
CONCEPTION LOGICIEL : RAPPORT DE DÉVELOPPEMENT DU PROJET

LEDOUX Eloi,
TAUPIN Erwann,
KITSOUKOU Manne Emile,
L1 informatique,
Groupe 2B,
PROJET SOKOBAN

Avril 2021



UNIVERSITÉ
CAEN
NORMANDIE

Sommaire

1	Introduction	1
1.1	Choix du sujet	1
1.2	Les objectifs du projet	1
2	Travaux préliminaires	1
2.1	Renseignement sur le Sokoban	1
2.2	Répartition des fichiers	2
2.3	La répartition des tâches	2
3	Structure du projet	3
3.1	Fichier de structure d'une grille	3
3.1.1	Création d'un fichier de grille global	3
3.1.2	Lecture d'un niveau de Sokoban	4
3.1.3	Adaptation des symboles pour le programme	4
3.2	Première version jouable en console	6
3.2.1	Les déplacements du joueur	6
3.2.2	Les déplacements de caisses	6
3.2.3	Les conditions de victoire	7
4	Approche graphique	8
4.1	Gestion des états du jeu par couches	9
4.2	La couche menu, la couche de sélection de niveau et la couche de fin	9
4.2.1	Les boutons	10
4.2.2	Le score	11
4.3	La couche de jeu	11
4.3.1	L'objet Image	12
4.3.2	La méthode react_to de la couche de jeu	13
5	Les ajouts de confort de jeu	13
5.1	Les indications textuelles	13
5.2	Les sons du jeu	14
5.3	Les touches de "sauvetage"	14
5.3.1	Le retour arrière	14
5.3.2	La remise à zéro du niveau	14
6	L'algorithme de résolution	14
6.1	Recherche d'un algorithme compatible	14
6.2	Adaptation de l'algorithme	15
6.2.1	Modifications du programme	15
6.2.2	Intégration de l'algorithme au jeu	15
7	Manuel d'utilisation et Illustration de l'usage	16
7.1	Manuel d'utilisation	16
7.2	Illustration de l'usage	17
8	Possibles améliorations	18
9	Conclusion	18
10	Annexes	19

1 Introduction

1.1 Choix du sujet

Suite à la création du groupe nous devons choisir un sujet parmi ceux proposés. Nous nous sommes rapidement entendus sur la réalisation d'un Sokoban.

Le Sokoban est un jeu vidéo dans lequel le joueur doit ranger des caisses sur des cases prédéfinies. Pour cela, le joueur peut se déplacer dans les quatre directions, mais il ne peut pousser qu'une seule caisse à la fois, de plus, il n'a pas la possibilité de la tirer.

Une fois que toutes les caisses sont sur les cases cibles, le joueur passe au niveau suivant. L'objectif du jeu est de réussir à compléter les différents niveaux avec le moins de coups possibles.

1.2 Les objectifs du projet

Dès le départ, nous avons plusieurs objectifs principaux que nous voulions réussir à mettre en place, mais aussi quelques objectifs secondaires que nous souhaitions implémenter s'il nous restait suffisamment de temps.

Au début du projet, nous avons trois objectifs principaux. Le premier était d'avoir un jeu jouable et plutôt joli. Le second était assez simple mais nous a demandé un peu de réflexion : nous souhaitions avoir la possibilité d'importer de nouveaux niveaux en les déposant simplement dans un répertoire spécifique. Et enfin, le troisième objectif était de laisser la possibilité au joueur de choisir le niveau sur lequel il souhaite commencer, et de pouvoir à tout moment revenir au menu principal et sélectionner un autre niveau.

Au fur et à mesure de l'avancement du projet, certains objectifs secondaires sont devenus des objectifs principaux, par exemple, la possibilité de revenir d'un ou plusieurs coups en arrière, la possibilité de réinitialiser un niveau, ou encore l'ajout d'un score.

Lorsque l'on a constaté que le projet avançait assez rapidement et sans trop de difficultés, nous avons décidé d'ajouter d'autres fonctionnalités. Parmi celles-ci, on peut citer l'ajout d'un écran de fin indiquant le score lorsque tous les niveaux sont complétés, mais aussi l'ajout de son (au début d'un niveau, mais aussi à chaque coup). Une fois le son ajouté, une autre fonctionnalité s'est imposée : la possibilité de couper le son !

Enfin, une fois les fonctionnalités précédentes ajoutées, nous avons voulu implémenter un solveur / un algorithme de résolution, qui permet de résoudre les niveaux automatiquement.

2 Travaux préliminaires

2.1 Renseignement sur le Sokoban

Avant de nous lancer dans la création de fonctions et l'écriture du code, nous avons cherché des renseignements sur le Sokoban, son fonctionnement, ainsi que sur le format sok. La page Wikipedia du Sokoban nous a bien aidé à comprendre le fonctionnement de ce jeu, et le site sokobano.de nous en a appris plus sur le format sok utilisé pour les fichiers des différents niveaux.

2.2 Répartition des fichiers

Une fois les recherches préliminaires effectuées, nous avons pu réfléchir à la façon dont nous allions répartir les différentes fonctions.

Nous voulions, au départ, avoir trois fichiers :

- un fichier Grille, regroupant les fonctions de base permettant de modifier une grille ;
- un fichier Level, qui hériterait de Grille, et qui appliquerait les fonctions de base à notre jeu ;
- enfin, un fichier Gestion, qui pourrait gérer intégralement une partie de Sokoban.

Cependant, au fur et à mesure que nous avançons, nous nous sommes aperçus que nous ne pourrions pas garder cette répartition.

Ainsi, nous avons désormais six fichiers :

- un fichier grille 10, qui regroupe les fonctions de base permettant de modifier une grille ;
- un fichier level 10, qui hérite de Grille, et qui applique les fonctions de base à notre jeu ;
- un fichier multimedia 10, qui permet de gérer les images ainsi que le son ;
- un fichier solver, qui contient le code du solveur que nous avons récupéré et dont nous nous servons ;
- un fichier frame 10, qui contient les différents écrans de jeu ;
- enfin, un fichier main, qui permet de gérer la partie de Sokoban.

2.3 La répartition des tâches

Dès le début, il a été clair que la réalisation du projet serait réalisée de la façon la plus collective possible. Ainsi, pour une meilleure efficacité, nous avons décidé que chacun devrait toucher en partie à l'ensemble des implémentations qui auront été effectuées. C'est ainsi que :

- Eloi et Emile se sont chargés du développement du module grille
- Le module Level a été développé par Erwann avec la collaboration de Eloi et Emile pour le développement des déplacements dans un niveau
- Erwann, en collaboration avec Eloi, s'est occupé principalement du module frame
- Erwann s'est occupé de la recherche d'un algorithme fonctionnel pour l'IA
- Emile s'est chargé du module multimédia
- L'intégration de l'algorithme a été pensée par l'ensemble du groupe
- L'ensemble du rapport et du diaporama a été réalisé collectivement

Cette répartition reflète uniquement de façon exhaustive les acteurs lors de l'implémentation de chaque partie. Mais dans le fond, il faut souligner la dévotion et l'engagement de chacun dans le développement du projet. Car, tous les membres du groupes ont apporté leur collaboration autant pour comprendre la tâche des autres mais aussi pour parfaire les idées de chacun.

