Le Mans Université

Licence Informatique 2ème année

Module 174UP02 Rapport de Projet

Wizard battle

Titouan Mokrani, Léo Guibert, Nassim Touissi

24 avril 2024

Table des matières

1 Introduction

Le Projet que nous avons choisi de réaliser est un jeu vidéo en deux dimensions nommé Wizard battle, à la croisée des genres du roguelike [1]. et du Beat them all [2]. Le joueur, incarné par un sorcier, apparaît sur une carte prédéfinie et doit résister aux assauts des ennemis l'attaquant (représentés sous forme de fantômes). Le jeu fonctionne par vagues [3] d'adversaires et l'objectif est de réussir à survivre le plus longtemps possible contre les offensives menées contre le joueur, sachant que la difficulté s'incrémente à la fin de chaque manche. En effet, pour éviter que la partie soit trop redondante et que l'utilisateur se lasse, les adversaires de chaque vague seront plus compliqués à battre que ceux de la vague précédente. Pour se défendre, le magicien incarné par le joueur a la possibilité d'envoyer des flammes qui blessent les attaquants (jusqu'à les tuer si les dégâts sont suffisants). Ces derniers de leur côté attaquent également le joueur et lui font perdre de la vie, si la barre de vie tombe à 0, alors c'est la fin de la partie. Les parties sont totalement indépendantes entre elles et ne possèdent pas de lien, la progression n'est pas sauvegardée [4] Voila le lien Git de notre projet.

2 Organisation

2.1 Répartition du travail

Pour la division des taches, trois missions principales ont été définies :
— La gestion de la caméra, de la carte et des déplacements du joueur
— La gestion des ennemis, des projectiles et du menu
— les tests unitaires et la documentation

La première partie a été supervisée et implémentée principalement par Titouan, la deuxième, par Léo. Nassim quand à lui a aidé à la réalisation des deux parties citées précédemment, et il s'est également occupé de la troisième partie. Dans le tableau 1 vous pouvez retrouver une répartition plus précise du travail (également disponible dans notre matrice GANTT).

Tâche	**Responsable**
Caméra	Titouan
Barre de vie	Léo
Gestion des événements	Léo
Gestion du personnage	Titouan & Léo
Menu	Léo
Carte	Titouan
Collisions	Titouan
Ennemis	Léo
Projectiles	Léo
Documentation	Nassim
Tests	Nassim

TABLE 1 – Répartition des tâches

1. Sous-genre de jeu vidéo de rôle dans lequel le joueur explore un donjon infesté de monstres qu'il doit combattre pour progresser
2. Sous-genre de jeu vidéo opposant un ou deux joueurs à un nombre important d'ennemis, arrivant souvent par vagues.
3. Une vague consiste en l'apparition d'un certain nombre d'ennemis que le joueur doit battre
4. voir concept de Permadeath

2.2 Outils utilisés et gestion du temps

Le principal outil utilisé pour ce projet est Github, pour sa capacité à permettre la mise en commun de notre code et la possibilité d'afficher à chaque séance les taches en cours de progression. En plus de cela, nous avons utilisé Google Sheets pour la répartition et l'agencement du travail à réaliser, ainsi que Discord pour la communication entre les membres du groupe. Au niveau de cycle du développement, un cycle itératif a été adopté, afin de favoriser une approche flexible et réactive aux changements. Cela a permis d'itérer rapidement sur les fonctionnalités du jeu, de les analyser objectivement et d'apporter des ajustements en conséquence.

3 Analyse et Conception

3.1 Approche choisie

Le jeu a été conçu en C avec l'aide de la bibliothèque logicielle SDL, en suivant une approche modulaire) afin de faciliter le développement, la maintenance et l'extension du code. Les fonctionnalités du jeu (affichage de la carte, gestion des collisions, barre de vie, etc.), ont été implémentées dans des modules séparés. Cette approche a permis à chaque membre de l'équipe de se concentrer sur des aspects spécifiques du jeu, tout en favorisant une collaboration fluide et une meilleure gestion du code source.

La partie concernant la carte, la caméra et les collisions est codée d'une manière impérative car cette approche permet la manipulation directe de l'état du jeu, ce qui est idéal pour l'affichage de la carte et de la caméra. La partie comprenant les projectiles et les ennemis est quand à elle implémentée dans le style de la programmation objet, cette approche étant plus adaptée pour modéliser les projectiles et les ennemis en tant qu'entités avec des comportements et des propriétés spécifiques.

3.2 Fonctionnalités du jeu

Pour lancer une partie, l'utilisateur sélectionne le bouton «jouer» de l'écran d'accueil (il peut avant cela modifier la difficulté de la partie). Le but est ensuite de simplement survivre au plus grand nombre de vagues possibles.

3.2.1 Actions possibles

Une fois la partie lancée, le personnage a alors six touches disponibles :

— Z, Q, S, D pour se déplacer (selon le schéma habituel [5] : Z pour se déplacer vers le haut, Q à gauche, S vers le bas et D vers la droite).

— Le clic gauche de la souris qui permet de tirer les projectiles vers l'endroit où se situe le pointeur de la souris [6].

— La touche Echap qui permet d'accéder au menu de pause

Il est important de noter que l'action d'appuyer sur une touche n'est pas exclusive : par exemple Z + Q vous fera avancer en diagonale vers le coin gauche de la fenêtre. Cependant, appuyer sur des touches antagonistes (Z + S ou Q + D) immobilisera le personnage ; Il est également possible de tirer des projectiles en se déplaçant.

5. Voir cet article

6. Il est très fortement recommandé d'avoir une souris pour jouer, le pavé tactile n'étant pas pris en compte si une touche du clavier est pressée en même temps

3.2.2 Limitations

Deux composantes du jeu imposent des limitations au gameplay : les projectiles et les déplacements. Les projectiles ont pour finalité d'atteindre les ennemis, les seules entités à être affectées par les flammes du joueur car le joueur ne peut ni se blesser lui-même, ni détruire les éléments de la carte. Le lancement des projectiles est conditionné à la quantité de mana disponible pour le joueur [7] tel que :

— un projectile = 5 de mana
— quantité maximum de mana = 100 de mana
— quantité de mana régénérée par seconde = 20 de mana [8]

Le joueur est également limité dans ses déplacements : en effet la carte est de dimension finie et il n'est pas possible d'aller au-delà de ses bords, la frontière est représentée par un mur invisible. En plus de cela, des objets présents sur la carte ne sont pas franchissables et peuvent bloquer le joueur (les ennemis, eux, ne sont pas affectés [9]).

