Le Mans Université

Institut d'Informatique Claude Chappe
Le Mans Université

Le Mans Université
Licence Informatique *2ème année*
Module 174UP02 Rapport de Projet
Battle of Time

Victor Poirier
Vincent Proudy
Louison Roquain
Thomas Wolter

24 avril 2024

Lien GitHub

Table des matières

1 Introduction

Dans le cadre de notre deuxième année de licence en informatique à l'université du Mans, nous avons entrepris la conception et la réalisation d'un jeu vidéo, sur la période de janvier à avril 2024. Ce projet nous a permis d'appliquer les connaissances acquises dans divers domaines, lors de notre cursus, telles que la programmation et la gestion de projet. Le jeu a été développé en langage C avec la librairie SDL2.

Nous avons choisi, pour ce projet, de nous orienter vers un jeu *Tower Defense* dans le style du jeu **'Age of War'**. C'est-à-dire un jeu se jouant à deux personnes (un joueur contre un joueur) dont le but principal est de détruire la base ennemie et défendre sa propre base en achetant des troupes.

Ce rapport présente toutes les étapes de développement de ce projet. Nous commencerons par présenter le contexte et les motivations qui ont conduit à la création de ce jeu, puis nous décrirons en détail la conception, le développement et les tests. Nous mettrons en lumière les défis rencontrés et les solutions que nous avons mises en œuvre pour les surmonter. Enfin, nous évaluerons les résultats obtenus, discuterons des leçons apprises et des perspectives futures pour le projet.

FIGURE 1 – Page d'accueil

2 Organisation

Concernant la répartition des tâches de travail nous avons utilisé un tableau de *Gantt* pour se répartir efficacement la charge de travail.

NUMÉRO	TITRE DE LA TÂCHE
1	Analyse et Conception
1.01	Choix du projet
1.02	Description du projet
1.03	Choix des graphismes du jeu (2D) / Design
1.04	Définitions des fonctionnalités du jeu
1.05	Choix des modes de jeu
1.06	Création et configuration GitHub
1.07	README
1.08	Conception du programme
2	Développement général
2.01	Affichage du menu
2.02	Gestion menu
2.03	Design du jeu
2.04	Structure bâtiment + Gestion des générations (différent âge)
2.05	Structure + Fonction personnage
2.06	Fonction du joueur + Structure
2.07	Fonction de l'ordinateur + Structure
2.08	Fonction + Structure ulti
2.09	Gestion combat + EXP + Gold
2.1	Gestion joueur
2.11	Gestion des troupes (bot)
2.12	Gestion bouton cliquable
2.13	Gestion collision
2.14	Affichage background + Obj statique
2.15	Affichage sprite
2.16	HUD
2.17	Spawn des entités
2.18	Mode de jeu réseau
2.19	Chargement partie depuis fichier / Sauvegarde
2.2	Fin de partie
3	Tests et Préparation soutenance
3.01	Rendre Projet (Git)
3.02	Documentation avec Doxygen
3.03	Rendre Rapport de Projet
3.04	Préparation Soutenance

FIGURE 2 – Extrait de notre *Gantt*

Pour communiquer en dehors de l'université nous avons utilisé *Discord*, un logiciel de communication en ligne pour discuter par texte, audio et vidéo. Nous avons également pu partager des versions de test sans forcément les mettre sur notre Git.

3 Concept

3.1 Règles du jeu

Dans Battle of Time, le joueur retrouvera deux modes de jeu. Le mode solo (contre l'ordinateur), et le mode en ligne où le joueur joue contre un autre joueur. Peu importe le mode de jeu sélectionné au lancement de la partie vous commencerez à l'Âge **Préhistorique**. Il existe cinq âges classés dans l'ordre suivant :

- **Préhistoire**
- **Antiquité**
- **Moyen-âge**
- **Moderne**
- **Futuriste**

FIGURE 3 – Les cinq âges de la **Préhistoire** à l'**Ère futuriste**

Une fois la partie lancée, le joueur trouvera sa base du côté gauche du terrain, il devra la défendre pour ne pas que son adversaire la détruise ! À l'opposé, du côté droit, se trouve la base de l'adversaire. Pour endommager celle-ci le joueur disposera de quatre unités différentes propres à chaque âge.

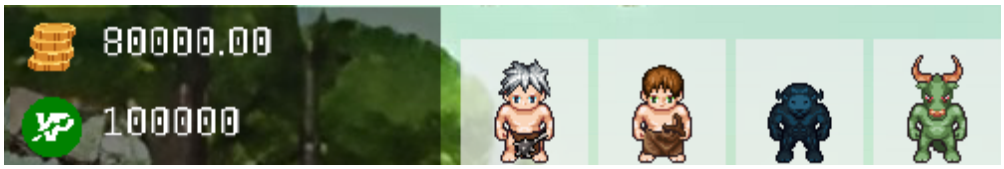

FIGURE 4 – HUD en partie

Pour déployer les unités, le joueur les achètera avec son argent (situé à gauche de la boutique). Il récoltera des pièces en éliminant des unités envoyées par son adversaire. De plus, pour rendre la bataille encore plus rude, un temps d'attente obligatoire est mis en place à chaque déploiement d'unité avant que celle-ci ne rentre dans l'arène. Les unités contenues dans la boutique évolueront et augmenteront en puissance quand le joueur passera à l'âge supérieur. Pour parvenir à l'époque suivante, il devra cliquer sur le bouton vert. Attention cela n'est pas gratuit, il déboursera l'expérience récoltée lorsqu'il éliminera des unités adverses.
Après avoir changé d'âge, le fond et la base du joueur auront changé de design et que les unités contenus dans le magasin également (à chaque âge le principe du jeu reste le même). C'est que le passage à l'âge supérieur a bien été effectué.

