Faculté des Sciences
Institut d'Informatique
UMONS

Speedy Roadie

Louis Dhanis, Corentin Dachy

Projet d'informatique, mai 2017

Table des matières

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.

2. 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 Antoine Fauville, 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].

  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.

- ## *Organisation du travail: GitHub*

  Afin de ne pas perdre notre travail en cas de bug informatique, on a choisi de travailler avec GitHub qui nous permettait de mettre à jour notre code chacun de notre côté tout en n'empiettant pas sur le travail de l'autre.

  Nous avons mis notre dépôt en "private" durant tout le développement du projet mais il est public désormais à l'URL suivante:

  https://github.com/BarberousseBinks/sokoban/tree/master/SpeedyRoadie

  Il est possible d'y voir une multitude d'informations et de statistiques. Nos pseudonymes sur Github sont:

  - BarberousseBinks (pour Louis Dhanis)
  - Alfattarte (pour Corentin Dachy)

- ## *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 modes, 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 explications sur l'implémentation complète ainsi qu'une petite discussion sur les choix principaux effectués.

[1] http://stackoverflow.com/questions/35264887/what-is-the-difference-between-unsing-java-fx-and-swing

○ *La classe Game :*

Toute les informations sur une partie devront se retrouver dans l'instance de la classe Game qui lui est associée (une partie = une instance de Game)

Dans cette instance, nous enregistrons le plateau (voir classe Board) et le nombre de mouvements effectués par le joueur jusqu'à là.

La partie graphique demandera des informations sur la partie en cours et voudra la modifier (de par le joueur effectuant un déplacement). Ces modifications de l'instance et l'obtention des informations passera donc des méthodes publiques de Game. En voici les principales :

- movePlayer(int x, int y) pour déplacer le joueur selon un vecteur unitaire (x, y)
- getNbSteps() qui renvoie le nombre de pas effectués jusqu'à là
- getRepr() qui renvoie un tableau de caractères bi dimensionnel où chaque caractère représente un élément du jeu (mur, sol, joueur, etc.)
- isGameWon() qui retourne un booléen : true si la partie est terminée et donc gagnée, false sinon.

À l'exception de getNbSteps, toutes ces méthodes utilisent les méthodes de la classe Board de par le plateau (getRepr() et isGameWon() ne font qu'appeler les méthodes éponymes de la classe Board et movePlayer() fait de même en plus d'éventuellement incrémenter le nombre de mouvements effectués par le joueur si le déplacement a bien eu lieu).

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

○ *Les éléments du jeu :*

Les éléments du jeu sont des classes qui implémentent l'interface Layoutable. Cette interface permet d'obliger les éléments à posséder les méthodes

getType() qui renvoie le string relatif à son type (par exemple la classe Wall renverra le string « wall »)

toChar() qui renvoie le caractère le représentant dans le tableau de caractère (une box sera représentée par le caractère « $ »)

Les différentes classes implémentant Layoutable (autrement dit, les différents éléments du puzzle) sont :

- ClassicBox pour les caisses
- EmptyCase pour le sol
- Player pour représenter Speedy, le personnage du jeu
- Wall pour les murs
- Goal pour les objectifs

La classe Goal est un peu plus complexe que les autres.

En effet, un Goal est un élément contenant un autre élément du jeu (une caisse, un vide ou le joueur). La classe Goal possède donc une variable content (de type Layoutable). Ainsi, toChar() et getType() renverront des réponses différentes en fonction du contenu du Goal. (Goal contient donc des méthodes, appelée par le Board qui le contiendra, permettant d'accéder à son contenu et de le modifier).

De plus, Goal implémente également l'interface Objectif. Cette interface contient la méthode isCompleted() renvoyant true si l'objectif en question est complété. Dans le cas des Goal, cette méthode renverra true si le contenu du Goal est une boite (ClassicBox)

- ***La classe Board :***

Le plateau (instance de la classe Board) contient tout ce qui est relatif aux éléments du puzzle et à leurs position à un temps donné. (C'est donc sans surprise que Game contiendra un Board ainsi que le nombre de pas effectués depuis le début, cette information n'ayant rien avoir avec la disposition actuelle des éléments du puzzle). Autrement dit, Board contiendra des objets Layoutable en plus de les organiser entre eux (les éléments Layoutable n'ayant pas d'information sur leur position).

Le plateau a donc une ArrayList d'ArrayList d'éléments Layoutable. Cette ArrayList<ArrayList<Layoutable>> porte le nom de tab. Ce tab contient la totalité des informations du plateau. Techniquement, aucune autre variable n'est nécessaire pour Board.

Cependant, Board comporte deux autres variables, toute deux visant à améliorer les performances du programme (pour un faible coût en espace de stockage) :

-pX et pY, deux int-objectives, une ArrayList<ArrayList<Objectif>>

En effet, afin de pouvoir retrouver à tout moment le joueur sans devoir passer en revue chaque élément de tab (nous aurions alors une complexité en $O(n^2)$ car récupérer un élément est en O(1), pour récupérer tous les éléments (donc des ArrayList) dans une boucle on a alors une complexité en O(n), pour récupérer les éléments contenus dans les deux dimensions on a alors une complexité en $O(n^2)$) le Board possède la position du joueur dans les variables entières (int) pX et pY (autrement dit, on aura toujours que le Player sera à tab.get(pY).get(pX), qui a une complexité en O(1)). Cela réduit considérablement le temps de calcul, d'autant plus qu'à chaque déplacement du joueur (chaque appel de movePlayer) la position du Player doit être connu (tout déplacement se faisant dans tab et par rapport au Player).

