



TER FMIN200

—

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

BONVILA Olivier

COUSEIN Kilian
TARDIEU Benjamin

PITOT Ludovic

Nous tenons à remercier

Table des matières

1	Introduction	4
2	Présentation	5
2.1	Projet	5
2.2	Plate-formes de développement	5
2.2.1	Android	5
2.2.2	iOS	6
2.3	Organisation	7
3	Analyse	9
3.1	Cahier des charges	9
3.1.1	Menus	9
3.1.2	Jeu	9
3.1.3	Serveur	10
3.2	Modélisation	10
3.2.1	Général	10
3.2.2	Menus	10
3.2.3	Editeur de carte	13
3.2.4	Jeu	14
3.2.5	Réseau	15
3.3	Différences entre Android et iOS	17
4	Développement	18
4.1	Mobile	18
4.1.1	Menus	18
4.1.2	Editeur de carte	21
4.1.3	Jeu	21
4.2	Serveur	29
4.2.1	JSON	29
4.2.2	Servlet	30
4.2.3	BDD	33

5	Manuel d'utilisation	34
5.1	Menus	34
5.2	Jeu	34
5.3	Editeur	34
6	Discussion	35
6.1	Problèmes	35
6.1.1	Android	35
6.1.2	iOS	35
6.2	Améliorations	35
6.2.1	Jeu	35
6.2.2	Serveur	35
7	Conclusion	36

Chapitre 1

Introduction

Chapitre 2

Présentation

- Nous + Tuteur

2.1 Projet

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

2.2 Plate-formes de développement

2.2.1 Android

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

Disponible via une licence Apache version 2¹, Android est fondé sur un noyau Linux, il comporte une interface spécifique, développée en Java, les programmes sont exécutés via un interpréteur JIT, toutefois il est possible de passer outre cette interface en programmant ses applications en C ou C++, mais le travail de portabilité en sera plus important.

Il nous est aussi permis 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 maximum de périphériques.

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

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

Le kit de développement d'Android comporte les outils de base pour développer une application en Java, le développement en C et C++ nécessitant d'utiliser le NDK². Il se compose des bibliothèques principales d'Android, de divers exemples d'applications et surtout d'un émulateur de terminal Android qui permet de pouvoir tester directement ses applications directement sur Linux, Mac ou Windows sans nécessairement avoir besoin d'un terminal mis à part pour OPENGL-ES 2 qui n'est pas encore pris en charge par celui-ci, sinon le kit de développement permet très bien de tester ses applications en temps réel sur tout périphériques tournant sur Android que ce soit en liaison directe avec un ordinateur ou en récupérant l'archive de l'application au format APK.

Le format APK ou Android Package est une variante du format JAR³, est utilisé pour la distribution et l'installation de composants regroupés sur le système d'exploitation Android.

2.2.2 iOS

Apple est une entreprise multinationale américaine qui a été créée le 1^{er} 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 répandus de l'entreprise se trouvent les ordinateurs Macintosh (1984), l'iPod⁴ (2001), l'iPhone⁵ (2007) et l'iPad⁶ (2010). Apple a développé deux systèmes d'exploitation (Mac OS X⁷ et iOS) permettant de contrôler ses différents produits, notamment les ordinateurs Macintosh, l'iPhone et l'iPod touch⁸.

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 TV⁹ (2007). C'est dérivé de Mac OS X dont il partage les fondations. Comme celui-ci, iOS comporte quatre couches d'abstraction : une couche « Core OS »¹⁰, une couche « Core Services »¹¹, une couche « Media » et

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

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

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

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

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

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

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

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

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

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

pour finir une couche « Cocoa »¹². Pour développer 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 Smalltalk¹³ et la rapidité du C. C'est une extension du C ANSI¹⁴ qui ne permet pas l'héritage multiple contrairement au C++¹⁵. Aujourd'hui, il est principalement utilisé pour développer des applications sur Mac OS X et iPhone. Une nouvelle version Objective-C 2.0 a été introduite avec Mac OS X 10.5 (Léopard) en octobre 2007.

