

HIGHWAY TO L2

Le Mans Université
Licence Informatique 2ème année

Nathan Gauteron - Nathan.Gauteron.Etu@univ-lemans.fr

Théo Duluard - Theo.Duluard.Etu@univ-lemans.fr

Rémy Georget - Remy.Georet.Etu@univ-lemans.fr

Matéo Gallais - Mateo.Gallais.Etu@univ-lemans.fr

<https://github.com/Natykk/Highway-to-L2>

13 avril 2023

Table des matières

1	Introduction	3
2	Conception	3
2.1	Analyse	3
2.2	Cahier des charges et fonctionnalités	3
3	Organisation	4
4	Développement	5
4.1	Maps	5
4.1.1	Structure des salles	5
4.1.2	Structure des étages	6
4.2	Entités	7
4.3	Objets et inventaire	8
4.4	Arbre de compétences	9
4.4.1	Compétences	9
4.4.2	Arbres	10
4.5	Système d'interactions	11
4.5.1	Attaques	11
4.5.2	Marchand	13
4.5.3	Interface de l'inventaire	13
4.6	Système de sauvegarde et de chargement	14
4.7	Recherche et conception des ressources	16
5	Conclusion	16
5.1	Résultats	16
5.2	Améliorations	17
6	Bibiographie	18
7	Annexes	19
7.1	Debogage avec l'outil GDB	19

1 Introduction

Highway to L2 est un jeu de type «Roguelike» en 2D. Le but du jeu est de descendre jusqu'aux tréfonds des enfers sans mourir. Le joueur traverse trois zones majeurs : la forêt, les mines et enfin les enfers. Il devra se mesurer à des monstres de plus en plus féroces tout au long de son aventure pour pouvoir améliorer ses techniques et pouvoir affronter les boss de fin de zones. La progression dans le jeu pourra être reprise n'importe quand grâce à un système de sauvegarde et de chargement.

Au cours de son aventure, le joueur rencontrera différents types d'ennemis, allant des gobelins et des loups aux orcs et aux démons.

Le jeu propose également un arbre de compétences pour chaque classe, permettant au joueur de choisir quelle compétence débloquer en fonction de son style de jeu. Les compétences peuvent être améliorées en collectant des objets sur les ennemis.

Enfin, les trois zones majeures traversées par le joueur ont chacune leur propre ambiance visuelle, créant une immersion totale pour le joueur.

2 Conception

Pour concevoir Highway to L2, il a fallu débattre. Nous avions chacun plusieurs idées de jeu et envies pour ce projet. Les jeux de type «Roguelike» étaient le parfait compromis entre nos idées. Ceux-ci nous plaisaient puisqu'ils rassemblaient toutes les suggestions que nous nous avions évoqué précédemment. Chacun y allait de son idée avec un système de combat, de compétences, d'objets, etc. La difficulté des combats et des niveaux était également sujet à débats.

2.1 Analyse

A l'origine, nous avions imaginé un jeu avec des déplacements fluides, des sprites qui colleraient au thème et qui s'y incrusterait parfaitement. Cependant la réalité nous a rattrapé. En effet, le déplacement fluide du personnage demandait trop de temps et de ressources alors que nous avions d'autres points à traiter. De plus, la recherche des sprites a été très complexe due au fait que nous voulions des sprites spécifiques afin de coller au mieux à l'ambiance du jeu et à notre angle de vue isométrique (vue de haut).

2.2 Cahier des charges et fonctionnalités

La première problématique du jeu fut la mise en place d'une carte style «Roguelike» où le personnage évoluerait au fur et à mesure de son aventure intégrant des salles, des niveaux, des monstres et des boss, tout en facilitant le placement sur la carte des divers éléments. Nous avons donc fait le choix d'utiliser des matrices pour la génération des salles et des étages.

Afin de garantir une expérience de jeu plus personnalisée, nous avons choisi d'implémenter des arbres de compétences. L'utilisateur pourra alors choisir quel type de combattant (mage, archer, assassin, guerrier) il souhaitera incarner et avec quelle spécialisation (dégâts, vie, vitesse de déplacement, vitesse d'attaque, réduction du diamètre de détection des mobs). A partir de cette implémentation, d'autres fonctionnalités doivent en découler. Comme par exemple, l'utilisation d'objets pour débloquer les compétences. Nous nous sommes alors interrogés sur le moment où le joueur pouvait améliorer son arbre de compétences. Nous avons fait le choix d'intégrer un marchand après chaque boss d'étage pour améliorer nos compétences pour l'étage suivant. Un jeu type «Roguelike» n'étant rien sans monstres, nous avons intégré douze types de monstres différents qui seront intégrés à l'étage correspondant : les quatre premiers au premier étage, les quatre suivants au deuxième puis les quatre derniers au troisième. Mais comment obtenir les objets ? Chaque monstre possède un inventaire et donc des objets variés. A la mort du monstre, cet inventaire est transféré dans

celui du joueur. Pour rendre le jeu plus portable, nous avons créé une sauvegarde et un chargement dans un fichier des données du joueur.

3 Organisation

L'organisation a été un élément crucial pour le développement de « Highway to L2 ». Nous avons utilisé plusieurs outils pour faciliter la gestion du projet. Par exemple le diagramme de Gantt qui nous a permis de planifier et de suivre l'avancement des tâches. Voici un tableau simplifié de la répartition de celles-ci :

Nathan	Remy	Théo	Matéo
<ul style="list-style-type: none">• Intégration des entités dans la map• Interface Inventaire• Construction du jeu	<ul style="list-style-type: none">• Réalisation des entités• Réalisation de l'inventaire• Menu des compétences	<ul style="list-style-type: none">• Réalisation de la map• Réalisation des arbres de compétences• Réalisation du menu principal• Sauvegarde et chargement	<ul style="list-style-type: none">• Attaques à distance• Recherche des sprites• Attaque au corps à corps

Nous avons utilisé Github pour gérer les versions de notre code et l'ajout de nouvelles fonctionnalités. Sans oublier Discord, notre moyen de communication à l'extérieur des séances de cours.

Nous avons organisé des réunions régulières pour discuter de l'avancement du projet et de la répartition des tâches. Ces réunions ont également été l'occasion de prendre des décisions importantes concernant les fonctionnalités à implémenter ou les problèmes rencontrés. En dehors des réunions, nous avons communiqué constamment pour échanger des informations, des mises à jour et annoncer les nouvelles versions améliorées de code résolvant les différents bugs rencontrés.

La communication a été un élément clé de notre organisation. Nous avons veillé à ce que chaque membre de l'équipe soit informé de l'état du projet et des tâches accomplies. Cela nous a permis de travailler efficacement en équipe et de résoudre rapidement les problèmes qui se sont présentés. Grâce à cette organisation rigoureuse, nous avons pu réaliser le projet dans les délais impartis et atteindre la majorité des objectifs fixés.

Concernant l'organisation du travail en tant que tel, au début du projet, nous nous sommes séparés en groupes de deux afin d'avancer le plus efficacement possible et de vérifier que les idées du projet étaient claires. Puis vers la fin du projet, nous avons effectué nos parties du travail seuls car les bases du projets étaient solides et chacun connaissait la direction que devait prendre l'ensemble du projet.

