérablement le temps d'exécution des algorithmes de détection de collisions.

1.4.2 Simplification des objets

La simplification des objets est une méthode simple et efficace qui permet de diminuer directement le nombre de calculs de l'algorithme de détection de collisions. Cette simplification, qui se fait au niveau des objets, ne modifie donc pas directement l'algorithme et elle peut donc être appliquée à n'importe quel type de détection qui fonctionne avec des modèles polygonaux.

En début de traitement, donc avant la détection de collisions, les objets sont simplifiés afin de diminuer le nombre de polygones de ceux-ci. Pour les modèles non déformables, la simplification sera faite une seule fois en début de traitement. Par contre, dans le cas de modèles déformables, la simplification devra être faite à chaque déformation, voir chaque itération temporelle. Cette technique devient alors très lourde avec ce type de modèles. Elle est à éviter dans ce cas.

L'objet 3D simplifié qui possède moins de polygones requiert alors moins de temps à l'algorithme de détection de collisions pour déterminer les collisions potentielles mais cet objet est bien sûr moins précis que l'objet d'origine. De nombreuses techniques permettent de diminuer le nombre de polygones d'un objet afin de perdre le moins de précision possible [17]. La figure 1.7 présente un objet très complexe qui a été simplifié.

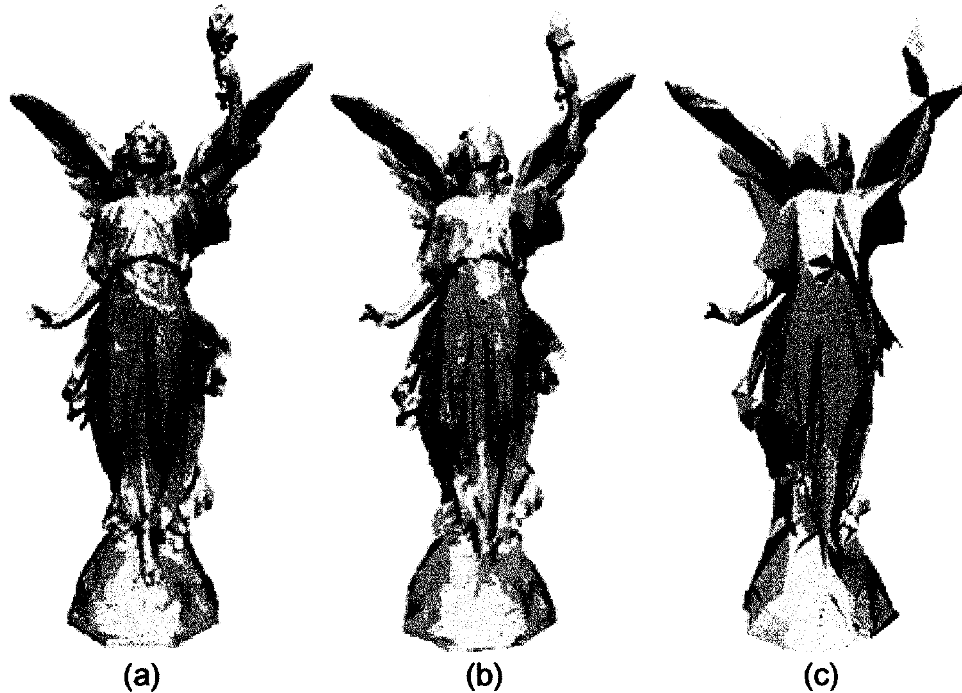


Figure 1.7. Statue de Lucy simplifiée[(1)].
(a) Forme originale.
(b) Forme simplifiée avec une faible réduction du nombre de polygones.
(c) Forme simplifiée avec une forte réduction du nombre de polygones.

La version simplifiée est utilisée par la détection de collisions afin de diminuer son temps de traitement. L'objet 3D d'origine est utilisé pour l'affichage afin de ne pas diminuer la qualité graphique de celui-ci. Les légères différences entre l'objet simplifié et l'objet d'origine causent des imprécisions sur la détection de collisions. Cette technique est donc à éviter lorsque la précision est primordiale.

La simplification est une technique qui permet de diminuer le nombre de polygones des objets. En diminuant le nombre de polygones traités par l'algorithme de détection de collisions, le temps de traitement est bien sûr diminué proportionnellement. Par contre, en enlevant des polygones, la forme de l'objet change et la détection de collisions devient alors moins précise.

[(1)] Prasun Choudhury et Benjamin Watson. Completely Adaptive Simplification of Massive Meshes. Northwestern University.

1.4.3 Limite de collision autour des objets

Les limites de collisions autour des objets sont des frontières construites à partir de formes simples qui entourent les objets. Elles permettent de déterminer rapidement si l'objet peut être en collision ou non. Elles sont particulièrement utiles dans les environnements à N objets en permettant d'éliminer rapidement des paires d'objets.

La forme la plus simple à utiliser est la sphère. En encerclant les objets de sphères, il suffit de calculer la distance entre les points centraux des sphères d'approximation pour déterminer si les objets peuvent être en collision ou non. La figure 1.8 présente un exemple en deux dimensions de la détection de collisions à l'aide de sphères.

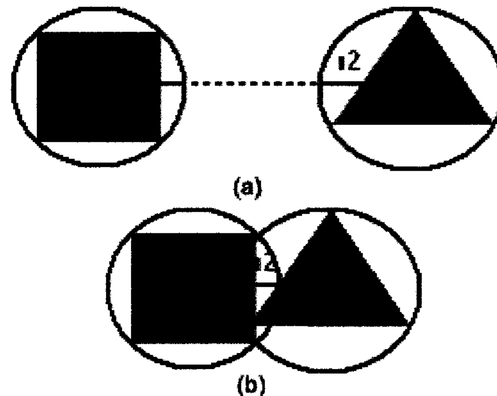


Figure 1.8. Déterminer si une collision est possible à partir de sphères.
(a) Collision impossible. (b) Collision possible.

Les variables r1 et r2 représentent le rayon des deux sphères. La distance est calculée entre les points centraux des deux sphères à l'aide de la formule bien connue :

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \quad (1)$$

Si la distance entre les deux sphères est plus grande que la somme des deux rayons (Figure 1.8 (a)), la collision entre les deux objets est impossible et la paire d'objets pour être éliminée du traitement. Si la distance est plus petite que la somme des deux rayons (Figure 1.8 (b)), une collision est possible et une détection plus précise est alors nécessaire.

D'autres formes peuvent servir à déterminer des limites autour des objets. La figure 1.9 en présente quelques unes.

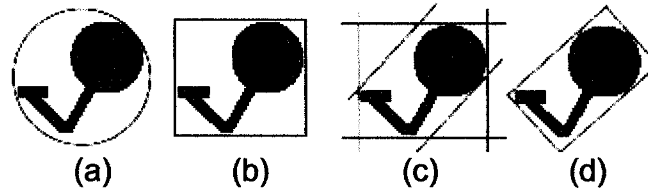


Figure 1.9. Frontières possibles autour des objets. (a) Sphères [35].
(b) Boîte orientée sur les axes (Axis Aligned Bounding Box) [18].
(c) Boîte orientée sur les axes généralisés (k-dop) [36].
(d) Boîte non orientée sur les axes (Oriented bounding Box) [19].

Chacune des limites offrent des avantages et des inconvénients.

- La sphère (figure 1.9 (a)) est simple à comprendre et à implémenter mais le calcul des distances n'est pas très rapide dû aux mises au carrée et la racine (Équation 1).
- La boîte orientée sur les axes (AABB, figure 1.9 (b)) est très simple à calculer et à détecter mais possède un volume plus grand que les autres approches. Un volume plus grand élimine moins de paires d'objets et la convergence vers un état de collision est donc plus long.
- La boîte non orientée sur les axes (OBB, figure 1.9 (d)) minimise au maximum le volume de détection. Par contre, elle est plus longue à déterminer et à détecter.
- La technique k-dop (figure 1.9 (c)) est un compromis entre les boîtes AABB et OBB.

Ces limites autour des objets sont beaucoup moins longues en temps d'exécution que les objets complets ce qui permet de déduire une première détection rapidement. Cette première détection permet de déterminer si un objet est susceptible d'entrer en collision avec un autre. Si ces objets ne sont pas susceptibles d'entrer en collision, l'algorithme passe à la prochaine paire d'objets ce qui élimine un grand nombre de calculs. C'est pourquoi cette optimisation est très utile dans les environnements à N objets.

1.4.4 Subdivision des objets

Comme présenté à la section 1.2.2 portant sur la modélisation, les modèles polygonaux sont décomposables. Cette division de modèle permet également une optimisation au niveau de la détection de collisions aussi bien au niveau des objets 3D convexes que non convexes. Cette technique permettra de délimiter un section particulière de l'objet qui est propice à une collision afin d'y faire une détection de collisions plus précise.

Le découpage des sections d'un objet 3D se fait principalement par division binaire successive et prend la forme d'un arbre binaire [19]. La section de l'objet 3D, dont la sphère englobante est en collision avec celle d'un autre objets, est alors divisée en deux et de nouvelles sphères de rayons plus petits sont calculées. Ces sphères sont à nouveau soumises à un autre test de collision, ainsi de suite jusqu'à ce qu'un critère d'arrêt soit atteint. Ce critère d'arrêt correspond au moment qu'une section devient insécable. La figure 1.10 présente le processus de division à l'aide de sphères.

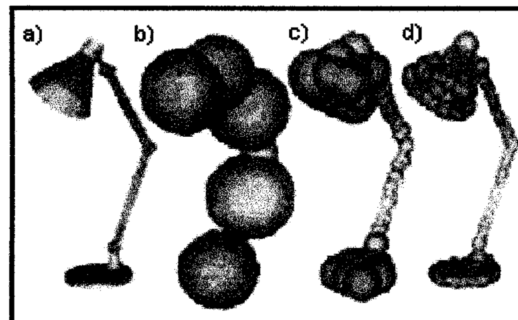


Figure 1.10. Division successive d'une lampe à l'aide de sphères. (1)
a) Objet d'origine.
b) Objet subdivisé grossièrement
c) Objet subdivisé modérément
d) Objet subdivisé considérablement

La limite de collisions autour des objets, vu à la section précédente (1.4.3), est utilisée sur les sections d'objet 3D afin de déterminer rapidement si une collision dans cette section est possible ou non grâce à des tests simples fait à l'aide de modèles simples. Si toutes les sections ne sont pas à risque de collisions donc à l'extérieur des limites, il n'y a pas de collisions possibles et l'algorithme peut arrêter. Dans le cas où une ou plusieurs sections sont à risque, ces sections sont divisées à nouveau et l'algorithme se poursuit.

Lorsque le critère d'arrêt est atteint, la division des modèles est arrêtée et des calculs plus poussés de détection peuvent être effectués sur les sections à risque afin de déterminer précisément s'il y a une collision ou non. Dans un environnement où la précision est moins importante, il est possible de déduire une collision uniquement si une ou plusieurs sections sont à risque à la fin du traitement sans faire les calculs précis.

Que ce soit pour déterminer rapidement les sections à tester précisément ou pour faire une détection approximative directement, la subdivision des objets est une optimisation très intéressante qui permet d'améliorer grandement le temps d'exécution. Cette amélioration découle des sections d'objet graphique éliminées sur lesquelles des calculs plus approfondis n'ont pas à être effectués. Cette technique largement utilisée est une partie importante des algorithmes I-Collide (section 1.6.2), Rapid (section 1.6.3) et V-Collide (section 1.6.4). Une

(1) Division successive d'une lampe à l'aide de sphères. hubbard 95.

technique similaire est utilisée pour la détection de collisions des cubes dans le logiciel développé dans le cadre de cette recherche et sera présenté au chapitre 2 du présent document.

1.4.5 Division de l'espace

Les environnements qui possèdent un grand nombre d'objets 3D génèrent un très grand nombre de paires d'objets. Un technique consiste à diviser l'espace 3D en sections. Les objets appartiendront à un nombre limité de sections et seul les objets qui partagent une même section pourront entrer en collision.

Les sections sont souvent déterminées à l'aide de boîtes alignées sur les axes [18]. La taille des boîtes est arbitraire mais peut être fixée en fonction de la densité des objets. La figure 1.11 décrit un exemple de division d'espace en 2D.

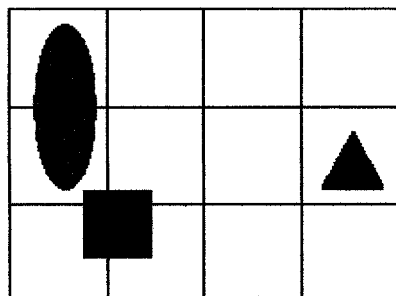


Figure 1.11. Méthode par grillage.

La figure 1.11 permet de visualiser un espace 2D composé de trois objets (A, B, C). Par conséquent, il existe théoriquement trois paires d'objets possibles (AB, AC, BC). Avec la méthode par division de l'espace, seules les paires d'objets partageant une même section sont susceptibles d'entrer en collision. Seule la paire AB est retenue pour des calculs plus précis. Il est possible que les objets appartiennent à plusieurs sections (Comme les objets A et B) lorsqu'ils chevauchent celles-ci.

Cette méthode permet alors d'éliminer un grand nombre de paires d'objets surtout dans les environnements peu dense où les objets sont peu propices à partager une même section. C'est donc une méthode intéressante et facile d'application de simplifier la détection de collisions. Elle est utilisée dans la détection entre balles et cubes du logiciel Arka3D.

1.4.6 Déterminer la distance entre deux objets

Dans un environnement où la direction et la vitesse des objets sont connues, la distance entre deux objets peut être déduite. Cette distance représente alors le parcours minimum avant d'entrer en collision. L'objet pourra donc se déplacer de cette distance sans entrer en collision.

Pour les modèles polygonaux, cette distance doit être calculée. Le fait de trouver les deux polygones les plus rapprochés revient à refaire la détection de collisions en entier pour chaque paire de polygones. Il faut donc approximer les objets 3D. La technique d'approximation à l'aide d'une sphère introduite à la section 1.4.3 peut alors être utilisée [20].

Une fois la distance entre les objets trouvée ces mêmes objets pourront alors se déplacer de cette distance sans entrer en collision. Si les deux objets sont en mouvement, la somme des déplacements constituera la distance minimum avant la collision. Ces deux objets pourront donc se déplacer librement pour un moment.

Cette technique permet donc d'éviter des calculs à l'extérieur de cette distance minimum. Deux objets très éloignés pourront alors se déplacer sans que la détection de collisions ne soit requise et ce tant que la distance séparant ces objets ne soit plus grande que la distance minimum.

1.3.7 Interférence par balayage

Dans une scène 3D où les objets se déplacent, l'environnement est en fait en 4D puisque le déplacement constitue la quatrième dimension. Plus l'environnement possède de dimensions, plus il devient difficile de déterminer si les objets sont en collision ou non. L'idée de l'interférence par balayage est d'éliminer des dimensions afin de simplifier les calculs. Cela permet alors de déterminer si une collision est possible ou non.

L'élimination d'une dimension s'opère en projetant l'objet dans une dimension inférieure. Par exemple, la projection d'un objet 3D est effectuée sur un plan 2D et la projection d'un objet 2D se fait sur une droite comme le montre la figure 1.12.

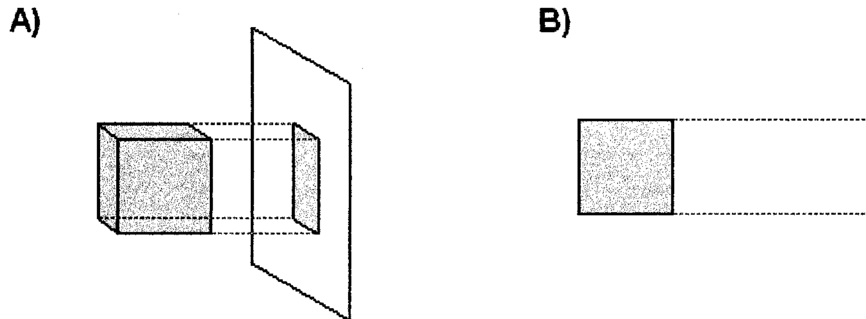


Figure 1.12. Projection d'objets dans une dimension inférieure. A) Projection d'un objet 3D sur un plan 2D. B) Projection d'un objet 2D sur une droite.

Ces projections se font habituellement sur les axes de référence puisqu'elles sont très simples. Il suffit alors d'éliminer une variable x, y ou z, les deux autres forment alors l'objet projeté. Il s'agit ici d'une projection parallèle [21].

Certains algorithmes comme OBB-Tree [19] ont besoin de projections sur d'autres plans que ceux formés par les axes de référence. Une rotation de l'objet est alors nécessaire ce qui nécessite un temps d'exécution beaucoup plus long que la projection sur les plans formés par les axes de référence.

Dans le cas d'objets en mouvement (4D), la projection en 3D représentera l'ensemble des positions d'objets durant un temps donné. La figure 1.13 présente des objets 2D en mouvement.

Quand deux objets sont en collision dans **toutes** les dimensions inférieures, il est **possible** que les deux objets soient en collision dans une dimension supérieure. Lorsque deux objets ne sont pas en collision dans **n'importe qu'elle** dimension inférieure, il est **impossible** que les deux objets soient en collision dans une dimension supérieure et l'algorithme de détection peut donc arrêter à ce moment. La figure 1.13 en présente un exemple.

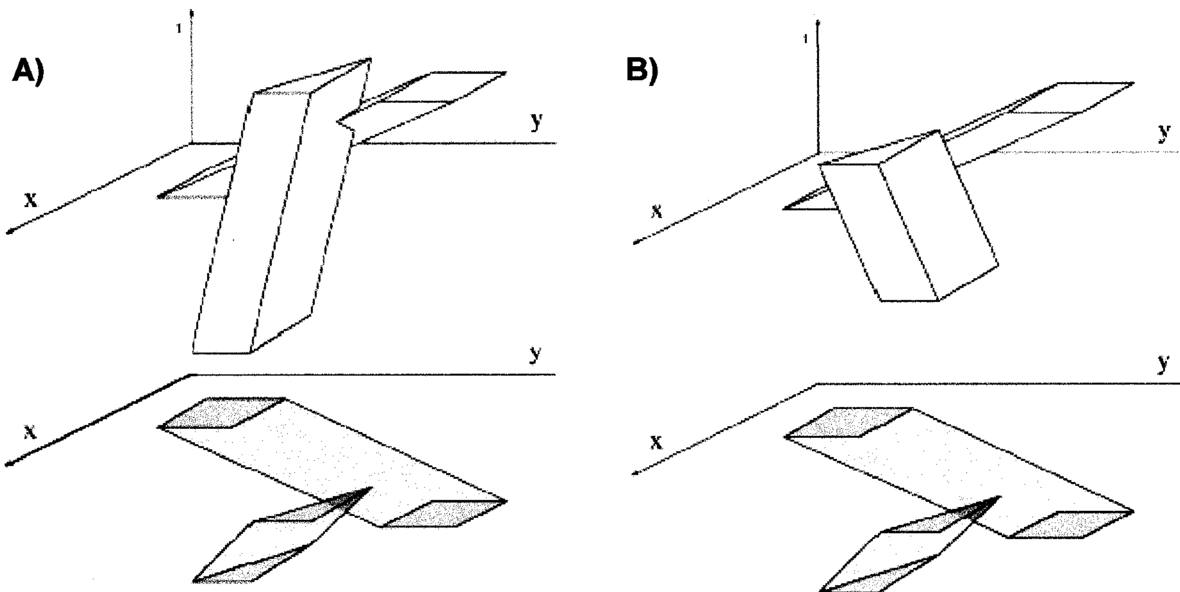


Figure 1.13. Projection à des fins de détection de collisions.[1] A) Collision possible détectée dans la dimension inférieure (plan x-y) lorsqu'une collision survient dans la dimension supérieure. B) Collision possible détectée dans la dimension inférieure dans le cas ou aucune collision ne survient dans la dimension supérieure.

Lorsque les deux objets projetés ne se touchent pas, la collision est alors impossible. Concrètement, il s'agit alors d'un angle de vue où l'on voit les deux objets qui ne se touchent pas.

Étant donné que les calculs sont plus simples dans une dimension inférieure, la détection de collisions sera grandement simplifiée.

Les boîtes alignées sur les axes, comme celles utilisée dans I-Collide [18], ont une propriété très intéressante. D'abord, les projections se font toujours sur les plans de projection (x, y, 0), (x, 0, z) ou (0, y, z) et les dimensions tronquées sont toujours perpendiculaires à un de ces plans. La propriété est la suivante : si toutes les projections inférieures sur les plans de projection sont en collision, l'objet est donc en collision avec un autre objet dans la dimension supérieure. C'est seulement avec les boîtes alignées sur les axes de référence que cette propriété est vraie.

Les boîtes OBB utilisées dans RAPID [19] ont également une propriété similaire, par contre, elle demande plusieurs projections sur des plans qui ne sont pas sur un des axes de référence et perpendiculaire à ceux-ci. On peut néanmoins déterminer exactement si les boîtes orientées sont en collision à partir de ces projections.

(1) Michael Ian Samos, Computational Geometry, Dissertation Information, 1978. page 175.

L'interférence par balayage permet donc une vérification supplémentaire simple sur les dimensions inférieures des objets. Cela permet de déterminer rapidement si une collision est possible avant d'appliquer une détection plus précise mais laborieuse. Plusieurs algorithmes publics, comme I-Collide [18] et RAPID [19] utilisent cette stratégie.

1.4.8 Interférence multiple

Une alternative possible afin de détecter la distance entre deux objets est d'utiliser les régions de Voronoï. Cette technique permet de projeter les sommets, les arêtes et les faces d'un objet afin de déterminer plus facilement la distance.

La région de Voronoï associée à une surface (sommet, arrête ou face) représente une zone qui s'étend souvent à l'infini et où les points situés dans cette zone sont inévitablement les plus proches de la surface concernée. À partir de ces zones, il est possible de converger rapidement vers les deux points les plus proches. Ainsi, un point x associé à la région X et un point y associé à la région Y seront les plus proches si x chevauche la région Y et que y chevauche la région X. Cela permet de réduire la complexité d'estimation de distance de N^2 à N log(N) où N est le nombre de sommets.

La figure 1.14 présente un exemple en 2D du phénomène.

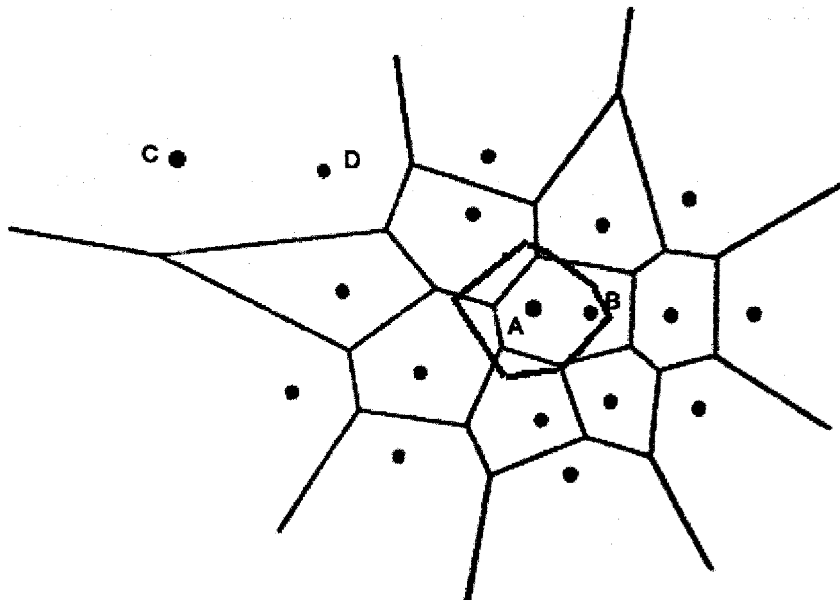


Figure 1.14. Les régions de Voronoï pour des points.

La figure 1.14 montre les régions de Voronoï d'un ensemble de points. Le point C est situé dans la région du point D. Le point D est donc le plus proche du point C. Pour vérifier si les deux points sont mutuellement le plus rapprochés, il faut vérifier que les deux points appartiennent à la région de Voronoï de l'autre. Par exemple, le point A est situé dans la région de B et le point B est situé dans la région (en rouge) du point A. Ces points sont donc les plus près l'un de l'autre.

Il existe donc une région de Voronoï pour chacun des sommets, des arêtes et des faces de l'objet 3D. Ces projections sont présentées dans la figure 1.15.

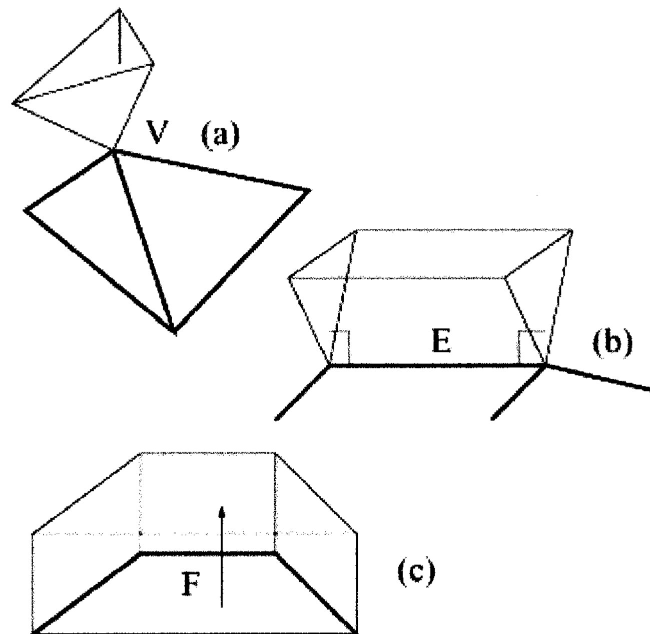


Figure 1.15. Régions de Voronoï.(1) (a) Région de Voronoï pour un sommet, (b) Région de Voronoï pour une arête, (c) Région de Voronoï pour une face.

Il existe donc 3 types de régions de Voronoï. Par exemple, un cube a 8 sommets, 12 arêtes et 6 faces. Chacun de ces éléments est associé à une région de Voronoï.

Des techniques permettent de déterminer ces régions, certaines sont utilisées seulement pour les boîtes rectangulaires [22] et d'autres permettent de les appliquer à n'importe quels objets convexes [23].

Certains algorithmes, comme celui de Lin-Canny [29], n'utilise que la région de Voronoï des sommets et d'autre, comme dans V-Clip [24], utiliseront l'ensemble de ces régions.

Les régions de Voronoï permettront alors de déterminer les points les plus rapprochés entre deux objets. Par contre, si les deux objets sont en intersection,

(1) P. Jiménez, F. Thomas, C. Torras. 3D collision detection: a survey. Page 277

le processus de détection bouclera à l'infini puisque les points à l'intérieur des objets appartiendront à plusieurs régions de Voronoï. Un pseudo-Voronoï est alors utilisé afin de détecter cette boucle et ainsi déterminer si les objets sont en collision. Cette technique est utilisée dans V-Clip [24].

Les régions de Voronoï aussi appelées interférences multiples permettent donc de déterminer la distance entre les deux points les plus proche de deux objets 3D. Cela permet implicitement de déterminer si deux objets sont en collision. Cette technique est utilisée dans l'algorithme V-Clip et dans I-Collide.

1.4.9 Cohérence temporelle.

Lorsque le temps et le déplacement font partie de l'environnement, il est possible de les exploiter. Lorsque la trajectoire est connue, une prédiction sur une collision future peut servir à éviter la redondance de certains traitements. Cette technique est utilisée dans Q-Collide [25], une version améliorée de I-Collide pour les objets en mouvement.

La section 1.4.6 présente une approche permettant de déterminer la distance minimum avant impact afin de diminuer le nombre de tests de collision. Cette technique mit en parallèle avec le temps peut être exploitée. La détection de collisions peut alors être interrompue sur le laps de temps où la distance minimum n'est pas atteinte.

Une autre application de la cohérence temporelle est l'élimination de polygones. Lorsque les objets se déplacent en ligne droite, ils peuvent entrer en collision seulement avec les polygones qui sont situés face à la trajectoire de l'objet. La figure 1.16 présente le phénomène dans un environnement 2D.

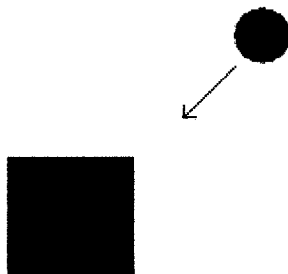


Figure 1.16. Collision possible d'une balle en mouvement (surligné en rouge).

Cette figure présente un objet 2D en bleu formé de quatre côtés et une sphère qui se déplace de manière linéaire vers celui-ci. Seuls deux cotés sur quatre sont sujets à entrer en collision avec la sphère.

Donc, cette technique permet d'éliminer en moyenne la moitié des polygones lors de la détection de collisions. Les polygones sujets à une collision sont ceux qui font face au vecteur de déplacement. Un produit scalaire entre le vecteur de déplacement et le vecteur de la normale du polygone est alors effectué. Seuls les polygones dont le produit scalaire est inférieur à zéro sont ainsi conservés. Cela permet de réduire le temps d'exécution de la détection de collisions en moyenne de moitié.

Une autre propriété intéressante que permet l'application de la cohérence temporelle concerne les sphères. Deux sphères en mouvement peuvent entrer en collision que si leurs centres se rapprochent. Alors, deux sphères dont la distance des centres augmente ne peuvent entrer en collision. Cette propriété peut être utilisée avec n'importe quel objet 3D si des sphères englobantes sont utilisées (Section 1.4.3). Cela permet de réduire la détection de collisions à un calcul de distance pour la majorité des objets qui s'éloignent.

Les objets qui se déplacent ont donc des propriétés particulières qu'il est possible d'exploiter afin d'optimiser le temps d'exécution en éliminant des calculs. Ces propriétés sont grandement exploitées dans le logiciel Arka 3D où les déplacements se font de manière linéaire. C'est alors une méthode intéressante de réduire à un seul produit scalaire ou un calcul de distance un maximum de détection de collisions entre deux objets.

1.4.10 Conclusion

Les algorithmes de détection de collisions de base sont très lourds en temps d'exécution lorsque les objets sont complexes et lorsque l'environnement comporte plusieurs objets. C'est pourquoi des optimisations sont nécessaires.

Ces optimisations utilisent des propriétés mathématiques et des propriétés des objets afin de déduire rapidement si une collision est possible ou non. L'objectif est donc de faire une pré-détection afin de réduire au minimum le nombre de détections complètes qui sont très laborieuses.

Les optimisations utilisent divers concepts comme la simplification des objets 3D, déterminer des limites, diviser l'espace, la projection et la cohérence temporelle. Ces techniques peuvent être combinées pour réduire le nombre de détections exactes.

Dans le logiciel Arka3D qui fait l'objet du présent document, où le temps d'exécution est primordial, des optimisations sont nécessaires. Plusieurs

concepts exprimés dans ces dernières sections sont utilisés conjointement afin de réduire au maximum ce temps d'exécution sans affecter la précision de la détection de collisions.

Les optimisations sont donc nécessaires afin de détecter les collisions dans un temps raisonnable.

1.5 Gestion de forces et d'événements

1.5.1 Introduction

Des forces sont présentent dans les environnements 3D et elles influencent le déplacement des objets. Ces forces peuvent changer la trajectoire des objets et elles affectent alors la détection de collisions. La gravité, par exemple, est l'une de ces forces. La détection de collisions devra alors tenir compte de ces forces.

La gestion des événements est un complément de la détection de collisions. Elle permet, entre autres, de réagir à une collision et de produire des effets sur les objets. Lorsqu'une collision se produit, un événement est généré. Dans un environnement 3D, la gestion d'événements permet de simuler les effets de la collision afin qu'ils soient intégrés dans les prochaines détections.

Les forces et événements décrits dans ce document sont :

- Les champs de forces.
- Attraction et évitement.
- Rebondissement et déviation (Événements)

Les forces influencent donc le mouvement des objets et les collisions génèrent des événements qui influencent également le mouvement des objets. Ces éléments doivent donc s'intégrer dans la détection de collisions.

1.5.2 Les champs de forces

Le champ de forces est la force la plus simple. Il s'agit d'un endroit dans la scène où les objets sont soumis à une force dont la direction est invariable.

L'exemple typique est la gravité qui attire les objets vers le sol. Cette force influence la direction et la vitesse des objets. Des champs de forces peuvent aussi représenter le vent ou un champ magnétique dont la provenance est éloignée. Les objets soumis à cette force auront tendance à se diriger dans la direction du champ. Cette direction peut être calculée par une somme pondérée de vecteurs.

Ces champs de forces changeront donc la vitesse et la direction des objets et la détection de collisions doit alors en tenir compte.

1.5.3 Attraction et évitement

L'attraction et l'évitement sont des forces qui sont générées à partir de points précis. Contrairement aux champs de forces, la direction de la force varie selon la position des objets.

La force d'attraction a pour effet d'attirer un objet vers un point ou un objet particulier. La direction de l'objet attiré est donc changée graduellement en direction de l'attraction. Pour déterminer le point d'attraction d'un objet attirant, le centre de gravité est utilisé. Dans un autre ordre d'idée, cette force augmente grandement la chance de collision.

La force d'évitement agit à l'inverse de la force d'attraction. Elle repoussera les objets plutôt que de les attirer. Elle est souvent utilisée comme une forme d'intelligence artificielle pour tenter d'éviter des collisions. Les points d'évitement sont rarement basés sur le centre de gravité de l'objet répulsif comme pour la force d'attraction. Celui-ci pourrait faire dévier l'objet tardivement ou aucunement si l'objet se déplace directement sur le centre de gravité. Le vecteur d'évitement sera donc calculé à partir du point d'intersection ou de la normale du plan tangent de l'objet 3D [26]. La figure 1.17 présente visuellement les deux méthodes.

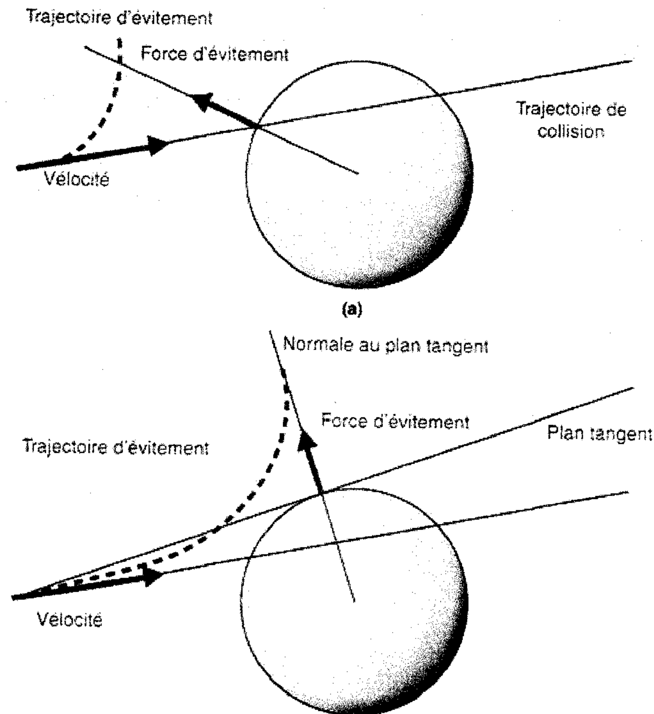


Figure 1.17. Trajectoire d'évitement [1] : (a) Déterminer une trajectoire d'évitement à partir du point d'intersection. (b) Déterminer une trajectoire d'évitement à partir du plan tangent.

La figure 1.17 montre que les deux méthodes peuvent être efficaces et que la méthode basée sur le point d'intersection (figure 1.17 (a)) diverge plus rapidement que la méthode basée sur le point tangent (1.17 (b)). Le taux de divergence désiré est donc un bon indice dans le choix de la méthode.

La force d'attraction fait donc converger les objets dans sa direction et la force d'évitement les fait diverger. Dans un contexte de détection de collisions, ces forces poussent les objets à entrer en collision ou de les éviter. Ces forces sont utiles dans le domaine de l'intelligence artificiel pour, entre autres, simulés des phénomène comportemental d'attirance et de répulsion.

1.5.4 Rebondissements et Déviations

Dans un environnement simulé, les conséquences d'une collision doivent être prises en compte. Les effets résultant d'une collision sont variés : changements de direction, déformations, bris, etc. Suite à une détection de collisions positive, aussi appelé un événement, il faut appliquer ces divers effets. Les rebondissements et les déviations font parties des conséquences des collisions.

[1]. Laurend Testud, Le programmeur : DirectX 9, Programmation des jeux 3D, CampusPress, 2003, page 185.

Les rebondissements et les déviations sont directement liés à la détection de collisions. Lorsqu'une collision se produit, les deux objets qui entrent en collision changent de direction. C'est ce phénomène qui est simulé par le rebondissement et la déviation.

Le rebondissement sur un plan est simple si la normale du plan est connue. Ce rebondissement correspond à une réflexion de vecteur tel que schématisé à la figure 1.18.

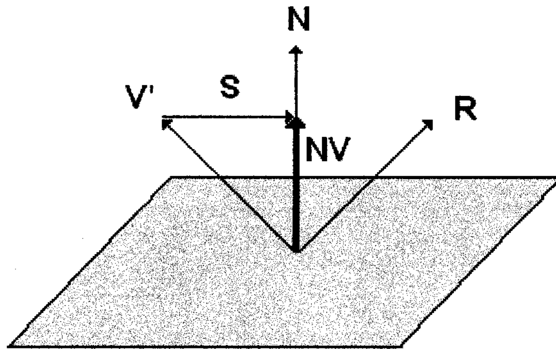


Figure 1.18. Réflexion d'un vecteur sur un plan.

