

Le Mans Université
Licence Informatique 2ème année
Module 174UP02 Rapport de projet
VoidBorn
Lien vers le Github

Baptiste M, Lucas R, Nathan M, Ilann T

4 avril 2025

Table des matières

1	Introduction	3
2	Analyse	3
3	Organisation du Projet	3
3.1	Planning Prévisionnel	4
3.2	Répartition des tâches	4
4	Développement	5
4.1	Création des ressources	5
4.1.1	Objet	5
4.1.2	Sprites et Personnages	5
4.1.3	Tuiles et Biomes	6
4.2	Implémentation	6
4.2.1	Map	7
4.2.2	Personnages	7
4.2.3	Compétences	8
4.2.4	Objets	8
4.2.5	Monstres	9
4.2.6	Évènement	10
4.3	Réseau	10
4.3.1	Networking	10
4.3.2	Client	11
4.3.3	Serveur	12
4.4	Rendu Graphique	12
4.4.1	Menu Principal	12
4.4.2	Jeu	15
5	Résultats et conclusion	16
6	Annexes	17
6.1	Débogage	17

1 Introduction

Nous sommes un groupe de quatre étudiants à l'université du Mans. Notre objectif à été de créer un jeu en respectant un délai d'environ 3 mois. Lors de notre phase de réflexion sur le jeu, nous souhaitions créer un jeu multijoueur en deux dimensions à l'aide de la librairie graphique SDL2 en langage C. Lors d'une partie de notre jeu, deux équipes s'affrontent pour détruire la base adverse tout en protégeant la sienne. Nous nous sommes fixé comme contrainte de développer ce jeu en réseau.

2 Analyse

Dans le menu, chaque joueur choisit son pseudo et la classe de son personnage, lui conférant des capacités uniques qu'il peut utiliser au cours de la partie pour prendre l'avantage sur l'équipe adverse. Dans le menu, vous avez le choix entre être le serveur de la partie, être joueur d'une partie ou être les deux à la fois.

Après cela, vous arrivez dans une salle d'attente pour que les autres joueurs se joignent à vous. Une fois que la personne servant de serveur lance la partie, le jeu se lance et il n'est plus possible de rejoindre la partie, même si vous étiez dedans, que vous quittez le jeu et que vous tentez de vous reconnecter.

La map de notre jeu se décompose en plusieurs parties que nous appelons tuiles, chacune ayant un biome(neige, plaine, etc) choisi aléatoirement.

Les joueurs ne peuvent visualiser qu'une tuile du jeu à la fois. Cependant, il est possible de changer de tuile en passant par des portes placées à différents endroits aux bords de la tuile à la manière de The Binding of Isaac.

Des obstacles sont générés aléatoirement sur la map, limitant les déplacements du joueur. Des monstres sont également présents sur la map, se dirigeant vers le joueur le plus proche pour l'attaquer. Lorsque le monstre est tué il lâche un objet que le joueur peut rammasser et ajouter à son inventaire.

Chaque objet a une rareté et confère au joueur différents effets. Plus la rareté de l'objet est élevée, plus les effets bonus conférés au joueur sont importants.

Un système de combat est mis en place entre les joueurs d'équipes différentes lorsqu'ils se situent sur la même tuile. Les points de vie ainsi que les objets du joueur sont affichés sur son écran à la manière d'un inventaire Minecraft.

La gestion des entités dynamiques et des tuiles de la map sont gérées grâce à l'implémentation d'une liste générique. Le serveur communique avec les clients en envoyant toutes les informations nécessaires à l'affichage du jeu par le client qui renvoie des informations au serveur en fonction des changements opérés par le joueur.

3 Organisation du Projet

Le groupe a organisé le travail autour de différents outils pour contrôler l'avancement du projet. Le dépôt git sert à stocker les fichiers. Un document partagé Google Drive a servi à noter les idées au lancement du projet et à préparer le compte-rendu. Puis,

l'application Trello et un diagramme de Gantt nous ont permis de répartir les tâches efficacement au cours du temps.

