1 Introduction

Le sokoban est un jeu vidéo japonais de réflexion créé par Hiroyuki Imabayashi en 1981. Dans ce jeu, vous incarnez un personnage dans un entrepot et vous devez déplacer des caisses pour qu'elles soient toutes sur un point de stockage.

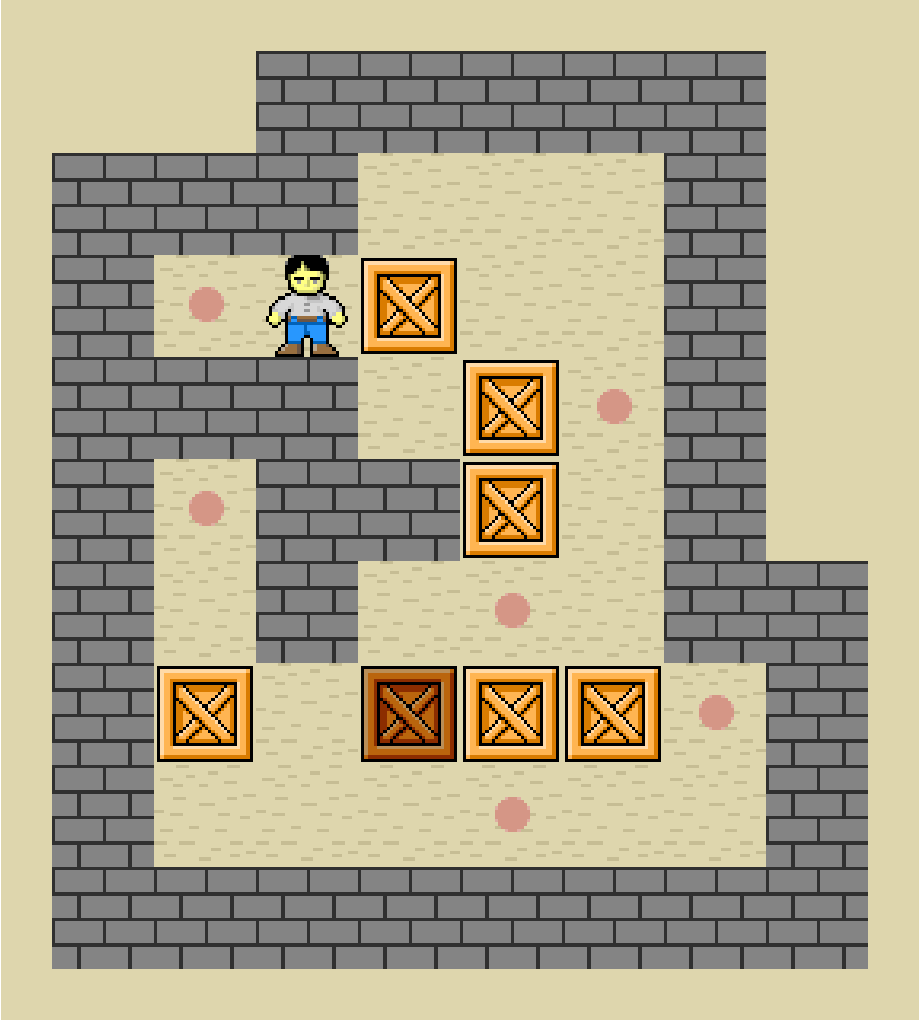

FIGURE 1 – Exemple de sokoban (sur wikipedia).

2 Règles du jeu

Afin de mieux comprendre les règles du jeu, vous trouverez un exemple de partie ici. Vous pouvez également jouer au jeu, par exemple sur cette page.

Ce jeu se joue sur une grille rectangulaire. Dans l'exemple précédent, la grille est de largeur 8 et de hauteur 9. Chaque case de la grille représente un mur ou du sol. Certaines cases contiennent des caisses et certaines cases sont des points de stockage.

Le joueur est toujours sur une case de la grille. Il peut se déplacer horizontalement ou verticalement vers une case vide (il ne peut pas entrer dans un mur ni dans une caisse). Le joueur peut également se déplacer vers une caisse, ce qui a pour effet de pousser la caisse. Les caisses ne peuvent être poussées que dans des cases vides (pas dans d'autres caisses ni dans un mur). On ne peut pas tirer les caisses. Un niveau est fini lorsque toutes les caisses sont sur un point de stockage.

