

Le Mans Université
Licence Informatique 2ème année
Rapport de projet
Final Fantasy Reloaded

Papot Alexandre, Karman Nathalie, Girod Valentin

19 Avril 2019

Table des matières

1	Introduction	3
1.1	Notions du jeu	3
1.2	Objectifs du projet	5
2	Organisation du travail	5
3	Conception	6
3.1	Détails des règles du jeu	6
3.2	Fonctionnalités du programme	7
4	Plan de développement	7
4.1	Back-end	7
4.1.1	Principales structures	7
4.1.2	Structure objet	8
4.1.3	Structure sort	8
4.1.4	À propos des fonctions	9
4.2	Front-end	12
4.2.1	Version terminal	12
4.2.2	Version SDL	12
4.2.3	Première partie SDL	13
4.2.4	Seconde partie SDL	13
4.3	Editeur de carte via SDL	14
5	Résultats	15
5.1	Améliorations possibles	15
5.2	Difficultés rencontrées	16
A	Exemple de débogage	16

1 Introduction

Ce papier est à l'origine de notre projet de deuxième année en licence informatique, il énonce les différentes étapes réalisées pour la confection d'un jeu de type Role Playing Game (RPG)



FIGURE 1 – Exemple de RPG : Final Fantasy 1 (1987) image issue de la video « Final Fantasy 1 Walkthrough Longplay PSP Part 1/3 » et de l'article : « The Best Version of Every Final Fantasy Game »

Tout d'abord, nous allons expliquer le principe du jeu en évoquant les objectifs et contraintes du projet. Seront ensuite décrits les détails de la gestion du projet et les principales phases qui en découlent : la planification du travail et le développement de celui-ci. La dernière partie du document comprend la mise en œuvre du jeu et les outils associés ainsi que les résultats obtenus.

