Marina Barannikov
Romain Rabreau

Rapport de projet - Invasion MIDO

Nous présentons notre compte rendu du projet Invasion MIDO, que nous avons appelé Licence VS Étudiant. Nous avons décidé de séparer ce compte rendu en plusieurs parties :

1. Nomenclature du code et des structures de données du jeu

Notre code est organisé en 10 fichiers. Le fichier contenant la fonction main principale lancée lors de l'exécution du jeu est dans `main.c` . Dans ce fichier il y a aussi la définition du menu principal et la fonction d'initialisation de partie. Le répertoire `src` contient les fichiers utilisés à des fins spécifiques du code :

- `Tourelles.c`

Définit les structures représentant les tourelles de défense, et les fonctions pour les interactions entre les tourelles et le jeu, notamment l'initialisation et la gestion de la mémoire.

- `ennnemis.c`

Définit les structures représentant les ennemis du jeu, l'initialisation de ces ennemis, les actions des ennemis spéciaux, et l'allocution mémoire.

- `Visuels.c`

Instructions dont l'intérêt principal est l'affichage (menus, titres, visualisation des vagues d'ennemis)

- `Jeu.c`

Fonctions pour jouer un niveau, avec la boucle de tour de partie, un niveau étant composé de plusieurs tours

- `Score.c`

Fonctions de mise à jour du score en cours de partie, et fonctions pour la gestion et l'affichage du leaderboard des joueurs par niveau.

- `Grille.c`

Affichage tour par tour de l'état du jeu, de la grille des ennemis et des tourelles

- `Sauvegarde.c`

Fonctions pour la sauvegarde de parties dans un fichier texte, et pour le chargement des données de parties sauvegardées dans le jeu courant.

- `helpers.c`

Fonctions auxiliaires de traitement de texte et de chemin de fichier, pour la sauvegarde et le leaderboard.

- `Header.h`

Contient les prototypes des fonctions et les bibliothèques externes que nous utilisons : `<stdio.h>, <stdlib.h>, <string.h>, <limits.h>,<unistd.h>,<dirent.h>, <sys/stat.h>`

Contient les macros pour le nombre d'ennemis, de tourelles, de niveaux, les dimensions du plateau, ainsi que les séquence couleurs que nous utilisons

Nous modélisons le jeu via les structures de données struct suivantes :

- Ennemis représentés par la structure "étudiant". Chaque type d'ennemi est représenté par un symbole, qui est sauvegardé en tant que int correspondant à la valeur ASCII de ce symbole

```c
typedef struct etudiant {
   int type; // type correspond à la valeur ASCII du symbole associé
   int pointsDeVie;
   int ligne;
   int position;
   int vitesse;
   int tour;
   int immobilisation; // attribut modifié lors d'une attaque de tourelle P
   int deplace;// 1 si l'ennemi a été déplacé, uniquement pour le type F
   struct etudiant* next;
   struct etudiant* next_line;
   struct etudiant* prev_line;
} Etudiant;
```

Ligne et position définissent l'emplacement de l'ennemi sur le plateau. La position 0 correspond à la ligne d'arrivée, qui marque la victoire des ennemis. Les ennemis entrent sur le plateau à la position 15 et sont en attente position 16.
Un double chaînage par ligne est implémenté, avec *prev_line* pointant sur l'ennemi étant entré sur le plateau avant, et *next_line* celui suivant. Un chainage simple dans l'ordre d'apparition au long de la partie est aussi implémenté. *Tour* correspond au tour auquel l'ennemi va apparaître.

- Une structure "TypeEnnemi" pour se référer aux caractéristiques pour chaque type d'ennemi, instanciée une fois en tant que constante du jeu

```
typedef struct {
   const char symbole; // Caractère représentant le type dans le fichier
   int pointsDeVie;
   int vitesse;
   const char* nom;
   const char* description;
} TypeEnnemi;
```

Avec ces types possibles :

- ❖ **Étudiant ('Z')** : 3 PV, vitesse 2. Étudiant de base, avance à moitié endormi.
- ❖ **Étudiant L1 ('L')** : 9 PV, vitesse 1. Résistant mais aussi le plus lent, fait de gros dégâts sur sa ligne.
- ❖ **Étudiant Talent ('X')** : 2 PV, vitesse 4. Plus rapide, moins résistant.
- ❖ **Syndicaliste ('S')** : 4 PV, vitesse 1. Augmente la vitesse des ennemis adjacents de sa ligne lorsqu'il meurt.
- ❖ **Fainéant ('F')** : 6 PV, vitesse 3. Fait des sauts aléatoires (parfois sur la ligne d'en dessous) ou ne bouge pas pendant plusieurs tours, résistant.
- ❖ **Doctorant ('D')** : 3 PV, vitesse 1. Résistant et soigne les ennemis de 1 PV par tour sur une zone de 3 cases et 3 lignes.

- Tourelles de défense représentées par la structure Tourelle

```
typedef struct tourelle {
   int type;
   int pointsDeVie;
   int ligne;
   int position;
   int prix;
   struct tourelle* next;
} Tourelle;
```

Les tourelles sont chaînées par ordre dans lequel elles vont tirer.

- Une structure TypeTourelle pour se référer aux caractéristiques pour chaque type de tourelle, instanciée une fois en tant que constante du jeu

```
typedef struct {
   const char symbole;
```

```
    int pointsDeVie;
    int prix;
    const char* nom;
    const char* description;
} TypeTourelle;
```

Ces tourelles sont actuellement disponibles :

- **Tableau noir ('T')** : 3 PV, 80 ECTS. Tourelle de base, envoie des craies sur les étudiants.
- **Diplôme LSO ('O')** : 1 PV, 40 ECTS. Tourelle mine, explose au contact et détruit l'étudiant puis s'auto-détruit.
- **BU ('B')** : 10 PV, 120 ECTS. Maxi mur de livres, ralentit les étudiants.
- **Feuille de présence ('P')** : 2 PV, 120 ECTS. Immobilise l'étudiant pendant 2 tours.
- **Emmanuel Lazard ('E')** : 2 PV, 150 ECTS. Multi-dégâts sur 3 lignes en même temps.
- **Eduroam ('R')** : 1 PV, 100 ECTS. Comportement aléatoire, une fois sur trois, l'ennemi recule, avance de 1, ou avance de 2.

- L'ensemble du jeu est modélisé par la structure "Jeu"

```
typedef struct {
    // pointeur vers la première tourelle
    Tourelle* tourelles;
    // pointeur vers le premier ennemi
    Etudiant* etudiants;
    int cagnotte;
    int tour;
    int score;
    char pseudo[50];
    char fichier_ennemis[255];
} Jeu;
```

`fichier_ennemis` correspond au fichier de niveau chargé en début de partie, cagnotte au solde de ce niveau permettant d'acheter des tourelles, tour correspond au tour en cours, et pseudo est choisi lorsque le jeu est lancé.

2. Déroulement global du programme

Lorsque le programme est lancé avec `./jeu.out`, une courte introduction est présentée, puis demande à l'utilisateur un pseudo. On rentre ensuite dans la boucle principale du jeu qui ne s'arrête qu'avec l'option Quitter. On effectue ces étapes :

2.1 **Affichage du menu principal** :

La fonction `Menu(&erreur)` est appelée pour effacer l'écran, afficher le titre et proposer cinq options au joueur grâce à la fonction `AfficherChoix(options, nbOpions, err)` (fichier Visuels.c). Elle affiche au centre du terminal les options en argument, et renvoie un entier entre 1 et *nbOptions* en fonction du choix de l'utilisateur. Ici, les options sont :

1. "Jouer les niveaux" (Niveaux préenregistrés)

On lit les niveaux préenregistrés dans le dossier "Niveau" avec la fonction `LectureNoms("Niveau", &nbNiveaux, err)`, (helpers.c). Elle renvoie la liste des noms de fichiers du dossier en argument. Ensuite, on rend les noms lisibles pour l'utilisateur avec `FormatterNoms(noms, nbNiveaux, err)` (helpers.c), qui renvoie la liste des noms de fichiers sans les "_" et l'extension .txt. Enfin, on demande à l'utilisateur de choisir un des niveaux proposés avec la fonction `AfficherChoix` et on retourne l'adresse du fichier texte choisi par l'utilisateur.

