Université Paris-Est Marne-La-Vallée

Rapport du projet de fin de Semestre
en Programmation, Synthèse d'image
et Architecture Logicielle.

Rapport Projet Smashstein

IMAC - Deuxième année

14 Janvier 2010

Lionel Augé
Adrien Benoist
Arnaud Casella
Vincent Hurbourque (Chef de Projet)
Julien Issartel
Gabrielle Luypaerts

Table des matières

Table des figures

Liste des tableaux

1 Introduction

Le but de ce projet est de créer un jeu vidéo temps réel en *C++* depuis la phase de réflexion jusqu'à la phase de réalisation. Ce projet, relativement complexe, a une portée dans différents domaines, allant de la confection de modèle 3D, à la création d'IA. Nous développerons toutes les étapes et tous les aspects présents dans notre jeu au cours de ce rapport. Le sujet était composé de trois parties peu dissociables dans un tel projet, à savoir la programmation, l'architecture logicielle et la synthèse d'image.

Dans un premier temps nous ferons une brève introduction de notre jeu pour comprendre ces principes et l'univers ainsi que les technologies utilisées. Puis nous verrons comment s'est organisée la communication interne sur un projet d'une telle envergure.
En ce qui concerne le projet à proprement dit, nous avons distingué deux parties importantes, l'architecture du projet permettant de comprendre comment interagissent les différents acteurs de notre jeu, et les fonctionnalités du programme, regroupant aussi bien l'interface *QT* que la gestion de l'Intelligence Artificielle, l'animation, le moteur physique, tout le chargement de notre scène par *XML* et les environnements sonores. Puis nous conclurons sur le bilan du projet avec le récapitulatif des tâches accomplies, des fonctionnalités non-implémentées, ainsi que les problèmes majeurs rencontrés. Enfin en annexe vous trouverez le manuel d'utilisation de notre jeu.

Notre groupe de travail est formé de six membres : Lionel Augé, Adrien Benoist, Arnaud Casella, Vincent Hurbourque, Julien Issartel, Gabrielle Luypaerts.

2 Introduction au projet JediSmash

2.1 Le Jeu

2.1.1 Présentation de l'univers

Nous avons décidé d'orienter notre jeu vers de l'action et du combat. Nous étions tous d'accord pour que notre projet soit dans la digne lignée d'un SmashBros. La modélisation de personnages tirés de Star Wars était donc obligatoire. Les animations devaient également représenter des mouvements plus ou moins sortis de cet univers, tel que le coup de sabre laser et l'utilisation de la "Force". Nous avons essayé, à travers la bande son, de retranscrire au mieux l'univers de George Lucas. Quant à l'arène, elle représente un demi-astéroïde sur lequel notre personnage et ses ennemis évolueront. Tout ces principes simples rendent plus facile l'immersion de notre joueur, car il est plus facile d'entrer dans un univers cohérent .

2.1.2 Principe de Jeu

Les principes du jeu sont relativement simples, puisque comme tout SmashBros, notre avatar (dirigé par le joueur, via le clavier) doit éjecter les ennemis présents dans l'arène. Cependant nous avons introduit une petite difficulté supplémentaire à notre projet, avec la présence d'un robot allié devant être protégé. Malgré ses compétences à attaquer les ennemis, sa fragilité l'empêche de rester longtemps en vie. C'est donc à notre Jedi (avatar) de le protéger, sous peine de perdre la partie.

Les conditions de défaite sont donc au nombre de deux ; soit notre personnage est éjecté, soit son allié meurt.
Les conditions de victoire quant à elles sont lorsque tous les ennemis sont éjectés.

2.1.3 Scène

Comme nous l'avons dit notre scène est un astéroïde correspondant à notre arène de combat. Initialement tous les personnages s'y trouvent. Elle est suspendue dans les airs et n'est pas fermée ni d'un côté, ni de l'autre, pour que le Jedi puisse tomber. Cela implique l'utilisation d'un moteur physique.

2.1.4 Personnages

Notre avatar sera représenté par un Jedi. Il aura donc tous les pouvoirs et accessoires qui incombent à un Jedi, à savoir qu'il possèdera un sabre laser dont il pourra se servir pour les attaques rapprochées, ainsi que des coups de "Force" qui éjecteront les ennemis en arrière. Il pourra aussi donner des coups de pieds. Bien sûr notre personnage étant libre dans la scène il pourra se déplacer sur les axes X et Y pour les déplacements et les sauts.

Le Jedi sera accompagné de robot allié sur l'arène, R2D2. Ce personnage ne sera pas jouable et sera contrôlé par une intelligence artificielle (IA). Il a une faible capacité d'attaque. Son unique rôle est de créer une difficulté supplémentaire.

Sur notre arène seront également présents des ennemis. Il en existera de deux sortes. Les premiers dotés d'une IA relativement basique, permettant de lancer des attaques à distance, ainsi que de porter des coups directement à notre avatar pour les attaques au corps à corps. Les robots pourront également se déplacer de manière horizontale sur un plan et de manière verticale pour changer d'étage. Leur IA leur permettra de fuir notre Jedi lorsqu'ils seront en mauvaise posture.
Le deuxième type d'ennemis présent dans la scène sera un maitre Sith. Il n'y en a qu'un et il possède une IA légèrement plus poussée. Il sera comme notre avatar et possèdera tout comme lui les mêmes coups (sabre laser, coup de pied, jet de "Force"). Ce robot n'attaque que le Jedi et l'assène de coups plus puissants et surtout à un rythme plus important. Sa vitesse de déplacement est également revue à la hausse.

2.2 Technologies utilisées

Pour mener à bien notre projet nous avons dû utiliser plusieurs technologies. Voici un bref récapitulatif exhaustif de celles dont nous nous sommes servis.

2.2.1 LaTeX

Notre rapport a été réalisé en LaTeX. L'utilisation de ce format nous a vivement été conseillé. De plus il donne un caractère professionnel à notre rapport, car très souvent utilisé par la communauté scientifique. Enfin, la

plupart d'entre nous n'ayant jamais fait de LaTeX, il était intéressant de prendre en main ce "langage", permettant de dissocier le fond de la forme.

2.2.2 OpenGL

Tout ce qui a trait au rendu visuel a été traité par de l'OpenGL. C'est à dire que toutes les textures ont été gérées par de l'OpenGL. Nous nous en sommes également servi pour tout les déplacements et rotations des objets de notre scène. Nous avons ainsi pu créer notre image d'arrière plan grâce à un *worldbillboard*, par exemple.

2.2.3 MD2

Le format MD2 est un format de fichier contenant les données des modèles en 3D chargées depuis un fichier. C'est un format relativement simple lorsque l'on en a compris le principe qui permet une animation par frame, donc correspondant tout à fait à notre projet. Le MD2 ne contient pas de texture, nous avons donc utilisé des formats *ppm* pour texturer tous nos objets.

2.2.4 C++

Pour un projet comme celui qui nous a été proposé, il fallait utiliser un langage orienté objet. Mise à part le fait que le C++ nous était imposé, il nous paraissait évident d'utiliser un tel langage.

2.2.5 XML

Le XML a joué un rôle très important dans le projet, car toutes les informations de notre scène sont contenues dans ce (ces) fichiers. On y stocke le nombre de d'objets "plateforme", ainsi que leur taille, mais également le nombre nécessaire de sauts entre les plateformes. Ils nous permettent de créer un FS/APS ainsi qu'un graphe entre les différentes structures sur lesquelles les personnages pourront se déplacer.
C'est aussi le XML qui stocke les personnages de notre scène à charger au début du programme avec leur md2, leurs animations mais aussi les sons liés à ces dernières.

2.2.6 QT

L'interface de notre projet est réalisée à l'aide de QT. Cet ensemble de bibliothèques multiplateforme permet de créer des GUI très aboutis.