4 Développement

Dans cette quatrième partie, nous vous parlerons des techniques de développement que nous avons mis en place, de nos idées et des problèmes rencontrés. Nous verrons le développement de la carte du jeu (map), des entités, des objets et des inventaires, de l'arbre de compétence, des systèmes d'interactions, des systèmes de sauvegarde et de chargement puis la recherche et la conception des ressources.

4.1 Maps

La map se partage en plusieurs parties : le niveau, les étages et les salles. Au vu du peu de temps que nous avions pour mettre en place un jeu qui soit à la fois fonctionnel et esthétique, nous sommes partis sur le système de map suivant : le jeu est composé en réalité d'un seul et unique niveau. Ce niveau est composé, lui-même, de trois étages, chacun possédant huit à douze salles générées aléatoirement avec à la fin une salle de boss. Dans cette partie, nous allons donc vous parler des générations de ces trois éléments.

4.1.1 Structure des salles

Chaque salle comprend plusieurs attributs dans sa structure. C'est-à-dire trois variables entières permettant de connaître le nombre de monstres présents dans la salle, le numéro de la salle ainsi que le nombre de portes. On y retrouve également une matrice *dim* de taille vingt-cinq par vingt-cinq, un statut pouvant prendre les valeurs *COMMUN*, *START* ou *EXIT*, et un tableau de pointeurs sur des monstres.

Pour initialiser chaque salle, on fait appel à une fonction qui génère celle-ci. Chaque salle est composée de plusieurs éléments tels que des murs, des portes, des monstres. La génération de la salle se découpe en plusieurs étapes : la première consiste à matérialiser les murs. Nous avons délimité le contour de la matrice *dim* par une valeur numérique représentant les murs. Ainsi chaque côté de cette matrice est délimité, et nous permettra de définir les limites de la salle et du déplacement des entités. La deuxième étape consiste à incruster, dans les murs, les portes en fonction du nombre de salles présentes autour de celle courante. Pour cela, on remplace la valeur numérique du milieu des murs concernés par celle de la porte (*PORTE*).

Cependant, cette gestion des portes n'est pas réalisé dans un cas particulier : lorsque l'on arrive sur la génération de la dernière salle (cf. FIGURE1). Pour celle-ci, on regarde autour de cette dernière salle, quelle est l'avant-dernière et on met une porte entre les deux. De ce fait, il faudra également vérifier pour toutes les autres salles si elle est son adjacente à la dernière et dans ce cas, ne pas mettre de porte entre les deux. La troisième étape de la génération consiste à placer des obstacles. Le nombre d'obstacles est déterminé aléatoirement entre trois et douze. Les obstacles sont placés sur la matrice de la salle aux endroits vides. Afin d'éviter d'entraver la progression du joueur, nous avons choisi de laisser au minimum une case d'espacement entre les obstacles et les murs. Ces obstacles seront utiles pour rajouter de la mixité au décor de la salle, la rendant moins monotone. La quatrième et dernière étape finalise la construction de la salle en y ajoutant les monstres qui s'y trouveront. Pour savoir quel type de mob on doit insérer dans la map, on parcourt le tableau *tab_mob*, en fonction de l'étage dans lequel le personnage se trouve. Pour le premier étage, on utilisera les quatre premiers types, pour le deuxième étage, les quatre suivants puis pour

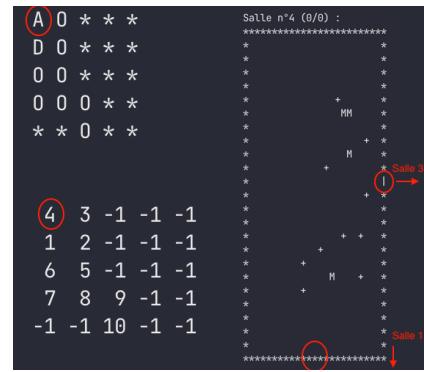


FIGURE 1 – Position salle *EXIT* et positionnement des portes
Légende : * ⇒ mur, | ⇒ porte

le troisième étage, les quatre derniers, auxquels on initialisera l'inventaire qui sera transféré dans celui du personnage lors de la mort du monstre (cf. 4.3).

Après que l'ensemble des salles ait été créé, le personnage peut se déplacer dans celles-ci jusqu'à atteindre la salle du boss. Cette salle diffère légèrement des autres. En effet, sa première particularité est son emplacement dans la matrice *etage*, puisqu'elle remplace la salle de sortie. La dernière salle du labyrinthe ne sera jamais atteinte puisque celle-ci est juste une salle dite "tampon". Dès que le personnage arrivera dans cette salle, il sera immédiatement transporté dans la salle du boss. La deuxième particularité de cette salle est sa taille. La salle du boss possède une longueur plus importante permettant de rendre les déplacements plus facile pour le joueur mais aussi pour le boss. Cette salle est également équipée d'obstacles ayant une position particulière, tous alignés formant un passage en direction du boss et séparés, permettant le passage entre chaque. Elle possède également une porte située juste derrière le boss qui s'ouvre uniquement si celui-ci est tué, laissant place à une salle de transition dans laquelle se trouve un marchand (cf. 4.5.2) qui vous permettra au joueur de débloquer des compétences (cf. 4.4.1). La petite salle possède au bout de celle-ci un escalier emmenant directement dans la première salle de l'étage suivant. La salle du boss est générée par une fonction qui alloue dynamiquement la mémoire pour la salle et lui implémente les murs, les obstacles et le boss. La salle du marchand est, quant à elle, générée par l'appel d'une fonction qui réalise les mêmes actions que pour la salle du boss mais remplace celui-ci par le marchand.

4.1.2 Structure des étages

Le niveau est caractérisé par sa structure *t_niv* possédant comme seul attribut un tableau de taille trois de type *t_etage*. Ce tableau nous permettra de générer les trois étages nécessaires au jeu. Les étages possèdent dans leur structure le numéro de l'étage (un, deux ou trois), le nombre de salles à générer, qui est assigné de manière aléatoire à une valeur comprise entre huit et douze, une matrice de taille cinq par cinq d'éléments de type *t_salle*, un pointeur sur un type *t_salle* correspondant à la salle du boss, un autre pointeur sur un type *t_salle* correspondant à la salle du marchand et deux entiers vérifiant chacun si les salles du boss et du marchand sont débloquées ou non.

Lors de la création du niveau, on génère le nombre de salle puis on vérifie que l'étage est conforme. Pour qu'un étage soit conforme il faut que toutes les salles aient été générées et qu'elles ne dépassent pas les limites de la matrice. Pour cela, la fonction de génération du niveau fait appel à la fonction de vérification de conformité. Celle-ci trouve la première salle grâce à l'appel d'une fonction qui parcourt une à une les salles de l'étage et cherche celle possédant le premier numéro. Si la salle est trouvée, on retourne alors ces coordonnées. Dans le cas contraire on retourne un marqueur d'erreur (ici -1). Après cela, nous vérifions, grâce à une fonction, que l'ensemble des salles a été généré. Celle-ci retourne alors un si notre indice de parcours est égal au nombre de salle qu'il fallait générer et zéro sinon. Si la conformité de l'étage n'est pas validé, on recréé l'étage en question et le chemin des salles.