3 Structure du projet

3.1 Fichier de structure d'une grille

Lors de nos premières réflexions sur le projet, l'idée qui nous a été confiée par nos enseignants fût d'envisager le jeu de Sokoban comme un jeu entièrement jouable au travers de tableaux Python en deux dimensions. Pour cela, il nous fallait donc trouver un moyen de gérer, en simultanée, le jeu à travers la grille, la lecture du niveau, ainsi que les actions du joueur. Notre réponse à cette problématique a alors été d'envisager ces paramètres à travers deux classes : une classe Grille regroupant le plus de possibilités d'interactions envisageables avec un tableau en deux dimensions, sans lien réel avec le Sokoban, et une classe Level héritant de Grille qui adapterait un niveau de Sokoban en tableau et permettrait de gérer ce même tableau au travers de fonctions plus spécifiques au jeu lui même.

3.1.1 Création d'un fichier de grille global

L'un de nos premiers objectifs lors de notre passage sur machine fût donc de concevoir un fichier Python de manipulation de listes en plusieurs dimensions. Ce fichier devait être suffisamment exhaustif pour proposer l'intégralité des opérations qui nous serait nécessaire par la suite lors de la création du jeu, mais devait être aussi réutilisable dans l'éventualité où nous souhaiterions créer un autre jeu/une autre application nécessitant des manœuvres sur grilles.

Tout d'abord, nous avons commencé par regrouper les fonctions "basiques" de manipulation de listes que nous connaissions dans une classe intitulée "Grille". Dans un souci d'exhaustivité, bien que nous ayons essayé de créer un maximum de fonctions par nous même, nous avons fini par le compléter en y ajoutant des fonctions reprises du cours de programmation du premier semestre.

Ainsi, la classe Grille devient le cœur de notre projet, où plutôt son squelette. Son fonctionnement est assez basique : la classe prend en paramètre un nombre de lignes et un nombre de colonnes puis utilise des paramètres lors de son initialisation pour créer un tableau vide aux dimensions convenues, récupérable sous le nom de "plateau". Ce plateau a pour intérêt de pouvoir : apporter des modifications à la grille, l'afficher d'une certaine façon en console, ou encore vérifier la présence/l'emplacement de certains éléments dans le tableau à partir des fonctions de la classe Grille. L'idée est ici de pouvoir mettre en place une classe qui pourra servir à une future classe, celle-ci nous permettra de transformer un fichier de niveau de jeu en une grille de plateau.

Le lot de fonctions le plus utile à ce stade du développement regroupe toutes les fonctions permettant de traiter des données du tableau à travers leur numéro de cellule. En pratique, pour chaque colonne de chaque ligne du tableau, la fonction `trouve case` associe un numéro de case qu'il sera possible de retrouver/modifier avec les fonctions de la classe. Cet ajout est particulièrement utile pour retrouver l'emplacement d'un certain symbole au sein du plateau créé/modifié par la fonction mais se révélera encore plus efficace lorsqu'il s'agira de reconnaître les différents éléments qui composent un niveau de Sokoban.

Enfin, dans cette optique de "préparation", nous avons amorcé la représentation graphique du jeu en appliquant en paramètre de la classe Grille la possibilité de choisir la taille d'une case du tableau, en pixel, ainsi que la marge qui l'accompagne, l'idée étant de se préparer à toute éventualité. Au moment de la création de la classe ce concept demeurerait cependant relativement abstrait, en effet, aucun d'entre nous n'avait auparavant travaillé avec Pygame ou une quelconque interface graphique.

3.1.2 Lecture d'un niveau de Sokoban

Comme dit précédemment, une fois la classe Grille fonctionnelle, il était ensuite nécessaire de la faire hériter par une autre classe de gestion de tableaux possédant une initialisation et des fonctions plus spécifiques au jeu du Sokoban. C'est dans cette optique que nous avons conçu la classe Level.

Pour parfaitement comprendre cette classe, il est important de comprendre son fonctionnement. A l'instar de la classe Grille, la classe Level va construire un tableau qui, cette fois, ne sera pas vide mais contiendra l'intégralité des symboles qui composent le niveau de Sokoban (en .sok) désiré. Pour se faire, la classe prend en paramètre un chemin vers le niveau de Sokoban puis à l'initialisation, la classe va lire ce niveau en le stockant dans une variable.

La construction du niveau s'opère ensuite grâce à la formation particulière d'un niveau de Sokoban. Il faut savoir qu'un fichier .sok comprenant un tel niveau s'organise comme un fichier texte dont la première ligne contient les données du nombre de colonnes et des lignes (dans cet ordre) du niveau en question. Ainsi, la classe peut alors récupérer ces données afin de les utiliser pour initialiser un tableau vide aux bonnes dimensions à partir de la fonction Grille. Ensuite, cette même grille sera remplie ligne par ligne à partir de la deuxième ligne du fichier où débute la représentation du niveau (voir annexe A). Pour ce faire, on utilise une boucle comme suit :

*pour ligne de fichier[2e ligne]
 plateau[ligne] = liste(replace plateau[ligne] par fichier[ligne])*

Ainsi, on obtient un tableau en deux dimensions aux proportions désirés contenant pour chaque case un symbole de la représentation du niveau. De ce fait, on peut à partir de ce moment là, considérer que l'on a obtenu une représentation du niveau manipulable par les fonctions de la classe Grille ainsi que par de futures fonctions de jeu.

Il est important de noter aussi qu'une fois ce travail effectué, la classe sauvegardera l'état initial du niveau dans une variable nommée "historique". Un ajout qui nous permettra, plus tard, de gérer une fonction de "reset" du niveau.

3.1.3 Adaptation des symboles pour le programme

Une fois le niveau adapté sous forme de plateau de jeu à travers un tableau Python en deux dimensions, il nous était désormais possible d'y apporter des modifications à l'aide de fonctions de la classe Grille. Si nous avions déjà envisagé comment gérer les déplacements du personnage lors de nos travaux préliminaires, il fallait d'abord trouver un moyen de communiquer à la machine comment identifier ce même personnage au sein de la grille ainsi que tous les obstacles qui pourraient être rencontrés par ledit personnage.

