

Le Mans Université
Licence Informatique 2ème année
Module 174UP02 Conduite de Projet
Rapport de Projet
Bloody Sanada

Ange Despert, Max Descomps, Antoine Bruneau, Rafaël Doneau

5 avril 2022

Lien de notre Github : <https://github.com/TheMisterPenguin/ProjetL2>

Table des matières

1	Introduction	4
2	Conception	4
3	Organisation	4
4	Développement	6
4.1	Module init	7
4.2	Module affichage	7
4.3	Fonctionnement du moteur d'affichage	8
4.4	Module personnage	9
4.5	Module monstres	10
4.6	Module objets et inventaire	11
4.7	Module coffres	12
4.8	Module map	13
4.9	Module listes et JSON	15
4.9.1	Listes	15
4.9.2	JSON	15
4.10	Gestion évènements	15
4.11	Main	16
5	Résultats	16
5.1	Attendus	16
5.2	Obtenus	17
6	Conclusion	17
7	Annexe	17

Table des figures

1	<i>Gantt prévisionnel</i>	5
2	<i>Structure d'affichage de texture</i>	8
3	<i>Structure personnage</i>	9
4	<i>Statut personnage</i>	10
5	<i>Structure objet</i>	12
6	<i>Structure coffre</i>	13
7	<i>La structure map</i>	14
8	<i>Les primitives de listes</i>	15
9	<i>Recherche erreur GDB</i>	18
10	<i>Erreur trouvée avec GDB</i>	18
11	<i>Inventaire personnage</i>	19
12	<i>Tableau de compétences</i>	20
13	<i>Statistiques personnage</i>	21
14	<i>Un exemple de fichier map</i>	22
15	<i>Gantt réel</i>	22

1 Introduction

Dans le cadre de notre deuxième année de licence informatique, nous réalisons un jeu vidéo programmé en C.

Le but principal est de réutiliser toutes les notions apprises au cours de nos deux années de licence, c'est-à-dire :

- créer et mettre en oeuvre des algorithmes,
- gérer un projet,
- mettre en place les outils de développement,
- présenter le résultat du travail par écrit et à l'oral,
- juger de nos capacités d'initiative,
- nous constituer une boîte à outils de fonctions utiles.

Notre jeu est de type Role Play Game (RPG) pour ce projet. Le principe de ce genre de jeux est d'affronter des ennemis avec son/ses personnage(s) afin de le faire suivre un scénario préalablement défini. La plupart de ces jeux mettent en place un système d'expérience afin de faire évoluer le(s) personnage(s) de façon claire pour le joueur.

Ce thème nous plaisant du fait de la diversité d'éléments à implémenter et à créer, nous avons rapidement choisi de nous lancer dans ce sujet.

2 Conception

Dans cette partie, nous vous expliquerons les différentes fonctionnalités générales du projet.

Comme mentionné dans l'introduction, les RPG mettent en scène un personnage se battant contre des ennemis. Notre personnage ne déroge pas à la règle et se bat contre des monstres en temps réel. Il gagne de l'expérience à chaque monstre tué, ce qui lui permet d'augmenter son niveau et ainsi améliorer ses caractéristiques. Le jeu offre la possibilité de jouer en multi-joueur localement, et chaque joueur peut augmenter les caractéristiques de son personnage en l'équipant d'objets (trouvés sur des monstres, dans des coffres ou par le biais de quêtes) et accéder à certaines zones selon les objets équipés (comme une clé). Il peut également utiliser des potions/herbes médicinales afin de se soigner et regagner de la vie. Il est possible de jouer sur Windows comme sur Linux, aussi bien avec une manette qu'avec un clavier et un souris.

3 Organisation

Concernant notre organisation au sein du groupe, nous avons mis en place un dépôt de gestion de version (Git) et nous avons réalisé un Gantt prévisionnel en répartissant les différentes tâches à effectuer permettant d'évaluer l'avancement du projet selon les délais établis pour chaque tâche (figure 1).

ID	Work Breakdown Structure	Editable User Area			Days	2022												
		Start	End	Person		De January					February				March			
						52	1	2	3	4	5	6	7	8	9	10	11	12
1	Définition des tâches et sous tâches	17/01/2022	17/01/2022	Tout le groupe	1	1												
2	Création du GIT	17/01/2022	17/01/2022	Ange	1	1												
3	Création du Trelio	17/01/2022	17/01/2022	Tout le groupe	1	1												
4	Création du GANTT	18/01/2022	19/01/2022	Rafael	2	2												
5	Description des sous tâches	18/01/2022	19/01/2022	Rafael/Max/Antoine	2	2												
6	Création du scénario	18/01/2022	18/01/2022	Tout le groupe	1	1												
7	Création du design du personnage	24/01/2022	11/02/2022	Rafael	19	15												
8	Création du design des monstres	24/01/2022	11/02/2022	Rafael	19	15												
9	Création des items	19/01/2022	17/02/2022	Max	24	18												
10	Création de la carte	19/01/2022	11/02/2022	Ange/Max/Antoine	24	18												
11	Gérer le déplacement du personnage	11/02/2022	25/02/2022		15	11												
12	Gérer le déplacement des monstres	11/02/2022	25/02/2022		15	11												
13	Gérer les sauvegardes automatiques	04/03/2022	25/03/2022		22	16												
14	Gérer la sauvegarde manuelle	04/03/2022	25/03/2022		22	16												
15	Gérer l'expérience du personnage	25/02/2022	04/03/2022	Rafael	8	6												
16	Créer les énigmes (coffres, actionneurs...)	04/03/2022	11/03/2022	Rafael	8	6												
17	Gérer les combats	25/02/2022	04/03/2022		8	6												
18	Créer une musique de fond	11/03/2022	25/03/2022	Ange	15	11												
19	Créer les sons et bruitages du jeu	11/03/2022	25/03/2022		15	11												
20	Créer la cinématique d'introduction	11/03/2022	25/03/2022		15	11												
21	Créer les différents menus (principal, pause, inventaire...)	25/02/2022	11/03/2022		15	11												
22	Gérer la résolution de l'interface	18/02/2022	04/03/2022		15	11												
23	Gérer les collisions (monstres/personnage, personnage/murs, monstres/murs)	25/02/2022	04/03/2022		8	6												
24	Créer l'interface de jeu principale	11/02/2022	18/02/2022	Ange	8	6												
25	Gérer la caméra de jeu (suivre le personnage en vue de dessus)	25/02/2022	04/03/2022		8	6												
26	Créer le makefile	17/01/2022	17/01/2022	Ange	1	1												
27	Ajouter l'inclusion de la SDL dans le makefile	17/01/2022	20/01/2022	Ange	4	4												
28	Préparer la SDL	17/01/2022	17/01/2022	Ange	1	1												
29	Tester l'affichage d'un sprite	18/01/2022	18/01/2022	Ange	1	1												
30	Ajout de cleanup au makefile	20/01/2022	20/01/2022	Ange	1	1												
31	Création de la configuration de la génération de la documentation	20/01/2022	24/01/2022	Ange	5	3												
32	Séparation en plusieurs fichiers	20/01/2022	20/01/2022	Ange	1	1												
33	Création des maquettes pour les interfaces des menus	26/01/2022	27/01/2022	Rafael	2	2												