Dans la figure 1.18, nous pouvons observer les divers vecteurs soient :

- le vecteur V' qui est l'inverse du vecteur incident qui représente la direction d'origine d'un objet,
- le vecteur N qui est la normale du plan de réflexion,
- le vecteur R qui est le vecteur réfléchi du vecteur V' et correspond à la direction du rebondissement.

Le vecteur NV sera la projection du vecteur V' sur le vecteur N à l'aide d'un produit scalaire soit $N \times (N \cdot V')$.

Le vecteur S est la différence entre le vecteur V' et NV.
Le vecteur R est donc le vecteur NV additionné au vecteur S.

Finalement, $R = (NV) + (S)$ (2)

$$R = (N \times (N \cdot V')) + (N \times (N \cdot V') - V')$$

$$R = 2\,N\,(N \cdot V') - V'$$

Cette formule permet donc de calculer la réflexion d'un vecteur et elle est utilisée pour les rebondissements. De plus, cette formule est utilisée dans les moteurs 3D afin de simuler la réflexion de la lumière sur les polygones [27] .

Un objet qui entre en collision avec un objet de masse inférieure ou avec un objet qui sera détruit ne fera que dévier et ne rebondira pas nécessairement. La force appliquée est alors déduite à partir du point de collision, comme le

rebondissement. L'énergie cinétique est aussi prise en compte et elle est le produit de la masse et de la vélocité.

Les déviations et les rebondissements sont donc une suite logique à la détection de collisions. Ces forces sont un effet engendré par la collision afin d'éviter la pénétration entre deux objets. Le rebondissement, déduit à partir d'une réflexion de vecteur, est surtout utilisé lorsque l'objet frappé est inerte. La déviation est utilisée lorsque les deux objets sont en mouvement et de masse différente. Il s'agit alors de force que l'on applique sur les objets. Ces deux effets sont largement utilisés dans le logiciel Arka 3D.

1.5.5 Conclusion

Il faut donc tenir compte des forces qui agissent dans l'environnement afin d'ajuster la détection de collisions. Les champs de forces et les forces d'attraction et d'évitement sont susceptibles de changer la trajectoire des objets et de modifier les collisions.

De plus, les objets ont souvent la contrainte de ne pas pouvoir se pénétrer les uns les autres. Alors, suite à une détection de collisions positive, des forces devront être appliquées sur les objets afin qu'ils rebondissent ou dévient. C'est la simulation de l'effet découlant de la collision qui est créée et elle est appelée un événement.

Ces forces, qui peuvent changer les trajectoires des objets avant ou après une collision, doivent faire partie intégrante de la détection de collisions.

1.6 Algorithmes existants

1.6.1 Introduction

Les différentes techniques de détection de collisions vues dans les sections précédentes sont utilisées conjointement afin de créer un algorithme de détection de collisions adapté au besoin. Les techniques utilisées varieront avec l'environnement dans lequel ces techniques évoluent et les besoins des utilisateurs.

Certains travaux visent à trouver des algorithmes de détection de collisions qui seront adaptés à un très grand éventail d'environnements et de besoins. Ces algorithmes existants tentent de réaliser la détection de collisions dans un temps acceptable pour ces divers environnements.

Dans cette section, cinq algorithmes de détection de collisions existant sont présentés :

- I-Collide [18].
- Rapid [19].
- V-Collide [28].
- V-Clip [24].
- SOLID [32].

Ces algorithmes sont utilisés dans divers domaines comme dans la robotique et la réalité virtuelle. L'efficacité de chacun des algorithmes varie grandement avec les environnements.

Voici les caractéristiques principales des environnements sur lesquels l'efficacité des algorithmes est jugée :

- Le nombre d'objets.
- Le nombre de polygones des objets.
- La densité de l'environnement (espace occupé versus espace disponible).
- La vélocité des objets.
- La vitesse de rotation des objets.

Ces caractéristiques permettent de comparer les algorithmes puisque certains d'entre eux sont plus influencés que d'autres par celles-ci. La comparaison des algorithmes se fait donc principalement sur ces critères.

1.6.2 I-Collide

I-Collide est un algorithme simple qui permet de faire une première détection rapide afin d'éliminer rapidement des paires d'objets. Il est basé sur les boîtes AABB vue dans la section 1.4.3 [18].

L'algorithme I-Collide a une architecture qui est principalement composée de quatre étapes comme le présente la figure 1.19.

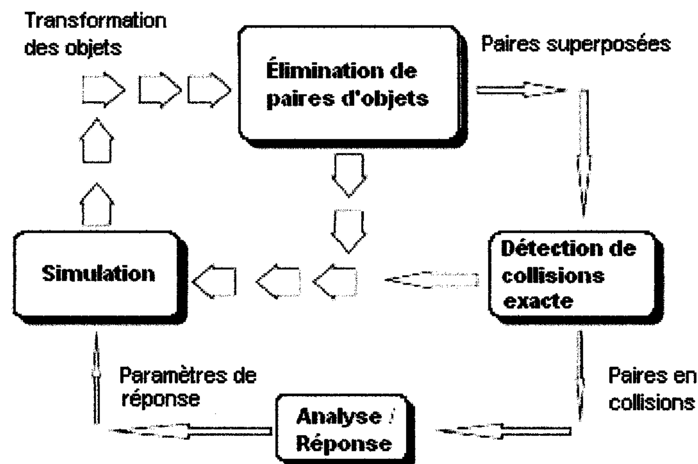


Figure 1.19. Architecture de détection de collision de I-Collide.(1)

Les étapes sont donc :

- La simulation de l'environnement.
- L'élimination de paires à l'aide des boîtes AABB.
- Détection exacte pour les paires dont les boîtes sont superposées.
- Analyse et réponse à la collision (gestion d'événements) pour la prochaine simulation.

Modèle d'objets de I-Collide

L'algorithme I-Collide utilise sa propre structure de modèle d'objets. Les objets doivent donc être convertis en ce modèle afin d'appliquer l'algorithme. La conversion se fait en un temps raisonnable en début de traitement dans le cas des modèles non déformables. Pour les modèles déformables, la conversion des objets vers le modèle I-Collide doit se faire à toutes les itérations et les résultats sur le temps d'exécution sont désastreux.

Le modèle d'objet I-Collide permet principalement de déterminer les points adjacents à un point donné d'un objet. Cela est particulièrement utile lors de la construction des boîtes AABB.

Construction des boîtes AABB

Suite à la conversion des objets en modèle d'objet I-Collide, les boîtes AABB sont calculées. À la première itération, les maximums et minimums des coordonnées x, y, z des sommets des polygones composant chaque objet sont déterminés en balayant chacun des points de ceux-ci. Aux itérations suivantes, les maximums et les minimums sont testés. Pour faire le test, les points adjacents du point extrême (minimum ou maximum) sont comparés afin de vérifier si le point est toujours extrême. Le test est appliqué de manière récursive

(1) Traduction de Architecture for Multi-body Collision Detection, I-Collide [18]

afin de trouver le nouveau point extrême. Cette méthode permet de trouver les nouveaux minimums et maximums, en utilisant la cohérence temporelle, sans avoir à balayer l'ensemble des points comme à la première itération.

Une fois les extremums des coordonnées x, y, z trouvés, ces valeurs constitueront les boîtes AABB. Les boîtes sont donc définies à l'aide de 6 vecteurs qui constituent les bordures d'un parallélépipède rectangle.

Détection de collisions des boîtes AABB

C'est avec les maximums et les minimums que la superposition des boîtes AABB est déduite. Pour ce faire, un tri de type « Quick Sort » est utilisé.

Chacune des dimensions, soient les coordonnées x, y ou z, de tous les objets sont placés dans une liste. Les valeurs des maximums et des minimums de cette liste sont associées au modèle de chaque objet. La déduction des paires qui se superpose se fait à partir du tri. La figure 1.20 présente cette déduction.

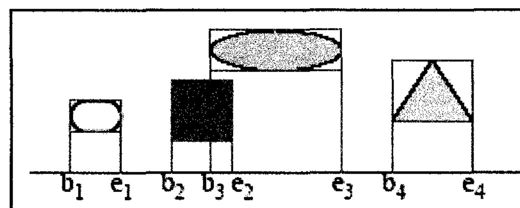


Figure 1.20 Boîtes superposées dans une dimension.

Dans cette figure, tous les objets sont représentés par leur minimum (b) et leur maximums (e). Si le minimum et maximum de l'objets sont un à la suite de l'autre dans la liste triée, l'objet n'est donc pas superposé avec aucun autre objet. Dans le cas contraire, les objets qui sont présents, par le maximum et/ou le minimum, entre les valeurs de l'objet concernés sont ajoutés à une liste de paires d'objets superposés pour cette dimension. L'opération sera répétée pour chacune des deux autres dimensions.

Les objets qui se superposent dans les trois dimensions ont donc des boîtes qui se superposent et ils sont placés dans une liste de paires d'objets et ils sont soumis à une détection de collisions plus précise.

Détection de collisions exacte

La détection exacte permet de déterminer précisément si les paires d'objets sont réellement en collision. La première étape de l'algorithme I-Collide ne permet que de déduire si les paires sont à risque d'entrer en collision.

L'algorithme I-Collide propose la méthode des pseudo région de Voronoï, il s'agit de l'algorithme de Lin-Canny [29]. Cet algorithme relativement instable est résumé dans la section 1.6.4

Il est également possible d'utiliser I-Collide de manière récursive en divisant le modèle, mais il est préférable d'utiliser par la suite les boîtes OBB comme dans V-Collide dû à la rapidité de convergence de l'approximation par ces boîtes. Les boîtes OBB ont un volume toujours inférieur ou égal aux boîtes AABB, comme présenté dans la section 1.4.3, ce qui réduit le nombre d'itérations lors de la détection de collisions.

Conclusion

L'algorithme I-Collide est donc une bonne pré-détection rapide afin d'éliminer rapidement des paires d'objets. Il faut par contre ajuster la détection de collisions exacte selon les besoins. L'efficacité de la technique de détection de collisions se limite par contre aux objets convexes et non déformables.

1.6.3 Rapid

L'algorithme de détection de collisions Rapid est basé sur les arbres de boîtes orientés (OBBTree). Il constitue une routine de détection de collisions complète et flexible permettant de trouver précisément les points de collisions entre deux ou plusieurs objets. Cet algorithme est particulièrement efficace lorsque les objets de l'environnement sont très rapprochés.

Voici les étapes de la détection de collisions à l'aide de l'approche OBB-tree.

1. Construction des boîtes OBB sur les parties d'objet à détecter.
2. Vérifier deux à deux les boîtes qui se superposent.
3. Diviser les boîtes qui se superposent en deux et recommencer l'étape 1 à 3.
4. Les boîtes indivisibles sont considérées comme en collision.

Modèle d'objets utilisé

Le modèle utilisé par Rapid est le plus commun soit l'ensemble de polygones non triés (« Polygon soup »). Cela rend l'algorithme très flexible puisqu'il peut être appliqué facilement, sans conversion de modèle d'objets, à la majorité des environnements.

Construction des boîtes OBB

Le calcule des boîtes OBB est numériquement et sémantiquement complexe. Un des principaux apports au monde scientifique de Rapid est sa technique innovatrice pour calculer ces boîtes.

En se basant sur un algorithme de déduction de la plus petite ellipsoïde englobante de Welzl [30], Rapid est un algorithme d'ordre linéaire qui permet de déduire rapidement la boîte OBB englobante minimale. Cet algorithme permet de déduire trois vecteurs qui constituent les axes principaux de la boîte.

Par contre, contrairement à I-Collide, l'ensemble des points d'un objet sert à la construction des boîtes puisque la cohérence temporelle n'est pas exploitée dans cet algorithme.

La construction d'un boîtes OBB est donc beaucoup plus laborieuse en temps machine que la boîte AABB.

Détection de collisions des boîtes OBB.

La méthode habituelle pour détecter si deux boîtes orientées se superposent est de vérifier si l'un des coins pénètre l'une des faces de l'autre boîte. Cette méthode est par contre d'ordre trop complexe ($O(n^2)$) comparativement au nombre de fois qu'elle est appelée.

Une approche développée par un des créateurs de Rapid, S. Gottschalk, permet de déduire plus simplement si deux boîtes OBB sont en collisions. Il s'agit du théorème de l'axe de séparation [31].

Cette technique permet de trouver un axe de séparation entre deux boîtes OBB avec une complexité linéaire. Quinze projections basées sur l'angle des boîtes OBB permettent alors de déduire exactement si un axe de séparation existe. Si aucun des axes de séparation ne permet de séparer les deux boîtes, celles-ci sont en collisions. Si un axe de séparation est trouvé, les boîtes ne sont pas en superposition et l'algorithme arrête sans compléter les autres projections.

Pour vérifier si les projections des boîtes englobantes se superposent ou non, une technique se basant sur le centre et le rayon de la boîte est employée. Le rayon est un vecteur partant du centre vers le coin de la boîte. La figure 1.21 présente la technique.

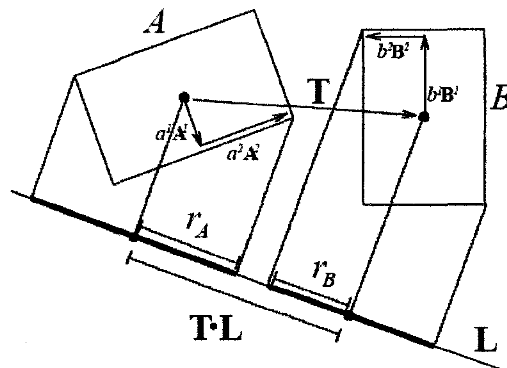


Figure 1.21. Projection des boîtes OBB sur un axe.[1]

Le vecteur T, qui est le vecteur qui relie le centre des deux boîtes, est projeté sur l'axe correspondant au vecteur L. Les rayons sont également projetés sur l'axe L et la valeur projetée la plus grande de chacune des boîtes est conservée. Si la projection du vecteur T est plus grande que la somme des deux rayons, les boîtes ne se superposent pas sur cet axe. Sinon les boîtes sont considérées comme superposées.

Un maximum de quinze projections comme celle-ci sont calculées afin de déduire la collision entre deux boîtes. Lorsqu'une des projections ne produit pas une superposition, les autres projections ne sont pas calculées parce que les boîtes ne sont pas en collision.

Étant donné que la détection de collisions par boîtes OBB est utilisée très souvent, l'optimisation des calculs de cette détection influence grandement la vitesse d'exécution de l'algorithme.

Division de boîtes OBB

Pour raffiner la précision de la détection de collisions, il faut alors poursuivre la construction de l'arbre OBB par la division successive des objets 3D. L'algorithme Rapid utilise encore une fois les boîtes OBB qui sont déjà construites lors de la détection de collisions.

Une fois la première estimation de la collision déduite, la même boîte utilisée pour faire la détection de collisions grossière est divisée en deux sur sa longueur. Les polygones sont alors acheminés vers deux nouvelles structures selon leur appartenance à l'une ou l'autre des divisions. La figure 1.22 présente le cheminement de OBBtree.

[1] S. Gottschalk, M. C. Lin et D. Manocha, OBBTree: A Hierarchical Structure for Rapid Interference Detection Environments.

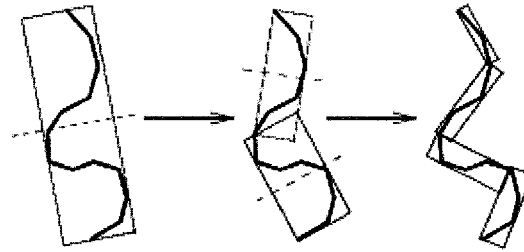


Figure 1.22. Division successive de OBBtree.[1]

À chaque itération, les boîtes OBB sont divisées en deux ce qui permet de se rapprocher rapidement de l'objet d'origine et d'ainsi améliorer la précision de la détection de collisions.

Si la boîte OBB est indivisible sur sa longueur, la division sur la largeur est alors tentée. Si la boîte est également indivisible sur la largeur, la collision pour l'objet complet est alors déduite et la section indivisible représente donc un point de collision. L'arbre OBB est alors remonter complètement afin de déterminer quels objets sont en collisions.

Conclusion

Bien que OBBTree semble très lourd comme algorithme, les optimisations faites par l'équipe qui a développé Rapid sont intéressantes. Les résultats de cette détection de collisions sont impressionnants [34] aux niveaux de leur stabilité et rapidité.

1.6.4 V-Collide

L'algorithme V-Collide est conçu pour le langage de modélisation de réalité virtuelle VRML 2.0. Il s'agit d'un module qui permet d'intégrer les collisions dans cet environnement. V-Collide est une fusion des algorithmes I-Collide et Rapid et il est également conçu par les mêmes auteurs.

La figure 1.23 présente le cycle de V-Collide et comment il s'intègre à VRML.

[1] S. Gottschalk, M. C. Lin et D. Manocha, OBBTree: A Hierarchical Structure for Rapid Interference Detection Environments.

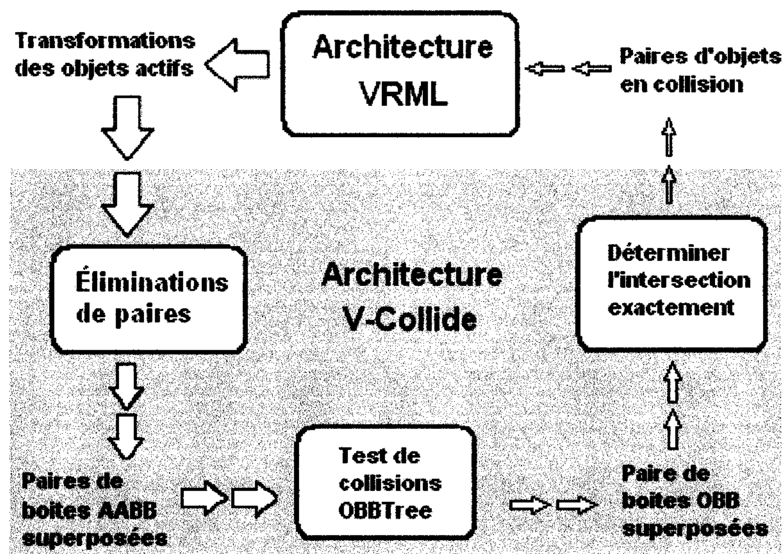


Figure 1.23. Architecture de V-Collide. (1)

La détection est faite en trois étapes :

1. Élimination rapide de paires d'objets à l'aide des boîtes AABB.
2. Test de collision à l'aide des OBBTrees.
3. Déterminer l'intersection exactement.

Étapes de détection de V-Collide

Cet algorithme utilise donc les boîtes AABB pour détecter rapidement les paires d'objets susceptibles d'entrer en collisions. Cette technique est très utile lorsque les objets sont dispersés.

Une fois que les paires d'objets à risque sont déterminées à l'aide des boîtes AABB, ces paires d'objets sont ensuite testées à l'aide de l'algorithme OBBTree exactement comme dans Rapid.

Les collisions exactes sont ensuite déterminées afin de trouver les objets et les régions exactes qui sont entrées en collision.

Modèle d'objets pour V-Collide

Le modèle d'objets de V-Collide doit s'intégrer parfaitement à ceux de VRML ce qui provoque quelques limitations et conversions.

(1) Traduction de The system architecture of V-Collide [28].

La structure des modèles est donc générale et une conversion est donc nécessaire pour chacun des objets. De plus, VRML supporte les surfaces implicites comme les quadratiques. Ces modèles ne sont pas composés de polygones. Il faudra donc générer ces polygones.

Dans VRML, la cohérence temporelle n'est pas applicable étant donné que les objets peuvent se déplacer aléatoirement selon l'utilisateur. L'optimisation faite pour la construction des boîtes AABB de I-Collide est alors beaucoup moins efficace.

Les modèles d'objets de VRML doivent contenir l'information sur les paires d'objets dont la collision est possible. Étant donné que cette information est d'ordre N^2 selon le nombre d'objets, l'algorithme est très lourd en espace mémoire.

Les limitations de VRML, qui sont tout de même très importantes afin de créer un standard dans le monde de la réalité virtuelle, réduisent considérablement l'efficacité de cet algorithme.

Conclusion

L'algorithme V-Collide est donc un bon mélange des détections I-Collide et OBBTree. Il utilise la détection rapide des paires d'objets de I-Collide et la convergence rapide vers le point de collision de OBBTree.

Il est principalement utilisé pour le langage de modélisation de réalité virtuelle VRML 2.0. Cet environnement limite l'efficacité par contre de V-Collide.

1.6.5 V-Clip

L'algorithme Voronoï Clip (V-Clip) utilise les régions de Voronoï. Il est une amélioration de la technique de détection du point le plus proche de Lin-Canny [29] utilisé dans I-Collide.

Bien que cet algorithme ne possède pas de processus d'élimination de paires comme dans I-Collide, il propose des solutions simples aux lacunes de la technique Lin-Canny.

L'algorithme Lin-Canny utilise seulement les régions de Voronoï des sommets. Cela cause certains problèmes de convergence. En proposant un automate qui distingue les relations entre les faces, les arrêtes et les sommets, V-Clip permet d'optimiser chacune de ces relations. La figure 1.24 présente cet automate.

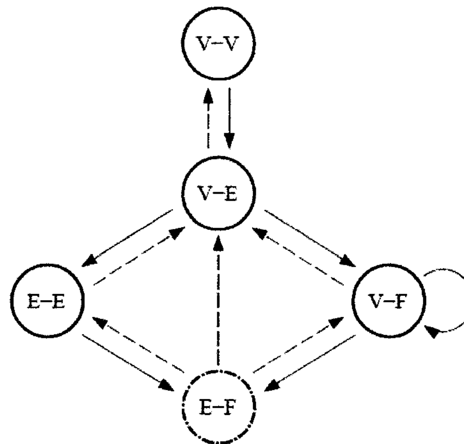


Figure 1.24. Transitions des états dans V-Clip. (1)
Où : V est un sommet,
E est une arrête et
F est une face.

La figure 1.24 montre les transitions possibles entre les états. Le but de l'automate est bien sûr de trouver les deux surfaces (faces (F), arrêtes (E) ou sommets (V)) les plus rapprochées. Les flèches pleines indiquent une réduction de distance tandis que les flèches pointillées correspondent à une distance équivalente. Les états V-V, V-E, E-E et V-F peuvent être à la fois des états initiaux et des états finaux. E-F est un état spécial, quand l'automate déduit que les surfaces les plus rapprochées sont une arrête et une face, cela indique que des objets se pénètrent.

Dans la technique Lin-Canny, il est impossible de déterminer avec certitude si deux objets se pénètrent. Si cette situation se présente, l'algorithme boucle indéfiniment puisque les régions de Voronoï se croisent à l'intérieur des objets. La solution apportée par Lin-Canny est un compteur qui arrête l'algorithme après un certain nombre d'itérations ce qui déduit qu'une collision est survenue. Cette boucle interrompue est désastreuse au niveau du temps d'exécution.

L'automate proposé par V-Clip permet donc de déterminer quand deux objets se pénètrent et ce en vérifiant que l'état E-F possède la plus petite distance. Dans ce cas, afin de terminer l'algorithme, un des objets en collision est découpé (« Clipping »). Le sommet coin pénétrant identifié à l'aide de l'arrête E est alors coupé par le plan déduit de la face F. L'automate poursuit ensuite le traitement à l'aide de l'automate afin de sortir de l'état E-F et de terminer sont cheminement vers les surfaces les plus proches. Il est possible que plusieurs sommets pénètrent les deux objets. De plus, il est également possible d'adapter l'algorithme afin de calculer précisément le volume d'intersection.

(1) States and transitions of the V-Clip algorithm. [24]

L'algorithme V-Clip est donc une autre application des régions de Voronoï. Étant donné que l'algorithme général boucle indéfiniment lorsque les objets se pénètrent, V-Clip propose une méthode simple de détecter et de découper les objets afin que le processus des régions de Voronoï puisse se terminer adéquatement. Bien que la technique de V-Clip puisse sembler lourde à cause de l'automate et les calculs nécessaires au découpage, elle donne des résultats bien supérieurs à sont prédécesseur, la technique de Lin-Canny.

1.5.6 SOLID

La librairie SOLID utilise la géométrie constructive (CSG) vu dans la section 1.2.3. Cette librairie est conçue pour fonctionner avec OpenGL et elle est bien plus qu'un algorithme de détection de collisions. Elle supporte le mouvement et les événements suite à une collision. En plus d'exploiter la cohérence temporelle, la librairie SOLID est le seul algorithme présenté qui supporte implicitement les modèles déformables.

L'algorithme utilisé par SOLID s'exécute en quatre étapes :

- Élimination rapide de paires d'objets à l'aide de boîtes AABB.
- Estimation de la distance entre les paires d'objets rapprochés.
- Calcul des points de contact et des volumes d'intersection.
- Gestion des événements de collisions.

Modèles objets de SOLID

L'algorithme SOLID utilise les modèles CSG construits à l'aide de boîtes, de cônes, de cylindres, de sphères et également d'autres polyèdres complexes prédéterminés. Les objets polygonaux sont donc convertis en objets CSG en début de traitement. Cette opération réduit sensiblement le raffinement de la représentation des objets.

La modélisation de SOLID permet également d'ajouter des paramètres de déplacement et de déformation qui sont appliqués dans la scène. De plus, ces modèles sont liés avec OpenGL pour le rendu graphique.

Élimination de paires d'objets

La détection de collisions de CSG est plutôt lourde étant donné que la décomposition des objets en formes simples entraîne plusieurs détections de collisions. Les boîtes AABB sont alors utilisées exactement comme dans I-Collide et V-Collide afin d'éliminer rapidement les paires d'objets éloignés. Dans ce cas, la distance est estimée à l'aide de ces boîtes.

Estimation de la distance

Une fois les paires d'objets éloignés éliminées, les paires rapprochées sont soumises à la détection GJK [33] adaptée au besoin de SOLID. Cette détection permet de déterminer la distance entre deux objets. Cela permet également de déterminer si les deux objets CSG sont en collisions.

Calcul des points de contact et du volume d'intersection

Lorsque deux modèles sont considérés comme étant en collisions, les points de contact sont déterminés à l'aide d'opérations d'intersection sur les modèles CSG. À partir de ces opérations, les volumes d'intersection sont également déduits à l'aide d'addition des volumes en intersection comme présenté dans la section 1.1.3.

Gestion d'événements de collisions

Les modèles objets de SOLID permettent de spécifier comment réagir à une collision. La masse, la porosité et même la déformation ont donc un effet sur la réaction suite à la collision. Cet événement modifiera le comportement des objets lors de la prochaine itération de l'algorithme. De plus, étant donné que le mouvement des objets est connu, plusieurs optimisations se basant sur la cohérence temporelle sont appliquées.

Conclusion

La librairie SOLID est beaucoup plus qu'un algorithme de détection de collisions. Elle permet en plus de gérer l'affichage graphique des objets et les événements comme le mouvement, la déformation et les collisions.

Cette librairie est donc un outil complet qui offre des informations abondantes sur les objets en intersection. Par contre, cette vaste gamme de fonctionnalités à bien sûr une incidence sur le temps d'exécution qui est sensiblement augmenté.

1.6.7 Comparaisons entre les algorithmes

La communauté scientifique travaillant sur les algorithmes de détection de collisions a de la difficulté à s'entendre sur une série de tests qui permettraient de déterminer quel algorithme est le plus efficace. La raison étant qu'il existe un nombre presque illimité d'environnements dans lesquels ces algorithmes peuvent évoluer. L'efficacité de ces algorithmes peut varier grandement d'un environnement à l'autre.

Une expérimentation intéressante a été faite afin de comparer les algorithmes vus précédemment dans un environnement de planification de mouvements probabilistes [34]. Cette expérimentation concorde assez bien avec les résultats d'autres expérimentations réalisées avec des environnements différents.

L'expérimentation compare les algorithmes V-Collide, Rapid, Solid, V-Clip et PQP. Tous ces algorithmes sont présentés dans le chapitre présent à l'exception de PQP (Proximity Query Package) [35].

L'algorithme PQP utilise un technique de balayage d'arbres de sphères (SphereTree) afin de déterminer la distance entre deux objets. Il a été développé par le même groupe de travail ayant conçu Rapid.

Quatre tests sont utilisés dans l'expérimentation afin de déterminer l'efficacité des algorithmes dans un contexte de planification de mouvement. La figure 1.25 présente graphiquement les tests effectués.

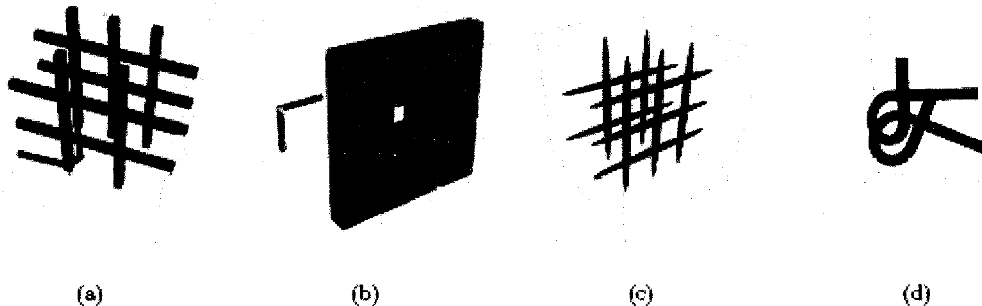


Figure 1.25. Environnement d'expérimentation des algorithmes de détection de collisions. (a) Test 1 : Planification de mouvement dans une grille simple. (b) Test 2 : Planification de mouvement à travers un trou. (c) Test 3 : Planification de mouvement dans une grille complexe. (d) Test 4 : Résolution d'un casse-tête.

Le premier test présenté dans la figure 1.25 (a) consiste à déterminer un chemin sans collision, pour un objet composé de trois rectangles, à travers une grille également composée de rectangles afin d'atteindre un endroit précis qui est choisi aléatoirement. La figure 1.26 présente le temps d'exécution des différents algorithmes de détection de collisions pour résoudre dix fois ce test.

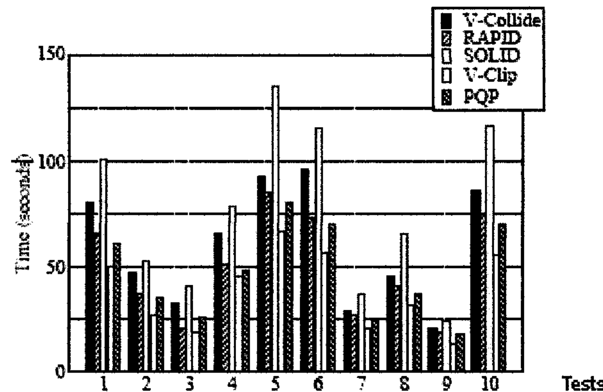


Figure 1.26. Temps d'exécution des différents algorithmes de détection de collisions pour résoudre dix fois le test de planification de mouvement dans une grille simple.

L'algorithme V-Clip est donc le plus rapide pour cette problématique suivie de près par PQP et Rapid. V-Collide est plus lent étant donné qu'il génère deux types de modèle (boîtes AABB et OBB) afin de trouver la solution. SOLID est loin derrière même si les objets modèles présentés font partie de sa librairie de formes. Les diverses formes utilisées dans SOLID sont plus lourdes en temps de calcul que les formes simples utilisées dans les autres algorithmes.

Le second test présenté dans la figure 1.25 (b) est également une prévision de chemin mais cette fois-ci, l'objet qui se déplace doit passer au travers d'un trou. La figure 1.27 présente les comparaisons entre les tests 1 et 2.

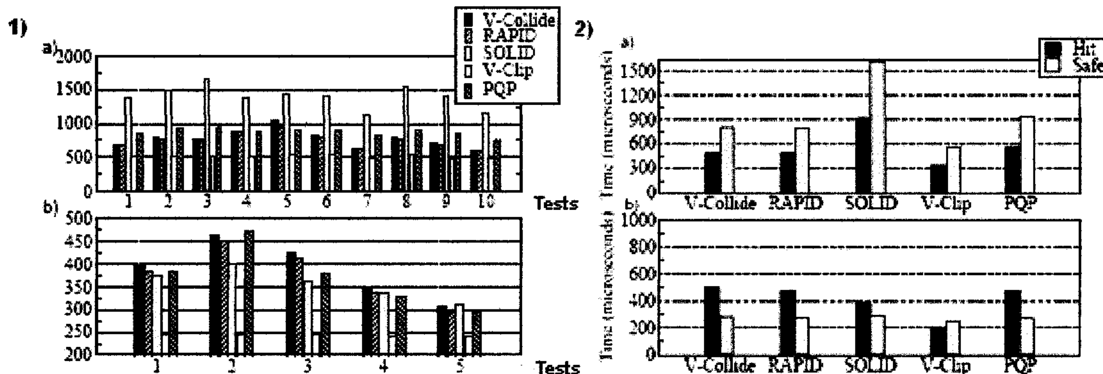


Figure 1.27.
Comparaison entre les tests 1 et 2 présentés aux figures 1.25(a)et (b).
1) Comparaison des temps d'exécution pour le test 1 en a) et pour le test 2 en b).
2) Comparaison des temps d'exécution moyen en cas de collision (Hit) et de non collision (Safe) pour le test 1 en a) et le test 2 en b).

Le test 2 est donc réalisé plus rapidement que le test 1 pour l'ensemble des algorithmes comme le décrit bien les figures 1.27 1 (a) et 1 (b). La raison étant que beaucoup plus d'espace est inoccupé dans le test du trou, la densité

de l'environnement est donc plus petite. De plus, l'objet, qui est formé de seulement deux boîtes et qui doit parcourir le chemin est plus simple comparativement au test 1 qui en contient trois.

La densité de l'environnement explique également pourquoi la majorité des algorithmes affichent un temps d'exécution nettement plus petit pour les tests de non collisions. La technique V-Clip fait exception et ne semble pas être affectée par la densité de l'environnement. La performance de l'algorithme SOLID dans le test 2 est probablement due à l'utilisation des boîtes AABB qui représente bien la structure environnante du trou. L'algorithme V-Clip donne encore la meilleure performance pour le test 2 mais il est suivi de près par les autres techniques.

Le test 3 (figure 1.25 (c)) est sensiblement le même que le test 1 mais la grille à été complexifiée par l'ajout de triangles qui sont au nombre de 8900. La figure 1.28 présente une première comparaison entre les temps d'exécution du test 1 et du test 3 sur la base de l'occurrence de collisions et une seconde dans le cas de non collision.

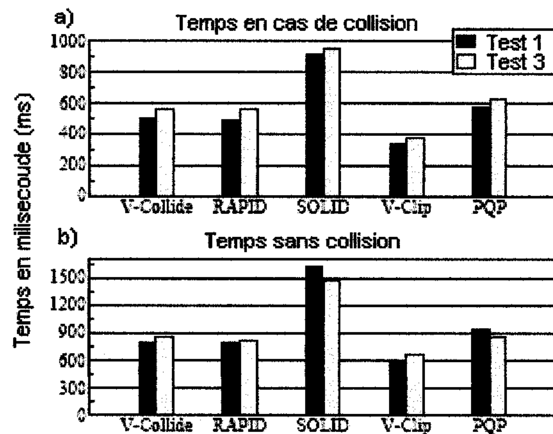


Figure 1.28. Comparaison entre les temps de collisions pour les tests 1 et 3.
a) Somme des temps en état de collision.
b) Somme des temps en état de non collision.

Le temps exécution est en général très légèrement supérieur pour le test 3 même si le modèle 3D de la grille est beaucoup plus complexe. La complexité des objets n'est donc plus un problème significatif pour la majorité des algorithmes de détection de collisions. SOLID et PQP donnent même de meilleurs résultats dans le test 3 que dans le test 1, ces deux algorithmes s'ajustent mieux aux objets sphériques.

Le test 4 présenté dans la figure 1.25 d) est un casse-tête formé de deux cylindres recourbés sur eux-mêmes. Ces deux cylindres forment alors des cercles non fermés et ils sont entrelacés l'un dans l'autre comme un maillon de chaîne. L'objectif du test est de séparer les deux objets et de défaire le maillon.

La figure 1.29 présente les temps d'exécution des algorithmes V-Collide, Rapid et PQP pour résoudre le casse-tête.

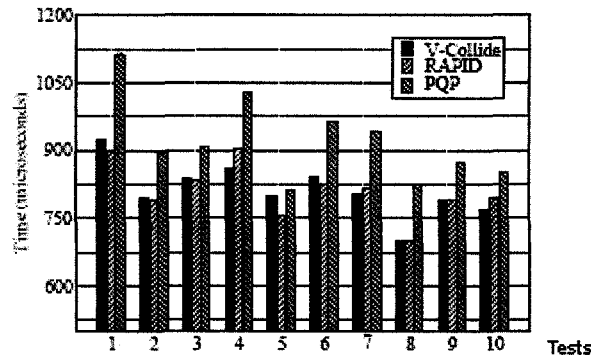


Figure 1.29. Temps exécution moyen pour une itération du test 4.

Les 3 algorithmes présentés dans le graphique de la figure 1.29 sont ceux qui donnent des résultats satisfaisants. L'algorithme RAPID et V-Collide donnent cette fois-ci les meilleurs résultats. L'optimisation des paires d'objets faites par V-Collide donne parfois des meilleurs résultats mais à d'autres moments, RAPID est plus rapide même s'il ne fait pas cette optimisation. La technique PQP suit légèrement derrière, tandis que SOLID, qui n'est pas présenté sur la figure 1.29, donne des temps cinq fois supérieurs aux trois algorithmes présentés. L'algorithme V-Clip, qui performe très mal avec les modèles non convexes, donne des résultats pitoyables.