2. "Charger une partie" (Partie du joueur)

Le programme demande à l'utilisateur d'entrer l'adresse du fichier de partie (par exemple, `"Dossier/2_Talents_De_Sprinteur.txt"`). Si la saisie est correcte (non vide), le chemin est enregistré et retourné.

3. "Reprendre une partie sauvegardée" (Partie déjà entamée)

Les parties sauvegardées sont dans le dossier `Sauvegardes` , et sont lues grâce à `LectureNoms()`. Les noms des fichiers sont rendus lisibles comme précédemment avec

`FormatterNoms` et une étape supplémentaire retire le suffixe `"save"` (qui permet de savoir qu'il s'agit d'une sauvegarde) pour améliorer la lisibilité. On demande à l'utilisateur de choisir l'un des niveaux sauvegardés avec la fonction `AfficherChoix` et on retourne l'adresse du fichier texte choisi par l'utilisateur.

4. "Classements"

La sélection appelle la fonction `ChoixLeaderboard` (helpers.c) qui, dans l'esprit des cas précédents, lit les fichiers se terminant par le suffixe "_leaderboard.txt" avec `LectureNoms`, puis appelle `FormatterNoms`, et enfin demande à l'utilisateur quel classement il souhaite consulter avec `AfficherChoix`. Toujours dans `ChoixLeaderboard`, on appelle `AfficherLeaderboard (score.c),` qui affiche le fichier texte de classement. Une fois la touche "entrée" appuyée, on sort de ces fonctions et de retour au Menu, on retourne "*NULL*" car on ne souhaite pas lancer de partie.

5. "Quitter" (Arrête le programme)

Cette dernière option arrête le programme avec `exit(0)`. Aucune mémoire n'est à libérer, car nous n'avons pas utilisé de mémoire dynamique jusqu'ici.

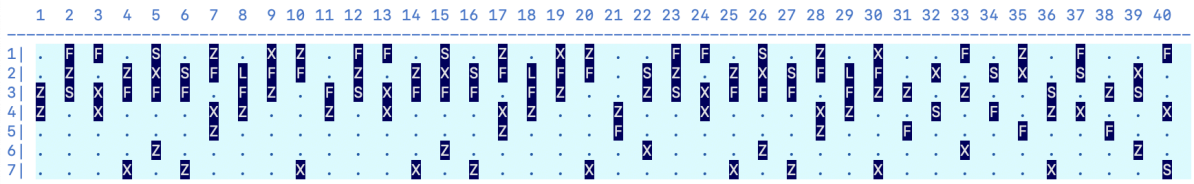
De retour dans le main, on crée une structure Jeu et on associe à la caractéristique "fichier_ennemis" de celle-ci le chemin renvoyé par le Menu. Si il a retourné NULL, on rentre dans `JouerPartie() (game.c)` mais on en ressort immédiatement car *'jeu'* n'est pas initialisé. *'jeu'* est immédiatement désallouée de la mémoire avec `LibererJeu(&jeu).` Si le menu renvoie une adresse, alors on rentre dans la phase de préparation/reprise de partie.

2.2 **Préparation ou reprise de la partie** :

Si le chemin commence par "Sauvegardes/", le programme relance une partie avec `RelancerPartie (sauvegarde.c)`, Sinon, il prépare une nouvelle partie via `PreparerPartie (main.c)`.

Dans le cas d'une nouvelle partie, on lit le fichier texte passé en argument (via la Jeu->fichier_ennemis). Puis, on effectue les actions suivantes :

- Lire la cagnotte en première ligne et l'associée
- Initialiser le chaînage des ennemis lus ligne par ligne avec `InitialisationEnnemis(ennemis.c)` , les affiche avec `VisualiserEnnemis (Visuels.c)` afin que le joueur puisse se préparer à l'assaut.



Visualisation des vagues d'ennemis

- Configurer le chaînage des tourelles en appelant `InitialisationTourelles() (tourelles.c)`. C'est à cette étape que le joueur rentre ses tourelles au clavier, ligne par ligne avec le format [Symbole de la tourelle] [position], [symbole 2] [position2] ect.
- Copier le pseudo rentré plus tôt dans la structure du jeu et initialise le score ainsi que le numéro de tour.

Dans le cas d'une reprise d'une partie sauvegardée, on lit le chemin passé en argument (via chemin_save), puis on effectue les actions suivantes :

- Lire le Tour et la cagnotte lors de la pause de la partie et les remettre dans la structure jeu.
- Remettre le pseudo de la personne qui a mis la partie en pause dans la structure jeu, car on ne vole pas la partie de quelqu'un !
- On refait le chaînage Next des Tourelles en lisant les lignes une par une. Les données des Tourelles sont données dans cet ordre : type, pointsDeVie, ligne, position, prix.
- Puis on refait le chaînage Next pour les Etudiants dans cet ordre : type, pointsDeVie, ligne, position, vitesse, tourEnnemi, immobilisation

Voici un exemple d'un fichier `txt` sauvegardé :

Sauvegardes > 3_Masters_En_Piste_(Romain)_proche de la défaite_save.txt

```
1   Tour 19
2   Cagnotte 100
3   Pseudo Romain
4   NbTourelles 6
5   Tourelle T 3 1 1 100
6   Tourelle T 3 2 3 100
7   Tourelle E 2 3 2 150
8   Tourelle T 3 4 4 100
9   Tourelle T 3 6 4 100
10  NbEtudiants 3
11  Etudiant Z 3 5 2 1 5 0
12  Etudiant M 4 1 5 1 8 0
13  Etudiant M 4 7 5 1 8 0
```

Une fois la phase de préparation ou de reprise de partie terminée, la fonction `JouerPartie(&jeu, &erreur)` (issue du fichier *game.c*) est appelée dans le `main`. C'est à l'intérieur de cette fonction que la boucle de jeu principale s'exécute, et elle ne se termine que lorsque l'on constate que la partie est gagnée, perdue ou mise en pause. Elle appelle successivement des fonctions nécessaires au bon déroulement du tour. En voici son fonctionnement :

2.3 Déroulement de la partie :

Au début de chaque itération de la boucle, on incrémente `jeu->tour` de 1 pour passer au tour suivant. Cela permet notamment de "déclencher" l'apparition des étudiants prévus à ce tour. (Note : Nous effectuons cette opération en début de boucle afin de ne pas rentrer dans toutes fonctions inutilement car aucun étudiant n'est encore sur le plateau.)

Apparition des ennemis – `ApparitionEnnemis(jeu, erreur)`
Les ennemis qui doivent apparaître au tour courant, ou à un tour précédant et qui ne sont pas encore apparus puisque la case était occupée sont identifiés. On choisit le premier qui correspond à ce critère, pour chaque ligne

Affichage du plateau
On efface d'abord l'écran du terminal afin de donner une impression de "fluidité" lorsque les tours s'enchaînent rapidement. Puis on appelle la fonction `AfficherPlateau(jeu)(grille.c)` qui affiche le tour courant ainsi que le score du joueur. Ensuite, on créer un tableau de dimension NB_LIGNES x (NB_EMPLACEMENTS + 1) remplit de "." pour désigner les cases possibles du plateau. On passe successivement dans les listes chaînées des Tourelles puis des Étudiants : on remplace le "." si un entité est présente et à une vie strictement positive par [points de vie][type].

