Loïs Aubree - Lucie Boutou
Benjamin Guillet - Théophile Madet

Projet de LO43

Réalisation d'un jeu AngryBirds-like

Rapport de projet

Automne 2011

Table des matières

Chapitre 1

Présentation du projet

Nous avions ce semestre en LO43 la possibilité de développer un jeu ludique similaire à Angry Birds.

Dans notre version, le joueur a à sa disposition un ensemble d'oiseaux «en colère» , et doit tuer les ennemis à l'aide d'œufs pondus en vol. Les oiseaux peuvent également se sacrifier comme dans Angry Birds en réalisant une «attaque suicide» et ainsi tuer les ennemis avec leur propre corps.

Le joueur a au départ entre 3 à 5 oiseaux, selon la difficulté du niveau. Chaque type d'oiseaux possède des capacités de ponte (nombres d'œufs) et des possibilités physiques qui lui sont particulières. Le score du joueur est calculé d'après son nombre d'oiseaux restants à la fin du niveau. Celui-ci est fini quand tous les ennemis sont morts.

Le jeu est organisé en suivant des difficultés qui vont de facile à extrême, et chacune d'elle possède un nombre défini de niveaux. Seul le premier niveau de chaque difficulté est au départ disponible, les niveaux suivants sont débloqués au fur et à mesure de la progression du joueur. Ce dernier ne peut accéder au niveau suivant qu'en ayant validé le précédent.

Chapitre 2

Organisation et répartition du travail

Notre groupe était composé de quatre personnes, nous avons régulièrement organisé des réunions chez les uns et les autres pour avancer le projet et faire le point.

La partie spécification UML a été faite une première fois tous ensemble afin de se mettre d'accord sur une base de départ puis a évolué pour mieux coller au cahier des charges : respect du pattern MVC, utilisation des threads et du polymorphisme, etc.

Nous avons utilisé **Eclipse**, **Skype** et un dépot **GitHub** pour la synchronisation et la gestion des versions. **GitHub** possède également un suivi des fonctionnalités à ajouter ou à corriger. Ce qui a permis à chacun de travailler les classes et fonctionnalités nécessaires qui lui plaisaient.

Globalement on peut dire que le travail s'est réparti ainsi :
- Benjamin : base du projet (respect du pattern MVC), gestion du joueur ;
- Loïs : collisions et envoi des oiseaux ;
- Lucie : décors et menus ;
- Théophile : gestion des sauvegardes, collisions.

Mais tout le monde a par moment débordé sur le «secteur» des autres pour corriger un bug ou améliorer une fonctionnalité. Rien n'était fixé.

Chapitre 3

Spécifications

3.1 Diagramme d'utilisation

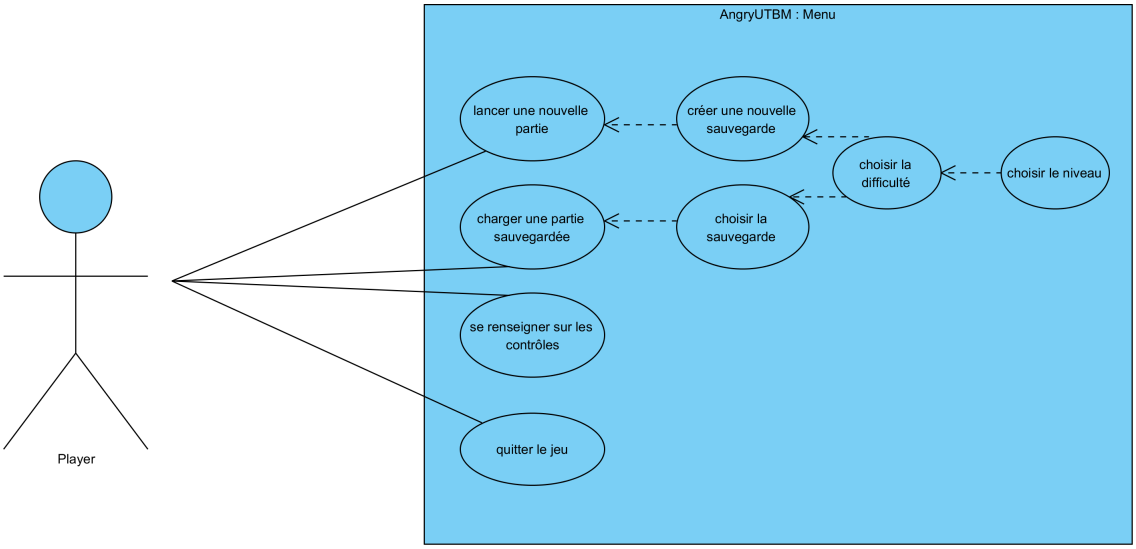

FIGURE 3.1 – Diagramme d'utilisation du menu

Diagramme d'utilisation du menu : (voir figure 3.1)

Lorsque le joueur lance le jeu, il a plusieurs choix possibles : lancer une nouvelle partie, charger une partie sauvegardée, se renseigner sur les contrôles de jeu ou quitter le jeu.

