

Le Mans Université
Licence Informatique *2ème année*
Module 174UP02 Rapport de Projet
The Last Nightmare

Enzo Moreau
Lina Pasquier
Emma Rasson

19 avril 2024

Table des matières

1	Introduction	3
2	Organisation	3
3	Conception	4
3.1	Règles du jeu	4
3.1.1	Règle de l'exploration	4
3.1.2	Système de cauchemar	5
3.2	Fonctionnalités	5
3.2.1	Combat	5
3.2.2	Menu	7
3.3	Aspect <i>rogue-lite</i>	8
4	Développement	8
4.1	Structure générale du programme	8
4.1.1	Arborescence	8
4.1.2	Structures	9
4.1.3	Génération de la carte	9
4.1.4	Chargements des <i>layouts</i>	11
4.2	Modules	12
4.2.1	Sauvegarde	12
4.2.2	Gestion dynamique des textures	13
4.2.3	Gestion du <i>frame-rate</i>	13
4.3	Évènements et affichage	13
4.3.1	Déplacements du personnage	14
4.3.2	Combat	14
4.3.3	Menus	16
4.3.4	Dialogues	17
5	Conclusion	18
5.1	Résultats	18
5.2	Bilan	18
5.3	Projection	18
6	Annexes	19

1 Introduction

Dans le cadre de notre module de gestion de projet de fin de deuxième année de licence nous avons eu l'opportunité de travailler sur un projet en développant notre propre jeu vidéo. Notre jeu est nommé «*The Last Nightmare*» il est d'un genre hybride entre le RPG (*role playing game*), par exemple : «*Pokemon*» et le *rogue-lite*, par exemple : «*Rogue Legacy*». Notre jeu s'inspire des premiers RPG avec : sa perspective 2D vue du dessus, ses combats au tour par tour et pour les statistiques des personnages. Tandis que le côté *rogue-lite* se retranscrit dans la gestion de la mort car après celle ci on doit recommencer une partie depuis le début mais les personnages deviennent plus fort et peuvent s'équiper de reliques aux puissants bonus.

2 Organisation

Nous nous sommes organisés en fonction de nos inclinations pour des concepts de programmation ainsi que nos compétences alors la répartition des tâches s'est faite naturellement.

Enzo s'est occupé de diviser le corps du programme en plusieurs parties :

- L'initialisation, là où les variables d'environnement sont créées et où la mémoire de ces variables est allouée.
- La boucle principale, divisée en deux parties, la première où les événements sont gérés et la deuxième qui s'occupe de l'affichage du personnage, des menus, des objets et cetera.
- La fin du programme, où les variables d'environnement sont détruites pour libérer l'espace alloué.

De plus, il était chargé du déplacement du personnage, l'animation lors du déplacement et la gestion des collisions. Entre autre il a implémenté la plus part des fonctions dans le programme principal ainsi que la sauvegarde et le chargement des données liées au programme. Il était responsable de l'organisation.

Lina s'est occupée de créer les menus du jeu et des interactions sur ces menus, comme le menu d'option avec les changement de commandes. Elle était aussi en charge d'habiller ces menus de textures. Elle a également implémenté l'affichage des dialogues dans le jeu.

Lina a, en outre, développé le système de combat du jeu, de son *gameplay* (système de jeu) à son interface. Elle était responsable du *gameplay*.

Quant à Emma, elle s'est occupée d'implémenter la structure de la carte du jeu ainsi que de sa génération. Elle a aussi réalisé les *sprites* (images d'un personnage du jeu), les textures qui habillent le fond du jeu et les objets. Elle a implémenté la gestion des textures SDL dans le jeu. Elle était responsable de la direction artistique.

3 Conception

Une fois que l'idée du jeu a germé, il a fallu concevoir les différentes étapes de celui-ci. Au cours du développement de nouvelles idées sont apparues mais les bases de la conception du jeu n'ont pas changé.

3.1 Règles du jeu

La première étape a été de poser les règles générales du jeu.

3.1.1 Règle de l'exploration

A chaque nouvelle partie, la carte globale de jeu est générée aléatoirement via un système de génération procédurale. Non seulement des différentes zones de la carte mais également des *layouts*, c'est à dire, de l'organisation même des cartes à l'intérieur de la carte globale (*map*). Ces *layouts* sont écrits à la main, pour un total d'environ cinquante documents texte, le choix est quant à lui aléatoire en respectant certaines règles imposées. Cela fonctionne pareil pour les *layouts* des objets.

Par défaut, le joueur a accès à la zone de la forêt, celle de la plage ainsi que le chemin menant au manoir. Cependant, il n'a pas accès à la partie sous-marine de la zone de la plage, l'édifice de la zone du manoir ou la zone de la grotte. Ces différents accès seront débloqués via l'acquisition d'objets : une combinaison de plongée pour la zone sous-marine, une clé pour le manoir et un talisman pour la grotte.

