**Speedy Roadie**

Louis Dhanis, Corentin Dachy

Projet d’informatique, mai 2017

Table des matières

[1. Répartition du travail 1](#_Toc482698801)

[2. Divers choix personnels 1](#_Toc482698802)

[ Thème du jeu et mode histoire 1](#_Toc482698803)

[ Swing ou JavaFX? 2](#_Toc482698804)

[ Package Backend: Implémentation d'une partie 2](#_Toc482698805)

[ Package Frontend: la GUI (Graphic User Interface) 8](#_Toc482698806)

[3. Points forts 9](#_Toc482698807)

[ Sauvegardes en continu et pendant la partie 9](#_Toc482698808)

[ Lecture de .mov en mode cinématique 9](#_Toc482698809)

[ Mode histoire 10](#_Toc482698810)

[ Réinitialisation de la partie 10](#_Toc482698811)

[4. Faiblesses 10](#_Toc482698812)

[ Mode histoire 10](#_Toc482698813)

[ Apparence du code 10](#_Toc482698814)

[5. Erreurs connues 11](#_Toc482698815)

[ Lecture de .mov en mode cinématique 11](#_Toc482698816)

[ Exécution sans Apache ant et ressources graphiques 11](#_Toc482698817)

[6. Apports positifs de ce projet 12](#_Toc482698818)

[7. Guide utilisateur de Speedy Roadie 12](#_Toc482698819)

[ Ant 12](#_Toc482698820)

[ Menu d'accueil 13](#_Toc482698821)

[ Mode Histoire 13](#_Toc482698822)

[ Mode aléatoire 14](#_Toc482698823)

[ Charger une partie 14](#_Toc482698824)

[ Continuer une partie en cours 14](#_Toc482698825)

[ Fonctionnement d'un niveau 15](#_Toc482698826)

[ Fin de partie 15](#_Toc482698827)

[ Générer l'output d'un fichier .xsb et .mov 16](#_Toc482698828)

[8. Remerciements 16](#_Toc482698829)



1. Répartition du travail

Afin de se répartir les tâches équitablement et selon les préférences de chacun, Corentin Dachy s’est occupé de la partie algorithmique back-end du travail tandis que je me suis attelé à la partie graphique du Sokoban.

Pour faciliter la tâche de l'implémentation, nous avons décidé de diviser le travail en deux packages: le Backend (géré par Corentin) et le Frontend (que j'ai implémenté)

Nous avons divisé le travail de la sorte pour ne pas interférer sur les classes de chacun. En plus d'exploiter un concept de Package, nous avons dû apprendre à travailler en binôme, mais cela sera expliqué plus amplement dans le point des apports de ce projet.

1. Divers choix personnels

Divers choix ont ponctué le développement de notre jeu. Tout du long, nous avons discuté les différents aspects de la mise en œuvre de chaque élément. Dans cette section nous allons aborder les choix les plus importants que nous avons effectués pour ce projet.

* Thème du jeu et mode histoire

En tant qu’amoureux de l’univers de la scène metal, nous avons vu en ce projet un moyen de raconter une histoire. En nous concertant nous nous sommes dit que le sokoban pouvait représenter une scène où un roadie (personne qui travaille pour un groupe et qui aide à l’organisation de la scène) nommé Speedy, devrait déplacer des caisses contenant des instruments de musique pour les ranger afin rétablir l’ordre des choses dans le metal, ce dernier ayant été corrompu par un circle pit maléfique.

A plusieurs reprises dans ce document, nous allons parler de mode Classic et de mode Histoire alors que ces deux modes sont en fait la même chose. Pour ce projet nous devions créer une liste de dix niveaux progressivement difficiles et s'enchaînant un à la suite de l'autre. Nous avons d'abord appelé ce mode comme le mode Classic (à contrario avec mode aléatoire ou encore le mode où le joueur peut charger une partie personnalisée) pour ensuite avoir l'idée d'y insérer une véritable histoire, d'où son deuxième nom le mode Histoire.

L’interface graphique devait suivre avec le thème. Corentin a donc demandé à <INSERER NOM DE TON AMI>, un graphiste doué dans le pixel art, de dessiner les différents sprites (éléments graphiques) du jeu.

* Swing ou JavaFX?

Nous avons eu une séance d'information à propos des interfaces graphiques à une des séances d'information du projet. A cette séance, on nous a expliqué qu'il était intéressant de préférer JavaFX à Swing pour la raison suivante: Swing est en fin de développement, cette bibliothèque graphique ne sera plus mise à jour.

Nous avons tout de même choisi Swing car son utilisation ne nous était pas étrangère et ce projet ne nécessitait pas tout ce qu'offre JavaFX (entre autres l'utilisation du CSS ou la 3D)[[1]](#footnote-1).

Nous avons donc préféré ne pas consacrer trop de temps à l'apprentissage d'une nouvelle technologie qui n'aurait pas eu plus d'utilité pour ce projet qu'une technologie que l'on savait utiliser mais qui allait ne plus être mise à jour.

* Package Backend: Implémentation d'une partie

Cette partie porte sur les choix de l'implémentation d'un puzzle, indépendamment du contexte de celui ci (niveau importé ou généré pour être joué dans l'un de ces mode, ou niveau du contexte du mode histoire) et de la représentation(visuelle)/manipulation de celui ci (à quoi ressemble l'interface graphique et comment l'utilisateur va s'en servir).

Vous y trouverez donc des explication sur l'implémentation complète ainsi qu'une petite discussion sur les choix principaux effectués.

Explication de l'implémentation:

Toute les information sur une partie devront se retrouver dans une instance de la classe Game, et modifier une partie devra se faire part l'appel de méthodes publiques de son instance de Game.

Obtenir des informations sur une partie donnée et modifier l'état actuel de cette partie (de part les mouvements autorisés) se fera donc via manipulation de l'instance de Game qui lui est associée. (Cette instance de Game sera donc manipulée par l'interface graphique qui appellera ses méthodes pour savoir quoi représenter et pour tenter de modifier son état)

Pour se faire, la classe Game possède deux variables: board et nbSteps. board est une variable de type Board, classe qui sera décrite plus bas. Elle représente tout ce qui est relatif au positionnement actuel des éléments du puzzle. Toute information non relative aux positions actuel des éléments du puzzle devront se retrouver dans d'autres variable de Game.

Techniquement il ne reste que le nombre de pas effectués depuis l'état initial du puzzle, représenté par le variable nbSteps de type int.