Ensuite on affiche le plateau avec une double boucle for, ainsi que des éléments visuels comme les numéros de ligne et de position. Voici une idée du rendu lors d'une partie :

```
========== DÉBUT DU TOUR 9 ==========
     1    2    3    4    5    6    7    8    9    10   11   12   13   14   15          Score : 400
-----------------------------------------------------------------------------------------------
  |
1 | 3T   1R   2E   10B  .    .    .    .    .    .    .    4S   .    6F   .
  |
2 | 3T   1R   2E   10B  .    .    .    .    3D   6F   .    3Z   2X   .    3D   3D
  |
3 | 3T   1R   2E   10B  .    .    .    4S   3Z   3D   3D   .    6F   3Z   3Z
  |
4 | 3T   1R   2E   10B  .    .    .    2X   .    .    .    .    .    .    .
  |
5 | 3T   1R   2E   10B  .    .    .    .    .    .    .    .    .    .    .
  |
6 | 3T   1R   2E   10B  .    .    .    .    .    .    .    .    .    .    .
  |
7 | 3T   1R   2E   10B  .    .    .    .    .    .    .    .    .    .    .
  |
```

Actions des tourelles – `ResoudreActionsTourelles(jeu, erreur)`
Pour chaque tourelle vivante (pointsDeVie > 0) de `jeu->tourelles`, on recherche l'étudiant pouvant être attaqué (car leur chaînage ordonné permet d'attaquer seulement le premier devant la tourelle). Il est cible de la tourelle si il est sur la même ligne, vivant, et a position strictement devant la tourelle (entre la tourelle et la fin du plateau pour ne pas toucher ceux qui ne sont pas encore dans la partie).

Selon le type de tourelle, on applique les effets correspondants :

- `'T'` : tourelle basique, inflige 1 point de dégât à l'ennemi détecté
- `'O'` : mine LSO, explose avec l'ennemi, s'il est une case devant elle (position tourelle + 1 = position ennemi).
- `'B'` : (BU) ne fait rien car c'est un mur défensif.
- `'P'` : feuille de présence, immobilise l'ennemi pendant 1 tours tous les 3 tours.
- `'E'` : Emmanuel Lazard, la tourelle la plus forte du jeu, fait une attaque de zone (réduit de 1 PV tous les ennemis situés à ±1 ligne et jusqu'à 4 positions devant).
- `'R'` : eduroam, ne fais rien ici, car elle affecte le déplacement des ennemis seulement, elle est donc dans `DéplacementsEnnemis()` que nous verrons plus bas.

Si l'ennemi meurt (pointsDeVie <= 0), on vérifie si c'était un `'S'` (syndicaliste) : dans ce cas, il augmente la vitesse des ennemis adjacents (précédent et suivant sur la même ligne) de 2 avant de disparaître. Puis on supprime l'ennemi de la liste chaînée via `SupprimerEnnemi(jeu, erreur, e) (ennemis.c)`. Cette dernière appelle également la fonction `AjouterAuScore(jeu, ennemi, erreur) (score.c)`, qui permet d'ajouter au score de la partie (qui est dans la structure Jeu) les points rapportés pour l'élimination de cet ennemi. Plus il est robuste, plus il rapporte :

- 'Z' (étudiant) : +25 pts
- 'L' (Étudiant L1) : +100 pts
- 'X' (Étudiant Talent) : + 50 pts
- 'S' (Syndicaliste) : +50 pts
- 'F' (Étudiant Fainéant) : + 75 pts
- 'D' (Doctorant) : +100 pts

Actions des ennemis – `ResoudreActionsEnnemis(jeu, erreur)`
C'est maintenant pour chaque ennemi qu'on vérifie si une tourelle est sur la même ligne et position de la tourelle = position de l'ennemi - 1. Si c'est le cas, l'ennemi attaque la tourelle et effectue plus ou moins de dégâts selon son type.

Cas particulier : `'D'` (Docteur) régénère les ennemis proches : dans un rayon de ±1 ligne et ±3 positions autour de lui, tout ennemi récupère 1 PV (sans dépasser son maximum).

Déplacement des ennemis – `DeplacerEnnemis(jeu, erreur)`

Une fois les dégâts appliqués, on parcourt de nouveau la liste chaînée des étudiants, et on lui applique les déplacements selon plusieurs critères :

- Si il a une **immobilisation** strictement positive, on décrémente de 1 e->immobilisation, et on passe à l'étudiant suivant car celui-ci ne bouge pas à ce tour.
- Sinon, on calcule son **déplacement potentiel.**
    - S'il a été touché par une tourelle de type **'R' (eduroam)**, son déplacement est fixé de façon équiprobable à 1 (il est ralenti si sa vitesse est supérieure à 1), ou 0 (il ne bouge pas), ou -1 (il recule si c'est possible).
    - Pour le type `'F'` (**Fainéant),** son déplacement est déterminé aléatoirement par la fonction `ActionFaineant(jeu, e)`, pouvant aller de 0 à 1 aléatoirement, car il est fainéant, il ne bouge pas toujours…
    - S'il n'est dans aucun de ces cas, alors son déplacement vaut sa vitesse.

Ensuite, on vérifie les collisions entre ennemis (on ne peut pas dépasser l'ennemi devant, ou entrer dans celui de derrière) et avec les tourelles (on ne peut pas empiéter sur la case de la tourelle juste devant). Pour réaliser cela, on regarde la différence entre la position des étudiants ou tourelles devant et derrière, avec la position de l'étudiant en question, auquel on ajoute le déplacement (calculé plus haut). Si cette différence est négative, c'est qu'il y a collision ou traversement. Dans ce cas, on déplace l'ennemi juste devant l'entité qui pose problème.

Enfin, on met à jour la position de l'ennemi : `e->position -= deplacement`.

Vérifications de fin de partie – Nous traiterons cela plus bas. Il s'agit de vérifier si la partie est perdue, gagnée ou mise en pause.

Fin de tour

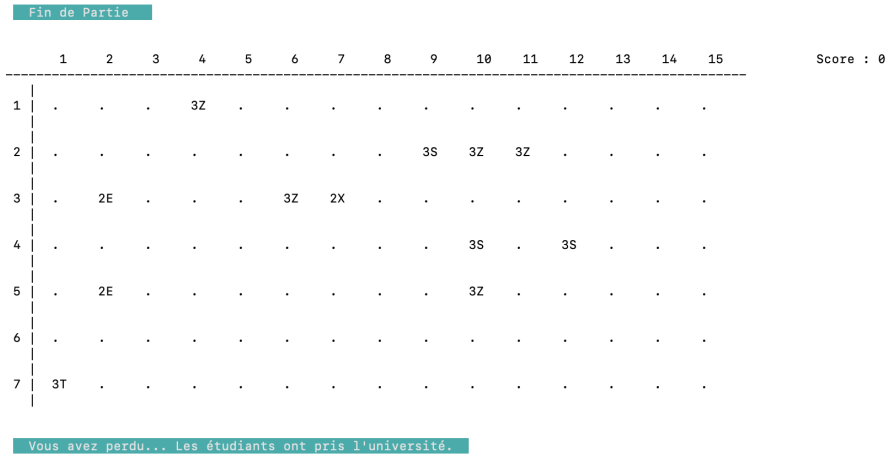
S'il n'y a ni victoire ni défaite à ce tour, on affiche un message de fin de tour. On demande à l'utilisateur d'appuyer sur *Entrée* pour passer au tour suivant, ou de taper 'S' pour sauvegarder la partie. On retourne ensuite au **point 1** de la boucle (incrémentation du tour), et le déroulement se répète.