D'autre part, les deux joueurs possèdent une compétence spéciale appelée **ultime**. Cette compétence aura pour effet d'éliminer toutes les troupes adverses simultanément et instantanément. Cette capacité possède toutefois un temps avant utilisation très long. De plus, elle dispose d'un délai de recharge, se caractérisant par une incapacité d'utilisation pendant ce laps de temps.

Durant la partie, le joueur aura accès au menu « paramètres ». S'il souhaite interrompre la partie prématurément et la continuer ultérieurement, il pourra sauvegarder sa partie à tout moment. Le jeu n'accepte qu'une seule sauvegarde, s'il sauvegarde une partie, la partie précédemment sauvegardée sera écrasée.

Notre jeu propose plusieurs difficultés ainsi présentées :
- Facile
- Intermédiaire
- Difficile

Selon son choix, au début de la partie, une certaine somme d'argent lui sera assignée. De plus, l'ordinateur aura une plus grosse soif de victoire.

3.2 Fonctionnalités du programme

L'utilisateur aura la possibilité de se balader dans un menu avec de nombreuses fonctionnalités, comme par exemple la résolution de la fenêtre, le son, le mode de jeu solo, etc. Arrivé en partie, il pourra poser des unités sur le champ de bataille. Il aura également la possibilité de changer d'âge et d'utiliser une compétence spéciale. De plus, à tout moment, il pourra sauvegarder et reprendre sa partie plus tard. Il n'aura plus qu'à sélectionner

Reprendre partie dans le menu. Par ailleurs, durant une partie, le joueur aura toutefois accès aux paramètres, par exemple il pourra réduire le volume si la musique lui semble trop forte ou encore changer la résolution...Après avoir gagné ou perdu une partie, l'utilisateur aura le droit a une animation pour clore celle-ci. S'ensuit un retour au menu, où il pourra alors recommencer une partie ou quitter notre jeu.

Enfin, notre jeu possède un mode réseau. Cela signifie que deux personnes sur deux machines différentes vont pouvoir jouer l'un contre l'autre. Pour cela, un des deux joueurs créera la partie et son adversaire la rejoindra. Le joueur qui rejoindra son adversaire hébergeant la partie, devra avoir connaissance de *l'adresse IP* de celui ci. S'ensuit le déroulé d'une partie classique.

Le dossier *Battle_ of_ Time* est organisé de façon à se repérer facilement. Le *makefile* a été écrit dans l'optique d'être le plus généraliste possible. Les dossiers *obj*, *bin* et *save* seront crées s'ils n'existent pas lors de la première compilation avec la commande `make all` ou `make -f makefile all`. La commande `make mrproper` permet quant à elle de supprimer l'exécutable et les fichiers objets.

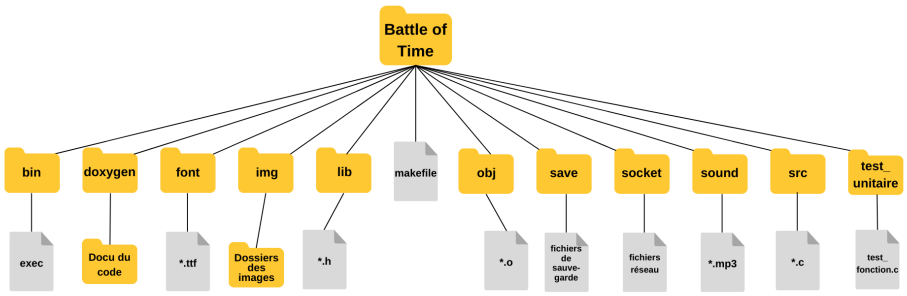

FIGURE 5 – Architecture dossier *Battle_ of_ Time*

3.3 Direction artistique du projet

L'idée principale pour la direction artistique était de créer nous-même les designs 2D/3D, c'est-à-dire les *sprites* ainsi que les images d'arrière plan. Cependant, nous nous sommes rapidement rendu compte que cela était long et compliqué à réaliser. Nous avons donc choisi le *pixel art* qui est plus facile à mettre en place avec de nombreux outils existants. Nous avons utilisé principalement 4 sites gratuits :

— *Freepik* (pour télécharger des images vectorielles libres de droit)
— *Online PNG Tools* (pour supprimer le fond des images)
— *Pixel It* (pour pixeliser les images)
— *Pixlr* (pour le montage photo)

Le processus pour créer les images d'arrière-plan du jeu 12 est le suivant :

1. Chercher une image d'arrière-plan vectorielle correspondant à l'âge
2. Chercher des images vectorielles d'éléments à placer sur le fond
3. Retirer l'arrière-plan des images (si cela n'est pas déjà fait)
4. Pixeliser les images pour être dans le style *pixel art*
5. Assembler l'image de fond et les éléments

En ce qui concerne la police d'écrire 13, nous avons choisi '**Handjet**', disponible sur *GitHub*. Pour la musique et les effets sonores, le site *Pixabay* nous a été d'une grande aide en nous permettant d'obtenir ces éléments.

4 Codage

4.1 Algorithme et structures de données

4.1.1 Outils

Le jeu est implémenté avec le langage C en utilisant *Visual Studio Code* pour plus de lisibilité et d'efficacité. La documentation du code est faite avec *Doxygen* et l'outil de gestion de versionning *GitHub* est utilisé pour la mise en commun, le suivi des modifications, la fusion et la gestion de dépôts de codes.

Dans un premier temps, c'est la partie fonctionnelle du jeu qui a été développée. Pour repérer les erreurs de programmation, le GNU Debugger (GDB) a été utilisé (voir en Annexe 6.1).

4.1.2 Structures de données

Pour les délais quels qu'ils soient, c'est l'utilisation de la fonction *difftime* qui vérifie l'écoulement du temps sans bloquer le programme. Il est également important de noter que le paramètre *owner* (présent dans les structures ci-dessous) permet de déterminer qui est le propriétaire de chaque objet.