Dans un premier temps, voici une liste des différents symboles qui composent un niveau de Sokoban et qui, par conséquent, devront être explicités à la machine afin de leur attribuer différents comportements :

- @ - Il s'agit du personnage joueur. C'est ce symbole qui se déplacera dans le tableau à l'aide de la fonction de mouvement à chaque input de l'utilisateur. Il peut aller vers le haut, vers le bas, vers la gauche et vers la droite.
- # - Il s'agit d'un mur. C'est le symbole qui délimite la zone de jeu, il est inamovible et ne changera jamais de coordonnées pendant la partie.
- \$ - Il s'agit d'une caisse. Ce symbole pourra, comme le joueur, se déplacer avec la fonction de mouvement. En revanche, il ne répondra pas directement aux inputs du joueur dans le sens où une caisse ne bougera que dans la direction vers laquelle elle est poussée par le joueur.
- . - Il s'agit d'un emplacement de caisse. Ce symbole ne se déplacera pas non plus au sein du tableau. L'objectif du joueur est de déplacer les caisses sur ces emplacements en les poussant, ce qui aura pour effet de les transformer en symbole * tant que la caisse est sur l'emplacement.
- * - Il s'agit d'une caisse lorsqu'elle est sur un emplacement de caisse. Chacune de ces cases incrémente un nombre initialisé à 0 au début de la partie qui, lorsqu'il atteint le seuil défini par le nombre d'emplacements de caisses sur le plateau, confirme la complétion du niveau. Cette case est aussi à considérer comme un symbole "inamovible" dans le sens où si le joueur pousse la caisse de son emplacement, la caisse réapparaît sur la case suivante dans le sens de la poussée tandis que la case d'emplacement, maintenant chevauchée par le symbole du joueur, se transformera en symbole + jusqu'au prochain déplacement.
- + - Il s'agit du joueur lorsqu'il est sur un emplacement de caisse. Cette case, au même titre que le symbole précédent, ne bougera pas au sens propre ; elle se retransformera en emplacement de case une fois que le joueur se déplacera. Elle est particulièrement utile pour permettre à la machine de suivre les mouvements du joueur lorsqu'il avance sur un emplacement de case sans pour autant totalement remplacer le symbole de ce dernier par le symbole du joueur et ainsi perdre une condition de victoire.
- " " - Le symbole vide (ou ESPACE) représente tous les endroits du plateau où le joueur peut se déplacer librement sans enclencher de comportement spécifique de la part du jeu.

La méthode la plus efficace que nous avons trouvée pour reconnaître les différents symboles du plateau est la fonction `identite` de la classe Grille. Cette fonction a pour effet de vérifier si un numéro de case pris en paramètre correspond à un symbole lui aussi pris en paramètre. Ainsi, dans la classe Level, des fonctions comme `est_une_boite` ou `est_un_mur` permettent de vérifier le contenu d'une case et ainsi d'y appliquer les propriétés désirées. Ces fonctions sont ensuite utilisées lors des déplacements du joueur afin de vérifier comment chaque input de l'utilisateur modifie le tableau en fonction de la direction dans laquelle se dirige le joueur. En conséquence, le jeu ne considère jamais vraiment le niveau dans son ensemble en attribuant à chaque symbole des propriétés, mais se contente de vérifier quels sont les symboles qui entourent le joueur (et les boites lorsqu'elles se déplacent) pendant la résolution de la fonction de déplacement.

3.2 Première version jouable en console

Avec la machine capable de comprendre comment envisager chaque partie du plateau, il était désormais possible d'entrevoir une version du jeu jouable sans utiliser d'interface graphique. Les questions qui demeuraient alors étaient celles relatives au système de déplacement du personnage ainsi que celui des caisses, dépendant directement du précédent. Enfin, il nous fallait observer comment définir une condition de victoire pour chaque niveau.

3.2.1 Les déplacements du joueur

Comme expliqué plus haut, les déplacements du personnage avaient déjà été évoqués au sein du groupe. Nous avons donc commencé à travailler à partir de l'idée que les inputs "h" (haut), "b" (bas), "g" (gauche), "d" (droite) suffiraient à faire évoluer le personnage au sein de la grille. Pour se faire, nous avons simplement décidé d'une condition **si** qui, suivant l'input, attribuerait à chacune des lettres un effet. Ainsi, si l'utilisateur entre "h", la console décrémente la position du joueur de 1 au niveau des lignes, au niveau des colonnes si il a entré "g"; pour "b" et "d" on incrémentera au lieu de décrémenter. Bien que cette solution soit efficace, on optera plus tard pour une reconnaissance des directions telles que : "h": 0, "d": 1, "b": 2, "g": 3 et une évolution par case à travers des fonctions comme **case player** et **case suivante** qui permettront de vérifier le symbole de la case en question. L'utilité principale de cette modification étant de simplifier l'organisation du code en évitant de se perdre à travers de trop nombreuses boucles conditionnelles. Plus tard, lorsque nous avons commencé à travailler avec Pygame, nous avons fait en sorte que les inputs "classiques" de déplacement tels que ZQSD ou les flèches directionnelles renvoient directement les lettres auxquelles ils sont associés afin de ne pas avoir à réorganiser tout le système. Bien qu'avec Pygame la gestion du jeu soit basée sur une boucle d'événements répétant la fonction de mouvement en continu pendant l'exécution d'un niveau, lors de la réalisation du jeu en console nous avons simulé cette exécution en répétant la fonction de déplacement **tant que** les conditions de victoire n'étaient pas remplies. Ainsi l'utilisateur était amené à entrer une nouvelle lettre en console jusqu'à voir apparaître le mot "victoire" en console bien que le jeu continue de tourner.

3.2.2 Les déplacements de caisses

Le déplacement des caisses fonctionne de façon relativement similaire à celui du personnage principal, dans le sens où il s'agira d'un changement de symbole au sein du plateau en fonction de l'emplacement de départ et celui d'arrivée. Pour convenir de la direction de la caisse, comme il n'est pas déterminé par un input de l'utilisateur, la fonction **mouvement boite** de la classe Level prend en variable la position du joueur et détermine la disponibilité de la case qui suit à partir de l'orientation du joueur. En effet, un déplacement de caisse aura forcément lieu pendant un déplacement du joueur, ainsi il sera possible de récupérer la direction en fonction de l'input de l'utilisateur. Pour approfondir cette idée de dépendance avec le déplacement du joueur, il faut comprendre que la fonction **mouvement boite** s'exécute à l'intérieur de la fonction **mouvement du personnage** qui elle s'exécutera à chaque input. Comme la fonction **mouvement du personnage** va vérifier la présence d'un symbole sur la case vers laquelle s'oriente le joueur, si elle y détecte une caisse alors elle pourra résoudre **mouvement boite**, dans le cas contraire elle agira en fonction

du symbole qui suit (dans la plupart des cas le joueur avancera simplement, à l'exception d'une rencontre avec un mur qui redéfinira l'emplacement du joueur à son emplacement avant l'input de l'utilisateur). Pour ce qui est de la détection du symbole dans la direction de la caisse, on a choisi de reprendre simplement le même type de vérification que pour le joueur en utilisant la fonction `case suivante` pour obtenir la case et les différentes fonctions `est un X` pour en identifier le symbole. Dans les faits, les transformations sont très similaires à celle du joueur :

- Si la caisse rencontre un mur, les déplacements sont "annulés" : le symbole de la caisse est transformé en un symbole de caisse et celui du joueur en un symbole de joueur, ce qui donne une impression d'inaction.
- Si la caisse rencontre un emplacement de caisse, le symbole de la caisse devient un symbole de joueur et celui de l'emplacement de caisse devient un symbole de caisse sur emplacement.
- Si la caisse rencontre une autre caisse, on applique la même transformation que si elle avait rencontré un mur puisqu'il est impossible de bouger deux caisses en même temps.