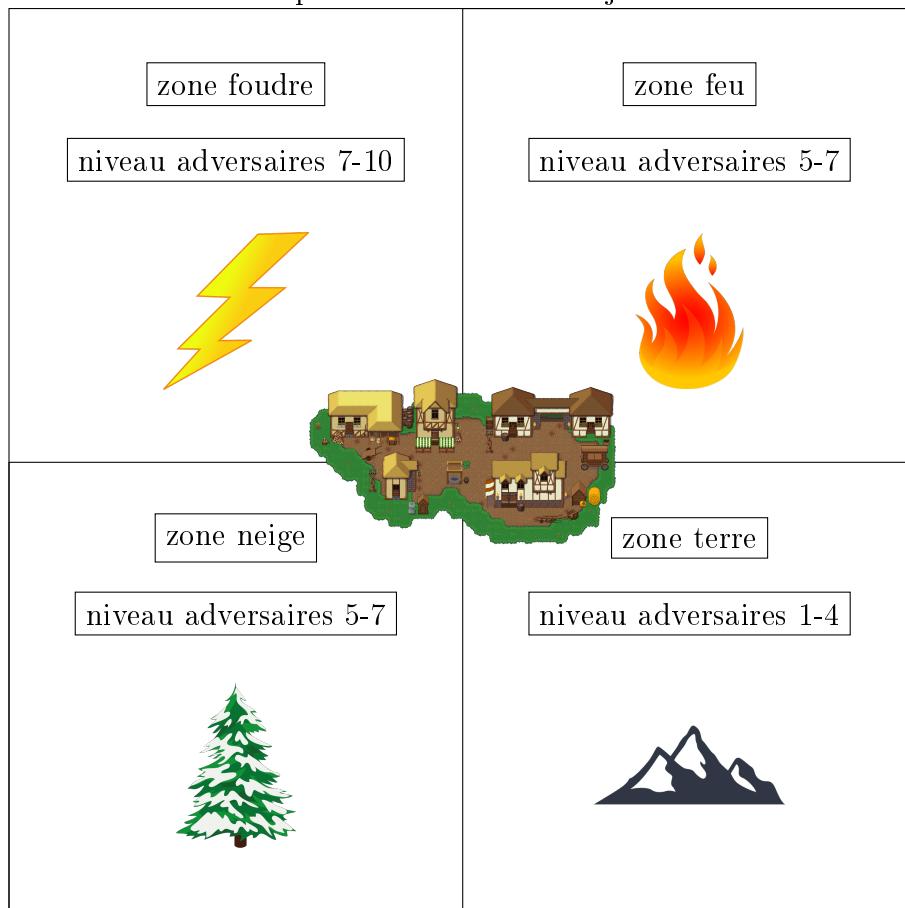
1.1 Notions du jeu

Le jeu se base sur un modèle qui existe depuis la fin des années 80, le joueur incarne un personnage dont il choisit la classe, le nom et le sexe. Le sexe est une caractéristique qui ne change en rien le déroulement du jeu, en contrepartie la classe du personnage influencera ses capacités. Si la classe « guerrier » est choisie, les attaques physiques seront plus efficaces, ou par exemple, si il joue le rôle d'un « mage », son point fort sera les sorts. Une fois la création du personnage achevée, le joueur se retrouve au milieu d'une carte où il peut se déplacer librement. Il va rencontrer des personnages non-joueurs (PNJ) qui lui donneront des missions, le but du jeu étant de réaliser toutes les missions.

La carte est composée de cinq zones :

- un village (zone pacifique située au milieu de la carte),
- et quatre autres zones dans lesquelles le joueur est susceptible d'être attaqué par des monstres.

Répartition des zones du jeu :



Le joueur pourra s'aider d'un inventaire pour stocker des objets qui pourront lui être utiles au cours du jeu.

1.2 Objectifs du projet

L'objet du projet consiste élaborer la conception et la création d'un jeu en groupe et ce, dans les temps impartis. Dans le cadre d'un travail en groupe, le projet doit être segmenté en tâches, chacune ayant une priorité : il faut établir les bases nécessaires pour un jeu de type RPG. Le concept de ces bases ne peut se former qu'après avoir établi une manière précise de travailler en groupe, les outils à employer et les possibles contraintes. En effet, le sujet étant divisé en problèmes bien distincts, le modèle de cycle de développement en V a été le modèle choisi pour réaliser notre jeu. Le travail devait se faire en équipe et à l'aide d'une plateforme qui gère différentes versions d'un même projet, il fallait une manière de pouvoir éditer les bouts de codes de manière indépendante avec des tests unitaires dans un fichier de test spécifique pour les modules plus importants pour ensuite procéder à des tests d'intégration.

2 Organisation du travail

Concernant l'organisation de l'équipe, le projet a été scindé en deux et il n'a pas été désigné de chef de projet. Le groupe s'est donc réparti comme ci-dessous :

- partie back-end
- une partie front-end

La partie front-end qui se sert de la bibliothèque Simple DirectMedia Layer (SDL), est assurée par une personne du groupe. Quant à la partie back-end, regroupant le cœur des fonctionnalités du jeu, elle est prise en charge par les deux autres membres du groupe. Les jeux de tests du back-end nécessitant un affichage terminal, la version terminal du jeu a donc été assurée par cette même équipe. Les implémentations des tests SDL ont eux aussi, demandés la création de quelques modules à priori réservées au back-end, mais qui finalement sont spécifiques à la version SDL.

Il est important de rappeler que les deux parties ne sont pas indépendantes, et que les choix du back-end sont fortement influencés par les contraintes de la SDL et inversement. L'outil collaboratif Github est l'outil qui permet à l'équipe de gérer les différentes versions du travail ainsi que la mise en commun des deux versions (SDL et terminal). Pour assurer une bonne compréhension de tout le code réalisé, Doxygen est l'outil de documentation

semi-automatique qui nous a été proposé : une personne du back-end a générée le fichier de configuration afin que celui-ci soit adapté au langage C, langage qui a d'ailleurs été celui utilisé pour tout le projet. À partir de ce fichier de configuration, tous les membres de l'équipe ont pu commenter leur code en respectant les règles de typographie Doxygen.

3 Conception

La modélisation du jeu repose sur les besoins et contraintes de celui-ci.