Chaque instance de Board contient également une ArrayList<Objectif> (du nom d'objectives) contenant chaque objectifs (des références vers chaque objectif). Bien que tous les objectifs soient des Goal déjà référencé dans tab, objectives permet d'y accéder directement sans devoir les chercher dans tab (même problèmes que cité précédemment). D'autant plus que la patrie graphique a intérêt à appeler la méthode isGameWon de Game (et donc de Board indirectement) à chaque mouvement pour vérifier si chaque Goal contient une caisse et donc si la partie est gagnée. De plus, cette organisation laisse l'opportunité d'implémentation d'autres objectifs que les Goal (voir partie discussion)

Les méthodes les plus importantes de Board sont :

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

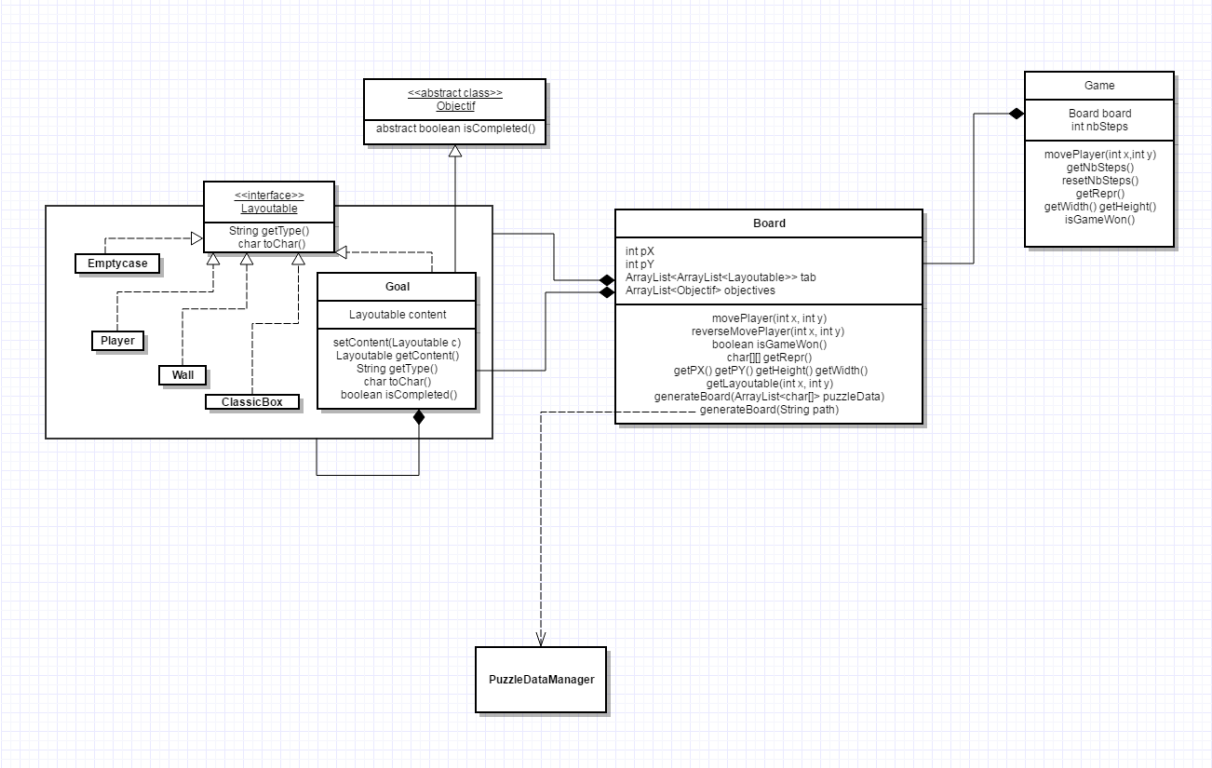
- isGameWon() regardera chaque Objectif de objectives et renverra true si chaque objectif renvoi true via isCompleted(), false sinon.

- movePlayer(int x, int y) qui déplacera le personnage dans tab dans la direction indiquée par le vecteur unitaire (x, y) (renvoie une exception si cette précondition n'est pas respectée). En plus de potentiellement (si le déplacement est possible) modifier tab, cette méthode renvoie l'entier qui sera stocké dans le .mov (voir 3. Standards des fichiers de sauvegarde) ou -1 si le déplacement a été impossible (déplacement vers un mur par exemple)

- reverseMovePlayer(int x, int y) qui fera le même travail que movePlayer mais à l'envers (au lieu de pousser les caisses, ça les tirera). Cette méthode (protected) n'a d'utilité que dans le générateur de niveau aléatoire et ne devrait jamais être appelée une fois un niveau créé.

- generateBoard(ArrayList<char[]> puzzleData) qui remplit les informations de Board (pX, pY, tab, objectives) en fonction de l'ArrayList passée en paramètres

Bien que d'autres méthodes existent, elles ne sont pas d'une importance capitale pour la compréhension mais sont accessibles en annexe décrites dans la Javadoc. Notons juste que des constructeurs de Board (et de Game) peuvent prendre en paramètre un chemin d'accès vers un fichier, qui sera transcrit en ArrayList<char[]> par la méthode readBoard du PuzzleDataManager (une classe boite à outils contenant des méthodes static relatives à la lecture écriture de fichiers qui entrent en jeu dans notre sokoban)



Attention, dans ce diagramme "UML", les normes et conventions d'un diagramme UML ne sont probablement pas toutes respectée. Voyez-le comme un schéma que partiellement rigoureux donnant cependant une visualisation de l'implémentation d'une partie

Discussion :

1) Les objectives auraient pu être une List de Goal et non d'Objectif, et Goal est la seule classe implémentant Objectif. Ceci est fait pour laisser l'opportunité d'ajout de fonctionnalités amenant d'autres conditions de victoire. (Par exemple : un interrupteur au sol devant être pressé un nombre pair de fois)

2) Si les box sont en réalité des ClassicBox implémentant la classe abstraite Box, c'est pour laisser l'opportunité d'ajouts 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 associées à movePlayer devraient être partiellement réécrites auquel cas.