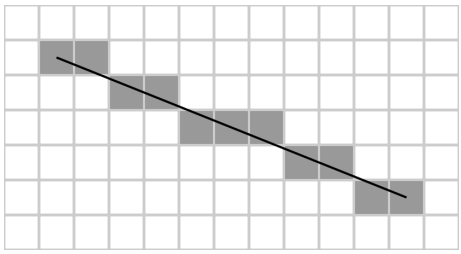
3.3 Algorithmes et structures de données

3.3.1 Algorithmes

Afin de représenter les distances entre les ennemis et les joueurs, l'algorithme de Bresenham a été utilisé [10]. Il est implémenté dans la fonction `DessinerLigneEnnemiJoueur` de la manière suivante :

$$\dot{y} = \left\lfloor \frac{y_2 - y_1}{x_2 - x_1} \cdot (\dot{x} - x_1) + y_1 + 0,5 \right\rfloor \tag{1}$$

Avec x_1 et x_2 les coordonnées du joueur, et y_1 et y_2 les coordonnées de l'ennemi. Le résultat produit est représenté dans la figure 3.3.1 :



3.3.2 Structures de données des entités et des projectiles

Pour le stockage des données, on peut différencier les deux cas principaux d'utilisation : le stockage des entités (joueur et ennemis) et des projectiles, et le stockage de la carte (et des collisions de cette dernière). Le stockage des entités et des projectiles est réalisé avec des structures, qui sont ensuite stockées sous forme de tableaux (plus d'informations dans les parties 4.7 et 4.8 .

7. représenté par la barre bleue en haut à gauche de l'écran
8. La régénération de mana est implémentée de manière à incrémenter de 10 toute les 0.5 secondes la barre de mana
9. Pour des raisons de cohérence, les fantômes peuvent passer à travers les murs et divers autres obstacles
10. Il n'a été utilisé qu'en phase de développement, son implémentation est toujours visible dans le code source mais il n'est pas utilisé dans la version finale

3.3.3 Structures de données de la carte et des collisions

Dans un souci d'efficacité de traitement, d'économie de mémoire vive et de stockage, nous avons choisi la solution de l'atlas de texture, aussi bien pour la représentation des joueurs que de la carte du jeu, car notre projet est «tile based». Pour plus de simplification, nous allons ici faire référence aux atlas de textures concernant la carte sous le nom de **tilemap**, et ceux concernant les personnages sous la forme de **spritesheet**.

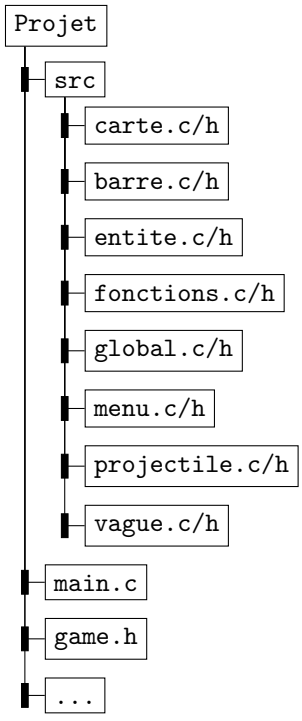
Dans le jeu, la carte est considérée comme un tableau à deux dimensions de 18 cases par 18 cases. Pour représenter les collisions, on utilise un tableau à deux dimensions, dans lequel sont présents des 0 et des 1. Un 0 signifie qu'il n'y a pas de collision prévue sur la case, et un 1 signifie que le joueur ne peut pas passer sur cette case.

Les graphiques de notre jeu sont réalisés à l'aide de deux tilemaps, l'une représente l'herbe, le sol du jeu, et l'autre les différentes structures qui apportent un peu de diversité à la carte. Pour plus de détails sur leur implémentation, voir partie 4.4.2

4 Développement

4.1 Organisation du code

Comme précisé dans l'introduction, le projet suit une approche modulaire pour la gestion de son code, qui est réparti selon l'arborescence ci-dessous (pour des raisons de simplification, ne sont représentés dans cette arborescence que les fichiers du dossier `src`, dans lequel se trouve la grande majorité du code que nous avons codé, les autres dossiers étant consacrés à la documentation, aux fichiers binaires etc.).



Le fichier game.h possède en en-tête les `include` de tous les autres fichiers du projet, ainsi que ceux de la bibliothèque SDL, le graphique de dépendance du dossier `src/` est disponible en annexe (figure 11). A la racine du programme, en plus des répertoires usuels, on retrouve également un script bash qui permet d'exporter automatiquement la variable d'environnement nécessaire au bon fonctionnement de la librairie SDL.

4.2 Flux d'exécution

Lors d'une exécution classique suivante du programme, le flux d'exécution est généralement semblable à celui représenté dans la figure 1. L'initialisation du programme se réalise avec les fonctions suivantes : `initFonctions` (la principale, qui initialise une grande partie des variables globales), `initEnnemis`, et `initBoutons`. Ensuite l'utilisateur rentre dans la boucle du jeu ainsi que celle du menu, et le programme est alors mis à jour en fonction des évènements (action du joueur ou des ennemis).

