Développement d'un jeu vidéo de tir subjectif, programmé en C++ avec IrrLicht et modélisé avec Blender

LEANDRO KIELIGER - NICOLAS JULIEN ROUSSEL

28 octobre 2012

Avant-propos

Dès leur invention, les jeux vidéo ont rencontré un immense succès. D'abord présents sous forme de bornes d'arcade dans les cafés et autres endroits très fréquentés, les jeux vidéos devinrent ensuite si populaires que de nombreuses personnes commencèrent à s'intéresser à ce nouveau marché qui deviendra très rapidement une industrie à part entière. Si à la base les jeux vidéo n'étaient créés que par quelques férus d'informatique et d'électronique, enfermés jour et nuit dans leur garage, cette époque est mainteant révolue. En effet, de nos jours, derrière un jeu vidéo complet se cachent des armées de professionels répartis dans une vingtaine de métiers différents. Allant du simple testeur au modeleur 3D en passant par le chef de projet ou encore le directeur marketing, cet univers est extrêmement riche et varié.

Mais un jeu vidéo c'est avant tout un jeu et en tant que tel, il doit être divertissant. Le joueur doit pouvoir rapidement se sentir à l'aise avec les commandes afin de mieux s'immerger dans l'univers du jeu. Si celui-ci est réussi, les utilisateurs ressentiront l'envie d'y rejouer, seuls ou à plusieurs et tout ceci sans se soucier du fonctionnement des mécanismes complexes mis en œuvre. C'est à cet endroit que nous intervenons, passionnés par l'informatique et les jeux vidéos. Il n'a pas fallu beaucoup de temps avant que notre curiosité s'éveille et cherche à comprendre comment une pression sur un bouton permet de faire sauter un personnage dans un jeu ? Quels sont les mécanismes qui se cachent derrière l'appui d'une simple touche ? En bref, comment fait-on un jeu vidéo ?

C'est ici que débute notre travail de maturité ; nous avons décidé de franchir le pas et de découvrir ce qui se cache derrière l'écran de notre télévision ou de notre ordinateur. Bien que n'ayant que des connaissances de base en programmation, nous nous sommes tout de même lancés dans un projet d'envergure puisqu'il est question ici de tenter de créer un jeu vidéo de type « First Person Shooter » (signifiant « Jeu de Tir Subjectif » en français et abrégé FPS). C'est un genre que nous apprécions particulièrement puisque le joueur est amené à incarner directement le héros du jeu, en voyant à travers ses yeux. Nous ne cherchons donc nullement à égaler les jeux actuels ou à inventer un concept original ou même révolutionnaire, loin de là, notre objectif est de reproduire des fonctionnalités que l'on trouve dans les jeux de nos jours afin de comprendre à quelles difficultés les programmeurs et autres professionnels de la branche sont confrontés et quels sont les solutions et moyens qu'il nous faudra trouver et employer pour les résoudre. Le jeu que nous avons ainsi réalisé au cours de ce travail est également disponible en téléchargement libre sur le site internet créé spécialement pour l'occasion, *www.lkieliger.ch*

Nous espérons que ce rapport changera votre perception du jeu la prochaine fois que verrez un personnage évoluant dans un univers virtuel et que votre regard passera outre l'aspect visuel pour atteindre une partie de la technique qui orchestre le tout.

Remerciements

Nous tenons à remercier en premier lieu notre maître répondant qui est également notre maître de classe, monsieur J.-M. Faillétaz, pour les conseils, directives et corrections apportés pendant la rédaction de ce dossier.

Nous remercions également notre doyen, monsieur M. Gelsomino pour nous avoir fourni des plans du Gymnase de Burier. Ceux-ci se sont révélés indispensables à la reconstitution 3D du gymnase que nous avons effectuée.

Finalement, nous souhaitons adresser un immense remerciement à nos familles respectives ainsi qu'à tous nos amis pour leur soutien et l'intérêt qu'ils ont porté à notre travail durant l'intégralité de sa réalisation.

Table des matières

Avant-propos **i**

Remerciements **ii**

I Travail de maturité **1**

1 Introduction **2**
1.1 Outils 2
1.2 Compétences 4

2 3D **5**
2.1 Blender 5
2.2 Sources 7
2.3 Modélisation 7
2.4 Application des textures 8
2.4.1 UV mapping 9
2.5 Animations 10
2.5.1 Import/export 11

3 2D **12**
3.1 The Gimp 12
3.2 Sources 13
3.3 Conception et réalisation 13
3.4 Intégration au jeu 14

4 Code **15**
4.1 Langage de programmation 15
4.1.1 Programmation orientée objet 16
4.2 Bibliothèques informatiques 17
4.2.1 IrrLicht 19
4.2.2 IrrKlang 20
4.2.3 IrrBullet 21
4.2.4 Mise en relation des bibliothèques 21
4.3 Aide 22
4.3.1 Forums & communauté 22

5 Conclusion **23**

6 Bibliographie **24**

A Modélisation du Gymnase de Burier **26**

B Conséquences d'un clic de souris **28**

C Texturage d'un modèle complexe **31**

Première partie

Travail de maturité

Chapitre 1

Introduction

1.1 Outils

Afin que vous compreniez tout ce qu'il est nécessaire de mettre en œuvre pour un projet tel que la création d'un jeu vidéo, nous nous sentons obligés de vous présenter les outils nécessaires à leur développement. Dans un premier temps, nous vous présenterons les choix qui s'offraient à nous pour la réalisation de ce projet. Puis, nous décrirons plus en détail les outils que nous avons utilisés.

Premièrement, il faut savoir qu'il existe plusieurs possibilités pour créer un jeu vidéo. Nous allons vous en présenter trois, que nous classerons par ordre de difficulté d'utilisation. En effet, en informatique, les outils que nous utilisons peuvent être classés selon ce critère, on parle alors de « niveau ». Plus un langage de programmation se rapproche du langage machine, autrement dit le langage binaire, plus il est dit de « bas niveau ». Dans le cas contraire, plus un langage possède des fonctions sophistiquées destinées à simplifier le travail de l'utilisateur, plus il est dit de « haut niveau ». Ainsi, les langages de haut niveau permettront d'obtenir des résultats concrets beaucoup plus rapidement que les langages de bas niveau grâce à leur utilisation simplifiée. Malheureusement, si les langages de haut niveau sont plus simples d'utilisation, ils ont l'inconvénient d'être également moins souples car l'utilisateur n'a dès lors plus un accès aussi grand aux possibilités de sa machine. En conclusion, un langage de bas niveau est plus puissant mais plus complexe à utiliser qu'un langage de haut niveau et vice-versa. Cette précision étant faite, revenons aux choix qui s'offraient à nous :

La première possibilité est la programmation à partir de bibliothèques[1] standard comme par exemple les bibliothèques « OpenGL » et « Direct3D », celles-ci rassemblent les fonctions considérées comme « basiques ». Le développement de jeux grâce à ce moyen, a été surtout utilisé dans les débuts du jeu vidéo. C'est la méthode considérée comme celle de plus bas niveau pour créer une application 3D. Pour réaliser un projet de cette manière, il nous aurait fallu énormément de temps, car les bibliothèques standard ne comportent pas des fonctions permettant de créer un environnement en 3D aisément. Afficher un « simple » cube qui pivote sur l'écran nécessiterait déjà de nombreuses lignes de code.

La seconde manière de procéder consiste à utiliser d'un moteur 3D[2] associé d'autres moteurs spécifiques. Cette méthode se situe entre les bibliothèques standard et les moteurs de jeu qui seront évoqués plus loin. L'utilisation de ces bibliothèques nécessite l'emploi d'un IDE, un « Environnement de Développement Intégré », un logiciel qui permet de compiler et débogguer le code afin de créer le fichier exécutable de notre jeu. Un chapitre plus détaillé sera consacré dans la suite de ce rapport aux bibliothèques utilisées.

La troisième option, celle de plus haut niveau, utilise un moteur de jeu. Les moteurs de jeu, sont des programmes qui assistent grandement le développeur dans la réalisation, car ils possèdent une interface graphique qui permet de voir en temps réel la scène dans laquelle se déroule le jeu. De plus, ils intègrent automatiquement la gestion d'éléments complexes tels la gestion des collisions dans une scène en trois dimensions ou encore la gestion de l'éclairage et des effets de lumière. Cette méthode, est actuellement la plus utilisée de nos jours car elle crée une certaine répartition des tâches

1. une bibliothèque est un rassemblement de fonctions spécifiques qui nous permet d'accéder à celles-ci sans avoir à toujours les réécrire.

2. un moteur 3D est un composant logiciel qui crée des images numériques grâce à des coordonées relevées sur 3 dimensions.

dans la création d'un jeu. En effet, elle permet à toute une équipe d'artistes, scénaristes et codeurs de travailler en collaboration dans le processus de création du jeu. Les spécialistes en programmation peuvent ainsi s'occuper en parallèle du développement du « moteur de jeu » en lui-même qui doit être capable de résoudre des problèmes complexes.

Nous avons décidé d'opter pour la seconde méthode, celle qui nous semblait la plus adaptée pour la réalisation de notre projet. En effet, la première imposait un travail et un investissement en temps titanesque et la troisième, même si elle nous aurait fourni rapidement un résultat beaucoup plus riche et intéressant, masquait trop de difficultés que nous souhaitions résoudre nous-mêmes.

Toutes les techniques décrites ci-dessus servent à créer la structure des jeux, mais nous avons également besoin d'autres outils, qui eux vont nous servir à créer le contenu concret du jeu, à savoir les modèles 3D, les images et les sons.

Pour ce qui est des modèles 3D, ils ont tous été réalisés par nos soins grâce au logiciel de modélisation gratuit nommé « Blender ». Afin de réaliser nos modèles avec le plus de détails possibles, nous utilisons une photographie de ceux-ci à partir de laquelle nous modélisons leur silhouette pour ensuite leur donner du volume. Puis, pour les images qui vont nous servir par exemple à obtenir les textures de nos modèles 3D, nous cherchons sur le web des images pouvant représenter notre modèle que l'on modifie ensuite avec le logiciel d'édition d'image « TheGimp » pour qu'elles s'ajustent à notre modèle. Et finalement, les sons vont être eux aussi récupérés sur le web ou alors enregistrés par nous-même puis modifié grâce à « Audacity ».

Pour récapituler, nous créons avec des logiciels libres ou récupérons sur le Web les données des éléments physiques du jeu que nous rendons ensuite intéractifs grâce au code que nous avons créé qui définit comment les différents objets vont interagir les uns par rapport aux autres, mais aussi bien sûr en fonction des actions décidées par le joueur.

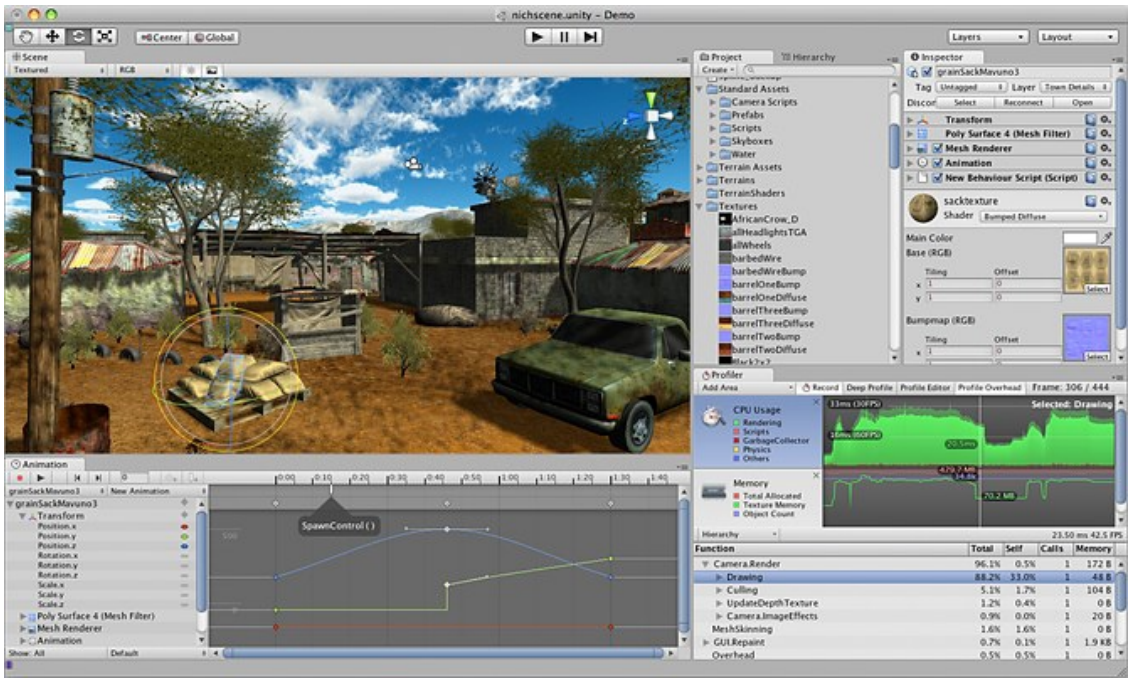

FIGURE 1.1 – Interface d'un moteur de jeu tel que Unity3D

1.2 Compétences

Il va sans dire que pour venir à bout du développement d'un jeu vidéo, de nombreuses connaissances sont requises et ce, dans des domaines informatiques variés. Le chapitre précédent décrivait les outils nécessaires à notre projet, et c'est principalement sur ceux-ci que doivent porter nos compétences. De la programmation à la modélisation, en passant par l'édition de sons, tous ces domaines constituent dans le monde professionnel des métiers distincts. Étant des néophytes dans chacun d'entre eux, notre but n'est bien entendu pas de tenter d'égaler le niveau d'un jeu d'envergure commerciale, mais plutôt d'apprendre à utiliser du mieux que nous le pouvons des outils pouvant parfois se révéler très complexes.

