

Underground Project Section - UPS

yaiche_c - Arys

gaulti_b - Stin

gayral_d - b95093cf

labadi_t - Feeloun

Rapport de soutenance finale

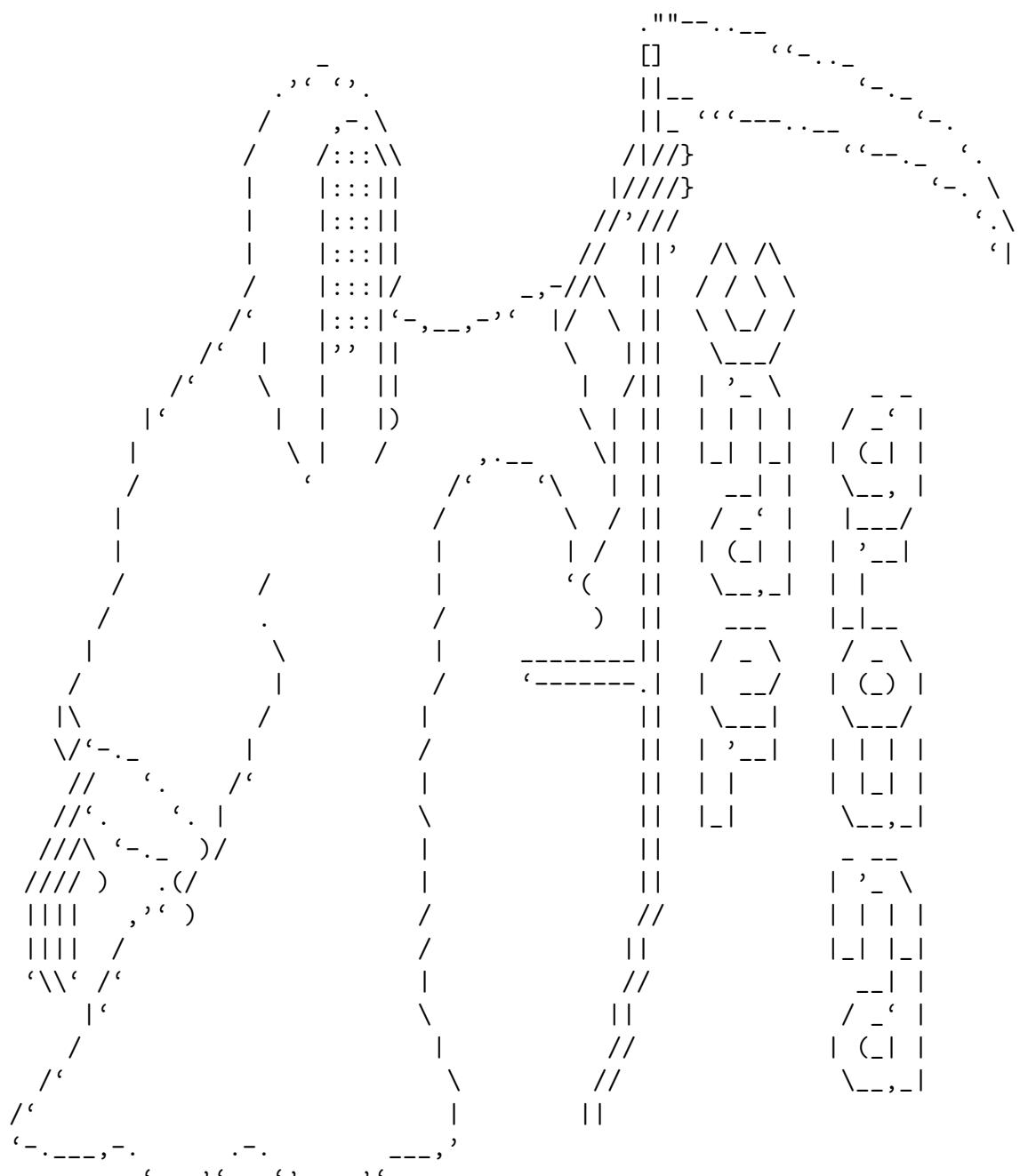


Table des matières

1	Introduction	3
2	Calendrier du cahier des charges	5
3	Charles <i>aka Arys</i>	6
1	Le moteur son	6
2	Le labyrinthe	8
4	Denis <i>aka b95093cf</i>	12
1	Le moteur graphique	12
2	La lumière	18
3	Les évènements	24
4	Interface utilisateur	25
5	Thomas <i>aka Feeloun</i>	27
1	Le menu	27
2	Les collisions	29
3	Le site web	32
6	Bastien <i>aka Stin</i>	34
1	Le choix du logiciel de modélisation 3D	34
2	La création de l'environnement	35
3	La création des monstres et les animations	40
7	Conclusion	43

CHAPITRE 1

Introduction

La soutenance 3.. enfin !

Afin de faire une petite piqûre de rappel, nous avions à la deuxième soutenance :

- Une importation d'environnement en 3D fluide avec peu de lag
- Un moteur sonore performant pour une ambiance d'horreur
- Des collisions fluide en utilisant les bounding box
- Un début de menu
- Une lumière simple : un halo autour de la caméra donnant un bon effet angoissant

Et pour la soutenance 3, nous avions prévu :

- Une gestion de monstre
- Un menu fini
- Un environnement créé aléatoirement
- Un affichage optimisé

... Mais en réalité, nous avons même de l'avance !

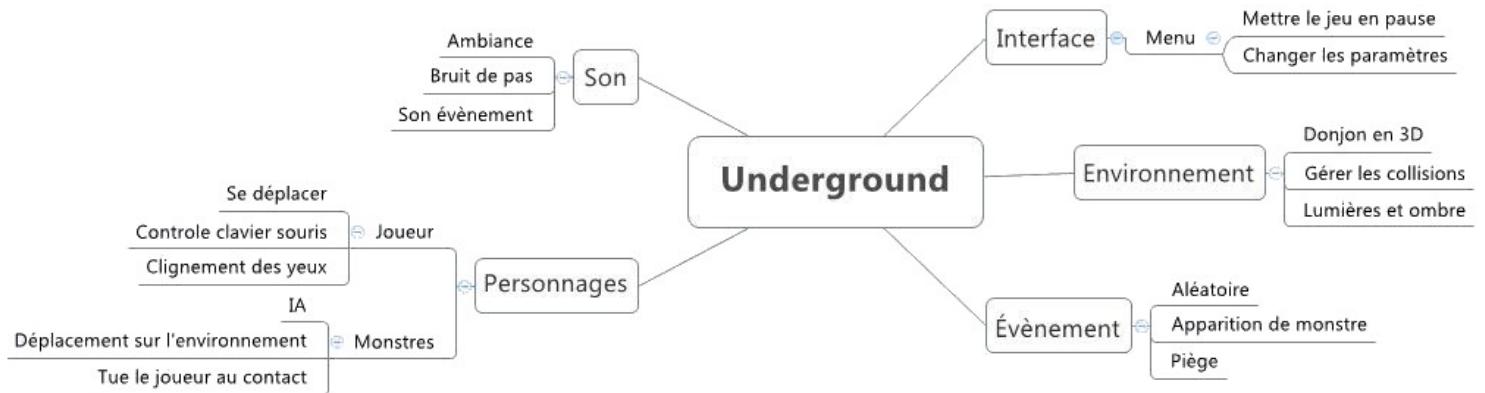
On a en effet réalisé :

- Un nouvel environnement qui est un labyrinthe parfait généré aléatoirement
- Optimisé au maximum l'affichage (résolution des leak de mémoire dûs à SharpDX qui ne possède pas de garbage collector)
- Ajouté des animations des menus, en plus de la gestion des résolutions et du volume
- Amélioré la lumière dans le style du jeu c'est à dire sous forme de lampe torche donnant un bon effet sur l'environnement
- Implémenté un clignement d'yeux pour ajouter un effet que le joueur ne contrôle pas totalement

En résumé... on est content ! Et on va vous expliquer les peines à travers lesquelles nous sommes passées dans les prochaines pages !

CHAPITRE 2

Calendrier du cahier des charges



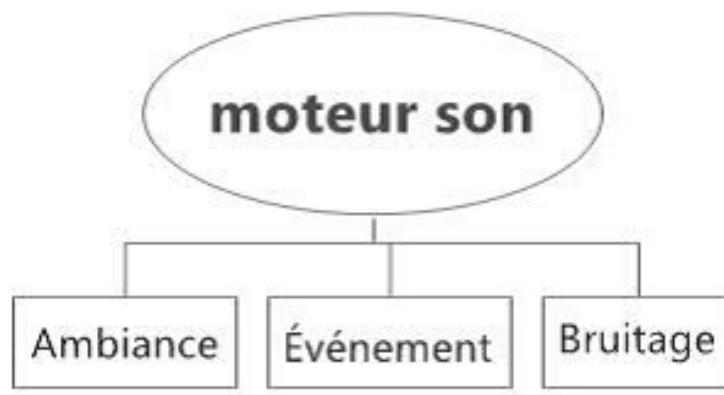
Soutenance	État du projet	État des membres
1	Présence d'un menu et d'une salle, implémentés avec DirectX	Stable
2	Gestion du son, finalisation de l'environnement et ajout des collisions	Drogué au café
3	Déplacement du personnage, clignements d'yeux, gestion d'événements et de lumières	Mauvais
4	Mise en place de l'ambiance et du scénario	Zombie
Bonus	Générateur d'évènement et d'environnement aléatoire, amélioration des textures	The Walking Dead

CHAPITRE 3

Charles aka Arys

1 Le moteur son

Il été découpé en 3 grandes parties très importantes :



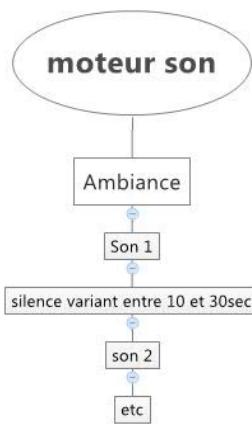
L'ambiance

Les musiques d'ambiance sont là pour renforcer l'immersion créée par les fonds sonores.

Bien sûr, ce n'est pas du rock, ni du rap ni du classique. Elles sont en général assez calmes mais avec un son très grave, très lourd. Bien évidemment, plus elles sont de bonne qualité, plus l'ambiance est renforcée.

Le volume des musiques dans un Survival Horror n'a pas besoin d'être haut, les musiques doivent plutôt s'apparenter à une musique de fond pour ne pas perturber ou masquer les fonds sonores d'évènement et les bruitages.