3 Organisation

3.1 Définition des rôles

Pour ce projet nous étions une équipe formée de six membres. Vincent en tant que chef de projet a été chargé de répartir les rôles selon les capacités mais aussi les souhaits de chacun. Tout au long du projet, il a fallu redéfinir les tâches des personnes selon les périodes afin de gagner du temps et de structurer le projet. Malgré que Vincent aie affecté des rôles et tâches nous avons tous essayé d'avoir une vision globale du programme et, nombre de fois un responsable d'une tâche est intervenu dans celle d'un autre. Voyons la répartition des rôles :

Lionel Augé : Responsable 3D. Il s'est chargé de la modélisation des personnages et du décor sous 3DSMax avec animations, textures, et exportation puis intégration au projet. Il s'est occupé de charger les modèles md2 dans le programme (chargement textures, rotation des modèles etc...).

Adrien Benoist : Responsable documentation. Recherche d'informations sur l'utilisation de doxgen ainsi que prise en main de Latex. Il a centralisé les documents liés aux projets et a mis en forme le rapport. Concernant le code celui-ci s'est initié à la bibliothèque SDL mixer afin de gérer les effets sonores.

Arnaud Casella : Responsable technique. Son rôle est d'avoir une connaissance totale du projet dans ces moindres parties et d'ainsi d'orienter les programmeurs et de les conseiller. Il s'occupe aussi de mettre en place l'AI avec les plug-ins.

Vincent Hurbourque : Chef de Projet. Son rôle en tant que chef de projet a été de bien dissocier toutes les parties à réaliser et de répartir leur création. Il a implémenté l'IA des bots avec les graphes.

Julien Issartel : Gestion du moteur physique et de la synchronisation des fenêtres.

Gabrielle Luypaerts : Gestion du l'XML. Création fichier, Ouverture/lecture, chargement ...

Les programmeurs principaux sont : Julien Issartel, Arnaud Casella, Vincent Hurbourque et Gabrielle Luypaerts.

3.2 Communication au sein du groupe

La communication sur des projets aussi important que celui-ci est très importante. Nous étions six sur ce projet, il était donc constamment en modification. Pour ne pas être complètement perdu dans l'avancement du projet, nous devions nous tenir au courant des avancés des uns et des autres. Nous nous sommes servi de trois canaux de diffusions principalement, les réunions, d'un googlegroup, et de mercuriale.

3.2.1 Réunions

Tout au long de la mise en place de notre projet nous avons effectué plusieurs réunions. Elles sont devenues très courantes à la rentrée, car durant les vacances, nous ne pouvions nous voir physiquement pour des raisons de distances. Nous nous réunissions à l'échelle d'une réunion physique minimum par semaine.
Nous avons également fait quasi-quotidiennement des réunions vsur Internet en utilisant des messageries instantanées ou encore des logicielles de VoIP. Les informations étant ensuite transmises aux autres personnes par email.

3.2.2 GoogleGroups

Pour permettre la diffusion des informations au sein de notre équipe de développement nous avons mis en place un système de liste d'envoie googlegroups. Nous avions pour adresse *(smashstein@googlegroups.com)*. De cette manière tous le monde était informé en temps et en heure des modifications et des difficultés que rencontraient chacun.

3.2.3 Mercuriale

Notre équipe est composée de six personnes et cela risque de poser problème quand à l'organisation du programme. En effet, en faisant un projet à un nombre élevé de personne, il est possible qu'il y ait des problèmes quand au rassemblement des avancées de chacun dans le programme. Pour éviter ces problèmes, nous avons mis en place un Mercurial.
Mercurial est un logiciel de gestion de version. Le but de celui ci est de pouvoir faire travailler plusieurs développeurs sur un même ensemble de fichiers sans qu'il y ait besoin d'une personne qui s'occupe d'assembler le travail de tout le monde. Chacun ajoute sa partie et tout le monde en profite.

Le projet est, pour cela, stocké sur internet, dans un dépôt ou repository – ici on utilisera le site intuxication.org pour héberger notre projet. Puis, Mercurial va communiquer avec le serveur du site en question pour lui envoyer ou recevoir les mises à jour des fichiers du projet en cours de traitement, et ce, grâce à certaines commandes utilisées.

Malgré quelques petits problèmes quant à sa mise en place, ce procédé nous a évité de nombreuses heures fastidieuses à rassembler toutes les différentes parties de chacun et de les compiler. Cependant nous sommes conscient du gain de temps dont nous a fait profiter l'utilisation de ce logiciel simple et gratuit.

4 Architecture et Structure du Projet

4.1 UML

4.1.1 DSS

Nous avons réalisé ce diagramme de séquence pour bien identifier les interactions des objets entre eux.

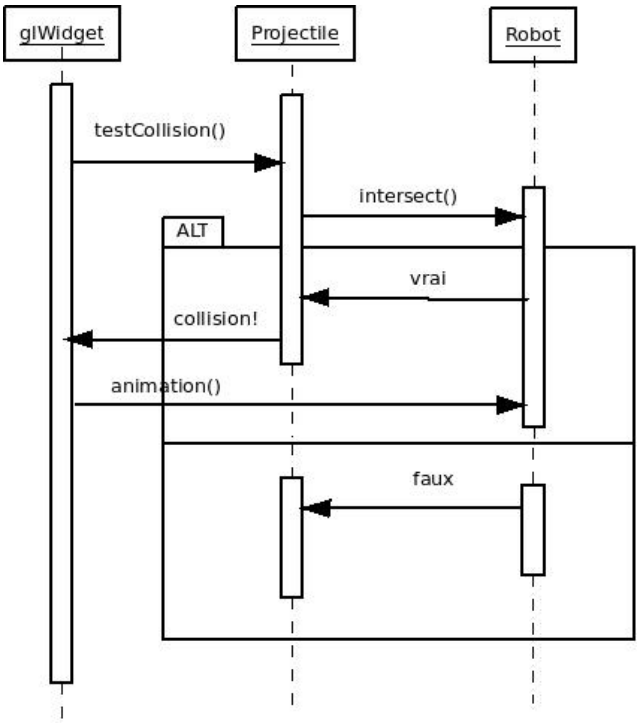

Figure 1 – Diagramme de Séquence

4.1.2 Use Case

Voici les Use Case que nous avons réalisé.

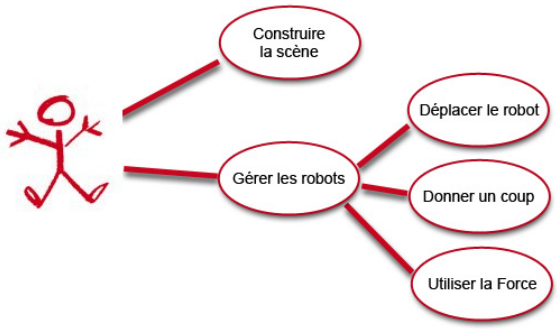

FIGURE 2 – Use Case 1

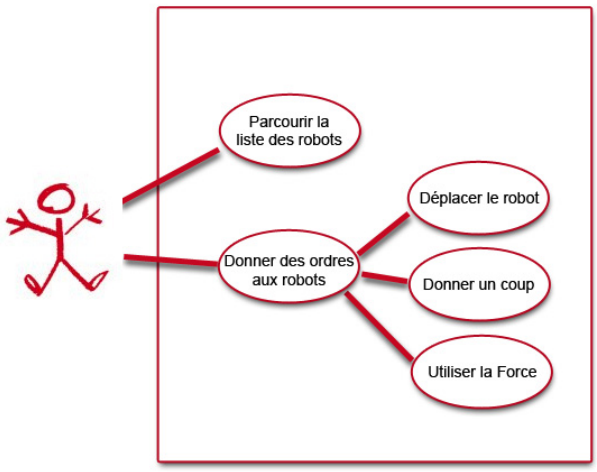

FIGURE 3 – Use Case 2

4.1.3 Diagramme de Classe

Le diagramme de classe de notre projet JediSmash se trouve sur la page suivante (page 15).

