Universitat de les Illes Balears

Robostrike: analysis and implementation of a multiplayer game with current technologies

MSc Candidate

Juan Rechach Piza

A MSc thesis submitted to *Departament de Ciències Matemàtiques i Informàtica* of the University of Balearic Islands in accordance with the requirements for the degree of
Màster Universitari Enginyeria Informàtica (MINF)

Author
Juan Rechach Piza

MSc Supervisor
Isaac Lera Castro

MINF Director
Antonia Mas Pichaco

28/05/2019

Robostrike: analysis and implementation of a multiplayer game with current technologies

Juan Rechach Piza
Tutor: Isaac Lera Castro
Treball de fi de Màster Universitari Enginyeria Informàtica (MINF)
Universitat de les Illes Balears
07122 Palma de Mallorca
<juanrechach@gmailcom>

Abstract

Robostrike, the game developed in this project, is currently built in Flash, which is an obsolete technology that will be retired in December 2020. This technology makes the date of death of the game approaches and the way in which the game is developed it makes not to be possible to play it from a smartphone. To solve these problems, the work of this project to build the game from scratch with modern technologies and adapting it to smartphones with a native application.

Knowing the history, the current problems, the requirements of the game and the future work after the project, an analysis and study have been carried out to plant the stones in a solid way and to choose the best-suited technology for each situation. Backend side architectures have also been studied due to the game also has a backend side. Several methodologies, techniques, good practices and principles has been followed due to the quality of code and of the project is important too.

Finally, a Continuous Integration / Continuous Deployment pipeline has been set up and backend services have been automatically deployed to hosting with Docker containers.

Keywords: smartphone game, microservice architecture, gateway ppattern, continuous integration, continuous delivery

1 Introduction

Smartphones were a small great revolution which turned out to be an indispensable tool for many people around the world. Over the years, smartphones have been improved very fast by offering better specifications and power. At the same time, video games for smartphones have also evolved being more powerful and with better graphics. But this improvement has no effect on the quality of video games, but only in amount, which has increased.

Just like smartphones, technologies are changing and improving very fast. Frequently, technologies are updated either to fix bugs or to add new features adapted to the new needs of the world, to make life easier for developers or to be able to build better products in an easier way and with higher quality. This means that if a company wants to be always up to date, it is forced to update the technologies of all its projects very frequently. Actually, much more often than desired. The result is not another that the companies end up using outdated technologies. This is due to two reasons mainly. First, because companies tend to think that if something works it does not have to be touched. Updates can break the code and the products and companies want to lose no money and give a bad image to customers. And second, because companies do not want to invest money in updating the technologies because they think that they do not receive any value in return. Companies hardly prevent future problems. In almost any technology, from time to time important failures are discovered, which are quickly fixed in future versions. If the technology used is up to date, then it's really easy to be updated, otherwise, it makes it an almost impossible mission.

A similar problem occurs with architectures too. The architecture is a key and an essential element for the success of a product. It determines the quality and longevity of the full system or product and it has a vital impact on the performance, security, interoperability, reliability, availability and scalability. A wrong solution can lead a low client confidence, performance or security issues. It usually happens that the scope of a project changes and grows as development proceeds by making the architecture obsolete. This way, architecture needs to be revised from time to time. If it is not reviewed, the difficulty of the project increases and it becomes hard to maintain and to understand. Even further, it is needed an exaggerated time to do non-valuable tasks and to refactor code, reaching a point where it stops being attractive for workers. A wrong wrong architecture can lead to the failure of a product or to a lack of quality. At first, when a product suffers the first changes in the scope, companies usually think that the architecture does not need to be reviewed, but slowly and over the years they realize that they have a technical debt.[1]

Lack of quality in projects is also a typical problem and it is related to the previous two points. Usually, projects do not include tests, the documentation is poor or non-existent,

spaghetti code is abundant due to grow without control and without foresight. This makes it difficult to fix bugs, to change the behavior of a product or to update the technologies and at the same time, it causes the quality to go down. Companies also think that writing unit tests or documentation is a waste of time where they do not earn money or produce value for their products. This helps not to discover bugs in development time, but in production, which needs to be fixed quickly in future versions and the quality of code gets worse. This also causes a technical debt.

2 About the game

2.1 History

The game developed in this project is already an existing game called Robostrike. It was built in Flash as a hobby among a few friends from France in 2001. The next year, in May 2002, the game was released and has been online since then. During the first years of life, the game had a great reception and was very successful with an active community from several countries. Since its release, the game has gone through different improvements, like the graphics, team game mode, better solo games, share personal boards and so on.

At that time, Flash Macromedia Reader was a small technological revolution. It opened many doors to the web in terms of new graphics, sound and animation. It also introduced new opportunities for developers. However, this technology that at the time was a great opportunity, at the end it was one of the main reasons of the death of the game: the death of Flash meant the death of the game. Flash began to die when the first iPhones were released. Apple and Steve Jobs did not like Flash[25] and it was not allowed on the company's IOS hardware products. Another reason was the search engines like Google. Robostrike's success was entirely based on the interest of the game itself and its online community, not on its optimization for search engines. However, in 2005 Google progressively relegated Robostrike and other full flash websites to the hidden alleys of the web. This is due to the results are based on a complex algorithm that works very effectively on text-mainly websites but is ineffective with flash code. These reasons, added to the fact that creators could not dedicate to the game, caused the number of players of the game started to decrease over time. Nowadays, just a few players play the game.

2.2 Game concept

Robostrike is a strategy and action multiplayer game where robots fight in boards in real time controlled by players. Each robot is controlled by one player who decides the actions of these. Games are played on boards formed by squares, which can have robots, walls, holes, power-ups, pushers, rotators and sliders. The game can be played in several modes. The first mode is a deathmatch the last survivor wins the game. The second mode is a race. The first player to capture all the marks or the last survivor wins the game. The third and last is a solo mode where players have to find out how to pass each level in as few moves as possible (indicated by a PAR for each level). This mode contains IA robots.

Figure 1: Example in the current game.

The mechanics of a game is as follows:

- Each player controls a robot.
- All players decide and program 6 actions of their robots at the same time.
- Then, when all players have completed programming, all robots execute their 6 actions simultaneously.
- When robots end the execution of their actions, players program again the actions as in the point 2.
- This process is repeated until the game ends when one of the conditions of completion is met depending on the game mode.

Boards, where games are played, are composed of different elements which interact with robots.

Besides, players can create their own boards to play and share with the community.

2.3 A unique game

Robostrike is a rich game in strategies, in the diversity of skills required, in game concepts, in the types of power-ups of robots and in the number of existing boards, which is almost infinite. This makes it a unique game and very different from the most successful games because it is neither a casual game, nor a pay to win game. Players cannot pay to beat other players, to have more chances or to get a better score. Players must think about the strategy that they have to follow to beat other players.