Dans Underground, le moteur son gère plus de 8 musiques d'ambiance de temps variant entre 1 et 5 minutes lancées dans un ordre aléatoire. Les



musiques sont bien évidemment angoissante, et bien que peu répétitives elles forcent le joueur à se concentrer sur l'ambiance générale.

De plus entre les musiques il y a un silence de temps variant entre 10 et 30 seconde pour rajouter un effet de tension et d'immersion, renforcé par la mise en place de ce que j'appelle des sons d'évènement ainsi que des bruitages.

Les sons d'évènement

Ce que j'appelle son d'évènement peut s'apparenter aux bruitages mais je fais une distinction entre les deux, surtout car ils ne s'implémentent pas de la même manière.

Ces fond sonore sont des sons très courts qui apparaissent de manière complètement aléatoire, le joueur aura donc tendance à avancer prudemment. Le vent fait aussi un très bon fond sonore, celui qui souffle dans les branches des arbres par exemple. On peut aussi penser à des cris de personnes se faisant attaquer.

Cela dit, une pièce complètement vide où on n'entend absolument rien peut aussi avoir un coté très angoissant (en général, la pièce en question est louche !) d'où les 10 à 30 seconde de suspension de musique !

Les fonds sonores ont en effet tendance à plonger assez facilement le joueur dans l'ambiance du jeu.

Underground ne compte pas moins de 18 fonds sonore allant du souffle du vent, des cris rauques de bêtes, des sons plus gores et inquiétants, des gouttes d'eau qui, placées au bon moment de silence, rendent le joueur très nerveux.

Les bruitages

Dans Underground les bruitages consistent surtout en le bruit de pas du joueur, des bruits de pas très lourd, un peu étouffes mais qui résonnent quand même sur la pierre dure de l'environnement, ils sont réalistes et de bonne qualité.

Autres bruitages assez intéressant et qui contribuent à l'atmosphère angoissante : ce sont les bruits de pas et les cris des ennemis.

Prenons un nouvel exemple : vous arrivez dans un couloir et entendez des bruits de pas autres que les vôtres, alors que vous ne voyez pas de monstres, c'est alors que les pas s'accélèrent puis soudain plus rien... si ce n'est un cri strident qui fera se dresser les cheveux de n'importe qui.

2 Le labyrinthe

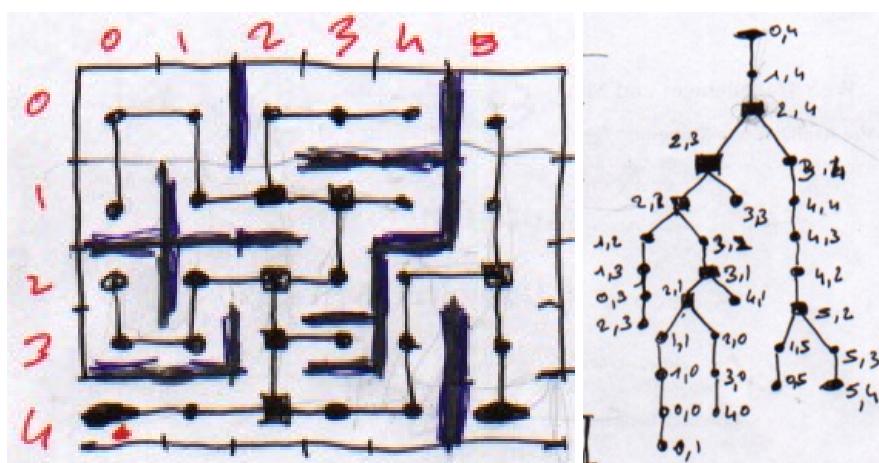
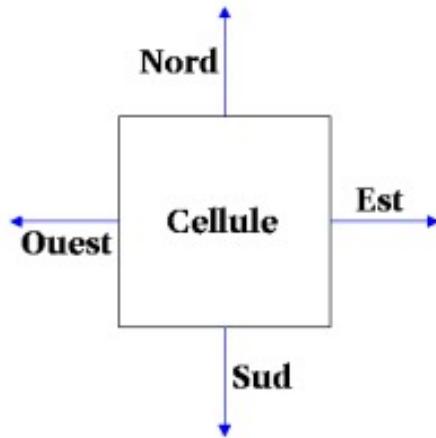


FIGURE 3.1 – Sous forme de tableau à deux dimensions et sous forme d'arbre binaire

Un labyrinthe rectangulaire parfait de y colonnes par x lignes est un ensemble de $x * y$ cellules reliées les unes aux autres par un chemin unique.

Chaque cellule est reliée à toutes les autres et, ce, de manière unique. L'intérêt est qu'il n'y a aucune perte de place au niveau du tableau.

Étant donné que nous parcourons les pièces, il me paraît plus simple, à priori de modéliser les cellules plutôt que les murs.



En première approche on pourrait comparer les ouvertures à des passages et de voir les portes comme des liens vers d'autres cellules.

Chaque cellule serait composée de 4 lien. Une porte serait un lien vers une cellule voisine, et un mur un lien nul. Nous avons besoin de coder l'état des murs Ouvert/Fermé. Il y a 4 portes par cellule, et chaque porte n'a que deux états. Nous n'avons donc besoin que de 4 int par cellules ainsi qu'un booléen qui définirais si la case a déjà été visitée.

Notre classe pour les cellules possède donc à première vu cinq attributs : le nord, l'est, l'ouest, le sud, et un booléen indiquant si elle a été visitée.

L'algorithme de création est plutôt simple, on part d'une case donnée, on la marque comme étant visitée et on passe à l'une de ses cases voisines(nous devons donc ajouter une liste de cases voisines aux attributs des cellules) de manière aléatoire en "cassant" les murs, puis on rappelle la fonction avec la nouvelle case.

La nature de cet algorithme est donc purement récursif.

Comme une démonstration vaut mieux que mille explications : Voici l'algorithme en langage Yaiche Ici nous nous trouvons dans le tableau bidi-

mensionnel MAZE comportant les attribut maze.length.x ainsi que maze.length.y, ceci étant la taille du tableau selon x et y, et MELANGER : une procédure qui mélange aléatoirement les éléments d'une liste.

Marquer toutes les cellules comme non visitées

Choisir une cellule C aléatoire et la marquer comme visitée

Procédure generate (Cell C) :

 Marquer C comme visitée

 CList <- melanger(cellules voisines de C)

 Pour toute cellule C2 de CList faire :

 Si (C2 n'est pas visitée) :

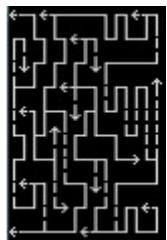
 Ouvrir la cloison la liant avec la cellule précédente
 generate(C2)

 Fin Si

Fin Pour

Fin Procédure

Cela nous donne ainsi en console :



Les flèches représentent des culs de sac, et le chemin est représenté par le trait blanc, décomposé ainsi, nous pouvons identifier chacune des cases pour les "tridimensionnaliser".

La visualisation sous forme d'arbre binaire nous permet bien de mettre en évidence le défaut majeur du labyrinthe parfait : il suffit d'un parcours main gauche pour en venir à bout.

C'est pourquoi pour la quatrième soutenance je me tournerai vers le second type de labyrinthe : le labyrinthe imparfait ou labyrinthe à îlot. Comme son nom l'indique : le labyrinthe à îlot possède plus d'un chemin pour arriver au résultat, mais aussi plus d'une manière de se perdre car au lieu d'une représentation d'arbre binaire, le labyrinthe imparfait utilise les graphes.

On crée une procédure qui va parcourir le tableau aléatoirement, pour le nombre de murs cassés on utilisera la formule `nombreMurCassé = (maze.length.x*maze.length.y)/2` ainsi qu'un nombre aléatoire (`rand`) qui correspond à l'orientation de la destruction du mur (nord/sud ou est/ouest).

(pour plus de lisibilité on utilisera la procédure `OUVERTUREMURCASSÉ(nord/sud)()` qui comme son nom l'indique « casse un mur » ainsi que la fonction `RANDOM(min,max)` qui renvoie un nombre aléatoire compris entre min et max)

Ainsi on a :

```
Pour i == 0 jusqu'à nombreMurCassé faire
    posX = random(0, maze.length.x -1) // -1 pour éviter de casser à
                                         // l'extérieur du labyrinthe
    posY = posX = random(0, maze.length.y -1)

    si rand%2 == 0 faire
        ouvertureDeMur(nord/sud)
    sinon
        ouvertureDeMur(est/ouest)
    fin si
fin pour
```

CHAPITRE 4

Denis aka b95093cf

1 Le moteur graphique

Le choix du format de fichier 3d

Le choix du format de fichier 3D a été un point névralgique du projet car les fonctions de traitement et d'import de fichiers 3D sont encore en développement par SharpDX, il nous a donc fallu créer notre propre « loader ».

Nous avions 3 formats préférentiels :

- **Le .vmf ou le .map** : Ce format est celui utilisé par la plupart des jeux développés par Valve Corporation (par exemple les séries Half-Life / Portal / Counter Strike donc une communauté des plus importantes).

Ayant déjà « mappé » pour ces jeux, j'ai une bonne connaissance des particularités de ce format, dont son système d'entités pour indiquer des mouvements d'objets, des déclencheurs, des systèmes logiques, des lampes, etc, est simple et m'est familier, de plus ayant déjà utilisé le logiciel de modélisation dédié à ce format, je pouvais obtenir très rapidement des fichiers 3D de test et ainsi implémenter ce format beaucoup plus rapidement.

Ce format a malheureusement été écarté car le nombre de sommets définissant une face pouvait être différente de 3 or même si certaines cartes graphiques peuvent gérer jusqu'à 4 sommets par faces, il est d'usage de ne pas envoyer des faces à plus de 3 sommets.

- **Le .ms3d (pour MilkShake3D)** : Lorsque nous avons commencé à nous documenter sur SharpDX, dans la mesure où la seule page de la documentation qui n'était pas « TBD » (To Be Done) était le disclaimer, nous avons appris à utiliser SharpDX sur le tas, en disséquant

des samples, et l'un d'entre eux chargeait un fichier .ms3d, un format très répandu.

Ce format a aussi été écarté car au moment de coder notre loader, nous cherchions désespérément à afficher un cube en entrant les coordonnées des sommets à la main et donc nous ne pouvions pas nous permettre de composer avec un format de fichier 3D qui serait binaire.

○ **Le .obj (pour waveform 3d OBject file)** : Un spé nous avait conseillé ce format sous prétexte qu'il était simple et il s'est avéré qu'il était effectivement très adapté pour notre besoin (qui était pour rappel d'afficher un simple cube).

C'est donc ce format qui a été retenu du fait de sa structure simple, et cette simplicité permet aussi à de nombreux logiciels de modélisation (par exemple Blender, ou la référence dans le domaine du level design : 3D Studio Max) d'exporter nativement ou via des plugins dans ce format.