4.2 Design Pattern Utilisées

4.2.1 Singleton

Le programme doit assurer l'unicité de l'instance qui va lancer le jeu, en effet, on doit interdire de pouvoir instancier plusieurs fois l'objet responsable du lancement du jeu. On utilisera, donc, dans la classe *Application.cpp*, une instance Application qui ne sera accessible qu'à partir d'une méthode *static getApplication()*. Cette fonction servira à gérer l'accès au constructeur qui ne doit pas être accessible de l'extérieur. Notre idée de départ était de rendre unique l'avatar contrôlé par le joueur mais anticipant l'ajout d'un deuxième joueur, nous avons préféré laisser cette idée de côté.

4.2.2 Stratégie

Le principe des Stratégies est à la base du concept de généricité de la classe Robot. La partie sur l'IA explique que l'on va envoyer à chaque bot des ordres. Au début nous gérions les ordres depuis la scène :

- on parcourt la liste de robots
- pour chacun, il y a un test sur l'ordre reçu
- on effectue une action adaptée

Avec l'implémentation de robots de différentes natures, il aurait fallu tester également le type de robot pour savoir comment agir. Plutôt que de faire cela, on va définir dans l'interface *Robot*, une méthode *traiterOrdre* qui prend en paramètre l'ordre. Ainsi, par polymorphisme, pour chaque élément de la liste de Robot, qu'il s'agisse d'un type DumbRobot ou Jedi, l'action *traiterOrdre* sera traitée indépendamment.

4.2.3 Observateur

À chaque frame et pour chaque projectile, une fonction *testCollision()* sera appelée. En testant l'intersection entre la bounding box du projectile et celle du personnage testé, on peut détecter le fait qu'un projectile frappe un protagoniste. Dans ce cas précis, il va falloir augmenter la barre de dégâts du robot touché puis, pour lancer l'animation, informer les 4 *glWidgets*. Pour

FIGURE 4 – Diagramme de Classe

cela, on met en place un système d'observateur :

- chaque projectile possède une liste d'observateur (les glWidgets)
- à chaque collision, on calcule les dégâts infligés à la cible
- on appelle une fonction *animHit()* pour chaque observateur

4.3 Interactions Logicielles

4.3.1 Les Robots

Elle contient une liste de pointeurs sur *DumbBot*, qui représente les bots de la scène et qui va être initialisée avec la fonction *populate()*. La fonction *populate()* va également associer à chaque plug-in un groupe de *DumBot*, ce vecteur de pair est l'attribut *botsPlugins*. Enfin, cette fonction va lire et chercher dans le .so la fonction *act()* qui va définir le comportement du bot, on aura ainsi, stockée dans *botsPluginsAct*, une liste de comportements. En toute logique, le nième groupe de bots stocké dans *botsPlugins* doit adopter le comportement du nième élément de la liste *botsPluginsAct*. C'est ainsi que les fonctions *act()* des plugins vont être appelées à travers la méthode *act()* de la classe Robots : pour chaque i de 0 à n, on appelle *botsPluginsAct[i]* en lui passant *botsPlugins[i].second* (le groupe de bot associé).

4.3.2 Robot

Robot se situe dans le package openkn : :smashstein et définit un certain nombre d'accesseurs aux paramètres suivants :

- robotState
- speaker
- radarReport
- radarRange
- maxSpeed
- weight
- gunFrequency
- gunPower
- gunProjectileVelocity
- getBoundingBox

NB : *robotState*, speaker et radarReport sont respectivement des objets *RobotState, Speaker et RadarReport* et seront décrits plus bas.

4.3.3 Radar Report

La classe DumbRadarReport, implémentant l'interface RadarReport, est faite pour gérer la visibilité de chaque bot. Les methodes getNearbyRobots, getNearbyProjectile et getNearbyObstacle vont respectivement renvoyer un vecteur de Robot, de Projectile et d'Obstacle correspondant aux entités visibles du bot passé en paramètre. Il était possible pour nous de gérer la visibilité d'un bot soit directement dans le radarReport soit dans l'IA (il suffit de sélectionner, à partir des bots envoyés par radarReport, ceux qui sont dans la zone de visibilité), or comme d'un type de bot à l'autre, la seule chose qui changerait pour son radarReport est sa visibilité, nous avons décidé de calculer celle-ci dans l'IA et ainsi factoriser notre code avec un seul et même type de radarReport pour n'importe quel Robot.

4.3.4 Speaker

Cette classe va contenir un couple chaîne de caractères/flottant que l'on appelle un ordre, c'est la fonction act() du plugin qui va envoyer les ordres en fonction du contexte et c'est dans GLWidget : :paintGL qu'il sera traité. Là encore, nous avons un seul type de speaker.

4.3.5 Robot State

Cette classe contient un paramètre velocity, un vecteur position et un vecteur direction. Ces informations sont utilisées dans le GLWidget pour, selon l'ordre passé dans le Speaker, modifier la position du bot. Ex : s'il l'ordre est d'avancer, on calcule la position vers laquelle il faut le translater avec sa direction et sa vitesse de déplacement (velocity). C'est également ici que sera gérée la gravité des objets mobiles de la scène.

5 Fonctionnalités du Programme

5.1 Interface Graphique

5.1.1 QT

Qt est une bibliothèque logicielle (en C++) qui offre de nombreux composants dont celui d'affichage graphique (dont nous nous sommes servis ici). L'environnent graphique KDE, utilisé par défaut par Mandriva ou Kubuntu, est notamment basé sur la bibliothèque logicielle Qt.
Nous nous sommes servis de cette dernière comme interface, autant graphique que physique (gestion Clavier-Souris).
La gestion du clavier et de la souris se fait de façon transparente par rapport à Opengl et fonctionne de la même manière que sous OpenGL. Qt « écoute » le clavier et transforme le message en ordre que l'on peut récupérer sous OpenGL.

Comme nous avons vu ci-dessus, Il est possible d'afficher de L'OpenGL sur Qt. C'est une des grosses raisons pour laquelle nous avons utilisé le framework Qt. Comme nous devions séparer notre fenêtre en quatre affichages OpenGL indépendants, nous avons utilisé le système de Widget. Un widget correspondant à un affichage indépendant sous Qt, nous en avons fait quatre, et dans chacun d'entre eux, nous avons affiché les fenêtres OpenGL désirées.

Si nous voulions créer quatre fenêtres pour gérer quatre caméras, nous voulions aussi afficher la même scène. Or, chaque Widget contient son propre code indépendant. Nous avions donc quatre affichages représentant quatre fois la scène, mais avec les mêmes personnages se déplaçant différemment (les personnages utilisent de l'aléatoire pour agir).
Pour afficher les personnages de la même façon dans les quatres widgets, nous avons donc déclaré les pointeurs du personnage principal ainsi que robots(la liste des bots de l'IA) dans le fichier instanciant les widgets (mainWindows.cpp), à qui nous avons passé en paramètre ces pointeurs.

Une fois ceci fait, lorsqu'on modifie la position d'un personnage dans un widget, on le fait pour tous les widgets (puisqu'ils partagent le même pointeur). Nous n'avons ainsi écrit certains codes qu'une seule fois, dans widget.cpp (widget de débuggage), notamment ceux déterminant la position des personnages.