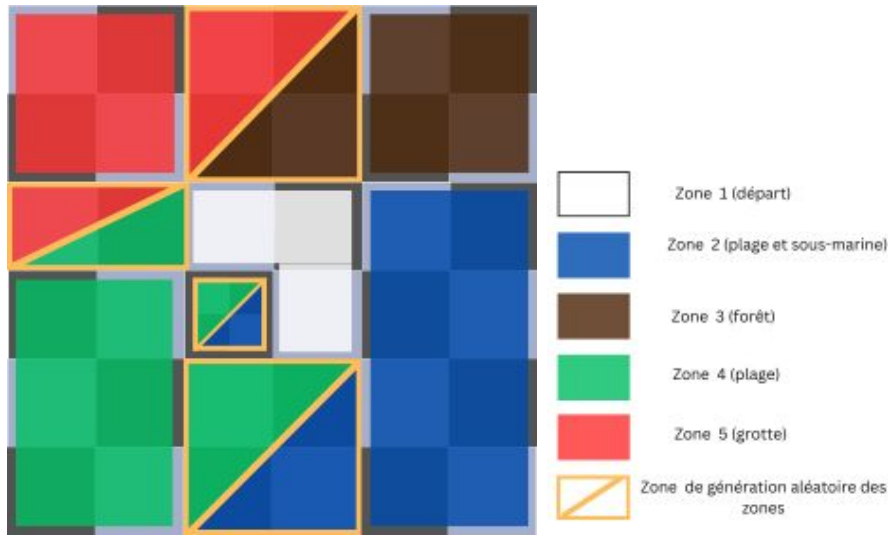


FIGURE 1 – Génération procédurale de la carte globale dans le cas général

3.1.2 Système de cauchemar

Le cauchemar est une mécanique de jeu qui est indiquée par une barre de type barre de vie qui progresse lors d'un combat. Si la barre atteint le maximum, le joueur passe en mode cauchemar. Au niveau graphique, cela est représenté par un passage des textures de base en mode cauchemar, dont les couleurs tournent autour du violet dans différentes teintes.

A chaque fin de combat en mode cauchemar, le niveau de la zone augmente d'un. Cela implique que les ennemis deviennent de plus en plus difficiles à battre. Tandis que le joueur gagne seulement la moitié d'un niveau.

La partie se termine alors de plus en plus vite car au bout d'un moment, les ennemis sont bien plus forts que le joueur.

3.2 Fonctionnalités

La deuxième étape est de développer les différentes fonctionnalités pour appliquer les règles du jeu au joueur.

3.2.1 Combat

1. Système de multiplicateur

Dans le système de combat, il y a la possibilité d'augmenter la

puissance de ses attaques ou de ses soins avec un multiplicateur qui ne dure que le temps de l'attaque ou du soin. Les joueurs ont un nombre de points au début du combat et à chaque tour, ils gagnent un point. Ces points servent à augmenter le multiplicateur, pour un point utilisé le multiplicateur augmente de 0.5. Le nombre de points utilisés ne peut pas dépasser trois (hormis en cas d'utilisation d'un artefact). Ce système de multiplicateur et de points permet d'ajouter une difficulté en plus dans le jeu. Le programme regarde si le joueur a des points et s'il ne dépasse pas les trois points. S'il en utilise la puissance de l'attaque ou du soin augmente.

2. Attaques spéciales

Tous les personnages ainsi que les ennemis ont une attaque spéciale. Cette attaque spéciale a un temps de recharge avant d'être utilisée. Il existe quatre types d'attaques spéciales différentes : un soin, une puissante attaque sur un ennemi, une attaque qui touche tous les ennemis et un malus qui fait passer le tour du combattant choisi. Le programme doit vérifier si les personnages et les ennemis peuvent lancer leur attaque spéciale ou non.

TABLE 1 – Fiche des attaques spéciales des personnages

Personnages	Attaque spéciale	Temps de recharge	Dégat/Soin
Alex	Attaque puissante sur une cible	2 tours	Attaque de base *3
Lou	Empêche la cible de jouer pendant un tour	3 tours	/
Finn	Soigne un allié	3 tours	+ 20% pv max de la cible
Ada	Attaque puissante sur toutes les cibles	2 tours	Attaque de base *2

3. Gestion du tour par tour

Le combat se joue en tour par tour, les ennemis et les personnages jouent chacun à leur tour. Ils jouent en fonction de leur vitesse. Si quelqu'un a la plus haute vitesse, il va jouer en premier. Le programme trie tous les combattants dans le combat par rapport à leur vitesse puis les fait jouer un par un.

3.2.2 Menu

1. Menu principal

Le menu, s'affiche lorsque l'on lance le jeu, il permet de faire plusieurs choses : il est possible de quitter le jeu, d'aller dans les options, de continuer sa partie et de faire une nouvelle partie. Il y a aussi le menu dans le jeu qui s'ouvre lorsque l'on appuie sur la touche «echap». Depuis ce menu, nous pouvons : accéder aux options, continuer le jeu et quitter le jeu. Pendant ce menu le jeu est mis en pause.

2. Menu des options

Le menu des options permet de changer les touches pour jouer. Il y a trois types de touche de déplacement : les flèches directionnelles, les touches d'un clavier azerty (zqsd) et les touches d'un clavier qwerty (wasd). Depuis ce menu, il est possible de faire retour pour retourner au menu précédent.

3. Inventaire

L'inventaire est accessible à partir du menu *game over*. Dans l'inventaire, se trouvent les artefacts ceux grisés sont ceux que le joueur n'a pas encore acheté dans le magasin. Il y a aussi l'emplacement des artefacts que le joueur a équipé, il y a maximum 4 emplacements d'artefact. Au début d'une partie le joueur n'a qu'un seul emplacement d'artefact, pour en avoir plus il faut les acheter dans le magasin. En cliquant sur un artefact que le joueur possède, il l'équipe et l'artefact s'affiche dans son emplacement. Pour le déséquiper, il suffit de cliquer sur l'artefact qui est dans son emplacement et il sera déséquipé. Pour quitter l'inventaire il faut cliquer sur le bouton retour et le joueur sera de retour dans le menu *game over*.

Le seul moyen d'aller dans le magasin est de perdre la partie, à ce moment-là, le magasin est proposé et permet d'acheter les artefacts. Les artefacts servent à aider lors des combats, chacun à un effet différent. Dans le magasin, le joueur peut voir l'argent qu'il a récupéré et peut le dépenser en achetant des artefacts ou des emplacements d'artefacts. Pour acheter un artefact, il faut cliquer dessus et celui-ci sera acheté, et se grisera pour signifier qu'il est acheté. Il est possible de n'acheter qu'une fois chaque artefact. Pour quitter le magasin, il suffit de cliquer sur le bouton retour et le joueur sera renvoyé à l'écran de *game over*.

