

XenoKart Eco+ Ultimate (Not) Definitive Edition

Licence Informatique *2ème année*

Mano Brabant
Benjamin Riviere
Erwan Serelle

22 avril 2019

Table des matières

1	Introduction	3
2	Organisation du travail	3
3	Analyse	3
4	Conception	4
4.1	Menu	4
4.2	Lobby	5
4.2.1	Trie	5
4.2.2	Choix de carte	5
4.2.3	Commerce	5
4.3	Carte	6
4.3.1	Génération	6
4.3.2	Pathfinding	6
4.3.3	Utilisation	6
4.4	Combat	6
4.4.1	Statistiques	7
4.4.2	Auto-attaques/Arts	7
4.4.3	Hostilité	8
4.4.4	Commandes	8

5	Réalisation	9
5.1	Menu	9
5.2	Lobby	9
5.2.1	Trie	10
5.2.2	Choix de carte	10
5.2.3	Commerce	10
5.3	Carte	11
5.3.1	Génération	11
5.3.2	Pathfinding	11
5.3.3	Utilisation	12
5.4	Combat	12
5.4.1	Structures	12
5.4.2	Organisation	13
5.4.3	Réseau	14
6	Résultats	14
6.1	Améliorations	14
6.1.1	Menu	15
6.1.2	Lobby	15
6.1.3	Carte	15
6.1.4	Système de combat	15
7	Conclusion	16
8	Annexe	17

1 Introduction

Pour ce projet, notre objectif était de faire un jeu fun avec des mécaniques plus originales qu'un dungeon crawler[1] classique. Le but du jeu est d'obtenir le meilleur score possible. On obtient des points en échange d'objet récupérés pendant l'exploration de différentes cartes et pendant des combats.

2 Organisation du travail

Nous avons défini les différentes tâches à réaliser pour le projet et nous les avons réparties de la manière suivante :

- Le menu : Benjamin Riviere
- Le lobby : Erwan Serelle
- La carte :
 - Génération, Pathfinding : Benjamin Riviere
 - Utilisation : Erwan Serelle
- Le système de combat : Mano Brabant

Le projet est stocké sur [GitHub](#). [2]

Le dépôt a été structuré en plusieurs parties :

- SDL2 : contient la bibliothèque Simple Direct Layer
- code : contient les différents .c et .h qui constituent le programme
 - menu
 - lobby
 - map
 - combat
- data : contient les différents textes, images et autres données nécessaires au programme.
- docs : contient les documents annexes.

3 Analyse

Le but du jeu est d'obtenir le meilleur score possible à la fin de la partie. On obtient des points en échange d'objets qu'on récupère en expédition. On peut aussi choisir de vendre une partie de ses objets contre de l'argent. Cet argent nous permet d'acheter des objets utilitaires à un marchand qui nous aideront au cours de notre partie. On peut ensuite partir pour l'exploration du lieu de notre choix.

On arrive alors sur une carte composée de cases hexagonales de différents types (forêt, désert, eau ...) et on doit rechercher et atteindre la case d'arrivée. Il y aura des ennemis à combattre sur le chemin, et des cases spéciales à explorer pour obtenir le plus d'objets possible.

