

# Stardust

Rapport deuxième soutenance

Koin-Koin z'Team

CRESTEY BENOÎT  
CALVET BRUNO  
CARRIGNON DAVID  
EGGENSPIELER RÉMI

INFOSUP

25 février 2003

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Répartition des tâches . . . . .	3
<b>2</b>	<b>Avancement du projet</b>	<b>4</b>
2.1	Moteur Graphique . . . . .	4
2.1.1	Loader 3D Studio Max . . . . .	4
2.1.2	Modélisation 3D . . . . .	5
2.1.3	Effet starfield . . . . .	5
2.1.4	Frustum culling . . . . .	6
2.2	Moteur de texte . . . . .	7
2.2.1	Principe . . . . .	7
2.2.2	Réalisation . . . . .	7
2.2.3	Modélisation de la console . . . . .	8
2.2.4	Conclusion . . . . .	8
2.3	Acquisition du clavier . . . . .	9
2.3.1	Travail réalisé à la première soutenance . . . . .	9
2.3.2	Nouvelle gestion du clavier . . . . .	9
2.4	Moteur de Gestion . . . . .	10
2.4.1	Scénario . . . . .	10
2.4.2	Moteur d'évènements . . . . .	10
2.4.3	Console . . . . .	11
2.4.4	Gestion des ennemis . . . . .	11
2.5	Collisions . . . . .	12
2.5.1	Documentation . . . . .	12
2.5.2	Implémentation des collisions . . . . .	13
2.5.3	Prévision pour les collisions . . . . .	14
<b>3</b>	<b>Avance du projet</b>	<b>15</b>
<b>4</b>	<b>Ressources</b>	<b>17</b>
<b>5</b>	<b>Travaux prévus pour la soutenance suivante</b>	<b>18</b>
<b>6</b>	<b>Conclusion</b>	<b>18</b>
<b>7</b>	<b>Annexes</b>	<b>19</b>

# 1 Introduction

Notre groupe est toujours composé de Benoît Crestey (SUP B2) le chef de groupe, David Carrignon (SUP A2), Rémi Eggenspieller (SUP D2), Bruno Calvet (SUP B1).

Nous développons un jeu de simulation spatiale en 3D qui est d'un style moins répandu ces derniers temps mais qui a tout de même connu des best-sellers tels que Wing Commander, X-Wing Vs Tie Fighter et le très célèbre Starlancer.

## 1.1 Répartition des tâches

Même si nous avons beaucoup travaillé ensemble, voici une courte liste des principales tâches de chacun pour cette soutenance :

Nom	Répartition des tâches
CALVET BRUNO	<ul style="list-style-type: none"><li>- Travaux sur les collisions.</li><li>- Gestion Acquisition des touches DirectInput.</li><li>- Site Internet.</li></ul>
CARRIGNON DAVID	<ul style="list-style-type: none"><li>- Importateur d'objets 3D Studio Max.</li><li>- Création du Starfield.</li><li>- Texturage.</li><li>- Modélisation.</li></ul>
CRESTEY BENOÎT	<ul style="list-style-type: none"><li>- Travaux sur les collisions.</li><li>- Gestion des événements.</li><li>- Gestion de la console.</li><li>- Gestion des ennemis.</li></ul>
EGGENSPIELLER RÉMI	<ul style="list-style-type: none"><li>- Affichage et texturage du texte.</li><li>- Affichage de la console.</li><li>- Site Internet.</li></ul>

## 2 Avancement du projet

### 2.1 Moteur Graphique

#### 2.1.1 Loader 3D Studio Max

Pour la première soutenance, nous avons réalisé un loader qui était en mesure de charger toutes les informations essentielles contenues dans les fichiers 3D Studio Max. Mais nous n'avions pas encore géré les textures appliquées à chaque objet. Ce fut la principale difficulté à résoudre pour cette soutenance. Le principe de lecture des informations fut le même pour les textures que pour les objet 3D. Nous utilisons pour cela la fonction read-buffer afin de lire les identifiants de 2 octets et la longueur des informations correspondant à l'identifiant stockée dans 4 octets. Ensuite, il ne reste plus qu'à traiter les informations suivant la valeur de l'identifiant.