La génération des étages commence par mettre à moins un de tous les numéros de salle de la matrice de l'étage. On va, par la suite, générer des coordonnées x et y de manière aléatoire. Ces valeurs seront comprises entre zéro et quatre. Puis on va assigner à la salle aux coordonnées x et y le numéro un, il s'agit de notre salle de départ. On va, par la suite, générer l'ensemble des salles de l'étage, à partir du numéro de la dernière salle positionnée jusqu'à obtenir l'ensemble des salles à générer. Pour générer les salles suivantes, on va récupérer les coordonnées de la salle précédente dans deux variables.

A partir de cela, un cas particulier à prendre en compte peut se présenter : Si la dernière salle générée est dans un coin ou n'a aucun voisin disponible, alors on lui assigne le statut de sortie *EXIT* (cf. FIGURE2). On va ensuite vérifier pour chaque salle, à partir de la première, si elle a au moins deux cases libres autour d'elle et que ces cases sont dans la matrice. Si c'est le cas alors le

labyrinthe pourra terminer sa génération à partir de l'une des deux cases libres de la salle vérifiée, sinon on recherche la salle suivante et on réalise les mêmes vérifications jusqu'à obtenir une salle qui satisfera ces conditions.

Dans le cas général, on réalise une boucle autour des coordonnées sauvegardées de la précédente salle et celles de la nouvelle. Si elles sont égales c'est que la salle n'a pas pu être générée et qu'il faut réessayer la génération. Dans le cas contraire, tout s'est bien déroulé et la salle est créée. Pour la générer, on va choisir de manière aléatoire une direction entre gauche, droite, haut et bas. Ces directions nous permettront de choisir où sera positionnée la prochaine salle en fonction de la celle courante. Mais avant de générer cette nouvelle pièce, on doit vérifier que celle-ci est disponible. C'est-à-dire qu'elle n'est pas à l'extérieur de la matrice et qu'aucune salle n'est déjà assigné à cet endroit. Lorsque elle est générée, on lui assigne un statut en fonction de son numéro. Si ce n'est pas la première salle (*START*) alors on lui assigne le statut *COLUMN* et enfin pour la dernière salle ce sera le statut *EXIT*. Lorsque l'ensemble des salles est généré, on va alors appeler la fonction *genererSalle* pour chacune d'entre elles qui se chargera de générer la structure de la salle.

4.2 Entités

Comme nous l'avons vu précédemment, dans chaque salle il y a des monstres qui sont générés. Ceux-ci peuvent être appelés plus généralement sous le nom d'entités. Dans cette partie nous détaillerons alors qu'est-ce qu'une entité ; plus précisément nous parlerons du personnage, des monstres et des boss.

Voyons dans un premier temps les entités dans leur globalité. Les entités possèdent plusieurs attributs : un nom qui nous permet de différencier chacune d'entre elles, différentes statistiques (dégâts, vitesse d'attaque, vitesse de déplacement, périmètre de détection) et un inventaire qui leur est propre. De plus, les entités possèdent dans leurs structures un arbre de compétences. Ces compétences servent à améliorer les statistiques de l'entité. Enfin, nous avons deux variables qui servent pour la direction et pour l'identification. Celle-ci nous permet à savoir si l'entité est un joueur ou un mob . La direction se caractérise par haut, bas, gauche et droite.

Maintenant, chaque type d'entité (Mob, Boss ou personnage) possède plus ou moins de caractéristiques. Par exemple, les boss ne possèdent pas d'inventaire et les mobs ne possèdent pas d'arbre de compétences du fait que celui-ci est réservé pour le joueur. Ce qui implique alors que nous devons séparer la création de chaque entité.

Pour référencer chacune d'entre elles, nous avons fait le choix de créer des tableaux afin d'avoir les valeurs de chaque mob et boss en dur dans le code. Pour pouvoir accéder à ces valeurs, nous avons créé une fonction qui, grâce au nom du mob, nous retourne son indice dans ce tableau. Le système est exactement le même pour les boss.

Chaque type d'entité possède sa fonction de création qui lui est propre. Chaque fonction se base sur le même principe car elle se réfère aux valeurs des tableaux d'entités. Pour le personnage, nous avons décidé de lui attribuer des statistiques de base. Pour la création du nom du personnage, celui-ci est saisi dans le menu principal du jeu lorsque le joueur souhaite commencer une nouvelle partie.

Pour que chaque entité puisse être détruite correctement, nous avons trois fonctions qui garan-

* * * 0 A	* * * * *
0 0 * 0 0	* 0 0 0 *
* 0 0 0 D	* 0 A D *
* * * * *	* 0 0 0 *
* * * * *	* * * 0 *
-1 -1 -1 4 5	-1 -1 -1 -1 -1
10 9 -1 3 2	-1 4 3 2 -1
-1 8 7 6 1	-1 5 8 1 -1
-1 -1 -1 -1 -1	-1 6 7 9 -1
-1 -1 -1 -1 -1	-1 -1 -1 10 -1

FIGURE 2 – Exemple de cas particulier de positionnement de la salle *EXIT*

tissent de n'avoir aucune fuite de mémoire concernant ces entités.

4.3 Objets et inventaire

Dans notre jeu, pour pouvoir avancer convenablement, il faut arriver à débloquer des compétences. Pour les débloquer, il faut des objets. Ces objets doivent être stockés quelque part. Pour cela, nous avons décidé de créer un inventaire.

Dans un premier temps, il faut définir ce qu'est un objet. Un objet est constitué d'un nom qui sert à le référencer et d'une description qui décrit l'objet en question. De la même manière que pour les entités, nous avons fait le choix d'un tableau de valeurs en dur (*tab_objets*) afin de connaître les attributs de chaque objet.

Dans un second temps, nous devons définir les attributs de l'inventaire. Pour cela nous utilisons une structure *obj_inv_t*. Celle-ci comporte deux tableaux de même taille définie par la taille de l'inventaire de chaque entité. L'un de ces tableaux nous permet de stocker concrètement les objets. Et l'autre est un tableau d'entiers contenant le nombre d'objets dans l'inventaire. Donc par exemple, prenons une entité qui possède trois cases d'inventaire avec dedans trois objets quelconques. Si on veut accéder au nombre d'objets de obj1 dans l'inventaire de l'entité. On utilisera l'indice zéro du tableau d'entiers.

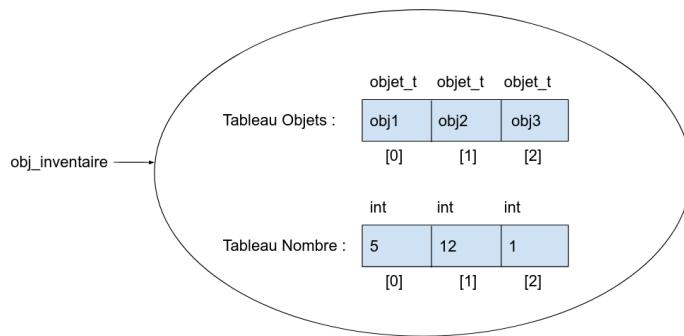


FIGURE 3 – Schéma de fonctionnement de l'inventaire

Comme nous pouvons le voir sur la FIGURE 3, si on veut accéder par exemple au nom de obj2, on utilisera l'indice 1 du tableau d'objet. Si on veut accéder au nombre d'objets de obj2 on utilisera l'indice un dans le tableau d'entiers. Pour connaître l'indice de l'objet, on utilise le même principe que pour les mobs. On utilise une fonction qui, avec le nom du mob, nous retourne son indice dans le tableau d'objets.