3) L'interface Layoutable ne possède que getType() et toChar(). Cela semble être une implémentation ne profitant pas des paradigmes de la programmation orientée objet (ressemblant presque à l'utilisation répétée de l'opérateur isinstanceof). C'est pourtant en connaissance de cause que cette implémentation a été choisie (surtout en dépit des désavantages que les autres implémentations engendraient).

Voici la philosophie des objets implémentant l'interface Layoutable :

Ces objets ne savent pas grand-chose, si ce n'est qu'ils sont des éléments pouvant constituant un plateau de jeu, ainsi que ce qu'ils sont dans ce plateau (des murs, des boites, ...). Cependant, ils ignorent tout de leur position sur ce plateau et des interactions qu'ils peuvent avoir avec d'autres éléments Layoutable. Ils ont une confiance aveugle en le plateau (Board ici) qui les contient et les manipule. Le plateau étant l'entité connaissant toute les règles d'interaction entre ses Layoutable et sachant comment les organiser.

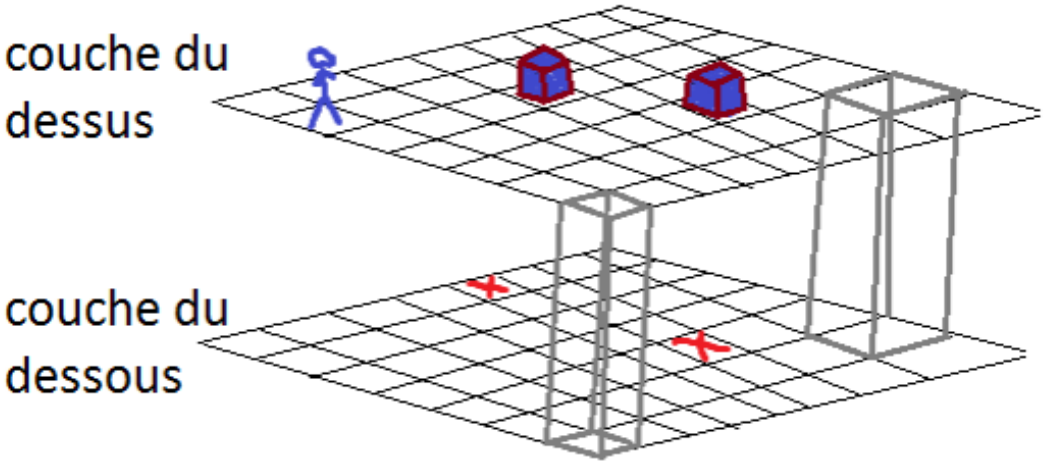
D'autres optiques, utilisant une interface Movable (a priori plus naturelle) au lieu de Layoutable ont été envisagées pour représenter les éléments du plateau de jeu. Les décrire en détails, ainsi que les soucis qu'elles amenaient, serait très fastidieux. Ceci dit il n'est pas inintéressant d'en parler, surtout qu'elles permettent de mieux comprendre la philosophie de l'implémentation via Layoutable finalement choisie. En voici donc simplement un (gros) résumé (des implémentations et des problèmes qui en étaient liées) :

Les éléments Movable connaissent leurs position (int x, int y) et la manière relative d'interagir (méthode move (Movable Adjacent, Movable Suivant)), et le Board ne gère donc qu'indirectement les positions (pas de List<List<Movable>> dans Board mais une simple List<Movable>). Problème : Nécessité de parcourir d'innombrables fois (potentiellement plusieurs fois pour un seul déplacement) la liste des éléments, afin de trouver les éléments adjacents à d'autres ainsi que les contenus des Goals.
-Les éléments Movable connaissent leurs positions et les manières relative d'interagir, mais le Board gère lui aussi leurs positionnements relatifs. Problème : En plus d'avoir redondance de l'information “position” pour chaque élément, les méthodes liées au déplacements (move) ne pouvaient se contenter de modifier les coordonnées représentant les positions des éléments, elles devaient également modifier leurs positions dans le tableau (la liste de liste) de Board. Ainsi, ces méthodes appelées sur des éléments contenus dans le tableau devaient prendre comme paramètre le tableau lui-même. Cette mécanique de serpent se mordant la queue ne semblait pas saine, en plus d'être couteuse. (Note : Peut-être que certains design pattern amèneraient une solution à ce problème, à considérer si le projet était à refaire)
-Les éléments Movable ignorent leur position mais connaissent leurs interactions relatives. Le Board les contenant connait leurs positions relatives. Le board appel move(Movable adjacent, Movable suivant (optional : Player p)) qui demandera à Player (qui lui-même demandera à adjacent et suivant) comment il peut se déplacer sachant qu'il est à côté d'adjacent lui-même à coté de suivant. (Via méthode howToMove et canMoveTo de Movable), après quoi le Board s'occupera de réaliser faire concrètement ce déplacement. Problème : Cette optique rendait l'implémentation inutilement compliquée, avec comme seul avantage par rapport à l'optique finalement choisie de donner de l'information aux objets Movable (les manières d'interagir entre eux). Information que le Board pouvait très bien avoir (il ne semblait pas plus intrinsèque que le Board ai cette information que l'élément qu'il contient. D'autres Board aurait pu utiliser eux aussi des murs, un joueur et des boites, mais avec des comportements très différents).

C'est pourquoi l'implémentation par l'interface Layoutable décrite plus haut a été choisie, implémentation assez proche de la dernière citée, mais étant plus claire. Le Board communiquant avec les éléments qu'il contient que par simple méthode avec un unique paramètre, suite à quoi il s'occupait lui-même de calculer quel sera le déplacement et il pouvait le réaliser concrètement sur le tas (en même temps que le calcul). (Notons également que la méthode toChar() de Layoutable rend l'implémentation de getRepr() extrêmement simple).

4) Le point 3 discute d'une dualité Movable – Layoutable. Cependant, il existe une autre dualité pouvant être discutable aussi bien les optiques d'implémentation “Movable” que “Layoutable” : la dualité Couche – Contenance.