4 Conception

4.1 Menu

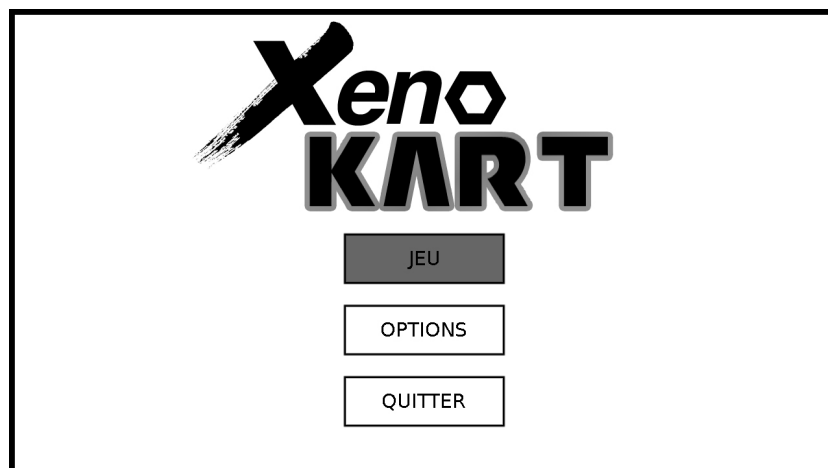
Le menu est la première interface sur laquelle l'utilisateur arrive au lancement du jeu. Tout d'abord il y a le logo du jeu et une invitation à appuyer sur n'importe quelle touche afin de lancer le menu.

Ensuite l'utilisateur arrive sur une interface avec des boutons qui peuvent être sélectionnés avec les touches du clavier :

- Z ou Flèche vers le haut : pour aller vers le haut.
- S ou Flèche vers le bas : pour aller vers le bas.
- Entrée ou Espace : pour sélectionner le bouton sur lequel on est positionné.
- Echap ou Retour : pour retourner en arrière.

Le bouton sur lequel on est positionné est symbolisé par le bouton de couleur rouge.

Le menu est composé de sous-menus, tel que le menu des options ou le menu du jeu. Leur fonctionnement est le même que pour le menu principal. Le menu est affiché comme le schéma ci-dessous avec des rectangles pour représenter les boutons, sur lesquelles sont inscrits les textes qui indiquent leurs fonctions.



Le menu se compose ainsi :

- Jeu
 - Charger (charge une partie)
 - Nouveau (commence une nouvelle partie)
- Options
 - Coop (permet de jouer en multijoueurs)
 - Serveur (indique que nous serons un serveur)
 - Client (indique que nous serons un client)
 - Solo (permet de jouer tout seul)
- Quitter (quitte le programme)

4.2 Lobby

Le lobby est une suite d'interfaces qui permet au joueur de se préparer pour l'expédition. Elle est composée de 3 modules : le premier sert à trier son inventaire, le second lui permet de choisir le lieu qu'il va explorer, et il pourra acheter des objet à un marchand dans le dernier. L'inventaire du joueur, un tableau de structure objet, y est initialisé. La structure objet est composée du nom de l'objet, de son effectif dans l'inventaire, de sa valeur en argent et de sa valeur en score.

4.2.1 Trie

Le premier module du lobby permet de trier son inventaire. Les objets de l'inventaire du joueur apparaissent un par un sur l'écran. On attend que le joueur choisisse quoi faire de son objet avant de passer au suivant. Il a le choix entre garder l'objet, l'échanger contre des points ou contre de l'argent. Une fois que tous les objets sont triés, on passe au choix de la carte.

4.2.2 Choix de carte

L'interface est constituée d'une carte du monde avec des drapeaux symbolisant les différents lieux explorables. On peut naviguer entre les différents drapeaux pour sélectionner la zone de notre choix. La valeur de la zone sélectionnée sera utilisée plus tard pour générer la carte. Une fois ce choix effectué, on passe au commerce.

4.2.3 Commerce

Dans cette dernière étape du lobby, le joueur pourra acheter des objets à un marchand. L'inventaire du joueur et celui du marchand sont affichés sur l'écran. Le joueur peut déplacer un curseur sur l'inventaire du marchand. Il

peut acheter les objets du marchand tant qu'il a assez d'argent. Les objets achetés sont alors transférés dans son inventaire.

4.3 Carte

La carte est ce sur quoi l'utilisateur va se déplacer en jeu afin de récolter des objets ou combattre en expédition, elle est composée de cases en hexagone ayant chacun un type (eau, plaine, forêt, etc). Il y a une case où on apparaît et une case de fin pour finir l'expédition.