Du fait de la structure des fichiers 3DS, la manière la plus simple de stocker les informations est de créer une structure fichier3DS contenant diverses champs dont deux listes chaînées. L'une contient les données de chaque modèles et l'autre contient celles des différents matériaux. Chaque objet possède une chaîne de caractère indiquant quel matériau lui est appliqué. Le problème de ce système de stockage de données est qu'il faut parcourir la liste chaînée des objets, puis celle des matériaux à chaque fois que l'on veut l'afficher. Or, dans notre projet, chaque objet 3D possède une texture que lui seul utilise. Nous avons donc préféré inclure les informations des matériaux dans la structure de chaque objet pour n'avoir à parcourir qu'une seule liste chaînée.

Dans ce but nous stockons tous les matériaux dans une liste chaînée comme nous l'avions fait pour les différents objets contenus dans chacun des fichier3DS. Ensuite, nous parcourons la liste des objets et nous associons la texture et ses paramètres à chaque objet. Enfin, nous créons une structure IDirect3DBaseTexture8 contenant les textures de chaque objet. Malheureusement un seul de nos objets, la sphère, utilise plusieurs fois la même texture, c'est pour cela que nous avons créé une procédure d'affichage qui n'utilise pas la texture d'objet à afficher mais celle du précédent ce qui permet de ne pas stocker plusieurs fois les mêmes données.

### 2.1.2 Modélisation 3D

Nous avons commencé la modélisation de nos vaisseaux en utilisant le logiciel 3D Studio Max 5 pour pouvoir exporter en fichier 3DS et charger nos objet avec notre loader. Notre toute première modélisation fut présentée à la première soutenance, mais ce modèle fut totalement abandonné vu son manque d'esthétisme. Pour réaliser un vaisseau d'une certaine qualité, nous avons dû acquérir des méthodes de modélisation. Pour cela nous avons lu des tutoriaux ainsi que des magazines tels que Studio Multimédia.

Nous avons retenu une methode particulière, la création d'objets à l'aide de lignes. Dans la pratique le vaisseau fut d'abord réalisé grâce aux lignes, puis, une fois la forme souhaitée obtenue, on utilise un modificateur qui crée la surface de l'objet. La plus grande des difficultés rencontrée fut d'obtenir un bon compromis entre le nombre de polygones et la qualité visuelle du vaisseau. Le modèle final que nous avons obtenu est composé de moins de 500 polygones. Une fois le modèle terminé, nous avons utilisé des techniques avancées de texturage tel que le UVW Unwrap. Ceci permet d'appliquer un morceau de texture comme on le souhaite sur un ou plusieurs polygones. Nous avons ainsi réussi à réaliser un vaisseau avec un design d'une certaine qualité.

### 2.1.3 Effet starfield

Pour donner un impression de vitesse dans la vue interne du vaisseau, nous avons voulu créer un effet ressemblant à l'écran de vielle starfield de windows. Pour cela nous avons cherché des tutoriaux sur internet, mais nous n'avons trouvé que des explications pour créer des starfield en 2 dimensions. Premièrement, nous pensions faire un starfield en post-rendu en créant nos étoiles grâce à des pixels. Après quelques essais, nous avons remarqué que l'affichage d'un pixel en Direct3D est à peine visible, et donc nous avons décidé de créer des particules pour afficher les étoiles en mouvement. En utilisant ceci, il était devenu impératif de ne plus gérer les étoiles en 2D mais en 3D, afin de leur donner un effet de profondeur. La première façon de gérer le starfield fut de déplacer les particules en fonction du déplacement du vaisseau. Dès qu'une particules sort de l'écran, elle est replacée au loin devant la caméra pour donner un effet de continuité. Cependant notre starfield comportait de nombreux problèmes de coordinations et l'effet ne paraissait pas très réaliste. Nous avons donc décidé de changer totalement la gestion des étoiles.