Conclusion

Les tests sont réalisés dans un contexte de planification de mouvements probabilistes. Étant donné le grand nombre de détections réalisées pour résoudre chaque problème, ces tests donnent des comparaisons très intéressantes entre les algorithmes présentés. Des tests concernant de grands nombres d'objets et sur la densité des environnements sont par contre manquants.

L'algorithme de détection de collisions qui donne les meilleurs résultats dans la majorité des tests présentés est V-Clip. Cette technique utilisant les régions de Voronoï donne des temps d'exécution inférieurs pour les tests 1, 2 et 3. Par contre, celui-ci est incapable de résoudre dans un temps raisonnable le test 4.

Les techniques utilisées par V-Collide et RAPID donnent de bons résultats dans tous les tests, avec un léger avantage pour RAPID. De plus, ces deux algorithmes, qui utilisent les modèles polygonaux, fonctionnent avec tous les objets, même les objets incomplets et excessivement non convexes.

Finalement, pour un environnement qui possède des formes complètes qui sont convexes, il est possible d'utiliser V-Clip pour diminuer légèrement le temps d'exécution. Dans les autres environnements où les formes sont plus complexes (incomplètes ou non convexes), il est préférable d'utiliser des algorithmes plus robustes comme RAPID et V-Collide.

1.6.8 Conclusion

Le domaine d'application de la détection de collisions est très large. Les environnements dans lesquels évoluent les algorithmes de détection de collisions ont des caractéristiques qui influencent grandement leur rapidité.

Les algorithmes de détection présentés dans ce chapitre permettent de s'adapter à un vaste nombre d'environnements. Ils visent à donner de bons résultats dans tous ces environnements.

Les cinq algorithmes présentés dans ce chapitre possèdent chacun des techniques et des caractéristiques différentes. Certains utilisent les boîtes englobantes, d'autres les régions de Voronoï et aussi la géométrie constructive.

Les comparaisons faites à l'aide des tests de planification de mouvements probabilistes montrent que les boîtes orientées (OBB) utilisées dans V-Collide et RAPID donnent de bons résultats dans tous les environnements. Les régions de Voronoï de V-Clip sont plus efficaces pour la majorité des tests mais par contre elles ne sont pas applicables pour un environnement avec des modèles non convexes.

Les caractéristiques et les besoins de l'environnement doivent être examinées attentivement afin de choisir le meilleur algorithme de détection de collisions. Un algorithme hybride conçu pour un environnement particulier donnera par contre de bien meilleur résultat quoiqu'il ne soit pas portable à tous les environnements.

Dans le logiciel d'expérimentation Arka3D, un algorithme hybride est implémenté. Cet algorithme, utilise les boîtes englobantes, les régions de Voronoï et la géométrie constructive, optimise également des aspects caractéristiques de l'environnement.

Les techniques utilisées par les algorithmes existants sont donc efficaces et elles sont appliquées dans un algorithme adapté à l'environnement du logiciel Arka3D.

1.7 Conclusion

Cette revue de littérature a permis de faire un survol des différentes techniques de détection de collisions en relation avec l'environnement dans lequel elles évoluent. De plus, la gestion de forces et d'événements, comme le rebondissement, est également traitée.

Dans les premières sections, nous avons examiné les modélisations d'objets et les environnements avec lesquels les algorithmes de collisions seront utilisés. Certains algorithmes sont plus efficaces que d'autres avec certaines modélisations ou certains environnements. La compréhension de ces éléments est donc essentielle afin de choisir un algorithme de détection de collisions efficace pour la situation.

La détection de collisions est un problème de complexité. Bien qu'il soit possible dans tout les cas de déterminer exactement les collisions dans un environnement 3D, le temps d'exécution peut devenir énormément long dû aux nombreuses comparaisons lorsque les objets sont nombreux et complexes. C'est pourquoi la section 1.4 présente des optimisations de calculs applicables sur les algorithmes de détection de collisions. Entre autres, la subdivision des modèles, qui permet de faire des arbres de formes simples et d'éliminer un grand nombre de polygones traités par l'algorithme de détection de collisions, et les régions de Voronoï, qui servent à déterminer rapidement les points les plus proches entre deux objets.

Des forces influençant le mouvement peuvent aussi être présentes dans l'environnement. Par exemple, notons l'influence de la gravité et les champs de forces. De plus, les collisions entre deux objets génèrent également des forces afin de respecter les règles de non pénétrations. Nous parlons ici de rebondissements et de déviations. Ces forces sont donc essentielles pour la gestion d'interactions dans un environnement 3D.

De nombreux algorithmes de collisions sont développés par des groupes de recherche et peuvent être appliqués à des environnements variés. Dans la section 1.6, I-Collide, RAPID, V-Collide, V-Clip et SOLID sont présentés. Ils utilisent les techniques présentées dans la section 1.4. Ces algorithmes sont ensuite comparés lors d'un test de planification de mouvement. L'algorithme le plus rapide dans la majorité des tests est V-Clip mais il est incapable de résoudre en un temps raisonnable l'un des quatre tests. Nous retenons donc RAPID pour avoir réussi en un temps raisonnable à tous les tests mais surtout pour sa flexibilité en s'adaptant à tous les environnements et les modélisations présentés dans le présent document.

De ces algorithmes et ces techniques d'optimisation de détection de collisions présentés s'inspire l'algorithme de détection de collisions utilisé dans le logiciel Arka 3D présenté dans le chapitre 2. Ce logiciel utilise un approche hybride adapté à l'environnement afin d'avoir un temps d'exécution optimal.

Chapitre 2 : Modélisation

2.1 Introduction

Au cours du dernier chapitre, plusieurs méthodes permettant de déduire une collision et de réagir à cette collision ont été abordées. Certaines de ces méthodes peuvent être utilisées conjointement afin de créer un algorithme hybride de détection de collisions et de gestion d'événements.

Dans le cadre d'une expérimentation visant à appliquer ces méthodes, nous avons développé un jeu vidéo, nommé Arka3D, évoluant dans un environnement 3D. À l'aide des techniques présentées au chapitre 1, nous avons créé un algorithme de détection de collisions hybride et de gestion d'événements adapté au jeu développé. Le jeu a été développé à l'aide de Visual C++ et DirectX.

Ce jeu a permis de mettre en pratique plusieurs techniques de détection de collisions et de tester l'efficacité de nombreuses optimisations.

Dans un premier temps, une description du fonctionnement du jeu et de la méthode de développement seront présentées.

Ensuite, chacun des objets sera examiné individuellement à l'aide de sa structure. Par la suite, le fonctionnement des relations entre les objets, principalement les collisions, sera expliqué.

Un bon jeu vidéo possède bien sûr plusieurs effets visuels spéciaux. C'est pourquoi nous avons inclus une section sur les effets visuels et les événements qui les produisent.

Ce chapitre décrit donc l'ensemble de la conception graphique et algorithmique du jeu Arka3D.

2.2 Application développée (Explication du jeu)

2.2.1 Introduction

Le jeu développé est une version 3D du jeu Arkanoïd développé dans les années 1980 d'abord en version arcade et ensuite pour les consoles de jeu comme Nintendo.

Ce jeu, joué partout à travers le monde est maintenant devenu un classique. De nombreuses versions du jeu ont été développées sur ordinateur et elles sont disponibles gratuitement sur internet.

Figure 2.1 Première version du jeu Arkanoid.

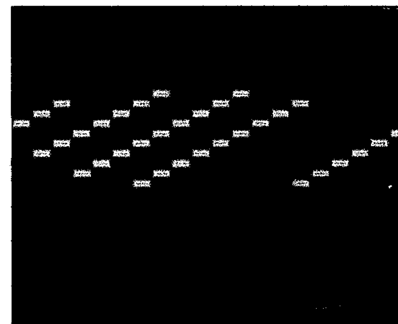


Figure 2.2 DX-Ball, une des nombreuses versions du jeu Arkanoid.

Les figures 2.1 et 2.2 présentent deux versions du jeu Arkanoïd. D'abord la version originale à la figure 2.1 développée par Taito en 1986 et une version gratuite nommé DX-ball développée en 1996 par un programmeur indépendant à la figure 2.2.

Le concept du jeu est assez simple. L'environnement du jeu est limité par des murs sauf pour la partie inférieure. Le jeu possède 3 éléments essentiels soient la balle, les cubes et le disque. Le but du jeu est de faire dévier la balle, qui est toujours en mouvement, à l'aide du disque sur un tableau de cubes afin de les détruire. Le disque est le seul élément que le joueur peut déplacer et il peut le faire seulement de manière horizontale afin de faire rebondir la balle. La balle rebondit également sur les murs. Si la balle passe derrière le disque et sort par la partie inférieure du jeu, le joueur perd une vie qu'il a en nombre limité. Lorsque le joueur n'a plus de vie, c'est la fin du jeu. Quand la balle détruit des cubes, le joueur amasse des points et lorsque l'ensemble des cubes sont détruits, le joueur passe au tableau de cubes suivant. L'objectif est donc d'accumuler un maximum de points.

Ce sont les règles générales du jeu auxquelles sont ajoutées souvent quelques options comme des cubes spéciaux et des bonus qui modifieront l'état du jeu, comme, par exemple, la vitesse de la balle et la taille du disque.

La version que nous avons développée est beaucoup plus complexe que son ancêtre. En effet, la dimension du jeu est passée de la seconde à la troisième dimension d'où le nom Arka3D. La détection de collisions, qui est beaucoup plus complexe dans un environnement 3D, a également été raffinée afin de tenir compte des collisions sur les coins des cubes, ce qui n'était pas implémentée dans la version originale. De plus, en 2D, les tableaux sont composés d'un maximum d'environ cent cubes (10^2) tandis que dans notre environnement 3D, ce maximum passe à mille cubes (10^3).

L'environnement 3D offre également beaucoup de fonctionnalités qui ont été utilisées dans le logiciel Arka3D. Le logiciel a été développé en DirectX, un langage de Microsoft principalement utilisé pour les jeux.

Cette section présente les règles du jeu ainsi que son fonctionnement général. Les contraintes liées au jeu et à la conception seront également examinées. De plus, nous présenterons notre processus de conception qui a permis de réaliser le projet.

2.2.2 Fonctionnement du jeu

Le jeu Arka3D respecte les règles du jeu arkanoid classique. L'environnement possède des cubes, des murs, une ou plusieurs balles ainsi qu'un disque que le joueur peut déplacer.

Le nombre de murs est passé de 3 dans la version 2D à 5 pour la version 3D afin de former une boîte rectangulaire ayant une seule ouverture. Les balles sont passées de cercles à des sphères et les cubes sont maintenant représentés en 3D. Le disque se déplacera maintenant sur 2 dimensions, soit sur le plan x,y.

Le objectif du jeu reste le même, soit faire rebondir la balle à l'aide du disque dans le but de détruire les cubes afin d'accumuler des points tout en évitant que la balle passe derrière le disque et sorte du jeu.

Le logiciel est composé de deux applications qui ont été développées en parallèle mais qui utilisent les mêmes structures. D'abord le logiciel de jeu qui permet au joueur de s'amuser et le constructeur qui permet de créer des tableaux de cubes qui pourront ensuite être chargés dans le logiciel de jeu.

Le constructeur est essentiel à la création de tableaux et ainsi qu'aux phases de tests. Il n'est pas très complexe ne posséde pas de balle ni de disque et il n'utilise donc pas l'algorithme de détection de collisions.

Le logiciel de jeu est donc le corps du logiciel Arka3D. La structure de ce logiciel est fondée sur celle de DirectX. La figure 2.3 présente le cheminement des étapes permettant le bon déroulement du jeu.

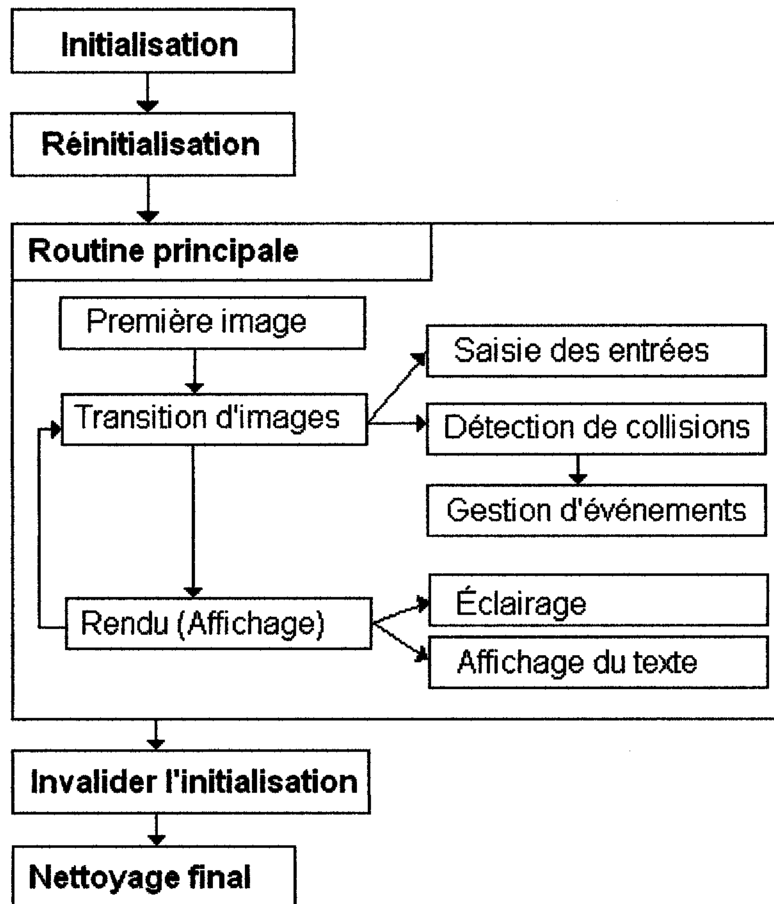


Figure 2.3 L'algorithme général du logiciel Arka3D.

Cette figure présente l'ordre des différentes étapes composant l'algorithme du jeu Arka3D. Voici en détail chacune des étapes.

- **Initialisation** : Créer les objets statiques comme les textures.
- **Réinitialisation** : Créer ou Recréer les objets dynamiques comme les modèles polygonaux.
- **Routine principale** : Boucle qui permet l'affichage de la scène.
    - Première image : Opération spéciale fait à la première image (Init.)
    - Transition d'images : Opération faite entre deux images affichées, gère surtout le déplacement des objets.
        - Saisies des données : Gestion du clavier et souris

        - Détection de collisions (Section 2.3)
            - Gestion d'événements (Section 2.4)
    - Rendu : Affichage par Direct X de la scène.
        - Éclairage : Ajustement des sources de lumières.
        - Affichage du texte : Affichage des points, etc.
- **Invalider l'initialisation :** Se produit lorsque le contexte d'affichage change (le mode d'affichage passe de fenêtre à plein écran par exemple). Cette étape détruit les objets dynamiques et appelle la réinitialisation pour les recréer.
- **Nettoyage finale :** Détruit les objets statiques et mets fin au jeu.

Chacune des étapes à ses propres fonctionnalités qui sont appelées à différents moments lors du déroulement du jeu. Évidemment, les étapes de transition d'images et de rendu sont appelées les plus souvent soient environ 50 fois par seconde.

Étant donné que l'habilité du joueur peut varier, le niveau de jeu peut être modifié. Par exemple, à la difficulté facile, la vitesse du jeu sera réduite et les objets plus gros, et ce, afin de permettre au joueur de viser plus facilement. Par contre, au niveau difficile, la vitesse des balles sera considérablement accélérée et la taille des objets réduite. Afin de motiver le joueur à passer à un niveau de jeu plus difficile, le nombre de points acquis est augmenté en fonction de la difficulté.

Le jeu est donc une version 3D du classique arkanoid et cette conversion rend la détection de collisions beaucoup plus complexe. Le nombre d'objets comme les murs et les cubes ont augmenté et les techniques de rebondissement ont été raffinées. Également, le déroulement du logiciel est fortement lié à la structure de DirectX.

2.2.3 Capacité et contrainte

L'environnement du jeu a des capacités limitées et des contraintes à respecter. Ces limitations sont instaurées afin de permettre une bonne jouabilité et de respecter les contraintes imposées par le matériel, comme la carte graphique, et le langage Direct X.

Ainsi, le nombre de balles maximum est de dix. Ce nombre est fixé du à une limite matérielle de la carte vidéo. Chacune des balles émet une lumière afin d'être localisées plus facilement par le joueur. Le nombre de lumières mis à la disposition du jeu par la carte vidéo est limité ce qui nous permet d'avoir jusqu'à dix balles. Dépasser ce nombre, les balles n'émettront plus de lumières et elles seront plus difficiles à repérer par le joueur.

La direction que les balles peuvent prendre est également limitée. L'angle des balles devra toujours permettre un déplacement raisonnable sur la profondeur (z). Cette limitation permet aux balles de revenir plus rapidement vers le disque déplacer par le joueur et les faire rebondir plus facilement.

Le nombre de cubes est limité a mille soit 10 par 10 par 10 (10^3) ce qui est amplement suffisant. La version classique en 2D du jeu possédait un maximum de 121 cubes. Un plus grand nombre de cubes rendrait les tableaux très longs à terminer pour le joueur qui se découragerait plus facilement. Les tableaux construits pour le jeu final utilisent rarement plus de 200 cubes.

La plus grande contrainte est sans doute temporelle. Étant donné que le jeu doit fournir une image fluide pour le joueur, celui-ci devra créer plus de 30 images par seconde. Cela laisse moins de 40 millisecondes pour déplacer les objets, effectuer la détection de collisions et afficher la scène. Étant donné que l'affichage des objets dans la scène prend beaucoup de temps et ne peut être optimisé énormément, la détection de collisions devrait prendre une minime fraction du temps requis pour créer chaque image. Voilà pourquoi la détection de collisions demande beaucoup d'optimisation.

Ces contraintes et limitations ont été établies en début de projet afin de bien cerner les problématiques qui pourraient surgir. Bien que certaines sont exprimées afin d'améliorer la jouabilité, d'autres contraintes sont imposées par les ressources limitées que nous offre le matériel. Bien établir ces capacités et contraintes nous ont permis de réaliser avec succès le projet Arka3D.

2.2.4 Technique de développement

Le jeu conçu est complexe et contient plusieurs caractéristiques. Une technique de développement devait donc être appliquée afin de mener le projet à bien. La technique utilisée au cours de ce projet s'inspire de la méthode de programmation XP (eXtreme programming).

La technique est assez simple. Le but est de développer rapidement des versions du programme en implémentant à chaque fois des nouvelles fonctionnalités. À chaque itération, ou à chaque version, l'ensemble du programme est testé afin de vérifier si les nouvelles fonctionnalités s'intègrent bien au reste du projet. L'annexe B présente un suivie de chaque version (voir également http://run.to/Arka3d).

L'ordre de conception a été soigneusement réfléchi afin d'intégrer les fonctionnalités le plus facilement possible. Les dépendances entre les objets ont été le principal critère de l'ordre d'intégration. La figure 2.4 présente l'ordre de développement selon la dépendance des objets.

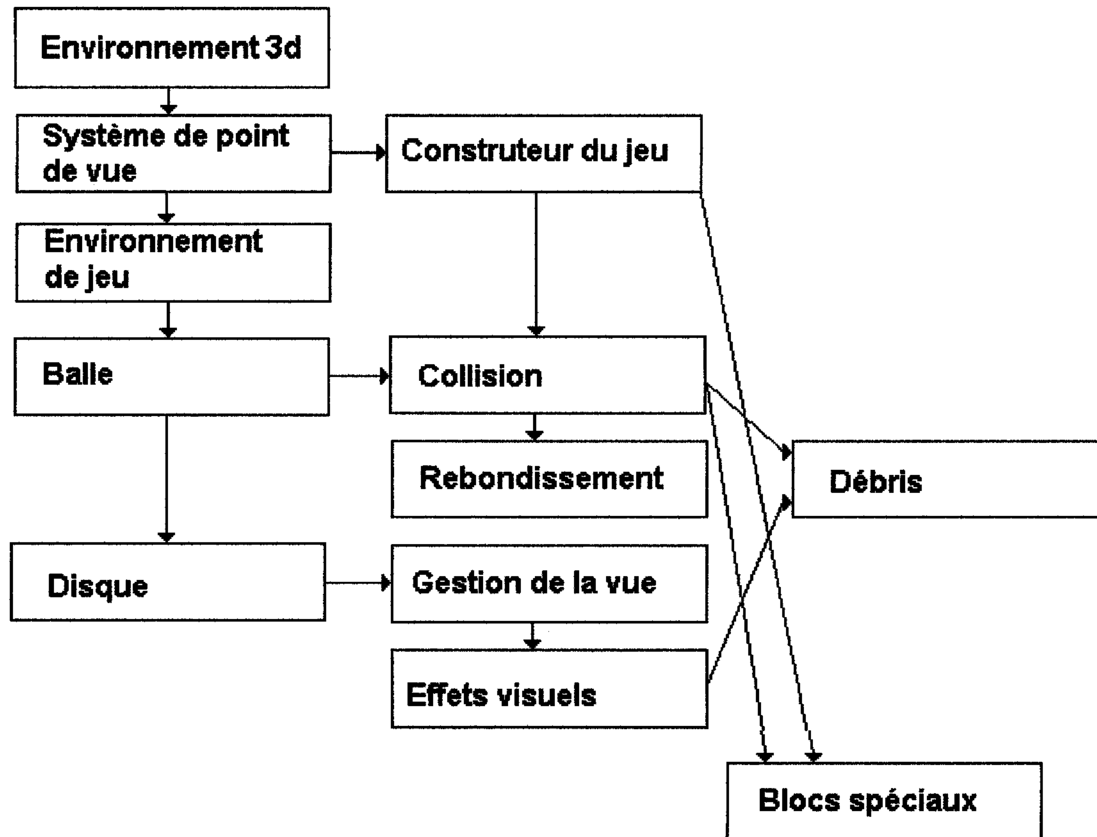


Figure 2.4 Structure de développement.

Les flèches de la figure 2.4 représentent le cheminement de la programmation. Ainsi, la dépendance suit le chemin inverse des flèches. Les éléments de la figure 2.4 sont présentés ci- dessous (Tableau 2.1).

Tableau 2.1 Cheminement de la programmation d'Arka3D

Élément du jeu	Description
Environnement 3D	Gestion de la routine d'affichage des objets et de la vue.
Système de point de vue	Système permettant de changer les points de vue afin de regarder la scène de différents angles. Essentiel à la phase de test.
Environnement de jeu	Environnement général dans lequel évolue le joueur. Il est constitué de murs et de blocs. Il permet également de charger des tableaux de blocs.
Constructeur de jeu	Permet de construire et d'enregistrer les tableaux de cubes qui seront ensuite chargés dans l'environnement de jeu.
Balle	Gestion de l'affichage de la balle et de ses mouvements.
Collision	Système de détection de collisions utilisé pour la balle et les débris.
Rebondissement	Gestion du rebondissement de la balle suite à une collision.
Disque	Gestion de l'affichage du disque et du contrôle du joueur sur celui-ci.

Gestion de la vue	Système permettant un suivi visuel du disque et des balles.
Effets visuels	Effets sonores et explosions causés par les collisions.
Débris	Création et gestion de débris suite à une explosion.
Blocs spéciaux	Gestion des blocs à plusieurs coups et des blocs explosifs

L'ordre du développement suit donc une suite logique. Les nombreuses fonctionnalités du logiciel ont été structurées afin de pouvoir les réaliser plus facilement en éliminant les redondances et en permettant des tests de qualité plus efficaces. Cette technique de développement est aussi un facteur important qui a permis le succès du projet Arka3D.

2.2.5 Conclusion

Voilà comment la conception du jeu a été planifiée. Arka3D, qui est une version 3D du jeu d'arcade Arkanoïd, a donc été conçue itérativement en versions successives en tenant compte des capacités matérielles et des contraintes de jouabilité.

Les principaux modules présentés dans la section 2.1.4 ont été implémentés de manière logique et ordonnés afin de bien les intégrer successivement les uns aux autres pour ainsi faciliter les tests.

Ces modules comportent un ou plusieurs objets qui sont détaillés dans la section suivante. Les algorithmes de détection de collisions font, bien sûr, partie de ces objets. La section 2.2 explique donc la structure des objets et la section 2.3 les algorithmes de détection de collisions qui ont été intégrés a ceux-ci.

2.3 Modélisation des objets

2.3.1 Introduction

La programmation objets est largement répandue comme parodigme informatique surtout dans les applications développées en C++. Direct X propose également des structures de classe d'objets intéressantes qui facilitent la programmation. La programmation à l'aide d'objets conceptuels (programmation objet) a donc été adoptée pour le développement de l'application Arka3D.

Dans cette section nous verrons les principaux objets conceptuels qui composent le jeu Arka3D. Nous verrons en détail leurs attributs et leurs fonctionnalités. Le tableau 2.2 présente un résumé de ces objets conceptuels.

Tableau 2.2 Liste des objets conceptuels composant le jeu Arka3D.

CArka3D	Corps du jeu qui contient tout les autres objets et en gère l'affichage.
CMur	Objet représentant les murs formant une boîte dans laquelle le jeu évolue.
CBalles	Objet gérant un ensemble de balles ainsi que leur mouvement.
CDisque	Objet gérant le disque et son mouvement.
CDébris	Objet gérant les débris ainsi que leur mouvement.
CCubes	Objet gérant les cubes et la construction de primitives associées à ceux-ci.
CExplosions	Objet permettant l'affichage d'explosions.

Ces objets conceptuels interagissent entre eux et ils ont pour la plupart une structure générale permettant de les construire correctement.

Ces classes d'objets ont été créées afin de permettre une meilleure compréhension du programme en regroupant les fonctionnalités associées à ceux-ci.

2.3.2 L'objet Arka3D

Description

Cet objet est le corps du jeu Arka3D et il contient tous les autres objets. En plus de gérer les interactions des autres objets, il permet également le chargement et le déchargement en mémoire des modèles polygonaux des objets 3D et des textures.

De plus, l'objet Arka3D gère le temps et l'affichage. À chaque itération d'affichage, il s'occupe de calculer le temps écoulé afin de déterminer les futurs mouvements du jeu et ainsi avoir des mouvements constants.

Il s'occupe également des contrôles du joueur en utilisant DirectInput. Cela permet à l'utilisateur d'interagir avec l'environnement à l'aide du clavier, de la souris et même d'une manette de jeu.

La principale fonctionnalité de l'objet Arka3D lors du déroulement du jeu est donc d'afficher les objets au bon moment. Il possède donc un automate qui détermine l'état du jeu et quels objets graphique 3D doivent être affichés. La figure 2.5 présente cet automate.

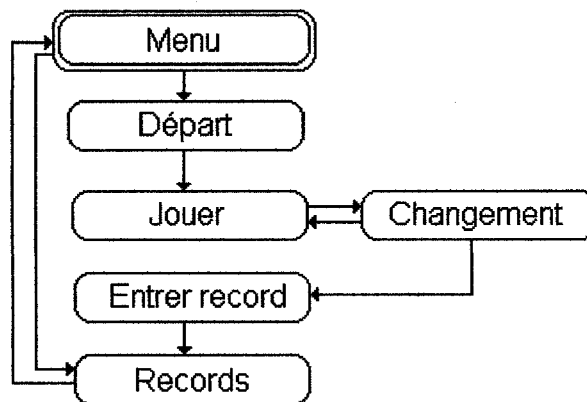


Figure 2.5. États de l'objet Arka3D.

Le bon déroulement du jeu est donc assuré par cet automate. Chacun des états possèdent des caractéristiques particulières qui sont décrites dans le tableau 2.3.

Tableau 2.3 États de l'objet Arka3D.

État	Description
Menu	Permet l'affichage du menu et permet au joueur de faire une sélection parmi quatre options soient : jouer, changer la difficulté, afficher les records et quitter le jeu. Cet état correspond à l'état initial et final.
Départ	Permet l'initialisation du début du jeu et passe à l'état jouer.
Jouer	Permet l'affichage de la scène de jeu et de gérer les contrôles du jeu. Affiche les balles, les cubes, le disque. Se termine lorsqu'il n'y a plus de cubes à briser ou lorsque les balles sont perdues (mort).
Changement	Appeler à la fin de l'état jouer afin de redémarrer le jeu ou terminer celui-ci. Il gère également le chargement de nouveaux tableaux de cubes et la fin du jeu.
Entrer record	Lors de la fin du jeu, cet état permet d'entrer un nouveau

	record de points. Il passe directement à l'état Record s'il n'y a pas de nouveau record.
Records	Affiche les records de points fait précédemment dans le jeu. Retourne ensuite à l'état Menu.

Il est possible de quitter le jeu à tout moment mais le nombre de points et le tableau de cubes en cours d'utilisation sont bien sûr perdus.

Les tableaux 2.4 et 2.5 décrivent la structure de l'objet Arka3d.

Tableau 2.4 Principaux attributs de l'objet Arka3D.

Attribut	**Description**
m_pd3dDevice	Pointeur sur le moteur 3D de DirectX.
Mur	L'objet mur (voir la section 2.3.3).
Balles	L'objet balles (voir la section 2.3.4).
Disque	L'objet disque (voir la section 2.3.5).
Cubes	L'objet cubes (voir la section 2.3.6).
Débris	L'objet débris (voir la section 2.3.7).
Explosions	L'objet explosions (voir la section 2.3.8).
UserInput	L'objet servant à la gestion des contrôles du jeu (DirectInput).
Etat	Entier représentant l'état actif.
Points	Entier long contenant le nombre de points.
Difficulté	Entier représentant la difficulté du jeu.
Vie	Entier représentant le nombre de vies restantes.
Highscore	Structure contenant une liste de records.

Tableau 2.5 Principales méthodes de l'objet Arka3D.

Méthode	**Description**
OneTimeSceneInit	Initialisation du moteur 3D.
InitDeviceObjects	Initialisation des objets statiques.
RestoreDeviceObjects	Réinitialisation des objets dynamiques.
LoadTexture	Permet de charger les textures (statiques).
Render	Construction de la scène et affichage de celle-ci.
FrameMove	Gestion des déplacements et des contrôles du joueur.
RenderText	Afficher le texte 2D à l'écran.
SetLights	Permet l'ajustement des lumières présentes.
LoadMenu	Permet de charger le menu.
LoadStage	Permet de charger un fichier de tableau de cubes.
ReloadStage	Permet de recharger le tableau de cubes actuel.
DeleteTexture	Permet de décharger les textures.
InvalidateDeviceObjects	Suppression des objets dynamiques.
DeleteDeviceObjects	Suppression des objets statiques.
FinalCleanup	Nettoyage final du moteur 3D et fin du programme.

Les figures 2.6 et 2.7 donnent un aperçu du menu et du jeu.

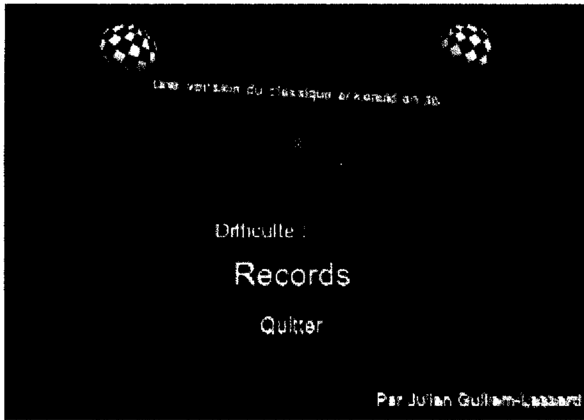


Figure 2.6. Aperçu du menu.

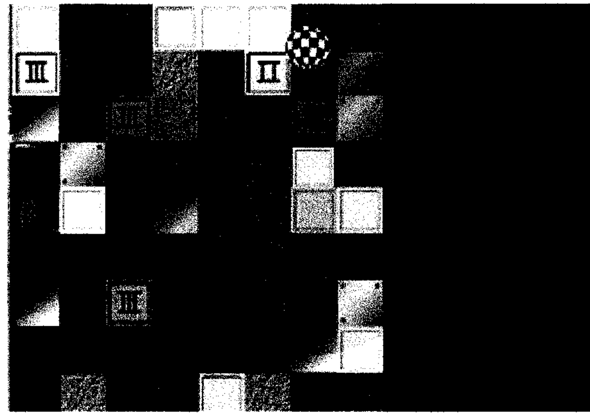


Figure 2.7. Aperçu du jeu.

2.3.3 L'objet Murs

L'objet mur gère les murs qui entourent l'environnement de jeu. Ces murs sont utilisés principalement dans l'état « Jouer » de Arka3D. Bien que les murs semblent uniformes et simples, ils sont en fait composés des plusieurs milliers de polygones afin de créer un effet de réflexion de lumière plus réaliste. Étant donné que les murs sont très complexes, ils sont définis comme des modèles dynamiques afin d'améliorer le temps d'affichage.

Par contre, lors de la détection de collisions, les murs sont considérés comme des plans simples ce qui améliore considérablement le temps d'exécution de l'algorithme de détection de collisions.

Les murs sont composés de deux parties soient l'extérieur et l'intérieur. L'intérieur est opaque et l'extérieur est translucide afin de pouvoir voir la scène de jeu de n'importe quels angles. Les effets de lumière des murs extérieurs sont également particuliers comme nous le verrons dans la section 2.6.2.

Par contre, l'objet mur est immobile ce qui en fait un objet très simple à gérer suite à sa construction.

Les tableaux 2.6 et 2.7 contiennent une description de la structure de l'objet Mur.

Tableau 2.6 Principaux attributs de l'objet Murs.

Attribut	Description
m_pD3DXWallMesh	Structure graphique Direct3D servant au stockage des polygones du mur intérieur.
m_pD3DXOutWallMesh	Structure graphique Direct3D servant au stockage des polygones du mur extérieur.

Tableau 2.7 Principales méthodes de l'objet Murs.

Méthode	Description
BuildMur	Permet de construire les « meshs » servant aux murs.
DessMur	Permet de dessiner les murs.
DeleteMur	Permet de supprimer les structures graphiques servant aux murs.

Les figures 2.8 et 2.9 décrivent pour leur part un aperçu des murs vus de l'intérieur et de l'extérieur respectivement

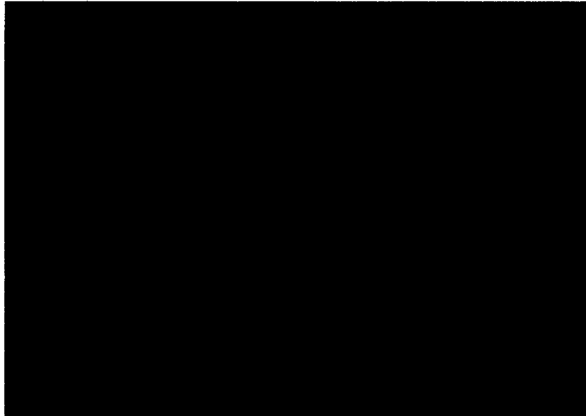


Figure 2.8 Aperçu des murs intérieurs.

Figure 2.9 Aperçu des murs extérieurs.

2.3.4 L'objet Balles

L'objet Balles gère l'ensemble des balles disponibles dans le jeu, autant au niveau de l'état « Jouer » que de l'état « Menu ». Les balles sont des modèles sphériques qui supportent les textures. La texture appliquée est une image de damier noir et blanc.

Les balles sont des objets conceptuels très importants dans le jeu. Elles sont les principaux objets en mouvement et les seules à pouvoir rebondir et être déviées. Elles sont utilisées par le joueur pour détruire les cubes. En effet, le joueur peut diriger une balle vers les cubes en la faisant rebondir à l'aide du disque. De plus, la perte de toutes les balles en jeu au travers de la partie inférieure des murs cause la perte de vies et la fin du jeu.

Les balles, tant comme les débris, ont de loin l'algorithme de détection de collisions le plus complexe. Mais en plus de gérer les collisions avec les autres objets, elles doivent réagir à la collision en rebondissant ou en déviant de trajectoire.

De plus, chaque balle émet une lumière qui est réfléchie par les autres objets. Cette lumière permet un meilleur positionnement de la balle par le joueur. Dans un autre ordre d'idée, les balles peuvent également servir de point de vue afin de pouvoir faire le suivi des balles dans l'environnement.

Voilà pourquoi l'objet balles est un objet très complexe et très important dans le jeu Arka3D. Les tableaux 2.8 et 2.9 décrivent les attributs et les méthodes de la classe balles.

Tableau 2.8 Principaux attributs de l'objet Balles.

Attribut	Description
balls	Liste de balles actives contenant les informations sur la grosseur, la vitesse et la direction de chaque balle.
nbballs	Nombre de balles actives.
ppas	Valeur numérique servant à la division en pas de l'algorithme de détection de collisions.
m_pD3DXBallMesh	Structure graphique Direct3D permettant le stockage de polygones du modèle sphérique de base.
m_pD3DXBallMeshTex	Structure graphique Direct3D permettant le stockage de polygones du modèle sphérique texturé.

Tableau 2.9 Principales méthodes de l'objet Balles.

Méthode	Description
CreerTexture	Charge la texture à damier des balles.
BuildBalle	Crée la structure graphique dynamique et texturé des balles.
InitBalle	Initialise les balles en début de jeu.
InitMenuBalle	Initialise les balles pour l'affichage du menu.
AddBalle	Ajoute une balle.
MoveBalle	Gère le déplacement des balles en appelant les algorithmes de détection de collisions.

CollisionMur	Algorithme de détection de collisions et de rebondissement avec les murs.
CollisionPad	Algorithme de détection de collisions et de rebondissement avec le disque.
CollisionBall	Algorithme de détection de collisions et de rebondissement entre deux balles.
CollisionCube	Algorithme de détection de collisions et de rebondissement avec les cubes.
CollisionDebris	Algorithme de détection de collisions et de déviation avec les débris.
dessBalles	Affiche les balles dans l'état « Menu » et l'état « Jouer ».
remBalle	Enleve une balle.
deleteBalle	Efface la mesh dynamique des balles.
deleteTexture	Efface la texture à damier des balles.

