Projet IF3:
Ikswall

Table des matières

Introduction

1. Choix du Projet

« Les projets sont l'occasion de vous confronter à la création de vrais programmes ».

J'ai donc du tout d'abord choisir le type de programme que je voulais réaliser. Comme ma machine tourne sous Linux, il fallait que je fasse un programme qui fonctionne sur plusieurs systèmes d'exploitation (GNU/Linux, Solaris, Mac OS, MS Windows) et plusieurs architectures différentes (x86, x86_64, powerpc, sparc, arm (certaines consoles de jeu portables)). Il fallait donc soit utiliser uniquement les fonctions standard du C (celles de la librairie standard), soit utiliser des librairies plus complexes, mais portées sur les différents systèmes d'exploitation et les différentes architectures voulues.

Je pouvais donc réaliser plusieurs types de programmes. La console n'étant pas vraiment un outil très utilisable sous MS Windows, et une application tournant en tache de fond n'étant pas forcement quelque chose de très « Démonstratif », j'ai décidé de faire une application plutôt graphique.

J'ai au début hésité entre la création d'un serveur ftp (basique), d'un client ftp, d'un logiciel de stéganographie (uniquement pour les images en bmp) , ou un jeu.

Ayant déjà fait plusieurs jeux, j'avais une banque d'images a ma disposition, j'ai donc décidé de me lancer dans la création d'un jeu. La librairie semblant la plus adaptée à ce genre de programme étant la librairie SDL (http://www.libsdl.org/) qui a été utilisée pour des jeux tels que « Unreal Tournament ».

2. Principe / Règles de base

J'ai eu plusieurs idées de jeux, dont un RPG (j'ai fait l'éditeur de carte et la gestion des déplacements du personnage sur la carte, avant de laisser tomber car le système de combats prévu sur papier était trop long a programmer pour être finit dans les temps), un Asteroids'like (que je finirais plus tard, les fonctions prévues étant trop nombreuses pour ce type de projet.

J'ai finalement décidé de plancher sur un projet plus simple, permettant de jouer a plusieurs.

Le Principe est assez simple, une balle par joueurs, des bonus, des murs a éviter, et un score. Les bonus étant facilement configurables, et les murs n'apparaissant que lors de la récupération de certains bonus. Pour plus de difficulté, les murs bougent, soit verticalement, soit horizontalement.

Il existe aussi plusieurs modes de jeux, de nouveaux modes pouvant être facilement développés.

3. Outils Utilisés

Éditeurs de texte:

Kate - http://kate.kde.org/

Emacs - http://www.gnu.org/software/emacs/emacs.html

Console :

Konsole - http://konsole.kde.org/

Bash - http://www.gnu.org/software/bash/bash.html

Compilation/Débuguage:

Gcc - http://gcc.gnu.org/

Gdb - http://www.gnu.org/software/gdb/gdb.html

ValGrind - http://valgrind.org/

Graphisme :

The Gimp - http://www.gimp.org/

(cette liste ne comprend que les outils principaux)

Portage Windows :

Dev Cpp - http://www.bloodshed.net/devcpp.html

Analyse du sujet

1. Structures de données

a) Les Options

Les options doivent être accessible dans tout le jeu. Dans le cas de ikswall, elles doivent comporter: le nombre de joueurs, le mode de jeu, la taille des murs, la vitesse de base des balles et des murs, la résolution, le mode vidéo (plein écran, fenêtre), et les touches de joueurs.

La plupart des paramètres sont des entiers non signés, la vitesse est un double car la plupart des opérations portant sur la vitesse sont faites avec des double, cela évite donc de caster dans tout les sens.

Une résolution d'écran est composée d'une largeur et d'une hauteur le tout en pixel, il était donc assez logique de faire une structure contenant la largeur et la hauteur pour chaque résolutions.

Les touches sont des entiers non signés avec SDL (voir SDL_keysym.h). Chaque joueur ayant 4 touches, les touches sont stockées sous la forme d'un tableau à deux dimensions, la première étant le joueur, la deuxième la touche.

Les modes de jeux sont constitués d'un nom et de flags (drapeaux) permettant de spécifier ce que fait ce mode de jeu. C'est donc une structure constituée d'un pointeur vers une chaîne de caractères et d'un entier non signé.