FIGURE 1 – *Gantt prévisionnel*

Vous pourrez également retrouver dans la partie 7 notre Gantt réel (figure 15).

Au sein du projet, Ange a réalisé :

- le module affichage sauf certaines fonctions (animation personnage et interface),
- le module map,
- le module liste,
- les fonctions de sauvegarde et chargement,
- les menus,
- l'introduction,
- le makefile,
- la définition des codes d'erreur,
- la création de macro pour les erreurs et les warnings,
- le module d'initialisation et fermeture,
- les événements, en particulier tout ce qui concerne l'utilisation de la manette,
- la fonction main avec le moteur de jeu,
- la création du dépôt git,
- l'installation de la librairie JSON-c,
- la conversion vers Windows.

De son côté, Antoine s'est principalement occupé du gameplay. Il a commencé par faire le design et l'animation du personnage qu'il a mis en ap-

plication grâce à des fonctions pour gérer les événements aux claviers et à la souris. Il a réalisé le module des monstres de bout en bout, du design en passant par l'affichage pour enfin gérer leurs interactions avec la carte et le joueur. Parmi ces monstres, il y a :

- le chevalier -> attaque aux corps à corps,
- le sorcier -> lance des sorts à distance et se replie,
- le boss -> invoque des clones, se téléporte et lance des sorts.

Pour ces monstres, il a donc eu besoin de créer un module exprès pour gérer les sorts ainsi qu'un autre pour le boss qui possède des contraintes supplémentaires. Enfin, il a géré une grande partie des interactions entre le personnage et les entités, comme les dégâts, la défense ou bien encore la parade des différents sorts.

Max lui a d'abord imaginé le concept du jeu, puis réalisé quelques fonctions d'affichage, notamment d'animation. Il a ensuite créé les modules gérant les coffres et les objets avec leurs effets sur le personnage, puis créé certaines fonctions relatives aux listes génériques.

Il s'est occupé d'une grande partie du débogage et de la suppression des fuites de mémoire, notamment par la création de jeux de tests avec la bibliothèque CUnit. Il a travaillé sur l'interface joueur par :

- la création de l'inventaire (concept et programmation),
- la barre de santé,
- le menu d'accueil.

Il s'est aussi occupé de la totalité des textures des cartes du jeu à l'aide de tilesset (jeu de tuiles) ainsi que de certains menus graphiques comme l'inventaire, de quelques sprites (éléments graphiques). Il a finalement contribué au tests du code avec l'implémentation de CUnit, à la documentation du code avec Doxygen et à sa compilation avec un makefile.

Rafaël s'est occupé de la réalisation du Gantt et de ce présent rapport. Il a ensuite réalisé les différentes maquettes des interfaces des menus du jeu. Il a également mis en place les fonctions de gain d'expérience et de montée de niveau du personnage ainsi que la modification des caractéristiques de ce dernier au fil des niveaux. Enfin, il a aidé Max à la réalisation de l'inventaire et de ses différentes fonctions.

4 Développement

Dans cette partie, nous expliquerons les différents modules que nous avons créés pour mener à bien ce projet. Pour des raisons de clarté et de flexibilité, les informations concernant les monstres, les cartes ainsi que les coffres sont stockés dans des fichiers de données structurés au format JSON. Ces fichiers sont lus une fois au lancement du jeu et insèrent leur contenu dans les structures correspondantes, créant les cartes et les différents modèles de monstres et de coffres implémentables.

Le fichier source de création de carte contient la position des monstres, coffres et leur personnalisation, qui sont enregistrés dans des listes situées dans la structure de la carte.

4.1 Module init

Ce module a pour but de permettre l'initialisation de tout ce qui est nécessaire à l'ouverture du programme et à sa fermeture.

On y retrouvera, notamment, l'initialisation des modules de la SDL, la création de la fenêtre principale et de son rendu. On y prépare également tout ce dont on a besoin pour fermer le programme sans fuites de mémoire.

Ce module implémente un système de liste de pointeurs de fonction similaire au fonctionnement d'*atexit()*. Cela a pour avantage d'avoir un fonctionnement similaire en tout point (des fonctions exécutées automatiquement lors de la sortie du programme) mais également d'éviter les problèmes que cause cette méthode avec les librairies externes, ce qui, à terme, empêche les comportements inattendus du programme voir un plantage complet.