4.3.1 Génération

La génération de la carte se fait case par case avec un nombre de cases par carte défini ainsi qu'un type de carte. Pour chaque case générée on lui assigne un type de case aléatoirement (influencer par le type de la carte et par ses voisins) et elle a une chance définie de contenir un ennemi ou un objet. A la fin on place aléatoirement une case de début et de fin.

4.3.2 Pathfinding

Dans ce jeu, on utilise le pathfinding pour la génération de la carte afin de vérifier que la carte soit jouable (pas de carte impossible à finir à cause des cases d'eau) ou soit trop facile (nombre de cases entre le début et la fin trop court). On s'en sert aussi pour la gestion des déplacements pour savoir quelles cases sont accessibles à l'utilisateur en un déplacement.

4.3.3 Utilisation

On gère dans cette partie le déplacement du personnage sur la carte ainsi que leur affichage sur l'écran en fonction de la position de la caméra. Le joueur doit simplement cliquer sur une case de la carte pour y déplacer le personnage et recentrer la caméra sur celui-ci.

4.4 Combat

Quand on rencontre un ennemi sur la carte on rentre en combat.

Le système de combat est une reprise du système de Xenoblade Chronicles un jeu sorti le 10 juin 2010 sur la Wii.

On contrôle en temps réel, dans un environnement en 2D vu de dessus, une équipe de trois personnages qui vont se battre contre un ou plusieurs ennemis. Pour gagner il faut réduire les PV des ennemis à zéro. Si les PV des trois personnages de notre équipe sont réduits à zéro on perd.

On possède une caméra qui nous permet d'afficher les éléments en fonction de la position du personnage que l'on contrôle (on affiche le personnage au centre de la fenêtre et on affiche les éléments qui se trouvent autour de lui).

Les ennemis sont situés au centre de la carte de combat et commencent à nous attaquer quand ils subissent des dégâts.

4.4.1 Statistiques

Les personnages et les ennemis possèdent plusieurs statistiques à prendre en compte durant le combat :

- Point de Vie.
- Force : Détermine la puissance des auto-attaques et des compétences physiques.
- Magie : Détermine la puissance des compétences magiques.
- Défense physique : Détermine la résistance aux dégâts physiques.
- Défense magique : Détermine la résistance aux dégâts magiques.
- Vitesse d'attaque : Détermine la fréquence des auto-attaques.
- Critique : Détermine le pourcentage de chance qu'une auto-attaque ou qu'une compétence provoque un coup critique.
- Agilité : Détermine le pourcentage de chance d'esquiver une attaque.
- Dextérité : Détermine le pourcentage de chance de toucher sa cible.
- Portée : Détermine la portée des auto-attaques.
- etc.

Ses statistiques sont enregistrées dans des fichiers que l'on charge au début du programme. Chaque personnage et ennemi possède des statistiques différentes.

4.4.2 Auto-attaques/Arts

Pour réduire les PV des ennemis on utilise deux moyens

- Les auto-attaques
- Les arts

Durant le combat chaque personnage cible un ennemi. Ce sera cet ennemi qui sera la victime des auto-attaques et des arts.

Les auto-attaques sont des attaques que les personnages réalisent automatiquement quand ils sont à portée de l'ennemi qu'ils ciblent. Les dégâts que les auto-attaques infligent sont déterminés par la force de l'attaquant ainsi que la défense physique de la victime. Les auto-attaques sont réalisées de façon régulière à une fréquence déterminée par la vitesse d'attaque.

Les arts sont des compétences spéciales utilisables au cours du combat. Les arts peuvent réaliser différentes choses :

- Infliger des dégâts.
- Soigner les membres de l'équipe.
- Améliorer les capacités des membres de l'équipe.
- Réduire les capacités des ennemis.

Les arts ne peuvent pas être utilisés plusieurs fois à la suite, après leur utilisation ils ont un temps de recharge durant lequel ils ne peuvent pas être utilisés. Les arts ont des effets bonus, en fonction de l'orientation du personnage par rapport à l'ennemi au moment où ils sont utilisés, par exemple certains arts font plus de dégâts quand ils sont utilisés dans le dos de l'adversaire.