Pour l'écran *game over*, il s'affiche lorsque le joueur a perdu, c'est-à-dire lorsque le combat est fini et que le personnage joué est mort. Depuis ce menu, il est possible d'accéder au magasin en cliquant dessus, de quitter le jeu et de continuer le jeu en relançant une partie.

3.3 Aspect *rogue-lite*

Le jeu prend le genre du *rogue-lite*, il en hérite une grande importance de la mort dans le *gameplay*.

Lorsque Le joueur meurt la partie prend fin, c'est le *game over*, pour autant le jeu ne prend pas fin. Une nouvelle partie commence avec un survivant de l'équipe de la partie précédente. D'autre choses sont également héritées de la partie précédente : l'argent, le niveau de l'équipe et les reliques. Ainsi le jeu devient plus simple mort après mort avec l'expérience de jeu acquise par le joueur et le niveau des personnages qui augmente.

4 Développement

Dans cette partie il sera vu comment les concepts et idées de «*The Last Nightmare*» ont été développé puis implémenté dans le code.

4.1 Structure générale du programme

Lors du développement du jeu, le programme a été divisé en plusieurs parties où chacune est structurée et documentée pour faciliter leurs usages dans les autres segments du programme.

4.1.1 Arborescence

Les fichiers sont tous répertoriés dans plusieurs dossiers : «src» pour les fichiers sources, «libs» pour les fichiers *headers*, «*sprite*» pour les textures des personnages (souvent des png) et cetera. La compilation de ces fichiers est gérée par un *makefile* qui va créer un dossier pour les fichiers objets en les nommant à partir des noms des fichiers sources via la fonction «*wildcards*» puis créer l'exécutable dans le dossier «bin» en le compilant avec *GNU*.

▼	libs	
	combat.h	
	commun.h	
	map.h	
	menu.h	
	musique.h	
	Obj.h	
	Pmov.h	
	printmg.h	
	save.h	
	texte.h	
▼	src	
	combat.c	Programme qui s'occupe du combat
	map.c	Programme s'occupant de la création, l'initialisation et la gestion de la carte de jeu
	menu.c	Programme pour afficher tous les menus
	musique.c	Programme qui gere les musiques du jeu
	Obj.c	Programme qui gere les objets du jeu
	Pmov.c	Programme pour afficher les personnages
	printmg.c	Programme pour afficher les FPS et le background
	save.c	Programme pour la sauvegarde
	texte.c	Programme qui affiche un texte sur la fenetre
	TLN.c	Programme principale

FIGURE 2 – Arborescence des fichiers du projet

4.1.2 Structures

Le programme principal est divisé en quatre parties :

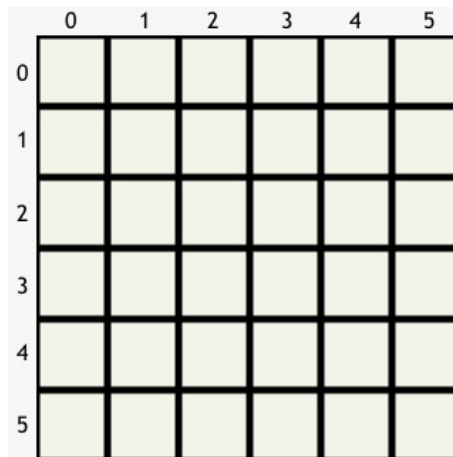
- En premier, il y a l'initialisation des valeurs, l'importation des fichiers *headers*, la création de la fenêtre *SDL*, le chargement des valeurs sauvegarder et cetera.
- En deuxième, il y a la boucle d'évènement qui fait partie de la boucle principale du jeu, tant qu'il y a des éléments dans la pile «*event*» de *SDL* les interactions du joueur avec le jeu sont traitées une à une, exemple : si la touche «*echap*» est pressée on ouvre un menu.
- En troisième, il y a l'affichage, il est dans dans la boucle principale après la boucle d'évènement. À chaque itération de la boucle principal, toutes les variables qui doivent être affichées le sont, exemple : le personnage ou les menus, cet affichage est impacté par les évènements car si la touche «*echap*» n'a pas été pressé, le menu ne sera pas affiché.
- En quatrième et dernier, il y a la destruction des variables du programme, on y fait aussi la libération des espaces alloués ainsi que la sauvegarde des données.

4.1.3 Génération de la carte

La génération de la carte est réalisée procéduralement, c'est-à-dire, qu'elle suit un algorithme pas à pas en faisant des choix à chaque étape cruciale en utilisant, dans notre cas, l'aléatoire. Pour pouvoir créer cet algorithme, il a

d'abord fallu établir les règles qui régissent celui-ci.

Le choix a été fait de réaliser des structures en cascade, c'est-à-dire, la structure *case* est présente dans celle de la grille qui est elle-même un attribut de celle de la carte qui, enfin, est présente sous forme de matrice dans la structure *map*. La structure *map* contient les informations globales qui peuvent servir à différentes échelles du programme.



	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

FIGURE 3 – Schéma de la matrice de la *map* avec coordonnées

Si l'on se penche maintenant sur la structuration de la *map*, il a d'abord fallu réaliser un croquis à la main pour identifier les différents cas et règles pour l'algorithme.