En effet, deux éléments peuvent se retrouver sur la même position si l'un des deux est un Goal. Il a été choisi de gérer cette fonctionnalité par ce que nous appellerons Contenance : le Goal **contient** un autre élément. Une autre optique a été envisagée, celle de visualiser le plateau de jeu selon deux couches : une contenant les Goal et éventuellement les murs, une autre contenant le joueur, les caisses et les “EmptyCase”.



(Pour les implémentation Movable : rendant non plus les murs et goals Movable mais Reachable, avec la couche de dessus ne contenant que des Movable et celle du dessous que des Reachable. Pour l'implémentation Layoutable tout est Layoutable mais le Board utilise différemment les objets se trouvant dans sa couche du dessus (List<List<Layoutable>> upperLayer) que ceux dans sa souche du dessous (List<List<Layoutable>> lowerLayer)

Cette optique a été abandonnée au bénéfice de la mécanique de Contenance car elle n'apportait pas d'avantage évident par rapport à cette dernière, alors qu'elle nécessitait de stocker beaucoup plus d'éléments (avec éventuellement une grande partie d'éléments null en fonction des détails d'implémentation), en plus d'être légèrement plus compliquée à réaliser.

- *Package Backend : Génération aléatoire de niveau*

La classe PuzzleGenerator possède plusieurs méthodes (static) permettant de générer aléatoirement une partie. Vous trouverez dans cette partie une explication des étapes permettant à arriver à ce résultat, ainsi qu’une discussion sur certain point en fin de cette partie.

Les explications qu’apportent Joshua Taylor et le prof. Ian Parberry vis-à-vis de leurs propre implémentation de générateur de niveau Sokoban, dans le cadre d’un projet de l’University of North Texas, nous ont beaucoup aidé pour notre implémentation qui se base en beaucoup de points sur la leurs. Vous pouvez retrouver les dites explications à l’adresse :

https://larc.unt.edu/techreports/LARC-2011-01.pdf

La génération de niveau aléatoire se fait par l'appel de generateBoard, la seule méthode publique de PuzzleGenerator (avec checkMap) et prenant comme paramètre la largeur et la hauteur (qui seront multipliés par 3) souhaités ainsi que le nombre de caisse souhaité.

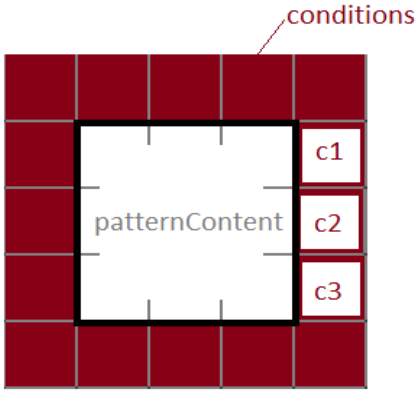
La création aléatoire d'un niveau est sous divisée en deux points principaux

-Générer un niveau vide, avec murs et « emptycase » uniquement

-Générer le joueur, les objets et les caisses dans ce niveaux dans ce niveau vide

- ***Génération de plateau vide :***

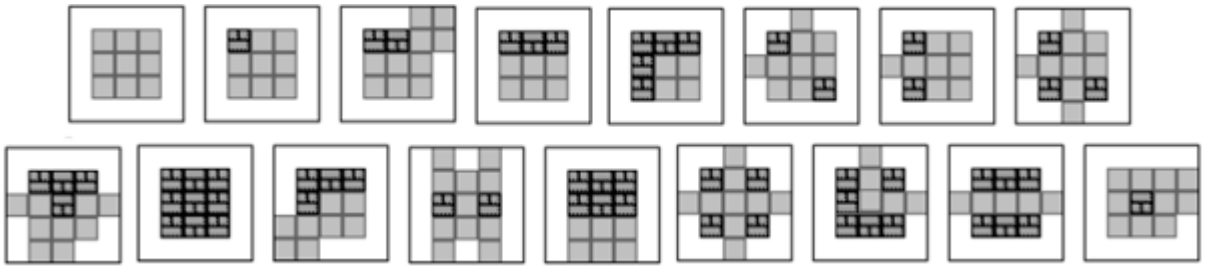
Générer un plateau (nommée map plus loin, par opposition à un board qui est une map complétée) dénuée de goals, de joueur et de caisses, revient à placer des «SPattern» (ou pattern) les uns à coté des autres. SPattern est une classe à part entière représentant une région 3x3 de carte (contenant un certain mélange de murs et d'emptycase) ainsi que 16 conditions potentielles portant sur les 16 cases en bordure de l'espace 3x3 du pattern. Les conditions d'un pattern, prenant toujours la forme d'une obligation de présence d'emptycase à un certain endroit, indiquent ce qu'un autre patterns doit présenter pour se retrouver à proximité du pattern étudié. Par exemple, si dans le dessin ci-dessous c1 indique une emptycase et c2 n'indique rien, alors pour placer un pattern à droite du pattern étudié, celui-ci doit obligatoirement avoir une emptycase dans sa position en haut à gauche mais peut cependant avoir un mur (ou une emptycase) dans sa position milieu gauche.



Représentation générique d'un SPattern

Ainsi, construire une map vide revient à piocher aléatoirement des pattern dans l'ensemble des pattern disponible et de les joindre, lorsque c'est possible, comme des pièces de puzzle. Ce qui explique par ailleurs pourquoi toute les map générée aléatoirement auront des hauteurs et largeurs multiples de 3.

Voici l'ensemble des pattern disponible (image provenant du document de Joshua Taylor et Ian Parberry) :



Notez bien qu'il existe en réalité bien plus que 17 possibilités différentes étant donné que chacun de ces pattern est susceptible de subir rotations et symétrie miroir lorsqu'il est sélectionné avant d'être potentiellement ajouté à la map.

