Compte rendu PG110 : BOMBEIRB

GABRIEL Guillaume & FRANCHI Loïc

ENSEIRB-MATMECA

Table des matières

1 Introduction

Le jeu est organisé de manières à séparer l'affichage des objets (caisses, bonus, portes...) de la gestion physique des entités (Personnage, Bombes, Monstres). Le jeu est sans cesse réactualisé aussi bien pour l'affichage que pour la mécanique du jeu. Un *game_loop* regarde sans cesse si un évènement a eu lieu (pose d'une bombe, naissance d'un monstre, déplacement du joueur).

2 Gestion des déplacements

La gestion des déplacements se fait dans la fonction *player_move*(), on divise les déplacements en quatre cas ($NORTH, EAST, SOUTH, WEST$) et on vérifie pour chacun si la position ou l'on veut se déplacer est bien dans les limites de la carte. On vérifie ensuite si l'on peut se déplacer suivant le type de la cellule suivante (fonction *player_move_aux*()). Le cas des caisses est également géré dans la fonction *player_move*(), on fait appel à la fonction *move_case* pour savoir si on peut déplacer la caisse (on regarde ce qui se trouve derrière la caisse et si elle se trouve aux limites de la map) et si c'est le cas on la déplace.

3 Gestion des Mondes

Les cartes sont enregistrées dans le dossier *data* et sont numérotées de 1 à 7. La carte numéro 0 représente le dernier numéro dans notre cas.

3.1 Chargement des cartes

Les cartes sont chargées dans le jeu avec la fonction *map_get*() qui utilise un fscanf pour lire chaque cellule. Elles sont chargées au fur et à mesure que le joueur passe les niveaux. Cette fonction prend en argument le numéro de la carte à lire et retourne la map demandée. Cette fonction est donc appelé à l'initialisation mais aussi lorsque le joueur passe une porte.

3.2 Gestion des portes

Lorsque l'on arrive face à une porte, on regarde tout d'abord (dans *player_move_aux*() si elle est fermée ou non et si le joueur possède une clé (la possession de la clé se fait dans la structure du joueur). Si les conditions requises pour passer la porte sont validées, on récupère le numéro du prochain niveau avec la fonction *next_level*() et on charge la map correspondante. Si les conditions requises pour passer la porte ne sont pas validées, le joueur reste bloqué devant la porte.

3.3 Gestion du panneau d'informations

Les informations relatives au nombre de vies, au nombre de bombes, leur portée, la présence d'une clé dans l'inventaire et le numéro de niveau sont stockées dans la structure du *player*. L'affichage de ces informations se fait grâce à la fonction *game_banner_display*() qui est appelé dans la fonction *game_display*(). Cette fonction récupère tout les attributs du joueur de façon continue, et les affiche dans la bannière à l'aide de *windows_diplay_image*().

4 Gestion des Bombes

Les bombes ont été implémentées à l'aide d'une liste chaînée inversée. En effet, pour la gestion des explosions il a été plus facile de supprimer le dernier élément de la liste, celui-ci correspond à la bombe la plus ancienne du jeu. Lors de l'explosion d'une bombe on regarde sur chaque direction ($NORTH, EAST, SOUTH, WEST$) puis sur chaque case d'une direction les évènements possibles (destruction de bonus, de caisses). On se concentrera sur une seule direction ($NORTH$) pour les explications. L'explosion possède sa propre cellule, à savoir un $CELL\_FIRE$.

On considère ici deux caisses à la suite. La *game_loop* du jeu a posé des problèmes quand à la gestion des explosions de caisses et des bonus. En effet, après explosion de la bombe, au premier tour de boucle du jeu où l'on rencontre la première $CELL\_CASE$ celle ci est transformé en un $CELL\_EMPTY$. Au tour suivant la première caisse n'est plus là, mais la deuxième $CELL\_CASE$ subissait alors le même sort, sa destruction.

Pour résoudre ce problème, nous avons alors décidé d'inclure à la structure de la bombe un tableau contenant la position de la première caisse qui sera rencontré, juste avant l'explosion de la bombe. Cette valeur (ici $block[NORTH]$) stoppe la propagation de l'explosion. Pour ce faire on choisit le min entre entre ces deux valeurs à l'aide de la macro suivante : $MIN(bomb\_range, block[NORTH])$.