Structure *building* 18 :

Le *building* correspond à la base que le joueur doit défendre (ou attaquer si c'est celle de l'adversaire). Elle possède une fonction d'initialisation et d'amélioration (`init_building` et `upgrade_building`).

Structure *character* 19 :

Pour des raisons de simplification, les statistiques de chaque « *character* »(combattant) sont initialisées par des valeurs prédéfinies. Seules les statistiques listées ci-dessous sont calculées par des formules mathématiques pour le bon équilibre du jeu :

— Le prix de sa formation
— Les dégâts qu'il peut infliger
— Ses points de vie
— Le temps de formation
— Sa vitesse de déplacement

Au début du jeu un tableau de combattant, répertoriant toutes les caractéristique des vingt unités, est initialisé (grâce à la fonction `initcharacter`). Les fonctions `character_attack_character`, `character_attack_building` et `building_attack_chatacter` permettent à deux combattant de s'attaquer entre eux ou d'attaquer la base adverse.

La structure *tab_ charactere_ t* 20 représente un tableau de combattants. La gestion des déplacements des combattants se fait par la fonction `deplacement`. Un combattant peut avancer si rien ne se trouve devant lui, sinon il ne bouge pas ou alors il attaque s'il se trouve au front. Lorsqu'un combattant meurt (ses points de vie tombent à zéro), `delete_character` le supprime et libère l'espace mémoire qui lui est réservé. Puis `tasser_tab` intervient pour replacer au bonne endroit les combattants encore en vie. Quand **l'ultime** est déclenché, `ulti` est appelée pour mettre à les points de vie de tous les combattants du camp adverse à zéro.

Structure du joueur 21 :

Player_ t possède également une fonction d'initialisation et de destruction (`initplayer` et `destroy_player`) pour réserver ou libérer la mémoire au bon moment. Si le joueur veut acheter un combattant, `buy_charater` vérifie si

celui-ci possède assez d'argent pour l'acheter. Puis, elle crée le combattant (réserve de la mémoire) en récupérant les information dans le tableau de combattant initialisé avec `initcharacter` et le place dans la file d'attente pour commencer sa formation. Intervient ensuit `envoie_char`, qui place les unités ayant fini leur formation (après quelques secondes) sur le champ de bataille.

En ce qui concerne l'ordinateur 22 :

L'envoi de personnage est illimité (pas de restrictions d'argent). Cela est décidé aléatoirement entre 1 et n secondes en fonction de la difficulté choisie dans un maximum de 15 secondes (en mode facile), 7,5 secondes (en mode normal) et enfin toutes les 5 secondes (en mode difficile).

$$o\text{->}delai = rand()\%(MAX\_DELAI/(o\text{->}difficulte)) + MIN\_DELAI;$$

Le choix du combattant (« *charactere* ») se fait aussi aléatoirement. Pour changer d'âge (et améliorer la base), l'ordinateur regarde si le joueur a gagner un certain nombre d'expérience. Ce nombre est déterminé en fonction du coût de l'amélioration et de la difficulté choisie.

L'ultime est également déclenché aléatoirement dans un intervalle de 5 à 15 minutes.

4.1.3 Sauvegarder une partie

Pour sauvegarder une partie, il a été décidé de créer deux fichiers de sauvegarde, un pour le joueur et un pour l'ordinateur (*joueur.txt* 2 et *ordi.txt* 3) contenant les informations nécessaires pour reprendre le jeu exactement là où il s'était arrêté.

4.2 Interface graphique (SDL)

4.2.1 *Sprites* et collisions

Pour réaliser les *sprites* des personnages présents dans notre jeu, nous avons utilisé ce générateur, sous licence CC BY-SA 3.0, donc libre d'utilisation si l'on crédite les auteurs de celui-ci dans notre projet. Ce générateur nous a permis de créer des personnages en choisissant leurs caractéristiques, le tout en *pixel art*. Les animations du personnage générées sont contenues dans une image png (voir figure 23).

Concernant les collisions, les personnages apparaissent les uns derrière les autres après avoir été sélectionnés. Ils ne peuvent pas se dépasser et si l'un des personnages se déplace plus rapidement que celui devant lui, il va alors ralentir à la même vitesse que celui-ci.

FIGURE 6 – Exemple de collisions

L'affichage des animations, des déplacements et des collisions des *sprites* a été réalisé grâce à la librairie SDL (Simple DirectMedia Layer).

Concernant les animations d'attaque des sprites, nous avons rencontré des problèmes d'implémentation tel que des bugs graphiques. C'est pourquoi nous avons décidé que les sprites utiliseront tous les mêmes animations d'attaques qui sont soit le coup de poing, soit l'animation de tir à l'arc.

4.2.2 Interface du menu et du jeu

L'interface graphique du menu et du jeu a été réalisée en SDL2 en utilisant les bibliothèques suivantes :

— **SDL.h**
— **SDL_image.h**
— **SDL_mixer.h**
— **SDL_ttf.h**

Lorsque le jeu est lancé, nous arrivons sur la page d'accueil 1 qui nous emmène ensuite sur le menu principal 14 qui permet d'accéder aux différents sous-menus comme les options, les crédits, jouer au jeu ou encore quitter le programme. Le fichier *afficher_menu.c* contient les fonctions d'affichage des différents sous-menus. Ce fichier contient également la fonction `void affichage(etat_t etat, ...)` qui gère à l'aide du paramètre `etat_t etat`, dans un `switch`, l'entièreté de l'affichage du programme en appelant,

en fonction de l'état d'affichage, les fonctions d'affichage des menus et du jeu.