Une fois la partie terminée ou en cas d'erreur, `LibererJeu(&jeu) (main.c)` est utilisée pour libérer les mémoires allouées dynamiquement (ennemis et tourelles).
Le programme attend alors que le joueur appuie sur Entrée avant de revenir au menu principal, permettant ainsi de démarrer une nouvelle partie, reprendre une partie sauvegardée, etc..

Ce cycle se répète jusqu'à ce que l'utilisateur choisisse de quitter le jeu.

2.4 Conditions d'arrêt de la partie

Vérification de la défaite – `PartiePerdue(jeu)`



Juste après le déplacement, on vérifie si au moins un ennemi est arrivé à la position 0 ou moins, (c'est à dire qu'il a atteint l'université). Si c'est le cas, la partie est perdue. On fixe alors `jeu->score = 0` et on affiche le message de défaite avant de quitter la boucle et de revenir au menu. (Note : en cas de défaite, le score n'est pas pris en compte dans le classement)

Vérification de la victoire – `PartieGagnee(jeu)`
Si tous les ennemis sont morts (la liste chaînée ne contient plus aucun étudiant, la partie est gagnée. On calcule un *bonus de score* pour les ECTS non dépensés en début de partie (`jeu->score += jeu-> cagnotte * 3`), puis on appelle `AjouterAuLeaderboard(jeu, erreur) (score.c)` pour ajouter le pseudo et le score (si il est dans le top 10 des meilleures parties) dans le fichier de classement (leaderboard) du niveau en cours.

Cette fonction ouvre (ou créer s'il n'existe pas) le répertoire "*data_leaderboard*", et cherche s'il existe un fichier texte nommé ainsi : "[Nom du fichier de la partie]*_leaderboard.txt". Pour trouver ce [Nom du fichier de partie], on regarde le chemin qui a permis d'initialiser la partie grâce à jeu->fichier*_ennemis, puis on enlève l'extension ".txt" ainsi que les répertoires (on parcourt la chaîne de caractère depuis la fin à l'aide d'un pointeur et on s'arrête dès qu'on rencontre un "/").

Une fois le fichier texte trouvé, ou créé, on l'ouvre pour lire les scores des joueurs précédents. On lit ligne par ligne MAX_SCORES (afin de ne pas prendre en compte d'éventuels scores superflus), et on les stocke dans un tableau, chaque ligne contenant un pseudo et un score. On ajoute le pseudo et le score de partie à la fin du tableau avant d'effectuer un tri quicksort (`qsort + comparerScores()`). En cas d'égalité, on prend les scores dans l'ordre alphabétique. Enfin, on réécrit le fichier avec les scores actualisés.

Enfin, on affiche le leaderboard de la partie qui vient de se terminer grâce à `AfficherLeaderboard()` et on sort de la boucle de jeu.

Demande de sauvegarde de la partie – `SauvegarderPartie(jeu, err)`

Si au lieu d'appuyer sur "Entrée" pour passer au tour suivant l'utilisateur choisit d'appuyer sur "S" ou "s", alors la partie se sauvegarde grâce à la fonction `SauvegarderPartie()` `(sauvegarde.c)`.

La fonction commence par demander à l'utilisateur d'entrer un nom pour la sauvegarde. (Note : cela permet à l'utilisateur de sauvegarder plusieurs fois le même niveau si il met un nom différent). On utilise `RecupererNom()` pour extraire le Nom du fichier de la partie à partir du chemin contenu dans `jeu->fichier_ennemis` (en retirant les répertoires et l'extension). Puis on ouvre (ou crée si inexistant) le dossier `Sauvegardes`, qui contient tous les fichiers texte sous la forme : `"[Nom du fichier de la partie]_([pseudo])_[Nom choisi par l'utilisateur]_save.txt".` On crée un fichier texte vierge avec le nom que l'on vient de composer.

On ouvre ce fichier texte et on y inscrit les informations de la partie à sauvegarder :

- Le numéro du tour
- La cagnotte actuelle
- Le pseudo du joueur
- Le nombre de tourelles restantes
- Chaque tourelle avec son type, ses points de vie courants, ligne, position et prix. (On parcours la liste chaînée)
- Le nombre d'étudiants restants
- Chaque étudiant avec son type, ses points de vie courants, sa vitesse, sa ligne, sa position, son état d'immobilisation

Une fois toutes les données écrites, le fichier est fermé. Enfin, un message est affiché pour indiquer à l'utilisateur que la partie a bien été sauvegardée, en précisant le chemin du fichier de sauvegarde. Une fois cette fonction terminée, on retourne au menu et la partie s'arrête.

3. Détails des fonctions ‘dynamiques’

Les fonctions dynamiques sont celles qui interagissent avec les données, au contraire des fonctions statiques servant uniquement à l’affichage.
Nous présentons les fonctions dans l’ordre dans lequel nous les avons implémentées, qui va des fonctionnalités basiques aux extras améliorant l’expérience joueur.

Nous utilisons le code couleur suivant pour indiquer les contributions
Contribution faite majoritairement par Romain
Contribution faite majoritairement par Marina
Contribution également partagée par les deux

- Définition des structures + Initialisation des ennemis + structure d’erreur

Tout d’abord, il a fallu créer toutes les structures de bases de jeu, et les fonctions permettant l’initialisation de ces structures.

La structure erreur a été implémentée dès le début. Plus d’informations sur la gestion d’erreur sont dans la partie 3.

Au début notre programme fonctionnait en prenant en entrée un fichier txt avec les vagues d’ennemis. Nous avions supposé que les niveaux pouvaient ainsi être modifiés par l’utilisateur, et il fallait implémenter un système de vérification de conformité de ce fichier txt. La fonction `void ResoudreFichier(FILE* fichier_ennemis, Erreur* erreur)` permettait de trier le fichier avec `qsort`, et de résoudre les éventuels doublons d’ennemis apparaissant au même tour et à la même ligne.

Il fallait donc reléguer les ennemis au tour d’après. Mais dans ce cas, le fichier que l’on venait de trier par tour n’était plus trié. Il aurait fallu reléguer la ligne de cet ennemi jusqu’à rencontrer un ennemi du tour suivant, et effectuer un tri à chaque fois que l’on a changé l’ennemi de tour. Cela complexifiait le programme pour peu.

Nous avons alors choisi de simplifier : nous avons imposé que le fichier soit déjà ordonné (par tour, puis par ligne) et qu’aucun doublon ne soit présent. Ainsi, `ResoudreFichier` se limite à vérifier la validité de chaque entrée (format, bornes d’emplacement) et à contrôler l’ordre (tour non décroissants et, si le tour est le même, lignes strictement croissantes). En cas de problème, la fonction renvoie immédiatement une erreur.

Nous vérifions la première ligne du fichier, la cagnotte, qui doit contenir uniquement un nombre et le caractère `\n`.
Pour initialiser la structure des ennemis, nous parcourons le fichier valide avec fgets et sscanf qui trouve les trois valeurs par ligne :

“Tour” “Numéro de ligne” “Symbole”

Nous avons un chaînage global, et un double chaînage par ligne : pour chaque ligne *l*, on conserve dans *lignes_ennemis[l - 1]* le dernier ennemi déjà inséré. Les lignes vont de 1 à 7.

Le nouvel ennemi sur cette ligne pointe vers le dernier dans prev_line, et celui-ci met à jour son next_line. On met ensuite à jour le dernier ennemi de la ligne.
La position est initialisée à une valeur hors plateau (=NB_EMPLACEMENTS + 99) pour être ajustée au moment où l'ennemi doit apparaître.
Les autres caractéristiques sont récupérées depuis la table constante d'inventaire `TYPES_ENNEMIS`.

Le pointeur final `premier` (vers le premier ennemi de la liste) est retourné au jeu, et c'est dans l'ordre du chaînage global que les ennemis seront parcourus.