Si le joueur lance une nouvelle partie, il devra renseigner son nom afin qu'une sauvegarde soit créée. S'il décide de charger une partie sauvegardée, il devra choisir parmi les noms des joueurs possédant déjà une sauvegarde.

Le joueur peut ensuite accéder au choix de difficulté puis de niveau.

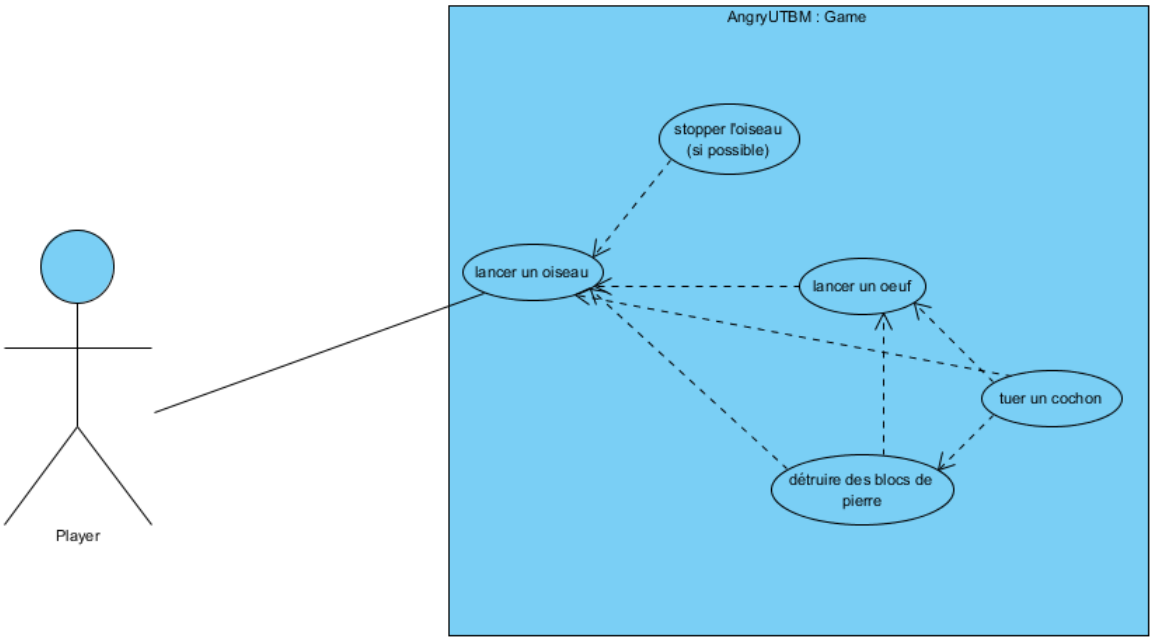

FIGURE 3.2 – Diagramme d'utilisation du jeu

Diagramme d'utilisation du jeu : (voir figure 3.2)

Après avoir choisi un niveau le jeu se lance. Le joueur doit donc lancer un oiseau.

Il peut, selon les caractéristiques de l'oiseau, décider de stopper celui-ci pendant son vol. L'oiseau peut aussi lancer un (ou plusieurs selon le type d'oiseau) œufs.

Le joueur peut détruire des blocs de pierres présents sur le décor à l'aide des œufs ou bien directement avec l'oiseau lui-même.

Enfin, l'utilisateur peut aussi tuer des cochons toujours à l'aide des œufs ou des oiseaux eux-même. Mais il peut être nécessaire parfois de détruire des blocs de pierre pour pouvoir atteindre certains cochons.

3.2 Diagramme de classes

Respect du pattern MVC : (voir figure 3.3)

L'architecture (le modèle) du programme réalisée dans ce projet est de type modèle-vue-contrôleur.

La fenêtre du jeu `GameFrame` héritée de `JFrame` est la classe principale du jeu. Elle créée et stocke les classes du modèle-vue-contrôleur.

`GameView` est la classe correspondante à la vue du pattern MVC. Elle va se contenter d'afficher les entités de la liste. Elle n'effectue aucun traitement.

`GameModel` est la classe (modèle du pattern MVC), elle possède la liste des entités du jeu. C'est là où l'ensemble des traitements est effectué comme la mise à jour des positions des entités ou le test des collisions : `GameModel` va parcourir la liste des entités en permanence pour effectuer les modifications qu'apporte le joueur ou les événements du jeu.

`GameController` est la classe (contrôleur dans le pattern MVC) qui va écouter(listener) les événements (clavier, souris, changement dans la liste des entités). Elle aura pour but de récupérer ces événements et d'appeler les fonctions de `GameModel` (ou des entités) correspondantes.

Le menu, lui, est de type vue-contrôleur. Une vue est crée pour chaque page du menu dans une classe adaptée (non représenté sur le diagramme de classes),et toutes héritent de la classe vue du menu `GameViewMenu` (abstraite). Le `MenuController` récupère les différents événements, et pour chacune des vues, il sera chargé de faire le changement des pages du menu.

