

De Visch Justine
Deryck Olivier

Projet d'informatique
bac 1



Sokoban

Première
Bachelier
Science Informatique

Année 2016-2017

Table des matières

Table des matière.....	2
Intodution.....	3
Répartition des tâches.....	3
Explication du jeu.....	3
Le jeu.....	4
Description des choix personnels.....	4
Dans la classe Game.....	5
La classe RandomLvl.....	6
Fenêtres d'aide.....	8
Fenêtre de niveau bloqué.....	8
Fenêtre de choix des apparences.....	8
Les classes qui récupèrent les informations pour les utiliser dans les autres classes ...	9
La classe de tests unitaires.....	9
Points positifs de notre projet.....	10
Points négatifs de notre projet.....	10
Erreurs connues du programme.....	10
Apports positifs.....	11
Apports négatifs.....	11
Mini guide utilisateur.....	12
Conclusion.....	14
Bibliographie.....	15

Introduction:

1) Répartition des tâches :

Deryck Olivier	De Visch Justine
Jeu Sokoban de base Test Ant Aléa	Images Espace Graphique Skin
Le reste des classes est un mélange de nos travaux	

2) Explication du jeu :

Le jeu est décomposée des plusieurs parties :

New game : Lance une nouvelle partie

Continue : reprend la partie en cours à la dernière progression sauvegardée

Choose Level : Ouvre une fenêtre avec un choix des 10 niveaux de base que l'on peut jouer uniquement si ils sont débloqués

Random Level : Lance une partie avec un niveau généré aléatoirement.

Select Level : Ouvre une fenêtre qui demande à l'utilisateur le chemin vers le niveau qu'il souhaite jouer

Level Editor : Ouvre l'éditeur de niveau, dans lequel l'utilisateur peut créer et éditer ses propres niveaux

Skins : Ouvre une fenêtre contenant des boutons sur lesquels il suffit de cliquer pour changer l'apparence des objets du jeu

Exit : Permet de fermer le programme.

Le jeux

Nous avons débuté la conception du projet après visionnage d'un tutoriel, car au début nous n'avions pas assez de connaissances pour réaliser certaines choses. Des éléments sont donc inspirés de ce tutoriel, hélas les vidéos explicatives qui étaient sur youtube ont été effacées et il nous est impossible de retrouver la chaîne associée. Nous ne pouvons par conséquent pas mettre le lien dans la bibliographie du projet.

1) Description des choix personnels :

Toutes les images et visuels ont été réalisés par Justine de Visch. Par choix de personnalisation afin d'avoir des éléments uniques créés par nous. Cela nous évite également le risque de plagiat et réduit notre liste de sources.

Dans la classe Game :

Tout d'abord, les variables initialisées sont :

- 🖱 Le tableau du jeu , de taille 21x15 (affiché tel qu'étant 15x15 car nous avons décalé la grille de 6 cases en abscisse afin de centrer le jeu dans la fenêtre).
- 🖱 Les listes qui vont servir à contenir les informations des objets du jeu.
- 🖱 Les autres classes qui vont servir à récupérer des informations via des Getter et des Setter (Wall, Char, Box, Goal, Lvl) ainsi que les FileReader et FileWriter pour les manipulations dans les fichiers.
- 🖱 Les éléments graphiques (boutons, icône de la fenêtre de jeu, police de la chaîne de caractère).
- 🖱 Les entiers qui serviront à représenter le nombre de pas, ainsi qu'un booléen qui servira à savoir quand la touche SPACE est pressée.
- 🖱 La classe est composée de 3 constructeurs, la décision de surcharger le constructeur vient du fait que, le chargement des parties est géré en fonctions d'arguments, ou non, donc il a fallu plusieurs constructeurs pour pouvoir choisir le déroulement des actions en fonction du type de partie.

Selon le type de partie lancée (Nouvelle partie, continuer une partie, choisir un certain niveau, lancer un niveau ajouté et choisis par l'utilisateur, ainsi que pour les tests unitaires). On a décidé de si, oui ou non nous allons prendre des arguments en paramètre, ce qui n'est pas le cas lors du lancement d'une partie « classique » mais est le cas en autre temps.

La méthode LoadLevel() charge le niveau en fonction du type de partie, qui a été choisi par l'utilisateur lors de l'appui d'un bouton dans le menu.

