Abed Ylies
Merad dhiaeddine

RAPPORT DE PROJET

1. Architecture du Projet

Le projet adopte une structure **MVC (Modèle-Vue-Contrôleur)**. Cette structure permet de séparer le jeu en trois parties permettant de gérer chaque chose séparément et, par exemple, comme c'est le cas dans ce projet, permettre deux vues différentes. Une section utilitaire pour ne pas rendre les codes du modules trop chargé.

1.1. Le Modèle (src/model/)

Cette partie gère tout ce qui est nécessaire au bon fonctionnement de la logique du jeu. Elle gère donc les états, les entités ainsi que la physique du jeu. Cette partie est donc divisée en trois sections. Cette partie n'ayant aucune dépendance directe vers les autres a requis certains ajustements, comme pour les animations où le modèle laisse un indice de frames en fonction du temps écoulé que la partie vue utilisera.

A. Les Entités

Cette section regroupe toutes les entités du jeu. Chaque entité est définie dans une structure, dans son fichier d'en-tête, son fichier source gérant quant à lui par exemple la logique (ex : le déplacement, le nombre de projectile jetable, etc...) ou tout ce qui est en rapport avec la gestion en mémoire de la structure (allocation, destruction).

- **player** : Ce module initialise et libère la mémoire du joueur avec ses propriétés (position, santé, score). Elle contrôle son déplacement horizontal limité aux bords de l'écran et gère un système de tir avec temporisation entre les tirs ainsi que son animation. Elle attribue aussi une valeur que l'on récupérera pour le score.
- **Enemy** : Ce module gère les vagues d'ennemis avec deux comportements distincts. Pour le premier niveau, elle crée un essaim se déplaçant horizontalement et verticalement en groupe, avec une augmentation de la vitesse par membre en moins et un système de tir aléatoire du membre le plus bas de chaque colonne de l'essaim. Pour le second niveau, un boss unique se déplace horizontalement en continu et effectuant des tirs triples.
- **projectile** : Ce module gère les projectiles utilisant un pool d'objets (mémoire pré-allouée et réutilisée) pour optimiser les performances. Elle permet de créer et de faire évoluer des projectiles avec différents comportements : déplacement horizontal (pour le joueur et les ennemis) et diagonal (pour les attaques spéciales du boss). Les projectiles sont automatiquement désactivés lorsqu'ils quittent les limites de l'écran ou entrent en collision.
- **bunker** : Ce module sert à la création des bunkers, le bunker est mis en place sous la forme d'une grille de bloc que l'on désactive si une collision est détectée (la détection se fait dans le module physics).
- **explosion** : Ce module gère les explosions à l'aide là aussi d'un pool d'objets. On les déclenche à l'aide de coordonnées. Chaque explosion possède une durée de vie, un état qui évolue avant sa désactivation.

B. Gestion du jeu

Ici sont les modules servant à gérer le jeu, la physique de ses entités, ses états et sa gestion du score.

- **physics** : Ce module sert à vérifier si des hitboxes entrent en collision et applique l'effet de la collision (ex : si une balle touche le joueur, il perd une vie et elle permet aussi de déclencher la fin de la partie si le joueur perd sa dernière vie), en faisant le lien entre l'entité touchée par le projectile et l'explosion.
- **game_state** : C'est une simple énumération qui sert de tour de contrôle pour tout le projet. Il permet de segmenter le programme en quatre phases distinctes. Quand le jeu est au menu, quand il est en cours, quand il est en pause et quand il est terminé.
- **storage :** Cette fonction récupère le high score d'un fichier json, si il n'existe pas elle récupere 0, et enregistre le score du joueur si il est supérieur au high score récupéré.

1.2. La Vue (src/view/)

Cette partie sert au rendu visuel à l'aide des données que la partie modèle nous donne. Le modèle étant indépendant, cela nous permet de créer deux interfaces graphiques différentes.

- **sdl_view** : Il s'agit d'une interface graphique. Cette interface utilise les bibliothèques SDL3 pour charger des sprites, gérer les fenêtres et jouer les sons. Elle transforme les coordonnées du modèle en pixels. Elle se charge aussi des menus.
- **ncurses_view** : Il s'agit d'une interface textuelle. Cette interface utilise la bibliothèque Ncurses afin de pouvoir dessiner le jeu dans un terminal. Pour cela, on traduit simplement nos entités en modèle ascii. L'interface se charge aussi d'afficher le menu et de gérer ses options.