Lorsqu'une map (de murs et d'empty case) est générée, elle doit encore passer la méthode checkMap qui la refusera si une de ces trois conditions n'est pas respectée :

- La map doit compter suffisamment d'emptycase (Elle doit contenir au moins une place libre pour chaque caisse que l'on compte y déposer, une pour chaque goal, une pour le joueur ainsi qu'une case supplémentaire libre)
- Elle ne doit pas contenir de cul de sac (afin d'éviter certaines maps inintéressantes)
- Tous les espaces libres doivent être connectés. En d'autres termes, la map ne peut pas être découpée en deux sous map n'ayant aucune liaison (afin également d'éviter des maps artificiellement trop petites)

Les maps générées ne respectant pas l'un de ces critères sont jetées et le travail est recommencé dès le début.

○ *Remplissage du plateau :*

Une map obtenue lors de l'étape précédente se verront essayée tentatives de complétion (afin de les transformer en véritable Board complet).

Afin de réaliser tentative de complétion, le joueur ainsi que le nombre de goals désiré seront placé sur la map. Chaque goal sera accompagné d'une caisse, qu'il contiendra. Cela donne donc un Board à l'état «gagné». Pour changer cet état où chaque caisse se retrouve sur un goal, la méthode reverseMovePlayer de Board sera appelé un grand nombre de fois (paramètre nbMovePerTry de PuzzleGenerator, valant 2000 sans modification) avec des directions aléatoires. Cette méthode déplacera le joueur avec les règles inverses de déplacement de caisse. Les caisses ne peuvent plus être poussées, mais peuvent (doivent) être tirées. Ce faisant, le programme joue la partie de sokoban à l'envers, commençant par la position de fin et finissant par la position initiale que le joueur retrouvera par la suite. Cette méthode à également l'avantage de garantir la faisabilité du niveau produit.

Mais cela n'est qu'une tentative de complétion. À chaque tentative de complétion, en plus d'un Board final, est associé un valeur range (de type int) qui sera incrémentée à chaque fois que reverseMovePlayer a comme effet de déplacer (tirer) une nouvelle caisse ou une caisse qui vient d'être tirée mais dans une nouvelle direction. Elle représente (de manière incertaines) la difficulté de la tentative complétion (voir point 3 de la partie discussion)

Ainsi, un certain nombre (paramètre nbTryLayout de PuzzleGenerator, valant 50 sans modification) de tentative de complétion seront effectuées. Celle qui sera finalement gardée sera celle ayant la valeur de range la plus élevée.

Discussion :

1) L'intérêt de cette implémentation par pattern 3x3 plutôt que par case aléatoire est de créer des niveaux plus intéressant, en plus diminuer largement le nombre de niveaux discontinus ou comportant un cul de sac évidents générés (mais ne réduisant pas ce nombre à 0, d'où l'intérêt de ces deux conditions dans checkMap afin de n'avoir finalement aucune map comportant ces caractéristiques).

   Le système de condition en est pour beaucoup, évitant au maximum les larges couloirs que des caisses ne pourraient passer. Par exemple, le pattern : 

   Pourrait former une zone inexplorable si n'imposait pas d'avoir au moins deux points libre continus pour y accéder. Mécanique qui aurait été impossible (ou tout du moins bien plus compliquée) à implémenter sur un système de génération case par case.

2) La fraction de map générée ne respectant pas les conditions de passage de checkMap dépend bien entendu de la taille désirée pour ces maps ainsi que du nombre de caisse que l'on souhaite y déposer.

   Voici quelques statistiques relatives au nombre de maps refusées par checkMap. Chacun des huit tests a été effectué avec un échantillon de 1 000 000 map générée aléatoirement.

   Maps (2,2,5) (map 6x6, en plus des murs extérieurs, devant contenir 5 caisses) :

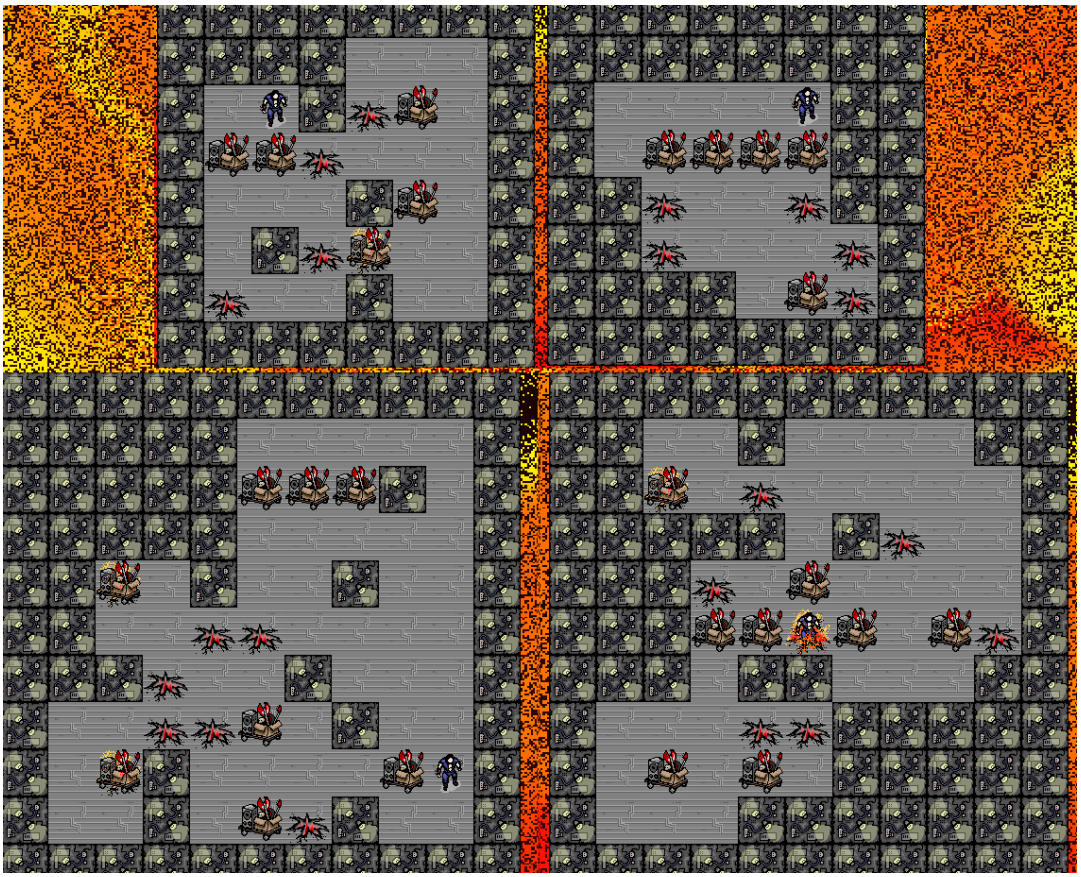
   - 0,004 % des map générées ne respectent pas la condition de suffisance de place
   - 68,555 % des map générées ne respectent pas la condition d'absence de cul de sac
   - 52,661 % des map générées ne respectent pas la condition de continuité