3.1 Planning Prévisionnel

Ci-dessous notre planning prévisionnel sur 3 mois. (Voir Figure 1)

TITRE DE LA TÂCHE	Janvier	Février	Mars
1 Definition et planification du projet			
1.1 Réunion de lancement	■		
1.2 Mise en place du GitHub	■	■	
2 Développement			
2.1 Génération graphismes	■	■	
2.2 Création du menu principal	■	■	
3 Client			
3.1 Mémorisation des données	■	■	
3.2 Client socket & threads	■	■	
3.3 SDL et gestion d'image	■	■	
3.4 Implémentation	■	■	
4 Réseau de Neurones			
4.1 Sauvegarde et Ouverture du NN	■	■	
4.2 Parcours des NN		■	
4.3 Classement, tri et évolution des NN		■	
4.4 Copie du jeu: plus rapide, sans réseaux et sans SDL			■
5 Serveur			
5.1 Serveur sockets & threads	■		
5.2 Génération des données du jeu		■	
5.3 Analyse des paquets		■	
5.4 Implémentation des personnages		■	
5.5 Génération		■	

FIGURE 1 – Diagramme Gant

3.2 Répartition des tâches

Nous nous sommes répartis les tâches de la manière suivante :

- Baptiste : affichage du client, déplacement d'un personnage, gestion de la carte et des tuiles. Réalisation de la partie réseau, l'envoi des informations au lancement du jeu, le format des messages, la mémorisation et le traitement des informations du jeu pour le client et le serveur.
- Lucas : réalisation du réseau de neurones, une partie qui n'est pas possible de rattacher au jeu dans l'état actuel du jeu. Lucas a aussi sa part dans la partie réseaux avec la création de la file et la mise en place des threads et de l'écoute des flux. Les compétences ont aussi été développées, dans l'état l'implémentation n'est pas concluante et ne peut pas être ajoutée au jeu. Résolution de nombreux bugs dans tout le projet.

- Ilann : réalisation menu, dans le but de récupérer le pseudo, le personnage ainsi que l'adresse IP d'un joueur ou serveur. Réalisation de l'affichage des objets dans l'inventaire des joueurs ainsi que leur récupération sur la carte et la possibilité de les relâcher.
- Nathan : réalisation des statistiques des personnages, création des objets permettant la modification des statistiques du joueur et implémentation des monstres (partiellement intégrés au jeu). Génération de beaucoup d'images et de textures de la map, laissant un libre choix de cohérence graphique dans le jeu.

4 Développement

Cette partie abordera le développement du jeu dans sa globalité. De la création des ressources graphiques à la création du client/serveur en passant par la gestion des de la carte, des objets etc.

4.1 Création des ressources

Cette section abordera la génération des ressources graphiques de base permettant la réalisation du jeu.

4.1.1 Objet

Quarante objets sont présents dans le jeu. Tous générés avec un prompt chat gpt. Quelques exemples d'objets ci-dessous (Figure 2, Figure 3).



FIGURE 2 – Bague du mage noir



FIGURE 3 – Orbe mystique

4.1.2 Sprites et Personnages

Quatre personnages sont présents dans le jeu :

- Archer
- Mage
- Ninja
- Vampire

Chaque personnage à un spritesheet de 12 images (3 images par direction). Quelques exemples de spritesheet ci-dessous (Figure 4, Figure 5).



FIGURE 4 – spritesheet Mage



FIGURE 5 – spritesheet Archer

4.1.3 Tuiles et Biomes

Six biomes différents sont implémentés au jeu.

Chaque biome a :

- une texture de base pour le sol (voir Figure 6)
- une texture pour un premier obstacle (voir Figure 7)
- une texture pour un deuxième obstacle (voir Figure 8)
- Une texture pour indiquer la sortie d'une tuile (voir Figure 9)

Exemple des 4 textures du biome plaine ci-dessous.

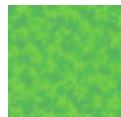


FIGURE 6 – Texture base



FIGURE 7 – Texture obstacle1



FIGURE 8 – Texture obstacle2

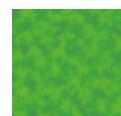


FIGURE 9 – Texture sortie

4.2 Implémentation

Nous allons maintenant présenter les différentes structures de données mises en œuvre pour la réalisation du jeu.

4.2.1 Map

La map est inspirée des jeux comme The Binding of Isaac, dans lesquels le joueur se déplace de salle en salle dans les quatre directions.