Objets actifs : (voir figure 3.4)

Tous les objets capables d'interagir au sein de notre application sont créés dans des classes adaptées héritant de la classe `Entity`. La classe `Entity` possède une méthode abstraite `move()` qui sera redéfini dans chaque classe enfant, avec les caractéristiques personnalisées du déplacement de l'élément concerné.

Les différents types d'oiseaux sont créés dans les classes correspondantes. Elles héritent toutes de la classe `Bird` qui contient les attributs et méthodes communs à tous les oiseaux.

Le même schéma est répété pour les ennemis, les différents types de cochons sont créés dans leur classe respective qui hérite elle-même de la classe `Ennemy`.

Cette implémentation nous permet de parcourir une unique liste d'entités, la liaison dynamique s'occupant ensuite d'appeler la bonne méthode `move()`.

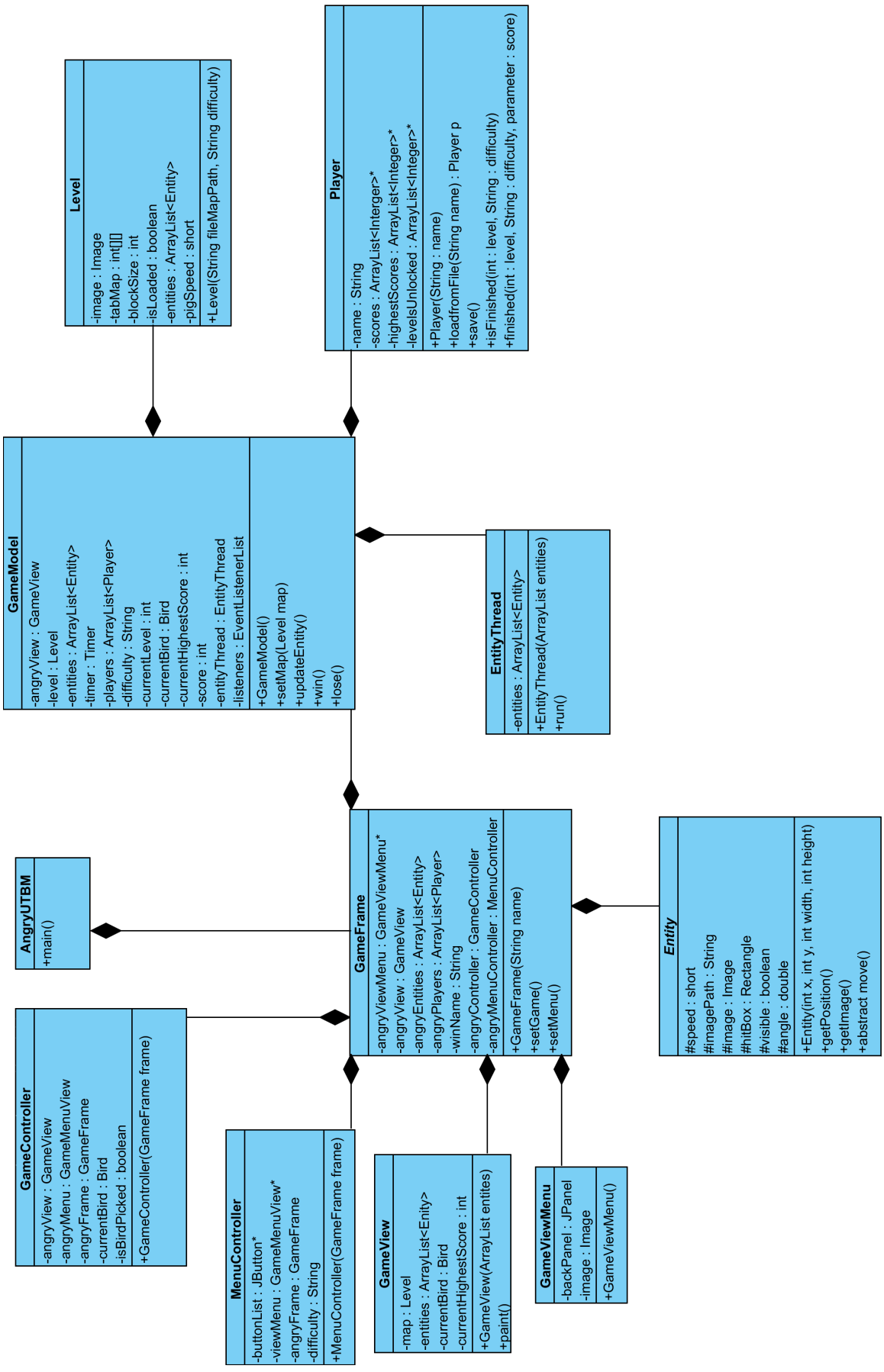

FIGURE 3.3 – Diagramme des classes

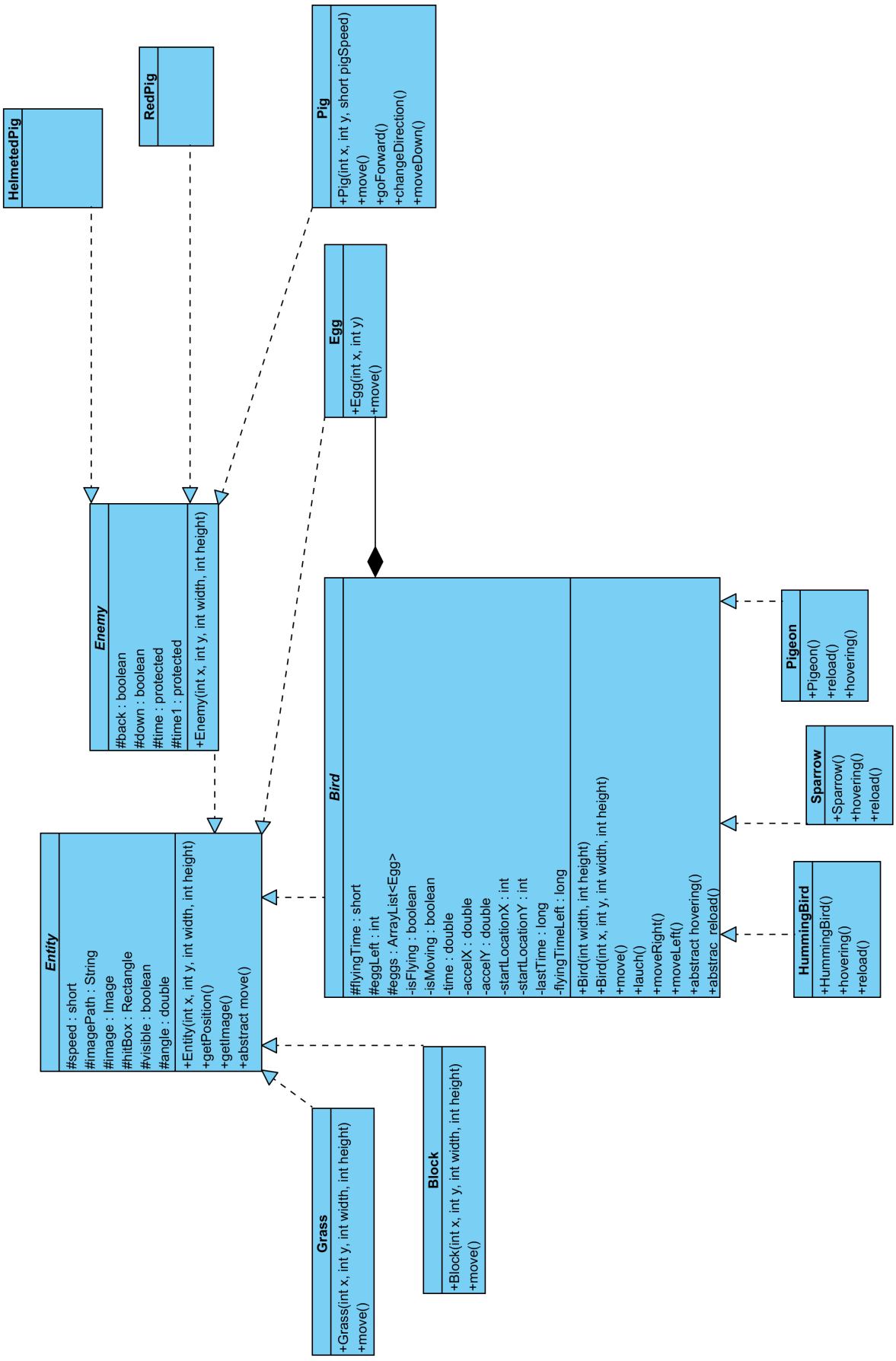

FIGURE 3.4 – Diagramme des classes

Chapitre 4

Conception et prise en main

4.1 Gestion du menu

La gestion du menu s'effectue au travers de 6 panneaux différents :
- `MenuHomeView`, page d'accueil au lancement du jeu.
- `MenuNewView`, page de création d'une nouvelle partie.
- `MenuLoadView`, page de chargement d'une partie.
- `MenuControlsView`, page d'informations sur les contrôles du jeu.
- `MenuDifficultyView`, page de choix de la difficulté du jeu.
- `MenuLevelView`, page de choix du niveau.

Chaque panneau est implémenté dans une classe propre à ses caractéristiques héritant de la classe `GameViewMenu` (abstraite) qui comporte les caractéristiques communes à tous les panneaux telle que le background du Menu.

La classe `GameViewMenu` hérite elle-même de la classe `JLayeredPane`, qui permet d'indiquer aux éléments placés sur le panneau un index de position en profondeur, ce qui nous autorise notamment à placer nos boutons par-dessus notre background.