3 Motivation goals

The development of this project is a motivation from several points. First, this project gives me the opportunity to build a from scratch a unique game that deserves a new opportunity in the current markets with new and powerful technologies.

Second, I like to investigate new technologies, play and learn them. Third and last, I like to learn better ways to do things, trying to do the best I can.

This project gives me the opportunity to learn from the mistakes companies do and do things better and to spend time to learn modern and powerful technologies that I would not do learn at work. Thus, a special emphasis has been placed in quality, the design of the architecture and the choice of most correct technologies for each situation.

The goal of this project is to rebuild the game to adapt it to today's world, with today's technologies and best practices.

The goals of this project are next:

- Rebuild an existing game to adapt it to smartphones (Android iOS -future versions).
- Give the game a new chance.
- Build a well-built prototype to continue the game.
- Deploy the backend side to a hosting.
- Study technologies to learn and decide which is the most appropriate for each situation.
- Learn in detail new and modern technologies.
- Deepen the knowledge of technologies already learned in the last years.
- Learn in detail how to structure the architecture of a system.
- Learn and use the Test Driven Development methodology.
- Automate the build, the test of code and the deployment.

4 Methodology

To ensure the quality of the project a set of methodologies, techniques, good practices and principles has been followed. In this section they are briefly explained and why they have been chosen.

As was previously explained, current technologies are rapidly changing. This means that if the technologies want to be updated easily, a development methodology that allows making these changes have to be used. The development methodology that which is better suited to changes is the Agile Software Development, which aims to focus on short cycles of development. The short cycles allow not to build all the parts the project at the same time, being able to group the development in any desired way. In this manner, the development of each component has been divided into several cycles, whose number is variant according to how big it is. A few methodologies are included within the Agile Software Development, such as Scrum. In this project, the Kanban board has been used, which is a work and workflow visualization tool that enables you to optimize the flow of your work. It has been used to group, manage and organize the tasks and keep track the status of tasks.

To ensure the quality either of the code and the product a software development technique and a practice development technique have been followed and some tools have been used. In one place, Test Driven Development (TDD) is found. The idea of TDD is that tests are written before the code. In the pure TDD cycle, first is written a single failing test, then enough code to pass the test. Then a second single failing test is written, enough new code to pass both tests and finally, code is cleaned up. This process is done until the task is made entirely. In another place is the Continuous Integration (CI)[28], which is the process of automating the build, testing the code and the deploy of every time a team member commits changes to version control. Among the benefits of use, one of the key benefits of integrating regularly is that errors can be detected quickly and locate them more easily. As each change introduced is typically small, pinpointing the specific change that introduced a defect can be done quickly. Another huge benefit is that code is built and deployed automatically if all the tests are green after a commit in the specified git branch. This can be easily integrated with docker. Other benefits are that tests run in the real world or the code coverage is increased.)[2])[24]

Among the tools used, one of them is the linter[27], which is a tool that analyzes source code to flag programming errors, bugs, stylistic errors, and suspicious constructs. In this case, a linter has been used both for TypeScript and JavaScript.

5 Requirements

In this section, the project requirements are presented and grouped in game requirements and technology and architecture requirements.

In this section, the project requirements are presented and grouped in game requirements and technology and architecture requirements.

a) Game requirements

- A full functional prototype must be developed. The game must be playable by multiple players at the same time.
- The game must be played from smartphones with Android.
- The deathmatch game mode must be able to be played.
- Multiple players must be able to play the same game.
- A fair game without cheating players.

b) Technology and architecture requirements

- All the repositories must contain unit tests and the coverage of the code must be higher than 90
- The game must be easily scalable. The architecture must be created in a way that the game must be able to scale easily in the number of players (both registered and connected) and in the number of games played at the same time. This scalability must not be expensive to achieve both at the level of resources and of code.
- The smartphone application must be native.
- Codebase must be single both for Android (currently) and iOS (future).

- Use the most similar technologies/programming languages in all the parts of the project.
- All the technologies used must open source.
- All the technologies used must be modern and the last stable version must be used.
- Use of the best possible practices and methodologies to ensure quality.
- Data such as boards, games and players must be stored and requested form services and a database.

c) Other requirements

- A fair game without cheating players.

6 Technologies

To select the technologies and the tools used, many of them have been analyzed and studied to find the most appropriate technologies for each situation. In this section, the technologies used in the project and the reasons behind their choice are briefly explained.

6.1 Node.js

Node.js[19] is a lean, fast, cross-platform JavaScript runtime environment that is useful for both servers and desktop applications built on Google V8 JavaScript engine. Node.js uses an event-driven, a single-threaded non-blocking event loop, and a low-level I/O API. Node.js allows using JavaScript on the server side as if it were a browser. This way and in some applications, the same code can be shared between the server and the client. Node.js works especially well for real time applications such as chats, games. It also comes with a package ecosystem behind called NPM, which is the world's largest software registry with more than half a million packages of free and reusable code. NPM can manage packages that are local dependencies of a particular project, as well as globally installed JavaScript tools. When used as a dependency manager for a local project, NPM can install in one command all the dependencies of a project through the package.json file.

.Net Core has been studied with Node.js. .NET Core is a free, open-source, Cross-platform and last generation web framework to build modern applications like websites or REST APIs. Along with the latest tools built like Azure Cloud and the programming language C, it allows to build and deploy REST APIs in a very simple way. As well as Node.js, .Net also contains a package manager called NuGet which provide the ability to produce and consume packages in the projects.

However, Node.js and JavaScript / TypeScript have been decided to be used instead of .Net Core and C due to the only need to use one web framework and one language in the full project. This way, both APIs, game server and the game can be developed using Node.js and JavaScript by making the developing of all the platforms easier.

6.2 JavaScript

JavaScript is a lightweight, interpreted, object-oriented language with first-class functions. It is best known as the scripting language for Web pages, but thanks to Node.js its use has become popular for the non-browser environments as well. It is a prototype-based, multi-paradigm scripting language that is dynamic, and supports object-oriented, imperative, and functional programming styles. JavaScript is the native language in the Node.js platform.

The main reason behind its use (although it ends up using a superset that compiles to JavaScript) is due to the full application can be built using one single programming language. From the native smartphone application to the backend APIS or the game server can be programmed with the same language. This makes the development of the application easier. This language makes sense with TypeScript.

6.3 TypeScript

TypeScript[26] is a typed superset of JavaScript that compiles to plain JavaScript developed by Microsoft, which provides optional types, classes, interfaces and modules to JavaScript. It adds readability to the code and maintainability to large-scale Javascript applications.

The choice of TypeScript over JavaScript is due to the value added. First, thanks to typings the readability of the code improves and this is a very important factor because the code is read much more often than it is written. Typings also helps to the maintainability due to the self-documentation code. Second, Javascript code is valid TypeScript, getting any TypeScript project to be able to use any JavaScript package. This means that no compatibility with the Javascript ecosystem is lost and any package developed by the community can be used. In addition to focusing on static typing, TypeScript provides great tooling and language services for autocompletion and code navigation.

