Projet Roguelike
Licence SPI 2ème année
Pavard Valentin, Decrand Baptiste et Gerbault Maxime
2015-2016

Référant : L.Barrault

Table des matières

I]Introduction

Dans le cadre de notre projet du semestre 3 de licence SPI, nous avons réalisé un Roguelike. Ce projet s'est développé depuis la plate-forme github à l'adresse suivante: https://github.com/jeyto/rogue_like.
Le Roguelike est un genre de jeu apparu dans les années 80, le joueur contrôle un héros qui parcourt un labyrinthe avec des objectifs à remplir et des obstacles à contourner.
Dans notre Roguelike, le joueur doit parcourir différents niveaux qui se génèrent aléatoirement.
Pour passer un niveau le joueur doit récupérer une clé et ouvrir un coffre qui lui dévoilera la sortie.
Le labyrinthe contient des bonus qui aideront le héros mais aussi des pièges et des monstres qui tenteront de le tuer.

II]Analyse

1)Structures

-typedef enum{nord=0, sud, ouest, est} t_direction :

Cette énumération sert à déterminer la direction dans laquelle les portes seront placées afin de par la suite pouvoir placer les couloirs.

-typedef enum{mur_contour=0, mur, vide, couloir, porte, coffre, cle, bonus, piege, hero, monstre_agressif, monstre_défensif, monstre_inactif, sortie} t_case :

L'énumération sert à déterminer toutes les cases de la matrice(items ou décors) afin de savoir ce qu'il y a dedans et de simplifier l'affichage du jeu par la suite. Elle est utilisée un peu partout dans le code.

-typedef struct{int x; int y;} t_coord :

Cette structure nous est utile afin de déterminer les coordonnées d'une case de la matrice.

-typedef struct{int PV; int score_bonus; t_coord position; int cle; int invisible; int armure;} t_personnage :

La structure sus-nommée est utile afin de gérer les différentes caractéristiques du personnage (vie, score, position, et les valeurs des objectifs ou bonus).

-typedef struct{int x_D; int y_D; int x_A; int y_A;} t_pos :

Cette structure ci sert à connaître la taille d'une pièce grâce à ces positions. En effet, nous stockons les coordonnées de départ D et d'arrivée A.

-typedef struct{int PV; t_coord position; t_case race_mob;} t_mob :

La structure ci-dessus sert à connaître les caractéristiques d'un monstre dont son nombre de points de vie, sa position et sa race.

-typedef struct element{t_mob valeur; struct element *pred; struct element *succ;}t_element :

Cette structure de liste est utilisée pour stocker les caractéristiques des monstres.

2)Main

Dans le fichier *main.c*, nous gérons l'affichage des différents menus. Afin de gérer cela au mieux nous utilisons des *switch* pour pouvoir rediriger l'utilisateur vers les menus qu'il souhaite, ce qui donne une meilleure expérience utilisateur. Le premier *switch* sert de relais entre le menu et les sous menus pour permettre à l'utilisateur d'avoir le plus de choix possibles.
De plus toutes les initialisations liées à la librairie Ncurses sont faites dans le *main.c* afin d'éviter les lignes redondantes et ainsi éviter les potentielles initialisations qui en écrasent d'autres et par la suite n'affiche plus ce que nous souhaitons.

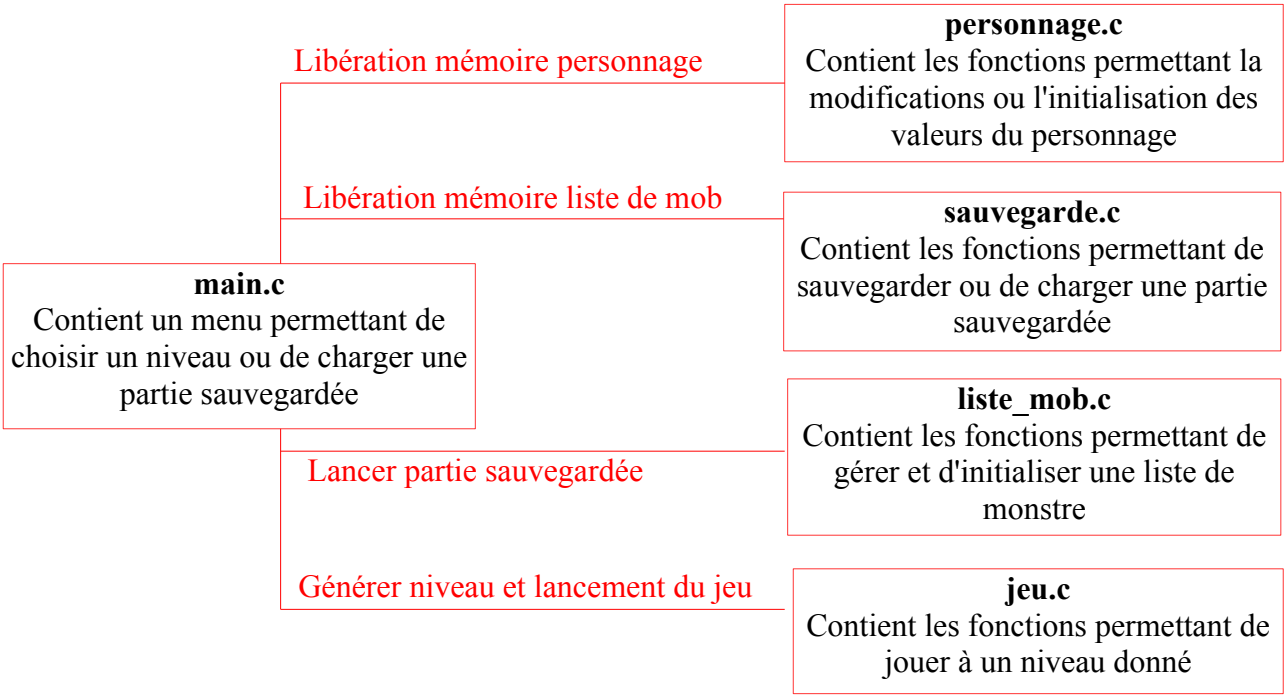

Figure 1 : Diagramme du module main.c

3)Map

Dans le fichier *map.c*, toutes les fonctions concernant la génération de la carte sont regroupées. Toutes ces fonctions permettent la génération des pièces, des couloirs, le placement des items et l'affichage de ceux-ci.

a)Explication des fonctions

-void init_matrice(...) :

Afin d'éviter toutes erreurs dans la gestion de la matrice,comme par exemple des valeurs non cohérentes, nous l'initialisons dans son ensemble. Tout d'abord, nous remplissons la matrice avec la valeur de l'énumération *mur_contour* afin de s'assurer que les contours de la matrice seront bien infranchissables, par la suite nous initialisons les cases par la valeur *mur*, c'est à dire toutes les cases sauf les contours de la matrice. La matrice est alors initialisée avec un encadrement de *mur_contour*.