Nous ne nous sommes pas lancés dans ce projet en connaissant parfaitement chaque environnement avec lequel nous allions travailler. En effet, nous n'avons suivi aucune formation en programmation, en modélisation 3D ou même en retouche d'image. Les connaissances que nous avons acquises proviennent exclusivement d'articles trouvés sur internet ou dans des livres. C'est principalement ce manque de connaissances qui nous distingue des professionnels qui, eux, ont une maîtrise parfaite des logiciels sur lesquels ils travaillent. Par exemple, pour le domaine de la programmation, l'apprentissage des fonctions fournies par notre moteur 3D Irrlicht s'est fait au cours du temps, par « tâtonnements ». Ainsi, un temps considérable à été consacré à la recherche et à l'expérimentation de différentes méthodes de travail dans le but de résoudre certains problèmes spécifiques. Problèmes qui n'auraient pas forcément mis en difficulté une personne maîtrisant parfaitement le moteur. À titre indicatif, le temps investi pour tenter d'appréhender les différents outils mis à notre disposition représente plus de la moitié du temps de travail total. L'exemple du moteur 3D peut bien sûr aussi s'appliquer aux autres logiciels et bibliothèques utilisés comme le programme de modélisation 3D Blender qui, lorsqu'il est correctement employé, peut rendre des images d'un réalisme époustouflant. Un écart impressionnant se retrouve donc à ce niveau par l'expérience qu'ont pu accumuler les développeurs, les graphistes et autres personnes du métier du jeu vidéo.

Au final, malgré des compétences modestes, s'attaquer à un projet tel que la création d'un jeu vidéo est le meilleur moyen de progresser dans les différents domaines d'activités dont il fait partie. Même si au premier abord le but fixé semble inatteignable, c'est à force d'essais et de persévérance que le travail avance peu à peu. L'aspect expérimentation est un élément clé de notre travail qui consiste à apréhender les mécanismes qui se cachent derrière un jeu vidéo.

Dans la suite de ce rapport nous vous présenterons tout d'abord les logiciels et techniques que nous avons employés pour réaliser le contenu concret notre projet puis dans un second temps viendront les présentations et explications des outils permettant leur mise en association.

Chapitre 2

3D

2.1 Blender

Pour entamer cette section consacrée à la création du contenu visuel du jeu, nous avons décidé de présenter le logiciel de modélisation 3D Blender. En effet, notre jeu a pour objectif d'immerger le joueur dans un environnement tridimensionnel dans lequel il peut évoluer et intéragir avec les objets et les ennemis. C'est pourquoi, dans le but de représenter cet environnement, nous utiliserons Blender pour créer de toutes pièces tous les éléments 3D du jeu tels les décors ou les personnages. Nous avons entrepris pour notre projet de reconstituer avec Blender le bâtiment principal du Gymnase de Burier dans la perpective de plonger le joueur dans un environnement qui lui est familier. Vous trouverez dans l'annexe A les différentes étapes de cette reconstitution que vous pourrez consulter en complément de la lecture de ce chapitre. Tous ces volumes sont appelés « modèles » et c'est le moteur 3D de notre jeu qui sera chargé de déterminer quelles sont les faces visibles de chaque modèle qui doivent être affichées à l'écran.

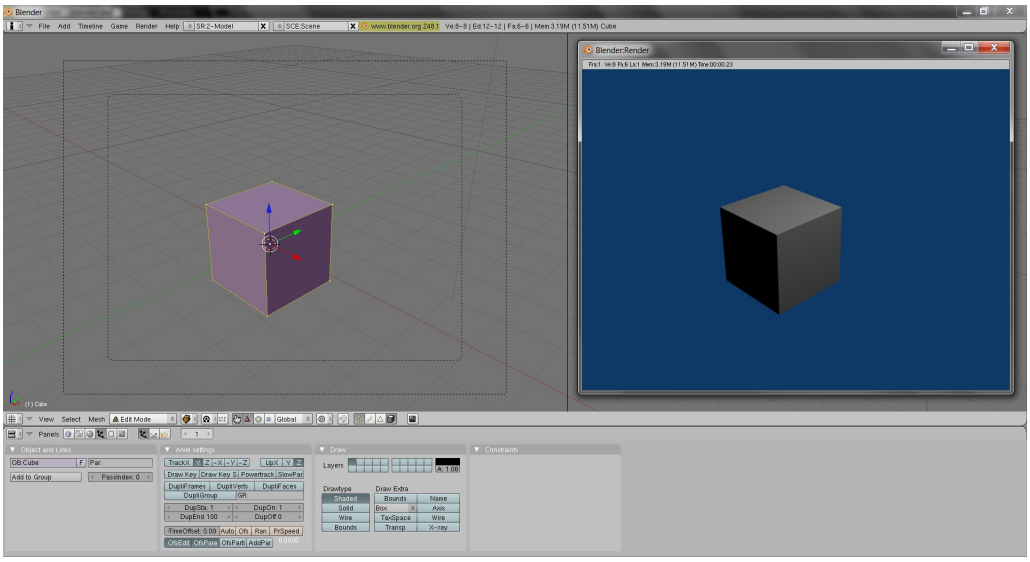

FIGURE 2.1 – Interface de Blender avec la fenêtre de rendu à droite

Blender est un outil de modélisation 3D très puissant, libre et gratuit. Bien entendu, il n'a pas été conçu exclusivement pour modéliser du contenu de jeux vidéo. En effet, le logiciel permet de réaliser des rendus [1], des films ou des animations de qualité à partir de scènes en 3 dimensions. Blender n'est pas un outil facile à prendre en main ; c'est pourquoi il nous a été nécessaire de chercher de l'aide

1. Un rendu est une image numérique réalisée par le logiciel à partir d'une scène d'objets, cette image peut être comparée à une photo.

et des explications sur internet afin d'utiliser tous les outils à disposition de manière apropriée et optimisée.

Le niveau de détails et de précision de la modélisation est nettement moins élevé dans un modèle destiné à un jeu que dans un modèle destiné à un film d'animation. En effet, si le rendu se doit d'être parfait (ou du moins le plus proche possible de la réalité) pour un film d'animation, ce n'est pas le cas pour un jeu, car le moteur 3D du jeu doit être capable de calculer et d'afficher plusieurs rendus par seconde sur l'écran du joueur. Plus le niveau de détails des objets est élevé, plus il devient difficile de réaliser plusieurs rendus par seconde, rendant ainsi le jeu saccadé. A titre de comparaison, si un ordinateur domestique tentait de calculer le rendu d'un film Pixar il lui faudrait plusieurs centaines d'années pour en venir à bout [2]. On comprend dès lors pourquoi les modèles 3D pour jeux vidéos se doivent d'être simplifiés au maximum.

Même si le résultat final que représente un rendu 3D n'est qu'une « simple » image numérique affichée sur un écran, il apparaît évident qu'obtenir ce rendu à partir d'objets modélisés en trois dimensions est une méthode très puissante. En effet, les objets peuvent facilement être placés, orientés ou éclairés différemment. Mais cela n'a pas toujours été le cas. Dans les débuts des jeux vidéos et quand la puissance de calcul et la capacité en mémoire des machines étaient très limitées, les développeurs durent trouver des astuces pour donner l'illusion de la troisième dimension sans qu'elle n'existe réellement (du moins dans le monde virtuel du jeu). C'est devenu fait rare de nos jours puisque la puissance des machines actuelles rend cette technique d'économie de ressources obsolète. Nous pouvons voir par exemple dans l'image qui suit, tirée du jeu « DoomII » sorti en 1994 que seul l'environnement où progresse le joueur (plus communément appelé niveau) est réellement en 3D, nous entendons par là que chaque sommet possède bel est bien des coordonnées x,y,z. Le reste des objets qui s'affichent à l'écran ne sont que de simples images animées [3] ayant la particularité d'être orientées en direction de la caméra de manière à constamment faire face au joueur. Ce type de représentation est parfois appelée dans le milieu du jeu vidéo la « 2.5D ».

FIGURE 2.2 – Capture d'écran du jeu Doom II (1994)

2. Le temps de rendu nécessaire pour réaliser une image est compris entre 5 et 15 heures. Un film étant constitué de 25 images par seconde, il ne reste plus qu'à multiplier le total par la durée du film.

3. Ces images sont généralement appelées *sprites*.

2.2 Sources

Même si tous nos modèles 3D on été réalisés par nos soins, il nous a été impossible de réaliser nous même l'intégralité du travail. En effet, nos compétences résident plus dans la partie technique que dans la partie artistique et c'est pourquoi, pour tous les modèles que nous avons créés, nous avons en premier lieu dû chercher des croquis et photos sur internet pour ensuite en tirer le profil général. Il faut savoir que la création de textures des objets et personnages d'un jeu constitue dans le monde professionnel un métier à part entière. Ainsi, pour certains cas où les photos ne suffisaient pas, nous avons été contraints d'utiliser des images et travaux d'artistes mettant leur travail à disposition sur internet. Dans chaque cas, nous avons dû apporter des modifications et retoucher les images nous-mêmes. Cette partie du travail fait l'objet d'une autre partie de ce rapport.

2.3 Modélisation

Il existe plusieurs méthodes pour créer un modèle 3D ; la plus utilisée d'entre elles est sans doute l'utilisation d'une silhouette, croquis ou photo du modèle que l'on désire numériser dans le but de guider le modélisateur dans la reproduction virtuelle de l'objet. Les images ainsi employées s'appellent des *blueprints*. Ci-dessous, un croquis d'être humain de face et de profil est utilisé dans l'objectif de modéliser une créature humanoïde pour notre jeu.

En général la modélisation se fait à partir d'un simple cube dont les faces sont ensuite dupliquées et les sommets déplacés pour former le volume désiré. Le logiciel Blender fournit un rendu en temps réel de notre modèle ainsi qu'une représentation précise des sommets, arrêtes et faces qui le composent. Avec des outils spécifiques intégrés à Blender nous avons ainsi la capacité de déformer le volume géométrique jusqu'à ce que sa forme nous convienne.