4.2 Module affichage

Le module d'affichage fournit des fonctions intermédiaires qui permettent de manipuler simplement les textures et les rectangles associés.

On retrouve notamment une fonction *creer texture* qui peut créer une texture et la placer à l'écran très facilement. On a aussi des fonctions qui permettent de facilement jouer la frame suivante d'une animation.

Toutes ces informations sont donc stockées dans la structure spécifique à l'affichage d'une texture. (figure 2)

Les deux variables qui sont particulièrement intéressantes sont *frame_anim* et *aff_fenetre*. *frame_anim* permet de sélectionner seulement une partie de texture pour l'affichage, ce qui est bien utile lors ce que l'on a un sprite avec des animations sur une seule texture. *aff_fenetre* permet, quant à elle, de choisir où l'on veut placer la texture à l'écran et quelle sera sa taille.

Le programme utilise intelligemment ces deux rectangles afin de pouvoir, par exemple, passer à la frame suivante d'une animation sans devoir se soucier du reste. La plupart des textures ayant du vide autour d'elles, on préférera faire appel à des rectangles pour les déplacements. Cela nous permet de faire d'une pierre deux coups et connaissant la position d'une entité et également sa taille, bien pratique pour la gestion des collisions.



FIGURE 2 – Structure d'affichage de texture

Pour finir, l'affichage joue également un grand rôle dans le déplacement du personnage principal, en effet lorsque que le personnage se déplace, il faut que la caméra se déplace pour ne pas perdre le personnage de vue, et que le personnage se déplace lorsque la caméra ne peut plus se déplacer. Pour cela, on fait appel à deux rectangles tx et ty stratégiquement placés afin de savoir si la caméra doit se déplacer au bien si c'est au personnage de se déplacer. On gère bien sur les collisions pour éviter que le personnage ne sorte de la bordure de l'écran ou bien qu'il traverse un objet solide.

4.3 Fonctionnement du moteur d'affichage

Le moteur du jeu est fait pour être le plus puissant possible. Il doit s'adapter à la résolution et au format de l'écran et pouvoir être le plus simple possible d'utilisation pour le programmeur.

Pour cela, on divise le jeu en 3 plans :

- l'arrière-plan qui contient uniquement la map et donc l'environnement dans lequel on évolue,
- le plan principal, là où se passe la majeure partie de l'affichage. C'est ici que l'on va afficher les entités (monstres, coffres...) sans avoir besoin de savoir si l'on doit les voir à l'écran,
- le 1^{er} plan, on y affiche tout ce qui se déplace avec la caméra. On y retrouve donc le personnage principal ainsi que l'interface.

Pour créer tous ces plans, on utilise la faculté que possède la SDL de pouvoir afficher le résultat des opérations de copies d'une texture dans une autre au lieu de l'écran.

Ainsi, on peut les textures ensemble afin de n'en former qu'une seule et unique texture qui s'affichera à la taille de l'écran. Il est important à noter que le moteur s'adapte à la vitesse de la machine et se verrouillera à une fréquence maximale de 60 images par secondes. Pour finir, le moteur simule

un environnement en 3 dimensions avec la possibilité d'ajouter une texture de superposition à la map.

4.4 Module personnage

Ce module est l'un des plus importants de notre projet, il nous permet de gérer le personnage principal de notre jeu.

Nous décidons de créer une structure afin de stocker plusieurs informations nécessaires pour le personnage principal. Cette structure est définie de façon suivante (figure 3) :



```
1 typedef struct joueur_s {
2     char * nom_pers; /*<Le nom du personnage*/
3     short int niveau; /*<Le niveau du joueur*/
4     int xp; /*<Le nombre de points d'expérience que possède Le joueur */
5     byte *trigger; /*<Une variable contenant des triggers Logiques concernant le personnage */
6     int maxPdv; /*<Le nombre de Pv max du joueur */
7     int pdv; /*<Les points de vie actuels du joueur */
8     int attaque; /*<attaque de base du joueur*/
9     int defense; /*<defense de base du joueur*/
10    int vitesse; /*<vitesse de déplacement de base du joueur*/
11    int attaque_actif; /*<attaque du joueur avec bonus d'équipement*/
12    int defense_actif; /*<defense du joueur avec bonus d'équipement*/
13    int vitesse_actif; /*<vitesse du joueur avec bonus d'équipement*/
14    statut_t *statut; /*<statut du joueur*/
15    t_l_aff *textures_joueur; /*<Tableau contenant toutes les textures du joueur*/
16    inventaire_t * inventaire; /*<Inventaire du joueur*/
17 }joueur_t;
```

FIGURE 3 – Structure personnage

Les statistiques principales du joueur sont :

- son attaque,
- sa défense,
- ses points de vie,
- sa vitesse.

Toutes ces statistiques seront modifiées en fonction du niveau ou des objets équipés par le personnage. Le personnage dispose d'un statut (expliqué dans la suite de cette section) ainsi que d'un inventaire (expliqué dans la section 4.6).

Concernant le statut du personnage (figure 4 ci-dessous), nous récupérons :

- s'il se déplace sur la map,
- son orientation,
- si un bouclier est équipé,
- la durée de l'action réalisée,

- l'action qu'il réalise,
- sa zone de collision.



```

1  typedef struct statut_s {
2      bool en_mouvement; /*<personnage en mouvement*/
3      t_direction orientation;/*<ordination du personnage*/
4      bool bouclier_equipe; /*<personnage à un bouclier d'équipé*/
5      int duree; /*<durée de l'action à réaliser*/
6      int duree_anim; /*<durée d'une animation éventuelle sur le joueur*/
7      action_t action; /*<L'action du personnage*/
8      action_t animation; /*<Animation sur le personnage*/
9      SDL_Rect zone_colision; /*<zzone de colision du personnage*/
10     SDL_Rect vrai_zone_collision; /*<La vrai zone de collision du J1 sur la carte */
11 }statut_t;

```

FIGURE 4 – *Statut personnage*

L'orientation du personnage est créée de façon à savoir dans quelle direction le personnage regarde (haut = NORD, gauche = OUEST, droite = EST, bas = SUD).