-void piece(...) :

Cette fonction permet de générer une pièce dans la matrice. Nous créons une pièce de taille hauteur*largeur à partir du point de coordonnées (pt_x,pt_y) du Nord-Ouest au Sud-Est. Chaque pièce est remplie de case *vide*, et possède un encadrement de case *mur_contour*. Enfin, cette fonction prend en paramètre une direction(Nord, Ouest, Sud, Est) et crée une case *porte* au milieu d'un coté choisi de la pièce. Une seule porte est créée par pièce.

-int generer_piece_matrice(...) :

Afin de générer une carte jouable, cette fonction crée un nombre de pièces données dans l'ensemble de la matrice en vérifiant que le placement de la pièce est possible. C'est à dire, qu'en fonction de la taille de la pièce souhaitée, la fonction vérifie qu'au point de coordonnée de départ et à l'emplacement de la future pièce, la matrice ne contient que des cases *mur*.
En fonction du niveau du jeu, cette fonction génère un nombre de pièces différent et de taille différentes. Pour un niveau strictement inférieur à 3, seulement 2 pièces de taille élevée sont créées, alors que pour le niveau 3, le nombre de pièces double et la taille diminue. Enfin afin de remplir la matrice d'un nombre important de pièces, pour les niveaux strictement supérieurs à 3, la taille des pièces est encore plus petite.
La fonction commence par créer une pièce principale à gauche lorsqu'il n'y a que 2 pièces où au milieu de la carte lorsque le nombre de pièces dépasse 2.
Cela permet d'obtenir une pièce centrale, et par conséquent, 2 pièces latérales. Pour chaque pièce, la direction de la porte est alors choisie.
Une fois que toutes ces conditions sont respectées (espace libre, choix de la porte, nombre de pièce), les pièces sont créées.
Enfin, pour chaque pièce créée, nous stockons la position de départ (Nord-Ouest) ainsi que la position d'arrivée (Sud-Est) afin de connaître tous les emplacements des pièces et leur taille. Comme le nombre de pièces varie, cette fonction retourne le nombre de pièces qu'elle a réussi à créer.

-void recherche_chemin_couloir(...) :

Cette fonction recherche le chemin le plus court entre deux portes en calculant le nombre de déplacements séparant chacune d'elle. Elle permet donc de connaître le chemin à parcourir pour rejoindre l'autre pièce. Autrement dit, elle permet la création des couloirs reliant deux pièces. C'est une version modifiée de l'algorithme utilisé dans la gestion des monstres puisque celle-ci calcule également le chemin avoisinant le chemin le plus court afin que les couloirs possèdent une largeur de deux cases.

-int generer_matrice_tot(...) :

Afin de générer la carte dans son ensemble, nous appelons cette fonction. Elle permet de lier les fonctions précédentes afin de générer les pièces et les couloirs. En effet, cette fonction appelle la fonction *int generer_piece_matrice(...)* pour l'ensemble des pièces. Une fois les pièces créées, nous connaissons la localisation de chaque porte et leur nombre. Nous pouvons alors utilisé l'algorithme du plus court chemin afin de relier toutes les portes entre elles. Cela permet que le joueur puisse accéder à toutes les pièces peu importe l'emplacement de son apparition.

-void spawn_item(...) :

Une fois la carte générée, nous avons besoin d'y placer différents items. En fonction des niveaux et donc de la difficulté, les items placés sont différents. La fonction permet donc de placer pour chaque niveau une case *héro*, une case *clé* et une case *coffre* qui correspondent aux objectifs de chaque niveau. Le placement de ces objectifs est fait de tel manière que la case *héro* soit le plus éloignée de la case *coffre*, et que la case *clé* soit entre ces derniers. C'est donc pour cela que nous recherchons la pièce la plus à gauche de la carte, ainsi que celle la plus à droite. Peu importe la carte, la case *héro* sera au milieu de la pièce la plus à gauche, alors que la case *coffre* sera dans l'angle Sud-Est de la pièce la plus à droite.
Chaque niveau a ses caractéristiques :

1: Seule présence des objectifs afin de se familiariser avec le principe du jeu
2 : Apparition de case *monstre inactif*, *piège* et *bonus*. Dans chacune des pièces générées, nous plaçons une case *monstre inactif* et une case *bonus*. Quant aux cases *piège* elles sont générées un peu partout dans la carte si l'emplacement est disponible. Elles vont souvent par 3.
3: Apparition de case *monstre défensif* dans la moitié des salles.
4 : Apparition d'une case *monstre agressif* dans la salle contenant la case *coffre*, afin d'éloigner au maximum la case *héro* et *montre agressif*.
5 : Apparition de case *monstre défensif* dans toutes les salles ainsi que quelques case *monstre agressif* en plus.

-void afficher_matrice(...) :

Comme vu précédemment, la matrice est composée de nombreuses cases qui peuvent être des cases décors ou items. Cette fonction permet tout simplement d'afficher à l'écran un caractère choisi en fonction de chaque élément de l'énumération *t_case*.
Nous avons également fait varier la couleur de l'affichage en fonction des items, fais varier la couleur de la vie du personnage pour savoir combien de points de vie il lui reste afin de rendre le jeu plus intéressant à jouer.