Les arts sont enregistrés dans des fichiers que l'on charge au début du programme. Chaque personnage possède des arts qui lui sont propres.

Quand des dégâts sont infligés ou quand on soigne des personnages, etc, le nombre de dégâts ou de soins apparaît au dessus du personnage ou de l'ennemi concerné.

4.4.3 Hostilité

Les ennemis disposent uniquement des auto-attaques pour infliger des dégâts à l'équipe.

Les ennemis vont cibler un personnage en particulier grâce au système d'hostilité.

Quand on inflige des blessures à un ennemi on augmente la valeur de notre hostilité pour cet ennemi

Quand on soigne notre équipe, que l'on utilise des arts, on augmente la valeur de notre hostilité pour chaque ennemi que l'on est en train de combattre. Chaque ennemi va donc cibler le personnage qui lui semble le plus menaçant. Les ennemis ainsi que les personnages non contrôlés par l'utilisateur vont automatiquement suivre le personnage ou l'ennemi qu'ils ciblent, jusqu'à ce que celui-ci soit à portée des auto-attaques. Il est impératif de gérer cette valeur afin que les ennemis attaquent les personnages les plus résistants du groupe et que les personnages possédant des arts avec des effets bonus puissent les utiliser dans le dos ou sur les côtés de l'ennemi.

4.4.4 Commandes

On peut déplacer les personnages sur la carte de combat à l'aide des touches ZQSD :

- Z pour aller vers le haut.
- Q pour aller vers la gauche.
- S pour aller vers le bas.
- D pour aller vers la droite.

On peut combiner les directions verticales et horizontales pour faire des déplacements en diagonale.

Il y a un curseur sur la palette d'arts qui nous indique quel art nous allons utiliser si nous appuyons sur la touche de confirmation (la touche E).

Les flèches directionnelles (droite et gauche) nous permettent de déplacer le curseur de la palette d'arts.

Les flèches directionnelles (haut et bas) nous permettent de changer la cible des arts de soutien à un membre de l'équipe.

Les touches I et P nous permettent de changer de cible.

Les touches & et " nous permettent de donner des ordres à nos alliés (changer de cible (avec la touche R), etc) ou de prendre le contrôle du personnage (avec la touche E).

La touche & est pour notre allié de gauche, et la touche " est pour notre allié de droite.

Par exemple : En restant appuyé sur & et en appuyant sur R nous donnons l'ordre à notre allié de gauche d'attaquer l'ennemi que nous sommes en train de cibler.

5 Réalisation

5.1 Menu

Le menu fonctionne exclusivement via la SDL. On crée des `SDL_Rect` pour chaque bouton et leurs coordonnées sont calculées afin de les centrer. On utilise une variable `nb_choix` qui contient le numéro du choix actuel de l'utilisateur et on utilise un switch quand on valide.

`nb_choix` varie en fonction de l'appui des touches grâce à la variable `state` qui contient le statut du clavier. En fonction de `nb_choix` et de la validation on appelle les fonctions adéquates telles que `fonctionJeu()` ou `fonctionOption()`.

Dans le menu les rectangles et les textes sont contenus au même endroit dans un tableau de structure nommé `affichage` qui contient un rectangle et un texte. Pour afficher le menu on parcourt ce tableau pour afficher les rectangles un par un et les coordonnées du rectangle de choix (le rectangle rouge) sont calculées en fonction de `nb_choix`.

5.2 Lobby

Les différentes interfaces du lobby sont gérées avec SDL et les actions du joueur se font au clavier grâce à `SDL_Event`. Chaque étape possède sa

fonction d’affichage.