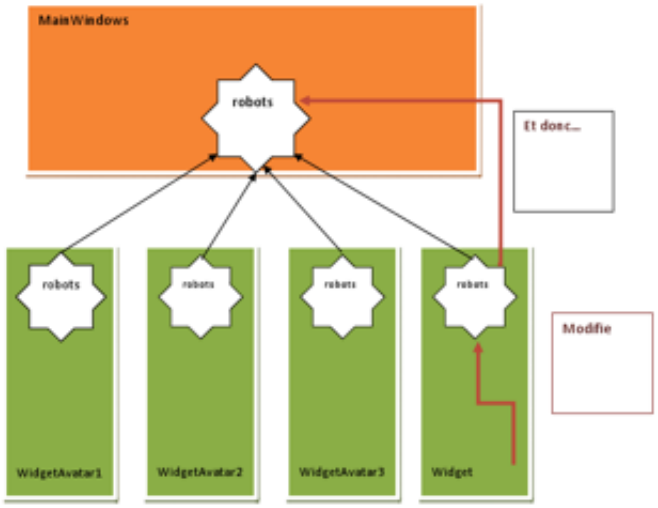

FIGURE 5 – Gestion des widgets

5.1.2 Caméra

Nous devions organiser chaque affichage de la façon suivante :
- Un devait représenter le héros de la scène, et le suivre tout le temps
- Un devait suivre un des bots de l'IA
- Un devait représenter la scène dans son ensemble
- Un dernier devait pouvoir permettre de se déplacer librement sur la scène pour le débuggage (déjà implémenté de base)

Pour chacune des trois premières caméras, nous avons tout simplement adapté le GluLookAt en fonction de ce que nous avions à afficher. Les deux premiers visent et se déplacent en fonction de la cible désirée (et proposent un plan rapproché). Pour l'affichage des bots, il est possible de switcher entre eux en appuyant sur + ou -. Si nous avons le temps, nous afficherons la scène globale en fonction des deux personnages de la scène les plus éloignés en x.

5.2 I.A

Avant tout il faut savoir que nous disposons d'une liste de robots qui sera transmise à chaque bot par son radar. L'enjeu de l'IA va être de rejoindre ou de fuir ses ennemis selon le profil (team à laquelle il appartient). Pour cela on va tester dans la liste de robots celui qui est le plus proche.

5.2.1 IA Smart

Nous construirons l'IA des bots en fonction du fichier XML du décor. Le fichier *XML* (décrit plus bas dans le rapport) contiendra les différents objets selon les niveaux. Chaque objet sera lié à d'autre s'il est accessible. Exemple :
<objet 1 niveau='1' posX='10' posY='10'>
<objet 2 niveau='2' saut='1'/>
<objet 3 niveau='3' saut='2'/>
</objet>

Ainsi, comme nous le voyons sur la FIGURE 6 ci- dessous, un bot pourra aller de l'objet 1 à 2 s'il possède une capacité de saut de 1 et pourra aller vers l'objet 3 si il possède une capacité de 2.Nous bloquerons la capacité de saut d'un bot au nombre 2.Il ne pourra ainsi pas sauter à des hauteurs démentielles. Voyons cela sur le schéma suivant.

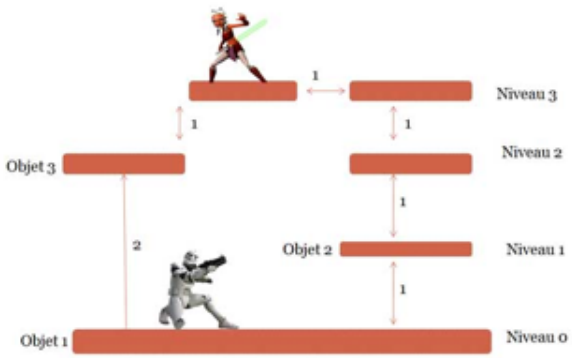

FIGURE 6 – Diagramme de Classe

Le bot peut se trouver dans 2 positions par rapport au bloc sur lequel il doit sauter :

- En dessous du bloc
- Eloigné du bloc

Comment agir pour que le saut soit fluide ? Et non avoir ce schéma de saut (FIGURE 6). Nous connaitrons :

- La hauteur et la largeur de l'objet
- La position de l'objet
- La position du bot

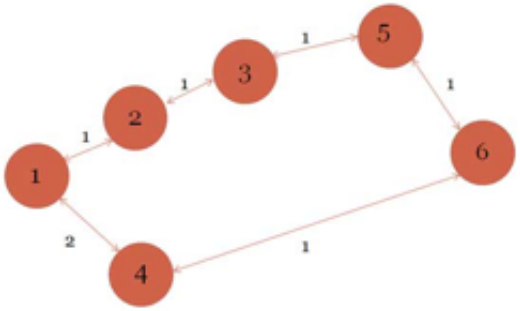

FIGURE 7 – Graphe Correspondant

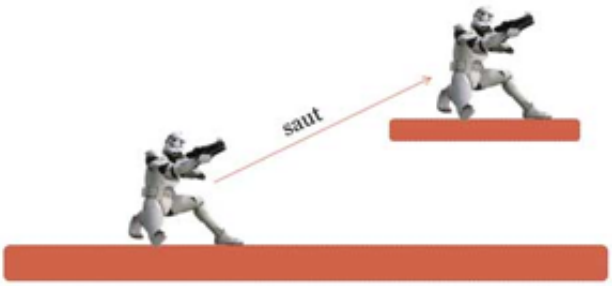

FIGURE 8 – Saut en ligne droite

Si le bot est loin, il suivra ce schéma (FIGURE 9)

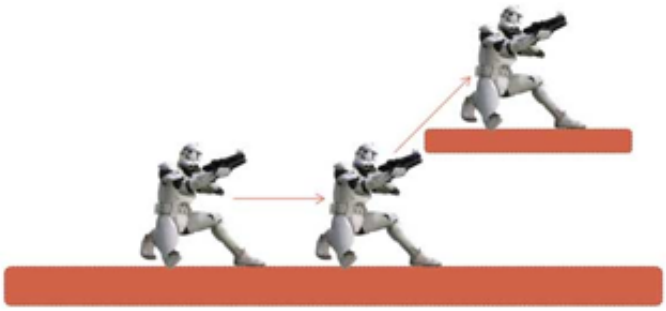

FIGURE 9 – Rapprochement du bot avant le saut

Voici un pseudo du mouvement du bot :

Si (Xd < X0) alors
Avancer bot tant que Xd< X0 – width/2 – epsilon puis
Sauter en (Xo – width/2 + gamma1 , Yo +gamma2)
Sinon si (Xd > X0) alors
Avancer bot tant que Xd > X0 + width/2 + epsilon puis
Sauter en (Xo + width/2 - gamma1 , Yo +gamma2)
Sinon si (Xd > Xo-width/2 et Xd < X0+width/2)
//saute droit
Yd = Y0 +gamma2

En revanche si le bot est sous la plate forme, il pourra passé a travers, mais uniquement dans un sens, lorsqu'il saute dessus. (FIGURE 10)

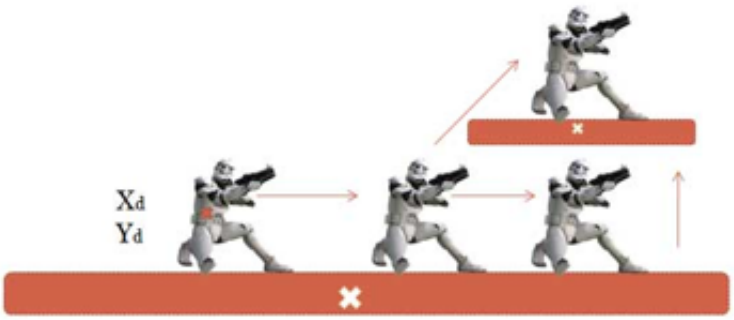

FIGURE 10 – Rapprochement du bot avant le saut

Il est à savoir que ce déplacement sera simulé par l'ordre *RunAndJump* dans notre programme. Cet ordre sera au moment ou Xo + width/2.