- **Menu Principal**
  - **Jouer**
    - **Solo**
      - **Nouvelle Partie**
      - **Reprendre Partie**
    - **En ligne**
      - **Créer Partie**
      - **Rejoindre Partie**
  - **Options**
    - **Musique/Son**
    - **Résolution**
  - **Crédits**
  - **Quitter**

FIGURE 7 – Architecture du menu

Chaque menu du jeu possède sa fonction d'affichage, permettant de se repérer plus facilement dans le code. Le menu *Jouer* permet d'accéder à deux autres sous-menus, pour jouer seul contre l'ordinateur ou bien en ligne avec un autre joueur. Le sous-menu *Solo* a deux éléments permettant de jouer une nouvelle partie contre l'ordinateur avec trois niveaux de difficulté (facile, moyen et difficile) ou bien de continuer une partie existante grâce à un système de sauvegarde. L'autre sous-menu de *Jouer*, *En ligne*, donne la possibilité à l'utilisateur de jouer contre une autre personne (voir le 4.3). Le sous-menu *Options* possède deux sous-menus, *Musique/Son* et *Résolution*, pouvant contrôler respectivement les sons et les dimensions de la fenêtre du programme.

Dans le menu *Musique/Son*, il est possible d'augmenter ou de diminuer le volume de la musique, grâce à un curseur amovible 15, ainsi que d'activer ou de désactiver les effets sonores. L'utilisation et la gestion de sons dans le jeu est possible grâce à la librairie **SDL_mixer.h**. Ensuite, dans le menu *Résolution*, nous pouvons modifier la taille de la fenêtre parmi un choix de dimensions disponibles dans une liste déroulante horizontalement 16. Un bouton plein écran permet, comme son nom l'indique, au programme d'occuper toute la taille de l'écran.

Le fichier *gestion.c* contient les fonctions de gestion des inputs, notamment la fonction de gestion des cliquables qui occupe une bonne partie de

ce fichier. En effet, cette fonction évalue si un clic de la souris a été fait dans une zone précise (ex : un bouton du menu) et effectue le traitement correspondant à l'action du bouton. L'évaluation du clic se fait avec les coordonnées de la souris que l'on récupère (appelées `mouseX` et `mouseY`) à l'aide des instructions suivantes :

— `mouseX = evenement.button.x;`
— `mouseY = evenement.button.y;`

Ainsi nous vérifions si ces coordonnées récupérées sont comprises dans un intervalle de coordonnées correspondant à la zone du bouton. Les autres fonctions de fichier gèrent le déplacement de la souris, qui est utilisée pour contrôler le survol des éléments dans le jeu, l'action d'appui sur une touche du clavier et la saisie de texte au clavier.

Lorsqu'une partie est lancée, l'interface du jeu apparaît 17. Nous y retrouvons en arrière-plan l'image correspondant à l'époque en cours, la base du joueur (à gauche) ainsi que celle de son adversaire (à droite). Également, au-dessus d'elles, s'affichent leur nombre de points de vie, l'HUD avec, en haut à gauche, les informations de la boutique, des personnages disponibles, le bouton pour améliorer sa base et changer d'âge, le bouton pour utiliser son **ultime**, et en haut à droite le bouton permettant d'accéder au menu des paramètres. Des informations supplémentaires et cachées sont visibles en survolant les boutons pour améliorer sa base, attaquer avec son **ultime** et acheter des personnages. L'affichage de cette interface se fait grâce aux fonctions disponibles dans le fichier *afficher_jeu.c*.

Les fonctions d'affichages, du menu et du jeu, prennent quasiment toutes en entrée les paramètres suivants :

— `SDL_Renderer* rendu` (le rendu qui contient le contexte d'affichage de la fenêtre)
— `SDL_Window* fenetre` (la fenêtre d'affichage du rendu)
— `TTF_Font* police` (la police d'écriture)
— `SDL_Texture* image` (les textures des images à afficher)

Ces paramètres sont nécessaires pour charger les images ou du texte dans le rendu et donc de les afficher dans la fenêtre du programme. L'utilisation de la police et le chargement des images sont possibles grâce aux librairies **SDL_ttf.h** et **SDL_image.h**.

En ce qui concerne le mouvement de l'écran par rapport au reste de l'image 8, celui-ci est géré dans la fonction `void afficherJeuFond(...)`. La position x de la caméra est initialisée à zéro quand on lance une partie,

puis elle est mise à jour en additionnant (quand le curseur de la souris est vers la droite) ou en soustrayant (quand le curseur de la souris est vers la gauche) à elle-même la vitesse de déplacement. Le déplacement de l'écran s'arrête quand on atteint les limites de l'image, c'est à dire quand la position x de la caméra vaut zéro ou quand celle-ci vaut : $LARGEUR\_IMAGE - LARGEUR\_ECRAN$

FIGURE 8 – Schéma du déplacement de l'écran

4.3 Le mode réseau

Notre jeu possédant un mode réseau, nous allons utiliser plusieurs librairies dédiées en C. Nous avons utilisé une architecture semblable à du *serveur-client*. Une machine prendra le rôle du serveur et une autre celui du client.

Pour que le déroulement d'une partie en ligne soit sans accroc pour les deux joueurs, nous allons devoir faire transiter entre les deux machines des informations essentielles. Dans un premier temps, nous allons mettre en contact les deux machines. *Dans le menu, on va effectuer la demande de l'adresse IP du serveur. Nous sommes obligés de demander à l'utilisateur de rentrer l'adresse IP de la machine avec laquelle il va jouer, car nous ne pouvons pas avoir de serveur fixe avec une adresse qui ne changera pas au cours du temps.* Note : La machine A prend le rôle du *serveur* et la machine B devient le *client*.

On suit ce principe d'initialisation :

1. Sur la machine A on **affiche son adresse IP**.
2. On met **le port X en écoute** sur la machine A.

3. Sur la machine B, on **rentre l'adresse IP de la machine A**.
4. La machine B **se connecte sur le port X** de la machine A.