Le kit de développement 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 on peut analyser un programme pour surveiller l'état de la mémoire, l'utilisation du réseau, du CPU, etc. Pour utiliser tous ces outils, il est obligatoire de posséder un ordinateur Macintosh sous Mac OS X et de s'enregistrer sur iPhone Dev Center pour pouvoir télécharger et installer le SDK d'iOS.

2.3 Organisation

Afin de garder notre projet cohérent et par sécurité nous avons choisi de le stocker sur les serveurs de *Google Code* qui mettent gratuitement à disposition des gestionnaires de versions (Subversion ou SVN en abrégé) distribués sous licence Apache¹⁶ et BSD¹⁷.

Les gestionnaires de versions comme leur nom l'indique, permettent d'avoir à portée 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 prix avoir modifié un fichier à partir de la dernière version de celui-ci. Dernier avantage d'avoir

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

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

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

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

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

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

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

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

Chapitre 3

Analyse

– Introduction

3.1 Cahier des charges

3.1.1 Menus

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

Nous avons pour ce faire établi un diagramme d’activité reflétant les différents parcours possibles par un utilisateur lors de sa navigation dans l’application (voir section modélisation p12).

3.1.2 Jeu

Le jeu est la partie la plus importante du projet. Il se décompose 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 hiérarchie de classe permettant de représenter l’ensemble des objets du jeu. La vue quant à elle est composée d’objets graphiques simples (Bouttons, images, ...) et d’une partie représentant 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 français et l’anglais. Elle doit permettre à l’utilisateur de jouer à des parties solitaires ou multijoueurs. Ce dernier possèdera un compte hors ligne et en ligne. Le premier permettra 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 préférences de connexion sur une base de donnée locale, mais aussi les scores du joueurs (nombre de parties gagnées ou perdus). Le jeu devra permettre à l’utilisateur

de pouvoir créer différents comptes hors lignes en cas de partage de téléphone avec un ami ou un membre de famille, pour pouvoir garder en mémoire ses scores et ses préférences. Le compte en ligne quand à lui servira seulement à établir une connexion avec le serveur distant pour pouvoir jouer en multijoueur. Un menu d'aide doit apparaître 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 dernières pourront être seulement utilisées en mode solitaire. Pour les parties solitaires une intelligence 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 manière 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'hébergeur pour les parties et il se chargera de faire s'interagir les joueurs, via leur mobile, entre eux. Nous parlons donc ici des parties multijoueurs.

Il devra être capable d'enregistrer des inscriptions de nouveaux joueurs, avec vérification qu'il n'y ait pas de doublons. Ces derniers seront inscrits dans la base de données du serveur. Les joueurs devraient ainsi pouvoir se connecter en utilisant le couple username/mot de passe, préalablement choisi. Suite à cela les utilisateurs seront à même de lister les parties en cours, ils pourront choisir de créer des parties ou de les rejoindre. Voilà concernant les fonctionnalités qui ont été demandées pour la partie serveur.

3.2 Modélisation

3.2.1 Général

- Langage utilisé
- Tout en anglais
- Modèle Vue Contrôleur
- Documentation - (Javadoc / appledoc)

3.2.2 Menus

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

Vous avez tout d'abord la section de création de parties locales. Vous aurez accès à une liste de cartes ainsi qu'au réglage de difficulté des bots, leur nombre et le temps de

jeu. Le type de partie sera une fonctionnalité à venir. Vous n'aurez plus qu'à créer la partie configurée.

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

Suite à ces deux sections vient ensuite l'éditeur de cartes. C'est depuis ce menu que vous déciderez la création d'une nouvelle map de jeu local ou à l'édition d'une d'entre elles. Choisissez 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 systèmes telles que le volume ou la langue de l'application(anglais, français). Une sous-section de gestionnaire de profil est aussi présente. Une édition de vos comptes locaux, multijoueurs ou même vos paramètres de jeu comme la position du menu, sont modifiable depuis ce menu à onglets.