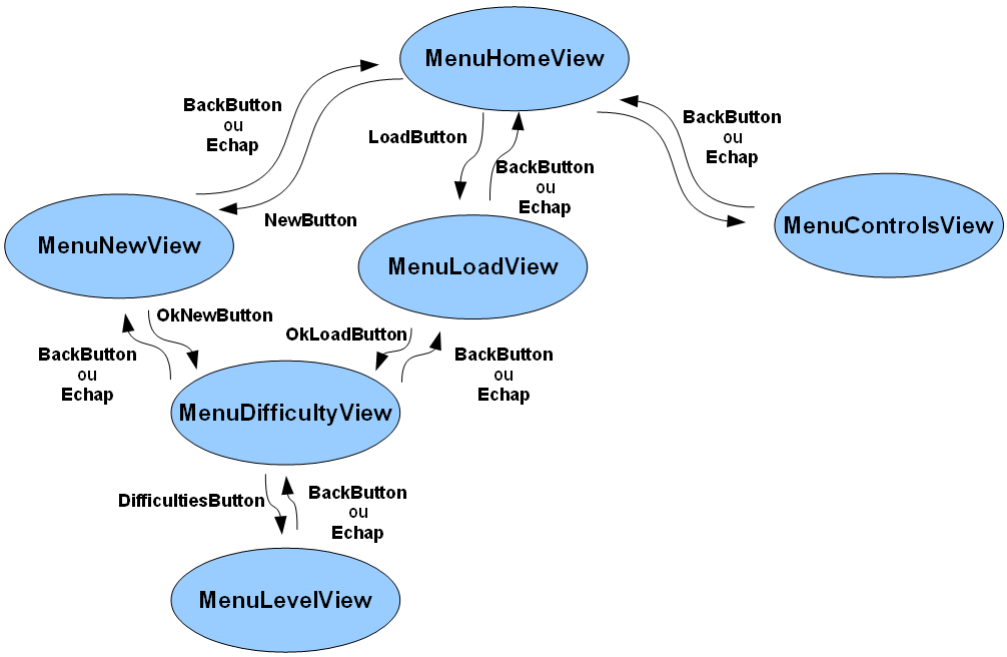

FIGURE 4.1 – Navigation dans le menu

La navigation dans le menu se fait au travers des différents boutons avec la souris, et pour le retour en arrière nous avons mis en place un contrôle supplémentaire par le clavier au moyen de la touche «échap» (équivalent au retour en arrière en utilisant le bouton «back» à la souris).

Il y a alors détection du panneau sur lequel on se trouve afin de charger le panneau «précédent» correspondant.

Pour le cas du panneau de choix de difficulté, l'accès au panneau a pu être réalisé depuis le panneau de création d'une nouvelle partie ou depuis le chargement d'une partie. C'est pourquoi une variable a été implémentée (`parentPanel`) afin de garder en mémoire depuis quel panneau a été fait le dernier accès au panneau de choix de difficulté.

4.2 Gestion des niveaux

La création d'un niveau du jeu s'appuie sur la lecture d'un fichier texte qui doit contenir les informations suivantes :
- les oiseaux disponibles pour réaliser le niveau.
- la position des différents blocs représentant le décor.
- la position des cochons dans ce décor.

On retrouve en tête du fichier la liste des oiseaux. Un retour à la ligne est effectué après chaque oiseau disponible dans le niveau afin de faciliter la lecture du fichier. De plus les oiseaux sont indiqués dans le même ordre que leur ordre d'apparition dans le niveau.

Un test d'égalité a lieu entre la chaîne de caractères contenue dans la ligne et les différents types d'oiseaux, afin de créer l'oiseau correspondant et de l'ajouter à un `ArrayList` d'entités.

Lors de la lecture du fichier texte, la rencontre du mot «Map» permet au programme de savoir que la liste des oiseaux disponibles pour le niveau est complète et que la suite du fichier contient la carte représentant le niveau.

Il s'agit de lignes de texte de taille égale contenant différents caractères qui seront implémentés dans un tableau de deux dimensions (chaque ligne du fichier texte correspond à une ligne du tableau 2D).

Le tableau à une taille adaptable aux dimensions de notre fenêtre et de nos éléments de décor. Chaque élément du décor a une image correspondante de 26*26 pixels, et pour correspondre ici à une fenêtre de taille 1200*600 pixels, une ligne du tableau contient 47 éléments et une colonne en contient 22.

Chaque caractère correspond à un élément différent sur le décor :
- le «0» indique que rien ne se trouve à cet endroit de la carte.
- le «1» indique qu'il y a un bloc d'herbe à cet endroit de la carte.
- le «2» indique qu'il y a un bloc de pierre à cet endroit de la carte.
- le «3» indique la position de départ d'un cochon ennemi.

Une lecture du tableau est ensuite réalisée et à chaque «1» , «2» , ou «3» rencontré, on créé l'entité correspondante (cochon, bloc d'herbe ou bloc de pierre). L'entité est créé via un constructeur prenant en paramètre sa position x (calculée à l'aide de l'index des colonnes du tableau et de la taille de l'image correspondante à l'entité) et sa position y (calculée à l'aide de l'index des lignes du tableau et de la taille de l'image correspondante à l'entité).