Initialement, la structure de Jeu contenait uniquement un pointeur vers le premier ennemi/étudiant et vers la première tourelle. Nous l'avons ajusté plus tard pour gérer le changement de niveau (voir p. 20)

- Initialisation des tourelles et vérification de l'entrée utilisateur

Il a fallu ensuite choisir la façon dont l'utilisateur allait donner l'emplacement des tourelles. Nous avons implémenté une approche avec entrée ligne par ligne, en demandant à l'utilisateur une unique chaîne de caractères de la forme « T 3, T 5, P 6 » pour chaque ligne.

Cela évite de faire plus appels (ligne, position, symbole) et rend l'entrée plus simple à saisir d'un coup.
Nous vérifions d'abord la validité de chaque ligne entrée avec `int VerifEntreeLigne(char * ligne_tourelles, Erreur* erreur)`, puis utilisons cette entrée pour créer de nouvelles tourelles.
La validation est systématique :

- Symbole autorisé
- Emplacement valide sur le plateau
- Et format de l'entrée correct, avec exactement une virgule après chaque couple (symbole, position)
- Cagnotte non dépassée
- Pas de doublons

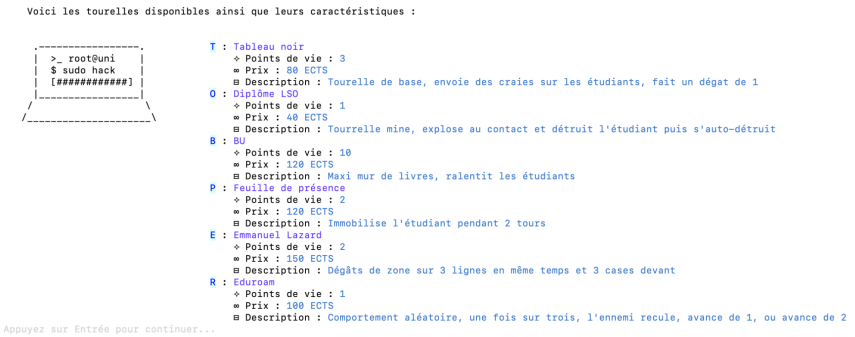
La mise en place de la fonction `VerifEntreeLigne` telle qu'elle n'a pas été immédiate. Plusieurs bugs ont dû être résolus pour avoir une validation robuste à tous les cas d'entrée.

- De multiples virgules ou les espaces superflus causaient une erreur. Par exemple, si l'utilisateur terminait sa ligne par « , » ou ajoutait un espace en trop avant la virgule, le pointeur de lecture avançait incorrectement. Nous avons résolu cela en vérifiant explicitement *ptr == ',' ou *ptr == '\n' et en sautant les espaces.
- Une entrée sans emplacement « T ,» ne passait pas la condition `sscanf(ptr, " %c %d", &symbole, &position)) == 2`, et ne renvoyait pas d'erreur. Il a fallu ajouter la condition à la fin que le nombre de matchs du `sscanf` était égal à 2.

Après une entrée valide, les tourelles sont créées et initialisées avec leurs caractéristiques depuis la table `TYPES_TOURELLES`. Le pointeur vers la première tourelle doit être stocké dans la structure de jeu, et `InitialisationTourelles` retourne `premier`. Une autre variable *`dernier`* a été créée pour ne pas avoir à reparcourir l'ensemble des tourelles lorsqu'on doit ajouter une tourelle au chaînage existant. Donc `Tourelle* AjouterTourelles` doit retourner la nouvelle dernière tourelle.

Or un bug a été rencontré à ce niveau, car la fonction peut chaîner plusieurs tourelles à la fois mais elle modifiait uniquement le pointeur *`dernier`*. Si la première tourelle de la liste était donnée dans une entrée avec une tourelle, *`premier`* correspondait à *`dernier`*, tout était ok. Mais si l'entrée comprenant la première tourelle du jeu était composée de plusieurs tourelles, par exemple (T 1, T 4), alors *`premier`* n'était pas correctement mis à jour, et seulement la dernière tourelle entrée était enregistrée (T 4). Il a fallu donner en argument un **pointeur vers le pointeur** ***`premier`***, pour qu'il soit mis à jour :
`Tourelle* AjouterTourelles(Tourelle* * premier, Tourelle* dernier,..)`

Lors de l'ajout, on n'a pas de vérification de validité à faire, donc on avance simplement le pointeur en fonction des virgules et des sauts de ligne.



Affichage des caractéristiques des tourelles

- Gestion dynamique de la mémoire

La gestion de l'espace mémoire pour les ennemis et les tourelles est similaire, pour chaque instanciation de la structure Étudiant ou Tourelle, nous allouons de l'espace mémoire avec `malloc`. Cela permet de gérer un nombre potentiellement fluctuant d'ennemis ou de tourelles pendant la partie, contrairement à une allocation statique.

Lors de l'initialisation, en cas d'erreur (fichier invalide, symbole non reconnu, manque de mémoire), on libère immédiatement tous les ennemis ou tourelles déjà créés grâce à LibererEnnemis ou LibererTourelles. Ces dernières parcourent les listes chaînées pour libérer chaque nœud.

Après avoir fini les dernières étapes du projet, nous avons vérifié l'absence de fuites mémoire et d'accès invalides avec Valgrind. Grâce à ces fonctions de destructions, il est possible de garantir que chaque allocation est compensée par une libération

- La boucle de partie et les 6 étapes d'un tour de jeu

Nous avons déjà expliqué l'ordre dans lequel différents éléments du jeu se mettent à jour. Il y a un découpage clair entre les actions des tourelles et des ennemis. Cela reproduit la logique classique des jeux tower defense : d'abord le joueur (ou ses tourelles) agit, ensuite l'ennemi réplique, et enfin se déplace. Cette structure limite évite des états intermédiaires indéfinis.
Grâce à cette logique, l'implémentation de toutes les fonctions a été simple.
Pour l'action des ennemis sur les tourelles, nous avons décidé que les dégâts seront faits lorsque l'ennemi se situe juste devant la tourelle, à 1 emplacement de plus. Cela facilite la gestion des collisions et l'affichage, pour ne pas avoir deux objets à la même position

Nous avons aussi introduit SupprimerEnnemi.
Cette fonction se charge de dé-chaîner proprement un ennemi de la liste principale et de la liste par ligne, avant de libérer la mémoire. Il faut vérifier si c'est le premier ou non, car cela indique s'il a un prédécesseur. Il faut parcourir ensuite la liste des étudiants jusqu'à trouver son prédécesseur `while (prec && prec->next != ennemi)`
Cette boucle est obligatoire, une simple vérification de `ennemi->prev_ligne` ne suffit pas, le prédécesseur pourrait être sur une autre ligne. Cela garantit qu'il n'y a pas de fuite et que la structure reste cohérente.

Des spécificités d'action de tourelles et d'ennemis qui changent autre chose que le nombre de points de vie ont été introduites plus tard. (voir p. 19)

Nous avons dû gérer la collision liée à l'apparition des ennemis sur le plateau dans le cas où la case est déjà occupée.
La première façon dont nous avons géré cette vérification de collision a été en reléguant l'apparition de l'ennemi au tour d'après. Mais les ennemis n'étaient plus chaînés dans le bon ordre, et cela modifiait le déplacement et les actions des ennemis, car un ennemi du tour d'après mais d'une ligne en dessous avançait avant cet ennemi relégué.