Donc nous avons décidé de rendre l'effet tel qu'il se produirait dans la réalité. Pour cet effet, Les particules ne bougent pas et l'effet de starfield est simplement créé par le déplacement de la caméra dans l'espace 3D. Nous fûmes alors confrontés à deux problèmes : d'un coté, il fallait faire pivotert les particules pour qu'elle apparaissent toujours face à la

caméra, et de l'autre, nous devons détecter quand les particules sortaient de l'écran pour les replacer dans celui-ci.

Pour que les particules soient toujours tournées vers l'écran, nous avons utilisé la fonction de Direct3D permettant de créer la matrice de vue avec comme paramètres, le vecteur position de la particule en question, celui de la position de la caméra et de l'orientation de la caméra. Ensuite on inverse la matrice obtenue et on l'utilise pour transformer le repère monde avant de positionner la particule.

Pour détecter les particules qui sont en dehors du champ de vision de la caméra, nous avons utilisé le Frustum culling.

#### **2.1.4 Frustum culling**

Le Frustum culling est une technique permettant de déterminer si un point est situé dans les six plans de l'écran. Dans la pratique, nous avons créé une procédure qui calcule les coordonnées des 6 plans en fonction de la matrice de vue et de projection, et pour déterminer si un point est entre les plans, nous avons créé une fonction booléenne qui prend en paramètres les coordonnées du point. Ces algorithmes furent créés à partir de tutoriaux trouver sur internet.

Le frustum culling n'est pour l'instant utilisé que pour l'effet de starfield. Mais nous envisageons de l'utiliser pour tous les objets à afficher. Cependant nous devons trouver un moyen de résoudre le cas où le centre d'un objet n'est pas dans l'écran et qu'un morceau de l'objet devrait être affiché.

## 2.2 Moteur de texte

Lors de la première soutenance, un moteur d’affichage de texte fut créé. Or, celui-ci ne permettait d’afficher qu’une seule police de caractères, et le placement de ceux-ci était hasardeux (problèmes mineurs d’alignement du fait de l’utilisation d’une fonte non générique). Ainsi, il fut décidé d’améliorer grandement les possibilités de celui-ci, en le rendant en partie générique. Il sera alors permis de choisir sa propre fonte et de placer le texte en coordonnées d’écran, à la manière de DirectDraw, tout en utilisant le même principe que précédemment.

En effet, afficher du texte en passant par le moteur graphique s’avère judicieux. Pour cette soutenance, le but est de concevoir un moteur d’affichage de texte générique, c’est à dire, à partir d’un modèle de texture de police de caractères, garder le même algorithme de découpage de la texture, ce qui évitera cette fois de devoir noter les coordonnées de chaque caractère sur la fonte.

### 2.2.1 Principe

Le principe repose sur une refonte quasi-complète du moteur d’affichage de texte. L’accès aux coordonnées des caractères doit se faire par la valeur ASCII et non par le caractère lui-même. De plus le nombre de caractère doit être limité aux stricts besoins de l’affichage de texte, en excluant les caractères exotiques. L’affichage doit pouvoir gérer autant de fontes différentes que désirées, et le placement du texte doit être plus intuitif en laissant supposer des coordonnées d’écran en pixels. Le principe d’alignement, lui, ne changeant pas.

### 2.2.2 Réalisation

La première étape fut de changer complètement le type du tableau contenant les coordonnées des caractères. Le type précédent contenait un champ avec le caractère voulu. L’accès à un caractère entraînait automatiquement un parcours du tableau jusqu’à la lettre voulue pour accéder à ses coordonnées. Le nouveau type qui fut créé fut un tableau de 93 caractères (expliqué ci-après) commençant à 32 et finissant à 126, c’est à dire le code ascii de chaque caractère. Ainsi, au lieu de passer par un parcours de tableau, on accède à la case voulue par le code ASCII de la lettre. Cette méthode s’avère être un gain précieux en temps de calcul.