5.2.2 IA Bot

L'intelligence des bots sera assez simpliste. On pourra avoir 2 types d' IA simples de bots :
- Corps à corps
- Tireur

Les bots essaieront toujours de se rapprocher de l'avatar afin de l'attaquer. Si le bot a un profil corps à corps alors il essaiera d'atteindre sa position en

x sinon si c'est un profil tireur, on définit une distance minimale à atteindre avant de tirer. Il faut aussi regarder à quel niveau se trouve l'avatar, si l'avatar se trouve à un niveau supérieur alors le bot devra atteindre la plateforme la plus proche et monter dessus. Il essaiera de monter au plus haut en sautant sur les plateformes autour de lui.

5.2.3 IA Victime

La victime doit fuir les ennemis afin de ne pas être tapée. Il est en perpétuel mouvement. Si un bot ennemi l'approche de trop près alors la victime doit s'écarter de la position. Elle dispose d'un radar continu, et essaye de rester toujours au plus loin des ennemis. Si une plateforme est libre alors la victime sautera dessus.
Si la victime est attaquée alors elle essaiera de se défendre via son attaque. Il faut noter aussi que si la victime et l'avatar sont sur le même niveau alors la victime ira vers l'avatar pour être protégé.

5.3 Animation

5.3.1 Modélisation

Dans notre jeu, l'univers choisi est celui de Star Wars. Il nous a ainsi semblé obligatoire le fait que nos personnages aient un rapport avec la célèbre sixtilogie. Nous avons décidé des personnages suivants :

- L'avatar sera un Jedi, l'ennemi le plus intelligent sera un Seigneur Sith, les soldats seront des Clones Trooper et enfin, le personnage que doit protéger l'avatar sera un robot de type R2D2.

Pour ce faire, avec le temps dont nous disposions, nous avons pris sur internet des bases de model de 3DS Max et nous les avons modifiés à nos fins. Pour la scène, nous avons choisi de modéliser une scène dans l'espace représentant des morceaux d'astéroïdes. (FIGURE 11)

5.3.2 Texture

Comme nos modèles seront exportés au format *md2*, il faut gérer leurs textures d'une certaine façon. En effet, le format *md2* n'incorpore pas les textures directement. Il faut donc utiliser de l'UVW Mapping. Cette technique consiste à plaquer une image 2D sur un objet en 3D. Il faut donc décomposer,

déplier, notre modèle en face sur un plan 2D et ainsi pouvoir appliquer les textures que l'ont veut sur notre modèle (FIGURE 12 et 13).

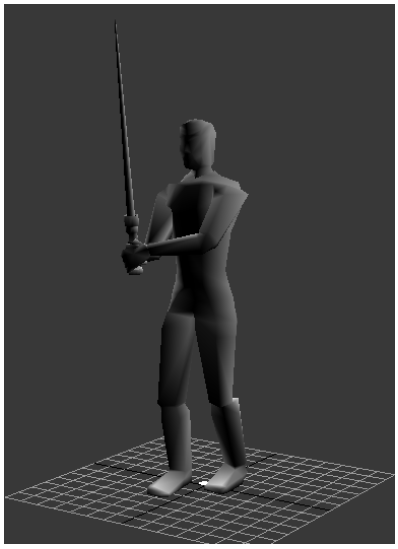

FIGURE 11 – Modelisation du Jedi sans Texture

Sur la FIGURE 12 et 13 ci-dessous, nous avons sélectionné toutes les faces du bras gauche et nous les avons bien placés sur l'image. Ainsi, nous avons la texture du bras associée au modèle.

Nous veillerons à n'utiliser qu'une seule image 2D avec toutes nos textures dessus car, dans le chargement de textures dans le programme, nous ne devons avoir qu'une seule image. Voici un aperçu de la scène et la réalisation final de notre bot (FIGURE 14 et 15).

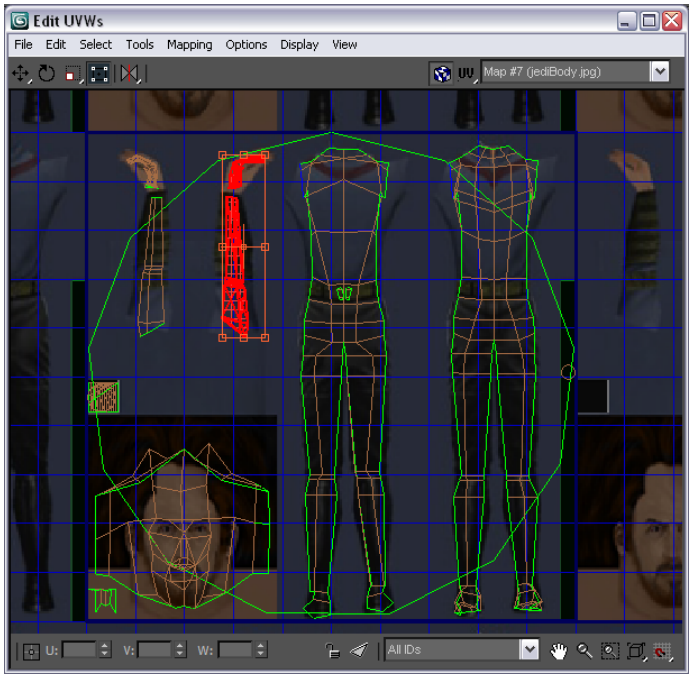

FIGURE 12 – Textures et UVW Mapping

FIGURE 13 – Textures appliquées au personnage

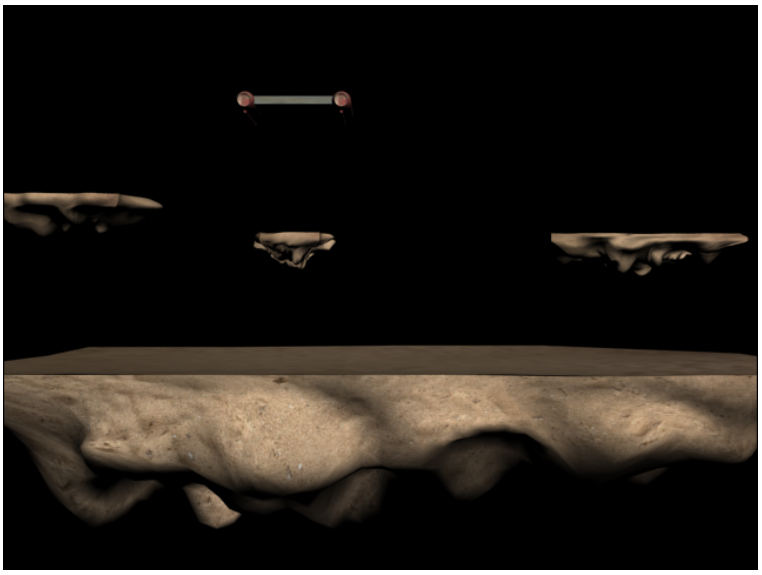

FIGURE 14 – Scène de notre jeu

FIGURE 15 – Bot ennemis

5.3.3 Animation

Une fois nos modèles finis et les textures appliquées, nous devons les animer. Chaque personnage possède un nombre d'animations qui lui est défini selon les différents états qu'il peut avoir dans le programme.
Pour animer un modèle, il faut lui incorporer un squelette et l'attacher ensuite au modèle. C'est ce squelette que nous animons. Ainsi, nos modèles pourront effectuer parfaitement leurs mouvements.
Les personnages ont des mouvements communs selon leurs états. Les attaques dépendent elles du personnage.

Les états (communs) :
- Stand : le personnage est à l'arrêt
- Run : le personnage cours dans la scène
- Jump : lorsqu'il saute
- Fall : lorsque le personnage se fait attaquer, il tombe au sol est se relève.