On appelle ces salles des tuiles. De plus, la map est constituée de différents biomes qui déterminent l'apparence de la tuile. Similaires aux biomes de Minecraft, les textures diffèrent selon les biomes (forêt, plaine, montagne, desert, neige, glace). Pour que le jeu soit re-jouable et intéressant, la map est générée procéduralement, en associant à chaque tuile un biome, puis en créant les tuiles en plaçant les textures selon le biome choisi.

Dans le programme écrit en C. La map est une matrice de tuile. Chaque tuile est une structure, qui contient l'information du biome, une matrice des textures qui la composent et les éléments du jeu qui y sont associés. On représente les informations par des listes. Elles sont variables au cours du jeu et selon les tuiles. Quatre listes tiennent compte des joueurs, des objets, des monstres, et des effets des compétences actives afin de les afficher facilement.

4.2.2 Personnages

Les personnages sont choisis par les joueurs au début de la partie. Ils sont caractérisés par une classe, comme dans certains jeux de rôle. Cette classe détermine les statistiques de base du personnage et les compétences accessibles. Les statistiques sont la vie, la force, la magie et la vitesse. Chaque personnage a un point fort qui le distingue des autres.

Le personnage est représenté par une structure C. Ses statistiques sont deux tableaux de quatre valeurs, une pour sauvegarder ses statistiques de base et l'autre pour avoir accès à ses statistiques améliorées par les objets. Un tableau de pointeurs permet de retrouver les objets possédés par le joueur.

La fonction creer_perso permet d'initialiser un joueur à partir de la classe choisie. Celle pour détruire est appelée à la fin du programme.

Pour gérer le déplacement des joueurs, on utilise SDL qui écoute les évènements du clavier. En utilisant l'état des flèches directionnelles, on peut faire changer la direction du joueur avec changer_dir. Puis la fonction avancer est appelée et change la position du personnage grâce à sa vitesse et sa direction en vérifiant qu'il ne dépasse pas les bords de la tuile. Pour simplifier le calcul, on utilise une table de hachage qui donne une structure qui est le multiplicateur en x et en y selon la direction.

Exemple : le joueur se déplace vers le haut et vers la droite, qui est converti en une valeur énumérée (type enum en C).

Cette valeur donne un indice pour le tableau deplacement. La valeur qui correspond à cette direction est la structure de valeurs $x = 1$ et $y = -1$, car un déplacement vers la droite signifie ajouter une valeur x (la vitesse du joueur) et un déplacement vers le haut implique de diminuer la valeur y du joueur de sa vitesse aussi.

4.2.3 Compétences

Les compétences sont les capacités des personnages, elles peuvent être de simples attaques, des techniques de déplacement ou bien des sorts.

Elles sont définies, de manière statique dans la structure compétence, et de manière dynamique dans la structure *cast*. Quand une compétence est lancée, la structure de *cast* est créée et ajoutée dans une liste. Le *cast* va évoluer en fonction des appels de la fonction update. De sorte à par exemple supprimer une compétence de zone, après sa durée de vie. La fonction d'update permet aussi de détecter des joueurs en collision pour leur infliger des dégâts par exemples etc. Les *casts* de compétences sont génériques, on peut ainsi avoir plusieurs types de compétences différentes.

L'implémentation des *casts* est différente entre le côté client et le côté serveur.

Le client, au moment de la détection d'une touche d'envoie de compétence, fait une requête au serveur avec les paramètres nécessaires, comme les positions etc.

Le serveur ajoute ce nouveau *cast* dans une liste et communique avec les joueurs sur la tuiles, les informations comme la position, la taille d'une zone etc. Le serveur va souvent faire l'update des *casts*, et ainsi envoyer les informations au joueurs.

Le client à la suite de la requête du lancement de la compétence ne s'occupe seulement de la récupération des informations envoyées par le serveur et de l'affichage de la compétence.

4.2.4 Objets

Les objets sont essentiels au jeu. Ils permettent d'augmenter les statisques de base de notre personnage. Quarante objets sont présents dans le jeu, séparés en 4 catégories de rareté :

- commun
- rare
- épique
- légendaire

Chaque objet a 4 statisques différentes :

- vie
- force
- magie
- vitesse

Ensuite pour calculer le bonus apporté au joueur par chaque objet on différencie trois cas de figure :

- bonus nul
- bonus à additionner
- bonus à multiplier

Pour garder une certaine cohérence dans les calculs de statisques du personnages un ordre de calcul a été décidé. On effectue d'abord les calculs de somme puis les calculs de pro-

duit. Pour ce faire chaque statisque de l'objet a un couple de valeur. La première valeur corresponds à ce qui est à sommer ou multiplier à la statisque de base du personnage. La deuxième valeur est un ordre de priorité permettant d'effectuer les calculs dans le bon ordre.