Les figures 2.10 et 2.11 permettent de visualiser un objet graphique balle et l'effet de lumière incorporé à celui-ci (figure 2.11).

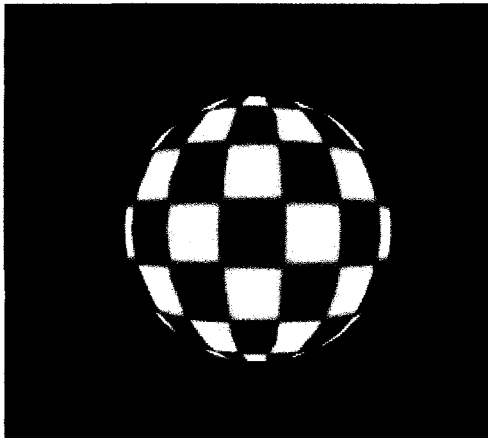


Figure 2.10 Balle texturée.

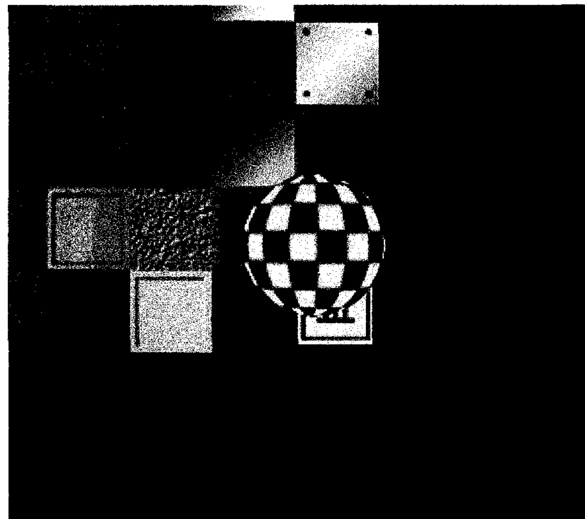


Figure 2.11 Effet de lumière de la balle.

2.3.5 L'objet Disque

Le disque est le seul objet que le joueur peut déplacer directement à l'aide du clavier et de la souris. Cet objet permet au joueur de faire rebondir les balles afin de changer leurs directions et d'éviter que celles-ci ne sortent par la partie inférieure des murs.

La structure graphique (Mesh) de l'objet disque est construite à l'aide d'une sphère coupée. Cette structure est chargée à partir d'un fichier « X

Mech », la structure de prédilection de Direct X. Cette structure a été construite à l'aide de MilkShape 3D. Cette structure est ensuite chargée dans le logiciel.

L'opacité du disque change selon la distance de celui-ci avec le point de vue. Si le disque est suffisamment proche du point de vue, celui-ci sera translucide afin de permettre au joueur de bien voir la scène (voir figure 2.13). Sinon, le disque devient opaque afin de permettre au joueur de voir correctement le disque même si celui-ci est éloigné (voir figure 2.12).

Le déplacement du disque est limité. D'abord, il ne peut être déplacé que sur le plan XY. Ainsi, il ne peut pas s'approcher ni s'éloigner du tableau de cube. Ensuite, le déplacement du disque est limité par les murs. Celui-ci ne peut donc pas sortir de l'environnement de jeu.

Le disque est donc l'entité que le joueur utilise afin d'influencer le jeu, de gagner des points et d'éviter de perdre les balles.

Les tableaux 2.10 et 2.11 décrivent la structure de l'objet Disque.

Tableau 2.10 Principaux attributs de l'objet Disque.

Attribut	Description
pos	Vecteur contenant la position actuelle du disque.
taille	Valeur numérique indiquant la taille du disque.
vitesse	Valeur numérique indiquant la vitesse du disque en rapport avec les entrées (DirectInput) du joueur (Invariable) .
m_pD3DXPadMesh	Structure graphique Direct3D contenant la structure de polygones du disque.

Tableau 2.11 Principales fonctions de l'objet Disque.

Fonction	Description
buildDisque	Charge la structure de polygones à partir du fichier pad.x
initDisque	Initialise le disque en début de jeu en centrant celui-ci.
deplaceDisque	Permet de déplacer le disque suite à une manipulation du joueur.
dessDisque	Dessine le disque opaque ou translucide dans la scène.
deleteDisque	Décharge la structure de polygones du disque.

Les figures 2.12 et 2.13 présentent graphiquement l'apparence visuelle du disque opaque et translucide respectivement.

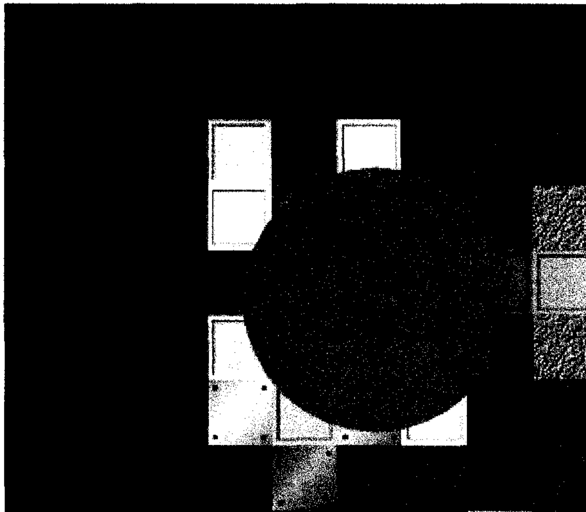


Figure 2.12 Disque opaque.

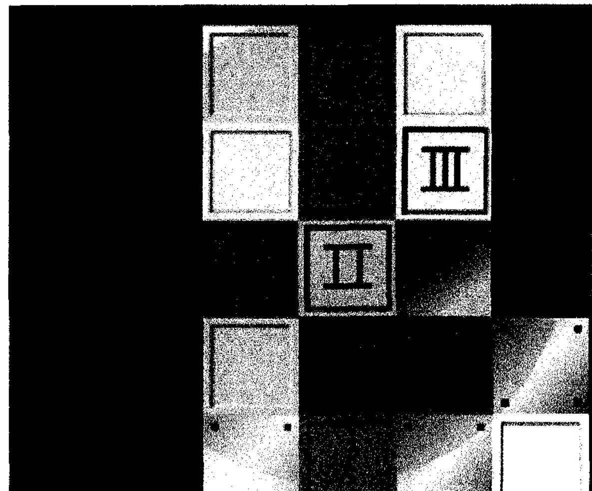


Figure 2.13 Disque translucide.

2.3.6 L'objet Cubes

L'objet Cubes gère l'ensemble du tableau de cubes. Le grand nombre de cubes et les optimisation faites sur cet objet le rend très complexe. En effet, les cubes sont des structures abstraites qui sont concrètement affichées en facettes afin d'éliminer un grand nombre de polygones à afficher.

Détruire les cubes avec les balles est l'objectif du jeu. Les cubes sont donc visés par le joueur afin d'accumuler des points et de passer au niveau suivant.

Il existe cinq types de cubes soient :

- Cubes normaux : qui explosent après une seul collision.
- Cubes deux coups : qui explosent après deux collisions.
- Cubes trois coups : qui explosent après trois collisions.
- Cubes indestructibles : qui n'explosent jamais.
- Cubes explosifs : qui, lors d'une collision, créent un grande explosion faisant exploser les cubes adjacents.

Ces cubes sont insérés dans un tableau 10X10X10 qui représente leur position dans d'espace. Le nombre maximum de cubes est donc de milles et leur position est statique.

Pour l'affichage et la détection de collisions, les cubes sont transposés en facettes carrées. Un cube a en théorie six facettes. Par contre, les facettes des cubes adjacents peuvent être éliminées. Par exemple, sur deux cubes adjacents, qui sont normalement composés de six facettes, il est possible d'éliminer les deux facettes entre les deux cubes puisqu'elles ne sont pas visibles et qu'aucune collision n'est possible sur ces facettes. Cette technique est particulièrement efficace lorsqu'il y a un grand nombre de cubes. Si le tableau de cubes est complet, donc composé de mille cubes, le nombre de facettes dessinées sera de six cent facettes plutôt que de six milles facettes. Cette optimisation innovatrice est particulièrement efficace dans l'environnement du jeu Arka3D. La figure 2.15 présente un ensemble de 8 cubes dont 24 facettes sont dessinées plutôt que 48 (8 cubes de 6 faces).

Les facettes sont donc créées dynamiquement en mémoire et dessinées en primitives. L'affichage de primitives est un concept de base de DirectX qui permet de dessiner directement des structures de polygones créées dynamiquement. Donc, à l'affichage et à la détection de collisions, les cubes ne sont plus qu'un ensemble de facettes. À chaque changement dans le tableau, les facettes sont recréées afin de s'adapter au nouveau tableau de cubes.

Les cubes possèdent chacun une texture propre qui est transmise aux facettes. De plus, les cubes indestructibles possèdent une texture dynamique, changeant dans le temps, afin de créer un effet réfléchissant.

Les cubes ont donc une partie abstraite et une partie concrète qui sont liées logiquement afin de pouvoir déterminer à quels cubes appartiennent les facettes. Les facettes servent principalement à réduire le nombre de polygones affichés et à diminuer le nombre de tests de collisions.

La structure de l'objet Cubes est présentée aux tableaux 2.12 et 2.13

Tableau 2.12 Principaux attributs de l'objet Cubes.

Attribut	Description
m_pVB	Structure de stockage des polygones qui serviront à l'affichage par primitive (Vertex Buffer).
m_pD3DXCubeTex	Liste des textures disponibles pour les cubes.
stage	Entité contenant les informations sur le tableau 10X10X10 de cubes. Contient également l'information sur chaque cube soient : le type, la couleur et la texture.
plaques	Liste de facettes reliées aux cubes. Contient également l'information sur chaque facette soit : la position, la normale et le cube associé.

Tableau 2.13 Principales méthodes de l'objet Cubes.

Méthode	Description
CreerTexture	Charge les textures des cubes en mémoire.
InitPrim	Initialise les facettes en créant un espace mémoire pour celles-ci.
InitStageCube	Crée une liste de facettes à partir du tableau de cubes.
BuildPlaqPrim	Construit une liste de polygones pour l'affichage des facettes.
AddCube	Permet d'ajouter les facettes d'un cube.
HaveCube	Vérifie s'il y a un cube à la position donnée.
dessPlaqPrim	Affiche dans la scène l'ensemble de facettes.
trierColPlaq	Permet de trier les facettes à l'aide de la distance par rapport à l'objet susceptible d'entrer en collision afin d'ordonner la détection de collisions.
RefreshStage	Permet de recréer la liste de facettes lorsque le tableau de cubes à changer.
DétruireCubeExplosif	Détruit un cube explosif et récursivement les cubes adjacents.
flushPrim	Remet à zéro le nombre de facettes à afficher.
deletePrim	Détruit l'espace mémoire alloué aux facettes.
deleteTexture	Décharge les textures des cubes.

Les figures 2.14 et 2.15 exposent un cube rudimentaire et un ensemble de 8 cubes adjacents.

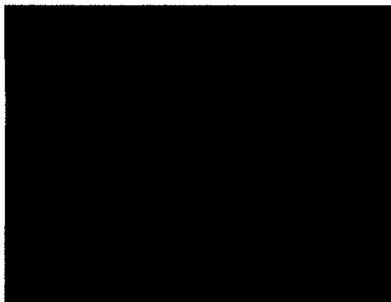


Figure 2.14 Cube simple.

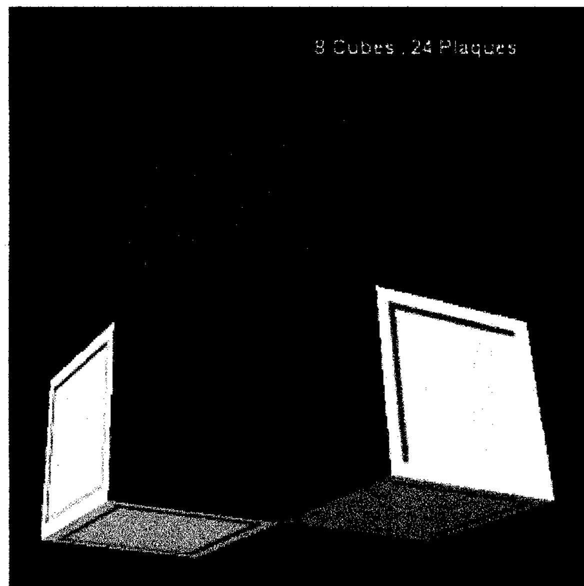


Figure 2.15 Aperçu des huit cubes adjacents.

2.3.7 L'objet Débris

Les débris sont des objets graphiques créés suite à la destruction de cubes. Ces objets graphiques se déplacent de manière linéaire dans l'environnement jusqu'à ce qu'ils entrent en collision et soient détruits.

L'objet débris gère donc une liste de débris et leurs déplacements. Suite à l'explosion d'un cube, un nombre aléatoire de débris est créé. Ces débris sont alors ajoutés à la liste de débris et l'objet conceptuel débris en gère le mouvement jusqu'à leurs destructions.

Les débris sont de trois formes différentes chargées à partir de fichiers X qui sont créés à l'aide de MilkShape3D. Ces structures de polygones supportent les textures.

Bien que la principale fonction des débris soit d'ajouter un effet visuel afin de rendre les explosions plus réalistes, ces entités peuvent entrer en collision avec les autres objets de l'environnement de jeu. Ils partagent donc un algorithme de détection de collisions similaire à celui des balles. Par contre, lorsqu'un débris entre en collision, il est détruit immédiatement créant une petite explosion (Voir la section 2.5.4). Les débris ne peuvent donc par rebondir ou être déviés.

Les débris ont aussi un effet sur l'environnement en pouvant influencer les balles. Lorsqu'une balle entre en collision avec un débris, celle-ci est déviée. L'importance de la déviation est définie en fonction de la taille du débris et par l'angle d'impact (voir la section 2.5.3).

Les débris sont donc plus qu'un effet visuel et ils peuvent également faire dévier les balles et donc influencer la suite des événements. Finalement, l'objet débris gère la liste des débris, le mouvement de ceux-ci ainsi que leur algorithme de détection de collisions qui s'apparente à celui des balles.

Les tableaux 2.14 et 2.15 contiennent la description de la structure de l'objet conceptuel Débris.

Tableau 2.14 Principaux attributs de l'objet débris.

Attribut	Description
debris	Liste de débris contenant des informations sur la position, la taille, le déplacement et la forme de ceux-ci.
ndebris	Nombre de débris dans la liste.
m_pD3DXDebrisMesh	Liste des trois structures de polygones des débris.
m_pD3DXDebTex	Texture des débris.

Tableau 2.15 Principales méthodes de l'objet débris

Méthode	Description
creerTexture	Charge la texture en mémoire.
buildDebris	Charge les trois structures graphiques « Mesh » à partir des fichiers X.
creerDebris	Ajoute dans la scène un nombre aléatoire de débris.
addDebris	Ajoute un débris à une position donnée.
moveDebris	Permet le déplacement des débris.
collisionMur	Détection de collisions entre un débris et les murs.
collisionDisque	Détection de collisions entre un débris et le disque.
collisionDebrisCube	Détection de collisions entre un débris et les cubes.
collision2debris	Détection de collisions entre deux débris.
dessDebris	Affiche les débris dans la scène.
clearDebris	Enlève tout les débris de la liste.
deleteDebris	Décharge les trois structures graphiques « Mesh » des débris de la mémoire.
deleteTexture	Décharge la texture de la mémoire.

Les figures 2.16 et 2.17 donnent un aperçu graphique d'un objet graphique débris individuel ainsi que plusieurs débris engendrant des explosions (fig. 2.17)

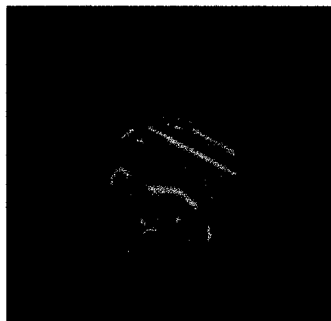


Figure 2.16.
Aperçu d'un débris.

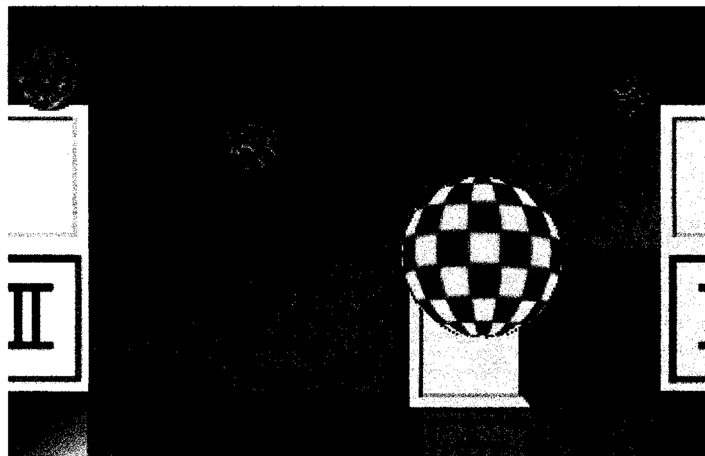


Figure 2.17. Plusieurs débris crées suite à la destruction d'un cube et explosions de quelques débris.

2.3.8 L'objet Explosions

L'objet explosions ajoute un autre effet visuel lors de la destruction d'un cube ou d'un débris. Il n'influence pas les autres objets graphiques. Les

explosions sont des sphères texturées basées sur le modèle polygonal des balles.

Les explosions ont une durée de vie limitée. À sa création, une explosion est une sphère d'un certain volume dépendant de l'importance de l'explosion. Ainsi les cubes explosifs créent de grandes explosions, les cubes normaux en créent de moyennes et les débris en créent de petites variant avec la taille de ceux-ci.

La taille et l'opacité des explosions évoluent ensuite dans le temps. Suite à sa création, la taille de l'explosion augmente et son opacité diminue pour devenir translucide. Cela permet de donner l'effet d'une dilatation de l'explosion. Après un certain temps, lorsque que la taille de l'explosion est à son maximum et que l'explosion est parfaitement translucide, l'explosion est terminée et celle-ci est enlevée de la liste d'explosions.

L'objet explosion gère donc une liste d'explosions en permettant d'en ajouter et en les faisant évoluer dans le temps jusqu'à la fin de leur durée de vie. L'effet visuel créé par l'explosion est impressionnant et permet au joueur de percevoir la destruction d'un objet. Les figures 2.18 et 2.19 permettent d'en visualiser l'effet.

Les tableaux 2.16 et 2.17 décrivent la structure générale de l'objet Explosion.

Tableau 2.16 Principaux Attributs de l'objet Explosions.

Attribut	Description
explosions	Liste d'explosions contenant des informations sur la taille, la position et la durée de vie de celles-ci.
nexplosions	Nombre d'explosions actives dans la scène.
m_pD3DXExpTex	Textures des explosions
*balles	Pointeur donnant accès à l'objet Balles permettant de dessiner des sphères sans créer de nouvelles structures de polygones.

Tableau 2.17 Principales méthodes de l'objet Explosions.

Méthode	Description
CreerTexture	Charge la texture en mémoire.
AddExplosion	Permet d'ajouter une explosion en spécifiant la taille, la position et la durée de vie.
DessExplosion	Dessine les explosions dans la scène.
clearExplosion	Vide la liste d'explosions.
DeleteTexture	Décharge la texture de la mémoire.

Figure 2.18 Petite explosion créée par un minuscule débris.

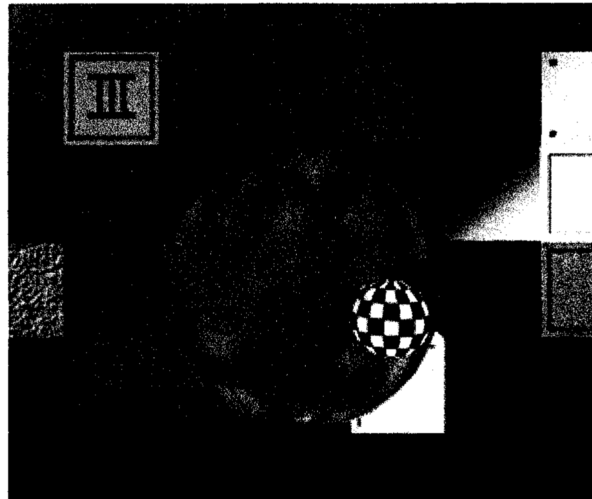


Figure 2.19 Grosse explosion créée par un cube explosif.

2.3.9 Conclusion

Les différents objets qui composent la scène ont été présentés dans cette dernière section. Ces objets forment, ensemble, le logiciel de jeu Arka3D. Chacun des objets a une utilité différente et se complète les uns les autres.

Certains objets sont très importants dans le fonctionnement du jeu. Les murs forment les limites de l'environnement de jeu. Le disque permet au joueur d'influencer le jeu en déviant les balles. Détruire le tableau de cube est l'objectif du jeu et les balles sont les seules à le permettre. Tous ces éléments sont essentiels à la jouabilité du jeu.

D'autres objets, comme les débris et les explosions, sont présents pour ajouter des effets visuels au jeu. Les explosions n'influencent en aucune manière les autres objets mais permettent au joueur de mieux voir les changements qui se produisent dans l'environnement. Les débris offrent également un autre effet visuel mais ajoute également un élément de difficulté en provoquant la déviation des balles lors d'une collision.

Nous verrons dans la prochaine section comment la détection de collisions à été implémentée et optimisée sur ces objets afin de se faire en un temps raisonnable (section 2.4). Ensuite, les différentes interactions entre les objets conceptuels à la suite d'une collision seront étudiées.

2.4 Détection de collisions

2.4.1 Introduction

La détection de collisions est sans aucun doute la partie la plus complexe d'un jeu vidéo qui contient un grand nombre d'objets graphiques comme dans l'environnement d'Arka3D. Voilà pourquoi une grande partie de la revue de littérature (chapitre 1) est axée sur cette problématique.

Le logiciel Arka3D utilise une approche hybride utilisant différents concepts présentés dans le chapitre 1. Les différents objets utilisent des détections de collisions adaptées qui utilisent des techniques comme : la division de l'espace (section 1.4.5), la simplification des modèles (section 1.4.2), la cohérence temporelle (section 1.4.9), la division des objets (section 1.4.4), les régions de Voronoï (section 1.4.8), les boîtes AABBs (section 1.4.3) et autres.

Le nombre de paires d'objets graphiques est sans contredit la plus grande problématique à gérer au cours de la détection de collisions. C'est pourquoi la gestion des cubes, qui peuvent être au nombre de mille, utilise beaucoup d'optimisations.

La détection de collisions est implémentée pour les objets en mouvement comme les balles, le disque et les débris. De plus, les objets immobiles, qui ne sont pas a risque de créer de nouvelles collisions, n'ont pas à être traité systématiquement de manière individuelle à chaque itération.

Certaines techniques de recouvrement après erreur seront également expliquées au fur et à mesure afin qu'aucune interaction suspecte ne se produit. Ces erreurs sont souvent causées par une imprécision mathématique dû aux nombreux calculs fait en virgules flottantes.

L'algorithme de détection de collisions hybride du logiciel Arka3D est en fait un ensemble d'algorithmes dont chacun est appliqué selon le type d'objets susceptible d'entrer en collision.

Cet algorithme de détection de collisions combiné à des techniques de recouvrement après erreur permet une détection rapide et cela sans qu'aucun événement invraisemblable se produise comme la pénétration entre deux objets.

2.4.2 Détection de collision entre 2 sphères

Les balles et les débris sont les deux seuls objets graphiques à utiliser ce type de détection. Une méthode de base pour la détection de sphères est donc utilisée pour détecter une collision entre ces deux objets (voir section 1.4.3).

La détection de collisions entre deux sphères est assez simple et elle a été vue dans la section 1.4.3. En résumé, il est possible de déterminer si deux sphères se pénètrent à l'aide de la formule de la distance.

Il existe donc trois types de collisions qui utiliseront ce modèle soient : les collisions entre deux balles, les collisions entre deux débris et les collisions entre une balle et un débris.

Dans le cas d'une collision positive entre deux balles, il est impératif de vérifier si les deux sphères s'approchent avant la collision et s'éloignent à la suite de la collision. Sinon les deux balles entreront dans un état de collisions récursives qui les immobilisera en les faisant tourner l'une autour de l'autre.

Pour régler ce problème, la distance précédente est comparée afin de vérifier si les deux objets de type balle s'approchent ou s'éloignent. Dans le cas où celles-ci s'éloignent, la collision devient négative peu importe la distance entre les deux sphères. En s'éloignant, elles sortiront inévitablement de cet état de pénétration.

Si par contre, les deux balles s'approchent l'une de l'autre et que le résultat après rebondissement les approchent encore, les deux balles sont légèrement déplacées de manière intemporelle afin de briser le cycle. Ce phénomène peut se produire lorsque les balles entrent en collision très près d'un mur. La figure 2.20 présente le phénomène.

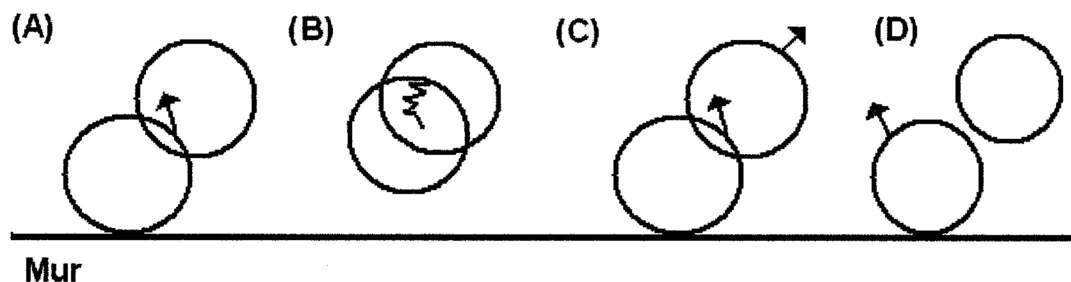


Figure 2.20 Phénomène de collisions cycliques entre 2 balles.
(A) Deux balles s'approchant du mur après une collision.
(B) Les collisions successives font pénétrer les deux balles.
(C) Ajout d'un mouvement intemporel (en vert).
(D) Sortie du cycle de collisions.

La figure 2.20 présente comment le mouvement intemporel peut régler la collision cyclique. La flèche rouge présente le mouvement de la balle après rebondissement. À la figure 2.20 (A), les balles se pénètrent encore plus après la collision ce qui cause une répétition de collisions observées en (B). À la figure 2.20 (C), l'ajout d'un mouvement (en vert) permet d'éloigner les balles et de sortir

de la collision cyclique (fig. 2.20 (D)) et le rebondissement devient donc possible (en vert).

Il est inutile de vérifier une telle situation lors de la collision entre une balle et un débris ainsi que la collision entre deux débris. Étant donné que le débris est détruit lors de la première itération de la collision, la collision cyclique est alors écartée.

Des phénomènes d'exception dû à certaines contraintes du jeu peuvent donc se produire lors d'une collision aussi simple que celle entre deux sphères. Nous avons vu comment le problème de collisions cycliques a été résolu. Ce recouvrement à un coût en temps d'exécution et il a donc été appliqué seulement à un des trois types de collision entre sphères soit la collision entre deux balles.

2.4.3 Collision sur les murs

Les murs forment les limites du jeu et doivent empêcher les objets évoluant dans l'environnement demeurent à l'intérieur de ces limites. Comme nous le verrons, les murs sont permissifs afin de s'adapter à certaines règles du jeu ou à certains recouvrements après erreurs.

Dans la section 2.3.3, nous avons montré que les murs étaient composés de plusieurs centaines de triangles afin de refléter correctement les effets de lumière. Par contre, au niveau de la détection de collisions, les murs sont considérés comme six plans statiques alignés sur les axes d'origine afin de simplifier les calculs et d'améliorer le temps d'exécution. La base, qui n'affiche pas de mur, possède quand même une limite afin de détecter les sorties de balles et de débris.

Dans le cas des débris, s'ils croisent l'un des six plans, ceux-ci sont détruits. Ce croisement est déterminé à l'aide de la position du débris additionnée à son rayon.

Pour les balles, c'est le même principe mais la direction des balles est également vérifiée. Un produit scalaire entre le vecteur de la direction de déplacement de la balle et la normale des murs intérieurs permet de vérifier si la balle se dirige vers l'extérieur ou vers l'intérieur des murs. C'est seulement si la balle se dirige vers l'extérieur que celle-ci rebondit afin de se diriger vers l'intérieur. Cela permet donc de récupérer une balle qui aurait dépassé les limites imposées par les murs comme la présente la figure 2.21.

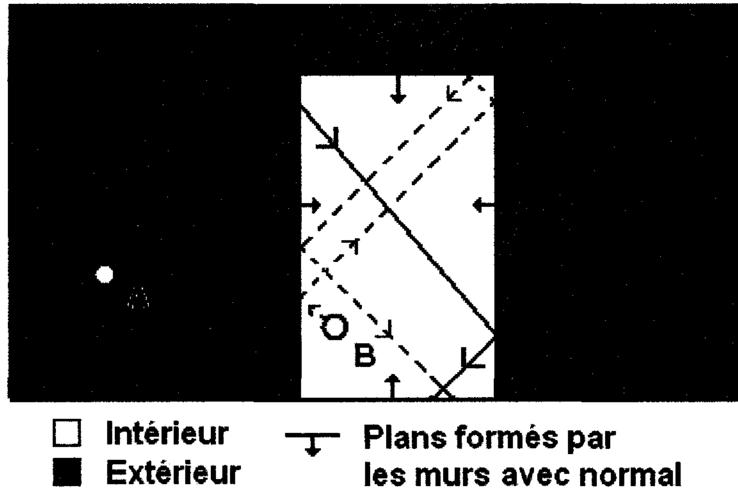


Figure 2.21 Rebondissement permissif des murs
La balle A située à l'extérieur qui est récupérée.
La balle B située à l'intérieur qui est conservée.

La figure 2.21 montre qu'une balle située loin à l'extérieur (balle A) des murs revient inévitablement à l'intérieur si le vecteur de déplacement est comparé avec la normale des murs. De plus, une balle située à l'intérieur (balle B) rebondit correctement. Cet exemple présente un balle située loin à l'extérieur ce qui est en pratique impossible. Par contre, il est possible qu'une balle se situe légèrement à l'extérieur suite à un recouvrement après erreur (déplacement intemporel). Dans ce cas, la balle est rebondie correctement à l'intérieur.

Le disque est un autre objet graphique qui est susceptible d'entrer en collision avec les murs. Étant déplacé par le joueur, celui-ci sera limité par les quatre plans croisant son plan de déplacement (XZ), soit les quatre murs de côté. Les murs sont également permissifs pour le disque, celui-ci peut dépasser à environ 30% la limite imposée par le mur afin de permettre au joueur de faire dévier les balles frôlant les murs avec une plus grande marge de manœuvre. Cette permission est donc accordée afin d'améliorer la jouabilité du logiciel. La figure 2.22 montre cette permission.

La vue associée au disque est également limitée par les murs. De la même façon que pour le disque, elle est permissive en permettant au joueur de regarder la scène de l'extérieur des murs. Les murs sont donc beaucoup plus permissifs avec la vue que le disque afin de permettre une meilleure visibilité du jeu. La figure 2.23 présente le résultat visuel de cette permission.

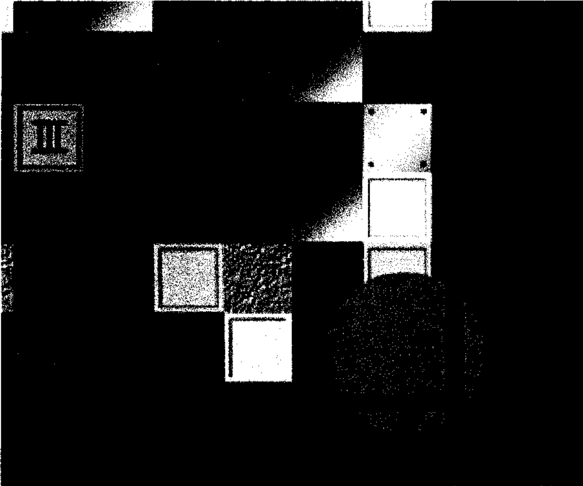


Figure 2.22
Permission accordée au disque.

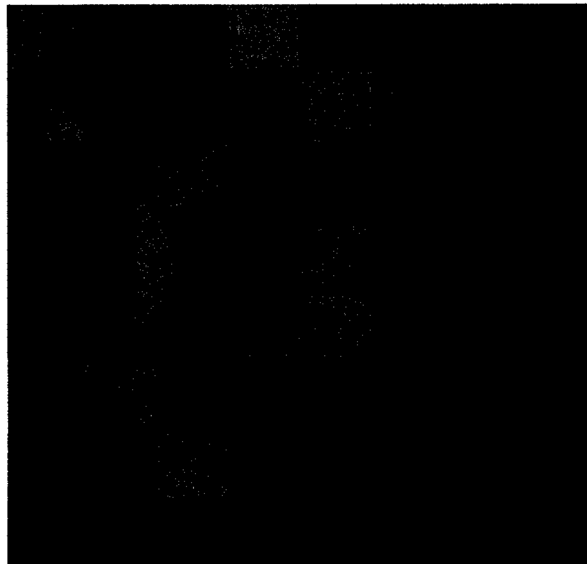


Figure 2.23
Permission accordée à la vue.

Dans les figures 2.22 et 2.23, nous pouvons observer visuellement l'effet de ces permissions. Ces permissions améliorent la jouabilité et permettent de meilleurs points de vue sur l'environnement de jeu.

La détection de collisions avec les murs est donc très simple. Il s'agit de vérifier la position et le rayon des objets par rapport à des plans alignés sur des axes d'origine. Pour les balles, la direction est aussi vérifiée afin de permettre une récupération des balles situées à l'extérieur. Les collisions sont permissives également afin d'améliorer la jouabilité du jeu par rapport au disque et au point de vue. Les murs sont donc simples et permissifs.

2.4.4 Collision sur le disque

Le disque est l'objet graphique que le joueur peut déplacer afin de faire dévier les balles sur les cubes. Les collisions avec le disque sont donc très importantes afin de permettre une bonne jouabilité du logiciel Arka3D.

La section 2.2.5 explique que le disque a été construit à partir d'une sphère coupée. Bien que, pour l'affichage, le disque soit transféré en modèle polygonal, la structure logique de la sphère coupée a été conservée dans l'algorithme de détection de collisions.

La technique de géométrie constructive (CSG), présenté dans la section 1.2.3 [7], a donc été utilisée pour cet objet. Le volume logique occupé par le

disque est formé par l'union d'une sphère et d'un plan orienté tels que présenté dans la figure 2.24.

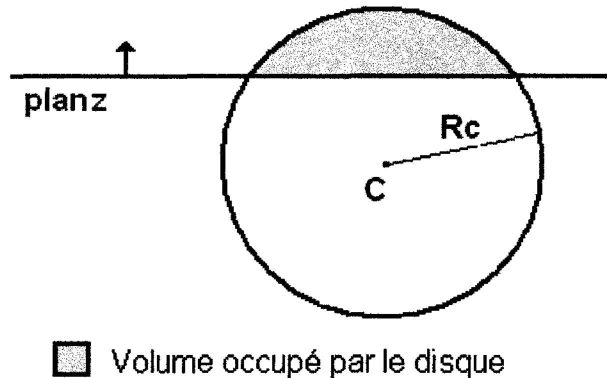


Figure 2.24 Volume occupé par le disque déterminé par l'union d'une sphère et d'un plan orienté en version 2D.

Dans la figure 2.24, le plan orienté, présenté en rouge, coupe la sphère présenté en noir pour créer un nouveau volume utilisé par le disque (en gris). L'équation du plan est simple puisque celui-ci se situe sur un axe d'origine soit : z = planz. L'équation de ce plan orienté est donc z >= planz. L'équation de la sphère est bien entendu la formule de la distance comme présentée dans la section 1.3.3. L'union de ces deux volumes est :

$$Z_p >= planz \cap \sqrt{(x_p - x_c)^2 + (y_p - y_c)^2 + (z_p - z_c)^2} <= Rc \quad (3)$$

où p : est un point donné à évaluer.
planz : est la valeur z du plan.
c : est le centre de la sphère.
Rc : est le rayon de la sphère.

Pour vérifier si ce volume spécifié par la sphère et le plan est en intersection avec une autre sphère, comme c'est le cas avec les balles et les débris, il faut ajouter le rayon de cette nouvelle sphère afin que la collision soit détectée au bon moment.

De plus, le disque est sujet aux déplacements incohérents et non temporels étant donné que l'ensemble de ses mouvements est géré par un utilisateur. Le disque peut donc en tout moment se situer au milieu d'une sphère. Afin de faciliter le jeu, le disque fait rebondir exclusivement les balles vers l'avant, soit la direction indiquée par la normale du plan Z. Cela permettra également d'éviter les collisions récursives en limitant de moitié les angles d'incidences possibles. Une balle provenant de l'arrière n'est donc pas sujette à des collisions.

En pratique, une balle ne peut jamais provenir de l'arrière puisqu'il n'y a pas de mur arrière permettant de faire rebondir la balle dans cette direction.

Le disque utilise donc la géométrie constructive (CSG) [7] afin de créer un volume dans l'espace qui est sujet à des collisions. De plus, étant donné que le disque est sujet à des mouvements brusques et non temporels, un processus vérifiant l'angle d'incidence permet d'éviter les collisions récursives.

