TER FMIN200

–

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

Olivier BONVILA Kilian COUSEIN Ludovic PITIOT
Benjamin TARDIEU

Nous tenons à remercier tout particulièrement :
– Notre tuteur : M. Deruelle pour ses conseils et son soutien technique.
– Les différents éditeurs des logiciels qui nous ont servis pour le développement mais aussi pour l'écriture de ce rapport.
– La faculté des Sciences pour les nombreuses connaissances qu'elle nous a permis d'acquérir, bien évidemment transmises par les professeurs mais également par les échanges inter-étudiants.

Table des matières

Chapitre 1

Introduction

Nous sommes amenés, dans le cadre de notre première année en master informatique à la faculté des Sciences de Montpellier, à travailler sur l'élaboration complète d'un projet, de son analyse à la conception puis à sa programmation. Tout au long de ce deuxième semestre, le projet nous permet de comprendre quelles sont les phases de développement d'un projet et comment celui-ci doit être conduit. Nous pouvons par ailleurs mesurer notre capacité à réagir face à des problèmes en nous impliquant dans ce projet. Le projet que nous avons choisi est de réaliser sur deux systèmes d'exploitations de téléphone différents un jeu de type Bomberman.

Ce projet nous permet de mettre en application l'enseignement que nous avons acquis tout au long de ce semestre. Ce dernier étant spécialisé pour chaque étudiant, il nous a permis de travailler en collaboration avec des personnes aux capacités différentes et d'ainsi mettre nos connaissances en commun. Mais il permet aussi de se faire une idée du travail demandé dans le monde des entreprises et d'ainsi nous préparer à notre stage que nous devrons réaliser l'an prochain.

Afin de comprendre la démarche que nous avons utilisée pour mener ce projet à son terme, notre rapport se compose de trois grandes parties :

Tout d'abord, dans une première partie nous présentons le cadre général de notre projet, c'est-à-dire le groupe de projet, les outils de developpement ainsi que la manière dont nous nous sommes organisés pour mettre à bien le developpement de ce dernier. Ensuite dans une seconde partie, nous présenterons le travail d'analyse que nous avons effectué pour pouvoir ensuite, dans une troisième partie, expliquer le developpement du projet.

Chapitre 2

Présentation

En tant que première année de master, nous avons dû developper un projet tout au long de ce semestre. Chaque année une liste de projets est présentée aux étudiants. Tous les étudiants doivent former des groupes pour développer l'un des projets choisi. Puis ces derniers sont associés à un tuteur. Etant donné que nous voulions développer notre propre projet, nous avons formé un groupe de quatre personnes : Olivier BONVILA, Kilian COUSEIN, Ludovic PITIOT et Benjamin TARDIEU. Ensuite, pour pouvoir valider notre sujet, nous avons dû trouver un tuteur voulant bien s'occuper de notre tutelle. Mr Laurent Deruelle a gentiment accepté de s'occuper de notre groupe, mais pour que ce dernier soit entièrement validé, des modifications du sujet ont été nécessaires.

2.1 Projet

Etant donné que nous voulions principalement créer une application iPhone, nous avons demandé à développer notre propre projet. Un jeu de type bomberman. Mais ce dernier a dû subir des modifications pour être approuvé. Nous avons donc dû développer le jeu sous iPhone et sous Android. Le but étant de comparer la différence de développement entre les deux types de téléphones et de développer des fonctionnalités en rapport avec les parcours d'enseignement que nous avons choisi ce semestre.

Un jeu de type Bomberman est un jeu d'action dont le but est simple. Le joueur incarne un poseur de bombes et ce dernier doit faire exploser ses ennemis pour pouvoir gagner la partie. A ceci s'ajoute toute une liste de bonus et malus permettant de complexifier le jeu et de le rendre plus amusant.

Pour pouvoir rapprocher le développement de cette application à notre parcours d'enseignement. Nous avons choisi de développer un mode solitaire qui permettra de jouer contre une ou plusieurs intelligences artificelles qui est en rapport avec le cursus Ingéniererie de l'Inteligence Articielle (I2A) qui nous est enseigné. Ensuite pour ce qui est du parcours Combinatoire, Algorithmique, Sécurité et Administrat nous avons décidé d'implémenter un mode multijoueur qui permettra à plusieurs joueurs connectés en Wi-Fi de jouer en réseau grâce à un serveur qui combinera un serveur d'applications et un serveur web. Puis pour ce qui est du parcours Données, Interaction et Web (DIWEB), la partie serveur permettra de palier à l'enseignement de ce dernier parcours.

2.2 Histoire du jeu

2.2.1 Histoire

Le Bomberman est un jeu culte des années 80. Il a été conçu par l'éditeur japonais Hudson Soft. Sa silhouette, reconnaissable parmi toutes, est celle d'un poseur de bombes muni d'un casque avec des membres blancs et un corps bleu. Cependant, à la sortie du premier Bomberman en 1983 sur MSX, notre héros n'arborait pas cette apparence puisqu'il s'agissait en fait d'un ennemi issu de Lode Runner. Ce n'est qu'en 1985 que l'aspect de Bomberman tel que nous le connaissons aujourd'hui est utilisé lors du portage sur NES. C'est cette adaptation qui a réellement permis de faire connaitre le personnage en partie grâce à des mécaniques de jeu très simples.

Au fil du temps et des versions, le poseur de bombes gagne des pouvoirs de plus en plus grands. Ainsi, dans le premier épisode, le personnage ne sait que poser des bombes. Les bonus du jeu se limitent aux bombes, aux flammes et aux bottes permettant respectivement d'augmenter le nombre de bombes que l'on peut porter, la porté des explosions et la vitesse de déplacement. Dans les versions suivantes le personnage peut, après avoir trouvé les bonus correspondants, lancer des bombes ou les pousser, traverser les murs, etc. Certaines versions ont vu apparaitre des montures apportant de nouveaux pouvoirs et modes de déplacement.

2.2.2 Principe

Pour gagner, il faut détruire ses ennemis à l'aide de bombes et obtenir des bonus ou malus qui permettent d'augmenter ou diminuer ses chances de gagner. La maniabilité, le rythme effréné et la possibilité de s'affronter à quatre joueurs ont grandement contribué à sa popularité (dix millions d'exemplaires vendus depuis sa création). Sa simplicité, aussi bien dans le gameplay que dans les graphismes, semble être l'élément moteur de son succès puisque toutes les tentatives de suites, augmentant le nombre de bonus ou passant à la 3D, ont été boudées par le public. L'absence d'évolution du concept n'empêche aucunement à ce jeu d'être l'un des plus addictif du genre, et à son personnage de connaître une immense popularité.

2.3 Plate-formes de développement

2.3.1 Android

Android est un système d'exploitation open source fournis par Google et developpé par la startup 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, 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 ANSI 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.

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

Le SDK d'Android comporte les outils de base pour developper une application en Java, le developpement en ANSI et C++ necessitant d'utiliser le NDK. 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 2.0 qui n'est pas encore prit en charge par celui-ci, sinon le SDK 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 est une variante du format JAR qui est utilisée pour la distribution et l'installation de composants regroupés sur le système d'exploitation Android.

2.3.2 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 concevoir 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 (2001), l'iPhone (2007) et l'iPad (2010). Apple a développé deux systèmes d'exploitations (Mac OS X et iOS) permettant de contrôler ses différents produits, notamment les ordinateur Macintosh, l'iPod et l'iPod touch.

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 (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 », une couche « Core Services », une couche « Media » et pour finir une couche « Cocoa ». 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 et la rapidité du ANSI. C'est une extension du ANSI qui ne permet pas l'héritage multiple contrairement au C++. 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 SDK d'iOS fournit les principaux outils nécessaires 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, il est possible 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.

2.4 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 (SVN)) distribués sous Licence Apache et BSD.

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 où pour pouvoir être mis à jour, il faut à tout prit avoir modifié un fichier à partir de la dernière version de celui-ci.

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

De plus *Google Code* met aussi à disposition de ses utilisateurs des wikis. Un wiki est un site Web dont les pages sont modifiables par les développeurs afin de permettre l'écriture et l'illustration collaboratives des documents numériques qu'il contient.

Malgré de nombreuses réunions quotidiennes afin d'organiser au mieux le développement de cette application dont nous ignorons tout à nos début. Nous avons utilisé le wiki qui nous a été fourni afin de partager au mieux les découvertes que nous faisions au fur et à mesure ainsi que les articles qui nous semblaient interressants.

En ce qui concerne la répartition du travail, celle-ci s'est faite naturellement car seulement la moitiée du groupe possédait de quoi développer sous iOS, de là nous avons donc créé deux équipes une sous Android et l'autre sous iOS.

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 [1].

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 multijoueur. 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 également 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

1. Modélisation « voir section 3.2, page 10. »

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'âge 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 débutant, 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 l'interagir des joueurs entre eux, via leur mobile. Nous parlons donc ici des parties multijoueur.

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 développer des applications iOS et Android, nous a imposé le choix du langage. En effet pour développer des applications iPhone, il faut utiliser le langage Objective-C, puis pour les applications Android c'est le langage Java qui est utilisé. C'est deux langages ont une syntaxe complétement différente mais sont quand même très proche car ce sont des langages orientés objets. Grâce à cela nous avons pu mettre au point une modélisation générale de l'application que nous avons ensuite adapté à chaque langage.

Nous avons décidé de développer en anglais car premièrement c'est la langue la plus utilisé dans le monde de la programmation et deuxièmement car la syntaxe des langages est toujours en anglais. Cela permettra à n'importe quel utilisateur de n'importe quelle nationalité 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 documentation pour l'application iPhone et 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 développer les deux applications selon le modèle Modèle Vue Contrôleur (MVC). 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 compte local. Les menus de l'application ont été réalisés pour que l'utilisateur puisse les utiliser de manière intuitive. Ils se divisent en quatre 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 s'il n'est pas 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.

