TER FMIN200

–

Développement d'un jeu de type Bomberman en réseau sous Android et iOS

BONVILA Olivier COUSEIN Kilian PITIOT Ludovic
TARDIEU Benjamin

Nous tenons à remercier

Table des matières

Chapitre 1

Introduction

Chapitre 2

Présentation

- Nous + Tuteur

2.1 Projet

- On l'a proposé
- Principe du projet et description
- Rapprochement avec nos cours (I2A, Casar, Diwed)
- Buts / Attentes recherchés

2.2 Plate-formes de développement

2.2.1 Android

Android est un système d'exploitation open source fournis par *Google* et developpé par *Android*. Il est majoritairement utilisé sur smartphones, PDA et autres terminaux mobiles mais aussi sur tablettes graphiques et même sur certains téléviseurs.

Disponible via une licence Apache version 2 [1], Android est fondé sur un noyau Linux, il comporte une interface spécifique, développée en Java, les programmes sont exécutés via un interpréteur JIT, toutefois il est possible de passer outre cette interface en programmant ses applications en C ou C++, mais le travail de portabilité en sera plus important.

Il nous est aussi permit d'utiliser du XML principalement pour les interfaces graphiques ou encore pour la portabilité des données entre chaque type d'architecture.

Sa première version date du 5 novembre 2007, actuellement la dernière version disponible est la 3.1 Honeycomb (Rayon de miel) mais nous utiliserons ici par soucis de compatibilité la version 2.0 dans le but de toucher un maximums de périphériques.

1. La licence Apache est une licence de logiciel libre et open source. Elle est écrite par l'Apache Software Foundation, qui l'applique à tous les logiciels qu'elle publie. Il existe plusieurs versions de cette licence (1.0, 1.1, 2.0). Cette licence n'est pas copyleft.

Afin de pouvoir développer nos propres applications, *Android* met à dispotition un kit de développement (SDK) ainsi qu'un greffon pour Eclipse (ADT) afin de simplifier l'utilisation de celui-ci.

Le kit de développement d'Android comporte les outils de base pour developper une application en Java, le developpement en C et C++ necessitant d'utiliser le NDK [2]. Il se compose des librairies principales d'Android, de divers exemples d'applications et surtout d'un emulateur de terminal Android qui permet de pouvoir tester directement ses applications directement sur Linux, Mac ou Windows sans necessairrement avoir besoin d'un terminal mis à part pour OPENGL-ES 2 qui n'est pas encore prit en charge par celui-ci, sinon le kit de développement permet très bien de tester ses applications en temps réel sur tout peripheriques tournant sur Android que ce soit en liaison directe avec un ordinateur ou en récupérant l'archive de l'application au format APK.

Le format APK ou Android Pakage est une variante du format JAR [3], est utilisé pour la distribution et l'installation de composants regroupés sur le système d'exploitation Android.

2.2.2 iOS

Apple est une entreprise multinationale américaine qui a été crée le 1er avril 1976 à Cupertino (Californie) par Steve Jobs et Steve Wozniak. Elle a pour but de conçoir et vendre des produits électroniques grand public, des ordinateurs personnels et des logiciels informatiques. Parmi les produits les plus répendus de l'entreprise se trouvent les ordinateurs Macintosh (1984), l'iPod [4] (2001), l'iPhone [5] (2007) et l'iPad [6] (2010). Apple a développé deux systèmes d'exploitation (Mac OS X [7] et iOS) permettant de contrôler ses différents produits, notamment les ordinateurs Macintosh, l'iPhone et l'iPod touch [8].

iOS, connu sous le nom de iPhone OS avant Juin 2010, est le système d'exploitation mobile d'Apple. Ce dernier a été développé originellement pour l'iPhone, puis a été étendu pour l'iPod touch, iPad et Apple TV [9] (2007). C'est dérivé de Mac OS X dont il partage les fondations. Comme celui-ci, iOS comporte quatre couches d'abstraction : une couche « Core OS » [10], une couche « Core Services » [11], une couche « Media » et

2. NDK ou Native Developpement Kit permet d'utiliser du code dit natif tel que le C ou C++ au sein des application Android

3. JAR (Java ARchive) est un fichier ZIP utilisé pour distribuer un ensemble de classes Java. Ce format est utilisé pour stocker les définitions des classes, ainsi que des métadonnées, constituant l'ensemble d'un programme.

4. L'iPod est un baladeur numérique conçu et commercialisé par Apple

5. L'iPhone est une famille de smartphones conçue et commercialisée par Apple

6. L'iPad est une tablette électronique conçue et commercialisée par Apple.

7. Mac OS X est une série de systèmes d'exploitation propriétaires développés et commercialisés par Apple, dont la version la plus récente est Mac OS X v10.6, dit Snow Léopard.

8. L'iPod touch est un baladeur numérique à écran tactile capacitif multi-touch, conçu et commercialisé par Apple.

9. Apple TV est un appareil conçu et commercialisé par Apple permet la communication sans fil entre un ordinateur et un téléviseur.

10. La couche « Core OS » contient les fonctionalitées bas niveau.

11. La couche « Core Services »contient les services fondamental du système.

pour finir une couche « Cocoa » [12]. Pour dévélopper une application Cocoa, il est imposé d'utiliser l'Objective-C comme langage de programmation.

L'Objective-C est le langage de programmation orientés objet. Il a été inventé au début des années 1980 par Brad Cox, créateur de la société Stepstone. Son objectif était de combiner la richesse du langage Smalltalk [13] et la rapidité du C. C'est une extension du C ANSI [14] qui ne permet pas l'héritage multiple contrairement au C++ [15]. Aujourd'hui, il est principalement utilisé pour développer des applications sur Mac OS X et iPhone. Une nouvelle version Objective-C 2.0 à été indrotuite avec Mac OS X 10.5 (Léopard) en octobre 2007.

Le kit de développement d'iOS fournit les principaux outils nécessaire pour réaliser une application sur iPhone. Il est composé notamment de Xcode, Interface Builder, iPhone Simulator et Instruments. Xcode est l'outil de développement Apple, il permet la création de projets iPhone, l'édition du code source Objective-C, la compilation et le débogage des applications. Interface Builder quant à lui permet de construire des interfaces graphiques. L'iPhone Simulator est un logiciel simulant le comportement d'un iPhone, ce qui permet de pouvoir tester les applications directement sur l'ordinateur. Grâce à Instruments on peut analyser un programme pour surveiller l'état de la mémoire, l'utilisation du réseau, du CPU, etc. Pour utiliser tous ces outils, il est obligatoire de possèder un ordinateur Macintosh sous Mac OS X et de s'enregistrer sur iPhone Dev Center pour pouvoir télécharger et installer le SDK d'iOS.

2.3 Organisation

Afin de garder notre projet coérent et par sécurité nous avons choisi de le stocker sur les serveurs de *Google Code* qui mettent gratuitement à disposition des gestionnaires de versions (Subversion ou SVN en abrégé) distribués sous licence Apache [16] et BSD [17].

Les gestionnaires de versions comme leur nom l'indique, permettent d'avoir à porté de main toutes les versions qu'il y a eu d'un fichier depuis sa création, cela permet donc de pouvoir revenir en arrière si une erreur a été commise. De plus grâce à cela notre projet reste cohérent dans le sens ou pour pouvoir être mit à jour il faut à tout prit avoir modifié un fichier à partir de la dernière version de celui-ci. Dernier avantage d'avoir

12. « Cocoa » est une API native d'Apple pour le développement objet sur son système d'exploitation Mac OS X.

13. Smalltalk est l'un des premiers langages orientés objets créé en 1972.

14. Le C est un langage de programmation impératif conçu pour la programmation système. Inventé au début des années 1970 avec UNIX.

15. Le C++ est un langage de programmation permettant la programmation sous de multiples paradigmes comme la programmation procédurale, la programmation orientée objet et la programmation générique.

16. La licence Apache est une licence de logiciel libre et open source. Elle est écrite par l'Apache Software Foundation, qui l'applique à tous les logiciels qu'elle publie. Il existe plusieurs versions de cette licence (1.0, 1.1, 2.0). Cette licence n'est pas copyleft.

17. La licence BSD (Berkeley software distribution license) est une licence libre utilisée pour la distribution de logiciels. Elle permet de réutiliser tout ou une partie du logiciel sans restriction, qu'il soit intégré dans un logiciel libre ou propriétaire.

utilisé un gestionnaire de version et qu'il est hébérgé sur le net et donc chaque membre de l'équipe peut y acceder où qu'il soit.

- Subversion
- Réunions
- Répartition du travail
- Diagramme de Gant

Chapitre 3

Analyse

– Introduction

3.1 Cahier des charges

3.1.1 Menus

Les menus se doivent d'être clairs et de rendre l'utilisation de l'application aisée. Il s'agit d'un jeu ne demandant aucune compétence particulière. Il va donc toucher un public large et doit pouvoir convenir à tout utilisateur. Cela passe d'abord par une navigation intuitive dans les menus.

Nous avons pour ce faire établi un diagramme d'activité reflettant les différents parcours possibles par un utilisateur lors de sa navigation dans l'application (voir section modélisation p13).

3.1.2 Jeu

Le jeu est la partie la plus importante du projet. Il se decompose en trois parties : Le model, la vue et le controlleur. Le modèle est composé du moteur physique, du moteur de rendu ainsi que de la hierarchie de classe permettant de representer l'ensemble des objets du jeu. La vue quant à elle est composée d'objets graphiques simples (Bouttons, images, ...) et d'une partie reprèsentant le jeu. Elle se doit d'être ergonomique et de permettre à l'utilisateur de pouvoir jouer très simplement. Le controlleur permettra de faire le lien entre les actions de l'utilisateur sur le modèle. Cette décomposition permettra dans le futur de pouvoir modifier facilement le modèle et/ou la vue.

L'application se doit de pouvoir changer de langue, avec comme langues initiales le francais et l'anglais. Elle doit permettre à l'utilisateur de jouer à des parties solitaires ou multijoueurs. Ce dernier possèdera un compte hors ligne et en ligne. Le premier permetta de personnaliser son profil comme par exemple pour modifier la couleur du joueur ou encore changer son pseudo... Il servira aussi à enregistrer les informations et les preferences de connexion sur une base de donnée locale, mais aussi les scores du joueurs (nombre de parties gagnées ou perdus). Le jeu devra permettre à l'utilisateur

de pouvoir créer différents comptes hors lignes en cas de partage de télephone avec un ami ou un membre de famille, pour pouvoir garder en mémoire ses scores et ses préferences. Le compte en ligne quand à lui servira seulement à établir une connexion avec le serveur distant pour pouvoir jouer en multijoueur. Un menu d'aide doit apparaitre pour pouvoir aider le joueur à comprendre le but du jeu et comment jouer. Ce dernier doit être simple et très explicite étant donné la large tranche d'age d'utilisateur que vise cette application. Ensuite un éditeur de carte permettra aux utilisateurs de créer un large choix de cartes, grâce à une multitude de différents objets qui composeront les cartes. Ces dernieres pourront être seulement utilisé en mode solitaire. Pour les parties solitaires une inteligence artificielle avec trois niveaux de difficulté devra permettre à un joueur debutant, intermédiaire ou confirmé de jouer comme bon lui semble pour pouvoir améliorer sa maniere de jouer.