Base de données

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

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

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

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

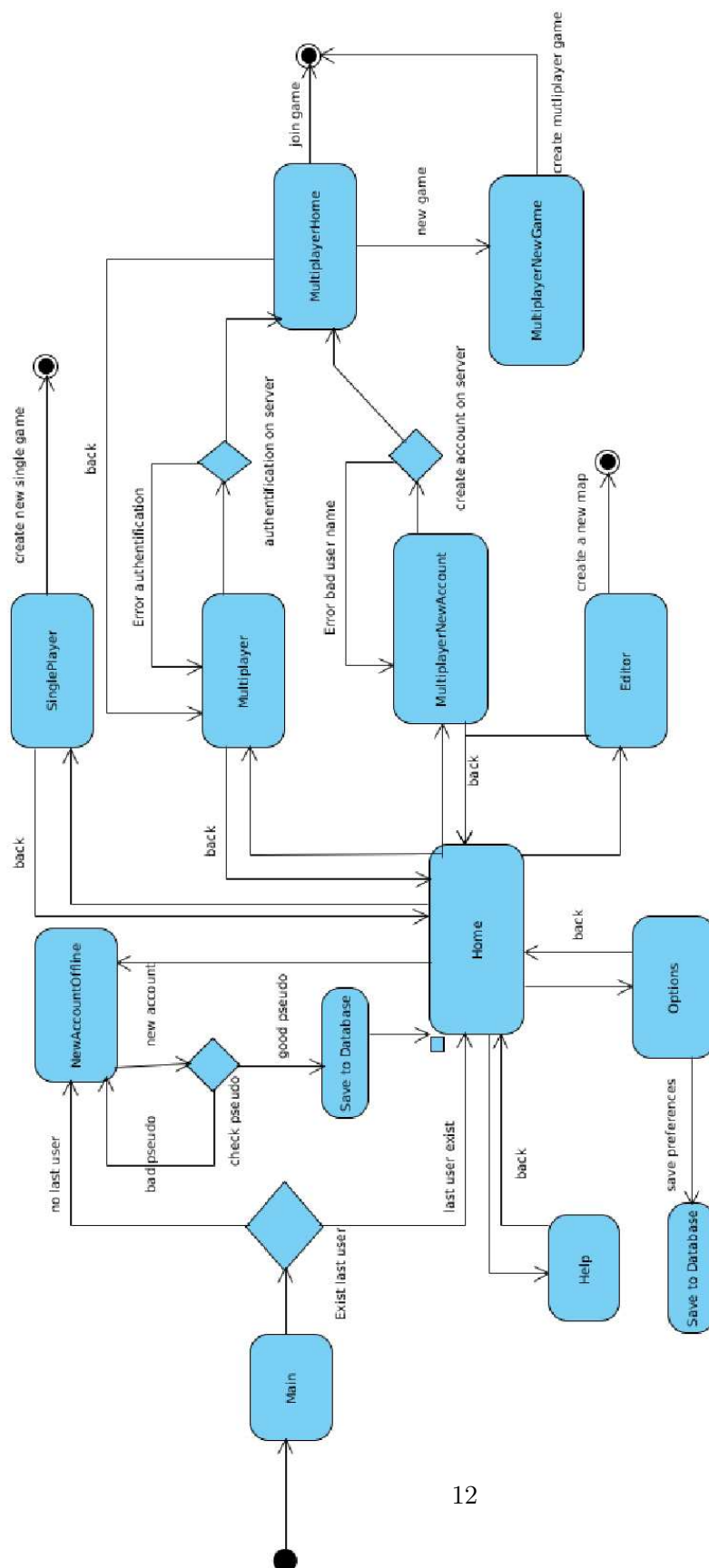


FIGURE 3.1 – Diagramme d'activité

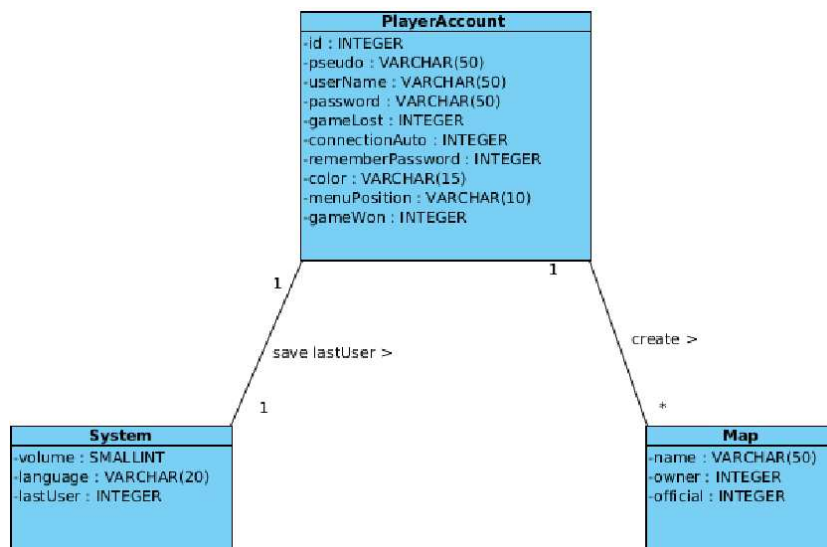


FIGURE 3.2 – Diagramme de classe Base de données

Scénarios

3.2.3 Editeur de carte

L'éditeur de carte est une fonctionnalité qui va permettre à un utilisateur de créer facilement ses propres cartes pour ensuite y jouer dessus contre l'intelligence 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 précédemment 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.

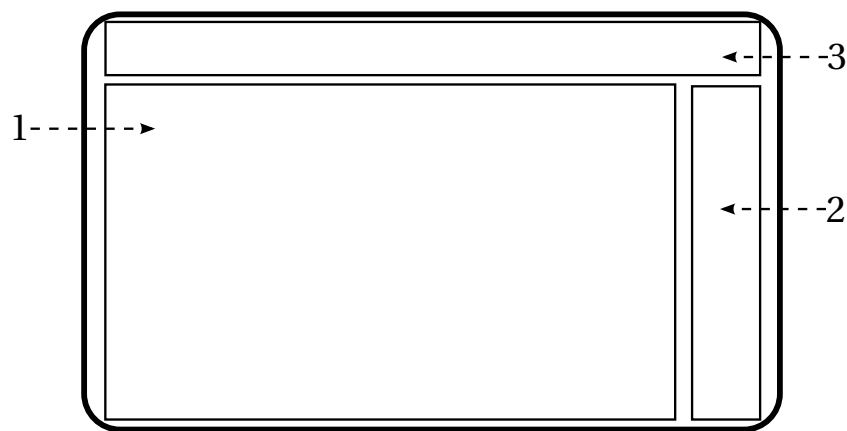
Pour réaliser cette partie de l'application, nous avons utilisé le modèle de conception MVC pour diviser le code de l'éditeur de carte. Grâce à cette décomposition, le code est plus lisible et plus facile à réutiliser.

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 représenter sous la forme de deux matrices, la première représentant les objets du premier niveau (le sol) et la deuxième 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 smartphone¹. Après plusieurs prototypes d'interface, nous avons décidé 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 contrôler 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'appuyer sur la carte pour placer un bloc dessus.



Controleur

Pour finir le controleur aura 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 global possédant les controleurs de chaque vue.

3.2.4 Jeu

- Diagramme classe
- Gameplay
- Gestion images / tile mapping
- Images
- Sons

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

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.

Les six éléments situés sur la partie haute du schéma ci-dessous(respectivement Inscription, Connection, GamesList, CreateGame, ConnectionGame et ManageGame), représente les différents tâches demandées. Elles sont reliées à une classe nommée contexte, 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 3.1).

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 cours(ConnectionGame), ou en créer une 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.

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 a été pensée comme demandé pour l'enregistrement de comptes. Une unique table nommée Users remplit donc cette fonction. Le serveur devra pouvoir y accéder en écriture(inscription) comme en lecture(connexion).