-void afficher_legende(...) :

Dans cette fonction on gère l'affichage de toutes les informations utiles au joueur (vie, armure, numéro de étage, score, s'il a la clé de l'étage ou non) de tel sorte qu'il puisse jouer tout en ayant les informations qu'ils lui permettent de gagner et de savoir où il en est dans le jeu.

-void afficher_ecran(...) :

Cette fonction est la fusion des deux fonctions précédentes qui nous permet d'afficher la matrice et la légende à droite de celle-ci car sans cette fonction l'affichage des deux fonctions simultanées ne serait pas possible à cause de l'utilisation de la librairie Ncurses (cf. *8)Ncurses*) et plus précisément de la fonction *clear()* .

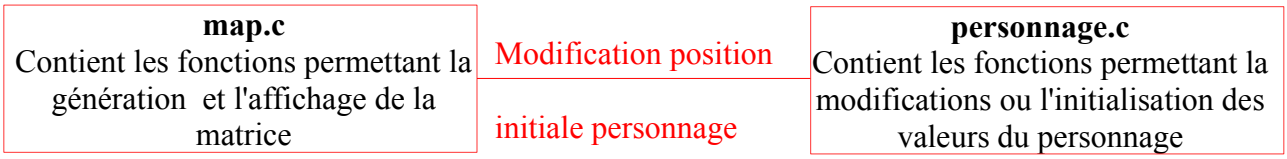

Figure 2 : Diagramme du module map.c

b)Fonctionnement, règles et contraintes lors de la création de la carte

L'ensemble de ces fonctions permet donc la génération de la matrice de jeu. Chaque fonction a son propre but, comme générer une seule pièce, un seul couloir. C'est donc l'utilisation groupée de celle-ci qui permet la génération de la carte et donc de permettre une jouabilité de niveau en niveau.
Pour chaque élément de la carte, un système d'aléatoire a été utilisé. Tous les éléments sont donc placés aléatoirement mais de façon contrôlée.
Tout d'abord, la position de chaque pièce, ainsi que sa taille(largeur et hauteur) sont calculée aléatoirement en vérifiant l'espace disponible lors la création, et sont recalculées si les contraintes ne sont pas respectées. De même, pour la création de la case *porte*, nous utilisons un nombre aléatoire entre 1 et 4 afin que la direction de celle-ci ne soit pas choisie volontairement.
Cependant, la taille des pièces est tout de même comprise entre deux valeurs afin de ne pas avoir des pièces de taille trop petite ou trop grande, limitant l'expérience du jeu. De même, chacune des pièces sont séparées d'un minimum de trois cases pour éviter qu'elles ne soient collées. Les couloirs quant à eux ne peuvent être créé que dans des cases *mur* afin d'éviter qu'ils ne coupent les pièces. Sachant qu'ils sont générés par la recherche du plus court chemin, leur génération n'est également pas choisie. De plus, le nombre de pièces étant souhaité maximal pour des niveaux élevés, nous testons la disponibilité d'espace libre pour la création de nouvelles pièces en recalculant jusqu'à 500 fois la position et la taille des pièces.
Enfin, le placement des items est contrôlé en fonction des pièces, mais nous les plaçons aléatoirement dans une case libre de cette pièce. Cela permet de jouer à un jeu qui n'aura presque jamais la même apparence, ce qui caractérise un Roguelike.

4)Monstre

Ce fichier .c contient les fonctions permettant de gérer la liste de monstres et leurs caractéristiques.

a)Explication fichier liste_mob

-void init_liste_mob(...) :

Cette fonction initialise une liste de type *t_mob* en allouant un emplacement mémoire pour le drapeau. Cette fonction est utilisée dès le lancement du programme.

-int liste_vide_mob(...) :

La fonction ci-dessus renvoie VRAI si la liste est vide et FAUX sinon. Cette fonction est utilisée afin de vérifier si on peut parcourir ou modifier la liste.

-int hors_liste_mob(...) :

Cette fonction renvoie VRAI si l'élément est placé sur le drapeau et FAUX si il est sur un monstre. Cette fonction est utilisée lorsqu'on parcours la liste, elle nous permet de dire quand on a atteint le bout de la liste

-void en_tete(...) :

La fonction place l'élément courant sur le premier monstre de la liste (c'est à dire l'adresse suivant le drapeau). Cette fonction est utilisée lors des parcours de liste afin que aucun élément ne soit oublié.

-void en_queue(...) :

La fonction au dessus place l'élément courant sur le premier monstre de la liste (c'est à dire l'adresse précédent le drapeau). Cette fonction est utilisée lors des parcours de liste afin que aucun élément ne soit oublié.

-void precedent(...) :

La fonction place l'élément courant sur celui qui le précède. Cette fonction permet de parcourir les listes en passant d'un élément à un autre.

-void suivant(...) :

La fonction place l'élément courant sur celui qui le suit. Cette fonction permet de parcourir les listes en passant d'un élément à un autre.

-void valeur_mob(...) :

Cette fonction récupère les caractéristiques du monstre de l'élément courant et le place dans un variable indiquée en paramètre. Cette fonction est utilisée lorsqu'on a besoin des coordonnées du monstre dans les fonctions IA.c ou quand on veut savoir si un monstre à encore des points de vie.

-void modif_mob(...) :

La fonction change la valeur de l'élément courant par celui entré en paramètre . Cette fonction est utilisée pour modifier les coordonnées ou les points de vies des monstres.

-void oter_mob(...) :

La fonction oter_mob supprime l'élément courant et libère la mémoire allouée puis l'élément courant devient le précédent. Cette fonction est utilisée quand un monstre meurt et dans la fonction *vider_liste_mob ().*

-void ajout_droit(...) :

La fonction ajout_droit alloue un place mémoire pour ajouter un monstre à droite de l'élément courant, l'élément courant devient alors l'élément ajouté. Cette fonction est utilisée dans la fonction *init_carac_mob()* et quand on veut ajouter un monstre.

