Signals

Introduction

The aim of this report is to describe the design, thought and implementation process of the game “Signals”. In overall, “Signals” is a strategy, real-time, multiplayer online game. The goal of a single game is to capture the base of your opponent. In order to win a game, a player has to be able to balance the economy, military and technology development.

The project had several requirements and they all have been fulfilled. The game runs on many browsers, however it is fully playable only on Google Chrome due to its technological advancements and early adaptation of bleeding edge technology. We chose to host our game on an external server provided by Nodejitsu. Not only it is free of cost but also provides support for all the required technologies, especially for WebSockets. We authorize users using their Facebook or Google+ accounts. The database stores player’s profile which consists of: rating, history of played matches and history of messages. Most importantly we do not store any personal information regarding player. The game requires simple database operations, therefore MongoDB has been used as a storage solution. The players have a possibility to message each other and all the conversations are saved on a server. We used various open source tools and resources and all the authors have been given credit in the acknowledgements section.

At the beginning of the development process we set ourselves multiple goals. The main one, was to create a real-time strategy game which will require from a player an ability to quickly make decisions and research innovative strategies. It was also crucial to make an advanced graphical user interface which will not affect overall game performance. Later in the implementation process, game seemed to get too complicated, therefore we set ourselves a target to make the user interface as intuitive as possible and avoid unclear functionalities. As the development progressed we started to encounter an increasing number of performance issues. Consequently, we made our main priority to limit the amount of graphical features.

Project Management

The first step of the development process was a brainstorming session. All the group members quickly reached a consensus and it was decided what type of game will be developed. It was essential at this stage to establish the rules of the game and the basic functionalities. We created a document which included the rules, basic game features, advanced game features and the division of workload. As we managed to complete some of the goals we added more ambitious targets.

One of the goals was to create an event driven, decoupled and asynchronous architecture. The very important step on the way was to design and fully understand all the process and internal mechanisms of the game. Before we started the implementation phase, we decided upon all the events that will

drive the game. Moreover, we came up with ideas how to guarantee a high level of control in an environment where function calls happen in arbitrary order and time.

The next stage was an implementation process. We were meeting daily in either Computing Department Laboratories or in the team members' houses. The group did not have any communication and organisation problems, as we all worked together before. It was essential that at least two members were acquainted with a particular element of the game and respective part of code. This type of group structure enabled us to keep track of the progress at all times. One of the drawbacks was that the group approached this project as a research opportunity into modern web technologies, rather than an assignment that has to result in a specific product. Consequently, the group members wanted to introduce new features and did not focus on debugging some of the most basic behaviours. Unfortunately, some of the features that were developed could not be included, because the team did not manage to fully implement them.

The last phase of the project was a code debugging and beta-testing phase. The group of beta testers included team members and their friends. One of the priorities was to find all the problems which may occur in corner cases. What is more, the testing enabled us to assemble some significant feedback, which helped to improve user interface and some game features. The game turned out to be surprisingly difficult to balance. It will take time and users feedback to adjust entities delays, units power and resource collection.

The following table contains the elements, which have been implemented by particular group members and their responsibilities.

Group Member	**Assignments Description**
Lukasz	Initial game rules. Database management. Synchronisation of WebSockets session with web server session. User panel interface. Terrain generation and rendering. Implementation of shading. Implementation of simple model of water interactions. Units animation. Radial menu generation. Website design and development Initial decisions about the technological stack.

Piotr	Implementation of the rendering engine, which included board grid, ownership, fog of war, platforms, channels, units. Finding and adding media resources, such as animations and bitmaps. Game engine debugging. Routing Algorithms. Game specification and rules development.
Robert	Implementation of the game mechanics. In game event handling Finding and providing tools for the rest of the group. Server communication. Server synchronisation. Database config. User login and social media integration.

Game Rules

Setting

The single game is set on an island where players are the leaders of two hostile tribes. Each tribe wants to claim the ownership of the island and in order to do that it has to capture the opponent's headquarter.