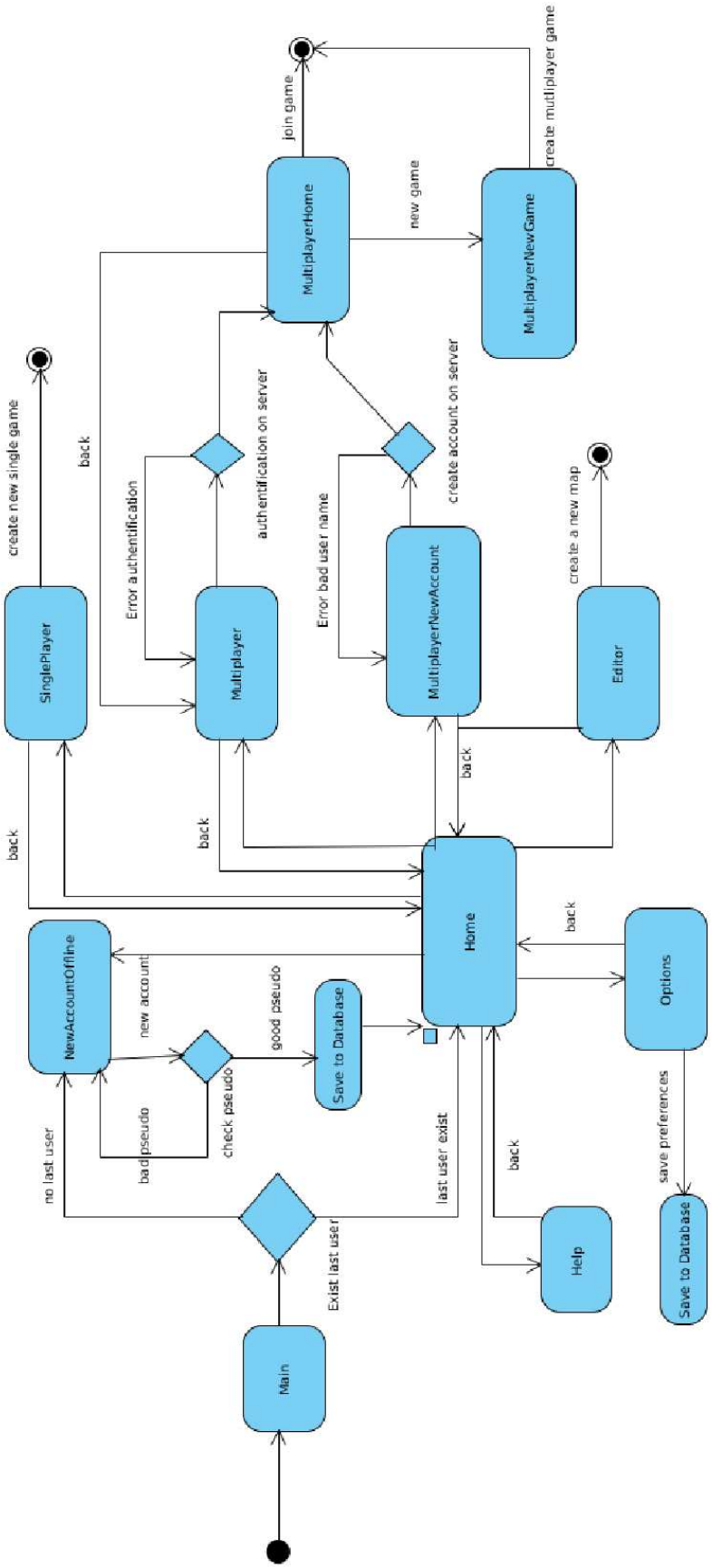
Suite à ces deux sections vient ensuite l'éditeur de cartes. C'est depuis ce menu que vous créerez une nouvelle map de jeu local ou éditerez l'une d'entre elles. Choisisez votre nom de carte et l'éditeur s'ouvrira ensuite à vous. Il vous sera possible, à la fin, d'enregistrer votre carte si vous désirez 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 créé 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 désirez, et il sera possible possible d'éditer ou choisir son compte.



12

FIGURE 3.1 – Diagramme d'activité

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(attribut 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).

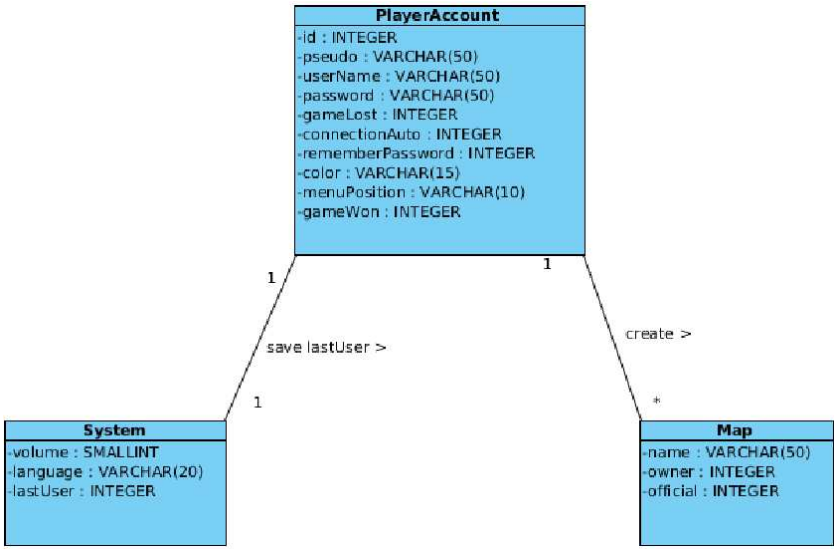


FIGURE 3.2 – Diagramme de classe Base de données

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.

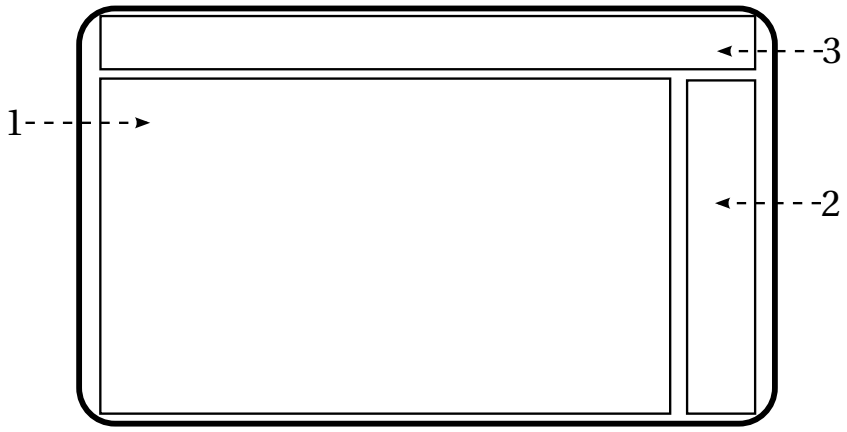
Modèle

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. 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.

L'intelligence artificielle dans notre jeu utilise des algorithmes de recherche opérationelle.

Pour une meilleure expérience de jeu nous avons séparé l'intelligence artificielle en trois niveaux.

- Facile
- Moyenne
- Difficile

Nous avons utilisé deux types d'algorithmes de recherche opérationelle basés sur le pathfinding.

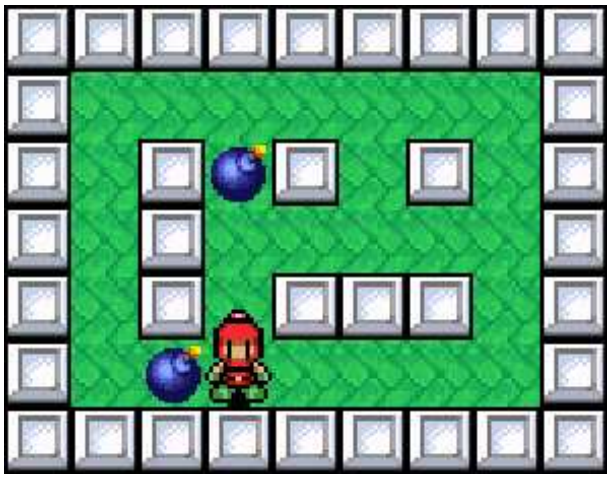
Pathfinding Le premier algorithme basé sur le parcours en largeur est utilisé quel que soit le niveau de l'intelligence artificelle choisie contrairement au second qui n'est utilisé que pour la difficulté moyenne et difficile.

Parcours en largeur

L'algorithme du parcours en largeur dans notre cas, consiste à partir d'un sommet S, lister d'abord tous les voisins de S pour ensuite les explorer un par un. Ici nous allons donc regarder toutes les cases autour de nous puis regarder tous leurs voisins et cela ainsi de suite jusqu'à trouver un point correspondant à nos attentes.

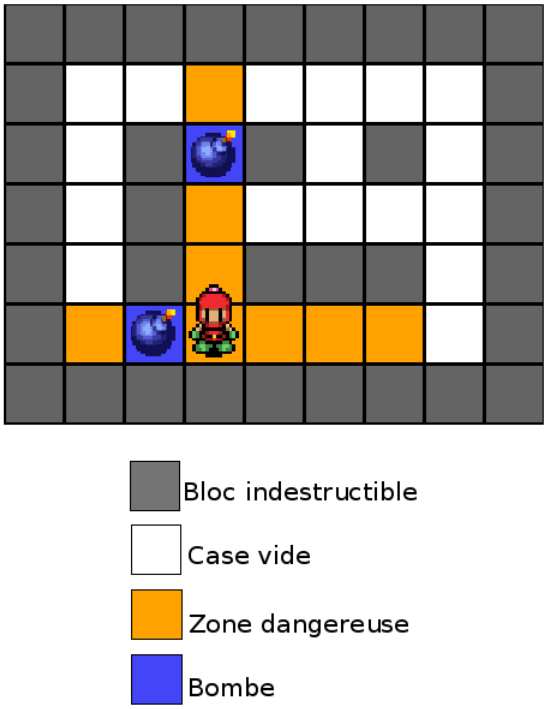
Le contexte est le suivant, un bot découvre qu'il est sur la trajectoire d'explosion d'une bombe est va donc fuir vers la case sûr la plus proche or il n'a aucune idée d'où elle se trouve.

L'image suivante représentera la situation initiale :



Comme dit précédemment il n'y a que les murs ou les bombes que l'on ne peut pas traverser sinon tous les autres objets ou joueurs sont traversables. Nous considerons que les bombes ont ici un champs d'explosion en nombre de cases de 5.

Représentons la carte ci-dessus d'une façon plus parlante en remplacant les divers objets mis à par le joueur par des couleurs leur correspondant, à savoir :



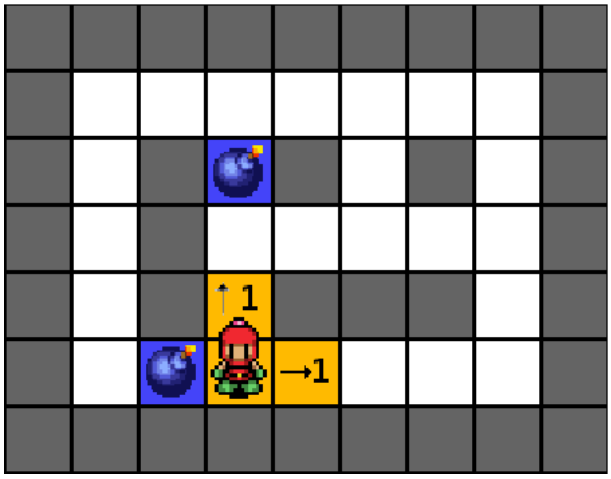
Mettons nous à present à la place du bot.

Nous allons donner un poid aux cases que nous allons parcourir correspondant à la distance par rapport à la case initiale, ainsi qu'une direction qui correspondra à la direction initiale que le bot devra empruter pour utiliser ce chemin, c'est à dire par exemple que tout chemin découvert dont l'origine est une case à droite de la notre aura comme direction droite.