3.1 Détails des règles du jeu

- Le joueur a des points de vie qui vont évoluer en accord avec son niveau.
- Le niveau est basé sur les points d'expérience qui sont obtenus à la fin de chaque combat.
- Les sauvegardes sont décisives : si le joueur meurt lors d'un combat, il retourne à la dernière sauvegarde. Les sauvegardes ne sont disponibles que par le biais d'un objet nommé « tente » ou si le joueur se trouve dans le village.
- Le niveau varie de 1 à 10, et ce pour tous les personnages (joueur et adversaires).
- Les quatre zones sont divisées en biomes : les biomes indiquent les bornes du niveau des adversaires et ont leur thème défini.
- Le joueur dispose d'un inventaire, avec un nombre d'emplacement fini, lui permettant de transporter des objets qui peuvent être ou non, équipés.
- Chaque fois que le joueur se déplace d'une case, le joueur peut « aléatoirement » se faire attaquer par un, deux, trois ou encore quatre adversaires.
- Tous les personnages possèdent un champ qui détermine leurs différents états (étourdi, empoisonné, etc). Ces états affectent le personnage lors du combat et peuvent perdurer après le combat.

3.2 Fonctionnalités du programme

Lorsque le joueur lance le jeu, il peut soit commencer une nouvelle partie ou choisir de charger une sauvegarde qu'il a réalisé préalablement. Lors d'une nouvelle partie, le joueur doit d'abord créer son personnage, en sélectionnant un sexe, un nom pour le personnage et une classe (guerrier, mage, chasseur, prêtre). Une fois la création du personnage réalisée, le joueur se retrouve dans le village où il peut librement se déplacer et faire des rencontres qui lui indiqueront les missions à faire. En se déplaçant en dehors du village, le joueur va devoir vaincre les monstres qui vont l'attaquer. Lors d'un combat le joueur a le choix :

- d'attaquer physiquement le monstre,
- de lancer un sort (offensif ou défensif),
- d'utiliser une potion disponible dans l'inventaire,
- d'essayer de s'enfuir.

Quand le joueur sort vainqueur d'un combat ou complète une mission il est récompensé en point d'expérience (xp) et reçoit des objets qui sont en accord avec le niveau des adversaires battus.

4 Plan de développement

4.1 Back-end

Pour le développement des bases du jeu, il faut partir de la création du personnage, la structure représentant le personnage est une structure qui est partagée avec les monstres. La mise en global de plusieurs variables s'est avérée nécessaire pour la version SDL : le personnage principal, l'inventaire, la position du joueur et les sorts.

4.1.1 Principales structures

Le champ qui définit les états des personnages (limité à 8 états) est un « dictionnaire » allant de 0 à 7 contenant des booléens VRAI ou FAUX.

Les objets obtenus à l'issue d'un combat sont quant à eux générés à partir du niveau des monstres vaincus. La fonction qui s'occupe de la génération des objets récompenses post-combat prend donc en compte le niveau de chaque adversaire.

Les objets sont modélisés à partir de la structure suivante :

4.1.2 Structure objet

- Nom de l'objet : une chaîne de caractère.
- Type de l'objet : un entier qui définit si c'est une armure, une arme, une potion ou encore, une tente. Cet entier varie donc de 0 à 3 et est représenté par un type enum avec le nom respectif de chaque type, pour la lecture du code.
- L'état de l'objet : ce champ définit la spécificité de l'objet ; une armure (type) en cuir ou en or (état), ou une potion (type) de mana ou super mana, etc.
- Valeur de l'objet : un entier qui représente soit le pourcentage d'effet de l'objet, soit une valeur allant de 0 à 7 (type enum) qui est celle utilisée pour affecter l'état d'un personnage. Par exemple la valeur 0 indique que l'objet aura un effet sur l'état *Stunt*, le tableau des états du personnage est mis à VRAI à l'indice 0.

Dans un soucis d'efficacité au niveau de la mémoire vive, les objets que le joueur possède sont créés dynamiquement : la structure de l'inventaire comprend un tableau de pointeurs sur des objets. À chaque nouvel objet dans l'inventaire, on crée la place nécessaire pour celui-ci et on incrémente le compteur d'objet, ce compteur est le champ de la structure qui nous aide à parcourir l'inventaire et savoir si il est plein.