D'autres informations auraient pu être représentée ici, tel que le temps écoulé depuis le début de la partie. Cependant, cette fonctionnalité n'a pas été implémentée et nbSteps reste la seule variable de Game qui n'est pas directement lié à la position actuelle des éléments du puzzle.

Ainsi, Game devra disposer des méthodes publiques permettant de lire et de modifier son contenu. Pour citer les plus importantes:

-movePlayer(int x,int y) pour déplacer le joueur

-getNbSteps() pour obtenir le nombre de pas effectués depuis l'état initial

-getRepr() pour obtenir, sous forme d'un char[][], la disposition actuelle est éléments de la partie

-isGameWon() pour obtenir un booléen indiquant si la partie et gagnée (et donc terminée) ou non

Toute ces méthodes à l'exception de getNbSteps appellerons des méthodes de la variable board. De plus, toute ces méthodes à l'exception de movePlayer se contente d'appeler des méthodes du même nom de board et de renvoyer le résultat de ces dernières. La différence avec movePlayer étant que cette méthode incrémente nbSteps si le déplacement a bien été effectué.

C'est donc dans board que le positionnement concret des éléments d'une partie sera implémenté.

Mais avant, voyons ce que sont les éléments d'une partie.

Ils ont tous en commun le fait d'implémenter l'interface Layoutable. Chaqu'un de ces éléments possèdera donc les méthodes getType() (renvoyant un string relatif à son type) et toChar() (renvoyant un char relatif à son type). Une seule des deux méthodes aurait suffit mais l'utilisation de l'une nous sembla plus commode dans un cas et inversement dans l'autre (les deux cas étant l'envoit d'information à l'interface graphique via getRepr et le déplacement des éléments vie movePlayer).

Ces éléments sont: ClassicBox, Goal, EmptyCase et Player. Tous n'ont en réalité que l'implémentation des méthodes toChar() et getType() imposé par l'implémentation de Layoutable. A l'exception de Goal qui a la particularité de contenir un autre Layoutable. Ainsi, une instance de Goal contiendra donc une instance d'EmptyCase, de Player ou de ClassicBox. Ses méthodes toChar() et getType() donneront des réponses différentes en fonction de l'objet contenu, et Goal comporte d'autres méthodes permettant de changer et d'obtenir son contenu. De plus, Goal implémenté également l'interface Objectif et possède donc une méthode isCompleted() renvoyant un booléen (true si l'objectif est completé, false sinon)

Ainsi, le classe Board devra non seulement contenir des éléments Layoutable mais aussi les organiser entre eux (les éléments Layoutable n'ayant pas d'information sur leurs positions).

Cela se fait via une ArrayList d'ArrayList, de Layoutable, formant la variable tab.

La totalité de l'information de Board est représenté dans tab. Ceci dit, Board contient d'autres paramètre afin de réduire la complexité de certaine de ses méthodes.

Ainsi Board comporte également deux variables de type int, pX et pY, représentant les coordonnées X et Y du joueur (concrètement, une instance de Player devrait être retrouvé à tout moment en tab.get(Y).get(X)). Cela est fait pour ne pas devoir parcour tout tab à chaque déplacement pour retrouver la position du joueur (tout déplacement étant fait autour du joueur).

Board contient également la variable objectives, une ArrayList d'Objectif. Cette variable contient des références vers tout les Goal du jeu (chaque Goal sera donc référencé dans Board deux fois, une fois dans objectives et une fois dans tab) servant encore une fois à ne pas devoir parcourir tout tab lorsqu'une méthode est appelé: il s'agit ici de la méthode isGameWon() qui demandera à tout les objectifs (tous les Goal) si ils sont completé (avec isCompleted() vu que les Goal implémenté Objectif). Étant donné que la méthode isGameWon() sera appelé (par l'interface graphique) après chaque mouvement, la nécessité d'avoir des références directes vers les Objectif (les Goal) -plutot que de devoir les chercher, noyés dans tab- devient évidente.

Ainsi les méthodes en suspend getRepr(), isGameWon() et movePlayer() deviennent assez évidente.

-getRepr() renvoyant un tableau de char de dimensions égale à celles de tab (et appelant la méthode toChar() de chaque Layoutable composant tab)

-isGameWon() regardera chaque Objectif de objectives. Il renverra est completé (chaqu'un renvoyant true via isGameWon()) et false si au moins un objectif n'est pas completé (isGameWon() renvoyant false pour au moins un objectif)

-movePlayer(int x, int y) est un peu plus compliquée si on regarde les détails, mais sa philosophie est en réalité à peine plus compliquée que celle des méthodes précédente: cette méthode prend un vecteur unitaire (x,y) (exception générée si cette précondition n'est pas respectée! Ce qui ceci dit ne devrait jamais arriver vu que c'est l'interface graphique qui fait appel à cette méthode) représentant la direction du déplacement du joueur et modifier tab autour du joueur (qui est directement retrouvé grace à pX et pY qui stocke en permanence sa position). Par ailleurs cette méthode a une valeur de retour servant essentiellement à avertir l'instance de Game qui appellera cette méthode si un déplacement a été effectué ou non (pour que l'instance de Game sache si elle doit, ou non, implémenter nbSteps).

Board contient bien d'autres méthodes qui ne seront pas expliquées ici en détails. (Voir javadoc pour plus de détails).

Je me contenterais seulement de citer les plus importantes de cette catégorie :

-La méthode reverseMovePlayer(int x,int y), qui fait avancer le joueur avec toute les règles de déplacement (de caisse) inversées. déplaçant le joueur "à l'envers" (il ne peut que tirer des caisses, pas les pousser). Cette méthode ne servira quand dans génération de niveaux aléatoires et ne devra jamais être appelée une fois un niveau créé. C'est donc sans surprise que cette méthode est protected

-La méthode generateBoard(ArrayList<char[]> puzzleData) ainsi que les constructeurs de Board. generateBoard(ArrayList<char[]>) remplit les information de Board (tab mais aussi objectives, pX et pY) à partir de la variable avec laquelle il est appelé. Les constructeurs de Board appellent tous generateBaord. Soit le constructeur est appelé avec une ArrayList<char[]> et passe directement à generateBoard, soit il est appelé avec le chemin d'un fichier xsb contenant l'information à envoyer à generateBoard, auquel cas le constructeur réalise une étape de plus qui est la transcription de ce fichier xsb en ArrayList<char[]> (travail que délègue à la méthode readBoard(String path) qui se trouve dans le PuzzleDataManager, une classe boite à outils contenant plein de méthodes (toute static) relatives à la lecture et écriture de fichiers en rapport avec notre projet)