Attaques (particulier selon les personnages) :
- Jedi et Sith
  - Kick : il donne un coup de pied de type mawachigueri
  - Saber Kick : il donne un coup de sabre laser
  - Force : il utilise la force pour projeter son ennemi

- Troopers
  - Kick : le trooper donne un coup de blaster à son ennemi
  - Shot : le trooper tire avec son blaster

- R2D2
  - Kick : il donne un coup de tête.

Il est bien entendu que les animations communes sont particulières aux personnages.

5.3.4 Exportation

Une fois les animations réalisées, nos modèles sont fin prêts pour l'exportation au format *md2*. Le format MD2 est un format de fichier contenant les données des modèles 3D de Quake II. Nous l'avons choisi car il est simple à incorporer dans notre programme et intéressant à utiliser. Il contient des données géométriques, comme les polygones ou les coordonnées de textures, mais aussi l'animation du modèle frame par frame. Il n'est pas un format courant

pour 3DS Max, il a donc fallu se procurer un script disponible sur Internet. Pour l'exportation au format *md2*, il a fallu respecter certains critères :

- des modèles basse résolution pour pouvoir les exporter facilement
- des modèles unis en maillage éditable.
- texture UV Mapping avec une seule image 2D

5.3.5 Chargement Texture/MD2

Une fois que nous avons nos différentes animations aux formats *md2*, il faut les incorporer dans la scène. Nous devons tout d'abord les charger dans notre programme. Pour cela, nous utilisons le chargeur de *md2* fourni avec le squelette du programme. Nous créons des objets *md2* dans nos GLWidget, puis nous les chargeons à l'aide de la méthode *LoadModel* qui prend en paramètre le chemin du *md2*.
Après, il faut charger les textures du modèle. Pour ce faire, nous avons converti notre image 2D qui était à l'origine au format *JPEG*, au format *PPM*. Ainsi, nous avons utilisé un chargeur d'image *LoadPPMRGB*, puis nous avons chargé les textures du modèle, en utilisant la fonction *LoadTexture* qui permet dans un premier temps, de charger l'image ppm en texture OpenGL, et dans un deuxième temps, d'attribuer cette texture au modèle *md2*.

5.3.6 Incorporation dans la scène

Pour une meilleure gestion des animations en fonction des ordres des robots et avatars, nous avons choisi de les centraliser dans une classe pour chaque personnage. Chaque classe de personnage possède des pointeurs vers ses *md2*, sa texture ainsi que les sons joués.
Il y a des méthodes qui permettent ainsi de jouer les animations des robots en utilisant la fonction *renderFrame()*. Cette méthode consiste à jouer le numéro de la frame, passé en argument du *md2*. Ainsi, il suffit de faire une boucle et d'incrémenter notre indice pour avoir une animation fluide. On arrête ou recommence l'animation, en fonction de l'animation, lorsque c'est la fin de celle-ci.

5.4 Son

Pour immerger le joueur d'avantage le joueur dans le jeu, nous avons mis en place une librairie pour lire des fichiers sonores. De cette manière, nous avons une musique de fond. Sur celle ci vient s'ajouter d'autres effets sonores en fonction des actions et conséquences des actions de chacun tels que les sons de tirs, de chutes, ...

5.4.1 Librairie

Notre choix de librairie c'est porté sur la SDL. En effet cette librairie est très complète et nous permettait donc de répondre à nos besoins. Elle est capable de jouer une musique et d'iphone Ca prise en main n'a pas été très difficile. Nous devions simplement importer les bons *includes* et récupérer les syntaxes pour implémenter le chargement et la lecture des sons.

5.4.2 Incorporation dans le code

A l'heure actuelle, le son n'est pas encore intégré au projet. Lorsque ce sera fait, les sons seront joués en même temps que l'animation. C'est à dire que l'action du personnage sera lancé lors d'un évènement clavier. La méthode alors appelé appellera à son tour la méthode permettant de jouer le son.

5.5 Moteur Physique

5.5.1 Déplacement

Sur la scène, les robots sont soit des bots (donc contrôlés par l'ordinateur), soit le personnage principal (contrôlé par le joueur). Si la façon de leur donner des ordres est différente, en revanche, ils agissent de la même manière. Comme précédemment dit, chaque robot écoute les ordres qu'on lui a donné (soit par le clavier, soit par l'IA), et agit en conséquence.

Courir : Tous les robots peuvent courir. Comme nous sommes sur une scène en 2D, faire courir des Bots correspond à déplacer leur position en x dans un sens ou l'autre grâce au clavier (et afficher l'animation correspondante). Il suffit pour cela d'ajouter à la position en x du robot sa vitesse (plus ou moins grande selon ce qui est désiré).
Lorsque le robot s'arrête de courir, on observe un effet d'inertie. En effet, au lieu de mettre la vitesse du bot instantanément à zéro, nous l'avons rapide-

ment diminuée pour donner une impression plus « réaliste ».

Sauter : Nous avons aussi fait sauter les robots. Pour le robot principal, nous lui avons appliqué une gravité (qui lui est propre, et que nous n'utilisons pas en respectant les lois de la physique), et qui s'applique constamment à lui (ce qui le fait naturellement redescendre). A tout moment, le robot teste son environnement pour détecter si des collisions ont lieu entre lui et un objet inférieur [1]. Lorsque c'est le cas, la gravité du robot s'annule. En revanche, dès qu'il n'observe plus de collision de haut en bas, le robot se remet à tomber. Maintenant, il s'agit de faire sauter le robot. Pour ce faire, nous avons détecté si le robot était en train de faire un saut (par un booléen). Si ce n'est pas le cas, alors nous avons ajouté une notion d'impulsion au robot, qui s'ajoute à la gravité du robot, et la rend positive, ce qui fait que le robot monte. Cette impulsion, d'abord positive, est décrémentée ce qui fait que plus le robot monte, moins il va vite. Ensuite, sa gravité devient nulle, et ensuite négative, ce qui refait tomber le robot, de plus en plus vite (car la gravité continue à être décrémentée). Comme vu précédemment, la chute du robot correspond au moment où il détecte un objet sous lui.

Quand aux robots de L'IA, ils réaliseront un saut d'une manière différente. Au jour où nous écrivons, ceci n'est pas encore implémenté mais non espérons le faire. Pour rejoindre le personnage principal perché en hauteur, les robots devront monter sur des plateformes prédéfinies. Pour ce faire, il faut qu'ils réussissent leur saut. Donc nous avons déterminé qu'ils devraient être guidés, d'un point A de l'étage où ils se trouvent, au point B de l'étage supérieur. Le saut serait donc parabolique : en fonction de la position de A et B (qui sont des positions clefs définies en fonction des plateformes), le robot calculerait ses positions tout le long de la parabole de A à B, en x et en y. Si nous comptons implémenter cette idée, c'est parce qu'ainsi le saut est totalement maitrisé et n'est pas influencé par une vitesse initiale trop importante (ou pas assez).
Le résultat sera probablement moins réaliste, mais il fonctionnera. De plus, la structure même du graphe facilite cette implémentation, puisque les positions clefs des sauts correspondraient mentalement aux nœuds du graphe de l'IA. Nous avons vu que nous avons utilisé les boites englobantes pour gérer les collisions, nous allons vous en parler un peu mieux.

1. pour savoir si la collision a lieu avec un objet inférieur, il suffit de détecter si la position en y du robot est plus proche du Y supérieur de la boite englobante que de son Y inférieur

5.5.2 Collision (Bounding Box)

Nous avons géré les collisions entre les personnages (personnage joueur et bots utilisant une AI), objets (sol, plateformes) et missiles. Le sol de la scène est un objet aussi, puisque nous avions besoin de lui pour détecter des collisions (avec les robots surtout). Expliquons précisément ce qu'est une boite englobante.

