

M2 SIS - Synthèse d'images et réalité virtuelle : Dungeon Master

TOFFANIN Marc

25 janvier 2016

1 COMPILEMENT ET EXECUTION

Pour compiler le programme, il suffit d'exécuter le fichier *build.linux*, ou d'exécuter directement le *Makefile* (testé sur Ubuntu 11.3-15.4)

Certains packages sont requis d'être installés sur la machine hôte, sur ubuntu en faisant :

sudo apt-get install libsdl2-dev libsdl2-image-dev libsdl2-ttf-dev libassimp-dev libglm-dev
Une fois le programme exécuté, la partie du jeu démarre. Une seule partie par exécution peut être jouée.

2 BIBLIOTHÈQUE UTILISÉES

L'application utilise certains nombre de bibliothèques, surtout de la SDL (crossplatform et open-source).

- SDL2.0 permet de créer le contexte et la fenêtre OpenGL
- SDL_image est une bibliothèque de chargement d'images utilisées pour les textures
- SDL_ttf sert à gérer le TrueTypeFont qui permet l'affichage de texte à l'écran
- Assimp est une bibliothèque open source de chargement de modèle 3D
- GLM est une bibliothèque header only servant à créer des vecteurs et matrices compatibles OpenGL

Certaines parties du code sont en partie prise de *learnopengl.com* (notamment le chargement de models) et de mon projet de l'année dernière.

3 PRÉSENTATION DU PROJET

Le projet de base consiste au développement d'un jeu type dungeon master, cependant je l'ai étendu à créer un type de jeu dungeon crawler. Ce type de jeu est en vue FPS et l'on se déplace librement dans un donjon en rencontrant créatures et trésors. Le but du jeu est de trouver la clé cachée dans un coffre pour ouvrir la porte de sortie.

3.1 PARTIE GRAPHIQUE

Au niveau graphique, on peut retrouver la mise en place de normal mapping sur les murs, de réflexion sur l'eau et d'un modèle de lumière blinn-phong.

Le normal mapping s'effectue en lisant les normales d'un objet depuis une texture et utiliser cette nouvelle normale pour le shading. Ceci permet d'émuler le fait que l'objet est constitué de normales qui changent en fonction de sa topologie virtuelle.

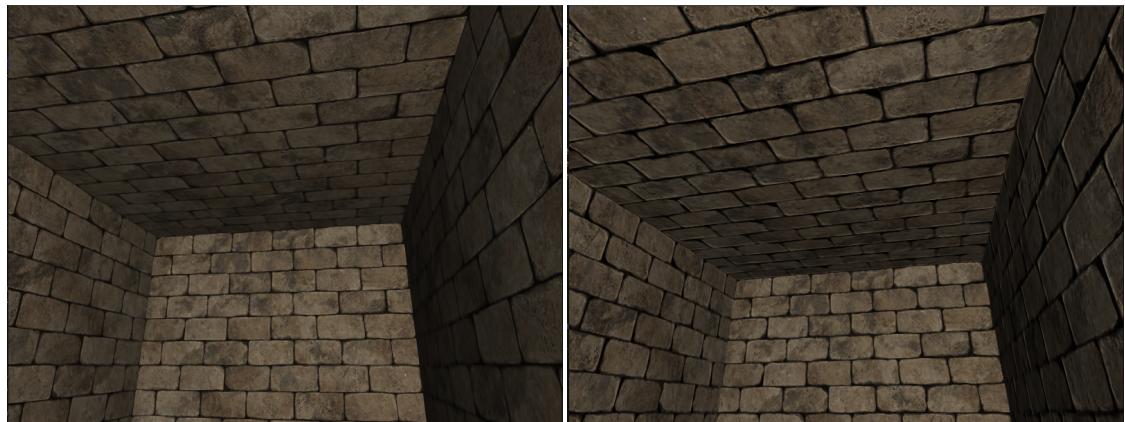


FIGURE 3.1 – La texture avec normal mapping à droite et celle sans à gauche. On peut remarquer que le mur parait avoir plus de profondeur avec le normal mapping

Les réflexions sur l'eau sont très simple. On fait un rendu de la scène depuis le dessous dans un FBO que l'on floute et applique au plan de l'eau.