3.1.3 Serveur

Le serveur représente la partie réseau de notre projet. Il doit pouvoir rendre fonctionnel le jeu entre plusieurs téléphones (qu'ils soient de type iOS ou Android). Autrement dit il servira d'hebergeur pour les parties et il se chargera de faire s'interagir les joueurs, via leur mobile, entre eux. Nous parlons donc ici des parties multijoueurs.
Il devra être capable d'enregistrer des inscriptions de nouveaux joueurs, avec vérification qu'il n'y ait pas de doublons. Ces derniers seront inscrits dans la base de données du serveur. Les joueurs devraient ainsi pouvoir se connecter en utilisant le couple username/mot de passe, préalablement choisi. Suite à cela les utilisateurs seront à même de lister les parties en cours, ils pourront choisir de créer des parties ou de les rejoindres.
Voilà concernant les fonctionnalités qui ont été demandé pour la partie serveur.

3.2 Modélisation

3.2.1 Général

Le fait de developper des applications IOS et Android nous a imposé le choix du language. En effet pour developper des applications IPhone, il faut utiliser le language Objective-C, puis pour les applications Android c'est le language JAVA qui est utilisé. C'est deux languages ont une syntaxe complétement différente mais sont quand meme tres proche car ce sont des languages orientés objets. Grâce à cela nous avons pu mettre au point une modélisation général de l'application que nous avons ensuite adapté à chaque language.

Nous avons décidé de développer en anglais car premierement c'est la langue la plus utilisé dans le monde de la programmation et deuxiement car la syntaxe des languages est toujours en anglais. Cela permettra à n'importe quel utilisateur de n'importe quelle nationalités de comprendre le code de l'application.

La documentation des deux applications est également en anglais. Cette documentation permettra à tout utilisateur de comprendre le fonctionnement de l'application. Nous avons créé une AppleDoc pour l'application IPhone et une JavaDoc pour l'application Android. Chacune de ces documentations est au format HTML et sera donc directement visible grâce à un navigateur Web.

Pour ce qui est de la modélisation globale du projet, nous avons choisi de developper les deux applications selon le modèle MVC [1]. Ce dernier est une architecture et une méthode de conception qui organise l'interface homme-machine (IHM) d'une application logicielle.

Le modèle représente le comportement de l'application : traitements des données, interactions avec la base de données, etc. Il décrit et contient les données manipulées par l'application. Il assure la gestion de ces données et garantit leur intégrité.

La vue correspond à l'interface avec laquelle l'utilisateur interagit. Sa première tâche est de présenter les résultats renvoyés par le modèle. Sa seconde tâche est de recevoir toutes les actions de l'utilisateur (clic de souris, sélection d'une entrée, boutons, etc).

Le contrôleur prend en charge la gestion des événements de synchronisation pour mettre à jour la vue ou le modèle et les synchroniser. Il reçoit tous les événements de l'utilisateur et enclenche les actions à effectuer.

Grâce à cette méthode de conception le code est décomposé en trois parties bien distinctes qui permettent la maintenance et l'amélioration du projet. Cela permet aussi de modifier chacune des trois parties sans avoir à modifier les deux autres. Cela permet par exemple de changer de vue et avoir des interfaces graphiques différentes.

3.2.2 Menus

Au démarrage de l'application vous arrivez sur un menu d'accueil. Depuis celui-ci vous pourrez accéder à l'aide, à la liste des comptes locaux ou à la création d'un nouveau. Les menus de l'application ont été réalisé pour que l'utilisateur ait une utilisation intuitive de l'application. Ils se divisent en 4 grandes sections.

Vous avez tout d'abord la section de création de parties locales. Vous aurez accès à une liste de cartes ainsi qu'au réglage de difficulté des bots, leur nombre et le temps de jeu. Le type de partie sera une fonctionnalité à venir. Vous n'aurez plus qu'à créer la partie configurée.

Dans la même catégorie se trouve la section des parties multijoueurs. En accédant à celle-ci vous allez pouvoir vous connecter à votre compte multijoueur, ou le créer si ne déjà fait. Vous accèderez ensuite à la liste des parties multijoueurs, que vous pourrez rejoindre, ou choisir de créer la votre. Dans le menu création le principe est proche des parties locales.

Suite à ces deux sections vient ensuite l'éditeur de cartes. C'est depuis ce menu que vous déciderez la création d'une nouvelle map de jeu local ou à l'édition d'une d'entre elles. Choisisez votre nom de carte et l'éditeur s'ouvrira ensuite à vous. Il vous sera

1. Modèle-Vue-Contrôleur

possible à la fin d'enregistrer votre carte si vous désirez la conserver et l'utiliser comme carte de jeu.

Et enfin vient le menu des options. Depuis ce dernier vous pourrez gérer vos préférences sytèmes telles que le volume ou la langue de l'application(anglais, français). Une sous-section de gestionnaire de profil est aussi présente. Une édition de vos comptes locaux, multijoueurs ou même vos paramètres de jeu comme la position du menu, sont modifiable depuis ce menu à onglets.

Base de données

Une base de données locale a elle aussi été conçue. Cette dernière a pour but de stocker plusieurs types de données.

En effet dès lors qu'un compte local est crée sur le téléphone dans la table PlayerAccount, il est possible de conserver ses préférences de joueur tel que la couleur du joueur, le pseudonyme ou même ses paramètres de connexion multijoueur. Vous pourrez créer autant de comptes locaux que vous le désirez, et il sera possible possible d'éditer ou choisir son compte.

L'application est par ailleur en mesure de conserver les valeurs sonores, la langue et même le dernier utilisateur de l'application, grâce à un son id qui est clé étrangère dans la table System(lastUser).

De plus l'application sera délivrée avec quelques cartes officielles, mais l'utilisateur aura libre droit de créer ses propres cartes de jeu via un éditeur. Elles seront alors stockées dans la table Map avec toujours une clé étrangère vers l'id de son créateur(owner).

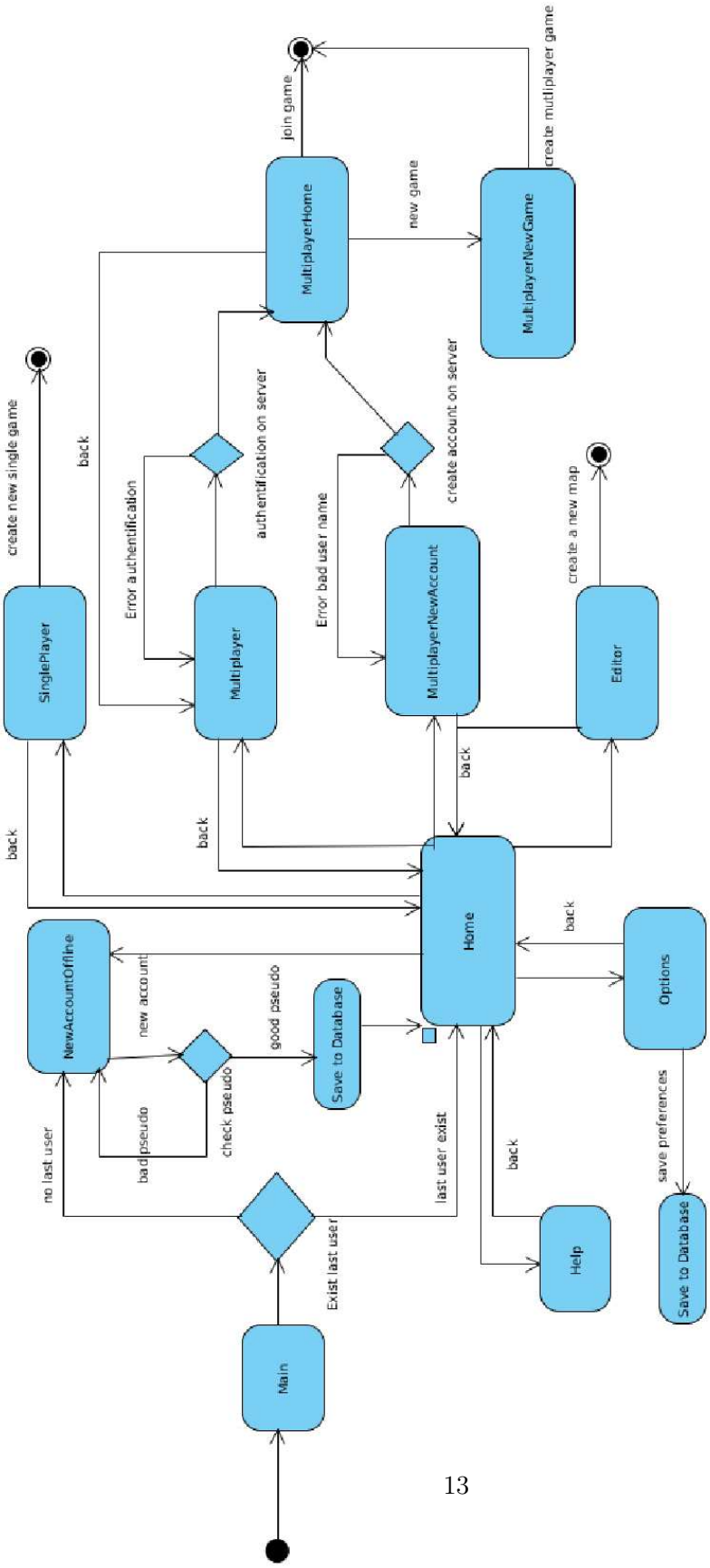


FIGURE 3.1 – Diagramme d'activité

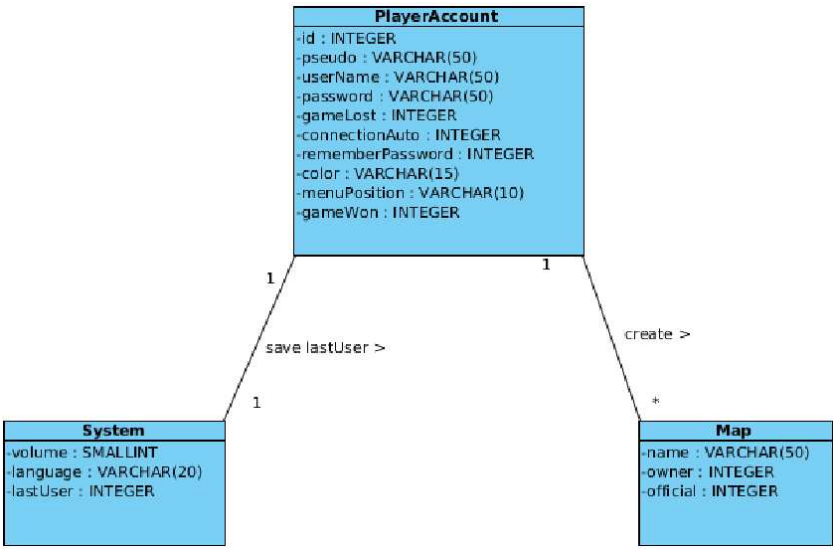


FIGURE 3.2 – Diagramme de classe Base de données

Scénarios

3.2.3 Editeur de carte

L'éditeur de carte est une fonctionnalité qui va permettre à un utilisateur de créer facilement ses propres cartes pour ensuite y jouer dessus contre l'intéligence artificielle. Après avoir réfléchi sur toutes les fonctionnalités que l'éditeur de carte devait remplir, nous avons retenu celles-ci : permettre à l'utilisateur de créer une nouvelle carte, mais aussi de charger une ancienne carte précèdemment créée. Ensuite lui donner la possibilité de modifier le sol de la carte et aussi ajouter ou supprimer des blocs de la carte et enfin la dernière fonctionnalité que l'éditeur de carte implémente c'est de pouvoir placer les différents points de départ des joueurs sur la carte.