Exemple : si un personnage a deux objets :

- 1^{er} objet : $\{\{0, \text{null}\}, \{10, \text{add}\}, \{0, \text{null}\}, \{-2, \text{add}\}\}$
- 2^e objet : $\{\{1.5, \text{mult}\}, \{0.8, \text{mult}\}, \{20, \text{add}\}, \{0, \text{null}\}\}$

On rappelle que les statisques sont de la forme {vie, force, magie, vitesse}. Dans cet exemple, on commencera par calculer les trois additions puis les trois multiplications (la somme étant représenté par «add» et le produit par «mult»).

Deux fonctions permettent de gérer les objets possédés par le joueur. La fonction ajouter_objet permet de placer un nouvel objet dans l'inventaire à une place libre. La fonction retirer_objet supprime l'objet à la position donnée. Ces fonctions sont appelées par des fonctions de plus haut niveau qui vérifient la présence d'un objet au sol et retirent ou ajoutent l'objet correspondant sur la tuile en fonction de l'action du joueur.

La fonction update_stats est appelée quand un joueur gagne ou perd un objet. Elle permet de recalculer les statistiques du personnage

4.2.5 Monstres

Le jeu intègre des personnages non joueurs. Ce sont des monstres, appelés mobs. Lorsqu'il sont tués il permettent d'obtenir des objets qui augmentent les capacités de base de notre personnage. Les mobs ont plusieurs fonctionnalités. La première est qu'ils se rapprochent du joueur le plus proche à l'aide d'une fonction de calcul de distance. Une fois que cette distance est calculée, la direction que le mob doit prendre est choisie avec un calcul d'angle. Un vecteur horizontal orienté à droite qui servira de base est créé ainsi qu'un vecteur représentant la direction personnage vers mobs. (Voir Figure 17) On calcule l'angle en radians formé par ces deux vecteurs.

On obtient donc une valeur du cercle trigonométrique que l'on exploite de la manière suivante :

- Le monstre va en haut si $\frac{\pi}{4} \leq \text{angle} < \frac{3\pi}{4}$
- Le monstre va à gauche si $\frac{3\pi}{4} \leq \text{angle} < \frac{5\pi}{4}$
- Le monstre va en bas si $\frac{5\pi}{4} \leq \text{angle} < \frac{7\pi}{4}$
- Le monstre va à droite si $\frac{7\pi}{4} \leq \text{angle} < \frac{\pi}{4}$

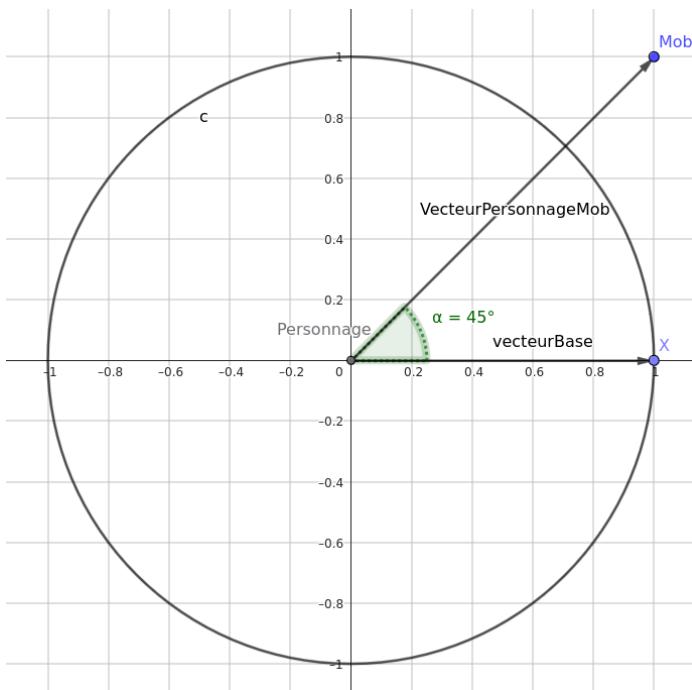


FIGURE 10 – Représentation d'un calcul d'angle en fonction de la position d'un personnage et d'un mob