Tout d'abord, la zone de *spawn* (zone de départ) est placée dans une des quatre cases du centre de la matrice. Ce choix est réalisé aléatoirement grâce à la fonction «rand». Il a été défini que les quatre autres zones en dehors de la zone de départ seraient chacune dans les quatre coins de la matrice (sauf dans le cas particulier). L'organisation est donc :

- Zone départ (zone une) : une case au centre de la matrice
- Zone plage (zone deux) : en bas à droite de la matrice
- Zone manoir (zone trois) : en haut à droite de la matrice
- Zone forêt (zone quatre) : en bas à gauche de la matrice
- Zone grotte (zone cinq) : en haut à gauche de la matrice

À partir de la zone de départ, le joueur a accès à la zone deux à l'est, la zone trois au nord et la zone quatre à l'ouest. La zone cinq est accessible par la zone quatre quand le talisman est trouvé. Les sous-zones dans les zones deux et trois sont accessibles par leurs zones respectives en ayant trouvé les objets associés.

En résumé, il y a donc deux cas :

1. Le cas général où les points d'apparition(*spawn*) du personnage se situe en [2][2], [2][3] et [3][3] (cf Figure 3). Dans ce cas, la zone du manoir (zone quatre) est située dans le coin haut droit de la matrice.
2. Le cas particulier où le *spawn* est en [3][2] (cf Figure 3), la zone du manoir est décalé de un par rapport aux bordures en haut à droite. Dans ce nouvel espace vide il y a un couloir de plage de la zone deux (cf Figure 4 dans Annexes).

Les zones de génération aléatoires sont ensuite quasiment les mêmes en fonction de ce qui a déjà été généré (cf Figure 1). Les choix pour les zones de génération s'effectuent de la même façon que le choix pour la zone de départ. Les affectations de zones sont, elles, faites à la main dans chaque cas.

4.1.4 Chargements des *layouts*

Pour charger les *layouts*, il faut dans un premier temps avoir effectué la génération des zones. De même que pour cette génération, celle des *layouts* est elle aussi procédurale. Elle est décidé en fonction de celle de la carte tout en respectant ses propres règles. Une fois encore il y a des choix aléatoires dans certains cas.

Les *layouts* sont écrits à la main et sont stockés dans le répertoire du même nom. Pour pouvoir les charger, la fonction «*load_layout*» lis dans un fichier de *layout*, les textures qui y figurent.

Dans chaque zone, on peut y retrouver des *layouts* similaires :

- Zone du boss
- Entrée vers la zone du boss
- Des cartes de transitions entre les différentes zones (surtout entre les zones et la zone de départ)
- Des cartes simples (sans transition avec d'autres zones ou d'éléments particuliers dessus)

4.2 Modules

Le programme est divisé en plusieurs modules, un pour les menus, un pour le personnage, un autre pour le combat et cetera. Certains de ces modules ou du moins une partie est d'une importance capitale dans le jeu, ceux sont derniers qui vont être abordés.

4.2.1 Sauvegarde

Le jeu possède un système de sauvegarde qui, lorsque on quitte le jeu en pleine partie, permet de la reprendre dans le même état à la prochaine session de jeu. Pour sauvegarder l'état du jeu, le module de sauvegarde écrit dans un fichier «*save.txt*» toutes les informations importantes : les «pv» du personnage et de l'équipe, la génération de la «*map*», les artefacts possédés et cetera. Lors du redémarrage du jeu, une vérification est faite pour savoir si un fichier de sauvegarde existe et si oui alors le fichier est lu et les données sont chargées dans les variables du jeu. Il y a deux cas distincts de variable où s'effectue le chargement des données : celles qui sont allouées dynamiquement et celles qui sont statiques. Dans le premier cas les données lues dans le fichier sont chargées directement dans la variable tandis que dans l'autre cas, un tableau de pointeur sur entier précédemment alloué est utilisé pour copier les valeurs dans le programme principal.

Si le joueur meurt et qu'il refait une partie, la sauvegarde se fait partiellement car seulement les données de l'aspect «*rogue-lite*» sont sauvegardées et chargées. Il est aussi possible depuis l'écran titre de lancer une nouvelle partie ce qui va supprimer le contenu du fichier de sauvegarde. Pour différencier dans quel type de sauvegarde le programme se trouve, il est écrit au

début du fichier un entier qui indique cet état : zéro si le fichier est vide, un pour une sauvegarde classique et deux pour une partie après une mort. ...

4.2.2 Gestion dynamique des textures

Le processus d'initialisation des textures passe par un fichier texte recensant tous les chemins vers les images des textures du jeu triés zone par zone. La lecture de ce fichier permet de remplir un tableau de tous les chemins.

Une partie importante du programme est la gestion efficace des textures chargées, il faut faire attention à ménager la mémoire. Dans le cas du jeu, il a été décidé de la gérer par le chargement de toutes les textures d'une zone à l'entrée dans celle-ci. Cela peut surcharger la mémoire, cependant la limitation zone par zone amoindrie cet effet. Outre cela, cette méthode a pour avantage de réduire le nombre d'occurrences de chargement dans la mémoire. Ainsi la mémoire n'est suscitée qu'au chargement d'une nouvelle zone.

4.2.3 Gestion du *frame-rate*

L'affichage du jeu est limitée, pour éviter les sauts d'image et autre bogue. Pour limiter l'affichage, le jeu est limité à 60 *FPS* (*frame per seconds*, *frame-rate* ou image par seconde). Pour ce faire, le programme capture le temps en temps zéro ($t0$) et ensuite capture le temps temps un ($t1$) jusqu'à ce que la différence $t1 - t0$ soit d'une seconde. À chaque capture de $t1$ un compteur de *frame* est incrémenté ainsi à chaque itération de la boucle principale, le suivi du *frame-rate* se fait image par image et lorsqu'une seconde s'est écoulée le compteur se réinitialise. Avec ces informations le programme va ajuster son affichage. À l'initialisation les itérations de la boucle principale du jeu sont espacées par un temps de latence de mille millisecondes divisé par un diviseur du *frame-rate* qui est initialisé à la valeur voulue de *FPS*, ici soixante. Ensuite si en une seconde les FPS voulus ne sont pas atteints on incrémente le diviseur du *frame-rate* est incrémenté. S'il dépasse il est décrémenté ainsi l'affichage s'adapte à la difficulté du programme à afficher le rendu.