4.1.3 Structure sort

La structure sort possède trois champs : son nom de type chaîne de caractère, son type (qui est un entier variant de 0 à 2, pour définir si c'est un sort offensif, défensif ou modificateur d'état) et enfin, la valeur du sort qui est soit un pourcentage soit une valeur symbolique. Les valeurs symboliques sont pour la plupart, déclarées en type enum pour la lisibilité du code. Les sorts sont d'abord créés dans un tableau statique mis en global. Étant donné que tous les personnages ont un champ *liste_sorts*, tous peuvent accéder à ce tableau global et ce grâce à une mise en œuvre par pointeur. Il s'agit d'une liste chaînée (*liste_sorts*), qui contient l'indice du sort respectif. Ce choix a été motivé pour éviter des redondances entre les monstres et le joueur.

4.1.4 À propos des fonctions

Le programme est lancé à partir d'une fonction *menu* qui propose au joueur soit de créer une nouvelle partie, soit de charger une partie ou quitter le jeu. Lors du chargement d'une partie, on affecte à la variable globale les données qui ont été sauvegardées et le jeu est lancé avec une fonction nommée *en_jeu*. Si on démarre une nouvelle partie, la fonction de création du personnage est lancée et attend que l'utilisateur choisisse les spécificités de son personnage. C'est seulement après, que la partie démarre et toutes les fonctions de début du jeu s'imbriquent.

La fonction *en_jeu* contient une boucle while vérifiant l'état de la valeur globale *etat_jeu* qui varie de trois états différents, tant que celle-ci ne correspond pas à une fin de partie, on propose au joueur de se déplacer, sauvegarder la partie, afficher son inventaire, afficher ses caractéristiques.

Quand le joueur se déplace, pour chaque case qu'il parcourt, la fonction *fight_rand* est appelé, celle-ci génère un nombre pseudo-aléatoire entre 1 et 100 et le compare avec la matrice qui est influencée par les coordonnées du joueur, si le nombre est inférieur ou égale au pourcentage de la matrice, un combat se déclenche.

Lorsqu'un joueur rentre dans un combat, la fonction *combat_on* s'active, celle-ci va tout d'abord déterminer aléatoirement le nombre de monstre que le joueur doit affronter et créer chaque monstre (la fonction qui les crée, prend en compte les coordonnées du joueur pour qu'ils aient un niveau cohérent). Une boucle do while définit si l'on reste ou pas en combat (la condition d'arrêt étant la mort du joueur ou la mort de tous les monstres).

On commence par afficher les noms des monstres avec le niveau et les points de vie. On propose ensuite au joueur les différents choix du menu de combat qui lui sont proposés. L'action est alors jouée et les monstres attaquent à leur tours. Les monstres présent en combat sont placés dans un tableau, lorsqu'un monstre est vaincu, il convient de supprimer le monstre du tableau et de diminuer la taille du tableau, ceci est assuré par une fonction *update_tab_monster*.

Pour la gestion des sorts, la fonction *init_tab_sort* initialise un tableau global défini en statique. Ce tableau de type sort possède donc une taille prédéfinie et l'on pourrait y ajouter d'autres sorts d'usage commun aux personnages si besoin. Les sorts du jeu en dur permettent au personnage et aux

monstres d'accéder à leurs sorts par le biais d'une liste chaînée *liste_sort* qui contient l'indice respectif de celui-ci. Lors de l'affectation d'un nouveau sort, la fonction *attribution_sort* crée dynamiquement l'espace nécessaire pour une entité donnée. Le concept de liste chaînée permet au joueur d'avoir une quantité non finie de sort : si c'est le premier sort ajouté, on indique que la liste est inaugurée et le début de cette liste n'est pas mis à *NULL* mais à l'adresse allouée. Sinon, chaque nouveau sort est ajouté à la fin de la liste, les uns à la suite des autres, le dernier pointant toujours sur *NULL*. Il a fallu faire une fonction *debut_liste* qui joue le rôle de drapeau et permet de retourner au début de la liste quelque soit la position de l'élément courant dans la liste. Une fonction *supprimer_sorts* permet de facilement supprimer toute liste de sorts à partir de l'entité.

Au cas où le joueur meurt, il est redirigé automatiquement vers le menu d'accueil et est contraint de recommencer une partie ou charger une partie ultérieure. Les points d'expérience sont calculés à la fin du combat, si le joueur a battu tous ses adversaires. Il retourne ensuite sur la carte.