La seconde étape fut de choisir 93 caractères parmi le code ASCII, c’est à dire du code 32 à 126 moins deux caractères inutiles. Un modèle de texture fut choisi, c’est à dire une texture carrée de 255 \* 255 pixels ; ainsi chaque caractère prendra un morceau de 25 \* 25 pixels de la texture, les 5 pixels restants ne seront pas pris en compte. Lors de la soutenance précédente, le principe reposait sur le fait de noter chaque coordonnée de chaque

caractère dans un tableau : tâche longue et fastidieuse. Cette fois, la procédure d'initialisation des coordonnées des caractères va incrémenter de 25 pixels en 25 pixels pour la première coordonnée du caractère, puis celle-ci va utiliser le fichier ".ini" de la police qui contient la largeur de chaque caractère en pixels. Ainsi, la seconde coordonnée significative du caractère peut être calculée. Les calculs suivants des coordonnées restant les mêmes. Ainsi, les calculs sont grandement simplifiés.

La troisième étape fut d'adapter la procédure d'affichage du texte, afin de pouvoir le placer à partir des coordonnées de l'écran (ici 1024\*768). En effet, le précédent moteur plaçait le texte par rapport aux unités de la scène. Ainsi, par des calculs résultant des coordonnées beaucoup plus significatives (car en pixels) le placement du texte fut grandement simplifié.

La quatrième étape fut d'initialiser les tableaux de coordonnées puis d'initialiser la liste des textures de chaque police. Les procédures d'affichage de texte proprement dites furent modifiées. En effet, il est désormais possible de passer en paramètre la fonte désirée dans la procédure d'affichage de texte, l'appel de la texture se faisant indépendamment.

### **2.2.3 Modélisation de la console**

Une première version de l'interface de la console fut alors créée. Celle-ci se présente sous la forme de six rectangles texturés. Le design de cette console ouvre la voie à l'élaboration d'un principe d'affichage de sprites dans le moteur 3D qui sera , par exemple, utilisé pour la conception de la vue du cockpit.

### **2.2.4 Conclusion**

La réalisation fut relativement complexe, mais très intéressante au niveau de la recherche d'optimisation d'un principe. En définitive, la réalisation d'un moteur d'affichage de texte et la modélisation graphique de la console ouvrent la voie à la création de la console du jeu, et ainsi interagir directement avec le programme en cours d'exécution.

## 2.3 Acquisition du clavier

### 2.3.1 Travail réalisé à la première soutenance

Pour la première soutenance, nous avons au tout début utilisé la gestion des touches par des contrôles Delphi mais ce n'était pas un façon pratique de gérer le clavier. Nous avons donc opté pour l'utilisation de Direct Input afin de gérer les touches pour que nous puissions déplacer le vaisseau comme bon nous semblait.

Pour la première soutenance nous avons intégré dans le jeu du Direct Input qui renvoyait les touches qu'il détectait comme enfoncées, mais nous ne pouvions savoir quelles touches venaient d'être enfoncées et lesquelles étaient enfoncées depuis la boucle précédente.

### 2.3.2 Nouvelle gestion du clavier

Pour cette soutenance, nous voulions avoir la possibilité de taper des commandes une fois le jeu lancé via une console. Pour cela, nous avons dû modifier la gestion du clavier qui, comme il a été dit précédemment, n'était pas assez efficace pour nous. Nous avons donc recherché différentes documentations sur Internet pour gérer le clavier en Direct Input avec un buffer. Nous avons trouvé beaucoup de documentation expliquant comment gérer les touches, mais très peu expliquaient comment gérer un buffer afin de stocker l'état de chaque touche à chaque moment du jeu. Notre but était de savoir en permanence l'état de chaque touche, si elles sont pressées ou non, et utiliser un buffer était la seule solution.

Au démarrage, tout comme l'ancienne version, une vérification de l'ancienne version de Direct Input est effectuée. Toutes les touches ne sont pas traitées de la même façon, mais le traitement permet de connaître l'état de chacune d'entre elles. Celui-ci n'est communiqué que lorsqu'il change, ainsi on peut aisément savoir quelles touches sont pressées. Toutes les touches ne sont pas traitées de la même manière : lorsque la console n'est pas activée l'état des touches de direction est stocké dans des variable booléennes qui sont alors testées pour voir s'il faut déplacer le vaisseau ; les autres touche étant gérées de telle manière que lorsqu'on appuie dessus, la commande liée est exécutée une fois. En mode console, les touches agissent comme s'il s'agissait d'un traitement de texte normal.

