Bomb'eirb

1 Introduction

Le présent document constitue notre rapport sur le projet **Bomb'eirb**, dont le code a été écrit par les professeurs de l'ENSEIRB-MATMECA, Florian MEIGNAN et Maël VIRIOT. Il s'agit ici d'expliciter le fonctionnement du jeu en décrivant brièvement chaque solution technique apportée au différents problèmes qui se posaient.

2 Le jeu

Le contexte global de jeu est géré par le type *struct game* qui contient toutes les variables nécessaires à son fonctionnement : le joueur(*struct player* player*), le niveau courant (*int level*) et le nombre de niveaux total (*int levels*), un tableau de maps *struct map*, et un booléen *onPause* permettant d'indiquer si le jeu est en pause ou non.

2.1 Démarrage

Après avoir initialisé la SDL il nous faut initialiser le contexte de jeu. Pour cela il nous faut tout d'abord tester si une sauvegarde existe. On appelle la fonction *fopen* pour tenter d'ouvrir le fichier en lecture. Si la fonction nous renvoie une résultat non nul, le fichier existe bel et bien et nous pouvons utiliser la fonction *save_load* pour charger le contenue (cette fonction est explicitée plus loin). Si la valeur de retour de *fopen* est *NULL* alors on initialise un nouveau jeu avec les valeurs et les maps par défaut.

2.2 Fin

Le jeu s'arrête en cas de victoire ou de défaite.

Condition de victoire : le joueur se trouve sur la même case qu'une cellule de type *SCENERY_PRINCESS* (sous type de cellule *CELL_SCENERY*).

Condition de victoire : La vie du joueur est tombée à 0. On utilise la fonction *player_get_lives* et on teste si la valeur est strictement positive.

Ces deux tests sont faits à chaque tour de boucle.

2.3 Portes et changements de niveau

Sur les cartes se trouvent des clés, une porte d'entrée et une porte de sortie (voir section sur les maps). Lorsque le joueur se déplace on effectue les tests suivants :

Si le joueur se trouve sur une clé, on la remplace par une cellule vide et la valeur keyState du joueur est mise à 1.
Si le joueur se trouve sur une porte ouverte, on change de niveau.
Si le joueur se trouve sur une porte fermée mais que *player.keyState != 0* alors on ouvre la porte (en changeant le type de cellule), on met le keyState à 0 et on change de niveau.

Le changement se fait à l'aide de la fonction *game_change_level* qui prend en argument le jeu, et une entier "offset". On change la valeur de *game.level* en lui aditionnant la valeur de offset. Si offset vaut -1 cela revient à revenir à la map précédente, s'il vaut 1 on va à la map suivante. De plus lors de tout changement de map, on actualise le point de spawn de la map que l'on quitte grâce à la fonction *map_set_player_spawn* en y mettant la position que le joueur quitte juste avant d'aller sur la porte. Ainsi le joueur revient toujours au bon endroit s'il venait à revenir sur une map précédente.

3 Les maps

L'objet map est représenté map le type *struct map* dont les champs sont : la grille (*unsigned char[]*), les dimentions *w* et *h*, la position de spawn du joueur (*Vector2*), le nombre de monstres, le tableau des monstres (*struct monster[]*), le joueur (*struct player**), et le contexte global de jeu (*struct game**).

Les maps sont soit initialisées par le jeu lors de la création d'une nouvelle partie, soit lues depuis un fichier de sauvegarde (voir section sur les sauvegardes). Lors de l'initisation la position de spawn du joueur est soit lue dans le fichier de sauvegarde (s'il existe) ou sur la grille directement à l'aide de la cellule *CELL_PLAYER_SPAWN*.

La grille est un tableau de $w \times h$ octets, chaque octet indique le type de cellule (*enum cell_type*) Les 4 bits de poids fort indiquent le type principal (*EMPTY, SCENERY...*) et les bits de poids faibles donnent des informations complémentaires comme le sous-type par exemple.