Les modes vidéos n'étant que deux (plein écran et fenêtre), il n'était pas très utile de faire une structure contenant les flags de ce mode, et le nom. J'ai donc uniquement créé un tableau de pointeurs vers des chaînes de caractères contenant le nom des modes.

Une variable globale option de type struct s_options (voir main.h) est donc créée en début de programme, elle contient les options courantes et qui peuvent être lues et modifiées par n'importe quelle fonction du programme.

Cela permet d'éviter d'avoir a passer dans la quasi totalité des fonctions les options en paramètres.

Cela permet aussi de faire des macros de ce genre :

```
/* Résolution Horizontale */
# define SCREEN_WIDTH vresols[options.vresol].w
/* Résolution Verticale */
# define SCREEN_HEIGHT vresols[options.vresol].h
```

b) Le Menu

Le menu est assez simple, il est constitué de boutons, si on clique sur l'un de ces boutons, une action est associée. Ces boutons sont aussi utilisables au clavier. J'aurai pu utiliser une librairie toute faite pour ce genre de chose (comme picogui http://picogui.org/), mais j'ai préféré recréer un système simple.

Chaque bouton a donc une position, et un état (appuyé, relâché), une surface (l'image de fond du bouton), et un texte.

Ici, les seules variables utiles à stocker pour un bouton sont la position (pour savoir si on clique dessus ou pas) et l'état. C'est une une structure comprenant un entier non signé qui indique l'état, et un SDL_Rect qui indique la taille et la position. De plus il est inutile de stocker pour chaque bouton le fond dans une surface séparée, car la plupart des boutons ont un fond commun. L'action associée pourrait être stockée comme un pointeur vers une fonction, mais dans le cas présent cela n'a pas vraiment d'utilité.

Pour écrire sur le menu, il faut une police (ou plusieurs de taille différentes), et une image de fond, quit sont donc stockées dans la structure menu, qui elle même contient un tableau de structures de bouton.

Une structure de ce type permet donc de stocker facilement l'état et la position des boutons. Les images et polices étant stockées dans s_menu. surf contient différentes images collées les une aux autres.

c) Le Jeu/L'interface

L'interface du jeu est assez simple: un fond fixe, et les informations sur les joueurs au premier plan.

Il faut donc une image de fond (différente en fonction de la résolution) et les polices pour écrire le texte.

Pour des raisons pratiques, toutes les images du jeu en lui même (fond, balles, murs, bonus ...) sont stockées dans la structure de l'interface, qui est globale. Cela facilite par exemple le processus d'affichage des scores à la fin.

d) Les Balles

Une Balle représentant un joueur, il y a plusieurs types de données à stocker:

- Les données concernant le joueur, comme le score, le nombre de vies, les touches appuyées, les bonus déjà récupérés, la vitesse maximum, le temps restant en mode invincible, etc ...
- Les données concernant la balle en elle même, comme la position, les différences de déplacements en x et y (permet de gérer l'accélération et de ne pas tout le temps de déplacement à la vitesse maximum).

La structure s_ball (voir ball.h) permet de stocker toutes ces données.

Les balles sont stockées dans un tableau de structures s_ball, ce tableau peut être utilisé et modifié par n'importe quelle fonction.

e) Les Murs

Les murs apparaissent quand on récupère un bonus, leur nombre n'est donc pas fixe comme le nombre de balle.

Il faut donc :

- Stocker le nombre de murs courant
- Avoir une structure de donnée dynamique permettant de stocker les informations sur les murs (position, déplacements, etc ...).

Une liste chaînée aurait put être utilisée, mais il semble plus indiqué d'utiliser un tableau dont on modifie la taille avec realloc quand il n'y a plus de place.

Par defaut, le tableau de structure s_wall (walls.h) (qui permet de stocker la position et les déplacements) de dimension 100, dès qu'il est plein, 100 nouvelles « cases » sont allouées.

Le nombre de murs et ce tableaux de murs sont stockés dans une structure de type s_walls (voir game.h), ce qui permet de limiter le nombre de paramètres à passer aux fonctions qui gèrent les murs.

f) Les Bonus

Les bonus, comme les murs, n'ont pas de nombre définis, mais en plus, peuvent disparaître.