3 Modélisation du jeu

3.1 Les cases du plateau de jeu

Chaque case de la grille peut être représentée par une combinaison de quatre facteurs. Pour stocker une case, je vous propose d'utiliser un seul nombre entier. On va pour cela jouer un peu avec la représentation du nombre. Soient b_{mur}, b_{joueur}, b_{caisse} et $b_{stockage}$ 4 bits définis par :

$$b_{mur} = \begin{cases} 1 & \text{si la case est un mur} \\ 0 & \text{sinon} \end{cases}$$

$$b_{joueur} = \begin{cases} 1 & \text{si le joueur est sur la case} \\ 0 & \text{sinon} \end{cases}$$

$$b_{caisse} = \begin{cases} 1 & \text{si une caisse est sur la case} \\ 0 & \text{sinon} \end{cases}$$

$$b_{stockage} = \begin{cases} 1 & \text{si la case est un lieu de stockage pour caisse} \\ 0 & \text{sinon} \end{cases}$$

On peut maintenant accoler ces bits pour former un nombre entier c de 4 bits. Il faut pour cela choisir dans quel ordre les placer. L'ordre choisi n'a pas d'importance mais il faut en fixer un et toujours garder le même pour être cohérent. Fixons $c = b_{stockage}b_{mur}b_{joueur}b_{caisse}$. Cette technique de stockage de l'information s'appelle des champs de bits (*bit fields* ou *bit flags* en anglais). En C, elle est le plus souvent utilisée avec un type énuméré où les valeurs n'utilisent qu'un bit :

```
typedef enum
{
        CAISSE = 0x01,
        JOUEUR = 0x02,
```

```
        MUR = 0x04,
        STOCKAGE = 0x08
}  Case;
```

Les différentes valeurs (ou *flags*) sont ensuite combinées grâce à des **opérateurs bit à bit**. Dans notre cas, les opérateurs intéressants sont :

— l'opérateur **et bit à bit** s'écrit **&** et prend deux paramètres entiers.
A & B renvoie un nombre dont les bits à 1 sont uniquement ceux étant à 1 à la fois dans A et B.

— l'opérateur **ou bit à bit** s'écrit | et prend deux paramètres entiers.
A | B renvoie un nombre dont les bits à 1 sont ceux étant à 1 dans A ou dans B.

— l'opérateur **non bit à bit** s'écrit ∼ et prend un seul paramètre.
∼A renvoie un nombre dont les bits sont inversés par rapport à A.

Voici quelques exemples de manipulation de tels nombres :

```c
// Case 1 : un joueur sur une salle de stockage
int c1 = JOUEUR | STOCKAGE;
// Case 2 : une case vide
int c2 = 0;
// Case 3 : une caisse sur une case vide
int c3 = CAISSE;
// Le joueur quitte la case c1
c1 = c1 & ~JOUEUR;
// Le joueur arrive sur la case c2
c2 = c2 | JOUEUR;
// Une caisse quitte la case c3
c3 = c3 & ~CAISSE;
// Une caisse arrive sur la case c1
c1 = c1 | CAISSE;
// Quelques tests
if ((c2 & CAISSE) != 0)
    printf("Il y a une caisse sur c2\n");
if ((c1 & (CAISSE | MUR)) != 0)
    printf("c1 est une caisse ou un mur\n");
if ((c2 & (CAISSE | STOCKAGE)) == (CAISSE | STOCKAGE))
    printf("c2 est une caisse sur un lieu de stockage\n");
```

3.2 Le plateau de jeu

Le plateau de jeu peut être représenté par une grille de largeur w et de hauteur h. On peut stocker cette grille dans un tableau d'entiers de taille $w \times h$, comme on le voit sur la figure 2. Veuillez noter qu'en informatique, la case de coordonnée (0,0) est souvent choisie en haut à gauche et non en bas à gauche comme vous avez pu le faire en cours de mathématique. Cela est dû à plusieurs raisons, la principale étant que les écrans historiques affichaient leurs caractères en balayant l'écran de gauche en droite, en commençant par le haut de l'écran. Ainsi, $x \in [0, w[$ croît lorsqu'on va vers la droite et $y \in [0, h[$ croît lorsqu'on va vers le bas.