Pour gérer les collisions de manière correcte il fallait prendre la surface globale d'on objet. En effet prendre les coordonnées de nos objets ainsi que la largeur et la hauteur ne suffisait pas. C'est là que les boites englobantes interviennent. Via le modèle md2 et ses vertices nous avons pu tracer un carré englobant l'objet. Prenons un exemple :

Ces boîtes ont plusieurs objectifs :

- détection des collisions missiles/bots et missiles/avatar
- détection robots et avatar avec le sol (qui est un modèle md2) ou objet qui permet de gérer le moteur physique comme expliqué précédemment

Voyons plus précisément comment nous avons géré les missiles.

5.5.3 Projectile

Un projectile est un type d'objet qui contient aussi sa boîte englobante. Il contient aussi sa position sa direction : tout ce qui peut lui permettre de se déplacer. Enfin, il contient aussi une variable représentant le nombre de dégâts qu'il peut occasionner.
A tout moment, il fait un test de collision voir si il ne détecte pas d'objet. Lorsque c'est le cas, il disparaît. S'il rencontre un objet « robot », celui-ci voit sa barre de dommages augmenter en conséquence, et subit un déplacement dû à l'impact proportionnel à son pourcentage de dégâts.

5.6 XML

Le chargement de l'arène se fait au lancement du programme. Il faut ensuite lui transmettre tout ce qui y est présent. Toutes les informations concernant l'arène sont contenu dans un fichier *XML*. Ce fichier renseigne sur le nombre d'objets qui sont présents. Il contient la position des niveau et leur taille, ainsi que le nombre d'objets accessibles depuis celui-ci avec le nombre de saut qu'il faut pour y accéder, ainsi que le nombre de saut nécessaire pour passer d'un objet à un autre.
Par exemple : Pour passer de la plate-forme numéro 0 situé au niveau 1, à plate-forme numéro 2 situé au niveau 3, il faudra passer par la plate-forme 1 se trouvant au niveau 2. Il a donc fallu implémenter une fonction qui permettais de lire dans un fichier *XML*.

5.6.1 Bibliothèque

5.6.2 Structure

Organisation des fichiers XML *Arena.xml* :

```
<arena nbObj="3">
  <environment background="background.png" music="music.mp3" />
   <avatar>
        <name>Bobby</name>
        <plugin>plugin1</plugin>
        <position x="0" y="0" z="0" />
        <dimensions width="5" height="2" />
</avatar>
<bot>
        <name>Bot1</name>
        <brain>libBrain.so</brain>
        <body velocity="1.0" resistance="1.0" />
        <position x="4" y="3" />
        <model nom="trooper" />
</bot>
<objet>
        <nom>Sol</nom>
        <id>0</id>
        <position x="0" y="0" />
        <dimensions width="20" height="3" />
        <model nom="sol" />
</objet>
</arena>
```

Le fichier xml *arena.xml* contient chaque objet présent dans la scène. La balise générale, *<arena>*, englobe une balise *<environment>* qui nous renseigne sur les éléments du décor comme l'image de fond (attribut *background*) et la musique d'ambiance (attribut *music*).
Une autre balise, *<avatar>*, possède les informations du personnage joué par l'utilisateur : son nom, le plugin utilisé, sa position initiale et ses dimensions.
Une balise <robot> représentera chaque robot ennemi placé dans la scène, avec pour données son nom, la bibliothèque correspondant à son intelligence artificielle, des caractéristiques physiques comme sa vitesse et sa résistance, et le nom du fichier *md2* qui le représentera.
Enfin, le décor étant constitué de plateformes, à chacune d'entre elles est attribuée une balise <objet> qui nous renseigne sur le nom de l'objet en question, son identifiant, sa position dans le décor, ses dimensions et, une fois encore, le modèle md2 qui lui sera rattaché.

Organisation du fichier XML *Modeles.xml* :

```
<model>
    <avatar>
            <stand>stand.md2</stand>
            <run>run.md2</run>
            <fall>fall.md2</fall>
            <kick>kick.md2</kick>
            <saber>saber.md2</saber>
            <force>force.md2</force>
            <jump>jump.md2</jump>
</avatar>
<bot1>
          <stand>standBot1.md2</stand>
          <run>runBot1.md2</run>
          <fall>fallBot1.md2</fall>
          <kick>kickBot1.md2</kick>
          <saber>saberBot1.md2</saber>
          <force>forceBot1.md2</force>
          <jump>jumpBot1.md2</jump>
  </bot1>
<objet>
           <sol>sol.md2</sol>
          <asteroide>asteroide.md2</asteroide>
          <platform>platform.md2</platform>
 </objet>
</model>
```

Le fichier modeles.xml regroupe tous les modèles *md2* qui seront utilises dans le jeu. Les modèles peuvent être de trois types : avatar, robot ou objet. La balise *<avatar>* regroupe les modèles correspondant aux sept actions que peut effectuer le personnage principal : être immobile, courir, tomber, donner un coup, utiliser le sabre laser, utiliser la Force, sauter.
Ces mêmes actions se retrouveront au sein des différentes balises filles de *<bot>*. Celle-ci correspond à la catégorie des robots, et chaque type de robot a une apparence différente *(troopers, sith, etc.)*.
Enfin, la balise <objet> renferme les trois catégories de plate-forme que l'on peut trouver dans la scène : le sol, un astéroïde et une plate-forme.

5.6.3 Chargement et Lecture

Pour effectuer le chargement des fichiers xml, nous avons utilisé la librairie *TInyXML*, qui permet un accès facile à tous les niveaux d'information, grâce, entre autres, aux classes *TiXmlNode*, qui représente un nœud de l'arbre, *TiXmlElement*, qui correspond à un élément, et *TiXmlAttribute*, pour les attributs de ces éléments.
Afin d'utiliser les données contenues dans les fichiers xml, il est nécessaire de parcourir leur arborescence.
Un objet de type *TiXmlNode* est créé, qui correspond à la racine de l'arbre. Un objet *TiXmlElement* est créé pour nous permettre de nous déplacer et de récupérer les informations. En partant d'un nœud, il est possible d'obtenir l'élément qui y est associé grâce à la fonction *ToElement()*. Depuis un élément, il est possible de parcourir ses fils grâce à la méthode *FirstChildElement()*, ou ses frères avec *NextSiblingElement()*.
Une fois que nous nous sommes placés sur l'élément à connaitre, il nous est possible de récupérer soit la valeur de ses attributs avec la méthode Attribute(), soit le texte compris entre la balise en question avec *GetText()*. Une fois arrivé à une feuille de l'arbre, il est possible de repartir de la racine afin d'explorer une nouvelle branche de la même manière.

Chargement des modèles *md2* :
Dans un premier temps, tous les modèles md2 sont chargés à partir du fichier *modeles.xml*, et stockés dans des tables de hachage, au sein des classes appropriées. Ces tables permettent d'accéder aux modèles directement par leur nom, sans avoir besoin d'un identifiant numérique, qui serait moins clair. La classe Avatar contient la table des sept modèles *md2* vus précédemment, la classe Robots celle de tous les modèles possible pour les robots, et la classe objet, la table stockant les trois modèles correspondant aux plate-formes.

Chargement et construction des objets de la scène :
Une fois les modèles *md2* chargés et stockés, chaque objet de la scène est créé grâce à une fonction spécifique du fichier *loadXML.cpp*. La fonction *loadAvatar* permet de créer l'avatar en remplissant les attributs de la classe avec les valeurs du fichier xml. De même, la fonction *loadRobots* remplit la liste de robots et *loadObjets* remplit la liste d'objets.

6 MANUEL DE JEU

Les commandes de jeu Dans notre jeu, nous avons opté pour un gameplay plutôt pratique. En effet, l'utilisateur disposera des touches directionnelles pour déplacer l'avatar dans la scène. Les commandes d'attaques seront côtes à côtes :

- a : coup de pied
- z : force
- e : coup de sabre laser

Il pourra sauter avec la barre d'espace.