- La partie correspond à une nouvelle partie, alors on charge le premier niveau.
- La partie correspond à une partie aléatoire, alors on charge le niveau aléatoire, qui aura été créé dans le constructeur, grâce à la classe générant un niveau aléatoire.
- La partie correspond à une reprise de partie, alors on charge les fichiers contenant le nombres de pas déjà effectués, l'état du niveau précédemment stoppé, et l'indice du niveau pour charger les prochains niveau. (une fois le niveau charger, on remet l'état du jeu comme lors du déroulement d'une partie classique afin de continuer normalement les niveaux) .
- Si le niveau a été choisi par l'utilisateur dans ses niveaux personnels, on le joue jusqu'à ce que la personne change manuellement de niveau.
- Si le niveau a été sélectionné depuis l'écran de sélection des niveaux, on charge le niveau sur lequel on a cliqué, puis on repasse l'état du jeu en partie classique afin de continuer normalement le jeu.

Pour charger les éléments d'un niveau, la classe lit dans le fichier en format .xsb l'apparence du niveau, remplit des listes contenant les coordonnées x et y associées à chaque élément du jeu, ainsi que remplir le tableau de jeu avec les éléments du fichier et le dessine ensuite avec la méthode paintComponent(). Après avoir chargé les éléments, on fait un CheckLevelDone() pour mettre les caisses en état valides si elles doivent l'être (car cette caractéristique est gérée dans cette méthode).

Dans la méthode checkCollision(), on vérifie les collisions de chaque objet avec un autre et on augmente le compte de pas lorsqu'on peut bouger:

- On parcourt d'abord la liste des murs, en vérifiant pour chaque mur l'intersection avec le personnage, si celui-ci entre en collision avec un mur, on décale ce premier contrairement à son mouvement effectué, afin d'annuler celui-ci.
- -On parcourt ensuite la liste des caisses, en vérifiant pour chacune de celles-ci si le personnage entre en collisions avec l'une d'elles. Si la case suivante la boîte bougée (en fonction de la direction de mouvement) est différente d'un mur ou d'une caisse, alors on peut bouger la caisse, sinon on fait en sorte d'annuler le mouvement. Dans cette méthode, on vérifie à chaque fois la collision entre les objectifs et les caisses, si les caisses ne sont pas sur un objectif alors on met l'état de la caisse sur « NORMAL »

La méthode checkTCollision() a pour fonction de vérifier les collisions dans le cas où le personnage tire les caisses, pour cela, le personnage n'avance pas quand on veut pousser une caisse, pareil pour celle-ci, on garde le même principe lors des collisions avec les murs, le personnage ne peut pas avancer. Par contre, lorsque un personnage n'entre pas en collision avec une caisse, alors il peut bouger, et si la case avant le personnage, avant son mouvement, est une caisse, alors le personnage et la caisse avancent afin de créer ce mouvement de traction.

La méthode CheckLevelDone() vérifie si le niveau est fini ou non, en comparant le nombres de caisses en état valide au nombre d'objectifs, si ceux-ci sont identiques, alors on passe au niveau suivant (dans le cas d'une partie classique), si on lance une partie de

niveaux aléatoires, alors on joue des niveaux aléatoires jusqu'à la décision d'arrêter. Dans le cas d'un niveau choisis par l'utilisateur, on relance ce même niveau si l'utilisateur ne le change pas. Quand le niveau est fini, on remet le compteur de pas du niveau à 0 tandis que le compteur de pas total continue d'augmenter.

La méthode autosave() sauvegarde automatiquement la progression de la partie, elle est lancée à chaque mouvement. Cette méthode lit l'état du niveau, c'est à dire qu'elle lit les éléments stockés dans le tableau du jeu, et récupère l'élément pour chaque emplacement dans le jeu, les réécrit dans un fichier de sauvegarde nommé « autosave.xsb » situé dans le dossiers des « Maps ». Il récupère aussi l'indice du niveau, ainsi que les nombres de pas du niveau actuel ainsi que les pas totaux dans des fichiers afin de pouvoir les recharger par la suite pour reprendre la progression.

La méthode ReplayLevel() sauvegarde les mouvements qui ont été effectués pour résoudre un niveau, elle permet ainsi de pouvoir regarder les déplacements faits pour atteindre la fin du niveau.