Lire le format OBJ

Un fichier OBJ s'organise ainsi :

- Début du fichier
- Liste de **coordonnées de sommets**, chaque coordonnée de sommet étant identifié par le mot clé "v" suivi par, vu que l'on travaille dans un espace tridimensionnel 3 valeurs décimales (une pour l'axe X, une pour l'axe Y, et une pour l'axe Z) puis un retour à la ligne.
- Liste de **coordonnées de texture**, chaque coordonnée de texture étant identifié par le mot clé "vt" suivi cette fois ci de 2 valeurs décimales puisqu'on applique une texture en 2 dimensions sur une face bien précise donc en 2 dimensions ici aussi, et les coordonnées de textures vont servir à étirer, orienter, n'utiliser qu'une parcelle de la texture. Nous avons bien évidemment un retour à la ligne après ces deux valeurs.
- Liste de **normales**, chaque normale étant identifiée par le mot clé "vn" suivi de 3 valeurs qui représentent un peu comme pour la définition d'un sommet, des coordonnées du vecteur et avec le retour à la ligne. Les normales servent via un simple produit vectoriel à avoir une lumière assez réaliste, ce que nous verrons dans un prochain chapitre.

- Liste de **faces**, chaque face étant identifiée par le mot clé ”f” suivi de 3 groupes de 3 valeurs entières (donc 9 valeurs entières en tout) et le séparateur des valeurs dans un même groupe est le slash (/), donc rappelons-le, une face doit posséder 3 sommets, c'est pour cela que nous avons 3 groupes et dans chacun de ces groupes nous avons 3 valeurs : l'identifiant de la coordonnée de sommet, c'est à dire que si cette valeur est à 1, nous allons chercher dans notre liste de coordonnée de sommet le premier élément ajouté, nous avons ensuite l'identifiant de la coordonnée de texture ou rien si la face n'est pas texturée et enfin l'identifiant de la normale ou rien si on n'a pas de normale pour cette face. Et comme toujours, à la fin de la définition de la face, nous avons un retour à la ligne.

Et disséminés dans le fichier d'autres mot clés comme ”mtllib” suivi du chemin vers un fichier .MTL (MaTeriaL) qui va être décrit plus tard, ”usemtl” suivi d'une chaîne de caractères qui permet d'utiliser les propriétés définies dans le groupe correspondant à notre fameuse chaîne de caractères dans le .MTL chargé, ”o” qui permet de définir un nouvel objet au sein de notre fichier .OBJ (cela sert notamment à dire que les 50 faces qui vont suivre font partie d'un même mur et on se sert de ça pour réduire drastiquement le nombre de Bounding Box dans certains cas, ce qui sera expliqué par Thomas).

La difficulté lors de la conception du système lisant les fichier .OBJ résidait dans le fait que les .OBJ contenaient quelques surprises.

L'avantage de ce format était également son problème, c'est à dire que c'est un fichier texte manipulable avec un simple éditeur de texte du type notepad, donc cela signifie qu'il n'y a pas nécessairement de retour à la ligne après la dernière coordonnée lue pour un sommet mais un espace puis un retour à la ligne, un espace peut être une tabulation, un retour à la ligne peut tenir sur 1 ou 2 octets (donc s'il y a ou non un retour chariot) et il a donc été plus simple de procéder par un système de liste blanche : tout ce qui n'est pas un caractère alphanumérique est un séparateur. On a néanmoins fait un cas à part du slash qui est un séparateur assez particulier : avoir plusieurs espaces (0x20) les uns à la suite des autres n'est pas important mais avoir un ou deux slash à un endroit ne va pas provoquer le même résultat et si vous avez compris ce qu'il y a au dessus vous savez ce qui change.

Lire le format .MTL

Bien que le format .MTL soit complémentaire au format .OBJ, le développement du lecteur de fichiers .MTL s'est terminé bien plus tard que le lecteur de fichier .OBJ pour la simple raison que pour achever ce lecteur il fallait totalement repenser la façon dont étaient stockées les données en mémoire et recoder la boucle de rendu sans compter que je devais aussi penser à rendre simple l'utilisation de ces données pour le moteur physique de Thomas, prévoir la future fonction de Charles qui devait retourner les salles vues par un joueur situé sur une salle déterminée pour appeler mes futures fonctions qui chargeront les salles qui doivent être vues et déchargeront les salles qui ne sont plus vues.

Ce n'était donc pas simple, il a fallu être très prudent, beaucoup discuter avec mes coéquipiers et cela a fini par une mise à jour qui modifiait la totalité des sources.

L'historique étant passé, passons à la description d'un fichier .MTL (pour MaTeriaL). Ce format possède un très grand nombre de mots clés pour définir des effets très spécifiques mais vu que le moteur graphique n'en utilise que six, il est inutile de décrire tout le format, nous avons donc :

- **"newmtl"** suivi d'une chaîne de caractères, par exemple "toto" correspond à ce qui a été dit dans le chapitre précédent avec le mot clé "usemtl", donc pour rappel, un usemtl toto dans un fichier .OBJ va appliquer toutes les propriétés définies sous "newmtl toto" dans le fichier .MTL aux prochaines faces définies dans le .OBJ.
- **"Kd"** suivi de 3 valeurs décimales comprises entre 0 et 1 servant à définir la couleur du sommet, donc les valeurs sont respectivement des nuances de Rouge, de Vert et de Bleu, 1 représentant l'intensité maximale et 0 l'intensité minimale.
- **"Ks"** suivi de 3 valeurs décimales comprises entre 0 et 1 servant à l'image de Kd à définir la couleur cette fois-ci dite spéculaire du sommet, ce qui veut dire qu'on va déterminer la couleur du reflet (on va développer un peu comment est déterminé un reflet plus tard, lors du bump mapping)
- **"Ns"** suivi d'une seule valeur décimale comprise entre 0 et +infini qui servira à définir l'intensité du reflet ce qui est plus expliqué sur la partie lumière et en particulier lors du bump mapping.

- ”**map_Kd**” suivi d’une chaîne de caractères indique que l’on va charger une texture qui est située au chemin défini par la chaîne de caractères et si ce champ n’est pas renseigné (il est possible qu’une texture ne soit pas nécessaire, mais il est aussi possible que le graphiste ait oublié d’insérer une texture), on charge une texture « par défaut » qui fut lors du développement du projet un pixel rouge pour repérer facilement les faces concernées mais qui a depuis été remplacé pour la version finale par un pixel noir (0;0;0) et vu que la couleur finale d’un pixel est déterminée par la somme de la couleur définie par ”Kd” et la couleur provenant du pixel de la texture devant être appliquée à cet endroit, le fait d’utiliser un pixel noir en tant que texture par défaut ne provoque aucun changement visuel et ne provoque aucun ralentissement vu qu’il ne s’agit que d’un unique pixel.
- ”**map_Ns**” suivi d’une chaîne de caractères indique que l’on va charger une texture qui est située au chemin défini par la chaîne de caractères et si ce champ EST renseigné, on active le bump mapping. Si la procédure par défaut de map_Ns diffère de la procédure par défaut de map_Kd c’est simplement pour optimiser la vitesse d’affichage, comme vous le verrez, le bump mapping est une opération très lourde et on ne peut à l’image de map_Kd utiliser une texture ”neutre”.

Les transformations

La grande difficulté de cette partie a sans doute été d’admettre qu’un ridicule tableau de nombres de dimension 4,4 pouvait provoquer n’importe quelle transformation, je parle bien entendu des matrices.

Pour les transformations, nous faisons ces calculs directement sur la carte graphique étant donné qu’elle est conçue pour faire ce type de calculs, et pour demander à la carte graphique des calculs, il faut coder et compiler un Effect File (qui se traduit par fichier effet), le notre est nommé Underground.fx.

Coder ce fichier a été un véritable challenge car il y a une myriade de mécanismes à connaître, comme ce que fait le vertex shader, ce que fait le pixel shader, connaître les prototypes des différentes fonctions disponibles, savoir que pour modifier la taille d’un tableau il faut recompiler le fichier effet etc.

Donc que ce passe-t-il dans le vertex shader et le pixel shader ?

○ Lorsque notre programme présente la scène à la carte graphique, une fonction est appelée, c'est le vertex shader, puis une seconde est appelée, c'est le pixel shader. Le vertex shader consiste à traiter les sommets des faces dans un espace tridimensionnel, c'est donc ici que l'on va d'abord placer les modèles 3D puis placer le joueur. Vu que placer un modèle 3D et placer un joueur sont des opérations quasi identiques, je vais expliquer comment on place le joueur ce qui est plus instinctif, donc cela se traduit concrètement pour chaque sommet du monde par :

- une rotation sur l'axe Y (donc si on tourne sur cet axe on regarde à gauche ou à droite)
- une rotation sur l'axe X (cette fois ci il s'agit de l'axe qui permet de regarder en haut ou en bas)
- une rotation sur l'axe Z (qui sert à faire pencher le personnage à gauche ou à droite, cet axe est très utilisé dans les jeux de simulation de guerre comme Arma pour se protéger les jambes par exemple mais n'est pas d'une très grande utilité pour notre jeu)
- et enfin on effectue une translation jusqu'à l'endroit où est censé être le joueur.

Notez que je vous ai donné ces opérations dans un ordre bien précis et c'est dans cet ordre que sont et doivent être exécutées ces opérations, sans quoi vous aurez un personnage qui aurait des déplacements très étranges (problème que nous avions à la première soutenance et qui a été depuis bien entendu corrigé).

○ Enfin dernière opération : la projection (dans notre cas avec perspective) qui permet comme son nom le suggère de projeter nos sommets qui sont dans un espace tridimensionnel sur un plan bidimensionnel (l'écran).

Au moment du retour de notre vertex shader, juste avant de passer la main au pixel shader, le vertex shader va effectuer une série d'opérations que nous n'avons pas codé visant à accélérer les prochains calculs et que je vais vous expliquer à titre purement informatif car cette information est selon moi vraiment importante pour bien comprendre comment marche notre programme, mais gardez en tête que c'est une opération totalement automatique.

- La carte graphique va d'abord effacer l'ensemble des éléments qui sont derrière la caméra.
- Elle va ensuite exécuter une opération nommée le *culling* qui permet

d'effacer les faces qui sont devant le joueur mais lui font dos, donc ce sont normalement des faces qui ne sont pas visibles, pour vous donner une idée plus précise, si le produit scalaire de la normale de la face et du vecteur d'origine (allant du joueur à l'endroit où il regarde) est négatif : on efface.

