UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

11/02/2022

Bomberman

Projet INFO0402

SAMI DRIOUCHE & WALID AIT-ERRAMI

S4F3B

Table des matières

1. Conception 2
   a. Présentation du projet 2
   b. Description du projet 2
   c. Diagramme 4
2. Développement 4
   a. Concept de la POO 7
3. Documentation Doxygen 9
4. Conclusion 10

1. Conception

a. Présentation du projet

Le projet a pour but de coder le jeu du Bomberman en C++ sans utiliser d'interface graphique (uniquement sur le terminal).

Nous avons un joueur (le Bomberman) sur une Map en 2 dimensions remplie de mur, d'item et de monstre

L'objectif est de tuer tous les monstres à l'aide des bombes pour pouvoir atteindre un objectif et gagner la partie.

Pour compiler et tester ce projet tapez la commande make all pour utiliser le makefile situé dans le dossier Bomberman. Celui-ci génèrera un fichier exécutable prog.exe.

Attention : le projet à été codé sur un système windows et nous utilisons certains appel système ce qui fait que sur mac ou linux il risque de ne pas compiler.

b. Description du projet

Pour le développement nous avons choisi de classer nos fichiers de la façon suivante :

Tous les dossiers ainsi que la classe main sont dans un dossier Bomberman.

- Tous les fichiers d'items seront dans un dossier Item.
- Tous les monstres dans un dossier ennemi,
- La Map et les différentes tuiles dans un dossier Map,
- Le fichier du player ainsi que la Bombe dans le dossier Joueur.

Dans chaque dossier de fichiers .cpp se trouve un dossier entête dans lequel sont rangés tous les .h .

Pour la compilation nous avons choisi d'utiliser un makefile qui cherchera dans chaque dossier les .cpp pour générer des fichiers objets.

Pour les fichiers d'Item nous avons choisi de créer un classe mère Item qui comporte comme attribut d'entier pour la position x et y pour lequel vont hériter tous les items

Nos éléments sont affichés de la façon suivante sur la map :

Le Bomberman et représenté par un emoji vert qui correspond a char(2) .

Les monstres sont affichés en rouge et sont respectivement un M un G et F pour monster, Ghost et bowman.

La bombe quand elle est un @ en jaune.

Les Items sont affichés en bleu et sont un cœur char(3) pour MoreLife, un point d'exclamation ! pour ScaleUp, un 0 pour MoreBomb , un 8 pour PowerUp , et enfin des guillemet fermant » pour SpeedUp .

Les Tiles sont affichées en gris et sont un symbole ▓ char(178) pour les wall et ▒ char(177) pour les sand.

Pour symboliser les murs destructibles et indestructibles.

Pour afficher les couleurs sur le terminal nous avons utilisé ceci :

SetConsoleTextAttribute(GetStdHandle(STD_OUTPUT_HANDLE), 8);

Grâce à cette partie de code :

```
#define SCREEN_WIDTH 90
#define SCREEN_HEIGHT 26
#define WIN_WIDTH 70

HANDLE console = GetStdHandle(STD_OUTPUT_HANDLE);
COORD CursorPosition;

void gotoxy(int x, int y)
{
    CursorPosition.X = x;
    CursorPosition.Y = y;
    SetConsoleCursorPosition(console, CursorPosition);
}
void setcursor(bool visible, DWORD size)
{
    if (size == 0)
        size = 20;

    CONSOLE_CURSOR_INFO lpCursor;
    lpCursor.bVisible = visible;
    lpCursor.dwSize = size;
    SetConsoleCursorInfo(console, &lpCursor);
}
```