Pour transformer des coordonnées (x, y) d'un tableau à deux dimensions en coordonnées (i) d'un tableau équivalent à une dimension, on peut appliquer $i = y \times w + x$. Pour effectuer la transformation inverse, on peut appliquer $y = i/w$ et $x \equiv i \pmod{w}$.

4	7	2	3
4	5	3	2
8	1	6	9

(a) Un tableau à deux dimensions de taille $w = 3$, $h = 4$

4	7	2	3	4	5	3	2	8	1	6	9

(b) Le même tableau stocké dans une seule dimension de taille $w \times h = 3 \times 4 = 12$

FIGURE 2 – La représentation du plateau de jeu

4 Abstraction jeu / interface

Lorsqu'on développe un projet informatique, il est souvent intéressant de découper le code en plusieurs modules liés plus ou moins fortement. Dans notre cas, il est important de séparer la gestion du jeu de la gestion de l'interface utilisateur. Il vous sera demandé dans ce projet de développer plusieurs interfaces utilisateurs. Vous devez obligatoirement faire l'interface basique en mode console. Vous pouvez ensuite faire une interface avancée en mode console et une interface en mode graphique, chacune d'entre elles vous octroyant des points supplémentaires au projet.

Le déroulement d'une partie se fait en plusieurs tours. Quelle que soit l'interface utilisée, à chaque tour, le joueur effectue une et une seule action parmi :
- Se déplacer vers la gauche,
- Se déplacer vers la droite,
- Se déplacer vers le haut,
- Se déplacer vers le bas,
- Réinitialiser le niveau,
- Quitter le jeu.

Chaque type d'interface doit au final faire deux choses : afficher le plateau de jeu et obtenir un choix d'action de l'utilisateur.

4.1 Interface basique en mode console

Cette interface est du même type que celle que l'on utilisait jusqu'à présent. L'affichage se fait via *printf* et les saisies via *scanf* et *fgets* (ou des fonctions plus poussées de la bibliothèque standard).

Pour afficher le plateau de jeu, on peut simplement se servir de *printf*. Voici comment les différentes cases doivent être affichées dans ce type d'interface :

— ' ' pour les cases vides,
— 'X' pour un mur,
— '.' pour un lieu de stockage,
— '@' pour le joueur qui n'est pas sur un lieu de stockage,
— '#' pour le joueur qui est sur un lieu de stockage,
— 'c' pour une caisse qui n'est pas sur un lieu de stockage,
— 'C' pour une caisse qui est sur un lieu de stockage.

Voici par exemple ce que donnerait l'affichage de la figure 1 (page 1 avec cette norme :

```
  XXXXX
XXX   X
X.@c  X
XXX c.X
X.XXc X
X X . XX
Xc Ccc.X
X   .  X
XXXXXXXX
```

Pour obtenir un choix d'action de l'utilisateur, vous pouvez par exemple lui demander ce qu'il souhaite faire grâce à *printf* en lui détaillant quelles sont les actions possibles. Vous pouvez ensuite lire ce qu'il a saisi (*via scanf* ou *fgets* par exemple) et renvoyer l'action correspondante. Vous devez gérer correctement les saisies invalides : il faut dire à l'utilisateur qu'il s'est trompé et lui redemander la saisie dans ce cas. Un exemple de cette interface est illustré en figure 3 (page 6).

4.2 Interface avancée en mode console

Cette interface est également en mode console mais beaucoup plus poussée puisqu'elle utilise la bibliothèque **ncurses**. Cette bibliothèque est disponible dans les paquets de toute distribution Linux usuelle (le paquet est *ncurses* sous Arch, *libncurses-dev* sous Ubuntu...). Faire fonctionner cette bibliothèque sous