Nous avons plutôt opté pour un système par "file d'attente" (une file par ligne), géré par la fonction `ApparitionEnnemis()`. Ce mécanisme suit un principe *FIFO* : pour chaque ligne, on recherche le premier ennemi éligible à apparaître (en fonction de son tour) et on vérifie si la position d'entrée (NB_EMPLACEMENTS + 1) est libre. Si oui, l'ennemi apparaît ; sinon, il attend le tour suivant. Le parcours préserve l'ordre d'apparition déterminé initialement et maintient l'intégrité du chaînage `prev_line/next_line/ next.`

- Gestion des collisions lors des déplacements des ennemis (étape 4/6)

Il a fallu ensuite gérer les cas où l'avancée des ennemis interférait avec la position de tourelles et d'ennemis déjà présents.
On calcule le déplacement souhaité avec la vitesse de l'ennemi, et on le réduit dans le cas de collisions.

Il faut donc vérifier la position de l'ennemi devant sur la même ligne, l'ennemi précédant sur la même ligne (dans le cas d'un déplacement négatif causé par l'action d'une tourelle).
Il faut aussi parcourir toute la liste des tourelles et vérifier les collisions.

De toutes les comparaisons, on garde celle qui résulte du plus petit déplacement.

Si l'ennemi ou la tourelle est en collision avec l'emplacement souhaité, on décale l'emplacement souhaité de 1. On continue de parcourir la liste des tourelles même après une collision, car il se peut que plusieurs tourelles soient situées les unes à la suite des autres et bloquent la deuxième position envisagée.

Nous avions implémenté cette gestion de collisions de plusieurs façons, d'abord en comparant à chaque fois la position directement et non pas l'emplacement. Cette méthode ne s'est pas avérée pratique pour gérer le cas de tourelles définissant un déplacement.

Toutes ces implémentations permettaient de gérer déjà plusieurs types d'ennemis qui se distinguent par une vitesse et des dégâts différents.

- Visualisation des vagues d'ennemis avant le placement des tourelles – `void VisualiserEnnemis(Etudiant* etudiants)`

Afin de permettre à l'utilisateur de voir les vagues d'étudiants qui vont l'attaquer, nous avons dû créer le tableau `char waves[NB_LIGNES][maxTour]`. NB_LIGNES est déjà déterminé dans le header (vaut 7), mais il nous a fallu connaître le nombre maximum de tours, MaxTour. Nous devons donc itérer la chaîne d'étudiant pour trouver ce nombre. Une fois ce tableau à créé, nous l'initialisons avec des "." pour donner une impression de cases. Puis, nous réitérons la chaîne d'étudiants pour remplacer les "." par le type de l'étudiant si il est présent à ces coordonnées.
Une fois le tableau waves rempli, nous allons commençons à l'afficher. Nous inscrivons d'abord les numéros de tours.

Ici, nous avons été confrontés à un problème de décalage entre les colonnes du tableau et les numéros, car dès que nous avons 2 chiffres à afficher, l'espace pris augmente. C'est pourquoi nous utilisons ici %2d qui permet de garder l'espace pour 2 chiffres même lorsqu'il n'y en à qu'un. (Note : nous n'avons pas souhaité garder plus que 2 espaces réservés puisqu'en pratique, nous avons rarement plus de 99 vagues d'étudiants.
Ensuite, nous affichons à l'aide de deux boucles imbriquées l'ensemble du tableau waves. (Quelques effets décoratifs sont ajoutés, notamment des couleurs (ANSI, voir header) ou encore des "---" ou " | " afin de donner une impression de tableau.

- Visualisation du plateau – `void AfficherPlateau(Jeu* jeu)`

C’est cette fonction qui affiche le plateau à chaque tour lors d’une partie. Elle est souvent précédée de `printf("\033[0;0H\033[2J");` qui efface l’écran pour donner une sensation de fluidité dans les mouvements des étudiants.
On initialise d’abord le tableau `char plateau[NB_LIGNES][NB_EMPLACEMENTS + 1][4]`, à 3 dimensions. Les deux premières permettent de stocker les positions des entités, et la dernière dimension stocke 4 caractères pour le type (un caractère), les pv (entre 1 et 99), et un \0 de fin de chaîne.
On initialise le tableau à l’aide d’une double boucle : on met `'.'` pour indiquer une case vide. On place un `'\0'` juste après pour que la chaîne soit correctement terminée. Cela permet aux fonctions de manipulation de chaînes (comme `printf` avec `%s`) de savoir où s'arrêter lors de l'affichage ou du traitement de la chaîne. On ajoute une colonne supplémentaire (d’où le NB_EMPLACEMENTS + 1) pour permettre aux étudiants d’arriver sur le plateau sans collision avec les emplacements possibles des tourelles. Cette colonne est marquée par un espace pour indiquer que ce ne sont pas des cases comme les autres, prévues pour le placement des tourelles.

Ensuite, le tableau est rempli en parcourant les listes chaînées des tourelles et des étudiants du jeu. Pour chaque entité, on convertit les positions (ligne et colonne) en indices commençant à 0 pour passer d’un indiçage de 1 à n vers 0 à n-1 et on vérifie qu'elles se trouvent bien dans les limites du plateau. Si l’entité est valide, on utilise `snprintf` pour formater une chaîne (par exemple "3T" pour une tourelle de type ‘T’ à 3 points de vie) dans la case correspondante du plateau.

Affichage final

Le tableau complété avec les informations des entités, on affiche maintenant celui-ci. Un en-tête est affiché indiquant les numéros des colonnes. Chaque numéro est affiché avec une largeur fixe pour conserver l'alignement et pour aérer le tableau (“%-6d”). Le score est affiché à droite de l'en-tête et une ligne de séparation horizontale est affichée pour délimiter l'en-tête du plateau.

Ensuite, chaque ligne du plateau est affichée. On affiche d'abord le numéro de la ligne suivi d'une barre verticale. Pour indiquer le début du champ de bataille. Puis, on parcourt toutes les cases de la ligne (y compris la colonne supplémentaire) en affichant le contenu de chaque case sur une largeur fixe (6 caractères) pour garantir l'alignement. Enfin, une nouvelle ligne de séparation est affichée après chaque ligne du plateau, pour l'aérer.

- Mise en place d’ennemis et tourelles plus avancés

Eduroam, Docteur, Emmanuel, Syndicaliste, immobilisation, random
Maxi flop du Fainéant, changement de chainage du fainéant donc nécésaire d’avoir indicateur s’il s’est déplacé ou pas -> comment réinitialiser l’indicateur.

Pour élargir la variété de gameplay, nous avons ajouté plusieurs types plus “avancés” :

- ★ Tourelles : Eduroam (R), Feuille de présence (P), Emmanuel Lazard (E)
- ★ Ennemis : Syndicaliste (S), Fainéant (F), Doctorant (D)

De l’aléatoire a été introduit, avec la tourelle R et l’étudiant F.

`srand((unsigned int)(uintptr_t)&erreur)` est un moyen d’initialiser la graine du générateur pseudo-aléatoire. Il suffit de le faire une fois dans tout le programme pour initialiser la graine de génération. Il était possible de le faire avec `srand(time(NULL))`, mais nous avons préféré ne pas introduire la bibliothèque `time.h` en plus.

Les actions de R et de P sont sur le déplacement de l’ennemi, il a fallu ajouter un parcours de la liste des tourelles pour vérifier si une tourelle R allait tirer sur l’ennemi.
Il faut qu’il n’y ait pas d’ennemi précédant, car la tourelle tire uniquement sur le premier ennemi qu’elle rencontre, on a donc la vérification :
`(t1->ligne == e->ligne && t1->position < e->position && e->prev_line == NULL)`
Pour P, Feuille de Présence qui immobilise 1 fois sur 3, un attribut immobilisation a été nécessaire. Lorsqu’on parcourt les ennemis pour les déplacer, on vérifie maintenant au début de la boucle si `e->immobilisation > 0`, auquel cas on passe à l’ennemi suivant.