-void ajout_gauche(... :

La fonction ajout_gauche alloue une place mémoire pour ajouter un monstre à gauche de l'élément courant, l'élément courant devient alors l'élément ajouté. Cette fonction est utilisée dans la fonction *init_carac_mob()* et quand on veut ajouter un monstre.

-void vider_liste_mob(...) :

La fonction vider_liste_mob va ôter les éléments un par un en libérant leur place mémoire (elle réutilise la fonction *oter_mob()*) jusqu'à ce que la liste soit vide. Cette fonction est utilisée quand un niveau est terminé pour pouvoir ensuite initialiser (avec *init_carac_mob()*) les monstres contenus dans le nouveau niveau.

-void init_carac_mob(...) :

La fonction init_carac_mob parcours la carte et ajoute à la liste les coordonnées des monstres et leurs caractéristiques. Cette fonction est utilisée à chaque fois que le joueur passe ou lance un niveau.

-int position_mob(...) :

La fonction position_mob place l'élément courant sur le monstre situé aux coordonnées entrées en paramètre et renvoie VRAI si le monstre est trouvé. Cette fonction est utilisée quand le personnage attaque un monstre et qu'il faut sélectionner le monstre correspondant .

-void mob_perte_PV(...) :

La fonction mob_perte_PV fait perdre des PV au monstre correspondant à l'élément courant, si celui en meurt un gain de bonus est apporté au personnage et on supprime le monstre de la liste et de la grille. Cette fonction est utilisé quand le monstre rencontre un piège ou quand il est attaqué par le personnage.

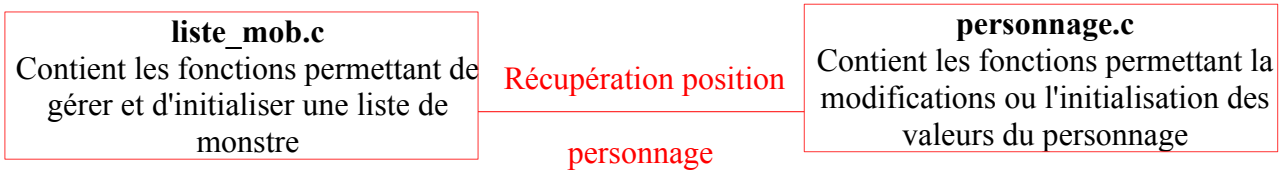

Figure 3: Diagramme du module liste_mob.c

b)Explication fichier IA.c

Le fichier IA.c contient l'ensemble des fonctions qui vont gérer les déplacements des monstres .

-void permutation_monstre_agr(...)

-void permutation_monstre_def(...)

-void permutation_monstre_alea(...)

Ces fonctions font avancer les monstres d'une case à une autre quand la case est vide, ou bien elle réalise des interactions en fonction de l'objet contenu dans la case *arr* et le monstre.

-void generation_mob_suivante(...) :

La fonction generation_mob_suivante va générer un déplacement pour chaque monstre en fonction de leurs caractéristiques et de leurs milieux.
Pour cela elle parcourt la liste de monstres et en fonction de leurs types, elle recherche un chemin disponible vers le héros et déplace le monstre, sinon quand le chemin n'est pas disponible le monstre se déplace aléatoirement d'une case.

-int recherche_chemin_monstre_def(...) :

-int recherche_chemin_monstre_agr(t...) :

Les fonctions recherche_chemin_monstre créent un chemin entre le monstre et le héros si le chemin est possible le monstre se place sur la première case du chemin et les fonctions renvoient VRAI sinon le monstre ne bouge pas et les fonctions renvoient FAUX.
L'algorithme de ces fonctions est expliqué en détails.

-void chemin_aleatoire(...) :

La fonction chemin_aleatoire déplace aléatoirement un monstre d'une case autour de lui.

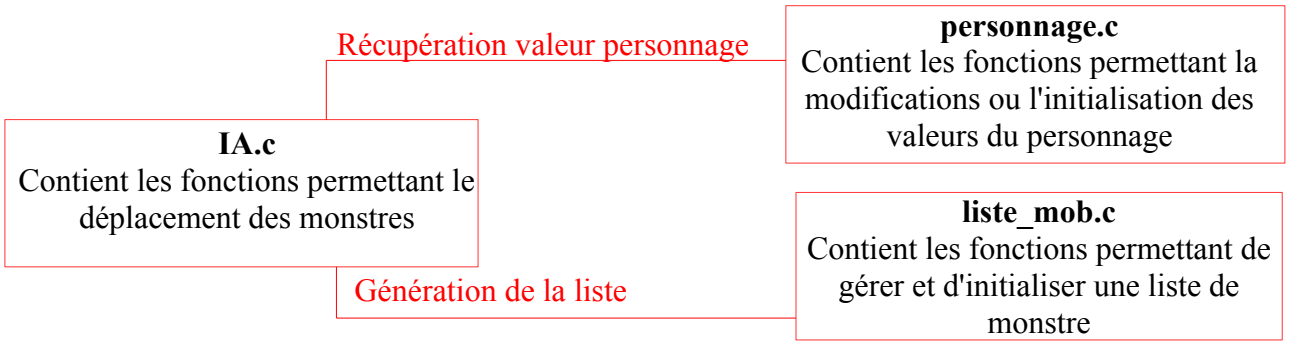

Figure 4 : Diagramme du module IA.c

c)Fonctionnement dans le jeu

Les fonctions contenus dans *IA.c* et *liste_mob.c* permettent à l'ensemble des monstres d'évoluer sur la carte à chaque génération en fonction des déplacements et des actions du personnage ainsi que les éléments de la carte.

d)Règles et contraintes lors des déplacements des monstres

On distingue trois types de monstres:

-le monstre agressif contient initialement 2 PV et a un comportement agressif, c'est à dire qu'il peut se déplacer comme le personnage et il cherche à rejoindre et attaquer le personnage où qu'il soit .

-le monstre défensif contient initialement 1 PV et a un comportement défensif, c'est à dire qu'il se déplace aléatoirement lorsqu'il ne peut pas atteindre le personnage et devient agressif quand il le peut.

Le monstre défensif ne peut pas traverser les portes.