Enfin, il faut initialiser l'inventaire. Le principe est différent selon si c'est l'inventaire d'un monstre ou l'inventaire d'un personnage. Pour initialiser l'inventaire du joueur, on prend chaque objet de *tab_objet* et on le met à zéro dans le tableau de nombre d'objets du joueur. Autrement dit dans l'inventaire du joueur, il y a tout les objets possibles initialisé à 0. Pour les mobs, on récupère son indice et on initialise un plus petit inventaire qui ne contiendra que les objets que les mobs pourrons nous donner.

Afin de pouvoir transférer l'inventaire d'un mob quand il est tué par un joueur, nous devons utiliser une fonction, qui, pour chaque objet dans l'inventaire du monstre, va regarder l'indice de l'objet i, puis ajouter le nombre d'objets de l'inventaire du mob dans l'inventaire du joueur.

4.4 Arbre de compétences

Pour faire progresser le personnage en même temps que les monstres présent dans le jeu, nous avons imaginé des arbres de compétences permettant d'améliorer des statistiques du joueur, au profit d'autres. Ce système garantit au joueur une certaine personnalisation de son aventure de son propre gré. Nous avons donc choisi de représenter quatre classes avec leur arbre (cf. FIGURE 4) : l'archer, le mage, le guerrier et l'assassin. Chaque classe à ses points forts et ses faiblesses. Dans cette section nous verrons, dans un premier temps, la création des listes de compétences et dans un second temps, nous verrons comment ces compétences sont introduites dans les différents arbres de compétences.

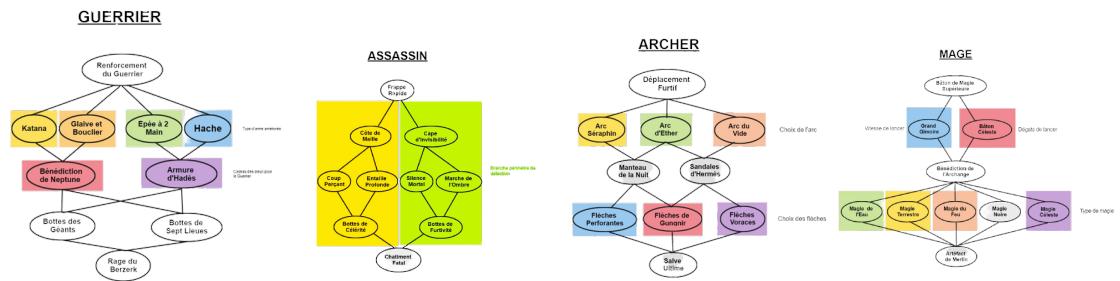


FIGURE 4 – Arbre des compétences pour chaque classe

4.4.1 Compétences

Les compétences du jeu sont toutes représentées par une structure `t_competence` possédant une variable vérifiant si la compétence a été acquise, une chaîne de caractère comme nom, une chaîne de caractère comme description, une structure de buffs de compétences qui seront ajoutées aux caractéristiques du joueur. On retrouve également dans sa structure un nombre de compétences suivantes, avec un tableau de pointeurs sur ces compétences, un nombre de compétences précédentes avec un tableau de pointeurs sur ces compétences. Les compétences sont aussi caractérisées par leur nom d'image qui sera utilisé pour afficher l'image de la compétence dans le menu, un pointeur sur une structure possédant les objets et leurs quantités nécessaire pour débloquer la compétence, avec un nombre entier définissant le nombre de types d'objets différents qui sont nécessaires, puis un numéro d'étage de la compétence dans l'arbre.

Quatre listes de dix éléments de type `t_competence` regroupent les compétences de jeu : `cpt_assassin`, `cpt_mage`, `cpt_archer` et `cpt_guerrier`. Pour les initialiser, on fait appel à la fonction `init_competences`. Cette fonction a pour objectif de remplir les zones vides ou à NULL des différentes compétences. Son premier rôle ainsi sera de compléter le nom de l'image de la compétences. Pour cela, la fonction `remplissage_nomImgCpt` est appelée et alloue dynamiquement la mémoire pour le chemin de l'image. Ensuite, selon le type de la classe rentrée en paramètre, on copie la chaîne du chemin dans le nom de l'image à laquelle on va concaténer les majuscules et les nombres ou numéros présents dans le nom de la compétence avec l'extension "._l.png" pour sélectionner l'image foncée de la compétence puisque celle-ci est bloqué (lock). Prenons l'exemple de la compétence "Marche de l'Ombre" du tableau de compétences `cpt_assassin`, on obtiendra le chemin : `../img/competences/assassin/MO_l.png`.

La prochain étape est le chaînage des compétences suivantes et précédentes (cf. FIGURE 5) en même temps que l'initialisation des objets nécessaires pour débloquer la compétence grâce à l'appel de la fonction *init_obj_necessaires*. Cette fonction alloue dynamiquement le pointeur sur

la structure possédant le tableau des éléments et le tableaux des quantités de ces éléments. Elle alloue également ces tableaux grâce au nombre de type d'objets différents inscrits dans la structure de la compétence. Ensuite, en fonction du type de la classe rentrée en paramètre et de l'indice, les emplacements des objets nécessaires et leur quantité sont remplis statiquement. L'initialisation se poursuit avec l'ajout des compétences suivantes et précédentes pour chacune d'entre elles. Pour cela, on alloue dynamique les tableaux des pointeurs sur compétences en fonction du nombre de précédents et de suivants connus pour chacune d'entre elles. On va incrémenter un indice qui va parcourir l'ensemble des compétences et remplir les tableaux. Cependant, un cas particulier peut se présenter à ce moment là. En effet, pour le tableau *cpt_guerrier*, les liaisons entre les compétences aux indices cinq et six permettent toutes deux de débloquer les compétences aux indices sept et huit. Il faut donc que notre indice décrémente ou incrémenté d'avantage en fonction du remplissage de ce tableaux de compétences précédentes ou suivantes.

4.4.2 Arbres

Les arbres de compétences permettent d'illustrer de manière plus compréhensive le fonctionnement de tableaux pointeurs de compétences inclus dans la structures des différents arbres. On y retrouve également le type de la classe et un tableau de types *t_acquis* correspondant aux étages de l'arbre en question. Ce tableau sera utilisé lors du déblocage des compétences que nous verrons ultérieurement dans cette partie. Chaque arbre de compétences, de type *t_arbre*, est lié ainsi à une classe et à un tableau de compétences.