Concernant le jeu : Pour blesser un ennemi il faut lancer des attaques(coup de pied, force ou coup de sabre laser). Si celui-ci est touché, cela fera augmenter la barre de dommage d'un ennemi et l'expulsera plus ou moins loin en fonction du cumul des dommages infligés. Plus un personnage aura encaissé de coups, plus sa barre de dommage sera élevée et plus la position de celui-ci sera affectée au prochaine impact. Un personnage meurt s'il est expulsé de l'arène.

But du jeu : Survivre dans l'arène et protéger son cohéquipier. Le jeu prend fin si tout les ennemis sont hors de l'arène ou encore si l'avatar ou le personnage à protéger est mort.

7 BILAN

7.1 Récapitulatif des fonctionnalitées

Récapitulatif des fonctions implémentées :

- gestion du gameplay par équipe (mode coopératif avec l'ordinateur)
- affichage synchronisé de 4 fenêtres QT possèdant diverse caméras
- animations visuelles et sonores des peronnages grâce aux models 3D md2
- gestion des textures pour les modèles md2
- construction d'une scène à partir d'un fichier xml (plateformes+décor)
- implémentation d'un moteur physique gérant la gravité
- implémentation de différents IA compilés en binaire pour chaque type de personnage
- recherche de plus court chemin, pathfinding, dans l'IA du jedi noir, pour rejoindre son ennemi
- gestion des collisions, grâce aux bounding box
- gestion d'un système de vie

Fonction que nous avions prévu d'implémenter :
- lanceur aléatoir d'animation pour une même action (pour ne pas que les coups soit répétitifs)
- ajouter d'ordre eventuelle pour pouvoir courir puis sauter
- agrandissement et switch des différentes vues
- affichage et gestion d'un score
- affichage des barres de dégâts
- ajout d'un menu de départ
- choix d'une arène et d'un modèle

7.2 Problème rencontrés

Exportation des modèles *md2* Lors de l'élaboration des modèles md2, nous avons rencontré quelques difficultés. En effet, nous faisions des modèles qui n'étaient pas assez optimisés pour l'utilisation que nous en avions. Du coup, l'exportation ne se faisait pas ou était trop longue. Nous avons opté par la suite pour des modèles de qualité moindre.

Architecture Nous avons eu quelques problèmes d'adaptation à l'architecture déjà mise en place. Nous nous sommes rendu compte qu'elle n'était pas totalement adaptée à notre vision du projet. En analysant bien la nature

des personnages, nous nous sommes aperçu qu'ils étaient plutôt similaires et que par conséquent, nous n'avions pas besoin de créer plusieurs classes pour des entités assez proches par leurs comportements.
Nous avons donc préféré factoriser le code. Par exemple, les radarReports sont les mêmes lorsqu'il s'agit d'un Jedi ou d'un DumbRobot. La liste contenue dans Robots est de type DumbRobot, or pour parcourir celle-ci de façon générique (cette liste peut contenir à la fois des Jedis et des DumbRobots), il faut parcourir et appeler des méthodes propres à l'interface Robot.

Mercurial Par désir d'une bonne organisation du travail, nous avons mis en place un système de gestion de version grâce au logiciel Mercurial. Néanmoins la prise en main a été difficile pour l'ensemble de l'équipe, à la fois pour ceux qui ne connaissaient pas ces procédés et également pour ceux qui l'administraient.

Gestion du son Initialement, nous voulions utiliser FMOD Ex, mais devant les difficultés de compilation, nous lui avons préféré la bibliothèque SDL Mixer qui est plus simple d'installation et d'utilisation.

Gestion de l'humain Un groupe de six personnes est difficile à gérer. En effet, chaque personnes ayant des niveaux différents dans divers domaines, la distribution des tâches a dut être rigoureusement pensé afin d'optimiser au maximum le travail.

7.3 Apports Personnels

Les notions vues en cours (pratique et théorique) à la fois en C++, en Architecture logicielle ou encore en Synthèse d'image nous ont permis de mieux aborder le projet et de le rendre plus accessible. Ce fut pour nous l'occasion d'apprendre à incorporer des objets 3D animés dans une scène OpenGl, rendant plus attractives nos réalisations. Il a fallu s'adapter à une structure logicielle informatique déjà en place et se l'approprier selon nos besoins.

Ce sujet très complet, nous a permis de mener un véritable projet de programmation, intégrant aussi bien des algorithmes connus et utilisés que des interactions visuelles avec l'utilisateur.
Le projet nous a permis d'apprendre, et d'acquérir une rigueur lors de la conception d'un programme en C++, d'Architecture Logicielle et d'OpenGL, cela provenant du fait que le projet nous a passionné. Il y avait un but concret, avec des résultats visibles et divertissants.

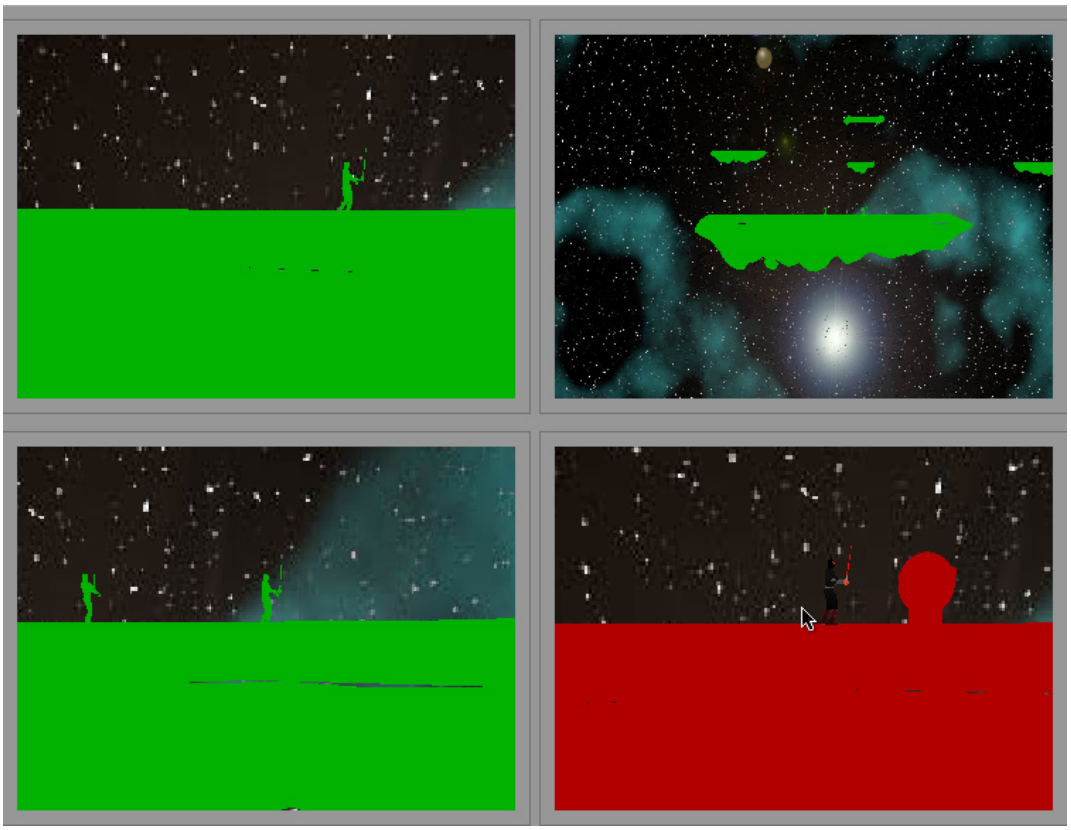

FIGURE 16 – Univers du Jeu (mode sans texture)