DRAI Alexis
EL CHEIKH Walid

17/12/2021
L2 - Prog. Av.

Projet 2048

Seules les principales fonctions responsables du fonctionnement du jeux sont présentées dans ce rapport, mais le reste des fonctions sont expliquées dans la documentation des fichiers "game.h" et "toolbox.h", où on prend soin des aspects les plus mondains du programme. On ne montrera pas dans ce rapport toutes les validations de *mallocs*, libérations de mémoire, et autres sécurités, mais elles sont là dans le code. Le versionnage du code est visible sur GitHub, à votre convenance.

Un README donne des détails sur la compilation et l'execution. Deux programmes sont livrés : le programme principal (`cd DM2048` puis `make` puis `bin/prog`) et le prototype sur console (`cd DM2048` puis `make console` puis `bin/console`).

2048 est un jeu dont l'objectif est de déplacer des cases et de les fusionner pour créer une case avec le nombre 2048.

Nous allons donc exposer des programmes qui mènent à ces résultats : d'abord la version console, puis la version SDL 1.2.

Console :

Pour la première partie du projet, on a codé 2048 comme un jeu sur console.

```
=_=_=_=_=_= 2048 _=_=_=_=_=_=
nouvelle partie:. n
charger partie:.. c
quitter:......... q
>n
score: 0
#### 0002 #### ####
#### #### #### ####
#### #### #### 0002
#### #### #### ####
(h)aut | (d)roite | (g)auche | (b)as || (q)uitter | (s)auvegarder | (c)harger
```

La structure "Game" contient une grille (matrice de 4 par 4 *unsigned short ints*), le score (*unsigned short int :* avec l'intervalle [0, 65535], le joueur pourrait presque atteindre 2048 trois fois : c'est certainement suffisant), et le nombre de cases vides (*unsigned short int)*. En conservant les variables liées au jeu dans cette *struct*, on facilite la manipulation de ces données.

On commence par afficher un menu : "n" fait appel à la fonction "InitGame" qui initialise le jeu et fait apparaître deux '2' dans des emplacements aléatoires. "l" fait appel à la fonction "LoadGame", qui charge une partie déjà sauvegardée. "q" libère la mémoire réservée par Game et coupe l'exécution du code.

Quand une partie est lancée, on itère dans la boucle *while* dans "main.c" tant que le jeu est en cours. On libère la matrice et coupe l'exécution du programme quand on quitte la boucle ; soit par fin de la partie, soit par choix du joueur. C'est une fonctionnalité commune pour un jeu-vidéo. Après le premier coup, on

génère une case de valeur '2' ou '4' dans une place aléatoire, affiche la grille, et demande au joueur de choisir une direction. Arbitrairement, on a choisi 75% de chance de générer un '2', et 25% pour un '4'.

```
while (isOn) {
    if (wasMove) {
        rdVal = ((rand() % 2) + 1) * 2;
        if (rdVal == 4) { rdVal = ((rand() % 2) + 1) * 2; }
        SpawnTiles(g, rdVal, 1);
    } DisplayGame(g);
    if (g->free_tiles == 0) { CheckLose(g); }
    PromptMove(&isOn, &wasMove, g);
}
```

Pour vérifier si la partie est perdue, on évalue d'abord "free_tiles" qui est tenue à jour le long du programme, quand la grille change. On évite donc d'évaluer chaque paire de cases possible à chaque itération pour détecter un *game over*.

"SpawnTiles" évite de faire apparaître des cases quand l'action précédente n'a pas changé la grille. On fait cela grâce au pseudo-booléen "wasMove", aperçu plus haut dans le *main*. C'était important de s'assurer qu'on écrase pas une case existante, et que la fonction continue d'essayer même après un échec.

```
int spawned = 0;
while (spawned < n) {
    int col = rand() % 4, row = rand() % 4;
    if (g->board[row][col] == 0) {
        g->board[row][col] = val;
        g->free_tiles -= 1; spawned++;
[...]
```

On affiche sur la console le score et la grille de jeu et la liste des commandes que le joueur peut taper. Les commandes " h, d, g, b " font appel à la fonction "PromptMove" qui provoque un mouvement. La commande "q" demande si le joueur veut sauvegarder la partie, sauvegarde par défaut sauf s'il tape "n", puis coupe l'exécution.