Initial State

At the beginning of the match, each player starts with a headquarter and a fixed amount of two basic resources: food and gold.

Buildings

An user has a possibility to build two types of buildings: platforms and channels. A channel may be built between two fields only if there is some other entity in the field. Channels can only branch out on platforms, otherwise they can have only one channel on each end. A platform may be build at a field, which is already connected to a channel, it the field contains resources, the platform will start production.

Units

Units carry the resources and are able to move along the channels. Player gains resources when a unit carrying it gets to the headquarter On the other hand, an unit can be used to attack other players. An attack occurs when an unit approaches opponent's channel or platform . When channel’s or platform’s life reaches 0, a player whose signal dealt the last hit gains the ownership of the entity. The game ends when one of the player’s gains ownership of all the headquarters on the board.

Routing

Each platform has a routing menu. A player may specify the interactions between platforms and channels. The user decides whether units can move from a platform to the specified channel and whether units may move from specified channel to the platform (The menu is included on one of the screenshots).

Game progress

In the beginning stages of the game players need to focus on the economy. It is necessary to gain ownership of couple of fields with the key resources. The constant income of gold and food is vital to survive in the mid and late game. It the subsequent stages, players need to focus on attacking opponent's entities. It often involves capturing key points of enemy's channel infrastructure. The main goal of a game is to capture the opponent's headquarters, therefore some of the strategies focus on the early attacks and infrastructure development.

Implementation details

From the beginning we were certain that we want to develop modern web based game with all cutting edge technologies. Hence, we picked Node.js as our runtime environment on the server side. Furthermore, due to the fact that Node.js and web browsers use JavaScript, we can share and reuse code between both environments. Although JavaScript is the most common programming language for these two environments, we decided to use CoffeeScript instead. It provides much clearer syntax and all the team members had previous experience with this language.

CoffeeScript is a programming language which is compiled to JavaScript. Apart from enhanced, bracket free syntax, it also introduces useful language constructs, such as list membership operator or foreach loops as a language constructs. It also provides very clear syntax for classes. Since we prefer bracket free and indentation oriented code, CoffeeScript emerged as the only reasonable choice.

Along with CoffeeScript we also decided to make heavy use of Underscore library. It provides many utility functions which one would expect from any functional language. Furthermore it ensures that native version of a supported function is used before Underscore's implementation is used.

To Further enhance development and data management we decided to use MongoDB instead of traditional SQL database system. The main advantage of MongoDB was the fact that its query language is JavaScript. Hence, the whole environment was easy to understand for all the group members. In addition, we decided to use mongoose module which provides object like access to MongoDB collections.

In order for our game to follow current web trends we decided to implement social logins and integrate our game with social services. Therefore, users can login with their facebook or google account. We do not store any credentials and we keep only information required to uniquely identify returning user and display his or hers history. Due to lack of traditional login, our database is extremely simple and any

information stored there is not critical to users.

The next thing is the use of canvas as drawing interface on the client side. Even though the technology might not be perfectly supported in every browser and lacks hardware acceleration in some of them, it was ideal for our project: real-time multiplayer game. It allowed us to access every pixel of the game. However, more importantly it allows to implement efficient animations which is not possible with many DOM (Document Object Model) elements.

Last but not least is the way we decided to solve synchronization issues around the system. In order to achieve persistent real time connection between server and client we used socket.io which is an abstraction layer on top of WebSockets. It simplifies communication as it adds support for channels which we heavily rely on. Socket.io is an excellent example of an event driven communication which we decided to implement across the whole game. Instead of simply calling another objects' functions we trigger an event on common publish/subscribe bus. This design has many advantages like asynchronicity and lack of direct coupling of objects. However, asynchronicity is also its drawback. In order to ensure that event has been received and processed we decided to introduce another technology called promises.