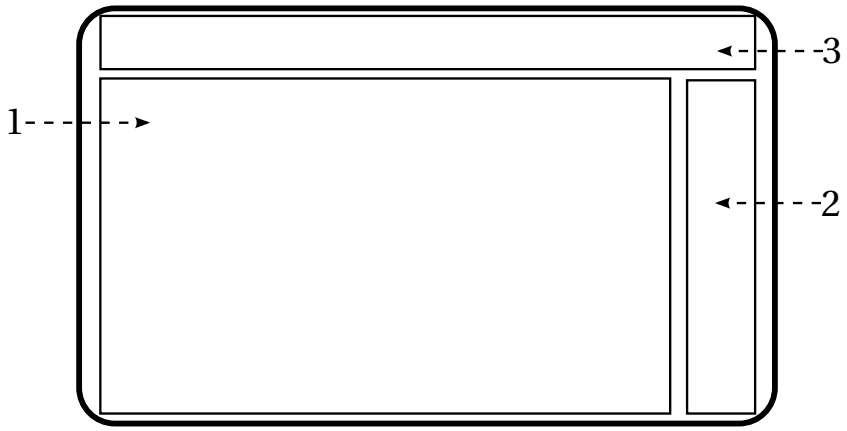
Pour réaliser cette partie de l'application, nous avons utilisé le modèle de conception MVC pour diviser le code de l'éditeur de carte. Grâce a cette décomposition, le code est plus lisible et plus facile à réutiliser.

Modèle

La partie modèle va contenir toute les données de l'éditeur de carte. Les cartes sont les principales données qu'il va devoir manipuler. Pour cela nous avons décidé de la réprésenter sous la forme de deux matrices, la première représentant les objets du premier niveau (le sol) et la deusième la matrice du second niveau (les blocs, les points de départ des joueurs, etc).

Vue

Ensuite, la vue représentera l'interface graphique de notre éditeur de carte. La principale difficulté pour réaliser l'interface graphique était de devoir rentrer toutes les informations nécessaires pour l'éditeur de carte dans un écran de type smartphone [2]. Après plusieurs prototypes d'interface, nous avons décide de séparer l'interface en trois parties. Tout d'abord la plus grande partie, l'affichage de la carte, qui comment étant la principale information à afficher, nous avons essayé de maximiser sa taille. Ensuite un menu à droite permettant au joueur de changer d'outil. Et la dernière partie affiche les différents éléments permettant de controler l'éditeur de carte. L'utilisateur aura juste à choisir l'outil qu'il veut placer sur la carte grâce au menu de droite et ensuite lui suffira d'appuier sur la carte pour placer un bloc dessus.



Controleur

Pour finir le controleur aurra pour but de faire la liaison entre les données du modèle et de la vue. Chaque vue possède son controleur, et il y a un controleur gobal possédant les controleurs de chaque vue.

3.2.4 Jeu

Intelligence artificielle

Comme nous avons vu dans le cahier des charges, nous avons mis en place une integellence artificielle permettant à un joueur de jouer en solotaire.

Tout d'abord nous avons du réfléchir à toutes les actions que les bots pourraient effectuer, lors d'une partie. Premièrement l'action la plus importante que l'intelligence artificielle doit savoir faire c'est de pouvoir se déplacer librement sur la carte en fonction des différents élements de la carte et des actions effectuées par les autres joueurs. Elles

2. Traduit littéralement comme « téléphone intéligent » en français, c'est un terme utilisé pour désigner les téléphones évolués, qui possèdent des fonctions similaires à celles des assistants personnels. Certains peuvent lire des vidéos, des MP3 et se voir ajouter des programmes spécifiques.

sont notamment capable de suivre le joueur le plus proche pour l'attaquer, fuire en cas de danger. Ensuite les bots sont capables de détruire des murs pour pouvoir atteindre un point de la carte et de poser une bombe pour essayer de tuer un ennimi.

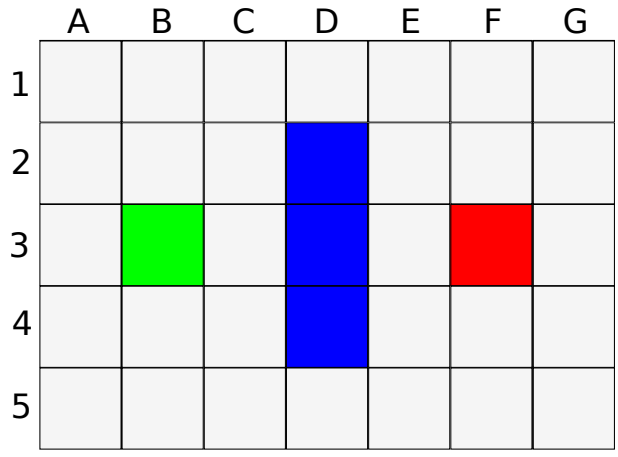
Comme nous veunos de le voir, la principale action que l'intélligence artificielle doit être capable effectuer, c'est de pouvoir se déplacer dans son environement. C'est pour cela que nous avons regardé comment déplacer un object dans un graphe, et nous avons trouvé plusieurs algorithmes étant capable de rechercher un chemin dans un graphe, mais parmi tous ces algorithmes, nous en avons retenu deux en particulé.

Pathfinding

Tout d'abord l'algorithme de recherche A* a pour but de rechercher un chemin dans un graphe entre un nœud initial et un nœud final tous deux préalablement définis. A* permet de trouver l'un des meilleurs (mais pas forcément le meilleur) chemins existant entre un point A et un point B (il retourne le premier chemin trouvé). La force de cette algorithme est le temps de calcul et l'exactitude des résultats, contrairement à Dijkstra qui est le deusième algorithme que nous avons retenu qui lui fournit toujours le meilleur résultat (le plus court chemin entre deux points) mais dans un temps d'exécution beaucoup plus long que l'algorithme de A*. Et comme il peut y avoir jusqu'à trois bots [3] et que l'intéligence artificielle doit réguliairement recalculer son chemin en fonction des actions effectuées par les autres joueurs, nous avons donc choisi l'algorithme de A*.

Maintenant que l'on sait quel algorithme utiliser pour rechercher un chemin dans un graphe, nous allons voir en détail comment marche l'algorithme de A*.

Pour comprendre comment l'algorithme, nous allons nous aider d'un dessin représentent une carte avec un point A (départ) est affiché en vert, le point B (arrivée) est en rouge, et où les cases en bleu représentent le mur.



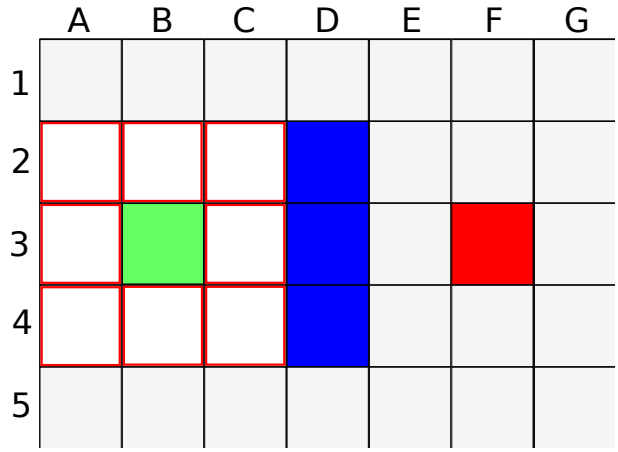
La premier chose que l'on peut observer, c'est que la carte est divisée en cases. Chaque case de la matrice représente un node qui peut être soit traversable, soit non traversable. Dans l'application, il y que les murs ou les bombes que l'on ne peut pas traverser sinon

3. Bot (diminutif de robot) désigne un personnage contrôlé par l'ordinateur.

tous les autres objets les joueurs peuvent les traverser. Donc le but de l'algorithme est de trouver un chemin entre A et B en évitant les murs.

Durant le déroulement de l'algorithme, nous avons utilisé deux listes qui contiennent des cases de la carte. Il y a une liste dite « listeOuverte » et l'autre « listeFermée ». La listeOuvrete contient une liste de cases qui pourraient éventuellement faire partie du chemin, mais pas forcément, pour le moment elle sera vide. Plus précisément c'est une liste de cases que nous devons vérifier. Ensuite, au niveau de la listeFermée, elle contient toutes les cases que nous aurons déjà vérifiées, au début de l'excution, elle contient que le point de départ (B3).

Commencons les explication du déroulement de l'algorithme. Tout d'abord, il faut savoir qu'un joueur peut se déplacer dans toutes les directs, donc nous allons ajouter toutes les cases adjacentes à la listeOuvert qui sont traversable, il y en a huite (A2, B2, C2, A3, C3, A4, B4, C4). Donc pour le moment nous devons avoir ça :



Les carrés avec un contour rouge sont les carrés qui sont dans la listeOuverte et les carrés qui on une couleur un peut plus foncé que les autre sont les carrés qui se trouvent dans la listeFermée.

Maintenant pour choisir la case par laquelle on doit passer, nous devons rajouter trois données « F », « G » et « H » :

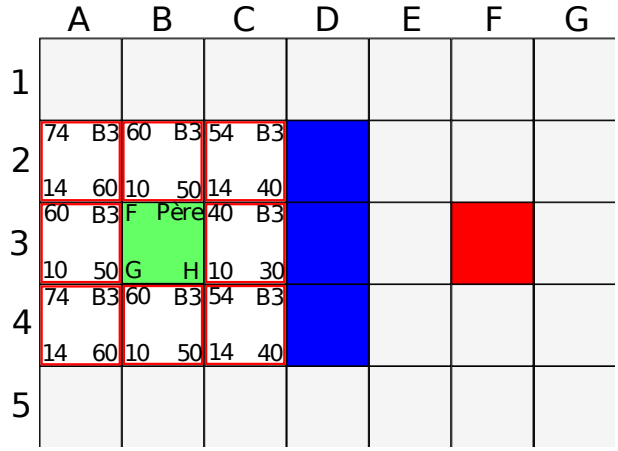
G : c'est le coût de mouvement pour aller de la case A à une case donnée sur la grille, en suivant le chemin généré jusqu'à cette dernière.

H : c'est l'heuristique, c'est à dire le coût estimé pour allé du point courant à l'arrivé. Comme nous ne connaissons pas vraiment la distance qu'il nous reste à parcourir, car toutes sortes d'obstacle peuvent se trouver sur notre chemin (objet non traversable). Donc nous allons devoir l'approximer grâce à une fonction, pour la calculer nous avons choisi d'utiliser l'heuristique de Manhattan, qui consiste à compter le nombre de bloc (à vol d'oiseau et sans prendre les diagonales) qui lui reste à parcourir.

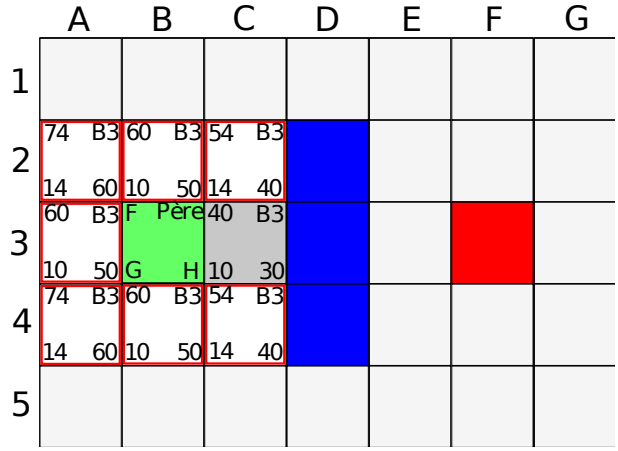
F : c'est G + H

Chaque case de la listeOuverte ou de la listeFermée vont devoir possèder toutes ces données, plus les coordonnées de leur père, c'est à dire les coordonnées de la case qui vient de les ajouter dans la la listeOuverte. Pour calculer G, nous allons assigner un coût de 10 pour chaque déplacement horizontal ou vertical, et un coût de 14 pour un mouvement en diagonale. Nous utilisons ces données car la distance nécessaire pour se déplacer est la racine carrée de 2, ou approximativement 1.41 fois le coût d'un déplacement vertical ou horizontal. Nous utiliserons donc 10 et 14 pour des raisons de simplification. Par conséquent, nous allons multiplier par 10 le coût H pour qu'il soit cohérent par rapport à G.