4.3 Évènements et affichage

Lors d'une partie, les interactions du joueur avec l'environnement provoquent différents événements qui se retranscrivent dans l'affichage du jeu.

4.3.1 Déplacements du personnage

Les déplacements du personnage réagissent à chaque fois qu'une touche de déplacement est pressée. Au départ, le personnage est dans une case de la carte et est affiché en fonction de la direction dans laquelle le personnage regarde. Si une touche de déplacement est pressée et qu'elle correspond à une autre direction que celle du personnage alors il change de posture sans se déplacer. Autrement il se déplace dans la case adjacente à la sienne dans sa direction.

Le personnage a des limites pour son déplacement. Si la case dans laquelle il veut aller est occupée alors il ne bougera pas, si la case est en bordure de carte et qu'il va en dehors de celle-ci alors il va aller dans la carte adjacente à la sienne et si la case dans laquelle il va est en bordure de *map* alors il ne changera pas de carte.

À chaque déplacement une animation du personnage s'effectue. À partir de la nouvelle case du personnage et de sa direction, le personnage est animé depuis sa précédente case vers la suivante. L'animation se fait en quatre dixième de secondes donc en vingt quatre frames qui sont réparties en quatre images qui sont en réalité deux qui alternent. Alors le premier et le troisième quarts de l'animation sont pour le premier *sprite* de l'animation tandis que le deuxième et le quatrième quarts sont pour le deuxième *sprite*.

4.3.2 Combat

Pour la gestion du combat, les informations des personnages et des ennemis sont dans une structure combattant qui regroupe les informations importantes pendant le combat (par exemple : les points de vie, la force d'attaque des personnages etc.). Les statistiques des combattants sont multipliées par le niveau de la zone pour les ennemis et par le niveau d'équipe pour les personnages. Ces informations sont gardées dans une structure « *map* ». Chaque ennemi sur la carte a une variable pour savoir s'il est en combat ou non. Cette variable change lorsque le joueur entre en combat avec un ennemi, à ce moment-là, le joueur entre dans la fonction de combat. Le combat se termine soit par la mort de toute l'équipe de personnages, soit par la mort de tous les ennemis. Concernant l'affichage du combat, il y a une fonction qui est appelée en continu tant que le combat n'est pas terminé.

Les personnages jouent chacun leur tour grâce au tri par vitesse. Lors du tour d'un joueur, il faut vérifier si le joueur peut jouer. S'il le peut la fonction

« `attaque_allie` » est appelée. Cette fonction a une boucle « `SDL_event` », la boucle permet de changer de personnage à attaquer en cliquant sur les flèches, puis lorsque le joueur appuie sur le texte attaquer, l'ennemi sélectionné perd des points de vie en fonction de la force d'attaque du personnage et du multiplicateur. Lors du tour d'un ennemi, il faut aussi vérifier s'il peut jouer et s'il le peut la fonction « `attaque_ennemi` » est appelée. Cette fonction permet à l'ennemi d'attaquer en fonction de sa forme, il existe quatre formes d'ennemi :

- les *slimes*,
- les nécromanciens,
- les créatures inspirés du Minotaure,
- les boss.

Chacune de ses formes a une façon spécifique d'attaquer les personnages, la fonction permet aussi aux ennemis d'utiliser leur attaque spéciale dès qu'ils l'ont. À chaque fois que le tour d'un combattant est terminé, il faut vérifier s'il a encore des points de vie, lorsque qu'il n'en a plus, une variable dans sa structure est mise à un pour signifier sa mort, à ce moment-là le combattant ne peut plus jouer et ne peut plus être sélectionné.

Lors du combat, pour savoir si le joueur est en mode cauchemar, il y a une variable dans la structure « `map` » qui indique son état(zéro s'il n'est pas en mode cauchemar, un sinon). Lorsque la barre de cauchemar est au maximum pendant le combat, le joueur passe en mode cauchemar et la variable est mise à un. À ce moment, les statistiques de tous les combattants sont augmentées, étant donné que les niveaux de zone et d'équipe augmentent. Les personnages dans ce mode ont un passif qui leur est propre :

- Alex peut ressusciter une fois lorsque il est mort,
- Finn soigne un peu ses alliés à chaque tour,
- Ada peut jouer deux fois de suite tous les deux tours,
- Lou lorsqu'elle fait passer le tour d'un personnage, il perd aussi des points de vie.

Le programme vérifie à chaque tour de boucle si le joueur est en mode cauchemar, si c'est le cas les passifs des personnages sont activés et l'affichage est modifié. Dans ce cas, il y a donc un changement dans les textures. Le moyen de sortir du cauchemar est de tuer un boss, si le joueur a tué un boss la barre de cauchemar est diminuée de moitié et si on tue un boss sans avoir été en mode cauchemar la barre retourne à zéro.

4.3.3 Menus

Pour chaque menu, dans leur fonction, il y a une boucle. Tant que le joueur a rien fait, le programme reste dans cette boucle. Il y a aussi un «SDL event» qui regarde si le joueur clique sur un des boutons. Si le joueur clique sur l'un des boutons l'action associée à ce bouton sera exécutée.