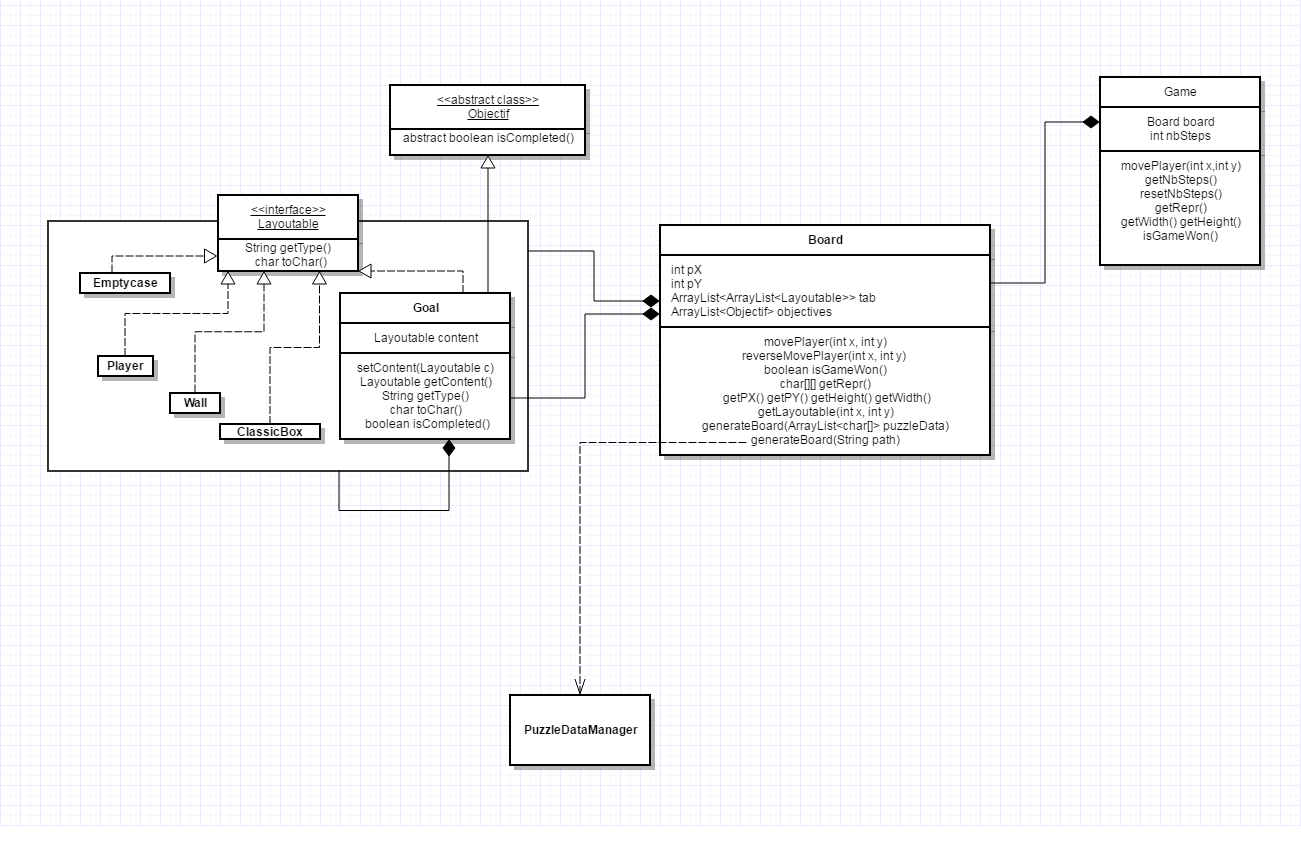


Figure - UML du package Backend

Si les objectives sont une List d'Objectif et non de Goal, avec Objectif une interface que seul Goal implémente, c'est pour laisser l'opportunité à l'ajout de fonctionnalité tel qu'il y a d'autres conditions de victoire que de mettre chaque caisse sur un goal. (par exemple: un interrupteur pressé un nombre pair de fois)

Si les box sont en réalité des ClassicBox implémentant la classe abstraite Box, c'est pour laisser l'opportunité de l'ajout de Box différente que la ClassicBox (par exemple: une boite avec un nombre limité de déplacements). A noter cependant que les méthodes movePlayer devraient être partiellement réécrites auquel cas)

La première question que l'on pourrait se poser est pourquoi l'interface Layoutable est implémentée avec ces méthodes toChar() et getType(). En effet, d'une part ces méthodes sont tout à fait générales et ne reflètent pas particulièrement le caractère "élément du jeu (du layout)" qu'elle sont sensé représenter. De plus, elle ressemble fortement à l'opérateur instanceof qui est généralement une mauvaise pratique, ne tirant pas profit de la programmation orientée objet.

Des objets implémentant une interface Movable avec une méthode move semble plus naturel. (C'est totalement sur cette optique que j'étais parti dans un premier temps, jusqu'à me rendre compte des problèmes qu'elle engendrait). Les Goal et les Wall n'étant pas déplaçable, ils n'implémenteront pas l'interface Movable et le jeu serait donc divisé en deux couches. Il existe encore plusiers manières d'implémenter le jeu à partir de là

Dans cette optique, il semblait naturel que les objets Movable connaissent leurs position. Et donc que ceux ci ne soient pas ordonnés dans un tableau (au sens mathématique du terme) dans Board. En effet, l'information de leurs position aurait été redondante pour chaque objet dans le cas contraire. Board aurait plutot contenu une simple List<Movable>. (Optique ayant l'avantage de ne pas stocker de "null" dans les interstice d'une map non rectangulaire, bien que cela représente un maigre avantage vu que même dans les cas extrèmes, cela représente un petit nombre de "null" potentiels évités). Cette implémentation avait l'énorme désavantage de forcer le parcours, plusieurs fois potentiellement, de la liste de tout les éléments pour obtenir ceux à proximité du Player lorsque nous voulions nous déplacer pour obtenir les deux éléments dans la direction choisie à passer en paramètre à move (ou de passer tout les éléments ainsi qu'un paramètre représentant la direction, ce qui aurait été encore pire) afin que move puisse mérifier si il est bien possible de se déplacer avec ces deux éléments comme cases adjacente dans la direction de choix du déplacement, et puisse modifier les coordonnées (x,y) de l'élément sur lequel move est appliqué ainsi que de son suivant, et potentiellement de celui d'apres (cas de déplacement d'une caisse, où la Box plus loins prendrait la place de l'EmptyCase encore après, l'EmptyCase la place du Player et le Player la place de la Box).  
Le fait de bien utiliser la programmation orientée objet ne me semblait pas un argument nécessaire que pour devoir potentiellement parcourir (plusieurs fois potentiellement aussi!) tout les éléments.