Enfin, les déplacements sont définis dans la méthode KeyReleased(), qui effectue des actions lors de la pression sur certaines touches du clavier. A chaque mouvement, la direction du personnage change(pour définir l'image de direction, entre autre), celui-ci se déplace en fonction de la direction et fait un checkCollision(ou checkTCollision lorsque la touche SPACE est pressée) qui permet de rendre les mouvements valides. On ajoute aussi les mouvements effectués à une liste, qui est écrite dans un fichier de sauvegarde des mouvements effectués grâce à la méthode ReplayLevel() expliquée précédemment. La touche R permet de recharger le niveau joué, N lance un nouveau niveau aléatoire dans le cas d'une partie aléatoire. Après chaque mouvements, on appelle un repaint() pour redessiner le jeu, CheckLevelDone ainsi quand l'autosave si le niveau est en état d'une partie classique (non aléatoire et non choisis par le joueur).

La méthode Automove() simule une suite de mouvement au personnage, elle est utilisée lors de l'application d'une séquence de mouvement .mov à une fichier de niveau .xsb, donc utilisée lors des simulation de niveau dans le terminal ou les unitaires.

La classe RandomLvl :

Cette classe génère un niveau aléatoire, jouable.

Pour générer un niveau aléatoire et jouable, nous avons procédés par étapes, comme suit :

- 👤 Premièrement, on commence par générer une grille, par les murs de côté, puis en remplissant l'intérieur de la grille par les autres éléments.
- 👤 Ensuite, en appliquant une suite de mouvement du personnage qui appliquera la méthode identique au CheckTCollision de la classe Game (c'est à dire le personnage qui tire les caisses plutôt que les pousser).
- 👤 On sauvegarde finalement l'état du niveau créé et mélanger dans un fichier qu'on jouera quand on sélectionnera une partie aléatoire.

Pour générer la grille de façon aléatoire et de façon à ce qu'elle reste jouable et fermée(pour éviter que le personnage ne sorte du plateau de jeu), on a procédé par une génération par

coté, c'est à dire, un mur sur le côté droit, un sur le côté gauche, un en haut et enfin un en bas.

Pour créer un mur sur un côté, on a pris une cordonné proche du bord, imposé un maximum (qui est le bord du plateau) un minimum (qui est le bord d'une zone centrée au milieu, qui restera vide afin de générer les autres éléments à l'intérieur).

Le coté subit aléatoirement une variation de positive ou négative ou reste constant sur l'axe des ordonnées pour les murs supérieurs et inférieurs au plateau, ou rester sur le même axe (généralisé avec une méthode Random) tandis que les murs des deux extrémités peuvent augmenter de un, diminuer de un ou encore rester constant sur l'axe des abscisses.

Après cela, on obtient les quatre cotés qui se rejoignent entre eux pour se fermer, tout en laissant une zone toujours vide au milieu.

Ensuite on génère un nombre de caisses (valides, sur un objectif) aléatoire entre 2 et 5, et on génère tour à tour des coordonnées pour chacune d'elles, en la posant, si la case à l'emplacement désiré est vide.

Pareillement, on génère des coordonnées aléatoires pour le personnage, jusqu'à lui avoir trouvé un emplacement libre, sur lequel on le posera.

Après, on parcourt chacune des cases du plateau, on vérifie si elle est libre, et si oui, on génère un mur avec une probabilité de 1/9.

Enfin, on sauvegarde une première fois le plateau obtenu dans un fichier.

Donc une fois notre niveau aléatoire obtenu, nous devons le mélanger avec la méthode de tirage de caisses.

On génère alors une chaîne aléatoire de mouvement, d'un nombre suffisamment élevé afin de pouvoir bien mélanger le niveau, laquelle on stocke dans un fichier.

On parcourt le fichier contenant la liste de mouvements, élément par élément, en déplaçant le personnage à chaque itération de la direction indiquée par cet élément lu.

Chaque déplacement applique la vérification de collision, avec la traction de caisses plutôt que la poussée de celles-ci. De cette façon, les caisses ne se retrouvent jamais coincées dans un coin, et le niveau reste donc toujours jouable.

Finalement, on sauvegarde encore une fois le résultat dans un fichier, celui-là même qu'on lira dans le cas d'une partie sur des niveaux aléatoires.