Donc maintenant, nous devons avoir cette matrice :

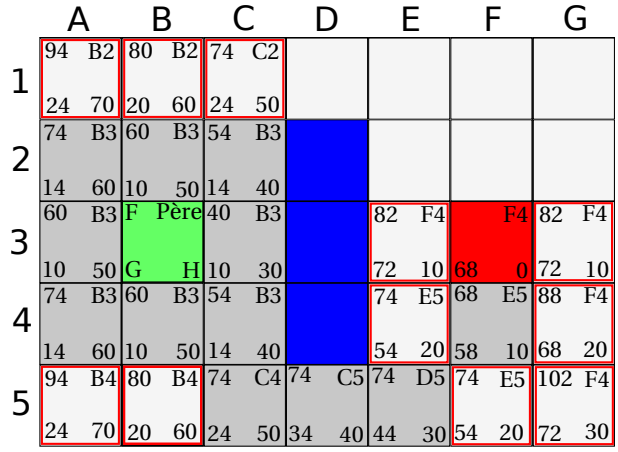


Après avoir ajouté toutes les cases adjacentes à la case courant, il suffit de prendre la case qui a le plus petit coût F et ensuite de la rajouter dans la listeFermée et de la supprimer de la listeOuverte. Nous obtenons donc :

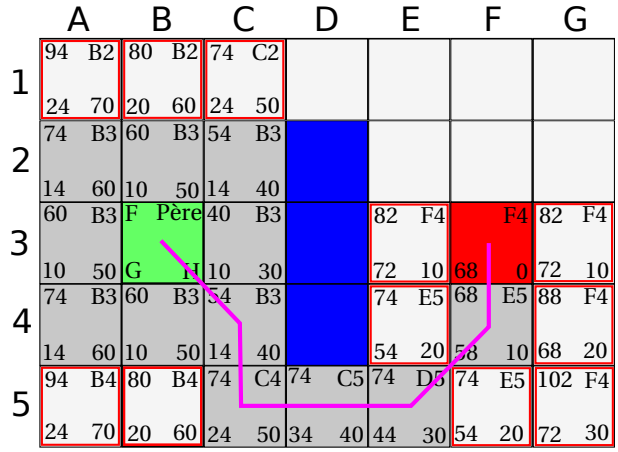


Ensuite on regarde toutes les cases adjacentes à la dernière case ajoutée dans la listeFermée. Si elles se trouvent déjà dans la listeOuverte, on vérifit que leurs coût soient inférieur au coût de la case correspondante déjà dans la listeOuverte, si oui alors on la remplace sinon on ne fait rien.

Pour finir on répète cette opération jusqu'on arrive à la case d'arrivée, nous obtenons ça :



Pour finir, il nous suffit juste de récupérer la case d'arrivée et de regarder son père, puis de repéter cette opération avec la case obtenu jusqu'à arriver à la case de dépard. Grâce à ça nous obtenons cette dernière étape de l'algorithme :



Comme on peut le voir sur le schéma prèdédent, le chemin qu'à trouvé l'algorithme de A* est le suivant : $B3 \rightarrow C4 \rightarrow C5 \rightarrow D5 \rightarrow E5 \rightarrow F4 \rightarrow F3$. L'algorithme aurait pu d'autre chemin équivalent à celui-ci mais le principe de cette algorithme c'est de renvoyer le permier chemin qu'il trouve.

– Diagramme classe

Comme vue dans le cahier des charges l'application est divisé en trois grandes parties : Le modèle, la vue et le controlleur.

Modèle

Le modèle constitue la *base* de notre projet. C'est sur celui-ci que nous avons construit le reste de l'application. Nous avons modélisé celui-ci de manière à ce qu'il soit clair et

simple. Ce dernier ce décompose en trois sous-partie : la partie hierarchie des objets, la partie moteur puis la partie éditeur de carte.

Hierarchie des objets

Pour concevoir la hierarchie des objets, il a fallu tout d'abord dinstinguer la totalité des objets qui seraient disponible dans le jeu ainsi que leurs différences et leur point communs. Nous avons donc distingué quatre grands types d'objets : les objets destructibles, les objets indestructibles, les objet animés ainsi que les objets inanimés. En sachant que les objets pouvaient être destructibles-animés, destructible-inanimés, indestructible-animés ou indestructible-inanimés. Nous avons choisi que tout objet serait considéré comme un objet animé pour éviter des soucis de modélisation. Ainsi les objets possèderont tous une sequence d'animation d'images qui sera dessiné a l'écran. Cette derniere sera répété dans le cas d'une animation qui se répété et qui comporte plusieurs images, sinon si elle possède simplement qu'une seule image, seule cette image sera dessiné a l'écran. Nous avons ensuite dinstingué une multitude de point communs entre les différents objets dont voici le nom de l'attribut et leur utilité :

position indique la position en pixel de l'objet sur l'écran

nom indique le nom de l'objet

hit est à 1 si l'objet peut être touché par une explosion (0, sinon)

level est à 1 si l'objet est animés (0,sinon)

fireWall est à 1 si l'objet ne laisse pas passer les flammes des explosions (0,sinon)

damages indique si l'objet peut infliger des domages

idle est une image qui représente l'objet en général

animates est une table de hachage contenant l'ensemble des images composant la sequence d'animation de l'objet (vide si inanimés)

destroy est une table de hachage contenant l'ensemble des images composant la sequence d'animation de destruction de l'objet (vide si indestructible)

currentFrame permet de connaitre le numéro de l'image courante de la sequence d'animation en cours d'affichage

Nous avons donc décider de concevoir une classe "Object" pour modéliser toutes ces propriétés que les objets ont en commun. Cette classe sera abstraite car tout objet est destructible ou indestructible et cela sera décris dans des classes plus spécialisés.

Ensuite nous avons pensé a créer deux autres classes : "Destructible" et "Undestructible" pour les objets destructible et indestructible. Ces deux classes héritent de "Object" car elles sont des objets. Ce qui différencie ces deux classes est le champ *life* qui permet de savoir combien de fois l'objet doit etre touché par une bombe avant d'être detruit.

Nous avons ensuite dinstingué deux autres types d"objets encore plus spécifique : Les joueurs et les bombes. Ces derniers sont des objets destructibles puisqu'il ne dure pas toute la partie selon le mode de jeu.

Les bombes possèdent deuxnouveaux attributs qui les différentie des autres objets :

type permet de connaitre le type de bombe

owner permet de connaitre le joueur qui a posé la bombe

Quand à la classe joueur celle-ci possède encore d'autres attributs :

color permet d'afficher à l'écran le joueur avec sa bonne couleur

bombsTypes permet de connaitre le type de bombe qu'il peut poser

powerExplosion permet de connaitre la porté d'explosion de ses bombes

timeExplosion permet de connaitre le temps d'explosion des bombes

speed permet de connaitre la vitesse du joueur

shield permet de connaitre la valeur du bouclier

bombNumbers permet de connaitre le nombre maximum de bombes que le joueur peut poser

isTouched permet de savoir si le joueur viens d'être touché

isKilled permet de savoir si le joueur est mort

isInvincible permet de savoir si le joueur est invincible

Cette hierarchie de classe nous permettra donc de modéliser l'ensemble des objets que le jeu pourra afficher.

Moteur

Pour ce qui est du moteur. Celui-ci est représenté par une classe "Engine". Cette classe va contenir l'ensemble des methodes qui vont permettre de d'établir les collisions. C'est aussi cette classe qui s'occuppe de mettre à jour les bombes ainsi que l'IA du jeu. Une instance d'un moteur est associé à une partie dont le nom de classe est "Game". Cette classe "Game" est abstraite et représente une partie avec toutes les options qu'elle contient. C'est à dire qu'elle possède des attributs permettant de décrire le type de partie, un tableau avec chaque joueur de la partie, ainsi que le carte du jeu. La classe possède toutes les méthodes d'initialisation, de mise à jour, de dessins et de fin de la partie. Enfin pour différencier les différents types de parties nous avons utiliser le pattern [4] décorateur. Ainsi il y aura deux grands types de parties : les parties solitaires et les parties multijoueurs. Puis ces parties sont décorés par le type de partie : survivor [5] ou death match [6].

Ensuite pour ce qui est des cartes, une classe abstraite nommé "Map" contient le nom et la taille de la carte ainsi qu'un tableau contenant la position initial de chaque joueur sur la carte. Puis une classe "GameMap" qui hérite de map permet de dessiner la carte a l'écran, elle se compose d'une image représentant le sol puis d'un tableau d'objets animés qui permet de représenter l'ensemble du reste des objets. Ensuite nous utilisons un autre

4. Un pattern est un modèle de conception en informatique

5. Le mode survivor est un mode où chaque joueur à un nombre de vie limité et dont le gagnant est celui qui sera le dernier a être en vie

6. Le mode death match est un mode où il y a un temps limité et dont le joueur gagnant est celui qui a tué le plus d'ennemis

type de carte qui va permettre de faciliter les collisions, que ce soit pour un joueur humain ou pour une intelligence artificielle. Cette classe est appelé "CollisionMap" et contient une matrice d'objet "CollisionCase" ainsi qu'un tableau qui contient l'ensemble des bombes posés sur la carte. Les "CollisionCase" sont composé de deux tableaux. Chaque valeur d'un des tableau est associé à une valeur de l'autre talbeau. Un tableau *types* contient les différents types de danger ou d'objet qu'il existe sur cette case et un tableau *counters* permettra en fonction de chaque type de compter le nombre de fois que ce danger ou objet est présent sur cette case. Par exemple on peut très bien avoir une case qui est dans le champ d'explosion de quatres bombes et qui contient un bloc de type destructible. Ainsi le tableau *type* contiendra deux champs , un pour le type zone dangereuse et un autre pour le type bloc destructible puis le tableau *counters* contiendra donc dans sa case associé à la case zone dangereuse une valeur égale à quatre et et une valeur de un pour l'autre.

Vue

L'interface graphique du jeu est décomposé en trois partie. Une partie qui représente le menu d'information de la partie, une autre qui réprésente le menu des actions possibles du joueurs et une dernière qui représente la partie en cours. Le menu d'information se situe en haut de l'écran. Il permet d'afficher le score des joueurs, le temps lors d'une partie death match ainsi que les bonus du joueurs (le nombre de bombes qu'il peut poser, la portée de l'explosion des bombes, la vitesse du joueur et les bonus de vie) et de mettre le jeu en pause. Le menu d'action se situe à droite de l'écran. Il permet à l'utilisateur de poser les bombes grâce à un bouton et aussi de changer de type de bombes grâce à une liste déroulante. Quand à la partie, elle est afficher au centre de l'écran et prend le maximum de place possible pour que le jeu soit le plus visible possible. Elle permet d'afficher la carte ainsi que les joueurs et les bombes. Mais elle sert aussi à écouter les mouvements du doigt de l'utilisateur pour modifier les coordonnées du joueur dans le modèle pour pouvoir ensuite rafraichir l'écran et voir le joueur se deplacer.