Les machines A et B sont maintenant en communication. Tant que la connexion n'a pas été rompue, les deux machines peuvent communiquer entre elles. Deuxièmement, nous devons faire transiter les informations, les librairies inclues nous permettant d'utiliser des fonctions déjà écrites et fiables. Les deux fonctions principales pour le partage de données sont :

— `send(int s, const void * buf, size_t len, int flags)`
— `recv(int s, void * buf, size_t len, int flags)`

Ces deux fonctions sont très similaires. En effet :

	int s	void * buf	size_t len
send	@ destination	pointeur générique sur un objet	taille de l'objet
recv	@ de l'expéditeur	pointeur générique sur un objet résultat	taille de l'objet

Table 1 – Récapitulatif des arguments

Les noms des fonctions ont été adaptés à notre programme. Ces dernières se nomment *envoyer* et *recevoir* :

— `void envoyer(int to, int * action, int * action2)`
— `void recevoir(int to, int * action, int * action2)`

Notre philosophie pour le mode réseau est plutôt simple, on partage les informations de chaque joueur en fonction de ses actions, puis on appliquera ses actions depuis notre machine. Par exemple :

— Si notre adversaire envoie une unité, alors on envoie dans *action* **0**. L'*action2*, elle va servir à connaître une information complémentaire si nécessaire. Par exemple, dans ce cas, le numéro unique pour reconnaître l'unité sera utile.
— Si notre adversaire souhaite passer à l'âge supérieur, alors *action* prendra la valeur **1**. Cette information étant suffisante, l'*action2* prendra la valeur **-1**.
— Enfin, si notre adversaire veut utiliser sa compétence spéciale alors, *action* sera égale à **2**. Cette information étant suffisante, l'*action2* prendra la valeur **-1**.

Finalement, la partie la plus complexe a été effectuée. En effet il ne reste plus qu'à appeler les fonctions d'initialisation (cf liste d'initialisation), puis

d'envoi ou de réception en fonction de la situation. Le schéma est donc le suivant :

1. Initialisation du serveur et client
2. Envoi et réception
3. Traitement (on applique les actions de notre adversaire)
4. Déconnexion

5 Conclusion et résultats

5.1 Fonctionnalités du programme réalisées

Les principales fonctionnalités du programme ont été réalisées. C'est-à-dire un menu interactif et intuitif permettant à l'utilisateur de modifier la résolution et le son, consulter les crédits et bien évidemment jouer au jeu. Les fonctionnalités de base du jeu ont été implémentées comme l'achat d'une unité et son apparition sur le champ de bataille, l'utilisation de **l'ultime**, le changement d'âge mais surtout les troupes qui se battent. La possibilité de sauvegarder et reprendre sa partie est également présente. Concernant le mode réseau, il a été compliqué pour nous de l'implémenter et de comprendre son fonctionnement, c'est pour cela qu'il est présent dans le menu du jeu mais qu'il n'est pas totalement fonctionnel.

5.2 Planning prévisionnel respecté

Malgré nos efforts pour tenir le planning, il n'a pas pu être respecté en raison de plusieurs problèmes de développement. Le débogage du programme a pris plus de temps que prévu. D'autre part, des fonctionnalités du jeu qui n'étaient pas prévues lors de l'étape de la conception ont dû être ajoutées.

5.3 Améliorations possibles

Ce projet possède certains axes d'amélioration.
Premièrement d'un point de vue graphique en jeu. Lorsqu'une partie est lancée, les effets visuels sont limités. En effet, notre philosophie est la simplicité et l'accessibilité pour tous les publics, qu'ils soient enfants ou adultes. Toutefois, il existe une possibilité de mettre en place des animations apportant au joueur une meilleure expérience de jeu. Nous pouvons notamment prendre l'exemple d'une animation plus poussée pour l'explosion de **l'ultime** lorsqu'un clic est effectué sur le HUD, etc.

Deuxièmement, pour le mode réseau, l'optimal serait d'avoir un serveur fixe qui puisse héberger une partie en ayant vraiment une architecture *serveur-réseau*. Cela éviterait au joueur d'aller chercher dans le menu l'adresse IP de la machine contre laquelle il veut jouer.

Par ailleurs, en raison du manque de temps et de compétence pour la direction artistique du jeu, nous avons été contraints de nous orienter vers le *pixel art*. Ce choix, nous garantit l'accès à des outils simples et gratuits. Notre idée de départ était d'avoir une direction artistique axée sur le réalisme pour mettre davantage en immersion le joueur durant une partie.

Troisièmement, les sprites ne possèdent pas leur propre animation d'attaque en raison de problèmes d'implémentation dans le code.

Enfin, nous n'avons pas pu rajouter certaines compétences qui permettraient au joueur d'explorer plusieurs stratégies pour remporter une partie. Cela pourrait passer par des missions à faire au cours d'une partie ainsi que l'ajout d'autres compétences comme **l'ultime** déjà disponible.

5.4 Apports du projet

Ce projet nous a permis d'appliquer les connaissances et les techniques que nous avons vu durant notre première et deuxième année de licence informatique. De plus, il nous a permis de découvrir la gestion de projet et toute l'organisation de groupe qu'il est nécessaire d'avoir pour la bonne réalisation du projet. Pour nous, la réalisation de celui-ci a été une réelle motivation car nous avons tout conçu du début à la fin.

6 Annexe

6.1 Debugage

6.1.1 Exemple 1

Exemple de situation d'utilisation du GDB : Une nouvelle fonction vient d'être implémentée (« delete_character »dans notre exemple) et l'on a remarqué lors d'une précédente exécution, pendant l'utilisation de **l'ultime**, la destruction des combattants ne se fait pas correctement. C'est ici qu'intervient GDB : *gdb ./bin/prog*

Un point d'arrêt est placé sur la fonction « delete_character » : *(gdb) b delete_character*
Breakpoint 1 at 0x4037 : file src/tab_character.c, line 3.

