COMPTE RENDU PG110

FELLAH Hicham & AHALLI Mohamed

ENSEIRB-MATMECA

23/04/2021

Sommaire

1. Introduction

L'objectif de ce projet est d'implémenter un jeu 2D. De manière générale le code du jeu fonctionne par une initialisation *game_new()* puis une boucle principale qui parcourt les fonctions qui gèrent : la mise à jour *game_update()* et l'affichage des images *game_display()*. Ainsi, le travail commence par un débogage concernant le déplacement du joueur et ses interactions avec les décors. Après, le travail est concentré sur l'ajout des fonctionnalités afin de gérer la gestion des différents éléments du jeu, notamment les monstres, les bombes, les bonus, les malus et les vies. Avant de finir le travail sur la gestion de pause et de sauvegarde et chargement de partie avec une interaction supplémentaire à partir d'un menu.

2. Gestion de déplacement

Le mouvement du joueur est géré dans la fonction *player_move()* qui modifie la position du joueur selon la direction modifiée dans la fonction *player_set_current_way() ,* et en utilisant la fonction *player_move_aux()* que l'interaction entre le joueur et les décors est gérée.

3. Gestion des mondes

Les données concernant chaque map sont stockées dans le dossier map où chaque carte est codée dans un fichier texte nommé map_N avec N le numéro du niveau. La gestion des mondes est gérée par la fonction *map_get_numero()* qui prend en argument le numéro de la carte à lire et retourne la map demandée. Elle est appelée à l’initialisation et au passage par une porte.

Au début la porte est fermée et après l’obtention d’une clef, la porte s'ouvre. À chaque passage par une porte ouverte, qui était fermée, le joueur passe vers le niveau supérieur, ainsi la valeur level de la structure game est incrémentée. Dans les cartes où deux portes existent, la porte déjà ouverte mène vers le niveau inférieur. Dans la dernière map, l’ajout d’une porte, qui s’ouvre aussi par la clef de la map, permet d’augmenter la difficulté pour rendre l'accessibilité à la princesse plus intéressante.

4. Gestion des bombes

L’idée de la gestion des bombes consiste à créer une file qui contient des bombes de la structure *Element_bomb*, son premier élément est pointé par un pointeur de la structure *bomb_ancienne*. La fonction *Bomb_update_global()* [1] parcourt l’ensemble de la file pour modifier l’état de toutes les bombes à la fois, mais cela est possible qu’après avoir inséré au moins une fois une bombe dans la file avec la fonction *insertion()* d’où l’utilité de la condition au début de la fonction [1]. Après, dans la boucle, la fonction *bomb_update()* permet de modifier à chaque fois l’état de la bombe jusqu’à l’explosion et contrôler la propagation des explosions et l'explosion des caisses. Le contrôle de la durée de changement d’état est géré par la fonction *SDL_GetTicks()* et la variable *temps_ref* qui correspond à l’instant du lancement de chaque bombe, ainsi une comparaison entre la différence entre ces deux valeurs et les multiples de 1000 permet une attente d’une seconde avant le prochain état. Après l’explosion, la bombe est supprimée de la file par la fonction *supprimer()*. Le choix d’utiliser une file se justifie par la possibilité défiler et donc supprimer la première bombe ajoutée de la file, et par l’optimisation de l’utilisation de la mémoire et l’ignorance du nombre exacte des bombes que le joueur peut avoir

à chaque partie, de plus ce choix s'adapte pour n'importe quelle amélioration futur du jeu si elle nécessite une utilisation infinie des bombes.

5. Gestion des bonus et malus

L'apparition des bonus et des malus est gérée par la fonction *set_bonus_type()* qui est incluse dans la fonction *bomb_update()*, le contenue des caisses apparaît suivant une loi de probabilité uniforme, et l'effet des bonus est mis en place dans la fonction *player_move_aux()* dans *"case CELL_BONUS"*.

6. Gestion des vies

Dans le cas d'une explosion il faut distinguer deux cas possible :
Soit la bombe est déjà explosée et c'est le joueur qui avance vers la case explosée : dans ce cas la diminution de la vie se fait dans la fonction *player_move_aux() dans "case CELL_BOMB"*. Soit le joueur est placé dans une zone accessible par la portée de l'explosion : dans ce cas la diminution de la vie se fait dans la fonction *bomb_update()* dans le "case explosion" en utilisant la fonction *player_dec_nb_life()*. L'interaction avec les monstres fait diminuer aussi la vie dans la fonction *player_move_aux().* Si le nombre de vie du joueur est négatif une image de "game over" sera affichée pour demander à l'utilisateur s'il veut rejouer ou quitter la partie, cela est traité dans la fonction *game_update().*

7. Gestion des monstres

Pour la gestion des monstres, nous avons opté pour un tableau où chaque case est un pointeur vers un monstre. En parcourant la carte entièrement lors de son chargement, et dès qu'une case de type "CELL_MONSTER" est repérée on incrémente le paramètre *nb_monster*. D'une part, l'allocation de la mémoire de fait pour l'apparition du monstre par la fonction *game_set_monster()*. D'autre part on a implémenté la fonction *update_monster_direction()* qui retourne aléatoirement une direction pour le monstre par une loi probabiliste uniforme. Anisi la fonction *monster_update()* parcourt le tableau des monstres et met à jour leur mouvement en faisant appel à *update_monster_direction()*. À chaque passage à un niveau supérieur, la vitesse des monstres augmente en utilisant l'astuce de la divisibilité du temps actuel avec une variable qui dépend du niveau et ayant à chaque fois une valeur plus petite pour une vitesse plus grande à l'aide de la fonction *get_vitesse()* pour le passage vers un niveau supérieur.

8. Gestion du panneau d'informations

Le panneau d'informations contient le nombre de vies, de bombes, de clefs, la portée des bombes et du level atteint, toutes ces informations se mettent à jour correctement et s'affichent grâce à la fonction *game_banner_display()*. l'idée consiste à diviser le panneau en 9 cases de taille *SIZE_BLOC* et appeler la fonction *window_display_image()* avec l'image correspondante.

9. Gestion de pause et de sauvegarde et chargement de partie

La gestion de pause est gérée par la fonction *pause_game()* appelée dans la fonction *input_keyboard()*. L'idée est de créer une boucle while qui affiche une image de pause tant que l'utilisateur n'a pas encore cliquer une autre fois sur p.

La gestion de sauvegarde est gérée par la fonction *save_game()* appelée dans la fonction *input_keyboard()* pour une sauvegarde automatique si la partie est quittée. L'idée consiste à récupérer plusieurs données sur la partie à cet instant et les sauvegarder dans un fichier "save_game" dans le dossier "*save*". Concernant la map actuelle, son codage est sauvegardé dans le fichier "map_5" dans le dossier "map"

La gestion de chargement est gérée par la fonction *load_game()* appelée au main.c dans le menu de départ. L'idée consiste à récupérer les informations contenues dans les fichiers de sauvegardes dans des variables et les insérer dans les structures associées par les trois fonctions: *player_set_data(), game_set_data(), bomb_monster_to_load().*