L'action du personnage se décompose en divers éléments :

- RIEN : le personnage se déplace ou ne fait rien,
- ATTAQUE : le personnage attaque,
- ATTAQUE_CHARGEE : le personnage fait une attaque chargée (attaque qui tournoie autour du personnage),
- CHARGER : le personnage charge son attaque,
- BLOQUER : le personnage bloque une attaque,
- ATTAQUE_OU_CHARGER : le personnage attaque ou charge son attaque,
- J_BLESSE : le personnage est blessé et n'a pas ses points de vie au maximum,
- SOIN : le personnage se soigne.

Ces diverses actions vont nous permettre de gérer au mieux les évènements liés au personnage.

La zone de collision du personnage permet de gérer les collisions entre le personnage, le décor de la map et les monstres.

4.5 Module monstres

Les monstres constituent un élément important de notre jeu, c'est pourquoi nous en créons plusieurs sortes :

- un chevalier, qui attaquera avec son épée,
- un sorcier, qui lancera des sorts,

— un boss.

Une fois les monstres en jeu, on gère leurs déplacements et leurs collisions en fonction ou non du joueur, si celui-ci se trouve dans la zone d'action du monstre. D'autre part, grâce au type enum « action_monstre », on adapte l'affichage du monstre qu'il soit en train d'attaquer ou qu'il soit blessé.

De plus, certains monstres, tels que le sorcier ou encore le boss, peuvent lancer des sorts. C'est pourquoi on utilise une structure et une liste dédiées aux sorts. Dans ce module sorts, on gère une fois de plus les collisions de ceux-ci sur la map. En effet, lorsqu'il rencontre un obstacle, on le supprime et si en plus l'obstacle s'avère être un joueur, on lui applique les dégâts du sort. D'autre part, lors de la création du sort il est directement orienté vers le joueur et se contente par la suite d'avancer.

On a aussi décidé de faire un module spécial pour le boss qui est un monstre particulier car il possède beaucoup de spécificités comme être rattaché à 2 clones ou bien encore avoir une orientation différente avec cette fois-ci 8 orientations possibles (nord,nord_ouest...).

4.6 Module objets et inventaire

L'inventaire s'ouvre en appuyant sur une touche qui bloque le reste du jeu et est divisé en deux parties :

- une liste d'objets équipés pouvant contenir autant d'objets qu'il y a de types d'objet afin de garantir que le joueur ne s'équipe pas de deux objets du même type comme de deux épées. Un objet équipé est donc placé dans la case de son type,
- une liste d'objets situés dans le sac.

Il fallait pouvoir sélectionner un objet dans une liste directement par indexage et sans avoir à parcourir toute la liste pour une exécution rapide du programme notamment lors de l'échange d'objets entre le joueur et son environnement, c'est pourquoi une structure spécifique utilisant un pointeur d'objets a été créée et le modèle d'une liste générique n'a pas été retenu. Un type de structure similaire vu en TP a permis sa mise en place rapide.

Nous avons fait ce choix après une analyse de la complexité des fonctions servant à équiper (mettre un objet de la liste du sac dans la liste des équipés) et déséquiper (opération inverse).

Pointeur :

déséquiper : $O(n)$ → on parcours le sac pour trouver une place.

équiper : $O(1)$ → on connaît l'index (i) de l'objet cliqué à sortir du sac.

Liste :

déséquiper : $O(1)$ → on ajoute au sac en début de liste.

équiper : $O(n)$ → on parcours la liste(liste_suivant()) i fois (index).

Or, le joueur équipe beaucoup plus souvent qu'il déséquipe.

Les statistiques d'un personnage, ses capacités et son apparence changent dynamiquement avec sa liste des objets équipés (notamment le bouclier qui permet la parade avec le clic droit et ajoute un bouclier à la texture du joueur). Il peut également se soigner et avoir un retour direct grâce à une animation (avec des potions ou herbes médicinales) ou encore utiliser des clés lui permettant d'ouvrir des coffres afin d'obtenir d'autres objets. Pour qu'un objet puisse être utilisé, il doit se trouver dans la liste des objets équipés.



FIGURE 5 – *Structure objet*

Lorsque l'on clique sur un objet du sac depuis l'interface du jeu, son index dans le sac est calculé à partir de la zone sélectionnée et il s'équipe automatiquement en remplaçant l'objet du même type déjà équipé s'il existe.

Inversement, lors d'un clic sur un objet équipé, son index dans la liste des équipés est calculé à partir de la zone sélectionnée grâce à la SDL et celui-ci est déséquipé et positionné dans la première place libre du sac. Les textures des objets changent de place dans l'interface graphique de l'inventaire et leur position s'adapte à la taille de l'écran.

La lecture des objets du jeu en langage C depuis un fichier texte est plus rapide que de passer par les fonctions JSON bien que le fichier source soit moins structuré, ce qui permet un chargement du jeu plus rapide. Chaque objet possède des informations comme ses bonus sur les statistiques du joueur.

4.7 Module coffres

Les coffres sont chargés depuis un fichier JSON qui indique leur contenu, l'identificateur de l'objet nécessaire pour l'ouvrir s'il est fermé à clé. Ces informations se retrouvent dans la structure du coffre, qui contient également une variable informant sur son état, ouvert ou bien fermé.