Pour le générateur de niveau, nous avons fait un Jpanel, qui contient un plateau de taille identique à celui du jeu, ainsi que des boutons qui servent à sélectionner les objets à placer. On a donc le constructeur, dans lequel on construit nos boutons aux emplacements et aux images correspondantes.

Les méthodes de souris, lorsqu'on bouge la souris, on redessine le jeu afin d'actualiser l'image au curseur, et le clique de la souris, qui ajoute l'objet avec le clique gauche ou l'enlève avec le clic droit.

Lors de l'ajout d'un objet dans le plateau, on l'ajoute également dans un tableau représentant l'éditeur.

Ensuite, la méthode paintComponent dessine l'image de fond ainsi que les éléments aux coordonnées correspondantes.

Enfin, nous avons les méthodes d'appel au clavier, pour les raccourcis de sauvegarder et de charge, respectivement S et L (pour save et load) :

- Dans la sauvegarde, on parcourt les éléments du tableau de jeu, et on récupère pour chaque case l'élément correspondant, qu'on va écrire dans un fichier spécifié par l'utilisateur.
- Dans la charge de niveau, on demande à l'utilisateur le niveau qu'il désire charger et modifier, on lit ensuite le fichier et parcourt chaque élément dans celui-ci. Pour chaque élément, on écrit la valeur correspondante dans le tableau de l'éditeur qu'on chargera dans la méthode de paint.

Les défauts de ce générateur sont que parfois après mélange, une caisse s'effaçait ou alors le fichier de réécriture est vide, on a pallié à ce problème en posant dans les conditions de la classe Game que lorsqu'on génère un niveau aléatoire, si le nombre de caisses diffères du nombre d'objectif, on relance la génération aléatoire, même processus lorsque le nombre de caisse est nul, ce qui veut dire que le fichier sera vide.

Ainsi nous nous débarrassons des seuls erreurs que nous avons.

Fenêtres d'aide :

Dans l'éditeur de niveau ou en partie, se trouve un bouton d'aide qui ouvre une fenêtre contenant une image indicative résumant les actions possibles dans la fenêtre actuelle. Cette fenêtre est créé dans la classe Help et est un Jdialog. Ce Jdialog est modal et dépend de la fenêtre mère (GameFrame), ce choix est dû au fait qu'après quelques recherches, il apparaît que une Jdialog modal permet de ne pas perdre le focus sur la fenêtre principal. Le contenu de ce Jdialog est simple, c'est une image que l'on dessine en fonction de l'état de jeu donné, soit un niveau aléatoire, soit un niveau normal, soit l'éditeur de niveau. L'image affichée correspond à l'état du jeu voulu.

Fenêtre de niveau bloqué (lorsqu'un niveau n'a pas encore été joué et n'est pas sélectionnable dans la sélection de niveau) :

Cette fenêtre est une simple Jframe affichant une image qui explique que le niveau n'a pas encore été débloqué est qu'il faut réaliser les niveaux précédents pour pouvoir le jouer. Les conditions pour ouvrir cette fenêtres sont situées dans la classe de fenêtre principale GameFrame. Pour chaque bouton du panel de choix de niveau, on lit le fichier de sauvegarde pour vérifier si le niveau maximum sauvegardé est supérieur ou égal au niveau que l'utilisateur souhaite jouer.

La fenêtre de choix des apparences :

Skin est la classe qui crée le panel qui comporte les éléments du menu de choix d'apparences du jeu, elle possède uniquement des boutons sur lesquels on va pouvoir cliquer pour sélectionner les apparences voulues, ainsi qu'un fond dessiné en arrière-plan. Par manque de temps, le « fond » des skins n'a pas été ajouté car il devait être implémenté d'une autre manière que les classes (Box,Goal,Wal,Char).

Tous les ActionListeners des boutons des différents panels sont gérés dans la classe de la fenêtre principale GameFrame.

Les classes qui récupèrent les informations pour les utiliser dans les autres classes :

- 🐼 LvlInfo: Cette classe possède 4 méthodes , deux qui servent à récupérer des informations (le type de partie ainsi que le niveau joué (pour la sauvegarde)) ainsi que deux qui servent à les définir.
- 🐼 NewSkin : Cette classe permet de récupérer et de gérer les informations relatives aux apparences.
- 🐼 Box , Char, Goal, Wall: Ces classes servent à récupérer et gérer les informations relatives aux objets en jeu; les coordonnées et les images (qui varient en fonction du thème d'apparence choisi).