An alternative to TypeScript is Flow[9], a static type checker for JavaScript developed by Facebook. Flows uses data flow analysis and infers types and tracks data as it moves through the code. There is no need to fully annotate the code before Flow can start to find bugs. However, the choice comes for what TypeScript provides and not Flow: better readability to code, autocompletion and code navigation.

6.4 Express

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It contains a large amount of HTTP utility methods and middleware to build quickly and easily a robust API. Many popular frameworks are based and built on Express. Ease of use, integration with Socket.io and TypeScript and prevent further layers and dependencies have been key in its choice.

6.5 Mocha/Chai

Mocha[17] is a popular testing framework on Node.js. Chai[4] is an assertion library that provides both the Behavior Driven Development and Test Driven Development styles of programming for testing the code in any testing framework. Both packages have been used together and allow to implement the unit tests required to achieve the desired quality in the project. This way, changes to the code or refactorings can be done quickly knowing in all moment that the code still works and has not been broken. Alternatives like Jasmine[14] or AVA[3] have also been studied, but simplicity, community and modularity have been valued above all, making Mocha and Chai the best pick.

6.6 GraphQL

GraphQL[10][11] is an application layer query language. It provides an abstraction layer to the requester of details of the stored data or procedures. GraphQL is designed to interpret a query from a server or client and return that data in an understandable, stable and predictable format. GraphQL was developed to cope with the need for more flexibility and efficiency. It solves many of the shortcomings and inefficiencies that developers experience when interacting with REST APIs. The major differences between REST and GraphQL come to fetching data from an API. With a REST, it usually happens that there is the need to request data from several endpoints or that is obtained more data than required. This problem can be fixed writing specific endpoints, meaning to have a considerable amount of code to write on the server to prepare the data for each specific client. But this is not a good solution, is not scalable and ends with duplicated code.

A GraphQL query is a string that is sent to a server to be interpreted and fulfilled, which then returns JSON back to the client. GraphQL let the clients define the data shape, which makes it easy to predict the shape of the data returned from a query, as well as to write a query once the data an app needs is known. GraphQL naturally follows relationships between objects and this relationship is easily visible. Data is strongly typed. Each level of a GraphQL query corresponds to a particular type, and each type describes a set of available fields. This allows GraphQL to provide descriptive error messages before executing a query. GraphQL is just a protocol in an upper layer to the business logic or the storage. This makes it easily integrable to any current REST API. Moreover, there is no version. The shape of the returned data is determined entirely by the client's query, so servers become simpler and easy to generalize. This allows to add new fields leaving existing clients unaffected. And otherwise, fields can be deprecated when continue to function. This gradual, backward-compatible process removes the need for an incrementing version number. In the GraphQL application layer, schemas are used to define the shape of the data. These schemas are both to fetch data or to mutate data.

The choice of GraphQL instead of REST API is due to those problems described that it fixes and what it provides.

6.7 Cocos2d-x

Cocos2d-x[5] is an open source game framework written in C++, with a thin platform dependent layer. It is widely used to build games, apps and other cross platform GUI based interactive programs. Cocos2d-x allows developers to build truly native games for Android and iOS from a single codebase with a variety of languages like C++, JavaScript and Lua. Cocos2d-x' features are next: rendering, multi-platform and multi touch support, sprite sheet and asset loader, accessibility, full scene graph, sound and deploy as a native App. To these characteristics is added the fact that it is open source, multiplatform and that it builds truly native apps.

Alternatives like Phaser[21] or Pixi.js[22] have also been studied. Phaser is a mobile HTML5 game framework for Canvas and WebGL that powers browser games. Pixi.js is an HTML5 creation engine that allows developers to create beautiful digital content with the fastest, most flexible 2D WebGL renderer. Both alternatives would be sufficient to create the game but would require additional technologies such as Cordova or PhoneGap to make them work as an application on a smartphone. Nevertheless, these technologies would compile the game into a hybrid application, which would not meet the requirements of a native application. This makes Cocos2d-x be the best option.

6.8 MongoDB

MongoDB[18] is an open source database that uses a document-oriented data model. It is part of the group of NoSQL databases. Instead of using tables and rows as in relational databases, MongoDB is built on an architecture of collections and documents. Documents comprise sets of key-value pairs and are the basic unit of data in MongoDB. Collections contain sets of documents and function as the equivalent of relational database tables.

For the microservices that are implemented in this project, a NoSQL database complements better. As will be explained in the next sections, both boards, games and player information will be stored in a JSON structure. Each type of information lives in a different database, so there is no direct relationship between data in the same database. if this is joined by the fact that the amount of data can be huge, leads to the conclusion that a NoSQL database is better than a SQL database.

6.9 Docker

Docker[8] is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow developers to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, thanks to the container, developers can be assured that the application will run on any machine that contains the docker environment installed

regardless of any customized settings that machine might have that could differ from the machine used for developing and testing the code.

The main characteristics are:

- A Docker container is guaranteed to be identical on any system that can run Docker. Dependencies or settings within a container will not affect any installations or configurations on your computer, or on any other containers that may be running.
- With important caveats (discussed below), separating the different components of a large application into different containers can have security benefits: if one container is compromised the others remain unaffected.
- Docker Hub is a repository with many well-maintained images available, such as MongoDB, Node.js. These images can be easily used just specifying in the dockerfile or as dependent services.
- Docker works well as part of continuous integration pipelines. Every time the source code is updated, pipeline tools can save the new version as a Docker image, tag it with a version number and then deploy it to production.

6.10 WebSockets

Websocket[7] is a communication protocol which provides bidirectional communication between the Client and the Server over a TCP connection, WebSocket remains open all the time so they allow the real-time data transfer. When clients trigger the request to the Server it does not close the connection on receiving the response, it rather persists and waits for Client or server to terminate the request. Both browsers and native applications for mobiles allow the use of this protocol.

The key feature of Websocket needed in this project is the real-time communication to allow players to play real-time games. For simplicity and ease of use, a Socket.io has been used, which is a library that abstracts the WebSocket connections.

7 Architecture

A good architecture helps increasing productivity, by making it easy to develop new features, either at the first stages of the development or after the product has been released to the market. When the structure is decided and built correctly, the location of every new piece of code is known beforehand and it helps to have a better quality code and maintainability. This way, it's also easy to find bugs and anomalies and fix them. Otherwise, a bad architecture leads to an exponential decrease in productivity and increases the costs. Special emphasis and a great importance have been placed in the design of the architecture, both at a general level and at a detailed level of each piece the system. It has been designed thinking in the future needs of the project. Single responsibility principle. When it is an online game, where game logic goes is an issue that matters a lot. So much that success also depends on it, becoming a critical point.

In this section, the architecture, components, communication are explained in detail. Also, the concept of the microservices architecture and the API Gateway Pattern are briefly described.

7.1 Microservice Architecture and API Gateway Pattern

A microservices[16] architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability.

