Rapport de Projet de C

Weng Jonathan, De Saint Léger Térence

Contrôles : 🎮

- **Échap** pour quitter le jeu
- **Barre espace** pour lancer la vague
- **Clique droit de la souris** pour déplacer la caméra
- **Clique gauche de la souris** pour construire des tours
- **Molette de la souris** pour zoomer/dézoomer
- **ZQSD/WASD [+Shift] [+Ctrl]** pour déplacer la caméra plus ou moins vite
- **F11** pour mettre le jeu en plein écran (fenêtré sans bordure)

Attention, le jeu vous demandera des informations dans le terminal avant de se lancer.

Merci de ne pas entrer de caractères spéciaux dans votre pseudo.

Spécificités du projet :

Tout d'abord, nous avons décidé de nous séparer du style proposé par l'énoncé pour nous tourner plus vers un style médiéval fantastique. Un univers de magie et de monstres divers et variés. Nous avons pour cela, dessiné chaque sprite (que ce soit de tour, de monstre ou de décors) à la main sur l'application *paint.net* (absolument aucun outils d'IA n'a été utilisé pour ce projet, que ça soit sur le plan technique ou esthétique), bien que cela nous ait pris plus de temps, nous souhaitions donner a notre jeu une identité propre tout en gardant un aspect familier des jeux de *tower defence* basiques (les couleurs

ne varient pas trop, nous avons choisi un style plutôt simple et épuré afin de faciliter et accélérer le processus de création).

Choix pour le projet et problèmes rencontrés

Décisions graphiques 🎨

Nous avons décidé de nous tourner vers la bibliothèque SDL2 pour les rendus graphiques de notre jeu, de plus, nous avons directement commencé avec une interface graphique, pensant qu'il serait plus facile de manipuler et d'avancer sur le projet a partir d'une interface graphique bien organisée et établie plutôt que de transposer une interface en ASCII vers une interface graphique complète. Cela nous revenait à repartir de zéro quant aux systèmes d'input/output et nous limitait trop dans nos choix de

conception (ex : il n'est pas possible de faire du mouvement avec des caractères ASCII). De plus, nous n'étions pas attiré par le fait de rendre des images sur des symboles ASCII.

Un autre points nous ayant poussé à nous tourner vers SDL2 en particulier est qu'en plus d'être proposé par le sujet, nous avons déjà de l'expérience avec le moteur *Pygame* (python) qui est basé sur SDL. Étant donné que nous apprécions tous les deux cette librairie, il nous a semblé logique de nous tourner vers SDL2 pour ce projet.

Bien que notre projet soit principalement basé sur la SDL2 , lors de la vérification des fuites de mémoires, nous avons remarqué que certaines semblaient impossible a réparer, après quelques recherches, nous nous sommes rendus compte que la SDL avait en elle même des fuites de mémoires qu'il nous n'était pas possible de corriger même par le biais de fonctions tels que `SDL_FreeSurface()` et autres du même genre.

Nous avions aussi décidé d'opter pour un système d'animation entièrement basé sur le déplacement (et étirement) de la zone d'affichage des sprites (au lieu de réellement les animer).

N'étant pas graphistes, il nous est déjà compliqué de créer 1 seul sprite pour une entité. Les animer représenterait une perte de temps beaucoup trop importante en vue de la durée du projet.

L'avantage de notre choix est qu'il est assez simple à implémenter et fonctionne pour tout. Par exemple aucun ennemi de notre jeu ne bouge vraiment. Ils sont simplement téléportés à destinations, mais avec notre système d'animation leur sprite semble faire le chemin en faisant de petits bonds. Cela permet donc de bien séparer l'aspect technique du jeu (savoir sur quelle case sont les ennemis) de l'aspect esthétique (savoir où représenter les ennemis).