   - 62,3143 % des map générées ne respectent pas au moins l'une de ces trois conditions (et engendre une nouvelle génération de map)

   Maps (3,3,8) (map 9x9, en plus des murs extérieurs, devant contenir 8 caisses) :

   - 2,544 % des map générées ne respectent pas la condition de suffisance de place
   - 35,324 % des map générées ne respectent pas la condition d'absence de cul de sac
   - 28,714 % des map générées ne respectent pas la condition de continuité

   - 58,771 % des map générées ne respectent pas au moins l'une de ces trois conditions (et engendre une nouvelle génération de map)



Exemple de Board (2,2,5) et (3,3,8) générées par le PuzzleGenerator

3) Range a comme but de représenter la complexité de la tentative de complétion de la map. Ou, vu d'une autre manière, la distance séparant l'état de réussite à l'état initial proposé par la tentative de complétion.

   Cette notion de distance pourrait avoir plusieurs définitions.

   La plus simple serrait de compter le nombre de pas réalisé par le Player (reverseMovePlayer pouvant aller dans la direction d'un mur, ne faisant donc concrètement pas de pas, la valeur de range selon cette définition ne serait pas nécessairement égale à nbMovePerTry). Cependant, cette vision de range n'est pas très intéressante car un grand Board avec des solutions longues et évidentes auraient alors une valeur de range élevée, contrairement à ce qui est désiré.

   Une autre manière serrait de voir cette notion de distance, et donc de range, comme le nombre de poussée(/tirées) de boite. Mais cette vision de range possède les même défauts (pousser une caisse le long d'un long couloir droit donnerait une grande valeur de range)

   La vision de range choisie par Joshua Taylor et Ian Parberry est donc une amélioration de la précédente : range est incrémenté chaque fois qu'une boite est poussée, mais les déplacements multiples de la même boite dans la même direction ne comptent que pour un seul déplacement (chaque boite ne pouvant apporter que

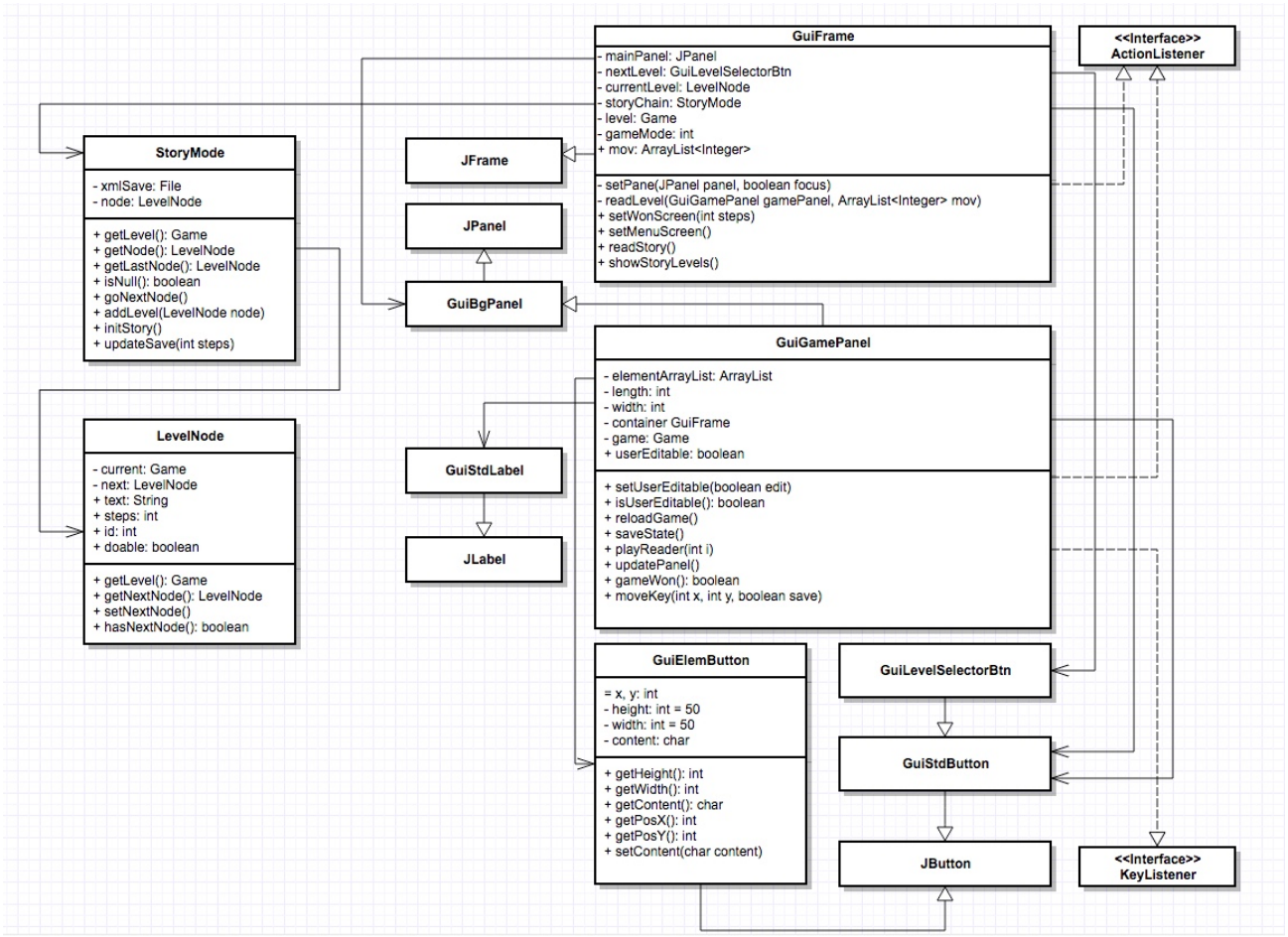
4 à la valeur de range). Cette vision de range se focalise donc plus vers un « nombre de réflexions » qu'un « nombre de déplacement ».