By definition taken from Promises/A specification provided by CommonJS: *"A promise represents the eventual value returned from the single completion of an operation".* Hence, we can observe promise can and be notified when underlying condition changes its state to fulfilled, rejected or failed. It allows us to avoid deep nesting of code. We do not have to introduce callback in a callback ad inifinitum but we can observe a promise regarding value on which callback operates and be notified when its state changes.

It is important to note how promises and events are almost identical in behaviour. Instead of executing a callback when the function has finished, we return a promise from this function and then we can decide what to do once the underlying condition gets resolved. Essentially, both of these mechanisms can be used interchangeably. However, we decided that we prefer events for public api of a class and promises for internal synchronization within function, mainly because of the clarity.

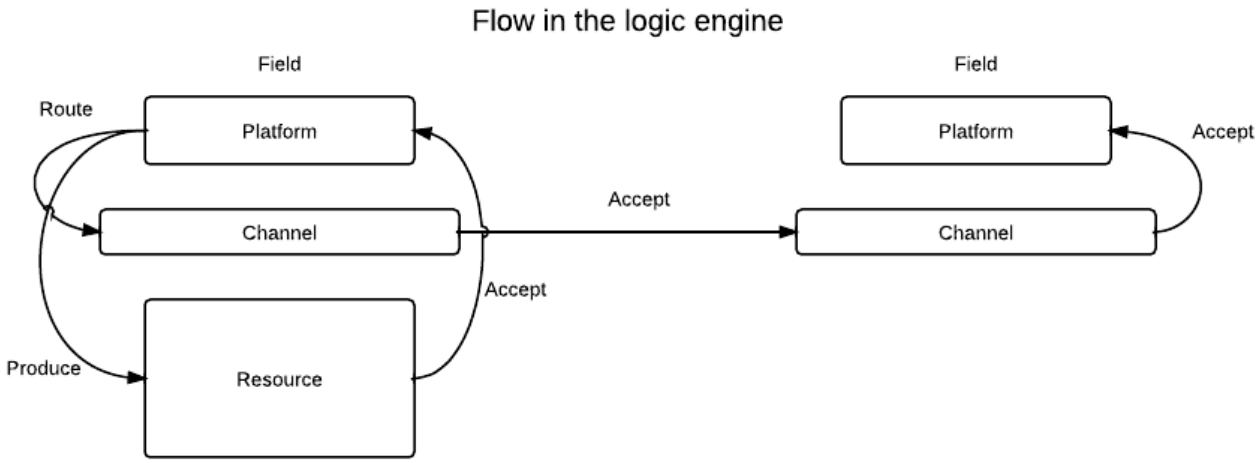
For back-ups we used git along with github service. Apart from version control system we always had authoritative version of code which could be retrieved in case of machine failure. Furthermore we used Google Drive to store diagrams of the system along with detailed descriptions of game mechanics.

To maximally streamline development we looked into common tools used by other programmers. We used Guard which tends to be very popular amongst Ruby programmers. It allows to launch specific tasks upon file change. Hence, we used it to compile our code and refresh browser window on source file save. With this approach we eliminated small annoyances.

We tried to mutually agree on every design decision. Therefore, if any member had to make an important decision about structure of code, he had to consult his idea with the rest of the group.