Lorsqu'un joueur se déplace, on vérifie s'il entre en collision avec un coffre par l'utilisation de la fonction `SDL_HasIntersection`, qui détermine si les rectangles de zone de collision du joueur et du coffre se croisent. Pour ce faire, on parcours la liste générique des coffres en comparant la zone de collision de l'obstacle rencontré par le joueur et celle des coffres de la liste. Si c'est le cas, on vérifie que le joueur soit face-à-face, pour ce faire on inverse l'orientation du coffre renseignée par sa structure avant de la comparer avec celle du joueur qui interagit avec. Le joueur doit être en possession de la clé, s'il en faut une, ce que l'on vérifie en inspectant le contenu de la case réservée aux objets de quête de ses objets équipés. S'il est en mesure d'ouvrir le coffre, alors l'éventuel objet qu'il renferme est placé à la fin de la liste des objets du sac du joueur.

Le coffre est animé et sa texture se dissocie de sa zone de collision de manière à s'approcher d'une représentation en 3 dimensions comme le reste du décor : seule sa base possède une collision et l'augmentation de l'espace occupé en hauteur par la texture lors de l'ouverture n'augmente pas l'espace occupé par son rectangle de collision.

Pour ne pas ouvrir plusieurs fois un même coffre, une variable indique son état actuel, ouvert ou bien fermé, ce qui évite une faille qui permettrait au joueur de récupérer plusieurs fois l'objet. On retrouve toutes ces informations dans la structure du coffre.

```
typedef struct coffre_s
{
    int id_cle; /*<Identificateur de l'objet de quête nécessaire pour ouvrir le coffre sinon 0*/
    int id_loot; /*<Identificateur de l'objet obtenu en ouvrant le coffre sinon 0*/
    type_coffre_t type; /*<Type de coffre*/
    t_direction_1 orientation; /*Orientation du coffre*/
    etat_coffre_t etat; /*Estat en cours par le coffre*/
    SDL_Rect collision; /*<Coordonnées*/
    t_aff* texture; /*<Texture*/
} coffre_t;
```

FIGURE 6 – *Structure coffre*

4.8 Module map

La map est une partie importante du programme, si ce n'est la plus importante.

Tout d'abord, on doit récupérer les informations qui concernent la map depuis un fichier. Nous avons choisi le format JSON, car c'est un format très

répandu, adaptable et qui a tendance à remplacer le format XML (pour plus d'information, se référer à la section 4.9.2). On stocke donc dans ce fichier, du moins au niveau des informations essentielles, la texture de la map, la taille d'une case de la map en pixels, les rectangles de collisions, les monstres, les coffres ainsi que les zones qui permettent la téléportation (pour plus de détails, se référer à la figure en annexe (figure 14)).

La map se trouvera stocker dans une structure map 7, on y retrouve notamment la texture de la map, mais également les listes (section 4.9.1) qui accueillent des « entités » comme les monstres, les coffres et les collisions. On note également l'id de la map qui il se trouve utile lors de la sauvegarde et de la téléportation entre map.

```

1  typedef struct s_map{
2      unsigned int id_map; /*<> L'identificateur de La map */
3      t_aff * text_map; /*<> La texture de La map */
4      t_aff *text_sol; /*<> La texture du sol */
5      t_aff *texture_superposition; /*<>La texture à superposer devant Le personnage */
6      unsigned int width; /*<>La longueur de La map */
7      unsigned int height; /*<>La hauteur de La map */
8      unsigned int taille_case; /*<> La taille d'une case */
9      unsigned int cases_x; /*<> Le nombre de cases affichées en x */
10     unsigned int cases_y; /*<> Le nombre de cases affichées en y */
11     list *liste_monstres; /*<> La liste des monstres de La map */
12     list *liste_sorts; /*<> La liste des sorts de La map */
13     list *liste_collisions; /*<> La liste de toutes les collisions */
14     list *liste_coffres; /*<> La liste de tous les coffres */
15     list *liste_zone_tp; /*<>La liste des points de téléportation */
16 }t_map;

```

FIGURE 7 – *La structure map*

On a 3 textures :

- *text_sol* qui est la texture de fond de la map,
- *text_map* qui est la texture du milieu où seront affichées les entités,
- *texture_superposition* qui est la texture qui sera affichée par-dessus le personnage principal.

En ce qui concerne la téléportation, on détruit l'ancienne map, on charge la nouvelle en recréant toutes les nécessités puis on téléporte le joueur à son nouvel emplacement.

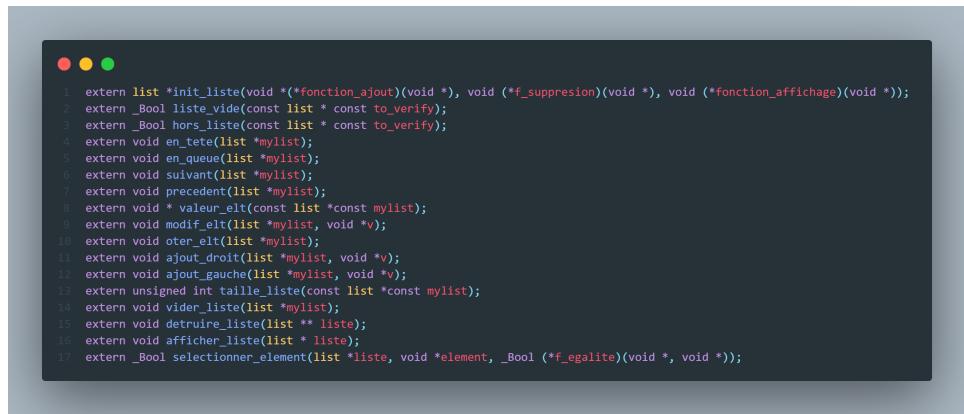
Bien évidement, on veille à gérer toutes les erreurs possibles avec le fichier map donné.