Pour afficher le menu d'écran titre, la fonction est appelée avant même de rentrer dans la boucle du jeu. Pour commencer le jeu, il faut cliquer sur «continuer» et le jeu se lancera en changeant la valeur d'une variable qui permet de rentrer dans la boucle du jeu et chargera la sauvegarde du joueur. Pour lancer une nouvelle partie, il faut cliquer sur le bouton du même nom et une nouvelle partie sera créée, la sauvegarde si elle existe ne sera pas chargée et une autre sera créée. Pour quitter le jeu, il faut cliquer sur «quitter» et la variable qui permet de rentrer dans le jeu ne sera pas changée donc le joueur ne rentrera pas dans le jeu, mais quittera le menu ce qui fait quitter le jeu. Pour aller dans les options, il faut cliquer sur le bouton «options» qui appelle la fonction qui affiche la fenêtre des options.

Pour le menu dans le jeu, il s'affiche avec l'événement d'appuyer sur *echap* ou si une variable à une certaine valeur. Une fenêtre qui ressemble au menu d'écran sera affichée, il y a juste le bouton «nouvelle partie» qui n'y est pas. Les boutons marchent exactement pareils que pour le menu d'écran titre.

Pour les options, un menu s'affiche qui permet de changer avec quelle touche se déplacer, pour cela la fonction a une variable qui change par rapport à ce que le joueur choisi comme touches de déplacement. Pour quitter le menu d'options, le joueur doit cliquer sur «retour» et retourne sur le menu dans lequel il était. Pour retourner à l'écran titre, la fonction rappelle la fonction qui affiche l'écran titre, alors que pour retourner au menu, une variable est mise à la bonne valeur pour ré-afficher le menu sans avoir ré-appuyer sur *echap*.

Pour l'écran *game over*, la fonction regarde si le personnage joué est mort et s'il l'est l'écran s'affiche. Depuis ce menu, nous pouvons accéder au magasin, la fonction appelle directement la fonction d'affichage du magasin pour y accéder. Il est possible de quitter le jeu et de relancer une partie.

Pour le magasin, tous les artefacts s'affichent et sont cliquables pour les acheter. Pour avoir les informations de tous les artefacts il y a une structure artefacts qui garde les informations importantes de chaque artefact et

dans la structure *map* il y a un tableau de structure artefacts qui a tous les artefacts qui existent et une variable pour savoir le nombre d'argent que le joueur possède. Pour ne pas acheter un artefact que le joueur a déjà il y a une variable dans la structure artefact qui dit si le joueur le possède déjà. Si c'est le cas l'artefact est grisé pour montrer que le joueur la possède déjà, sinon la variable est mise à un pour dire que le joueur vient de l'acheter et l'argent que le joueur possède est diminué du prix de l'artefact. La fonction marche de la même manière pour les emplacements à acheter. Pour quitter le magasin il suffit de cliquer sur le bouton «retour» et la variable qui met le jeu en pause reprend. Cependant comme le joueur est mort, l'écran *game over* revient automatiquement.

Pour le menu inventaire, tous les artefacts s'affichent, mais ceux que le joueur ne possède pas sont grisés et ne peuvent pas être équipés. Pour équiper un artefact, le joueur doit cliquer sur l'artefact qu'il veut mettre et la variable qui indique si l'artefact est équipé ou non se met à un. Ainsi l'artefact sera affiché dans une des cases d'emplacement d'artefacts. Pour déséquiper un artefact, il faut cliquer dessus dans son emplacement et sa variable sera mise à zéro. De plus, dans l'emplacement l'artefact ne sera plus affiché et laissera une case vide. Pour quitter l'inventaire, il faut cliquer sur le bouton «retour».

4.3.4 Dialogues

Pour lancer un dialogue, il faut appuyer sur la touche « e » à un endroit où il y a un dialogue. Concernant le lancement du dialogue, il y a une fonction qui regarde si le joueur appuie sur « e » à côté d'un endroit où il y a du dialogue et une variable « *etat_dialogue* » va se mettre à un pour dire que le joueur est en dialogue. Cette variable mise à un permet de rentrer dans la fonction « *dialogue* », cette fonction a une boucle d'état qui n'est pas quittée tant que le joueur n'a pas appuyé sur « e ». Le dialogue marche comme un menu, le jeu est mis en pause. La différence, c'est que le joueur peut encore voir le jeu derrière, pour cela, il fallait sauvegarder la dernière image du rendu pour mettre par-dessus un filtre noir. Dans le jeu, il n'y a pas beaucoup de textes de dialogue donc lorsque le joueur lance un dialogue, l'objet ou le personnage a un numéro de dialogue et avec un *switch* le programme récupère le texte associé et l'affiche. Pour quitter le dialogue, il suffit d'appuyer sur « e » est la variable d'état se mettra à zéro pour retourner au jeu.

5 Conclusion

Maintenant que le projet est arrivé à son terme, il est maintenant possible d'en tirer des conclusions.

5.1 Résultats

Le projet, malgré qu'il n'est pas entièrement suivi la ligne directive de départ, a abouti à un résultat plus pragmatique et plus en phase avec nos capacités et nos idées. L'histoire a notamment beaucoup évolué passant d'un remède à trouver à un boss final à tuer. De plus, certaines fonctionnalités ne semblaient pas réalisables, il a donc été décidé d'en ajouter des différentes à la place comme l'effet de la lumière dans la zone de la grotte ou encore des différents types des ennemis à affronter.

De nombreux tests ont été effectués sur le programme permettant au fur et à mesure de son développement d'être sûr de la stabilité du programme.

5.2 Bilan

Ce projet nous a permis de comprendre l'importance d'avoir une bonne organisation et une bonne communication au sein d'une équipe.

La complexité de se lancer dans ce genre de projet peut sembler, à première vue, décourageante. Cependant la finalité est gratifiante et très enrichissante.