On pourrait donc imaginer que les objets Movable comme connaissant leurs position ET qu'ils soient organisés dans une List<List<Movable>> dans Board. Bien que cela rendait l'information sur la position redondante (une fois en variable de movable, une autre en position dans la List de List). Ainsi, obtenir les Movable adjacents à Player, à passer dans move, serrait beaucoup plus simple. Mais dans ce cas move ne pouvais plus se contenter de changer les coordonnées de l'objet sur lequel il était appelé ainsi que du/des adjacents, move devait aussi modifier le tableau (au sens mathématique, la List de List concrètement) du Board dans lequel il était appelé. Il aurait donc été nécessaire de passer le tableau entier à une méthode d'un de ses élément. Passer le contenant à un élément contenu. Cela semblait être particulièrement une mauvaise pratique et cette optique a été abandonnée. Me faisant tirer la conclusion que si implémentation par tableau (List de List) à lieux d'être, alors la méthode move faisant les déplacement doit se trouver dans le Board et non dans les Movable eux même. Rendant l'interface Movable très discutable vu que les Movable ne s'occupent plus eux même de se déplacer.

Il existe une autre complexité à cette implémentation, n'étant pas spécialement un désavantage mais méritant d'être citée, qui est que les murs et les goals ne sont pas susceptible de bouger, et ne devraient donc pas implémenter l'interface Movable. Cependant, l'information sur leurs position devrait bien être présente, et un Goal peut se trouver au même endroit qu'un Movable, mais ne pouvait facilement contenir de Movable (comme c'est le cas dans mon implémentation) car déplacer un Movable aurait impliqué de regardé dans chaque Goal si il n'était pas contenu quelque part. Cependant une autre optique aurait pu résoudre ce problème: une conception du plateau sous deux couche: une couche avec les goals (et les murs éventuellement), une couche avec les EmptyCase, les joueur et les caisses (et les murs éventuellement). Cette optique a donc elle aussi été envisagée pour être abandonnée car les problèmes cités plus haut restaient bien présent.

Voici une implémentation, fort proche de celle que j'ai réalisé pour ce projet, qui aurait pu correctement respecter les principes de la programmation orientée objet et ne pas engendrer de soucis majeurs cités précédemment:   
Board contient un tableau (List de List) de Movable ainsi qu'un tableau (List de List) de même dimension de Goal. Ni les Movable ni les Reachable ne connaitraient leurs position (pas de redondance d'information). Les déplacement seront effectués dans une méthode moveObjects de Board, qui est la classe possédant ces tableau et donc l'organisation de ces objets. Ceci dit, pour savoir concrètement quels déplacements faire, le Board appelera howToMove, méthode du Player, en lui passant les deux éléments Moveable adjacent dans une direction donnée. Cette méthode howToMove devrait appeler les méthodes canMove, méthode des objets Movable, sur ces objets adjacent. Ensuite, en fonction du résultat, howToMove renverrait un int: -1 si impossible de se déplacer, 0 si il faut échanger les place de players et du Movable adjacent (déplacement vers un lieu vide) et 1 si il faut faire un échange entre les trois objets (déplacement d'une caisse ver un lieu vide. lieu vide -> joueur, joueur -> caisse et caisse -> lieu vide), déplacement que la méthode moveObjects du board réalisera.   
Toute la complexité de cette implémentation réside dans les méthode howToMove et canMove, que le Board pourrait lui même faire plus simplement si il regardait directement le type des objets composant ses tableau (et lui même sait qu'un joueur ne peut aller sur un mur, lui même sait qu'une caisse ne peut être déplacée que si la case suivante n'est qu'un vide) et donc un seul paramètre devrait être passé par les méthodes de Movable pour obtenir de l'information!

C'est donc pour cela que j'ai abandonné cette dernière implémentation pour arriver sur mon choix final d'implémentation. Il est donc important de noter que, comme c'est le Board qui sait comment fonctionne le déplacement des Layoutable ET qui connait leurs position, les objets implémentant l'interface Layoutable n'ont de sens que contenu dans le Board. (Qui pourrait implémenter une interface "Boardable" pour rendre cela plus général, comme un design pattern).

La conclusion de tout cela est que, bien que l'implémentation des objets composant le Puzzle sous forme de Layoutable n'ayant que les méthodes toChar() et getType() ne semble pas idéale, celle ci peut prendre plus de sens dans le cadre du Board et que les autres implémentations des défaux majeurs. Cependant, je ne doute pas qu'il y ai une meilleure implémentation.

* Package Frontend: la GUI (Graphic User Interface[[2]](#footnote-2))

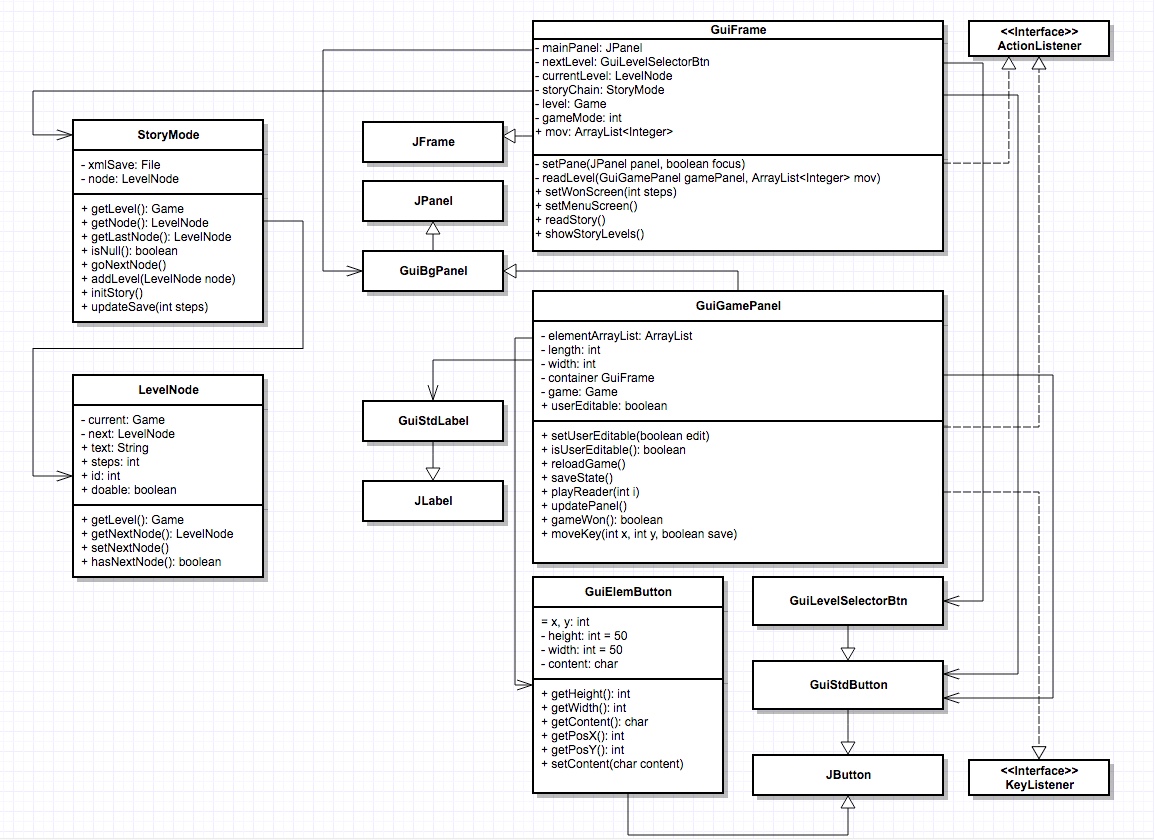
L'interface graphique (GUI) est composée de plusieurs classes qui permettent sa réalisation qui se retrouvent dans le package Frontend.