Pour les initialiser, on fera appel à la fonction *init_arbre*. Celle-ci alloue dynamiquement l'espace mémoire pour l'arbre, le tableau de pointeurs sur compétence et assigne à la classe de l'arbre celle passée en paramètre de la fonction parmi **aucune_classe**, **assassin**, **archer**, **guerrier** et **mage**. La prochaine étape de l'initialisation de l'arbre concerne le tableau d'étage de compétence acquises. En effet, pour les cinq étages la valeur **non_acquis** sera assigné. La fonction va ensuite donner à chaque pointeur sur compétences de l'arbre l'adresse de compétences à l'indice *i* dans le tableau de compétences passé en paramètre. A ce moment là, les compétences ne sont pas initialisées. Pour cela, la fonction réalise une dernière étape consistant à appeler la fonction *init_competence* (cf. 4.4.1) en lui donnant les compétences et la classe de l'arbre.

Maintenant que les quatre arbres sont initialisés, le joueur peut alors débloquer la première compétences d'un des arbres lorsqu'il arrivera dans la première salle de marchand (cf. 4.5.2). Pour débloquer une compétences plusieurs cas doivent être vérifiés :

- Le personnage possède suffisamment de ressources dans son inventaire.
- La précédente compétence est déjà débloquée sauf première compétence de l'arbre.
- La compétence que le personnage souhaite débloquer est bien dans celui pointé dans sa structure (cf. 4.2).
- La compétences que le joueur cherche à débloqué n'est pas sur le même étage qu'une compétences déjà débloquée.
- La compétence n'a pas été déjà acquise.

Si l'une de ces condition n'est pas respectée, la compétence ne sera pas déblocable et le joueur reviendra sur le menu des compétences et l'image restera inchangée. Maintenant que nous avons

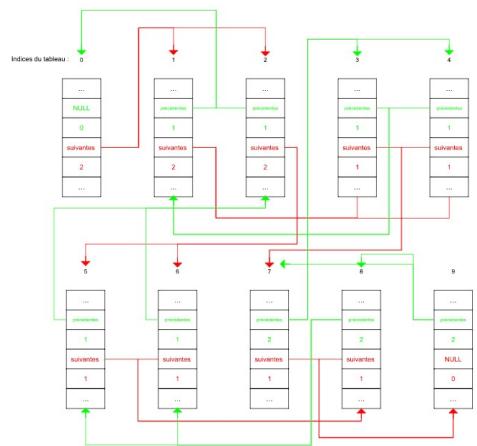


FIGURE 5 – Exemple de chaînage de compétences avec le tableau *cpt_assassin*

détaillé les différentes conditions de déblocage d'une compétences, voyons comment ce système de gestion de déblocage est géré. Lorsque le joueur arrive le menu des compétences, il a la possibilité de cliquer sur l'une des quatre classes du jeu, laissant apparaître l'arbre associé à cette classe. Le joueur peut ensuite choisir l'une des compétences et apercevoir les améliorations de statistiques que propose celle-ci. Chaque classe possède deux atouts majeurs. Par exemple, l'arbre du guerrier propose des améliorations concernant soit la vie, soit les dégâts. Le joueur peut également soit revenir en arrière grâce à un bouton en bas de la page, soit débloquer la compétence si cela est possible. Pour vérifier les conditions du déblocage de la compétence, le menu fait appel à la fonction *competence_debloquer*. Le premier rôle de cette fonction est de vérifier si l'arbre du joueur est vide ou pas. Si c'est le cas et que le joueur veut débloquer la première compétence d'un arbre alors l'arbre de la compétences sera pointé de manière définitive par le pointeur sur arbre présent dans la structure du joueur. La fonction va ensuite appeler celle de vérification des conditions pour débloquer la compétence : *peut_debloquer_cpt*. Celle-ci vérifie que les conditions énoncées précédemment sont bien respectées. Dans ce but, elle va utiliser les fonctions *one_preced_cpt_debloq* qui recherche si l'une des compétences précédentes est débloquée et *cpt_in_arbre_joueur* qui parcours l'arbre du joueur pour vérifier que la compétence provient bien de celui-ci, et va réaliser des comparaison avec le statut de la compétence (acquise ou non) et avec le tableau de statut d'acquisition de compétence en fonction de l'étage où se situe la compétence dans l'arbre. Lorsque ces conditions sont vérifiées et valides, on parcours le tableau d'objets nécessaires. Pour cela, on récupère l'indice dans le tableau *tab_objet* de l'objet nécessaire à l'indice *i* du tableau d'objets nécessaires. Puis on vérifie que le joueur possède suffisamment de cet objet dans son propre inventaire. Si c'est le cas, toutes les vérifications sont passées et la fonction renvoie alors un, témoignant que la compétence est déblocable. De retour dans la fonction de déblocage, si après vérification, la compétence peut être déverrouillée, alors plusieurs action sont réalisées :

- Changement du nom de la compétence pour changer l'image et la faire apparaître en clair.
- Ajout et multiplication des avantages de compétence aux statistiques du joueur.
- Suppression des objets utilisés pour débloquer la compétence.

Maintenant que la compétence est acquise, le bouton de déblocage disparaît afin de garantir une sécurité supplémentaire autour du double déblocage d'une compétences. Cependant, si au bout des vérification le retour est différent de un, on affiche un message d'erreur dans le terminal.

4.5 Système d'interactions

Dans «Highway To L2», le joueur est confronté à plusieurs sources d'interactions. Ne serait-ce que pour se mouvoir ou lancer des attaques au monstres qui l'entourent. Il est également nécessaire pour son progrès de garder un œil sur son inventaire ainsi que d'améliorer ses compétences. Cette partie dévoile la manière dont nous avons offert au joueur la possibilité d'interagir avec l'environnement pour faire évoluer son expérience de jeu. Tous les moyens mis en œuvre impliquent l'utilisation de la SDL, que ce soit pour générer du son, pour la gestion des contrôles, ainsi que l'affichage.

4.5.1 Attaques

Il existe quatre classes : archer, assassin, guerrier et mage. Ces classes se distinguent en deux groupes qui définissent la manière dont ils interagissent pour lancer des attaques. La première catégorie est composée de l'assassin et du guerrier, ce sont des personnages qui se battent aux corps à corps. Tandis que la seconde catégorie, composée de l'archer et du mage, nécessite un système d'attaque à distance. Il est donc nécessaire de prendre en compte le choix de la classe du personnage. Ce choix s'effectue au moment où le joueur débloque sa première compétence dans

l'arbre qu'il choisit. Le temps que le joueur débute et récupère les objets nécessaires à l'achat de sa première compétence, il utilise une attaque par défaut de corps à corps.

Quand un assassin ou un guerrier souhaite attaquer, le programme fait appel à la fonction *attaque_cac*. Lorsque un archer ou un mage souhaite attaquer, le programme fait appel à la fonction *attaque_proj*. Un pointeur sur fonction *fonc_attaque* devra être utilisé pour référencer une de ces deux fonctions en conséquence. Elles doivent donc avoir la même signature. Comme évoqué plus haut, ce pointeur sur fonction sera initialisé avec *attaque_cac*.