Désormais, du point de vue de la gestion des contrôles, il nous reste à intégrer le paramétrage des touches au menu lorsque celui-ci sera créé et, éventuellement, la gestion de la souris pour le menu.

## 2.4 Moteur de Gestion

### 2.4.1 Scénario

Comme exprimé dans notre précédent rapport, le jeu devra proposer au joueur des scénarios, lesquels seront stockés dans des fichiers de scénarios. Comme l'étape précédente fut de créer un éditeur de scénario, la nouvelle fut de gérer le chargement et l'exécution dans le jeu.

### 2.4.2 Moteur d'évènements

Le fichier de scénario est constitué d'une succession d'instructions, contenant :

- Le temps à laquelle l'opération sera exécutée.
- La commande à exécuter.
- Les différents paramètres d'exécution des vaisseaux concernés, ou des coordonnées par exemple.

Chaque ligne d'instruction est composée suivant une syntaxe définie :

|tps|commande|paramètres

- tps : c'est le temps à laquelle est exécutée la commande. Ce temps, incrémenté par la boucle principale du jeu, peut être défini soit relativement au temps défini depuis le début de la partie, soit relativement au temps déjà écoulé. Si rien n'est entré, l'action est exécutée immédiatement.
- commande : c'est tout simplement la commande à effectuer. Quelques exemples : vso\_position, vso\_vitesse, vso\_pv, ...
- paramètres : les différents paramètres seront découpés suivant le caractère ~ .

exemples de commandes :

- |100|vso\_position|joueur~10~0~0 : Positionne le vaisseau du joueur au cycle 100, à la position x=10, y=0, z=0 dans le repère du jeu.
- ||vso\_vitesse|robert~1000 : Le vaisseau nommé robert accélérera (ou décelérera, suivant sa vitesse actuelle) progressivement à la vitesse 1000 (nous n'avons pas encore d'unité de vitesse établie). Il faut noter que cette vitesse sera atteinte progressivement, pour des raisons de réalisme. Comme aucun temps n'est précisé, cette action interviendra immédiatement.

Le fichier est lu directement : les lignes de commentaires ou d'espace ne sont pas prises en compte, et le programme parse chaque ligne d'action et les ajoute dans la liste des actions à accomplir.

La liste des actions à accomplir est en fait une liste chaînée contenant dans chacun de ses enregistrements les différents paramètres. celle-ci est triée en fonction des moments d'appels à chaque action, et à chaque appel de la boucle de jeu, il suffit de regarder les premiers éléments de la liste et de les exécuter si leurs temps d'exécutions sont égaux ou inférieurs au temps de la boucle de jeu.

### 2.4.3 Console

Afin de pouvoir intervenir directement sur les données et de diminuer considérablement les temps de développement, nous avons également mis en place une console qui nous permette d'exécuter directement les commandes du scénario, ainsi que quelques autres destinées au débogage. Ceci nous permet de tester directement les nouvelles fonctionnalités de notre jeu et d'afficher les résultats nous permettant de vérifier si les algorithmes sont corrects.

La console est composée d'une ligne nous permettant de saisir les commandes, et de l'affichage des résultats dans une partie supérieure. Nous avons également prévu le cas où le texte dépasserait de celle-ci, et nous avons mis en place un système de déplacement pour pouvoir consulter les anciennes données (les résultats affichés sont stockés dans une liste chaînée, on peut donc en stocker autant que la mémoire le permet).

### 2.4.4 Gestion des ennemis

Afin de gérer l'affichage et la gestion des ennemis, qui seront bientôt pilotés par l'ordinateur, nous utilisons une liste chaînée contenant les données de chaque vaisseau (position, direction, points de vie, ...). Cette liste chaînée est parcourue à chaque cycle de calculs : on regarde si le vaisseau doit accomplir certaines actions (tourner ou tirer par exemple), puis la position et la trajectoire du vaisseau sont mises à jour.