1.3. Le Contrôleur (src/controller/)

C'est cette partie qui fait le lien entre les entrées de l'utilisateur et le modèle. Deux contrôleurs différents sont présents ici car les bibliothèques SDL3 et Ncurses gèrent le déplacement de manière différente. Les contrôleurs prennent aussi en compte l'état du jeu (Les entrées n'ont pas les mêmes utilités selon l'état du jeu)

- **sdl_controller** : Le module sert à faire le lien entre les entrées détectées et les actions qu'elles sont censées produire dans le modèle. Les contrôles sont faits à l'aide des bibliothèques de SDL3.
- **ncurses_controller** : La même chose que le précédent sauf que celui-ci la fait avec la bibliothèque Ncurses.

1.4. Les Utilitaires (src/utils/)

Ici sont regroupé les modules ne servant pas la logique du jeu, mais servant une logique utilitaire.

- **json_helper :** Un utilitaire servant à récupéré le plus grand int présent dans un fichier 0 sinon, 0 et pouvant écrire dans un dossier un couple clé valeur.

1.5. Le main

Le main assure la liaison entre les trois parties en trois étapes clés :

1. **Initialisation** : Configuration des structures du modèle.
2. **Implémentation** : Chargement de la vue et du contrôleur sélectionnés.
3. **Boucle de jeu** : Cycle continu récupérant les entrées du contrôleur pour mettre à jour le modèle, puis demandant à la vue d'afficher l'état actualisé.

2. Les Décision de Conception

2.1. Les pools

Afin d'éviter une trop grosse utilisation des allocations dynamiques car ceux si étant coûteuses et pouvant générer des fuites mémoires. Nous avons opté pour les pools d'objets. Ils sont utilisés pour les entités qui, sans le pooling, doivent être allouées plusieurs (les projectiles et les explosions). Cela permet alors de faire une seule allocation au lancement et de simplement les activer et les désactiver au besoin.

2.2. Animation cyclique

Pour le joueur a décidé de faire une animation cyclique, c'est-à-dire que l'animation part d'un point A et effectue les autres animations avant de revenir à ce point A et de recommencer. Nous avons choisi cela pour le player au lieu d'animer les mouvements afin que le jeu nous donne un retour visuel même à l'arrêt.

2.3. Logique de tir pour l'essaim

Nous avons implémenté une logique dans le tir de l'essaim, le membre nous tirant dessus est toujours celui le plus bas de sa colonne. Cela permet un rendu visuel meilleur pour le joueur, il sait de quel membre de l'essaim le tir peut venir et permet aussi de ne pas avoir à gérer le cas où l'essaim se tire lui-même dessus.

2.4. Modularité des composants du jeu

Toutes les entités se trouvent dans un module séparé. Cette division nous a permis de développer, tester et modifier chaque composant indépendamment.

3. Les Difficultés rencontrées

3.1. La mise en place de l'animation

Trouver une solution pour intégrer des animations tout en gardant le modèle indépendant de la vue, nous a demandé de faire en sorte de laisser le modèle se charger de donner un indice de frames que la vue récupérera pour y appliquer l'animation appropriée.

3.2. Évitez les fuites de mémoires

La gestion de la mémoire était la difficulté principale rencontrée lors de ce projet, car SDL3 contient certaines fuites internes indépendantes de notre code dues aux drivers requis pour son fonctionnement. Cela nous a demandé de créer mysuppressions.supp qui fonctionne lors de l'appel De Valgrind qui ne prend pas ces leaks en compte, ce qui nous permet de nous concentrer exclusivement sur la détection et la correction des fuites réelles de notre programme.

4. Les Tests réalisés

4.1. Test de portabilité du modèle

Pour vérifier que le modèle était bel et bien indépendant et que l'expérience ne change pas selon l'interface graphique. Nous avons fait des tests systématiques afin de vérifier si des différences apparaissaient entre les interfaces (ex : vitesse des projectiles).

4.2. Test de Mémoire

Ce qui a composé la majorité de nos tests vu les tests de fuites de mémoires à l'aide de Valgrind, comme expliqué plus haut, la création de mysuppressions.supp nous a permis de pouvoir nous concentrer sur notre programme et le fait que nous avons isolé au mieux la plupart de nos modules a permis de faire les tests sur chaque module de manière indépendante.

5. Diagramme des modules