2.4.5 Collision avec les cubes

La collision avec les cubes est de loin la plus complexe. Elle utilise plusieurs optimisations et concepts vus dans la revue de littérature (sections 1.3 et 1.4) ainsi que quelques méthodes adaptées au projet plus innovatrices. Étant donné que le tableau peut contenir jusqu'à mille cubes, le nombre de paires d'objets doit être limité au maximum.

Premièrement, seules les balles et les débris peuvent entrer en collision avec les cubes, étant donné que ce sont les seuls objets à pouvoir se déplacer dans la partie de l'environnement où les cubes sont présents. Ces deux objets sont représentés par des sphères.

Une routine de détection de collisions permettant d'éliminer un maximum de tests a été établie afin de réduire au minimum les calculs à faire. Cette routine peut s'arrêter à tous moments lorsque tous les cubes à vérifier sont éliminés logiquement par la routine. Voici cette routine en quatre étapes :

1. **Déterminer les cubes en collision possible.**
    - Transposer la position actuelle de la sphère sur la position des cubes (le tableau 10X10X10).
      (Trouver vis-à-vis quel cube se situe le point milieu actuel de la sphère).
    - Ajouter les cubes adjacents à ce cube, formant une boîte autour de celui-ci. (27 cubes maximum ou 2.7%).
      (Le diamètre de la sphère doit toujours être plus petite que les cubes sinon plus de cubes sont retenues).

2. **Déterminer les plaquettes en collision possible**
    - Déterminer les facettes à partir des cubes (six par cubes).
    - Éliminer la moitié des plaquettes à partir de l'orientation de la direction de la sphère selon la règle de cohérence temporelle.
      (Le produit scalaire des normales de facettes avec la direction de déplacement de la sphère détermine si les facettes n'entrent jamais en collision) (élimine approximativement 50% des facettes).
    - Calcul de la distance avec la sphère pour chaque facette.
    - Trier les facettes par distance (quick sort).

- Détection de collisions pour les facettes en ordre croissant jusqu'à une valeur maximum de distance (où aucune collision n'est possible) (Élimine un nombre très variable de facettes).

3. **Détection de la collision des facettes**
   - Déterminer le vecteur entre l'ancienne position de la sphère et la nouvelle position plus le rayon de la sphère.
   - Trouver le point d'intersection avec le plan de la facette.
   - Si le point fait partie du domaine de la facette, il y a collision et un rebondissement plat (Région Voronoï de la face de la facette).
   - Sinon, si le point fait partie du domaine de la facette étendue du rayon de la balle, il y a collision et un rebondissement sur coin (Région de Voronoï de l'arrête et des sommets de la facette.).

4. **Collision détectée**
   - Arrêter la détection des autres facettes.
   - Appliquer les modifications sur le cube.
   - Répéter la routine jusqu'à ce qu'aucune collision ne soit détectée.
   - Le maximum devrait être de 3 par sphère.

Explications

La première étape consiste à sélectionner un minimum de cubes susceptibles d'entrer en collision avec la sphère en mouvement. Pour ce faire, une méthode de division de l'espace, similaire à celle présentée dans la section 1.4.5, est utilisée afin de sélectionner un nombre limité de cubes. La division de l'espace utilisée est dynamique, elle constitue une zone autour de la sphère se déplaçant avec elle. Cette technique permet de retenir un maximum de 27 cubes (3^3) lorsque la sphère est plus petite que les cubes.

Ensuite, dans la seconde étape, ont utilise la seconde partie logique des cubes soit les facettes. Les cubes sont donc décomposés en six facettes afin de permettre l'élimination de certaines d'entre elle. La norme sur l'élimination des facettes des cubes adjacents s'applique également ici. Un ensemble de 27 cubes générera donc 64 facettes plutôt que 162. La conversion des cubes en facettes présentée à la section 2.6.5 permet cette optimisation. De plus, en utilisant la technique de la cohérence temporelle sur le déplacement d'une sphère présentée à la section 1.4.9 (Figure 1.16), il est possible d'éliminer ensuite en moyenne 50% des facettes non sujettes à des collisions. Les facettes restantes sont alors triées selon la distance avec la sphère susceptible d'entrer en collision avec elles. Ce tri à deux utilités : la première étant de traiter d'abord les facettes les plus rapprochées afin de choisir la plus appropriée pour le premier test de collision et la seconde étant d'interrompre le traitement lorsque la collision est impossible dû à la distance entre la sphère et la facette (En rouge sur la figure 2.25).

La troisième étape permet la détection entre les sphères et les facettes. Pour ce faire, un vecteur de déplacement entre l'ancienne et la nouvelle position de la sphère est déterminé en tenant compte du rayon de celle-ci. Si ce vecteur traverse le plan sur lequel se situe la facette, les régions de Voronoï de cette facette sont alors déterminées (voir la section 1.4.8). Ces régions de Voronoï sont présentées à la figure 2.25.

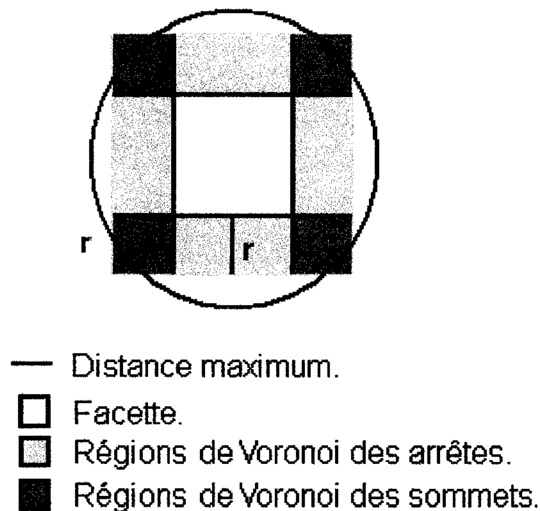


Figure 2.25 Régions de Voronoï d'une facette.

Dans cette figure 2.25, les différentes régions de Voronoï d'une facette (en gris) sont presentées ainsi que la distance limite de possibilité de collision qui intervient au niveau du tri (en rouge). Le rayon de la sphère susceptible d'entrer en collision (r) délimite à la fois la distance maximum et les régions de Voronoï. L'union de ces deux régions délimite un espace où la collision se produit et des calculs complets sur le point de collision sont alors effectués. Selon la région dans laquelle se situe la sphère, un rebondissement approprié est effectué dans le cas d'une balle.

Suite à une détection de collision positive, un changement dans la scène est observé si la sphère est une balle : celle-ci changera de trajectoire et un cube pourrait être détruit. Voilà pourquoi à la quatrième étape, l'algorithme de détection de collisions des cubes est arrêté, les changements sont effectués et l'algorithme complet incluant les autres objets est relancé jusqu'à ce que aucun changement ne soit apporté. Il est très rare que plusieurs collisions avec une balle se produisent au même instant mais la possibilité est bien réelle et doit être gérée correctement.

Des mécanismes de recouvrement après erreurs sont également implémentés afin de gérer les rares cas où l'imprécision mathématique fait rater

un test de collision ou un rebondissement. Les nombreux calculs sont effectués afin de déterminer les régions et le point de contact causent cette erreur. Bien qu'elle soit minime, elle peut dans de rares cas (environ 1 sur 10000) biaiser la détection de collisions et rater un rebondissement. Il existe deux mécanismes permettant la correction de cette erreur. D'abord au niveau de la détection de collisions, un test est fait afin de vérifier si la sphère se situe à l'intérieur d'un cube (ce qui est en réalité impossible). Dans ce cas, un rebondissement approprié (balle) ou une destruction (débris) est effectué. Ensuite, si une balle entre deux fois au même instant en collision avec la même facette, un déplacement intemporel est effectuer afin d'éliminer la redondance cyclique détectée.

La détection de collisions entre les sphères et le tableau de cubes est donc la plus complexe. La détection se fait en plusieurs étapes qui permettent d'éliminer un maximum de facettes afin de réduire le temps d'exécution. Plusieurs techniques vues dans la revue de littérature ont permis l'optimisation de l'algorithme comme : la division de l'espace (voir section 1.4.5), la cohérence temporelle (voir section 1.4.4) et les régions de Voronoï (voir section 1.4.9). De plus, l'algorithme est robuste en supportant la gestion des erreurs. Cet algorithme de détection de collisions entre sphères et cubes donne de très bons résultats.

2.4.6 Conclusion

L'algorithme de détection de collisions du logiciel Arka3D est donc une approche hybride inspirée des méthodes et des algorithmes présentés dans le chapitre 1. Chaque objet utilise un algorithme de détection de collisions particulier selon l'objet avec lequel il entre en collision.

La collision entre deux sphères est théoriquement très simple. Par contre, implémentée avec d'autres types de collisions et combinée avec des rebondissements, la technique de base peut générer des erreurs. Un processus de recouvrement après erreur est donc implanté afin d'éliminer une redondance cyclique qui peut se produire.

Les collisions avec les murs qui sont des plans alignés sur les axes d'origine sont sans doute les plus simples. De plus, les murs sont permissifs afin de supporter les balles situées à l'extérieur et améliorer la jouabilité ainsi que la visibilité.

Le disque, qui permet de faire dévier les balles sur les cubes, est un élément essentiel du jeu. De plus, il peut être déplacé abruptement et n'est alors pas conforme aux règles de la cohérence temporelle. Construit par union d'une

sphère et d'un plan, l'algorithme de détection de collisions du disque a aussi pour but d'améliorer la jouabilité en n'étant pas trop restrictif.

Le tableau de cubes, dû au très grand nombre d'objets qui le composent, est de loin l'objet le plus complexe et l'algorithme de détection de collisions qui l'implique doit être optimisé. Une routine en quatre étapes permettant de décomposer les cubes en facettes et d'éliminer rapidement un maximum de celles-ci a donc été développée.

La technique de détection de collisions du logiciel Arka3D est donc très complexe algorithmiquement mais elle résout de manière linéaire le problème de détection.

Dans le chapitre 3 portant sur l'analyse des résultats, nous présenterons les temps d'exécution des différentes détections et nous testerons également l'efficacité des différentes optimisations.

2.5 Gestion d'événements

2.5.1 Introduction

La gestion d'événements permet de créer une réaction suite à une collision. Une application graphique ne réagit pas automatiquement à une collision en appliquant l'effet désiré.

Les événements doivent donc être gérés en influençant le déplacement des objets tout en évitant qu'ils se pénètrent. Cela rend la scène beaucoup plus réaliste en respectant les règles de non pénétration des objets tangibles.

Le rebondissement est l'effet le plus utilisé dans le logiciel Arka3D. Il permet de changer drastiquement la direction de la balle suite à une collision sur un objet statique.

Les sphères entre elles, qu'elles soient balles ou débris, ont aussi un type de rebondissement qui leur est propre. Ce rebondissement devrait vérifier certains critères afin de gérer certains cas particuliers. Dans certains cas, une déviation est utilisée plutôt que l'utilisation d'un rebondissement.

La destruction met fin à la présence d'un objet 3D dans l'environnement. Nous verrons comment cet effet est géré et quel impact il a sur le reste de l'environnement.

Ces différents effets appliqués suite à des collisions permettront d'ajuster la trajectoire des objets et de rendre la scène réaliste.

2.5.2 Rebondissement

Les balles sont les seuls objets à pouvoir rebondir. N'étant pas détruite suite à une collision, une balle se doit de changer de trajectoire afin de ne pas pénétrer dans l'objet avec lequel celle-ci entre en collision.

Le rebondissement se fait donc à l'aide d'un plan comme présenté dans la section 1.5.4. Donc, à l'aide du vecteur incident et la normale d'un plan, il est possible de calculer le vecteur réfléchi qui servira au rebondissement.

Le vecteur incident, qui est la direction de balle qui, est déjà connue. La normale du plan de réflexion doit par contre être déterminée.

Rebondissement sur les murs

Lorsque la balle entre en collision avec un mur, la normale du plan de réflexion est toujours la normale du mur sujet à la collision étant donné que seul le rebondissement plat est possible. Étant donné que les murs sont alignés sur les axes d'origine, il est possible de simplifier le calcul de réflexion en renversant seulement l'une des trois valeurs du vecteur de direction de la balle. Par exemple, si la balle entre en collision sur le mur formé par le plan YZ, la valeur x du vecteur de direction de la balle devient –x.

Rebondissement sur le disque

Le disque est représenté par une sphère coupée par un plan (voir la section 2.4.4). Les rebondissements sur le disque dépendent de l'endroit sur le disque où la collision se produit. Si la balle entre en collision sur le centre du disque, elle rebondit normalement, un peu comme sur les murs. Si elle entre en collision sur les extrémités du disque, elle est déviée plus fortement. La figure 2.26 présente quelques déviations possibles.

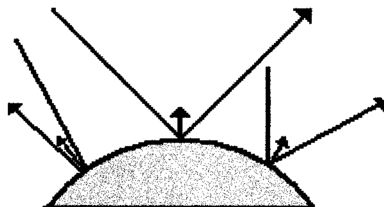


Figure 2.26 Rebonds sur le disque.

La figure 2.26 présente donc trois différents rebonds sur le disque. La normale du plan de réflexion change selon l'endroit de la collision sur le disque.

Rebondissement sur les cubes

Au niveau des cubes, il est possible que la collision se produise sur un coin. Cela aura pour effet changer le plan de réflexion. Lors de la détection de collisions sur les cubes, présentée à la section 2.4.4, les régions de Voronoï [24] utilisées permettent de déterminer si la balle est entrée en collision sur la face, une arrête ou un sommet. Cette déduction oriente la gestion d'événements vers le type de rebondissement approprié.

Dans le cas où le rebondissement est sur la face d'un cube, le même rebondissement que les murs est alors effectué. Les cubes sont également alignés sur les axes d'origine et les normales sont connues (Figure 2.27 (a)).

Lorsque la détection de collisions détermine que la collision se produit sur une arrête ou un sommet, la normale du plan de réflexion doit être calculée. Pour ce faire, le point d'impact doit être connu. Voilà pourquoi l'algorithme de détection de collisions présenté à la section 2.4.4 prend le temps de déterminer le point d'impact. La soustraction entre le point d'impact et le centre de la sphère, qui possède un rayon, nous donne alors le vecteur de la normale du plan de réflexion. La figure 2.27 présente des exemples de réflexions calculées à l'aide de cette technique en version 2D.

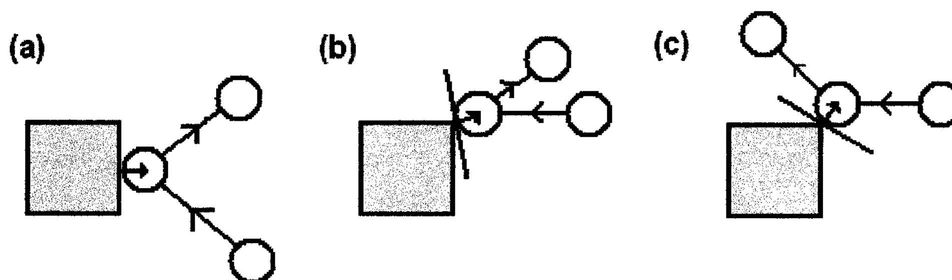


Figure 2.27 Rebondissement sur une face et sur un coin.
(a) Rebondissement sur une face.
(b) Léger rebondissement sur un coin.
(c) Fort rebondissement sur un coin.

Dans la figure 2.27 (a), nous observons un exemple de rebondissement sur une face où la normale de la face est utilisée comme plan de réflexion. Aux figures 2.27 (b) et (c), la normale du plan de réflexion est calculée à partir du point d'impact et du centre de la balle. Une différence est alors observable si la balle rebondit légèrement sur le coin (figure 2.27 (b)) et fortement sur celui-ci (figure 2.27 (c)) dû au changement du plan de réflexion.

Rebondissement entre deux balles

Lorsque deux balles entrent en collision, un rebondissement doit alors être effectué également afin d'éviter que les deux balles ne se pénètrent.

L'angle de réflexion se calcul également à l'aide de la normale d'un plan de collision. Le vecteur de cette normale est déduit à partir de la différence entre les points centraux des deux balles. La figure 2.28 présente cette déduction.

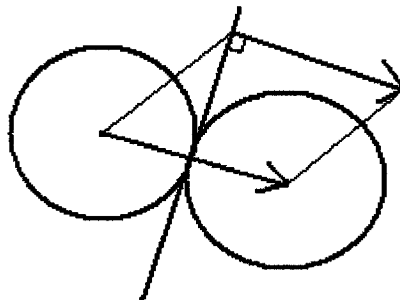


Figure. 2.28 Plan de collision entre deux sphères.

Dans la figure 2.28 montre que le vecteur de la différence entre les deux centres des balles est le même que la normale du plan de collision.

Le vecteur correspondant à la direction de rebondissement est ensuite calculé à l'aide de l'équation (2) présentée à la section 1.5.4.

Dans un autre ordre d'idée, les rebondissements entre les balles doivent être classifiés en deux catégories (rebondissement correct et rebondissement incorrect) afin que le changement de direction soit approprié. Si les rebondissements ne sont pas classifiés, un changement de direction irréaliste peut se produire comme le présente la figure 2.29

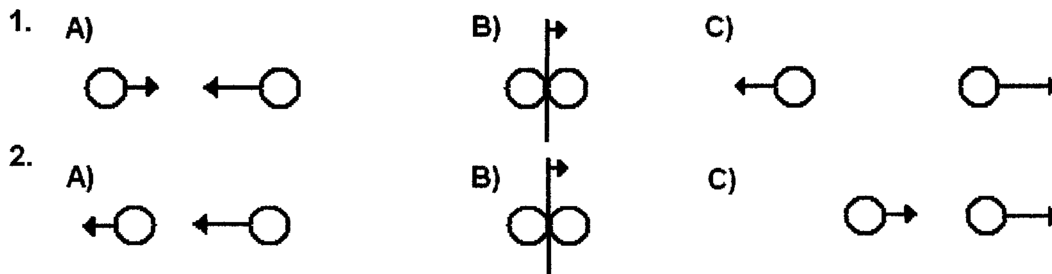


Figure 2.29 Problème de rebondissement avec les sphères.

1. Rebondissement correct
    (a) Deux balles de vitesses différentes avec une direction opposée.
    (b) Plan de réflexion des balles.
    (c) Les deux balles rebondissent correctement.

2. Rebondissement incorrect
   (a) Deux balles de vitesses différentes avec une direction similaire.
   (b) Plan de réflexion des balles.
   (c) Les deux balles rebondissent incorrectement.

Dans la figure 2.29, le phénomène de rebondissement irréaliste est présenté lorsque les balles n'ont pas la même vitesse mais le phénomène peut aussi se produire lorsque deux balles ont la même vitesse mais des angles de déplacement légèrement différents. Le cas présenté à la figure 2.29 1) est un cas simple de rebondissement rationnel. Par contre, en 2), les balles vont dans la même direction et l'une des balles se rapproche de l'autre. Lors de la collision, les deux balles rebondissent à l'aide du même algorithme qu'en 1). Suite à cette collision, les deux balles changent de direction ce qui est complètement irréaliste.

Pour régler ce problème, nous avons établie que deux types de collisions sont possibles : un type normal et un type déviation. Le produit scalaire entre les deux vecteurs direction permet de classifier la collision. Si le produit scalaire est inférieur à zéro, les balles vont dans des directions différentes et un rebondissement normal est effectué. Sinon, le produit scalaire est supérieur à zéro, la balle la plus rapide est rebondie et la balle la plus lente est déviée (voir la section 2.5.3 sur les déviations).

Le problème de rebondissement est causé par une contrainte du jeu qui veut que la vitesse des balles soit constante afin d'éviter qu'une balle soit trop rapide ou trop lente, ou bien soit carrément immobile. Les balles n'ont donc pas de masse. Rappelons que le logiciel Arka3D ne tente pas de simuler des phénomènes réels mais bien de simuler un environnement 3D agréable pour le joueur mais tout de même véridique.

Finalement, le rebondissement des balles est classé en deux catégories soient normal ou dévié. Le rebondissement normal applique un rebondissement sur chaque balle et un rebondissement dévié applique un rebondissement sur une des balles et une déviation sur l'autre. Cette catégorisation dépend des directions des balles et elle permet d'éviter un rebondissement irréaliste.

Conclusion

Les rebondissements sont donc appliqués selon les types d'objets qui sont entrés en collision. L'important est de trouver le bon plan de réflexion afin de faire le rebond approprié. Finalement, les rebonds permettent de respecter les règles de non pénétration des objets.

2.5.3 Déviation

Les déviations sont une simulation d'un objet qui change légèrement de direction suite à une collision. Les seuls objets pouvant être déviés sont les balles.

Les balles sont les seuls objets en mouvement qui résistent à une collision et qui ne sont pas contrôlés directement par le joueur, ce qui font d'elles les seuls objets à pouvoir être déviés. Les cubes restent immobiles et les débris sont détruits.

Une déviation ce produit lors de la collision entre une balle et un débris ainsi que lors de la collision de type déviation (voir la section 2.5.2) entre deux balles.

Pour ces deux types de collision, un vecteur de déviation est calculé. Ce vecteur est déterminé à partir de la direction de l'objet faisant dévier (un débris par exemple) ainsi que la normale du plan de collision déterminée à partir des centres des sphères qui entrent en collision (voir la figure 2.28). La masse des objets, estimée à partir du volume de ceux-ci, affecte la déviation. La formule suivante est alors utilisée :

$$V_R = V_O + (V_D \times M) + (V_N \times M) \tag{4}$$

Où : V_R : est le vecteur de déplacement résultant.
V_O : est le vecteur de déplacement d'origine.
V_D : est le vecteur de déplacement de l'objet faisant dévier.
V_N : est le vecteur de la normale du plan de collision.
M : est un rapport entre les masses des objets.

Le vecteur résultant est donc déduit en fonction du vecteur de déplacement de l'objet avec lequel une collision survient et la normale du plan de collision et le tout est influencé par la masse des objets. Un objet est donc dévié plus fortement si la masse de l'objet avec lequel il entre en collision est plus grande.

Finalement, la déviation est une simulation du phénomène de changement de direction suite à une collision. La direction de l'objet faisant dévier et le point de collision en rapport avec la masse viennent alors influencer la direction de la balle. Cela permet de donner un effet réaliste de déviation suite à une collision.

2.5.4 Destruction

Des objets dans la scène peuvent disparaître ou être détruits durant le déroulement du jeu. Certains de ces objets créeront un effet visuel lors de leur fin de vie. C'est le cas pour les balles, les cubes et les débris.

Les balles peuvent sortir par la partie inférieure de l'environnement de jeu. Lorsque la balle passe derrière le disque et sort suffisamment loin, celle-ci disparaît tout simplement et elle est retirée de la liste des balles actives. Elle n'explose pas, elle est tout simplement partie vers l'infini. Lorsqu'il n'y a plus de balles actives, le jeu est fini.

Les cubes, par contre, explosent. Lorsque qu'un cube atteint son nombre de coups maximum, celui-ci disparaît et crée une explosion ainsi que des débris. L'explosion ajoute un effet visuel et permet au joueur de bien percevoir les changements dans la scène. Les cubes explosifs créent de plus grandes explosions pour donner l'impression d'une explosion plus puissante qui détruit les cubes adjacents.

Les débris, qui sont détruits à la moindre collision avec n'importe lequel des autres objets, ont une taille d'explosion variable. Cette taille est proportionnelle à la taille du débris. Les gros débris créer donc de plus grosse explosion. Lors de l'explosion, le débris est retiré de la liste des débris actifs.

Les destructions sont donc là pour ajouter un effet visuel au jeu. Ces effets visuels permettent au joueur de bien percevoir les changements dans la scène.

2.5.5 Objets parentés

Les balles peuvent être parentées au disque. Ainsi, en début de jeu ou à certains rebondissements particuliers sur le disque, la balle se déplace exclusivement en fonction du déplacement du disque. Cet effet permet au joueur de viser les cubes beaucoup plus facilement.

En début de jeu, la balle est collée au centre du disque. La balle se déplace alors en même temps que le disque. Chacun des déplacements du disque sont alors transmis à la balle. Cette technique est appelée « parentage ». La balle ne se déplace donc plus par elle-même. Le joueur peut donc déplacer le disque et la balle afin de démarrer le jeu à l'endroit voulu. La balle est retournée à son état normal lors d'un clic de la souris par le joueur.

À la difficulté « Facile » de l'objet Arka3D qui est choisie à partir du menu, la balle colle sur le disque à chaque collision sur le disque plutôt que de rebondir. Les paramètres de rebondissement sont alors conservés pour la relance de la

balle par l'utilisateur. Cela rend le jeu beaucoup plus facile en permettant au joueur de bien viser à chaque collision avec la balle en la déplaçant à l'aide du disque.

Un objet parenté se déplace donc en relation directe avec un autre objet. Les balles collées se déplacent alors en fonction du disque facilitant ainsi de beaucoup les règles du jeu Arka3D.

2.5.6 Conclusion

La gestion d'événements dépend donc de plusieurs facteurs. Dans le logiciel Arka3D, la nature des objets graphiques 3D qui entrent en collision est primordiale et à un effet direct sur le type de réaction qui découle de cette collision.

Pour la majorité des collisions concernant les balles, un rebondissement est effectué. Ce rebondissement permet de respecter les règles de non pénétration des objets qui subsistent à la collision comme les murs et certains cubes. Chaque type d'objet conceptuel qui est sujet à une collision à son propre type de rebondissement en déterminant de manière différente son plan de réflexion.

Moins drastique que le rebondissement, la déviation permet de changer légèrement la trajectoire des objets. C'est une réaction à une collision plus légère.

Suite à une collision, des objets peuvent également être détruits. La destruction est alors représentée par un effet visuel d'explosion qui informe le joueur sur les changements dans l'environnement.

Les objets parentés sont une classe à part. Cette technique permet aux objets de s'ancrer à d'autres en suivant leur déplacement. Elle est utilisée avec la balle et le disque afin de faciliter le jeu.

La gestion d'événements est donc essentielle afin de traiter les collisions. Suite à ces collisions, des effets sont ajoutés dans l'environnement afin de le rendre plus réaliste. Ces effets sont donc gérés par la gestion d'événements.

2.6 Effets visuels

2.6.1 Introduction

Les effets visuels sont primordiaux dans le domaine du jeu vidéo. Les premières impressions du joueur se baseront sur les premières images du jeu qu'il perçoit. Il est donc important que les effets visuels soient attrayants.

Dans le logiciel Arka3D, de nombreux effets visuels sont donc présents dont plusieurs ont déjà été présentés à travers la description des objets de la section 2.3. L'utilité de l'objet Explosions (voir section 2.4.8) et de l'objet Débris (voir section 2.3.7) est principalement d'ajouter des éléments graphiques dans la scène afin quelle soit plus agréable pour le joueur. Plusieurs autres effets reliés aux objets sont également présents dans la scène.

La présente section fait un résumé de ces effets visuels en expliquant brièvement comment ils ont été conçues et quels événements les produits.

2.6.2 Murs Translucides

La section 2.3.3 présente comment les murs ont été construits. En résumé, les murs sont construits en deux parties soient une partie extérieure et une partie intérieure.

La partie extérieure est translucide afin de pouvoir voir l'ensemble de la scène de n'importe quel point de vue situé à l'extérieur. Pour dessiner ces murs translucides, le paramètre d'opacité alpha de DirectX est tout simplement utilisé afin de créer un effet de transparence.

Les murs extérieurs sont donc affichés lorsque le point de vue se situe à l'extérieur de l'environnement de jeu. La figure 2.30 présente une vue de côté de la scène à travers les murs translucides.

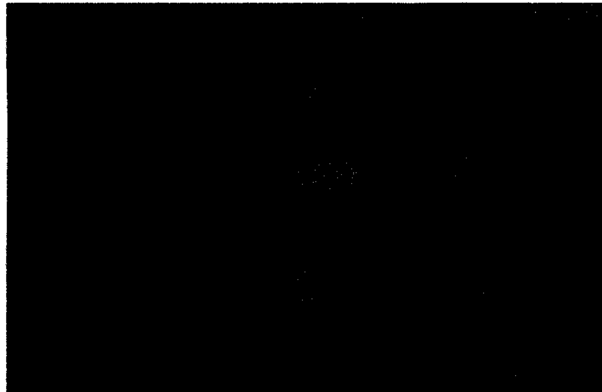
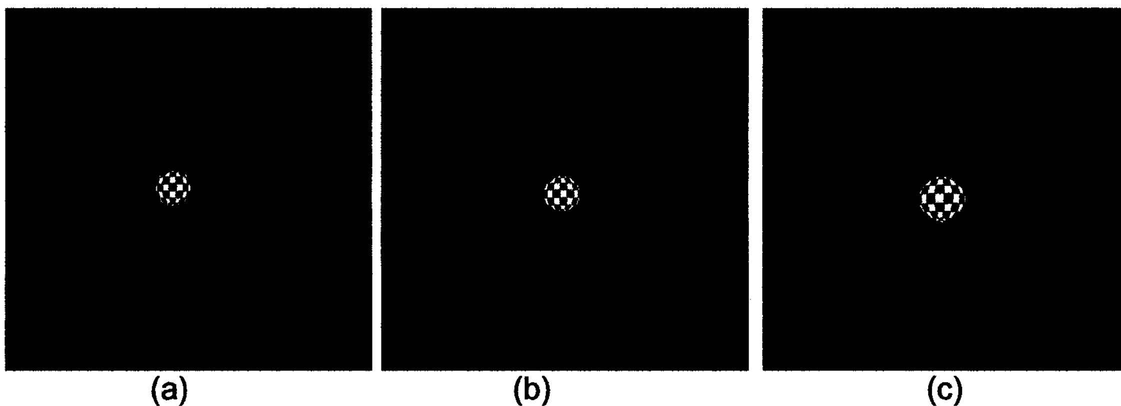


Figure 2.30 Vue de côté à travers un mur translucide.

2.6.3 Effets de lumière sur les murs

En plus de pouvoir être translucides, les murs permettent également de réfléchir la lumière de manière détaillée. Dans la section 2.2.3, nous avons expliqué que les murs étaient composés de plusieurs centaines de polygones afin de bien réfléchir la lumière de type point. Ce type d'éclairage est émis par chaque balle afin de pouvoir les positionner plus facilement en rapport aux murs.

Une lumière de type point permet un éclairage sur 360 dégré dont l'intensité décroît avec la distance. Ce type de lumière est supporté par Direct X qui se base sur la normale de chacun des polygones afin de créer un dégradé de couleur dans ceux-ci. Pour donner un effet de lumière crédible, il nous faut donc un grand nombre de polygones. La figure 2.31 présente les effets d'éclairage sur les murs à l'aide d'un nombre variable de polygones.



(a) (b) (c)

Figure 2.31 Effet d'éclairage d'une balle sur un mur avec un nombre de polygones variables.
(a) Éclairage sur un mur de 18 polygones.

(b) Éclairage sur un mur de 96 polygones.
(c) Éclairage sur un mur de 7938 polygones.

La figure 2.31 présente bien l'importance du nombre de polygones pour l'effet d'éclairage de type point. Dans la figure 2.31 (a), l'effet de lumière est flou avec 18 polygones ce qui est nettement insuffisant et en dessous de ce nombre, l'effet de lumière sur les murs est imperceptible. En (b), l'effet de lumière commence à se préciser avec 96 polygones. Et finalement en (c), l'utilisation de 7938 polygones donne un cercle presque parfait de lumière. Dans la version finale de Arka3D, nous utilisons 1922 polygones qui donnent un effet de lumière largement suffisant sans trop surcharger la scène.

L'éclairage de la scène Direct X dépend directement des normales des polygones. Un polygone face à la lumière réfléchie donc plus de lumière qu'un polygone de côté. Le produit scalaire est aussi utilisé ici pour modifier la luminosité des polygones.

Afin que les murs extérieurs puissent avoir un effet de lumière, la normale de ceux-ci à été renversée afin qu'ils soient sensibles à la lumière intérieure. Cela donne l'impression que la lumière émise par les balles traverse les murs. La figure 2.32 présente le résultat visuel de cette technique.

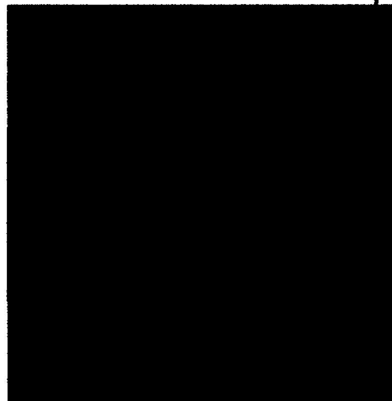


Figure 2.32 L'effet de lumière à travers un mur translucide.

L'effet de lumière des balles est donc perceptible de l'extérieur des murs comme le montre la figure 2.32 grâce au renversement de la normale.

Les effets de lumière sur les murs sont donc efficace dû au grand nombre de polygones qui les composent. De plus, les effets de lumière, qui sont utilisés par les balles, permettent au joueur de mieux percevoir la distance des objets qui compose la scène.

2.6.4 Séquence de textures

Afin de créer un effet de mouvement sur une surface statique, il est possible de créer une séquence de textures. Les cubes indestructibles disposés dans la scène utilisent cette technique.

Les cubes indestructibles utilisent donc une séquence de texture afin de se donner un effet réfléchissant. Pour cela, il utilise un ensemble de onze (11) textures qui sont chargées en entier en mémoire dès le lancement du logiciel Arka3D. Ces textures sont ensuite affichées successivement afin de créer une image en mouvement, un peu comme le fait la télévision. La figure 2.33 présente la liste de ces textures.

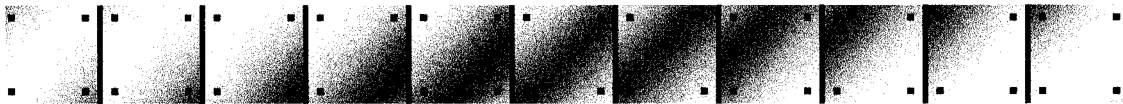


Figure 2.33 Liste des textures utilisée pour les cubes indestructibles.

La figure 2.33 présente donc la suite d'images qui permet de créer l'effet réfléchissant des cubes indestructibles.

Cet effet visuel est assez simple et donne de bons résultats. Il permet de rendre la scène plus animée et l'effet aide également le joueur à remarquer plus facilement les cubes qu'il ne peut pas détruire afin de les éviter.

2.6.5 Explosions et débris

Les exposions vues à la section 2.3.8 et les débris vus dans la section 2.3.7 font également partie des effets visuels permettant d'ajouter du mouvement dans la scène.

Ces effets sont créés lors de la destruction d'un objet et permettent au joueur de mieux percevoir les changements dans la scène tout en améliorant la qualité graphique du logiciel.

Les résultats visuels de ces objets sont présentés dans leurs sections respectives (explosions en 2.3.8 et débris en 2.3.7).

2.6.6 Disque opaque ou translucide

La section 2.3.5, qui présente la structure de l'objet Disque, explique également que le disque peut être translucide ou opaque. Cet effet visuel permet au joueur de toujours bien voir le tableau de cubes et les balles en mouvement.

La distance entre le point de vue et le centre du disque est donc calculée et si cette distance est inférieure à une valeur critique, le disque devient translucide sinon il est affiché complètement opaque. La valeur critique est une constante déterminée arbitrairement afin de permettent un bon visionnement de la scène. Pour rendre le disque translucide, un paramètre alpha est utilisé lors de l'affichage du disque par DirectX de la même façon que les murs extérieurs.

Les résultats visuels de cette technique sont présentés à la fin de la section 2.3.5 portant sur l'objet Disque.

2.6.7 Points de vues multiples

Afin que le joueur puisse bien observer la scène sur tous les angles, une panoplie de points de vue est disponible. L'utilisateur du logiciel Arka3D peut donc changer comme bon lui semble son point de vue sur la scène.

En appuyant sur des touches au clavier, le joueur peut donc sélectionner différents points de vue. Il existe trois types de vue que le joueur peut utiliser :

- Vue derrière le disque.
- Point de vue statique.
- Suivi des balles.

Vue derrière le disque

Cette vue permet de suivre le disque en étant toujours derrière celui-ci. C'est une vue de type troisième personne qui peut être déplacée à l'aide de la souris et le clavier en même temps que le disque. Cette vue permet au joueur de mieux viser les balles et elle est par conséquent la plus utilisée.

Dans ce type de vue, le disque peut devenir translucide dû à la proximité du point de vue. De plus, le joueur peut utiliser la roulette de la souris (mouse wheel) afin d'ajuster la proximité du point de vue en rapport avec le disque. La figure 2.34 présente un exemple de la vue lorsque le disque est translucide.

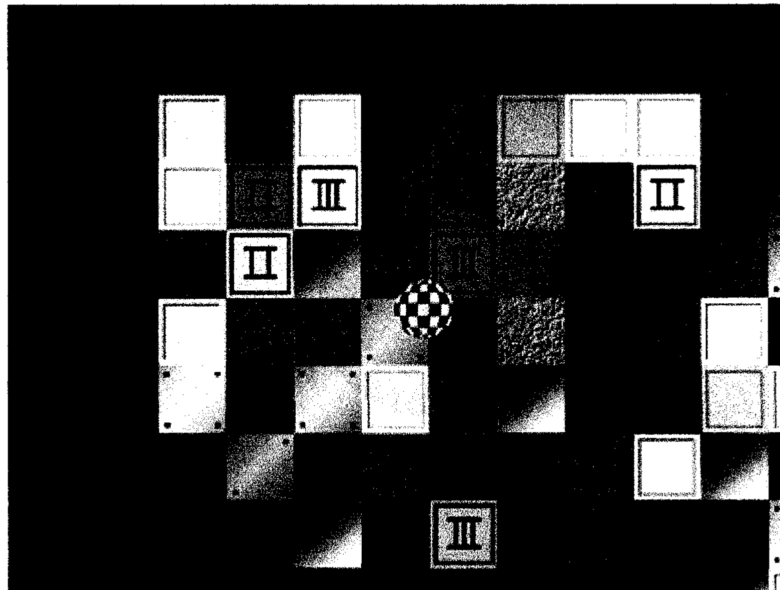


Figure 2.34 Vue derrière le disque translucide.