La récompense obtenue après un combat, se fait grâce à la fonction *create_loot* qui prend en paramètre le niveau du monstre vaincu. La fonction alloue dynamiquement un *objet_t*, et fait appel aux trois fonctions *loot_type*, *loot_state*, *value* qui respectivement :

- remplit le champ du type de l'objet créé, en générant un nombre pseudo-aléatoire : il y a 5% de chance d'obtenir une armure, 15% de chance d'obtenir une arme, 30% de chance d'obtenir une tente et 70% d'obtenir une potion,
- remplit le champ « état » prenant en compte le niveau du monstre, pour attribuer un objet correspondant à la tranche de niveau,
- attribue à l'objet, une valeur à partir du type et de l'état.

Dans le cas d'une récompense de mission, on ne souhaite pas attribuer de récompense aléatoire, on fait alors appel à *create_object* qui prend directement en paramètre l'état et le type de l'objet.

Aussitôt l'objet créé, on emploie la fonction *fill_up_inventory* qui se charge de placer l'objet dans l'inventaire, si il reste un emplacement libre (retourne un code d'erreur sinon). Le joueur peut retirer un objet ou plus pour les remplacer par d'autres objets.

Le déplacement du joueur, en version terminal, se fait en entrant les coordonnées de la position où il souhaite se déplacer (x et y). Le déplacement va alors se faire automatiquement à l'aide de la fonction *deplacement_joueur*.

Concernant la sauvegarde de la partie du joueur, la fonction *sauvegarde_partie* inscrit dans un fichier texte ayant un nom au choix de l'utilisateur contenant les champs à sauvegarder avec le séparateur « ; »(caractéristiques du joueur, objets dans l'inventaire et position du joueur sur la carte). La fonction *charger_partie* quant à elle se charge de créer un personnage et remplir les champs à partir du fichier de sauvegarde pour ensuite relancer la partie.

Le projet dispose également d'un fichier commun.[ch] contenant les fonctions dites « outils ».

Pour faciliter la création des chaînes de caractères de manière dynamique, la fonction *creer_string* prend en paramètre un pointeur sur une chaîne de caractère qui est celle de sortie et une chaîne de caractère statique : on alloue la mémoire requise pour exactement la taille de la chaîne statique. Pour supprimer une chaîne, une fonction *supprimer_string* est aussi disponible. Encore pour la saisie de chaîne de caractère et d'entiers de manière sécurisée, une fonction qui s'occupe de vérifier si le tampon est plein nommée *vider-Buffer* fait en sorte de vider le buffer.

La fonction *entier_aleatoire* permet de générer un nombre entier pseudo-aléatoire comprise entre une borne min et max.

Pour le débogage, les outils GNU Debugger (GDB) et Valgrind ont été utilisés. Valgrind étant suffisamment clair pour indiquer les problèmes d'adressage et fuites de mémoire, c'est l'outil qui a été le plus employé.

4.2 Front-end

4.2.1 Version terminal

Le principal soucis de l'affichage terminal est de faire en sorte que le joueur ait une expérience proche de celle qui peut se faire en SDL, l'affichage de la map se fait à l'aide de la fonction *afficher_map* qui permet d'afficher la carte sur terminal avec une mise à l'échelle pour plus de lisibilité sur terminal (axe y 40 fois moins important, axe x 10 fois moins important). Le joueur est affiché à l'aide d'une légende qui indique que « X » représente la position actuelle du joueur et le village avec le caractère « . ». Les combats sont donc orchestrés de manière dynamique et jouent sur le suspense de l'affichage à l'aide d'une fonction *POSIX* appelée *sleep* .

Le menu de combat représente un menu typique de combat au tour par tour de RPG : le joueur est libre de choisir entre une des actions données et peut changer d'avis avant d'avoir vraiment validé son coup. Par exemple, le joueur peut choisir de regarder dans son inventaire et choisir une potion, mais si il a changé d'avis, il peut retourner au menu précédent, et choisir d'attaquer ou essayer de s'évader. La fonction « *running_away* », qui joue sur l'aléatoire, peut ne pas aboutir et les monstres attaquent à leur tour puisque le joueur a choisi son coup. Le jeu affiche tous les ennemis qu'il reste encore à vaincre avec leurs points de vie et aussi les points de vie du joueur, ce qui permet au joueur d'élaborer une stratégie. Si l'issue de la rencontre est la mort du joueur, on redirige l'utilisateur sur le menu principal, sinon on affiche les points expériences obtenus et retourne au menu de navigation.