-le monstre inactif contient 1 PV et a un comportement aléatoire, c'est à dire qu'il se déplace aléatoirement, il ne peut pas traverser les portes et s'il mange un bonus un monstre défensif apparaît.

Chaque monstre perd un point de vie s'il marche sur un piège.

5)Personnage

Les fonctions contenus dans ce fichier permet de gérer l'ensemble des caractéristiques du personnage (points de vie ,coordonnées ,bonus,...).

-void init_personnage(...) :

La fonction init_personnage permet d'allouer en mémoire la variable personnage de type *structure t_personnage*. On utilisera cette fonction au tout début du programme.

-void init_valeur_personnage(...) :

La fonction init_valeur_personnage va initialiser l'ensemble des caractéristiques du personnage. On utilisera cette fonction lorsque le joueur lancera une partie.

-void init_etage_personnage(...) :

La fonction init_valeur_personnage va initialiser l'ensemble des caractéristiques du personnage hormis le score. On utilisera cette fonction à chaque changement de niveau (le score est conservé).

-void gain_bonus_personnage(...) :

La fonction gain_bonus_personnage attribue un gain (valeur en paramètre) au score du personnage. Cette fonction est utilisée lorsque le personnage rencontre des bonus ou tue des monstres.

-void valeur_personnage(...) :

La fonction valeur_personnage attribue l'ensemble des valeurs du personnage au pointeur valeur en paramètre. Cette fonction est utilisée lors de la sauvegarde.

-void modif_personnage(...) :

La fonction modif_personnage change l'ensemble des valeurs du personnage pour celles entrées en paramètre.

-void modif_position_personnage(...) :

La fonction modif_position_personnage change la position du personnage pour celle entrée en paramètre. Cette fonction est utilisée lorsque le personnage est déplacé.

-void valeur_position_personnage(...) :

La fonction permet de récupérer les coordonnées du personnage dans la variable *pos*. Cette fonction est utilisée à chaque fois où l'on veut connaître les coordonnées du personnage.

-void degat_personnage(...) :

La fonction degat_personnage enlève un point d'armure et si le personnage n'a pas de point d'armure elle enlève un point de vie. Cette fonction est utilisée quand le personnage se fait attaquer par un monstre ou quand il marche sur un piège.

-int valeur_cle_personnage(...) :

La fonction valeur_cle_personnage renvoie la valeur de la clé. Cette fonction est utilisée pour savoir si le personnage peut ouvrir le coffre.

-void modif_cle_personnage(...) :

La fonction modif_cle_personnage permet de modifier la valeur de la clé. Cette fonction est utilisée quand le personnage trouve la clé.

-int valeur_PV_personnage(...) :

La fonction valeur_PV_personnage renvoie la valeur des points de vie du personnage. Cette fonction est utilisée pour savoir si le personnage est encore en vie et pour afficher ses points de vie à l'écran.

-int valeur_score_personnage(...) :

La fonction valeur_score_personnage renvoie la valeur du score du personnage. Cette fonction est utilisée pour afficher le score à l'écran.

-int valeur_invi_personnage(...) :

La fonction valeur_invi_personnage renvoie la valeur de l'invisibilité du personnage. Cette fonction est utilisée pour savoir si le personnage a été récemment attaqué et s'il est immunisé.

-void modif_invi_personnage(...) :

La fonction modif_invi_personnage permet de modifier la valeur d'invisibilité du personnage. Cette fonction est utilisée quand le personnage est attaqué, il possède alors l'immunité qui disparaît au cours du temps.

-int valeur_armure_personnage(...) :

La fonction valeur_armure_personnage renvoie la valeur de l'armure du personnage. Cette fonction est utilisée dans *degat_personnage()*.

6)Jeu

L'ensemble des fonctions est utilisé dans le fichier *jeu.c* et *main.c*. En effet, ces fichiers permettent de créer le jeu en lui-même, mais il est nécessaire de créer des fonctions permettant de lier les autres afin de pouvoir jouer au jeu.

a)Explication fonction

-void generation_level(...) :

Cette fonction permet la génération complète d'une carte, de l'initialisation de la matrice jusqu'à l'affichage en passant pas la génération des pièces, de la gestion des items et de l'initialisation des monstres. Cela permet de regrouper l'ensemble des fonctions et de créer un niveau de difficulté donnée.

-void game_message(...) :

Cette fonction permet d'afficher le message final d'une partie en fonction des résultats du joueur. Par exemple, si le joueur parvient à finir tous les niveaux jusqu'au cinquième, le message « good game » apparaîtra, alors que s'il perd c'est le message « game over » qui apparaîtra.
Cette fonction remplace toutes les valeurs de la matrice à l'aide des valeurs contenues dans un fichier texte. En effet, deux fichiers textes ont été créés afin de pouvoir remplir la matrice de case *mur* et *vide* pour permettre l'affichage d'un message. Nous avons besoin de la fonction qui convertit des entiers en éléments de l'énumération *t_case* du fichier *sauvegarde.c*.

-void jeu(...) :

Cette fonction permet la gestion du jeu. Tout d'abord, elle permet de gérer la gestion vers le niveau suivant :

1: Positionner la case *sortie* proche du coffre sans l'afficher
2: Sauvegarde au début du niveau

Ensuite, nous faisons une boucle tant que le joueur a de la vie ou qu'il n'a pas fini le niveau, afin de :

3 : Déplacer les monstres en fonction du temps. C'est à dire qu'une nouvelle génération de monstre est mis en place toutes les 1,5 secondes même si le joueur ne joue pas.
4 : Déplacer le personnage en fonction de la touche qu'il a pressé. En effet, le joueur peut se déplacer en appuyant sur Z, Q, S ou D, ou bien z, q, s ou d, et même les touches directionnelle. En fonction de la case où il veut se déplacer, la fonction jeu va traiter sa demande, voir si son déplacement est possible, et faire l'action adéquate.