Point de vue statique

Un certain nombre de points de vue statiques sont également disponibles dans l'environnement de jeu afin de permettre au joueur de bien pouvoir suivre l'action dans la scène.

Ces points de vue sont disposés partout autour de la scène et le joueur peut les sélectionner à partir du clavier. Ceux-ci sont souvent éloignés afin de donner une vue d'ensemble à l'utilisateur. Les figures 2.35 et 2.36 présentent deux de ces vues. La figure 2.35 montre une vue loin derrière la scène et la figure 2.36 présente une vue de côté.

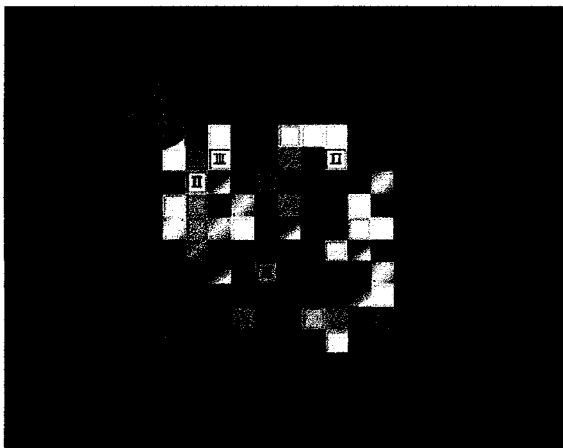


Figure 2.35 Vue loin derrière la scène.

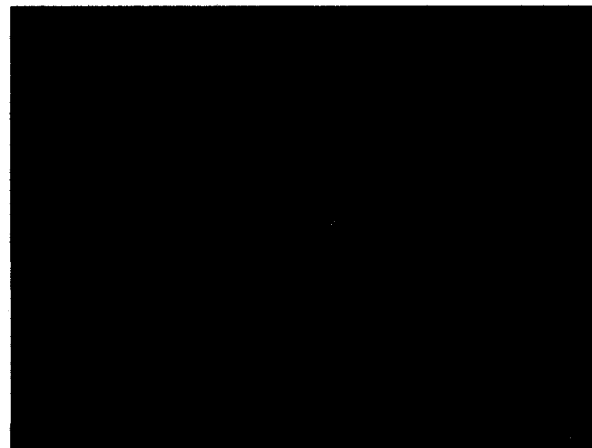


Figure 2.36 Vue de côté sur la scène.

Suivie des balles

Un point de vue très intéressant est de suivre une balle afin de voir ses déplacements et ses rebondissements. Bien que le joueur ait l'impression d'être derrière la balle, un peu comme le disque, celui-ci ne peut pas déplacer directement ni la balle ni la vue.

Ce point de vue est très pratique lorsque la balle se situe à l'intérieur du tableau de cubes et nous assure de toujours pouvoir observer les balles, le joueur peut donc, à partir du clavier, suivre n'importe quelle balle active dans la scène. La figure 2.37 présente un exemple de suivi d'une balle.

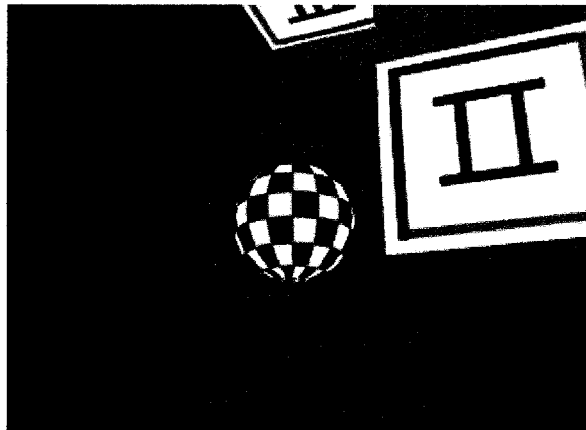


Figure 2.37 Suivi d'une balle.

Changement de vue

Le changement des vues peut être exécuté à partir de touches au clavier. À tout moment, le point de vue peut donc être changé sans affecter le déplacement des objets et la gestion d'événements reliés à ceux-ci. Par contre, le changement de vue se fait de manière saccadée si aucune transition n'est faite entre les changements de vues. Cela a pour effet de désorienter l'utilisateur qui doit alors se questionner sur la position du point de vue qui lui est présenté.

C'est pourquoi une transition lors des différents changements de vue où lors du suivie d'une balle a alors été instaurée. Pour ce faire, la position du point de vue actuelle est conservée et à chaque itération, le point de vue converge graduellement vers le nouveau point de vue. L'équation suivante est alors utilisée :

$$V_R = \frac{(C-1) \times V_O + V_F}{C} \quad (5)$$

Où : C : est une constante inversement proportionnelle à la vitesse de

convergence.

V_O : est la position d'origine du point de vue déduite à l'itération précédente.

V_F : est la position future désirée du point de vue.

V_R : est la position résultante du point de vue.

Cette formule permet donc au point de vue de s'ajuster graduellement à une nouvelle position lors d'un changement de vue ou d'un changement de direction de la balle pour le suivi d'une balle. Le seul cas où cette technique n'est pas utilisée est pour la vue derrière le disque afin que la vue se repositionne instantanément pour permettre au joueur de mieux viser.

Conclusion

Les différents points de vue sont très intéressants en permettant la visualisation de la scène sous plusieurs angles. Cela permet au joueur de bien regarder où ce situe les différents objets qui composent l'environnement.

De plus, une transition entre les points de vues est instaurée afin de permettre un changement graduel de l'angle de vue et de ne pas désorienter le joueur.

Ces nombreux points de vue sont donc essentiels pour une bonne visibilité de la scène et de ses objets.

2.6.8 Conclusion

Tous les bons jeux vidéo possèdent de bons effets spéciaux qui rendent l'environnement visuellement plus attrayant. Cette dernière section a permis d'explorer les effets spéciaux qui ont été intégrés au jeu Arka3D.

Ces effets visuels font partie intégrante des objets conceptuels présentés à la section 2.3. Les cubes, par exemple, utilisent une séquence de texture afin de créer un effet réfléchissant. Certains objets, comme l'objet Explosions et l'objet Débris, sont présents principalement pour ajouter des effets spéciaux dans la scène.

La translucidité, utilisée sur le disque et les murs, permet au joueur de toujours bien voir les principaux objets présents dans l'environnement de jeu de Arka3D. De plus, le changement de point de vue permet au joueur de pouvoir

observé l'action qui se déroule dans la scène sous plusieurs angles sans être désorienté.

Les effets visuels sont donc essentiels dans tout bon jeu vidéo. Voilà pourquoi le jeu Arka3D en utilise plusieurs et qu'une section complète de ce mémoire explique ces effets.

2.7 Erreurs de détection de collisions et de gestion d'événements.

Pour diverses raisons, certaines erreurs d'ordre contextuelle et algorithmique ont dû être corrigées. Ces erreurs peuvent causer des événements irréalistes et même provoquer des boucles algorithmiques qui affecteront la stabilité du logiciel.

Certaines limitations du jeu, vues aux sections 2.2.2 et 2.2.3, qui ont été instaurées afin de permettre une meilleure jouabilité, ne sont pas conforme aux règles physiques conventionnelles ce qui causent certaines incohérences dans certaines situations.

Nous avons vu que les balles ont toujours une vitesse constante et n'ont pas de masse. Cette contrainte ne permet pas l'échange d'énergie de vélocité entre les balles. Cela cause des situations lors de collisions entre plusieurs balles près d'autres objets (les murs par exemple) provoquant la pénétration entre les balles. Cette erreur a été corrigée par un déplacement intemporel tel que vue à la section 2.4.2.

Ce déplacement intemporel est par contre lourd de conséquence. En effet, lors de ce déplacement, aucune détection de collisions n'est effectuée puisque cette détection est erronée. Cela peut avoir pour effet la pénétration des objets déplacés intemporellement avec d'autres objets (cubes et murs). Les murs permissifs permettent de résoudre ce problème comme nous avons vu à la section 2.3.3. Pour les cubes, une seconde détection implicite basée sur la division de l'espace sera utilisée pour vérifier si les objets en mouvement sont à l'intérieur des cubes (voir la section 2.4.5).

La détection entre les objets et les faces des cubes se fait à l'aide de vecteurs de déplacement. Ces vecteurs permettent de déduire l'intersection avec le plan des cubes afin de déterminer la collision. Ces vecteurs ont par contre une minime erreur mathématique dû à la précision des variables à virgule flottante qui peut causer un intervalle vide de détection entre deux vecteurs à des temps successifs. Si la collision se produit à cet intervalle, l'algorithme ne détectera pas cette collision. La seconde détection implicite permet également de générer l'événement de collisions. L'instauration d'une marge d'erreur à l'aide d'une constante Epsilon a également été testée. Cette marge d'erreur faisait

malheureusement décupler le nombre d'erreurs de rebondissement récursif sur les coins.

L'erreur la plus problématique a été le rebondissement récursif sur les coins qui causait une boucle infinie au niveau de la détection de collisions entre les balles et les cubes indestructibles. Cela avait pour effet d'arrêter le déroulement du jeu. Deux causes peuvent générer ce type de bouclage. La première étant l'imprécision mathématique qui peut générer des rebondissements dans deux sens opposés au même instant. L'autre étant la contrainte de direction des balles, exposée à la section 2.2.3, qui pourra occasionner la même erreur que l'imprécision mathématique. Pour sortir de cette boucle, un déplacement intemporel sera effectué lorsque deux détections au même instant sont déduites avec les deux mêmes objets.

Les deux principales causent d'erreurs découlent essentiellement des contraintes du jeu et des imprécisions mathématiques. Ces erreurs sont maintenant détectées et résolues à l'intérieur de l'algorithme de détection de collisions et de gestion d'événements du logiciel Arka3D permettant ainsi le bon déroulement du jeu et le réalisme de la scène.

2.8 Conclusion

Le chapitre 2 décrit le fonctionnement général du logiciel Arka3D. Ce logiciel d'amusement a été conçu à l'aide de Visual C++ et DirectX dans le but de mettre en pratique les différentes approches concernant la gestion d'événements et d'interactions entre les objets graphiques décrites dans le chapitre 1.

Dans la première partie du chapitre 2, nous avons montré le fonctionnement général du jeu en rapport avec ces prédécesseurs en 2D. La technique de programmation, inspirée de la programmation extrême mais surtout de la programmation objet, à également été présentée.

Ensuite, les différents objets qui constituent l'environnement de jeu ont été expliqués en détail ainsi que leurs principaux attributs et méthodes. L'objet Arka3D, qui intègre tous les autres objets comme les objets Cubes, Balles et Disque, est donc le corps du jeu.

Les différents algorithmes de détections de collisions ont été présentés dans la section 2.4. Notons que le logiciel Arka3D utilise une méthode hybride inspirée de techniques existantes [18] [24] [25]. L'algorithme tient compte de la nature de chacun des objets graphiques afin d'optimiser le temps d'exécution. De plus, l'algorithme de détection de collisions de Arka3D détecte et solutionne les erreurs.

La gestion d'événements, décrite dans la section 2.5, permet à l'environnement de réagir aux collisions pour le rendre plus réaliste. Ces réactions, principalement des rebondissements, sont présentes dans le jeu Arka3D afin de respecter les règles de non pénétration des objets.

Finalement, à la section 2.6, nous avons expliqué les différents effets visuels utilisés dans le logiciel Arka3D. Ces effets spéciaux, tels que les explosions, la rotation de textures et la translucidité de certains objets, permettent de rendre la scène plus vivante. Les nombreux points de vue aide également le joueur en lui présentant la scène sous plusieurs angles. Les effets spéciaux augmentent par contre le temps de rendu de la scène.

Voilà donc comment le jeu complet, dans le cadre du projet Arka3D, a été développé pour cette maîtrise sur la gestion d'événements et d'interactions dans un environnement 3D. Le chapitre suivant présente différents tests d'efficacité et le temps d'exécution afin de comparer les différentes optimisations qui ont été utilisées dans le logiciel Arka3D.

Chapitre 3 : Expérimentation et discussion

3.1 Introduction

Les différents algorithmes de détection de collisions et les optimisations de ceux-ci ont une efficacité relative à leur environnement. C'est pourquoi une approche hybride a été développée afin d'optimiser au maximum la détection de collisions utilisée avec l'environnement 3D du logiciel Arka3D.

Dans le but d'évaluer les différentes optimisations de détection de collisions vues dans le chapitre 1 et appliquées dans le modèle présenté au chapitre 2, nous avons soumis le logiciel Arka3D à plusieurs scénarios. Dans ces différents tests, plusieurs éléments sont changés afin de modifier l'environnement le plus possible. Le nombre de cubes dans le tableau et la disposition de ceux-ci, la taille du disque et des balles, ainsi que le nombre et la vitesse des balles sont tous des éléments qui sont susceptibles de varier d'un scénario à l'autre. Chacun des scénarios sera exécuté une seule fois et jusqu'à deux cents mille informations sont enregistrées à chaque test. C'est sur les moyennes de ces informations que nous baseront nos comparaisons. Notons que lancer plusieurs fois le même test donne des résultats très similaires.

De plus, afin de comparer l'efficacité des différentes optimisations, nous avons délibérément retiré certaines d'entre elles pour vérifier leurs effets sur le temps d'exécution.

Finalement, les différents scénarios seront comparés principalement sur les temps d'exécution afin d'en tirer des conclusions lors de la discussion. L'efficacité des principales optimisations sera alors évaluées.

Les résultats de l'expérimentation du logiciel Arka3D sont donc divulgués dans ce chapitre et une évaluation à l'aide d'une comparaison entre les différents scénarios est également présentée et commentée.

3.2 Expérimentation

3.2.1 Introduction

Le logiciel Arka3D a été soumis à un grand nombre de scénarios afin de tester son efficacité et sa stabilité. Nous présenterons dans ce chapitre six scénarios avec différentes configurations afin d'évaluer les performances de

l'algorithme de détection de collisions de ce logiciel et l'efficacité des optimisations utilisées.

L'objectif de l'algorithme de détection de collisions et de gestion d'événements est de résoudre de manière réaliste les différentes interactions entre les objets en un temps raisonnable afin de laisser un maximum de temps pour l'affichage graphique. Le logiciel doit fournir en tous temps, sur un ordinateur commun (Pentium 4) un minimum de 40 images par seconde afin d'avoir une image fluide. Étant donné que la construction de la scène et l'affichage de celle-ci sont extrêmement gourmant, la détection de collisions doit donc se faire en une fraction du temps requis pour l'affichage.

Les tests ont été réalisés sur un Pentium 4 3000mhz, muni de 1 gigaoctet de mémoire vive et une carte vidéo 3D standard (Asus GeForce FX5200). Le logiciel utilise des structures en mémoire statique afin de réduire le temps d'exécution qui occupe environ 20 méga octets.

Le nombre de détections de collisions effectuées en une seconde varie énormément, soit de 40 à 400 dans les différents scénarios présentés dans ce chapitre. Les moyennes de toutes les itérations sont donc utilisées afin d'avoir un aperçu général sur l'efficacité de la détection de collisions.

Dans les différents scénarios, les temps d'exécution de l'affichage (Output), de la gestion des contrôles (Input) et enfin de la détection de collisions et la gestion d'événements (Traitement) sont comptabilisés afin d'observer leurs pourcentages d'utilisation du temps de CPU.

De plus, l'ensemble des détections de collisions entre les balles et les cubes sont regroupées afin de vérifier l'efficacité de cet algorithme. Notons que cette détection est la plus complexe et la plus optimisée étant donné le grand nombre d'objets que celle-ci doit traiter.

Nous avons donc testé différents scénarios afin de compiler des résultats dans le but de comparer l'efficacité de la détection de collisions et les différentes optimisations.

3.2.2 Scénario 1 : Niveau simple

Le premier scénario est le test le plus simple soit avec une seule balle, peu de cubes et à la difficulté de jeu « facile ». La difficulté « facile » implique une plus grosse balle qui se déplace plus lentement. L'objectif de ce scénario est de tester l'efficacité de la détection de collisions dans un environnement simple.

Ce scénario est en fait le premier tableau de cubes du jeu final utilisable par l'utilisateur moyen. Le tableau possède 44 cubes et 210 facettes au départ.

L'élimination de facettes, vue dans le chapitre 2 (section 2.3.6), permet donc d'éliminer 54 facettes sur 264, soit environ 20%, à l'aide de cette élimination des surfaces cachées. C'est peu comparé aux autres scénarios. La raison est simple, s'il y a moins de cubes, la probabilité d'avoir des cubes adjacents est diminuée ce qui réduit l'efficacité de cette première optimisation. La figure 3.1 présente l'aperçu visuel du premier scénario.

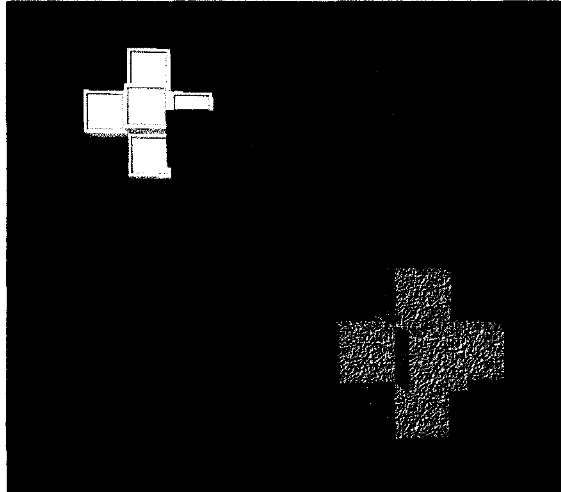


Figure 3.1. Aperçu du tableau de cubes du scénario 1.

Le tableau de cubes, présenté dans la figure 3.1 est donc constitué de sept ensembles de sept cubes formant des croix dont le cube central est un cube explosif. Ce cube explosif, s'il est touché par une balle, détruit tous les cubes adjacents soit tous les cubes de l'ensemble de sept cubes où il est situé. La meilleure stratégie pour le joueur est donc de faire exploser l'un des cubes d'un ensemble afin d'avoir accès aux cubes explosifs et de pouvoir ensuite détruire facilement les cubes restant de cet ensemble.

Le premier graphique, à la figure 3.2, présente la distribution du temps d'utilisation du CPU moyen au cours de l'exécution du scénario 1.

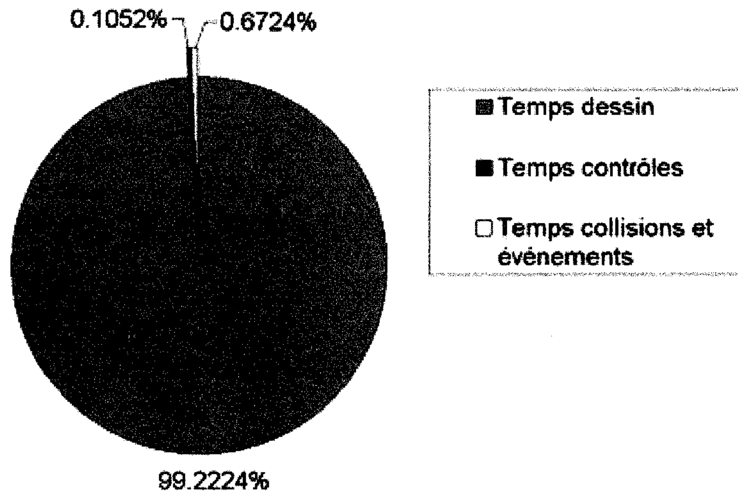


Figure 3.2 Répartition du temps d'exécution des différentes routines pour le scénario 1.

L'objectif de réduction du temps de la détection de collisions et la gestion est donc atteint étant donné que celui-ci représente moins de un pourcent du temps d'exécution. Les contrôles, gérés par DirectInput de DirectX, prennent un temps six fois inférieurs au temps de collisions et événements. L'affichage graphique peu donc prendre la majorité du temps de d'exécution soit plus de 99% afin de rendre la visualisation de la scène plus fluide en permettant plus de cinquante images par seconde.

Le logiciel est stable pour le scénario 1. De plus, dans les données analysées, il n'y a eu aucun recouvrement après erreurs. Cela indique que, dans un environnement simple avec une seule balle, l'efficacité de l'algorithme de détection de collisions provoque très rarement des erreurs.

Le scénario 1, qui est un tableau de cubes utilisé dans la version finale du jeu, est donc réalisé avec succès par le logiciel Arka3D et les temps d'exécution fournis par celui-ci sont largement satisfaisants.

3.2.3 Scénario 2 : Niveau intermédiaire

Le scénario 2 est également un niveau présent dans la version finale du jeu Arka3D. Celui-ci est d'une complexité moyenne et le niveau de difficulté est également passé à « Intermédiaire ». Ce niveau implique une balle un plus petite et plus rapide qu'à la difficulté « Facile ». Le but de ce scénario est de tester le logiciel dans un environnement moyennement complexe.

Le tableau de cubes est composé de 144 cubes et 372 facettes. Il y a donc ici une élimination de 492 facettes, ou 57%, seulement avec l'élimination

des facettes cachées. Cette optimisation devient donc très intéressante. Visuellement, le tableau de cubes est composé de trois carrés de cubes superposés de couleurs différentes. La figure 3.3 présente visuellement le tableau de cubes utilisé dans le scénario 2.

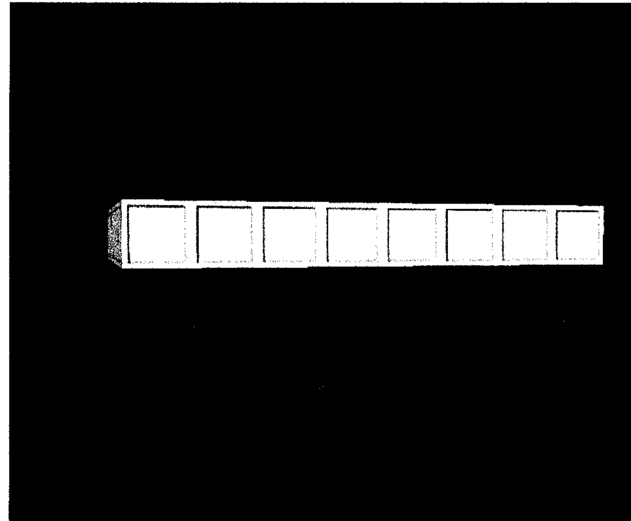


Figure 3.3 Aperçu du tableau de cubes du scénario 2.

La figure 3.3 présente le tableau de cubes qui est composé de trois carrés de cubes (8X6) de couleur bleu, blanc et rouge. L'objectif est un peu plus abstrait que dans le premier scénario, la meilleure stratégie est de faire passer la balle entre les carrés de cubes afin qu'elle rebondisse d'un carré à l'autre. Avec un angle suffisant, la balle rebondira de nombreuse fois et détruira un grand nombre de cubes.

Les temps d'exécution des différentes routines sont répartis dans la figure 3.4 qui suit.

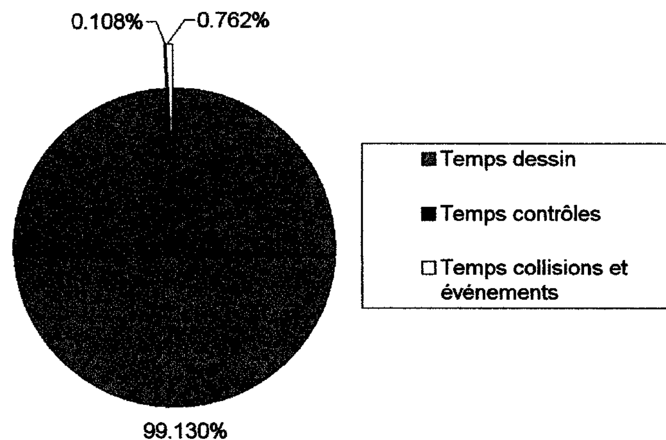


Figure 3.4 Répartition des temps d'exécution des différentes routines pour le scénario 2.

Nous pouvons donc observer dans la figure 3.4, la répartition du temps de dessin (ou affichage), du temps de contrôle ainsi que le temps de la détection de collisions et de la gestion d'événements. Les temps sont similaires au scénario 1. L'algorithme de détection de collisions prend légèrement plus de temps soit 0.76%, environ 0.1% de plus que dans le scénario 1. La raison est simple, il y a trois fois plus de cubes à traiter. L'affichage graphique utilise donc plus de 99% du temps de CPU ce qui donne un rendu très fluide à l'écran.

Le logiciel Arka3D supporte très bien le scénario 2. Encore ici, il n'y a eu aucune reprise après erreur. Malgré la proximité des carrés de cubes, l'algorithme de détection de collisions détecte correctement les collisions et applique les bons rebondissements.

Le scénario 2 est donc réalisé avec succès par le logiciel Arka3D en donnant des temps comparables au scénario 1. Malgré un nombre de cubes plus important et une balle plus rapide, il résout la détection de collisions en un temps très légèrement supérieur.

3.2.4 Scénario 3 : Niveau difficile

Ce scénario a été utilisé durant le développement pour un grand nombre de tests de stabilité. Il s'agit d'un tableau de cubes complet avec des types de cubes aléatoires. En plus d'utiliser 4 balles, les balles sont plus petites et plus rapides (difficulté « difficile »). Le but de ce test est de vérifier la stabilité du logiciel Arka3D et l'efficacité de l'algorithme de détection de collisions avec un tableau de cubes le plus complexe possible.

Nous avons mentionné précédemment (voir la section 2.3.6) que le nombre maximum de cubes possibles dans un tableau est de 1000 (10X10X10). Le scénario 3 possède donc 1000 cubes de type variable : cubes normaux, cubes 2 coups, cubes 3 coups, cubes explosifs et cubes indestructibles. Il y a donc 600 facettes d'afficher en début de traitement plutôt que 6000, une optimisation de 90% faite par l'élimination de facettes cachées. En moyenne, l'optimisation sera d'environ 60%. Cette optimisation est donc très efficace dans un tableau de cubes très lourd. Le tableau de cubes est donc un méli-mélo de cubes différents comme le présente la figure 3.5.

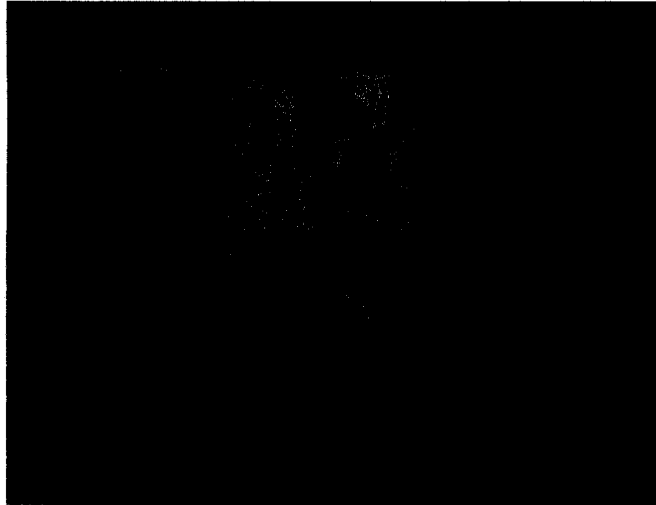


Figure 3.5 Aperçu de tableau de cubes du scénario 3.

Les cubes aléatoires qui forment le tableau de cubes du scénario 3 n'ont pas un aspect graphique et une jouabilité intéressants. Ce tableau de cubes n'est donc pas retenu dans la version finale du jeu Arka3D. Il est exclusivement utilisé à des fins de test d'efficacité et de stabilité. Par ailleurs, les scénarios 4, 5 et 6 utilisent le même tableau de cubes afin de tester l'efficacité des optimisations.

Afin de faciliter, d'allonger et de complexifier considérablement le test, les quatre balles rebondissent par le mur du fond qui cause habituellement la destruction d'une balle. Cela a pour effet de conserver les quatre balles en jeu sans avoir à les faire rebondir avec le disque. Sans cette modification, il est très difficile d'enregistrer plus d'une minute de test dû à la complexité du jeu.

La figure 3.6 présente la répartition du temps d'affichage graphique, de la saisie des contrôles, de la détection de collisions et de la gestion d'événements.

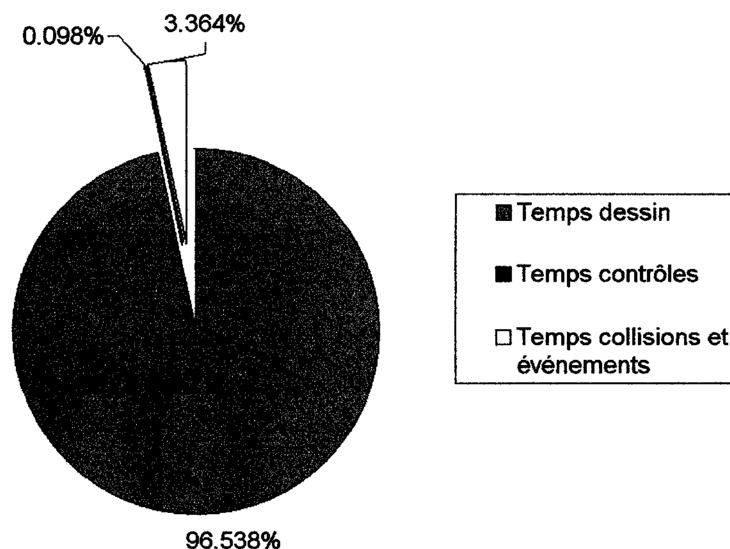


Figure 3.6 Répartition du temps d'exécution des différentes routines pour le scénario 3.

La figure 3.6 nous permet d'observer une augmentation du temps de traitement de la détection de collisions qui est cinq fois supérieure aux scénario 1 et 2. La principale raison est le nombre de balles qui a augmentés de un à quatre ce qui provoque beaucoup plus de paires d'objets et de tests de collisions. Par contre, les optimisations faites sur le type de collisions balles avec cubes permettent d'exécuter un test de collision sensiblement similaire que dans les autres scénarios malgré le nombre de cubes plus élevé. Nous comparerons ces tests lors de la discussion.

Le pourcentage de temps alloué pour le dessin est de 96%, ce qui est suffisant pour produire un minimum de 45 images par seconde. Ce taux d'affichage est fluide et très correct.

Par contre, étant donné la complexité du tableau de cubes, du nombre de balles et des nombreux tests de collisions effectués dans ce scénario, soit plus de dix milles avec au moins une facette retenue, un recouvrement après erreur à dû être effectué. En effet, une balle aurait pénétrée à l'intérieur d'un cube sans rebondir. Cette erreur pourrait être causée par plusieurs facteurs, comme l'imprécision mathématique ou le rebondissement de deux balles à proximité d'un cube. Cette erreur a été recouvrée avec succès et le jeu s'est poursuivi normalement.

Malgré un recouvrement après erreur, le scénario 3 est considéré hautement satisfaisant dû à son temps d'exécution très raisonnable. Cela montre que l'algorithme de détection de collisions du logiciel Arka3D est rapide, stable et efficace même dans un environnement anormalement lourd.

3.2.5 Scénario 4 : Test sur la division d'espace

Le scénario 4 a pour but de vérifier l'efficacité de la division dynamique de l'espace vue à la section 1.4.5 et appliquée à l'étape 1 de la détection de collisions entre la sphère et les cubes vue à la section 2.4.5. Pour ce faire, le même tableau de cubes que le scénario 3 est utilisé afin de comparer les résultats.

Dans ce scénario, l'optimisation de la division de l'espace est retirée afin de vérifier son efficacité. Une balle en mouvement est donc testée avec l'ensemble des cubes plutôt qu'avec seulement les cubes à proximité. Les autres paramètres sont les mêmes que ceux présentés dans le scénario 3 à la section 3.2.4 soit quatre balles à la difficulté « difficile ». Le tableau de cubes utilisé est donc le tableau complet vue à la figure 3.5.

La figure 3.7 présente les temps d'exécution des différentes routines pour le scénario 4.

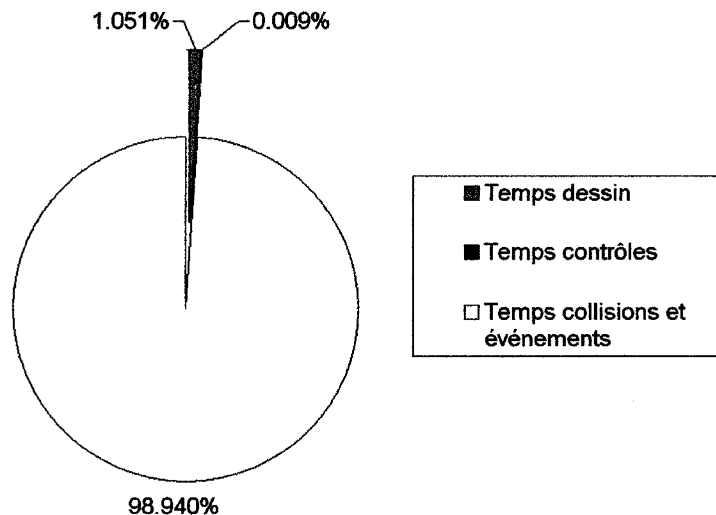


Figure 3.7 Répartition du temps d'exécution des différentes routines pour le scénario 4.

Dans cette figure, le temps d'exécution de l'algorithme de détection de collisions prend maintenant près de 99% du temps de CPU ce qui est inacceptable. Le nombre d'images par seconde affichées tombe en dessous de 3 ce qui rend la séquence d'images très saccadée. De plus, étant donné que les contrôles du jeu sont validés à chaque image, le temps alloué au contrôle est également réduit se qui rend le jeu encore plus difficile à utiliser.

La raison est simple. Sans la division de l'espace, les facettes sont testées pour l'ensemble des cubes à chaque détection de collisions. Dans la seconde

étape de la détection de collisions entre les sphères et les cubes (section 2.4.5), l'algorithme calcule la distance entre les facettes et les balles et tri ensuite les facettes. Cela donne environ 800 calculs de distance qui est très lourd et un tri sur 800 éléments à chacune des détections de collisions.

La détection de collisions plus lourde et complète n'est pas faite sur l'ensemble des facettes grâce à l'optimisation arrêtant les calculs lors de la vérification de la distance minimum où une collision est possible (Étape 3). Sans cette seconde optimisation, les résultats seraient encore plus désastreux. C'est donc la présélection des facettes et le tri de celles-ci qui prennent un temps déraisonnable.

Dans le test enregistré, le nombre de recouvrements après erreurs est également très élevé soit 61. Plusieurs raisons expliquent ce grand nombre d'erreurs. La principale étant que les balles se déplacent en fonction du temps écoulé ce qui provoque un grand déplacement lorsque le nombre d'images par seconde est réduit comme dans le scénario 4. Ce grand déplacement occasionne un plus grand nombre de collisions récursives. Également, plus de paires d'objets sont sélectionnées ce qui engendre plus de tests de collisions donc plus de risque d'imprécisions mathématiques. Par contre, tous les recouvrements après erreur ont été réalisés avec succès et le logiciel Arka3D à réaliser l'ensemble du test sans montrer de signe d'instabilité.

Le scénario 4 est donc une évaluation de la division de l'espace dynamique utilisée dans l'algorithme de détection de collisions entre les sphères et les cubes du logiciel Arka3D. Nous avons montré que sans cette optimisation l'algorithme devient très lourd et l'affichage graphique devient inadéquat. Cela prouve l'efficacité de la division de l'espace. Finalement, le scénario 4 a été tout de même complété par Arka3D ce qui montre la grande stabilité du logiciel Arka3D.

3.2.6 Scénario 5 : Test sur la cohérence temporelle

Encore dans le but de vérifier une optimisation de l'algorithme de détection de collisions, l'optimisation sur l'élimination des facettes selon leur orientation a ici été retirée. Le même tableau de cubes et la même configuration que le scénario 3 ont ici été utilisés afin de comparer les résultats.

Le tableau de cubes utilisé est donc le tableau complet de cubes différents présenté dans la section 3.2.4 et la figure 3.5. Quatre balles sont encore utilisées et la difficulté du jeu est fixée à « Difficile » afin d'avoir des petites balles rapides.

L'optimisation retirée est celle qui permet, à l'étape 2 de la détection de collisions avec les cubes, de vérifier la normale de la facette avec la direction de la balle afin de vérifier si une collision est possible. La cohérence temporelle est

ici exploitée. Cela permet d'éliminer en moyenne la moitié des facettes à l'aide d'un simple produit scalaire.

La figure 3.8 présente le résumé des temps d'exécution pour le dessin, la saisie de contrôle ainsi que la détection de collisions et la gestion des événements.

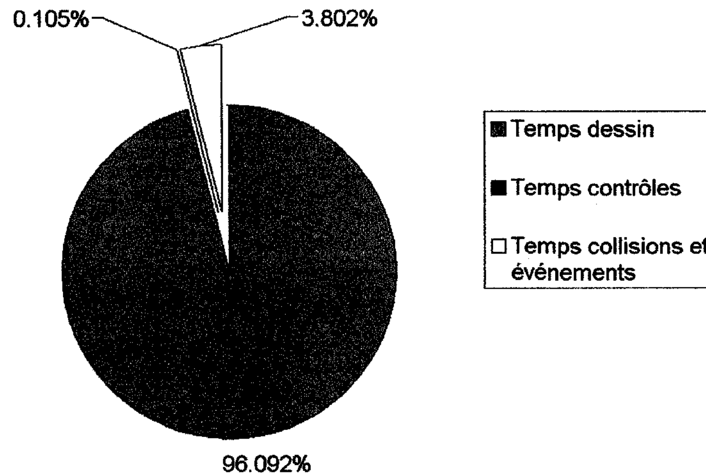


Figure 3.8 Répartition du temps d'exécution des différentes routines pour le scénario 5.