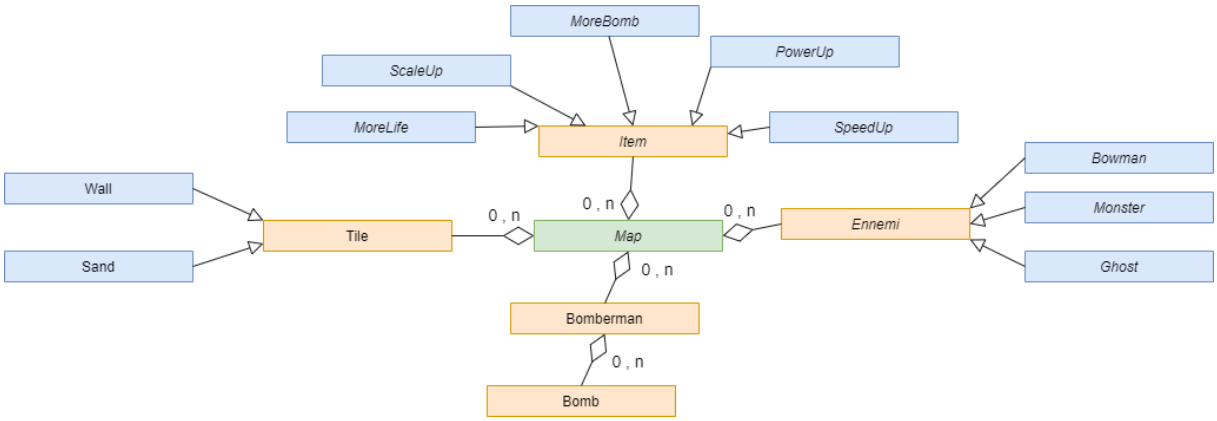
Et à l'inclusion de windows.h :

#include <windows.h>

c. Diagramme

Diagramme de classe

Réalisé a sur draw.io.



2. Développement

Pour le développement du jeu nous avons choisi d'utiliser un tableau d'entiers, car c'est beaucoup plus simple à manipuler. Tous les objets représentés sur la carte auront donc un nombre qui les caractérise comme attribut (comme une sorte d'identifiant) qui se nommera « valeur » et que nous récupèreront à l'aide d'un getter getValeur() .

Gestion de la Map :

Notre tableau est donc initialisé avec une taille stockée dans 2 variables static nbligne et nbcolonne dans le fichier Map.h Ce tableau nommé tab sera donc rempli de 0 à la création (les 0 symbolisent l'herbe ou le vide) grâce à la méthode creerMap() nous ajoutons ensuite à notre Map les murs indestructible grâce à remplirMur() les murs destructibles grâce à remplirSable() ou nous passeront en paramètre un objet de type tuile, ces murs seront ajoutés à la Map de façon arbitraire et séquentielle, c'est-à-dire nous poseront un mur toutes les x lignes et toutes les y colonnes.

```
int const static nbligne = 14;
int const static nbcolonne = 14;
int tab[nbligne][nbcolonne];
```

Ensuite le remplissage des différents Items, que nous avons choisi de créer à l'aide du constructeur par initialisation à l'aide de 2 paramètres leur position x et y générer aléatoirement à chaque partie pour que les items ne soient jamais 2 fois au même endroit la méthode est donc remplirItem() qui attend un item en paramètre et le place à l'endroit où l'item a ces coordonnées.

```
void Map::remplirItem(Item &k)
{
    tab[k.getX()][k.getY()] = k.getValeur();
}
```

Puis le remplissage des monstres qui comme pour les items utilisent le constructeur par initialisation à l'aide de remplirMonstre() .

Pour finir nous ajoutons le player grâce à remplirPlayer(), nous l'ajoutons en dernier pour être sur qu'aucune autre valeur n'écrasera celle du player et éviter que le jeu commence sans joueur.

Une fois le remplissage de la Map terminée ; il faut faire bouger les éléments de celle-ci et faire en sorte que tous nos objets se comportent comme nous le souhaitons.

Gestion des mouvements :

Dans notre classe main nous appelons toutes les fonctions de mouvements et l'affichage dans une boucle do while(), tant qu'il reste des points de vie au player le jeu continue.

Pour que l'affichage de la Map sur le terminal soit propre et qu'il n'affiche pas plusieurs Map à la suite nous appelons une commande système dans la boucle pour nettoyer le terminal à chaque mouvement system("cls").

Lors de l'affichage chaque int de notre tableau correspondant à un élément de notre jeu sera converti à l'aide d'une fonction convertir en chaîne de caractères qui permettra d'avoir l'affichage que nous souhaitons.

Cette fonction convertir() se situe dans la classe Map et est appelée dans la fonction afficher de la Map.