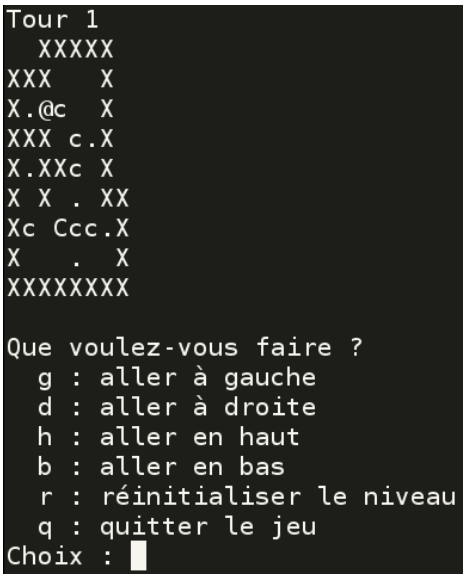

FIGURE 3 – L'interface console

Windows semble être un parcours du combattant, mais vous pouvez vous y essayer si vous vous sentez l'âme d'un guerrier.

Vous pouvez trouver un tutoriel sur ncurses ici. Je n'ai pas trouvé de tutoriel correct sur cette bibliothèque en français, vous pouvez essayer d'en trouver un si vous n'êtes pas à l'aise en anglais.

Pour obtenir un choix d'action de l'utilisateur, je pense qu'utiliser les flèches directionnelles du clavier pour se déplacer, la touche 'r' pour réinitialiser le niveau et la touche 'q' pour quitter le programme est le plus intuitif.

Pour afficher le niveau vous pouvez utiliser deux caractères par case. Je vous propose de mettre un fond jaune pale pour les murs, un fond noir pour le vide et un fond rouge pour les lieux de stockage. Vous pouvez afficher le joueur par " : :" et les caisses par "[]", les deux avec une couleur d'écriture blanche. Un exemple d'un tel affichage se trouve sur la figure 4 (page 7). Si vous souhaitez faire cette interface, vous pouvez vous rendre en section 6.5 (page 10).

4.3 Interface graphique

Cette interface utilise une fenêtre graphique pour afficher le plateau de jeu et utilise les mêmes commandes que l'interface ncurses. La bibliothèque SDL est probablement la manière la plus simple d'avoir ce type de résultat. Des tutoriels de la SDL se trouvent ici, là où encore là. Vous devez vous servir de la version

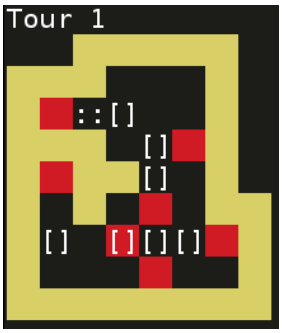

FIGURE 4 – L'interface ncurses

1.2 de la bibliothèque, pas de la version 2 (ou si vous le faites, assurez-vous que votre code compile et marche avec la version 1.2.15-7 de la SDL). Cette interface est représentée sur la figure 5 (page 8)". Si vous souhaitez faire cette interface, vous pouvez vous rendre en section 6.6 (page 10).

L'archive du projet contient un ensemble d'images libres dont vous pouvez vous servir. Le Tux a été réalisé par creek23 et a été récupéré ici. L'image de caisse a été réalisée par lunavorax et a été récupérée ici. L'image de base utilisée pour les murs a été faite par Iwan Gabovitch et a été récupérée ici. Vous pouvez vous servir d'autres images si vous le souhaitez mais vous devez vous assurer que leurs licences le permettent.

5 Programmation par contrat

Il vous sera demandé dans plusieurs fonctions de vérifier grâce à des assertions que les préconditions ou les postconditions de certaines fonctions sont remplies. Ce paradigme de programmation s'appelle la programmation par contrat. Par exemple, pour modifier une case d'un plateau, il vous est demandé de vérifier que la case demandée est bien dans le plateau. Faire ce type de vérification avec des assertions n'est pas juste esthétique :

— Les assertions n'ont pas la même sémantique que les tests dans des instructions conditionnelles. Un if dont la condition n'est pas respectée n'est pas un bug, cela fait juste partie du flot de contrôle du programme. Une assertion non respectée implique par contre qu'une partie du code n'a pas été appelée correctement (ou avec des variables dans un état qui n'est pas le bon) et donc que le programme comporte un bug.
— Contrairement aux instructions conditionnelles, les assertions ne ralentissent pas le programme lorsqu'il est compilé en mode *release*.
— Les assertions sont rapides à écrire et peuvent vous faire économiser beaucoup de temps en debug. En effet, si depuis n'importe quelle ligne de n'importe quel fichier source de votre projet, vous souhaitez modifier une case invalide du plateau de jeu et qu'une assertion vérifie que la case est