Décisions techniques 🔧

Pour les structures de notre projet, nous avons décidé d'ajouter certains attributs dans celles demandées par l'énoncé tout en gardant la même

racine et d'en rajouter de nouvelles telles que `waves` , `Animation` , `TextElement` , afin d'ajouter plus de détails et de créativité à notre projet.

Pour des raisons pratique, toutes les entrées textuelles (écrire du texte) se font dans le terminal. En effet, gérer des entrées de texte complètement graphiquement (et donc à implémenter nous même) s'avère beaucoup trop long en vu du temps qui nous est accordé. C'est ce qui nous a pousser à en partie utiliser le terminal.

L'utilisateur doit en effet entrer son pseudo dans le terminal puis choisir quel niveau il souhaite lancer. Il peut aussi choisir de lancer le mode infini ou de charger sa dernière sauvegarde (s'il en a une).

Le choix des niveaux est automatiquement mis à jour en fonction des fichiers niveaux disponibles.

Nous avons aussi essayé le plus possible de concaténer les fonctions le permettant, tels que `GetEnemyAndTowerAt` servant a récupérer l'objet se trouvant sur la tuile passé en argument, ou `drawEnemiesAndTowers` . En revanche, nous avons décidé de nous limiter dans la mesure du possible vis a vis de fonctions natives de bibliothèques importées et à la place de cela, les implémenter nous même bien que cela rallongea le code. D'un point de vue construction pour notre `main` , nous avons essayé de le raccourcir le plus possible en nous focalisant sur la construction de fonctions annexes.

L'entièreté du code est écrit en anglais dans une optique d'uniformisation de la langue.

Nous avons aussi décidé de ne pas fixer le joueur à un emplacement et de lui permettre de se déplacer autour de la carte , cet aspect plus complexe de gestion de l'espace, ajoutant une dimension de coordonnées relatives à caméra, rajoutés aux coordonnées fixes (utilisés par exemple pour les menus) nous ont permis une plus grande liberté dans la personnalisation du jeu.

Toutes nos fonctions ont linéaires ou constante sauf pour la fonction permettant de ranger les scores, se servant d'un algorithme de tri par

insertion, celui ci a une complexité plus élevé que le reste. Chaque fonction est commentée afin de faciliter la compréhension du développeur.

Mécaniques de jeu ⚙️

Système de vagues 🌊

Nous avons repris l'idée original et avons pensé qu'il était intéressant d'avoir plusieurs vagues d'ennemis de suite au lieu de seulement 1 seule.

Notre système de vagues revient à faire jouer plusieurs niveaux de suites au joueur, à la différence près qu'il garde ses tours entre chaque vagues et gagne un peu plus d'argent en plus de ce qu'il lui restait de la vague précédente.

Nous utilisons une structure `Wave` contenant la liste d'ennemis à faire apparaitre durant la vague ainsi que la quantité d'argent à remettre à l'utilisateur avant que celle-ci commence (afin qu'il puisse construire ses défenses).

Lorsque la vague précédente se termine dû à l'élimination de tous les ennemis, la vague suivante est chargé à la place, jusqu'à arriver à la dernière vague.

Après avoir vaincu la dernière vague, un message de victoire s'affiche et les meilleurs scores sur ce niveau sont affiché.

Système de prévisualisation des vagues 🔎

Avant chaque vague, des tuiles supplémentaires (d'une couleur différente des tuiles de base) apparaissent à droite de l'écran et sur elles sont dessinés les ennemis qui apparaitront durant la vague. Il est possible que vous ayez à déplacer la caméra pour voir l'entièreté de la vague d'ennemis.

Durant la vague, cette prévisualisation n'est pas affiché.

Les ennemis utilisés pour la prévisualisation sont les même que ceux qui tenteront d'envahir le château (objectif à protéger).

Les phases de tour ⏰

Contrairement aux versions avec représentation ASCII, une frame n'équivaut pas à un tour de jeu complet ([...] → les tours attaquent → les ennemis attaquent/se déplacent/apparaissent → les tours attaquent → [...]).