FIGURE 3.2 – L'image qui sert à la reflection (gauche) comparé à la vision du joueur (droite)

On peut noter aussi la création d'une shadow-map pour la lumière, mais comme elle vient exactement de la position de la caméra on ne voit pas les ombres. De plus j'ai décidé de faire le jeu très sombre donc les ombres ne seraient de toutes manières pas très importantes.

3.2 PARTIE PHYSIQUE

La partie physique du moteur consiste à gérer les collisions. Ces collisions sont gérées à l'aide de AABB (Axis Aligned Bounding Box) associées à chaque modèle. Une fois la collision détectée on réagit en conséquence, qui dans notre cas consiste à faire glisser l'objet rentrant en contact le long de l'objet qu'il rencontre. Cependant cette physique ne s'applique que aux objets et aux joueurs, les monstres sont collision-free.

3.3 PARTIE IA

Pour l'intelligence artificielle, elle est très simple et créée donc de nombreux problèmes. Les monstres ont un rayon d'agression qui une fois franchit par le joueur déclenche leurs agression. Cette agression consiste à aller se coller au joueur lui faisant perdre de la vie.

On voit vite la limitation avec le fait que les monstres n'ont donc aucune connaissance de la topologie du niveau et peuvent donc passer à travers les murs, attaquer un joueur depuis l'extérieur de la carte, etc. Faire une intelligence artificielle plus efficace prends un peu plus de temps. Une amélioration serait notamment de faire un algorithme A* basique, ou plus intéressant une navigation mesh en plus de gérer les collisions.

3.4 PARTIE GAMEPLAY

Le gameplay reste très basique. Le joueur a un nombre d'attaque par secondes, un coefficient d'attaque, de la vie, de l'or et des clés. Les monstres ont les mêmes stats et quand les

deux entre en collisions ils s'attaquent mutuellement. Il n'y a aucun retour à l'écran comme quoi on est effectivement en train d'attaquer le monstre. Une fois le monstre mort, il nous donne notre récompense et le jeu ne l'affiche plus.

De la même manière, une fois une porte ouverte, une clé disparaît de l'inventaire du joueur et elle n'est plus affichée. Enfin les coffres donne leurs contenu au joueur la première fois qu'il rentre en collision avec.

Le donjon est créé à partir d'un fichier texte et d'une image ppm. Le fichier texte doit respecter un certain format (figure 3.3)



FIGURE 3.3 – La carte du donjon avec : en blanc les pièces, en rouge l'entrée, en vert la sortie, en orange les portes et enfin en bleu l'eau (droite) ainsi qu'un exemple de fichier texte (gauche)

3.5 STRUCTURE DU CODE

Si l'on s'intéresse à la structure du code, la majeure partie se trouve dans la classe Game. Game va s'occuper de créer la fenêtre, charger les modèles, charger les shaders pour ensuite lancer la boucle principale qui va effectuer la logique et le rendu à chaque image. Chaque objet important est encapsulé dans une classe (camera, model, player, shader ...). Les objets qui peuvent créer des collisions sont tous des classes filles de la classe actor. Un acteur est un objet avec un modèle, une position et qui peut réagir aux collisions entre lui et le joueur.