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

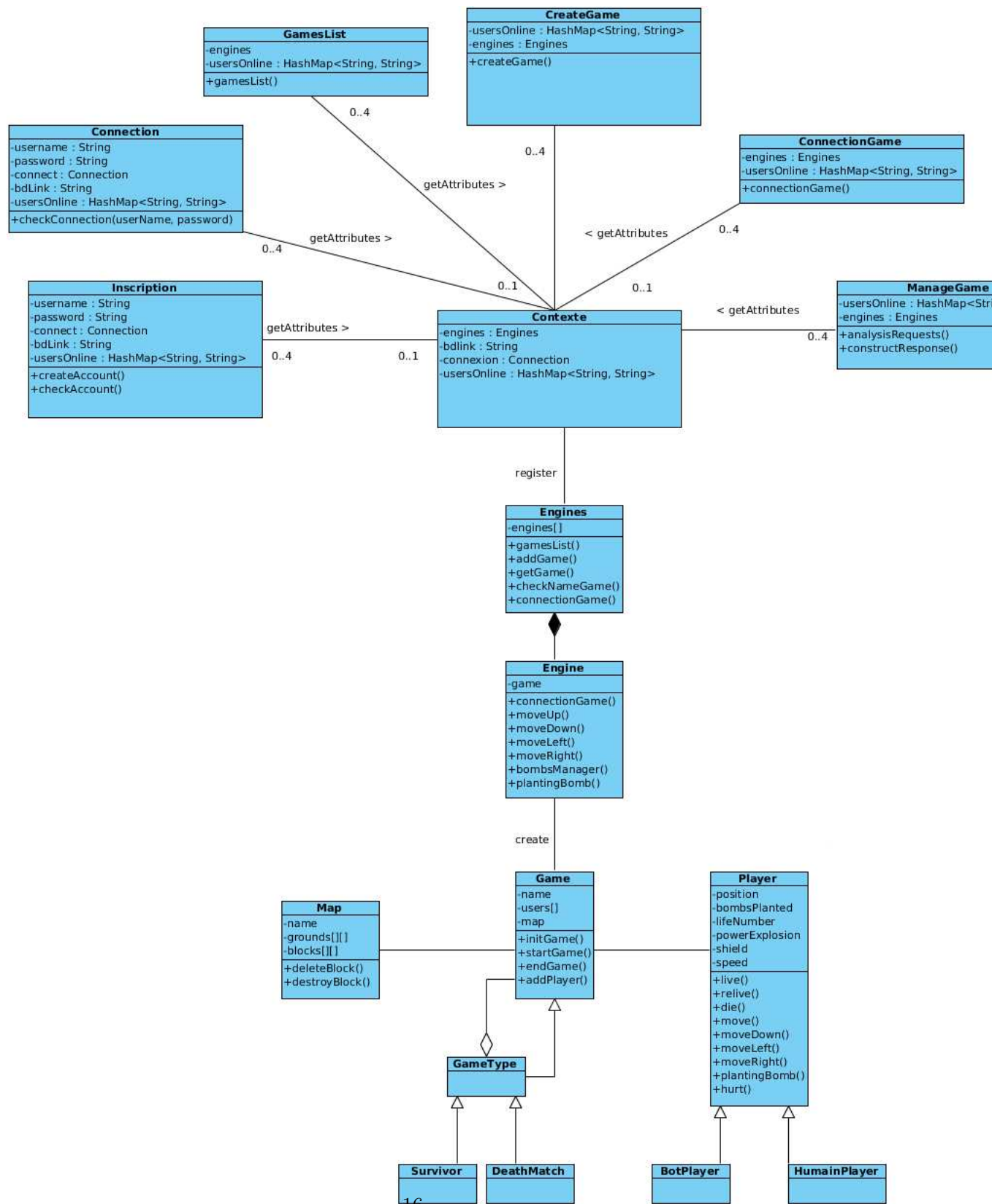


FIGURE 3.3 – Serveur

3.3 Différences entre Android et iOS

Chapitre 4

Développement

4.1 Mobile

4.1.1 Menus

API et widget

Android

Afin de créer les différents menus de notre application, Android met à la disposition des développeurs une API¹ très bien fournie. Parmi celle-ci le package Widget nous a été très utile.

