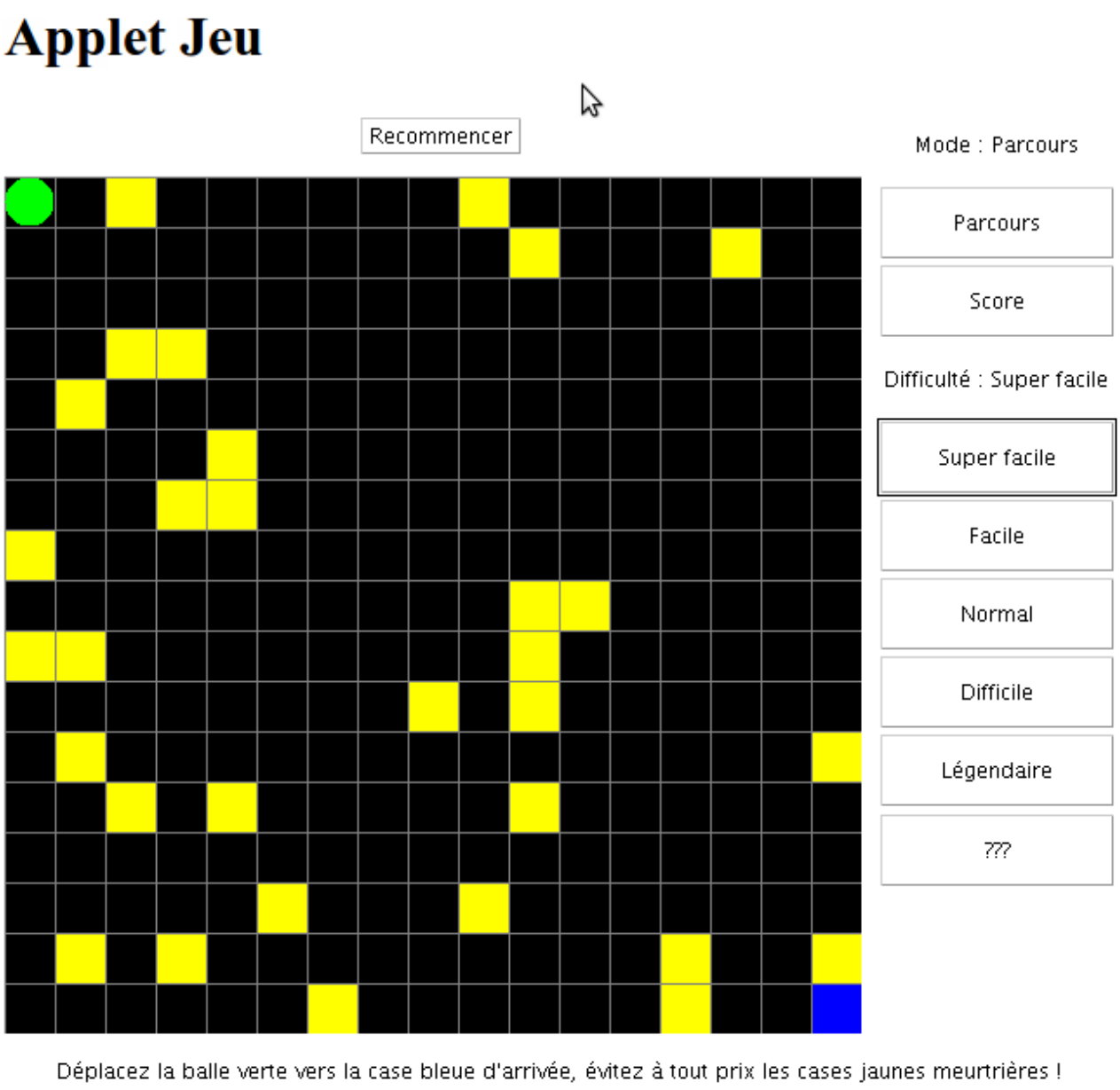
RAPPORT SUR L'APPLET JAVA

Le travail demandé fut de continuer le développement d'une application, codée en Java, dans le cadre d'un travail d'équipe. Au-delà de l'habileté à utiliser les différentes ressources proposées par les librairies disponibles, c'est surtout l'adaptabilité du code source qui a été mise à l'épreuve.