4 BUGS CONNUS ET AMÉLIORATION POSSIBLE

Plusieurs bugs sont connus :

- Au niveau des collisions, il est possible de sortir de la carte car on effectue une vérification objet par objet. On peut donc gérer la collision avec un objet en déplaçant le joueur au-delà d'un autre objet du niveau. Pour remédier à cela il faut prendre en compte les objets adjacents. Si cela arrive il faut redémarrer une partie, ou revenir sur la carte à travers un coin.
- Au niveau des déplacements, le fait de faire deux directions à la fois (avant et droite par exemple) fait aller plus vite que d'aller uniquement tout droit. Il faudrait faire la racine du résultat pour être plus précis. La vitesse augmente aussi considérablement si l'on se colle aux murs, dû à la manière dont sont gérés les collisions (sliding).

Au delà des différentes améliorations déjà citées, on pourrait aussi rajouter des animations aux modèles, d'autre lumières dans le niveau ou encore un modèle pour le joueur (un bras avec une épée qui se déplace avec la caméra par exemple). Au niveau graphique, on aurait tout intérêt à définir un champs de vision du joueur et de ne charger que les objets qu'il peut voir, mais étant donné la nature du projet, le backface culling est amplement suffisant pour des performances acceptables sur la plupart des machines actuelles.

5 CONCLUSION

Ce petit projet était assez intéressant, même si je trouve dommage que la plus grande partie du travail soit au final à faire sur le gameplay plus que sur la synthèse d'image. Je dois avouer n'être absolument pas satisfait par le résultat que j'ai eu mais le temps me manquait pour polir le jeu. En effet, je trouve qu'il y a encore trop de duplication de code et il devrait gagner en lisibilité.