Avec cette méthode de déplacement de caisse liée au déplacement du personnage, on évite tous les problèmes que nous avons envisagés en amont. Ainsi, pas de gestion de collision, de physique des caisses ou d'algorithme de poussée mais simplement des permutations de symboles en fonction de certaines conditions.

3.2.3 Les conditions de victoire

A partir du moment où il était possible de déplacer en continu le personnage tout en le faisant interagir avec les objets du décor, il ne manquait à la version jouable en console qu'une condition de victoire qui viendrait mettre fin à cette boucle de déplacement.

Bien que notre première approche fut d'incrémenter un nombre à chaque caisse poussée sur un emplacement en le comparant à un autre défini par la quantité d'emplacements du niveau à chaque déplacement, nous sommes parvenus à trouver une méthode plus efficace assez rapidement. Cette méthode se repose sur une fonction de la classe Grille nommée `compteur de valeur` qui prend en paramètre un symbole et renvoie un nombre entier correspondant à la quantité de symboles désignés au sein du tableau. A partir de cette fonction, il est possible d'en créer une nouvelle définissant le nombre de boîtes sur le terrain en comptant à la fois les symboles `*` et `$`. Cette fonction aura pour principe de définir le total de boîtes à "activer" afin de réussir le niveau. Elle sera comparée à une autre fonction nommée `sont actives` qui utilisera le même principe mais ne comptera que les symboles `*` afin de compter le nombre de boîtes activées. Cette comparaison aura lieu dans une troisième fonction `victoire` qui s'initialisera à la fin de chaque mouvement afin de vérifier si toutes les boîtes ont été activées ou non. Si tel est le cas, la fonction `victoire` retournera Vrai sinon elle retournera Faux. Le principe étant de faire en sorte que le programme s'arrête ou écrive simplement le mot "victoire" en console.

Avec cela, le prototype du jeu était alors fonctionnel puisqu'il était désormais possible de jouer un niveau entier en console. Il était alors nécessaire de commencer nos recherches sur les différentes possibilités d'adaptations que nous offrait le moteur graphique Pygame afin de passer nos projets à l'état de jeu.

4 Approche graphique

Avant d'entamer l'approche graphique, nous avons au préalable réfléchi sur les cas d'utilisation côté utilisateur. Le but purement recherché était de fournir une interface graphique facile, agréable et compréhensible pour l'utilisateur. De ce fait, nous avons décidé de développer diverses couches graphiques que l'on pourrait manipuler en fonction des choix ou événements de l'utilisateur.

Nous nous sommes alors mis à la place des utilisateurs et avons défini quelques choix et événements qui auraient pu se passer lors d'un lancement du jeu. Ceux qui ont particulièrement retenu notre attention sont :

- **JOUER** : lancer une partie
- **GO_MENU** : retourner au menu principal du jeu
- **RUN_SOLUTION** : demander à l'IA (Intelligence Artificielle) de rechercher une solution permettant de résoudre le niveau en cours
- **AUTOMATIQUE** : lancer la résolution automatique du niveau après la recherche de solution
- **ECRAN_FIN** : c'est un événement qui se produit lorsque l'utilisateur a terminé de résoudre l'ensemble des niveaux disponibles
- **NIVEAUX** : accéder à la liste des niveaux disponibles

Gérer cette approche avec des couches nous contraignait à développer en parallèle des fonctionnalités, mais aussi à gérer les données et informations qui pouvaient exister entre les diverses couches. Ainsi, l'approche graphique n'était pas seulement une étape au cours de laquelle nous nous sommes occupés de l'interface graphique. C'est aussi une étape durant laquelle nous avons géré une programmation événementielle et également gérer les données récupérées côté utilisateur.

Pour le développement, deux(2) choix s'offraient à nous par rapport aux bibliothèques que l'on devrait utiliser :

- **Pygame**
- **Tkinter**

Nous nous sommes alignés sur celle de **Pygame** pour trois(3) principaux points :

- C'est une bibliothèque conçue principalement pour le développement des jeux en deux dimensions(2D)
- Elle nous offrait les meilleures possibilités pour développer facilement notre jeu du point de vue graphique, sonore, et événementiel
- Elle nous donnait les outils de développement qui correspondait à la meilleure approche de notre jeu en console

4.1 Gestion des états du jeu par couches

Une gestion des états du jeu par couches s'appuie essentiellement sur le principe de la programmation événementielle. En effet l'interface graphique qui est affichée à l'écran dépend principalement des événements qui peuvent se produire. Partant de ce principe, il a fallu modéliser ce que nous entendons par couches. Ainsi, une couche est un objet possédant des données, des informations pouvant être manipulées et modifiées à travers deux(2) méthodes :

- **react_to** : Cette méthode s'occupe du côté événementiel de la couche. Elle permet de poster ou réagir à des événements. Elle respecte un certains nombre de principes :

Réagir aux évènements claviers

Modifier les données

Poster des évènements utilisateurs

- **draw** : Cette méthode est liée à l'aspect graphique. Elle permet de dessiner, afficher des éléments sur l'écran associés à la couche. Cela est implémenté dans cette logique :

Mettre un fond de couleur

Afficher des images

Écrire et afficher du texte

Afficher des rectangles (boutons)

Le passage d'une couche à une autre est rendu possible par la détection du programme d'évènements précis défini plus haut.

Côté programme, ces évènements sont définis comme l'addition d'un **pygame.USEREVENT** avec une constante.

```
JOUER = pygame.USEREVENT + 1 # Lancer le jeu
GO_MENU = pygame.USEREVENT + 2 # Aller au menu
NIVEAUX = pygame.USEREVENT + 3 # Aller à la selection des niveaux
ECRAN_FIN = pygame.USEREVENT + 4 # Fin de la resolution de tous les niveaux
RUN_SOLUTION = pygame.USEREVENT + 5 # Lancer la recherche de solution de l'IA
AUTOMATIQUE = pygame.K_ESCAPE + 6 # Lancer le solveur
```

FIGURE 1 – Initialisation des évènements

4.2 La couche menu, la couche de sélection de niveau et la couche de fin

Ces couches ont été développées principalement pour améliorer l'interactivité côté utilisateur. Ainsi du côté utilisateur, ces couches se présentent comme une interface dans laquelle on peut défiler et sélectionner un bouton.

- **La couche menu** : Sur cette couche, l'utilisateur a le choix entre démarrer une partie, aller à la sélection des niveaux ou quitter la partie. Côté programmation, nous effectuons principalement l'affichage graphique et postons des événements.