4.9 Module listes et JSON

4.9.1 Listes

On fait appel à des listes génériques homogènes afin de pouvoir facilement ajouter et supprimer des valeurs d'un contenant.

On a donc accès aux primitives suivantes 8 :



```
1 extern list *init_liste(void *(*fonction_ajout)(void *), void (*f_suppression)(void *), void (*fonction_affichage)(void *));
2 extern _Bool liste_vide(const list * const to_verify);
3 extern _Bool hors_liste(const list * const to_verify);
4 extern void en_tete(list *mylist);
5 extern void en_queue(list *mylist);
6 extern void suivant(list *mylist);
7 extern void precedent(list *mylist);
8 extern void * valeur_elt(const list *const mylist);
9 extern void modif_elt(list *mylist, void *v);
10 extern void oter_elt(list *mylist);
11 extern void ajout_droit(list *mylist, void *v);
12 extern void ajout_gauche(list *mylist, void *v);
13 extern unsigned int taille_liste(const list *const mylist);
14 extern void vider_liste(list *mylist);
15 extern void detruire_liste(list ** liste);
16 extern void afficher_liste(list * liste);
17 extern _Bool selectionner_element(list *liste, void *element, _Bool (*f_egalite)(void *, void *));
```

FIGURE 8 – *Les primitives de listes*

Cela est amplement suffisant pour tous nos besoins.

4.9.2 JSON

Pour la plupart des fichiers lus par le programme, il a été décidé d'utiliser le format JSON. Grâce à sa clarté et sa facilité d'utilisation grâce à la librairie *json-c* ce format paraît tout à fait adapté. Effectivement, il nous permet de rentrer les informations dans n'importe quel ordre, d'en omettre certaines en pouvant le détecter, d'en rajouter d'autres sans gêner la lecture du fichier. De plus, ce fichier en modifiable extrêmement facilement par nous et donc permet un travail plus efficace.

Pour finir, le format JSON nous permet de créer des tableaux d'éléments (comme les collisions dans la map) et donc ainsi pour avoir un fichier qui peut recevoir les informations « un peu dans tous les sens ».

4.10 Gestion événements

Pour la gestion des événements, le programme utilise un procédé non bloquant. Une fois par frame il va analyser tous les événements qu'il a reçus et les traitera en fonction de l'état actuel du jeu. Par exemple, si le joueur relâche la touche de déplacement, on va arrêter son mouvement.

Ce module est également compatible avec l'utilisation d'une manette, dans ce cas, le programme ne traitera pas les événements liés au déplacement du stick directionnel, car il y en a beaucoup trop de créés même lorsque que l'on n'actionne pas ce dernier. On va donc préférer vérifier l'emplacement du stick directionnel une fois par frame, on se servira ensuite du cercle trigonométrique pour diviser le cercle en quatre quarts et ainsi pouvoir obtenir une direction utilisable. On veillera à ajouter une zone morte pour éviter les cas de déplacement intentionnels (drifts).

Hors ce cas précis, il n'y a aucune différence entre la manette et l'utilisation du clavier et de la souris si ce n'est les boutons.

4.11 Main

Dans la boucle du programme principal, on va d'abord initialiser tous les systèmes du moteur de jeu. On veillera à jouer l'introduction du jeu, puis on démarrera la partie.

Au début et à la fin de chaque frame on regarde le temps système afin de savoir pour combien de temps le programme doit stopper son exécution pour ne pas dépasser le nombre limite d'images par seconde. On vérifie les événements et on applique les actions adéquates. Puis on affiche nos 3 couches de jeu en veillant bien à coudre les texture comme il le faut. On affiche le tout à l'écran et on recommence.

5 Résultats

Dans cette partie, nous vous présenterons les résultats que nous souhaitons obtenir et ceux réellement obtenus au terme de ce projet.

5.1 Attendus

Pour ce projet, nous pensions créer plusieurs choses :

- un personnage se battant contre des monstres,
- une map où se déplace le personnage,
- un système d'inventaire (figure 11 en annexe),
- des coffres/clés disponibles sur la carte,
- un système d'expérience,
- un système de points de compétences (figure 12 en annexe),
- un système d'augmentation de statistiques (figure 13 en annexe),
- une vidéo d'intro pour expliquer le scénario,
- de la musique (bonus).

Vous pourrez trouver différentes maquettes de nos idées dans la partie 7.

5.2 Obtenus

Maintenant, quant aux résultats obtenus au terme de ce projet, nous avons pu en réaliser une bonne partie et inclure de nouvelles idées. Concernant nos premières pensées, nous avons pu développer :

- le personnage se battant contre des monstres,
- des maps où se déplace le personnage,
- un système d'inventaire,
- des coffres/clés disponibles sur la carte,
- le système d'expérience,
- le système d'augmentation de statistiques,
- un mode à deux joueurs en local,
- une adaptabilité pour manettes,
- une version sur Nintendo Switch.

6 Conclusion

Ce projet avait pour but de nous faire utiliser les notions vues au cours de ces deux années de licence mais également de nous juger sur les différents aspects nécessaires au bon fonctionnement d'un projet.

Nous avons pu réaliser l'essentiel de ce que l'on voulait faire malgré le fait qu'on avait prévu de réaliser beaucoup de fonctionnalités. Étant donné que nos principales fonctionnalités ont été implémentées et sont fonctionnelles, nous avons pu réaliser quelques éléments bonus (tels que l'adaptabilité manette ou encore le multijoueur local).

Ce projet nous a permis de nous familiariser avec une librairie inconnue (SDL2) afin de réaliser un jeu vidéo fonctionnel. Nous avons également pris du plaisir à réaliser ce projet et réussir à le mener à bien.