La classe de tests unitaires (GameTest):

Dans cette classe, nous exécutons les tests unitaires, pour cela, nous appliquons une suite de mouvements sur un fichier de niveau.xsb et nous sauvegardons le résultat dans un autre fichier .xsb, nous comparons ensuite le résultat avec un fichier contenant l'état du niveau attendu (lui aussi en format.xsb).

Pour comparer deux fichiers entre eux, il a fallu créer une méthode qui compare les fichiers entre eux et renvoie un booléen true/false si les fichiers correspondaient ou non.

La méthode qui effectue cette vérification est nommée equalFiles et prend en entrée les deux fichiers à comparer. Il lit chacun des fichiers lignes par ligne jusqu'à les avoir parcouru entièrement. Si toutes les lignes sont identiques, la méthode retourne un equal qui a pour valeur true, sinon elle sera false.

Nous avons trouvé cette méthode grâce à quelques recherches car nous n'arrivions pas à comparer efficacement les fichiers (voir bibliographie).

Ensuite vient les tests, nous en avons fait cinq , chacun des test simule une partie en exécutant la classe Game avec 3 arguments ; le niveau de départ(input), la chaîne de mouvements(moves), ainsi que le niveau après application de la chaîne de mouvements (output).

Il vérifie ensuite l'égalité entre le fichier après l'application des mouvements (output) et le fichier contenant l'état de jeu attendu (expected).

Le résultat est vérifié avec un assertTrue(equalFiles(expected, output) qui retourne true si les fichiers correspondent et false si ils ne correspondent pas.

Nous avons créé 5 tests unitaires :

- 🐼 Dans le premier, le personnage pousse une caisse sur un emplacement libre.
- 🐼 Dans le second, le personnage se déplace sur une case libre, sans rien déplacer.
- 🐼 Dans le troisième test, le héros se déplace contre un mur (pour vérifier la collision avec celui-ci, qu'il ne puisse pas le traverser).
- 🐼 Dans l'avant dernier, le personnage pousse une caisse contre un mur, vérifiant ainsi la collision entre les caisses et les murs.

- Dans le dernier test, une caisse est poussée sur un objectif et le personnage finit ses déplacements sur un objectif, ainsi nous vérifions l'état des deux objets lors de leur collisions avec un objectif.

2) Points positifs de notre projet :

- L'apparence générale qui est plutôt soignée et qui offre une sélection de thèmes diverses via un menu de choix.
- La sauvegarde automatique, qui se fait à chaque mouvement et qui ne sauvegarde que lorsque la progression dépasse la progression sauvegardée, l'utilisateur n'a donc pas à se soucier de sauvegarder par lui-même ou d'avoir peur de perdre sa progression lorsqu'il joue un niveau antérieur. (La progression est toutefois écrasée lorsqu'on lance une nouvelle partie).
- Un compteur de nombres de pas qui indique à l'utilisateur le nombre de mouvements effectués.
- Les fenêtres d'aide qui explique de manière agréable le fonctionnement du jeu au lecteur.
- Une interface facile d'utilisation.
- La possibilité de créer et éditer des niveaux personnels d'une façon simple, ainsi que pouvoir les jouer.
- Etant donné que l'on a rajouté le fait de trirer un caisse. Le jeu est toujours jouable même lorsque l'on coinse la caisse.

3) Points négatifs de notre projet :

- Programme peu optimisé dans l'ensemble, qui peut allonger les temps d'exécution, par exemple.
- Certaines erreurs non réglées mais juste contournées.
- Pas d'écran ou d'interactions de fin de partie.
- Fenêtre de taille non redimensionnable.
- Certains points de programmation vus au cours non exploités.
- Dans l'éditeur de niveaux, l'utilisateur doit vérifier si le niveau est jouable ou non par lui-même.