Enfin, la fonction va traiter la sortie de la boucle. Elle va soit afficher un message « game over» si le joueur n'a plus de vie, ou bien va vérifier si le niveau est réussi. Si le niveau réussi est inférieur à 5, alors la fonction va générer une nouvelle carte avec un niveau supérieur. En revanche, si le niveau vaut 5, alors un message « good game » va s'afficher à l'écran. Le jeu est fini.

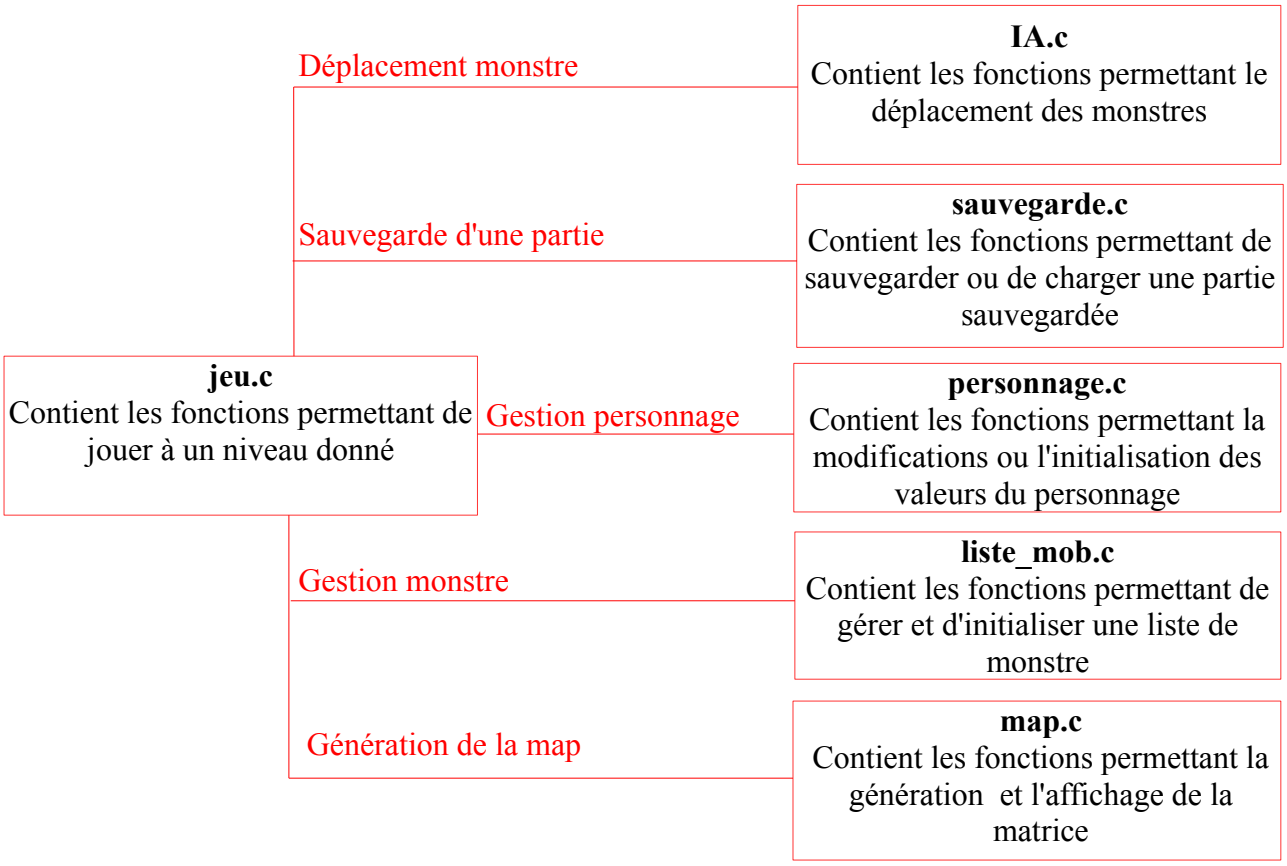

Figure 5: Diagramme du module jeu.c

b)Règle du jeu

Le joueur apparaîtra dans une pièce située à gauche de la carte. Il devra éviter les monstres et les pièges, récupérer des bonus, la clé puis le coffre, et enfin sortir par la case *sortie*.
Nous venons de voir qu'en fonction d'un déplacement demandé, la fonction jeu va réaliser un déplacement ou une action. Si le joueur demande à se déplacer dans une case *vide* ou *couloir,* alors le déplacement s'effectue et la position du héros varie. En revanche, s'il souhaite se déplacer dans une case *mur* ou *mur contour* alors il reste immobile. Dans la matrice des cases *porte* sont placées dans chaque pièce. Seul le joueur et les monstres agressifs peuvent les passer. Cela permet de bloquer les monstres défensifs et inactifs dans les pièces. Si le joueur décide de passer une porte, alors il quitte la pièce et les monstres défensifs arrêteront de le poursuivre. Cependant, s'il se heurte à un piège, ou à n'importe quel monstre, il perdra une vie. Le joueur a tout de même la possibilité de tuer un monstre, en se déplaçant sur celui-ci. Si le monstre a plusieurs vies, il faudra alors attaquer plusieurs fois avant de tuer le monstre. De ce fait, si le monstre vient sur le héros, le joueur perd une vie, mais si le joueur déplace son héros sur un monstre, il réalise une attaque.
De plus, à chaque monstre tué ou bonus ramassé, le joueur gagne des points. Il peut même avoir la chance de gagner un point d'armure en ramassant un bonus, qui lui permet d'éviter de perdre une vie

lors du prochain combat. Enfin, si le joueur souhaite terminé le niveau, il faudra qu'il récupère la clé, puis le coffre. Si celui-ci souhaite tout d'abord récupérer le coffre avant la clé, un message lui avertissant qu'il n'a pas la clé apparaîtra. Une fois la clé prise, et si le joueur récupère le coffre, une nouvelle pièce apparaît. Il s'agit d'un escalier avec une case *sortie*. Si le joueur se dirige sur cette case, il termine le niveau. Un nouveau niveau est généré, il conserve son score, récupère sa vie et continue les niveaux jusqu'au cinquième.