Puis le programme est lancé normalement : *(gdb) r*
Starting program :

Vient ensuite le premier arrêt du programme *ORDINATEUR*
Troupe :
<—– characters possedees —–>
:0 : combattant caillou, owner : 3, pv : 0
:1 : gorille, owner : 3, pv : 0
:2 : combattant caillou, owner : 3, pv : 0
:3 : gorille, owner : 3, pv : 0
:4 : combattant dinosaure, owner : 3, pv : 0
:5 : combattant dinosaure, owner : 3, pv : 0
:6 : combattant caillou, owner : 3, pv : 0
:7 : combattant massue, owner : 3, pv : 0
:8 : combattant dinosaure, owner : 3, pv : 0
:9 : gorille, owner : 3, pv : 0
<——————————————>
L'ultime vient d'être utilisé. Premier arrêt du programme :
Breakpoint 1, delete_character (characters=0x55555555d880 <afficher_player+138>) at srctab_character.c :3
*3 booleen_t delete_character(tab_charactere_t ** characters) (gdb) n*
*17 for(int i=0 ; i<(*characters)->nb ; i++)*
*(gdb) display (*characters)->tab*
*1 : (*characters)->tab = {0x55555555dab0, 0x55555555db90, 0x55555555dc70, 0x55555555dd50, 0x55555555de30, 0x55555555df10, 0x55555555dff0, 0x55555555e0d0, 0x55555555e1b0, 0x55555555e290}*

En défilant le programme pas à pas, l'erreur apparaît :

*19 if ((*characters)->tab[i]->pv <= 0)*
*1 : (*characters)->tab = {0x0, 0x0, 0x0, 0x0, 0x55555555de30, 0x55555555df10, 0x55555555dff0, 0x55555555e0d0, 0x55555555e1b0, 0x55555555e290}*
(gdb)
*26 free((*characters)->tab[i]) ;*
*1 : (*characters)->tab = {0x0, 0x0, 0x0, 0x0, 0x55555555de30, 0x55555555df10, 0x55555555dff0, 0x55555555e0d0, 0x55555555e1b0, 0x55555555e290}*
(gdb)
*27 (*characters)->tab[i] = NULL ;*
*1 : (*characters)->tab = {0x0, 0x0, 0x0, 0x0, 0x55555555de30, 0x55555555df10, 0x55555555dff0, 0x55555555e0d0, 0x55555555e1b0, 0x55555555e290}*
(gdb)
*29 (*characters)->nb−;*
*1 : (*characters)->tab = {0x0, 0x0, 0x0, 0x0, 0x0, 0x55555555df10, 0x55555555dff0, 0x55555555e0d0, 0x55555555e1b0, 0x55555555e290}*
*(gdb) display (*characters)->nb*
*2 : (*characters)->nb = 5*

La variable « (*characters)->nb »est décrémentée au mauvais endroit.

6.1.2 Exemple 2

Au lancement du programme, celui-ci s'est arrêt à cause d'une erreur de segmentation. Pour trouver cette erreur, nous avons utiliser le débogueur GDB et placer, avec celui-ci, un breakpoint sur une instruction qui pouvait poser problème. Ensuite, le programme a été lancé jusqu'au breakpoint.

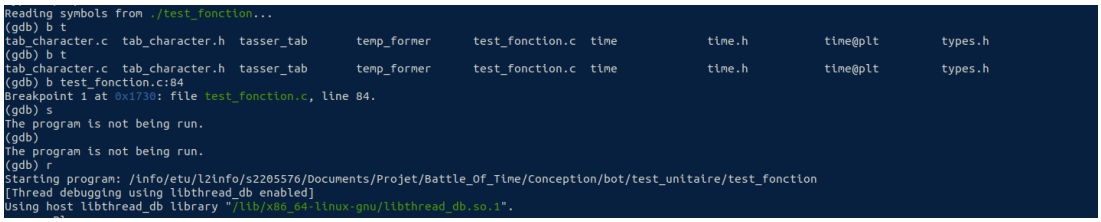

FIGURE 9 – Début du Debogage

Comme on peut le voir ci-dessous, l'erreur de segmentation est dû à une case du tableau à l'indice 0. On en déduit que nous essayons d'accéder à une case du tableau dont l'espace mémoire n'a pas été alloué. Grâce au débogueur, nous avons pu remarquer que le tableau était est réalité vide.

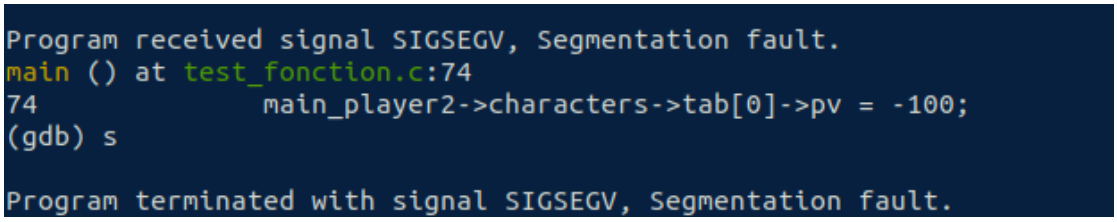

FIGURE 10 – Fin du Debogage

6.2 Tests

Des tests unitaires ont été réalisés sur les fonctions principales du jeu. C'est-à-dire, les fonctions suivantes :