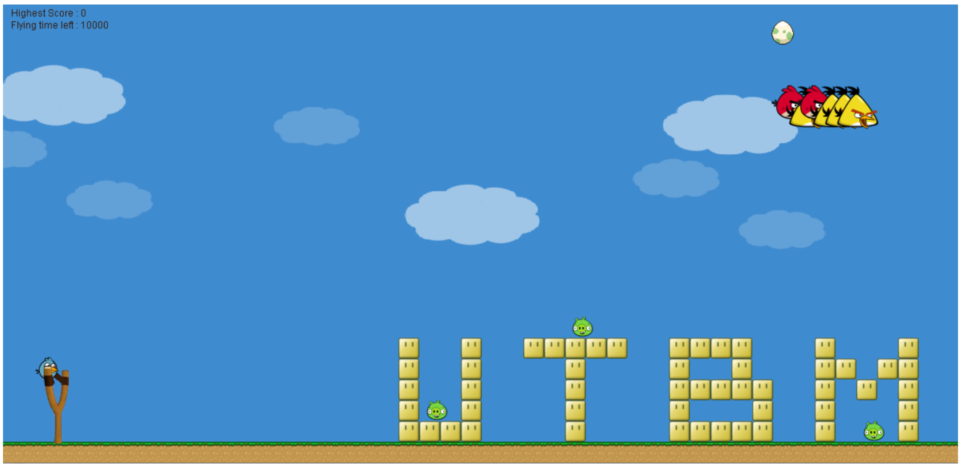

FIGURE 4.2 – Screenshot niveau 1

```
Sparrow
Humming Bird
Pigeon
Humming Bird
Pigeon
Pigeon
Pigeon
Map
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000
00000000000000000000000000030000000000000000000
00000000000000000002002002222200222200020002000
00000000000000000002002000020000200200022022000
00000000000000000002002000020000222220020202000
00000000000000000002302000020000200020020002000
00000000000000000002222000020000222220020302000
11111111111111111111111111111111111111111111111
```

FIGURE 4.3 – Fichier texte niveau 1

La figure 4.2 est le fichier texte correspondant au niveau représenté en figure 4.1. On remarque que la dernière ligne ne comportant que des «1» concorde bien

à une ligne de blocs d'herbe sur la partie basse de notre fenêtre. De même les «2» concordent à des blocs de pierres et les «3» à des cochons.

On aperçoit dans le coin en haut à droite de la figure 4.1 les oiseaux disponibles dans le niveau, coïncidant aux premières lignes du fichier texte de la figure 4.2.

D'une manière générale on remarque que la création de nouveaux niveaux ou la modification de niveaux existants est relativement simple.

4.3 Gestion des collisions

La gestion des collisions et les événements qui en découlent sont effectués dans la fonction `updateEntity()`.

Chaque entité créée possède ce qu'on appelle une `hitBox` ; c'est-à-dire que chaque objet graphique est englobé d'un rectangle de collisions.

Pour assurer le traitement des collisions, on parcourt la liste des entités du `GameModel`. Ensuite, suivant l'entité courante traitée, on gère ses collisions avec les autres entités en testant si sa `hitBox` est en intersection avec la `hitBox` des autres.

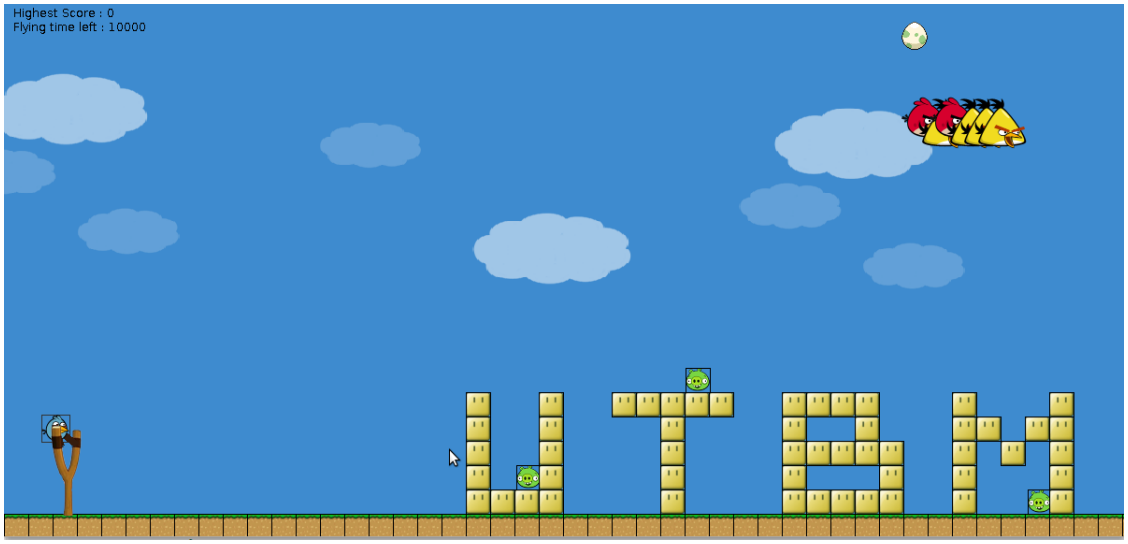

FIGURE 4.4 – Niveau 1 avec hitBoxes visibles

Si la collision d'un élément avec un autre doit entraîner la suppression d'un des deux objets, le programme ajoute celui ou ceux-ci dans une liste d'entités (appel à `toRemove()`). Celle-ci est parcourue à son tour en fin de fonction pour effectuer la suppression des entités dans le `GameModel`. Cela permet de supprimer toutes les entités concernées au même niveau dans l'algorithme et aussi de s'assurer que tous les traitements sur les entités ont été réalisés avant d'en supprimer.