7 Annexe

Dans cette annexe, vous pourrez retrouver un exemple de débogage avec GDB ainsi que nos ébauches et maquettes de certaines fonctionnalités du jeu.

Voici un exemple du débogueur GDB :

Le débogueur GDB s'est révélé être d'une grande aide tout au long du projet pendant lequel nous avons découvert son potentiel, notamment lors d'une modification importante de la structure des fonctions du programme dans l'objectif de se débarrasser de certaines variables globales superflues et de pouvoir appliquer les fonctions à n'importe quel élément passé en paramètre (dans cet exemple n'importe quel joueur) afin de respecter les concepts de la programmation modulaire. Il s'agit ici du cas d'une erreur de segmentation dont la cause a pu être découverte rapidement et sans avoir à relire

méticuleusement l'intégralité du code concerné. L'exécution du programme sous GDB (figure 9) informe d'une erreur de segmentation dans la fonction « équiper_sac_slot » et suggère un déréférencement illégal vers la variable « sac ».

```
Thread 1 "jeux.bin" received signal SIGSEGV, Segmentation fault.
equiper_sac_slot (joueur=0x555555de69e0, slot=1) at src/inventaire.c:141
141         lobjet_t * sac = joueur->inventaire->sac;
(gdb) up
#1  0x00005555556048b in afficher_inventaire (joueur=0x555555de69e0) at src/menus.c:236
236             equiper_sac_slot(joueur, slot_selectionne);
(gdb)
#2  0x0000555555592e3 in keyDown (ev=0x7fffffff410, joueur=0x555555de31a0) at src/event.c:46
46             afficher_inventaire(joueur->inventaire);
(gdb)
#3  0x0000555555596d1 in jeu_event (joueur=0x555555de31a0) at src/event.c:174
174             case SDL_KEYDOWN : keyDown((SDL_KeyboardEvent*)&lastEvent.key, joueur); break;
```

FIGURE 9 – Recherche erreur GDB

Aucun paramètre nul n'étant observé, la pile des fonctions appelées est remontée en utilisant la commande « up ». On observe alors les valeurs à la recherche d'une incohérence. Dans cet exemple, on remarque que l'argument « joueur » est modifié entre l'appel des fonctions « keyDown » et « afficher_inventaire », il est alors judicieux d'investiguer dans le code de la première fonction. Les valeurs proches des deux arguments mettent sur la piste du passage d'un membre de la structure joueur en paramètre plutôt que du pointeur sur la structure, cependant aucune erreur n'est déclarée à la compilation. Dans le doute je supprime les fichiers objets et exécutables du projet à l'aide de la fonction mrproper du makefile avant de le recompiler, ce qui révèle enfin le problème (figure 10) :

```
src/event.c: In function 'keyDown':
src/event.c:46:43: warning: passing argument 1 of 'afficher_inventaire' from incompatible pointer type [-Wincompatible-pointer-types]
46 |         afficher_inventaire(joueur->inventaire);
|             |
|             inventaire_t * {aka struct inventaire_s }
In file included from include/commun.h:19,
                 from src/event.c:1:
include/menus.h:35:4: note: expected 'joueur_t *' {aka 'struct joueur_s *'} but argument is of type 'inventaire_t *' {aka 'struct inventaire_s *'}
 35 | extern void afficher_inventaire(joueur_t * joueur);
```

FIGURE 10 – Erreur trouvée avec GDB

Comme envisagé, le compilateur informe que le type d'argument ne correspond pas au paramètre attendu. GDB a beaucoup servi à suivre la valeur des variables lors des phases de débogage, en particulier pour suivre la valeur des variables avec la commande « display » et le pistage de leur modifications avec la commande « watch ».

Voici maintenant quelques ébauches et maquettes de différentes mécaniques du jeu (figures 11 à 13).

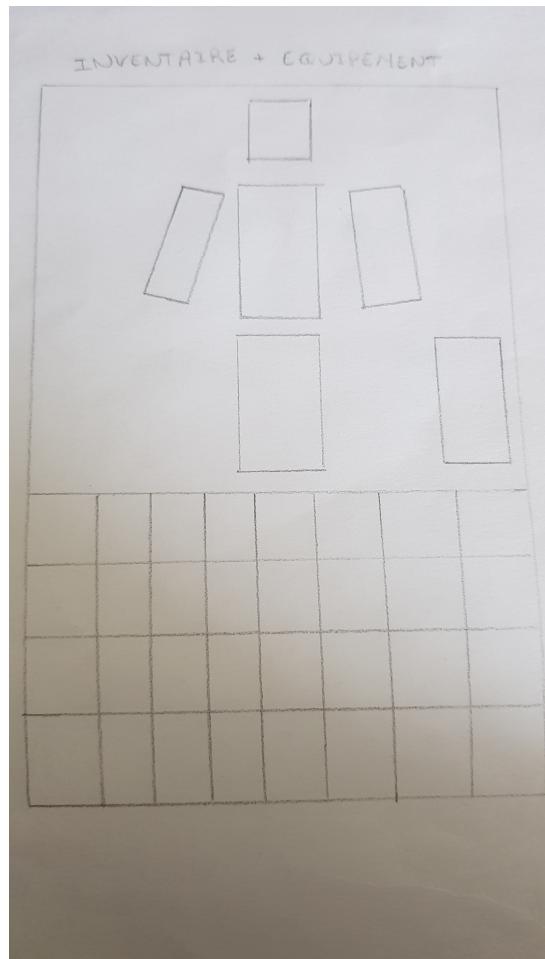


FIGURE 11 – *Inventaire personnage*

L'inventaire permet au personnage d'avoir ses statistiques modifiées et un sac est disponible (le tableau du dessous) afin de garder certains objets du personnage.