4.2.2 Version SDL

L'affichage SDL est découpé en deux parties. Un premier fichier nommé *fonction_sdl.c* contient les fonctions de bases pour préparer et afficher des éléments (sprites du jeu sous format image jpg ou png), fonctions qui vont être utiles pour l'affichage général. La seconde partie est gérée par le fichier *affichage_sdl.c* qui se sert des fonctions définies dans le premier fichier pour afficher les menus, les missions et la map.

4.2.3 Première partie SDL

Concernant la première partie, elle contient *drawImage* qui affiche une image d'une taille donnée à des coordonnées données, *loadImages* qui charge toutes les images d'un répertoire et les met dans un tableau de textures, pour les garder en mémoire et les utiliser dans *drawImage*.

init_affichage ouvre une fenêtre en plein écran, prend en compte la taille de l'écran, charge la police de caractères et lance le chargement des images avec *loadImages*. *drawText* quant à lui écrit du texte donné, d'une taille donnée à des coordonnées données.

unloadImages permet de libérer la memoire, pour des raisons de lisibilité du code une *fonction fond blanc* affiche un fond blanc (utile pour le menu de sélection), une *fonction quitter affichage* est aussi disponible pour quitter la SDL.

4.2.4 Seconde partie SDL

Dans la seconde partie, les fonctions font appel aux outils de la première partie, *afficher_Map* affiche l'environnement, compte le nombre de sprite affichables en fonction de la taille de l'écran, vérifie quel est le biome de chacun de ces sprites, puis les affiche avant d'afficher le joueur. Pour la gestion des collisions *detecter_mouvement* détecte les mouvements du joueur, en prenant en paramètre les coordonnées en pointeur et les modifie selon le mouvement. Pour l'affichage de l'inventaire *afficher_inv* prend en paramètre les dimensions de l'image d'inventaire et ses coordonnées à l'écran, et affiche les barres de statistiques (vie, mana), les objets, et le niveau du joueur.

Pour la gestion des cliques dans l'inventaire, *showInventory* gère les choix de l'utilisateur.

Le menu principal *afficher_menu* gère l'affichage, qui contient 4 boutons pour : démarrer, charger, quitter le jeu et encore, éditer la map. Il prend en paramètre un tableau de chaîne de caractères, qui sont affichées sur les boutons du menu. L'affichage des combats *afficher_combat* prend en entrée la liste des monstre et la quantité des monstres, la fonction affiche ensuite l'interface de combat qui contient :

- une image de fond « champ de bataille » avec un menu de sélection des actions du joueur,
- les caractéristiques des monstres et du joueur,
- les images des monstres et du joueur.

Pour la sélection des actions du joueur *affich_choix* permet de saisir l'ac-

tion du joueur (sur ce qu'il clique), qui renvoie un *entier* et détecte les cliques de la souris sur les actions à choisir.

Pour la sélection de l'ennemi *choisir_ennemi* prend en compte les monstres qui sont en combat et donne la possibilité au joueur d'attaquer. L'affichage des quêtes *afficher_quete* permet d'être chargé à partir d'un fichier d'extension *txt*, il y a dans ce fichier, toutes les informations qui permettent d'afficher le personnage non-jouable sur la carte (les coordonnées). Le fichier contient aussi : les textes des quêtes (début/fin quête), un caractère indiquant le statut de la quête (si on est au début de la quête, si on peut voir l'objectif de la quête, si on est à la fin de la quête, ou si la quête est simplement disponible sur la carte), une chaîne de caractères qui affichera soit le nom du PNJ, soit une image de celui-ci (cela dépend de la distance entre le PNJ et le joueur). Si on est dans le cas où on est proche du PNJ, la fonction est exécutée pour afficher le texte de la mission, avec son nom sinon on affiche plutôt que son sprite, les coordonnées pour afficher l'image qui correspond au statut de la quête, ou le texte.