La position de spawn point est utilisée et pour l'apparition dujoueur en début de partie, et pour les changements de niveau.

3.1 Les types de cellule

CELL_PLAYER_SPAWN Valeur : 0x0f. Indique la position d'apparition du joueur sur la carte cette position est sujette à changement en cours de jeu et n'est pas remise à jour sur la grille, elle est uniquement ici pour initialiser une nouvelle carte à partir d'une grille par défaut.

CELL_BOX Valeur : 0x20. Une caisse au sens large, le contenu est indiqué par les bits de poids faible.

CELL_DOOR Valeur : 0x30. Une porte. Une porte peut être une entrée ou une sortie (la différence étant qu'une sortie mène vers le niveau suivant quand une entrée mène vers le niveau qui précède), et peut être ouverte ou fermée. Toutes ces informations sont encodées en poids faible.

CELL_KEY Valeur : 0x40. Indique une clé sur la grille.

CELL_MONSTER Valeur : 0x60. Indique la présence d'un monstre. Les 2 bits de poids faible indiquent la direction pointée par le monstre. La gestion des monstres sera détaillée dans la section les concernant.

CELL_BOMB Valeur : 0x70. Indique la présence d'une bombe mais pas son type.

CELL_EXPLOSION Valeur : 0x80. Indique une case enflammée.

4 Le joueur

4.1 Fonctionnement global

Le joueur est représenté par une *struct* ayant pour champs ses coordonnées *x* et *y*, le nombre de vie et de bombes, la range desdites bombes, un booléen indiquant si le joueur a une clé ou non, un timer de damage (dont nous reparlerons) et enfin un *enum direction* indiquant dans quelle direction le joueur est tourné.

Le joueur est connu au niveau global du jeu via un pointeur *struct player**, mais également au niveau de chaque map via un champ du même type.

4.2 Mouvements et dommages

Dans la boucle principale du jeu existe une fonction qui teste les inputs claviers. Si une touche de déplacement est pressée, le déplacement se fait en deux temps :
-changer la direction du joueur
-déplacer le joueur

Le déplacement du joueur se fait grâce à la fonction *player_move* qui prend en argument le joueur et la map dans laquelle il se trouve.

Tout d'abord on récupère la direction stockée dans le champ direction, et on la convertit en *Vector2* (par souci de simplicité dans la suite). Il nous faut ensuite tester si le mouvement est valide, ce pour quoi nous utilisons une autre fonction : *player_move_aux*. Elle prend en argument la map, le joueur et le vecteur directionnel.

En ajoutant ce vecteur à la position du joueur on obtient la position de la cellule cible, il nous suffit de faire les tests suivants : si la cellule est de type *CELL_EMPTY* on accepte le déplacement (c'est à dire que l'on renvoit 1). Si la cellule est une clé, une princesse, une porte, un bonus, un monstre ou une explosion, on accept le déplacement. Si la cellule est une caisse on appelle la fonction *map_cell_move*. Cette fonction prend en argument la map, la position de la cellule à déplacer et un

vecteur de déplacement. Si le déplacement est valide (i.e. si la cellule destination est vide) la cellule est déplacée dans la grille, et 1 est renvoyé, sinon 0 est renvoyé. Enfin si aucun des cas précédents n'est atteint, on renvoit 0.

Une fois que l'on sait si le déplacement est valide il suffit de mettre à jour les coordonnées du joueur.

C'est également à ce niveau que sont gérés les dommages : lors du déplacement on teste si la cellule de destination est une explosion ou un monstre. Si tel est le cas, on enlève une vie au joueur en décrémentant le champ lives.

On initialise ensuite un timer avec la fonction *SDL_AddTimer*. Elle nous permet de créer un timer géré par la SDL, que l'on stock dans la struct du joueur. Ce timer, une fois le délai défini dépassé (ici 1 seconde) appelle automatiquement une fonction callback qui prend en arguments le délai et un pointeur *void** qui pointe vers n'importe quel objet de notre choix, ici la map.