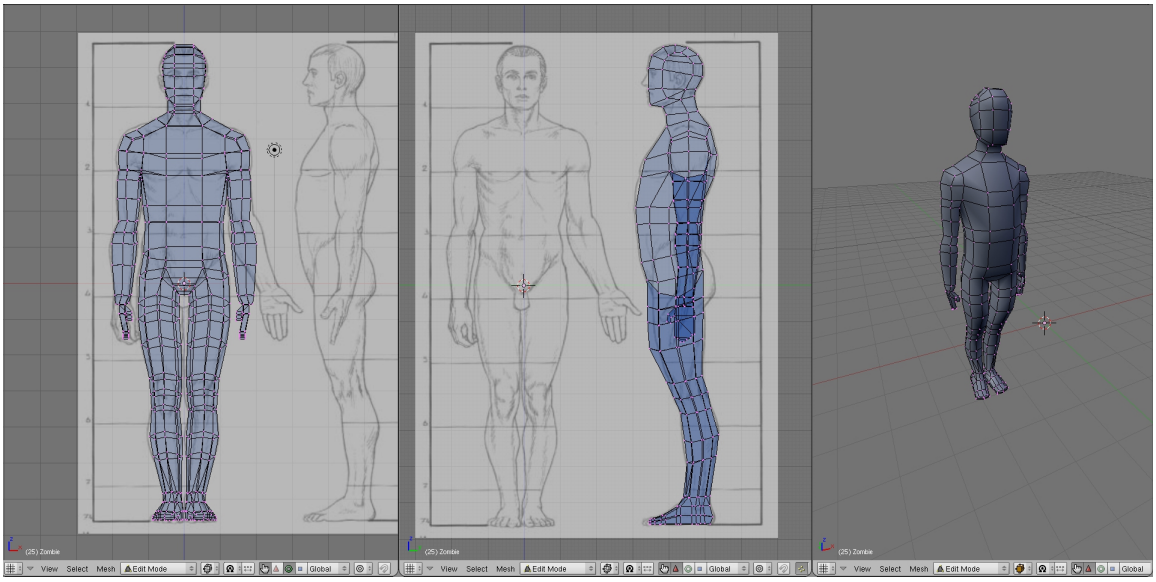

FIGURE 2.3 – Exemple d'utilisation d'un blueprint avec multiples points de vues

2.4 Application des textures

Il ne suffit pas de modéliser un objet soigneusement avec Blender pour le rendre réaliste, à cette étape de la création le modèle apparaît de couleur grise sur le rendu (comme on peut le remarquer sur la figure illustrant l'interface de blender, au début de ce chapitre). Pour remédier à cela, il faut utiliser ce que l'on appelle des *textures* que nous allons appliquer à notre modèle 3D. Une texture est une image numérique pouvant provenir d'un dessin ou d'une photo. Dans notre cas, sans aucune expérience de designer, nous avons décidé d'exploiter des photos numériques présentes sur le Web. Nous les avons ensuite retouchées avec des logiciels d'édition d'images. Une texture peut changer du tout au tout l'apparence d'un modèle, en effet, elle peut être considérée comme un habit que l'on place sur l'objet pour lui donner l'apparence que l'on souhaite. Ainsi, le simple cube gris de base de blender peut se tranformer en caisse de transport de marchandises en bois avec la texture adéquate, mais aussi en boîte aux lettres si le modeleur le désire.

Appliquer correctement une texture est un point capital pour un modèle 3D destiné au jeu vidéo, car il doit être simplifié au maximum. Mais grâce à l'utilisation d'une texture de qualité, des détails peuvent ainsi être récupérés avec une exploitation bien moindre de la mémoire que si ceux-ci avaient été modélisés un par un. Par exemple, si nous souhaitons modéliser un personnage féminin avec des cheveux longs et blonds, nous créerons un volume global comprenant la chevelure en lui appliquant ensuite une texture représentant des cheveux blonds.

Une autre technique que nous avons utilisée pour donner un degré de réalisme supplémentaire s'appelle le *normal mapping*. Elle consiste à simuler graphiquement (donc sous la forme d'une texture) un niveau de détails géométriques élevé. L'astuce consiste ici à fournir au moteur 3D un double de la texture où, cette fois ci, chaque pixel indique un degré de profondeur. Lors du rendu, le moteur 3D calculera des effets de lumière supplémentaires sur le modèle qui donneront l'impression d'un niveau de détails élevé. Par exemple, pour notre modèle 3D d'arme, la « normal map » donnera l'impression que le numéro de série inscrit sur l'arme est gravé en profondeur et non pas juste dessiné. Malheureusement il est difficile de montrer toute la puissance de cette technique sur papier car comme nous l'avons déjà précisé ci-dessus, cette technique joue essentiellement sur des effets de lumières et par conséquent le résultat est beaucoup plus visible lorsque le modèle est en mouvement ou que les sources lumineuses se déplacent.

FIGURE 2.4 – Amélioration du réalisme du modèle avec texture et normal map

FIGURE 2.5 – La normal map d'une des armes présentes dans le jeu

2.4.1 UV mapping

Malheureusement, la plupart du temps, appliquer une texture n'est pas chose aisée. En effet, il faut indiquer avec précision quelle portion de la texture s'associe avec quelle face. La technique universellement utilisée pour texturer des modèles complexes s'appelle l' « UV mapping » et son principe est très simple. On découpe le modèle 3D de manière à pouvoir en fabriquer un développement ; on obtient ainsi une image en deux dimensions [4]. On utilise ensuite ce développement comme guide pour dessiner la texture puis une fois le travail terminé, Blender appliquera de manière automatisée notre texture sur le modèle en respectant les proportions données lors de la création de la texture. Même si le principe reste relativement simple, la partie pratique se révèle parfois très complexe. Ci-dessous sont illustrées les étapes de texturage pour un simple cube. Cependant, il faut bien garder à l'esprit que dans un jeu vidéo les modèles sont toujours bien plus sophistiqués que de simples cubes. Vous trouverez en annexe un exemple de texturage d'un modèle beaucoup plus compliqué puisqu'il est formé de près d'un milier de faces. Il s'agit d'un modèle de fusil de précision réalisé par nos soins que nous n'avons pas pu intégrer à notre projet par faute de temps.

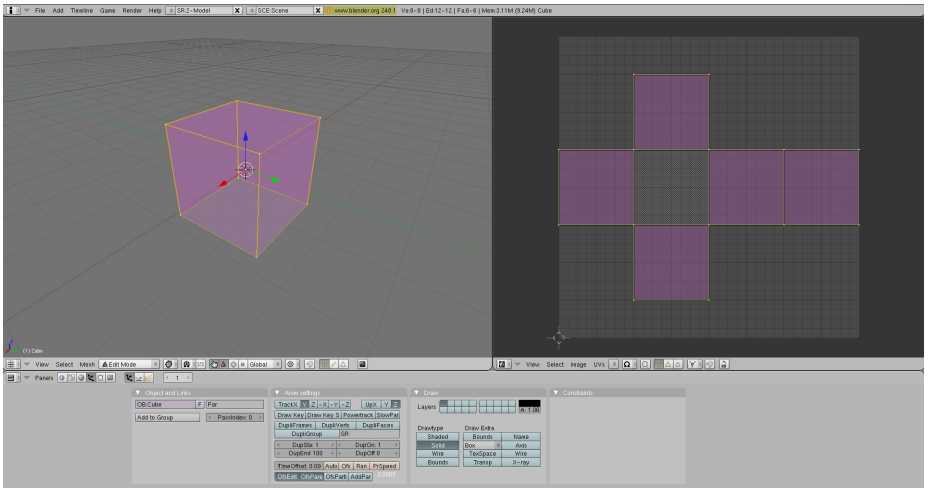

FIGURE 2.6 – Un des possibles développement du cube

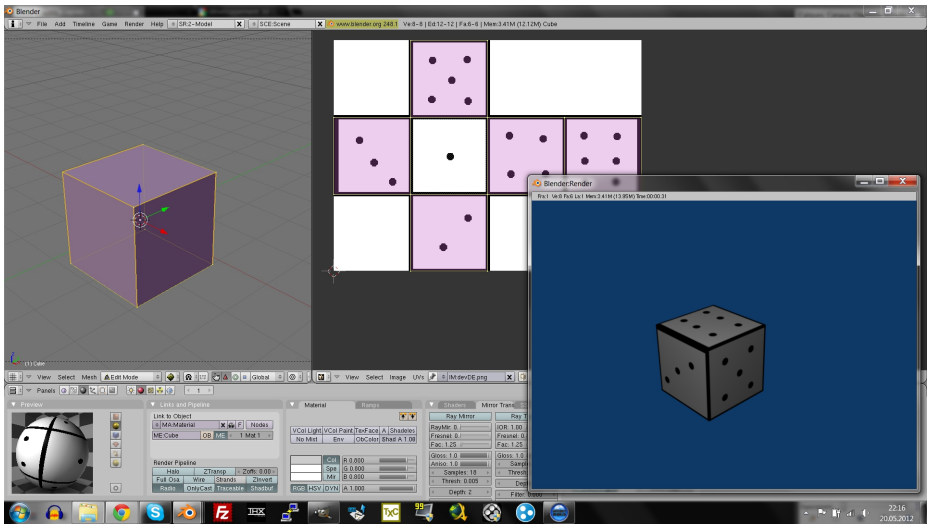

FIGURE 2.7 – Modèle final obtenu après texturage.

4. D'où le nom UV mapping, U et V étant les axes x,y pour les images.

2.5 Animations

Bien sûr tout ceci resterait très statique pour un jeu vidéo s'il n'y avait pas la possibilité d'animer nos modèles. Il existe ici également de nombreuses techniques d'animation. Celle que nous utilisons pour notre projet est une technique très répandue appelée animation squelettale. Elle est particulièrement efficace quand il s'agit d'animer des personnages. Elle consiste en une création d'ossature que l'on place à l'intérieur du modèle 3D. Chaque sommet est ensuite lié à un os constituant le squelette et sera ainsi déplacé lorsque l'ossature se mettra en mouvement. Cela permet une déformation du modèle extrêmement réaliste puisque les sommets se déplaceront en fonction de points de pivots. Cette technique simplifie l'animation puisque le modélisateur n'a plus qu'à agir sur quelques os et non sur chaque sommet du modèle.

Finalement la position et la rotation de chaque os par rapport au repère orthonormé du logiciel sont enregistrées en fonction du temps et pourront ensuite être récupérées par le moteur 3D du jeu qui sera ainsi capable de rejouer l'animation telle que le modeleur l'a créée. Blender nous apporte une nouvelle fois une aide précieuse puisqu'il suffit d'enregistrer quelques positions « clés », le logiciel s'occupant ensuite de générer automatiquement la transition entre chaque point.

À gauche de la figure 2.8, on peut observer notre modèle humanoïde (en gris) avec à l'intérieur, le squelette qui est utilisé pour l'animation (en bleu). À droite se trouve le graphique représentant les points clés de l'animation pour chaque os ainsi que celui représentant la position et la rotation d'un des os formant le squelette en fonction du temps. Pour être exact, dans la modélisation, on ne parle pas réellement de temps mais plutôt de *frames*, ainsi un objet est animé à 25 frames par seconde par défaut (cette valeur pouvant être modifiée par la suite). Par exemple, dans l'image qui suit nous pouvons observer le modèle tel qu'il sera affiché dans le jeu à la 25ème frame, soit après 1 seconde d'animation.

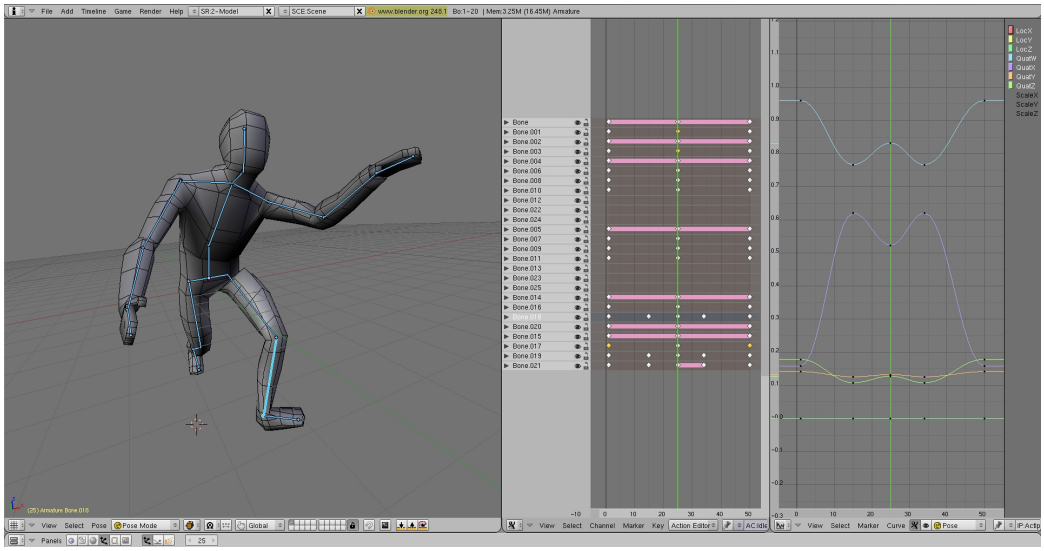

FIGURE 2.8 – L'animation d'un zombie, visible dans notre jeu avec Blender

2.5.1 Import/export

Une des grandes difficultés que nous avons rencontrées lors de la création de modèles avec Blender est le fait que les fichiers créés ne sont pas compatibles avec le moteur 3D de notre jeu. En effet, il existe une multitude de format de fichiers 3D comme il existe une multitude de format de fichiers textes (comme par exemple les extension .txt, .doc ou .pdf). Ainsi, certains formats de fichiers sont plus appropriés que d'autres pour certaines circonstances et tous ne sont pas exploitables par le moteur 3D. Par exemple, on privilégiera des formats différents si le modèle est animé ou pas ou s'il possède une normal map ou non. Ceci étant dit, les développeurs de Blender ont pallié au problème en instaurant un système d'exportation de modèle. Autrement dit, le modeleur 3D travaille sur son projet avec le type de fichier propre à Blender (d'extension .blend) puis lorsqu'il est satisfait de son travail, il a la possibilité d'enregistrer son modèle dans un type de fichier différent, exploitable ensuite par un autre logiciel. De plus, Blender possède une interface de script [5] Python (un langage de programmation). Cela signifie qu'il est possible aux utilisateurs de programmer de nouvelles fonctionnalités qui seront greffées au logiciel Blender de base. Ces fonctionnalités additionnelles sont appelées « greffons »ou « plug-ins » en anglais. Ainsi, si le modeleur souhaite exporter sa création sur un fichier de type non supporté normalement par Blender, il a de fortes chances de trouver sur internet un script écrit en Python lui permettant de rendre le format de fichier qu'il désire compatible avec Blender. Le principe d'exportation fonctionne aussi en sens inverse ; on parle alors d'importation. Cette opération est surtout utile pour récupérer et ensuite modifier des modèles 3D qui n'ont pas été créés dès le début avec Blender.