Lorsque nous sommes dans cette boucle faire tant que nous appelons donc les fonctions mouvementMonstre() prenant en paramètre un objet de type Ennemi et la fonction mouvementPlayer() prenant en paramètre un objet de type Bomberman .

```
w.mouvementPlayer(b, boom, mor, spe, mbo, sca, pow);
for (size_t i = 0; i < w.tabM.size(); i++)
{
    w.mouvementMonstre(*w.tabM[i], b);
}
```

Les monstres bougent donc avant le player ce qui fait que pour la gestion des dégâts qu'infligent les monstres on regarde si le player va bouger sur une case ou un monstre est présent. Si oui, le player ne bouge pas et le monstre inflige des dégâts.

Tous les monstres essaient de se placer d'abord sur la ligne du joueur puis dans sa colonne. Sauf s'ils rencontrent un mur qu'ils ne peuvent passer, dans ce cas pour plus de fun nous avons fait en sorte que les monstres ont un mouvement aléatoire. Tous les monstres se déplacent de la même façon à l'exception du ghost qui lui va traverser les murs (ceux qui le rend redoutable). En revanche, il ne traverse pas la bombe. Le ghost

possède un attribut int tmp dans lequel il stockera la valeur de la case dans laquelle il voudra aller pour dès qu'il quitte celle-ci elle revienne comme avant.

Tous les monstres du jeu sont stockés dans un vecteur afin d'automatiser le code et ajouter autant de monstre que l'on veut.

```
Monster mon1(7, 7);
w.tabM.push_back(&mon1);
```

A chaque création d'un objet monstre on l'ajoute au tableau et à l'appel de chaque méthode de monstre on parcourt ce tableau pour que tous les monstres soient traités.

Lorsque le jeu se termine on vide le tableau avec la fonction clear() pour que les monstres des parties précédentes ne restent pas.

Gestion des Bombes :

Pour la gestion des Bombes nous avons choisi de créer dans la classe Bomberman un tableau dynamique de bombe :

Std ::vector<Bomb> tabB

Que l'on remplît à 3 bomb lors de la construction du Bomberman. Ceci nous permet à chaque utilisation des bombes de pouvoir stocker leur position ainsi d'ajouter à chaque tour de jeu 1 au compteur de bombe à la bonne bombe du tableau. Si elle est posée sur la Map. Le player peut donc poser plusieurs bombes à la suite elles n'exploseront pas toutes en même temps.

Lorsque que notre joueur prend l'item MoreBomb on ajoute une case à notre tableau.

Grâce à la fonction push_back() .

Lorsque l'item Scaleup est pris la portée de la bombe augmente d'une case. On réitère simplement l'explosion à une case de plus de chaque côté grâce à l'attribut portée de la bombe.

L'explosion de la Bombe :

A chaque tour on appelle la fonction explosionBombe() qui parcourt notre tableau de bombe et fait en sorte que si la bombe est posée ; on augmente la valeur de l'attribut compteur de la bombe. Lorsque ce compteur atteint 3 la bombe explose et lorsqu'il atteint 5 on efface les cases de l'explosion à vide. Si l'explosion a lieu sur une case occupée par un mur indestructible il ne sa passe rien, et lorsque l'explosion a lieu sur la case d'un monstre elle tue le monstre.

Enfin si la case est un joueur, le joueur reçoit simplement les dégâts.

La mort d'un monstre :

A chaque tour nous appelons une fonction tuerMonstre() qui regarde si aucun de nos monstre contenu dans notre tableau de monstre est mort à l'aide de l'attribut Vivant de nos monstres qui est un boolean mis à true lors de la construction d'un monstre.

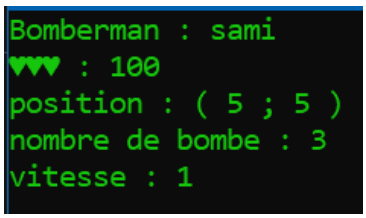
Si un monstre a son nombre de vie inferieure à 0 ; ce boolean est mis à false ce qui fait que le monstre ne pourra plus bouger car lors de l'exécution de la fonction mouvementMonstre on vérifie d'abord si le monstre est en vie.