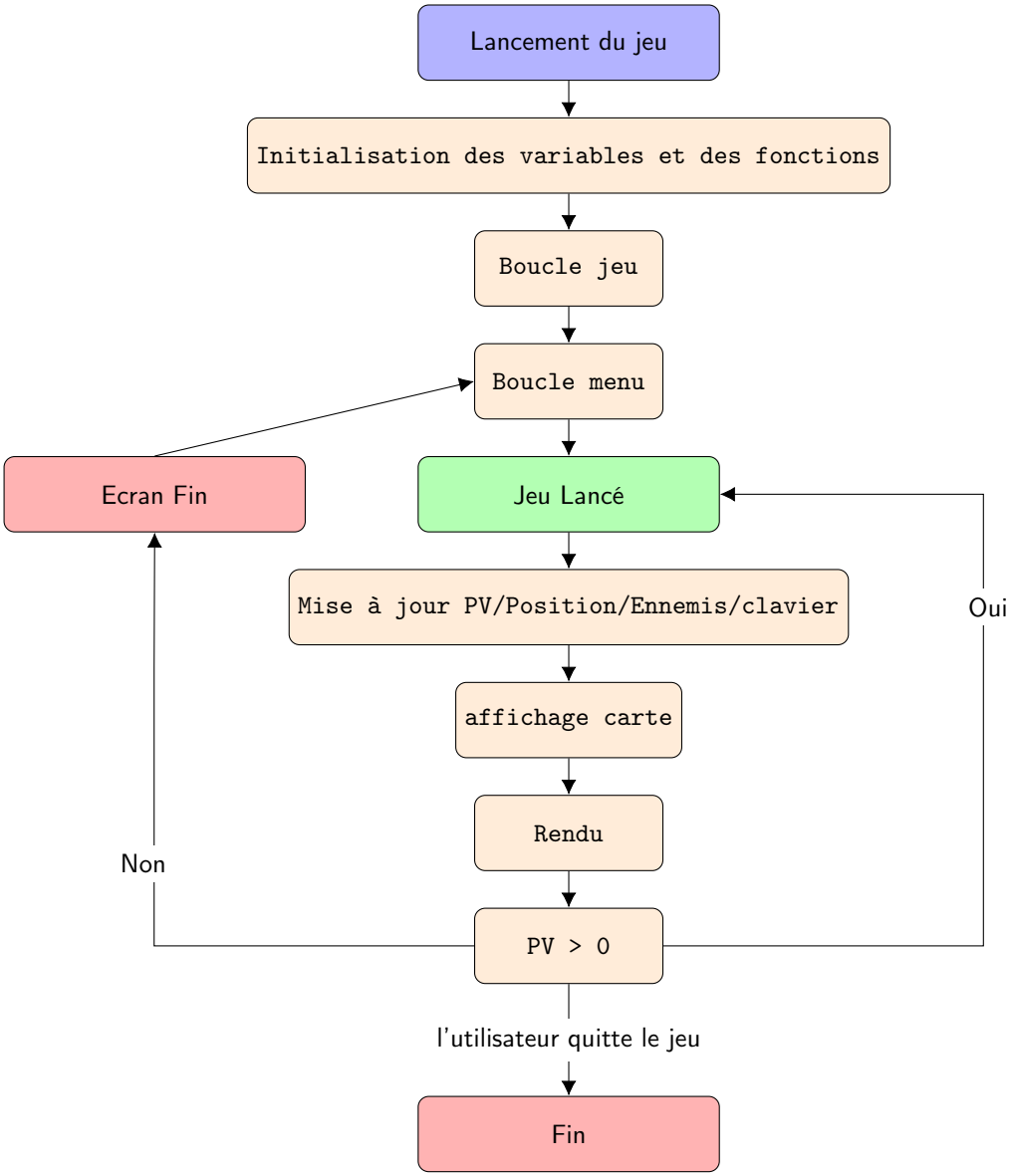


FIGURE 1 – Schéma du flux d'éxécution

4.3 Mouvements et animation du joueur

4.3.1 Mouvements

Le joueur est, en plus de sa structure REF ANNEXE, défini par un `SDL_Rect *` qui détermine le personnage à rendre à l'écran. Cette structure est envoyée à la fonction `void action(const Uint8 *clavier, SDL_Rect *pers_destination, collision_t *collision, int *direction)`, dans laquelle la variable clavier vaut `SDL_GetKeyboardState(NULL);` et permet de connaître les touches pressées par l'utilisateur (son utilisation est traitée dans la sous partie suivante) ; la variable direction permet, elle de savoir duquel côté du personnage se trouve la souris (la variable `collision` sera traitée dans la partie traitant des collisions).

Dans cette fonction se trouve quatre `if` vérifiant chacun l'une des quatre touches pressables [11]. Si l'une de quatre conditions suivantes est vérifiée :

1. `(clavier[SDL_SCANCODE_W] && pers_destination->y > 0 )`
2. `clavier[SDL_SCANCODE_S] && (pers_destination->y < WINDOWS_HEIGHT - DIM_SPRITE_PLAYER_X))`
3. `clavier[SDL_SCANCODE_A] && pers_destination->x > 0 )`
4. `clavier[SDL_SCANCODE_D] && (pers_destination->x < WINDOWS_WIDTH - DIM_SPRITE_PLAYER_X`

Alors on incrémente ou décrémente `pers_destination->x` `pers_destination->y` de `VITESSE_JOUEUR _X` ou de `VITESSE_JOUEUR _Y` en fonction de la touche pressée. De cette manière, au prochain rendu de `pers_destination`, sa position dans la fenêtre aura évolué. La suite des conditions sera traitée dans la partie portant sur les collisions.

4.3.2 Animation du personnage

Pour un souci d'immersion, une animation des entités est nécessaire, cette dernière permettant de donner une impression de mouvement. La fonction réalisant ce traitement est : `void actualisationSprite(int nb_sprite, int frame, int *direction, SDL_Rect *src, SDL_Rect *dst, SDL_Renderer *rendu)`. La fonction se sert d'un spritesheet (comme celui représenté ci dessous) déclaré en tant que variable globale (`SDL_Texture`) et va, grâce aux variables `nb_sprite` et `frame` savoir quelle partie afficher. En effet, la variable frame est incrémentée de la manière suivante dans le `main` :

```
1 if(delta_temps >= 100){ // ms entre les images du sprite
2     delta_temps = 0;
3     frame = (frame + 1) % 6;
4 }
```

Dans la fonction on retrouve les instructions suivantes :

```
1 src->x = (frame % nb_sprite)*DIM_SPRITE_PLAYER_X;
2 src->w = DIM_SPRITE_PLAYER_X;
3 src->h = DIM_SPRITE_PLAYER_Y;
4 // On affiche les sprites :
5 SDL_RenderCopy(rendu, texSprite, src, dst);
```

Concrètement, ici notre spritesheet contient 4 images, donc la fonction est appelée avec 4 en premier argument, et le programme va rendre à chaque fois un quart de l'image (dans l'ordre), en se positionnant sur le pixel `x` de l'axe horizontal de l'image (1re ligne du 2e snippet de code).

Mais avant d'effectuer ces opérations, la fonction doit choisir si elle effectue ces opérations sur le sprite gauche ou droite. La décision est prise en fonction de la variable direction (2 = Gauche,

11. voir partie 3.2.1 sur les touches

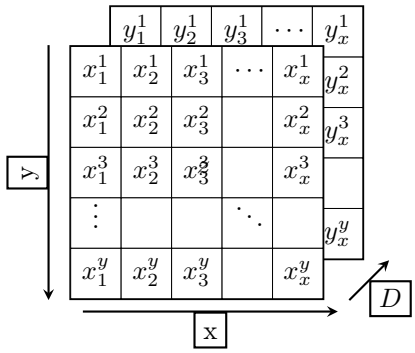
FIGURE 2 – Spritesheet (gauche) du sorcier

3 = Droite), et la valeur de cette dernière est déterminée en avant, dans la fonction `action`. La variable est là-bas déterminée par la fonction `getMousePositionDirection`, qui renvoie par paramètre les positions `x` et `y` du curseur. On calcule ensuite sa position relative [au joueur] en lui soustrayant les positions `x` et `y` du joueur. Pour finir, si la valeur relative de `x` est supérieure à 0, alors la souris est à droite, sinon elle est à gauche.

4.4 Gestion de la Carte

4.4.1 Chargement et initialisation de la carte

Comme expliqué précédemment, notre personnage se déplace sur une carte de N blocs par N blocs (dans la version présentée du jeu, $N = 18$, composé à base de tuiles. Ces tuiles proviennent de deux fichiers et sont présentées comme ci-dessous dans les figures 3 et 4. La position de ces tuiles dans la taille du jeu est gérée par une matrice tri-dimensionnelle de la forme suivante :



Ici la première matrice représente les tiles [12] à charger, sous la forme d'un nombre entier entre 1 et 64 pour la première couche, et entre 1 et 16 pour la seconde. Il y a 64 cases car le tableau de tiles est un tableau de $8 * 8$. La matrice est déclarée de la manière suivante : `int tilemap[2][NB_TILE_WIDTH][NB_TILE_WIDTH];`. La première dimension de la matrice correspond aux cases de la figure 3, et la deuxième aux cases de la figure 4.

Le chargement de la carte se fait à partir d'un fichier .txt (un fichier nécessaire par dimension souhaitée) composé de N lignes et colonnes d'entiers avec $N > 0$, et $N \leq N^2$ pour le cas d'une carte avec des cases hétéronormées de dimension 1 comme c'est le cas de la figure 3, ou sinon $N \leq$ au nombre d'éléments disponibles (figure 4. Un fichier de chargement suit le même modèle que celui présenté en figure 5. C'est la fonction `void chargerCarte(char * fichier, int tab[2][NB_TILE_WIDTH][NB_TILE_HEIGHT], int nb` qui s'occupe de charger le fichier dans la matrice, plus précisément dans la couche `nb` passée en paramètre. La fonction lit le fichier et inscrit les coordonnées dans la matrice passée en paramètre.