5.2.1 Trie

Le tableau inventaire (fourni en paramètres de la fonction du tri de l’inventaire) contient tous les objets existants dans le jeu. Chaque objet possède un nom, une valeur en argent et en score ainsi que son effectif dans l’inventaire. La fonction va afficher les objets présents dans l’inventaire du joueur un par un, en ignorant ceux dont l’effectif est 0. Le joueur peut décider d’échanger l’objet contre du score en appuyant sur la touche 1 ou de le vendre en appuyant sur la touche 3. L’effectif de l’objet diminue donc et on incrémente la valeur d’argent ou de score du joueur. S’il décide de garder l’objet en appuyant sur la touche 2, l’effectif affiché dans la fonction diminue mais celui enregistré dans le tableau ne change pas (pour qu’il reste présent dans l’inventaire). La fonction boucle tant que l’utilisateur n’a pas effectué d’action pour chaque objet. On teste la valeur de state à chaque itération de la boucle en attendant un évènement au clavier.

5.2.2 Choix de carte

Les deux images nécessaires à l’affichage (le drapeau et la carte) sont chargées dans des `SDL_Surface`. La carte est affichée en boucle avec les drapeaux dispersés dessus. A chaque itération, on teste la valeur de state pour vérifier les actions de l’utilisateur. Une variable correspond au drapeau sélectionné par le joueur. Elle s’incrémente ou se décrémente quand l’utilisateur appuie sur les flèches directionnelles du clavier. On affiche le type de carte à coté du drapeau sélectionné. L’utilisateur peut valider son choix en appuyant sur entrée. La fonction retourne alors un entier correspondant au type de carte sélectionné.

5.2.3 Commerce

Dans cette fonction, on utilise un second inventaire pour le marchand en plus de celui du joueur. La fonction boucle tant que l’utilisateur ne quitte pas en appuyant sur echap. A chaque itération, on appelle la fonction d’affichage de l’inventaire pour le marchand et pour le joueur. Elle prend en paramètres un tableau d’objets, la fenêtre SDL, ainsi que les coordonnées et la taille du `SDL_Rect` dans lequel on veut afficher l’inventaire. On affiche des cases blanches dans cet emplacement. On parcourt ensuite l’inventaire passé en paramètre et on affiche les objets présents dedans dans les cases blanches en ignorant les objets dont l’effectif est 0. On affiche ainsi leur nom, leur prix et leur effectif. De retour dans la fonction commerce, une image de curseur a été

chargée dans un `SDL_Rect`. Celui-ci s’affiche dans l’inventaire du marchand. Une variable correspond à sa position dans l’inventaire et l’utilisateur la fait varier en utilisant les flèches du clavier (on vérifie les événements du clavier à chaque itération de la boucle). Quand elle varie, on l’incrémente ou la décrémenté tant qu’elle correspond à un objet dont l’effectif est 0, pour ne pas que l’indice corresponde à un objet non présent dans l’inventaire. S’il a assez d’argent, l’utilisateur peut appuyer sur entrée pour acheter un objet. L’effectif dans l’inventaire du marchand diminue donc, l’objet apparaît dans l’inventaire du joueur s’il n’était pas présent (sinon son effectif augmente), et l’argent du joueur diminue du prix de l’objet. Si c’était la dernière itération de l’objet dans l’inventaire du marchand, la valeur de l’indice est incrémentée pour réafficher l’inventaire du marchand sans case vide.

5.3 Carte

La carte est un pointeur sur `mapt_t` une structure dynamiquement allouée qui contient un type de carte et une matrice de cases `case_t` qui elle-même contient un type, des coordonnées ainsi que des valeurs utiles.

5.3.1 Génération

Pour générer la carte, on appelle la fonction de création de carte avec le type de carte en paramètre, cette fonction elle-même appelle d’autres fonctions pour générer complètement la carte.

On commence par générer case par case la carte en ajoutant les valeurs utiles et en les stockant dans la matrice de la carte.

Dans les valeurs utiles, il y a le type de case qui lui est généré aléatoirement en regardant le type des voisins de la carte ainsi que le type de la carte en établissant des probabilités pour chaque type.