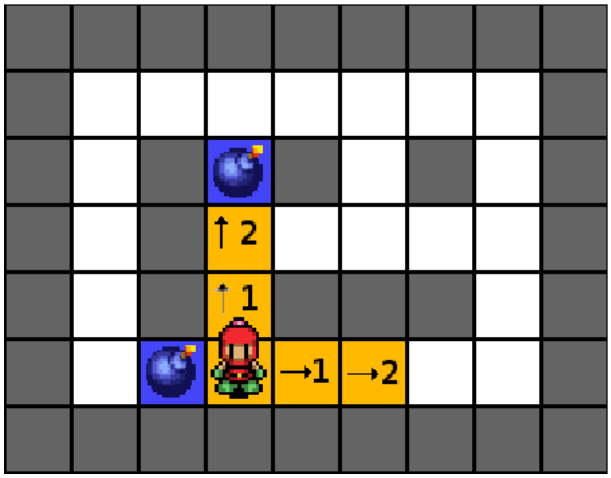
A partir d'une case donnée, nous ne regarderons que les voisines ayant un poids de 0 car si leur poids est différent cela voudra dire que nous les avons déjà vu precedemment et bien évidemment, nous ignorerons les murs ainsi que les bombes.

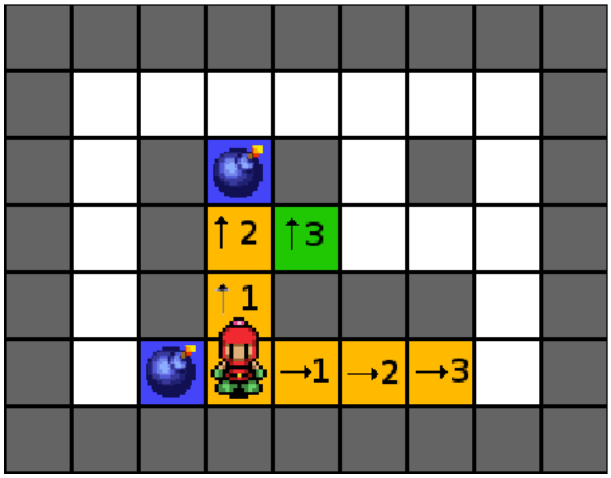
Nous avons changé la valeur des cases intraversables pour ne pas confondre avec le poids des cases visitées.

Appliquons l'algorithme de parcours en largeur aux cases voisines de la notre.



Toutes les cases découvertes étant considérées comme dangereuses (voir le schéma précédent) nous continuons à appliquer l'algorithme jusqu'à arriver à notre but.





Ici nous avons découvert une case non dangeureuse en $(13, 6)$.

Nous récupérons donc la direction enregistrée dans cette case et bougeons en fonction de celle-ci.

Si plusieurs cases avaient été découvertes le choix aurait été arbitraire car elles auraient toutes été à la même distance.

A*

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 objets présents sur la carte et des actions effectuées par les autres joueurs.

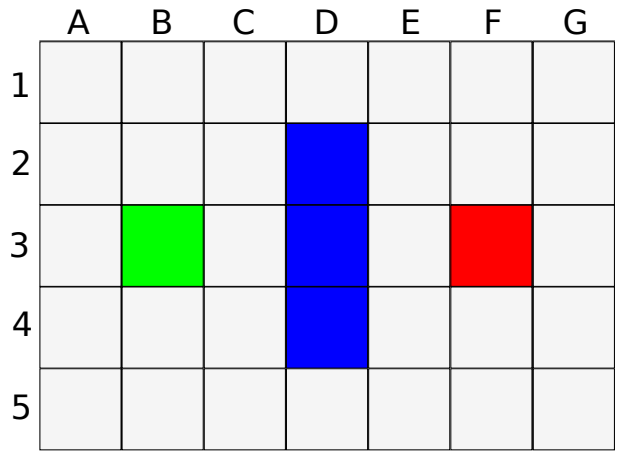
L'algorithme de recherche A* a pour but de rechercher un chemin dans un graphe entre un nœud initial et un nœud final tous deux préalablement définis. A* permet de trouver l'un des meilleurs (mais pas forcément le meilleur) chemins existant entre un point A et un point B (il retourne le premier chemin trouvé).

La force de cet algorithme est le temps de calcul et l'exactitude des résultats, contrairement à Dijkstra 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*.

Sachant qu'il peut y avoir jusqu'à trois bots et que l'intelligence artificielle doit régulièrement 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 marche, nous allons nous aider d'un dessin représentent une carte avec un point A (départ) affiché en vert, un point B (arrivée) en rouge, et où les cases en bleu représentent les murs.



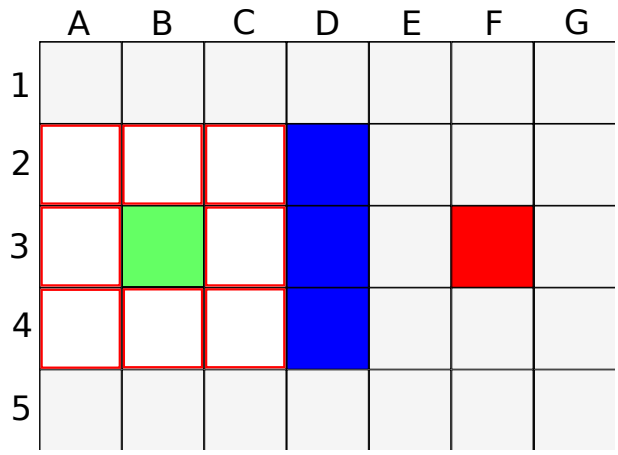
La premier chose que l'on peut observer, c'est que la carte est divisée en cases. Chaque case de la matrice représente un node qui peut être soit traversable, soit non traversable. Dans l'application, il n'y a que les murs ou les bombes que l'on ne peut pas traverser sinon tous les autres objets ou joueurs sont traversables. Le but de l'algorithme est donc 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 explications du déroulement de l'algorithme. Tout d'abord, il faut savoir qu'un joueur peut se déplacer dans toutes les directions, donc nous allons ajouter toutes les cases adjacentes à la listeOuvert qui sont traversables, il y en a huit (A2, B2, C2, A3, C3, A4, B4, C4).

Ce qui nous donne :



Les carrés avec un contour rouge sont les carrés présents dans la listeOuverte et les carrés qui ont une couleur un peu plus foncée que les autres sont ceux 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 » :

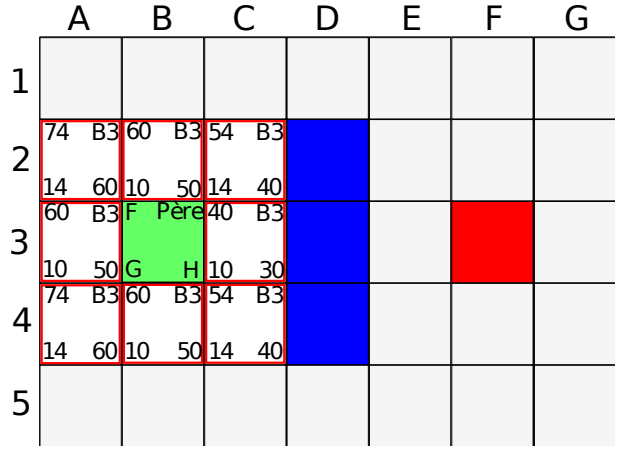
G : c'est le coût de mouvement pour aller de la case A à une case donnée sur la grille, en suivant le chemin généré jusqu'à cette dernière.

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.

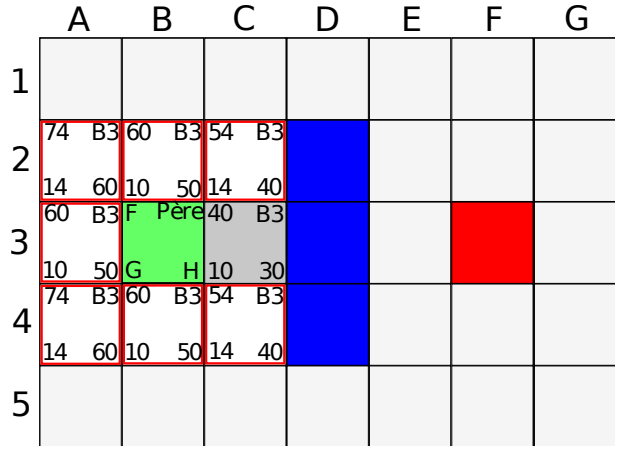
F : c'est G + H

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 :

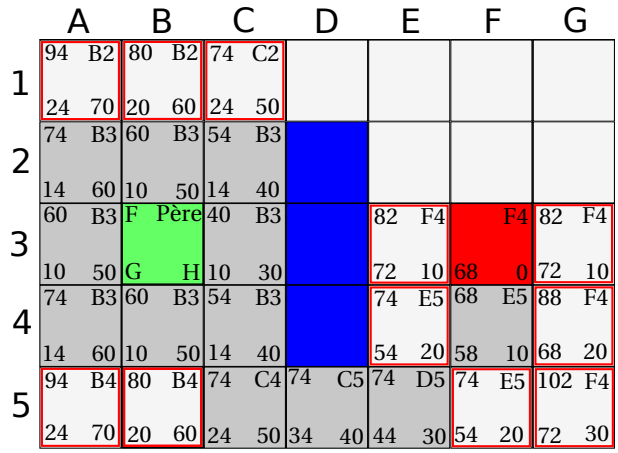


Après avoir ajouté toutes les cases adjacentes à la case courant, il suffit de prendre la case qui a le plus petit coût F et ensuite de la rajouter dans la listeFermée et de la supprimer de la listeOuverte. Nous obtenons donc :

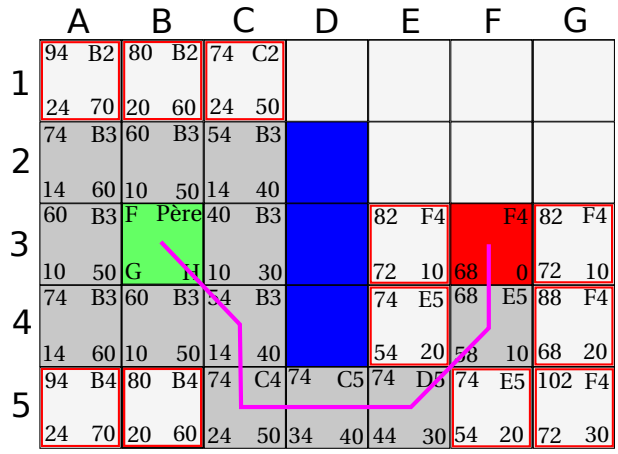


Ensuite on regarde toutes les cases adjacentes à la dernière case ajoutée dans la listeFermée. Si elles se trouvent déjà dans la listeOuverte, on vérifit que leurs coût soient inférieur au coût de la case correspondante déjà dans la listeOuverte, si oui alors on la remplace sinon on ne fait rien.

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 nous pouvons levoir sur le schéma precedent, le chemin qu'a 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 divisée en trois grandes parties : le modèle, la vue et le contrôlleur.

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 méthodes qui vont permettre 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 associée 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 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 ou death match.

