Introduction

J’ai développé un jeu 2D intitulé **Shooter**. Il s’agit d’un jeu de type arcade dans lequel le joueur doit survivre face à des vagues successives de monstres et tenter de réaliser le meilleur score possible. L’objectif principal de ce projet est d’appliquer des notions de programmation bas niveau, de gestion des ressources, et de conception modulaire via la SDL.

Installation et Pré-requis

Pour faire fonctionner le jeu, les éléments suivants sont nécessaires :

- La bibliothèque **SDL2** (ainsi que SDL_image pour le chargement des textures),
- Le **Makefile** fourni permet de compiler le projet facilement via la commande `make`.

  Un alias a également été ajouté pour regrouper les commandes de compilation, d’exécution et de nettoyage des fichiers intermédiaires.

Présentation du Jeu

Le gameplay est simple et accessible : le joueur contrôle un personnage qui doit **éliminer des monstres** apparaissant en boucle à l’écran. La partie prend fin lorsque le joueur est tué ou choisit de quitter.

Le jeu est ponctué par un **événement spécial** : la **pluie de comètes**, qui survient après un certain temps de jeu pour augmenter la difficulté.

Le score est basé sur le nombre de monstres éliminés, chacun apportant un certain nombre de points selon son type.

Types de Monstres

Deux types principaux d’ennemis sont implémentés :

- **Mummy**
  Petit monstre peu résistant, apparaissant en grand nombre.
    - Dégâts faibles
    - Vitesse variable
    - Faibles points de vie
- **Alien**
  Ennemi de plus grande taille, plus lent mais plus redoutable.
    - Dégâts élevés
    - Résistance importante
    - Apparition moins fréquente

Les Comètes

Les **comètes** sont des éléments environnementaux aléatoires qui tombent du ciel après un certain temps de jeu. Elles représentent un **danger supplémentaire** :

- Leur apparition est aléatoire,
- Une collision inflige des **dégâts aléatoires** au joueur,
- Elles renforcent la difficulté et apportent du dynamisme au jeu.

Architecture du Projet

Le code est **modularisé** en plusieurs fichiers source (.c) et d'en-tête (.h), chacun responsable d'un aspect spécifique du jeu :

- `player.*` : gestion du joueur (déplacements, points de vie, affichage).
- `monster.*` : gestion des ennemis, avec support pour différents types.
- `projectile.*` : logique de tir et collision avec les monstres.
- `game.*` : boucle principale, gestion des événements, score, et affichage global,
- `comet.*` : apparition et gestion des comètes.

Chaque header déclare des structures de données et fonctions claires, et favorise la **réutilisabilité** du code voici le detaille de certainne fonction en détaile la suite dans le projet.

`player.c` – Gestion du joueur

`void MoveLeft(Player *player)`

Déplace le joueur vers la gauche, si celui-ci n'est pas collé au bord gauche de l'écran.

- Vérifie que la position `x` est > 0.
- Diminue `rect.x` de la valeur `velocity`.

`void MoveRight(Player *player, Monster *monster)`

Déplace le joueur vers la droite tout en vérifiant deux conditions :

- Qu'il ne dépasse pas la largeur de l'écran (`SCREEN_WIDTH`)
- Qu'il ne rentre pas en collision avec un monstre (grâce à `checkcollision`)

`void HealthBarPlayer(SDL_Renderer *render, Player *player)`

Affiche une barre de vie au-dessus du joueur :

- Trace un rectangle rouge (bordure),

- Puis un rectangle vert proportionnel à la santé restante (`health / max_health`).

`void DamagePlayer(Player *player, double damage)`

Inflige des dégâts au joueur :

- Si la santé restante est suffisante (`health - damage > damage`), elle est réduite.
- Sinon, le joueur est marqué comme mort (`is_alive = false`).

`void LaunchProject(Projectile *project[], SDL_Renderer *render, Player *player)`

Permet au joueur de tirer :

- Cherche un projectile inactif dans le tableau `project`.
- Initialise ce projectile avec `initProjectile`.
- Active ce projectile (`is_projectile_active = true`).

`void initPlayer(Player *player, SDL_Renderer *render, const char *imgpath)`

Initialise toutes les propriétés du joueur :

- Position, taille, texture, santé, vitesse, etc.
- Associe aussi les **fonctions de comportement** du joueur via des pointeurs (comme `MoveLeft`, `DamagePlayer`, etc.).