Pour notre projet, les problèmes sont apparus au niveau de l'exportation avec Blender et de l'ouverture du fichier avec le moteur 3D de notre jeu, IrrLicht. En effet, nous avions besoin de modèles animés par ossature. Or, le seul format supporté par IrrLicht offrant la possibilités d'animer nos modèles de telle manière était le format Blitz3D (d'extension .b3d) et ce dernier n'existait pas pour blender. Après de longues recherches sur internet nous avons finalement trouvé un script d'export en .b3d créé par un particulier. Malheureusement, le script n'est de loin pas parfait et quelques particularités de nos modèles ne purent ainsi pas être exportées. Ce fut là une grande source de problèmes pour notre projet car il nous a fallu essayer des dizaines voire des centaines de fois d'exporter un même objet en modifiant à chaque fois quelques paramètres avant d'obtenir un résultat satisfaisant. Parmis les problèmes rencontrés nous pourrions relever des animations qui ne fonctionnaient pas, des faces transparentes ou encore des modèles qui semblaient s'écraser sur eux-mêmes lorsque nous tentions de les animer. Étant donné que le site web de la personne ayant créé ce script n'est plus disponible, il ne sera vraisemblablement pas possible d'obtenir un code parfait à l'avenir, bien que quelques versions corrigées commencent à faire leur apparition sur la toile.

5. Un script est un fichier texte contenant des commandes qui vont être lues et interprétées par un programme (appelé interpréteur) avant d'être exécutées.

Chapitre 3

2D

3.1 The Gimp

The Gimp est un logiciel de traitement d'image que l'on compare bien souvent avec un autre logiciel extrêmement connu du même type : Photoshop. The Gimp, bien que moins puissant, a l'avantage notable d'être un logiciel libre et gratuit. C'est pourquoi nous l'avons choisi pour traiter toutes nos images numériques. The Gimp permet d'étoffer tout le contenu créé dans le chapitre précédent, en créant des textures pour les modèles 3D. De plus, c'est grâce à lui que nous pouvons créer les différents éléments de l' « Affichage Tête Haute » ou plus simplement l'*ATH* [1]. Les opérations réalisées vont du simple redimensionnement à la création complète d'une image. Le champ d'utilisation du programme étant grand, la maîtrise du logiciel nous est devenue nécessaire. C'est sur internet que nous avons trouvé l'aide et les informations requises pour utiliser au mieux ses capacités.

The Gimp permet également de modifier les propriétés d'une image, comme par exemple ses dimensions ou son format. Les propriétés de chaque image doivent suivre une certaine logique car dans le cas contraire, si chaque image possède son propre « style artistique », l'ensemble visuel pourrait devenir désagréable pour le joueur. Par exemple, si l'environnement est constitué de différents tons de gris, il apparaît évident que des objets de couleur fluorescente ne sont pas assortis au thème visuel. De plus, l'utilisation d'une texture trop petite pour le modèle 3D qu'elle recouvre, donne un rendu pixélisé ou comportant des tâches grises (le gris étant la couleur par défaut des modèles qui ne possèdent pas de texture). D'autres paramètres importants doivent être pris en compte lors de la réalisation des images, comme par exemple la taille de l'écran du joueur. En effet, ce dernier influence le positionnement de chaque élément par rapport à l'ensemble visuel. Ainsi, certains éléments peuvent paraître trop grands ou trop petits, handicapant alors le joueur au niveau de la lisibilité.

Nous illustrons nos propos dans la suite de ce chapitre avec la création de toutes pièces d'une image présente dans le jeu ; la minimap [2].

1. Traduction du terme anglais « Head-Up Display » abrégé *HUD*. L'*ATH* est la partie visuelle du jeu constamment visible à l'écran et permettant au joueur de connaître certaines caractéristiques de son personnage comme par exemple son emplacement ou les munitions qu'il lui reste. Nous utiliserons l'abréviation anglaise *HUD* pour la suite de ce rapport car c'est cette dernière qui est universellement utilisée dans le domaine des jeux.

2. La minimap est l'élément du *HUD* qui permet au joueur de se repérer dans le jeu. Le joueur y est représenté sous la forme d'un point rouge, se déplaçant sur une vue aérienne du monde virtuel

3.2 Sources

Une grande partie des images de notre jeu a été créée par nos soins. Néanmoins, pour certaines d'entre elles, ce n'est pas le cas. En effet, l'utilisation d'images tirées du Web à été nécessaire pour éviter l'immense travail que représente la création d'une image en partant de rien. Surtout pour des personnes qui, comme nous, n'ont pas de talent particulier en dessin numérique. Ainsi, pour les images provenant de sources extérieures, nous n'avons procédé qu'à des retouches. Ceci implique tout de même de retravailler chaque image avec The Gimp pour qu'elles puissent ensuite être parfaitement intégrées au jeu. L'exemple que nous avons choisi, la minimap, a été réalisé à l'aide du plan du Gymnase de Burier que nous pouvons voir à la figure 3.1.

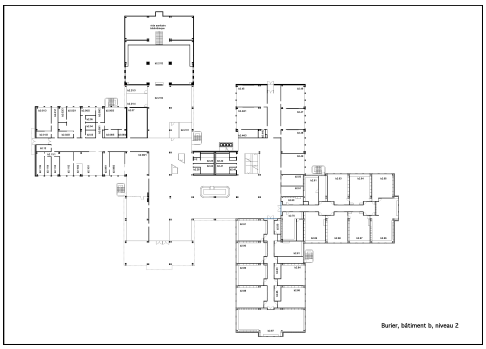

FIGURE 3.1 – Plan du deuxième étage du gymnase de Burier

3.3 Conception et réalisation

Tout d'abord, il faut savoir que le travail artistique qui est réalisé dans le cadre du développement d'un jeux vidéo est celui qui nécessite le plus de personnel. En effet, le rapport entre le nombre de programmeurs et le nombre de graphistes peut aller jusqu'à 1 programmeur pour 1000 graphistes [3] (ce ratio varie selon le jeu et le studio de développement). Il va sans dire que nous n'avons pas les moyens d'égaler le travail des artistes du milieu professionnel. C'est cela qui justifie notre choix d'utiliser pour notre jeu des images à disposition sur le Web. Comme expliqué dans la section 3.2, ces images subissent tout de même des modifications avant d'être intégrées au jeu de telle manière à ce qu'elles s'adaptent à son style. C'est justement à l'aide du logiciel The Gimp que nous procédons à ces modifications.

Voici quelques exemples de manipulations effectuées pour parvenir au résultat final de notre minimap. Premièrement, on découpe et redimensionne l'image « source » de manière à obtenir un résultat qui ne gêne pas le joueur car rappelons-le, ces images sont constamment présentes à l'écran. Par exemple, si l'image est trop grande, le joueur ne peut plus percevoir ce qui se passe dans le jeu, et si au contraire elle est trop petite, il ne peut pas discerner correctement ce qui est y est représenté. On procède ensuite à des retouches de couleurs pour qu'elle corresponde au ton général du *HUD*. Arrivé à ce stade, l'image peut être intégrée au jeu.

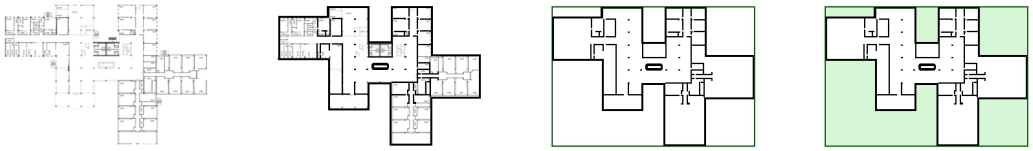

FIGURE 3.2 – Quelques étapes de le conception de l'image servant de minimap

3. Source : conférence sur le métier d'informaticien au forum horizon 2012

3.4 Intégration au jeu

Comme cela a été expliqué précédemment, les images peuvent servir de textures pour les modèles 3D mais ce n'est pas leur seule utilité. Il y a la partie appelée le « HUD », qui est entièrement faite à partir d'images et de texte. Chacun de ces éléments est intégré au jeu grâce au code. Bien sûr, pour les images servant de textures cela se passe exactement de la même manière. Le code permet de modifier certains aspects des images lors de l'exécution du programme comme par exemple la possibilité de redimensionner une image.

C'est également à l'aide du code que nous réalisons certaines spécialités 2D comme la représentation dynamique de la position du personnage sur la minimap. Faire bouger une image n'est qu'une des nombreuses possibilités qui s'offrent à nous par la programmation. Il est ainsi possible d'envisager des éléments du HUD qui changent de taille ou encore qui ne s'affichent que partiellement. Sur la capture d'écran ci-dessous, le programme dispose les nombreux éléments qui constituent le *HUD* en fonction de la résolution d'écran choisie par le joueur. Cela permet à des joueurs possédant des écrans différents d'obtenir la même expérience de jeu, on parle alors de positionnement *relatif*. Cela signifie qu'un élément n'est non plus fixé sur une coordonnée précise, mais placé en fonction de la largeur et de la hauteur de l'écran. En conclusion, comme pour les modèles 3D, le code permet de rassembler et d'utiliser efficacement toutes les ressources créées ou retouchées avec The Gimp.

FIGURE 3.3 – Affichage des différents éléments du *HUD* à l'écran pendant une partie.