12. On appelle une tile une case de la tilemap

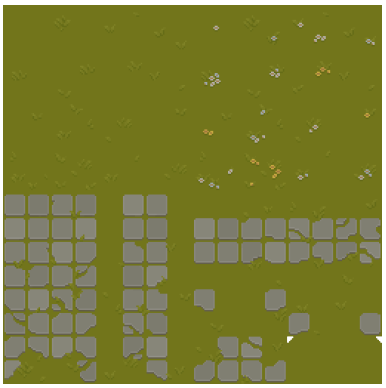


FIGURE 3 – tilemap herbe

FIGURE 4 – tilemap structures

```
12 25 01 01 41 01 25 25 01 01 01 04 01 25 25 01 01 01
01 31 01 01 42 01 01 01 01 301 01 01 01 01 4 01 01
01 25 04 01 43 01 01 14 01 01 04 25 01 31 01 01 01 01
25 01 01 01 44 01 21 01 25 25 25 01 01 01 01 01 01 01
01 01 25 01 45 01 25 01 04 01 01 31 01 14 01 01 25 01
49 48 47 46 40 42 40 01 01 01 01 01 01 14 01 01 25 01
01 25 25 01 01 01 21 01 01 48 01 01 01 01 16 01 01 01
01 01 06 01 31 01 01 01 01 50 01 31 01 01 25 01 01 01
01 01 01 01 01 01 21 01 01 51 01 01 17 01 01 01 01 01
25 25 01 01 04 01 24 24 01 52 01 31 01 01 31 01 01 01
01 01 07 01 01 01 01 01 01 54 01 01 01 01 21 01 01 01
12 25 01 01 41 01 25 25 01 01 01 04 01 25 25 01 01 01
01 31 01 01 42 01 01 01 01 301 01 01 01 01 4 01 01
01 25 04 01 43 01 01 14 01 01 04 25 01 31 01 01 01 01
01 01 25 01 45 01 25 01 04 01 01 31 01 14 01 01 25 01
49 48 47 46 40 42 40 01 01 01 01 01 01 14 01 01 25 01
01 25 25 01 01 01 21 01 01 48 01 01 01 01 16 01 01 01
01 01 06 01 31 01 01 01 01 50 01 31 01 01 25 01 01 01
```

FIGURE 5 – Fichier de chargement de la tilemap

4.4.2 Affichage de la carte

L'affichage de la carte est effectué à partir de la fonction `updateCamera` (expliquée dans la partie 4.5), et est effectué couche par couche. Concrètement pour le cas de ce projet, on va appeler la fonction une première fois pour afficher la carte présente sur la figure 3, puis une deuxième fois pour rendre les structures de la figure 4. L'affichage fonctionne de la manière suivante : on alloue dynamiquement deux `SDL_Rect *` (un pour la partie de la tilemap à prendre en compte pour le rendu, et un autre pour la zone à rendre sur le rendu final), et leur largeur et hauteur valent les dimensions d'une tuile de base. On rentre ensuite dans une boucle imbriquée, et la gestion de l'affichage diffère en fonction de la couche affichée.

1. Couche 1

```
1       val = tab[nb][y][x];
2       case_y = val / 8;
3       case_x = val % 8;
4
5       origin->h = TILE_HEIGHT;
6       origin->w = TILE_WIDTH;
7       origin->y = TILE_HEIGHT * case_y;
8       origin->x = TILE_WIDTH * case_x;
9
10      dest->x = x * TILE_HEIGHT - camera->x;
11      dest->y = y * TILE_WIDTH - camera->y;
```

```
12
13      SDL_RenderCopy(rendu, tilemap, origin, dest);
```

Les variables `case_y` et `case_x` représentent les coordonnées de la case à rendre, ici les cases sont toutes de la même dimension et on peut donc les sélectionner avec les opérateurs / et % car elles sont numérotées de la manière suivante par rapport à l'image (on imagine que derrière chaque nombre se cache une tuile) :

$$figure\ 3 = \begin{bmatrix} 1 & 2 & \cdots & 8 \\ 9 & 10 & \cdots & 16 \\ \vdots & \vdots & \ddots & \vdots \\ 57 & 58 & \cdots & 64 \end{bmatrix}$$

Pour le `SDL_Rect *` de destination, on soustrait ses coordonnées à celles de la caméra pour avoir un affichage relatif à la position du personnage.

2. Couche 2

   Pour la deuxième couche, le principe reste le même sauf que la façon de déterminer les caractéristiques des structures `origin` et `dest` ne se font plus de la même façon, et ce pour deux raisons : les dimensions des structures ne sont plus de dimension $1 * 1$, et on ne peut plus déterminer leur coordonnées sur la tilemap avec les opérateurs / et % . Il faut donc déterminer les paramètres suivants manuellement :

```
1       origin->h
2       origin->w
3       origin->y
4       origin->x
5       dest->x
6       dest->y
```

   On soustrait toujours les coordonnées du `SDL_Rect *` de destination aux coordonnées de la caméra afin de garder un affichage relatif.

4.5 Gestion de la caméra

Comme son nom l'indique, la caméra sert à afficher (rendre) un rectangle dynamique autour du personnage, ajoutant une fluidité supplémentaire au gameplay.

4.5.1 initialisation

La variable caméra est un `SDL_Rect *` initialisé par la fonction SDL_Rect * initCamera() avec les caractéristiques suivantes :

```
1       cameraRect->x = 0;
2       cameraRect->y = 0;
3       cameraRect->w = CAMERA_WIDTH;
4       cameraRect->h = CAMERA_HEIGHT;
```

4.5.2 fonctionnement

C'est ensuite la fonction `updateCamera` qui gère les paramètres de la caméra, et qui appelle les fonctions d'affichage de la carte, et de collisions. Le principe de cette fonction est de définir la zone à rendre par la fonction `SDL_RenderCopy`. Les coordonnées de la caméra sont définies par rapport aux coordonnées du joueur, et ajustées avec un facteur d'interpolation pour lisser le déplacement de la caméra :

```
1     const float interpolationFactor = 0.1;
2
3     // differentiel de position entre la cam ra et le joueur
4     int dx = (pers_destination->x * 2) - cameraRect->x;
5     int dy = (pers_destination->y * 4) - cameraRect->y;
6
7     // Application de l'interpolation lineaire
8     cameraRect->x += (int)(dx * interpolationFactor);
9     cameraRect->y += (int)(dy * interpolationFactor);
```

Concrètement, les coordonnées de la caméra peuvent se définir de la manière suivante, donnant ainsi plus de clarté au fonctionnement du facteur d'interpolation.

```
1     cameraRect->x = cameraRect->x + (dx * interpolationFactor)
2     cameraRect->y = cameraRect->y + (dy * interpolationFactor)
```

De la même manière que pour les déplacements du joueur (définis dans la sous partie 4.3.1), la caméra possède des conditions permettant de vérifier que rien n'est rendu en dehors du cadre défini.