Cela nous a permis de découvrir de nouvelles façons de travailler en autonomie ainsi que d'approfondir nos connaissances du langage C et d'apprendre à utiliser de nouveaux outils tels que *GitHub*, le diagramme de Gantt ainsi que la librairie SDL.

5.3 Projection

Notre projet, bien qu'abouti, peut être soumis à des améliorations dans le futur, tels que : l'optimisation des différentes fonctions, l'enrichissement de la génération procédurale ainsi que l'ajout de bruitages.

6 Annexes

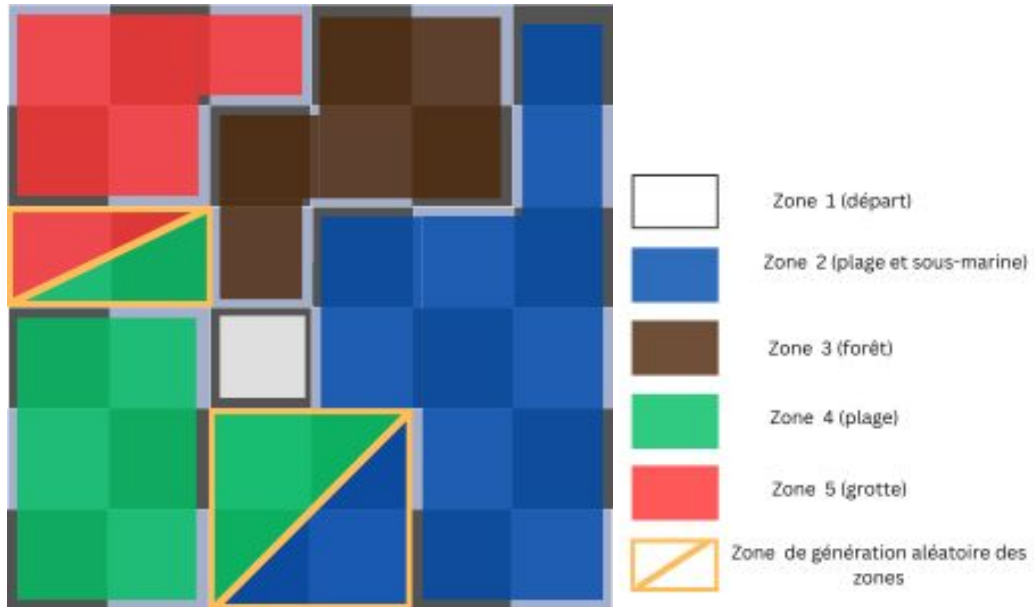


FIGURE 4 – Génération procédurale de la carte globale dans le cas général

```
s2201960@ic25114-11:~/Documents/workspace/ProjetL23 gdb ./bin/TLN
GNU gdb (Ubuntu 9.2-0ubuntu1-20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./bin/TLN...
(gdb) next
The program is not being run.
(gdb) run
Starting program: /info/etu/l2info/s2201960/Documents/workspace/ProjetL2/bin/TLN
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
[New Thread 0x7fffd788700 (LWP 4966)]
[New Thread 0x7fffd788700 (LWP 4967)]
[Thread 0x7fffd788700 (LWP 4967) exited]
[New Thread 0x7fffd788700 (LWP 4968)]
[Thread 0x7fffd788700 (LWP 4968) exited]
[New Thread 0x7fffd788700 (LWP 4970)]
[Thread 0x7fffd788700 (LWP 4970) exited]
[New Thread 0x7fffd788700 (LWP 4971)]
non
1920 1080

Thread 1 "TLN" received signal SIGSEGV, Segmentation fault.
0x0000555555557c96 in creation_map (w=<error reading variable: Cannot access memory at address 0x7ffff353c14>, h=<error reading variable: Cannot access memory at address 0x7ffff353c10>)
    at src/map2.c:56
56      map_t creation_map (int w, int h){
(gdb) quit
A debugging session is active.
```