Grâce à ce dernier de nombreux objets ont été utilisés 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 syntaxe bien particulière. Cette méthode est vraisemblablement préférable, du moins lorsque l'interface graphique est figée, connue à l'avance. Exemple :

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

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

Exemple :

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

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

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

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

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

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

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

Voici un exemple représentant la cohabitation listener Activity :

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

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

iOS

BDD

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

Android

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

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

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

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

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

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

    ContentValues entree = new ContentValues();

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

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

iOS

Première utilisation

Création utilisateur

Gestion utilisateur

Gestion des préférences système

Création de carte (charger)

Création partie solo (tout)

Création partie multi (officielle)

4.1.2 Editeur de carte

Rendu

Interface utilisateur

Sauvegarde

4.1.3 Jeu

Moteurs

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

Moteur de rendu

Contrairement à celui que nous avons vu dans la section précédente pour l'éditeur de carte² 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 physique³. , de l'intelligence artificielle⁴. et des divers sons⁵. 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 physique³ c'est à dire par exemple qu'ici un bloc indestructible sera présent

2. Editeur de carte « voir section 4.1.2, page 21. »

3. Moteur physique « voir section 4.1.3, page 24. »

4. IA « voir section 4.1.3, page 28. »

5. Sons « voir section 4.1.3, page 28. »

tout au long de la partie et ne pourra pas être supprimé, il n'est donc plus nécessaire 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 nécessaire à chaque rafraichissement de regarder si un bloc existe dessus, ce test se fait en début de partie et sera valide jusqu'à la fin de celle-ci.

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

Si l'on avait eu un sol animé représentant de l'eau, il aurait été composé de plusieurs images et aurait donc nécessité 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 cartes² 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 clés 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 mémoriser le sol se trouvant au dessous.

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

Voici un exemple concret de la methode décrite au dessus :

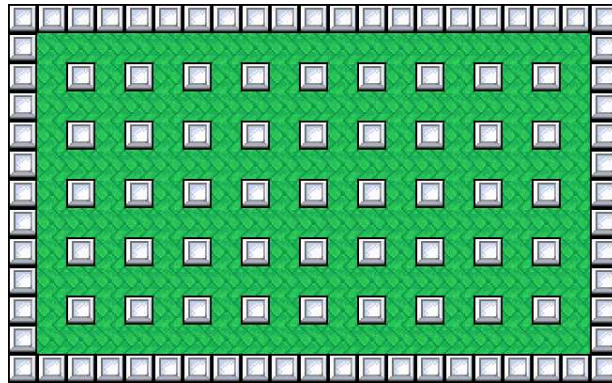


FIGURE 4.1 – L'image représentant la totalité des objets non animés

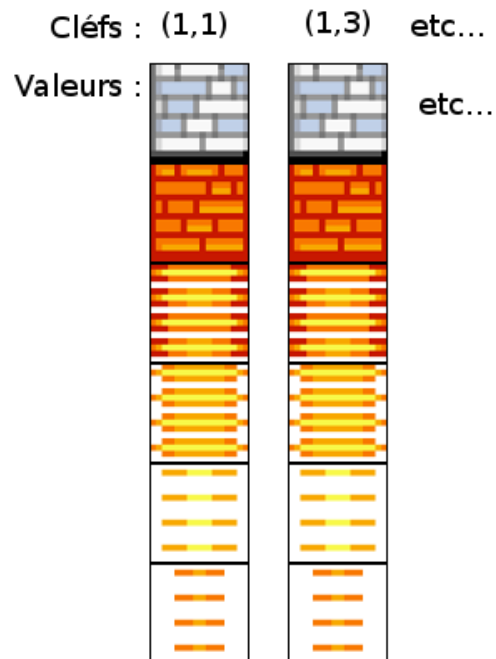


FIGURE 4.2 – La hashmap des objets animés

Les avantages d'avoir utilisé une telle structure est qu'ici au lieu de parcourir les $21 \times 13 \times 2$ 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 énormément plus si il existe d'autres objets, nous n'affichons qu'une image plus au maximum 197 objets.