Chapitre 4

Code

4.1 Langage de programmation

La programmation est un élément fondamental de notre travail. C'est grâce à elle que les différentes ressources et outils dont nous disposons pourront intéragir. Le code est en quelque sorte l'élément liant de tout notre travail et c'est par le biais de son utilisation que nous sommes capables de passer des ordres et des commandes qui seront exécutés par la machine. Pour ce faire, il est indispensable de choisir un langage de programmation. Il faut savoir qu'un ordinateur ne parle ni français, ni anglais, ni quelqu'autre langue humaine imaginable. Au niveau le plus basique qui soit, un ordinateur ne comprend et ne manipule que des 0 et des 1 appelés *binary digits*, ou plus simplement *bits*. Alignés en grande quantité, ces bits constituent ce que l'on appelle le langage binaire. Toutes les commandes que le programmeur désire envoyer a l'ordinateur consistent en suites de 0 et de 1. Heureusement pour nous, simples mortels, il existe un langage qui permet de « parler » à la machine, ce langage est appelé « Assembleur ». Malheureusement, « coder » un logiciel en assembleur est une tâche ardue même si le programme que l'on désire créer est très simple.

```
0040DC35|   891424        |MOV DWORD PTR SS:[ESP],EDX
0040DC38|   E8 6B180F00   |CALL 004FF4A8
0040DC3D|   8D45 F0       |LEA EAX,[EBP-10]
0040DC40|   890424        |MOV DWORD PTR SS:[ESP],EAX
0040DC43|   C745 B8 000000|MOV DWORD PTR SS:[EBP-48],0
0040DC4A|   E8 056C1600   |CALL 00574854
0040DC4F ^  E9 3DFFFFFF   |JMP 0040DB91
0040DC54|   884D B0       |MOV ECX,DWORD PTR SS:[EBP-50]
0040DC57|   890C24        |MOV DWORD PTR SS:[ESP],ECX
0040DC5A|   E8 49180F00   |CALL 004FF4A8
0040DC5F ^  E9 2DFFFFFF   |JMP 0040DB91
0040DC64|   55            |PUSH EBP
0040DC65|   89E5          |MOV EBP,ESP
0040DC67|   83EC 78       |SUB ESP,78
0040DC6A|   C745 CC 74ED4F|MOV DWORD PTR SS:[EBP-34],OFFSET 004FED
0040DC71|   C745 D0 0AD259|MOV DWORD PTR SS:[EBP-30],OFFSET 0059D2
0040DC78|   8D45 F8       |LEA EAX,[EBP-8]
Stack [0028FF88]=0
EBP=0028FF94
```

FIGURE 4.1 – Exemple d'instructions en assembleur

Les informaticiens ont ainsi inventé des langages de programmation. Ces derniers permettent à un utilisateur d'entrer des commandes relativement compréhensibles qui seront ensuite traduites en code binaire. Ce code pourra ensuite être exécuté par la machine. Voici par exemple à quoi ressemble du code en language C++. C'est un exemple des plus simples qui soit puisqu'il ne fait qu'afficher la phrase « Hello World ! » dans une fenêtre. Ce code une fois compilé ressemblera à une suite de 0 et de 1.

Code C++

```cpp
#include <iostream>
int main()
{
  std::cout<<"Hello World !"<<std::endl;
  return 0;
}
```

Ces langages qui jouent le rôle d'intermédiaire entre l'homme et la machine, il en existe des milliers. En fait, n'importe qui avec un peu de connaissance mais surtout beaucoup de courage, peut commencer à créer son propre langage de programmation. Il est capital de bien le choisir car si le principe de base se veut le même (intermédiaire homme-machine), tous diffèrent par leur syntaxe, leurs fonctionnalités et leurs performances. Nous retrouvons donc ici le principe du langage de haut et de bas niveau vu au début de ce rapport. Un langage se rapprochant du fonctionnement de la machine offrira une souplesse d'utilisation et des performances exceptionnelles mais ceci au détriment de la facilité d'utilisation. Par exemple, pour un langage de très bas niveau, chaque morceau de mémoire utilisé par le programme devra être libéré « manuellement » par le programmeur car le nettoyage automatique de la mémoire ne fera pas partie des fonctionnalités du langage. Au contraire, certains langages de haut niveau se proposent d'automatiser certaines tâches afin d'alléger le travail du programmeur avec le désavantage notable d'empêcher ce dernier de manipuler à son gré tous les composants à sa disposition. Notre choix du langage de programmation s'est porté sur le C++. Un langage extrêmement puissant et réputé dans le domaine du jeu vidéo à tel point qu'il est devenu presque incontournable. Sa force réside non seulement dans le fait qu'il soit rapide et souple d'utilisation mais également par sa particularité d'être « orienté objet » . Le chapitre suivant expliquera en détails la particularité des langages dits orientés objet ainsi que l'avantage qu'ils offrent pour la réalisation de certaines tâches.

Le C++ a la caractéristique très importante d'être un langage performant en matière d'efficacité calculatoire. Pour un même temps donné il pourra effectuer un plus grand nombre d'opérations. C'est ce dernier point qui attire les développeurs de jeux vidéos car ils ont besoin d'un langage fiable et rapide capable de créer des programmes qui fonctionnent en « temps réel ». Ceci signifie que le comportement du programme doit pouvoir changer à chaque instant en fonction des différents paramètres qui lui seront fournis pas l'utilisateur.

4.1.1 Programmation orientée objet

Les languages de programmation se répartissent en plusieurs catégories selon leur structure de fonctionnement. Il existe par exemple les langages procéduraux tels le *Fortran*, qui consiste à faire exécuter par la machine une suite de commandes. Lorsque le programme atteint la fin de la liste d'instructions il se ferme. D'autres en revanche auront une structure plus compliquée comme c'est le cas du C++ ; ce dernier fait partie de ce que l'on apelle les langages orientés objets. Le principe ici est de créer des morceaux de code appelés *objets*, représentant chacun un élément fondamental du programme. Ces différents *objets* pourront ensuite agir sur le fonctionnement du programme à l'aide de fonctions simples, utilisables par le programmeur. C'est ce type de langage que nous utilisons dans notre projet car c'est celui qui s'adapte le mieux à nos besoins. En effet, nous pouvons facilement imaginer le joueur comme un objet évoluant dans un monde (lui aussi un objet) et se battant contre des hordes de créatures, elles aussi, toutes des objets. Par exemple, le joueur aura des fonctions du type « infliger des dégâts » qui agiront sur les points de vie de ses ennemis ou « récupérer des munitions » alors que les créatures auront quand à elles des fonctions comme « émettre un grognement » qui apporteront une ambiance particulière au jeu. Les objets possèdent ce que l'on appelle des *méthodes*, soit fonctions qui leur permettent d'agir et des *attributs* qui sont, quand à eux, des valeurs propres à l'objet. Ici, la fonction du joueur qui inflige des dégâts à ses ennemis est une méthode et ses attributs seront les montants de points de vie et de munitions. Nous voyons dès lors qu'il est très facile de mettre en relation nos objets entre eux ; le joueur ayant fait appel à sa méthode infligeant des dégats à une créature, celle-ci fera appel à la fonction « émettre un grognement » et ainsi de suite. Cet enchaînement d'interaction nous permettra de rendre notre jeu dynamique et ce, de manière extrêmement logique au niveau du code. Un autre avantage de la programmation orientée objet en ce qui concerne les jeux vidéos est qu'elle propose un système d'héritage. L'héritage consiste à dire qu'un objet 1 est enfant d'un objet 2 et que par conséquent, il héritera de toutes les méthodes et attributs de l'objet parent. Ainsi le programmeur économise un grand temps de travail car il peut très facilement reprendre la structure de base d'un objet pour l'appliquer à d'autres.

Prenons un exemple : imaginez un jeu d'aventure dans un univers fantastique médiéval. Nous avons donc des personnages tels que des sorciers, ou des chevaliers. Tous ces personnages peuvent être considérés comme des objets et donc des classes distinctes à l'intérieur du code. Pourtant, après reflexion, ils possèdent tous certains points communs. En effet, quelque soit la race du personnage, il possède obligatoirement des attributs comme un total de points de vie, une ou plusieurs armes ou

encore un degré d'expérience. Ces personnages auront également des méthodes communes comme la possibilité de boire une potion de vie, d'attaquer ou de se déplacer. Ainsi, le programmeur va tout d'abord coder une classe de base *humain* avec les différentes méthodes et attributs communs aux personnages. C'est là qu'intervient la notion d'héritage. Notre classe *sorcier* possèdera tous ces attributs et méthodes mais également, un nombre défini de points de magie (appelé mana) qui sera stocké dans une variable *pointsMana*, la possibilité de recouvrer ses points de magie avec la méthode *recupererPointsMana* mais aussi la possibilité de lancer de multiples sortilèges avec la méthode *lancerSortilege*. Concrètement dans le code, le programmeur n'aura plus qu'a définir les nouvelles méthodes spécifiques au sorcier et de dire que la classe sorcier *hérite* de la classe *humain* pour que l'objet final possède toutes les méthodes et attributs nécessaires à son bon fonctionnement. La notion d'héritage n'est pas limitée à une « génération », on peut imaginer des sorciers manipulant des éléments différents comme le feu l'eau et la terre, chaque nouveau personnage étant un nouvel objet qui hérite de la classe *Sorcier* elle-même héritant de la classe *Humain*.

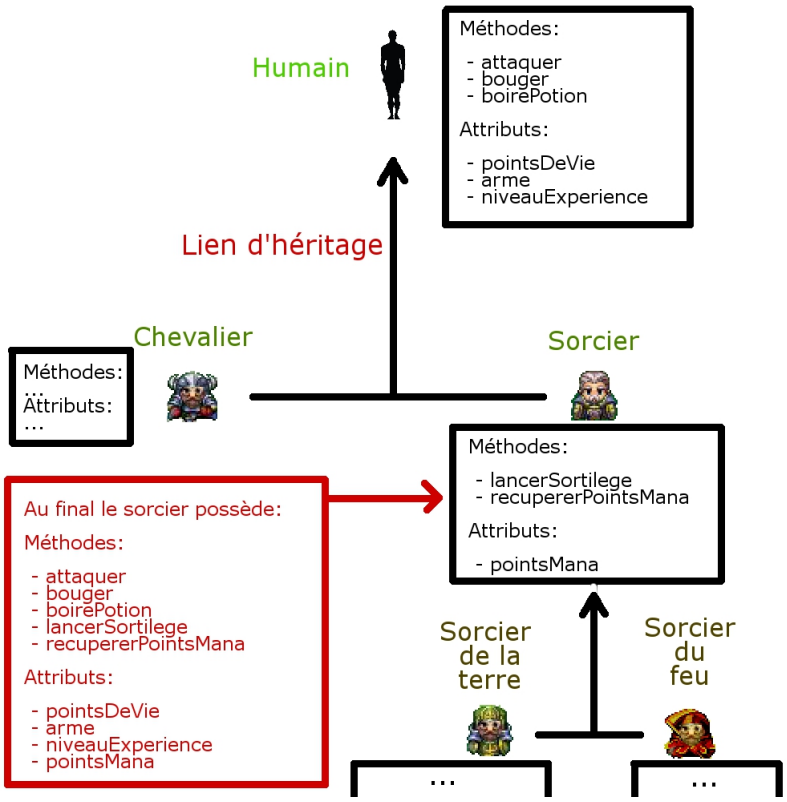

FIGURE 4.2 – Schéma illustrant les liens d'héritage entre plusieurs personnages d'un jeu vidéo.

4.2 Bibliothèques informatiques

En informatique il existe des outils que l'on appelle *bibliothèques logicielles* qui facilitent la tâche du programmeur dans la mesure où une grande partie du travail au niveau du code est déjà réalisée. Ces bibliothèques sont des groupement de fonctions utilisables directement par le programmeur sans qu'un travail supplémentaire ne soit requis. Lors de l'utilisation d'une bibliothèque, le code est intégré au projet et ensuite compilé par la machine dans l'exécutable final (autrement dit le programme en lui-même). Il existe un nombre gigantesque de bibliothèques et toutes celles que nous avons utilisées pour notre projet seront décrites dans les chapitre suivants. De base, le language C++ possède une bibliothèque dite standard qui pose certaines structures de base comme par exemple les différentes méthodes de stockage d'information dans la mémoire. Mais il existe des bibliothèques bien plus complexes comme *Qt* qui permet de créer rapidement des applications avec une interface

utilisateur avancée [1]. L'installation d'une bibliothèque se fait très rapidement puisqu'il suffit de télécharger les fichiers constituant ladite bibliothèque et d'indiquer au compilateur l'emplacement où sont sauvegardés les fichiers. Ainsi, à chaque fois que le programmeur désire utiliser une fonction propre à une bibliothèque, il lui suffit d'écrire au début du fichier source une ligne indiquant le nom des bibliothèques employées. Grâce à cette ligne le compilateur saura à quel endroit rechercher le code correspondant pour le compiler avec le projet à chaque fois que le programmeur employera une fonction spécifique. Ces lignes s'appellent des *includes* (voir première ligne du code ci-dessous).

Code C++

```
#include <iostream>
//La ligne ci-dessus indique au compilateur que nous désirons employer la bibliothèque
    iostream dans ce fichier par la suite, cette dernière est nécessaire pour afficher par
    exemple le message 'Hello World' sur la console !
int main()
{
  std::cout<<"Hello World !"<<std::endl;
  return 0;
}
```

Comme décrit dans le chapitre 1.1 de ce dossier, nous avons décidé de créer notre jeu à partir de bibliothèques distinctes ayant chacune un rôle bien précis. Chaque bibliothèque bénéficiera d'une section qui lui sera consacrée dans la suite de ce chapitre.