## 2.5 Collisions

Notre jeu Stardust est un jeu de combat spatial en 3D. Il est donc primordial qu'une gestion des collisions y soit intégrée afin de pouvoir gérer tout ce qui est tirs d'armes, collisions entre vaisseaux, collisions entre les vaisseaux et les éléments du décor (astéroïdes, débris, etc...).

Le problème majeur des algorithmes de collisions entre plusieurs éléments est qu'ils sont soit assez économiques et peu précis, soit plus précis et beaucoup plus gourmands au point de vue calculs. Or, pour un jeu tel que le notre il faut le moins de perte de vitesse possible durant la partie. Il nous fallait donc trouver des algorithmes efficaces et économiques.

### 2.5.1 Documentation

Nous avons bien sûr commencé par nous documenter sur Internet concernant les différentes manières de détecter des collisions en 3 dimensions. Nous avons ainsi trouvé diverse manières de procéder :

- Détection par pixels : cette manière serait extrêmement coûteuse pour notre jeu, car beaucoup de pixels sont chargés. Cette technique est néanmoins très efficace, mais plutôt destinée aux jeux en 2 dimensions.
- Détection via Oriented Bounding Box (OBB) : cette technique est assez complexe, car il faut calculer la boîte entourant l'objet puis ensuite faire les calculs de collisions, ce qui met assez de temps mais peut être assez précis.
- Détection grâce à des Axis Aligned Bounding Box (AABB) : cette technique est assez simple car il suffit de définir une longueur, une largeur et une profondeur par rapport au centre de l'objet et la boîte est ainsi créée.
- Détection par Bounding Sphere, cette technique est elle aussi très économique car il suffit de définir un rayon de l'objet et la sphère est créée.

Nous n'avons répertorié ici que les techniques qui nous ont paru efficaces pour notre projet, mis à part la détection par pixel que nous n'avons jamais envisagé d'utiliser.

Lors de nos recherches, nous avons trouvé des sites conseillant de trier les objets en fonction de leur coordonnées afin de simplifier les calculs de collisions, ainsi que d'éviter certaines comparaisons pour encore simplifier les calculs. En effet il serait inutile de tester les collisions entre plusieurs missiles tirés par une même personne par exemple ou encore

les collisions entre des tirs de laser. Ceci va nous permettre de simplifier un nombre non négligeable de calculs.

Il existe d'autres manières de détecter les collisions telles que les arbres BSP, la détection suivant les secteurs apparaissant à l'écran, où chaque objet étant situé dans un secteur défini est alors comparé aux autres objets étant dans ce même secteur, et ainsi on peut voir si ils se collisionnent. Mais cette technique n'est pas du tout pratique pour un univers en 3D, bien d'autres techniques sont utilisables mais ne nous paraissent pas utiles.

### 2.5.2 Implémentation des collisions

Nous avons tout d'abord décidé que les collisions seraient gérées grâce à des Axis Aligned Bounding Box, car la documentation disponible était assez claire et que cette technique est assez simple à mettre en oeuvre.

Nous avons créé un algorithme nous permettant de trier tous les éléments en fonction de leur coordonnées par rapport à l'axe des abscisses. Nous obtenons ainsi une liste chaînée triée de tous les objets chargés dans le jeu.

Lorsqu'un nouvel objet est chargé dans le jeu, celui-ci est ajouté à une liste de collisions. Le tri est alors lancé. Il s'agit en fait de comparer la coordonnée x de l'élément actuel à celle de l'élément suivant, si celle-ci est plus grande alors les deux éléments sont inversés dans la liste chaînée. Le tri peut prendre un certain temps, car il doit tout de même comparer les coordonnées de beaucoup d'éléments. Pour le simplifier, nous avons fait en sorte qu'une fois un élément trié, c'est à dire que tous les éléments le suivant dans la liste ont une coordonnée d'abscisse plus grande, alors le tri pour l'élément courant est terminé. Pour ce faire nous avons utilisé un booléen contenant l'état du tri pour l'élément courant. Si celui-ci est à TRUE alors le tri repart du premier élément de la liste nouvellement triée.