- `booleen_t buy_character(player_t ** player,...);`
- `player_t * initplayer(int owner);`
- `void destroy_player(player_t ** player);`
- `void envoie_char(player_t ** player);`
- `booleen_t get_ressources(player_t * player1, player_t * player2);`
- `booleen_t afficher_player(player_t * player);`
- `ordi_t * init_ordi();`
- `int detr_ordi(ordi_t ** ordi);`
- `int envoie_char_ordi(ordi_t * ordi, character_t * tab);`
- `booleen_t give_ressources(player_t * player, ordi_t * ordi);`
- `void afficher_ordi(ordi_t * ordi);`
- `void jeu_ordi(ordi_t * o, player_t * p, character_t * tab);`

Le fichier .c contenant les différents tests est disponible sur le GitHub

retour de la fonction : TRUE si achat valide, FALSE sinon

argument de la fonction : un pointeur sur joueur, le tableau de toutes les unités et l'indice de l'unité à acheter

Constante *: NB_UNITÉ 20, MAX_POSSÉDÉ 10*

Cas valide	**Jeu données**	**Résultat**
→ Pointeur sur joueur != NULL → tableau d'unité initialisé → indice > 0 && indice < NB_UNITÉ → le joueur possède assez de ressources pour acheter l'unité → le liste d'unité du joueur n'est pas pleine	Joueur : j1 tableau : tab_unité indice = 0	1

Cas invalide	**Jeu données**	**Résultat**
Pointeur sur joueur == NULL	player_t * j1 ; — puis — achat(j1,tab_unité,0);	0
Indice < 0 || indice >= NB_UNITÉ	-2 ou 21	0
L'argent du joueur – coût de l'unité ≥ 0	Argent du joueur = 100 coût de l'unité = 20	0
Si le nb d'unité possédé par le joueur + celle qu'on va acheter est >= MAX_POSSÉDÉ	Nb possédé par le jouer = 9 (de 0 à 9 donc 10)	0
Si on ajoute une unité (tout les cas précédents sont valides), mais que l'allocation dynamique pour ajouter l'unité dans le tableau du joueur échoue	Si j1→ tab[i] == NULL	0

FIGURE 11 – Exemple de jeu de test pour fonction achat d'unité

6.3 Images

FIGURE 12 – Image de fond pour l'Âge **Préhistorique**

FIGURE 13 – Exemple de la police **Handjet**

FIGURE 14 – Menu Principal

FIGURE 15 – Curseur amovible

FIGURE 16 – Liste déroulante avec les dimensions de la fenêtre

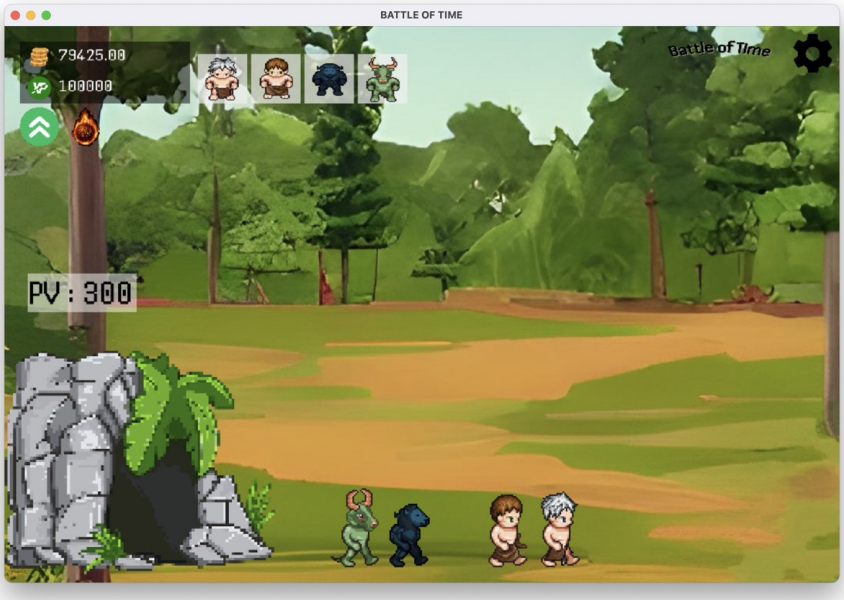

FIGURE 17 – Interface du jeu

```c
/**
 * @brief Structure représentant un bâtiment.
 */
typedef struct building_s {
    int pv;          /**< Points de vie du bâtiment. */
    int owner;       /**< Propriétaire du bâtiment. */
    int dammage;      /**< Dégâts infligés par le bâtiment. */
    int max_pv;      /**< Points de vie maximum du bâtiment. */
    int XP_cost;     /**< Coût en expérience pour améliorer le bâtiment. */
    int level;       /**< Niveau du bâtiment. */
} building_t;
```

FIGURE 18 – Structure de *building*

```c
/**
 * @brief Structure représentant un personnage.
 */
typedef struct character_s {
    int age;                        /**< Âge du personnage. */
    int classe;                     /**< Classe du personnage. */
    int cost;                       /**< Coût du personnage. */
    int dammage;                     /**< Dégâts infligés par le personnage. */
    char description[MAX_DESCRIPTION]; /**< Description du personnage. */
    int first_Attaque;                 /**< Si l'attaque du personnage est disponible.. */
    int max_pv;                     /**< Points de vie maximum du personnage. */
    char name[MAX_STR];             /**< Nom du personnage. */
    int owner;                      /**< Propriétaire du personnage. */
    int pv;                         /**< Points de vie actuels du personnage. */
    float ratio_ressources;          /**< Ratio de ressources du personnage. */
    char sprite[MAX_DESCRIPTION];   /**< Sprite du personnage. */
    int time;                       /**< Temps de jeu du personnage. */
    int vector;                     /**< Vecteur du personnage. */
    int x;                          /**< Position horizontale du personnage. */
    int x_pred;                     /**< Position horizontale précédente du personnage. */
    int y;                          /**< Position verticale du personnage. */
} character_t;
```

