Projet Dungeon Master

Quentin Lebon[1]

[1]ESIEE Paris

Prise en main du jeu

Le déplacement se fait avec les touches ZQSD (ou WASD si clavier QWERTY). La touche A permet de tourner la caméra à gauche (Q pour QWERTY) et la touche E à droite. La barre espace permet d'interagir avec les échelles, les trous, les portes, les items et t'attaquer les monstres. Si le joueur meurt, la touche R permet de recharger le jeu. Les échelles permettent de monter au niveau supérieur, les trous de descendre.

À tout moment, le joueur pour prendre connaissance de ses statistiques via l'interface en haut à droite. Un texte est éventuellement affiché lors d'une action pour mieux permettre de comprendre ce qu'il se passe dans le jeu.

But du jeu

Le but du jeu est d'arriver à l'échelle du dernier niveau du donjon.

Travail à réaliser

Chargement de la carte

Le chargement de la carte à partir d'image, nous utilisons la bibliothèque PPMIO pour lire les images PPM. Les spécifications de l'image sont les mêmes que dans le sujet, la seule modification appliquée est l'absence de code couleur pour la porte. Celle-ci doit être définie dans le fichier JSON.

Chargement des trésors et des monstres ainsi que leurs caractéristiques

Pour implémenter les trésors et les monstres, nous avons choisi le format JSON. En plus d'être visuel, il est bien supporté par la communauté. Nous utilisons la bibliothèque Json for Modern C++ pour parser facilement le Fichier JSON.

Le fichier contient plusieurs couches :

- Levels : cette couche est une liste de tous les niveaux du jeu (avec leurs informations propres).
- Numéro du niveau : cette couche contient le nom de l'image du niveau ainsi qu'une liste de toutes les entités du niveau.
- Items, Monster et Door : Ces couches contiennent une liste de tous les items/monstres/portes présents dans le niveau.
- Item : contient la position, le type, les 2 montants possibles et l'ID de son modèle 3D.
- Monstre : contient la position, le type, la vie, la défense, l'attaque et l'ID de son modèle 3D.
- Porte : contient la position et le prix de la porte.

Les types et les ID sont ici des int. Ces entiers sont listés fichier *Data.hpp* qui contient toutes les constantes du jeu pour une meilleure lisibilité du code C++.

Navigation simple dans le donjon

Pour déplacer le joueur dans le donjon lors de chaque input détecté, nous regardons si le potentiel déplacement est possible. Si la case n'est pas un mur, de l'eau ou si une éventuelle porte est fermée, le déplacement n'est pas possible nous ignorons l'input.

Interaction

La caméra est la FreeflyCamera implémentée lors des TD. Elle permet de se déplacer de case en case et de pivoter à 90°. Cette caméra nous est utile pour interagir avec les trésors et les montres. En récupérant le $frontVector$ et la position du joueur, nous pouvons déterminer les coordonnées de la case en face de celui-ci. Lorsque la barre espace est appuyée nous appelons une fonction callback pour déterminer quelle action va être effectué.

En effet, tous les objets interactifs héritent de la même classe $InteractiveObject$. Cette classe nous permet de mettre à jour les objets via une fonction, mais aussi de définir leur comportement lors des interactions. Par exemple, lorsqu'une interaction est engagée avec un item, nous pouvons déterminer grâce à son type s'il ajoute de la vie, de l'or, de l'attaque ou de la défense. Alors qu'un monstre calculera le nombre de dégâts reçu.

Combat

Les combats sont très simples, lorsqu'une entité combattante (monstre ou joueur) est attaqué, nous appliquons cette formule $vie\_defenseur := vie\_defenseur - (attaque\_attaquant - defense\_defenseur)$. Ici $(attaque\_attaquant - defense\_defenseur)$ est un entier positif.

Le joueur et les monstres héritent d'une classe $CombatEntity$ qui permet de les faire interagir entre eux simplement.

Comportement des monstres

Les monstres font une action toutes les 1 secondes.

Les monstres attaquent le joueur s'ils sont dans la 4-adjacence de celui-ci.

Les monstres peuvent "sentir" le joueur s'il est à moins de 6 cases de distance. Cette distance est simplement une distance cartésienne entre la position du monstre et celle du joueur. Ceci même s'il y a un mur entre le monstre et le joueur.

Si le monstre "sens" le joueur, il se déplace.

Dans l'idéal, le monstre devrait utiliser un algorithme de Dijkstra pour trouver son chemin. Cependant, la contrainte de temps ne nous a pas permis d'implémenter cet algorithme.