Tout d'abord, cherchons à identifier les quatre différentes tâches que notre programme doit exécuter. Notre programme étant intéractif, nous avons besoin d'un moyen pour communiquer avec l'utilisateur. Cela se fait par le biais des entrées/sorties. Sous ce terme sont regroupés tous les signaux que l'utilisateur peut envoyer au programme pour le contrôler ou que le programme peut lui retourner. Ainsi nous y trouverons des périphériques bien connus comme un clavier ou un écran. L'appui sur une touche déclenchera un signal qui sera interprété par le programme, ce signal est donc considéré comme une entrée. À l'inverse, le programme est capable d'envoyer des informations vers d'autres périphériques de la machine comme les hauts-parleurs pour jouer une musique ; cela constitue une sortie. La deuxième tâche, la plus ardue, est de calculer et d'afficher les rendus numériques à l'écran. C'est le moteur 3D IrrLicht qui se charge de ce travail. C'est la bibliothèque la plus importante de notre projet, elle se charge de définir quels objets doivent être affichés à l'écran ainsi que de toutes les « mathématiques dans l'espace » que comporte notre jeu, comme par exemple la distance séparant un joueur d'un ennemi. À cela viennent s'ajouter les services du moteur physique IrrBullet [2]. Cette bibliothèque comporte toutes les fonctions nécessaires pour gérer les collisions entre les différents objets du jeu ainsi que les calculs nécessitant l'usage de formules de physique comme par exemple le calcul de la décélération qui doit être appliquée à un objet en frottement avec le sol. Finalement, nous ferons usage d'un moteur de son qui nous aidera quand à elle à gérer l'émission de sons pouvant également être spatialisés. Ainsi, si dans notre jeu une voiture de police file à toute vitesse de droite a gauche de notre écran d'ordinateur, le rendu sonore sera calculé de sorte que le son de « déplace » avec l'objet. (le bruitage sera perceptible d'abord dans l'écouteur droit de l'utilisateur puis progressivement dans le gauche)

L'utilisation de toutes ces bibliothèques se fera tout simplement par l'appel de méthode spécifiques après avoir bien entendu indiqué au compilateur quels fichiers compiler à l'aide des *includes*.

1. L'interface utilisateur est l'ensemble des fenêtres et boutons dont l'utilisateur du programme se sert pour intéragir avec lui

2. Le véritable nom de ce moteur est Bullet Physics Engine, IrrBullet est en réalité une version de Bullet qui a été modifiée dans le but de s'adapter au fonctionnement et à la structure d'IrrLicht

4.2.1 IrrLicht

IrrLicht est de loin la bibliothèque la plus importante de notre projet. En effet, c'est un moteur 3D couplé à une gestion des entrées/sorties et ce sont ces deux éléments qui réaliseront la plus grande partie du travail au niveau de notre jeu. Si un jeu peut se passer de son ou de gestion avancée de la physique, on imagine mal un jeu sans rendu visuel et sans interaction avec le joueur. IrrLicht va nous permettre de charger facilement en mémoire des modèles 3D, des textures et des animations mais également de les manipuler dans un système à trois dimensions. Voici en exemple ci-dessous le code (simplifié) permettant d'initialiser le modèle 3D d'un ennemi fraîchement créé :

Code C++

```
//entête de la méthode
void CZombie::initZombie3D(scene::ISceneManager* smgr,video::IVideoDriver* driver)
{
    //stockage en mémoire du fichier texture
    m_zombTex = driver->getTexture("MODELS/ZOMBIES/zombie_tex.png");
    //stockage en mémoire du fichier 3D
    m_zombieNode = smgr->addAnimatedMeshSceneNode(smgr->getMesh("MODELS/ZOMBIES/zombie.b3d"
        ),0,IDFlag_IsPickable);

    //le 'triangle selector' nous permettra par la suite d'effectuer des calculs d'
        intersection avec des droites, comme par exemple la trajectoire et le point d'
        impact d'une balle sur notre modèle 3D de zombie.
    scene::ITriangleSelector* selector = 0;
    selector = smgr->createTriangleSelector(m_zombieNode->getMesh(), m_zombieNode);
    m_zombieNode->setTriangleSelector(selector);

    //toutes ces méthodes nous permettent de paramétrer le modèle 3D selon nos envies...
    m_zombieNode->setJointMode(EJUOR_NONE);
    m_zombieNode->setMaterialTexture(0, m_zombTex);          //...on lui applique une
        texture
    m_zombieNode->setScale(core::vector3df(1.6, 1.6, 1.6)); //...on défini une dimension
    m_zombieNode->setRotation(core::vector3df(0,0,0));       //...une inclinaison
    m_zombieNode->setMaterialType(EMT_SOLID);
    m_zombieNode->setMaterialFlag(EMF_FOG_ENABLE, true);
    m_zombieNode->setAnimationSpeed(50);                     //...animé à 50frames par
        seconde (cf. chapitre sur l'animation des modèles)
    m_zombieNode->setPosition(m_pos);                        //et finalement une position
    m_zombieNode->setLoopMode(true);

}
```

IrrLicht nous fournit également pléthores de méthodes et types de variable nous aidant dans la gestion de nos modès dans l'espace tridimensionnel, comme nous le montre l'extrait de code suivant qui réalise ce que l'on appelle un *raycast*. Le principe du *raycast* ou tir de rayon, est de créer un segment AB, A étant la position de la caméra et B le point qu'elle regarde, dans le but de contrôler si des objets s'y trouvent. Cela se révèle particulièrement utile quand il s'agit par exemple de vérifier si un ennemi a été touché par le tir du joueur.

Code C++

```

core::line3d<f32> rayTir; //création de la variable rayTir qui contiendra le segment décrit
     précédemment
rayTir.start = camera->getPosition();
rayTir.end = rayTir.start + (camera->getTarget() - rayTir.start).normalize() * 1000.0f;

if(m_shootingRay) //'m_shootingRay' devient vraie si le joueur appuie sur le bouton tirer.
  {
    m_shootingRay = false;

    //IrrLicht nous renvoye le modèle traversé par notre segment.
    ISceneNode* selectedSceneNode =collMan->getSceneNodeAndCollisionPointFromRay(rayTir,
        intersection,hitTriangle,0,0);
    IAnimatedMeshSceneNode* selectedMeshSceneNode = static_cast<IAnimatedMeshSceneNode*>(
        selectedSceneNode);

    //si le modèle touché est un ennemi, on lui inflige des dégâts
    if (NodeZombieMap.find(selectedMeshSceneNode))
```

```
    {
      CZombie* zombImpact = NodeZombieMap[selectedMeshSceneNode];
      //le montant des dégâts est donné par l'arme que le joueur utilise au moment de tirer
      zombImpact->recevoirDegats(armeJoueur->getDegats());
    }
  }
}
```

Mais au délà de la manipulation d'objet, de la gestion des entrées/sorties et la gestion des différents types d'objets nécessaires à un jeu en trois dimensions, IrrLicht nous permet avant tout d'obtenir le rendu visuel de notre scène[3] et de l'afficher sur l'écran du joueur. Le principe est le même qu'au cinéma où pour donner une impression de mouvement on affiche à l'écran 24 images chaque seconde[4] sauf que dans le cas d'un jeu vidéo les images ne sont pas stockées sur une bobine ou un disque dur ; il faut les créer sur le moment. C'est pourquoi on parle alors de *rendu en temps réel*, IrrLicht va déterminer quels sommets, faces et arrêtes doivent être affichées, cachées ou estompées par de l'ombre pour ensuite convertir ces informations en image numérique et ce, jusqu'à 60 fois par seconde. (Le nombre de 60 images par seconde est appelé le *taux de rafraîchissement* et il peut bien sûr dépasser la limite des 60 IPS si la puissance calculatoire le permet mais en général les écrans d'ordinateur ne dépassent pas ce taux, rendant ainsi le surplus inutile.)

4.2.2 IrrKlang

IrrKlang, comme peut le laisser supposer son nom est une bibliothèque développée spécialement pour fonctionner de pair avec IrrLicht. Elle regroupe une multitude des fonctions nous permettant de jouer de la musique et des bruitages ainsi que de leur appliquer des effets particuliers tels des échos ou des reverbérations. Mais IrrKlang se révèle particulièrement utile lorsqu'il s'agit de jouer des sons dans un espace en trois dimensions. En effet, nous sommes en mesure d'attribuer à chaque source sonore une position sur les trois axes de l'espace, IrrKlang s'occupant ensuite de définir à quel volume doit être joué le son dans chaque haut-parleur utilisé. La quasi-totalité des jeux vidéos en 3D utilisent cette fonctionnalité de nos jours car elle permet une meilleure immersion du joueur dans l'univers du jeu. La gestion des sons peut également être beaucoup plus poussée, notamment pour les jeux où l'ambiance sonore joue un rôle prédominant comme par exemple dans les jeux d'horreur. Ainsi, le sentiment de peur peut être instauré à l'aide de bruitages bien choisis comme un bruit de pas émit par une source se trouvant derrière le joueur. Le code légèrement modifié ci-dessous illustre l'utilisation des sons dans l'espace pour notre projet. Ici, des grognements sont émis aléatoirement pour chaque zombie présent dans le jeu.

Code C++

```
//un nombre aléatoire est généré
  int n = rand()%(100) +1;

//nous testons la valeur de ce nombre
  switch(n)
  {
//si elle vaut 1:
    case 1:
//nous jouons le bruitage intitulé 'm_frenzy1' à l'emplacement de la créature en question.
      m_soundEngine->play3D(m_frenzy1,m_tableZombies[m_parseurZombies]->getPos());
    break;
    case 2:
      m_soundEngine->play3D(m_frenzy2,m_tableZombies[m_parseurZombies]->getPos());
    break;
//et ainsi de suite
    case ...
  }
```

3. La scène n'est rien d'autre que l'environnement dans lequel sont affichés nos objets
4. Il est courant de retrouver l'abréviation *IPS* ou *FPS* pour Frame Per Second en anglais.

4.2.3 IrrBullet

IrrBullet est une bibliothèque conçue pour IrrLicht utilisant et adaptant les fonctions d'une bibliothèque bien plus grande qui quant à elle est destinée à des supports beaucoup plus variés. (Réalisation de films, de jeux vidéos pour consoles de dernière génération ou même pour des applications intégrées aux navigateurs Web.) L'utilisation de cette bibliothèque dans notre projet était facultatif mais nous avions à coeur de posséder une gestion avancée de la physique dans le jeu. Nous apréciions particulièrement le degré de réalisme que cette bibliothèque ajoute à notre travail, notamment pour les rebonds des objets mobiles lorsque le joueur tire dessus. De plus, son aide se révèle précieuse pour sa gestion des collisions bien plus sophistiquée que celle présente de base avec IrrLicht. La gestion des collisions est ce qui empêche tel modèle 3D d'entrer dans un autre, c'est donc grâce à IrrBullet que le joueur ou les créatures ne peuvent pas passer à travers les murs de la reconstitution 3D du gymnase

4.2.4 Mise en relation des bibliothèques

Toutes les bibliothèques décrites précédemment seraient inutiles si elles étaient prises séparément. C'est grâce à la programmation en C++ que nous pourrons créer des liens entre elles. Tel un poste de commande centralisé, nous avons écrit dans notre projet toutes les lignes de code et les relations qu'elles doivent avoir entre elles pour utiliser au mieux toutes les fonctionnalités des différentes bibliothèques. Illustrons ces propos avec un exemple bien concret ; celui du joueur cliquant sur le bouton gauche de sa souris en visant une créature du jeu. Vous trouverez dans l'annexe B les différentes parties simplifiées et commentées de code qui correspondent aux évènements ci-dessous.

1. Tout d'abord le système d'entrées/sorties va envoyer un signal indiquant que l'utilisateur a généré un événement. Un rapide test nous permettra de déterminer qu'il s'agit d'un événement de type *souris* puis un nouveau test nous indiquera qu'il s'agit précisément du clic gauche.
2. Connaissant maintenant l'action entreprise par le joueur il convient de tester si ce dernier est autorisé à tirer. Le cas inverse peut arriver si le joueur est à cours de munitions ou si il clique plus vite que la cadence de tir de son arme ne le permet.
3. Si tous les tests ont été passé avec succès le programme exécute des actions « simples » comme animer les modèles 3D de l'arme et de la main, retirer une munition au total possédé par le joueur, jouer un bruitage de tir d'arme à feu, créer une douille qui sera ensuite projetée dans le décors et finalement afficher un flash à l'écran.
4. Ensuite, nous devrons tester si le joueur à touché quelque chose en tirant. Nous procédons à un raycast [5] qui nous retourne si un objet a été traversé par le rayon et si c'est le cas, quel est le type de cet objet. (dans notre cas le type sera un objet Zombie puisque nous indiquions précédemment que le joueur visait une créature)
5. Arrivé à ce stade, le programme va procéder à une modification de la variable contenant les points de vie de la créature et teste si le total tombe en dessous de zéro. Si tel est le cas, la créature meurt et le programme efface l'objet, libérant au passage un peu de mémoire.
6. Finalement, le score du joueur sera augmenté et si ce score dépasse le meilleur score jamais atteint auparavant par le joueur, l'ancien *highscore* sera effacé et remplacé par le nouveau dans un fichier texte externe au programme, mais situé tout de même sur le disque dur, aux cotés des autres fichiers du jeu.