À la fin, on génère une case de début (généralement la première case de la matrice) ainsi que la case de fin aléatoirement (chaque case a la même chance d’être la case de fin que les autres).

Pour afficher la carte on parcourt case par case la matrice de la carte et on affiche via la SDL une image en fonction du type de la case dont les coordonnées sont calculées en fonction de leurs coordonnées dans la matrice.

5.3.2 Pathfinding

Le pathfinding fonctionne grâce à la valeur utile nommée `path` dans les cases ainsi que grâce à une file. On utilise des fonctions qui nous renvoient les cases de début et de fin et on cherche à aller du début à la fin. En partant

du début, on regarde la case sur laquelle on est et on regarde les voisins de cette case, si un voisin est valide (que c'est une case dans la matrice et que ce n'est pas une case d'eau) et que sa valeur de `path` est égale à zéro, on l'ajoute à la file et sa valeur de `path` devient égale à la valeur de la case dont on regarde les voisins plus un. On fait ça pour chaque voisin puis on enlève une case de la file et on recommence avec celle-ci, jusqu'à la fin (ou jusqu'à ce que la file soit vide). Au final, la valeur de la case de fin est égale à soit la distance début-fin ou alors à zéro (indiquant alors qu'il n'y a pas de chemin possible).

5.3.3 Utilisation

La fonction d'affichage du personnage prend en paramètres le sprite du personnage chargé dans un `SDL_Surface`, ses coordonnées en x et en y, la caméra (un `SDL_Rect`), et la fenêtre SDL. Elle affiche le personnage aux coordonnées voulues en fonction de la position de la caméra. Le déplacement du personnage va s'effectuer à l'aide du clic de la souris. Quand on clique sur la carte, la caméra va se déplacer sur la position de notre souris. La fonction centrage va alors comparer les distances entre chaque case de la carte et la caméra grâce à leurs coordonnées et retourner la case la plus proche de là où l'utilisateur a cliqué. Le personnage et la caméra sont alors positionnés sur cette case.

5.4 Combat

Cette partie du programme est divisée en plusieurs fichiers `.c` et `.h`. Dans les `.h` on définit une structure et les primitives qui lui correspondent. Dans les `.c` on écrit le corps des primitives définies dans le `.h`.

5.4.1 Structures

On dispose de plusieurs structures pour organiser le combat :

- `Personnage`
- `Ennemi`
- `Art`
- `degatsTxt`

Ainsi que de plusieurs tableaux et une matrice :

- `ennPool` : contient tout les ennemis que l'on peut rencontrer
- `ennemis` : contient tout les ennemis contre lesquels on est en train de se battre
- `equipe` : contient les trois personnages de l'équipe

- **artJeu** : contient les huit arts qu'utilisent chacun des personnages (donc 24 arts)
- **dgtsTxt** : contient tous les textes informatifs (dégâts, soin, esquive, coup critique, etc) durant le combat

Quand on entre en combat on va prendre un ennemi dans **ennPool** et le copier dans **ennemis**.

Les personnages possèdent un certain nombre d'arts (9 chacun pour le moment) mais ils peuvent en utiliser seulement 8 différents durant un combat. Ces arts utilisables en combat représentent la palette d'arts. Les arts sont enregistrés dans des fichiers que l'on charge au début du programme et que l'on affecte à chaque personnage. **artJeu** est une matrice de pointeurs sur art en [3][8], cette matrice contient les adresses mémoires de la palette d'arts des trois personnages de l'équipe.

dgtsTxt est un tableau de **degatsTxt** (une structure qui permet d'afficher du texte à l'écran). On retient le nombre de textes affichés avec une variable **nbDgtTxt**. Quand on inflige des blessures, et quand on se soigne on appelle une fonction qui ajoute un texte à afficher (cette fonction modifie les valeurs de **dgtsTxt[nbDgtTxt]**, puis incrémente **nbDgtTxt**). La structure **degatsTxt** possède une variable "vie" qui est décrémentée quand on affiche le texte à l'écran, quand cette variable devient négative on retire le texte de notre tableau, puis on décrémente **nbDgtTxt**.