A microservice architecture has the following characteristics:

- Services are small, independent, and loosely coupled.
- Services can be deployed independently.
- Services are responsible for persisting their own data or external state.
- Services do not need to share the same technology stack, libraries, or frameworks.
- Services communicate with each other by using well-defined APIs.

The API gateway[20] is the entry point for clients. Clients do not call services directly. Instead, they call the API gateway, which forwards the call to the appropriate services on the back end. The API gateway might aggregate the responses from several services and return the aggregated response. This pattern decouples clients from services. Services can be versioned or refactored without needing to update all of the clients. Also, it can perform other cross-cutting functions such as authentication, logging, SSL termination, and load balancing.

7.2 Architecture

The system is divided in three main components. One component is the Smartphone App, which is the only component that belongs to the client side. This component contains the parts of the game that interact with the players and their animations. It has no game logic because it remains the server side for security. A second component is the Multiplayer Game Server which contains the core logic of the game and the management of the games, rooms and players. The third and last component is the WebAPI, which contains the data and business logic. At the same time, this component is composed by several small components, just like the BoardAPI, PlayerAPI or the GraphQL API Gateway. This component uses a microservice architecture with an API Gateway pattern. Both Multiplayer Game Server and WebAPIs belong to the server side.

All components communicate with each other:

- Smartphone App - Multiplayer Game Server: there is

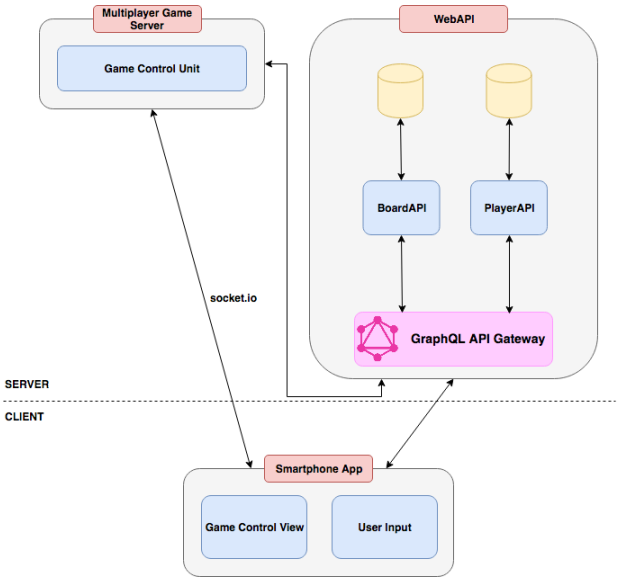

Figure 2: The general architecture of the system

two-way communication with the use of sockets. This communication is born from the need to notify in real time all players in the platform or in one game when one event occurs. Communication from Smartphone App happens from the interaction of the players, as such validation of the actions, start a game. Communication from Multiplayer Game Server happens when an event from a players needs to be replicated to the other players. For example, when a player validates its moves, an event is sent to the server and then the server notifies all the players in the game that that player validated its moves.

- Smartphone App - WebAPI: Communication occurs from the Smartphone App to the WebAPI when data is requested or stored. For example, when the list of the currently shared boards is requested and displayed on the user app.
- Multiplayer Game Server - WebAPI: The communication happens from the Multiplayer Game Server to the WebAPI. For example, when a game starts to request the data of the board or when a game ends, to store the result.

In the next section, each component is explained in more detail.

7.3 Components of the architecture

7.3.1 Control Unit

The Control Unit is the core of the application where game logic belongs. The development has been carried out with special care with the aim of having the code as simple as possible and well structured to facilitate growth in future functionalities by making code easier to reason and understand. There are three key components which should be able grow: robots, boards and engines. Robots should be able to grow with new skills, boards with new objects and engines with new gameplays. This way a special emphasis has been placed in the design of its architecture.

A robot is built by some basic properties like so id, damage, damage to be destroyed, position, direction or stunned. These properties just allow robots to do only a few things: move one square to any of the four directions, be damaged, be destroyed, rotate or be stunned. If skills or behaviours want to be improved or added, it has to be done with power ups. Robots have a set of power ups, which gives special skills, such as damaging other robots, moving more than one square in one action, repairing and so on. This causes that the only way to increase or improve the skills of a robot is adding new power ups keeping the logic of the robot intact.

Boards can grow in terms of elements in the squares such as ice, destroyable walls or barrels. Boards and elements have been designed in such a way that there is no need to modify any code when a new element has to be added, but the specific code for the element has to be added.

The prototype developed contains one gameplay but it is not the only possible mode that can exist. Other game modes can be a solo game (which the player plays alone with robots controlled by an AI), a race where players must capture all the marks in the board or a team mode where players play in teams. The way to grow is by adding new game engines keeping unbroken the existing engines, reusing all the common logic and adding the specific logic for that kind of mode. This way, engines remain simple and do one thing.

7.3.2 Multiplayer Game Server

The Multiplayer Game Server controls the management of the games, rooms and players. It also uses performs the simulation of the games by using the engine available on the Control

Unit.

On one side, it saves in memory all the players connected to the game and all the games and their states. On another side, all the events of the game in the smartphone app go through this component. Any time an event happens, such as a player joins or leaves, a game is opened or a player validates its actions, the Multiplayer Game Server manages it by updating all the necessary data and notifying all the players involved.

7.3.3 WebAPI

The WebAPI Component follows a microservice architecture pattern and an API gateway pattern. Each component that belongs to WebAPI is independent and is packaged up in a Docker Container. Besides, all components except GraphQL API Gateway manage its own data which save and request from a local database. Thanks to the microservice architecture and to Docker, functionality can be easily expanded just adding new components. This way, components are not impacted when a component is added, removed or restarted and can be scaled independently. Currently, the API consists of 4 components: BoardAPI, PlayerAPI and a GraphQL API Gateway. Below each component is explained. All the private components use the architecture REST to be communicated by the GraphQL API Gateway.

GraphQL API Gateway The GraphQL API Gateway is the only visible part to the outside world and all requests go through this component, whatever the request. Any request, whatever the client is, has to point to this component. It is responsible for performing all the needed requests to all the private APIs to get all the requested data, merge the data if required and return the data in the shape specified in the query. Two schemas have been described. One to define how data can be fetched and another to define how data can be mutated.

In the fetch schema, all the available types in the root, such as board, boards, player, players, search with its resolvers have been specified. Resolvers are nothing other than a request obtaining the data from the appropriate API. Resolvers can also exist in the fields of a type. This way, these fields or properties can be transformed or can be fetched from another API because the current type does not have this data. For example, saved boards in the database only store the id of the creator, but not the username or the rest of its fields. In a typical REST API, a client would need two requests to recover the username of the creator of a certain board. Nevertheless, with GraphQL a client only needs one request because once the board fetch ends the defined resolver takes care of fetching its creator data and adding it to the decided field.