Ce callback teste si le joueur se trouve toujours sur une cellule "toxique" (explosion ou monstre) et si tel est le cas, lui inflige un dégat supplémentaire avant de retourner 1000 (le délai en millisecondes avant le prochain callback). Ainsi tant que le joueur se trouvera en danger il prendre un dégat chaque seconde.

4.3 Vies

Le jeu teste la valeur de vie du joueur à chaque tour de boucle. Si celle ci est nulle le jeu s'arrête. Bien que nous aurions aimé faire un écran de fin, par faute de temps nous nous sommes contentés d'un print avant de mettre fin à l'exécution.

5 Les monstres

Quel intérêt y aurait-il à jouer s'il aucun danger ne vevait se dresser sur notre chemin ? Ils nous faut bien entendu des monstres !

5.1 Fonctionnement global

L'objet monstre est représenté par le type *struct monster* contenant les champs suivants : la position *x* et *y*, la direction (*enum direction*), la map dans laquelle se trouve le monstre, le contexte de jeu et un timer utilisé pour les déplacements.

Les monstres sont gérés au niveau des maps : chaque map possède un tableau pouvant contenir un maximum de 16 *struct monster**. A l'initialisation du jeu, les positions des monstres sont lues depuis la grille. On appelle la fonction *map_spawn_monster* autant de fois que nécessaire. Cette fonction alloue l'espace pour un monstre, initialise les différents champs et place le monstre dans le tableau.

5.2 Mouvements et attaques

Les monstres peuvent être nombreux, et potentiellement cachés dans des caisses. En outre il ne peuvent être placés que sur des cases vides. Il nous faut donc utiliser un type de cellule spécifique : *CELL_MONSTER* (0x80).

Les déplacements sont aléatoirement générés par un générateur de nombres pseudo-aléatoires implémenté dans le fichier rng.c. Il s'agit d'un algorithme empruntant des fonctions à l'algorithme d'extension de clé de ChaCha20 qui a l'avantage d'être rapide. Si l'algorithme a été bien entendu grandement simplifié, le résultat n'en apparait pas moins aléatoire.

On génère un entier entre 0 et 3, qui nous permet de déterminer la direction à emprunter. La marche à suivre est ensuite d'appeller la fonction *map_cell_move*. Tant que le mouvement n'est pas valide on génère un nouvel entier. On déplace ensuite le monstre, en vérifiant si la case de destination est un joueur. Si tel est le cas on lui inflige un dégât et on initialise son timer de dégâts.

Chaque monstre possède un timer qui permet d'effectuer cet algorithme chaque seconde.

6 Les bombes

6.1 Explication globale

La gestion des bombes se fait de la manière suivante : lorsque le joueur appuie sur espace, une CELL_BOMB est posé au x et y correspondants. Toutes les secondes, le sprite de la bombe décroit jusqu'à l'explosion. L'explosion se fait dans les quatres directions, dépendant de la portée qu'a le joueur.

6.2 bomb.c/bomb.h

L'ajout principal au fonctionnement des bombes se trouve dans les fichiers bomb.c/bomb.h La structure "struct bomb" défini plusieurs caractéristiques importantes des bombes - La position de la bombe sur la carte avec son x et son y.
- Le timer de la bomb, qui va donner dans quel état doit être la bombe.
- D'autres caractéristiques mineures qui affectent peu la manière de fonctionnement des bombes.

Ensuite, un tableau de pointeurs de struct bomb (bomb_tab) est créé afin de stocker les bombes si plusieurs bombes sont actives en même temps.

Lorsque bomb_tab_add est appelé, il va initialiser - avec la fonction bomb_init - le compteur de la bombe, la position de la bombe et la mettre dans le premier slot disponible dans bomb_tab.

bomb_seek va chercher quelle bombe correspond aux x et y donnés.

bomb_erase quant à lui, va supprimer la bombe correspondant aux *x* et *y* donné dans bomb_tab et libérer la mémoire.