4) Erreurs connues du programme :

- Dans le menu de choix des niveaux de base, lorsqu'on souhaite jouer un niveau encore verrouillé, une fenêtre contenant un message d'erreur s'ouvre. Toutefois cette fenêtre ne possède pas toujours la taille attendue qui devrait être adaptée à l'image.
- Lors de la création des niveaux aléatoires, le fichier où est sauvegardé le niveau aléatoire est parfois vide ou se retrouve avec une caisse manquante, pour pallier à ce bug, nous avons posé une condition dans le déroulement des parties aléatoires :
 - Lorsqu'on crée le niveau, si celui-ci possède un nombre de caisses nul ou différent du nombre d'objectifs, alors on relance la création jusqu'à obtenir un niveau correct (l'erreur n'est pas fréquente mais présente) La vérification se fait avant de jouer le niveau.
 - Dans le déroulement des parties aléatoires, il se peut que lorsqu'on lance un niveau, celui-ci soit déjà validé, alors on augmente le compteur de niveaux terminés sans pour autant que ce soit l'utilisateur qui l'aie réalisé.

- Dans la sélection de niveau par l'utilisateur, lorsque celui-ci ne rentre aucune information ou que le niveau qu'il veut jouer n'existe pas, ou qu'il ferme la fenêtre d'entrée le programme produit une erreur de fichier inexistant ou de fichier null, toutefois cette erreur n'est pas visible pour le joueur et il n'a qu'à entrer un autre nom de niveau. Cette erreur est une erreur normale et gérée par le programme, qui renvoie une erreur lorsque le fichier entré n'existe pas.
- Dans l'éditeur de niveau, lorsqu'on essaye de poser un objet en dehors du tableau de jeu, le programme rejette un erreur « `outOfBoundException` » car on ne peut forcément pas poser un objet hors des limites du jeu. Cette erreur est donc normale et n'a aucune incidence sur l'utilisation de l'interface par l'utilisateur.

5) Apports positifs:

- Étant notre premier projet d'informatique, il nous a permis de concrétiser les aspects appris lors du cours afin de se rendre compte de l'utilisation de ceux-ci, cela nous a permis d'améliorer notre compréhension de la programmation en générale, de plusieurs concepts tels que les héritages, les implémentations, les méthodes de classes et bien d'autres, qui restaient généralement assez flou après un cours ou un TP, mais qui se sont vite précisés après l'utilisation répétitive de ceux-ci.
- Notre esprit logique s'est aussi amélioré à force de réfléchir à chaque cas possible, d'essayer de remplir chaque condition, de les écrire sans en oublier au risque de devoir tout revérifier, de prévoir chaque erreur et d'en régler d'autre, bref une amélioration de notre logique en elle-même.
- Le projet nous a également permis de nous familiariser avec les environnement de développement tel qu'éclipse, qui a été notre outil principal lors de l'écriture de notre projet ainsi que ant qui nous a servi pour les tests unitaires, on saura donc l'utiliser pour nos projets futurs.
- Le projet fut dans l'ensemble plaisant à mettre en place, parfois certes agaçant lorsqu'on arrivait pas à accomplir un objectif mais tout aussi plaisant lorsque l'on y arrivait.
- Ce fut le premier travail de groupe réel(hormis les TP) à l'Université pour notre part, on a donc appris à travailler en conséquence, en répartissant le travail et l'accomplissant pour éviter de décevoir notre binôme.
- Beaucoup de recherche était nécessaire pour ce projet, dans la documentation Java ou alors sur divers forums expliquant des points que nous n'avions pas bien compris, ou expliquant des méthodes ou principes restés flous, on a donc appris à rechercher efficacement des éléments.





6) Apports négatifs :

Le projet prend beaucoup de temps, principalement quand on approche de la fin, et qu'un test dans une autre matière est prévue aux environs de la date butoir du projet, alors on préfère finir celui-ci plutôt que de travailler de manière approfondie sur d'autres matières.

7) Mini guide utilisateur :

En lançant, le jeu s'ouvre le menu principal :




Dans ce menu principal se retrouvent les boutons suivants :

-  New game, qui lance une nouvelle partie au niveau 1 en écrasant les derniers fichiers de sauvegarde, c'est à dire le compteur de pas, les anciens états du jeu et le niveau maximum atteint.
-  Continue, qui reprend la partie où elle en était, au dernier niveau joué, au nombre de pas effectués et dans l'état de partie sauvegardé.
-  Choose Level, qui ouvre un nouveau panel avec l'affichage des 10 niveaux de base que le l'utilisateur ne peut lancer que lorsqu'il a débloqué les niveaux précédents lors de sa partie normale.
-  Select Level, qui demande à l'utilisateur quel niveau il souhaite jouer, depuis les niveaux qu'il a créé et qui sont stockés dans le fichier « Maps/UserLvl/ ».