Présentation du projet



Le projet est accessible à l'adresse suivante : http://users.polytech.unice.fr/~mkrtchia/ .

1) Jouabilité

L'application à développer se présente sous la forme d'un jeu. Le joueur manipule via sa souris une balle qui doit atteindre son objectif. Pour pimenter la partie, des obstacles se dressent entre la balle et l'objectif, ces obstacles peuvent bouger voire même changer de forme et le joueur devra faire preuve d'habileté pour réussir la partie.

Ce jeu propose deux modes : le mode Parcours et le mode Score. Chacun de ces modes proposent différents niveaux de difficulté.

1.a) Le mode Parcours

Dans le mode Parcours, la balle doit entrer en contact avec une case d'arrivée bleue. Le chemin entre la balle et cette case est parsemé d'obstacles sous la forme de carrés jaunes qu'il faut à tout prix éviter. Il n'y a que la possibilité de gagner ou de perdre avec de mode de jeu.

1.b) Le mode Score

Dans le mode Score, la balle doit entrer en contact avec des gemmes et faire le plus haut score avant de perdre au contact d'un obstacle. Lorsqu'une gemme rouge (rubis) est assimilée, le score augmente, un obstacle apparaît sur la grille et il faut attraper une autre gemme rouge. Assimiler une gemme bleue (amulette de pouvoir) débloque un pouvoir temporaire pouvant aléatoirement rendre la balle invincible contre les obstacles, ou accélérer ou ralentir le temps. Enfin, les gemmes oranges (amulettes d'absorption) permettent à la balle d'absorber les obstacles limitrophes au lieu d'y être vulnérable, cependant la balle grandira pour chaque bloc absorbé, ce qui peut ensuite être une désavantage...

Dans ce mode de jeu, il n'y a pas de notion de victoire ou de défaite, dès que le jeu est terminé par un faux pas du joueur, son score détermine le degré de sa réussite du jeu.

1.c) Le point sur la difficulté

La difficulté du jeu réside entièrement dans le fonctionnement des obstacles. Plus la difficulté est haute, plus un obstacle peut utiliser d'actions :

- Super Facile : disponible uniquement en mode Parcours, les obstacles n'ont aucun mouvement.
- Facile : les obstacles sont capables de se mouvoir horizontalement ou verticalement.
- Moyen : la vitesse des obstacles est accrue par rapport au mode Facile.
- Difficile : les obstacles peuvent aussi se déplacer de façon oblique.
- Légendaire : les obstacles peuvent changer de forme pour s'arrondir et déployer des pics mortels prêts à embrocher la balle sur une plus grande distance, à tout moment durant la partie.
- Apocalypse : cette difficulté est bloquée, pour l'utiliser il faut connaître le mot de passe qu'on obtient en réussissant le jeu en difficulté Difficile ou Légendaire ou en réalisant un score élevé en mode de jeu score et en difficultés difficile ou légendaire. Le mot de passe est : toast.

  Dans ce mode de difficulté, certains blocs sont atteints d'un comportement viral qui leur confère un mouvement frénétique aléatoire et fait augmenter leur taille jusqu'à ce qu'ils explosent. D'autres cases peuvent alors naitre pour les remplacer.

2) Détail du code source

2.a) Le choix des technologies utilisées

Le langage Java possède de nombreuses librairies, nous nous sommes intéressés à certaines d'entre elles pour soit parvenir à nos fins, soit optimiser le rendu de notre application. Voici celles que nous avons retenu :

- Le double buffering : nous avons recours à cette technologie pour le rendu graphique. Nativement dans notre applet, un objet dont on change la position est « détruit » puis recréé à l'emplacement voulu, et étant donné le nombre élevé d'objets qu'on redessine naturellement et en forçant avec la commande *repaint()* la fréquence d'éléments crées et détruits est trop grande, ce qui provoque un effet de scintillement. Essayé sous Windows©Vista, l'application clignotait plus qu'elle n'était jouable ! C'est là qu'intervient le double buffering qui joue sur deux contextes graphiques (objet de type *Graphics*) : l'un reste en mémoire et c'est sur lui que les objets sont crées sans être affichés, et un autre qui va afficher l'ensemble en une fois par un objet de type Image. On surcharge par ailleurs la méthode *update()* pour qu'elle ne fasse appel qu'à *paint()* et ne fasse rien d'autre.
  On a alors une application bien plus fluide et agréable.