FIGURE 19 – Structure de *character*

```c
/**
 * @brief Structure représentant un tableau de personnages.
 */
typedef struct tab_charactere_s {
    character_t * tab[MAX_POSSESSED]; /**< Tableau de personnages */
    int nb; /**< Nombre de personnages dans le tableau */
    int ind_first_vivant; /**<indice dans le tableau du premier combattant encore en vie.*/
} tab_charactere_t;
```

FIGURE 20 – Structure de *tab_character*

```c
/**
 * @brief Structure représentant un joueur.
 */
typedef struct {
    char name[MAX_STR]; /**< Nom du joueur */
    int xp; /**< Expérience du joueur */
    int delai; /**< Délai pour former les troupes */
    unsigned long int debut; /**< Début du délai */
    unsigned long int fin; /**< Fin du délai */
    float gold; /**< Quantité d'or du joueur */
    int owner; /**< Numéro propriétaire du joueur */
    tab_charactere_t * characters; /**< Tableau des personnages du joueur */
    tab_charactere_t * file_attente; /**< File d'attente des personnages à former */
    building_t * building; /**< Bâtiment du joueur */
} player_t;
```

FIGURE 21 – Structure du joueur

```c
/**
 * @brief Structure représentant un adversaire contrôlé par l'ordinateur.
 */
typedef struct {
    int owner; /**< Numéro du propriétaire */
    int difficulte; /**< Niveau de difficulté */
    int delai; /**< Délai pour former les troupes */
    int xp; /**< Points d'expérience */
    int delai_ulti; /**< Délai pour l'utilisation de l'ultime */
    tab_charactere_t * characters; /**< Tableau de personnages */
    building_t * building; /**< Bâtiment */
} ordi_t;
```

FIGURE 22 – Structure de l'ordinateur

Information	**joueur.txt**
golde	0.000000
owner	1
nom	Player1
xp	0
delai	2
début	-1
fin	1713942684
attaque_du_building	500
niveau	0
max_pv	48000
coût_amélioration	10000
pv	48000
nombre_combattant_sur_le_terrain	3
pv_première_unité	8000
classe	3
position_x	954
position_y	0
position_x_précédant	938
classe	0
position_x	650
position_y	0
position_x_précédant	635
classe	0
position_x	290
position_y	0
position_x_précédant	275
nombre_combattant_en_formation	1
class	2

TABLE 2 – Exemple de sauvegarde pour le joueur

Information	**joueur.txt**
owner	3
difficulté	3
delai	4
xp	0
délai de l'ultime	650
attaque_du_building	500
niveau	0
max_pv	48000
coût_amélioration	10000
pv	48000
nombre_combattant_sur_le_terrain	2
pv_première_unité	8000
classe	2
position_x	3060
position_y	0
position_x_précédant	3075
classe	0
position_x	3635
position_y	0
position_x_précédant	3650

TABLE 3 – Exemple de sauvegarde pour l'ordinateur

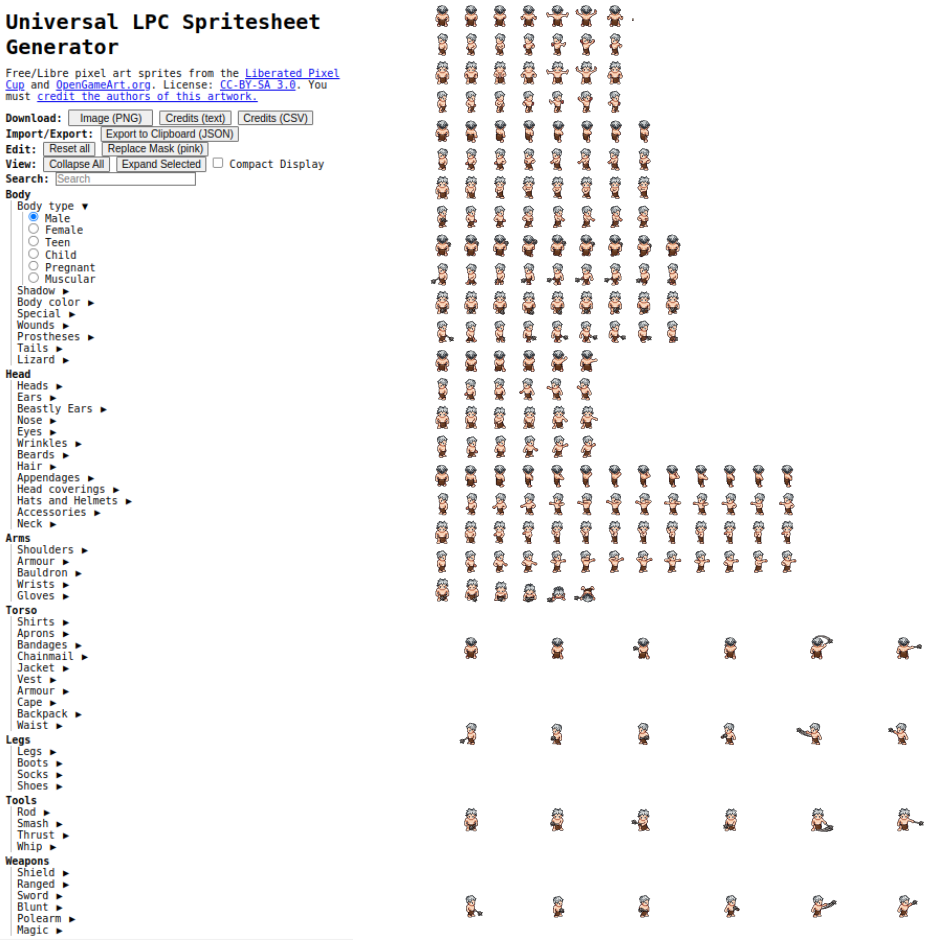

FIGURE 23 – Exemple du générateur et d'un *sprite*