Projet : Bomberman

{victor.allombert,luidnel.maignan,pascal.vanier}@u-pec.fr

Attention ! Le sujet peut être mis à jour dans les prochaines semaines, pensez à le reconsulter de temps à autre.

Il faut lire le sujet intégralement avant de commencer le projet [1].

1 Consignes

Le projet devra être codé intégralement en C, tout ajout non demandé ne sera pris en compte dans la note que si les fonctionnalités de base sont implémentées. Le projet est à réaliser **seul ou à deux**.

Les ouvertures/lectures/écritures de fichiers devront être faites avec les appels systèmes, vous avez cependant le droit d'utiliser les fonctions de `<string.h>` ainsi que les fonctions `sprintf`, `snprintf` et `perror`. Il devra comporter un `Makefile` et il devra compiler avec les options `-std=gnu99 -Wall -Werror`.

Vous joindrez à votre projet une documentation expliquant son fonctionnement, ses limites, les problèmes que vous avez rencontrés et les solutions que vous avez trouvées et les contributions des divers membres du groupe.

La notation tiendra compte de divers paramètres : la qualité du code (lisibilité, modularité, commentaires), la qualité des structures de données, la qualité algorithmique, le fonctionnement du programme (phase réussie, présence ou absence de bugs), ainsi que la soutenance.

Il est obligatoire d'utiliser le gestionnaire de versions `git`, un compte sera créé pour chacun d'entre vous sur le serveur https://git-etudiants.lacl.fr dès que les groupes auront été formés, avec ces comptes seront créés des dépôts pour chacun des groupes.

Attention : Tout projet qui ne sera pas rendu sur `git` ne sera pas noté. Vous devez vous en servir *régulièrement* [2].

2 Dates

Les groupes sont à former avant le **8 avril 2015**, sur éprel. Après cette date les personnes n'étant pas dans un groupe en auront un tiré aléatoirement.

La version finale du projet doit être sur `git` avant le **20 mai 2015**. La soutenance aura lieu le **19 mai 2015**.

3 Le principe

Le but de ce projet sera de faire un jeu : un clone de *bomberman* [3]

Le principe est que le joueur manipule un personnage se déplaçant sur une carte en deux dimensions faite de murs plus ou moins destructibles (cela va du mur destructible en une/-deux/... explosion, explosions, jusqu'au mur indestructible)

1. Oui, je sais, cela paraît évident...
2. Cela signifie en particulier que les excuses comme "j'ai perdu mes fichiers par accident", "nous avons rajouté des bugs au dernier moment" ne seront pas acceptées.
3. Il est possible d'y jouer en ligne si vous ne connaissez pas, votre moteur de recherche préféré est votre ami.

Un second personnage est manipulé par un autre joueur, le but pour chacun des joueurs est d'éliminer le joueur adverse en posant des bombes. Chaque personnage ne peut poser qu'une bombe à la fois et doit attendre que celle-ci aie explosé pour pouvoir en poser une autre. Un joueur est éliminé quand il est pris dans le souffle d'une bombe. Une bombe met un certain temps à exploser. Les bombes seront représentées par un `@` rouge sur la carte.

Le jeu permettra de jouer sur différentes cartes : les cartes pourront être chargées à partir d'un fichier. Le jeu devra gérer plusieurs ensembles de niveaux que l'on appellera `mods`. La syntaxe de ces fichiers est donnée en annexe A.

Attention : le programme ne devra être constitué que d'un seul thread.

4 Phase 1 : un film dans le terminal

La première étape sera de vous familiariser avec `termios`, allez faire un tour sur la page `man termios`, afin d'être capables d'afficher une "image" ascii dans votre terminal. On souhaite également qu'à cette étape les entrées au clavier ne soient pas affichées (le terminal ne devra donc pas rester en mode canonique).

La première étape du projet sera donc d'être capables d'afficher une carte à l'écran, sans que les personnages ne bougent. Vous devez donc à cette étape déjà être capables de lire les fichiers contenant les cartes, mais vous pouvez ignorer une très grande partie du contenu de ces fichiers. Vous devrez pour cela créer au moins un mod par défaut et créer des cartes dans celui-ci [4].

5 Phase 2 : faire bouger les personnages et gérer les collisions

Phase 2.1 : Il faut maintenant faire bouger les personnages. Il faudra pour cela surveiller l'entrée standard, et régulièrement mettre à jour l'écran dans le temps. On pourra pour cela utiliser le `timeout` de `poll` ou `select`. Pour déplacer chaque personnage, on utilisera respectivement les flèches et les touches ZQSD.

Phase 2.2 : Il faut maintenant gérer les collisions : lorsque le personnage contrôlé par le joueur entre en collision avec un mur ou un ennemi, il faut que celui-ci ne puisse pas le traverser.

Note : Une fois cette phase terminée, on a normalement un jeu d'exploration de labyrinthe avec deux personnages.

6 Phase 3 : gestion des bombes

Phase 3.1 : Il faut que les joueurs puissent poser des bombes, les bombes doivent exploser après un certain nombre de secondes (par exemple 2,5s) et avoir un rayonnement de quelques cases (3 par exemple). Il faut également que tout joueur pris dans le souffle d'une bombe perde une vie (par défaut les joueurs peuvent avoir 3 vies par exemple).

