ENSIIE

Initiation à la Programmation Impérative, Projet 2016

Morpion 3D

Vianney Touchard

Pour le 3 janvier 2017

Table des matières

1 Sujet

Le projet de programmation du premier semestre à l'ENSIIE consiste à implémenter un morpion 3D (plutôt similaire à un puissance 4 d'ailleurs).

Le jeu se joue au tour par tour, en empilant des pions dans des piles sur un plateau carré de taille n*n choisie au début de partie. Le but du jeu est d'être le premier joueur à aligner n pièces.

Deux variantes sont jouables : la variante vue du dessus où le but est de former des lignes avec les dessus des piles, et la variante 3D où les pièces doivent être alignées dans l'espace.

2 Outils de développement

Par habitude, j'ai choisi de développer sous Windows avec l'IDE Code : :Blocks et l'implémentation de gcc pour Windows MinGW. J'ai également eu recours à l'outil de gestion de versions Git et à l'hébergement gratuit github. Il est ainsi facile de tester le code de façon transparente sous Linux et Windows en utilisant les mêmes commandes (`git`, `make`, `gcc`).

3 Choix techniques

3.1 Langue

La langue utilisée dans le code est l'anglais (pour avoir une homogénéité avec l'utilisation des fonctions standard qui sont nommées en anglais et en règle plus générale avec le monde de la programmation qui utilise plus communément cette langue).

Les commentaires sont en revanche écrits en Français (exception faite des `@requires`, `@assigns`, etc. imposés)

3.2 Modules

Dans le projet, les différentes déclarations et implémentations sont séparées dans des fichiers différents, qu'un makefile permet de compiler en une commande `make`.

Le module `pile` avait été implémenté lors d'un TP durant le semestre et a été réutilisé tel quel.

3.3 La structure s_jeu

Pour regrouper les variables de partie tout en permettant leur utilisation facilement au travers de plusieurs fonctions, j'ai décidé d'utiliser une structure `s_jeu` qui ne sera instanciée qu'une seule fois pour une partie. Les fonctions associées requièrent systématiquement un pointeur vers une instance de cette structure.

3.4 Les préfixes

Par souci de lisibilité, ces fonctions sont préfixées (par le préfixe `j_`). L'implémentation de la structure de pile suit également cette logique.

3.5 Les booléens

En C, le type `bool` est différemment implémenté (ou pas) selon les standards (cf Wikipedia). Ainsi, la solution choisie a été d'utiliser les entier pour cela (sachant que les tests `if(a!=0)` et `if(a)` sont équivalents)

4 Solutions apportées

4.1 Le modulo euclidien

Pour pouvoir bouger le curseur entre les différentes parties de l'écran sans avoir à vérifier qu'il reste dans les bornes `[0,n]`, j'ai voulu utiliser un modulo. Il s'avère que le modulo standard (`%`) peut donner des résultats négatifs. Il a fallu implémenter sous forme de macro un modulo qui renvoie une valeur utilisable.

4.2 Le module console

La gestion de la partie et l'affichage sont gérés indépendamment. Pour simplifier ce dernier, j'ai eu recours à un tableau de symboles ASCII qui est modifié à volonté et ensuite affiché ligne par ligne en une seule fois. Cela permet de séparer les différentes fonctions de dessin et d'appeler un affichage à n'importe quel moment. Le module `console` apporte ces fonctions.

Il apporte également les fonctions de saisie sécurisées pour demander à l'utilisateur un caractère ou un nombre. Celles-ci sont basées sur la description de l'utilisation de scanf avec des expressions régulières donnée ici.

4.3 Les flags

On a recourt dans le programme à des flags pour marquer les options. Cette technique permet de coder plusieurs booléens dans un seul entier en utilisant des opérations bit à bit.

En pratique, cela permet de par exemple combiner les options aisément (par exemple la variante 3D avec l'option séisme est codée par un `int options = EARTHQUAKE | TRIDIM`). Cela a aussi permis de rajouter des informations dans le tableau `events` sans avoir à en re-créer.

4.4 Les tests de victoire

Les tests de victoire sont très similaires entre les variantes.

Ils reposent sur une recherche dans toutes les directions possibles des pièces alignées à partir de certaines cases. D'abord à partir de toutes les cases, on peut ensuite optimiser.

En effet, une ligne n'apparaît que si un événement y concourt. Ainsi les fonctions peuvent restreindre leur recherche aux seules piles effondrées (pour la vue de dessus) ou jouées (pour les 2 variantes). On marque durant le tour les événements en question dans des flags et on ne recherche des lignes seulement sur ces cases.

4.5 L'aléatoire

L'autre problème rencontré était que lors de l'activation de l'option séisme, le premier tour voyait une pile s'écrouler à chaque test. La probabilité donnée par la formule ne correspondait pas à une fréquence si haute.

Il s'est avéré que l'appel à `rand()`, même après qu'une graine ait été fournie par `srand(time(NULL))`. Après affichage des séquences de `rand()`, il s'est avéré que les premiers résultats étaient similaires. L'ajout de quelques appels non utilisés au début du programme semble avoir réglé le problème.

4.6 Le tableau events

L'affichage des éboulements aurait pu se faire lors de leur génération mais cette solution n'était pas satisfaisante parce que l'affichage devait se poursuivre ensuite durant le tour du joueur suivant.

Pour garder en mémoire quelles piles se sont éboulées, j'ai créé un tableau `events` dans la structure `s_jeu` qui marquait les piles touchées.

Par la suite, comme les fonctions de détection demandaient de connaître où un pion avait été joué, j'ai rajouté cette information dans le tableau grâce à un flag supplémentaire. Un dernier flag a été rajouté pour déboguer `j_checkUp`, qui codait les piles marquées commes alignées (le bogue était finalement un oubli du test de vacuité des piles observées)

5 Manuel d'utilisation

L'utilisation du jeu ne devrait pas poser de problème car l'interface est remarquablement bien conçue.

En revanche, quelques informations seront utiles aux developpeurs : Les sources sont à compiler avec `make puissance4` ou bien `make run` (qui lance le programme immédiatement) grâce au `makefile` fourni (les options `clean` et `mrproper` sont également disponibles). Ou bien directement avec `gcc` et l'option `-lm`.

En outre, il est possible d'augmenter la taille de l'affichage (nécessaire si un message lors du choix de la taille du plateau l'indique). Pour cela, il suffit de modifier les `#define WIN_H et WIN_W` (respectivement hauteur et largeur de l'affichage -en nombre de caractères-) dans console.h et de recompiler. Il est également possible de modifier le `#define FALLOUT` dans `jeu.h` pour changer la durée d'affichage des gravats lors d'un séisme.

6 Limites du programme

Bien que fonctionnel, le code du programme pourrait être mieux structuré selon les exigences. Il gagnerait à profiter de standards plus récents du C ou à passer à du C++.

Le jeu est assez peu amusant à jouer, surtout avec l'option séisme. Pour améliorer ce point, on aurait pu mieux calibrer les probabilités de chute des piles qui s'effondrent trop souvent.

L'interface est très limitée, et cela pourrait être amélioré avec la mise en place de couleurs, de touches directionnelles qui s'activent sans avoir à appuyer sur entrer, un mode plein écran et des caractères plus exotiques (ASCII étendu ou unicode).

De telles améliorations requièrent cependant l'utilisation de bibliothèques additionnelles comme ncurse, et rendent le programme plus dépendant (en particulier de l'OS).

De nombreuses améliorations seraient également imaginable : parties en réseau, affichage 3D, etc.