Ici une liste chaînée est particulièrement utile. Cela permet d'ajouter et de supprimer facilement des bonus.

Des pointeurs vers le premier et le dernier bonus sont stockés dans une structure de type s_bonus. Ces pointeurs pointent vers des structures de type s_1bonus qui contiennent le type, la position, et des pointeurs vers le bonus précédent et le bonus suivant.

Pour que les bonus disponibles soient facilement modifiables, un tableau de structure s_bonusconf permet de stocker la configuration des bonus.

2. Fonctions Principales

```
int main();
```
Fonction de base du programme, appelle Init(), puis StartMenu() si Init() à réussis

```
int Init();
```
Initialise SDL.

```
void Free();
```
Libère proprement la mémoire utilisée lors de l'initialisation.

3. Fonctions Utiles

Les fonctions utiles un peu partout sont dans le fichier util.c. Ces fonctions sont principalement des fonctions d'abstraction qui permettent d'éviter de répéter du code.

```
void limitFps();
```
Permet de limiter le nombre d'images par secondes, la macro FPS permet de régler ce nombre.

```
SDL_Surface *LoadSDLImage(const char *filename, unsigned int alpha);
```
Permet de charger une image, et d'afficher une erreur si l'image n'existe pas ou n'est pas lisible. Renvoi directement un pointeur vers l'image chargée.

```
TTF_Font *LoadSDLFont(const char *filename, unsigned int size);
```
Permet de charger une police, et d'afficher une erreur si la police n'existe pas ou n'est pas lisible. Renvoi un pointeur vers la police chargée.

```
unsigned int myrand(unsigned int a, unsigned int b);
```
Permet de générer un nombre pseudo-aléatoire situé entre a et b. Initialise le générateur lors du premier appel.

```
void PrintText(SDL_Surface *surf, char *text, SDL_Rect *dst, TTF_Font
```

```
*font, SDL_Color color);
```
Affiche un texte text a la position définie par dst, sur la surface surf. Le texte utilisera la police font, et la couleur color. dst est automatiquement modifié et contient après affichage du texte, la largeur et la hauteur du texte affiché.

```
void DrawGraph(SDL_Surface *surf, SDL_Rect dst,  unsigned int value,
unsigned int valuemax);
```
Affiche un « graphique » du type barre de vie ... pourait être utile, mais pas utilisé dans ikswall.

```
unsigned int get_flags(unsigned int flags);
```
Ajuste automatiquement les flags en fonction du système et du mode vidéo.

```
unsigned int SetVideoMode( unsigned int x, unsigned int y, unsigned int
bits_per_pixel, unsigned int flags );
```
Permet de changer le mode vidéo, et théoriquement d'éviter que le programme plante si il est impossible d'utiliser le mode vidéo spécifié par l'utilisateur.

4. Fonctions Générales

Le jeu est séparé en éléments distincts (interface, balles, murs, bonus ..).

Chacun de ces éléments a besoin d'être:
- Initialisé
- Libéré
- Démaré
- Affiché
- Géré (déplacements ...)
- Associé à des événements (clavier, souris, etc ..)

Chaque élément est donc dans un fichier séparé constitué la plupart du temps des fonctions :
-InitElement()
-FreeElement()
-StartElement()
-HandleElement()
-EventElement()

Avec bien sûr quelques autres fonctions permettant de ne pas trop mélanger du code ne faisant pas la même chose.

5. Séparation en différents fichiers

Comme le jeu est divisé en plusieurs éléments distincts, il était logique de créer un fichier source différent pour chacun de ces éléments.

Problèmes rencontrés

Général – Compatibilité Windows

Le Portage sous Windows a demandé quelques modifications au programmes, même si SDL est censé être portable.

Pour que le programme se lance et fonctionne correctement sur n'importe quel machine Windows il faut modifier la variable d'environnement SDL_VIDEODRIVER et mettre sa valeur à « windib », sinon SDL tente d'utiliser le driver directx qui à tendance à très mal fonctionner sur certains PC (ceux de la fac en particulier).