- AWT contre Swing : Nous avons choisi globalement de respecter le sens qu'avait pris le TD, et avons gardé *AWT* pour gérer l'affichage de notre application, car cette librairie étant plus ancienne, elle est plus stable et est reconnue par plus de systèmes anciens. Cependant *Swing* permet de créer souvent plus facilement, une plus grande diversité d'éléments. Nous avons utilisé *Swing* pour générer une boite de dialogue pop-up signalant au joueur s'il a gagné ou a perdu la partie ou tout autre message, et pour récupérer un mot de passe que le joueur est invité à taper. Même si il aurait été mieux de n'utiliser qu'une interface graphique au lieu de les mélanger, n'ayant besoin que d'une boite de dialogue générique, l'utilisation de *JOptionPane* nous a permis un gain de temps certain, en évitant de recréer une classe pour gérer la boite de dialogue. Elle permettait de centrer le pop-up facilement là où avec *AWT* il aurait fallu « réinventer la roue » ce qui dans les temps impartis aurait été trop pénalisant. Cela fait partie des points à réviser dans une version future.
- La gestion d'un thread : Pour que les obstacles puissent se mouvoir pendant que la balle se déplace, on s'est heurté à un problème de taille. Le jeu étant presque entièrement basé sur les mouvements des obstacles, il nous fallait donc une solution et nous l'avons trouvé : un thread, qui permet l'exécution de plusieurs tâches à la fois. Nous avons directement implémenté *Runnable* dans la classe Damier et crée un objet de type thread, que nous pouvons activer avec la méthode *start()* appliquée à cet objet. La méthode *run()* est alors appelée et s'exécute indépendamment des autres actions tels que le mouvement de balle avec la souris. La méthode *run()* est l'une des plus importantes des classes *Damier* et *DamierScore*, elle gère le déplacement de tous les objets autres que la balle et de nombreux tests d'objets annexes pour ne jamais s'arrêter d'elle-même, le contenu de cette méthode étant dans une boucle *while(true)*. Concernant la vitesse de déplacement des cases qui peut différer selon le niveau de difficulté, elle est réglée par un petit temps d'arrêt du thread entre chaque itération de la boucle *while(true)*, à l'aide de la commande *Thread.sleep(temps)*.

2.b) Structure de l'applet

L'applet (et sa classe principale *AppletJeu*) structure l'aspect visuel du jeu disponible sur la page html. Pour manipuler plus facilement les différents éléments tels que les boutons, le damier, les textes etc. nous avons crée plusieurs objets de type *Panel*, et avons structuré l'applet avec un *BorderLayout*.

Le damier de jeu apparaît au centre et est ajouté dans un Panel qui ne contient que cet objet, de manière à pouvoir le vider facilement pour recréer et rajouter un objet de type *Damier* ou *DamierScore*. A droite, un panel structuré avec *GridLayout* permet d'afficher les boutons de choix du mode de jeu et la difficulté. Ces boutons n'étant pas suffisamment nombreux pour remplir toute la longueur de l'applet et garder eux même une longueur agréable, nous avons dû ajouter des objets Label de texte vides. Cette technique n'est pas propre et pourrait pourrait probablement être réglée par l'utilisation du type de structure *GrigBagLayout*, mais nous n'avons pas utilisé ce type par manque de temps. En haut, le bouton Recommencer permet à tout moment de recommencer une partie. Dans le cas d'une partie en mode Parcours, ce bouton utilise la méthode *restart()* de la classe *Damier*, qui remet la balle à sa position de départ. Dans le cas du mode Score, un nouveau *DamierScore* est généré pour permettre de réinitialiser le nombre d'obstacles. Enfin en bas nous avons placé deux objets *Label* qui indiquent les règles du jeu courant.