Figure 2 - UML du package Frontend

Sur le schéma UML (Figure 1) on distingue deux composantes principales, à droite nous avons les composantes graphiques rassemblant les classes GuiFrame, GuiGamePanel, GuiBgPanel, GuiStdLabel, GuiElemButton et GuiLevelSelectorBtn qui héritent tous des classes d'interface graphique de Javax.Swing (voir 3. Swing ou JavaFX?). Ces classes permettent d'afficher ce que verra l'utilisateur. A gauche nous avons deux classes (StoryMode et LevelNode) qui permettent d'initialiser le mode Classic (mode histoire).

Nous avons choisi d'implémenter ce mode via une liste simplement chaînée de niveaux. En effet, la classe LevelNode a comme attribut "current" qui représente un niveau du LevelNode, "id" qui est un identifiant entier unique représentant le niveau (le niveau 1 aura l'id 1, le niveau 2 l'id 2 etc.) et "next" qui est le LevelNode représentant le niveau suivant. Cette chaîne est enregistrée dans la classe StoryMode. On y retrouve des méthodes permettant le déplacement de niveaux en niveaux (pour passer du niveau "n" au niveau "n+1" où n est l'identifiant unique d'un niveau, on appelle la méthode goNextNode()).

Dans StoryMode, seulement deux attributs sont enregistrés: xmlSave, le fichier de sauvegarde dans lequel la progression de l'utilisateur est enregistrée (ce fichier est expliqué dans le point 4. "Mode Histoire" de ce document) et node, le niveau que le joueur a sélectionné (soit via la méthode "showStoryLevels()" de GuiFrame permet d'afficher la liste des niveaux du ClassicMode en utilisant des GuiLevelSelectorBtn, soit en utilisant le GuiLevelSelectorBtn qui s'affiche en fin de partie contenant le texte "Niveau suivant..").

Les boutons du menu d'accueil lançant une partie (voir point 8. Guide utilisateur de SpeedyRoadie) créent un nouveau GuiGamePanel (JPanel contenant l'interface de la partie en cours) et, en fonction du mode sélectionné, génèrent selon des constructeurs différents, une partie (S'il s'agit du mode histoire, nous affichons un menu intermédiaire pour sélectionner le niveau souhaité).

1. Points forts

* Sauvegardes en continu et pendant la partie

Une fois que le joueur a commencé une partie, on en enregistre une copie dans le dossier PermanSave au format .xsb (format de fichier des plateaux de jeu Sokoban). Dans ce même répertoire, on enregistre le fichier .mov (format de jeu de l'historique des mouvement du joueur) qui se met à jour automatiquement avec chaque mouvement du joueur.

On peut donc garder une sauvegarde en continu de l'état d'avancement du joueur dans le niveau courant. Si le joueur quitte inopinément la partie et relance le jeu, une petite fenêtre lui demandera s'il souhaite continuer sa partie à l'endroit où il était.

De plus, il est également possible pour le joueur de sauvegarder sa partie à tout moment en un simple clic sur un bouton.

* Lecture de .mov en mode cinématique

Si le joueur charge un fichier .xsb et un fichier .mov, le jeu "lira" le fichier de mouvements comme une série d'instructions déplaçant progressivement le joueur sur le plateau avec un intervalle de 0,25 secondes entre chaque déplacement.

Pour ce faire, on a implémenté une classe étendant ActionListener nommée ClockListener qui s'exécute périodiquement grâce à un Timer.

On ôte le focus de l'interface graphique afin d'éviter que le joueur fausse les déplacements en déplaçant le personnage pendant la lecture du .mov

Une fois le déplacement terminé, on rend le focus à l'élément graphique représentant la partie courante et le joueur peut reprendre la partie.

* Mode histoire

Comme expliqué dans le point 3. "Thème du jeu et mode histoire" de ce document, nous avons mis en place un mode histoire où le personnage principal du jeu, Speedy, passe de scènes en scènes.

Pour ce mode, nous avons également mis en place une sauvegarde de l'état d'avancement du personnage dans un fichier sauvegarde.xml se trouvant dans le dossier "ClassicMode".

Dans ce fichier sont enregistrés les textes s'affichant avant chaque niveau, l'état d'avancement (si on peut faire un niveau, l'attribut "doable" du fichier de sauvegarde est à la valeur "true" et donc le bouton dans la liste des niveaux du menu ClassicMode est cliquable, sinon il est grisé) et le nombre de pas nécessaires au joueur pour terminer le niveau.

Afin de réinitialiser la sauvegarde sans trop de difficultés (pour éviter que le joueur ait accès aux niveaux suivants avant d'avoir les niveaux précédents) nous avons ajouté une commande à notre build.xml qui est "ant reset" qui permet de remettre à zéro la sauvegarde du jeu.

* Réinitialisation de la partie

En cas d'erreur de la part du joueur, il est possible de totalement recharger le niveau en cours de partie en un simple clic sur un bouton.