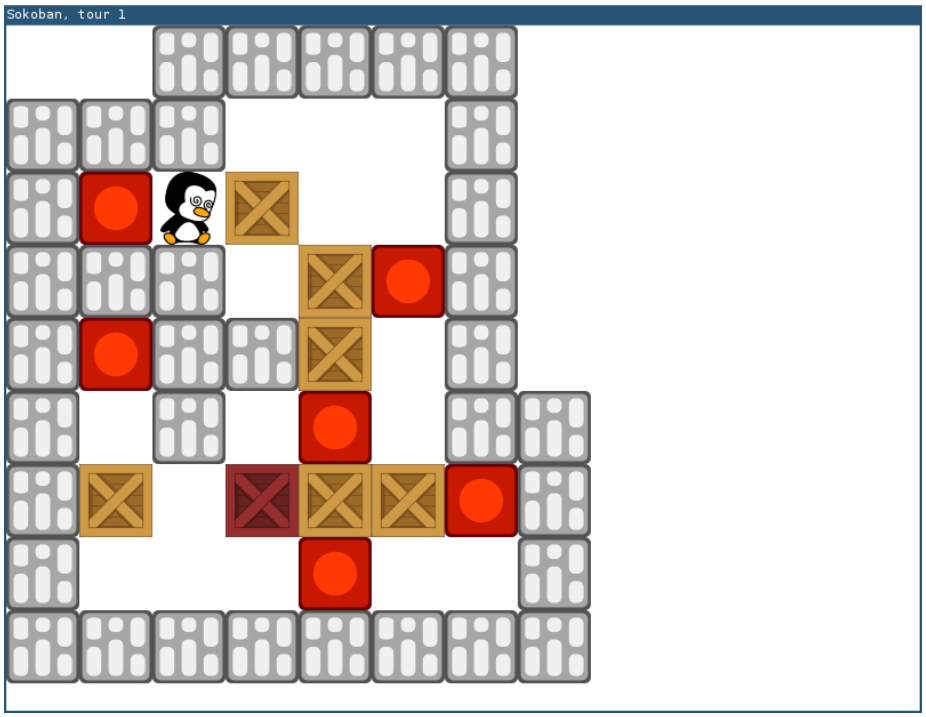

FIGURE 5 – L'interface utilisateur en mode graphique, qui utilise la bibliothèque SDL

valide à l'intérieur de cette fonction, vous saurez tout de suite que votre programme comporte un bug puisqu'il s'arrêtera lors de l'exécution en vous précisant qu'une assertion n'a pas été respectée. Il suffit ensuite de lancer votre programme dans un debugger (*nemiver* par exemple) et vous saurez tout de suite dans quel contexte votre fonction a mal été appelée.

6 Comment se servir de la base de code fournie ?

6.1 Appel de make

Pour compiler la base de code fournie, vous devez vous placer dans le dossier fourni et utiliser la commande make. Pour obtenir un exécutable pouvant être debuggé facilement grâce à un debugger, vous pouvez taper :

```
make debug # ou tout simplement make
```

Si vous souhaitez compiler votre projet en mode *release* (le code sera optimisé et beaucoup moins facilement debuggable), vous pouvez taper :

```
make release
```

6.2 Configuration du Makefile

Le Makefile que je vous ai fourni est un fichier texte, il peut être modifié par n'importe quel éditeur de texte. Les lignes commençant par un '#' sont des commentaires dans un Makefile.

6.2.1 Choix du compilateur

Vous pouvez choisir votre compilateur en jouant sur les commentaires des deux premières lignes du fichier. Ainsi, vous pouvez transformer les lignes

```
# CC=clang
CC=gcc
```

en

```
CC=clang
# CC=gcc
```

afin d'utiliser clang plutôt que gcc.