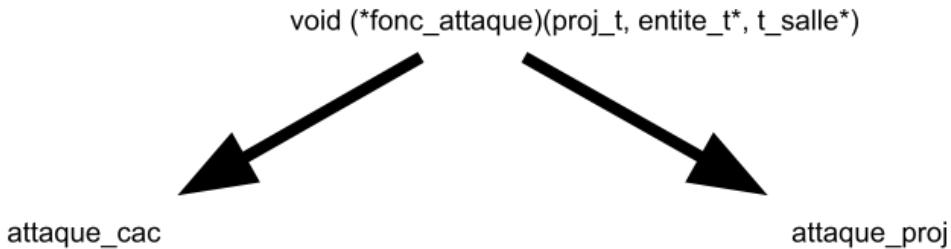


FIGURE 6 – Pointeur sur fonction d'attaque

La fonction *attaque_cac* garde un fonctionnement assez basique par rapport aux projectiles. Il suffit de vérifier la présence d'un mob dans la case adjacente concernée par la direction dans laquelle se trouve le personnage et on appelle une fonction qui applique les dégâts de l'attaque au mob concerné. Si la vie contenu dans la structure entité du mob est inférieur ou égale à zéro, on appelle la fonction *looter* pour récupérer les objets de l'entité mob puis on procède à sa destruction.

Les attaques à distance sont quant à elles plus délicates à gérer. Tout d'abord, il s'agit de définir ce qu'est un projectile. C'est dans la structure projectile *proj_t* que nous retrouvons ses paramètres, que sont :

- **Vitesse de déplacement** : *vitesse_depl*, entier représentant la vitesse du projectile.
- **Portée** : *portee*, entier représentant la portée du projectile.
- **degats** : *degats*, entier représentant les dégâts du projectile.
- **Touché** : *touche*, booléen qui vaut **true** quand une cible est atteinte.
- **Coordonnées en x précédentes** : *xp*, entier.
- **Coordonnées en y précédentes** : *yp*, entier.
- **Coordonnées en x actuelles** : *x*, entier.
- **Coordonnées en y actuelles** : *y*, entier.
- **Direction du projectile** : *dir*, *t_dir*.

Les projectiles sont créés en appelant la fonction *creer_projectiles*. Cette fonction alloue dynamiquement la place mémoire d'un projectile. En récupérant un type enum *proj_t*, elle initialise les valeurs de *vitesse_depl* et de *portee* avec les valeurs contenus dans un tableau de statistiques projectiles suivant sa nature (flèche pour archer, et boule pour mage).

Chaque projectile lancé, puisque le personnage peut en lancer plusieurs à la fois, est traité dans une liste. Cette liste est composée de pointeurs sur *element_t*. Un élément caractérise un pointeur sur projectile contenant un de ceux en mouvement, un pointeur sur l'élément suivant, et de même pour le précédent.

La liste est mise à jour à l'aide de la fonction *maj_proj*. Cette fonction calcule la trajectoire du projectile pour chaque nouveau rafraîchissement d'écran en appelant la fonction *calcul_position*.

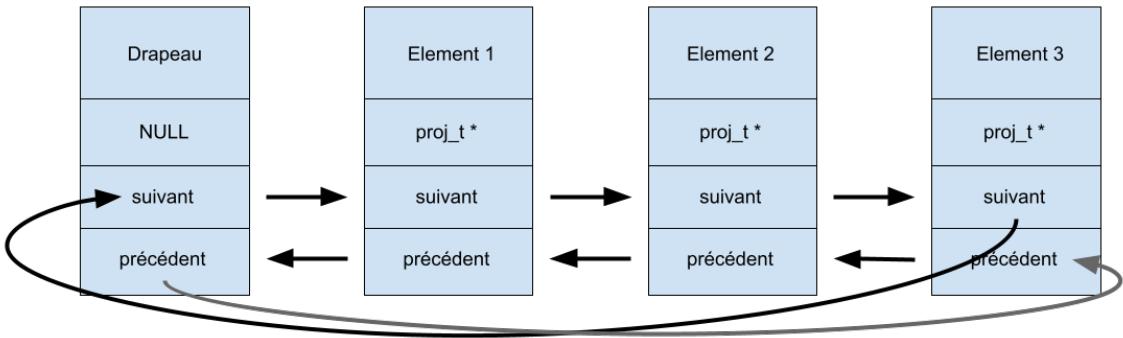


FIGURE 7 – Liste de projectiles

Cette dernière sauvegarde alors les coordonnées initiales dans *xp* et *yp* puis incrémente ou décrémente les coordonnées en *x* ou *y* en fonction de la direction *dir* du projectile. Lorsque *calcul_position* a fini de mettre à jour ces projectiles, on effectue une boucle en prenant comme repère les coordonnées *xp* et *yp* jusqu'à ce qu'elles soient égales à *x* et *y*. On suit ainsi le voyage du projectile à travers la salle. Pour faire disparaître un projectile, il suffit de vérifier au cours de cette boucle si il est rentré en collision en mettant à **true** le flag *touche*. Si c'est avec un mob, on appelle la fonctions «*degats*» qui fonctionnera ensuite comme pour le corps à corps, puis on détruit le projectile avec la fonction *détruire_projectiles*.

4.5.2 Marchand

Afin de pouvoir débloquer les compétences de l'arbre, il nous faut une interface graphique. Ainsi, nous avons créé différents menus :

- Une interface pour choisir l'arbre
- L'arbre en question
- Une compétence en fonction de celle choisie

Au niveau de l'affichage d'une compétence, on remarque qu'il y a le bouton débloquer qui permet de débloquer la compétence choisie (cf. 4.4.2). Il y a également des chaînes de caractère afin de décrire les buffs qu'octroie la compétence et les items requis pour la débloquer. Pour construire ces chaînes, on va passer par plusieurs étapes. Prenons l'exemple de la création d'une chaîne pour le nombre requis d'items. On commence par récupérer les informations requises : Le nom de l'item, la quantité requise et la quantité dans l'inventaire du joueur. Puis on construit la chaîne de la façon suivante grâce à l'ajout dans une chaîne avec la fonction prédéfinie *strcat* de la librairie «*string.h*». On obtient ainsi le résultat suivant : *nom_objet : quantite requise / quantite dans l'inventaire*.

4.5.3 Interface de l'inventaire

Afin que le joueur puisse regarder le nombre d'items qu'il a récupéré, nous avons intégré la fonction *inv* qui gère l'affichage de l'inventaire d'un personnage dans une interface graphique en utilisant la bibliothèque *SDL*. Elle crée une fenêtre de dimensions spécifiques et charge les textures des items pour afficher des rectangles représentant chaque objet de l'inventaire. Lorsque l'utilisateur clique sur un rectangle, la description de l'objet correspondant s'affiche à côté de celui-ci. Si cette description est affichée depuis plus de deux secondes alors elle est effacée de l'écran. La fonction

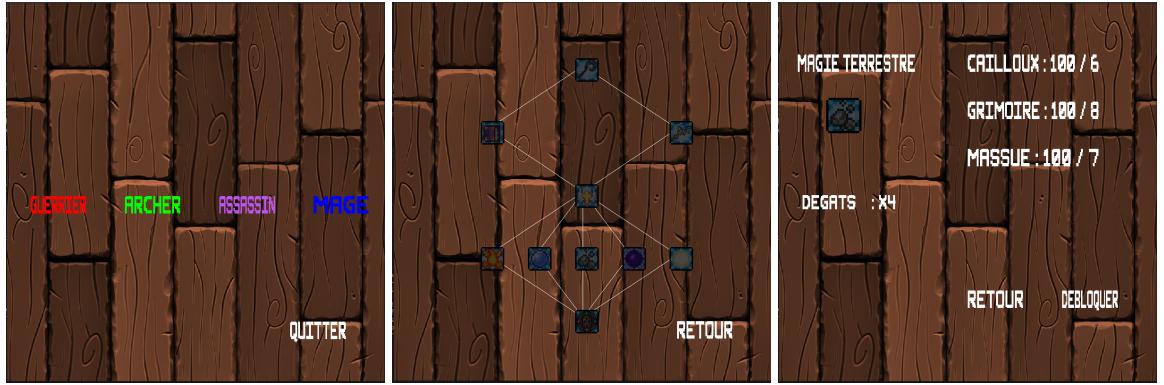


FIGURE 8 – Exemples du menu du marchand

utilise également une image de fond pour améliorer l'esthétique de l'interface. Lorsque l'utilisateur veux quitter l'inventaire, il suffit d'appuyer sur le bouton «TAB».