1. Faiblesses

* Mode histoire

A chaque fin de niveau, on affiche un bouton permettant de passer au niveau suivant.

Or, pour savoir vers quel niveau rediriger, on utilise une classe étendant JButton dans laquelle on passe en paramètre l'id du niveau. Pour récupérer le niveau correspondant à cet id, on doit parcourir toute la chaîne. La complexité de cette opération est en O(n).

* Apparence du code

Tant dans le package Frontend que Backend, nous avons eu quelques difficultés à garder un code propre pour l'entièreté des méthodes et classes.

Dans le package de Corentin (Backend) nous pouvons citer le code de la méthode movePlayer de la classe Board qui peut sembler un peu "spaghetti" et qui aurait pu être partitionné en sous méthodes pour éviter le recopiage du code à certains endroits

Il en est de même pour la classe GuiFrame du package Frontend que j'aurais pu découper en sous méthodes et de ce fait éviter une multitude de répétitions.

1. Erreurs connues

Malgré le temps qui nous a été accordé pour la mise en place de ce Sokoban et l'expérience que nous avons acquis tout du long, nous avons rencontré des erreurs que nous n'avons pu résoudre. Nous n'avons pas la prétention d'avoir retrouvé tous les bugs qui se cachent dans notre programme mais ce sont ceux que nous avons trouvé mais que nous n'avons pas su résoudre.

* Lecture de .mov en mode cinématique

En essayant notre SpeedyRoadie sur différentes plateformes et sur des ordinateurs ayant différentes configurations, nous avons remarqué que la lecture du .mov en mode cinématique pouvait poser problème dans le cas où la taille du fichier .mov était de grande taille (nous avons essayé des fichiers contenant 50, 100, 200, 300, 500 instructions de déplacement et à partir de 200 mouvements, le programme "oublie" certains déplacement)

Il arrive que parfois des déplacements ne soient pas pris en compte, faussant la cinématique. Nous avons recherché la raison de cette erreur mais nous ne sommes pas parvenus à en trouver l'origine ou la solution.

L'ordinateur sur lequel les erreurs sont survenues dispose d'un processeur Intel Celeron G3900 dual core 2,7 ghz avec 2 Go de RAM. Je me doute que ce sont ses caractéristiques matérielles, le système d'exploitation (windows XP 32bits) ou la configuration de la machine virtuelle Java (elle est configurée pour n'utiliser que 512 mo de RAM) qui ont provoqué l'oubli de mouvements car sur mon ordinateur personnel (Intel core i7 4 core 3,7 ghz avec 16 Go de RAM) je n'ai jamais rencontré cette erreur.

* Exécution sans Apache ant et ressources graphiques

Notre SpeedyRoadie a été développé dans l'optique d'être exécuté via les commandes ant build, ant run, ant clean, ant test et ant reset (voir point 4, Mode histoire).

Si le jeu est exécuté via les commandes java directement (et donc depuis un autre dossier que le dossier où se trouve le fichier build.xml) le chemin vers les images de fond (logo de bienvenue, texture de boutons, etc.) ne fonctionne pas car c'est un chemin relatif au dossier racine.

Par exemple si vous souhaitez générer un fichier .xsb à partir d'un input.xsb et d'un input.mov (voir 8. Générer l'output d'un fichier .xsb et .mov) en oubliant de mettre des arguments à frontend.Main, le jeu se lancera mais sans les images de fond.

1. Apports positifs de ce projet

Ce projet nous a été très instructif. D'abord nous étions tous deux avec d'autres personnes mais les choses ont fait que ces dernières ont quitté la faculté des sciences en cours d'année.

Nous avons, grâce à ça, appris à développer seul un moment avant de retrouver un monôme pour le binôme que nous formons.

Programmer en binôme n'est pas une chose facile car nous avions tous deux des points de vue différents sur certains éléments du développement du jeu alors nous avons dû, à plusieurs reprises, nous concerter pour discuter du projet et de son avancée.

Nous avons également dû gérer des problèmes de notre côté avant de faire une mise en commun. On a, grâce à ces problèmes, dû se coordonner sur la marche à suivre pour leur résolution sans empiéter sur le travail de l'autre. Utiliser GitHub a été une véritable avancée pour le développement puisque nous pouvions mettre à jour du code ensemble sans devoir vérifier à la main les changements de chaque fichier.

Chaque deadline que nous nous imposions n'a pas forcément été respectée mais nous avons pu terminer ce programme. Nous savons, grâce à ça, qu'il ne faut pas s'y prendre la veille pour avoir un sokoban qui fonctionne.

1. Guide utilisateur de Speedy Roadie

Afin de pouvoir profiter pleinement du jeu, ce petit guide utilisateur vous expliquera comment en exploiter au maximum les options.

* Ant

Pour commencer une partie, placez-vous avec le terminal dans le dossier racine du jeu (dossier où se trouve le fichier build.xml).

Dans ce même terminal, vous pouvez utiliser l'instruction "ant reset" avant de jouer.