4.6 Collisions

4.6.1 Initialisation des collisions de la carte

Pour gérer les collisions, le jeu se base entre autres sur un tableau à deux dimensions de taille $N * N$ avec N le nombre de cases, défini en amont. La fonction `chargercollisions` qui va initialiser ce tableau, à partir d'une dimension du tableau d'initialisation de la carte (la 2e, qui contient les structures). Le tableau va être parcouru, et si la valeur d'une case est égale à l'une des valeurs définie comme représentant un objet infranchissable est détectée, alors dans le tableau de collisions cette valeur vaudra 1. On se retrouve alors avec un tableau bidimensionnel binaire qui représente les endroits ou se trouvent des éléments nécessitant des collisions.

4.6.2 Position du joueur

Il reste maintenant à déterminer la position du joueur, afin de savoir quand est-ce qu'il se retrouve en situation de collision avec un bloc. Il est pour cela possible de calculer avec un point central la position du joueur mais cette méthode manque de précision. Le jeu possède donc un système de position se basant sur les quatre coins de l'image du personnage, donnant donc plus d'indication sur l'endroit où une potentielle collision peut se produire. On possède, pour représenter cela, des trois structures suivantes :

```
typedef struct {
int casx;
int casy;
} case_t;
```

```
typedef struct{
case_t case_hg;
case_t case_hd;
case_t case_bg;
case_t case_bd;
}positionJoueur_t;
```

```
typedef struct{
int haut;
int bas;
int gauche;
int droite;
positionJoueur_t *
position;
}colision_t;
```

Premièrement à chaque itération du programme la position du joueur est mise à jour avec la fonction `initialiser_position_joueur`, qui va attribuer à chaque coin du joueur la position de la case sur laquelle il se trouve, et renvoyer le tout via un pointeur sur une structure `positionJoueur_t`

Cette variable est ensuite envoyée à la fonction `collisions` qui va vérifier si l'une de ses positions se trouve sur une case du tableau `tabTilesColision` ou la valeur est 1. Si c'est le cas, il va modifier la valeur de l'entier correspondant dans la structure `colision_t`.

4.6.3 Gestion des collisions de la carte

La gestion des collisions se déroule dans la variable action : lorsqu'une est pressée, en plus de vérifier si l'on ne sort pas de la fenêtre, le programme vérifie également que le personnage n'a pas une collision détectée à cet endroit. Si c'est le cas, la valeur de sa position `x` ou `y` est alors incrémentée ou décrémentée suivant le cas.

4.6.4 Projectiles et collisions

La détection des collisions des projectiles est divisée en deux cas : les collisions avec les éléments de la carte, et les collisions avec les entités (joueur ou ennemi). Pour les collisions avec les éléments du décor, elles sont détectées avec la fonction `verifCollisionProj`. Elle vérifie si les coordonnées x et y sont sur une case non nulle du tableau de collision, et si c'est le cas alors l'animation d'impact du projectile est lancée.

Pour les collisions entre projectiles et entités, elles sont assurées par la fonction `collision ProjEntite` (qui est la seule fonction déclarée directement dans `main.c` pour des raisons de références circulaires. Si le projectile vérifie les conditions suivantes, alors une collision est détectée et l'animation d'impact du projectile est lancée.

FIGURE 6 – Conditions de collisions

```
1  /* Condition pour projectile joueur*/
2  if (((projectile->x +25 + projectile->w > ennemi->x) &&
3  (projectile->x + 25 < ennemi->x + ennemi->rect.w)) &&
4  ((projectile->y + projectile->h/2 > ennemi->y) &&
5  (projectile->y + 50 < ennemi->y + ennemi->rect.h)))
6
7  /* Condition pour projectile ennemi*/
8  if (projectile->x + projectile->w/2 > cameraRect->x + playerRect->x &&
9  projectile->x < cameraRect->x + playerRect->x + playerRect->w &&
10 projectile->y + projectile->h/2 > playerRect->y + cameraRect->y &&
11 projectile->y < playerRect->y + playerRect->h + cameraRect->y)
```

Le code ci-dessus vérifie si le projectile entre dans l'aire de l'entité.

4.7 Ennemis

4.7.1 Initialisation des ennemis

La structure de données ennemi_t est utilisée pour représenter un ennemi. Cette structure contient des informations telles que la position (x, y), la vitesse, les points de vie (pv), l'attaque, etc. (pour la liste exhaustive voir annexe). Les structures sont gérées sous forme de tableaux, et initialisées au début de la manche par la fonction `init.Ennemi` présente dans leur structure. C'est là que sont définis leurs points de vie, leurs dégâts et les autres caractéristiques.

4.7.2 Actualisation des ennemis

C'est là que sont mis à jour la position, le déplacement, les points de vie et le rendu graphique des ennemis. La Fonction `updateEnnemi` gère une partie de ces évènements. On vérifie déjà que

les points de vie de l'ennemi sont supérieurs à 0, car dans le cas contraire l'ennemi est mort (`ennemi.mort = 1`) et dans ce cas il n'est plus pris en compte dans les fonctions et conditions suivantes. Pour que le personnage bouge, il faut également que la variable `ennemi->détection` ne soit pas nulle, dans le cas contraire l'ennemi reste immobile. Si le joueur est détecté, l'ennemi va chercher à aller le rejoindre, avec une application du théorème de Pythagore :

```
1 float dirx = (playerRect->x + 100/2) - ennemi->rect.x ;
2 float diry = (playerRect->y + 100/2) - ennemi->rect.y ;
3 float hyp = sqrt(dirx * dirx + diry * diry);
4 dirx /= hyp;
5 diry /= hyp;
6 ennemi->x += dirx * ennemi->vitesse;
7 ennemi->y += diry * ennemi->vitesse;
```

Mathématiquement cela correspond à la figure suivante.

Soit e_x et e_y les coordonnées de l'ennemi, j_x et j_y les coordonnées du joueur et V la constante de vitesse de déplacement, alors les nouvelles valeurs de e_x et e_y s'obtiennent par :

$$e_x = \frac{((j_x + 50 - e_x)}{\sqrt{((j_x + 50 - e_x)^2 + ((j_y + 50 - e_y)^2}} * V$$
$$e_y = \frac{((j_y + 50 - e_y)}{\sqrt{((j_x + 50 - e_x)^2 + ((j_y + 50 - e_y)^2}} * V$$

Cependant, si l'ennemi est trop proche, il va alors s'arrêter pour continuer d'envoyer des projectiles à distance, et non au corps à corps.

Pour le rendu, une fonction similaire à celle de actualisationSprite (voir sous partie 4.3.2) est utilisé, la direction du sprite utilisée est conditionnée à la position du personnage par rapport au joueur (on peut savoir le côté du quel se situe le joueur grâce à la variable dirx : si elle est positive alors le sprite est à gauche du personnage).

4.8 Projectiles

Les projectiles sont contenus dans deux tableaux de structures `projectiles_t` : `projEnnemi` et `projJoueur`.