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

Le meilleur des cas ici décrit une carte vide donc composée que de sol non animé ainsi 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 indépendamment cela équivalait à 337 objets.

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

Nous voyons très clairement les différences de coûts entre les deux méthodes 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 collisions qu'auront les joueurs avec l'environnement les entourant ainsi que les interactions qu'auront les divers objets du décors entre eux ainsi qu'avec les joueurs.

Tout comme le moteur de rendu⁶ le moteur physique d'un jeu doit être optimal dans les traitements qu'il a à effectuer vu que son utilisation est permanente au cours du jeu.

Afin d'optimiser ces traitements lors des collisions nous avons fusionné les deux matrices présentes dans l'éditeur de cartes⁷ afin de n'en obtenir qu'une seule reprenant le principe décrit dans le moteur de rendu².

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

6. Moteur de rendu « voir section 4.1.3, page 21. »

7. Editeur de carte « voir section 4.1.2, page 21. »

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

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

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

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

Deplacements

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

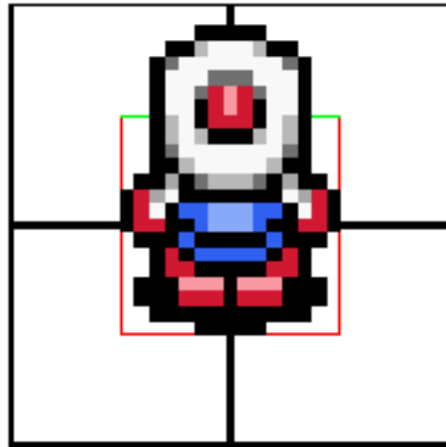
Quant à la gestion des colisions bien que chaque équipe utilise la matrice décrite ci dessus, les façons de concevoir la chose ont divergé proposant au final deux méthodes de rendu. Dans chacune le joueur se deplace verticalement, horizontalement ainsi qu'en diagonales.

1. Colisions sous Android

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

Pour cela nous allons étudier un mouvement qui sera celui vers le haut afin de mieux comprendre ce terme. Nous aurions pu prendre n'importe quel autre 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 entièrement traversable.

Il existe donc quatre types de collisions possibles :

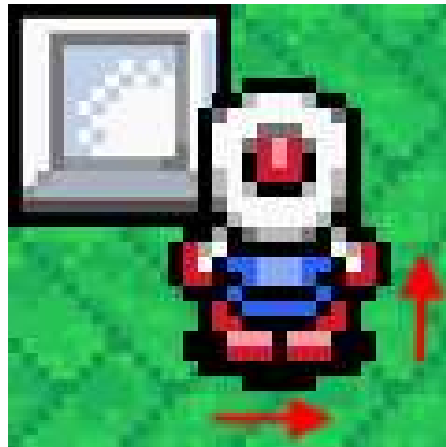
- (a) Le premier cas correspond à celui où les cases sont traversables, il n'y a donc qu'à déplacer le joueur vers le haut.



- (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 embêtant pour l'utilisateur de devoir se décaler 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 et l'opposé du précédent et marche de la même manière.



2. Colisions sous iOS

Gestion des bombes

- Threads

IA

Pathfinding

A*

Aléatoire

Prise de décision

Sons

Interface utilisateur

Android

iOS

4.2 Serveur

Nous avons choisi de créer notre serveur sur une base de servlet. Ce fût ici aussi un point nouveau pour nous, réitérant 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.

4.2.1 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 JSON⁸. 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 reconnaît la philosophie des services web exposant une interface d'échange : il s'agit d'envoyer et de recevoir des informations dans un format facilement manipulable par le protocole de transport HTTP.