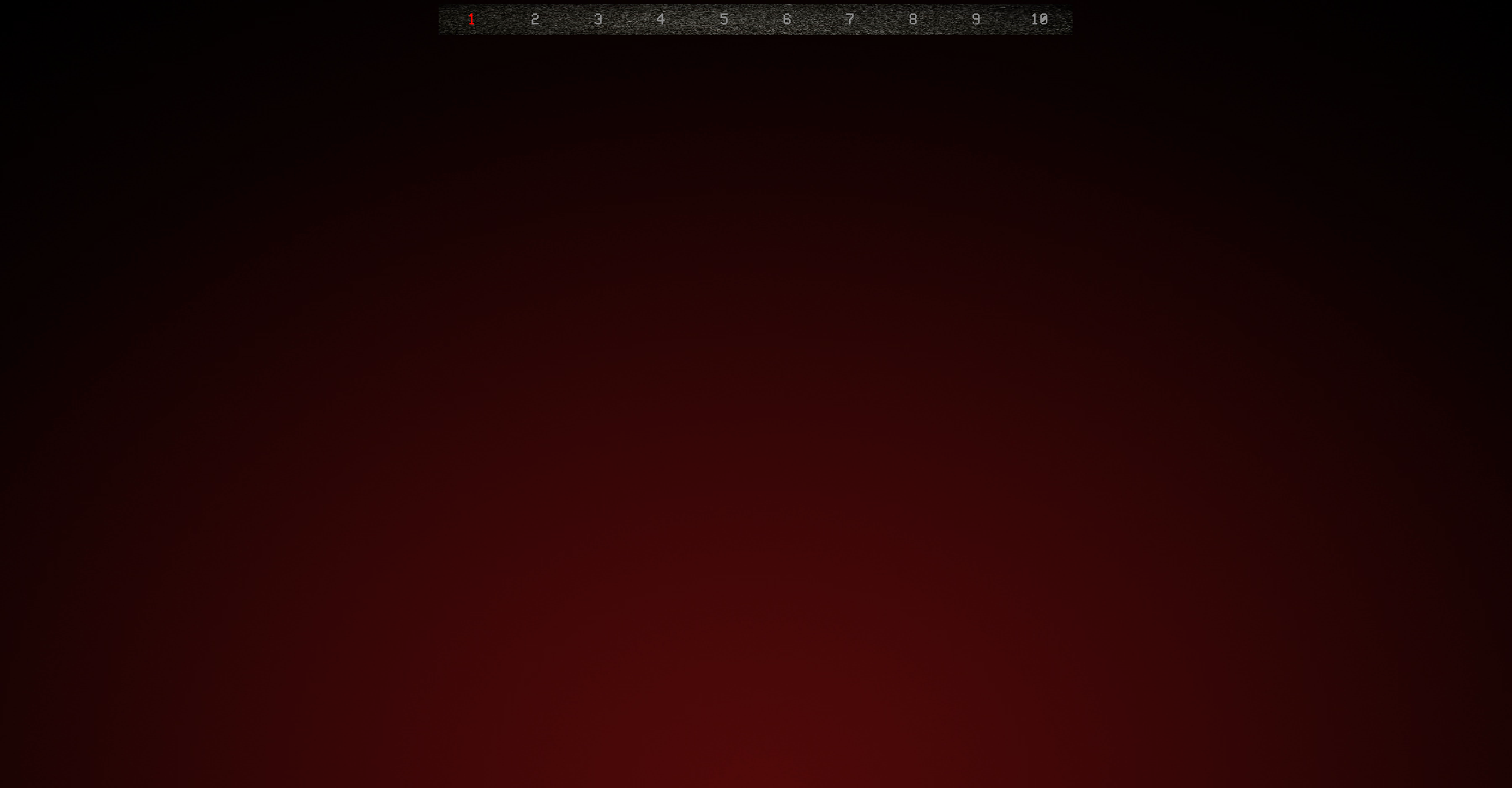
Cette commande permet de réinitialiser la sauvegarde du mode Histoire.

Utilisez ensuite les commandes "ant build" (le clean se fait automatiquement avec le build) et "ant run" afin de lancer la partie.

* Menu d'accueil

Le menu d'accueil dispose de 4 boutons (voir Figure 3)

* Mode histoire lançant le menu de sélection des niveaux du mode histoire
* Mode aléatoire, affichant un sélecteur de difficulté
* Chager une partie, permettant de charger un fichier .xsb et/ou un fichier .mov pour continuer une partie enregistrée
* Quitter le jeu, permettant d'arrêter l'exécution du jeu.
* Mode Histoire



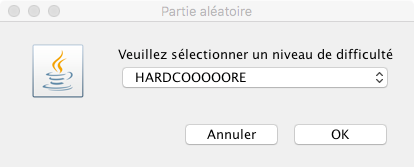
Pour le moment, vous ne pouvez que jouer au niveau un. Pour pouvoir avancer dans l'histoire, vous devrez avoir terminé les niveaux précédents.

Si vous quittez le jeu après avoir terminé un niveau, ne vous en faites pas, la sauvegarde est automatique.

Cliquez sur le niveau 1 pour en voir l'histoire et pour jouer à ce niveau.

* Mode aléatoire

En cliquant sur le bouton "Mode aléatoire" vous aurez un sélecteur de difficulté qui va s'afficher. Vous aurez le choix entre différents modes de difficulté, variant entre des plateaux de jeu plus ou moins grand et plus ou moins de flightcases à déplacer.

Choisissez la difficulté du niveau à jouer et cliquez sur Ok.

Votre partie se lancera et vous n'aurez qu'à jouer!

* Charger une partie

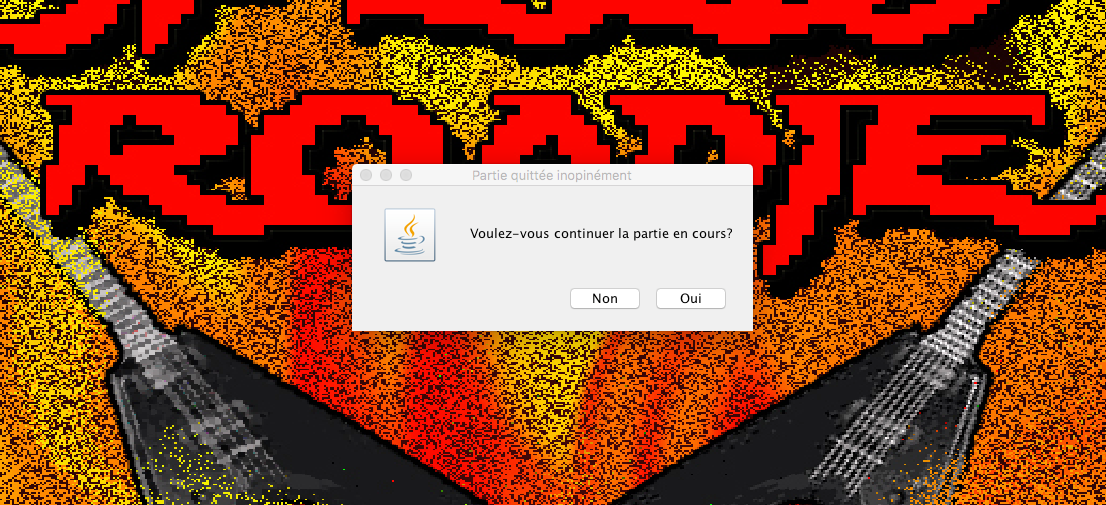
Si vous avez des fichiers de partie personnalisés et/ou un fichier de sauvegarde .mov à charger, vous pouvez cliquer sur le bouton "Charger une partie"

Un petit menu contextuel s'ouvrira pour chercher vos fichiers.