"s" sauvegarde la partie en cours et "c" en charge une, que ce soit dans le menu ou pendant une partie. "SaveGame", ci-dessous, et "LoadGame", pas illustrée ici, fonctionnent très similairement.

```
FILE *fp = fopen("data/save.txt", "w+");
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 4; j++) {
        fprintf(fp, "%d ", g->board[i][j]);
    }
    fprintf(fp, "\n");
}
fprintf(fp, "%d ", g->score);
fprintf(fp, "%d ", g->free_tiles);
fclose(fp);
```

"PromptMove" fait appel à "Move" qui effectue le déplacement vers la droite et les fusions possibles (à l'aide des fonctions "Slide" et "Fuse").

Et pour les autres directions, on appelle la fonction "Rotate" qui tourne la matrice vers la droite, on effectue les mouvements, puis on tourne la matrice pour qu'on revienne à la forme initiale.

```
for (int a = 0; a < n; a++) {
   int **aux = CopyBoard(board);
   for (int i = 0; i < 4; i++) {
       for (int j = 0; j < 4; j++) { board[j][3 - i] = aux[i][j]; }
   } FreeBoard(aux);
}
```

"Fuse" est la fonction responsable d'effectuer des fusions quand c'est possible en vérifiant si deux cases successives ont la même valeur, et fait appel a "YouWin" si une des cases obtient 2048 comme valeur.

```
for (int i = 0; i < 4; i++) {
    for (int j = 2; j >= 0; j--) {
        if (g->board[i][j] != 0 && g->board[i][j + 1] == g->board[i][j]) {
            unsigned short int newVal = g->board[i][j + 1] << 1;
            if (newVal == 2048) { YouWin(g); }
            g->score += newVal;
            g->board[i][j + 1] = newVal;
            g->board[i][j] = 0;
            g->free_tiles += 1;
            *hasFused = 1;
[...]
```

"Slide" est responsable du déplacement des cases s'il existe de la place libre du côté choisi par le joueur.

```
for (int a = 0; a < 3; a++) {
    for (int i = 0; i < 4; i++) {
        for (int j = 2; j >= 0; j--) {
            if (g->board[i][j + 1] == 0 && g->board[i][j] != 0) {
                g->board[i][j + 1] = g->board[i][j];
                g->board[i][j] = 0; *hasMoved = 1;
[...]
```

"CheckLose" vérifie s'il existe une fusion possible, et appelle "YouLose" si on n'en trouve aucune.

```
for (int i = 0; i < 3; i++) {
   for (int j = 0; j < 3; j++) {
       if (g->board[i][j] == g->board[i][j + 1] ||
           g->board[i][j] == g->board[i + 1][j]) { return; }
   }
}
YouLose(g);
```

"YouLose" et "YouWin" informent le joueur de la fin de la partie, affichent le score final, invoquent "FreeGame", et coupent l'exécution du programme.

SDL :

Pour cette partie, on a ajouté les bibliothèques SDL 1.2 et TTF. Cette nouvelle version implique principalement de l'affichage, mais la logique du programme a aussi changé par endroits.

Notamment, la *struct* "Game" contient maintenant les variables "msecs" (pour enregistrer le temps passé sur une partie, "status" (qui entre en jeu dans la nouvelle vision de la fin du jeu), "isOn" et

"wasMove" (qu'on utilisait jusqu'alors depuis le *main*), et "screen", "screen_clr", "fnt", et "fnt_clr", qui contiennent des constantes du jeu liées à l'affichage en SDL. Certaines de ces variables sont maintenant prises en compte par les fonctions "SaveGame" et "LoadGame".

La fin du jeu est gérée différemment : on veut pouvoir afficher la grille actualisé depuis l'écran de *game over*, et on veut pouvoir retourner au menu ou quitter le programme volontairement. Pour cette première contrainte, on retarde la fin du jeu jusqu'à après l'appel à "DisplayGame", en modifiant un code "status" si la partie est gagnée ou perdue, et on évalue ce "status" à la fin de chaque itération du *while* ; au lieu de lancer "YouWin" ou "YouLose" depuis "Fuse" ou "CheckLose" pour sortir prématurément. De plus, on utilise la fonction unique "EndGame" pour gérer cette situation, puisque la différence entre gagner et perdre tient dans une chaîne de caractères. Pour la seconde contrainte, "EndGame" laisse le joueur quitter le programme, ou retourne 1 pour aller au (m)enu grâce à un *goto* associé à un *label*.

```
if (g->status != 0) { if (EndGame(g)) { goto start_game; } }//dans main()
```

```
[...]//dans EndGame()
switch (evt.key.keysym.sym) {
   case 'm':
       [...]
       return 1;
[...]
```