- Et enfin elle va modifier ou effacer les faces qui sont derrières d'autres faces : c'est le *Z-buffer*.

Bien, ceci étant expliqué, revenons à nos moutons, le vertex shader passe la main au pixel shader qui récupère presque ce qu'il y aura à l'écran mais le pixel shader va affiner le travail : il va traiter non pas comme son prédecesseur les sommets mais va traiter les pixels ce qui est bien plus précis (sauf si bien sur on a un objet 3D microscopique avec un nombre de polygones astronomique).

C'est donc ici que nous allons appliquer pour chaque pixel la couleur qui était, si vous vous souvenez du chapitre sur la lecture des fichiers .MTL, définie par le mot clé "kd" ainsi que le pixel associé à la texture chargée par "map_kd" et que l'on trouve (cette fois ci je vous renvoie au chapitre sur la lecture des fichiers .OBJ) au moyen des coordonnées de textures.

Vous l'aurez donc compris, le pixel shader sert à donner de la couleur à ce monde encore très uniforme, et c'est aussi à cet endroit que nous allons ajouter de la lumière.

2 La lumière

Le calcul simple de la lumière

L'élément à savoir ici est que les structures disponibles pour la fonction pixel shader sont très limitées, il était prévu par les développeurs de DirectX de laisser le programmeur avoir accès à la position du pixel sur l'écran, aux coordonnées des pixels correspondant sur les textures à appliquer et à la couleur du pixel.

Or bien qu'ayant d'abord développé un éclairage dans le vertex shader nous avons par la suite développé un éclairage dit *per-pixel lightning* ce qui signifie que l'éclairage n'est pas calculé en fonction de la face mais en fonction du pixel ce qui vous vous en doutez est bien plus précis.

Donc le problème, vu que l'on développe un éclairage dans le pixel shader, est qu'il nous manque les informations indispensable à un bon éclairage soit : la position du joueur et la normale de la face (attention, il ne s'agit pas de la normale telle qu'elle est donnée par le .OBJ, pour être précis, elle a été multipliée par l'inverse de la transposée de la matrice qui a permis de placer l'objet 3D au bon endroit). Pour avoir accès à ces données, nous avons triché, nous avons utilisé le fait que les développeurs de DirectX ont prévu l'utilisation de plusieurs textures pour faire passer ces informations dans des coordonnées de textures.

Maintenant que nous avons ces informations, nous ajoutons les lumières du jeu dans des tableaux dans la carte graphique de la taille du nombre de lumières construits si vous vous souvenez du chapitre précédent à la compilation (anecdotiquement c'est réussir à faire ces tableaux "dynamiques" qui m'a pris le plus de temps pour le développement de l'éclairage (je met des guillemet car en réalité, il s'agit plus d'un code source dynamique)), tableaux contenant bien évidemment la position des lampes, leurs couleurs, et leurs distance d'affichage donc une valeur basse sera une bougie et une valeur haute sera plutôt un incendie.

Cette fois ci nous avons tout le nécessaire pour débuter le calcul de la lumière. On va donc pour chaque lumière du jeu lumière du jeu :

Déterminer le vecteur entre la position de la lumière et la position du joueur qui pour rappel nous est accessible puisqu'on l'a mise dans une coordonnée de texture.

Calculer le produit scalaire entre le vecteur trouvé et la normale qui comme pour la position du joueur est cachée dans une coordonnée de texture. Si la normale et le vecteur trouvé sont colinéaires, le pixel va recevoir un maximum de lumière et s'il sont orthogonaux on n'a aucune lumière.

Diviser la distance d'affichage de la lumière par la distance entre la lumière et le joueur car dans de la 3D, plus on est loin d'une source lumineuse, moins on reçoit de lumière.

Cela peut paraître très fastidieux à coder mais l'accès aux fonctions mathématiques fait que ça se code avec une rapidité déconcertante. Quelques multiplications plus tard, on va rassembler les effets des différentes sources lumineuses donc :

On va sommer ces résultats entre eux et les ajouter au pixel final. On va ajouter ce résultat au pixel final aux cotés de la texture et de la couleur (qui elles ont été ajoutées via le procédé expliqué à la fin du chapitre précédent).

On a donc normalement terminé de calculer la lumière, sauf qu'on a remarqué que si un pixel était très proche d'une forte source lumineuse, le pixel était tellement saturé de lumière qu'il devenait tout blanc, ce qui ne collait pas avec l'ambiance que nous voulions instaurer dans le jeu, alors nous avons établit une couleur maximale au pixel qui est simplement sa couleur multipliée par le pixel correspondant sur la texture, ce qui fait que le calcul de l'éclairage ne peut que diminuer la couleur du monde et non l'augmenter.

Vous aurez peut-être noté que je n'ai pas encore parlé de couleur spéculaire, c'est simplement dû au fait que nous utilisons des salles avec très peu de polygones et le fait d'avoir du reflet sur des murs tout plat est vraiment très laid, c'est pour ça que nous l'utilisons lors du bump mapping.

Le bump mapping

Le bump mapping est une technique qui consiste à utiliser une texture un peu spéciale pour améliorer l'éclairage d'une face et ainsi donner l'impression d'avoir beaucoup plus de détails.

Les textures bleutés utilisées pour le bump mapping contiennent en réalité des normales et les composantes XYZW de ces normales sont bien évidemment les composantes RVBA (Rouge Vert Bleu Alpha) de chaque pixel. J'ai donc sauvagement directement utilisé ces normales sur les sommets ce qui m'a donné ceci comme rendu :

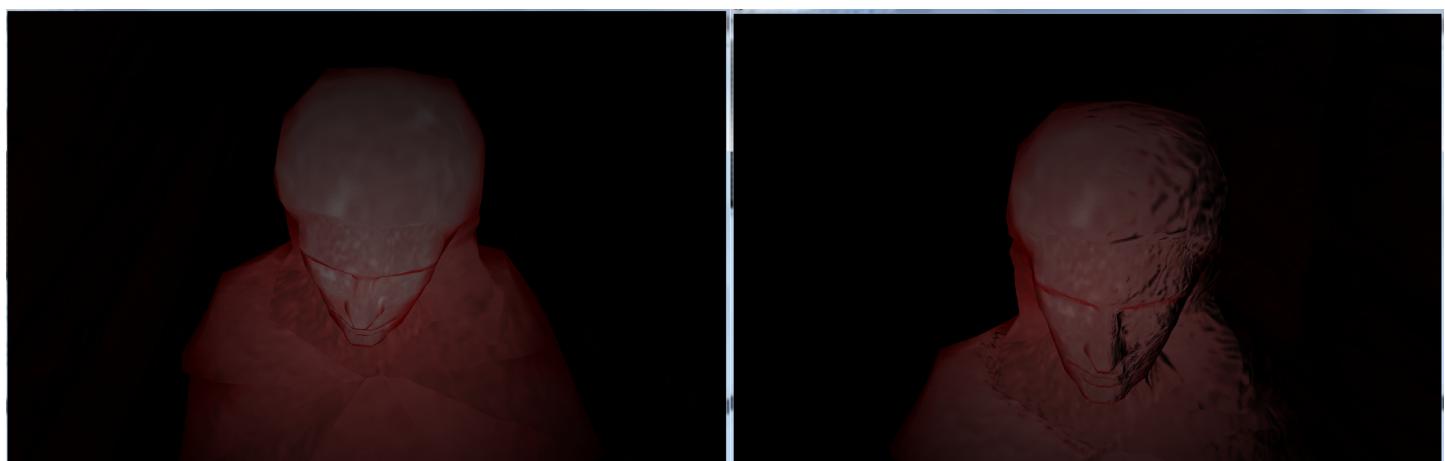


FIGURE 4.1 – Avant & après

Il y avait donc un mieux mais après avoir réfléchi un peu plus fort je me suis rendu compte qu'il devait manquer quelque chose, et effectivement, les normales étaient dans l'espace tangent, il faut donc calculer la tangente et la bitangente de la face (qui pour rappel est formé de 3 sommets qu'on nommera S1, S2, S3, de 3 coordonnées de textures qu'on nommera T1, T2, T3 et de 3 normales qu'on nommera N1, N2, N3).

On pose 4 vecteur VS1, VS2, VT1 et VT2 tels que

$$\text{VS1} = \text{S2} - \text{S1} \text{ (donc VS1 est le vecteur allant du premier au second sommet)}$$

$$\text{VS2} = \text{S3} - \text{S2} \text{ (donc VS2 est le vecteur allant du second au dernier sommet)}$$

$$\text{VT1} = \text{T2} - \text{T1} \text{ (donc la même chose que ci dessus)}$$

$$\text{VT2} = \text{T3} - \text{T1}$$

On va ensuite réduire ces vecteurs pour que leurs longueur soit égale à 1 et on a ensuite le système suivant :

$$\text{VS1} = (\text{VT1.x} * \text{tangente}) + (\text{VT1.y} * \text{bitangente})$$

$$\text{VS2} = (\text{VT2.x} * \text{tangente}) + (\text{VT2.y} * \text{bitangente})$$

Donc en résolvant le système par la méthode des déterminants de matrice on a :

$$\text{tangente} = \pm (\text{VT2.y} * \text{VS1} - \text{VT1.y} * \text{VS2}) / (\text{VT1.x} * \text{VT2.y} - \text{VT1.y} * \text{VT2.x})$$

$$\text{bitangente} = (-\text{VT2.x} * \text{VS1} + \text{VT1.x} * \text{VS2}) / (\text{VT1.x} * \text{VT2.y} - \text{VT1.y} * \text{VT2.x})$$

Vous noterez le signe « plus ou moins » devant l'expression de la tangente, car le produit scalaire entre la tangente et la bitangente ne doit pas être négatif.

La couleur finale (Cf) d'un pixel avec le bump mapping est donc la somme du calcul suivant pour chaque lumière :

$$\begin{aligned} \text{Cf} = & (\text{Couleur de la lumière} * \text{N produit scalaire L} \\ & + \text{couleur de la lumière spéculaire} * \text{amplification spéculaire}) \\ & * \text{atténuation} \end{aligned}$$

Avec :

N = la normale présente dans la texture bleutée (donc la couleur RGBA * 2 – 1)

N = la multiplication du vecteur allant du joueur au pixel calculé divisé par la portée de la lumière par la matrice TBN

TBN = la matrice :

[tangente.x, bitangente.x, normale.x]

[tangente.y, bitangente.y, normale.y]

[tangente.z, bitangente.z, normale.z]

normale = la normale extraite du .OBJ et bien sur multipliée par l'inverse de la transposée de la matrice qui place l'objet 3D dans le monde

amplification spéculaire = le produit scalaire entre N et (L + le vecteur allant de la position du joueur au pixel calculé multiplié par la matrice TBN) le tout à la puissance déterminée par Ns dans le .MTL

attenuation = 1 – vecteur allant du joueur au pixel calculé divisé par la portée de la lumière



FIGURE 4.2 – Après l'après !