7) sauvegarde

Ce fichier contient les fonctions permettant de sauvegarder une partie dans un fichier texte, ainsi que la fonction permettant de générer une partie à partir d'un fichier texte.

a)Explication fonctions

-void convertion_int_enum(...) :

L'énumération t_case contient différents noms d'items, chacun étant représenté par un entier dans la matrice. Afin de pouvoir remplir une matrice en récupérant des entiers d'un fichier texte par exemple, dont chaque case de celle-ci est un élément d'une énumération, nous avons besoin d'une conversion. Cette fonction est donc utilisable pour l'énumération *t_case*. En effet, pour un entier donné, compris entre 0 et 13 (14 éléments dans l'énumération), la case de la matrice possédant un entier est remplacée par la valeur de l'énumération liée à cet entier. Par exemple pour la case *matrice[i][j]*, si celle-ci vaut 9, alors nous indiquons que la case *matrice[i][j]* vaut *hero*.

-void convertion_int_enum_monstre(...) :

De la même manière, si nous avons besoin de convertir un entier en sa valeur dans une énumération nous utilisons cette fonction. La différence avec la fonction précédente est qu'elle n'est utilisable que pour les types de monstres, et qu'elle ne permet pas de remplacer une valeur d'une case, mais plutôt une caractéristique d'un monstre.

-void sauvegarde_map(...) :

Au cours d'une partie, nous avons besoin de sauvegarder l'ensemble des caractéristiques du niveau. Pour cela nous faisons appel à cette fonction. Elle permet de sauvegarder dans un fichier texte nommé *save_map.txt* :

-le niveau de la partie,

-l'ensemble des caractéristiques du héros : score réalisé, nombre de vie, sa position dans la matrice au moment de la sauvegarde, l'indication s'il a pris la clé, ainsi que l'indication d'invisibilité,

-l'ensemble des caractéristiques des monstres : le type de monstre, son nombre de vies, ainsi que sa position dans la matrice,

-l'ensemble des cases de la matrice sont représentées par des entiers.

-int generer_map_sauvegarde(...) :

A l'inverse de la fonction précédente, cette fonction permet de générer le niveau grâce à une sauvegarde et donc de permettre au joueur et aux monstres de se déplacer depuis leur position sauvegardée. Nous récupérons les données correspondant au héros et nous initialisons ses caractéristiques à partir ce celles-ci. De même, nous remplissons la liste contenant toutes les indications des monstres, puis nous remplissons la matrice en récupérant la valeur de chaque case en la convertissant avec les fonctions ci-dessus.

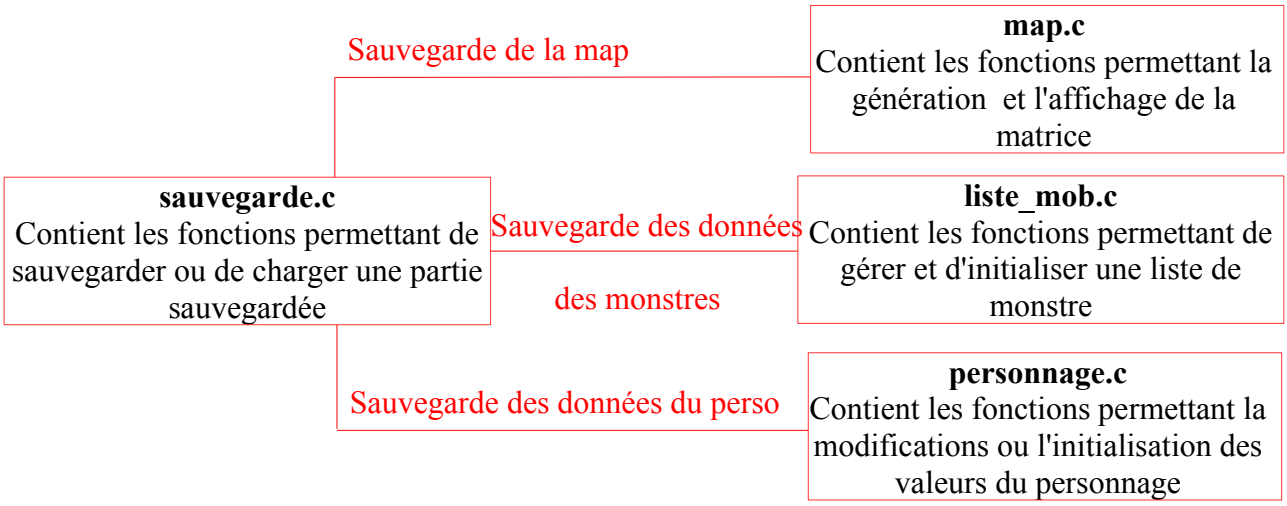

Figure 6: Diagramme du module sauvegarde.c

b)Fonctionnement dans le jeu

A chaque début de niveau, nous sauvegardons toutes les données du niveau actuel, afin de pouvoir générer de nouveau le niveau. En effet, la création de la carte ainsi que la position des monstres est aléatoire. Nous ne pouvons pas, ou très rarement, retrouver la même composition en relançant le niveau. Si par mégarde, ou par abandon, le joueur décide de quitter sa partie, il pourra alors la reprendre sans soucis. De même, son score et sa vie sont alors conservés.