Le graphique présenté à la figure 3.8 montre que le temps d'exécution de la détection de collisions est sensiblement supérieur à celui présenté dans le scénario 3 à la figure 3.6. La raison étant que plus de facettes sont retenues à l'étape 2 de la détection de collisions entre les sphères et les cubes. Donc plus de facettes seront triées de même que plus de facettes seront soumises à un test complet.

Le problème le plus notable causé par le retrait de cette optimisation est au niveau du nombre d'erreurs détectées qui est ici de 18. Deux causes sont susceptibles de créer des erreurs. D'abord, le nombre de détections de collisions qui peuvent causer une erreur d'imprécision mathématique est augmenté. Ensuite, si la balle arrive presque parfaitement sur le coin cube, celle-ci pourra entrer en collision avec 3 facettes différentes et certaine de ces facettes pourraient avoir une normale similaire à la direction de la balle ce qui a pour effet de faire rebondir la balle à l'intérieur du cube. L'optimisation sur l'élimination de facettes selon leur orientation et la direction de la balle permet donc également de retirer des facettes susceptibles de causer des erreurs. Encore une fois, le recouvrement après erreur a permis la bonne continuation du déroulement de la scène.

L'optimisation sur l'élimination de facettes selon leur orientation et la direction de la balle permet donc de réduire sensiblement le temps d'exécution

de la détection de collisions mais aussi d'éviter certaines erreurs. Malgré cela, sans cette optimisation le logiciel Arka3D fonctionne tout de même correctement.

3.2.7 Scénario 6 : Test sur l'élimination de facettes à l'aide de la distance

Ce dernier test permet d'évaluer l'efficacité de l'élimination de facettes grâce à la distance entre celles-ci et la sphère. Suite au tri de facettes, à l'étape 2 de l'algorithme de détection de collisions entre les sphères et les cubes (section 2.4.5), la détection de collisions s'arrête lorsque la distance rend impossible la collision entre les deux entités. Pour faire ce test, la même configuration qu'au scénario 3 a été utilisée afin de comparer les deux scénarios lors de la discussion.

L'environnement utilisé est donc le tableau de cubes aléatoires complets présenté à la figure 3.5. Encore une fois, quatre balles petites et rapides évolueront dans la scène.

L'optimisation portant sur l'arrêt de la détection des facettes triées selon leurs distances avec la balles à donc été volontairement retirée afin de tester l'efficacité de la technique. Cela entraîne inévitablement un plus grand nombre de tests complets de collision.

La figure 3.9 présente les temps d'exécution pour les routines de dessin, de contrôle ainsi que de collisions et d'événements.

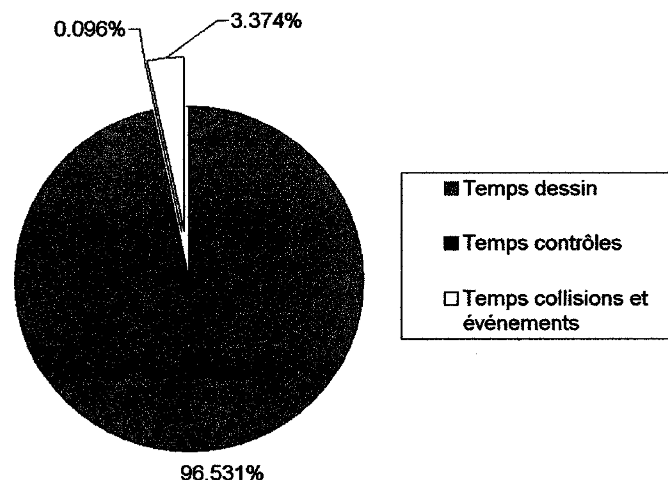


Figure 3.9 Répartition du temps d'exécution des différentes routines pour le scénario 6.

Les temps d'exécution sont très similaires aux temps présentés au scénario 3, avec un temps d'exécution pour l'algorithme de détection de

collisions et de rebondissement légèrement supérieur. La raison est simple, plus de détections complètes sont effectuées. Par contre, étant donné les nombreuses optimisations qui précèdent celles-ci, le nombre de facettes est déjà considérablement réduit ce qui ne provoque pas une élévation importante du temps d'exécution. Cette optimisation est efficace lorsque beaucoup de facettes sont retenues par les optimisations.

Étant données le grand nombre de détections complètes qui comportent plusieurs calculs, il est prévisible de voir le nombre d'erreurs augmentées dû à l'imprécision mathématique. Le nombre d'erreurs passe donc à 5 dans le scénario 6 pour la simulation effectuées, ce qui n'est pas énorme. Une augmentation significative du nombre d'erreurs est tout de même notable. Ces erreurs sont également reprises correctement par le recouvrement après erreur.

L'effet de l'optimisation sur l'élimination de facettes à l'aide de la distance est donc minimisé par l'efficacité des autres optimisations. Le logiciel Arka3D fonctionne très bien sans cette optimisation malgré un nombre d'erreurs plus élevées. Avec une légère diminution du temps d'exécution et un nombre d'erreurs inférieures, l'optimisation à donc sa place dans l'algorithme de détection de collisions.

3.2.8 Conclusion

Les six scénarios présentés ont permis d'enregistrer une panoplie d'informations qui permet d'évaluer l'efficacité de l'algorithme de détection de collisions du logiciel Arka3D.

Les trois premiers scénarios évaluent l'algorithme de détection de collisions complet dans différents environnements. Pour ce faire, des environnements simples, intermédiaires et complexes sont utilisés. Nous avons observé qu'avec toutes les optimisations utilisées, le logiciel Arka3D résout très bien la détection de collisions.

Dans les trois autres scénarios, des optimisations ont été retirées afin de vérifier l'efficacité de celles-ci. Un seul des trois scénarios, portant sur la division de l'espace (section 3.2.5) donne un résultat insatisfaisant prouvant la nécessité et l'efficacité de l'optimisation. Bien que les autres scénarios montrent que les autres optimisations ne sont pas nécessaires, celles-ci améliorent sensiblement le temps d'exécution et réduisent considérablement le nombre d'erreurs ce qui est une valeur ajoutée pour l'algorithme de détection de collisions.

Dans la section suivante portant sur une discussion sur ces scénarios, nous comparons sous plusieurs critères les différents résultats acquis dans cette section. Ces comparaisons permettront d'évaluer les différentes techniques d'optimisation utilisées dans le logiciel Arka3D.

3.3 Discussion

3.3.1 Introduction

Au cours des tests faits à l'aide des scénarios présentés à la section 3.2, un grande nombre d'informations a été enregistrées dans le but d'analyser les performances du logiciel Arka3D. L'objectif principal était d'analyser en profondeur l'efficacité de l'algorithme de détection de collisions et les optimisations qui le compose, surtout au niveau de la collision avec le tableau de cubes qui est la plus problématique dû au grand nombre de paires d'objets.

Nous comparerons ici les différents tests sur des critères comme le temps d'exécution moyen des différentes routines, les temps utilisés pour résoudre les différentes optimisations ainsi que les erreurs engendrées par les scénarios. Rappelons que chaque scénario est exécuté une seule fois ce qui donne suffisamment d'information pour tirer des conclusions pertinentes.

Rappelons que les scénarios 1, 2 et 3 sont des tests où l'ensemble des optimisations est disponible afin de comparer l'efficacité de l'algorithme de détection de collisions et de gestion d'événements sur des environnements différents.

Par contre, dans les scénarios 4, 5 et 6, certaines optimisations ont été retirées afin d'évaluer l'efficacité de celles-ci. Ces scénarios évoluent tous dans le même environnement que le scénario 3 afin de les comparer à celui-ci.

Ainsi, à l'aide de ces comparaisons, nous pourrons tirer des conclusions sur les performances de l'algorithme de détections de collisions et la gestion d'événements utilisées dans le logiciel Arka3D.

3.3.2 Comparaison entre les environnements

Dans la section 3.2 portant sur les scénarios de tests, nous avons souligné que les scénarios 1, 2 et 3 utilisent des environnements différents alors que les trois derniers scénarios utilisent le même environnement que le scénario 3.

Dans ces scénarios, nous avons fait varier la taille et la vitesse des balles ainsi que leur nombre afin de tester la stabilité de l'algorithme de détection de collisions. Mais se sont principalement les tableaux de cubes qui feront varier grandement les temps d'exécution puisqu'ils utilisent la détection de collisions la plus complexe. La figure 3.10 présente le nombre de cubes de chacun des

environnements ainsi que le nombre de facettes de départ et le nombre moyen de celles-ci enregistrés au cours du test.

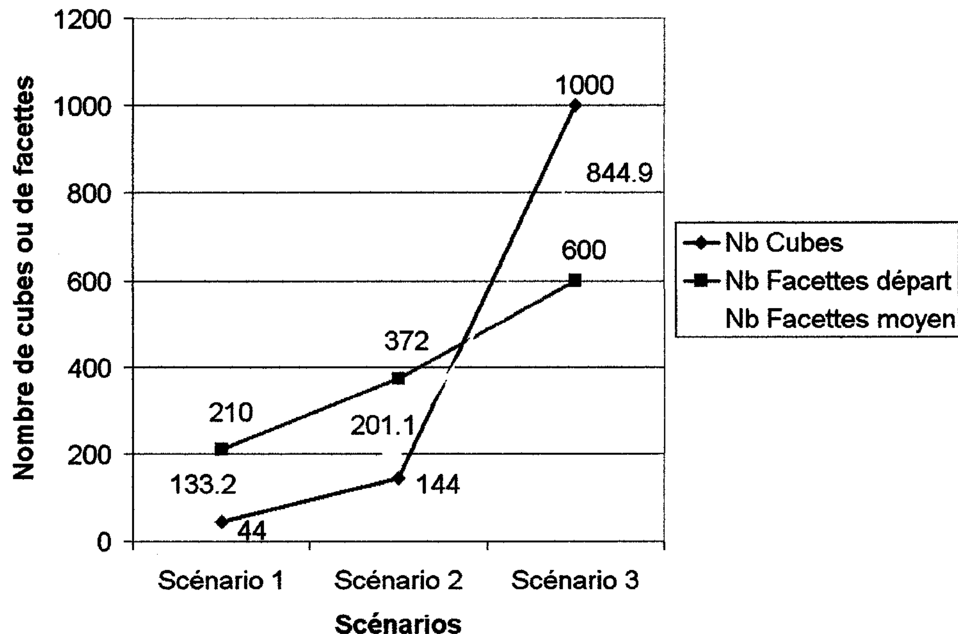


Figure 3.10 Comparaisons entre les environnements.

À l'aide des courbes présentées dans la figure 3.10, nous pouvons observer l'augmentation de la complexité des scénarios 1 à 3.

Les deux premiers scénarios sont des tableaux de cubes utilisés dans la version finale de Arka3D. Les tableaux de cubes utilisés dans la version finale du jeu ne possède pas plus de 200 cubes afin qu'ils soient réalisables en un temps raisonnable par le joueur. Nous avons donc testés un tableau simple, le scénario 1, et un tableau plus complexe, le scénario 2.

Le troisième scénario ne sera pas disponible dans la version finale du jeu Arka3D puisqu'il est presque impossible à compléter et prendrait au joueur un temps de jeu déraisonnable. Par contre, ce scénario est idéal pour un test d'efficacité et de stabilité dans les environnements très lourd.

Dans le graphique présenté à la figure 3.10, le nombre de cubes des scénarios augmente de manière exponentielle. Par contre, le nombre de facettes de départ et en moyenne augmente beaucoup moins rapidement. Dans le scénario 1, le nombre de facettes est supérieur au nombre de cubes, alors que dans le scénario 3, c'est l'inverse. La raison de cela est l'optimisation d'élimination de facettes disponible sur l'objet cubes (voir la section 2.3.6) qui élimine les facettes entre deux cubes adjacents puisqu'elles ne sont pas visibles. Donc, plus il y a de cubes, plus l'élimination de facettes est importante ce qui

rend les tableaux complexes aussi faciles à dessiner et à résoudre que les tableaux moyennement complexes. Voilà pourquoi le nombre de facettes augmente de manière linéaire alors que le nombre de cubes augmente de manière exponentielle.

Donc, nous démontrons ici l'efficacité de l'élimination de facettes surtout au niveau des tableaux possédant de nombreux cubes puisqu'un plus grand nombre de facettes est éliminées lorsqu'il y a un grand nombre de cubes. Cela permet de grandement simplifier les tableaux complexes, tant au niveau de l'affichage que de la détection de collisions.

3.3.3 Comparaison entre les routines

À la section 3.2, nous avons vu des graphiques à secteur qui présentaient les distributions du temps d'exécution de chacune des routines. Ces routines étant la construction de la scène et l'affichage de celle-ci (output), la gestion des contrôles (input), et la détection de collisions et la gestion d'événements (traitement).

La graphique présenté à la figure 3.11 présente un résumé du pourcentage de chacun des scénarios tirés des mêmes donnés qui ont servi à construire les graphiques à secteur de la section 3.2.

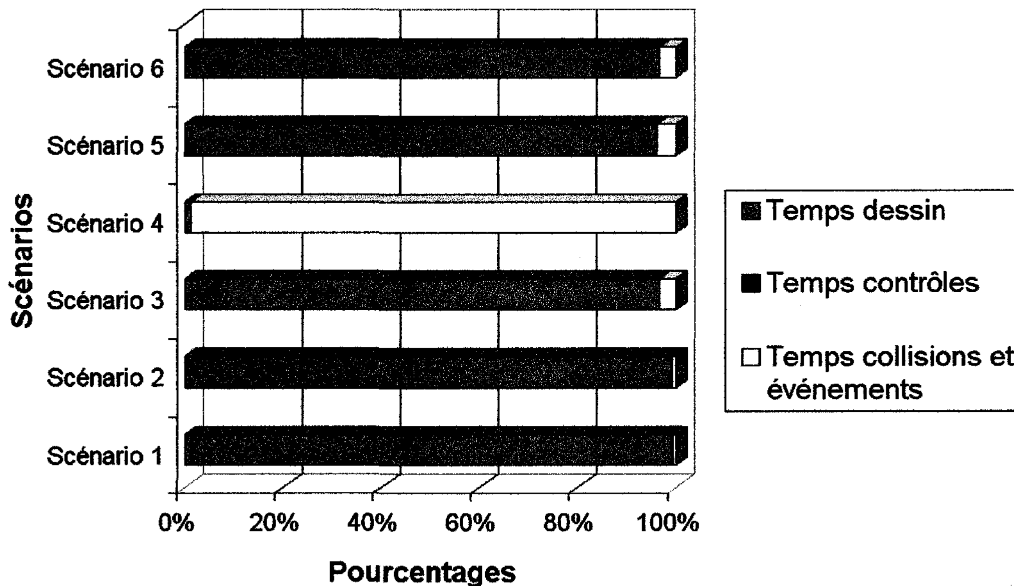


Figure 3.11 Résumé des répartitions du temps d'exécution des différentes routines.

Idéalement, un maximum de temps doit être alloué à l'affichage afin d'avoir des séquences d'images fluides. C'est le cas de la majorité des scénarios

qui offrent un temps supérieur à 96% pour le dessin. Seul le scénario 4, basé sur le même environnement que les scénarios 3, 5 et 6 est trop lent pour offrir un temps suffisant à l'affichage. Rappelons que le scénario 4 n'utilise pas la division de l'espace afin d'éliminer rapidement un grand nombre de paires d'objets. Cette optimisation est donc nécessaire et très performante dans le contexte de l'environnement d'Arka3D.

Le temps de contrôle, quant à lui, est toujours une fraction négligeable du temps d'exécution proportionnelle au temps d'affichage. Celui-ci ne cause alors aucun problème et DirectInput semble donc gérer les contrôles adéquatement et très rapidement.

Les scénarios 1 et 2, qui sont des tableaux de cubes utilisées dans la version finale du jeu offre 99% du temps de CPU à l'affichage, ce qui est amplement suffisant.

Les scénarios 3, 5 et 6 sont fait à l'aide d'un tableau de cubes utilisé exclusivement à des fins de tests et prennent malgré tout un temps très raisonnable à résoudre en lassant assez de temps pour un affichage fluide. Une légère augmentation du temps de résolution de la détection de collisions est tout de même perceptible dans le test 5 qui n'utilise pas la cohérence temporelle.

Finalement, la division de l'espace est une optimisation nécessaire afin de réduire considérablement le nombre d'itérations de la détection de collisions utilisée dans le logiciel Arka3D.

3.3.4 Répartition des temps pour la détection de collisions

Dans cette section, nous observerons comment sont distribués les temps d'exécution des différentes parties de la détection de collisions entre les balles et les cubes. Cette détection est sans aucun doute la plus lourde étant donné le nombre important de paires d'objets 3D à tester et le processus de gestion d'événements qui suit une collision.

Nous décomposerons donc le temps de détection de collisions entre les balles et les cubes en trois parties soient :

- **La présélection :** Qui permet d'éliminer des facettes avant d'appliquer la détection de collisions complète.
- **Le traitement :** Qui permet d'appliquer la détection de collisions complète.
- **Le rebondissement :** Qui calcul le rebondissement et applique les changements au tableau de cubes.

Le graphique de la figure 3.12 présente donc la répartition des temps d'exécution de chacune des parties de la détection de collisions entre les balles et les cubes.

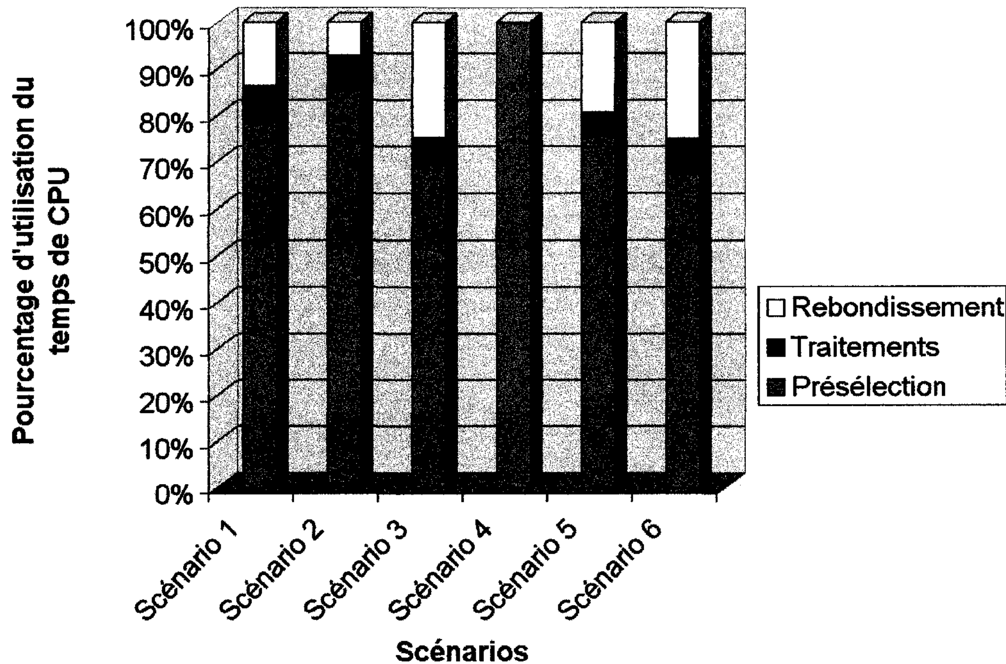


Figure 3.12 Répartition des temps pour la détection de collisions entre les balles et les cubes.

À première vue, nous pouvons observer dans la figure 3.12 que la majorité du temps de CPU est utilisé par la présélection de facettes. La présélection est nécessaire à toutes les itérations et les distances des facettes avec la balle doivent être calculées et triées, ce qui explique ce temps élevé. Dans le scénario 4, où un très grand nombre de facettes sont triées, la présélection prend un grande majorité du temps ce qui explique le temps désastreux de ce scénario.

Le temps de traitement est similaire pour tous les scénarios à l'exception du scénario 4 où il est complètement écrasé par la présélection. Il est supérieur dans le scénario 6 étant donné qu'il est fait sur un plus grand nombre de facettes comme nous le verrons à la section 3.3.6. Par contre le temps de présélection de celui-ci est réduit dû au retrait d'une optimisation (voir la section 3.2.7).

Le temps de rebondissement est également considérable. Bien qu'une collision se produise en moyenne 3 fois sur 100 000 tests de collision, celle-ci prend environ 10 à 30% du temps total de l'algorithme de détection de collisions et de gestion d'événements entre les cubes et les balles. Il y a deux raisons à cela, d'abord, le calcul du rebondissement est inévitablement lourd en utilisant plusieurs calculs vectoriels (voir la section 1.5.4), et ensuite, le changement sur

le tableau de cubes entraîne une recompilation complète des facettes à partir des cubes restants ce qui est également très lourd. Dans les scénario 1 et 2, il y a moins de cubes et moins de balles donc les collisions sont plus rares (2 sur 100 000) ce qui diminue le temps requis pour le calcul des rebondissements. Dans les autres scénarios, le grand nombre de cubes et de balles occasionne plus de collisions (4 sur 100 000) ce qui augmente considérablement le temps de traitement de cette partie.

Donc, la présélection, qui est essentielle pour éliminer un grand nombre de facettes prend la majeure partie du temps d'exécution de la détection de collisions entre les cubes et les balles. Cette étape est essentielle et permet au traitement de prendre beaucoup moins de temps. La gestion d'événements, le rebondissement en l'occurrence, prend également beaucoup de temps mais ne se produit que très peu souvent. Les temps d'exécution sont donc raisonnablement répartis dans tous les scénarios sauf le 4, qui néglige une optimisation très importante.

3.3.5 Comparaison des facettes présélectionnées et traitées.

Dans le but d'évaluer la performance de l'algorithme de détection de collisions entre les balles et les cubes dans des environnements de complexité variable et d'évaluer l'efficacité des optimisations, nous avons regroupé les nombres moyens de facettes retenues durant les différents scénarios.

Rappelons que la détection de collisions entre les balles et les cubes est de loin la plus complexe et, bien sur, la plus optimisée (voir section 2.4.5).

Dans le premier graphique, présenté à la figure 3.13, les nombres moyens de facettes présélectionnées de chacun des scénarios sont présentés.

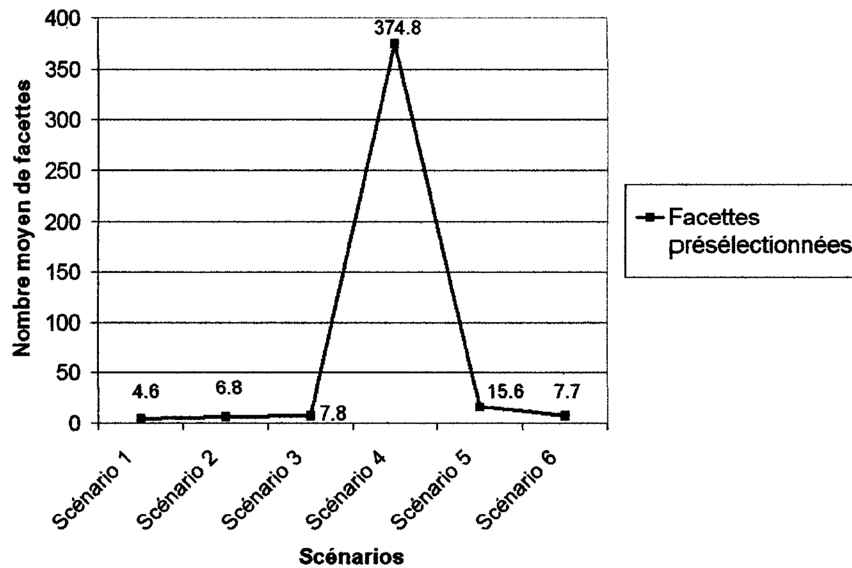


Figure 3.13 Nombres moyens de facettes présélectionnées.

Les facettes présélectionnées sont les facettes qui sont conservées suite aux optimisations de division de l'espace et de cohérence temporelle.

Dans ce graphique (Figure 3.13), nous pouvons observer, pour les trois premiers scénarios, que le nombre de facettes présélectionnées est augmenté légèrement avec le nombre de facettes présentes dans le tableau de cubes (voir figure 3.10). Cette augmentation semble être logarithmique. Cela montre que la division de l'espace est particulièrement efficace dans les environnements lourds.

Pour le scénario 4, où nous avons délibérément retiré l'optimisation de division de l'espace, le nombre de facettes présélectionnées est largement supérieur aux autres scénarios. C'est la raison qui explique les mauvaises performances présentées à la section 3.2.5.

Dans le scénario 5, l'optimisation sur la cohérence temporelle a été retirée. Cela nous permet de confirmer que cette optimisation élimine en moyenne la moitié des facettes si elle est comparée avec le scénario 3.

Dans le scénario 6, aucune optimisation sur la présélection n'a été retirée. Le résultat est donc similaire au scénario 3. Par contre, les résultats sont légèrement différents, ce qui montre que les résultats sont toujours différents d'une simulation à l'autre.

Le second graphique, présenté à la figure 3.14, présente le nombre de facettes traitées par l'algorithme de détection de collisions complet.

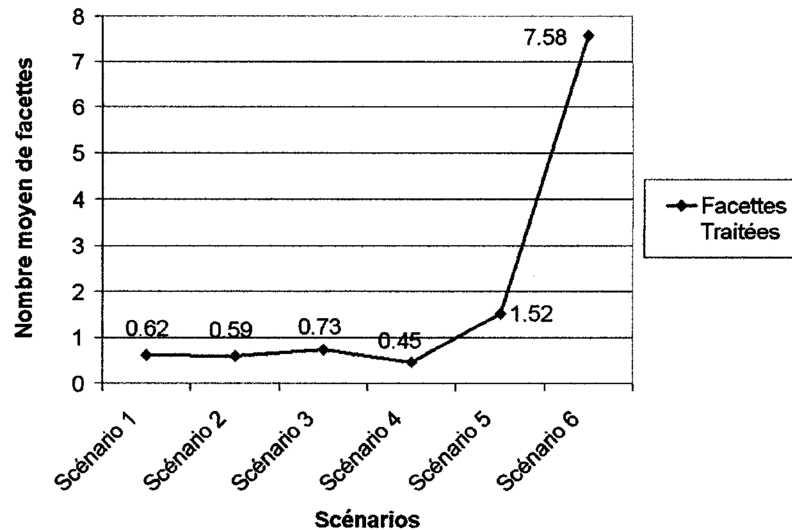


Figure 3.14 Nombres moyens de facettes traitées pour chaque scénario.

À la figure 3.14, nous pouvons observer que le nombre de facettes traitées à chaque itération est sensiblement le même pour les quatre premiers scénarios. Le scénario 2 enregistre un nombre de facettes traité légèrement inférieur au scénario 1. Cela s'explique par le fait que le tableau de cubes utilisé au scénario 2 est plus difficile à compléter que celui du niveau 1. Cela provoque plus d'itérations où aucune facette n'est retenue.

Dans le cas du scénario 3, le nombre de facettes traitées est légèrement supérieur étant donné que les collisions sont plus fréquentes et donc plus de facettes sont retenues pour permettre leurs détections.

Le scénario 4 présente un résultat inférieur aux trois premiers tests. Cela s'explique par un grand nombre de tests inutiles effectués, dû au retrait de l'optimisation de la division de l'espace.

Dans le scénario 5, nous observons une nette augmentation du nombre de facettes traitées par rapport au scénario 3 qui est exécuté dans le même environnement. La cohérence temporelle permet donc d'éliminer près de la moitié des facettes soumises à des tests de collisions complets.

Finalement, le scénario 6 n'arrête pas de traiter les facettes lorsque la distance maximum de collision possible entre une facette et une balle est atteinte. Cela entraîne le test complet de toutes les facettes présélectionnées. Le nombre de facettes traitées est donc similaire au nombre de facettes présélectionnées vu à la figure 3.13. Cela démontre l'importance de cette optimisation. La légère différence est causée par l'arrêt du traitement lorsqu'une collision est détectée.

L'algorithme de détection de collisions utilisé dans le logiciel Arka3D pour gérer les collisions entre les cubes et les balles permet donc d'éliminer un grand nombre de facettes avant le traitement. La présélection est essentielle afin de conserver seulement les facettes à risque d'être impliquées dans une collision. Les optimisations faites sont donc efficaces et réduisent considérablement le nombre de facettes présélectionnées ou traitées si nous les comparons aux scénarios qui les excluent. Nous pouvons donc conclure à l'efficacité de ces techniques d'élimination de facettes.

3.3.6 Comparaison des erreurs

Malgré un processus de recouvrement après erreurs efficace, celles-ci doivent être évitées le plus possible puisqu'elles occasionnent des rebondissements moins réalistes. C'est pourquoi cette section compare les erreurs générées par les différents scénarios.

Il y a deux types d'erreurs qui peuvent se produire. La première étant qu'une balle pénètre à l'intérieur d'un cube faisant ainsi échouer les tests de collisions. La seconde se produit lorsqu'un rebondissement récursif se produit sur le coin d'un cube à cause d'un rebondissement dans une mauvaise direction (Voir la section 2.6).

La figure 3.15 présente donc un graphique qui compare les erreurs générées par les différents scénarios pour un enregistrement.

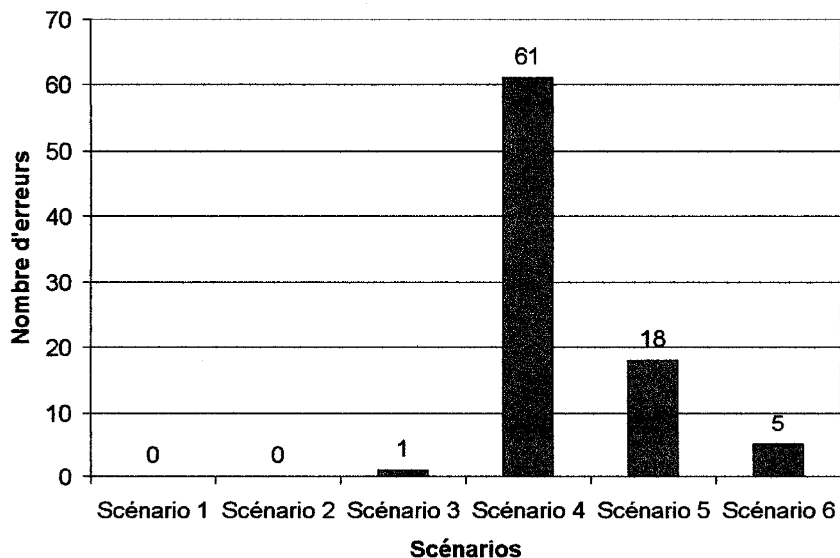


Figure 3.15 Nombres d'erreurs générées par les scénarios.

Le graphique présenté à la figure 3.15 montre que les trois premiers scénarios comportent pratiquement aucune erreur, soit seulement une pour le

scénario 3 qui est considéré comme très complexe. Cela montre que les optimisations jouent un rôle important pour la minimisation du nombre d'erreurs.

Dans le scénario 4, le nombre d'erreurs imposant s'explique par la lenteur du traitement qui cause de grands déplacements des objets 3D entre les itérations d'affichage de la scène.

Le scénario 5 démontre que l'élimination de facettes grâce à la cohérence temporelle, qui élimine la moitié des facettes n'étant pas sujettes à des collisions, permet ainsi d'éliminer un grand nombre de mauvais rebondissements.

Finalement le scénario 6, qui applique une détection de collisions complète sur un grand nombre de facettes génère également plus d'erreurs. Ceci démontre l'importance de traiter un minimum de facettes dans la détection de collisions complète.

Rappelons que toutes ces erreurs ont été recouvrées et que le logiciel Arka3D a pu poursuivre sont traitement normalement.

Beaucoup d'erreurs sont donc évitées grâce aux optimisations qui permettent d'éliminer un grand nombre de facettes non sujettes à des collisions. Alors, en plus de réduire considérablement le temps d'exécution de l'algorithme de détection de collisions, les optimisations améliorent la stabilité et le réalisme de la gestion d'événements.

3.3.7 Conclusion

Les différents scénarios ont donc été comparés afin de valider l'efficacité de l'algorithme de détection de collisions et des optimisations qui les composent.

Dans un premier temps, nous avons comparé les trois environnements qui ont servis à réaliser les scénarios, soient un simple, un complexe et un intermédiaire. Le scénario complexe a été utilisé pour les quatre derniers scénarios puisqu'il est le plus problématique. En comparant le nombre de facettes créées par les tableaux de cubes, nous avons montré l'efficacité de l'optimisation portant sur l'élimination des facettes cachées.

Ensuite, nous avons comparé les différents temps d'exécution des routines d'affichage, de gestion de contrôles ainsi que de la détection de collisions et la gestion d'événements. Tous les scénarios offrent un temps raisonnable pour l'affichage sauf le scénario 4, qui montre l'importance de la division de l'espace dynamique utilisée dans Arka3D qui évite une explosion du nombre de paires d'objets.

Nous avons ensuite comparé les différentes phases de la détection de collisions entre les cubes et les balles. Le temps d'exécution de la présélection prend la majorité du temps mais elle a un rôle très important pour l'élimination d'une grande majorité de calculs. La gestion d'événements prend également un temps significatif mais se produit plus rarement.

De plus, au niveau de la détection de collisions entre les balles et les cubes, qui est de loin la plus complexe, nous avons présenté le nombre de facettes utilisées lors de la présélection et le traitement complet. Nous avons alors vu l'importance des optimisations de divisions de d'espace, de cohérence temporelle et de vérification de distance.

Finalement, le nombre d'erreurs qui se produisent en cours d'exécution a été comptabilisé. Cette comparaison a montré une seconde utilité aux optimisations qui limite grandement le nombre d'erreurs produites.

L'efficacité de l'algorithme de détection de collisions et la performance des optimisations qui le composent ont donc été prouvées.

3.4 Conclusion

Dans ce chapitre, différents résultats sur l'efficacité de la détection de collisions et de l'affichage ont été présentés sous six différents scénarios.

Les trois premiers scénarios, dont deux sont utilisés dans la version finale du logiciel, sont réalisés avec l'ensemble des optimisations et satisfont les critères de temps pour l'affichage.

Les trois autres scénarios ont permis de comparer l'efficacité des optimisations en les retirant délibérément. Cela permet de déduire que l'optimisation basée sur la division de l'espace, qui a été retirée dans le scénario 4, est essentielle au bon fonctionnement du logiciel. Les autres optimisations permettent de réduire légèrement le temps d'exécution de l'algorithme de détection de collisions mais permettent aussi de réduire significativement le nombre d'erreurs produites.

Les environnements difficiles et le retrait d'optimisations soumis au logiciel Arka3D ont éprouvé considérablement sa stabilité. Malgré un temps désastreux pour le scénario 4, celui-ci n'a jamais arrêté son exécution malgré un grand nombre d'erreurs. Le recouvrement après erreurs est donc très efficace.

Les algorithmes de détection de collisions et de gestion d'événements passent donc tous les tests de manières stables et résolvent très bien les environnements auxquels ils ont été soumis lorsque toutes les optimisations sont disponibles. Les algorithmes satisfont donc toutes les attentes.

4 Conclusion

Les techniques de gestion d'interactions et d'événements dans un environnement 3D présentées dans ce document sont donc efficaces. Les algorithmes de détection de collisions et de gestion d'événements du logiciel Arka3D satisfont les attentes à tous les niveaux soient de la stabilité, de la précision et surtout de la rapidité.

L'algorithme de détection de collisions du logiciel présenté se base surtout sur des concepts établis examinés lors de la revue de littérature présentée au chapitre 1 mais utilise aussi des concepts innovateurs et adaptés à la problématique présenté au chapitre 2.

L'algorithme de détection de collisions du logiciel Arka3D utilise des techniques de géométrie constructive et de cohérence temporelle afin d'éliminer un grande nombre d'erreurs qui se produisent avec les algorithmes existants présentés à la fin du chapitre 1.

La principale problématique de la simulation d'environnement est donc sans aucun doute la détection de collisions. Bien que plusieurs travaux ont été réalisés dans ce domaine, ceux-ci se limitent surtout au modèle polygonaux à l'exception de la géométrie constructive (CSG). La géométrie constructive est par contre beaucoup moins rapide et moins stable que les algorithmes basés sur les modèles polygonaux. Les surfaces paramétrées sont également une avenue intéressante pour le développement futur d'algorithmes plus précis et plus rapides.

Avec l'augmentation continuelle de la puissance de calcul des ordinateurs et la venue récente de modules de physique intégrés (comme pour la nouvelle console Microsoft XBOX360), les ressources disponibles pour la simulation d'une environnement 3D sont de plus en plus importantes. Cela aura pour effet d'augmenter la précision et la rapidité des algorithmes de détection de collisions et de gestion d'événements.

La gestion d'événements et d'interactions dans un environnement 3D est donc encore à un stade embryonnaire qui mènera a des simulations de plus en plus précises qui permettront des améliorations dans plusieurs domaines d'application notamment ceux de l'animation et du jeu vidéo.

Annexes

Annexe A : Résumé de Direct X

Le DirectX SDK (software developpement kit) fournit par Microsoft est la trousse de développement pour DirectX 9.0 en environnement Visual Studio .NET. Bien que la trousse de programmation fonctionne aussi avec Visual Basic et les technologies .NET, il est fortement conseillé d'utiliser Visual C++ (sans MFC) afin de faire une application optimum. De plus, la trousse ne semble pas fonctionner avec .NET, l'application compile mais refuse de s'exécuter...

Les travaux et les explications seront donc fait en Visual C++.

Qu'est-ce que DirectX?