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 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.

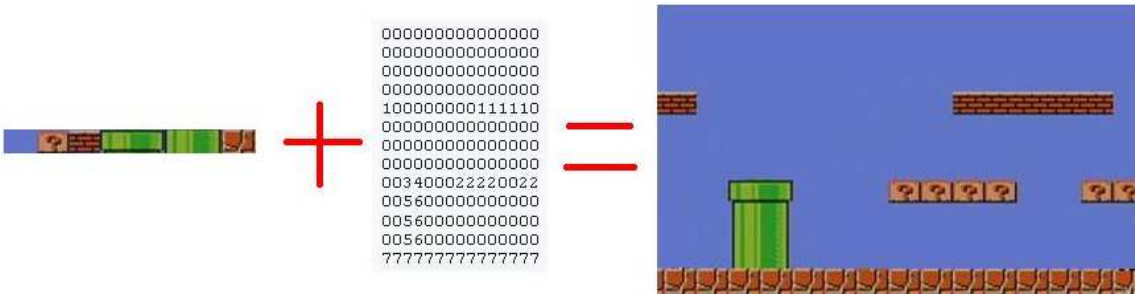
Pour la pause des bombes un simple boutton est mis en évidence en bas à droite de l'écran. Celui-ci est assez gros pour que l'utilisateur n'est pas à appuyer sur une zone trop précise en cas de manipulation rapide.

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 ou la Game Boy. Nous avons donc conçu le moteur du jeu selon le principe du Tile Mapping 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 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. Le fichier XML peut être réprésenté sous forme d'un arbre. Les fichiers XML ressemble tous plus au moins a ce genre d'arborescence :

```
<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.

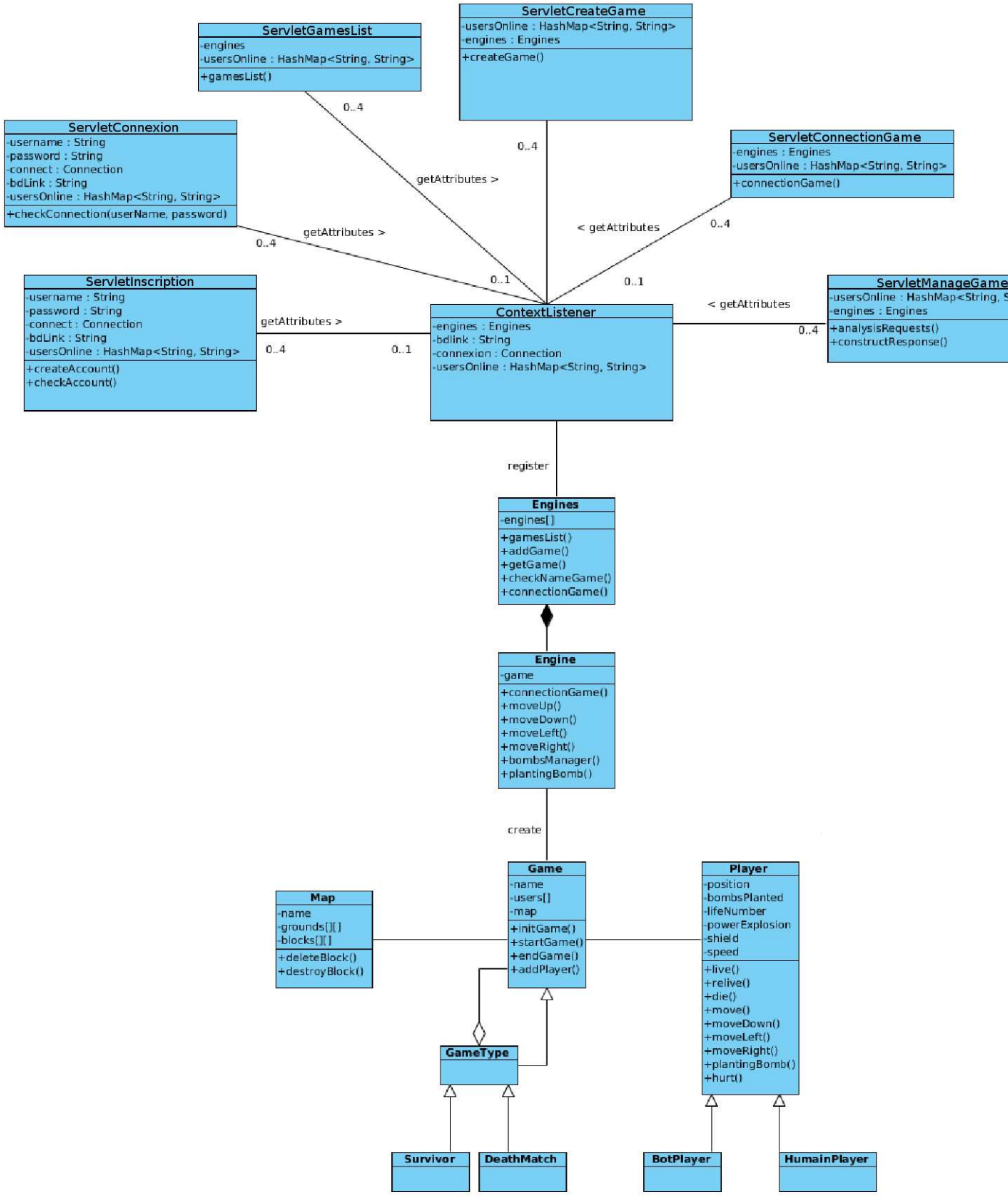


FIGURE 3.3 – Serveur

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. 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 Hypertext Transfer Protocol (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>,
"game": {"name":<name>, "type":<type>, "map":<map>, "ennemiesNumber": <ennemiesNumber>}}

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

ServletConnectionGame:
```

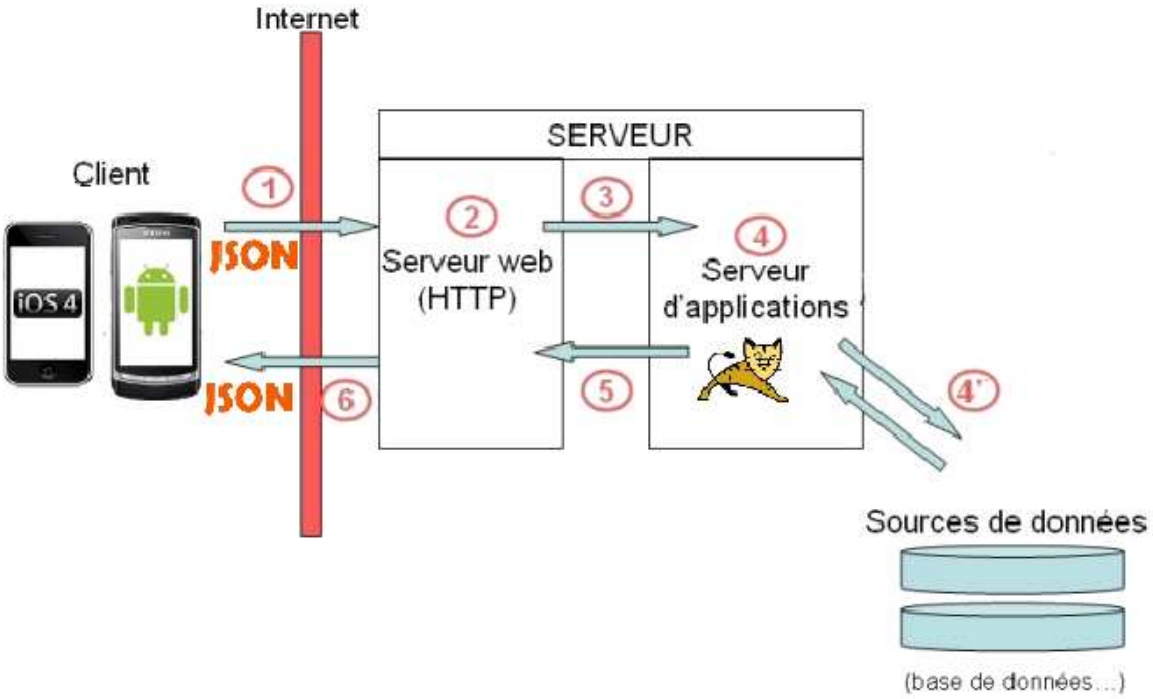
```
Player => Server:
{["userKey", "gameName"]}

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). Aléatoire 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

Lors du développement sous en Android et iOS, nous avons été confronté à divers problèmes. Notamment des problèmes liés aux différences entre ces deux systèmes d'exploitations. Dans un premier temps, le langage utilisé n'est pas le même, le premier utilise le langage Java alors que l'autre utilise l'Objective-C. Ces deux langages étant des langages orientés objets, la difficulté réside surtout dans la différence des syntaxes. Ensuite, lors de l'implémentation de notre diagramme de classe notre premier soucis a été les différences de modélisation des vues et des controlleurs. En effet sous Android, les écouteurs d'une vue sont directement situés dans le controlleur de cette dernière alors que sous iOS les écouteurs peuvent être implémentés soit dans la vue soit dans son controlleur. Pour respecter au maximum le modèle MVC, nous avons choisi de placer les écouteurs dans la vue sous iOS (ce qui est recommandé par Apple).

Chapitre 4

Organisation

4.1 Conventions de code

Par soucis de réutilisabilité et de partage de notre projet, nous avons choisi d'utiliser l'anglais pour la documentation du code mais aussi pour le code en lui-même (nom de fonction, nom de variable, etc).

4.2 Répartition du travail

La répartition du travail nous a été indirectement imposée par la structure du projet.

Tout d'abord la partie analyse du projet c'est fait par le groupe entier car l'analyse c'est la base d'un projet, donc nous avons choisi de la faire tous ensemble pour essayer d'avoir une analyse représentative du groupe et plus poussée. Comme le projet est divisé en deux parties, la partie développement Android et la partie iOS, de ce fait comme seulement Kilian Cousein et Benjamin Tardieu posséder un Macintosh, ces derniers ont développés la partie iOS et l'autre partie à été développée part Olivier Bonvila et Ludovic Pitiot. Enfin en ce qui concerne la partie serveur, c'est Ludivic Pitiot qui l'a implémenté.

4.3 Gestion du projet

Pour gérer le projet, nous avons mis en place un serveur SVN hébergé par GoogleCode (`http://code.google.com/p/bomberman-android-ios/`). Nous avons aussi utilisé le wiki et le gestionnaire de problème fourni par GoogleCode.

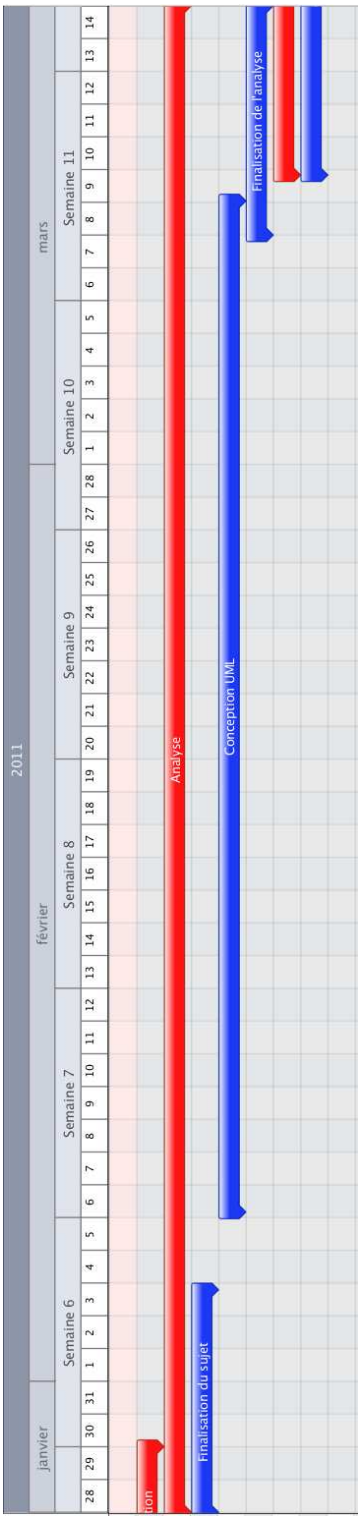
Ensuite, nous avons fait des réunions pour contrôler l'avancement du projet, environ deux fois par semain. Mais lorsque nous ne pouvions pas ou lorsque que nous avions des problèmes, nous avons utilisé Gmail et son chat.

4.4 Gestion du temps

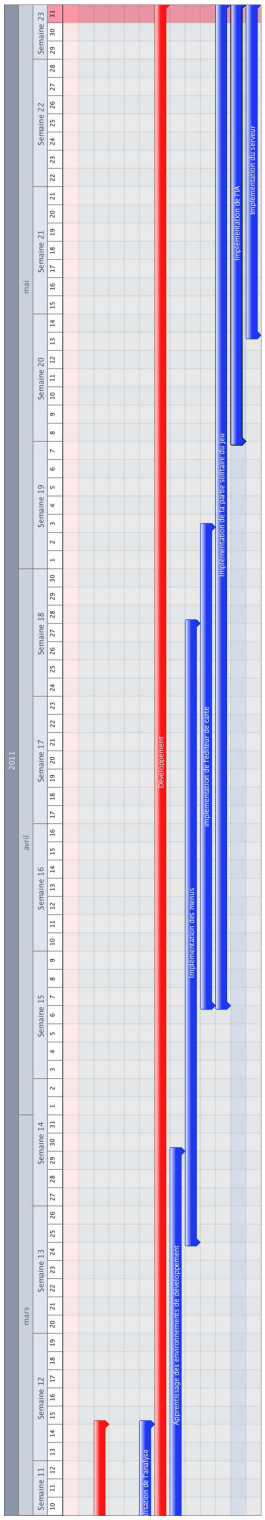
La réalisation de notre projet se divise en trois axes. Voilà les différents diagrammes de Gantt de ces derniers.

Organisation

Conception


2011
janvier
février
mars
Semaine 6
Semaine 7
Semaine 8
Semaine 9
Semaine 10
Semaine 11
Finalisation du sujet
Analyse
Conception UML
Finalisation de l'analyse

Développement



Chapitre 5

Développement

5.1 Mobile

5.1.1 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 :

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

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

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

Comme Android, nous avons utilisé les widgets [2] pour réaliser notre interface graphique. iOS fournit lui aussi une API couplet à une documentation très complète facilitant la réalisation d'une interface utilisateur ergonomique. Parmi les widgets les plus utile, nous retrouvons les UIView, UITableView, UITableViewCell, UILabel, UITextField, UIButton qui permettent respectivement de créer une vue, un tableau, une cellule du tableau, des labels, des champs de texte et des boutons.

2. Widget est un élément de base d'une interface graphique avec lequel un utilisateur peut interagir (par exemple : une fenêtre, une zone de texte, un bouton, etc).

Pour manipuler ces différentes composant graphique, il est possible et beaucoup plus pratique d'uliser Interface Builder pour réaliser des menus. Interface builder permet de gagner du temps lors de la réalisation d'une interface graphique standard (de type menu). Cette réalisation est possible via un outil visuel et non des lignes de code, Interface Builder va créer des fichiers xib [3] qui vont générer à la compilation des fichiers nib [4]. Ce dernier stock les informations relative à l'interface sous forme d'un fichier xml, mais nous avons pas besion de savoir réellemet comme ça fonctionne car il est inutile de modifier directement le fichier xml.

Ensuite pour controller les différents l'élements de l'interface préalablement créées avec Interface Builder, il suiffit de les lier au code de la vue via ce dernier. Grâce a ces liaisons, il nous sera possible de les modifiers ou les controllers avec du code et donc de pouvoir les accorder avec les données du modèle.

La gestion des écouteurs se fait grâce au même procédé qui est utilisé pour lier les éléments de l'interface avec la vue sauf que ça créra une métohde au lieu de créer un champs. Voilà un exemple d'écouteur :