Nous voyons maintenant la complexité que représente la programmation dans un tel projet mais également la puissance et la liberté qu'elle représente. Elle nous permet en quelque sorte d'orchestrer à nos souhaits les centaines (voire les milliers) de fonctions que nous avons créées ou que nous avons à disposition dans nos différentes bibliothèques.

5. (c.f chapitre 4.2.1 sur IrrLicht)

4.3 Aide

4.3.1 Forums & communauté

Pour parvenir à l'aboutissement de notre travail de maturité, il nous a bien sûr fallu de l'aide et des explications. Notamment pour ce qui s'agit des difficultés que nous avons rencontrées avec nos différentes bibliothèques informatiques ou nos logiciels. Ces difficultés ont toutes pu être résolues soit par nos propres soins à force d'essais répétitifs, soit grâce à de l'aide trouvée sur des forums informatiques. Lorsque des problèmes engendrés par des techniques plus sophistiquées faisaient surface, il devenait alors nécessaire d'exposer le sujet directement sur les groupes d'entraide spécifiques à chaque bibliothèque ou moteur informatique. Heureusement, sur les forums, l'ensemble des membres n'hésitaient pas à prendre un peu de leur temps libre pour nous venir en aide. Cependant, il est arrivé que certaines complications se soient révélées sans issue et il a fallu à ce moment là, non pas chercher comment résoudre le problème, mais comment le contourner. Comme ce fut le cas lors de l'exportation de nos modèles réalisés avec Blender vers le format *Blitz3D*, où les animations créées étaient déformées et parfois même totalement supprimées.

Ces forums sont donc notre principale source d'aide. Mais cela ne s'arrête pas là, c'est également ces groupes de passionnés qui rendent l'informatique en général accessible à l'utilisateur « lambda ». Ils dynamisent et maintiennent l'esprit d'entraide indispensable à tout débutant quel que soit le domaine qui l'intéresse. C'est donc naturellement qu'avec l'expérience acquise au fil de notre travail, nous avons pu inverser les rôles pour, à notre tour, rendre service en aidant les « nouveaux ». C'est en quelque sorte nos remerciements envers les personnes qui mettent à disposition de chacun leur temps et leurs connaissances.

Chapitre 5

Conclusion

La réalisation de ce projet a été pour nous un travail de longue haleine puisque nos premiers essais datent d'il y a environ de deux ans. Nous en n'étions alors qu'au stade de prise en main des différents logiciels et à la lecture des nombreux tutoriels présents sur internet. N'ayant donc aucune connaissance en la matière lorsque que nous avons débuté, nous ne savions pas vraiment à quoi nous attendre pendant la réalisation de notre travail. Par conséquent, nous n'avions pas de « processus » précis pour relever le défi car nous ignorions encore quelles étaient les tâches que nous étions à même de mener à bien. La réalisation de notre projet s'est révélée ardue, souvent même frustrante mais ô combien instructive au final. Nous avons avancé par « tâtonnements », d'abord en essayant des morceaux de codes, puis en les modifiant, pour bien souvent finir par les effacer et tout recommencer.

Avec du recul, nous éprouvons toutefois un petit sentiment de frustration. Un peu à cause de l'impression qu'il aurait été possible de faire encore mieux. En effet, avec le temps d'apprentissage que représente le développement de ce projet, nous avons énormément progressé dans les différents domaines auxquels il touchait. Ainsi, alors qu'au début l'accomplissement d'une certaine tâche inspirait en nous fierté et satisfaction, cette même réalisation nous semble imparfaite maintenant. De plus, n'étant que deux à travailler sur le projet (une broutille en comparaison des équipes de professionnels) nous avons dû distribuer notre temps qui était limité pour chaque fonctionnalité de notre programme.

Quelques exemples : Le joueur ne peut pas manier plus de deux armes différentes dans le jeu alors que le code que nous avons conçu a été prévu pour en gérer 4 et que nous en avions modélisé 6, chacune en 3D. Ou encore l'absence d'objet permettant de regagner ses points de vie alors que le code a été construit de telle manière à ce que l'implémentation de cette fonctionnalités soit aisée et que l'objet 3D en question a déjà été modélisé. Pour résumer, notre programme possède tous les mécanismes d'un jeu complet, mais uniquement le strict minimum au niveau du contenu. C'est là une concession que nous avons dû réaliser. En effet, se rendre au cœur de l'intéraction entre les différentes bibliothèques pour en comprendre le fonctionnement (c.f chapitre 1.1), demande énormément de temps pour obtenir un résultat concret. Notre jeu n'a donc pas la prétention de se distinguer par son originalité ou sa beauté mais plutôt par le travail et la recherche fournis dans le but de comprendre et reproduire le fonctionnement d'autres jeux vidéos. Ainsi notre projet compte plus de 3'800 lignes de code, 150 fichiers servant à son bon fonctionnement tels que des images numériques, des enregistrements audio ou encore des modèles 3D, le tout représentant plusieurs centaines d'heures de travail. De plus, ce total n'inclut pas les efforts supplémentaires qui ont été fournis pour mettre sur pied le site internet du projet ainsi que la rédaction de ce rapport.

Le choix de la méthode de travail que nous avons fait s'est révélé extrêmement instructif et on peut sans crainte dire qu'il a rempli sa mission. Nous espérons que vous avez apprécié, tout comme nous, la découverte des nombreux mécanismes qui fonctionnent de façon complexe au sein d'un logiciel de jeu en 3D. Vous avez maintenant de nouvelles connaissances qui vous permettront de porter un regard de connaisseur sur les images provenant du monde des jeux vidéos. Pour nos futures réalisations, individuelles ou communes, il serait intéressant de manipuler directement un moteur de jeu pour passer outre l'aspect purement technique des rendus, de la physique et des sons et se concentrer sur la qualité intrinsèque du jeu, le rendre original et innovant. C'est très probablement dans cette direction que nous allons nous diriger maintenant.

Chapitre 6

Bibliographie

1. Liste des sites internet nous ayant apporté des connaissances indispensables :
   (a) (***siteduzero.com***)
   Site proposant des cours et explications gratuits sur de très nombreux sujets touchant à l'informatique. Les cours ont une difficulté croissante, les rendant ainsi abordables même pour les néophytes en la matière.
   (b) (***irrlicht.sourceforge.net***)
   Site hébergeant le projet du moteur 3D IrrLicht ainsi que le forum rassemblant ses utilisateurs.
2. Liste des sites internet ayant fourni les ressources de notre projet :
   (a) (***ambiera.com/irrklang***)
   Site hébergeant le moteur de son IrrKlang.
   (b) (***sourceforge.net/projects/irrbullet***)
   Site hébergeant le moteur physique IrrBullet, voir *bulletphysics.org* pour le moteur physique original.
   (c) (***cgtextures.com***)
   Site proposant de très nombreuses textures de grande qualité.
   (d) (***universal-soundbank.com & sound-fishing.net***) Deux sites proposant un large choix d'effets sonores.
   (e) (***jeffroussakis.wordpress.com***)Mais également la texture des créatures créée par Jeff Roussakis pouvant être trouvée en haute résolution sur son site.

Table des figures

1.1 Interface d'un moteur de jeu tel que Unity3D 3

2.1 Interface de Blender avec la fenêtre de rendu à droite 5
2.2 Capture d'écran du jeu Doom II (1994) 6
2.3 Exemple d'utilisation d'un blueprint avec multiples points de vues 7
2.4 Amélioration du réalisme du modèle avec texture et normal map 8
2.5 La normal map d'une des armes présentes dans le jeu 8
2.6 Un des possibles développement du cube 9
2.7 Modèle final obtenu après texturage. 9
2.8 L'animation d'un zombie, visible dans notre jeu avec Blender 10

3.1 Plan du deuxième étage du gymnase de Burier 13
3.2 Quelques étapes de le conception de l'image servant de minimap 13
3.3 Affichage des différents éléments du *HUD* à l'écran pendant une partie. 14

4.1 Exemple d'instructions en assembleur 15
4.2 Schéma illustrant les liens d'héritage entre plusieurs personnages d'un jeu vidéo. . . . 17

A.1 Utilisation du *blueprint* pour tracer la silhouette générale. 26
A.2 Constitution du volume à partir de la silhouette. 27
A.3 Application des textures avec l'UV mapping. 27

C.1 Placement des lignes de découpe (en orange sur la première image) du modèle en vue du texturage puis mise en place du développement du modèle 3D sur la texture. . . . 31

Annexe A

Modélisation du Gymnase de Burier

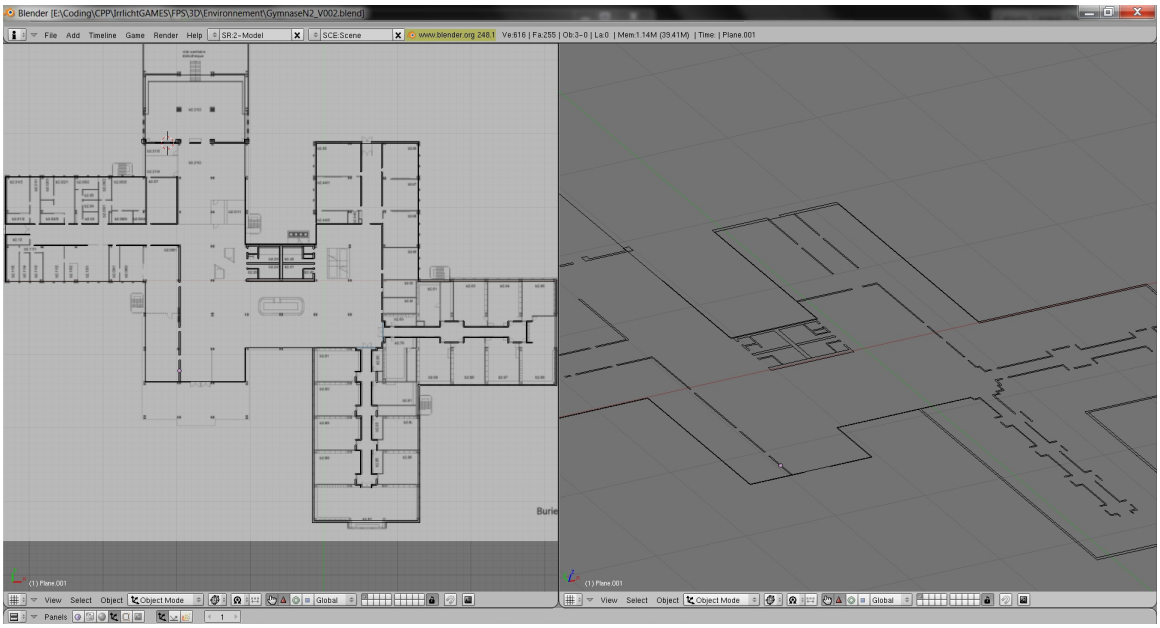

FIGURE A.1 – Utilisation du *blueprint* pour tracer la silhouette générale.