6.3 Génération de la documentation

Si vous préférez lire la documentation des fonctions dans un navigateur web ou un lecteur PDF plutôt qu'au milieu du code, vous pouvez installer doxygen et taper cette commande depuis le dossier racine fourni :

```
doxygen
```

Cette commande devrait générer différentes sorties en fonction de ce qui est installé sur votre machine. Vous devriez dans tous les cas pouvoir ouvrir la version HTML grâce à la commande :

```
firefox doc/html/index.html
#ou chromium doc/html/index.html
```

6.4 Compilation en utilisant les bibliothèques annexes (ncurses, SDL)

Le code de base qui vous est fourni est fait pour fonctionner sans bibliothèques annexes, mais permet également de les prendre en compte. Si ncurses et la SDL sont installées sur votre machine, vous pouvez jouer avec les commentaires du Makefile pour que les lignes

```
# LDFLAGS=-lncurses -lSDL -lSDL_image
LDFLAGS=
```

deviennent

```
LDFLAGS=-lncurses -lSDL -lSDL_image
# LDFLAGS=
```

Vous pouvez bien sûr modifier ces lignes, par exemple si vous souhaitez utiliser uniquement la SDL et pas ncurses, vous pouvez taper

```
LDFLAGS=-lSDL -lSDL_image
```

6.5 Que faire si on souhaite utiliser la bibliothèque ncurses ?

Après avoir installé la bibliothèque sur votre machine, modifié le Makefile en conséquence et lu un tutoriel sur ncurses, vous devez modifier une ligne du fichier interface_ncurses.c pour que

```
// #include <ncurses.h>
```

devienne

```
#include <ncurses.h>
```

Vous n'aurez ensuite plus qu'à compléter les fonctions dudit fichier.

6.6 Que faire si on souhaite utiliser la bibliothèque SDL ?

Après avoir installé la bibliothèque sur votre machine, modifié le Makefile en conséquence et lu un tutoriel sur la SDL, vous devez décommenter certaines lignes de certains fichiers. Tout d'abord, dans interface.h, vous devez modifier la ligne

```
// #include <SDL/SDL.h>
```

pour qu'elle devienne

```
#include <SDL/SDL.h>
```

Vous devez également modifier, dans interface.h, les lignes suivantes

```
typedef struct
{
    // SDL_Surface * screen;       //! La surface qui représente la fenêtre
    // SDL_Surface * crate;        //! La surface d'une caisse
    // SDL_Surface * crate_ok;     //! La surface d'une caisse sur un lieu de stockage
    // SDL_Surface * player;       //! La surface d'un joueur
    // SDL_Surface * player_ok;    //! La surface d'un joueur sur un lieu de stockage
    // SDL_Surface * storage;      //! La surface d'un lieu de stockage
    // SDL_Surface * wall;         //! La surface d'un mur
} UI_data;
```

pour qu'elles deviennent

```
typedef struct
{
    SDL_Surface * screen;       //! La surface qui représente la fenêtre
```

```
    SDL_Surface * crate;        //! La surface d'une caisse
    SDL_Surface * crate_ok;     //! La surface d'une caisse sur un lieu de stockage
    SDL_Surface * player;       //! La surface d'un joueur
    SDL_Surface * player_ok;    //! La surface d'un joueur sur un lieu de stockage
    SDL_Surface * storage;      //! La surface d'un lieu de stockage
    SDL_Surface * wall;         //! La surface d'un mur
} UI_data;
```

Vous devez ensuite décommenter une ligne dans le fichier interface_sdl.h de telle sorte que

```
// SDL_Surface * charger_image(const char * nom_fichier);
```

devienne

```
SDL_Surface * charger_image(const char * nom_fichier);
```

Vous devez ensuite décommenter les inclusions du fichier interface_sdl.c afin que les lignes

```
// #include <SDL/SDL.h>
// #include <SDL/SDL_image.h>
```

deviennent

```
#include <SDL/SDL.h>
#include <SDL/SDL_image.h>
```

Enfin, vous pouvez décommenter la fonction charger_image dans interface_sdl.c afin que les lignes

```
// SDL_Surface *charger_image(const char *nom_fichier)
// {
//     return NULL;
// }
```

deviennent

```
SDL_Surface *charger_image(const char *nom_fichier)
{
    return NULL;
}
```

Vous n'avez maintenant plus qu'à compléter les fonctions des fichiers interface_sdl.h et interface_sdl.c.