Le placement des objets en temps réel

D'abord, il faut savoir que les objets 3D utilisés pour construire les salles sont au nombre de 3, on a un objet 3D qui contient le sol (un carré) + le plafond (un autre carré), un autre qui contient un mur + une gouttière, et le dernier qui contient une colonne. En disposant d'une certaine manière ces objets 3D, on peut créer les 5 salles élémentaires nécessaires à la construction du labyrinthe (par exemple pour créer une salle en X on a besoin de 5 objets 3D contenant le sol et le plafond).

On récupère les salles à afficher en fonction de la position du joueur donc en divisant la position du joueur dans l'espace par la taille des salles (qui pour vous donner une idée était de 16 unités pour les salles de test et de 1200 unités pour les salles finales), ce qui me permet de récupérer dans le tableau créé par le code de Charles et contenant le labyrinthe, quels sont les types de salles à afficher (en L en X en T, en I ouvert ou en I fermé) et quelle est la rotation à appliquer. On place donc tous ces éléments à coup de rotations et de translation en « annulant » la rotation appliquée sur les sols et plafonds pour garder la continuité des textures entre les salles.

Mais pour placer ces salles, il faut les récupérer, on traite les fichiers .OBJ correspondants que l'on met dans une liste d'objets 3D avec quelques paramètres comme la couleur diffuse, la couleur spéculaire, le chemin de la texture, le chemin de la texture bleutée (la normal map pour le bump mapping), etc, la matrice qui permet de placer l'objet à sa place, un identifiant (qui par soucis de simplicité, est un tableau de 2 entiers correspondants aux indices de la salle dans le tableau bidimensionnel de Charles néanmoins on a décidé de donner des identifiants négatifs pour les objets 3D qui ne dépendent pas du labyrinthe comme le monstre par exemple), et un booléen qui indique si on affiche cet objet 3D ou pas. On a aussi d'autres paramètres comme par exemple, un Effet qui est un clone d'un Effet de base qui est en fait le fichier effet Underground.fx compilé mais contenant entre autres l'inverse de la transposée de la matrice permettant de placer l'objet 3D dans le monde ce qui permet d'éviter des calculs superflus à chaque rendu.

Donc une fois que les salles ne sont plus vues par le joueur (ceci est expliqué dans la partie de Charles) on désactive dans les liste des objets 3D les éléments contenant un des identifiants des salles qui ne plus vues ou les vide complètement de la carte graphique si on dépasse un certain nombre de doublons pour ne pas surcharger la mémoire de la carte graphique. Et lorsque qu'on charge une nouvelle salle, on va chercher dans les objets 3D désactivés si on ne peut pas réutiliser ce qui est déjà en mémoire, si on trouve on change juste la matrice qui permet de transformer l'objet et quelques petites choses dans l'effet correspondant, sinon, on charge complètement l'objet 3D cette fois ci non pas du disque dur mais de la mémoire vive car au premier chargement d'un objet 3D, on stocke en mémoire l' objet 3D épuré pour accélérer les prochains chargements.

Tout à l'heure, j'ai dit que l'on stockait le chemin des textures, c'est une optimisation que nous avons effectué vu qu'une texture est très longue à charger. On a une liste de texture contenant la texture (la vraie, RGBA) et le chemin de la texture, on a donc directement accès à une texture qui a déjà été utilisée auparavant et on évite même les doublons de texture.

Il faut aussi savoir qu'au début du jeu, on construit, charge et désactive l'affichage de 4 salles en L, 3 salles en I fermé (cul de sac), un nombre arbitraire de salles en X (à l'heure où j'écris ces lignes 2) en I ouvert (à l'heure où j'écris ces lignes 6) et en T (à l'heure où j'écris ces lignes 6), ce qui évite d'avoir de gros chargement en cours de partie au début du jeu (vu que l'on charge les salles en fonction de la position du personnage).

A chaque clignement d'yeux, on place le monstre devant le joueur à une distance de plus en plus proche, s'il n'est pas devant le joueur on le place à sa gauche, ou à sa droite ou derrière, la difficulté a été de la faire apparaître dans les salles, tâche qui a été terriblement simplifiée grâce aux paramètres des « cases » créées par Charles : h b g d qui permettent de dire si on a un mur respectivement en haut en bas à gauche ou à droite des cases.

3 Les évènements

Notre gestion des événements est dans un fil d'exécution différent du principal (qui contient la boucle de rendu) et c'est l'endroit où se font les calculs un peu coûteux comme par exemple le calcul d'une distance.

Notre premier et unique monstre a une certaine influence sur le joueur, en fonction de la distance les séparant, on ajoute des parasites aux rotations X et Y lors du placement du joueur (évoquées dans le chapitre sur les transformations) les parasites étant des nombres aléatoires multipliés par un coefficient, ce coefficient étant dépendant de la distance entre le joueur et le monstre, donc concrètement plus on est proche du monstre, plus la camera tremble.

Le calcul de la distance entre le joueur et le monstre permet aussi de créer une autre influence : plus on est proche du monstre, plus la portée de la lumière du monstre (rouge) est grande et moins la portée de la torche

du joueur est grande moins on est proche du monstre, plus la portée de la torche du joueur est grande et moins la portée de la lumière du monstre est grande.

Et encore une dernière : le négatif, plus le joueur est proche du monstre, plus le négatif appliqué sur la texture est important. Bien évidemment, la « négation » de la texture est appliquée dans le Pixel Shader soit sur la carte graphique, avec le processeur on va juste calculer un coefficient entre 0 et 1 qui va déterminer à quel point le négatif est appliqué.

Autre calcul qui se fait ici, le calcul de la sinusoïde qui parasite la translation sur l'axe Y lors du placement du joueur (le joueur décrit un mouvement haut bas lorsqu'il marche).

C'est aussi ici que sont calculés (pas affichés) le clignement d'yeux qui consiste à réduire progressivement la portée de toutes les lumières puis les augmenter et les barres de progressions (barre de progression avant le clignement et celle de l'endurance).

4 Interface utilisateur

L'interface utilisateur est en fait un ensemble d'outils que j'ai développé à la base pour faciliter l'intégration d'éléments en 2 dimensions dans des espaces en 3 dimensions. C'est donc une fonction qui est dans le fil principal et dans la boucle de rendu, une fois que l'on rentre dedans, on informe la carte graphique que l'on va envoyer des sommets légèrement différents des sommets envoyés pour créer une face en profondeur, on a donc pas besoin d'avoir de normale ou de tangente ou de couleur, il suffit d'avoir la position du sommet et de la coordonnée de texture.

On a donc une liste d'éléments qui contiennent la position de l'élément, la taille de l'élément, l'adresse de la texture et le type de l'élément.

Donc on a par l'exemple la lampe torche qui est en fait un élément quelconque placé aux coordonnées 0 ; 0, de la taille de la fenêtre, avec une texture qui masque les bords de l'écran en étant noire sur les bords et transparente au milieu.

C'est aussi ici que sont affichés les barres de progression qui bougent en multipliant la taille initiale de l'élément par les coefficients correspondants calculé par le fil des événements décrit dans le chapitre précédent.

Pour chaque élément, on crée un rectangle grâce à 6 points (il s'agit donc de 2 triangles côtés à côtés) à l'aide de la position et de la taille sur laquelle on applique la texture.

CHAPITRE 5

Thomas aka *Feeloun*

1 Le menu

Dès le début, on savait qu'il en faudrait un.

On était jeunes et innocents, on pensait que DirectX aurait un moyen de créer une interface simplement, un peu comme avec les Windows Forms (enfin peut-être pas *aussi simple que ça* mais bon).

Et puis... On a eu mal. Enfin surtout moi, vu que c'était la partie dont je m'occupais. Pour résumer, il fallait tout créer, de l'implémentation des boutons aux différentes options.



FIGURE 5.1 – Le menu de la soutenance 1. "Hodor".

Il fallait donc que je choisisse entre deux options :

- implémenter un gestionnaire d'interface déjà existant
- ... ou en faire un à partir de rien

Même si la première option est la plus tentante, elle pose plusieurs problèmes : déjà, il faut arriver à tout rendre compatible avec SharpDX, mais vu les problèmes que nous avions, cela aurait été un pari assez risqué.

Je me suis donc mis à faire un "gestionnaire de menu", et à ce moment j'ai compris la puissance des arbres. À l'aide d'un arbre général, dont ses

fils peuvent faire appel à des fonctions, et où chaque fils connaît son père (pour permettre aux boutons "retour" de fonctionner), ce fut moins pénible que prévu.

De plus, sachant quels boutons allaient être appelés grâce à cet arbre, j'ai pu mettre en place des animations. C'est pas forcément très utile, mais c'est toujours agréable !

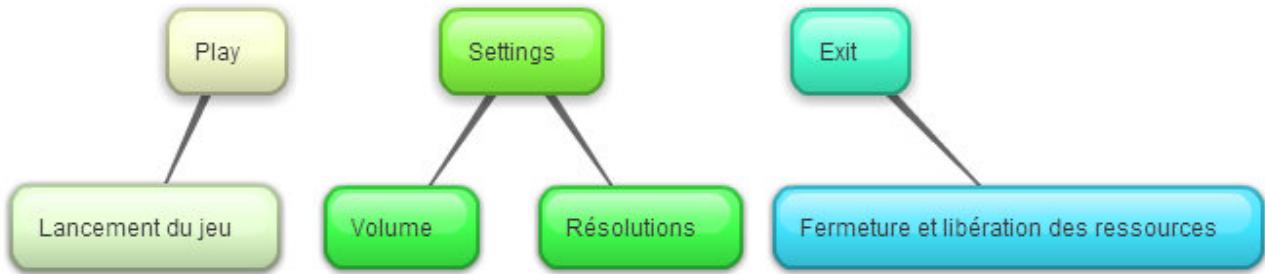


FIGURE 5.2 – Schéma simplifié des menus.

Cependant, un dernier problème s'est posé : la gestion des résolutions. En effet, utilisant une implémentation de DirectX 9, on ne peut pas changer la résolution "à la volée".

Il faut en effet enregistrer tous les éléments que l'on a mis dans la carte graphique, les libérer, puis changer la résolution et relancer le jeu. C'est pas si dur une fois qu'on sait, mais ça explique pourquoi de nombreux jeux (tous ceux développés sous DirectX 9 en fait), doivent être relancés pour permettre un changement de résolution.



FIGURE 5.3 – Le menu fini ! Hodor² !