Controlleur

Le controlleur va permettre de faire la liaison entre la vue et le modèle. La majorité de ces méthodes sont appelé lorsque l'utilisateur interagie avec la vue. Ces méthodes vont par la suite modifier le modèle qui va permettre de mettre à jour la vue. Le controlleur est divisé en quatre sous-controlleur. Chacun des trois premiers est respectivement associé à l'une des trois vues cité ci-dessus. Puis le quatrième est plus général, il permet de faire la liaison entre les trois autres controlleurs. Car en effet chacune des actions effectués sur l'une des vue peut modifier l'une des deux autres.

GamePlay

Nous avons choisi d'établir un gameplay immersif. C'est à dire que notre interface graphique sera utilisée de manière souple et intuitive. Notre gameplay est divisé en trois

parties : les déplacements, la pose des bombes et la gestion des différents types de bombes.

Pour ce qui est des déplacements du joueurs. Nous avons décidé que l'utilisateur utiliserais toute la surface de l'écran pour se déplacer. Ainsi si il veut se déplacer vers la droite, il fait glisser sont doigt de la gauche vers la droite, puis tant qu'il restera appuyer sur l'écran, le joueur continura de se déplacer. Cette manière de se déplacer est précise et très intuitive contrairement à l'utilisation d'un joystick virtuel ou de l'accéléromètre.

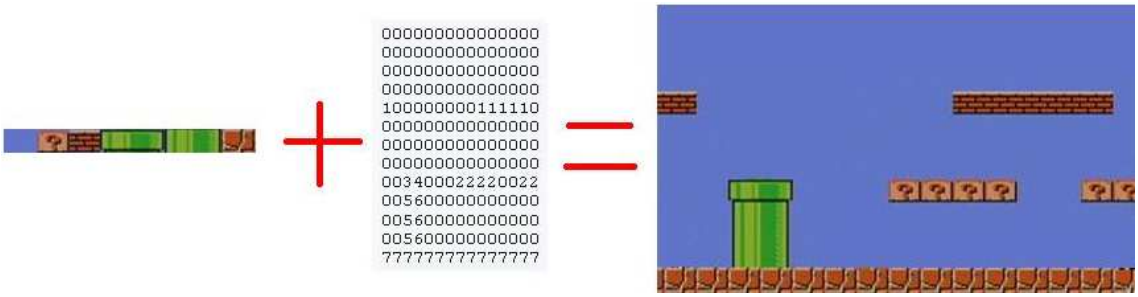
Pour la pause des bombes un simple boutton est mis en évidence en bas à droite de l'écran. Celui-ci est assez gros pour que l'utilisateur n'est pas à appuyer sur une zone trop précise en cas de manipulation rapide.

Puis pour le choix des différentes bombes, une simple liste déroulante est mise en évidence à droite du jeu, pour pouvoir changer de bombe rapidement.

Tile Mapping

La gestion des images a été une partie très importante de l'analyse. Car le développement mobile impose plusieurs contraintes, notament la gestion de la mémoire et la vitesse de calcul. Nous nous sommes donc inspiré des premiers jeux consoles comme Super Mario Bros ou Zelda qui ont été développé sur les premieres consoles comme la NES [7] ou la GameBoy [8]. Nous avons donc conçu le moteur du jeu selon le principe du Tile Mapping [9] pour minimiser l'utilisation des ressources des téléphones.

Le principe du Tile Mapping est d'utiliser des petites images que nous appelerons *tiles*. Ces tiles sont contenu dans une image appelé sprite (cf ci-dessous). Ensuite une matrice de nombre entier est utilisé pour représenter l'environnement du jeu. Chaque nombre entier représente un tile. Puis grâce à cette association, la carte sera dessiné à l'écran selon la matrice. Voici un exemple :



Nous avons donc repris ce principe et nous l'avons amélioré. En effet grâce à la programmation objet, les entiers sont réprésenté directement par des objets contenant le tile qui le représente. Ainsi, nous stockons dans une matrice tous les objets inanimés (les objets dont le tile restera le meme tout le long de la partie) et nous parcourons cette matrice pour dessiner chaque objet à sa position pour obtenir une nouvelle image au format png. Ainsi à chaque rafraichissement de l'écran, seule la nouvelle image est

7. La Nintendo Entertainment System, ou NES, est une console de jeux vidéo 8 bits
8. La Game Boy est une console de jeux vidéo portable à la puissance comparable à celle de la NES
9. Le Tile Mapping est une technique de modélisation graphique

dessiné et non pas chaque tile. Cette méthode permet d'éviter un parcours intempestif de la matrice. Ensuite pour le reste des objets (les objets animés composés d'une sequence de tiles) nous avons décidé de les stocker dans une table de hachage dont la clé est la position de l'objet (pour y accéder plus rapidement). Ainsi à chaque rafraichissement de l'écran on va dessiner l'image des objets inanimés puis parcourir entierement la table de hachage et dessiner le tile courant de la sequence d'animation de chaque objet.

La gestion des images et du son

Chaque image, tile et chaque son dont est composé le jeu sont chargé au lancement de l'application. Tous les tiles sont stocké dans des images appelés sprites. Un sprite est donc un ensemble d'image. L'ensemble des tiles sont stockés dans différents sprites. Il y a un sprite pour les bombes, un pour les joueurs et un autre pour le reste des objets. Chaque sprite contient l'ensemble des images des objets qu'il représente. Chaque sprite est associé à un fichier XML [10]. Le fichier XML peut être réprésenté sous forme d'un arbre. Les fichiers XML ressemble tous plus au moins a ce genre d'arborescence :

10. Extensible Markup Language, (langage de balisage extensible) est un langage informatique de balisage générique

```xml
<objects>

    <object_type name="object_name1" hit="0" level="0" fireWall="0" life="0" damages="0">

        <animation name="idle" canLoop="false" sound="sound_path">
            <framerect top="0" left="30" bottom="30" right="60" delayNextFrame="0" />
        </animation>

        <animation name="animate" canLoop="true" sound="sound_path">
            <framerect top="0" left="30" bottom="30" right="60" delayNextFrame="1" />
            <framerect top="0" left="60" bottom="30" right="90" delayNextFrame="1" />
            <framerect top="0" left="30" bottom="30" right="60" delayNextFrame="1" />
            <framerect top="0" left="0"  bottom="30" right="30" delayNextFrame="1" />
        </animation>

        <animation name="destroy" canLoop="false" sound="sound_path">
            <framerect top="0" left="90"  bottom="30" right="120" delayNextFrame="1" />
        </animation>

    </object_type>



    <object_type name="object_name2" hit="1" level="1" fireWall="1" life="1" damages="0">

        <animation name="idle" canLoop="false" sound="sound_path">
            <framerect top="30" left="30" bottom="60" right="60" delayNextFrame="0" />
        </animation>

        <animation name="animate" canLoop="true" sound="sound_path">
            <framerect top="30" left="30" bottom="60" right="60" delayNextFrame="1" />
            <framerect top="30" left="60" bottom="60" right="90" delayNextFrame="1" />
            <framerect top="30" left="30" bottom="60" right="60" delayNextFrame="1" />
            <framerect top="30" left="0"  bottom="60" right="30" delayNextFrame="1" />
        </animation>

        <animation name="destroy" canLoop="false" sound="sound_path">
            <framerect top="30" left="90"  bottom="60" right="120" delayNextFrame="1" />
        </animation>

    </object_type>


</objects>
```

Chaque fichier XML commence par une balise racine permettant de lister les objets qu'elle contient (*<objects>*). Ensuite chaque objet est décris par une balise qui contient le type, le nom de l'objet ainsi que l'ensemble de ses propriétés. La propriété *hit* est à 1 si l'objet peut être touché par une explosion (0, sinon), le champ *level* est à 1 si l'objet est animés (0,sinon), le champ *fireWall* est à 1 si l'objet ne laisse pas passer les flammes des explosions, le champ *life* indique le nombre de fois que l'objet doit etre touché par des flammes pour qu'il soit entierement détruit, et enfin le champ *damages* indique si l'objet peut infliger des domages. Par exemple *<destructible name="herb" hit="1" level="1" fireWall="1" life="1" damages="0">* décris un objet de type *destructible*, qui peut-etre touché par des flammes, qui est animé, ne laisse pas passer les flammes des bombes, possede une vie, n'inflige pas de domage et dont le nom est *herb* . Ensuite chaque objet possède au plus trois balises animations (sauf pour les joueurs). Tout objet possède au moins la premiere balise : <animation name="idle" canLoop="false" sound="sound_path">

qui est celle dont le nom est *idle*. Elle représente l'image standard de l'objet. Par exemple pour un objet bombe, ce sera l'image qui sera affiché dans la liste des bombes pour pouvoir selectionner ses bombes. Le champs *canLoop* permet de savoir si l'image doit être répété en boucle et le champ *sound* contient le chemin d'acces au fichier de son de l'animation si elle en possede un. Il y a ensuite la même balise mais dont le nom est *animate*, celle-ci contiendra toutes les sequences d'images d'un objet qui est animés. Puis la dernière est celle dont le nom est *destroy*. Cette derniere contiendra l'ensemble des images composant la sequences d'animation de destruction de l'objet. Enfin chaque balise de type animation contient des balises de type *framerect*. Ces balises permettent de donner la position de chaque image de la séquence d'animation dans le sprite ainsi que le delay de rafraichissement entre chaque image. Par exemple *<framerect top="60" left="30" bottom="90" right="60" delayNextFrame="1" />* représente une image dont le bord du haut est situé à 30pixels, le bord du bas à 90pixels, le bord de gauche à 30pixels et le bord de droite à 60pixels en partant du coin en haut à gauche du sprite. Puis pour le champ *delayNextFrame* celui-ci informe que l'image suivante sera déclenché apres un delay de 1s.

Grâce à cette modélisation, l'application va parcourir au démarrage l'ensemble des fichiers XML et va créer une table de hachage pour chaque ensemble d'objet du fichier. Lors de ce parcours, l'application va instancier chaque objet avec toutes propriétés que lui indique le document XML, ainsi que ses sequences d'animation. Ensuite lorsqu'un objet devra être utilisé dans le jeu, il suffira d'utiliser une copie de l'objet déja chargé en mémoire pour éviter de devoir reparcourir le fichier.

3.2.5 Réseau

Serveur

Il a été fixé dans le cahier des charges que notre serveur devrait pouvoir effectuer plusieurs tâches particulières séparées. Nous avons donc décidé de les compartimenter en classes.

Notre serveur est crée sur une base de servlet. Ce fût ici aussi un point nouveau pour nous, réiterant les phases d'analyse, de découverte, de test et de mise en place. Le fonctionnement est basé sur les échanges de requêtes type HTTP, où à chaque demande correspond une réponse.

Une servlet est une classe Java qui permet de créer dynamiquement des données au sein d'un serveur HTTP. Une servlet s'exécute dynamiquement sur le serveur web et permet l'extension des fonctions de ce dernier, typiquement : accès à des bases de données.

Les six éléments situés sur la partie haute du schéma ci-dessous(respectivement ServletInscription, ServletConnection, ServletGamesList, ServletCreateGame, ServletConnectionGame et ServletManageGame), représente les différentes tâches qu'un utilisateur puisse demander au serveur. Elles sont reliées à une classe nommée ContextListener, qui leur permettra d'accéder aux mêmes données sans qu'il y ait de conflits. La partie basse représente les objets qui seront utilisés pour les parties en multijoueurs. Bien évidemment ces objets sont très proches de ceux utilisés dans les parties locales(Schéma

3.3).

Comme il a été dit précédement, notre serveur est accessible via des requêtes HTTP contactant des servlets. Ces servlets sont stockées dans un serveur d'application nommé Apache Tomcat. Il s'agit d'un conteneur libre de servlets Java 2 Enterprise Edition, mais il fait aussi office de serveur Web.