Pour passer d'un mode de jeu à l'autre, deux méthodes : *newBoutons()* et *newDamier()* permettent de régénérer le bon objet *Damier* ou *DamierScore* et les bons boutons pour chaque mode. Étant donné qu'un jeu statique n'aurait aucun sens en mode Score, la difficulté Super facile, ainsi que le bouton associé ne sont disponibles qu'en mode Parcours.

2.c) Le réglage des difficultés

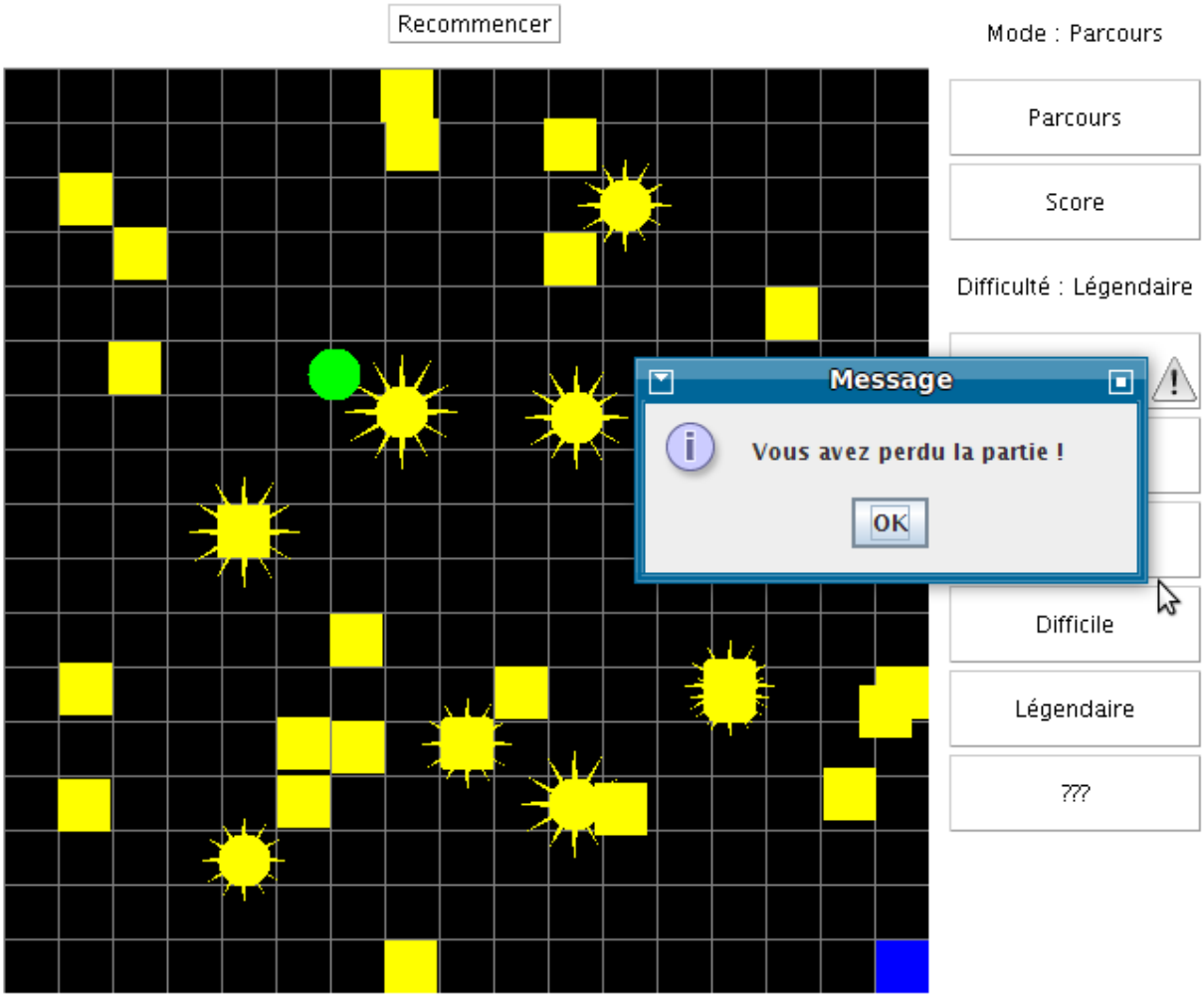
Chaque niveau de difficulté entraine ses propres effets sur le comportement des obstacles. Ces comportements sont tous définis dans la classe *Case* :