2 Les collisions

Les démoniaques, les affreuses, les terribles... Les collisions.

Tout le monde vous le dira : il faut utiliser des *Bounding Boxes*. (ou boîtes englobantes, en Français, non mais sans rigoler...)

D'après Wikipedia :

Une Bounding Box est, pour plusieurs points dans N dimensions, la boîte de la plus petite taille (aire ou volume suivant la dimension) dans laquelle tous les points souhaités peuvent tenir.



FIGURE 5.4 – L'agression des bounding boxes, un fléau récent

En soi, le principe est assez simple : créer une liste de boîtes qui contiennent les différents sommets d'un objet, pour pouvoir vérifier à chaque fois qu'on avance les collisions.

Mais... malheureusement, notre importateur de .obj est assez compliqué et ne différencie pas les différents objets présents, et nous renvoie simplement la liste de tous les sommets de la carte.

J'étais donc devant mon écran, avec 68 000 sommets, et des bounding boxes à créer. J'ai choisi la solution la plus simple, c'est-à-dire prendre les sommets 2 par 2, et créer une bounding box à chaque fois. Ce qui donne en effet... 34 000 boîtes.

J'ai alors commencé tel un bleu : à chaque mouvement de la caméra, ou plutôt, chaque pression minime d'une touche, ces 34 000 bounding boxes

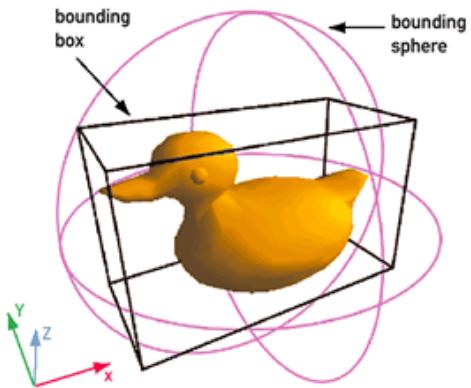


FIGURE 5.5 – Dans un monde parfait...

étaient testées pour leur collision avec la caméra (enfin, la bounding sphere de la caméra).

En effet, c'est imaginable, le résultat était simplement affreux. Vu la vitesse de déplacement, ça s'apparentait plus à un documentaire Arté qu'à un jeu d'horreur. Puis je me suis dit.. pourquoi tester des collisions si souvent, alors qu'on met à jour la fenêtre bien moins souvent ?

J'ai alors changé mon implémentation des collisions pour ne les tester que lors de la boucle de rendu : en effet, au lieu de les tester pour chaque mouvement minime, elles sont désormais vérifiées que lorsqu'il y a un mouvement conséquent, ce qui a fait disparaître toute trace de lag.

... Mais à ce moment là, le jeu n'en était encore qu'à ses débuts. Désormais, grâce au travail des autres membres, la carte est générée aléatoirement et fait un labyrinthe.

Elle peut donc contenir encore plus de sommets, ce qui rend le tout plus complexe, et la recherche linéaire ne semble plus être une solution. Mais que reste-t-il ? Les arbres et les graphes d'intervalle.

Les arbres m'étant plus familier, et bénéficiant de l'aide de Tommi Laukkonen, un développeur finlandais qui a accepté de m'aider à comprendre et à réaliser cet algorithme (et que je remercie encore d'ailleurs), nous avons désormais un arbre tridimensionnel d'intervalles fonctionnel !

Comment cela marche ?

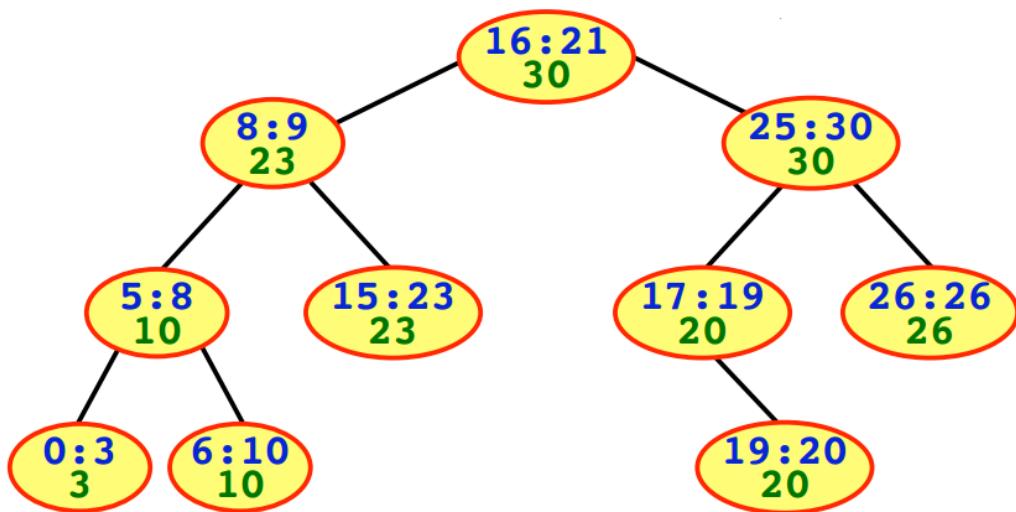


FIGURE 5.6 – Arbre d'intervalles simple

Dans chaque noeud, les deux nombres en bleu représentent l'intervalle, et celui en vert (appelé dmax) est le maximum des extrémités droites des intervalles du sous-arbre issu de ce noeud.

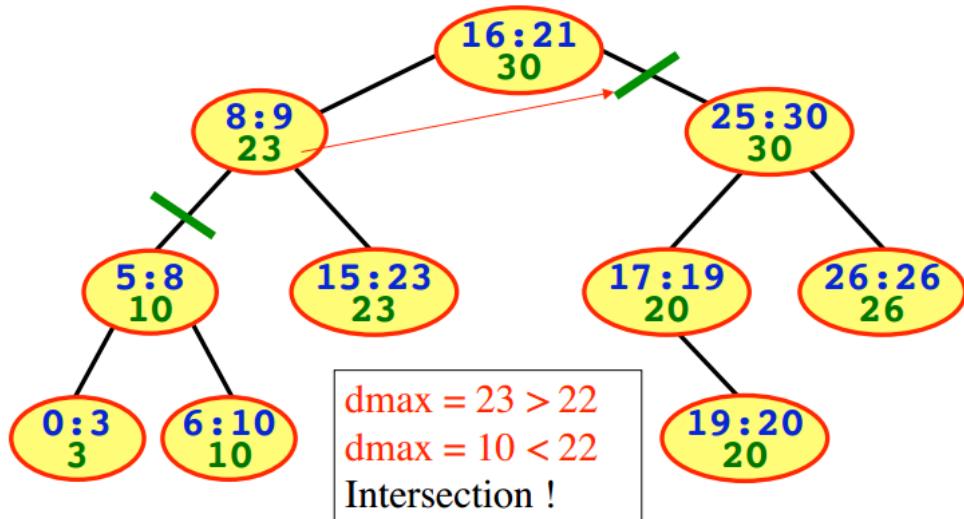
On applique deux règles :

1. Si filsG.dmax < g, inutile de chercher dans le sous-arbre gauche.
2. Si filsG.dmax $\geq g$, inutile de chercher dans le sous-arbre droit.

Prenons deux cas de recherche :

– Recherche positive : l'intervalle [22, 25]

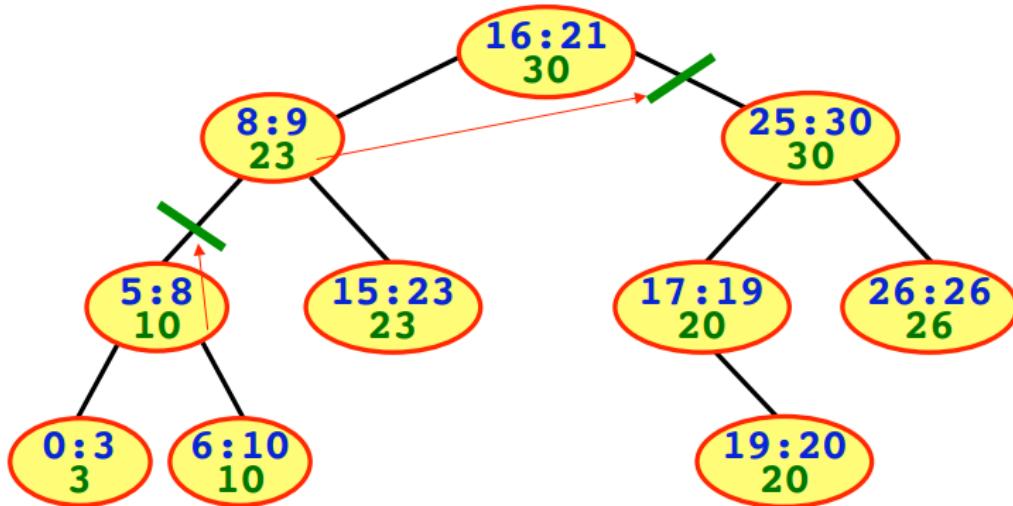
Grâce à la règle 1, on descend dans le sous-arbre gauche. Or, en re-



descendant, $10 < 22$. Puis avec la règle 2, comme $23 > 22$, on se rend compte que l'intervalle est compris dans l'arbre.

- Recherche négative : l'intervalle $[11, 14]$

En descendant ici, nous trouvons que le $d_{max} = 23 > 11$, puis que le



d_{max} suivant = $10 < 11$. Il n'y a donc pas d'intersection.

Maintenant, comment gérer ça sur plusieurs dimensions ?

Le principe est plutôt simple : à chaque fois qu'on descend, on teste pour une autre dimension. Donc pour 3 dimensions, au premier niveau on testera les x, au deuxième les y, et au troisième les z.

3 Le site web

Réalisé avec les moyens habituels - HTML, CSS, PHP et MySQL.

La page d'accueil est mise à jour presque chaque mois pour montrer l'avancée et quelques images de développement.

La partie média contient l'ensemble des images utilisées pour nos soutenances, et permet de rapidement voir l'avancement du projet.

La partie téléchargement est assez explicite, et la "à propos" permet de découvrir les différents membres du groupe.