Le scénarios le plus probable serait le suivant. Un utilisateur désire jouer en ligne contre de vrais joueurs. Il va alors passer par l'inscription et créer son compte sur le serveur(Inscription). Une fois cette étape obligatoire faite, il pourra choisir entre rejoindre une partie en ligne en cour(ConnectionGame), ou en créer un nouvelle(CreateGame). Dès lors qu'il aura accès à une partie en ligne, un contact régulier avec le serveur sera obligatoire afin de réaliser les interactions entre les différents joueurs(ManageGame). Tout ceci devra se réaliser bien sûr dans une durée infime afin de ne pas pénaliser les joueurs.

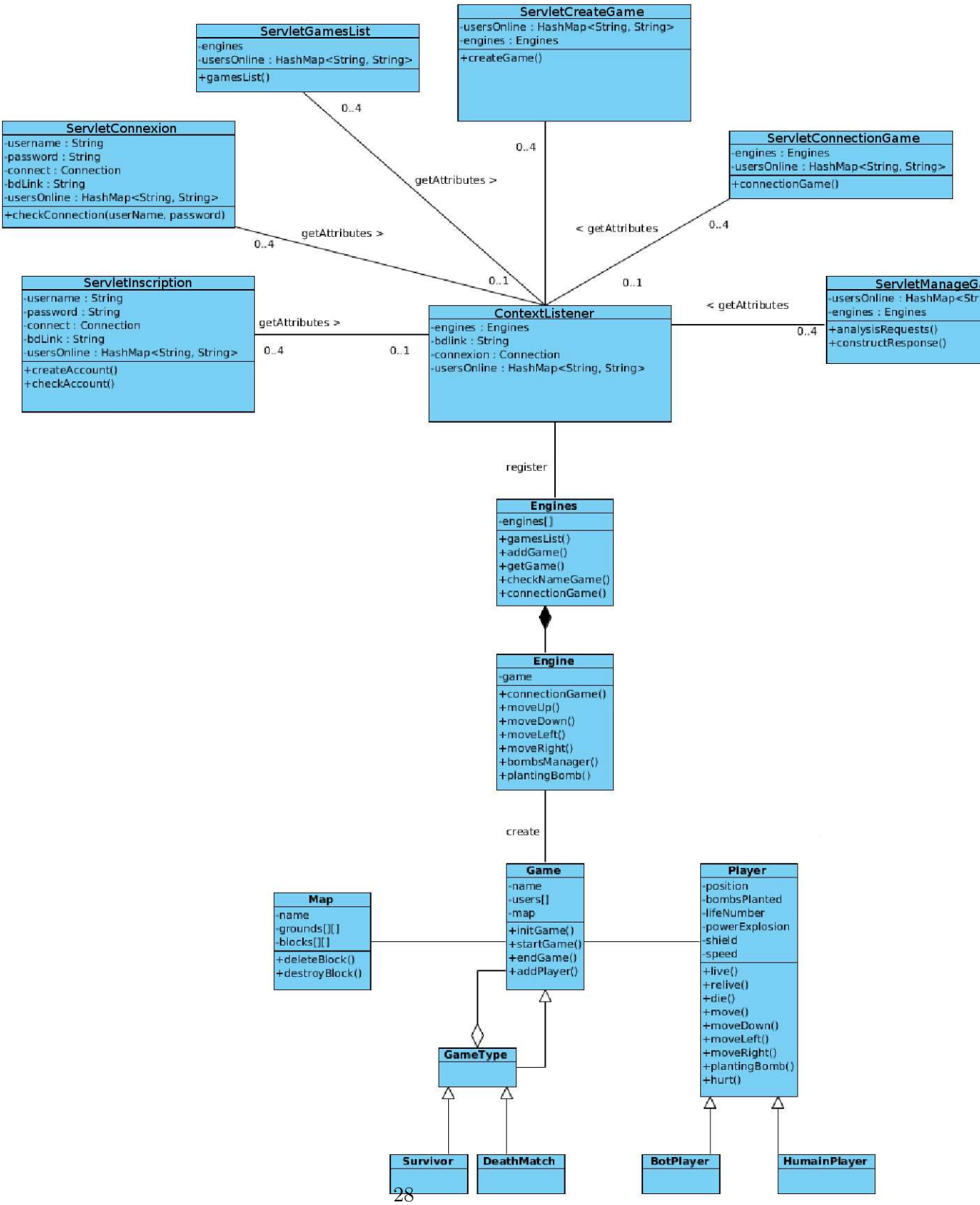


FIGURE 3.3 – Serveur

JSON

Soucieux des performances et de la rapidité des échanges entre applications et serveur, nous avons mis en place un protocole de communication client/serveur où les messages transitant sont des flux JSON [11]. Contrairement au XML qui peut représenter des données orientées document, JSON se focalise sur la description d'objets. Un autre avantage reconnu de JSON par rapport à XML est qu'il est nettement moins verbeux que ce dernier. Quoi qu'il en soit JSON reconnait la philosophie des services web exposant une interface d'échange : il s'agit d'envoyer et de recevoir des informations dans un format facilement manipulable par le protocole de transport HTTP.

Voilà pourquoi le JSON semblait être un format de données d'échanges optimal pour véhiculer le plus d'informations avec une taille moindre. Il est aussi en adéquation avec notre politique d'utilisation web pour un serveur. De plus étant beaucoup utilisé, nos deux langages mettent à disposition des outils de sérialisation de leurs objets en JSON.

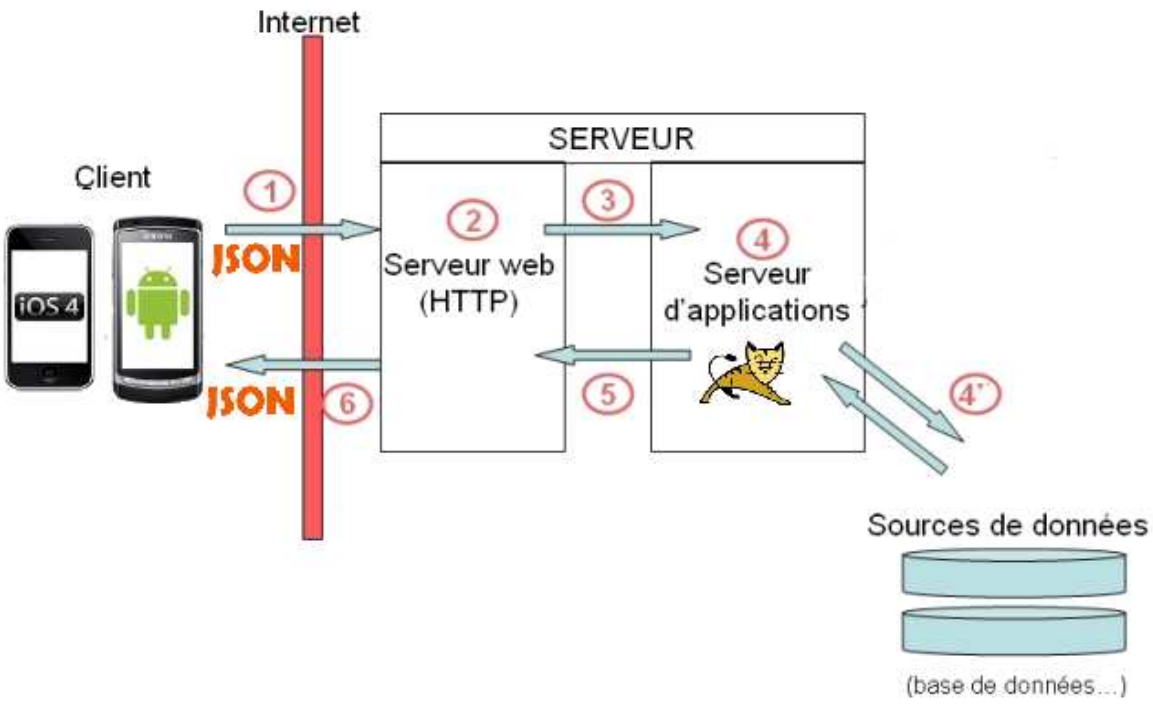
Ci-dessous un exemple concret de notre protocole de communication JSON entre serveur et application cliente.

```
ServletInscription
Player => Serveur
{["username","password"]}

Serveur => Player
{"OK"} ou {"BU"}

ServletConnexion
Player => Serveur
{["username","password"]}

Serveur => Player
{"OK"} ou {"BU"}

ServletGameList
Player => Serveur
{"userKey"}

Serveur => Player
{[{"class":"Game","map":"mapName","name":"gameName",
 "playerNumberConnected":nbConnected,"type":"gameType"},{..},{..}]}

ServletCreateGame:
Player => Server:
{"userKey": <userKey>,
```

11. JavaScript Object Notation : format de données textuel issu du JavaScript (ECMAScript pour plus exact) où il était employé comme une syntaxe pour décrire les valeurs des instances d'objets

```
"game": {"name":<name>, "type":<type>, "map":<map>, "ennemiesNumber": <ennemiesNumber>}}

Server => Player:
{"OK"} ou {"errorType"}

ServletConnectionGame:
Player => Server:
{["userKey", "gameName"]}

Server => Player:
{[<1/2/3/4>, "play<true/false>", "map", "time<mm:ss>"]}
ou
{"errorType"}

ServletManageGame:
Player => Server:
{"userKey", "gameName", "action"}

Server => Players: (Player, bombs, blocs, score, time)
{[
 [ ["x", "y", "direction", "dead <true/false>"],[...] ],
 [ ["x", "y", "type", "explode <true/false>" ], [...] ],
 [ ["position": {"x", "y"}, "bonus": <bonus>], [..] ],
 [1,2,3,4],
 "time <mm:ss>"]}
 ou
{"errorType"}
```

Schéma de fonctionnement



1. Le client émet une requête pour demander une ressource au serveur. Par exemple la création de son compte multijoueur, qui pourrait se situer `http://Bomberklob.com/inscription`
2. Côté serveur, c'est le serveur web qui traite les requêtes HTTP entrantes. Il traite donc toutes les requêtes, qu'elles demandent une ressource statique ou dynamique. Seulement, un serveur HTTP ne sait répondre qu'aux requêtes visant des ressources statiques.
3. Ainsi, si le serveur HTTP s'aperçoit que la requête reçue est destinée au serveur d'applications, il la lui transmet. Les deux serveurs sont reliés par un canal, nommé connecteur.
4. Le serveur d'applications (dans notre cas Tomcat) reçoit la requête à son tour. Lui est en mesure de la traiter. Il exécute donc la servlet correspondante à la requête, en fonction de l'URL, en récupérant les valeurs dans le flux JSON entrant. Cette opération est effectuée à partir de la configuration du serveur, grâce un fichier web.xml faisant le mapping entre URL et servlet associée.

   La servlet est donc invoquée, et le serveur lui fournit notamment deux objets Java exploitables : un représentant la requête, l'autre représentant la réponse. La servlet execute sa fonction et génère la réponse à la demande, sous forme de flux JSON. Cela peut passer par la consultation de sources de données, comme des bases de données (4' sur le schéma).

Base de données

Afin de pouvoir conserver les utilisateurs en ligne ainsi que leurs infos personnels et permettre une authentification, nous avons dû établir une base de données sur le serveur. Cette dernière à été pensé comme demandé pour l'enregistrement de comptes. Une unique table nommée Users remplie donc cette fonction. Le serveur devra pouvoir y accéder en écriture(inscription) comme en lecture(connexion).