Cet algorithme a une complexité de  $(n^2)$ , mais il suffit de  $(n^2 - n)/2$  calculs pour faire le tri ce qui, d'après la documentation que nous ayons pu trouver, est le nombre de calculs le plus simple en général. Pour le tri, nous utilisons une liste chaînée et non une liste doublement chaînée, car elle nous aurait peut-être permis de créer la liste de collisions plus aisément, mais le maniement aurait été bien plus complexe. Qui plus est, pour les tests de collisions, elle n'aurait été d'aucune utilité. Nous avons alors pensé utiliser l'algorithme de tri rapide, mais la liste que nous utilisons n'était pas équilibrée, et cela nous aurait mené à un algorithme plutôt lent.

Une fois cet algorithme de tri créé, nous avons pu commencer la réalisation de l'algorithme de collisions

Comme nous l'avons dit plus haut nous avons choisi d'utiliser des AABB. Pour ce faire, nous avons défini un paramètre rayon pour chaque objet du jeu : celui-ci permettra de créer la boîte. Notre algorithme fonctionne de la manière suivante : il part du premier élément de la liste de collisions et compare sa coordonnée d'abscisse à tous les éléments le suivant dans la liste de collisions. En réalité, il ajoute à sa coordonnée d'abscisse son rayon et compare si la coordonnée résultante est plus grande que la différence de la coordonnée d'abscisse de l'élément suivant et de son rayon, plus précisément en posant  $x_1$  l'abscisse du premier point,  $x_2$  l'abscisse du deuxième et  $r_1$  et  $r_2$  leurs rayons respectifs. Voici la comparaison qui est faite :  $(x_1 + r_1) \Rightarrow (x_2 - r_2)$ . Si la condition précédemment citée est remplie alors on teste la même chose pour les coordonnées en  $y$ . Si elles correspondent, on fait de même pour les coordonnées en  $z$ . Si finalement, tout correspond, on lance un autre algorithme que je vais expliquer ultérieurement. Si l'algorithme de collisions ne trouve aucune correspondance lors de la première comparaison ou des autres, alors l'algorithme compare avec le suivant et ce jusqu'à la fin de la liste. Ainsi, les éléments ne sont jamais testés deux fois entre eux, ce qui simplifie beaucoup les calculs et donne le même nombre de calculs que l'algorithme de tri.

L'opération effectuée lorsque deux AABB de deux objets se touchent, on passe alors à une détection plus précise en utilisant des Bounding Sphere. Pour ce faire, on regarde si la différence des coordonnées en  $y$  des deux objets est inférieure à la somme des rayons, puis on fait de même pour les coordonnées en  $z$  :  $(y_1 + y_2) \leq (r_1 + r_2)$  puis  $(z_1 + z_2) \leq (r_1 + r_2)$ . Ces calculs faits par Bounding Sphere augmentent quelque peu la précision de la collision.

### 2.5.3 Prévision pour les collisions

Pour les prochaines soutenances nous avons prévu diverses modifications :

- Simplification des algorithmes existants. En effet, dans la mesure du possible, nous simplifierons les algorithmes afin d'accélérer leur exécution lors de l'appel par le jeu
- Définition de "sous bounding Sphere" afin d'affiner la précision des collisions. Lorsque deux objets se toucheront, on fera des tests sur les sous Bounding Sphere du vaisseau. Ces Bounding Sphere seraient définies avec l'objet, et on pourrait ainsi les comparer.
- On pourra aussi éventuellement intégrer l'utilisation de Oriented Bounding Box pour les premiers tests afin d'être dès le départ plus précis qu'une AABB ou qu'une Bounding Sphere.

### **3 Avance du projet**

**Tâches prévues dans le cahier des charges :**

#### **Importateur d'objets 3D Studio Max**

L'importateur d'objet est totalement opérationnel dans le projet, il est finalisé et ne sera probablement plus modifié.

#### **Démonstration du moteur graphique**

Nous présenterons une version du moteur graphique avec affichage du vaisseau du joueur, vue interne, ainsi que des ennemis et des armes. Le nombre d'images par secondes affichées est pour le moment acceptables, même si l'on tente d'afficher un grand nombre de vaisseaux.