4.2.6 Évènement

4.3 Réseau

Nous allons maintenant voir comment connecter les joueurs avec un serveur pour permettre la connexion et le jeu. Tout d'abord la communication à travers le réseau se fait par des écoutes et des envois dans différents flux. Le client n'a donc comme flux, seulement le serveur. Le serveur a bien plus de flux, un pour chaque joueur. La parallélisation est donc obligatoire.

4.3.1 Networking

Tout d'abord le client se connecte au serveur, suite au choix réalisé dans le menu et suite à la l'initialisation du serveur. Plusieurs structures vont être créées et la parallélisation démarre.

L'envoi et la réception de données s'assure avec deux fonctions principales : La fonction `send` (qui n'est pas bloquante) permet l'envoi d'un paquet en unicast. La fonction `recv` (qui est bloquante) permet l'écoute du flux et la réception des paquets.

La propriété bloquante de `recv` force l'utilisation de threads. Ainsi dès la première connexion, une fonction parallèle d'écoute est créée. Le programme possède donc à ce moment :

- Le programme principal qui a appelé la création des autres threads.

— Un (ou plusieurs pour le serveur) thread(s) d'écoute.

La connexion entre eux se fait par une file. Les threads d'écoute enfilent et le programme principal défile les informations envoyées par le serveur. Ces informations passent ainsi du serveur à l'écoute au main pour ensuite être traitées.

Cette file permet aussi au serveur de traiter les différents clients.

Différents code peuvent être utilisé lors de l'envoi de paquets (Voir Table 1) :

Code	Nom	Catégorie
11	JOUEUR_CHANGE_DIR	Joueur
12	JOUEUR_MV	Joueur
13	JOUEUR_MV_TUILLES	Joueur
...
20	ADD_OBJET	Objet
21	RM_OBJET	Objet
...
301	UPDATE_ZONE	Compétence de zone
302	RM_ZONE	Compétence de zone
...

TABLE 1 – Différents codes d'action entre le client et le serveur

4.3.2 Client

Au début du jeu, le client se connecte au serveur via socket. Il envoie les informations de son personnage au serveur, puis récupère celles de tous les autres joueurs grâce à lui. Il initialise le jeu grâce aux fonctions *init_sdl* et *init_jeu* qui créent respectivement la fenêtre du jeu et les ressources pour la gestion de la map.

L'élément principal du programme du client est la boucle de jeu, qui récupère les actions effectuées par le client grâce à SDL. Le joueur se déplace, récupère un objet ou lance une compétence.

Puis, en récupérant un à un dans un buffer les messages stockés dans la file qui écoute le réseau, le client traite les informations qu'il reçoit du serveur. Il récupère de cette manière la position des autres joueurs, leurs actions, s'il doit les afficher, l'ajout et la suppression d'objets sur la tuile, le changement de tuile, etc. La fonction *sscanf* de la bibliothèque standard a été très utile pour ça, ainsi que *data_skip*, écrite pour faciliter la lecture des données dans le buffer.

Ensuite le client informe le serveur de ses actions comme un changement de direction, l'utilisation d'une compétence, un objet équipé ou lâché. Pour cela, la fonction *sendf* inspirée de *sprintf* est utilisée pour mettre en forme les informations à envoyer au serveur.

Enfin, le client affiche l'état du jeu. Pour cela, des fonctions s'occupent de charger et

d'afficher la tuile courante du joueur. Grâce aux listes actualisées des entités, le client affiche les textures correspondant à tous les personnages, objets et compétences utilisées.

4.3.3 Serveur

Le serveur est le premier qui démarre, puis se met en attente des joueurs. Une fois tous les joueurs connectés, il envoie un message start avec une graine de génération de la carte. Une fois la partie commencée, le serveur communique les informations sur le jeu à chaque joueur. C'est à ce moment que le serveur se met en écoute, pour gérer la communication entre les joueurs et le jeu de manière générale.