Ici, le monstre cherche à se déplacer d'abord sur l'axe X. S'il ne peut pas aller à droite, il va à gauche. S'il ne peut pas aller à gauche, on se déplace alors sur l'axe Y.

Si aucun déplacement n'est nécessaire sur l'axe X, nous nous déplaçons alors sur l'axe Y. Dans la même idée, si on ne peut aller en haut, nous allons vers le bas et si ce n'est pas possible, on se déplace sur X.

À tout cela, nous rajoutons une dernière contrainte qui force le monstre à ne pas retourner sur la case précédemment visitée. Les niveaux étant souvent de long couloir de 1x1, cela permet au monstre de ne pas se bloquer dans un cul-de-sac.

Gestion des modèles 3D

Les modèles 3D, les textures et les shaders sont gérés par des managers.

D'abord, nous chargeons les shaders dans le GPU grâce au $ShadersManager$. Ensuite, nous chargeons chaque texture dans le GPU grâce au $TexturesManager$. L'ID de chaque texture est gardé en mémoire dans le $TexturesManager$.

Pour finir, nous créons un objet $Model$ pour chaque modèle 3D, ces objets sont gérés par le $ModelsManager$.

Lors de la création de l'objet $Model$ nous lui associons l'ID de sa texture et un $ShaderProgram$. Cet objet contient aussi le VBO, VAO, IBO et ainsi que les propriétés du matériau du modèle 3D (KS,KD et shininess).

L'idée ici n'est d'avoir mémoire chaque modèle 3D qu'une fois. Pour faire le rendu de notre modèle 3D, nous passons une $ModelMatrix$ qui est unique à chaque objet du donjon.

Les managers créent les modèles et les textures dans un ordre défini. Cela qui permet de retrouver facilement les différents modèles et textures stockés via un simple entier (cet ordre est défini dans le fichier *Data.hpp*)

Ces managers nous permettent de cloisonner les ressources graphiques du jeu, pour une utilisation plus simple de celle-ci à partir des composants de génération de la carte ou de logique de jeu.

Rendu du donjon

Pour faire le rendu des objets à l'écran, nous utilisons la classe *RenderedObject* qui stocke un objet *Model*, une Model Matrix et la case de l'objet.

Le modèle de lumière utilisé est un Blinn-Phong avec une atténuation quadratique. Ce modèle utilisant les normales des modèles 3D, nous devons les orienter dans le bon sens pour qu'ils soient affichés à l'écran.

Lors de l'initialisation d'un niveau, nous parcourons l'image. En fonction de la nature de chaque case et de celle de ses voisins, nous créer une matrice 4x4 (Model Matrix) qui permet de dessiner le modèle 3D au bon endroit. Nous stockons le tout en créer un *RenderedObject*.

Pour dessiner le donjon, il nous suffit de parcourir notre liste de *renderedobjects* et d'appeler leur fonction *draw*().

Rendu des items/monstres

L'idée est la même que pour le donjon, sauf que ici l'item, le monstre ou l'échelle est placé au centre de la case.

Pour les items, le matériau utilisé est très brillant tout comme pour l'eau. Cela permet de voir plus facilement les items de loin.

Pour aller plus loin

Déplacement monstre

Le déplacement des montres modifié a été décrit plus haut. Ils se déplacent si le joueur est à moins de 6 cases de lui.

Animation Joueur

Lors de la première implémentation, les déplacements avaient effet "diaporama". Le fait de se déplacer instantanément de 1 case nui à l'immersion et à la compréhension des profondeurs.

Pour y remédier, nous avons implémenté une interpolation de déplacement et de rotation. Lorsqu'un déplacement est effectué, les inputs sont désactivés le temps de celui-ci. Il nous suffit d'interpoler la position de la caméra ou sa rotation en fonction du temps. (la durée est de 900ms pour la rotation et 1200ms pour le déplacement)

Grâce à cette technique, lorsque le joueur déplace la caméra, le mouvement est fluide et "naturel".

Facing object

Les items, les monstres et l'échelle sont de simples quads texturés. Si leur orientation est fixe, lorsque le plan de la caméra est perpendiculaire au quad, celui-ci devient invisible.

Il y a deux méthodes pour corriger ce problème. Faire un cube, solution que nous trouvons peu esthétique. Ou alors orienter le quad en permanence vers le joueur.

Cette deuxième solution est assez simple à implémenter. Les monstres, item et l'échelle ont un vertex shader différent qui prend en plus une variable uniforme : la position du joueur. À partir de celle-ci, ils nous suffit de calculer l'angle entre le vecteur (objet->joueur) et le vecteur normal. Une fois l'angle calculé, nous appliquons une rotation de valeur ce même angle selon l'axe Y.

Grâce à cela, notre quad est toujours face au joueur.