Elle ne comportera que deux champs, userName et password. Dès lors que l'utilisateur désirera créer un compte multijoueur, il renseignera dans l'application son userName souhaité ainsi que son mot de passe. Ce couple sera alors envoyé au serveur qui vérifiera dans cette base de données, que le userName(unique) n'est pas déjà utilisé. Auquel cas un nouveau n-uplet sera inséré et permettra l'authentification de l'utilisateur par la suite. Les mots de passe seront bien évidement crypté pour des raisons de sécurité.

3.3 Différences entre Android et iOS

Chapitre 4

Développement

4.1 Mobile

4.1.1 Menus

API et widget

Android

Afin de créer les différents menus de notre application, Android met à la disposition des developpeurs une API [1] très bien fournie. Parmis celle-ci le package Widget nous a été très utile.

Grâce à ce dernier de nombreux objets ont été utilisé afin de mettre en oeuvre et rendre pleinement fonctionnel nos menus. Parmi les plus utilisés il y a eu bien sûr Button, TextView, CheckBox et EditText pour les plus explicites. Les objets comme Spinner, SeekBar et Gallery étant respectivement utilisés pour les menus déroulants, les barres de progression et les galeries d'images pour la sélection de cartes de jeu.

Les composants graphiques sont créés ici au travers du fichier déclaratif XML via une synthaxe bien particulière. Cette méthode est vraisemblablement préférable, du moins lorsque l'interface graphique est figée, connue à l'avance. Exemple :

```
<Spinner android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:id="@+id/accounts"
 android:layout_gravity="center_horizontal"
 android:prompt="@string/ChooseUserAccount"></Spinner>
```

Pour récupérer la référence d'un widget créé depuis le fichier xml de layout, il convient d'utiliser la méthode findViewById de la classe Activity dans nos classes Java.

Exemple :

1. API : Application Programming Interface, c'est un ensemble de classes mis à disposition par une bibliothèque logicielle.

```
Spinner sp = (Spinner)findViewById(R.id.accounts);
```

On peut remarquer que cette méthode accepte en paramètre un int et non un String comme on pourrait s'y attendre. En effet, l'id est exprimé sous forme de constante int, on ne passe pas à la méthode la chaîne de caractères proprement dite. Grâce à cela, la méthode est sûre et on évite ainsi les erreurs à l'exécution qu'on pourrait avoir si on spécifiait un id ne correspondant à aucun widget.

Concernant leur positionnement, un système de layout est utilisé. Les layouts sont des ViewGroup responsables du dimensionnement et du positionnement des widgets à l'écran. Il en existe plusieurs, chacun adoptant une stratégie bien spécifique.

En ce qui nous concerne nous avons principalement utilisé les *ListView, LinearLayout, TableLayout* et enfin les *RelativeLayout*. Ce dernier nous a été très utile. En effet les widgets contenus dans un RelativeLayout peuvent déclarer leur position relativement par rapport à leur parent ou par rapport aux autres widgets. De ce fait nos menus et autres interfaces graphiques conservent leur proportion et leur agencement originel.

Les listeners présents dans les classes java permettent à leur tour d'écouter les évènements utilisateurs ou system et réagir en fonction comme l'accès au menu suivant ou le lancement d'un partie.

Les activités sont des composants centraux des applications. Ce sont également les composants qui portent les éléments visuels de l'interface utilisateur agencés sur l'écran. La navigation entre les écrans se fait dans notre cas de façon explicite. L'ordre de changement d'activité est véhiculé par un objet de type Intent (intention en anglais). Les activités s'enchaînent les unes aux autres par invocation directe. C'est-à-dire qu'une activité donnée déclenche l'affichage d'une autre activité en appelant la méthode startActivity avec un Intent mentionnant clairement le nom de l'activité.

Voici un exemple représentant la cohabitation listener Activity :

```
private Button create;
this.create = (Button)findViewById(R.id.SinglePlayerGame);
this.create.setOnClickListener(this);

public void onClick(View view) {
       Intent intent = null;
       if( view == create ){
             intent = new Intent(SinglePlayer.this, SinglePlayerLayout.class);
             startActivity(intent);
             this.finish();
       }
}
```

iOS

BDD

Après avoir effectué divers recherches, il s'est avéré que les mobiles utilisent un moteur de base de données relationnelles, accessible par le langage SQL. Dans notre cas il s'agit de SQLite 3. Sa particularité est de ne pas reproduire le schéma habituel client-serveur mais d'être directement intégrée au programme. L'accès à la base de données SQLite se fait par l'ouverture du fichier correspondant à celle-ci : chaque base de données est enregistrée dans un fichier qui lui est propre, avec ses déclarations, ses tables, ses index mais aussi ses données.

Android

Pour manipuler aisément les bases de données depuis l'application, nous avons crée une classe héritant de *SQLiteOpenHelper*. Cette dernière fournit des outils de manipulations. Un attribut y est instancié, il s'agit de la base de données elle même, de type *SQLiteDatabase*.

Nous y avons crée 3 tables, *PlayerAccount* sauvegardant toutes les informations sur les utilisateurs locaux, *System* concervant les propriétés du système, et enfin *Map* décrivant les informations relatives au cartes de jeu crées par l'utilisateur.

Ainsi de nombreuses fonctions ont été implémenté dans le but de simplifier les interactions avec cette base de données depuis l'application. Il est par exemple possible de créer un nouvel utilisateur local, modifier ses préférences, gérer les configurations systèmes comme la langue ou le volume du son, ajoûter de nouvelles maps ou même récupérer toutes les informations concernant un utilisateur.

Voici un exemple d'insertion d'un nouveau compte local dans la base de donnée. Rappelons que les tests d'existance du compte ont été fait depuis l'application même. Dans cet exemple vous verrez ainsi que nous commençons par récupérer les droits en écritures sur la base de données locale, puis nous créons un container qui servira à l'insertion de valeur dans la base. Et enfin l'insertion est faite. Nous terminons tout de même en fermant l'accès à cette base.

Il s'agit la d'un schéma classique de fonction d'interaction avec notre base.

```
/** ajout compte hors ligne **/
public long newAccount(String nomCompte){
base = getWritableDatabase();

ContentValues entree = new ContentValues();

entree.put("pseudo", nomCompte);
long var = base.insert("PlayerAccount", null, entree);

base.close();
return var;
}
```

iOS

Première utilisation

Création utilisateur

Gestion utilisateur

Gestion des préférences système

Création de carte (charger)

Création partie solo (tout)

Création partie multi (officielle)

4.1.2 Editeur de carte

Rendu

Interface utilisateur

Sauvegarde

4.1.3 Jeu

Moteurs

Au sein d'un jeu vidéo plusieurs types de moteurs sont mis en place. Chacun a un travail bien précis. Ici nous en retrouvons trois au total à savoir un pour le rendu graphique, un pour s'occuper de la physique et un dernier gerant les actions de l'intelligence artificielle. Commencons par le moteur de rendu.

Moteur de rendu

Contrairement à celui que nous avons vu dans la section précédente pour l'éditeur de carte [2] le moteur de rendu se doit être beaucoup plus léger car le jeu doit dans son ensemble rester le plus fluide afin d'offrir à l'utilisateur une meilleure experience vu qu'ici il faut en plus de gérer le rendu, s'occuper du physique [3]. , de l'intelligence artificielle [4]. et des divers sons [5]. qui seront joués au cours de la partie.

L'utilisateur ne pourra plus modifier la carte à sa guise et sera entièrement dépendant du moteur physique[3] c'est à dire par exemple qu'ici un bloc indestructible sera présent

2. Editeur de carte « voir section 4.1.2, page 37. »
3. Moteur physique « voir section 4.1.3, page 40. »
4. IA « voir section 4.1.3, page 44. »
5. Sons « voir section 4.1.3, page 44. »

tout au long de la partie et ne pourra pas être supprimé, il n'est donc plus necessaire de savoir quel type de sol se trouve dessous, de plus comme celui-ci ne peut pas être détruit et qu'il n'est pas animé son état sera toujours le même et ne correspondra qu'à une seule et unique image.

Un autre exemple est celui d'un sol inanimé tel que l'herbe où si il n'y a aucun bloc (destructible) au dessus en début de partie il en sera de même à la fin donc il ne nous est pas necessaire à chaque rafraichissement de regarder si un bloc existe dessus, ce test se fait en début de partie est sera valide jusqu'à la fin de celle-ci.

Cette remarque s'applique sur tous les objets dits *non animés* dont l'état ne changera jamais au cours du jeu et seulement eux.

Si l'on avait eu un sol animé représentant de l'eau, il aurait été composé de plusieurs images et aurait donc necessité un rafraichissement constant.

Concrètement ce que nous faisons ici à chaque début de partie est de créer une image vierge qui aura la taille de la carte affichée sur l'écran dans laquelle nous dessinerons tous les objets *non animés*.

Pour cela nous allons parcourir les deux matrices définies dans l'éditeur de cartes[2] et regarder s'il existe un bloc, si oui est-ce qu'il est destructible ?

Si ce bloc est destructible alors il nous est obligatoire de savoir ce qui se trouve en dessous. Nous allons donc stocker ce bloc dans une hashmap dont les cléfs sont les coordonnées de l'objet et dont la valeur est l'objet lui même et dessiner le sol sur l'image citée au dessus.

Si ce bloc est indestructible alors inutile de mémorisé le sol se trouvant au dessous.

Sinon s'il n'existe pas de bloc nous allons regarder si le sol est animé, si oui alors nous le stockons dans la hashmap comme un bloc destructible sinon nous le dessinons dans l'image comme un objet inanimé.

Voici un exemple concret de la methode décrite au dessus :

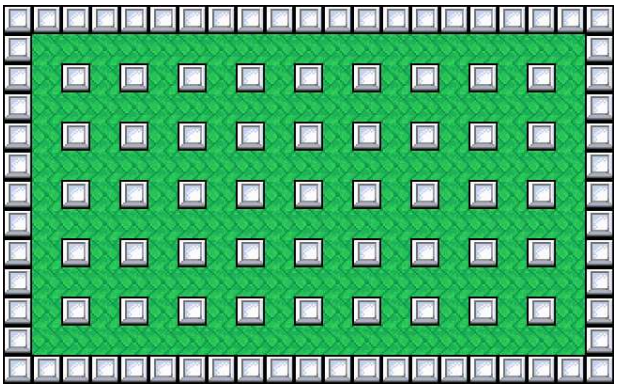


FIGURE 4.1 – L'image représentant la totalité des objets non animés

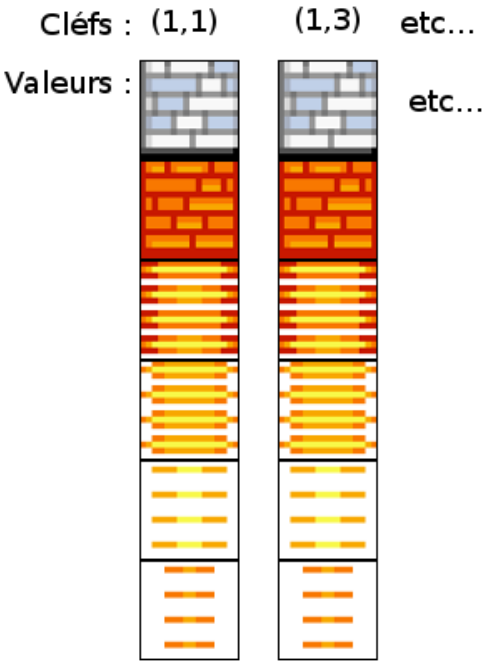


FIGURE 4.2 – La hashmap des objets animés

Les avantages d'avoir utilisé une telle structure est qu'ici au lieu de parcourir les 21*13*2 cases des deux matrices à chaque rafraichissement (c'est à dire toutes les 50 millisecondes environ) et d'afficher au minimum 21*13 objets pour le sol et 64 objets pour les bordures si la carte est vide donc énormement plus si il existe d'autres objets, nous n'affichons qu'une image plus au maximum 197 objets.