Le serveur fonctionne grâce à une boucle dans laquelle il traite les événements arrivant dans le jeu. A partir de cela, il peut modifier les informations du jeu qu'il possède, pour ensuite les communiquer avec les clients. Les informations sont récupérées par une file partagée entre plusieurs threads d'écoutes pour pouvoir écouter chaque client. Par exemple, le serveur vérifie, au moment du déplacement d'un joueur, qu'il ne sorte pas du jeu ni ne crée de collision avec des éléments bloquants sur la carte. Il renvoie ensuite à tous les joueurs présents sur la tuile, la position de ce joueur après un déplacement ou non.

Le serveur s'arrête à la fin du jeu, c'est-à-dire au moment où les joueurs se déconnectent en renvoyant le code d'arrêt '!'.

4.4 Rendu Graphique

Cette partie traitera de l'utilisation de la librairie SDL2 qui sert pour le rendu graphique du jeu.

4.4.1 Menu Principal

Nous allons maintenant présenter la mise en œuvre du menu principal de notre jeu. Voici à quoi ressemble le menu lorsqu'on le lance. (voir Figure 11)



FIGURE 11 – Image menu principal lors du lancement du jeu

Fonctionnalités du menu Dès qu'il est lancé, une musique de fond se lance. Le menu renvoie un entier qui peut être :

- -1 : quitter le menu sans jouer
- 0 : rejoindre une partie créée par une autre personne en saisissant son adresse IP
- 1 : être joueur et serveur à la fois
- 2 : être uniquement serveur de la partie

Dans la page d'accueil du menu, il est possible de saisir son pseudo et de changer la classe de son personnage. Il est également possible de quitter le jeu directement sans le lancer ni être serveur. En cliquant sur l'onglet paramètres, une nouvelle section s'ouvre.

Nous y retrouvons :

- un bouton de retour au menu d'accueil
- un bouton de sauvegarde des paramètres
- un bouton charger paramètres qui charge les paramètres dans ce même fichier
- un bouton plus et un bouton moins qui changent la valeur du volume de la musique du menu

Affichage général

Le menu peut être dans différents états représentés par un type enum pos_actuelle. Cet enum est mis à jour en fonction des boutons cliqués par le joueur.

La fonction aff_menu permet d'afficher la bonne partie du menu en fonction de la valeur de l'enum.

Images

Plusieurs images sont présentes dans le menu. On y retrouve différentes catégories :

- Personnages : les personnages disponibles pouvant être sélectionnés par le joueur
- Boutons : les boutons principaux du menu sont des images utilisant un style boisé

- Arrière-plan : une image dans le thème de la nature

Les images des personnages vus de face sont chargées en fonction du personnage sélectionné par le joueur. Toutes les images chargées sont stockées dans un tableau de structure `img_t`. Chaque structure `img_t` possède deux champs : une `SDL_Texture`¹ et la position de l'image représentée par un `SDL_Rect`².

Nous utilisons principalement quatre images de boutons boisés pour servir d'arrière-plan aux textes présents.

Textes

Les textes sont quant à eux chargés et sauvegardés dans un tableau de structure `texte_t` qui contient les mêmes champs que `img_t`, avec en plus un `char*` correspondant au texte affiché.

À la création d'un texte, nous utilisons une police importée (Go-Regular), créons une `SDL_texture`, mettons le nouveau texte dans le champ contenu.

Certains textes doivent être modifiés dynamiquement. On utilise donc une fonction `maj_texte` qui recrée la texture si l'ancien texte est différent du nouveau.



FIGURE 12 – Affichage lorsque l'IP est valide



FIGURE 13 – Affichage lorsque l'IP est invalide

Gestion des événements du menu

Les événements en SDL sont gérés à l'aide d'une structure `SDL_Event` fournie par SDL. Nous nous en servons dans notre projet pour détecter les clics de la souris et les touches pressées au clavier.

souris

Les clics souris sont assez simples à gérer : si l'événement est de type clic souris, on crée un `SDL_Point`³ correspondant aux coordonnées du curseur de la souris. De nombreuses conditions testent si le clic de la souris est dans les coordonnées d'un bouton à l'aide de

1. image ou surface graphique que l'on peut afficher à l'écran à l'aide de la fonction `SDL_Render_Copy`.

2. structure de SDL à 4 champs : (coordonnées x et y de départ, longueur et largeur de la zone où sera affichée l'image).