Il est aussi nécessaire d'inclure la libraire « windows.h » pour accélérer la création de la fenêtre.

Pour que ces modifications ne s'appliquent que pour la version compilée pour Windows, il suffit d'utiliser la macro WIN32 censée être définie pour ce système. Par exemple, pour la modification de la variable d'environnement il suffit de faire :

```
#ifdef WIN32
putenv("SDL_VIDEODRIVER=windib");
#endif
```

Général – Changement de Résolution / Mode Vidéo.

Changer le mode Vidéo (Plein écran/Fenêtre), n'est pas très compliqué, et utiliser la fonction « SetVideoMode » qui adapte automatiquement les flags et vérifie que le changement est possible facilement encore plus la chose. Le Plus dur a donc juste été de créer cette fonction.

Changer la résolution est par contre un peu plus compliqué. En les macros utilisées pour récupérer la taille de la fenêtre utilisent la résolution choisie dans les options. De plus le fond a une taille fixe et SDL ne permet pas de redimensionner les images.

Le premier problème, qui causait des problèmes d'affichage lors du changement de résolution sans Appuyer sur le bouton validé à été réglé en utilisant « screen->w » et « screen->h » pour les boutons du menu qui étaient placé en fonction de la taille de la fenêtre, et en forçant le changement de résolution lors du lancement du jeu, même sans appuyer sur Valider (uniquement si la résolution à été changée bien entendu).

Pour le deuxième il y avait deux solutions, la première étant de créer une fonction permettant de redimensionner une image à la volée, la deuxième de créer à l'avance des images pour les résolution possibles. Créer une fonction permettant de redimensionner étant long et nécessitant pas mal de modifications, j'ai préférer créer à l'avance les images. De plus le programme « convert » permet de faire ce genre de chose très facilement.

Jeu – Gestion des Touches.

La gestion des touches est compliquée pour la possibilité de jouer à 6 sur le même clavier. Il n'est pas possible d'activer une répétition des touches trop rapides, sinon l'utilisation des touches pour l'interface (déplacement au clavier sur le menu, touche Esc) est rendue impossible. La gestion des événement liées aux touches est largement facilitée par le stockage des touches des joueurs dans la structure utilisée par les options.

Pour réglé ce problème, le plus simple à donc été de stocker les touches appuyées, et ce pour

chaque joueur. Pour stoquer ces touches, le moins couteux en mémoire est d'utiliser un entier non signé, dont on modifie le bit « n » lorsque la touche correspondante est appuyée. Par exemple, si on appuie sur « haut » le bit 0 de « ball[0]->keydowns » et mis à 1, lorsqu'on relache cette touche, il est remis à 0.

Pour savoir si on doit déplacer la balle ou pas, le programme n'a plus qu'a regarder si la touche correspondant à « haut » est appuyée.

Jeu – Perte d'une Vie.

Un problème c'est posé lors de la perte d'une vie (collision avec un mur). En effet, une fois qu'il y'avait collision, le mur continuait a traverser la balle, qui continuait à perdre des vie. La solution appliquée est largement utilisée dans un grand nombre de jeux : rendre la balle invincible quelque secondes lors de la perte d'une vie.

Jeu – Apparition des Bonus.

Au début, un bonus apparaissait lors de la récupération de n'importe quel bonus. Le problème est qu'il suffisait d'éviter de prendre de la nourriture (qui crée des murs) et de prendre n'importe quel autre bonus pour gagner un grand nombre de points. Un flag spécifique à donc été rajouté au bonus nourriture qui permet de ne créer de nouveaux bonus que lors de la récupération de nourriture

Jeu – Collisions entre deux balles.

Pour le mode « Arcade » il fallait implémenter un système de collision entre les balles. Créer un moteur physique réaliste n'étant pas vraiment la meilleure solution, j'ai tenté de trouver une méthode plus simple mais moins réaliste.

Les déplacements horizontaux et verticaux des balles sont stockés dans la structure balle. Quand deux balles rentrent en collision (quand la distance entre le centre des deux balles et plus petit que le diamètre d'une balle), les déplacements de l'une sont ajoutées deux fois à ceux de l'autre.

