Project Report

## Mobile Computing - 2013/14

## 

## Achievements

The following table is a list of features and its implementation status.

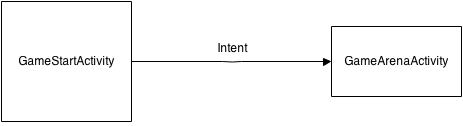
|  |  |
| --- | --- |
| **Feature** | **Implemented (Fully / Partially / Not implemented)?** |
| Game scene | **Fully** |
| Movement and life cycle of players | **Fully** |
| Bomb drop off and explosion | **Fully** |
| Movement and life cycle of robots | **Fully** |
| Collision detection | **Fully** |
| Score and game duration | **Fully** |
| Pausing / resuming the game | **Partially** |
| Handling of relevant activity lifecycle events (e.g., pressing home button) | **Partially** |
| Level selection | **Fully** |
| Multiplayer support | **Fully** |
| Clients leaving / joining the game | **Fully** |
| Server hand-over | **Not implemented** |
| Group merging (or group splitting) | **Not implemented** |

The pausing / resuming features, as well as handling relevant activity lifecycle events, has been partially implemented. The code handles gracefully a home button press action, but it is unable to get back to the game when the user restores the app (the board surface stays black).

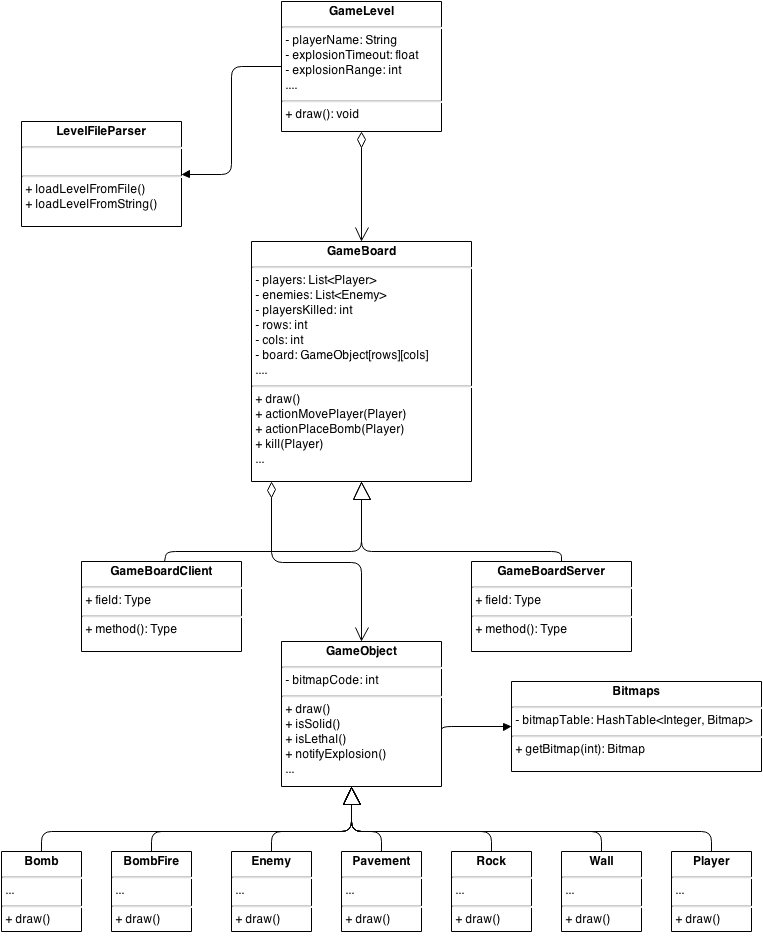
## 2. Specification

From an Android point of view, the game consists of 2 activities: *GameStartActivity* and *GameArenaActivity*. Unsurprisingly, the starting activity is *GameStartActivity*. This activity is responsible for showing the main user interface, where the player gets to choose to act either as a host or as a client. In either case, it is required that a wi-fi direct group has been previously created using the WDSim Console application. Also, the player is given a chance to pick a username of his choice, and if it will be hosting a new game, a menu to choose a level.

Essentially, *GameStartActivity*, is an initial screen interface that allows the player to set up the game environment. After retrieving the necessary information (player name, level, etc.), *GameArenaActivity* is started. The information from the previous activity is passed on to *GameArenaActivity* via intents:



The game implementation, living in *pt.cmov.bomberman.model*, is derived from the following UML model:



## 3. Design

A decentralized multiplayer architecture is used. The communication between players is carried out via Wi-fi Direct (simulated), where one of the nodes becomes the group owner. The group owner acts as a central server, and it is the only node that can update the game board. It is worth noting that regular client players can’t just update their local game board according to the user actions, because another network player may be doing some other action concurrently that conflicts with this one (for example, 2 players moving to the same position).

Therefore, to keep the system in a consistent state, every time a client wants to perform an update on the board, it sends a request to the server, and returns immediately. Eventually, the server will process this request, and if it is ok to proceed, a board update message is broadcasted to every known player. Upon receiving such a message, a client player updates its board accordingly. If the request cannot be granted, then the server just ignores it, and no more messages are exchanged.

This approach has the convenient and desirable property of allowing clients to return as fast as possible; there is no blocking I/O with the server waiting for a decision on a specific request. In effect, clients act as a simple replica of the server’s state, they just accept the server’s orders, hoping that these are consistent and correct. This yields a simple yet efficient synchronization protocol – it would be hard to reduce the network traffic used.