3. une position sur la fenêtre représentée par deux coordonnées x et y.

la fonction *SDL_PointInRect* et de la valeur de *pos_actuelle*. En fonction des boutons cliqués, différentes actions se passent. Il est possible de naviguer entre les différentes sections du menu de cette manière.

clavier

Les saisies clavier servent à modifier des chaînes de caractères mais aussi à se déplacer dans les menus à l'aide des flèches et des touches entrées. La saisie de l'adresse IP se fait via un switch dans la fonction *saisie_touche_ip* vue précédemment.

Pour la saisie du nom du joueur, on ajoute simplement chaque caractère alphanumérique à la chaîne du nom du joueur.

Il est possible de supprimer les caractères dans les deux saisies.

Il est possible de se déplacer dans le menu avec les flèches haut et bas.

```
//gestion du deplacement avec flèche dans le menu d'accueil
else if(pos_actuelle == MENU_PRINCIPAL){
    modifications = 1;
    if (touche == SDLK_DOWN) {
        bouton_select = (bouton_select + 1) % 3;
    }
    else if (touche == SDLK_UP) {
        bouton_select = (bouton_select - 1 + 3) % 3;
    }
}
```

FIGURE 14 – voici un exemple de la gestion des déplacements dans le menu via les flèches

Si l'événement SDL est de type *SDL_KEYDOWN*, on crée un *SDL_Point* qui correspond aux coordonnées du bouton.

Lorsque l'on clique sur echap, cela force l'arrêt du menu.

4.4.2 Jeu

Dans cette partie, nous présenterons les moyens d'affichage des différents composants du jeu.

Dans le jeu, le rendu de la map est généré en plusieurs étapes. Premièrement, la fonction *init_biomes* charge toutes les textures du jeu qui composent les tuiles (voir Figure 6, 7, 8 et 9). Ensuite, les fonctions *init_map* et *init_tuile* génèrent pour chaque tuile une instance d'un biome. Enfin, il reste la phase d'exploitation : quand un joueur entre sur une tuile. La fonction *charger_tuile* s'occupe de créer le rendu qui sera affiché sur l'écran (qui est fixe) à partir de la matrice de la tuile (voir exemple biome Figure 15 et 16). Dans la boucle du jeu, cette texture est affichée avant les autres éléments du jeu à

l'écran, grâce à la bibliothèque SDL, car elle est le décor.

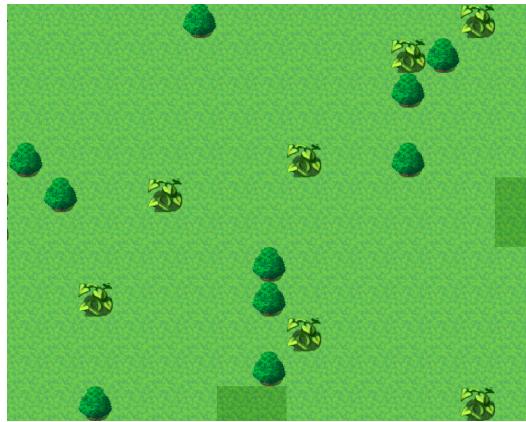


FIGURE 15 – Biome Forêt



FIGURE 16 – Biome glace

Pour l'affichage des personnages. Nous avons développé une fonction *charger_sdl_joueurs*, qui prend en paramètres le tableau des joueurs et un tableau à deux dimensions qui stocke les différentes textures des personnages. La fonction s'occupe de charger les bonnes images en fonction de la classe de chaque joueur. Dans la boucle principale du client, la liste des personnages présents est parcourue, on affiche alors la texture du personnage qui correspond à sa direction, à sa position courante.

L'affichage des objets est similaire, les textures sont préchargées, puis, grâce à la liste qui tient compte de ceux qui sont présents sur la même tuile que le joueur, ils sont affichés. Les fonctions de destruction et de libération des ressources sont appelées à la fin du programme. Les textures des objets, personnages, biomes doivent être détruites par la fonction SDL adéquate. Tous les éléments des listes de chaque tuile sont aussi supprimés par *détruire_tuile*, faisant appel à *détruire_liste*.