Mutations work the same way. But in this schema, fields are named as operations even when the performance is the same, such as cregateateBoard, deleteBoard, updateBoard, where the resolver functions are simple calls to the APIs.

7.3.4 PlayerAPI and BoardAPI

These components are only visible by GraphQL API Gateway and can only be accessed through REST requests. Between them, there is no communication, with which no component accesses another. These components share technologies with which they are built. Each service uses Express to create the API server, its data is stored in a json format in a MongoDB database, uses Mongoose to access to the database and to define the model and uses Mocha and Chai to unit test the resources.

The difference between these components lies in the data that they deal and they own. So, the PlayerAPI owns the players data, BoardAPI owns the board data. In each case, resources to mutate and read data have been defined. The data that belongs to them can only be accessed and mutated from the operations that the service offers.

7.3.5 Smartphone App

The Smartphone App contains that part of the game in which players can interact and play the game. It is the only accessible part by the players. From here, players can open new games, join current games, see how many players are connected in that instant and play real-time games with other players.

This app was built with Cocos2dx and JavaScript as a programming language, which are the technologies that fit better with the requirements.

The application is composed by four screens:

- Welcome: It is a screen that waits for the player to connect to the server and to fetch the data used in the Home screen. It contains a picture of the game and a button to go to the Home.
- Home: This screen contains a list of open games, in which any player can join and only the creator can start, the number of online players and a button to open a new game.
- Game creator: Screen with the options to open a new game, the name of the board and a button to open it. The options are the max number of players, the maximum time per turn.
- Gameplay: Screen that contains everything needed to play. On the left and in most of the screen is composed by the board with the robots where animations are shown. And on the right there are the selection of actions, the actions already selected, button to validate the selected actions and a counter with the remaining time of the turn.

7.4 The phases of a game

In the phases of a game, since a player opens a game until it ends all components have their function.

The phase starts when a player opens a game by selecting the board, the number of players and the time per turn in the mobile application. Then the Multiplayer Game Server is notified, saves this game and notifies all the players that a game

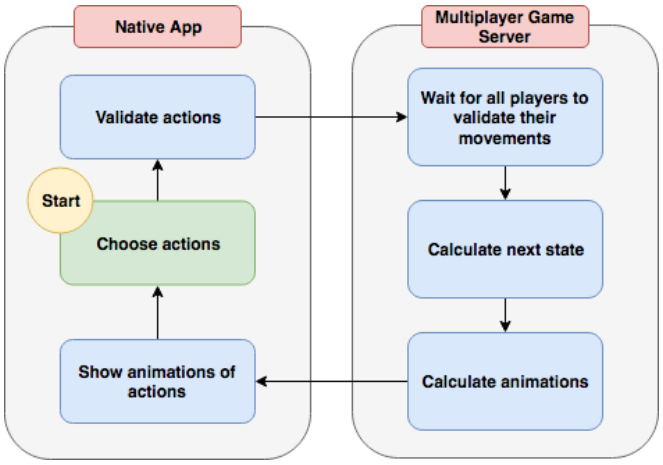

Figure 3: Phases of a turn

was opened. It is at that moment when the rest of players can join. Every time a player joins the game, the Multiplayer Game Server and the rest of the players are notified. When the game starts, the Multiplayer Game Server fetches the board from the WebAPI, sends it to all the players that plays the game and at the same time notifies them that the game is starting.

At this moment, there is a cycle that repeats until the game ends:

- Choose actions: Each player chooses all the actions of its robot.
- Validate actions: Each player validates its actions, which are sent to the Multiplayer Game Server.
- The Multiplayer Game Server waits for all players to validate their actions.
- Calculate next state and animations: The Multiplayer Game Server calculates the next state of the robots and the board and the animations of the current turn. Then, the animations and some data of the new state are sent to all the players.
- Show animations: When players receive the animations, these are shown in the smartphone application.
- Wait for all players to end showing the animations.
- After all, animations are shown, step 1 is repeated until games end.

7.5 Justification of the architecture

The idea behind the choice of the Microservices Architecture, the API Gateway Pattern and the GraphQL is to avoid a single, huge and uncontrollable project. Logic must be decoupled in several small independent services more controllable and easy to understand and reason. This is just like a microservices architecture offers. The game developed is a prototype that can grow in functionality, as adding teams, tournaments and so on. This functionality should be added to the back side. This architecture allows add this almost without breaking existing code since this new code has to be added in a new component. Actually, the only component that has to be updated is the GraphQL API Gateway. This way, the rest of the components remain intact. Uncoupling complicates the client code due to the exaggerated large number of requests needed in different endpoints to get all the required data. The API Gateway Pattern solves this problem bypassing all the requests in one entry point. Nevertheless, like in any API, clients still need multiple requests to get all the desired data, which makes the code more complex. With GraphQL clients can get all the desired data in one single request just specifying it in the GraphQL query.

Games are source of cheating players trying any trick to win such, as looking for the remains of the source code of the game any track, bug or way to change the score or know in advance what other players do. A way to combat these tricks is to move as much code as possible to the server side, place where players cannot access and keep the client side as simple as possible in terms of game logic. This is the reason why it has been decided to implement the code on the server. Of course, this code could also exist on the client, but the difficulty to avoid possible tricks would be too great, what makes to be easier to treat on the server side.

7.5.1 Benefits of the architecture

In terms of the Microservice Architecture, firstly, it offers independent deployments. Any time a service is updated there is no need to redeploy the entire application, and roll back an update if something goes wrong, but the affected service is enough. This way, bug fixes and feature releases are more manageable and less risky. Too, if a service goes down, it will not take out the entire application. Secondly, there is an independent development. A single development team can have a complete control of a service and can build, test, and deploy it using the more appropriate or desired technology for the picked service. The result is continuous innovation and a faster release cadence. Thirdly, this separation of services makes the scope to be smaller by making the code base easier to understand and to test. And finally, the granular scaling is also a benefit. Services can be scaled independently. At the same time, the higher density of services per VM means that VM resources are fully utilized. Using placement constraints, services can be matched to a VM profile (high CPU, high memory, and so on). As regards the API Gateway pattern, firstly it insulates the clients from how the application is partitioned into microservices and the problem of determining the locations of service instances. Secondly, it insulates the clients from the problem of determining the locations of service instances. Finally, it reduces the number of requests by making it possible for clients to retrieve data from multiple services with a single round-trip. Logic in the client becomes simpler without needing a large number of requests.

GraphQL comes with a more elegant methodology and experience concerning data retrieval. Clients only need to specify the type of data that they want to retrieve. GraphQL also

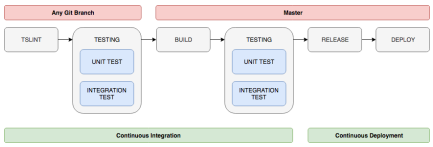

Figure 4: Pipeline of the Continuous Integration and Continuous Delivery.

increases efficiency, both in the client side and on the server side, reducing the call data demands for the client. When new services or fields are added, clients do not need to do more calls or change them, but they only need to specify the new fields.