Il nous a donc fallut un système pour déterminer quelle action devrait être poursuivie.

Les phases de tours répondent justement à ce problème. Elles permettent de savoir si la vague à commencée, si le joueur à gagné/perdu ou encore quand faire une sauvegarde de la partie (faites en fin de tour et avant le début des vagues).

Menu d'achats et améliorations 💰

Menu permettant de construire/détruire/améliorer ses tours.

Il suffi de sélectionner une case et de cliquer sur les boutons correspondants.

Nous avons décidé d'empêcher l'achat de tour lors de l'attaque des monstres pour mettre en valeur l'aspect stratégique et décisionnel de notre jeu, en effet, lancer une vague sans optimiser sa défense mènera a l'inévitable défaite.

Score et Économie

Chaque ennemi rapporte un nombre de points plus ou moins élevé selon la difficulté de son élimination, les meilleurs joueurs sont répertoriés dans un dossier et sont affichés a chaque fin de partie.

Capacités spéciales des tours et ennemis 💥

La plupart des tours et ennemis ont des capacités originales afin de diversifier au maximum le gameplay.

Parmi les tours, on retrouve (liste non exhaustive) :

- **Canon amélioré :** tire 1 fois tous les 3 tours un boulet explosif faisant des dégâts de zone
- **Mage amélioré :** en plus de ralentir les ennemis, 3 projectiles d'un coup sur sa ligne et celles adjacentes
- **Mur amélioré :** fait apparaitre des soldats tous les 5 tours
- **Soldats :** Portée très limité mais peut attaquer les lignes voisines

Parmi les ennemis, on retrouve (liste non exhaustive) :

- **Gellé (ou gros slime) :** se divise en 2 petits slimes sur les lignes adjacentes à sa mort
- **Gobelin :** change de ligne à chaque fois qu'il est blessé
- **Nécromancien :** fait apparaitre des squelettes autour de lui
- Sorcière : augmente la vitesse et soigne les ennemies autour d'elle

Sauvegarde automatique 💾

Sauvegarde automatique en début de chaque tour, permettant de reprendre sa progression n'importe quand.

La sauvegarde automatique peut chargé au démarrage (si l'utilisateur possède une sauvegarde à son nom).

Mode infini ∞

Vagues infinies d'ennemis, chaque vague étant plus dure que la précédente, jusqu'à l'inévitable défaite.

Le joueur commence avec des fonds prédéfinit et doit survivre un maximum de vague afin de gagner un maximum de points (accordé à l'élimination des ennemis, selon leur type).

Impossibilité de retoucher à ses tours lorsque la partie est lancée.

Ce mode utilise un système de niveau de difficulté déterminant la probabilité que des ennemis plus puissants apparaissent ainsi que le nombre d'ennemis apparaissant et leur densité d'apparition.

Problèmes rencontrés et implémentations supplémentaires/futures possibles

Système d'entrée textuelle via l'interface graphique ⌨️

Tout d'abord, lors de l'implémentation de polices d'écritures, nous nous étions penchés dans un premier temps, sur la bibliothèque SDL_TTF, celle-ci étant relié a la SDL que nous utilisions déjà, cela nous paraissait un choix logique d'implémentation. Cependant cette bibliothèque nous posa de nombreux soucis, notre binôme n'étant pas sur les mêmes systèmes d'exploitations, nous nous efforcions de créer un programme fonctionnant à la fois sur Linux et sur Windows, cependant avec la bibliothèque SDL_TTF nous avions tour a tour des problèmes lors de compilation. Ces nombreux soucis n'ont ont amené a implémenter notre propre méthodes d'affichage de polices que nous avons abordés précédemment.