Un qui a pris beaucoup de temps par rapport à l’intérêt qu’elle représentait a été la mise en place de l’ennemi Fainéant. Initialement, un changement de ligne une fois sur six été prévu. Mais cela a posé des difficultés avec le chaînage par ligne, dans le cas où la position prévue sur la ligne d’en dessous était occupée. Il fallait parcourir les ennemis jusqu’à trouver une position libre. Or si toutes les positions jusqu’à la fin étaient occupées, cela posait problème, l’ennemi étant relégué au bout du plateau + 1, et ne s’affichant plus.

La fonction ChangerLigne se trouve encore dans helpers.c, nous l’avons gardé dans le code pour des raisons de documentation du travail effectué, mais elle n’est pas utilisée.

Nous avons donc simplifié le comportement de Fainéant.
`int ActionFaineant(Jeu* jeu, Etudiant* e)` effectue l’action du Fainéant, basée sur un choix aléatoire entre 1 et 6. 2 fois sur 3, l’ennemi ne bouge pas, 1 fois sur 6 il se place derrière son prédécesseur sur la même ligne, 1 fois sur 6 il saute sur la ligne du dessous, uniquement

Le dernier bug rencontré dans le développement du jeu a été sur la tourelle ‘E’ Emmanuel Lazard. Elle n’attaquait pas les ennemis qui l’entouraient si aucun ennemi n’était présent sur sa ligne. Cela était dû au fait que nous cherchions d’abord l’ennemi à attaquer sur la même ligne, et ensuite faisions une distinction de l’action en fonction du type de la tourelle.

- Création d'un Menu de début de partie

Menu principal – `char* Menu(Erreur *err)`
C'est cette fonction qui gère l'interface de sélection au début du jeu (ou à chaque retour au menu). Elle propose 5 options qui sont stockées dans un tableau `char menu_options[5][MAX_NAME_LEN]`. Max Name Len est définie dans header.h, elle vaut 256 et permet de contrôler la taille maximale des chaînes de caractères.

Chaque élément du tableau est une chaîne de caractères que l'on affiche à l'utilisateur via la fonction `AfficherChoix`. Cette fonction se charge d'afficher les différents choix et d'écouter la saisie de l'utilisateur pour savoir quelle option a été sélectionnée.

L'utilisateur entre un nombre qui correspond à l'option désirée. Par exemple, saisir « 1 » permet de « Jouer les niveaux », etc.

Variable `chemin`
La fonction renvoie généralement un pointeur vers une chaîne de caractères (un *chemin*) vers un fichier de niveau ou de sauvegarde. Pour ce faire, on déclare une variable `char chemin[MAX_NAME_LEN]`.

Si l'utilisateur sélectionne l'option « Classements » ou bien s'il annule/retourne au menu, la fonction renvoie `NULL`, signifiant qu'il n'y a pas de fichier à charger.

Sinon, on construit la chaîne de chemin en fonction du choix fait par l'utilisateur (par exemple « Niveau/NomFichier.txt » ou « Sauvegardes/NomFichier_save.txt »).

Détails par option du menu

- Jouer les niveaux

On lit la liste des fichiers présents dans le dossier `Niveau` (fonctions `LectureNoms` et `FormatterNoms`) pour constituer des options lisibles. Chaque nom correspond à un niveau potentiel. L'utilisateur choisit un niveau grâce à `AfficherChoix` (ex. saisir « 2 » pour le deuxième niveau). On construit le chemin correspondant sous la forme « Niveau/… » et on le renvoie.

- Charger une partie (saisie manuelle)

Ici, l'utilisateur doit entrer le chemin complet du fichier à charger (par exemple « Dossier/2_Talents_De_Sprinteur.txt »). On lit la chaîne au clavier (via `fgets`), on retire le `\n` final et on vérifie qu'elle n'est pas vide. Si tout est correct, on la copie dans `chemin` et on retourne ce dernier pour l'utiliser plus tard lors du chargement de la partie.

- Reprendre une partie sauvegardée

Comme pour les niveaux, on lit le contenu du dossier `Sauvegardes`. Chaque fichier correspond à une partie précédemment enregistrée (ex. « 4_Etudiants_A_La_BU_save.txt »).

Pour plus de clarté, la fonction supprime la mention « save » lors de l'affichage. On ajoute également une dernière option « Retour » dans le menu, pour permettre à l'utilisateur de revenir en arrière sans charger de sauvegarde. Si l'utilisateur choisit une partie existante, on construit la chaîne « Sauvegardes/NomDeFichier » et on la renvoie.

- Classements

Aucun chemin n'est renvoyé. La fonction appelle simplement `ChoixLeaderboard` (une autre fonction gérant l'affichage et la navigation dans les classements). On retourne `NULL` pour indiquer que l'on ne charge pas de niveau.

- Quitter

On sort immédiatement du programme en appelant `exit(0)`. Il n'y a aucun problème

- **Gestion du score – Fichier score.c**

Afin d'enregistrer et de classer les performances des joueurs, la gestion du score est répartie dans plusieurs fonctions. Dès qu'un étudiant est vaincu, la fonction `SupprimerEnnemi` appelle `AjouterAuScore` afin d'attribuer un nombre de points dépendant du type d'ennemi éliminé.

Lorsque la partie est gagnée, la fonction `AjouterAuLeaderboard` lit le classement existant dans un fichier texte (dans le dossier `data_leaderboard`). Pour se faire, on récupère le nom de fichier de la partie avec jeu->fichier_ennemis, puis on le formatte en enlevant les répertoire et l'extension .txt et tout ce qu'il y a avant le premier "/" en partant de la droite.

Ensuite on créer le chemin d'accès vers le leaderboard en ajoutant le préfixe "data_leaderboard/" avant le nom extrait, puis le suffixe "_leaderboard".

`snprintf` est une fonction qui permet de formater une chaîne de caractères et de l'écrire dans un buffer (ici, `cheminLeaderboard`). Elle évite les débordements de tampon en spécifiant la taille maximale du buffer. Détaillons son utilisation :

- **(`cheminLeaderboard`)** : C'est le buffer dans lequel la chaîne formatée sera écrite.
- **(`sizeof(cheminLeaderboard)`)** : indique la taille totale du buffer. Ici, `sizeof(cheminLeaderboard)` renvoie la taille en octets de ce tableau, assurant ainsi que l'écriture ne dépassera pas ce nombre de caractères.
- `"data_leaderboard/%s_leaderboard.txt"` est la chaîne qui sert de modèle pour construire le chemin du fichier. Elle contient un `%s` qui sera remplacé par la chaîne contenue dans `nomBase`.

Cette construction permet de stocker les résultats de plusieurs parties dans le dossier data_leaderboard, et de les différencier des autres fichier texte avec le suffixe "_leaderboard".

Une fois le bon fichier texte ouvert, on lit ligne par ligne au plus MAX_SCORES (fixé à 10), contenu dans la variable int capacite. On récupère les pseudo des joueurs et les scores associés, et on y ajoute le pseudo et le score du joueur qui viens de terminer sa partie. Afin de garantir un ordre cohérent, la fonction `comparerScores` est employée pour trier les scores : elle compare d'abord la valeur numérique, puis, en cas d'égalité, s'appuie sur l'ordre alphabétique du pseudo.

`AfficherLeaderboard`, qui présente la liste des scores de manière structurée, est appelée soit depuis ChoixLeaderboard, fonction du menu permettant de consulter les scores des parties dans le dossier data_leaderboard, soit à la fin d'une partie gagnée.