The benefit of the logic on the server side is none other than to avoid traps of players. But this leads to a problem of scalability, in which more resources are needed when the number of players increases.

8 Continuous Integration, Continuous Deployment and Docker

8.1 Continuous Integration, Continuous Deployment

A Continuous Integration / Continuous Deployment[6] pipeline has been set up for all the components of the project. The Continuous Deployment has only been set up for the backends components. Thanks to this pipeline, at the end of the process the code is automatically deployed only if all the stages run successfully. The pipeline is show in the figure 5.

The pipeline consists in five stages. The first two stages run every time a commit in any git branch is pushed. The next three stages only run when a commit is pushed to the master branch. Commits to the usually master branch usually happen with the goal of deploying the project. If any stage fails, the pipeline stops running and notifies of the failure. Whether it is the master branch, the code is deployed. This pipeline enables to deploy the code automatically to the Heroku[12] hosting. The stages defined are next:

- **Stage 1 Linter**: Code is analyzed to flag programming errors, bugs, stylistic errors, and suspicious constructs.
- **Stage 2 testing**: Code is tested in a Node.js environment through the run of unit and integration tests.
- **Stage 3 build**: With the docker compose, a Docker image is built and is pushed to the Gitlab registry.
- **Stage 4 testing**: Both unit and integrations tests run again, but with the difference that this time it is tested with the same image that is will be deployed then. This allows to run code in an environment that is guaranteed to be identical to the deployed. This way it can be assured that the code will not fail once is deployed.
- **Stage 5 deploy**: The image is obtained from the Gitlab Registry, is pushed to the Heroku registry and is deployed to the Heroku app.

8.2 Docker

All the backend side components have been packaged up in a Docker Container. Some components own the data, such as PlayerAPI and BoardAPI. As data is stored in a MongoDB database, a MongoDB image has been defined in the dockerfile as a service. This simplifies the development and the deployment of the component. On one side it is because there is no need to manually install MongoDB with possible problems such as versions. And on another side, because it does not require any additional script or manual work to run the own component and all its dependencies, but running the docker container is enough.

9 Conclusion

The aim of this work is to build an existing game from the very beginning adapting it to the current smartphone market and to new and modern technologies. With the implementation of this project, all the goals have been achieved.

First, for the first time ever, playing this game from a smartphone has been possible. An APK with the game has been generated and installed to several Android smartphones. All the backend components have also been deployed to a Heroku hosting, being accessible from the APK. This allows users with the APK installed to play the game in real time with other users from anywhere in the world.

Second, the base has been built to being able to continue the game using everything that has been developed in this project. Knowing in advance the possible future features of the game, the architecture has been designed in such a way that makes easy the growth of the game and the implementation of the new features. This has affected the project in two sites. The first site is the architecture of the backend. As business logic will grow, a microservice architecture has been used so that this business logic can be grouped into small and independent components. Currently, data is only accessible from the Multiplayer Game Server and the smartphone app and with a single level of access, but in a future, it will be accessible from more sites and with different levels of access. It is here where GraphQL and the gateway pattern helps with the treatment of the access and the data. The second site is the core of the game, which has been modularized to be able to add new game modes, to add new elements on boards and new power ups to robots.

Third, some related technologies have been learned in detail both to learn them as to implement them. In a way to select the technologies that fit better with what is developed and to the goals, many more have been analyzed and studied.

And fourth, tools and methodologies to ensure the quality of the project have been used. This way, the project has to be developed following some standards that try to make the code as simple as possible, readable and easy to understand. And

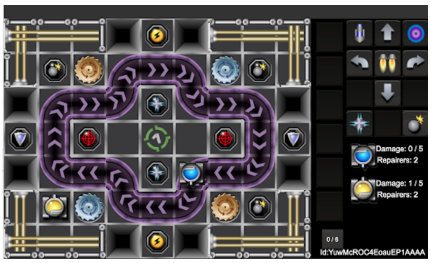

Figure 5: Example of the game developed.

not only this, but the code is to automatically deployed after is committed only if nothing is broken.

Thanks to this project, many of the capabilities acquired in the master, both at the level of technology and methodologies have been practiced, used and learned in more detail.

10 Future work

In this delivery, only one first phase has been carried out. But the first and most important stones have been placed. A functional prototype has been built. Players can start and play multiplayer games in real time from a smartphone app on Android.

In the next phases of the projects the next will be implemented:

- Continue App for smartphones.
- Watch mode: Players will be able to see other games in live.
- Board creation: Players will be able to create from scratch, play and share their own boards thanks to a tool in the smartphone app.
- New graphical design for boards and game animations.
- Build and test the smartphone app on iOS.
- Improve the game app.
- New game modes, like races, solo, teams.
- A friendly start tutorial to introduce the game to new players.
- Add security to the data.
- Add logging system on the backend.
- Deploy App to Play Store and App Store.

Appendix A Projects

Appendix A.1 Grouping of components and projects

The components and the projects have been grouped in two mono-repositories with the projects related to each other and in one repository. The idea behind putting together the related projects in one repository is to simplify the dependencies because most of them are shared, to reuse code in a simple way and to save project maintenance time. The first one is the **game-api** repository and is composed of the GraphQL API Gateway, the Board API, the Player API, the game API and a shared component that has shared functionality of the other component. The second one is the **game-engine** repository and is composed of the game-control-unit, the multiplayer game server and also a shared component that has shared functionality of the other component. The last repository is the **game-app** which contains the native application.

To facilitate the management of dependencies and maintenance a tool called Lerna[15] has been used.

Appendix A.2 Structure of the projects

Game Control Unit

- src: this folder contains all the files with the logic of the project.
  - components
    - board: folder with all the board's components and its logic. Each board's component is defined as a class with its specific logic.
    - powerUps: folder with all the power-ups' components and its logic. Each power-ups' component is defined as a class with its specific logic.
    - Robot.ts: Class with the definition of a robot and its functionality.
  - creator: Functionality to create the basic objects from JSON structures that are used in the engines.
  - engines:
    - DestructionEngine.ts: File that contains all the specific logic related with the destruction mode.
    - Engine.ts: Abstract class with the shape of an engine and the shared functionality that each engine gas to run. All engines must extend this class.
  - models: It contains files with the definition of the models.
  - utils: It contains modules with common functionality of this project.
- tests: this folder contains the files with the tests.

Multiplayer Game Server

- src: this folder contains all the files with the logic of the project.
  - rooms: Folder with the different kinf of rooms
    - rooms: IndividualRoom.ts: File that contains all the logic of an individual room.
    - Room.ts: Interface with the specification of a Room. All rooms must implemenet this interface.

  - models: It contains files with the definition of the models.
  - utils: It contains modules with common functionality of this project.
  - server.ts: File with the creation and run of the server.
- tests: this folder contains the files with the tests.