Cette méthode présentait un problème quand les balles étaient contre un bord, elles finissaient par être l'une dans l'autre et par ne plus pouvoir bouger. Quand une balle frappe une balle contre un bord, ses déplacements sont inversés, et la balle est repoussée, ce qui lui évite ainsi de rentrer a l'intérieur de la balle.

Guide d'utilisation

Compilation

UNIX

Sous n'importe quel UNIX, sous GNU/Linux en particulier, il suffit d'installer les librairies SDL a l'aide du gestionnaire de paquet de votre distribution (sous debian, `apt-get install libsdl1.2-dev`).

Le script bash compile.sh contient les commandes pour la compilation, le lancer comme ceci suffira :

```
sh compile.sh
```

Pour compiler à la main :

```
gcc src/*.c -c
gcc *.o -lSDLmain -lSDL -lSDL_mixer -lSDL_image -lSDL_ttf -o ikswall
```

Windows

Pour compiler Ikswall sous MS Windows, il est conseillé d'utiliser gcc et Dev C++. Il est nécessaire d'avoir les librairies SDL installées. Si ce n'est pas le cas, il suffit d'installer les packages (regroupés içi: http://xf.iksaif.net/sdl/windows/) à partir de Dev C++.

Il suffit ensuite d'ouvrir le fichier Ikswall.dev, et de cliquer le le bouton compiler.

Utilisation

UNIX

Lancez le script « launch_ikswall.sh ». Il est possible de lancer ikswall directement avec le binaire correspondant, mais cela peu poser des problèmes, notamment lorsqu'on double-clique sur le binaire via Konqueror.

Windows

Double-cliquez sur « Ikswall.exe ».

Règles

Balles

Chaque joueur possède une balle qu'il peut déplacer à l'aide du clavier. Chaque joueur a un certain nombre de vies (chaque fois que sa balle touche un mur, il perd une vie).

Si le mode « Arcade » est activé, les balles peuvent se pousser.

Bonus

Il existe plusieurs types de Bonus, dans l'ordre:

Pépite: 1 point, un seul a la fois, crée un mur, peut créer un bonus

Or: 5 points

Vie : 10 points, donne une vie

Vitesse: 5 points, augmente de 1 la vitesse

Vitesse++: 10 points, augmente la vitesse de 2

Bouclier I: 5 points, rend invincible 10 secondes

Bouclier II: 10 points, rend invincible 20 secondes

Mithril: 15 points

voilà comment sont configurés les bonus dans le code (voir définitions des structures).

```
{"Pépite",1,100,BONUS_IS_UNIQUE|BONUS_MAKE_WALL|BONUS_CAN_MAKE_BONUS},
{"Or",5,10,0},
{"Vie",10,5,BONUS_GIVE_LIFE},
{"Vitesse+",5,4,BONUS_SPEED_UP_1},
{"Vitesse++",10,2,BONUS_SPEED_UP_2},
{"Bouclier I",5,3,BONUS_INVIC_10},
{"Bouclier II",10,1,BONUS_INVIC_20},
{"Mithril",15,4}
```

Murs

Un mur est soit vertical, soit horizontal. Un mur se déplace à la vitesse définie dans les options (comme les balles), et a une taille définie (par les options).

Un mur rebondit lorsqu'il est trop près d'un bord, ou lorsque le mode arcade est activé, et qu'il touche une balle.

Score

Le score est modifié par les bonus, le joueur ayant le plus de points à la fin à gagné.

La formule pour calculer le nombre de points que rapporte un bonus est la suivante:

```
ceil(points_du_bonus*log(vitesse+1)*log(taille_des_murs+1))
```

Conclusion

Ce Projet m'a permis d'apprendre à utiliser SDL pour faire un jeu marchant sur plusieurs architectures différentes. J'ai aussi appris quand et comment utiliser des listes chaînées et doublement chaînées.

D'un point de vue algorithme, le code reste extrêmement simple, mais d'autres fonctions, comme une IA permettant de jouer contre l'ordinateur seront sûrement ajoutées.

La gestion du réseau est sûr papier devrait aussi être assez simple à implémenter (le plus dur étant le menu pour se connecter à une partie réseau !).