* Continuer une partie en cours

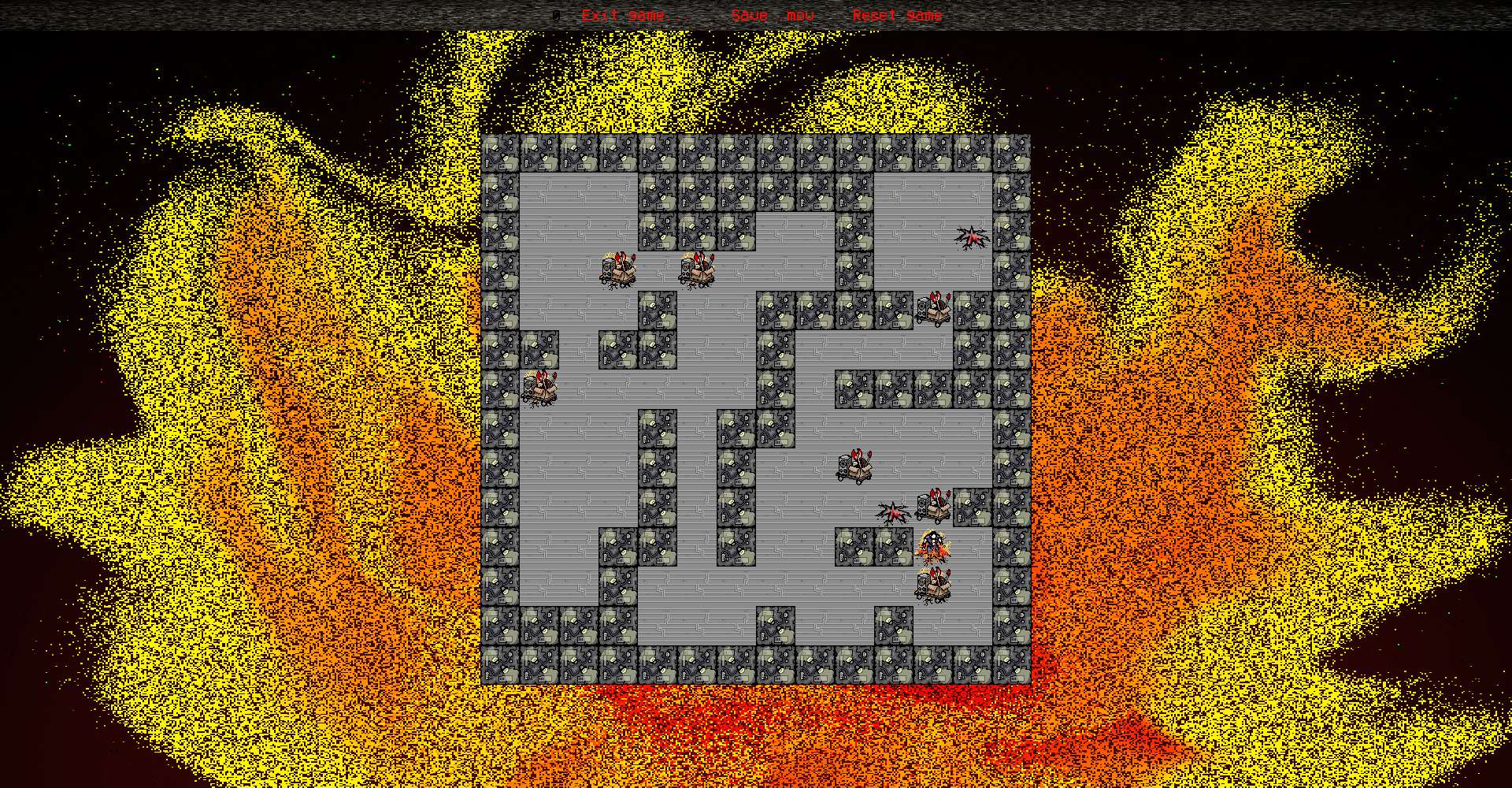
Si vous avez quitté inopinément votre partie en cours de jeu, vous avez la possibilité de continuer cette dernière.

Après avoir quitté la partie, au prochain lancement du jeu, un menu contextuel s'ouvrira



Vous n'avez qu'à cliquer sur Oui pour continuer la partie où vous étiez. Afin de vous rappeler tous les pas que vous avez fait, le jeu va passer en mode cinématique et va rejouer chaque déplacement que vous avez fait avant de quitter le jeu.

* Fonctionnement d'un niveau

Pour vous déplacer au sein d'un niveau, vous avez deux possibilités. Vous avez le choix entre le déplacement au clavier (via les touches directionnelles) ou en cliquant sur une des cases autour de notre roadie préféré pour vous y déplacer. Pour pousser une caisse en utilisant la souris, cliquez sur cette dernière.

Vous pouvez à tout moment de la partie enregistrer le fichier .mov de sauvegarde en cliquant sur "Save .mov".

Il est également possible de réinitialiser le niveau joué en cliquant sur le bouton "Reset game". Les pas que vous avez alors réalisé jusque-là ne seront pas pris en compte et vous recommencerez simplement le niveau à partir de zéro.

* Fin de partie

En fin de partie, vous avez le choix (si vous êtes dans le mode histoire) de continuer au niveau suivant ou de revenir au menu d'accueil.

Si vous êtes en mode aléatoire ou que vous avez chargé une partie, vous n'avez pas d'autre choix que d'accepter votre sort de grand vainqueur du niveau pour retourner humblement au menu d'accueil.

* Générer l'output d'un fichier .xsb et .mov

Il est possible de calculer le résultat de l'application d'un fichier d'instructions de déplacement (.mov) sur un fichier de plateau sokoban (.xsb) via une commande. Pour ce faire, suivez ces étapes:

* Placez vous avec un terminal à la racine de SpeedyRoadie
* Compilez le projet avec la commande "ant build"
* Toujours dans le terminal, déplacez-vous dans le dossier "run"
* Entrez la commande "java frontend.Main input.xsb input.mov output.mov" où les arguments sont:

 - input.xsb, le chemin vers un fichier xsb de base

 - input.mov, le chemin vers le fichier mov à appliquer à l'input

 - output.mov, le chemin vers le fichier de sortie

1. Remerciements

Nous remercions assistants qui nous ont grandement aidé pour ce projet, qui ont répondu à toutes nos questions et nous ont guidés dans le développement de ce Sokoban.

Nous remercions également <INSERER LE NOM DE TON AMI ICI> qui a dessiné les images du plateau de jeu, les murs, le sol, les caisses, Speedy et les objectifs.

1. http://stackoverflow.com/questions/35264887/what-is-the-difference-between-unsing-java-fx-and-swing [↑](#footnote-ref-1)
2. Interface graphique de l'utilisateur, partie du logiciel qui permet à l'utilisateur d'interférer avec les actions de ce dernier. [↑](#footnote-ref-2)