Le choix d'implémenter la logique de score dans des fonctions séparées, de la mise à jour immédiate du total au tri final, s'explique par un souci de maintenance et de clarté. Chaque partie du code gère une étape précise : `AjouterAuScore` se concentre sur l'accumulation de points lors du combat, `AjouterAuLeaderboard` agrège et trie les résultats pour une perspective plus globale, et `AfficherLeaderboard` se charge uniquement de la mise en forme visuelle. Cette structure modulaire rend le code plus facile à modifier ou à étendre : si l'on souhaite, par exemple, introduire de nouveaux types d'ennemis.

- Améliorations Visuelles

Après avoir découvert la possibilité d'ajouter des couleurs grâce à des séquences de clavier spéciales, les séquences *ANSI*, nous avons définit une table de macros de couleurs dans le `header`.
Nous avons ajouté un ordinateur en ASCII art à côté des explications sur les types de tourelle, qui est aligné verticalement. Cet alignement est possible grâce à un printf d'une chaîne de caractère d'une longueur fixe, `printf("%-35s", ordinateur[l])` remplit la chaîne avec des espaces pour arriver à 35 caractères.
Nous avons ajouté des éléments tels des barres de chargement, un affichage progressif de texte, et un titre en ASCII art, obtenu sur www.patorjk.com/software/taag/

- **Gestion de la sauvegarde – Fichier `sauvegarde.c`**

La fonction **SauvegarderPartie** demande d'abord à l'utilisateur de saisir un nom de sauvegarde. Cela permet au joueur d'avoir plusieurs sauvegardes pour un même niveau. Elle crée ou ouvre ensuite le dossier "Sauvegardes" et construit le chemin complet du fichier de sauvegarde grâce à *snprintf* vu précédemment . Ce chemin intègre le nom de la partie, le pseudo du joueur et le nom de sauvegarde :
`Sauvegardes/[Nom_fichier_partie]_([Pseudo])_%[Texte entré par l'utilisateur]_save.txt`

Pour enregistrer l'état du jeu, le code utilise des boucles pour parcourir les listes chaînées des tourelles et des étudiants et écrit chaque information ligne par ligne dans le fichier avec *fprintf*. (voir 2.4)

Le système de sauvegarde et de relance est conçu de manière modulaire pour séparer clairement chaque étape. La fonction **ExtraireNomFichierSauvegarde** utilise des fonctions de manipulation de chaînes comme *strrchr* et *strchr* pour parcourir le chemin du fichier. D'abord, *strrchr* permet de trouver le dernier '/' et d'isoler le nom complet du fichier. Ensuite, *strchr* cherche le premier '_' pour séparer la partie utile (le "NomFichier") des informations supplémentaires. Enfin, *strncpy* copie cette portion dans un tableau en s'assurant que la chaîne est bien terminée par un '\0'. Cette approche permet de créer facilement des sauvegardes sauvegardes le plus unique possible par leurs noms.

La fonction **RelancerParti**. Elle utilise *fscanf* pour récupérer les données comme le tour, la cagnotte, le score et le pseudo, en vérifiant à chaque étape que le format est correct. Nous récupérons le pseudo car nous avons fait le choix qu'on ne puisse pas "voler" la partie d'un autre joueur qui a été mise en pause en y mettant notre pseudo. Des boucles permettent ensuite de reconstruire les listes des tourelles et des étudiants en allouant la mémoire nécessaire. Une fois l'état du jeu reconstitué, la fonction appelle à nouveau **ExtraireNomFichierSauvegarde** pour mettre à jour le champ correspondant dans la structure du jeu. Cela permet d'éviter la création de doublons dans le leaderboard lors de la relance de la partie. Pour terminer, le fichier de sauvegarde est supprimé avec *remove*, évitant ainsi l'accumulation de sauvegardes inutiles.

- Création de niveaux supplémentaires et tests

Pour vérifier que notre fonction de vérification d'entrée utilisateur servant à l'initialisation des tourelles était robuste, nous avons créé une fonction `test_VerifEntreeLigne` dans le dossier test. Elle vérifie tous les formats d'entrée possibles.

4. Système de gestion d'erreur

Nous proposons une gestion et propagation d'erreur simplifiée grâce à une unique structure stockant le statut et le message d'erreur. Un statut à 0 signifie qu'aucune erreur n'est rencontrée. Si une erreur a lieu à un niveau, le statut est mis à 1. Ainsi, la vérification d'erreur est simplifiée, puisqu'il suffit de faire une vérification de condition unie dans les niveaux supérieurs.

Prenons l'exemple de l'initialisation des ennemis à partir du fichier txt décrivant l'apparition d'un type à un tour, à une ligne donnée.
Chaque ligne est analysée une par une, et la conformité du symbole est vérifiée.

```
const TypeEnnemi* type = VerifType(&tour, &num_ligne, &symbole_type, erreur);
```

Dans VerifType, si un symbole non conforme est détecté, il ne correspond à aucun type, le statut d'erreur est mis à jour, ainsi qu'un message d'erreur :

```
    if (type_ennemi == NULL) {
        erreur->statut_erreur = 1;
        strcpy(erreur->msg_erreur, "type d'ennemi invalide\n");
        return NULL;
    }
```

Ensuite, l'initialisation des ennemis est interrompue, on supprime les cases mémoires allouées pour les ennemis précédents, et on return. Après l'appel de `Initialisation Ennemi`, appelé depuis `PreparerPartie`, si une erreur est détectée via un statut à 1, on return directement, et on revient au programme main principal. Il y a une vérification du statut de l'erreur, et c'est là qu'on affiche la cause initiale.

5. Améliorations possibles

1. Simplifier la gestion des chemins de fichiers

Au fur et à mesure, différentes fonctions de traitement de chemins ont été créées pour obtenir un nom de fichier à partir d'un chemin. Par exemple, `ExtraireNomFichierSauvegarde` et `RecupererNom`.

Ou pour au contraire construire un chemin complet en partant d'un dossier et d'un nom de fichier. Ces fonctions se recoupent en grande partie et certaines étapes sont répétées (comme rechercher le dernier caractère `'/'`, enlever l'extension `.txt`, ou ajouter un suffixe comme `"_save"`). Il serait judicieux de **centraliser cette logique** dans une unique fonction qui prend en entrée un chemin complet et renvoie de manière fiable le nom de fichier sans répertoire ni extension. Cela éviterait la duplication de code et rendrait l'ensemble plus lisible. Une seconde fonction pourrait alors faire l'opération inverse : recevoir un dossier (e.g. `"Sauvegardes"`) et un nom de fichier (e.g. `"NomFichier"`) pour construire un chemin complet (i.e. `"Sauvegardes/NomFichier.txt"`). Cette refonte permettrait d'unifier le comportement, de réduire le risque de bugs liés à la manipulation de chaînes, et de simplifier la maintenance du code sur le long terme.

2. Améliorer le positionnement des tourelles

Actuellement, le placement des tourelles repose sur une saisie rapide et fixe, sans possibilité de prévisualiser la position d'une tourelle avant de la valider définitivement. Pour rendre l'expérience plus fluide, on pourrait introduire **un affichage dynamique** qui montre en temps réel l'emplacement sélectionné. Puis il donnerait à l'utilisateur la possibilité de revenir sur la ligne précédente pour modifier ou retirer une tourelle mal placée. Cela éviterait de devoir relancer entièrement le niveau en cas d'erreur. De plus, on pourrait autoriser le joueur à **poser des tourelles pendant la partie**, à condition de dépenser les *ECTS* gagnés lorsque

des étudiants sont éliminés, ce qui apporterait une **dimension stratégique supplémentaire**. Le joueur pourrait alors choisir de renforcer ses défenses en cours de jeu, voire vendre certaines tourelles pour récupérer une partie de leurs coûts. Cette amélioration demanderait bien sûr quelques modifications dans la boucle de partie (afin de gérer les actions du joueur à chaque tour), mais ouvrirait la voie à un gameplay plus riche et interactif.