Il y a plusieurs fonctions donnant les données de la bombe : - bomb_get_x et bomb_get_y donnent le *x* et le *y* d'une bombe donnée.
- bomb_timer donne le timer de la bombe.
- bomb_display qui affiche l'état de la bombe en fonction du timer.

Enfin, pour les explosions, il y a : - explosion_display, qui va fixer les CELL_EXPLOSION aux bons endroits en prenant compte la portée du joueur et l'environnement autour du joueur.
- explosion_swipe, qui au bout d'une seconde, va remplacer les CELL_EXPLOSION par des CELL_EMPTY

6.3 Fonctionnement détaillé

Dans le fichier game.c, lorsque le joueur appuie sur espace, si le joueur possède au moins une bombe, on ajoute une CELL_BOMB à l'endroit où est le joueur ainsi qu'une bombe dans le bomb_tab.

Ensuite, dans map.c, à chaque fois qu'une CELL_BOMB est rencontrée, la fonction bomb_seek est appelée, afin de savoir quelle bombe correspond aux *x* et *y* de la CELL_BOMB.

Ensuite, une fois que la bonne bombe est trouvée, on vérifie si le timer est inférieur 4, et si tel est le cas, on affiche la bombe correspondante avec bomb_display et on passe à la suite.

Si jamais la bombe a un timer supérieur à 4, alors la fonction explosion_display est appelée, et l'explosion est affichée.

Ensuite, on supprime la bombe avec bomb_erase.

7 Les bonus

7.1 Affichage après explosion des bombes

Dans le fichier bomb.c, une fonction box_explosion est définie. Cette fonction est appelée dans explosion_display, lorsqu'une case de type CELL_BOX est rencontrée.
Plusieurs masques sont appliqués afin de ne garder que le contenu de la boite, pour pouvoir ensuite utiliser map_set_cell_type pour le bonus correspondant.

7.2 Prise des bonus

Une fois les bonus affichés, le joueur peut prendre le bonus qui en est sorti.
Une fonction bonus_effect est définie, elle va appliquer un masque sur les bits de poids faible du

CELL_BONUS et gérer les effets du bonus.

Concrètement, lorsque le joueur avance sur une CELL_BONUS, la fonction player_move_aux va appeler la fonction nommée ci-dessus, et remplacer la CELL_BONUS par une case vide.

Pour le maluce du monstre, suite à de nombreux bugs que nous avions rencontrés, nous avons décidés de laisser ce bonus en commentaire afin d'éviter que le jeu s'arrête.

8 Les sauvegardes

8.1 Enregister une partie

Les sauvegardes sont gérées par le jeu à la fin de la boucle : on ouvre en écriture le fichier save.s contenu dans le dossier saves, et on écrit toutes les données nécessaires en binaire dans des entêtes délimités par des chaînes de caractères indiquant le type d'entête.

Entête "game"
Cet entête contient les données globales de jeu : combien de niveau il y a au total, quel est le niveau courant et l'état du jeu (c'est à dire si le jeu est en pause ou non). Chaque valeur est codée sur un octet.

Entête "player"
Contient toutes les informations sur le joueur : sa position, ses vies et bombes restantes, son "key_state" (un bolléen indiquant si il a une clé) et sa range. Chaque valeur est codée sur un octet.

Entête "bombs"
La première valeur de cet entête est le nombre de bombes posées. En suite pour chaque bombes nous trouvons dans l'odre : sa position, la valeur de son timer encodée en little endian sur 4 octets.

Entête "maps"
Cet entête contient directement les données de chaque map, à la suite (le nombre de maps étant indiqué plus tôt dans le fichier). Pour chaque map on indique sa largeur et sa hauteur, la position du spawn du joueur. Ensuite vient la grille avec toutes les valeurs qui la compose écrites de façon continue.

8.2 Charger une partie

Pour charger une partie, il suffit de lire le chaque entête contenu dans le fichier save.s. On intialise d'abord le jeu avec les valeurs globales, ensuite le joueur, les bombes et enfin les maps en prenant soin de créer les monstres et les bombes au fur et à mesure qu'on les rencontre.