- Super Facile : Les obstacles sont fixes, il n'y a donc aucune action d'affiliée.
- Facile : La balle se déplace à l'aide de la fonction *bougerXY()*. Celle-ci agit selon les valeurs de deux booléens : *XY* pour déterminer si le mouvement est vertical (true) ou horizontal (false), et *XX* pour déterminer si la case se déplace vers le haut / la gauche (true) ou vers le bas / la droite (false) du damier.
  Lorsqu'une case entre en contact avec un des bords du jeu, la position de départ de la balle ou la case d'arrivée en mode Parcours, *XX* change de valeur pour inverser le sens du mouvement de la case.
- Moyen : le déplacement est ici similaire à celui du niveau Facile, la même fonction est utilisée. Le mouvement des cases est accéléré, ce réglage se trouve au niveau du thread directement.
- Difficile : En plus de se déplacer horizontalement et verticalement, les cases peuvent aussi se déplacer en diagonale à l'aide de la fonction *bougerXYZ()*. Celle-ci sait s'il faut traiter un mouvement oblique à l'aide du booléen *XYZ*. Celui-ci prend la valeur true s'il faut bien déplacer la case en diagonale et donne alors le travail à la fonction *bougerOblique()*, et la valeur false s'il faut effectuer un mouvement horizontal ou vertical, auquel cas la fonction *bougerXY()* est appelée.
  Comme pour la fonction *bougerXY()*, le mouvement est transformé si la case est au contact d'une paroi, de la case de départ ou de la case d'arrivée si elle existe. Cependant la balle peut aléatoirement soit changer sa direction, soit garder la même direction tout en en inversant le sens du mouvement.
- Légendaire : en plus des mouvements du mode difficile, la case d'obstacle peut se transformer et prendre l'apparence d'une terrible boule de pics. La fonction *bougerL()* est alors appelée pour gérer les actions des cases. Chaque case possède une probabilité faible mais existante de s'animer et se transformer en boule de pics à chaque instant, si un des obstacles possède cette chance ou si elle est déjà en train de se transformer (on peut le savoir si le booléen L est true) ,alors on fait appel à la fonction *pics()* chargée de régir cette animation.
  La fonction pics() possède un scénario en trois étapes : un premier déploiement des *pics()*, puis ils sont rétractés, après ils sont redéployés et le carré changé en rond, enfin après un court temps de stagnation la boule de pics reprend la forme d'un simple carré. Selon l'évolution du

scénario, la fonction pics fait appel à la fonction *dessinerUnPic()* pour dessiner la forme que doivent prendre les pics.
Quoi qu'il arrive, on fait ensuite appel à *bougerXYZ()* pour le déplacement de la case, car se transformer en boule de pics mortelle n'empêche pas à l'obstacle de se mouvoir.

- Apocalypse : les cases ne prennent pas l'apparence de boule de pics dans ce niveau de difficulté. Certaines cases dès leur création peuvent trembler si le booléen *T* est true, environs une case sur quatre. La fonction *bougerT()* est appelée dans ce mode de difficulté. Elle fait appel à *bougerBizarre()* si la case a bien un mouvement frénétique, sinon c'est *bougerXYZ()* qui est appelée.
Lorsque *bougerBizarre()* est appelée, le mouvement de la case est aléatoire. De plus, la taille de la case augmente aléatoirement au fil du temps. Mais lorsque la case atteint une certaine taille, elle s'autodétruit.