4.8.1 Projectiles ennemis

Les projectiles ennemis sont tirés toutes les deux secondes en difficulté `normale` (en `facile` la fréquence est de 2,8 secondes , en `difficile` (1,5 secondes). Au moment où un projectile est tiré, il est initialisé avec la fonction `ProjEnnemi.initProj` qui prend (entre autres) en paramètre les coordonnées p_x, p_y, d_x et d_y qui correspondent respectivement aux coordonnées de départ et de destination (la position du joueur au moment du tir).

FIGURE 7 – Spritesheet du projectile

4.8.2 Projectiles joueurs

Pour le joueur, les projectiles sont tirés lorsque l'utilisateur clique sur le bouton gauche de sa souris, si il possède assez de mana. Les règles de dépense et de régénération de mana sont expliquées dans la partie 3.2.2. Quand un projectile est tiré, il est initialisé avec la même fonction `ProjEnnemi.initProj` sauf que cette fois les paramètres passés d_x et d_y correspondent à la position du pointeur de la souris.

4.8.3 Gestion des projectiles

Une fois la fonction d'initialisation du projectile appelée , la variable `projectile.angle` va être calculée, pour permettre un affichage cohérent. Pour cela c'est la fonction `calculerAngle` qui va être utilisée, renvoyant la formule suivante [13] (ou m_x et m_y représentent les coordonnées de la caméra :

$$\theta = 2 * arctan\Big(\frac{c_y + d_y - (p_y + 50)}{\sqrt{((c_x + d_x - (p_x + 50))^2 + ((c_y + d_y - (p_y + 50))^2}} + (c_x + d_x - (p_x + 50))\Big)$$

Ensuite le projectile est mis à jour pour évoluer continuellement, les fonctions suivantes sont appelées dans le main :

La fonction `actualisationSpriteProj` va afficher les différentes parties du sprite (figure 4.3.2) de la même manière que l'on met à jour les sprite des joueurs et des ennemis. Pour faire progresser leurs coordonnées, la fonction `updateProj` va actualiser les coordonnées des projectiles de la manière suivante :

```
1 projectile->x += projectile->vx * projectile->vitesse;
2 projectile->y += projectile->vy * projectile->vitesse;
3 projectile->rect.x = projectile->x - cameraRect->x;
4 projectile->rect.y = projectile->y - cameraRect->y;
```

Les variables `vx` et `vy` sont déterminées au moment de l'initialisation et prennent les valeurs suivantes :

Soit

$$d_x = d_x + c_x - p_x - 100$$
$$d_y = d_y + c_y - p_y - 100$$
avec p_x et p_y les coordonnées du projectile

alors

$$v_x = \frac{d_x}{\sqrt{d_x^2 + d_y^2}}$$
$$v_y = \frac{d_y}{\sqrt{d_x^2 + d_y^2}}$$

4.9 Vagues

Les vagues (ou manches) rythment la partie, elles contiennent un nombre d'ennemis à battre (5 en difficulté facile, 7 en normal et 10 en difficile pour la manche 1). Si le joueur arrive à tuer tous les ennemis de la manche, il peut passer à la manche suivante. A la fin de chaque vague, le nombre d'ennemis augmente de 5 (pour la manche suivante), les barre de vie et de mana se régénèrent entièrement. Les ennemis sont placés aléatoirement au début de chaque manche avec

13. effectuée par la fonction atan2

la fonction `initEnnemisVague` ou pour chaque ennemi des coordonnées aléatoires sont calculées avec la fonction `rand()`. Pour afficher le début d'une nouvelle vague, un message est affiché avec la fonction `afficherVague`

4.10 Menus

Les menus du jeu permettent d'afficher des informations et de sélectionner des options, avant ou après que l'utilisateur joue une partie. Ils sont nécessaires pour ne pas accueillir le joueur dans une partie de manière trop abrupte, et sont au nombre de quatre.

4.10.1 Fonctionnement d'un menu

Lorsque l'utilisateur est dans un menu, une image s'affiche en arrière plan et les boutons interactifs sont représentés par les structures suivantes :

```
1 typedef struct {
2     SDL_Rect rect;  /* Position et taille. */
3     SDL_Texture* texture;   /* Texture du bouton. */
4 } Button;
```

Un bouton est crée avec la fonction `afficherMessage` (et affiché avec la fonction `drawButton`) qui définit la taille du bouton, sa position dans la fenêtre et le texte qu'il affiche. On vérifie si un bouton est clique avec un event de type `SDL_Event` et la fonction `int clickButton(SDL_Event event, Button button)` qui retourne une valeur non nulle si le bouton passé en paramètre est cliqué. En fonction du menu dans lequel on se trouve, les suites d'instruction effectuées sont différentes.

4.10.2 Les différents menus

Les quatre menus disponibles sont les suivants :

1. La page d'accueil
2. Le choix de la difficulté
3. Le menu de pause
4. Le menu de fin de partie

Dans le menu d'accueil, le joueur a la possibilité de lancer directement une partie, ou de changer la difficulté dans le menu dédié. Dans ce menu, le joueur peut sélectionner un niveau de difficulté parmi les trois disponibles, puis de revenir au menu principal pour lancer une partie. Le menu de pause s'active lorsque la touche Echap est enfoncée et permet au joueur de faire une pause dans sa partie, puis de la reprendre, de la relancer ou de rejoindre le menu principal en appuyant sur le bouton correspondant. Pour finir, le menu de fin s'affiche lorsque la partie se finit (quand le joueur n'a plus de vie) et affiche les informations suivantes : la vague que le joueur a réussi à atteindre, le temps survécu et le nombre d'ennemis tués. On peut alors relancer une partie en appuyant sur le bouton **retry**.

L'appel des menus se fait de la manière suivante : un `switch(event.type)` détermine le bouton sur lequel le joueur a cliqué, indiquant l'action qu'il souhaite effectuer. Si cette action implique un changement de menu, alors la variable `menu` est changée et la fonction correspondante parmi les suivantes est appelée : `menuPrincipal`, `menuDifficulte`, `menuGameOver`, `MenuPause`.

5 Bilan et résultats

5.1 Fonctionnalités

Dans l'ensemble les fonctionnalités du programme ont été réalisées et permettent de jouer au jeu de manière fluide et agréable, tout en limitant les bugs et autres problèmes indésirables. Parmi les principales fonctionnalités réalisées, on compte le système d'affichage de carte dynamique (relatif à la position du joueur), le système de collisions, les ennemis, les projectiles, des menus fonctionnels. A la date de rédaction du rapport (18/04/2024), il ne reste plus qu'à optimiser certaines parties du code et quelques erreurs mineures.

Si le projet devait être continué et que des améliorations pouvaient être effectuées, voila la liste (non exhaustive) des fonctionnalités que nous aurions aimé implémenter :

1. Génération procédurale des cartes
2. Différents types d'ennemis et d'attaques
3. Items à récupérer au sol
4. Meilleur système de collision

5.1.1 Planning

Le planning prévisionnel n'a, dans l'ensemble, pas été respecté. Cela s'explique notamment par la sous-estimation de certaines taches du jeu, comme la caméra, la carte ou encore la gestion des ennemis. De plus le choix a été pris de créer à chaque fois les prototypes des fonctionnalités et de revenir plus tard sur ces dernières, chose non prévue dans le planning initial. Pour finir, la révision des partiels de fin d'année et le travail demandé par certaines matières (notamment l'algorithmique) nous a parfois détourné du projet et faussé nos estimations de réalisation des tâches. Cependant, nous ne pensons pas avoir été impactés de manière trop importante par ces déboires, car nous avons su nous organiser en fonction des évènements cités précédemment et revoir nos planning en cours de route, en faisant des choix et en priorisant certaines tâches.