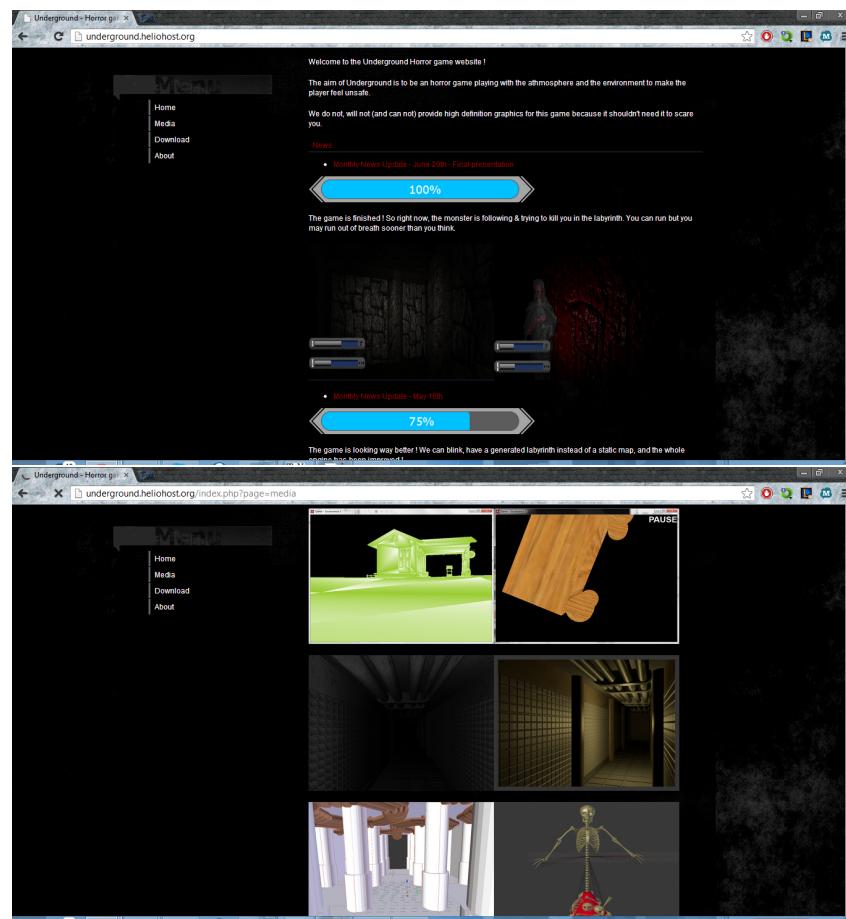


FIGURE 5.7 – Tadaaaaaaa !

CHAPITRE 6

Bastien aka Stin

Au cours de cette dernière année, Underground Project Section a décidé de réaliser un jeux vidéo en 3D, mon rôle était au sein de ce projet de s'occuper de donner vie à notre projet, je m'occupais donc de créer et d'animer tout ce qui se passait à l'écran : j'étais graphiste.

1 Le choix du logiciel de modélisation 3D

Afin de créer nos objets composant l'environnement 3D de notre jeu, nous avons dû choisir un logiciel de modélisation 3D, nous avons longtemps hésité entre deux d'entre eux, à savoir Blender et 3DsMax (nous avons également testé Maya), tous ces logiciels présentant leurs avantages et leurs inconvénients, Blender avec comme principal avantage d'être un logiciel que je connaissais déjà, mais dont les objets produits était difficilement implémentables sous SharpDX, et 3DsMax logiciels jusqu'alors inconnu, prodiguant une facilité d'exportation, mais dont je devais apprendre à me servir.

Le problème était un peu plus compliqué que de choisir entre deux jolies interfaces, en effet pour réaliser notre projet nous avons choisi d'utiliser SharpDX, une implémentation de DirectX pour le C#, mais qui manque cruellement de documentation, il fallait donc se concentrer plus sur la maîtrise de SharpDX, que sur la production d'un environnement. On a finalement décidé d'utiliser Blender, et il nous aura fallu un certain temps pour mettre au point un procédé nous permettant de simplement afficher les objets produit sous Blender.

Le principal problème de Blender a été la triangulation des faces, sous Blender les objets produits possèdent des faces à 4 sommets, or SharpDX

ne gère uniquement des faces à trois sommets, il fallait donc découper le code de l'objet de façon à ce que chacune des faces soient des triangles. Finalement on a utilisé un module de Blender qui permettait de le faire lors de l'exportation et qui le faisait plus ou moins bien.

2 La création de l'environnement

La détermination de l'environnement

La création d'un environnement est en soi pas si compliqué, cependant suite à de nombreux problèmes, j'ai dû recommencer un très grand nombre de fois.

Les premiers niveaux réalisés étaient basés sur les couloirs de métro, ci-dessous vous pouvez voir l'environnement de la soutenance 2.



L'environnement a subi de nombreuses modifications entre la soutenance deux et trois, ainsi en partant d'un univers moderne j'ai finalement produit divers environnements de type donjon et produit plusieurs niveaux.

Finalement je suis revenu aux sources et ai produit pour la soutenance finale un environnement inspiré de la soutenance 2, visibles si dessous.

Les artefacts

Lors de la conception des environnements nous nous sommes aperçus que tous les environnements que je produisais n'étaient pas forcément affichables. Certains d'entre eux possédaient des lignes de code incorrectes dans les fichiers obj, ces lignes définissaient des faces à deux sommets non affichables par SharpDX que vous pouvez voir ci-dessous.

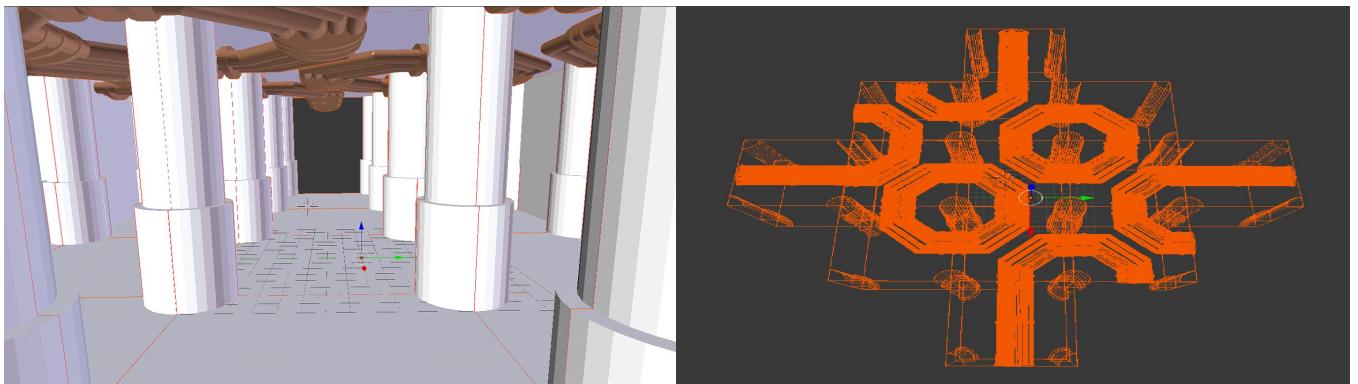


FIGURE 6.1 – Pièce en "+" non texturée

5374 f 393/213/393 1067/966/1052 1065/1068/1058
 5375 f 1066//1059 1068//1053 392/400/400
 5376 f 999/257/966 325/393/307 1001/1318/962
 5377 f 328/389/304 326/1296/308 1000/307/967
 5378 f 1063/966/1060 389//401 1065//1058
 5379 f 392/400/400 390/326/402 1064/1063/1061
 5380 f 13 39
 5381 f 14 40
 5382 f 239 261
 5383 f 615 617
 5384 f 616 618
 5385 f 617 619
 5386 f 618 620
 5387 f 33 57
 5388 f 35 53
 5389 f 13 53
 5390 f 14 54
 5391 f 32 64
 5392 f 27 63
 5393 f 27 67
 5394 f 23 69

Ceci est un artefact

La une liste d'artefacts

La première idée, pour résoudre ce problème, fut d'effacer ce surplus de lignes qui gênait au bon déroulement du programme. Ce ne fut pas une très bonne idée, en effet après suppression de ces lignes, et sauvegarde du fichier on s'est aperçus que les fichiers continuaient à faire planter l'obj loader (fonction qui gère les obj), nous avons alors rouvert le fichier et constaté que l'intégralité des faces contenues dans le fichier s'étaient décalées de manière à avoir le même nombre de faces qu'avant la suppression des artefacts et a reformé ces derniers. On arriva donc à la conclusion que les artefacts étaient un défaut de conception de l'objet, et que tous fichiers contenant ces derniers devaient être détruits car inutilisables.

C'est la principale raison au grand nombre d'environnements créés entre la soutenance 2 et 3. Cependant les artefacts n'étant visibles que dans les fichiers obj après exportation, on ne pouvait les détecter qu'après que l'objet soit terminé.

Cependant lors de la modélisation des monstres j'ai été forcé de résoudre ce problème car je ne pouvais pas me permettre de recommencer la modélisation des monstres qui prenait beaucoup trop de temps.

Je me suis donc servi du premier monstre modélisé pour résoudre ce problème. Contrairement aux objets jusqu'alors créés ce dernier possédait différentes copies des divers petits objets qui le compossaient, et qui plus est, ces copies se situaient à différents stades de sa réalisation, chose que je ne n'aie faite que pour cet objet particulier à cause de la difficulté de réalisation, et la peur de devoir revenir plusieurs stade en arrière suite à l'apparition d'une erreur qui m'aurait échappée.

De plus je pus ainsi remarquer un fait étrange qui était que les artefacts n'étaient pas présents dans tous les sous objets qui compossaient le modèle 3D. La première chose que j'ai faite fut donc d'exporter toutes les étapes que j'avais à ma disposition, et de regarder à partir de quel moment apparaissaient ces artefacts, et il s'est trouvé que ces derniers n'étaient pas dûs à une erreur de conception car ils apparaissent à l'étape qui précédait l'exportation durant laquelle j'utilisais des fonctions intégrées a Blender pour améliorer mon objet de façon à éviter que ce dernier soit trop carré, mais je ne faisais plus de modifications à proprement parler.

Pour se faire j'utilisais des fonctions qui subdivisaient les faces et multipliaient les sommets pour arrondir les angles, et d'autres pour aplatisir les surfaces, je me suis alors dit que la subdivision des faces devait faire planter la triangulation lors de l'exportation, mais à mon grand étonnement ce n'était pas le cas, les fichiers produits lors des exportations de diverses combinaisons de ces fonctions, ne possédaient aucun artefact.

Il ne me restait plus qu'une fonction à tester, la fonction « Mirror », cette fonction très particulière était le point commun à tous les objets contenant des artefacts, pourtant je l'avais écartée au tout début à cause de son mode fonctionnement, en effet cette fonction ne fait que reproduire, comme un miroir, une partie ou la totalité d'un objet selon un axe de symétrie (ce qui est très utile pour un graphiste).