We started our work on implementation from the very basis, from understanding the relations between objects within the game and stating their clear roles. There are four general types of objects in our program: Resources, Platforms, Signals, and Fields which serve as containers for Resources and Platforms. Signal is the most basic element of our game, it is essential to all the processes, they represent quantifiable amounts of goods, and are transported, mined, stored and used to affect the state of the game. Resources are objects responsible for mining signals from resource fields and passing them to the platform above. In fact a Resource can be seen as a section of a platform responsible for mining external resources (HQs have internal production capabilities and so they do not need a Resources to help them). Once a Resource receives a production order it mines some amount of goods and sends it back up to the platform. The production cycle consists of a Platform initializing production from a field by calling a Resource, which extract the goods, and then passes them to the platform. Platforms are storage and routing centres. They determine which of the possible gates will be used to try to emit a signal. Routing is a complex process, at its bases is a simple route table which tells the platform which of the gates should be considered when accepting and which when emitting. Then once a gate has been chosen, a platform exchanges a handshake with the object behind the gate, and only when they both agreed on the exchange, the signal will be transmitted. In the case of foreign signals, coming from different players, the channels and platforms transmitting them will not attempt a handshake. Instead they will force the signal inside the object, performing an attack. Channels are a special case of Platforms, which serve as a connecting medium between platforms, and themselves does

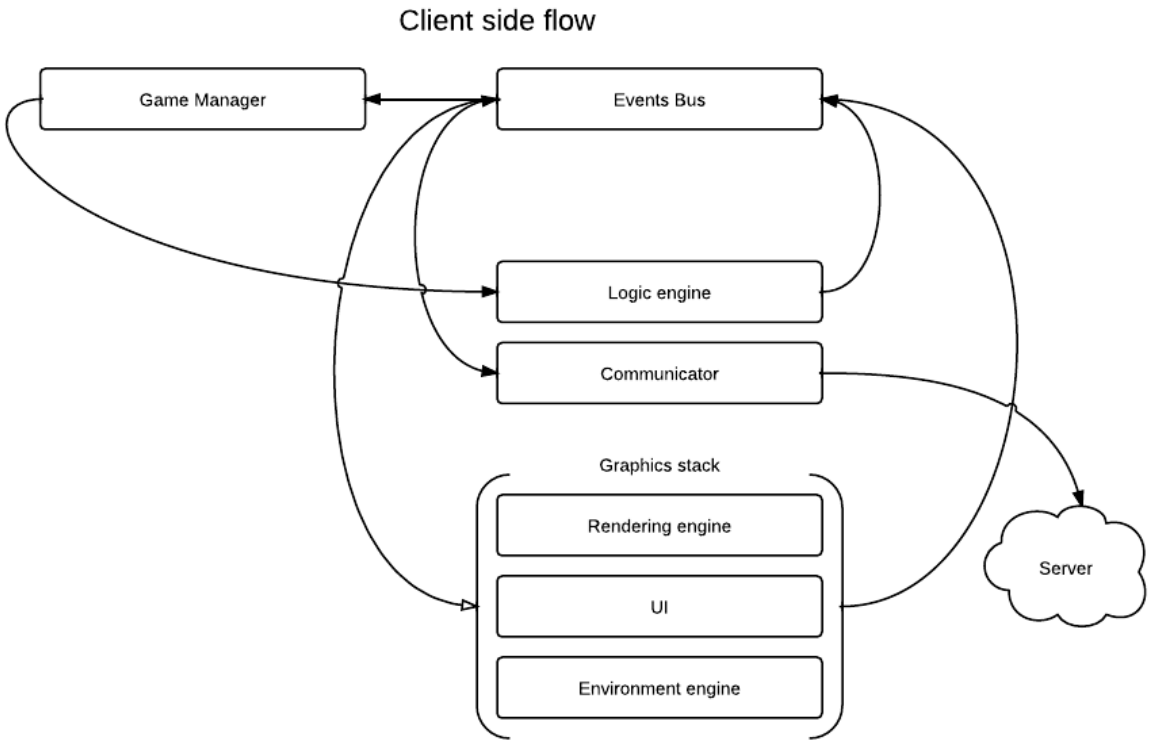
not have any production capabilities. Overall a structure of a single field consists of up to four layers, with the field itself being at the bottom, the resource above it, then the channel and the platform on top.