6 QUELQUES CAPTURES D'ÉCRAN

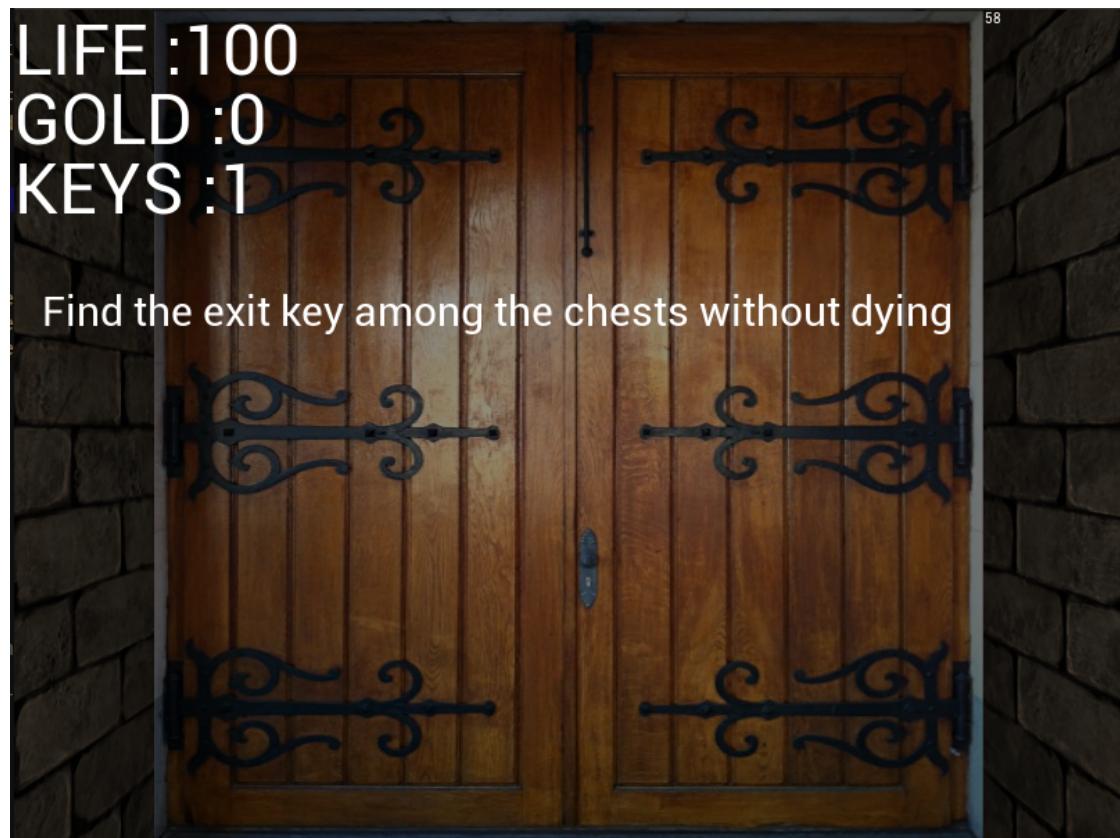


FIGURE 6.1 – La première vision du joueur dans le jeu

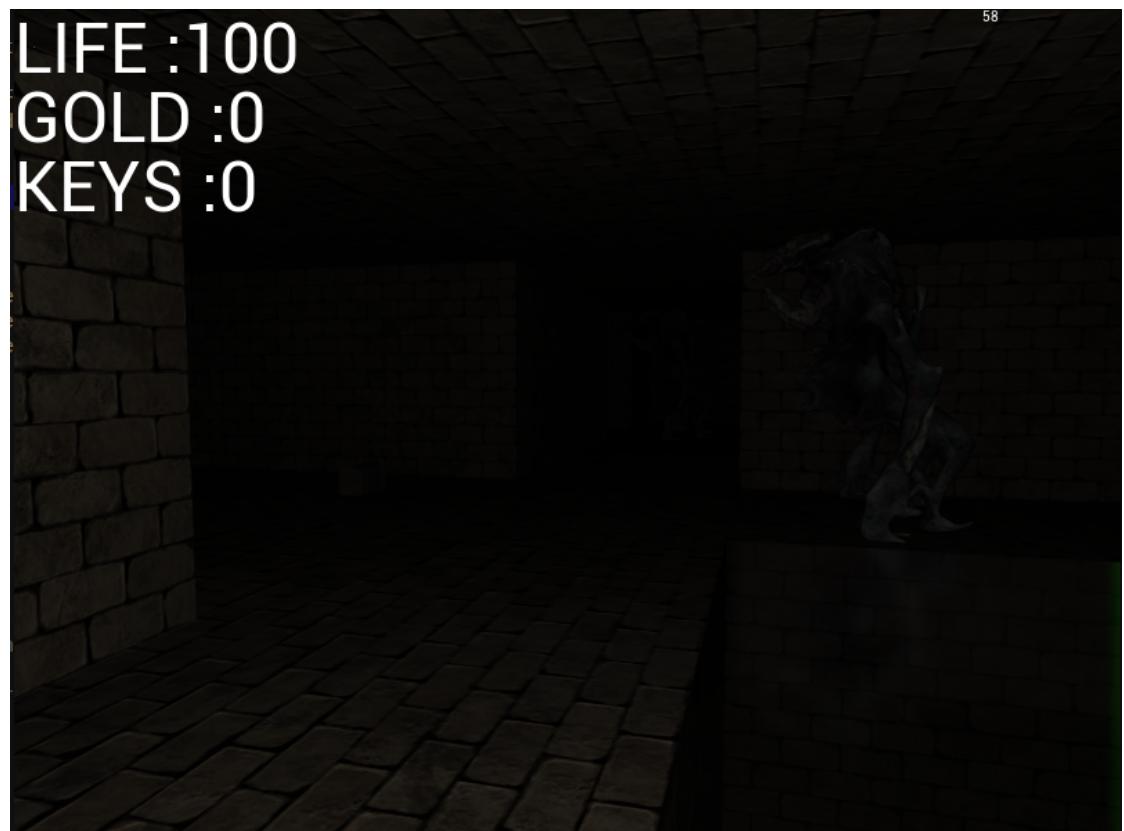


FIGURE 6.2 – It's getting dark in here!



FIGURE 6.3 – Un monstre nous attaque!



YOU WIN !! Press Esc to close

FIGURE 6.4 – Best end screen EVER !