As mentioned before, the information gathered from *GameStartActivity* is passed to *GameArenaActivity* through an *Intent*. *GameArenaActivity* will then take the appropriate actions to start the main game loop and instantiate the right board type. There are two board types: *GameBoardClient,* and *GameBoardServer*. The former acts more like a board stub – as mentioned earlier, all the client does is transforming user actions into a request to the server. The latter is used by players hosting a game, turning each user action into messages being broadcasted to every other player, to replicate the board status. Network communication is encapsulated in *pt.cmov.bomberman.net.[client|server]*. The classes in each package invoke the necessary actions on the existing board. In the case of a client, for example, a dedicated thread runs listening for incoming messages from the server, parses each message, and depending on its meaning, invokes the corresponding method on the generic board interface (which turns into a polymorphic call to a method on *GameBoardClient*). In the case of a server, it works the other way around: incoming requests are equally forwarded to the game board interface, but this time, the polymorphic call resolves to a *GameBoardServer*. The *GameBoardServer* implements the whole game logic, controls concurrent access to the board, and performs the necessary checks to make sure that the request is indeed valid. If it is valid, the action is executed on the server’s board, and replicated to every player through the network, otherwise, it is discarded. Since clients do not block waiting for server replies on a specific request, a discarded invalid request is just like it had never happened in the first place.

Whether a player is acting as a client or as a server, there always exist 2 threads:

* *GameArenaActivity*: mainly responsible for user interaction. This thread captures the user’s actions, such as buttons pressed, and invokes the appropriate methods to carry it out. The methods invoked mostly belong to the game board implementation;
* Main game loop (*MainThread.java*): this thread is minimalist, it merely draws the current board state into the screen surface. This is done by calling *draw()* on *GameLevel*, which in turn calls *draw()* on the current board. The game board is the only component with enough knowledge about the current state of the game; its *draw()* implementation loops through every position in the board – a 2D array of *GameObject* – and invokes draw() on the *GameObject* stored in that slot, if there is one. This final step turns out to be a polymorphic call: each object knows how to draw itself. In particular, each object is represented by a corresponding bitmap image, which can be loaded and retrieved using the utility class *Bitmaps.java*. The purpose of this utility class is to save memory and processing time by managing bitmaps allocation and storage, guaranteeing that the same bitmap is only loaded once, no matter how many objects use it.

If running as a client, *ClientThread.java* creates another thread to listen for and dispatch incoming messages from the server. If running as a server, there are two more threads:

* The main server thread accepting new connection requests (*ServerThread.java*);
* One thread per connected client, to handle requests coming from that client (*CommunicationThread.java*).

When acting as a server, a new connection is only accepted if there is still room for another player: the map layout assigns a predefined location to each player that can possibly join the game; if the number of players joined reaches the number of predefined locations for players, new connection attempts are denied.

## 4. Implementation Choices

The app has been targeted to run under WDSim. As a consequence, it has never been tested on real devices, but we believe that the Wi-fi Direct simulator provides a reasonably good testing ground. Devices must run at least version 19 of the Android SDK.

The dependencies to run the project are the WDSim library files, as well as the WDSim Console.

Since the levels description is static and stored in a text file in a similar fashion to that of the example given in the project assignment, the size of the map for each level is known *a priori*. This means that the size of the map is the same regardless of the device’s screen size. Thus, a map must be appropriately framed in the case that the maximum aspect ratio still leaves out a lot of empty space in the screen. There are two options to consider here: either the whole map is shown, and borders are drawn around the map field to fill the empty space, or a specific portion of the map is showed, moving the camera around as the player moves within the game scene.

For simplicity, the first option was our choice. Drawing borders to frame the map is way simpler than messing with the camera and the view, it is less error prone, and in fact players can even benefit from this approach, since they can visualize the whole map at any moment, giving an opportunity for a better strategy overall.

## 5. Conclusions

Overall, it has been a good experience, although we feel that a lot of development work that had to be done is absolutely unrelated to mobile and ubiquitous computing. This is mainly because game development is quite different from traditional apps development. This is a universally accepted fact, and that’s why there are dedicated courses to game design and development. Our game design skills were rusty enough to the point of making us wonder how to start, and what to do next. Since no one in the group is especially skilled and experienced in game design, this was a huge turn down and a bottleneck in the development process – after all, the last time we had to deal with game development was 2 years ago during the Computer Graphics course.

The net effect is that the majority of time was spent dealing with game implementation issues, rather than with Mobile Computing issues. Maybe worse than that, the huge amount of time spent on this component left very little spare time to think and design the networking protocols, which we believe were the killer feature in this project.

In that sense, perhaps a better choice, and definitely more effective, would have been to provide a project assignment where students are not required to have specific, detailed knowledge of a development niche market like game design, or, alternatively, provide the students with a solid, existent game implementation and ask them to extend it to support online multiplayers using Wi-Fi Direct groups. This would ensure that students don’t spend a blatant amount of time focusing on issues that are not interesting for the course.

As a consequence, we really wish that we had more time to work on the game. The ambitious deadline date made it extremely hard to deliver a fully functional game with networking features. Nevertheless, it has been a great challenge, and the final product, even if slightly unfinished, can offer some amusing game experiences.