#### **Moteur Physique avec gestion du déplacement**

Le déplacement est totalement opérationnel et intégré dans le moteur graphique, et couplé au module de gestion des touches.

#### **Début des travaux sur les collisions**

Nous avons déjà une collision simplifié, nous allons continuer afin de proposer uen détection plus fine.

#### **Gestion des touches**

Le module de gestion des touches est finalisé, et seules de nouvelles touches serons ajoutées dans les veriosn suivantes.

#### **Site Internet Simplifié**

Le site internet est déjà bien avancé et présentable, il présente notre groupe, notre projet, et propose le téléchargement des différents rapports.

**Tâches réalisées non prévues dans le cahier des charges :****Gestion et affichage des ennemis**

Nous sommes déjà capables d'afficher et de gérer simplement des ennemis.

**Début de la gestion des armes**

Nous sommes capable d'afficher, de faire progresser et de détecter les collisions des armes avec des vaisseaux.

**Gestion du script d'événements**

La base du script d'événements est créée, nous allons continuer d'implémenter différentes actions.

**Modélisation / Texturage**

Un premier vaisseau à été modélisé, il sera surement inclut lors de la version finale du jeu.

## 4 Ressources

Différents sites Internet visités :

- <http://www.dev-gallery.com/programming/opengl/colision/collision.html>, un site contenant un petit tutorial pour les collisions, assez clair
- [http://www.gamasutra.com/features/20000330/bobic\\_01.htm](http://www.gamasutra.com/features/20000330/bobic_01.htm), site très complet concernant les collisions bien que certains articles soient assez techniques
- <http://www.ddj.com/documents/s=983/ddj9513a/9513a.htm>, site intéressant fournissant des explications et des conseils concernant les collisions, à visiter si on recherche des infos sur les différentes méthodes de collisions.
- <http://francis.dupont.free.fr/coindev/collision.htm>, site intéressant et assez clair concernant les collisions
- <http://www.jedi-project.org>
- <http://www.delphisource.com>
- <http://www.game-dev.net>, site contenant divers tutoriaux sur l'utilisation des quaternions
- *L'intro DirectX* aux éditions CampusPress, livre de DirectX intéressant et assez clair
- *DirectX Delphi Game Programming*, Difficile d'accès, car en anglais, mais très clair et pratique pour une initiation à DirectX
- *L'Intro DirectX* - Campus Press - Difficile d'accès car les exemples sont en C++, mais traite de parties très intéressantes au niveau de DirectX3D
- *Delphi 4 guide du développeur* - Campus Press - Très pratique par l'initiation à l'environnement Delphi
- *Turbo Pascal 7* - Eyrolles - Bien expliqué, pratique pour se former au langage Pascal

## 5 Travaux prévus pour la soutenance suivante

### Collisions

Nous comptons continuer le travail sur les collisions en simplifiant le plus possible les algorithmes existant et en affinant la précision de la détection.

### Gestion du Son

Nous pensons peut être inclure une ébauche de la gestion du son qui nous donnera plus d'interactivité à notre jeu.

### Début de la gestion de l'IA

Nous avons prévu d'intégrer une IA à notre jeu et nous vous montrerons surement une ébauche de celle-ci.

### Moteur graphique

Pour la troisième soutenance nous présenterons un menu assez simple.

### Affichage des armes

Nous allons créer des nouvelles armes notamment l'affichage de laser réaliste.

### Site Internet

Nous continuerons bien sur d'améliorer le site Internet.

## 6 Conclusion

Nous avons complété tous les travaux pour cette deuxième soutenance, et nous avons largement conservé notre avance sur le cahier des charges. Nous avons l'impression d'avoir été efficaces dans notre organisation et nos méthodes de travail et nous pensons continuer ainsi.

## **7 Annexes**

Planche de Texture de notre vaisseau.

Modélisation du vaisseau sous 3D Studio Max.

Nouvelle console.

Vaisseau rendu dans le moteur 3D.