FIGURE 9 – Interface SDL de l'inventaire du personnage

4.6 Système de sauvegarde et de chargement

Lorsque le joueur lance le menu, trois choix se présentent à lui :

- **New game** : le joueur peut lancer une nouvelle partie.
- **Continue** : continuer une partie déjà enregistrée.
- **Quit** : quitter le jeu.

Le menu est implémenté par une fonction qui initialise et lance la sortie audio du menu. Cette fonction s'occupe également d'appeler une autre fonction qui gère l'affichage du menu. Après que l'appel de cette fonction et des actions internes aux fonctions d'interactions du menu et de nommage du personnage soient terminé, la fonction de gestion du menu éteint et quitte la sorte audio.

Lorsque le joueur clique sur **New game**, une nouvelle page s'affiche l'invitant à saisir un nom pour son personnage (cf. FIGURE 10a). Pour cela, les rectangles (appelés une «rects») de la page du menu sont détruites pour laisser place à celles de la saisie du nom. Mais la saisie du nom demande quelques vérifications avant de pouvoir être validée. En effet, le nom ne peut pas dépasser un nombre de caractères maximum garantissant le fait que la saisie n'est pas en dehors de la page. Celle-ci ne doit également pas être vide. Pour cela et afin d'être certains qu'il n'y ait



(a) Menu de saisie du nom du nouveau joueur (b) Différents états du bouton **Continue** du menu

FIGURE 10 – Images des menus

pas de problème de saisie, la taille de la saisie doit être supérieure à zéro et ne pas être vide. Si le nom respecte les deux conditions énoncées précédemment, la fonction *name* retourne alors le nom qui sera assigné directement au personnage. Si le joueur souhaite tout de même revenir sur le menu avant de saisir le nom, il le peut grâce à un bouton **back**. Pour cela, les «rects» de la page de saisie sont détruites pour laisser place à celles du menu créées par l'appel de la fonction responsable du menu.

La première sauvegarde de la partie se réalise lorsque le joueur arrive dans la première salle du premier étage, c'est-à-dire au tout début du jeu. Cette sauvegarde permet au jeu d'enregistrer le nom du personnage et permet en cas de décès dans ce premier étage de pouvoir recommencer en première salle avec le même nom de personnage. Cette première sauvegarde écrasera les données présentes, si il y en a, dans le fichier de sauvegarde *sauvegarde.txt*. Les prochaines sauvegardes se feront automatiquement lorsque le joueur arrivera dans la salle du marchand (cf. 4.5.2). La sauvegarde ne conserve pas toutes les données du joueurs. En effet, le fichier de sauvegarde est composé des éléments suivant : le nom du personnage, le numéro de classe (s'il en a, sinon -1), le nom des compétences déjà acquises et ajoute le drapeau de fin **END_OF_CPT** si le personnage en possède sinon le marqueur **None** sera sauvegardé. La sauvegarde comprend également le numéro de l'étage courant et la quantité de chaque objet du jeu présent dans l'inventaire du joueur.

Concernant le bouton de **Continue**, deux états différents peuvent lui être attribués (cf. FIGURE 10b). Si le fichier de sauvegarde existe et qu'il n'est pas vide, le bouton s'affichera en clair et sera cliquable. Sinon, celui-ci sera foncé et non cliquable. Le chargement du fichier de sauvegarde se réalise en plusieurs étapes. Dans un premier temps, on essaie d'ouvrir le fichier. Dans le cas où on ne peut pas, on affiche un message d'erreur, dans le cas contraire on récupère le nom du personnage en première ligne. Dans un second temps, on va récupérer le numéro de classe du personnage (cf. 4.4.1). Lors de la première sauvegarde le personnage n'a pas de classe. Il faut donc prendre en compte que le personnage peut ne pas avoir encore obtenu de classe. C'est pour cela que le marqueur moins un permet de gérer ce cas. Dans un troisième temps, on va récupérer les compétences du personnage. Lorsque le joueur débute une nouvelle partie, il n'a pas de compétence c'est pour cela qu'à la première sauvegarde, le marqueur **None** sera sauvegardé. Dans le cas où le joueur possède des compétences, on va lire le nom des compétences qui auront été sauvegardées jusqu'à obtenir le drapeau de fin **END_OF_CPT**. Lorsque les compétences sont lues, la fonction *appliquer* est appelée et associe au personnage les buffs de statistiques qui sont attribués à celle-ci. Dans un quatrième temps, on récupère le numéro de l'étage à partir duquel le personnage doit repartir. Puis dans un dernier temps, on récupère la quantité sauvegardée pour chaque objet du jeu. Ces quantités seront appliquées à l'inventaire du joueur.

Le dernier bouton présent sur le menu est le bouton **Quit**. Celui-ci permet de quitter la page du

menu et de fermer le jeu. Pour cela, on libère toutes les mémoires allouées que ce soit la fenêtre, le rendu... et on quitte SDL.

Enfin, au niveau de la sauvegarde, nous avons fait en sorte que le joueur ne puisse pas la modifier pour tricher. Pour cela, nous avons chiffré la sauvegarde en AES 128 qui est un chiffrement par bloc. Puis nous avons fait en sorte que le jeu vérifie ce chiffrement. Grâce à la fonction de hashage SHA-256 nous vérifions si le fichier n'a pas été modifié.

4.7 Recherche et conception des ressources

Les ressources utilisées peuvent être divisées en quatre catégories :

- Les sons
- Les musiques
- Les images
- Les polices d'écriture

Dans le dossier *music*, on retrouve les musiques qui seront jouées dans le jeu à différents moments. On peut alors y retrouver la musique du menu de lancement du jeu et les musiques de boss qui seront lancées lorsque le personnage arrivera dans la salle du boss. La musique du menu a été récupérée du jeu "Dead Cells". Celles des deux premiers boss viennent d'Elden Ring [1] et la musique du boss final a été composée par Carl Orff et s'appelle Carmina Burana[2].

Le dossier *img* est décomposé en plusieurs sous-dossiers en fonction de l'utilisation des images. Ces sous-dossiers sont au nombre de dix, on y retrouve les images de fond et décors des menus, les images de boutons du jeu, les images des compétences, les décorations diverses telles que les sprites et les textures. L'ensemble de ces images ont été récupérées sur le site « The Sprites Resource » [3] [4].

Le dossier *font* contient quant à lui la seule police d'écriture utilisée dans le jeu.