Transparence

L'ajout de la transparence a été fait comme expliquer dans le sujet. Cependant, lorsque des objets transparents sont l'un devant l'autre, nous ne pouvons pas les dessiner dans un ordre arbitraire. Sinon, nous aurons un effet de chevauchement. Il faut faire le rendu du plus loin au plus proche. Pour cela, nous trions les objets avant de faire le rendu du plus loin au plus proche. Cela nous permet de toujours rendre l'objet le plus loin en premier et le plus proche en dernier.

Multi-niveau

Pour charger un autre niveau, nous détruisons tous les *RendenredObject* ainsi que tous les items, monstre et échelle. Puis, nous relançons un nouveau chargement, mais cette fois sur le niveau demandé.

Échelle et trou

Pour se déplacer de niveau en niveau, nous avons ajouté des échelles et des trous. Ces objets se comportent comme des *InteractiveObject*. Lorsque l'interaction est demandée, nous effectuons les actions vues dans la section précédente.

Text rendering

Nous avons voulu intégrer une interface par-dessus le rendu 3D. Cette dernière nous permet d'afficher les statiques du joueur, mais aussi des messages en rapport comme les statiques des items ramassées ou le prix d'une porte.

Pour cela, nous avons suivi la ressource de cours via le site LearnOpenGL. Par conséquent, le code de rendu de texte est très similaire à celui du tutoriel.

Pour intégrer le rendu de texte, nous utilisons une classe *TextRenderer*. Cette classe permet de générer les glyphes, de garder leurs ID et caractéristique pour dessiner plus tard les caractères sur l'écran.

Ouverture des portes

L'ouverture des portes est identique à l'animation de mouvement, sauf qu'ici, nous interpolons une translation vers le haut.

Nous avons aussi ajouté un prix à payer pour ouvrir la porte.

Damage feedback

Pour ajouter un peu plus d'immersion, nous avons ajouté un effet de tremblement et de vision "sang" lorsque le joueur prend des dégâts. Cela est effectué en appliquant une fonction sinus en fonction du temps sur la matrix de projection. Il y a donc un effet de déformement de la vision qui se produit. Pour l'effet "sang", nous diminuons les valeurs des canaux bleu et vert et augmentons celui du rouge.

Écran de mort

Lorsque le nombre de points de vie du joueur est inférieur ou égale à 0 la simulation est stoppée. Nous affichons alors un message de mort et l'écran prend une teinte rouge. L'appui sur la touche "R" permet de recharger complètement le jeu et de relancer une nouvelle partie.

Améliorations possibles et défauts

- L'architecture logicielle n'est pas parfaite. Dans le *renderedObject* il n'y a aucune différences entre les facing objects (items, monstres, etc) et les objets statiques. Celle-ci pourrait être sous forme d'une meilleure hiérarchie de classe qui cloisonne mieux spécifications graphiques et celles de gameplay.
- Lorsque qu'on interagit avec un objet, nous sommes obligés de chercher dans le vecteur de stockage. Il faudra modifier toutes les structures de données pour y avoir accès en temps constant.
- Implémenter Dijkstra pour une meilleure IA pour le déplacement des monstres.
- Avoir la possibilité de faire interagir les objets entre eux. Exemple : si le joueur tue un monstre, la mort de celui-ci ouvre une porte.
- La possibilité d'utiliser des modèles 3D.

Références

Ressources graphiques

- Item sont issus Pixel Art Icon Pack RPG par Cainos.
- Monstres sont issus Monsters Creatures Fantasy par LuizMelo.
- Textures des murs, eau, sol et portes sont issues Tiny Texturepack 2 par Screaming Brain Studios . (Upscale grâce à Upscayl)
- Police d'écriture utilisée Dungeon SN Font.

Code C++ utilisées

La fonction GLSL $mat4\ rotationMatrix()$ dans le fichier $3D\_Facing.vs.glsl$ est issue du site de Neil Mendoza.

Rendu de texte

Pour cette partie du projet, j'ai suivi le site conseillé dans les ressources de TD : LearnOpenGL section Text Rendering.

Les fichiers $text.vs.glsl$ and $text.fs.glsl$ sont similaires au tutoriel (delta les noms de variables). La classe $TextRenderer.cpp$ qui permet de stocker les glyphes, générer les quads correspondants à chaque lettre d'une string et calculer la taille (en pixel) d'une chaine de caractères. Le constructeur et la fonction $renderText()$ utilisent du code du tutoriel, celui-ci a été modifié pour l'adapter pour à la POO.

Bibliothèques utilisées

GLFW, GLAD, Glimac, GLM, PPMIO and Json for modern C++.