DirectX est un ensemble de librairies de bas niveau qui sert surtout pour le développement de jeux. Il a été développé pour les ordinateurs utilisant Windows. DirectX a récemment fait une percée dans le monde des consoles avec la X-Box et pourrait bientôt apparaître sur Mac sous la forme de MacDX.

DirectX est principalement un engin de graphique 3D qui utilise les plus récentes technologies d'affichage. Son efficacité et comparable à OpenGL. Selon Microsoft, DirectX serait plus simple que OpenGL; mais a mon avis, c'est différent. Une chose qui est sûr, c'est que DirectX est plus complet pour le développement de jeux. Il supporte, en plus du 2D et du 3D, le son et la musique, les périphériques d'entrée comme les manettes de jeu, l'affichage de média comme les films et il permet même la gestion du réseau.

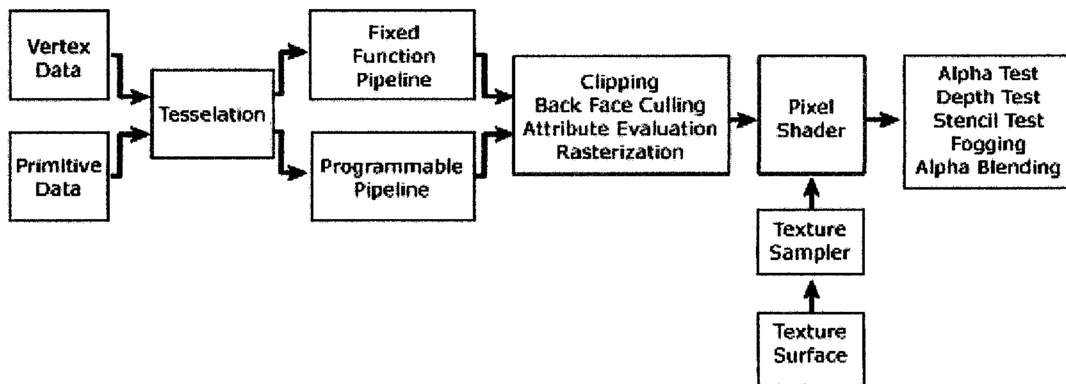
DirectX est divisé en plusieurs librairies (APIs) qui sont séparées par sujet :

- **DirectX Graphics :** gère l'affichage de scènes 2D et 3D.
- **DirectInput :** gère les périphériques d'entrée.
- **DirectPlay :** gère l'accès au réseau.
- **DirectMusic :** permet de créer de la music tel un synthétiseur.
- **DirectSound** : gère les sons à de bas niveau dans un environnement 3D.
- **DirectShow :** gère l'affichage de films.

Ces librairies sont toutes reliées entre elles afin de pouvoir créer facilement un environnement de jeu très flexible. DirectX peut bien sûr être utilisé pour n'importe quel type d'applications multimédias.

DirectX Graphics

DirectX Graphics sert à créer une scène en 2D (DirectDraw) ou en 3D (Direct3D) optimisée et très rapide en temps d'exécution. Il est principalement utilisé pour le 3D avec **Direct3D**. Plusieurs fonctionnalités sont gérées automatiquement comme le « clipping » et certaines fonctions d'amélioration esthétiques comme le « shading ».



L'engin 3D sert a créé un environnement 3D et possède un panoplie de fonctions et de propriétés modifiables. Il est définie comme suit :

```
LPDIRECT3DDEVICE9 d3dDevice = NULL;
App3D->CreateDevice( D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, hWnd,
                     D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                     &d3dparameter, &d3dDevice );
```

Direct3D support également les modèles d'objets 3D sous une forme qui lui est propre : les fichiers X. Ces fichiers contiennent à prime abord des modèles 3D qui peuvent être chargées rapidement par Direct3D. De plus, il est possible d'ajouter dans ces fichiers des références à des textures et même du mouvement. Ces fichiers X seront définis en « Meshs » qui pourront être affichées par l'engin 3D.

De plus, l'engin supporte des fichiers FX qui permettent de créer des effets spéciaux préprogrammés et réutilisables.

L'engin 3D est très efficace dans l'affichage de primitives, qui sont des listes de points dessinées en temps réels dans la scène. Cela évite de faire des transformations lourdes dans la scène.

Finalement, Direct3d permet une multitude d'effets spéciaux comme la transparence, les effets de lumières, la brume et le découpage de modèles.

DirectInput

DirectInput lie directement l'engin graphique avec des périphériques d'entrée. Les périphériques sont traitées rapidement et beaucoup plus efficacement que les méthodes traditionnelles. En étant inclus dans l'engin 3D, ils minimisent le délai de réponse ce qui donne l'impression d'un meilleur contrôle de l'environnement.

Il y a certainement une lacune dans les entrées de périphériques standard dans Windows. C'est pourquoi que, même avec OpenGl, il est fortement suggéré d'utiliser DirectInput, qui peut aussi être utilisé indépendamment.

Cette librairie gère également plusieurs périphériques à la fois. L'objet CInputDeviceManager est très utile en gérant toutes les périphériques possibles en même temps:

- Clavier.
- Souris.
- Manette de jeux (avec n'importe quel nombre de boutons).
- Joystick pour simulateur de vol.
- Volant et pédale pour simulateur de course.
- Écran tactile.
- Gérer indépendamment certaine fonctionnalité des périphériques.

De plus, DirectInput vient avec un système des configurations dynamiques qui permet à plusieurs joueurs d'enregistrer ses contrôles favoris.

Il est aussi possible de gérer les périphériques indépendamment en n'utilisant qu'un seul périphérique à la fois. Des structures adaptées sont alors instaurées afin d'utiliser les périphériques plus efficacement.

DirectInput permet également les sorties en générant certains événements.

Cette librairie est donc nécessaire pour un environnement contrôlé par périphériques.

DirectSound et DirectMusic

DirectMusic est un genre de gros synthétiseur de bas niveau qui gère plusieurs sons. Il utilise DirectSound et possède plus d'effets que celui-ci. DirectSound permet aussi d'enregistrer des sons en gérant les périphériques d'entrées.

DirectSound permet non seulement de jouer plusieurs fichiers de types musicaux, il permet aussi de modifier ceux-ci. Il peut bien sûr modifier la fréquence et la vitesse du son. Mais ce qui est le plus intéressant, c'est qu'il peut également le soumettre à plusieurs effets :

- Changer l'intensité sonore de chaque haut-parleur.
- Simuler un objet à une certaine distance et de coté.
- Simuler l'effet doppler.
- Simuler l'effet de roulement.
- Faire un écho.
- Faire plusieurs effets comme la distorsion, choral, gargariser, eau, etc…

Il permet également un accès direct au périphérique de son afin de permettre plus de fonctionnalités.

DirectMusic permet de créer un ensemble de sons harmonieux à partir de DirectSound. Il permet de faire des transitions entre deux sons et peut même ajouter des sons en suivant le tempo de la musique. De plus, il est possible d'utiliser un groupe de sons afin de créer la musique en temps réel.

Encore un fois, DirectMusic utilise son propre type de fichier midi (sgt) et on peut également enregistrer des effets.

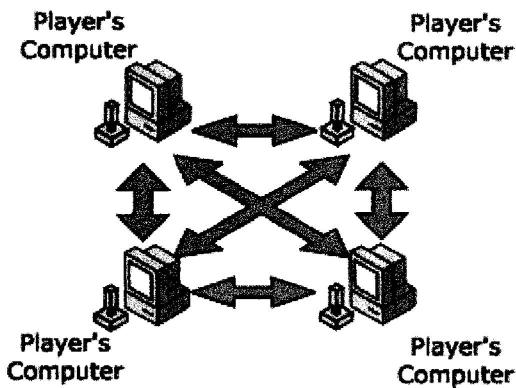
Dans un environnement 3D, DirectSound et DirectMusic rende le son plus réaliste.

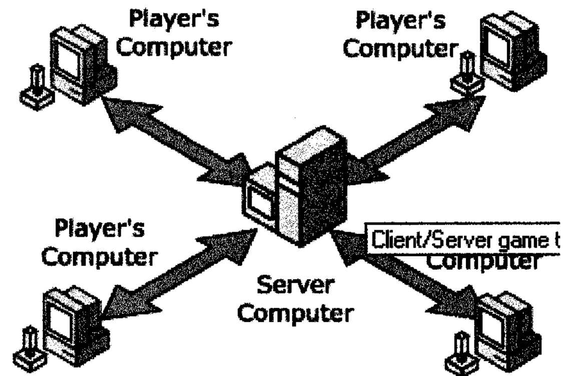
DirectPlay

Il ne faut pas confondre avec DirectPlay avec le DirectPlay de la version 7 de DirectX qui est devenu DirectInput. DirectPlay gère la connexion réseau en TCP/IP afin de permettre le multi-joueurs.

DirectPlay utilise la philosophie client/serveur et aussi le personne-à personne (« peer-to-peer »). Plus il y a de joueurs, plus la topologie client/serveur est conseillée.

Peer to peer **Client/serveur**





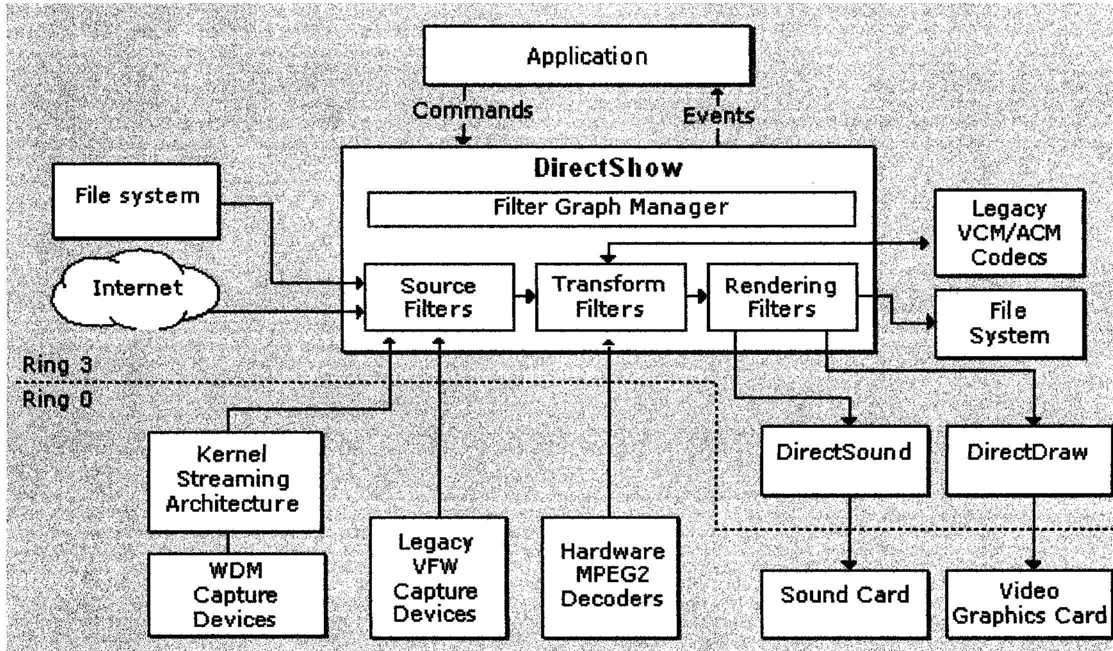
De plus, DirectPlay inclut des architectures largement utilisées comme le clavardage (« chat ») et la communication par micro/écouteur.

DirectPlay simplifie beaucoup la gestion multi-joueur tout en l'optimisant pour les jeux. En fonctionnant de pair avec engin DirectX, on s'assure d'un temps de réponse plus rapide.

DirectShow

DirectShow permet de faire jouer des films a partir d'un fichier ou a partir d'un flux de donnés. Il support une panoplie de types de fichier, y compris les fichiers Quicktime (ce qui est rare de la part de Microsoft).

De plus DirectShow permet de faire plusieurs types de filtres. Voici sontarchitecture :



La librairie permet également l'enregistrement de flux vidéo et un système d'édition vidéo. De plus, on y a ajouté un système qui fonctionne un peu comme la télévision (Microsoft TV Technologies).

Étant lié avec les autres modules de DirectX, il est possible, par exemple, de d'appliquer un film comme texture sur un modèle 3D.

DirectShow est donc plus qu'un lecteur de film, il pourrait servir à créer un logiciel de montage vidéo des plus complet.

Conclusion

Bien que DirectX à été principalement inventé pour la création de jeux vidéo, son mandat a été largement étendu. On n'a qu'à penser au système avancer de communication par réseau ou à la versatilité du traitement de vidéo.

DirectX reste tout de même l'outil par excellence pour la création de jeux vidéo en environnement Windows. Il gère à la fois l'affichage, les périphériques d'entrée, les sons et le mode réseau. Sans compter la multitude de fonctions intégrées comme la gestion de matrices et de vecteurs.

Par contre, on dénote plusieurs bugs dans toutes les versions de DirectX. DirectInput est souvent concerné. Les mises à jours fréquentes de Microsoft ajoutent souvent plus de bug quelles en corrigent. Il faut travailler avec ça.

Mais le meilleur choix dans le développement de jeux en environnement Windows est sans doute DirectX : déjà la majorité des jeux l'utilisent à différent niveau.

C'est avec un peu d'hésitation que j'ai choisi DirectX pour le développement l'environnement 3D qui fait le sujet de ma thèse. Maintenant que Direct3d et Directinput sont correctement configurés et que les principales fonctions d'affichage sont terminées, les principales difficultés ont été résolues en ce qui conserne DirectX.

Annexe B : Résumé des versions de Arka3D.

Pas d'image

Arka3d version 22
(11-11-2005)
Dix nouveaux tableaux de cubes ont été construits. Les records sont maintenant intégrés. Le jeu est maintenant complet et utilisable par un joueur.

Pas d'image

Arka3d version 21
(7-10-2005)
Les classes d'objets sont maintenant complètement intégrées aux jeux. La routine de jeux permet maintenant la perte de balles avec un nombre de vies.

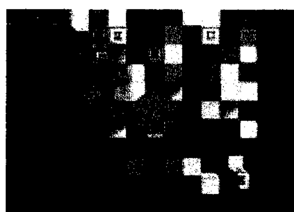
Pas d'image

Arka3d version 20
(30-09-2005)
Apres une longue revue de littérature, voici une nouvelle version où le code à été clarifié. Les fonctions sont en partie intégrées à des classes d'objets.

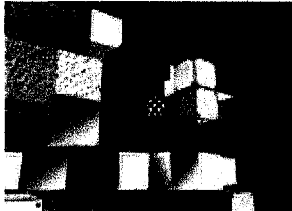
Pas d'image

Arka3D version 19
(4-01-2005)
Un menu est maintenant disponible. La routine de jeux a également été intégrée. Il n'est pas possible de mourir encore!

Arka3d version 18
(21-12-2004)
Des débris apparaissent maintenant lorsque des cubes explosent, ceux-ci entre en collision avec les autres objets de la scène. L'algorithme de collision entre balles et cubes a été encore modifié afin de tenir compte de certains cas particuliers.
Le projet tire à sa fin!

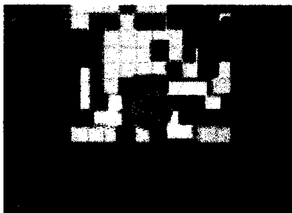


Arka3d version 17 et ArkaBuilder version 5
(26-11-2004)
Des cubes explosifs ainsi que des cubes à coups multiples sont maintenant disponibles. Les collisions semblent encore bien fonctionner! Pour les tests, un fichier output.txt est créé durant l'exécution, à garder en cas de bug! Les sons ont été changés...

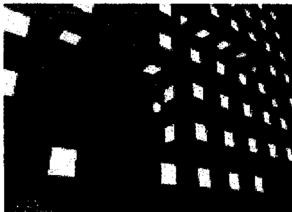


Arka3d version 16 et ArkaBuilder version 4
(19-11-2004)
L'algorithme de collisions avec les cubes a été revu, il semble fonctionner très bien. Il passe le test du beigne... (consiste à coincer les balles à l'intérieur d'une forme : un anneau (beigne) a été utilisé). Le " Sphère Map " a été ajusté. Également beaucoup d'optimisation qui améliore le temps d'exécution d'environ 20%. Des cubes indestructibles peuvent maintenant être placés.

Arka3d version 15
(12-11-2004)
Des textures sur les balles et des explosions ont été ajoutées. Le " Sphère Map " est à revoir. Il y a encore des corrections à apporter au niveau des collisions avec les cubes.

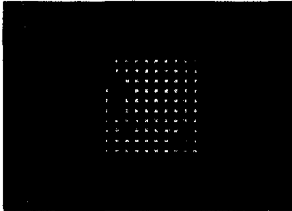


Arka3d version 14 et ArkaBuilder version 3
(05-11-2004)
La structure des objets cubes a été remodelée afin de supporter jusqu'à 32 textures. De plus le builder peut également afficher les niveaux par tranche, se qui sera utile pour vérifier les cubes du centre qui sont masqués par d'autres. Je suis ouvert à la suggestion de textures pour les cubes et les murs.



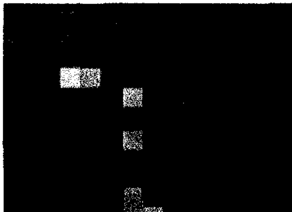
Arka3d version 13
(25-10-2004)
Les balles peuvent maintenant rebondir si elles ont des vitesses différentes. Un bug se produisait lorsqu'une balle très rapide était prise dans un coin par une balle plus lente. L'algorithme accélère certaines balles pour un moment. De plus, les collisions sur les coins sont maintenant fonctionnelles, ce fut plus simple que prévu...

Arka3d version 12
(15-10-2004)
Les balles entrent maintenant en collision entre elles. Ces collisions doivent tenir compte des autres types de collisions ce qui rend la tâche complexe. Cette version est la première soumise au beta testing! Voir la section téléchargement pour m'aider dans le développement de Arka3d!

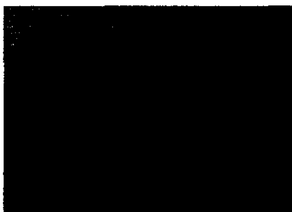


Arka3d version 11
(8-10-2004)
Je passe à Visual Studio 2003 et à DirectX SDK summer 2004 avec quelques difficultés... J'ai encore corrigé mon algorithme de détection de collisions sur les cubes. Je savais avant de commencer que se serait une partie difficile... Les murs et les cubes supportent maintenant les textures.

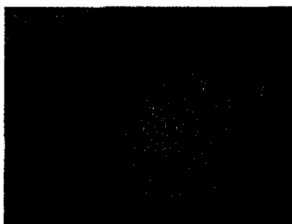
Arka3d version 10
(1-10-2004)
J'ai enfin corrigé tous les "warnings" qui s'accumulaient depuis le début. Il y a définitivement encore des bugs dans mon algorithme de collisions. J'ai également fait un nouveau système de points de vue qui devrait être utilisé dans la version finale.



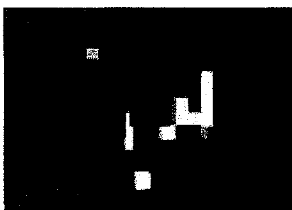
Arka3d version 9
(27-09-2004)
L'algorithme de détection de collisions avec les cubes a subi plusieurs corrections. Il fait maintenant disparaître les cubes touchés.



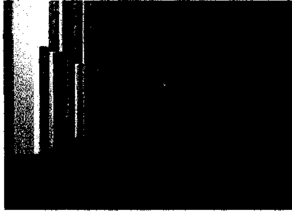
Arka3d version 8
(24-09-2004)
Première version de détection de collisions avec les cubes. C'est assez complexe et il y a encore beaucoup de bugs. Faut pas se décourager.



Arka3d version 7
(27-08-2004)
Les cubes sont ajustés dans le jeu également. La souris est maintenant fonctionnelle.



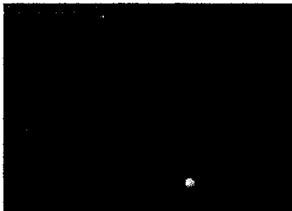
Arkabuilder version 2
(27-08-2004)
Les cubes sont plus gros pour en avoir seulement 1000 maximums. Les contrôles sont améliorés.



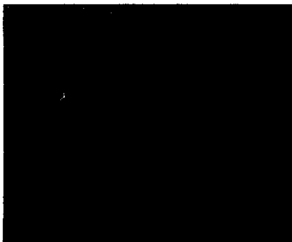
Arka3d version 6
(17-08-2004)
J'ai réussi à intégrer les 8000 cubes à l'environnement du jeu. C'est définitivement trop. Ici on voit 1000 cubes.

Arkabuilder version 1
(13-08-2004)
Maintenant, il me faut un constructeur de tableaux de cubes. Les tableaux peuvent contenir 8000 cubes (c'est énorme!). Les objets cubes sont mis en relation avec les objets facettes afin de minimiser l'affichage.

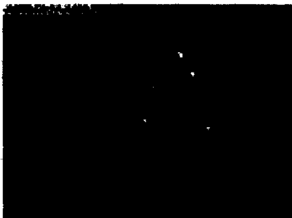
Arka3d version 5
(06-08-2004)
Le pad peut maintenant faire des rebondissements. La face avant peut dévier les balles et l'arrière les rebondies simplement. L'angle des balles est limité.



Arka3d version 4
(05-08-2004)
La mesh pad créée avec MilkShape3d est maintenant affichable. Elle sera translucide de proche et opaque de loin. Il est possible de la déplacer au clavier avec les flèches. Un petit test de texture...



Arka3d version 3
(26-07-2004)
Les murs qui sont à l'extérieur sont rendus translucides! La normale des murs a été renversée afin que l'effet de lumière des balles soit visible. Certains tests de chargement de meshs en fichier.x ont également été fait. Les sons sont également introduits.



Arka3d Version 2
(14-07-2004)
Construction des murs et des objets balles. Les murs sont construits de plusieurs triangles (64X64) afin d'avoir un effet de lumières avec les balles plus réaliste. Le mur est recopié 18 fois afin de former l'environnement de jeu.

Arka3d version 1
(7-07-2004)
Premier test d'affichage avec les primitives et l'éclairage, 20000 facettes sont affichées soit le double du maximum prévu utilisé dans le jeu. De plus, des meshs en forme de balle pivotent autour de ces plaquettes en produisant un éclairage. Il est également possible de se déplacer à l'aide du clavier afin d'explorer l'environnement.

Références

[1] Laurend Testud, Le programmeur : DirectX 9, Programmation des jeux 3D, CampusPress, 2003, pages 30 – 34.

[2] Chazelle B. et Palios L. Decomposition algorithms in geometry. In algebraic geometry and its applications. C. Bajaj Ed. volume 5. 1994. page 419-447.

[3]Bajaj C, Dey T. Convex decomposition of polyhedra and robustness. SIAM Journal on Computing 1992,339-364.

[4] Chazelle B. Convex partitions of polyhedra: a lower bound and a worst-case optimal algorithm. SIAM J. Comput. 1984,488-507.

[5] Bern M. Triangulations. In: Goodman JE, O'Rourke J, editors. Handbook of discrete and computational geometry. Boca Raton, FL, CRC Press, 1997.

[6] Chazelle B, Palios L. Decomposing the boundary of a nonconvex polytope. In Proceedings of the Third Scandinavian Workshop on Algorithm Theory, 1992. p. 364-375.

[7] C.M. Homann. Geometric and Solid Modeling. Morgan Kaufmann, San Mateo, Californie,2001.http://www.cs.purdue.edu/homes/cmh/distribution/books/geo.html

[8] Jules Bloomenthal et Brian Wyvill. Interactive techniques for implicit modeling. In Rich Riesenfeld and Carlo Sequin, editors, Computer Graphics (1990 Symposium on Interactive 3D Graphics), volume 24,March 1990.

[9] About Nonuniform Rational B-Splines – NURBS, a summary by Markus Altmann, http://www.cs.wpi.edu/~matt/courses/cs563/talks/nurbs.html

[10] G. Farin. Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide. Academic Press Inc., 1993.

[11] David Anson, BezierPatchApplet, http://www.nbb.cornell.edu/neurobio/land/OldStudentProjects/cs490-96to97/anson/BezierPatchApplet/

[12] M.C. Lin. Efficient Collision Detection for Animation and Robotics. PhD thesis, Department of Electrical Engineering and Computer Science, Université de la Californie, Berkeley, Décembre 1993.

[13] S. Cameron et R. K. Culley. Determining the minimum translational distance between two convex polyhedra. Proceedings of International Conference on Robotics and Automation, pages 591-596, 1986.

[14] M.C. Lin and Dinesh Manocha. Interference detection between curved objects for computer animation. In Models and Techniques in Computer Animation, pages 43-57. Springer-Verlag, 1993.

[15] Doug L. James et Dinesh K. Pai. BD-Tree: Output-Sensitive Collision Detection for Reduced Deformable Models, http://graphics.cs.cmu.edu/projects/bdtree/

[16] J. Stewart. Analyse, concepts et contextes. Volume 2. Fonctions de plusieurs variables. Pages 704-738. 2001

[17] Prasun Choudhury et Benjamin Watson. Completely Adaptive Simplification of Massive Meshes. Northwestern University.

[18] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha et Madhav K. Ponamgi. I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments. Department of Computer Science. University of North Carolina.

[19] S. Gottschalk, M. C. Lin et D. Manocha, OBBTree: A Hierarchical Structure for Rapid Interference Detection Environments, Department of Computer Science, University of North Carolina.

[20] Tornero J, Hamlin J, Kelley RB. Spherical-object representation and fast distance computation for robotic applications. Proceedings of the IEEE International Conference on Robotics and Automation, Sacramento, CA, vol. 2, Avril 1991. p. 1602-1608.

[21] Laurend Testud, Le programmeur : DirectX 9, Programmation des jeux 3D, CampusPress, 2003, pages 54– 59.

[22] Meyer W. Distance between boxes: applications to collision detection and clipping. In Proceedings of the IEEE International Conference on Robotics and Automation. San Francisco. vol. 1. Avril 1986. p. 597-602.

[23] Lin MC, Canny JF. A fast algorithm for incremental distance calculation. In Proceedings of the IEEE International Conference on Robotics and Automation. Sacramento. Vol. 2. 1991. p. 1008-1014.

[24] Brian Mirtich. V-Clip: Fast and Robust Polyhedral Collision Detection. TR-97-05. June 1997.

[25] Chung Tat Leung, Q_COLLIDE - Quick Collision Detection Library, http://www.geocities.com/kelchung220/collision_library.html.

[26] Laurend Testud, Le programmeur : DirectX 9, Programmation des jeux 3D, CampusPress, 2003, pages 184 – 185.

[27] Laurend Testud, Le programmeur : DirectX 9, Programmation des jeux 3D, CampusPress, 2003, pages 90 – 94.

[28] Thomas C. Hudson, Ming C. Lin, Jonathan Cohen, Stefan Gottschalk, Dinesh Manocha, V-COLLIDE: Accelerated Collision Detection for VRML, Department of Computer Science, University of North Carolina.

[29] Ming Chieh Lin, Efficient Collision Detection for Animation and Robotics, PhD thesis, University of California, Berkeley, Decembre, 1993.

[30] E. Welzl, Smallest enclosing disks (balls and ellipsoids), Technical Report B91-09, Fachbereich Mathematik, Freie Universitat, Berlin, 1991.

[31] S. Gottschalk, Separating axis theorem, Technical Report TR96-024, Department of Computer Science, UNC Chapel Hill, 1996.

[32] G. van den Bergen, FreeSOLID: Software Library for Interference Detection, http://www.win.tue.nl/~gino/solid.

[33] G. van den Bergen, A Fast and Robust GJK Implementation for Collision Detection of Convex Objects, Department of Mathematics and Computing Science Eindhoven University of Technology, July 6, 1999.

[34] S. Caselli, M. Reggiani, M. Mazzoli, Exploiting advanced collision detection libraries in a probabilistic motion planner, Italy.

[35] Eric Larsen, Stefan Gottschalk, Ming C. Lin, Dinesh Manocha. Fast Proximity Queries with Swept Sphere Volumes, Technical report TR99-018, Department of Computer Science, University of N. Carolina, Chapel Hill.

[36] Christoph Fünfzig, Dieter W. Fellner, Easy Realignment of k-DOP Bounding Volumes,Institute of ComputerGraphics, Technical University of Braunschweig, Germany.

Liste des figures

Figure 1.1. Classification des modèles. Traduction de : A Taxonomy of 3D Model Representations. Ming C. Lin & Stefan Gottschalk, Collision detection between geometric models: a survey, University of North Carolina, page 2.

Figure 1.2. Modèle convexe et modèle non-convexe. Julien Guillem Lessard, Uqtr.

Figure 1.3. Division d'un beigne. Julien Guillem Lessard, Uqtr.

Figure 1.4. Opérations d'union, de différence et d'intersection dans CSG, Wikipedia, the free encyclopedia : http://en.wikipedia.org.

Figure 1.5. Représentation des deux normales former par une intersection. Julien Guillem Lessard.

Figure 1.6. Nombre de pairs selon le nombre d'objets, Julien Guillem Lessard.

Figure 1.7. Statue de Lucy simplifiée. Prasun Choudhury et Benjamin Watson. Completely Adaptive Simplification of Massive Meshes. Northwestern University.

Figure 1.8. Déterminer si une collision est possible à partir de sphères. Julien Guillem Lessard.

Figure 1.9. Frontière possible autour des objets. Thomas Larsson, Collision detection. Department of Computer Science, Mälardalen University.

Figure 1.10. Division successive d'une lampe à l'aide de sphères. hubbard 95.

Figure 1.11. Méthode par grillage. Julien Guillem Lessard.

Figure 1.12. Projection d'objet dans une dimension inférieur. Julien Guillem Lessard.

Figure 1.13. Les régions de Voronoï pour des points. Michael Ian Samos, Computational Geometry, Dissertation Information, 1978. page 175.

Figure 1.14. Projection à des fins de détection de collisions.
P. Jiménez, F. Thomas, C. Torras. 3D collision detection: a survey. Page 271

Figure 1.15. Région de Voronoï. P. Jiménez, F. Thomas, C. Torras. 3D collision detection: a survey. Page 277

Figure 1.16. Collision possible d'une balle en mouvement. Julien Guillem Lessard.

Figure 1.17. Trajectoire d'évitement. Laurend Testud, Le programmeur : DirectX 9, Programmation des jeux 3D, CampusPress, 2003, page 185.

Figure 1.18. Réflexion d'un vecteur sur un plan. Julien Guillem Lessard.

Figure 1.19. Architecture de détection de collision de I-Collide. Traduction de *Architecture for Multi-body Collision Detection*, I-Collide [18].

Figure 1.20. Boîtes superposés dans une dimension. Basé sur Figure 2: Bounding Box Behavior, I-Collide [18].

Figure 1.21. Projection des boîtes OBB sur un axe. S. Gottschalk, M. C. Lin et D. Manocha, OBBTree: A Hierarchical Structure for Rapid Interference Detection Environments.

Figure 1.22. Division successive de OBBtree. . S. Gottschalk, M. C. Lin et D. Manocha, OBBTree: A Hierarchical Structure for Rapid Interference Detection Environments.

Figure 1.23. Architecture de V-Collide. Traduction de *The system architecture of V-Collide* [28].

Figure 1.24. Transitions des états dans V-Clip [24].

Figure 1.25. Environnement d'expérimentation des algorithmes de détection de collisions [34] .

Figure 1.26. Temps d'exécution des différents algorithmes de détection de collisions pour résoudre dix fois le test de planification de mouvement dans une grille simple [34] .

Figure 1.27. Comparaison entre le test 1 et 2 [34].

Figure 1.28. Comparaison entre les temps de collisions pour le test 1 et 3 [34].

Figure 1.29. Temps exécution moyen pour une itération du test 4 [34].

Figure 2.1 Première version du jeu arkanoid. Taito. 1986.

Figure 2.2 DX-Ball, une des nombreuses versions du jeu Arkanoid. Micheal P. Welch, 1996-1998.

Figure 2.3 L'algorithme général du logiciel Arka3D. Julien Guillem Lessard.

Figure 2.4 Structure de développement. Julien Guillem Lessard.

Figure 2.5 États de l'objet Arka3D. Julien Guillem Lessard.

Figure 2.6 Aperçu du menu. Julien Guillem Lessard.

Figure 2.7 Aperçu du jeu. Julien Guillem Lessard.

Figure 2.8 Aperçu des murs intérieurs. Julien Guillem Lessard.

Figure 2.9 Aperçu des murs extérieurs. Julien Guillem Lessard.

Figure 2.10 Balle texturé. Julien Guillem Lessard.

Figure 2.11 Effet de lumière de la balle. Julien Guillem Lessard.

Figure 2.12 Disque opaque. Julien Guillem Lessard.

Figure 2.13 Disque translucide. Julien Guillem Lessard.

Figure 2.14 Cube simple. Julien Guillem Lessard.

Figure 2.15 Aperçu des huit cubes adjacents. Julien Guillem Lessard.

Figure 2.16. Aperçu d'un débris. Julien Guillem Lessard.

Figure 2.17. Plusieurs débris créer suite à la destruction d'un cube et explosions de quelques débris. Julien Guillem Lessard.

Figure 2.18 Petite explosion créée par un minuscule débris. Julien Guillem Lessard.

Figure 2.19 Grosse explosion créée par un cube explosif. Julien Guillem Lessard.

Figure 2.20 Phénomène de collisions cycliques entre 2 balles. Julien Guillem Lessard.

Figure 2.21 Rebondissement permissif des murs. Julien Guillem Lessard.

Figure 2.22 Permission accordée au disque. Julien Guillem Lessard.

Figure 2.23 Permission accordée à la vue. Julien Guillem Lessard.

Figure 2.24 Volume occupé par le disque déterminé par l'union d'une sphère et d'un plan orienté. Julien Guillem Lessard.

Figure 2.25 Régions de Voronoï d'une facette. Julien Guillem Lessard.

Figure 2.26 Rebonds sur le disque. Julien Guillem Lessard.

Figure 2.27 Rebondissement sur une face et sur un coin. Julien Guillem Lessard.

Figure 2.28 Plan de collision entre deux sphères. Julien Guillem Lessard.

Figure 2.29 Problème de rebondissement avec les sphères. Julien Guillem Lessard.

Figure 2.30 Vue de coté à travers un mur translucide. Julien Guillem Lessard.

Figure 2.31 Effet d'éclairage d'une balle sur un mur avec un nombre de polygones variables. Julien Guillem Lessard.

Figure 2.32 L'effet de lumière à travers un mur translucide. Julien Guillem Lessard.

Figure 2.33 Liste des textures utilisée pour les cubes indestructibles. Julien Guillem Lessard.

Figure 2.34 Vue derrière le disque translucide. Julien Guillem Lessard.

Figure 2.35 Vue loin derrière la scène. Julien Guillem Lessard.

Figure 2.36 Vue de coté sur la scène. Julien Guillem Lessard.

Figure 2.37 Suivi d'une balle. Julien Guillem Lessard.

Figure 3.1. Aperçu du tableau de cubes du scénario 1. Julien Guillem Lessard.

Figure 3.2 Répartition du temps d'exécution des différentes routines pour le scénario 1. Julien Guillem Lessard.

Figure 3.3 Aperçu du tableau de cubes du scénario 2. Julien Guillem Lessard.

Figure 3.4 Répartition des temps d'exécution des différentes routines pour le scénario 2. Julien Guillem Lessard.

Figure 3.5 Aperçu de tableau de cubes du scénario 3. Julien Guillem Lessard.

Figure 3.6 Répartition du temps d'exécution des différentes routines pour le scénario 3. Julien Guillem Lessard.

Figure 3.7 Répartition du temps d'exécution des différentes routines pour le scénario 4. Julien Guillem Lessard.

Figure 3.8 Répartition du temps d'exécution des différentes routines pour le scénario 5. Julien Guillem Lessard.

Figure 3.9 Répartition du temps d'exécution des différentes routines pour le scénario 6. Julien Guillem Lessard.

Figure 3.10 Comparaisons entre les environnements. Julien Guillem Lessard.

Figure 3.11 Résumé des répartition du temps d'exécution des différentes routines. Julien Guillem Lessard.

Figure 3.12 Répartition des temps pour la détection de collisions entre les balles et les cubes. Julien Guillem Lessard.

Figure 3.13 Nombres moyens de facettes présélectionnées. Julien Guillem Lessard.

Figure 3.14 Nombres moyens de facettes traitées. Julien Guillem Lessard.

Figure 3.15 Nombre d'erreurs générées par les scénarios. Julien Guillem Lessard.

Liste des tableaux

Tableau 2.4 Cheminement de la programmation d'Arka3D.

Tableau 2.2.1.1 Liste des objets composant le jeu Arka3D.

Tableau 2.5 États de l'objet Arka3D.

Tableau 2.6 Principaux attributs de l'objet Arka3D.

Tableau 2.7 Principales méthodes de l'objet Arka3D.

Tableau 2.8 Principaux attributs de l'objet Murs.

Tableau 2.9 Principales méthodes de l'objet Murs.

Tableau 2.10 Principaux attributs de l'objet Balles.

Tableau 2.11 Principales méthodes de l'objet Balles.

Tableau 2.12 Principaux attributs de l'objet Disque.

Tableau 2.13 Principales fonctions de l'objet Disque.

Tableau 2.14 Principaux attributs de l'objet Cubes.

Tableau 2.15 Principales méthodes de l'objet Cubes.

Liste des équations

Équation 1. Formule de la distance.

Équation 2. Réflexion d'un vecteur sur un plan.

Équation 3. Volume occupé par le disque.

Équation 4. Formule de la déviation.

Équation 5. Formule de transition de point de vue.