La fonction "PromptMove" a changé pour recevoir de nouveaux types d'inputs, et avec "SDL_PollEvent". On y a ajouté l'option de lancer une nouvelle partie. Plus intéressant, on a mis en place la logique du chronomètre dans le *main* : "SDL_GetTicks" nous permet de récupérer le nombre de millisecondes depuis l'initialisation de SDL. "SDL_PollEvent" n'attends pas un input, mais les traite à mesure qu'ils arrivent. On mesure le temps écoulé entre le début et la fin du de chaque itération, et si c'était moins de 38ms (notre intervalle "ITV"), on invoque "SDL_Delay" pour cesser l'exécution jusqu'à atteindre 38ms. Ce choix vient du fait que 38ms par *frame* équivaut à 26 FPS. Or, en dessous de 25, l'affichage peut paraître saccadé pour l'œil humain. Ainsi, on économise de l'usage du CPU, et le rendu visuel reste décent.

```
Uint32 prv_time = SDL_GetTicks();
[...]
Uint32 spent = SDL_GetTicks() - prv_time;
if (spent < ITV) { SDL_Delay(ITV - spent); g->msecs += ITV; } else { g->msecs += spent; }
```

Un problème est survenu quand on est passé de *Wait* à *Poll*, car soudain les inputs par souris (curseur bougeant par-dessus le jeu, etc.) ralentissaient beaucoup le programme. On a donc neutralisé ces inputs par des appels à "SDL_EventState" dès l'entrée du *main*.

L'écran et la police apparaissent pour la première fois dans la fonction "Menu", où on les initialise ; puis ils sont stockés dans notre "Game" pour la suite du programme.

```
SDL_Surface *screen_ = SDL_SetVideoMode(WID, HEI, BPP, SDL_HWSURFACE);
TTF_Font *fnt_ = TTF_OpenFont("/usr/share/fonts/truetype/freefont/FreeMonoBold.ttf", 24);
```

Finalement, c'est la fonction "DisplayGame" qui a le plus changé avec SDL, comme il se doit. Notons tout d'abord qu'on a préparé des substituts pour les fonctions SDL et TTF usuelles, "MyFlip", "MyBlit", etc., visibles dans "toolbox.c". Ces nouvelles fonctions acceptent les mêmes arguments et appellent les

fonctions originales, mais gèrent aussi la validation des données. D'abord, "DisplayGame" libère l'écran actuel, puis le remplit avec un fond.

```
SDL_FreeSurface(g->screen);
MyFillRect(g->screen, NULL, g->screen_clr);
```

On affiche chaque ligne de texte et chaque case en spécifiant une surface et une position à *blit*ter.

```
SDL_Rect pos;
pos.x = PAD, pos.y = PAD;
char *prompt_str = "(s)auver | (c)harger | (n)ouveau";
SDL_Surface *prompt_dis = MyRenderText(g->fnt, prompt_str, g->fnt_clr);
MyBlit(prompt_dis, NULL, g->screen, &pos);
[...]
```

Pour chaque case, on utilise sa valeur pour déterminer une couleur, selon un schéma codé en dur. encore une fois, on spécifie ensuite une surface et une position à *blit*ter sur l'écran. Pour les cases non nulles, on affiche leurs valeurs en leur centre. Dans la boucle intérieure, on décale notre position vers la droite de la largeur d'une case; dans la boucle extérieure, on remet notre position contre le bord de gauche et on descend de la hauteur d'une case.

```
for (int i = 0; i < 4; i++) {
   for (int j = 0; j < 4; j++) {
       unsigned short int val_int = g->board[i][j];
       switch (val_int) {...}//couleurs
       tile = SDL_CreateRGBSurface(SDL_HWSURFACE, TIL, TIL, BPP, 0, 0, 0, 0);
       MyFillRect(tile, NULL, clr); MyBlit(tile, NULL, g->screen, &pos);
       if (val_int != 0) {...}//valeurs
       pos.x += TIL + PAD;
   }
   pos.x = PAD, pos.y += TIL + PAD;
}
```

Enfin, on affiche le nouvel écran.

```
MyFlip(g->screen);
```