5 Gestion des bonus et des malus

Lorsque le joueur se déplace sur un $CELL\_BONUS$, on récupère le type de bonus avec la fonction *map_get_sub_type*() et on agit en fonction du bonus pour augmenter/diminuer les attributs du *player* suivant le bonus rencontré. Cela est fait dans la fonction *player_move_aux*().

6 Gestion des vies

Dans le cas de la rencontre avec un monstre, on appelle la fonction *player_nb_hp_dec*() qui diminue la vie du joueur ou stoppe le jeu si sa vie est trop faible. Lorsque le monstre se déplace, on appelle la fonction *player_attacked* pour voir si le monstre se trouve sur le joueur et pouvoir réduire ses hp le cas échéant. Pour le joueur c'est si il marche sur un $CELL\_MONSTER$ on lui enlève de la vie en conséquence. La baisse de vie du joueur se fait également lors de l'explosion d'une bombe dans la fonction *set_fire_xy*(). Si le feu passe sur une $CELL\_PLAYER$, toute de suite après l'explosion, on regarde si le joueur n'a pas été touché récemment et dans ce cas là on appelle la fonction *player_nb_hp_dec*() qui diminue la vie du joueur . Enfin on sauvegarde ce moment où il a été attaqué. Le reste du temps c'est quand le joueur marche sur un $CELL\_FIRE$.

7 Gestion des Monstres

Tout comme les bombes, on ne peut savoir à l'avance combien de monstres seront sur la carte. Ce nombre varie au cours du temps ainsi qu'en fonctions des niveaux. Ainsi, nous avons opté pour une liste simplement chaînée pour la gestion des monstres. Il existe deux moyens de créer un monstre. Soit on visualise la carte entièrement lors de son chargement, et dès qu'un $CELL\_MONSTER$ est repéré on alloue de la mémoire pour sa naissance. (fonction utilisée $monster\_from\_map()$). Soit une autre fonction peut être appelé, $p\_set\_monster\_ad()$. Celle-ci est appelé lors d'une explosion d'une caisse contenant un monstre. Cette fonction est un *void* et comme elle est appelé dans le fichier *bomb.c* elle doit prendre en entrée un pointeur des monstres du jeu, soit un double pointeur.

Le mouvement des monstres et la gestion des collisions avec l'environnement se fait de la même manière que pour le joueur. L'affichage des monstres se fait directement dans la fonction $game\_meca()$ afin d'avoir un affichage indépendant entre chaque monstres. Les monstres disposent d'une vie à leur naissance, et s'ils se font touchés par une explosion, leur vie passe à zéro. On regarde cela dans la fonction $set\_fire\_xy()$. De plus, $delete\_monster()$ est sans cesse appelé et vérifie en continu la vie des monstres, si celle d'un monstre est à zéro, elle se charge de retourner seulement la liste des monstres vivants de façon récursive.

8 Pause

La gestion de la pause se fait lorsque l'on appuie sur la touche P, on note alors le temps ou la pause a été lancée et on affiche une image indiquant qu'on est en pause. On lance alors une boucle qui ne s'arrête que si l'on appuie à nouveau sur la touche P. A la sortie de cette boucle on modifie le timer des bombes et des monstres pour qu'ils ne prennent pas en compte le temps de pause. On utilise les fonctions $set\_new\_monster\_timer()$ et $set\_new\_bomb\_timer()$ qui prennent en argument le timer du temps initial de la pause.

9 Sauvegarde / Chargement partie

9.1 Sauvegarde

La sauvegarde est ici appelée lorsqu'on appuie sur la touche S. On affiche alors une image pour indiquer que l'on sauvegarde le jeu. La sauvegarde se fait dans deux fichiers prédéfinis, un fichier *Save* qui va enregistrer tout les attributs du joueur avec $fprint$ et un fichier map_s qui va enregistrer l'état de la carte au moment de la sauvegarde.

9.2 Chargement

La chargement est ici appelée lorsqu'on appuie sur la touche L. On affiche alors une image pour indiquer que l'on charge le jeu précédemment sauvegardé. On charge le jeu à partir des deux fichiers de sauvegarde séparément et on modifie la structure de *game* passé en entrée de la fonction. On récupère les données des fichiers en utilisant $fscanf$.