```
- (void)playerAction:(id)sender {
    if (sender == playerWhite) {
        self.editorInformation.colorPlayer = @"white";
    }
    else if (sender == playerBlue) {
        self.editorInformation.colorPlayer = @"blue";
    }
    else if (sender == playerRed) {
        self.editorInformation.colorPlayer = @"red";
    }
    else if (sender == playerBlack) {
        self.editorInformation.colorPlayer = @"black";
    }

    [editorInformation changeTool:@"player"];
}
```

Pour finir, l'un des principaux point de la gestion de l'interface graphique repose sur le modèle MVC et donc chaque vue doit être associée à un controlleur.

3. xib est l'acronyme de Xml Interface Buillder.
4. nib es l'acronyme de NeXT Interface Builder.

Première utilisation

Création utilisateur

Gestion utilisateur

Gestion des préférences système

Création de carte (charger)

Création partie solo (tout)

Création partie multi (officielle)

5.1.2 Editeur de carte

Le modèle

La carte est la principale information que l'éditeur de carte doit gérer. Pour représenter une carte, nous avons choisi de la représenter sous la forme de deux matrices ayant les mêmes dimensions que la carte. La première matrice contient des *Objects* représentant le sol et la deusième contient elle aussi des *Objects* mais ces derniers représentent des blocs et non le sol. Pour gérer les deux matrice nous avons utilisé des *NSMutableArray* qui sont des tableaux Objective-C que l'on peut modifier, ils contiennent eux aussi des *NSMutableArray* car il n'existe pas des constructeurs de matrice comme en JAVA. Ensuite l'autre information que la carte doit possèder c'est la position de départ des joueurs. Nous avons utilisé un *NSMutableDictionary* permettant de faire la liaison entre la couleur du joueur et son point de départ.

Lorsque l'utilisateur à terminé de créer ça carte, il a besoin de la sauvegarder. Pour cela nous avons donc utilisé la sérialisation des objets et lors de l'enregistrement de la carte ça génère un fichier « .klob » qui contiendra les informations necessaires pour la reconstruction de la carte lors de son chargement qui la reconstruira en la déserialisant. Pour utiliser la serialisation en Objective-C, nous avons juste besoin implémenter le protocol *NSCoding* qui comporte deux methode à définir : - *(id)initWithCoder :(NSCoder *)aDecoder* permettant de déserialisé et - *(void)encodeWithCoder :(NSCoder *)aCoder* s'occupant de la serialisation.

La vue

Pour facilité le développement mais aussi ça réutilisabilité, nous avons découpé la vue de l'éditeur de carte en trois zones. Chaqu'une de ces vues est associée à un controlleur qui permettra de faire la liaison avec le modèle. Pour concevoir les vues nous avons utilisé la classe UIView qui est prévue pour réaliser des vues pour une interface graphique sur iPhone.

Tout d'abord, la vue la plus important que nous avons créé, c'est la vue affichant la carte permettant au joueur d'intéragir avec celle-ci et de pouvoir créer des cartes selon ses envies. Pour réaliser cette intéraction, nous avons dû surcharger trois methodes de la classe *UIResponder* : *touchesBegan*, *touchesMoved* et *touchesEnded* permettant de gérer tous les évènement générés lorsque l'utilisateur appuiera sur la vue. La methode *touchesBegan* permet de récupérer l'évènement lorsque l'utilisateur va appuyer sur la vue, grâce à cette évènement, on peut savoir à quelle position l'utilisateur à pressé la vue. Ensuite pour réccupérer l'évènement lorsque l'utilisateur continu d'appuyer sur l'écran, il nous suffit d'utiliser *touchesMoved*. Et pour finir lorsque l'utilisateur retire son doigt de la vue, nous utilisons la méthode *touchesEnded* qui nous permettra par exemple de savoir où l'utilisateur à retiré son doigt de la vue. A l'aide de ces trois methodes, nous avons implémenté les différents gestes que l'ulisateur peut effectuer pour créer une carte. Pour dessiner la carte, nous avons dû surcharger la methode - *(void)draw :(CGContextRef)context*, celle-ci permet de modifier le comportement de l'affichage de la vue et notamme elle donne la possibilité de dessiner n'import quel objet dans la vue.

Ensuite, nous avons créer une vue permettant de changer d'objet que l'on veut mettre sur la carte. Cette dernière se trouve à gauche de la carte. Dans cette vue, nous avons des *UIButton* permettant de créer des boutons, mais la partie la plus important de la vue c'est la liste déroulante pour choisir l'objet que l'on placer sur la carte. Cette liste déroulante est une vue que nous avons dû implémenter, la particularité de cette liste c'est qu'elle est générique ce qui permet de l'utiliser dans plusieurs vues différentes et notamment dans les menus pour afficher le choix des cartes. Grâce à celle-ci l'utilisateur peut changer d'objet selectionné en exerçant un simple mouvement vertical du doigt.

Enfin la dernière vue que nous avons réalisé, est une vue contenant plusieurs *UIButton* mais aussi un *UISegmentedControl* qui permet de basculer du mode affichage complet (dessiner tous les éléments de la carte) de la carte, au mode d'affichage partiel (qui affiche que le sol et cache les blocs de la carte) facilitant la modification du sol malgrès les blocs.

Le controlleur

Le controlleur est une partie très important de l'architecture MVC, elle permet de faire le lien en les interactions que l'utilisateur effectue dans l'éditeur de carte et les données relative à celui-ci. Nous avons donc créé des objets de type *UIViewControler*, ces objets nous ont permis de réaliser cette liason. Grâce à celle-ci, le modèle et la vue sont complètement indépend ce qui permet de pouvoir changer totalement de vue sans avoir besoin de modifier le modèle. Comme nous avons divisé l'éditeur de carte en plusieurs sous vues, nous avons donc créé un « pseudo »controlleur (car se ne sont pas des *UIViewControler* mais seulement des objets normaux) pour chaqu'une entre elles, elle même possède un controlleur global qui lui sera bien un *UIViewControler*. Lorsqu'un utilisateur va réaliser une action dans la vue de l'éditeur de carte, cette dernière va appeler une methode de son pseudo controleur, qui lui appelera une methode du controlleur global, qui se chargera d'appeler la bonne methode du modèle en fonction de l'action effectuée par l'utilisateur.

5.1.3 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[5] 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[6]. , de l'intelligence artificielle[7]. et des divers sons[8]. 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 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.

5. Editeur de carte « voir section 5.1.2, page 43. »
6. Moteur physique « voir section 5.1.3, page 48. »
7. IA « voir section 5.1.3, page 51. »
8. Sons « voir section 5.1.3, page 52. »

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 :

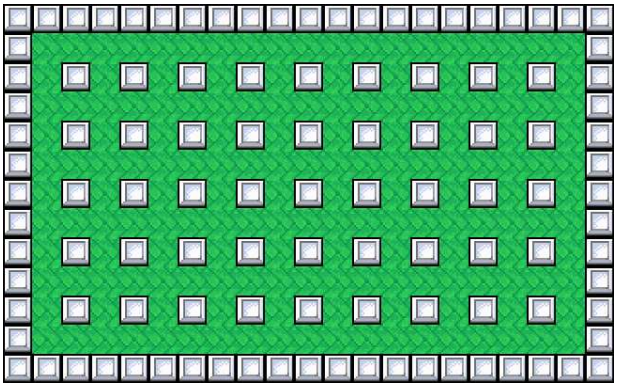