Fin de partie :

Cette fonction parcourt notre tableau de monstres et vérifie à chaque tour si tous les monstres du tableau sont morts ou pas. Si tous les monstres sont morts elle retourne un boolean a true qui va arrêter la boucle do while du main pour l'exécution du jeu et donc nous afficher un message de victoire.

Affichage des statistiques du Player :

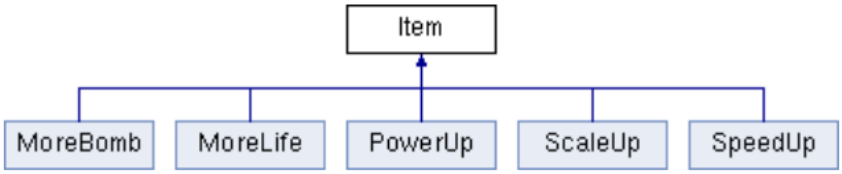
Nous affichons les statistiques de notre joueur juste au-dessus de la map grâce a la surcharge de l'opérateur cout <<, avec le nom de notre joueur, sa position x y, son nombre de Bombe et de point de vie (affiché avec des petits cœurs).



a. Concept de la POO

Héritage :

Nous avons donc une classe mère pour les 5 items tels que : Morelife, morebomb, speedup, scaleup et powerup.

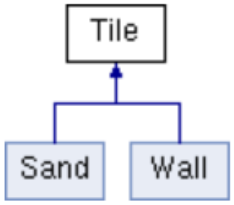


Polymorphisme :

Item possède 2 méthodes virtual effetBomb() et effetPlayer(), les classes filles vont en fonction de leurs capacités redéfinir une de ces méthodes .

Héritage :

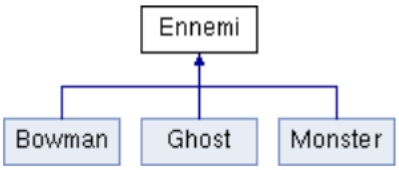
Une classe mère tile avec deux classes filles Sand et Wall pour les murs destructibles et indestructible.



Tile n'as pas de méthode virtual.

Héritage :

Une classe mère Ennemi pour nos 3 monstres.



Polymorphisme :

Ennemi possède 2 méthodes virtual attaquerPlayer() et recevoirDegat() pour que le monstre fasse des dégâts différents en fonction du type de monstres.

La classe bombe agrège la classe Bomberman de sorte que le bomberman possède des bombes.

La Map possède tous les types d'objets dont nos trois classes mère ; Tile, Item et Ennemi puis notre classe Bomberman.

3. Documentation Doxygen

Pour la documentation nous avons choisi d'utiliser doxygen. Toutes les méthodes de notre projet sont commentées de tel sorte à avoir l'auteur ainsi que les paramètres de la méthode.

```
/**
 * @brief Construct a new Bomberman object
 * @author sami DRIOUCHE
 * @param nom
 * @param vie
 * @param x
 * @param y
 * @param vitesse
 * @param nbBomb
 * @param boom
 */
Bomberman(std::string nom, int vie, int x, int y, int vitesse, int nbBomb, Bomb boom);
```

Puis on génère la documention doxygen dans le dossier Doxygen.

Et nous avons donc les liens pour la documentation en ligne avec les fichiers html.

Bomberman Class Reference

classe du bomberman More...

```
#include <Bomberman.h>
```

Public Member Functions

Bomberman (std::string nom, int vie, int x, int y, int vitesse, int nbBomb, **Bomb** boom)
Construct a new **Bomberman** object. More...

Bomberman ()
Construct a new **Bomberman** object. More...

Enfin nous avons généré grâce au dossier latex la documentation en pdf que vous pouvez retrouver dans le fichier refman.pdf.

4. Conclusion

Malheureusement par manque de temps nous n'avons pas pu implémenter plus de fonctionnalités et le jeu n'est pas complet. Il manque par exemple la flèche du Bowman et l'objectif final à atteindre.

La vitesse du Player n'a pas non plus été géré. Cependant le jeu est tout de même jouable et nous en sommes assez fiers. Il nous a permis d'en apprendre davantage sur la programmation orienté objet et le langage du C++ de façon tout à fait ludique et intéressante.