5.4.2 Organisation

Le système de combat est organisé comme ci-dessous :

- Détection des événements clavier à l'aide de SDL.
- Utilisation de différentes fonctions qui gèrent le combat (auto-attaque, utilisation des arts, ordre que l'on donne à ses alliés, etc).
- Déplacement des personnages.
- Affichage de tous les éléments.
- Destruction des éléments temporaires.
- (Si en coop) Réception des messages et interprétations de ceux-ci.

La vitesse du jeu est basée sur le framerate (nombre d'images par seconde (IPS ou FPS en anglais)). Pour garder une vitesse constante on limite le nombre d'itérations de notre boucle principale en fonction du temps (pas plus de 60 itérations par seconde pour notre programme). Pour réaliser cela, on mesure le temps au début d'une itération et à la fin de celle-ci. On fait la différence entre ces deux valeurs pour avoir le temps qu'a mis le programme pour réaliser les actions qu'on lui demandait. Tant que cette différence est inférieure à 1000/IPS millisecondes on attend.

5.4.3 Réseau

Le jeu peut se jouer en solo ou à trois joueurs. Quand on joue en coop, on définit un serveur et deux clients. Quand le personnage que le serveur contrôle se déplace, lance une auto-attaque, utilise un art, etc, on envoie aux deux clients le résultat de ses actions (la position du personnage quand celui-ci s'est déplacé, les dégâts infligés par une auto-attaque, si elle a été esquivée, bloquée, si c'était un coup critique, etc), et les clients mettent à jour leur programme. Si c'est un client qui réalise une action alors aucun calcul n'est fait, on envoie au serveur une requête pour qu'il fasse le calcul, puis il renvoie le résultat aux deux clients. Un des problèmes rencontré lors de l'implémentation du réseau était que le système de combat est en temps réel et que la fonction qui nous permettait de recevoir les messages des autres machines est une fonction bloquante.

```
int recv(int s, void *buf, int len, unsigned int flags);
```

Nous avons donc utilisé la bibliothèque "libpthread" afin de pouvoir utiliser des thread supplémentaires.

Quand on définit le serveur, on attend que les deux clients se connectent, puis le serveur affecte un thread d'écoute à ses deux clients, et les deux clients affectent un thread d'écoute au serveur. Le programme se déroule normalement et quand on reçoit un message sur le thread d'écoute, on effectue les actions correspondantes.

Les messages envoyés suivent une syntaxe pour être lus. Le premier caractère correspond à l'action à réaliser, les caractères suivants sont des paramètres séparés par des points-virgules. Par exemple : "a;0;5" (demande du client au serveur pour savoir si on peut utiliser l'art pointé par `artJeu[0][5]`)

6 Résultats

En ce qui concerne ce projet, le planning prévisionnel n'a pas été respecté, le jeu fonctionne mais n'est pas réellement jouable.

6.1 Améliorations

Nous pensons que des améliorations sont possibles, comme l'histoire du jeu, les personnages non-joueurs, l'utilisation de `SDL_Texture` au lieu de `SDL_Surface` et les détails tels que le déplacement sur la carte qui est pour le moment infini.

- Finir les liens entre les modules pour rendre le jeu jouable.

- Sauvegarder l'état de l'expédition pour pouvoir la charger plus tard.
- Améliorer la façon dont se connecte le serveur et les clients (Demander à l'utilisateur l'adresse IP du serveur quand on est un client).

6.1.1 Menu

Les améliorations possibles pour le menu sont :

- Amélioration des graphismes du menu (des vraies textures à la place des rectangles de couleur)
- Ajout d'options supplémentaires (par exemple : la taille de la fenêtre)
- Ajout des crédits du jeu

6.1.2 Lobby

Les améliorations possibles pour le lobby sont :

- Ajout d'un système de quête, un objectif bonus lors de l'expédition.
- Ajout de la possibilité de recruter un allié avant de partir en expédition.