FIGURE 5.1 – L'image représentant la totalité des objets non animés

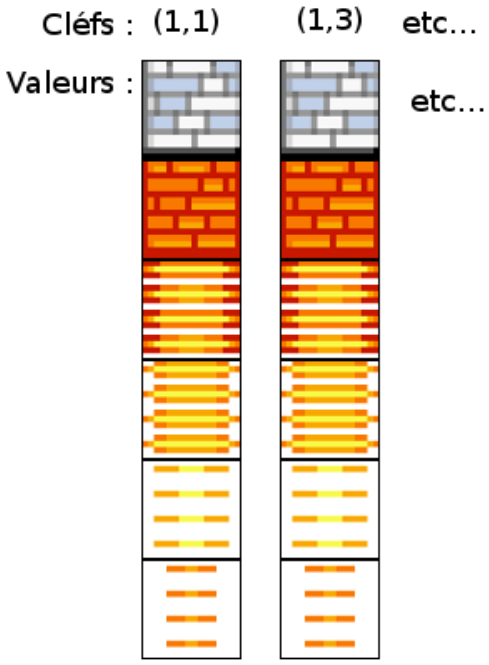


FIGURE 5.2 – La hashmap des objets animés

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	229

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 [9] 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 [10] 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 :

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

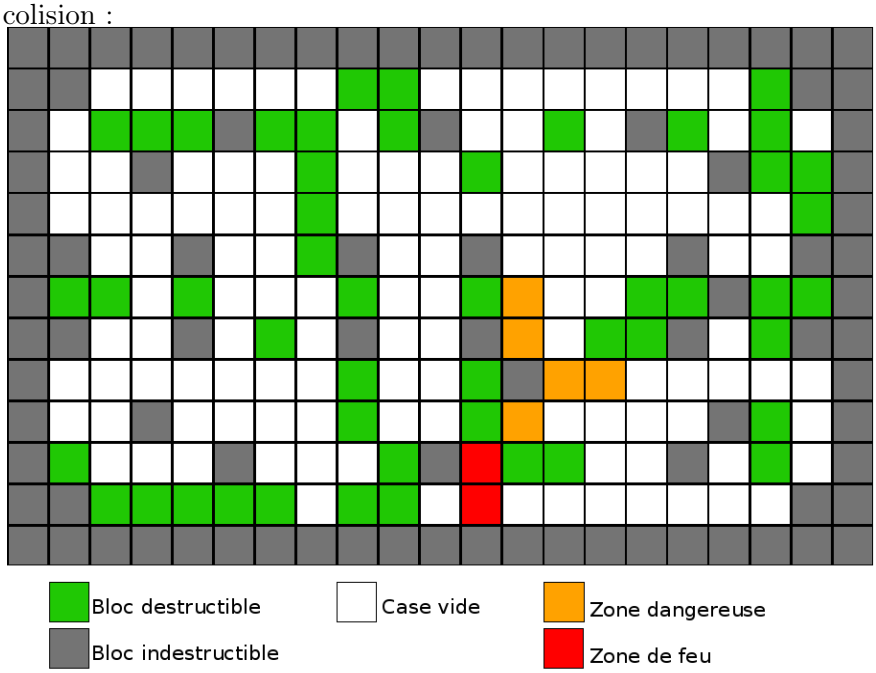
De cette façon il est rapide de savoir si un joueur rentre en collision avec un objet quelconque ou si celui-ci subit des dommages.

Exemple

– Carte originale :

9. Moteur de rendu « voir section 5.1.3, page 45. »
10. Editeur de carte « voir section 5.1.2, page 43. »

– Carte de colision :



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.

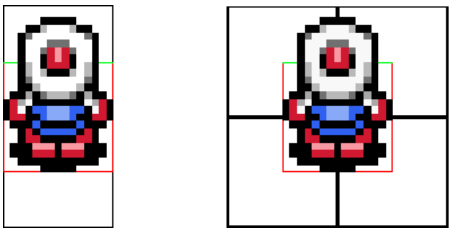
Quant à la gestion des colisions bien que chaque équipe utilise la matrice décrite ci dessus, les façons de concevoir la chose ont divergé proposant au final deux méthodes de rendu. Dans chacune le joueur se deplace verticalement, horizontalement ainsi qu'en diagonales.

1. Colisions sous Android

   Le principe de collisions 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



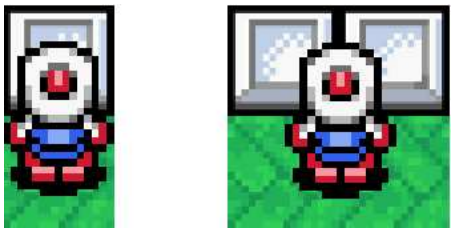
   Le trait vert représente le côté que nous étudierons lors d'un déplacement vers le haut.

   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. Collisions sous iOS

   Le principe des collisions sous IOS a été réflechie de maniere a ce que les déplacements soit fluides et que les collisions avec des blocs ne freinnent pas la fluidité du jeu. Car en effet lors des collisions, si le joueur doit passer entre deux blocs, celui-ci doit passer au pixel pret. Cela implique que l'utilisateur doit être très minutieu dans ses déplacements et cela rend le jeu casiment injouable lors de passage entre deux blocs. Donc pour remédier à ce problème nous avons décidé d'établir un système de marge sous IOS . C'est à dire que lors de la collision avec un bloc. On va vérifier si au moin un pixel du joueur touche au moins un pixel du bloc moins une marge. Voici le tout en image :

   Il faut donc que le rectangle rouge entourant le joueur ne touche aucun autre rectangle rouge entourant les blocs pour que le joueur puisse se déplacer. Les rectangles entourant les blocs sont réduit grâce à la marge (fleche jaune) pour permettre à l'utilisateur de passer facilement.

Gestion des bombes

– Threads

IA

Pathfinding

A*

Aléatoire

Prise de décision

Sons

Interface utilisaeur

Android

iOS

5.2 Serveur

5.2.1 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. Les requêtes font appel à la fonction post des servlets. Le flux entrant étant de type JSON, il faut déserialiser le flux dans un objet correspondant. Exemple l'utilisateur envoie son nom d'utilisateur ansi que 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 encapsulés dans un flux JSON, pour être envoyés 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 au nom d'utilisateur correspondant, le tout est ajoûté dans le tableau d'utilisateurs connectés. Cette userKey est ensuite nécessaire pour contacter les servlets suivantes. Si cet identifiant n'est pas envoyé ou n'est pas présent dans le tableau des utilisateurs connectés, il sera alors impossible d'accéder aux ressources du serveur.

5.2.2 BDD

La base de données du serveur n'est pas très complexe. En effet elle ne fait qu'accueillir les couples (nom d'utilisateur,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 de données. 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 6

Manuel d'utilisation

6.1 Menus

6.1.1 Premier lancement

Création compte local

Au premier lancement de l'application il vous est demandé de créer un compte un local. Ce dernier est nécessaire pour pouvoir utiliser l'application. Il vous suffira de renseigner dans le champs prévu à cet effet, votre pseudonyme 1).

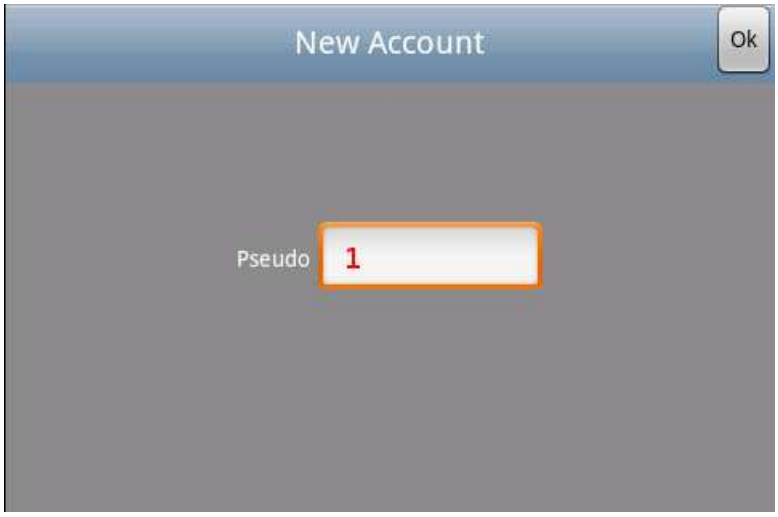


FIGURE 6.1 – Création compte local

6.1.2 Menu

Le menu d'accueil vous permet d'accéder à la section des parties locales **1**, des parties multijoueurs **2**, l'éditeur de carte **3** pour créer vos propres cartes de jeu, le menu d'options **4**, l'accès aux comptes locaux **5**, la création d'un nouveau compte local **6** et enfin l'aide **7**.

FIGURE 6.2 – Ajoût nouveau compte local

6.1.3 Jeu local 1

Paramétrage

Voilà un moment crucial précédent votre lancement de jeu, sa configuration. Rien de bien compliqué en soit, vous choisissez le type de partie **1**, la difficulté de vos adversaires(robots) **2**, le nombre d'ennemis sur la carte **3**, et le temps de jeu **4**, et enfin grâce à un défilement de la galerie **5**, la carte de jeu, et enfin vous validez **6**.

FIGURE 6.3 – Paramétrage partie locale

Jeu

Après avoir choisi vos préférences pour votre partie, celle-ci se lance. Pour vous déplacer votre joueur effleurez l'écran de jeu dans le sens et la direction vous souhaiter aller **1** . Sur le

panneau de droite en appuyant sur le bouton prévu **2**, vous posez des bombes. Le panneau du haut résume les informations de la partie. Le temps de jeu **3** restant, et vos différents bonus . Ces derniers correspondants à la portée de vos bombes **4**, le nombres de bombes que vous pouvez poser simultanément **5** , la vitesse de déplacement de votre joueur **6**, et votre compteur de vies **7**.

FIGURE 6.4 – Jeu en cour

Pause

Lors d'une partie vous pouvez la suspendre en appuyant sur le Menu **1** en haut à gauche. Il vous est possible de reprendre la partie **2**, accéder aux options de jeu **3**, redémarrer la partie **4**, ou tout simplement la quitter et revenir au menu **5**.

FIGURE 6.5 – Pause lors d'une partie

6.1.4 Jeu multi 2

Inscription Cette étape est inévitable afin d'accéder au mode multijoueur. Vous devez renseigner userName **1**, mot de passe **2**, et confirme ce dernier **3**. Vous pouvez choisir une connexion automatique **4** ou simplement ne pas ressaisir votre mot de passe aux connexions suivantes **5**. Dès que ces champs sont remplis, validez via la connexion au serveur **6**. Le userName est unique, s'il est déjà pris le serveur vous renverra une erreur, sinon vous accéderez à l'Accueil du mode multijoueur.

FIGURE 6.6 – Création compte multijoueur

Connexion Une fois seulement l'étape précédente accomplie vous devrez passer par le menu de connexion. Dans ce dernier il faut donner le couple userName **1**/mot de passe **2**, et vous pouvez aussi choisir, dans le cas d'une identification correcte, la mémorisation de votre complet **3** c'est à dire un accès direct sans passer par le menu de connexion, ou la mémorisation unique de votre mot de passe **4**. Enfin il est toujours possible de créer un nouveau compte multijoueur **5**. Là encore une vérification sur le serveur est faite lorsque vous choisirez la connexion **6**.