5 Résultats et conclusion

L'état final du jeu rendu n'est pas conforme à nos premières attentes. De nombreuses fonctionnalités n'ont pas pu être ajoutées : les quêtes, l'ajout complet des objets dans le jeu (l'envoi des informations nécessaires entre le serveur et les clients est codé mais ils n'impactent pas le jeu) et l'implémentation des compétences des personnages (malgré la présence des bases pour le faire également). Cependant, les fonctionnalités les plus importantes sont réussies : la discussion entre les clients et le serveur permet aux joueurs de se rencontrer dans le jeu, les tuiles et la map sont réussies au niveau graphique, tout comme les personnages.

Le planning prévisionnel n'a pas été respecté car nous avons abandonné de multiples fonctionnalités, mais celles qui ont été réalisées ont généralement suivi les dates imposées. Le développement du serveur et du client ont finalement duré du début à la fin du

développement, il fallait les développer en parallèle des autres fonctionnalités car elles y sont toutes intégrées. La partie qui a le plus dépassé les prévisions était la gestion des collisions avec les obstacles de la map.

Si nous avions un peu plus de temps pour améliorer le jeu, nous nous concentrerions sur le système de combat pour permettre la victoire d'une des équipes et développer les compétences, dont l'implémentation dans le code était en voie, pour que le jeu soit plus intéressant.

6 Annexes

6.1 Débogage

```
void init_joueurs_server(perso_t * joueurs, int nb){  
    char * data;  
    int equipe=1;//alterne à chaque boucle  
    classe_t classe;  
    char *nom;  
    int ind;  
    /*On ne sait pas dans quel ordre on va lire, on utilise l'indice  
    que le client envoie.*/  
    while(nb){//nb de messages attendus  
        if(fileVide(serv_file))  
            sleep(1);  
        else{  
            data=defiler(serv_file);  
            sscanf(data, "%d %d %s", &ind, (int*)&classe, nom);  
            creer_perso(joueurs + ind, classe, nom, ind, equipe!=equipe);  
            free(data);  
            nb--;  
        }  
    }  
}
```

FIGURE 17 – Fonction à tester

```

Thread 1 "serv" received signal SIGSEGV, Segmentation fault.
__vfscanf_internal (s=s@entry=0x7fffffffdea0,
    format=format@entry=0x55555555b1c7 "%d %d %s",
    argptr=argptr@entry=0x7fffffffde88, mode_flags=mode_flags@entry=2)
at vfscanf-internal.c:1100
1100 vfscanf-internal.c: Aucun fichier ou dossier de ce type.
(gdb) bt
#0 __vfscanf_internal (s=s@entry=0x7fffffffdea0,
    format=format@entry=0x55555555b1c7 "%d %d %s",
    argptr=argptr@entry=0x7fffffffde88, mode_flags=mode_flags@entry=2)
at vfscanf-internal.c:1100
#1 0x00007ffff7c7d382 in __GI_isoc99_sscanf (s=0x7ffe8000b60 "0 3 lulu ",
    format=0x55555555b1c7 "%d %d %s") at isoc99_sscanf.c:31
#2 0x0000555555581ed in init_joueurs_server (joueurs=0x7fffffff0c0, nb=2)
at serv_jeu.c:117
#3 0x000055555556a8c in main_server (port=2020, nb_clients=2) at serv.c:54
#4 0x00005555555743f in main (argc=2, argv=0x7fffffff458) at serv.c:170
(gdb) print ind
No symbol "ind" in current context.
(gdb) up
#1 0x00007ffff7c7d382 in __GI_isoc99_sscanf (s=0x7ffe8000b60 "0 3 lulu ",
    format=0x55555555b1c7 "%d %d %s") at isoc99_sscanf.c:31
31 isoc99_sscanf.c: Aucun fichier ou dossier de ce type.
(gdb) print ind
No symbol "ind" in current context.
(gdb) up
#2 0x0000555555581ed in init_joueurs_server (joueurs=0x7fffffff0c0, nb=2)
at serv_jeu.c:117
117         sscanf(data, "%d %d %s", &ind, (int*)&classe, nom);
(gdb) print ind
$1 = 0
(gdb) print classe
$2 = mage
(gdb) print nom
$3 = 0x18 <error: Cannot access memory at address 0x18>
(gdb) whatis nom
type = char *

```

FIGURE 18 – *nom* ne doit pas être un pointeur mais un tableau