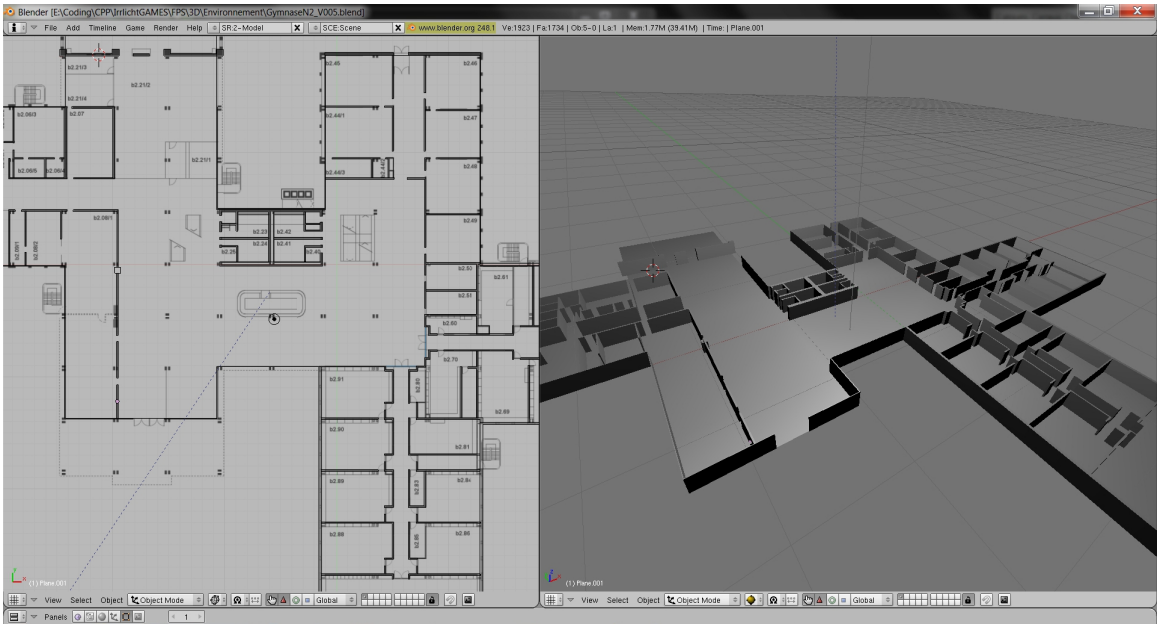

FIGURE A.2 – Constitution du volume à partir de la silhouette.

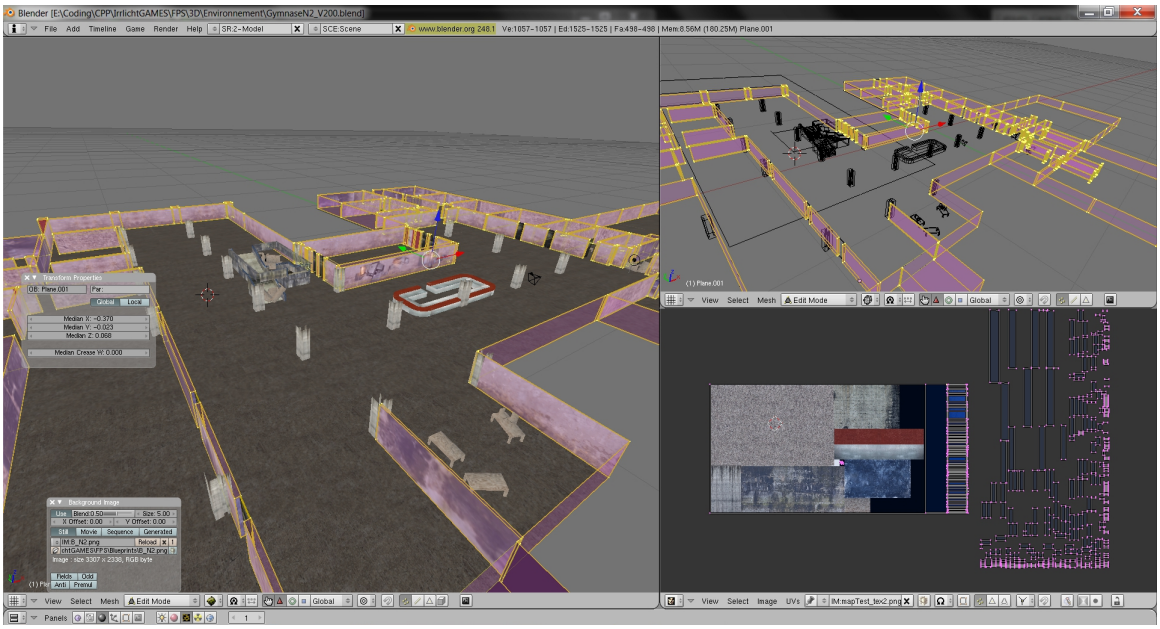

FIGURE A.3 – Application des textures avec l'UV mapping.

Annexe B

Conséquences d'un clic de souris

Réception de l'événement généré par le joueur, les commentaires explicatifs sont de couleur verte :

Code C++

```
bool CFps::OnEvent(SEvent const& event)
{
    //il s'agit d'un event concernant un appui/relâchement de touche
    if(event.EventType == EET_MOUSE_INPUT_EVENT)
    {
        switch(event.MouseInput.Event)
        {
            //si le bouton gauche de la souris a été pressé:
        case EMIE_LMOUSE_PRESSED_DOWN:
              //si le joueur n'est pas a court de munitions et qu'il n'est pas entrain de
                  recharger:
            if(m_joueur->getArme(m_armeActuelle)->getMunitions()>0
               && !m_joueur->isReloading())
            {
                //on définit que le joueur est entrain de tirer
                m_joueur->setShooting(true);
                return true;
            }
            else if(m_joueur->getArme(m_armeActuelle)->getMunitions() == 0)
            {
                m_soundEngine->play2D(m_outOfAmmoSound);
            }
            break;
         }
     }
}
```

Gestion des différents éléments liés au tir :

Code C++

```
//si le joueur a été défini comme étant entrain de tirer:
if(m_joueur->isShooting()&& m_joueur->getDeltaTempsTir()>armeJoueur->getCadenceTirArme())
{
  //nous attribuons la valeur 'vraie' à la variable qui détermine si oui ou non nous devons
      procéder à un raycast
  m_shootingRay = true;

  //si l'arme du joueur n'est pas déjà animée lors du tir
  if(armeJoueur->isLooped() == false)
  {
    switch(m_armeActuelle)
    {
      //[...]
    }
    //on anime l'arme et la main simultanément
    meshArmeJoueur->animateJoints();
    meshMainDroiteJoueur->animateJoints();
  }

  //[...]
  //on retire une munition au total
  m_joueur->getArme(m_armeActuelle)->majMunition(-1);

  //on éjecte une douille, juste pour l'effet visuel
  m_tableBodiesDouilles.push_back(ejecterDouille(m_repereDouilles->getAbsolutePosition(),
      m_repereDouilles2->getAbsolutePosition() ,m_vel));
  m_parseurDouilles++;

  //pour ne pas trop encombrer la mémoire si le nombre de douilles dépasse 30, on supprime
      la plus vieille
  if(m_parseurDouilles > m_maxDouilles)
  {
    m_tableBodiesDouilles[m_parseurDouilles-m_maxDouilles]->remove();
  }

  //on joue un bruitage en fonction de l'arme utilisée
  switch(m_armeActuelle)
  {
    case 1:
      m_soundEngine->play2D(m_berettaShootSound);
    break;
    case 2:
      m_soundEngine->play2D(m_uziShootSound);
    break;
    default:
      m_soundEngine->play2D(m_berettaShootSound);
    break;
  }
}

//[...]
```

Réalisation du raycast :

Code C++

```
//création de la variable qui va contenir notre raycast
core::line3d<f32> rayTir;
rayTir.start = camera->getPosition();
rayTir.end = rayTir.start + (camera->getTarget() - rayTir.start).normalize() * 1000.0f;

if(m_shootingRay)
{
  m_shootingRay = false;

  //l'objet traversé par le rayon est stocké dans la variable 'selectedSceneNode'
  ISceneNode* selectedSceneNode =collMan->getSceneNodeAndCollisionPointFromRay(rayTir,
      intersection,hitTriangle,0,0);

  //[...]

  if (NodeBodyMap.find(selectedSceneNode))
  {
    //[...]
  }

  //si l'objet touché est une créature:
  if (NodeZombieMap.find(selectedMeshSceneNode))
  {
    CZombie* zombImpact = NodeZombieMap[selectedMeshSceneNode];
    //on lui inflige des dégâts
    zombImpact->recevoirDegats(armeJoueur->getDegats());

    int n = rand()%(3) +1;
    //on joue un bruitage qui indique que la créature est touchée
    switch(n)
    {
      case 1:
        m_soundEngine->play3D(m_hit1,intersection);
      break;
      case 2:
        m_soundEngine->play3D(m_hit2,intersection);
      break;
      case 3:
        m_soundEngine->play3D(m_hit3,intersection);
      break;

    }

  }
}
```

Test des points de vie restants de la créature :

Code C++

```
bool CZombie::isAlive()
{
  if(m_HP < 1)
  {
    m_zombBody->remove();
    return false;
  }
  else
  {
    return true;
  }
}
```

Annexe C

Texturage d'un modèle complexe

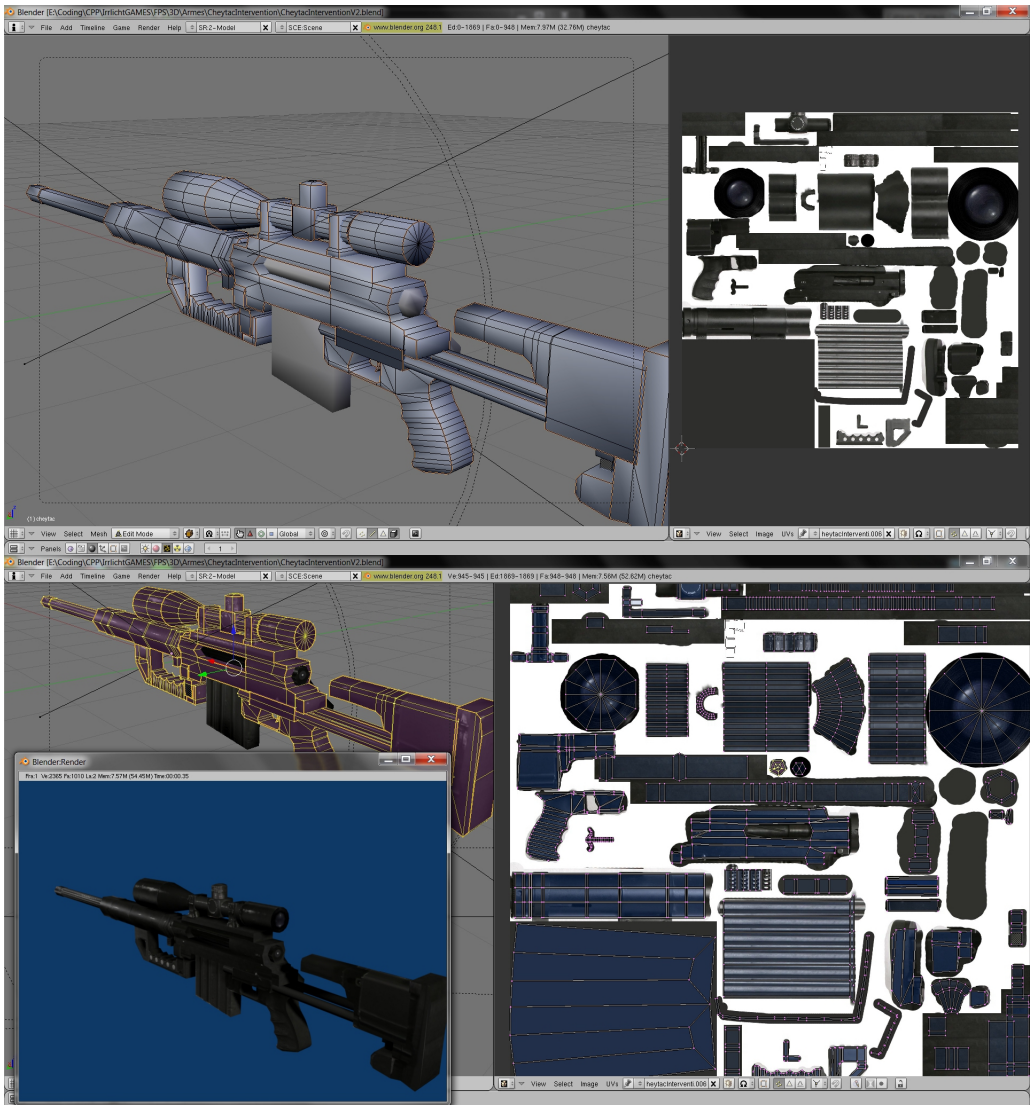

FIGURE C.1 – Placement des lignes de découpe (en orange sur la première image) du modèle en vue du texturage puis mise en place du développement du modèle 3D sur la texture.