Note : À ce stade le jeu est réellement fonctionnel.

Phase 3.2 : Il faut gérer des *powerups*, c'est à dire des objets permettant d'améliorer un joueur : sa vitesse maximale de déplacement, la portée de ses bombes ainsi que le nombre de bombes qu'il peut poser simultanément.

4. quelques cartes suffiront

7 Phase 4 : un serveur

À l'aide soit de tubes, soit de socket UNIX ou réseau (à vous de voir ce qui est le plus facile !), vous écrirez un serveur auquel plusieurs clients pourront se connecter pour jouer ensemble sur une même carte.

Attention : Avant de vous attaquer à la phase 4, assurez vous que votre code soit propre, modulaire, lisible et bien commenté (ce qui ne signifie pas qu'il faille commenter chaque ligne). Avoir un jeu qui fonctionne parfaitement jusqu'à la phase 3 permettra déjà d'avoir une bonne note.

A Syntaxe des fichiers

A.1 Niveaux

Chaque fichier de niveau est constitué de la manière suivante :

- Une ligne qui contient deux nombres h et l séparées d'un espace.
- Les h lignes qui suivent sont de longueur l et chaque "case" contient soit un chiffre soit un espace
- Puis il y a à nouveau h lignes de longueur l, celles-ci contiendront les *powerups* : un `+` pour une augmentation de vitesse une `*` pour une augmentation de la portée des bombes et un `@` pour une augmentation du nombre de bombes que peut poser un personnage à la fois.

```
10 44
00000000000000000000000000000000000000000000
0            00000        111      22      0
0   1111111  00000 111111 111 1111 11 1111 0
0 11 1111111 00000 118991 111 1111 11 111  0
0 11 1111111 00000 100011 111 1111 1  1    0
0 11 1     1 11111 1   11 111 1111 1 11 1  0
0 8  1 111 1 10000 1   11 111 1111 1 11 11 0
0 1 11 111 1 00000 111111 111 1111   11 11 0
0   11     1 11111                         0
00000000000000000000000000000000000000000000

                           +
      *                                 @

       @
               @           +    *
                                *       @

               @

```

A.2 Mods

Un "mod" (un ensemble de niveaux) sera constitué d'un répertoire dont le nom sera le nom du mod et qui contiendra un sous répertoires. Dans le répertoire du mod figurera un fichier `deroulement` le informations sur le jeu : la première ligne devra contenir le nom du jeu et les lignes suivantes les identifiants des niveaux par ordre d'apparition. Le répertoire du mod devra contenir un sous-répertoire `niveaux`. Chaque niveau sera stocké dans un fichier dont le nom sera un nombre **sans extension**, le fichier `deroulement` contiendra donc dans la première ligne le titre du mod, puis sur chaque ligne le numéro correspondant au niveau à charger ensuite.

Ceci doit permettre de facilement installer de nouveaux "mods" dans votre jeu. Simplement en ajoutant un répertoire correspondant à un mod, on doit pouvoir dans le menu d'entrée du jeu sélectionner ce mod si on le souhaite.

B Manipuler le terminal

B.1 Taille du terminal

Votre jeu doit fonctionner pour une taille fixée de terminal que vous pourrez choisir. Ne choisissez pas une taille trop grande, il faut pouvoir jouer sur les ordinateurs des salles informatiques. Un bon moyen de s'assurer que cela puisse fonctionner dans les salles informatiques est de lancer votre jeu avec la commande suivante :

```
xterm -geometry 80x24 -e /chemin/vers/jeu
```

vous pouvez remplacer 80 et 24 par les valeurs que vous souhaitez. Il est également possible d'accèder à la hauteur/largeur d'un terminal en accédant aux variables d'environnement `LINES` et `COLUMNS`. Il serait de bon aloi de vérifier avant de lancer une partie sur un mod si le terminal permet d'afficher ce mod.

B.2 Mode du terminal

Le mode par défaut dans lequel est le terminal est le mode canonique, c'est à dire qu'un read débloque uniquement quand l'utilisateur tape sur la touche entrée et tout ce qui est entré au clavier est affiché à l'écran. Il faut, afin que le terminal n'affiche pas les touches utilisées et permette de lire les touches sans que l'on utilise entrée, passer le terminal en mode `raw`, on peut pour cela faire appel à la fonction `tcgetattr` afin de récupérer les attributs du terminal, modifier ces attributs avec `cfmakeraw`, puis les attribuer au terminal avec `tcsetattr`. Le `man termios` ainsi que de chacune de ces fonctions est votre ami.

B.3 De la couleur dans le terminal

On peut utiliser de la couleur dans un terminal, afin de faire cela, on peut écrire avec `write` le résultat de `sprintf(buffer, "\x1b[91m%s", acolorier);`. Après cela, toutes les autres chaînes qui seront écrites seront de cette couleur, il faut donc remettre la couleur initiale si l'on ne veut pas cela.

Vous pouvez consulter la page http://jafrog.com/2013/11/23/colors-in-terminal.html pour plus d'informations sur les couleurs et autres attributs du texte and le terminal (lien en anglais).

B.4 Se déplacer dans le terminal

Vous pouvez déplacer le curseur à la position (x, y) en écrivant le résultat de

```
sprintf(buffer, "\x1b[%d;%dH", x, y);
```

avec `write`.