6.1.3 Carte

Les améliorations possibles pour la carte sont :

- Ajout de la gestion d'objets sur la carte
- Ajout de la gestion des déplacements sur la carte
- Ajout de cases spéciales pouvant être explorées.

6.1.4 Système de combat

Les améliorations possibles pour le système de combat sont :

- Au niveau de la réalisation :
 - Les structures Ennemi et Personnage ainsi que les fonctions qu'utilisent ces structures se ressemblent beaucoup (ex : typeCoupPerso et typeCoupEnnemi (qui détermine si une auto-attaque est un coup critique, si elle est esquivée, bloquée, etc)), on pourrait fusionner toutes les fonctions qui se ressemblent.
 - Régler certains bugs (par exemple : de temps en temps les dégâts infligés par certains arts ne sont pas ceux attendus)
- Au niveau de la conception :
 - Ajouter différents effets en fonction de l'environnement :
 - case d'eau (peu profonde, profonde), sable, pierre, terre, forêt, feu,

boue, buisson, neige, glace, trou, obstacle, etc

- météo (pluie, neige, tempête, foudre, éruption, tremblement de terre)
- Ajouter la tension des personnages : Attitude d'un personnage en fonction de comment se passe le combat (exemple : découragé, terrifié, confiant, etc)
- Ajouter le niveau de relation entre les personnages : Attitude d'un personnage par rapport à un autre (peut modifier des statistiques durant le combat, refuse d'écouter certains ordres, etc)
- Ajouter les traits de caractère : Réaction d'un personnage par rapport à une situation (la mort d'un ennemi, d'un allié, recevoir des ordres, etc)
- Utiliser des fichiers pour charger des groupes d'ennemis prédéfinis.
- Pouvoir changer la palette d'arts en dehors des combats.
- Pouvoir donner plus d'ordres à nos alliés (battre en retraite, se placer devant, à droite, à gauche ou derrière un ennemi).

7 Conclusion

Nous avons pu implémenter les principaux modules du programme pendant la conduite de projet. Cependant, nous aurions aimé avoir plus de temps pour finir le jeu.

Ce projet nous a permis de travailler en groupe sur un sujet qui nous plaisait tous.

C'était très enrichissant au niveau du code grâce à la découverte et l'utilisation de la SDL et du réseau en C. Nous sommes très heureux d'avoir pu travailler dessus.

8 Annexe

```
valgrind --leak-check=full ./incroyable.c
==20181== 35,392 (1,152 direct, 34,240 indirect) bytes in 12 blocks are definitely lost in
loss record 169 of 184
==20181==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==20181==    by 0x4EBF110: SDL_calloc_REAL (SDL_malloc.c:5344)
==20181==    by 0x4F0F40E: SDL_CreateRGBSurfaceWithFormat_REAL (SDL_surface.c:75)
==20181==    by 0x4F0F40E: SDL_CreateRGBSurface_REAL (SDL_surface.c:166)
==20181==    by 0x53EE671: TTF_RenderUTF8_Solid (SDL_ttf.c:1309)
==20181==    by 0x53EE9C9: TTF_RenderText_Solid (SDL_ttf.c:1275)
==20181==    by 0x111C5C: hudEnnemi (ennemi.c:376)
==20181==    by 0x10C592: main (incroyable.c:1342)
```

Une SDL_Surface n'était pas libérée après son utilisation.

```
valgrind --leak-check=full ./izi.c
==22832== 8 bytes in 1 blocks are definitely lost in loss record 5 of 1,299
==22832==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==22832==    by 0x109F98: main (izi.c:80)
```

Un pointeur n'était pas libéré après son utilisation.

Références

- [1] Wikipédia Dungeon Crawler
- [2] Lien du GitHub du Projet