Nous souhaitions implémenter un système de saisie de pseudonyme directement intégré au jeu, afin de ne pas avoir besoin de passer par le terminal du lancement du jeu a la fermeture de celui ci, cependant, il semblait impossible de l'implémenter dans le temps qui nous était imparti. En effet, n'utilisant pas de bibliothèque pour les polices, il aurait fallut créer

tout un système afin d'afficher les caractères (a l'aide de fonctions tels `drawTextSurface`) que l'utilisateur tape afin qu'il puisse voir ce qu'il écrit et se corriger, de plus il aurait aussi fallut vérifier chaque pression de touche de son clavier dans le `switch(event.type)` `case (KEYDOWN)` , car en effet il n'existe pas de fonction semblable à un `scanf`. Cela aurait fait un switch d'au moins 27 case différents pour concaténer le pseudonyme de l'utilisateur dans une chaine de caractères.

Déplacement vertical des ennemis 🏃‍♂️💨

Dû aux nombreux pointeurs à gérer, nous avons rencontré un grand nombre de problème lié au déplacement vertical des ennemis. En effet, à cause d'erreurs souvent assez bêtes, nous avons perdu beaucoup de temps à essayer de trouver la raison de bugs assez étrange (ex : un ennemi qui attaque dans le vite, attend 1 tour sans bougé mais reprend normalement juste après, etc...).

Ce qui nous a beaucoup compliqué la tâche est que ces bugs étaient très rare, mais provoquait le plus souvent un crash du jeu après quelques temps.

Le point commun à tout ces bugs est qu'ils étaient dû à un problème de gestion des pointeurs `prev_line` et `next_line` qui étaient mal actualisés lors de l'apparition et du déplacement des ennemis.

Compilation cross plateforme 💻→🖥️

Lors de ce projet, nous avons appris à nos dépends qu'il n'est pas possible (facilement) de rendre le projet non seulement compilable sous Linux mais aussi sous Windows ou encore MacOS.

En effet, nous avions chacun un OS différent, ce faisait de la portabilité du projet une question clé.

Initialement, nous voulions faire en sorte d'inclure toutes les bibliothèques nécessaires avec notre projet. Malheureusement, nous n'avons réalisé que très tard que les OS tels que Linux n'utilisent pas de librairies statiques,

contrairement à d'autres OS comme Windows où ces librairies sont indispensables.

Bien que nous ayons perdu beaucoup de temps avec ce problème, cela nous a permis de nous renseigner sur les outils utilisés par chacun des nos OS pour compiler et exécuter du code C. Cela nous a aussi amené à complètement restructurer notre projet de manière plus conventionnelle.

Ordonner les actions des ennemis et des tours

Un soucis que nous avons rencontré est le fait d'ordonner les actions des tours et des ennemis afin d'avoir un rendu visuel plus agréable et moins brouillon.

Pour ce faire, nous avons eu recours à un système complexe de conditions visant à détecter (avant toutes actions) si toutes les entités de notre jeu avait bien finies leur action. Cela se fait principalement par la vérification du type d'animation actuellement jouée.

En effet, si un ennemi n'est pas dans l'animation de base (animation d'attente), cela peut soit signifier qu'il attaque, reçoit des dégâts ou est en train de se déplacer. Dans tous les cas, il n'a pas fini son action et donc il faut l'attendre. De même pour les tours à l'exception près qu'au lieu de vérifier si elles bougent (ce qui n'est presque jamais le cas), on regarde s'il y a actuellement des projectiles actifs (et donc qu'une tour vient de tirer).

Système d'amélioration de tours

Pour l'amélioration de tour nous utilisons la fonction `upgradeTower(Tower **tower_list)` qui va `destroyTower` celle actuelle pour essayer d'acheter son amélioration et qui si n'y arrive pas par manque de fond va reconstruire celle de base. Cependant ce fonctionnement n'est pas vraiment optimisé, bien que fonctionnel, pour l'améliorer nous pourrions redéfinir les attributs de la tourelle actuelle par les caractéristiques de sa version améliorée mais ceci améliorerait la vitesse d'exécution du programme au détriment d'une longueur de fonction bien plus conséquente