- **La couche de sélection de niveau** : L'intérêt principal de cette couche est de présenter une liste de niveaux que l'utilisateur pourra sélectionner. Du point de vue programme, cette couche permet de récupérer le niveau sélectionné par l'utilisateur, gère le côté graphique et permet de poster l'évènement **JOUER** quand on sélectionne un niveau.
- **La couche de fin** : Cette couche apparaît quand l'utilisateur termine la résolution du niveau 10. Il affiche le nombre de coups Total et donne les options de poursuites du jeu.



(a) La couche menu

(b) La couche de sélection

(c) La couche de fin

FIGURE 2 – Les couches intermédiaires

4.2.1 Les boutons

Les boutons ont été considérés comme des rectangles de couleurs sur lesquels on peut écrire du texte et modifier également la couleur en fonction de la présence de l'utilisateur sur ce dernier.

Ainsi pour chaque couche de jeu où il est nécessaire de faire apparaître des boutons sur l'écran, on procède par étape :

- Définir la taille de chaque rectangle à travers **pygame.Rect**.
- Associer chaque bouton à un rectangle mais aussi les dessiner et les positionner sur l'écran avec **pygame.draw.rect**.
- Définir le texte qui sera affiché sur le rectangle.

Sur un écran, on peut défiler entre les différents boutons grâce aux touches directionnelles. Actionner un bouton est possible grâce à un attribut des couches **bouton_token**. Cet attribut prend le numéro du bouton sur lequel on est actuellement positionné. Lors de leur activation, selon le bouton, un évènement précis est posté dans la file d'attente des évènements. C'est grâce à cela que l'on peut actionner la bonne fonctionnalité.



FIGURE 3 – Illustration d'une couche possédant des boutons

4.2.2 Le score

Le score a été considéré comme étant le nombre de niveau que l'utilisateur a résolu lors d'une partie de jeu. Ainsi, on peut l'incrémenter de 1 uniquement quand l'utilisateur a terminé la résolution d'un niveau. C'est une donnée qui est disponible uniquement quand l'utilisateur est dans une partie en cours (**couche de jeu**).

4.3 La couche de jeu

La couche de jeu est l'une des plus importantes car elle représente l'état lorsque l'utilisateur est en pleine partie. Sur l'implémentation, elle ne diffère pas des autres couches. Toutefois, elle possède des données supplémentaires et fait également appel à d'autres objets déjà implémentés.

Cette couche représente le carrefour de l'ensemble du projet. En effet, elle fait appel à l'utilisation des objets importants déjà implémentés (**Level**, **Image**).

Dès l'initialisation de la couche de jeu, on définit :

- Le numéro du niveau : de 0 à 10
- On construit le niveau à partir de l'objet Level
- On charge les images de jeu sokoban avec l'objet Image
- On définit le score, le nombre de coup, le nombre de coups total
- On définit les touches de déplacement que l'on utilisera dans la partie.

C'est dans cette couche que l'on gère à proprement dit la résolution d'un niveau de la part de l'utilisateur.



FIGURE 4 – Illustration de la couche de jeu

4.3.1 L'objet Image

La classe Image est une classe qui nous permet de charger des images du jeu et de les associer à des caractères précis. Elle possède deux(2) principales méthodes qui permettent de pouvoir afficher sur un écran les images associées à ces caractères.

- **dessiner_grille** : Elle permet une représentation graphique d'un tableau 2D grâce à un parcours par ligne et colonnes.
- **dessiner_level** : C'est une méthode qui est d'avantage liée à la conception de notre projet car elle permet la représentation graphique d'un Objet Grille ou level grâce à un parcours par numéro de cases.

Les deux(2) fonctionnent selon un même algorithme.

Début de la méthode :

itération sur les éléments du tableau 2D(par ligne et colonne ou par numéro de case) :

tester si le caractère est dans le dictionnaire des caractères prédéfinis

afficher le caractère sur l'écran à des coordonnées calculées selon sa position dans

le tableau

Fin de la méthode

```
def dessiner_level(self, niveau: Union[level.Level, grille.Grille], ecran: Union[Surface, SurfaceType]) -> None:
    """
    methode permettant de dessiner une grille(Grille) ou un niveau(Level) par numero de case

    :param niveau: un objet Level ou Grille à dessiner
    :param ecran: surface sur laquelle il faut afficher les caractères graphiques
    """
    for numero_case in range(niveau.taille):
        val = niveau.valeur_case(numero_case)
        if val in self.representation:
            ecran.blit(self.representation[val], niveau.coordonnees_graphique(numero_case, self.taille))
```

FIGURE 5 – Exemple pour la méthode **dessiner_level**

4.3.2 La méthode `react_to` de la couche de jeu

La méthode `react_to` permet de récupérer les événements claviers et de modifier ainsi certaines données. A travers les événements récupérés, elle permet notamment le déplacement de l'utilisateur. Elle parvient à modifier les données à travers trois(3) éléments :

- Les touches de jeu : Quand l'utilisateur utilise l'une des touches nécessaires aux déplacements du personnage, le nombre de coup et nombre de coup, le nombre de coups total sont incrémenter, le plateau de jeu (construct) est modifié pour appliquer le déplacement.
- La touche résolution : Elle permet de basculer vers la couche de résolution qui permet de passer à la résolution du niveau courant.
- La victoire de l'utilisateur : On teste la condition de victoire, si elle est vraie alors on charge un nouveau niveau et on incrémente le score, ou alors on bascule sur la couche de fin.

5 Les ajouts de confort de jeu

Dans le but d'améliorer l'expérience de l'utilisateur, certaines fonctionnalités supplémentaires ont été pensées.

5.1 Les indications textuelles

Durant une partie, il semblait nécessaire d'ajouter certaines indications à l'utilisateur. Ces indications portent sur les informations que l'on recueille sur la partie en cours (score, nombre de coups, nombre de coups total....) mais aussi sur les touches qui permettent la navigation.



FIGURE 6 – illustrations d'un exemple d'indications textuelles

5.2 Les sons du jeu

Le son a été envisagé pour rendre le jeu moins banal, et ajouter une véritable sensation d'interaction du côté utilisateur. Cela est réalisé dans le module multimédia qui contient un objet **Audio**, celui-ci charge l'ensemble des sons, et grâce à ces modules peut jouer l'un des sons. Ainsi deux(2) principaux sons peuvent être écoutés par l'utilisateur :

- Un son lorsqu'il appuie sur une touche du clavier.
- Un son pour le lancement d'une partie.

5.3 Les touches de "sauvetage"

Les touches de "sauvetage" sont des touches qui nous permettent de nous donner une seconde chance lors de la résolution d'un niveau. C'est à dire elles nous permettent d'effectuer soit un retour à la position précédente, ou tout simplement de recommencer le niveau au début.

5.3.1 Le retour arrière

Cette fonctionnalité s'avère pratique dans la mesure où l'utilisateur peut décider de modifier son déplacement. Cela fait appel à la méthode **dernier_plateau** de l'objet **Level**. Cette méthode effectue un **pop** sur l'attribut **historique**¹ de **level** ce qui nous permet de remplacer le plateau courant par le dernier des plateaux tout en le supprimant de l'historique. Ainsi durant une partie, l'utilisateur, en appuyant sur la touche "u", peut revenir au déplacement précédent.