Applet Jeu



2.d) Le mode Parcours et le mode Score

Les deux modes de jeu, Parcours et Score, ont été divisés dans deux classes pour plus de clarté. Ils génèrent un damier 17x17 cases et placent 32 obstacles aléatoirement sur la grille. La balle doit aller de la case en haut à gauche à celle en bas à droite qui est colorée en bleu.

Dès le début du jeu, pour les modes super facile, facile, normal et difficile le joueur ne peut pas perdre car les obstacles s'arrêtent avant la première case et évitent également d'entrer dans la dernière case. Ceci permet d'éviter que le joueur soit sur la case Gagner et se fasse piéger par une case jaune cachée derrière celle-ci qui ressortirait tout juste, et permet par exemple éventuellement d'arrêter de jouer sans perdre, en mettant la balle à l'abri dans une case invulnérable pendant un moment. Dans les difficultés supérieures : légendaire et apocalypse, ceci n'est plus garanti.

La fonction *run()* traite l'essentiel du mouvement des cases, des tests de gain ou perte. Dans le mode parcours cependant, en mode super facile, le thread n'étant pas activé puisque les cases sont statiques, c'est la méthode *mouseDragged()* qui s'occupe des tests de collision avec des cases.

Concernant le déplacement de la balle, celle-ci est permise dans le cas où la souris est sur la balle et clique dessus en gardant le clic maintenu. Si le clic est relâché, ou si le joueur a gagné ou perdu, la balle arrêtera de se déplacer. Si le joueur sort de la grille de jeu en maintenant la balle en son contrôle, la balle s'arrêtera au bord de la grille de jeu en débordant de moitié tout en restant sous le contrôle du joueur tant que le bouton du clic reste maintenu. Ceci est pratique au niveau de la jouabilité pour ne pas casser la dynamique de jeu dans le cas de sortie du terrain de la souris.

Concernant le mode Score en particulier, les obstacles n'étant pas en nombre défini, et étant potentiellement infinis, nous avons décidé d'utiliser une liste pour les représenter plutôt qu'un tableau : nous avons utilisé un objet du type *ArrayList* pour ça.

La classe *DamierScore* est dépourvue de la gestion du mode d'obstacles statiques. Il incorpore plusieurs autres éléments tels que les rubis qui sont les éléments principaux à récolter, les amulettes de pouvoir et d'absorption qui permettent d'utiliser un pouvoir pendant quelques secondes etc.

L'amulette de pouvoir bleue apparaît avec une probabilité de 1 fois sur 5 rubis capturés. Elle permet l'utilisation de 3 pouvoirs différents choisis aléatoirement lorsqu'elle est capturée. Ces pouvoirs durent quelques secondes et consistent à soit diviser la vitesse du jeu (donc du déplacement des obstacles) par 3, soit multiplier la vitesse du jeu par 3, ou encore de devenir invincible. L'un de ces pouvoir est désavantageux puisqu'il accélère le jeu.

L'amulette d'absorption orange apparaît sur le terrain avec une probabilité de 1 fois sur 8 rubis capturés. Elle permet à la balle d'absorber pendant quelques secondes tout obstacle qui sera touché par celle-ci. Ce pouvoir a cependant un désavantage, il fait grossir la balle à chaque fois qu'elle absorbe un obstacle. Le pouvoir disparaît quand la balle reprend sa couleur verte.

Pour finir...

Ce projet nous a permis de découvrir plusieurs outils, comme le double buffering ou la gestion de threads. Nous aurions pu continuer à implémenter de nombreuses fonctionnalités au jeu. Quelques idées parmi tant d'autres :

- Ajout de musique et de sons car l'ambiance sonore influe sur le comportement du joueur;
- Remplacer les figures par des images : Le résultat est plus beau, plus appréciable par le grand public;
- Manipulation de la balle au clavier, pour diversifier l'utilisation de l'applet;
- Animer l'explosion d'un obstacle nerveux en difficulté Apocalypse et faire qu'un de ces objets ait des probabilités de se détruire en fonction de sa taille.

Mais le temps nous manque pour réaliser toutes ces extensions. Cependant le nombre de fonctionnalités implémentés a mis le code source à rude épreuve, pour le rendre le plus souple possible.