La vision que nous avons choisie pour range lors de notre implémentation est une variante de cette dernière tel que range est incrémenté à chaque déplacement non successif dans une même direction d'une boite. Ainsi, pousser un boite le long d'un long couloir droit (dans une direction X) n'incrémente range que de 1 (au premier déplacement) (équivalent à « 1 réflexion »). Déplacer la boite dans une autre direction ou déplacer une autre boite permettra à nouveau d'incrémenter range, et range sera à nouveau incrémenté si la boite poussée initialement est à nouveau poussée dans la direction X. D'après nos observations (peut être étonnées), cela augmentait légèrement la probabilité d'obtenir une partie intéressante par rapport à la vision précédente. En revanche, elle est plus sensible aux allez retours inutiles que la vision précédente.

Il est important de noter que, peu importe la vision, range n'est qu'une tentative, parfois ratée, de représenter la complexité d'un niveau. Un gros facteur d'erreur est le fait que faire des aller retours inutiles impact à tort la valeur de range (de manière moindre dans la vision de M.Taylor et M.Parberry).

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

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

[2] Interface graphique de l'utilisateur, partie du logiciel qui permet à l'utilisateur d'interférer avec les actions de ce dernier.

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).

3. 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 mouvements 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. De plus, il est également possible pour le joueur de sauvegarder sa partie à tout moment en un simple clic sur un bouton (voir 7. Guide utilisateur de SpeedyRoadie)

- ## 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.

- ## Reprise de la partie en cours

  Si le joueur quitte sa partie avant de l'avoir terminée (imaginons que son ordinateur redémarre sans crier gare), au prochain lancement du jeu, une fenêtre s'ouvrira et demandera au joueur s'il souhaite reprendre sa partie là où il était.

  Le jeu se mettra alors en mode cinématique et rejouera tous ses mouvements avant de le laisser jouer par lui-même.

- ### *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.

4. 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.

- ### *.xsb trop grands ou résolution trop petite…*

  En chargeant des fichiers .xsb de taille trop grande (au-delà de 25 caractères en largeur ou en hauteur) le plateau se charge mais n’est pas entièrement visible par l’utilisateur.

Ceci est dû au fait que nous avons défini une taille fixe aux boutons (50 pixels * 50 pixels).

En chargeant des fichiers trop grands sur des résolutions trop petites (nous travaillons tous les deux en 1920*1080) l'erreur survient.

5. 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.

6. 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.

Comprendre le code de quelqu'un d'autre n'est pas une chose facile, même avec la JavaDoc à côté de soi, nous avons dû nous rencontrer pour nous expliquer toute l'implémentation.

7. 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.

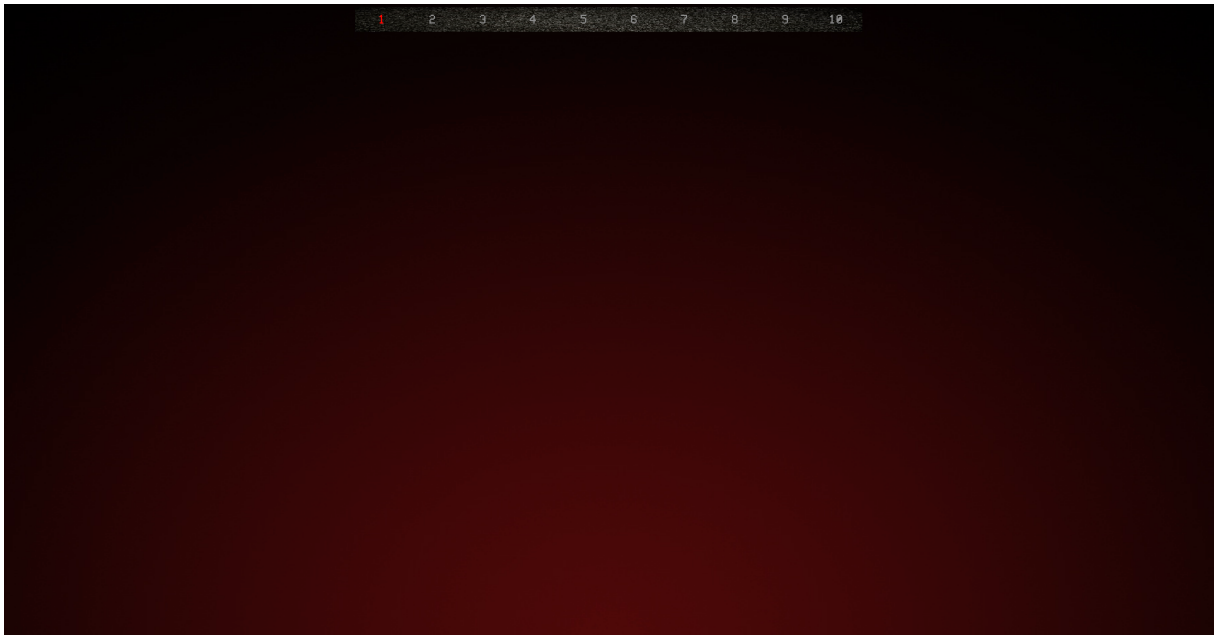
  Utilisez ensuite les commandes "ant build" (le clean se fait automatiquement avec le build) et "ant run" afin de lancer la partie.