5.3.2 La remise à zéro du niveau

A l'initialisation d'un niveau, une copie de son plateau est préalablement enregistrée ce qui permet de remettre tout à zéro sans pour autant faire une nouvelle initialisation de l'objet.

6 L'algorithme de résolution

6.1 Recherche d'un algorithme compatible

Au départ, nous souhaitions implémenter un algorithme écrit par nos soins, nous avons donc commencé à nous renseigner sur l'algorithme A*, cependant faute de temps, nous avons dû nous résoudre à trouver un algorithme compatible que nous pourrions adapter.

Pour cela nous nous sommes rendus sur GitHub et avons, dans un premier temps, cherché des solveurs de Sokoban codés en Python.

Une fois que nous en avons trouvé plusieurs, nous avons commencé à les tester. Lors de ces tests, un solveur a particulièrement retenu notre attention, car nous pouvions utiliser plusieurs algorithmes : le BFS (Breadth First Search, ou Algorithme de parcours en largeur), le DFS (Depth First Search, ou Algorithme de parcours en profondeur), l'UCS (Uniform Cost Search) et le A*.

1. l'attribut historique de level est une liste python qui contient l'ensemble des plateaux

Ce solveur a été écrit par *KnightofLuna*, et vous pouvez y accéder en cliquant sur ce lien [GitHub](#).

Cependant, nous nous sommes heurtés à un problème : les temps de résolution des niveaux complexes par les différents algorithmes. Pour plus de détails, vous pouvez consulter l'annexe C 10.

6.2 Adaptation de l'algorithme

6.2.1 Modifications du programme

Afin que le solveur puisse directement fonctionner avec nos fichiers .sok, nous avons dû effectuer une modification sur les caractères utilisés pour définir le personnage, les murs, les caisses, les emplacements de caisse, et les caisses sur un emplacement de caisse (qui sont respectivement @, #, \$, ., *). De plus, nous avons dû ajouter la possibilité que le personnage soit sur un emplacement de caisse grâce au caractère +. Nous avons également modifié les actions possibles afin que celles-ci puissent être directement utilisées dans notre solveur automatique, à savoir :

- un coup vers le haut est désormais représenté par un h si le joueur ne pousse aucune caisse, et par un H si le joueur déplace une caisse ;
- un coup vers le bas est désormais représenté par un b si le joueur ne pousse aucune caisse, et par un B si le joueur déplace une caisse ;
- un coup vers la gauche est désormais représenté par un g si le joueur ne pousse aucune caisse, et par un G si le joueur déplace une caisse ;
- un coup vers la droite est désormais représenté par un d si le joueur ne pousse aucune caisse, et par un D si le joueur déplace une caisse.

Enfin, nous avons décidé de ne garder que l'algorithme A*, et donc de supprimer les algorithmes BFS, DFS, et UCS, car ceux-ci prennent trop de temps à résoudre les niveaux pour être utilisés dans notre solveur automatique.

6.2.2 Intégration de l'algorithme au jeu

L'algorithme a été intégré au jeu en implémentant une nouvelle couche. Cette couche ne diffère pas des autres du point de vue de l'implémentation. En particulier, lors de l'initialisation, la recherche de solution par l'IA est lancée. Vu la complexité de certains niveaux, la recherche peut être longue. De ce fait, nous avons choisi de limiter le temps de recherche à **5 secondes**. L'algorithme A* étant le plus pratique du point de vue du temps, pour certains niveaux, nous avons dû faire en parallèle la recherche d'une solution par d'autres méthodes. Ainsi, pour l'ensemble des niveaux présents dans le fichier de base, nous sommes à mesure de fournir une solution.

7 Manuel d'utilisation et Illustration de l'usage

7.1 Manuel d'utilisation

" **A Sokoban Adventure** " est un jeu à l'interface graphique simple et compréhensible par l'utilisateur. Votre but principal est de terminer la résolution de chacun des niveaux. Pour terminer un niveau, vous avez besoin de placer l'ensemble des caisses sur les emplacements prédéfinis. Au cours de vos parties, diverses fonctionnalités du jeu vous permettront de parvenir à ce but.

Ce jeu ne demande aucun prérequis particulier pour y jouer.

a - Comment jouer ?

Pour jouer, vous auriez besoin d'un clavier alphanumérique. L'utilisation d'une souris n'est pas nécessaire.

Pour démarrer le jeu, il vous faudra lancer la lecture du fichier "main.py". Pour cela deux(2) possibilités s'offrent à vous :

- Posséder un **IDLE** : Il vous faudra alors ouvrir le répertoire du jeu dans l'idle et lancer la lecture du fichier **main.py**
- Lancement dans un **Terminal** : Dans le terminal vous devrez vous trouver dans le répertoire du jeu et lancer la commande ***python main.py***

Une fois lancer vous auriez la possibilité de lancer directement une partie ou de choisir le niveau que vous souhaitez résoudre

b - Guide de commande











Touches	Utilisations
   	Se déplacer entre les boutons
 /  ou 	Valider
 ou 	Revenir au menu
	Couper le son

TABLE 1 – Les touches utilisables entre les écrans

Touches	Utilisations
← ↑ → ↓ ou Q Z D S	Déplacer le personnage respectivement à gauche/ haut/ droite/ bas
R	Réinitialiser le niveau en cours
U	Effectuer un undo (revenir à la position précédente)
P	Lancer la recherche de solutions (solver)
Space	Démarrer la résolution automatique
↵ ou Enter	Si vous êtes dans le solveur, revenir à l'écran de jeu
echap ou esc	Revenir au menu
m	Couper le son

TABLE 2 – Les touches utilisables lors d'une partie

7.2 Illustration de l'usage

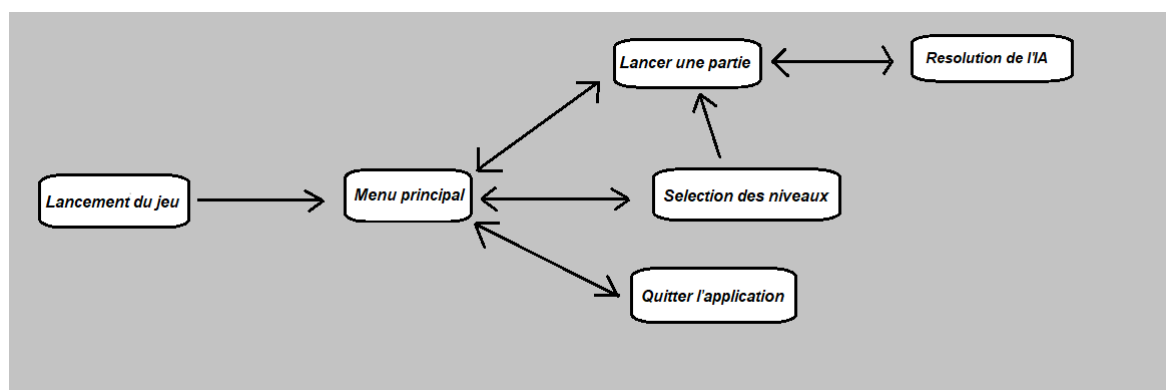


diagramme d'utilisation

Ce diagramme résume l'ensemble de cas d'utilisation que l'on peut avoir quand l'utilisateur souhaiterait démarrer le jeu.