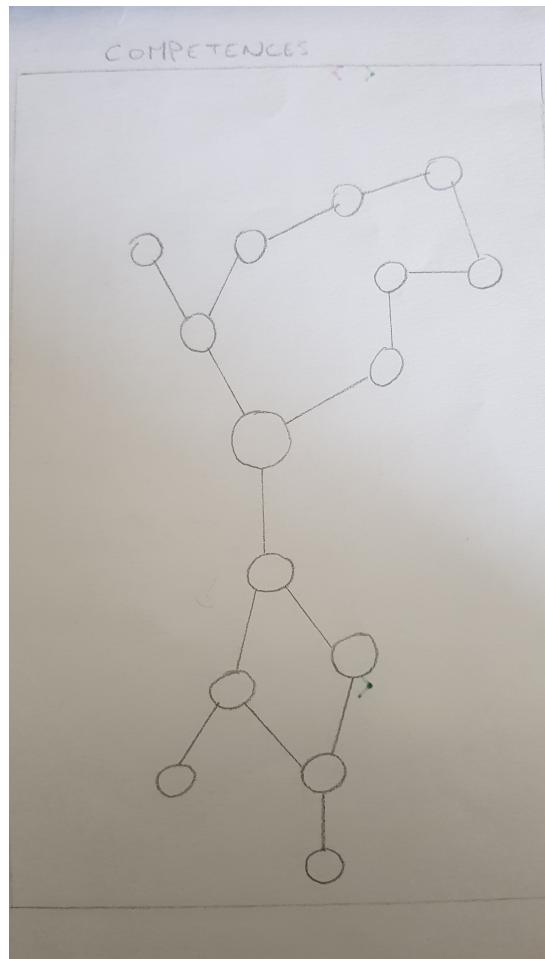


FIGURE 12 – *Tableau de compétences*

Nous avions pour idée de permettre au joueur de choisir des compétences au fil de ses niveaux afin de lui permettre d'avoir plus de facilités dans le jeu. Ces compétences pouvant chacune débloquer ensuite d'autres grâce aux liens les reliant.

STATISTIQUES		
Points disponibles : 6		
<input type="checkbox"/> +	Attaque	30
<input type="checkbox"/> +	Défense	25
<input type="checkbox"/> +	Points de vie	2000
<input type="checkbox"/> +	Agilité	15

FIGURE 13 – *Statistiques personnage*

Nous avions prévu au départ de laisser le choix au joueur pour augmenter ses différentes statistiques sous forme de points obtenus lors d'une montée de niveau du personnage.

```
1  {
2      "id" : 2,
3      "file-path" : "ressources/background/map/outside_sanada.bmp", /* La texture de fond de la map */
4      "superposition": "resources/background/map/outside_sanada_superpose.bmp", /* La texture de superposition de la map */
5      "width" : 55,
6      "height" : 35,
7      "taille case" : 32, /* La taille d'une case en pixel */
8      "starting-point" : [5, 5],
9      "monsters" : [ /* Un tableau représentant les monstres présents */
10          {"type" : "witcher",
11          "position" : [300,100]},
12      ],
13      "wall" : [ /* Un tableau contenant des zones de collision */
14          {
15              "x" : 5,
16              "y" : 3,
17              "h" : 3,
18              "w" : 1
19          }],
20      "zones tp" : [ /* Un tableau de zones de téléportation */
21          {
22              "x" : 7,
23              "y" : 1,
24              "w" : 2,
25              "h" : 2,
26              "destination" : 1,
27              "coords" : [224,382]
28          }
29      ],
30      "chest" : [ /* Un tableau de coffres */
31          {"type" : "profilferme",
32          "position" : [15,12],
33          "id_cle": 9,
34          "id_loot": 3}
35      ]
36  }
```

FIGURE 14 – *Un exemple de fichier map*

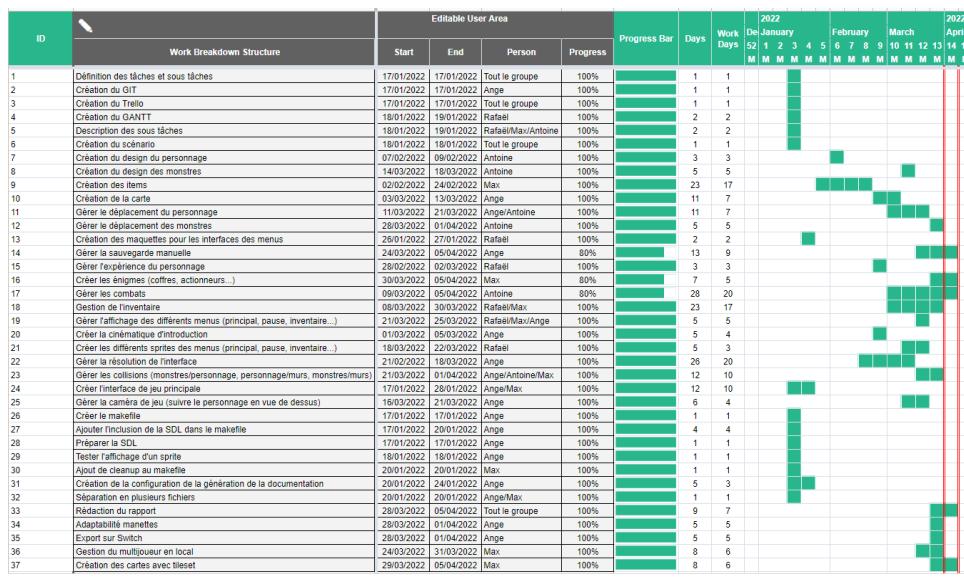


FIGURE 15 – *Gantt réel*