FIGURE 5 – Déboggage de la création de la *map* de jeu

```

==18467==
==18467== HEAP SUMMARY:
==18467==    in use at exit: 12,036,138 bytes in 15,666 blocks
==18467==    total heap usage: 127,190 allocs, 111,524 frees, 52,001,581 bytes allocated
==18467==
==18467== 17 bytes in 1 blocks are definitely lost in loss record 10 of 2,212
==18467==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==18467==    by 0x4EDD826: _XlcDefaultMapModifiers (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==18467==    by 0x4EDDBFA: XSetLocaleModifiers (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==18467==    by 0x4941824: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.10.0)
==18467==    by 0x494845A: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.10.0)
==18467==    by 0x491AF6A: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.10.0)
==18467==    by 0x488A8E6: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.10.0)
==18467==    by 0x10984B: main (TLN.c:21)
==18467==
==18467== 3,012 bytes in 251 blocks are definitely lost in loss record 2,164 of 2,212
==18467==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==18467==    by 0x109722: aff_Fps (printImg.c:44)
==18467==    by 0x109C40: main (TLN.c:111)
==18467==
==18467== 4,152 bytes in 35 blocks are possibly lost in loss record 2,171 of 2,212
==18467==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==18467==    by 0x48DC447: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.10.0)
==18467==    by 0x49E2C9: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2_ttf-2.0.so.0.14.1)
==18467==    by 0x49E4A43: TTF_RenderUTF8_Blended (in /usr/lib/x86_64-linux-gnu/libSDL2_ttf-2.0.so.0.14.1)
==18467==    by 0x49E4DD8: TTF_RenderText_Blended (in /usr/lib/x86_64-linux-gnu/libSDL2_ttf-2.0.so.0.14.1)
==18467==    by 0x10975E: aff_Fps (printImg.c:47)
==18467==    by 0x109C40: main (TLN.c:111)
==18467==
==18467== 186,000 bytes in 6 blocks are possibly lost in loss record 2,204 of 2,212
==18467==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==18467==    by 0x48DC447: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.10.0)
==18467==    by 0x49E354F: TTF_OpenFontIndexRW (in /usr/lib/x86_64-linux-gnu/libSDL2_ttf-2.0.so.0.14.1)
==18467==    by 0x1096FC: aff_Fps (printImg.c:37)
==18467==    by 0x109C40: main (TLN.c:111)
==18467==
==18467== 7,764,155 (7,595,000 direct, 169,155 indirect) bytes in 245 blocks are definitely lost in loss record 2,212 of 2,212
==18467==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==18467==    by 0x48DC447: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.10.0)
==18467==    by 0x49E354F: TTF_OpenFontIndexRW (in /usr/lib/x86_64-linux-gnu/libSDL2_ttf-2.0.so.0.14.1)
==18467==    by 0x1096FC: aff_Fps (printImg.c:37)
==18467==    by 0x109C40: main (TLN.c:111)
==18467==
==18467== LEAK SUMMARY:
==18467==    definitely lost: 7,598,029 bytes in 497 blocks
==18467==    indirectly lost: 169,155 bytes in 1,422 blocks
==18467==    possibly lost: 190,152 bytes in 41 blocks
==18467==    still reachable: 4,078,802 bytes in 13,706 blocks
==18467==    suppressed: 0 bytes in 0 blocks
==18467== Reachable blocks (those to which a pointer was found) are not shown.
==18467== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==18467==
==18467== For lists of detected and suppressed errors, rerun with: -s
==18467== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 2 from 2)
s2103776@ic2s124-08:~/Documents/Projet/The_Last_Nightmare$

s2103776@ic2s124-08:~/Documents/Projet/The_Last_Nightmare$ valgrind --leak-check=full ./bin/TLN
==19219== Memcheck, a memory error detector
==19219== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==19219== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==19219== Command: ./bin/TLN
==19219==
==19219== HEAP SUMMARY:
==19219==    in use at exit: 318,254 bytes in 2,914 blocks
==19219==    total heap usage: 106,232 allocs, 103,318 frees, 39,355,457 bytes allocated
==19219==
==19219== 17 bytes in 1 blocks are definitely lost in loss record 10 of 2,166
==19219==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==19219==    by 0x4EDD826: _XlcDefaultMapModifiers (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19219==    by 0x4EDDBFA: XSetLocaleModifiers (in /usr/lib/x86_64-linux-gnu/libX11.so.6.3.0)
==19219==    by 0x4941824: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.10.0)
==19219==    by 0x494845A: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.10.0)
==19219==    by 0x491AF6A: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.10.0)
==19219==    by 0x488A8E6: ??? (in /usr/lib/x86_64-linux-gnu/libSDL2-2.0.so.0.10.0)
==19219==    by 0x109872: main (TLN.c:21)
==19219==
==19219== LEAK SUMMARY:
==19219==    definitely lost: 17 bytes in 1 blocks
==19219==    indirectly lost: 0 bytes in 0 blocks
==19219==    possibly lost: 0 bytes in 0 blocks
==19219==    still reachable: 318,237 bytes in 2,913 blocks
==19219==    suppressed: 0 bytes in 0 blocks
==19219== Reachable blocks (those to which a pointer was found) are not shown.
==19219== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==19219==
==19219== For lists of detected and suppressed errors, rerun with: -s
==19219== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 2 from 2)
s2103776@ic2s124-08:~/Documents/Projet/The_Last_Nightmare$

```

FIGURE 6 – Débogage des FPS

```

Thread 1 "TLN" received signal SIGSEGV, Segmentation fault.
__vfprintf_internal (s=0x0, format=0x55555555b25f "%c",
  argptr=argptr@entry=0x7ffffffbc2f0, mode_flags=mode_flags@entry=2)
  at vfprintf-internal.c:345
345      vfprintf-internal.c: Aucun fichier ou dossier de ce type.
(gdb) next
[Thread 0x7ffffed788700 (LWP 6403) exited]
[Thread 0x7ffffedfc9700 (LWP 6398) exited]

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.

```

FIGURE 7 – Débogage du chargement des zones de la *map*

```

1031      for (i=0;i<4;i++){
(gdb) s
1032          if(pp->equipe[i]!=NULL){
(gdb) s
1033              allie++;
(gdb) s
1031      for (i=0;i<4;i++){
(gdb) s
1032          if(pp->equipe[i]!=NULL){
(gdb) s
1033              allie++;
(gdb) s
1031      for (i=0;i<4;i++){
(gdb) s
1036          combattant_t *tabAllie[allie];
(gdb) s
1040          for(i=0;i<allie;i++){
(gdb) s
1041              copieAllie[i]=init_combattant(pp->equipe[i]->nom,pp->equipe[i]->pv,pp->equipe[i]->vitesse,pp->equipe[i]->camp,pp->equipe[i]->indice_portrait,pp->equipe[i]->indice_sprite,pp->equipe[i]->type,pp->equipe[i]->temps_recharge_max,pp->equipe[i]->puissance,pp->equipe[i]->forme);
(gdb) s
1041      init_combattant (nom=0xf0000001c <error: Cannot access memory at address 0xf0000001c>, pv=64, vitesse=64, camp=1016,
  indice_portrait=1792, indice_sprite=2, type=0, temps_recharge_max=2, puissance=15, forme=0) at src/combatt.c:62
62      combattant_t *init_combattant(char* nom,int pv,int vitesse,int camp,int indice_portrait,int indice_sprite,int type,int temps_recharge_max,int puissance,int forme){
(gdb) s
63          combattant_t * combattant=malloc(sizeof(combattant_t));
(gdb) s

```

FIGURE 8 – Débogage du chargement des *layouts* d'objet