8 Possibles améliorations

Bien que le rendu du projet nous paraisse plus que complet, de par sa stabilité et ses nombreuses fonctions améliorant le confort de jeu, nous gardons à l'esprit qu'il reste grandement améliorable car limité par notre période de développement et les limitations de nos compétences. Si nous devions revenir dessus avec plus de temps et de savoirs, nous avons pensé à quelques améliorations.

- Dans un premier temps, il serait intéressant de pouvoir intégrer des niveaux personnels au jeu simplement par l'intermédiaire d'une interface sur le logiciel qui permettrait à l'utilisateur de définir un chemin vers ce niveau sur l'ordinateur puis l'ajouter à la liste des niveaux. Un ajout de confort qui irait dans ce sens serait l'implémentation d'un nouveau bouton pour le niveau ajouté sur l'interface de sélection des niveaux du Menu.
- Dans le même registre mais nécessitant un savoir bien plus poussé, un générateur de niveau ou une interface de création de niveaux serait aussi une possible amélioration du logiciel. Cela ajouterait de la variété au jeu tout en évitant à l'utilisateur de passer par un fichier .sok pour ajouter son niveau au jeu, rendant la visualisation des niveaux créés bien plus claire.
- Un des problèmes que nous avons rencontré lors de l'implémentation de l'algorithme de résolution au jeu a été de savoir comment proposer une résolution de niveau efficace sans bloquer l'écran trop longtemps à l'utilisateur. Une amélioration envisageable serait de permettre à l'algorithme de tourner pendant que l'utilisateur joue afin qu'il puisse commencer sa résolution à partir de l'endroit où se trouve le joueur au lieu de se contenter de montrer comment réussir le niveau du départ.
- Un système de sauvegarde des scores, de comparaison avec un highscore pourrait aussi être un élément de comparaison entre les utilisateurs à la manière d'une borne d'arcade. De même, pouvoir voir le temps de résolution d'un niveau et diverses statistiques de joueur comme le nombre de déplacement moyen et le temps de résolution moyen par niveau pourrait s'avérer être un ajout intelligent.

Voici un petit échantillon des nombreuses améliorations possibles du projet. Il serait intéressant d'y revenir en fin de cursus pour voir comment nos nouvelles connaissances nous permettraient d'améliorer le projet de différentes façons et surtout d'observer si les améliorations auxquelles nous avons pensées sont réellement réalisables.

9 Conclusion

Grâce à ce projet, nous avons pu approfondir nos connaissances en Python, notamment nos connaissances sur Pygame, mais également nos connaissances en algorithmie / algorithmique. De plus, nous avons pu découvrir le travail en équipe sur un projet d'assez longue durée, ce qui nous a permis de nous répartir le travail, en nous rapprochant du mode de fonctionnement que l'on peut trouver dans les entreprises.

10 Annexes

A - Fichier .sok

```

1  14 07
2      ####
3  #####. .##
4  # $ @ $ .###
5  # $$ ## . .#
6  # $ $$ ## . .#
7  #          **####
8  #####
9

```

FIGURE 7 – Exemple de contenu d'un fichier .sok

B - Les sources des fichiers multimedia

- Les images ont été récupérées du projet Sokoban, réalisé par *Kazantzakis*, déposé sur ce lien *GitHub*.
- Les sons ont été téléchargés sur *freesound*.

C - Les différents algorithmes

Une absence de donnée (valeur égale à 0) signifie que nous n'avons pas pu exécuter l'algorithme jusqu'à ce qu'une solution soit trouvée car cela prenait beaucoup trop de temps et / ou beaucoup trop de mémoire vive (RAM).