`player.h` – Déclaration du joueur

`typedef struct Player`

Définit la structure du joueur avec :

- Son rectangle d'affichage (`rect`), texture, vie, score...
- Des pointeurs de fonction pour encapsuler son comportement (`MoveLeft`, `HealthBarPlayer`, etc.).

`projectile.c` – Gestion des projectiles

`void initProjectile(Projectile *project, SDL_Renderer *render, const char *imgpath, Player *player)`

Prépare un projectile :

- Positionné à droite du joueur, centré verticalement.

- Chargement de sa texture.
- Angle initial = 0, vitesse fixée.

`void MoveProject(Projectile *project, Monster **monster, Player *player)`

Fait avancer le projectile :

- Augmente sa position `x` selon sa vitesse.
- Fait tourner le projectile (angle += 12).
- Si le projectile sort de l'écran, il est désactivé.
- Vérifie la collision avec chaque monstre :
    - Si un monstre est touché, on réduit sa santé et désactive le projectile.

`void resetProjectile(Projectile *project, Player *player)`

Réinitialise un projectile à sa position d'origine :

- Recalcule sa position par rapport au joueur.
- Le désactive (`is_projectile_active = false`).

`projectile.h` – Déclaration des projectiles

`typedef struct Projectile`

Structure d'un projectile :

- Rectangle, texture, vitesse, angle de rotation.
- Flags pour savoir s'il est actif ou hors de l'écran.

`monster.c` – Comportement des monstres

`void forward(Monster *monster, Player *player)`

Fait avancer le monstre vers la gauche :

- Si pas en collision avec le joueur → avance.
- Si collision → inflige des dégâts au joueur.

`void DamageMonster(Monster *monster, double damage)`

Inflige des dégâts au monstre :

- S'il lui reste suffisamment de santé, on réduit `health`.

- Sinon, il est marqué comme mort (`is_alive = false`).

`void HealthBarMonster(Monster *monster, SDL_Renderer *render)`

Affiche la barre de vie du monstre au-dessus de lui :

- Rectangle rouge (bordure),
- Rectangle vert (vie restante).

`void resetMonster(Monster *monster)`

Réinitialise un monstre après sa mort :

- Réinitialise position `x`, vitesse aléatoire, santé, etc.
- Fonction différente selon qu'il s'agisse d'un `MONSTER` ou d'un `ALIEN`.

`void freeMonsters(Monster *monster[])`

Libère la mémoire de tous les monstres :

- Libère chaque texture,
- Puis libère la structure `Monster`.

`void initMonster(Monster *monster, SDL_Renderer *render, const char *imgpath)`

Initialise un monstre de type `MONSTER` :

- Position, vitesse, points de vie, points de loot.
- Associe les comportements via des pointeurs (`forward`, `DamageMonster`, etc.).

`void initAlien(Monster *alien, SDL_Renderer *render, const char *imgpath)`

Initialise un monstre de type `ALIEN` :

- Similaire à `initMonster`, mais :
    - Plus gros,
    - Plus résistant,
    - Plus lent,
    - Plus de dégâts.

`monster.h` – Déclaration des monstres

`typedef enum MonsterType`

Définit deux types de monstres :

- `MONSTER` : standard
- `ALIEN` : plus dangereux

`typedef struct Monster`

Structure d’un monstre :

- Coordonnées, texture, vie, vitesse, type, etc.
- Pointeurs de fonction pour comportements personnalisés.

Remerciements

Ce projet a été inspiré d’une série de tutoriels vidéos réalisés par **Graven Développement** sur YouTube, initialement proposés avec **Pygame** (Python). J’avais déjà réalisé une première version de ce jeu en Python, et ce projet m’a permis de **reproduire et adapter l’ensemble en langage C avec SDL**, une bibliothèque beaucoup plus bas niveau.

Bibliographie

- **Zeste de Savoir** – *Tutoriel "Utiliser la SDL en langage C"*
  Pour une introduction rapide à la SDL.
- **Site officiel de SDL** – *https://www.libsdl.org/*
  Documentation et guide d'installation.
- **Chaîne YouTube de Graven** – *Développement de jeux en Python avec Pygame*
  Source d’inspiration initiale.
- **ChatGPT** – *Assistance à la correction et à l’ajout des commentaire*