5.1.2 Enseignements et leçons

Ce projet nous a beaucoup appris, en particulier dans les domaines de la programmation, de la gestion de projet et l'utilisation d'outils annexes (notamment `git`). La prise en main d'une bibliothèque comme SDl a permis d'appréhender des concepts comme le rendu graphique, l'architecture du code, la gestion de la mémoire et bien d'autres encore. La combinaison de ces différentes notions de programmation a rendu le projet très instructif pour les membres du projet, chacun ayant dû se montrer versatile en fonction des tâches qui lui étaient confiées. Au-delà des compétences techniques acquises, cette expérience nous a permis de développer des aptitudes transversales essentielles à la réussite d'un projet collaboratif comme celui là ; on peut notamment citer la communication, la gestion du temps et la répartition du travail à accomplir.

6 Annexe

6.1 Structures de données

6.1.1 Structures de données des ennemis et des projectiles :

```
struct projectiles_s {
int id;

float vitesse;
float x;
float y;
int w;
int h;

float vx; /**< Vitesse horizontale */
float vy; /**< Vitesse verticale */

double angle; // Angle du projectile
int animation_impact;

SDL_Rect rect;
SDL_Rect sprite;

int collision;

void (*initProj)(projectiles_t *proj,
float px, float py, float mx, float my,
int, SDL_Rect *camera);

void (*verifCollisionProj)
(projectiles_t *,
int tab[NB_TILE_WIDTH][NB_TILE_HEIGHT]);

void (*updateProj)(projectiles_t *,
SDL_Rect *,
int tab[NB_TILE_WIDTH][NB_TILE_HEIGHT]);

void (*renderProj)(SDL_Renderer *,
projectiles_t *, int frame);

} ;
```

```
struct ennemi_s {
int id;
float x;
float y;

float vx;
float vy;

int pv;
int pvMax;
int attaque;
float vitesse;
int detection;

int gauche;
int droite;

int mort;

Uint32 delta;

SDL_Rect rect;
SDL_Rect sprite;

void (*initEnnemi)(ennemi_t *ennemi,
float x, float y, int id, int pvMax,
int attaque);

void (*updateEnnemi)(ennemi_t * ennemi,
SDL_Rect * cameraRect, SDL_Rect * playerRect,
int tabColision[NB_TILE_WIDTH][NB_TILE_HEIGHT],
projectiles_t projEnnemi[MAX_PROJ],
int *projNbEnnemi, int temp_vivant);

void (*renderEnnemi)(SDL_Renderer * rendu,
ennemi_t * ennemi, int frame);

void (*renderVecteur)(SDL_Renderer * rendu,
ennemi_t * ennemi, SDL_Rect * playerRect);
};
```

6.1.2 Structure de donnée du joueur :

```
struct joueur_s {
    int id;
    int x;
    int y;
    int mana;
    int manaMax;
    int pv;
    int pvMax;
    int attaque;
    int vitesse; };
```

6.2 Tests unitaires

Dans le cadre de notre projet, l'implémentation de tests unitaires revêt une importance capitale. Deux tests ont été identifiés comme étant essentiels à cette fin. Dans le tableau ci dessous vous pouvez retrouver plus d'informations sur les deux tests.

	Tests de déplacement	**Tests de dégâts**
Jeu de données	x = 100, y = 100	joueur (`initialiserJoueur(&joueur)`), ennemi : initEnnemi (&ennemi, 0, 0, 1, 100, 10))
Fonctions testées	`SDL_GetMouseState_ptr`, `SDL_RenderCopy_ptr`	`initialiserJoueur`, `initEnnemi`
Objectif	Envoyer le personnage sur des coordonnées x et y	Décrémenter la barre de vie si des dégâts sont subis
Résultats attendus	Position en x et y	`barreDeVie` = 90 (car 100 - 10 = 90)
Sortie	Succès	Succès

FIGURE 8 – tableau des tests unitaires

6.2.1 Test de mise à jour des dégâts (joueur/ennemi) :

Ce test vise à vérifier le bon fonctionnement de la gestion des dégâts infligés aux personnages et aux ennemis. Pour ce faire, nous simulons une série d'attaques entre un joueur et un ennemi, en nous assurant que les points de vie sont correctement mis à jour après chaque interaction. Le déroulement du test est le suivant :

1. Initialisation du joueur et de l'ennemi avec leurs caractéristiques respectives.
2. Affichage des points de vie initiaux du joueur et de l'ennemi.
3. Simulation d'une attaque de l'ennemi sur le joueur et mise à jour de la santé du joueur.
4. Simulation d'une attaque du joueur sur l'ennemi et mise à jour de la santé de l'ennemi.
5. Vérification que les points de vie du joueur et de l'ennemi ont été correctement ajustés.
6. Affichage d'un message de succès ou d'échec du test en fonction des résultats obtenus.

Résultat du test :

6.2.2 Test de déplacement du personnage

Ce test évalue la fonctionnalité de déplacement du personnage en vérifiant que sa position est correctement mise à jour après une action de déplacement. Le test procède comme suit :

1. Initialisation de la position initiale du personnage.
2. Simulation d'un déplacement vers le haut en appuyant sur la touche W.
3. Vérification que la position du personnage a été correctement modifiée en comparant sa position actuelle avec la position attendue.
4. Affichage d'un message de succès ou d'échec du test en fonction des résultats obtenus.

Résultat du test :
Résultat du test :

```
$ ./test_degats

Début du test de msie à jour de dégâts (joueur/ennemi) :

Santé du joueur : 100
Santé de l'ennemi : 100

Le joueur subit une attaque de l'ennemi
Santé du joueur après l'attaque : 90

Le joueur attaque l'ennemi
Santé de l'ennemi après l'attaque : 90

Test réussi avec succès !!
```

```
$ ./test_deplacements

Position de la souris : x = 100, y = 100
Le test a été réussi avec succès !!!
```

FIGURE 9 – Résultat du test de déplacement

6.3 Exemple de débogage

Au moment de gérer les polices, nous nous sommes aperçus que lorsque le jeu était fermé, une erreur de segmentation se produisait à la fin. Nous avons donc utilisé **GDB** pour retrouver la source du problème. Nous savons de ce dernier qu'il est lié à la fonction qui libère la mémoire (appelée en ligne 237 du fichier `main.c`). On utilise alors la commande `b 237` pour positionner un breakpoint sur la fonction de libération de la mémoire, puis la commande `next` est utilisée pour analyser les étapes suivies par le programme (voir figure 10). On s'aperçoit alors que l'appel à la fonction `TTF_CloseFont()` (censée libérer la mémoire de la police utilisée par le jeu) provoque une erreur de segmentation. La fonction était en fait appelée deux fois par le programme, et devait alors libérer une zone mémoire déjà libérée au préalable. Pour corriger l'erreur de segmentation de notre programme, il a donc suffit de supprimer le deuxième appel à cette fonction.