Accueil L'accueil du mode multijoueur est composé d'un champ éditable, permettant le tri des parties en lignes **1**, le rafraichissement des parties **2**, une liste déroulante clickable de celles en cour **3**, et enfin un accès au menu de création d'une nouvelle **4**.

Créer partie La création multijoueur et semblable à celle en locale. Vous choisissez le type de partie **1**, la difficulté de vos adversaires(robots) **2** en attendant les vrais joueurs , le nombre d'ennemis sur la carte **3**, et le temps de jeu **4**, et enfin grâce à un défilement de la galerie **5**, la carte de jeu, et finalement vous validez **6**.

FIGURE 6.7 – Connexion compte multijoueur

6.1.5 Editeur de map 3

Création de map

Comme dans tout jeux d'arcade qui se respecte, vous avez la possibilité de créer vos niveaux via un éditeur. Vous n'avez qu'à donner le nom de la carte **1** que vous souhaitez créer puis valider **2**. La encore les doublons ne sont pas possibles.

Edition de map

Une carte vierge s'offre à vous **1**. Le choix des structures(destructibles ou non) **2**, de la texture du sol **2'** mais aussi le positionnement des joueurs **3** vous est alors mis à disposition. Pour passer des structures aux texture du sol, utilisez l'interrupteur situé en haut à droite de l'écran **4**. Vous pourrez ensuite choisir l'image que vous souhaitez dessiner. Remarquez qu'en slidant de haut en bas, les images défiles ainsi, et inversement. Pointez sur la case que vous souhaitez dessiner avec la structure ou la texture pour dessiner une case. Il est aussi possible de rester appuyer sur l'écran tout en glissant, l'image choisit sera alors appliquée sur toutes les cases survolées.

Enregistrement map

Une fois votre carte achevée ou non, vous pouvez accéder au menu d'option via le bouton Menu **1** . Ce dernier vous propose alors de retourner sur l'éditeur **2**, d'enregistrer votre carte et quitter **3**, de remettre à zéro votre carte **4**, et enfin quitter sans enregistrer **5**.

Charger map

Après avoir enregistré une carte que vous ou un autre compte local aurait crée, il vous sera possible de l'éditer et de l'enregistrer et ce en la choisissant **1** parmi toutes les cartes crées et de valider **2**.

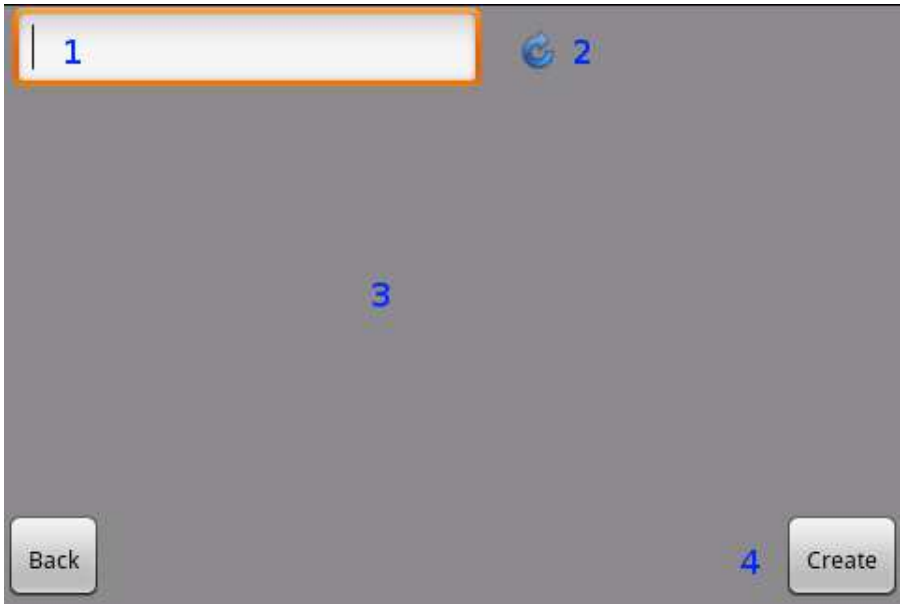


FIGURE 6.8 – Accueil des parties multijoueurs

6.1.6 Options 4

Le menu d'options est à votre disposition pour régler vos paramètre systèmes mais aussi profils d'utilisateur local et multiplayer.

Réglage sytème

Ce type de réglage a été conçu pour ajuster le son **1** à votre convenance mais aussi choisir la langue **2** utilisée dans votre jeu. Sont à votre disposition le français mais aussi l'anglais.

Gestionnaire profil

Dans ce menu vous avez accès à l'édition de vos profils locaux et multijoueurs. Une fois vos modifications accomplies n'oubliez pas de les valider en cliquant sur le bouton Ok situé en haut à droite de votre fenêtre.

Local

Il vous est ici possible de changer le pseudonyme **1** de votre compte local en cour, mais aussi de choisir la couleur de votre personnage **2** , toujours pour les parties locales.

Multijoueur

Une fois un compte multijoueur crée sur le serveur, cet onglet vous donnera la possiblité d'éditer **1** userName et password mais aussi de changer de compte multijoueur **2**. Les paramètres d'accès comme la connexion automatique **3** ou la sauvegarde du mot de passe **4** sont ici configurables dès lors que le couple userName/password sera renseigné et valide.

Général

Grace au menu déroulant **1** vous pouvez fixer le côté de l'écran où sera positionné le menu

FIGURE 6.9 – Création partie multijoueur

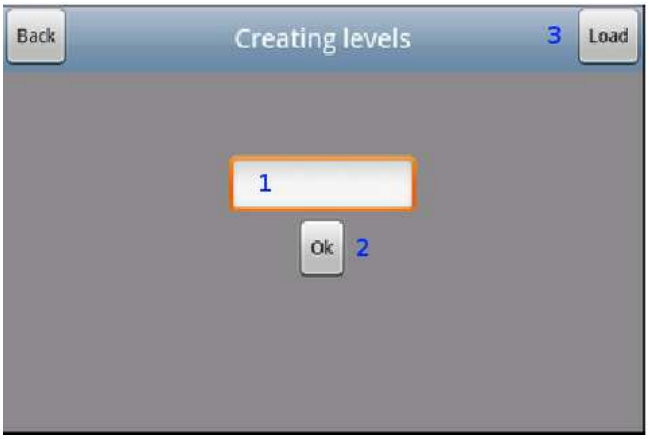


FIGURE 6.10 – Création de map

de jeu, c'est à dire celui contenant le bouton de posage de bombes pendant une partie, suivant si l'on est gaucher ou droitier.

Choisir compte local 5

Afin de permettre l'utilisation de l'application par plusieurs joueurs et donc comptes. Il vous est possible de les sélectionner grâce à une liste des comptes locaux.

Ajouter compte local 6

Il vous est demandé comme au premier lancement, de renseigner un pseudonyme 1. Dans ce cas là une vérification que votre pseudonyme n'existe pas déjà serait faite. Une fois valide vous revenez sur le menu d'accueil avec votre nouveau compte local sélectionné.

TODO ludo

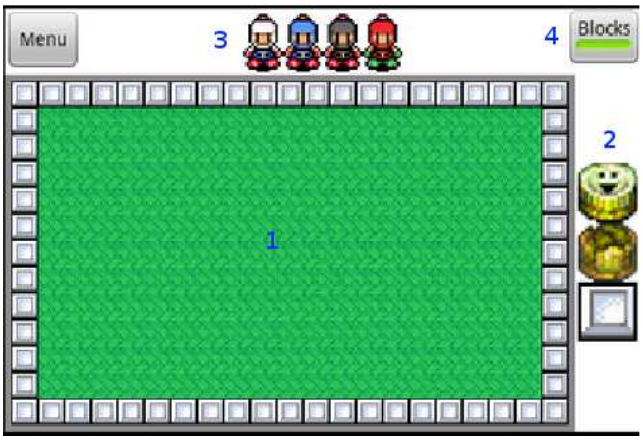


FIGURE 6.11 – Edition de map

FIGURE 6.12 – Option éditeur

6.2 Jeu

6.3 Editeur

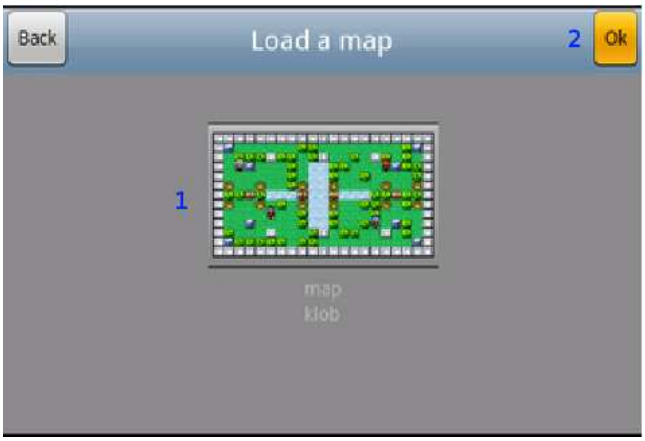


FIGURE 6.13 – Charger une map préexistante

FIGURE 6.14 – Charger une map préexistante

FIGURE 6.15 – Gestion profil local

FIGURE 6.16 – Gestion profil multijoueur

FIGURE 6.17 – Général

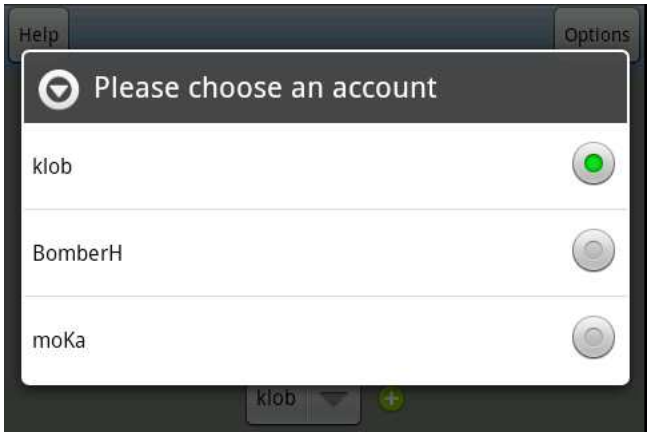


FIGURE 6.18 – Choix compte local

FIGURE 6.19 – Ajoût nouveau compte local

Chapitre 7

Réutilisabilité

7.1 Généralités

Il nous a paru important lors de la réalisation de ce projet de mettre au point un code facilement améliorable et surtout réutilisable par le spersonnes qui souhaiteraient personnaliser le jeu à leur façon.

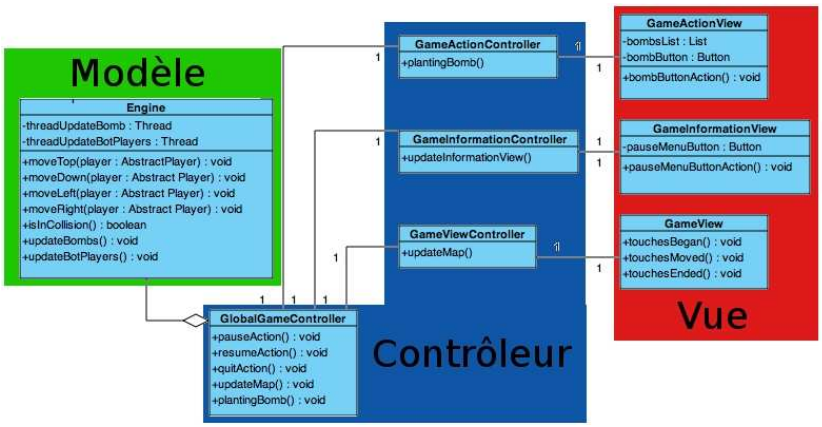
C'est pourquoi un des points primordial à été tout d'abord de rédiger notre code ainsi que notre documentation totallement en Anglais afin qu'une majorité de personne puisse les comprendre.

7.2 Client

7.2.1 Modèles de conception

MVC

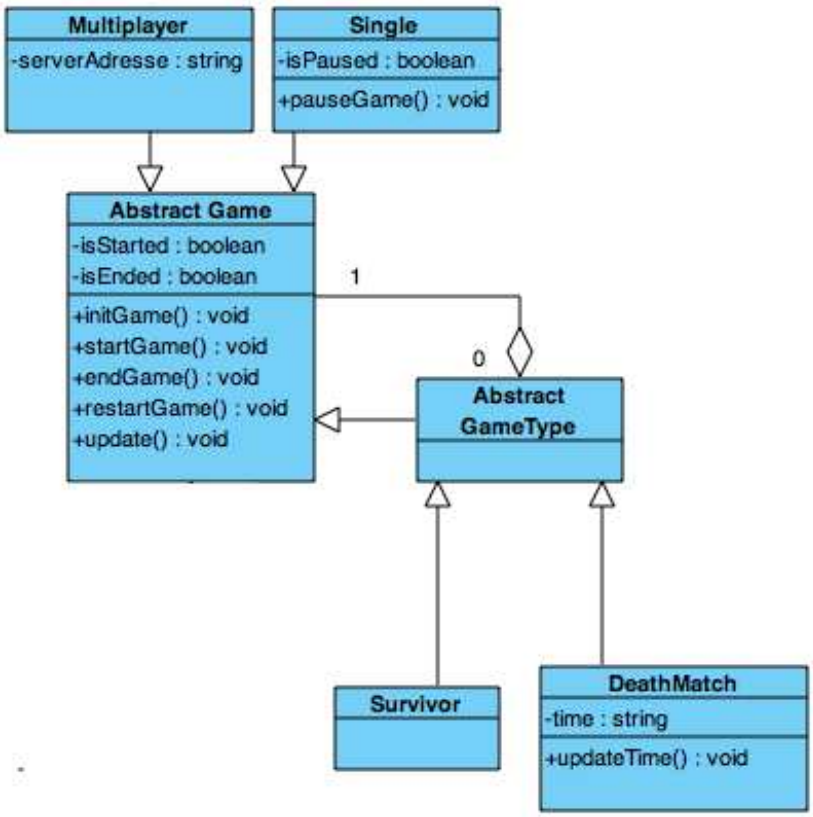
Le Modèle-Vue-Contrôleur est une architecture et une méthode de conception qui organise l'interface homme-machine (IHM) d'une application logicielle. Pour cela il divise l'IHM en un modèle (modèle de données), une vue (présentation, interface utilisateur) et un contrôleur (gestion des événements, synchronisation), chacun ayant un rôle précis dans l'interface.



L'avantage de cette méthode de conception est qu'elle permet la modification d'une des parties sans affecter les autres. Il serait dans notre cas possible de pouvoir modifier le gameplay,

l'interface graphique ou encore d'améliorer le moteur du jeu sans pour cela devoir s'occuper du reste du code.

Design Pattern Décorateur



7.2.2 XML

Le fait d'avoir utilisé le XML permet de pouvoir modifier les diverses caractéristiques de nos ressources sans necessiter la moidre modification du code source.

Internationalisation

Dans notre jeu nous avons prit en charge le support de l'internationalisation. Pour cela nous nous avons dédié à chaque langue un fichier XML contenant la globalité des mots apparaissants dans notre application. Si l'utilisateur souhaite en rajouter une nouvelle langue, il lui suffira d'ajouter un fichier XML dans le répertoire correspondant à la langue voulu et d'y traduire tous les mots présents dans les XML des langues déjà existants.

Personnalisation

Là aussi l'utilisation du XML montre ses avantages lors de la personnalisation des diverses ressources mises à disponibilité de l'utilisateur.

Prenons par exemple le code XML d'un objet indestructible :