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

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

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

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

```
ServletConnexion
Player => Serveur
{"username","password"}
```

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

```
ServletGameList
Player => Serveur
{"userKey"}
```

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

```

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"}

```

4.2.2 Servlet

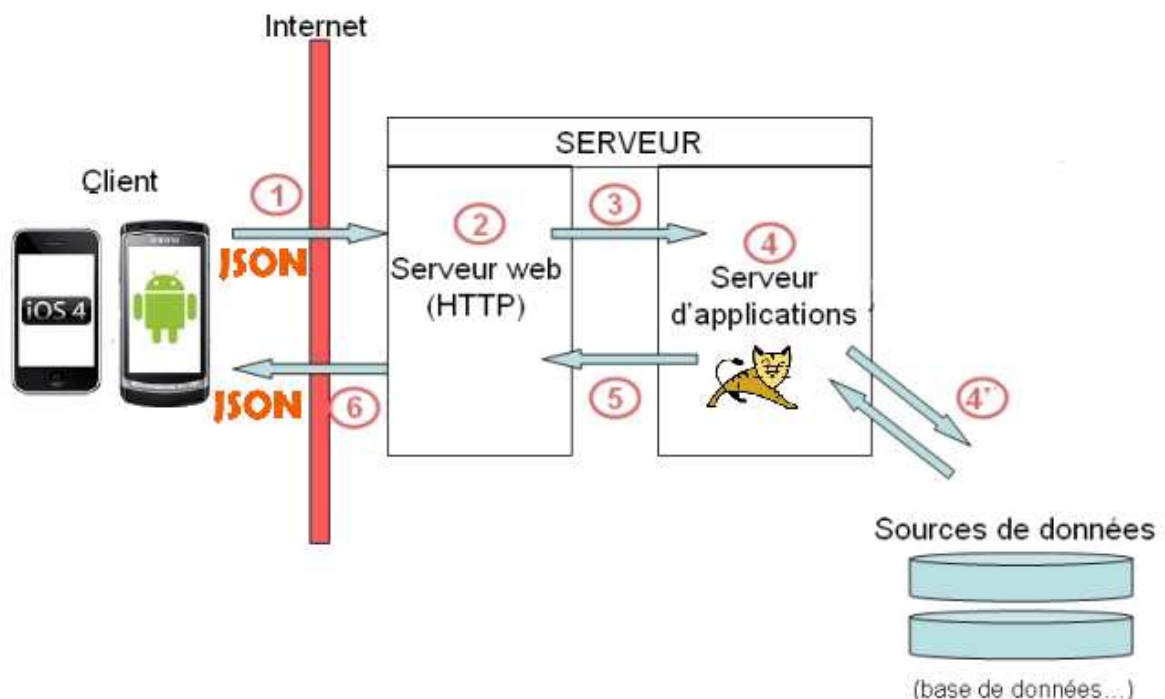
Comme il a été dit précédemment, notre serveur est accessible via des requêtes HTTP contactant des servlets. Ces servlets sont stockées dans un serveur d'application nommé Apache Tomcat. Il s'agit d'un conteneur libre de servlets Java 2 Enterprise Edition, mais il fait aussi office de serveur Web.

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.

Lors du développement nous avons eu simplement à remplacer les classes (diagramme partie Modélisation) représentant les tâches par des classes Java de type Servlet. Quant à la classe Context elle est devenue ContextListener. Cette dernière est invoquée lorsque l'objet ServletContext est créé. 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.

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 exécute 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).

En pratique

Les requêtes font appel à la fonction post des servlet. Le flux entrant étant de type JSON, il faut désérialiser le flux dans un objet correspondant. Exemple l'utilisateur envoie son userName et son mot de passe crypté dans un tableau, sérialisé en JSON, pour pouvoir récupérer les informations nous procédons comme suit :

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

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

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

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

    ...}
```

La sécurité

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

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

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

4.2.3 BDD

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

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

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

Chapitre 5

Manuel d'utilisation

5.1 Menus

TODO ludo

5.2 Jeu

5.3 Editeur

Chapitre 6

Discussion

6.1 Problèmes

6.1.1 Android

6.1.2 iOS

6.2 Améliorations

6.2.1 Jeu

6.2.2 Serveur

- Pooling
- Session

Chapitre 7

Conclusion