8)Ncurses

C'est la librairie que nous utilisons afin d'avoir un affichage dynamique de notre jeu.

a)Explication de la librairie

Dans le but d'avoir un affichage dynamique basé sur le temps et non sur les déplacements du joueur mais aussi libérer le joueur d'avoir à valider la direction de son déplacement avec la touche entrée à chaque fois, nous avons opté pour l'utilisation de la librairie Ncurses qui nous permet d'outre-passer, via quelques fonctions, l'affichage non dynamique de notre jeu. Elle nous permet aussi d'utiliser quelques fonctionnalités simplifiant l'affichage du jeu mais aussi de rendre l'expérience utilisateur plus épanouissante.

b)Explication modification du code

Dans la version initiale de notre jeu, l'affichage est géré sans la librairie Ncurses ce qui ne convient pas au projet ni à notre vision que nous nous faisions de notre jeu, car la gestion des déplacements se fait encore avec l'attente d'une confirmation. C'est à ce moment que la librairie Ncurses intervient, en modifiant les *printf()* en *mvprintw()* il nous est possible de pouvoir afficher les éléments de notre jeu à l'exact emplacement auquel nous voulons qu'ils se trouvent. Dans la fonction *mvprintw()* nous rentrons au minimum 3 paramètres qui sont la ligne du terminal, la colonne du terminal et enfin le contenu que nous souhaitons afficher à l'écran, de ce fait nous pouvons placer ce que nous souhaitons où nous souhaitons sans avoir à se soucier des éventuels problèmes de superpositions de caractères qui viendraient gâcher l'affichage du jeu.
De plus grâce a la fonction *getch()* qui attend que l'utilisateur appuie sur une touche sans que celui-ci ait à confirmer sa sélection, contrairement au *scanf()* qui lui attend la confirmation de la sélection avant de passer à la prochaine instruction, cependant cette fonction reste une fonction bloquante ce qui veut dire que si le joueur appuie sur aucune touche le jeu sera figé et attendra l'appuie d'une touche avant d'afficher la prochaine génération. A ce stade le jeu n'est pas tel que nous le souhaitons car il est encore généré en fonction des déplacements du joueur de part la caractéristique bloquante de la fonction *getch()*.

Il nous a donc fallu trouver une solution afin de le générer à l'aide du temps et donc supprimer la caractéristique bloquante du g*etch()*, c'est ainsi que nous sommes arrivé à la solution d'utiliser la fonction *kbhit()*. Cette fonction nous permet, à l'aide de fonctions intermédiaires, de supprimer l'effet bloquant de la fonction *getch()* en vérifiant à chaque itération si le joueur a appuyé sur une touche, si oui il videra le buffer ce qui générera les déplacements du joueur sinon le jouer sera immobile tandis que les monstres eux continueront à se déplacer.
Seul petit bémol à l'utilisation de la librairie Ncurses, nous sommes obligé d'utiliser les fonctions *clear()* et *refresh()* qui font respectivement la suppression du contenu de l'écran et à l'inverse l'affichage du contenu, sans cela l'affichage ne pourrait se faire ou tout du moins pas avec Ncurses.

III]Explication fonction difficile

a)Calcul chemin le plus court

Pour calculer le chemin le plus court entre un point de départ et un point d'arrivée dans une carte contenant des obstacles, on procède en trois étapes :

1/ La première étape consiste a créé une matrice jumelle de la carte dans laquelle on note l'ensemble des obstacles par -2, l'ensemble des cases atteignables par -1 et le point de départ par 0. Lorsque la valeur d'une case est positive, elle représente la distance (c'est-à-dire le nombre de case) la séparant du point de départ ainsi le nombre 0 du point de départ signifie qu'il y a 0 case à parcourir pour atteindre le point de départ.
2/ Cette étape est la plus importante, elle va construire l'ensemble des chemins depuis la position de départ en notant pour chaque case parcouru leurs distances avec ce point de départ. Pour cela l'algorithme va parcourir la matrice et ne s'arrêtera que quand le point d'arrivée est atteint ou bien quand il ne pourra plus avancer, à chaque parcours de la matrice lorsqu'il trouve une case non atteinte (-1) qui possède un ou plusieurs voisins que l'on a atteint (valeur>=0) alors cette case devient une case atteinte avec une valeur correspondante à la valeur la plus petite parmi ses voisins plus un.

Si pendant un des parcours de la matrice aucune case non atteinte n'a été changée alors cela signifie que le chemin entre le point de départ et le point d'arrivée n'est pas possible, la fonction retourne 0 et s'arrête. Sinon le point d'arrivée est atteint, la boucle s'arrête et l'algorithme passe à l'étape 3.

3/ Cette étape permet de récupérer les coordonnées de la première case du chemin le plus court. Pour retrouver cette case on parcourt le chemin à l'envers:

- on se place sur la position d'arrivé et on va remonter le chemin jusqu'à atteindre la première case.

La case d'arrivée contient une certaine valeur, on cherche alors parmi ses voisins la case contenant cette valeur moins un, puis on se place sur cette case et ainsi de suite jusqu'à ce que la valeur de la case soit de un ce qui correspondra à la première case que l'on recherchait.

IV]Organisation du travail

Pendant ce projet nous nous sommes répartis les tâches de telle sorte que le projet avance le plus vite possible mais aussi le plus logiquement possible c'est à dire que tout le monde ne travaille pas sur la même partie afin d'obtenir trois fois le même code mais en contre-partie que certains aspects du jeu soient négligés au vu du temps impartis.

-Baptiste D. s'est donc occupé de la partie génération de monstre (recherche du chemin pour venir tuer le joueur, passivité ou bien agressivité à l'approche du joueur, caractéristiques du joueur quand il se prend des dégâts de la part des monstres ou des pièges).

-Valentin P. s'est quant à lui occupé de toute la création de la carte (création des salles, leurs positionnement aléatoire, la création aléatoire des couloirs, gestion des items à mettre dans chaque salle), de la gestion du jeu avec la gestion des niveaux ainsi que la sauvegarde et le chargement de partie.

-Maxime G. s'est lui occupé de tout l'affichage du jeu dans le terminal (utilisation de la librairie Ncurses, affichage du jeu, gestion de la place occupé par la légende et le jeu).

VI] Conclusion et résultats

Durant ce projet nous avons pu mettre en place un travail d'équipe et une communication efficace qui nous a permis d'arriver à un jeu fonctionnel. Ce projet nous a aussi permis de mettre en pratique des notions vues en cours d'algorithme et programmation et algorithme et programmation avancée. De plus, les objectifs que nous nous étions fixés au début du projet ont tous été remplis dont certains objectifs annexes que nous voulions réaliser, s'il nous restait du temps, ont pu être implémentés au jeu.

Comme dit précédemment la version finale du jeu est comme nous voulions le voir fonctionner, le projet est donc pour nous une réussite au vu du travail effectué et des résultats obtenus.