5 Conclusion

En conclusion, «Highway to L2» est un jeu vidéo d'action-aventure en 2D, où le joueur doit parcourir différents niveaux en affrontant des ennemis et des boss, tout en collectant des objets et en améliorant ses compétences. Le jeu est conçu pour être accessible pour tous les âges, avec une courbe de difficulté progressive et des commandes simples et intuitives.

Le développement du jeu a impliqué plusieurs étapes clés, notamment la définition du concept de base, la conception de niveaux, la création de graphismes et de sons, ainsi que la programmation des mécaniques de jeu et des fonctionnalités tout au long du processus de développement.

Enfin, «Highway to L2» tire parti de nombreuses ressources pour créer un univers cohérent et immersif, notamment des images, des sons et des musiques provenant de différents jeux et sites Web. Grâce à cette combinaison de facteurs, le jeu offre une expérience engageante et stimulante pour les joueurs de tous horizons.

5.1 Résultats

Le développement de «Highway to L2» a été mené avec pour objectif de créer un jeu d'action en 2D en utilisant la bibliothèque SDL2 en langage C. Globalement, ces objectifs ont été atteints grâce à une méthodologie rigoureuse, un travail d'équipe efficace et une planification minutieuse dû au diagramme de Gantt mis en place.

Les principales fonctionnalités attendues ont été implémentées avec succès, notamment le mouvement, l'inventaire, l'arbre de compétences, les ennemis, les objets à collecter, les effets sonores et musicaux, ainsi que les différents étages avec leurs propres thèmes visuels.

Cependant, certaines fonctionnalités n'ont pas pu être implémentées. Par exemple, la fonctionnalité de multijoueur n'a pas été intégrée dans le jeu, par manque de temps. Cela aurait pu offrir une expérience de jeu encore plus divertissante et interactive. De plus, la posture du personnage, qui aurait pu ajouter une dimension supplémentaire à la mécanique de combat, n'a pas pu être mise en place en raison de ces contraintes de temps.

5.2 Améliorations

- Mode multijoueur en ligne : en ajoutant la prise en charge du mode multijoueur en ligne, les joueurs pourront jouer ensemble et affronter d'autres joueurs en ligne cela donnerait un autre niveau de défi et de compétitivité au jeu.
- Mode multijoueur en local : en ajoutant la prise en charge du mode multijoueur local, les joueurs pourront jouer ensemble sur le même ordinateur.
- Ajout de nouveaux modes de jeu : il pourrait être intéressant d'ajouter de nouveaux modes de jeu pour offrir une plus grande variété de gameplay. Par exemple, un mode histoire avec différents niveaux et objectifs, un mode survie où les joueurs doivent survivre le plus longtemps possible contre des ennemis de plus en plus difficiles, ou encore un mode arcade avec des scores élevés et des défis chronométrés.

En somme, il y a beaucoup de façons d'améliorer «Highway to L2» en ajoutant des fonctionnalités multijoueur et de nouveaux modes de jeu. Cela augmenterait la jouabilité, la durée de vie et l'intérêt.

6 Bibiographie

Références

- [1] FROMSOFTWARE. *Youtube : ELDEN RING - All Boss Theme Songs OST*. URL : <https://www.youtube.com/watch?v=Ug2kIzGOnI4&t=4098s>.
- [2] Carl ORFF. *Carl Orff - O Fortuna Carmina Burana*. URL : <https://www.youtube.com/watch?v=GXF5K0ogeg4>.
- [3] The Spriters RESOURCES. *PC / Computer - Crypt of the necrodancer*. URL : https://www.spriters-resource.com/pc_computer/cryptofthenecrodancer.
- [4] The Spriters RESOURCES. *PC / Computer - Terraria*. URL : https://www.spriters-resource.com/pc_computer/terraria/.

7 Annexes

7.1 Débogage avec l'outil GDB

Dans cette partie nous aborderons un exemple de l'utilisation de GDB dans le cadre de notre projet

Dans notre jeu nous possédons des entités, ces entités nous devons les initialiser (4.2). Lors de l'initialisation, pour les boss nous avons rencontré une segmentation fault, nous utilisons alors l'outil de débogage GDB afin d'identifier d'où viens la segmentation fault.

```
Program received signal SIGSEGV, Segmentation fault.
strcpy_avx2 () at ../sysdeps/x86_64/multiarch/strcpy-avx2.S:593
593  .../sysdeps/x86_64/multiarch/strcpy-avx2.S: Aucun fichier ou dossier de ce type.
(gdb) ■
```

FIGURE 11 – Capture d'écran de la segmentation fault

Comme nous ne pouvons pas identifier précisément l'origine il va falloir rentrer un peu plus dans le code :

```
(gdb) break creer_boss
Breakpoint 1 at 0xcb8e: file src/entite.c, line 193.
(gdb) run
Starting program: /home/remy/Documents/L2/ProjetL2/bin/menu_cpt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
*      Test sur la création des entités :      *
-----  

Breakpoint 1, creer_boss (entite=0xfffff702f16f, nom=0x555555568e5b "Satan") at src/entite.c:193
193      int emplacement = acces_boss(nom);
(gdb) next
194      int taille_nom = strlen(tab_boss[emplacement].nom);
(gdb) next
196      strcpy(entite->nom, tab_boss[emplacement].nom);
(gdb) next
Program received signal SIGSEGV, Segmentation fault.
strcpy_avx2 () at ../sysdeps/x86_64/multiarch/strcpy-avx2.S:593
593  .../sysdeps/x86_64/multiarch/strcpy-avx2.S: Aucun fichier ou dossier de ce type.
(gdb) ■
```

FIGURE 12 – Capture d'écran du tracage avec next

Grâce aux commandes *break* qui permet de poser un point pour que le programme, et *next* qui permet nous arrivons à rentrer dans la fonction qui pose problème : *creer_boss*

Pour essayer de voir si il y a des problèmes dans nos variables on utilise la commande *print* afin de connaître le contenu de nos variables.

```
(gdb) run
Starting program: /home/remy/Documents/L2/ProjetL2/bin/menu_cpt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
*      Test sur la création des entités :      *
-----  

Breakpoint 1, creer_boss (entite=0xfffff702f16f, nom=0x555555568e5b "Satan") at src/entite.c:193
193      int emplacement = acces_boss(nom);
(gdb) print emplacement
$1 = 0
(gdb) next
194      int taille_nom = strlen(tab_boss[emplacement].nom);
(gdb) print emplacement
$2 = 2
(gdb) next
196      strcpy(entite->nom, tab_boss[emplacement].nom);
(gdb) print taille_nom
$3 = 5
(gdb) next
Program received signal SIGSEGV, Segmentation fault.
strcpy_avx2 () at ../sysdeps/x86_64/multiarch/strcpy-avx2.S:593
593  .../sysdeps/x86_64/multiarch/strcpy-avx2.S: Aucun fichier ou dossier de ce type.
(gdb) ■
```

FIGURE 13 – Capture d'écran du traçage avec print

En regardant les print des variables on remarque que il n'y a pas de problèmes à ce niveau. Le problème vient donc du fait que le nom de notre entité n'était pas alloué. GDB nous a permis donc ici d'isoler le problème et de découvrir la solution « indirectement »