Dans chacun des trois boutons cités précédemment, l'utilisateur lance une partie aux caractéristiques normales, il peut donc utiliser les flèches directionnelles du clavier pour déplacer son personnage et pousser des caisses. En maintenant la barre espace appuyée lors d'un déplacement, le personnage ne pousse pas les caisses mais les tire.


La progression est sauvegardée automatiquement à chaque mouvement si le niveau n'est pas un niveau antérieur à la dernière sauvegarde.

Dans la fenêtre de jeu classique, se trouvent les boutons :

-  Help, qui ouvre une fenêtre contenant les explications des touches du clavier.
-  Menu, si l'utilisateur souhaite retourner au menu principal.
-  Exit, si celui-ci souhaite quitter le jeu.

Se trouvent à gauches, les informations suivantes :

- Le niveau que l'utilisateur est en train de jouer.
- Le nombres de pas effectués dans le niveau.
- Le nombres de pas effectués dans la partie.

-  Random Level, qui génère un niveau de taille et au nombres de caisses et murs aléatoire.

Ce mode de jeu contient presque les mêmes informations que les modes cités précédemment :

- Le nombre de niveaux aléatoires déjà finis.
- Le nombre de pas du niveau actuel.
- Le nombre de pas de la session de jeu de niveaux aléatoires.

Les boutons suivants :

- Menu, qui permet aussi de retourner au menu principal.
- Exit, pour quitter le jeu.
- Le bouton de fenêtre d'aide, qui possède une image différente pour préciser l'utilisation de raccourci « N » pour relancer un niveau lorsque celui-ci ne plaît pas.

- 👤 Editor, qui ouvre l'éditeur de niveau, dans lequel l'utilisateur peut librement créer son niveau dans le plateau de jeu disposé, sauvegarder ses propres niveaux et charger des niveaux déjà conçus afin de les modifier.

L'éditeur possède une fenêtre d'aide qui explique la procédure de création de niveau : Il suffit de cliquer sur les boutons à gauche afin de sélectionner l'objet à placer sur le plateau, ensuite il faut cliquer à l'emplacement voulu pour déposer l'objet. Si l'on se trompe lors de l'emplacement de l'objet, il suffit de faire un clic droit sur celui-ci afin de le supprimer.

Une fois le niveau créé, on peut le sauvegarder grâce au raccourci « S » ou alors charger un niveau pour le modifier grâce au raccourci « L ».

On peut également effacer le contenu du plateau grâce au raccourci de clavier "R". L'éditeur possède aussi les boutons Menu et Exit ayant les mêmes fonctions que précédemment expliquées plus haut.

- 👤 Exit, qui ferme le jeu.

Conclusion





Le projet fut agréable à réaliser, on a eu l'occasion d'approfondir nos connaissances, en réalisant des recherches et en s'entraînant par la pratique.

Le projet, malgré un thème imposé nous laisse libre cours à notre imagination, en incluant des bonus et une personnalisation tant dans le code que dans les apparences.

Au début lors de la réception de l'énoncé du projet, nous n'avions pas les connaissances nécessaires pour faire ce projet, et les connaissances (en suivant les TP effectués en séance) n'arrivent que proches de la date buttoir, ce qui peut poser problème lorsqu'on bloque sur une partie de la réalisation du programme.

Dans l'ensemble, malgré l'empressement en approche de la date finale et la nécessité de mettre certains cours de côté pour réussir à le finir, le projet fut amusant et instructif.

Bibliographie

-  Certains éléments inspirés de tutoriels sur une chaine youtube (malheureusement supprimée, si les vidéos sont nécessaires, nous les avons enregistré mais le lien de la chaine n'est plus accessible).
-  La méthode qui compare deux fichiers entre eux pour les tests unitaires : <http://stackoverflow.com/questions/466841/comparing-text-files-w-junit>
-  De nombreuses recherches notamment dans la documentation de java ou sur des forums expliquant certains éléments (points de matière ou méthodes) de java.
-  Graphismes inspirés de diverses images trouvées sur le net. (Toutefois ce ne sont que des inspirations) (Se trouvent dans le fichier « Pictures/others »)