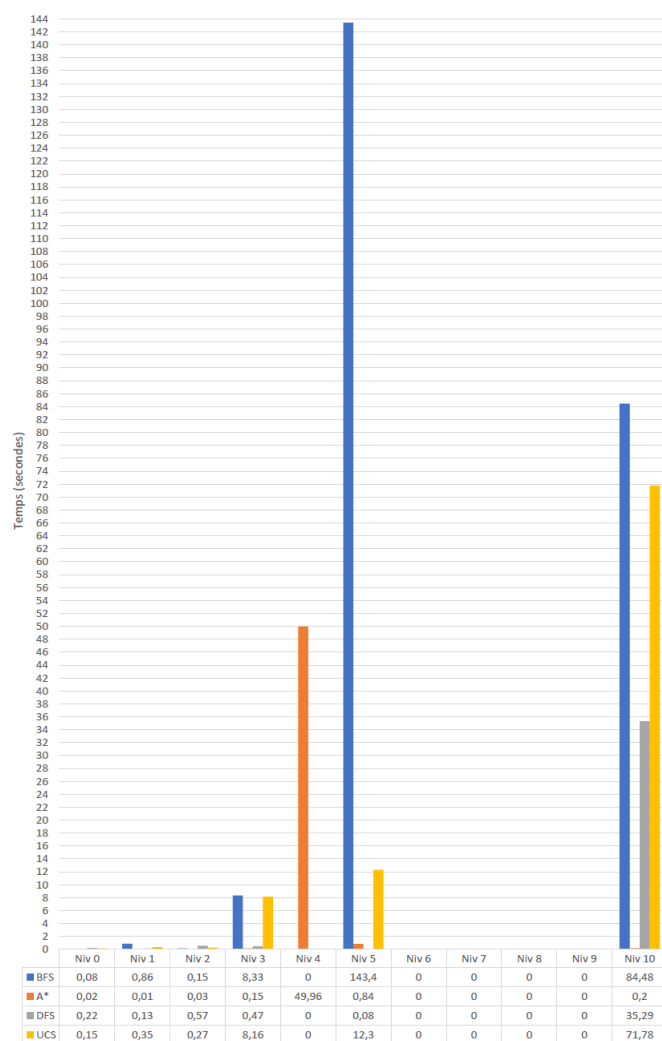


FIGURE 8 – Temps de résolution des niveaux en fonction des algorithmes utilisés

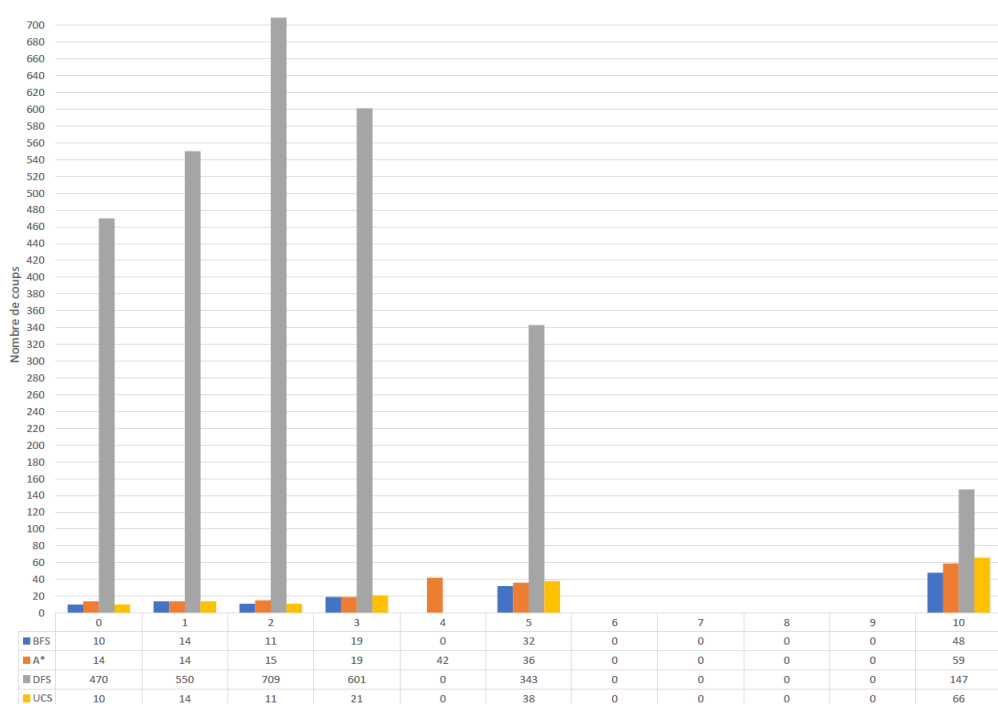


FIGURE 9 – Nombres de coups proposés pour résoudre les niveaux en fonction des algorithmes utilisés

D - Les diagrammes

a - Diagramme de classe Grille

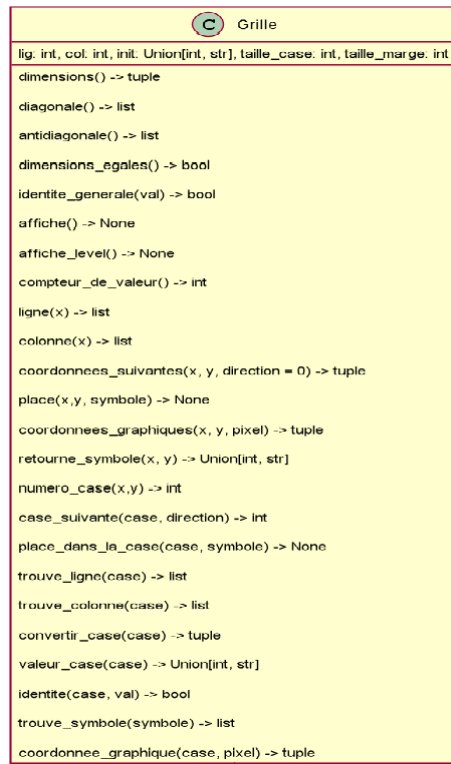


FIGURE 10 – Diagramme de classe Grille

b - Diagramme de classe Level

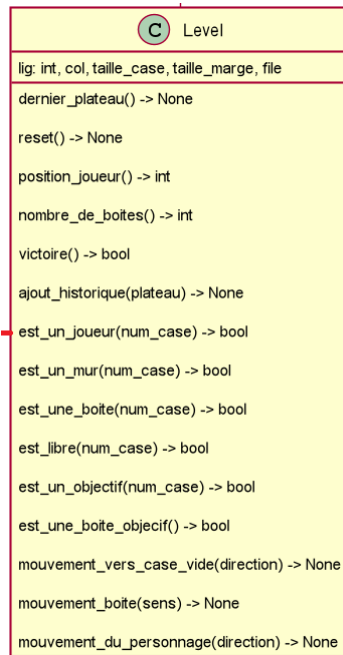


Diagramme de classe Level

c - Diagramme Image et audio

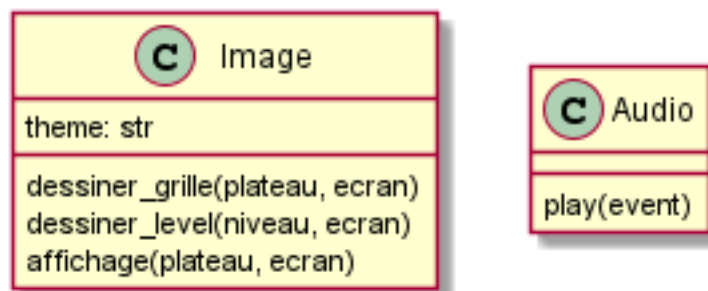


FIGURE 11 – Diagramme de Image et Audio

d - Diagramme couches

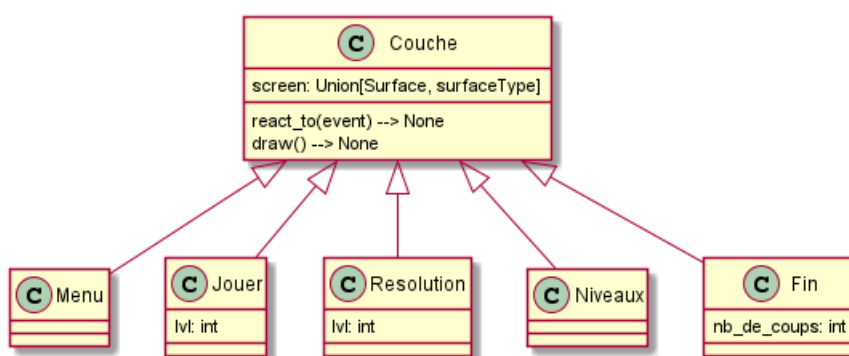


FIGURE 12 – Diagrammes des différentes couches

E - Sources

Voici la liste des sites consultés dans le cadre de la réalisation de notre projet :

- <https://github.com/kazantzakis/pySokoban> - GitHub - 23/10/2017 - kazantzakis
Il s'agit du GitHub de la personne qui a développé l'algorithme que nous avons adapté à notre projet.
- <https://fr.wikipedia.org/wiki/Sokoban> - Wikipédia - 08/03/2021
Page wikipédia décrivant les principes du Sokoban.
- http://www.sokobano.de/wiki/index.php?title=Main_Page - 23/06/2018
Wikipédia/Encyclopédie dédié au Sokoban réunissant de nombreuses informations sur le jeu et ses différentes possibilités.
- [https://fr.wikipedia.org/wiki/Algorithme_A*](https://fr.wikipedia.org/wiki/Algorithme_A%2A) - Wikipédia - 23/03/2021
Page wikipédia décrivant les principes de l'algorithme de résolution A*.
- https://fr.wikipedia.org/wiki/Algorithme_anytime - Wikipédia - 28/09/2017
Page wikipédia décrivant le principe d'un algorithme anytime en informatique.