6.4 Graphes

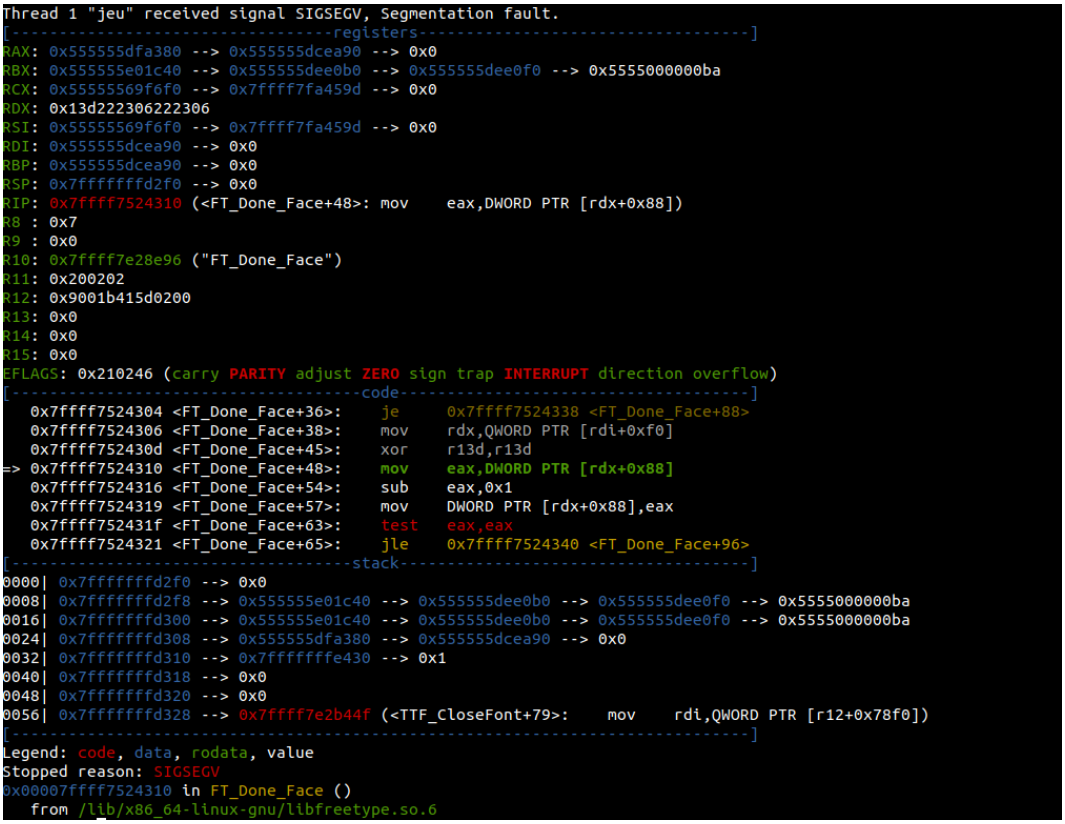


FIGURE 10 – débogage avec GDB

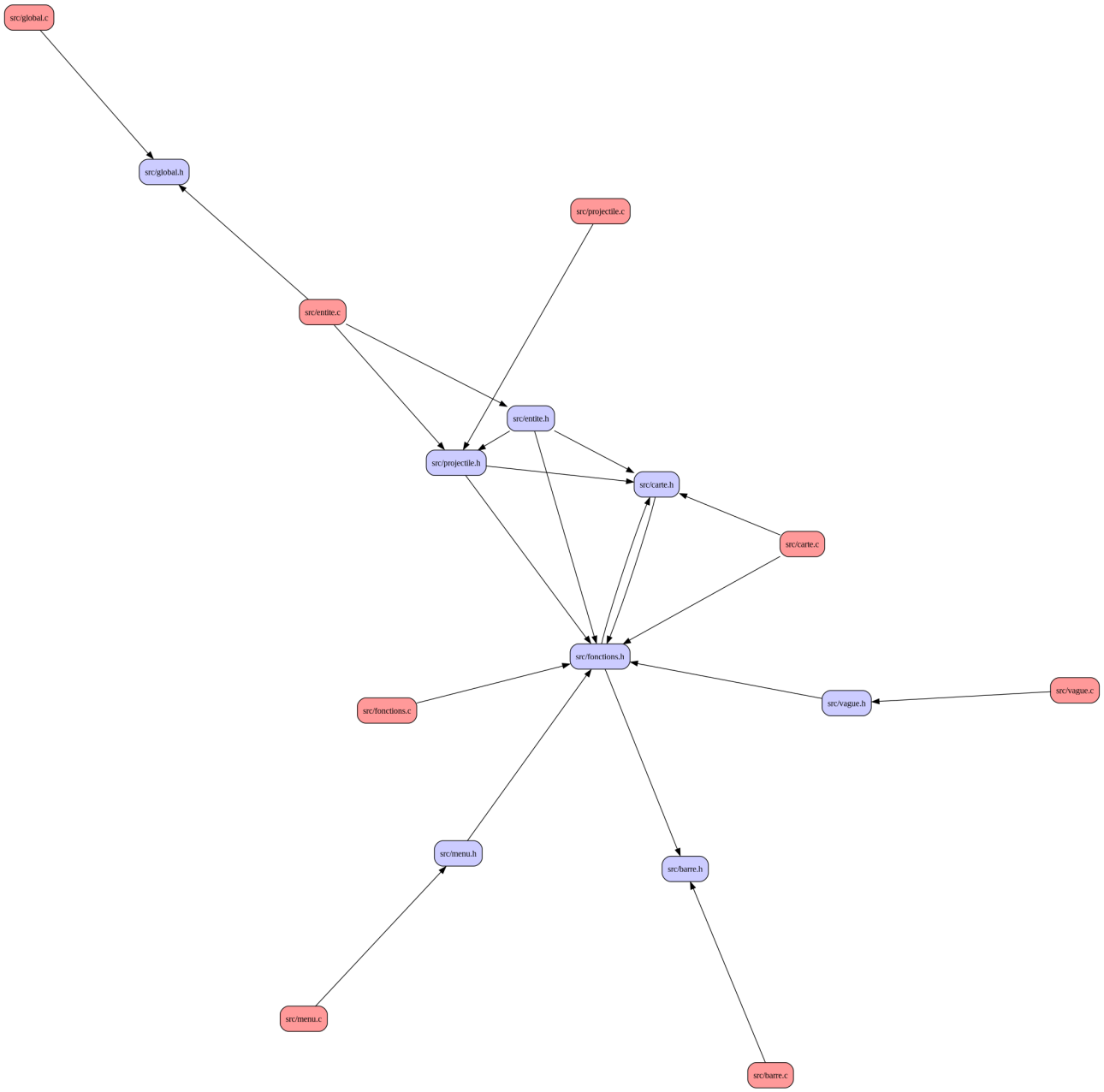


FIGURE 11 – Graphe de dépendance du dossier `src`