- *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
  - Charger 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*

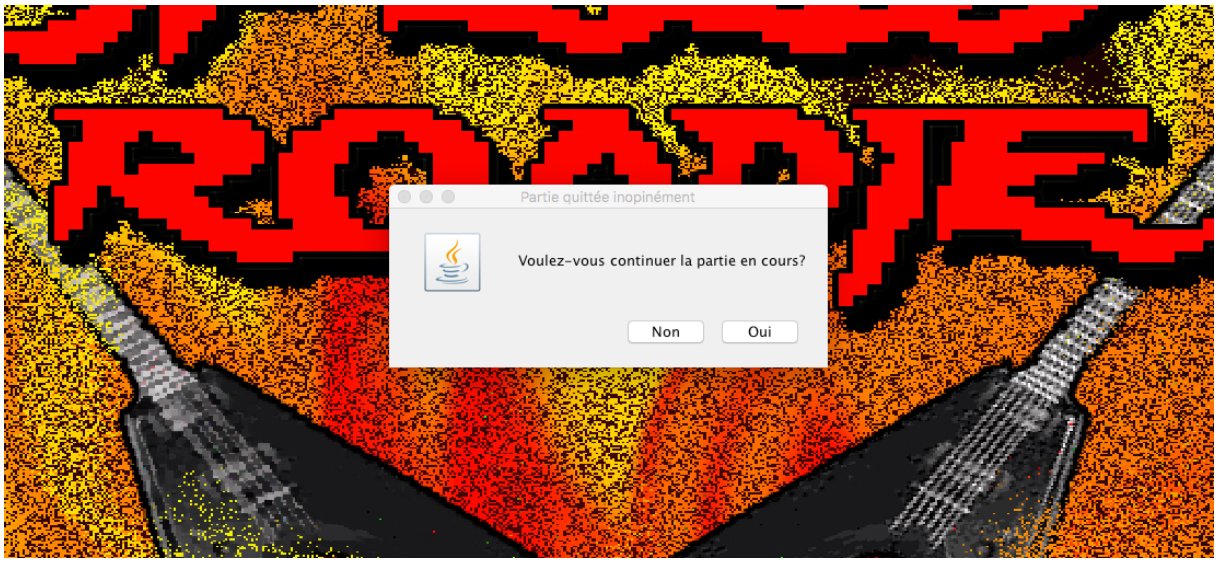
Si vous avez des fichiers de partie personnalisés et/ou un fichier de sauvegarde .mov à charger, vous pouvez cliquer sur le bouton "Charger une partie"

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

- *Continuer une partie en cours*

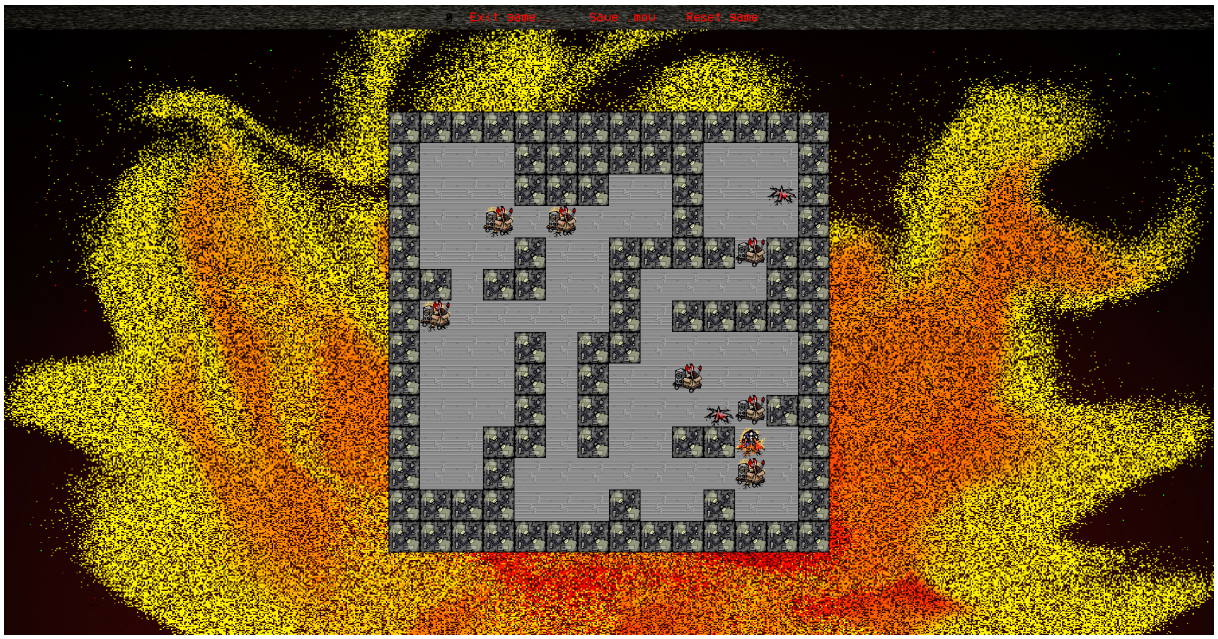
  Si vous avez quitté inopinément votre partie en cours de jeu, vous avez la possibilité de continuer cette dernière.

  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

8. 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 Antoine Fauville qui a dessiné les images du plateau de jeu, les murs, le sol, les caisses, Speedy et les objectifs. Son travail remarquable dans le design des sprites (éléments graphiques du jeu) nous plait énormément et nous préférions travailler avec un ami pour avoir quelque chose de vraiment unique plutôt que de prendre quelque chose qui existe déjà et est vu partout.