```
<objects>
 <undestructible name="grass1" hit="0" level="0" fireWall="0" damages="0">
  <animation name="idle" canLoop="false" sound="">
   <framerect top="270" left="120" bottom="300" right="150" delayNextFrame="0" />
  </animation>
 </undestructible>
</objects>
```

Si l'on souhaite associer un son à cet objet il suffit d'éditer l'attribut *sound* et de rajouter le nom de celui que l'on souhaite. Il est en de même pour les différentes caractéristiques de notre objet.

7.3 Serveur

7.3.1 Servlets

Chapitre 8

Discussion

8.1 Difficultés

Lors du développement de l'application, nous avons rencontré de nombreuse difficultés.

Tout d'abord la principale difficulté était le fait que nous devions développer une application mobile, ce qui nous limitait au niveau des ressources disponible. De plus nous avions jamais développé une application sous Android et iOS.

8.1.1 Android

La difficulte que nous avons eu durant le développement de l'application sous Android était d'implémenter le multi-touch pour que le joueur puisse à la fois bouger et poser une bombe.

8.1.2 iOS

Pour iOS, la principale difficulté a été d'apprendre l'Objective-C car c'est un langage qui était totalement nouveau, malgrès le fait qu'il est relativement similaire au Java. De plus, ce dernier ne posséde pas de garbage collector pour iOS, ce qui nous a obligé de gérer la mémoire manuellement, ce que nous avions jamais fait.

8.1.3 Serveur

Enfin, pour le serveur, nous avons eu des problèmes los du déploiment local du serveur d'application et aussi des servlets car nous avions jamais réalisé ces derniers. La communication entre le serveur d'application est la base de données, nous a aussi posé des problèmes.

8.2 Problèmes

8.2.1 Mobile

Le problème majeur a été de tester notre appliction. Tout d'abord au début du projet, nous ne possédions pas de téléphone sous android. Ensuite au niveau d'iOS, nous avons pas pu tester notre application sur un iPhone car il faut obligatoirement posséder un compte au centre de développement d'Apple (ce dernier coût 99$ par an), ce que nous pouvions pas nous payer.

L'autre problème était de savoir si nous allons utiliser OPENGL-ES pour le développement de l'application. Ce qui nous aurait permi d'avoir un code portable donc il aurait été compatible pour Android et iOS. Grâce à cette portabilité, cela nous aurait permi de développer qu'un code source et donc ça nous aurait fait gagné du temps en revanche il aurait encore fallu apprendre un nouveau langage.

8.2.2 Serveur

Au niveau du serveur, nous avons eu un problème de pour déployer ce dernier sur internet.

8.3 Améliorations

Le développement de ce type de jeu permet d'ajouter un grand nombres d'améliorations. Notamment la gestion des bonus et des malus qui rendent le jeu beaucoup plus amusant et qui est disponible en annexe. Il y a aussi la gestion de différents types de parties, comme : la capture de drapeau, les parties en équipe, etc. Nous aurions aussi pu ajouter un mode histoire pour les plus gourmands et pour rendre le jeu plus attractif. Puis une des améliorations les plus importantes, est l'ajout du mode multijoueur en WI-FI au niveau national (puis international ?) car ce dernier n'existe pas encore sur mobile et aurait très bien pu révolutionner le jeu bomberman sur mobile.

8.3.1 Serveur

- Pooling
- Session

Chapitre 9

Conclusion

L'application que nous avons réalisée est fonctionnelle et respecte les contraintes de fonctionnalités minimales imposées. Malheureusement nous n'avons pas eu le temps de developper le mode mutlijoueur qui aurait été très enrichissant. Mais nous avons quand même implémenté la partie réseau qui nous permettra dans un futur proche de pouvoir continuer à developper ce dernier dans le but de partager notre application sur l'AppleStore et l'AndroidMarket.

Toutes les fonctionnalités implémentées fonctionnent parfaitement malgres que durant le développement et les tests que nous avons réalisé, nous avons décelé quelques bogues minimes qui ont été corrigés au jour d'aujourd'hui.

Les langages que nous avons utilisés nous ont permis d'étoffer notre maitrise de la programmation objet et de découvrir le developpement mobile. L'analyse du projet quand à elle s'est avéré suffisante et efficace car l'application fonctionne et que l'implémentation d'une multitude de fonctionnalités est envisageable.

Nous avons du développer cette application à quatre. Cela a été un handicap car nous disposions d'un temps limité et que le développement de deux applications comme celle-ci demande une vraie équipe de projet composée de plusieurs membres. Heureusement, gâce à notre organisation, à la bonne entente et la bonne communication au sein du groupe, nous avons pu travailler de manière efficace. Mais cela nous a aussi permis de partager nous connaissances et de nous entraider tout au long de ce projet.