Pour le menu en *SDL* *afficher_menu_perso*, il prend en paramètre le nom, la classe et le genre du perso (en pointeurs) et affiche des boutons de sélections de la classe et du genre, un sprite du personnage selon ces paramètres, et affiche un champ permettant la saisie du nom du personnage lors de la création de celui-ci.

4.3 Editeur de carte via SDL

L'éditeur de carte créé en C via *SDL* est géré par la fonction appelée *gestion_editeur* qui montre la map du jeu en permettant les déplacements de manière très fluide dans celle-ci. Un menu de sélection en bas de l'écran qui contient les différents sprites du jeu, ainsi qu'un bouton d'enregistrement pour enregistrer les changements faits sur la map dans le fichier *map.txt* et un autre de retour au menu principal. Les changements sont donc faits par de simples cliques et changent le fichier avec les entiers respectifs des images du jeu : on peut y placer des arbres, des rues, et autres. Les collisions ne sont pas prises en compte pour faciliter l'édition de la carte et grâce à ça on peut bouger de manière très rapide.

La fonction *afficher_creation* prends les mêmes paramètres que la fonction précédente, gère le menu de création de personnage (en détectant les cliques sur les différents boutons, les touches du claviers pour le nom du perso), et utilise la fonction précédente pour afficher le menu dès qu'une modification est faite dans le nom, le genre ou la classe.

5 Résultats

Actuellement, le jeu dispose d'un éditeur de carte fonctionnel en C à partir de la librairie SDL.

Dans le jeu le joueur peut :

- se déplacer sur la carte,
- tomber en combat à partir de la matrice de menace,
- vaincre les monstres et obtenir des récompenses adaptées,
- sauvegarder sa partie ou charger une partie déjà sauvegardée.

Le jeu est jouable en version SDL et Terminal.

5.1 Améliorations possibles

Les fonctionnalités du jeu peuvent encore évoluer, avec par exemple des marchands d'objets dans le village et à quelques points stratégiques de la map. Cela permettrait au joueur d'obtenir des objets en utilisant une monnaie propre au jeu.

Le jeu devrait avoir en vérité, une version finale qui obligerait le joueur à former une équipe de quatre personnages. Les classes ne se limiteraient pas à quatre, mais à huit, et seraient importantes pour la stratégie de combat du joueur.

A propos des sorts, il serait possible d'implémenter plusieurs tableaux de sorts, qui pourraient être réservés à des personnages spécifiques. Cela serait primordial pour que le jeu soit complet et moins monotone.

5.2 Difficultés rencontrées

Lors de la mise en oeuvre de ce projet, l'usage d'un nouvel outil (*Github*) et le travail en équipe a demandé un temps d'adaptation (bien communiquer afin de rendre les différents modules compatibles).

De plus, il a fallu rapidement dès le début du projet restructurer le code pour régler des problèmes de dépendance circulaire avec le Makefile. La deadline a été également une difficulté majeure, en effet les jeux de type RPG étant souvent longs avec beaucoup de contenu, il a fallu se concentrer sur les fonctionnalités minimales (personnage, combat, inventaire, map), et sur l'aspect technique plutôt que le contenu du jeu (narration, bestiaires etc).

A Exemple de débogage

```
nom du sort: Hologramme ; valeur du sort: 0% type du sort offensif
==4995== Invalid write of size 4
==4995==   at 0x109093: bestiaire_neige (perso.c:187)
==4995==   by 0x1091D2: monster_creation (perso.c:216)
==4995==   by 0x108972: main (test_perso.c:10)
==4995== Address 0x522ecac is 28 bytes after a block of size 16 in arena "client"
==4995==
```

FIGURE 2 – Exemple de bug rencontré

L'outil valgrind par la commande `--leak-check=full ./nom_prog` nous affiche exactement si il y a une adresse qui n'est pas allouée correctement (`invalid write of size 4` indique que l'on essaie d'écrire une donnée à une adresse qui ne peut pas contenir toute la donnée). Dans ce cas-là, les lignes de code qui sont problématiques sont affichées ainsi que les modules qui sont ceux qui comprennent l'anomalie.