	Editeur de carte	Jeu
Meilleur des cas	337	1
Pire des cas	534	198

Le meilleur des cas ici décrit une carte vide donc composée que de sol non animé ansi que des bordures de la carte, ce qui représente dans le nouveau moteur de rendu une seule et unique image contrairement à l'ancien où chaque objet étant affiché indépendament cela equivaut à 337 objets.

Le pire des cas est une carte remplie au maximum de bloc destructibles, obligeant dans les deux cas à connaitre le type de sol se trouvant dessous.

Nous voyons très clairement les différences de coûts entre les deux methodes de rendu et l'optimisation qu'engendre la deuxième.

De plus ici l'utilisation de la hashmap permet dans un premier temps de retrouver directement un objet de par ses coordonnées mais aussi de ne pas avoir à parcourir n cases vides comme lors de l'utilisation des matrices car au fur et à mesure de la partie il existera de moins en moins d'objets donc garder une structure aussi grosse qu'une matrice n'est pas optimal.

Moteur physique

Un moteur physique est, en informatique, une bibliothèque logicielle indépendante appliquée à la résolution de problèmes de la mécanique classique. Les résolutions typiques sont les collisions, la chute des corps, les forces, la cinétique, etc. Les moteurs physiques sont principalement utilisés dans des simulations scientifiques et dans les jeux vidéos.

Ici notre moteur physique se contentera simplement de s'occuper des diverses colisions qu'auront les joueurs avec l'environnement les entourant ansi que les intéractions qu'auront les divers objets du décors entre eux ainsi qu'avec les joueurs.

Tout comme le moteur de rendu [6] le moteur physique d'un jeu doit être optimal dans les traitements qu'il a à effectuer vu que son utilisation est permanente au cour du jeu.

Afin d'optimiser ces traitements lors des collisions nous avons fusionné les deux matrices présentent dans l'éditeur de cartes [7] afin de n'en obtenir qu'une seule reprenant le principe décrit dans le moteur de rendu[2].

Cette matrice est composée de sept types d'objets différents à savoir :

6. Moteur de rendu « voir section 4.1.3, page 37. »
7. Editeur de carte « voir section 4.1.2, page 37. »

Objet	Description
EMPTY	Zone vide représentant un sol quelconque
BLOCK	Un bloc
GAPE	Un trou
DAMAGE	Une zone de dommages (piques, lasers, etc ...)
DANGEROUS_AREA	Une zone dangereuse
BOMB	Une bombe
FIRE	Feu resultant de l'explosion d'une bombe

Les zones dangereuses sont les zones qui seront touchées lors de l'explosion d'une bombe, ces zones sont spécifique à l'intelligence artificielle et leur permettent de savoir si elles sont en danger ou non.

Cette matrice est mise à jour lors de la pose et de l'explosion d'une bombe

De cette façon il est rapide de savoir si un joueur rentre en collision avec un objet quelconque ou si celui-ci subit des dommages.

Deplacements

Globalement la gestion des mouvements est identique sur Android comme sur iOS. Elle consiste à poser le doigt sur l'écran et à le faire glisser dans la direction souhaitée ainsi le personnage avancera jusqu'à que le doigt soit relevé.

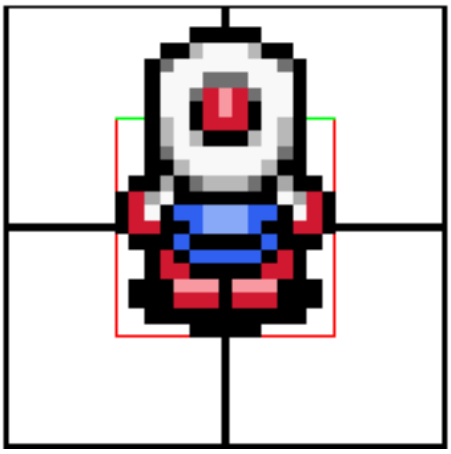
Quant à la gestion des colisions bien que chaque équipe utilise la matrice décrite ci dessus, les façons de concevoir la chose ont divergé proposant au final deux méthodes de rendu. Dans chacune le joueur se deplace verticalement, horizontalement ainsi qu'en diagonales.

1. Colisions sous Android

   Le principe de colisions sous Android a été de façon à faire "glisser" ou non le joueur lorsqu'il rencontre des obstacles.

   Pour cela nous allons étudier un mouvement qui sera celui vers le haut afin de mieux comprendre ce terme. Nous aurions pu prendre n'importe quel aute mouvement cela serait revenu au même.

   Il faut juste retenir qu'au mieux le joueur est dans une case et au pire dans deux et non dans quatre car lors d'un déplacement nous n'allons regarder que la face correspondant à cette direction comme le montre l'image ci-dessous



Le trait vert représente le côté que nous étudierons lors d'un déplacement vers le haut.

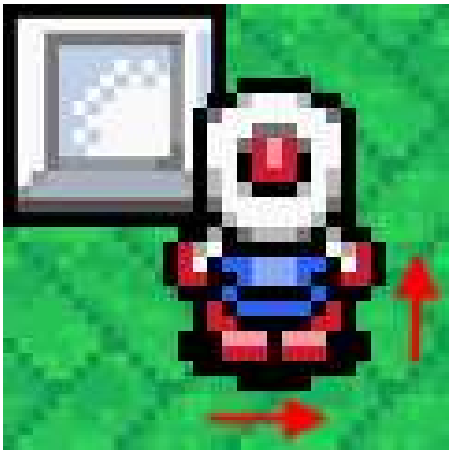
Nous partons du principe qu'il n'y a pas de vérification à faire tant que le joueur ne change pas de case car du moment où il y est entré c'est qu'elle est entierement traversable.

Il existe donc quatres types de colisions possibles :

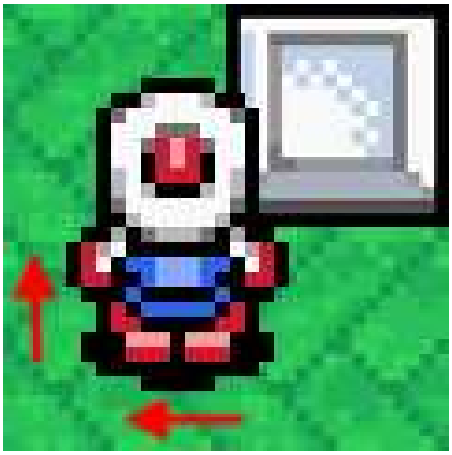
(a) Le premier cas correspond à celui où les cases sont traversables, il n'y a donc qu'à déplacer le joueur vers le haut.

(b) Le second est celui où la ou les cases en face sont des murs, il n'y a alors rien à faire, le personnage ne bouge pas.

(c) C'est à partir de ce cas que l'on rencontre le glissement cité plus haut. Ici nous essayons de monter mais la case de gauche correspond à un mur, étant donné la petitesse des écrans et la rapidité à laquelle le jeu se déroule il serait embetant pour l'utilisateur de devoir se decaler à droite de façon à être bien en face de la case libre. Nous avons donc pris en compte le cas où le joueur aurait dépassé la moitié du bloc intraversable, dans ce cas là nous le faisons glisser sur la droite de façon à ce que celui-ci se place convenablement en face de la case et monte normalement.



(d) Le dernier cas est l'opposé du précédent et marche de la même manière.



2. Colisions sous iOS

Gestion des bombes

– Threads

IA

Pathfinding

A*

Aléatoire

Prise de décision

Sons

Interface utilisaeur

Android

iOS

4.2 Serveur

4.2.1 Servlet

Comme expliqué précédement notre serveur est conçu grâce aux servlets. Après avoir décrit leur fonctionnement, nous allons montrer dans cette partie comment elles ont été utilisé.

En pratique

Au lancement du serveur la classe ContextListener est invoquée lorsque l'objet ServletContext est crée. Sa méthode contextInitialized(ServletContextEvent event) sera alors appelée, permettant ainsi de définir des objets communs à toutes les servlets, tels qu'un accesseur à la base de données, ou le tableau qui va contenir les utilisateurs connectés. Le requetes font appel à la fonction post des servlet. Le flux entrant étant de type JSON, il faut déserialiser le flux dans un objet correspondant. Exemple l'utilisateur envoie son userName et son mot de passe crypté dans un tableau, sérialisé en JSON, pour pouvoir récupérer les informations nous procédons comme suit :

```
BufferedReader req =
    new BufferedReader(new InputStreamReader(request.getInputStream()));
OutputStreamWriter writer =
    new OutputStreamWriter(response.getOutputStream());
String message = req.readLine();

if (message != null) {
  response.setContentType("text/html");

   // désérialisation des infos de l'utilisateur dans une arraylist
  JSONDeserializer<ArrayList<String>> jsonDeserializer =
    new JSONDeserializer<ArrayList<String>>();
  ArrayList<String> identifiers;
  identifiers = jsonDeserializer.deserialize(message);

  username = identifiers.get(0);
  password = identifiers.get(1);

  ...}
```

La sécurité

Ce serveur de jeu étant hebergé sur internet et contenant des informations sensibles d'utilisateurs, tels que des mots de passes, il était crucial d'instaurer des règles de sécurité et de cryptage.

En effet lors des inscriptions ou connexion au serveur pour le mode multijoueur, les mots de passes sont tout d'abord cryptés côté client et ensuite encapsulé dans un flux JSON, pour être envoyé au serveur. Il stockera ainsi la chaine de caractères extraite de l'objet déserialisé. De cette manière à aucun moment les données confidentielles ne transiteront en clair.

De plus un mécanisme semblable aux sessions est en place. Dans la confirmation de connexion ou d'inscription, une userKey est générée. Elle correspond en réalité à l'identifiant de session envoyé par le serveur. Une fois associée à l'username correspondant, le tout est ajoûté dans le tableau d'utilisateurs connectés. Cet userKey est ensuite nécessaire pour contacter les servlets suivantes. Si cet identifiant n'est pas envoyé ou n'est pas présent dans le tableaux des utilisateurs connectés, il sera alors impossible d'accéder aux ressources du serveur.

4.2.2 BDD

La base de données du serveur n'est pas très complexe. En effet elle ne fait qu'accueillir les couples userName/password des utilisateurs dans la table Users. Pour son accès, chaque servlet peut récupérer un objet de type Connection, instancié à l'initialisation du serveur. Il permettra à son tour de récuperer un objet de type Statement. L'application va l'employer pour transmettre des instructions à la base. Exemple d'insertion :

```
Connection connection =
  DriverManager.getConnection("jdbc:mysql://127.0.0.1/Bomberklob", "user","user");
Statement theStatement = connection.createStatement();
theStatement.execute(
     "INSERT into Users VALUES ('"+ username +"','"+password+"')");
```

Bien evidemment cette adresse est remplacée par une variable, elle aussi présente dans le ContextListener, contenant la véritable adresse de la base de données.

Chapitre 5

Manuel d'utilisation

5.1 Menus

TODO ludo

5.2 Jeu

5.3 Editeur

Chapitre 6

Discussion

6.1 Problèmes

6.1.1 Android

6.1.2 iOS

6.2 Améliorations

6.2.1 Jeu

6.2.2 Serveur

- Pooling
- Session

Chapitre 7

Conclusion