Les différentes collisions gérées sont les suivantes :

Collision des œufs :

- avec les cochons : lorsque la collision est détectée, on ajoute le cochon et l'œuf courant dans la liste des entités à supprimer.
- avec les blocs de pierre : la collision provoque l'ajout des entités bloc de pierre et œuf concernées dans la liste des entités à supprimer.
- avec les blocs d'herbe : la collision provoque l'ajout simple de l'œuf courant dans la liste `toRemove()`. Le bloc d'herbe étant indestructible.

Collision des cochons :

- avec les blocs de pierre :
  - gestion de la collision : la gestion de la collision est différente, ici, puisqu'elle ne se fait pas sur le rectangle de collision du bloc de pierre avec celui du cochon. En effet, les cochons peuvent se déplacer librement sur les blocs de pierre, mais, lorsqu'ils percutent un bloc de pierre par le côté, la collision doit provoquer le changement de direction dans le déplacement du cochon. On doit être en mesure de tester avec quelle arête de la `hitBox` du bloc de pierre, la `hitBox` du cochon est en intersection.

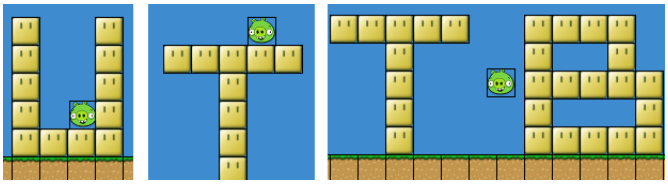

FIGURE 4.5 – Différentes collisions des cochons

  - événement : lorsque le cochon touche une des arêtes gauche ou droite d'un bloc de pierre, le cochon change de direction de déplacement (gauche ou droite) ; lorsque le cochon n'est pas en collision avec une seule des arêtes supérieures d'un des blocs de pierre (c'est-à-dire que le cochon se déplace sur aucun des blocs de pierre) il tombe, La chute du cochon se fait jusqu'à ce qu'il rencontre l'arête supérieure d'un des blocs (de pierre ou d'herbe) ; la gestion avec l'arête inférieur des blocs n'a aucun intérêt ici.
- avec les cochons : lorsque deux cochons se rencontrent, cela provoque le changement de direction des deux cochons concernés.

Collision des oiseaux :

- avec les blocs de pierres : lorsqu'un oiseau entre en collision avec la `hitBox` d'un bloc de pierre, cela provoque l'ajout de ces deux entités dans la liste des suppressions. On peut une nouvelle fois justifier ici le choix du traitement des suppressions en fin de fonction car la `hitBox` des oiseaux est assez grande pour entrer en collision avec plusieurs blocs à la fois. Seulement, si la suppression de l'oiseau était faite directement avec la détection de la collision pour le premier bloc, les autres blocs aussi en collision ne seraient pas supprimés car ils seraient non testés.
- avec les cochons : la collision d'un oiseau et d'un cochon provoque l'ajout de ces deux entités dans la liste des suppressions.

Dans la fonction `updateEntity()`, il est aussi géré la visibilité des entités lorsque la position de celles-ci dépasse les bords de la fenêtre (gauche, droite ou bas). L'élément concerné est ajouté dans la liste des suppressions.

4.4 Gestion des oiseaux

Tous les oiseaux héritent de la classe abstraite `Bird`.

Création de l'oiseau courant :

L'oiseau courant, celui qui est positionné au lance pierre est le premier oiseau trouvé en parcourant la liste des entités dans l'ordre.

Pour définir un oiseau comme courant, on parcoure la liste des entités et lorsque la première instance d'un `Bird` est détectée, le pointeur `currentBird` est placé sur cette instance.

Vie de l'oiseau :

Lorsqu'un oiseau est courant, il possède deux phases : une première de préparation au vol et une seconde de vol.

- Préparation au vol : l'oiseau est positionné sur le lance-pierre, le joueur a alors la possibilité de tendre l'élastique du lance-pierre pour déterminer la vitesse initiale et l'angle initial avec lequel le lancer de l'oiseau est effectué. Pour cela, le joueur doit cliquer sur l'oiseau situé sur le lance pierre puis effectuer un glissé tout en gardant le bouton de la souris enfoncé. Lorsque le joueur lâche le bouton de la souris, l'oiseau passe en mode vol (le booléen `isFlying` passe à true).

FIGURE 4.6 – Lancement de l'oiseau

- Phase de vol : l'oiseau en vol est soumis à la formule d'accélération, une accélération de 9.81 (simulant la gravité) est appliquée en Y. La position de l'oiseau est modifiée en fonction du temps, de l'angle initial imposé et de la vitesse initiale imposée. En vol, le joueur a la possibilité de lâcher des œufs (dans la limite du nombre d'œufs attribués : `eggLeft`). Le joueur peut aussi, si le type d'oiseau le permet (pigeon ou colibri) effectué un vol stationnaire.