En effet, pour pouvoir détruire les artefacts, il fallait d'abord les localiser efficacement, car il était rare d'avoir plus de dix artefacts dans un objet achevé, même si le plus grand nombres d'artefacts que j'ai pu recenser dans un objet était de 31 pour un obj comportant 8145 sommets, et malgré les récentes découvertes qui permettaient de limiter la zones de recherches à quelques centaines de points, voir seulement a une cinquantaine pour le meilleures des cas, cela revenait quand même à chercher une aiguille dans une botte de foin. J'ai donc continué à faire quelques expérimentations de

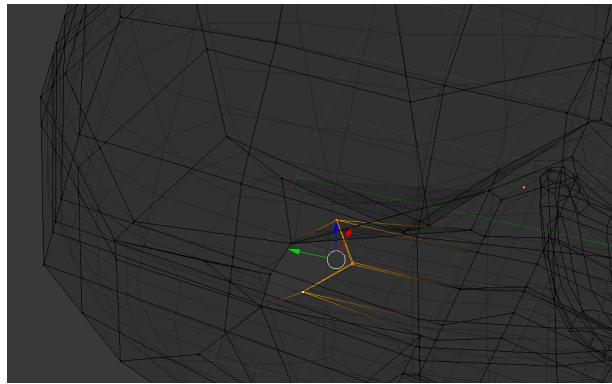


FIGURE 6.2 – En jaune, deux artefacts d'un objet

façon à trouver une façon simple de localiser et d'effacer tous les artefacts d'un coup (moyen que j'ai fini par trouver d'ailleurs), et surtout de les éviter lors de la création d'un environnement.

La constitution des environnements

Comme évoqué précédemment, l'environnement dans lequel évolue le joueur n'a cessé de changer et même la version qui se trouve ci-dessous et qui est la dernière version en date, n'est probablement pas celui qui apparaîtra lors de la soutenance finale, car ce rapport a été rédigé plusieurs jours avant la soutenance cependant tous ces environnements, à l'exception des premiers qui étaient des niveaux complets, avaient un certain nombre de points communs.

Ainsi, tous les niveaux se composent uniquement de cinq salles. Ces salles sont un tournant, un couloir droit, un croisement à 4 branches, un autre à 3 branches et enfin un cul de sac. Après ça, à l'aide d'un algorithme mis au moins par Charles, on affiche les différentes salles et on leur applique des rotations de façon à constituer un niveau sans trous possédant une fin et un début.

Cette manière de procéder, bien qu'efficace pour créer des environnements variés à partir d'un groupe d'objets réduit, et permettant d'allonger considérablement la durée de vie du jeu présente cependant un inconvénient : les caractéristiques techniques des salles. En effet, ces dernières doivent avoir les mêmes proportions, mais aussi pouvoir s'assembler sous de très nombreuses possibilités différentes.

L'un des premiers problèmes auxquels j'ai dû faire face a été le problème

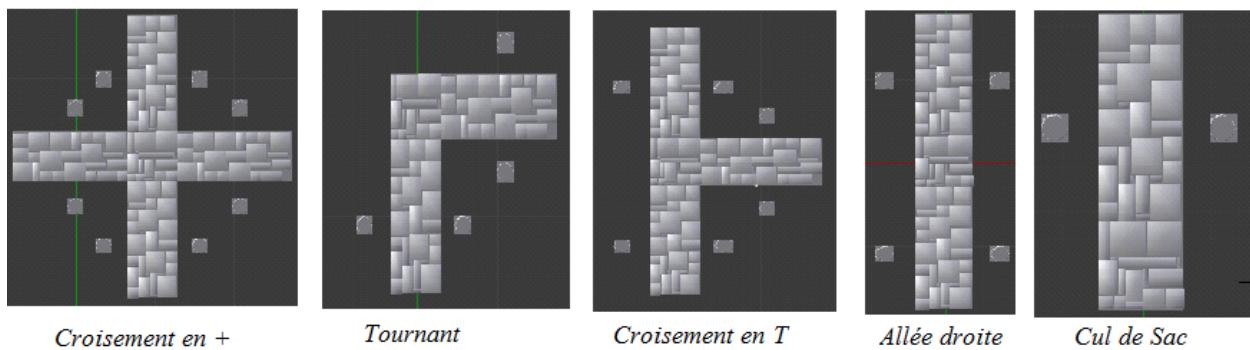


FIGURE 6.3 – Les cinq salles de base

lié au raccord des salles entre elles, ce problème s'est montré très tôt, même avant de décider de créer des niveaux aléatoires.

En effet, lorsque je créais mes niveaux entiers au début du projet, je me servais déjà de petites salles que j'assemblais entre elles pour obtenir la disposition souhaitée, je me suis aperçu dès lors que lorsque l'on juxtapose deux salles on voit nettement où commence l'une et où finit l'autre, notamment au niveau des textures.

Pour résoudre ce problème deux choix s'offraient à moi. Le premier consistait à faire coïncider les textures des murs pour avoir une impression de continuité, cependant cette option me paraissait difficilement envisageable, car extrêmement difficile à mettre en place pour tous les raccords possibles (même si j'en avais moins à gérer au début du projet, qu'aujourd'hui), c'est pourquoi j'ai opté pour la seconde possibilité.

Cela consiste à camoufler le raccord en mettant quelque chose devant, dans notre cas il s'agit de piliers, en effet à chaque fois que vous passez devant un pilier dans notre jeu vous changez de salles. Les piliers ne servent pas uniquement à camoufler les raccords, ils servent également au gameplay, il donne l'impression que l'environnement est bien plus étroit qu'il ne l'est en réalité, cela ajouter au phénomène de répétition, puisqu'il se trouve à une distance régulière, et est là pour angoisser le joueur, pour faire en sorte qu'il se sente oppressé, en plus cela réduit le champ de vision qui est déjà peu grand, et empêche de voir les tournants ce qui fait qu'un monstre peut très bien s'y cacher.

Pour optimiser la construction des environnements, j'ai découpé les salles de façons à les réduire à des cubes de 12 unités Blender de côté

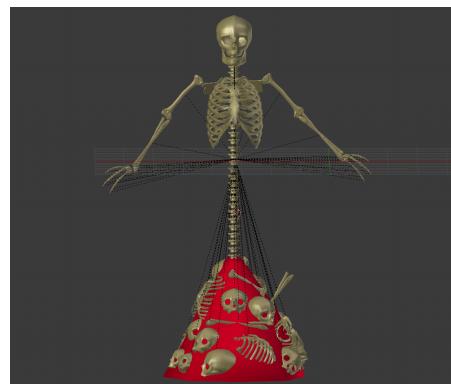
constituées de 4 quarts de pilier, un sol, un plafond et un nombres de murs qui varient en fonction de la place du cube dans la salle.

3 La création des monstres et les animations

La création des monstres

Le titre exagère un peu, en fait malgré le fait que j'ai mené des études pour trois types différents de monstres, je n'en ai réalisé qu'un seul, le squelette visible ci-dessous, en fait c'est à cause de ce squelette que j'ai fait une « chasse aux artefacts », puisque le crane et la cage thoracique contenaient tous deux des artefacts, et c'est sur les copies du crane que j'ai pu découvrir la nature réelle des artefacts.

Je vous ai dit que je ne voulais pas recommencer ces objets depuis le début juste pour effacer les artefacts, et bien c'est qu'il m'a fallu trois jours pour réaliser ce squelette, 2 pour réaliser le crane et la cage thoracique, une demi-journée pour faire le reste dont les textures, 1 jour pour résoudre le problème des artefacts et encore une demi-journée pour mettre en place les animations du squelette. Bon, je sais que ça fait quatre (la base du squelette était fixé le troisième jour) mais l'essentiel c'est de noter que la cage thoracique et le crane m'ont pris le plus de temps et qu'il est hors question de recommencer depuis zéro quelque chose d'aussi complexe qui pourrait à nouveau contenir des artefacts.



Pour me justifier sur le pourquoi il m'a fallu autant de temps pour réaliser ce squelette, c'est je suis parti de croquis anatomiques en 2D pour réaliser chacun des objets 3D qui le constituent, donc chaque objet qui compose ce squelette est basé sur de véritables os humain, je dis basé par ce que j'ai effectué quelques modifications (après tout c'est censé être un monstre et pas simple squelette), comme augmenter la longueur des avants

bras et le nombre de côtes, de plus j'ai limité le nombre de doigt à trois légèrement allongés et grossis, la colonne vertébrale est démesurément allongée pour donner un aspect de serpent à l'ensemble, enfin je l'ai fait sortir d'un tas d'ossement et de chair, pour garder l'idée d'un monstre "statue".

Enfin j'ai réalisé une animation du personnage en animant chacun des os qui constituent le corps de ce squelette à l'exception de ceux de la cage thoracique, chacune des 25 lombaires peut donc changer de position, ainsi j'ai pu créer des poses pour mon personnage de façon à faire en sorte que mon squelette ait l'air moins statique.



La raison pour laquelle je n'ai pas fabriqué d'autres personnages, c'est que mon projet de monstres a été refusé car s'intégrant mal à l'univers du jeu, personnellement je ne vois vraiment pas pourquoi.



Je suis pourtant sûr que d'un point de vue marketing, un jeu contenant une succubus est bien plus vendeur qu'un jeu contenant des statues d'anges qui pleurent, mais bon il paraît que ce n'est pas assez effrayant, et en plus d'après mes dessins préliminaires (qui ne sont pas dans ce rapport pour éviter que ce dernier ne deviennent un rapport illicite), les poitrines étaient trop grosses à leurs gouts et auraient empêché le joueur de se concentrer, ce qui aurait été considéré comme de l'anti-jeu.

L'animation

Bien que le squelette ne soit pas à proprement parler animé, le jeu possède cependant de nombreuses animations, telles que le clignement d'œil, l'animation de mort, ou encore des secousses et autres. Je n'ai pas fait toutes les animations du jeu, mais j'en ai suffisamment fait pour pouvoir en parler.

Pour faire nos animations on se sert d'un timer et de mouvements de caméra. Ainsi on effectue à chaque rafraîchissement d'écran on effectue un léger mouvement de camera, constitué de translations et de rotations par rapport à la position initiale, le tout synchronisé sur notre timer, donnant ainsi l'impression que le personnage principale est sujet à un mouvement fluide et continu.

CHAPITRE 7

Conclusion

Cette troisième soutenance nous aura permis de vraiment rentrer dans les arcanes de Direct X, ce qui nous a permis de transformer quelque chose qui ressemblait à un programme de visualisation de modèle 3D à en un vrai jeu.

La dernière soutenance sera consacrée à approfondir le gameplay au maximum avec comme objectif de pousser le réalisme à son paroxysme créant une expérience immersive totale.

tl;dr Vive DirectX¹ et vive la mémoire libre !

1. mais pas trop non plus