Native App

- src: this folder contains all the files with the logic of the project.
  - actions: Functionality to add and update the actions in the game scene. It also contains the related logic for the treatment of the selection of the actions.
  - animations
    * board: Folder whose files contain all the logic to show the animations of the components of the board.
    * powerUps: Folder whose files contain all the logic to show the animations of the power-ups of the robots, such as the explosion of a bomb, the shoot, the user of a repair and so on.
    * robot: Folder whose files contain all the logic to show the animations of the robots, such as be damaged, be destroyed, move, turn and so on.
  - layers: Folder with the functionality to create and show in the application each layer of the board.
  - models: It contains files with the definition of the models.
  - robotStatus: Functionality to add and update the status of the robot in any scene. In this case, it is used for the
  - scenes: Folder with the functionality to show each scene.
    * app.js: Initial scene when the app is opened.
    * game.js: Scene with the game.
    * gameCreator.js: Scene to create a new game
    * home.js: Home scene with the list of current games.
  - server: Functionality to communicate the native application with the Multiplayer Game Server with Socket.io
  - utils: It contains common functionality of the application.

GraphQL API Gateway

- src: this folder contains all the files with the logic of the project.
  - fetch: Modules with the request to create, delete, get and update boards and players.
  - modules: Definition of the GraphQL modules for the Board, the Player and the common functionality.
  - utils: It contains common functionality of the application.

BoardAPI and PlayerAPI

Both BoardAPi and PlayerAPI share the same structure.

- src: this folder contains all the files with the logic of the project.
  - controllers: Specific business logic for the project.
  - models: It contains files with the definition of the models and the mongoose schemas.
  - repository: Functionality to access to the database.
  - routes: Definition of the public API of the project.
  - utils: It contains common functionality of the application.
- tests: this folder contains the files with the tests.

Appendix B Third Libraries

All the third libraries used in the different Node.js projects are briefly explained in this appendix.

TypeScript

Package that contains the TypeScript compiler to transpile the code to JavaScript.

ts-node

ts-node is an executable that allows to run TypeScript code in Node.js directly without having to transpile to JavaScript first. It is used to run the projects in development and to run the unit tests.

TSLint

TSLint is an extensible static analysis tool that checks TypeScript code for readability, maintainability, and functionality errors. It is widely supported across modern editors build systems and can be customized with the own lint rules, configurations, and formatters.

Lerna

Lerna is a tool for managing large scale JavaScript projects with multiple packages. Lerna can also reduce the time and space requirements for numerous copies of packages in development and build environments - normally a downside of dividing a project into many separate NPM package.

Mongoose

Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box. It acts as an intermediate between MongoDB and the server side language. Its use facilitates access and writing of data.

Immutable.js

Immutable.js[13] is a set of immutable collections for JavaScript. It provides many Persistent Immutable data structures,like List, Stack, Map, OrderedMap, Set, OrderedSet and Record. Immutable data is data that cannot be changed once created. It leads to much simpler application development, reasoning of the code simpler and change detection techniques with simple logic. Persistent data presents a mutative API which does not update the data in-place, but instead always yields new updated data. This set of immutable collections have been used to facilitate and simplify the use of immutable data.

Socket.io

Socket.IO[23] is a JavaScript library that enables real-time bidirectional and event-based communication. It works on every platform, browser or device, focusing equally on reliability and speed. The reason behind its choice is the need of constant communication between the game server and the smartphone application, so that players can send to the server their actions or for the server can send the players the animations to display.

@types/chai, @types/mocha

TypeScript types to run mocha and chai packages in a TypeScript project.

@types/graphql

TypeScript types to run the graphql package in a TypeScript project.

@types/request

TypeScript types to run the request package in a TypeScript project.

@graphql-modules/core

GraphQL Modules is a toolset of libraries and guidelines dedicated to create reusable, maintainable, testable and extendable modules out of your GraphQL server.

apollo-server-express

apollo-server-express is the Express and Connect integration of GraphQL Server.

dotenv

Dotenv is a zero-dependency module that loads environment variables from a .env file into process.env

graphql

JavaScript reference implementation for GraphQL.

graphql-tag

It is an utilities library to parse GraphQL query strings into the standard GraphQL AST.

graphql-tools

It is a library that provides different useful ways to create a GraphQL schema, such as a GraphQL type language string.

request

Request is a library designed to make http calls.

Appendix C Explanation of the game

In this subsection all the features of robots, power-ups and boards are briefly explained.

Appendix C.1 Robot

Robots are what players control.

- Direction where robots are looking at. The direction can be north, east, south and west.
- Points of damage that robot has received. Robots start with 0 points of damage and they can be damaged by other robots or by elements of the board. Robots are destroyed when their damage reach 5 points.
- A set of power-ups where robots can increase their skills. Power-ups can be divided in three types:
  - Power-ups that can only be got from the board: these power-ups are the bomb, autolaser, x-cross, partial repair and shield. Each robot can only have one at the same time.
  - Power-ups that are restored at the beginning of each turn and that all robots have. These power-ups are the shoot, the boost and the blast and are restored to six, five and three power-ups respectively.
  - Power-ups that are added at the start of the game and cannot be got again. This is the repair and it is used automatically at the end of a turn every time a robot is destroyed.

- Stunned: While robots are stunned they cannot execute any action.

Appendix C.2 Power-ups

Power-ups are the only way that allow robots to improve their skills.

- Bomb: It explodes in a range of 2 squares at the end of the turn and damages the robots within the range. If the robots are not destroyed, they are stunned during the next turn and they cannot do any action.
- Repair: Robots recover until they have 0 points of damage. This power-up is currently used automatically after a robot is destroyed.
- Partial Repair: It allows robots to recover 3 points of damage or until the damage is 0 points.
- X-cross: Shot of 2 points of damage in the four directions. Each shoot is independent and it stops when it damages a robot or there is a wall.
- Autolaser: Robots automatically shoot in each action.
- Shield: Shield of 3 points of damage that protects the robot until the end of the turn.
- Shoot: Shot of 1 point of damage in the direction that the robot looks. The shot stops when it damages a robot or there is a wall.
- Blast: It damages one point to all the robots that are placed in one of the 8 neighboring squares.
- Boost: It accumulates actions doing nothing to move several squares at one time. For example, if a robot uses one boost, the next action it would move 2 squares in one direction. In other case, if it uses 4 boosts, then it would move 5 squares in the desired direction. Robots cannot pass through holes, but can pass through slides without being moved or lasers without being damaged.

Robots cannot pass through holes, but can pass through slides without being moved or lasers without being damaged.

Appendix C.3 Board

board: it is a 2 dimension board composed of squares. Size goes from 5x5 to 13x9.

A square is composed of 5 parts:

- Center: It occupies most of the square.
- The four edges (north, east, south and west).

A square can contain zero or one element in any of the parts or in all.