A ce moment là, l'oiseau arrêtera son vol puis le joueur aura la possibilité de reprendre le cours du vol dans la limite du temps de vol restant pour l'oiseau.

Mort de l'oiseau courant :

Lorsque l'oiseau courant meurt (collision avec un bloc, collision avec un cochon, plus visible dans la fenêtre ou temps de vol terminé), l'oiseau est ajouté dans la liste des suppressions de la fonction `updateEntity()` puis il est supprimé. La liste des entités est alors à nouveau parcourue pour déterminer le nouvel oiseau courant.

4.5 Gestion de la victoire ou défaite d'un niveau

Chaque niveau possède un nombre d'ennemis (cochons) définit. Le jeu est gagné lorsque tous les cochons du niveau ont été tués (supprimer de la liste des entités).

Le test de victoire est effectué dans la fonction `updateEntity()`. Lorsqu'on parcourt la liste des entités, on effectue la mise à jour du comportement des cochons, or si la liste des entités ne comporte plus aucune instance de cochon alors le niveau est terminé.

Si le niveau est terminé, le programme entre dans la fonction `win()` qui se chargera d'effectuer tous les traitements de la victoire.

De la même façon que pour la gestion de la victoire, l'événement de défaite est provoqué dans la fonction `updateEntity()`. Mais, cette fois, le test n'est pas effectué sur les instances de cochons mais sur les instances des oiseaux dans la liste des entités. Lorsqu'il n'y a plus d'oiseaux dans la liste, la fonction `lose()` est appelée.

Dans ce cas il est important de faire les traitements sur les cochons en premier, pour qu'il n'y ait pas d'appel à la fonction `lose()` lorsqu'il n'y a plus de d'oiseaux ni de cochons (l'appel à la fonction `win()` passera donc en priorité).

4.6 Gestion du joueur et des sauvegardes

Il nous était demandé de sauvegarder la progression du joueur pour lui permettre de reprendre la partie là où il en était plus tard. Nous avons donc mis en place un système de «profil» : le dossier `save` contient un fichier de sauvegarde par joueur, qui porte son nom. Ce fichier contient la sérialisation de l'objet correspondant au joueur. La classe `Player` a été conçue spécifiquement pour ce besoin.

Le joueur a donc la possibilité de choisir parmi la liste des profils déjà sauvegardés, ou d'en créer un nouveau.

Données associées à un joueur :

Pour chaque joueur, on souhaite stocker son nom, les niveaux qu'il a réussi, ainsi que les scores obtenus. La classe `Player` contient donc un champ de type `String` pour le nom et deux `ArrayList` par niveau de difficulté, une contenant les niveaux débloqués pour la difficulté correspondant, l'autre contenant les scores pour les niveaux réussis.

Sérialisation :

Pour sauvegarder les données entre les parties, il était nécessaire de sérialiser la classe `Player`. Pour cela, le plus simple a été d'utiliser les facilités offertes par Java avec les `ObjectOutputStream` et `ObjectInputStream`. Nous n'avions ainsi pas besoin de nous occuper des détails d'implémentation de la sérialisation des objets, ou de leur lecture depuis un fichier.

Cette technique empêche la lecture des fichiers de sauvegarde par d'autres programmes que le notre, mais cela n'est pas un problème pour ce projet.

Un autre problème pourrait venir du nom des joueurs : en effet, les fichiers de sauvegarde sont nommés «nom.save» , or aucune vérification n'est faite sur les noms entrés lors de la création d'un nouveau profil.

Scores :

Seuls les plus hauts scores sont conservés pour un joueur et un niveau donné. Lorsque le joueur termine un niveau, on appel la méthode `finished()` de la classe `Player`, avec en paramètres le niveau, la difficulté et le score. `finished()` se charge en suite de vérifier si le score donné est un nouveau record.

Le score correspond au nombre d'oiseaux restants une fois le niveau réussi.

Chapitre 5

Bilan

5.1 Conclusion

Ce projet était très intéressant et motivant. Son côté ludique nous a permis d'apprendre le langage Java avec une vraie motivation et une bonne ambiance. Un groupe important comme le notre nécessite une bonne communication afin que chacun sache le travail à accomplir et éviter les répétitions. Heureusement aujourd'hui nous avons de nombreux outils à notre disposition pour nous faciliter la tâche, en plus des classiques réunions ou e-mails.

5.2 Améliorations possibles

Un projet comme celui-ci est toujours perfectible, voici quelques-unes des améliorations qui nous viennent en tête et que nous aurions bien voulu implémenter :

- il serait facile d'ajouter une musique d'ambiance, des bruits lors des collisions ou lorsque l'on gagne ou perd.
- on pourrait afficher le meilleur score du joueur actuel sous chaque niveau.
- un ajout de contrôle sur le nom des profils serait nécessaire, pour ne pas rencontrer de problèmes à la création du fichier de sauvegarde.
- une diversité des ennemis pourrait facilement être implémentée.
- une amélioration de la chute des cochons serait envisageable, celle-ci ne dépend actuellement pas de la gravité et ne possède aucune accélération.