- Wall: wall that occupies the center of the square. It blocks robots, shoots or lasers in any of the four directions, but bombs can.
- Wall in the edge: wall that only occupies the edge. It blocs robots and shoots in the same direction or the opposite. For example, one wall in the north or in the south would block robots or shoots coming from the north or the south, but not from the east or the west.
- Mark: It indicates the initial position of the robots or where they are placed when they are destroyed.
- Hole: It destroys the robots that move or place on top.
- Gear: Rotates the robots places on top. Gear can rotate robots to the left or to the right according to the type.
- Power-up: Robots get power-ups when they are placed on top of each other during at least one action. The power-up of square can only be got once per turn. At the end beginning of the turn they are restored. Board can have power-ups for bomb, partial repair, x-cross, autolaser and shield.
- Sliders: Sliders move robots one square in one direction. There are one kind of sliders that also rotate robots.
- Laser: Lasers are placed in the edges and can damage 1, 2 or 3 points to the first that find in that direction.

Appendix C.4 Actions

Actions represent the way in which players can control their own robots. The basic actions are as follows:

- Move forward: Robots are moved one square forward.
- Move backward: Robots are moved one square backward.
- Move right: Robots are moved one square to the right. This action is only available after the use of the boost power-up.
- Move left: Robots are moved one square to the left. This action is only available after the use of the boost power-up.
- Turn left: Robots turn 90^{o} to the left. For example, if the direction of a robot is north, the next direction after turning left would be west.
- Turn right: Robots turn 90^{o} to the right.
- Non action: the robots does nothing. This action cannot be chosen by the player and it is added automatically when the robot is stunned.

More advanced actions or special skills actions can only be executed having power-ups in the lot. These are the actions available with the current defined power-ups:

- Bomb: put a bomb in the square where the robot is in that action.
- Partial Repair: The robot recovers 3 points of damage or until its damage is of 0 points.
- X-cross: Shoot in the four directions a shot of 2 points of damage.
- Autolaser: Start to shoot automatically in each action.
- Shield: Protect itself with a shield of 3 points of damage.
- Shoot: Shoot of 1 point of damage in the direction that the robot looks.
- Blast: Damage to all the players in neighboring squares.
- Boost: It accumulates actions doing nothing to move several squares at one time. After the use of one boost, robots cannot only move to one of the direction or use again a boost. Boosts cannot be used in the last action.

In the algorithm, actions and interactions with the elements of the board run in a specific order. Actions in the same point are executed at the same moment.

1. Shield, partial repair, put of a bomb and autolaser.
2. Shoot, blast, x-cross.
3. Boost, move in any of the four directions, turn.
4. Slider.
5. Gear.
6. Hole.
7. Damage of the lasers of the board.
8. Bomb explosion (only in last move of a turn).

Moreover, animations of the actions and its interactions with the board are shown in its own specific order:

1. Shield, partial repair.
2. Put a bomb.
3. Shoot, blast, x-cross.
4. Destruction of a shield and of a robot.
5. Robot being damaged.
6. Move and rotation of a robot.
7. Slider moving a robot, move of the slider.
8. Robot being destroyed by a hole.
9. Robot turning, gear rotating a robot, rotate of the gear
10. Laser of the board, robot being damaged and shield being damaged, robot being, destruction of shield, destruction of robots.
11. When is last action of the turn.
12. Bomb explosion.
13. Robots being damaged, shield being damaged, destruction of shield, destruction of robots.
14. Robots being frozen.
15. Unfreeze robots non bombed.
16. Put robot on the initial coordinate.

Appendix C.5 Information in the screen of a game

On the screen of a game, players see all the information related to the game. In the left and occupying most of the screen there is the board with the robots of all the players. In its right, there is the list of the current actions selected and the button to validate them once the 6 actions are selected. In the right top of the screen, there are the available actions that the player can select. Just below there are the status of all the robots in the game with its damage and the number of remaining repairs. The player's robot always appears above. Finally, at the bottom the id of the robot appears.

References

[1] 5 big problems caused by bad application architecture. https://www.mrc-productivity.com/blog/2012/02/5-big-problems-caused-by-bad-application-architecture/. Accessed: 2019-05-01.

[2] 7 reasons why you should be using CI. https://about.gitlab.com/2015/02/03/7-reasons-why-you-should-be-using-ci/. Accessed: 2019-05-01.

[3] AVA. https://github.com/avajs/ava. Accessed: 2019-05-01.

[4] Chai. https://www.chaijs.com/. Accessed: 2019-05-01.

[5] Cocos2d-x. https://cocos2d-x.org/. Accessed: 2019-05-01.

[6] Continuous integration vs. continuous delivery vs. continuous deployment. https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deploy
Accessed: 2019-05-01.

[7] Difference Between WebSocket vs Socket.io. https://www.educba.com/websocket-vs-socket-io/. Accessed: 2019-05-01.

[8] Docker. https://www.docker.com/. Accessed: 2019-05-01.

[9] Flow. https://flow.org/. Accessed: 2019-05-01.

[10] GraphQL. https://graphql.org/. Accessed: 2019-05-01.

[11] GraphQL: A data query language. https://code.facebook.com/posts/1691455094417024/graphql-a-data-query-language/. Accessed: 2019-05-01.

[12] Heroku. https://www.heroku.com. Accessed: 2019-05-01.

[13] Immutable.js. https://facebook.github.io/immutable-js/. Accessed: 2019-05-01.

[14] Jasmine. https://jasmine.github.io/. Accessed: 2019-05-01.

[15] Lerna. https://github.com/lerna/lerna. Accessed: 2019-05-01.

[16] Microservices architecture style. https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices. Accessed: 2019-05-01.

[17] Mocha. https://mochajs.org/. Accessed: 2019-05-01.

[18] MongoDB. https://www.mongodb.com/. Accessed: 2019-05-01.

[19] Node.js. https://nodejs.org/en/. Accessed: 2019-05-01.

[20] Pattern: API Gateway / Backend for Front-End. https://microservices.io/patterns/apigateway.html. Accessed: 2019-05-01.

[21] Phaser. https://phaser.io/. Accessed: 2019-05-01.

[22] PixiJS. http://www.pixijs.com/. Accessed: 2019-05-01.

[23] Socket.io. https://socket.io/. Accessed: 2019-05-01.

[24] The benefits of continuous integration in the cloud. https://blog.codeship.com/the-benefits-of-continuous-integration-in-the-cloud/. Accessed: 2019-05-01.

[25] Thoughts on Flash. https://www.apple.com/hotnews/thoughts-on-flash/. Accessed: 2019-05-01.

[26] TypeScript. https://www.typescriptlang.org/. Accessed: 2019-05-01.

[27] What are JavaScript linters? https://www.codereadability.com/what-are-javascript-linters/. Accessed: 2019-05-01.

[28] What is continuous integration. https://docs.microsoft.com/en-us/azure/devops/learn/what-is-continuous-integration. Accessed: 2019-05-01.