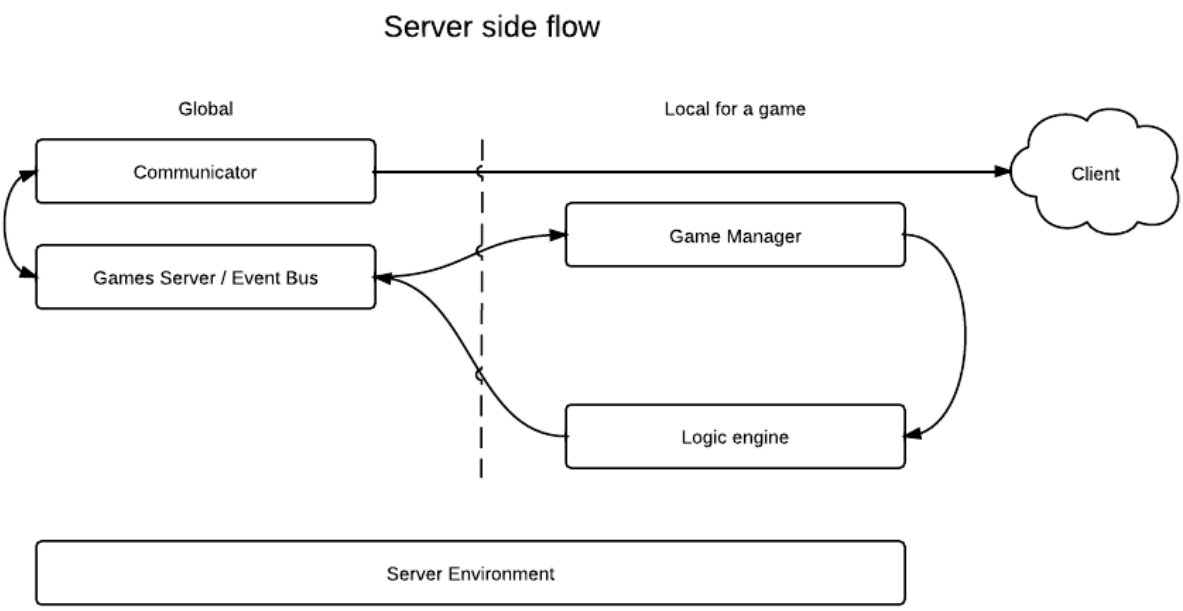
The are four main components existing on the client side of the application. Game Manager is an object responsible for game flow and handling of players. It starts, and stops the game, and makes sure that all players are ready to play before the game starts. It is essential but at the same time one of the smallest components in the entire game. Event Bus provides a common API for other elements of the game, so they can communicate and cooperate in a decoupled environment. Together with the Communicator they take care of the connection and existence of sound state of the game on this side. Only EventBus operates directly on communication module, while every other element has to communicate through it, and it is up to the Event Bus to decide whether Server should be informed about a change caused by user interaction. Logic engine is the core of the game which implements all the rules and mechanics. Our graphics stack consists of three separated elements: User Interface, Rendering Engine and Environment Engine.

User Interface is responsible for handling user interactions, displaying useful informations and passing them further to other components. For user experience reasons we have decided to implement a radial menu. Options with the menu are displayed on an orbit around the object. It is not only an original and unique solution, but also provides better experience by not occupying screen space, which instead can be used by the game. It also allows user to operate around the area of his interest, unlike in classic menus, where user has to first select object he wants to operate on and then go to another area of screen to choose and perform the action.

Environment engine generates surroundings of the game. It provides context in which the game is played. For efficiency reasons all operations are performed first in memory with computations done for every pixel, and once everything is rendered it is transferred onto the canvas. There are three main elements of the environment: water surrounding the island, the terrain covering the surface and shading, which shows shape of the island. The terrain generation in our game is quite complex. It starts on the server side when the game manager initializes game state by generating a height map using diamond replacement algorithm. A height map describes height points of the terrain, with two hundred points for a hexagonal field. Once it is generated we decide on the types of terrain that will cover the map. Currently in our game there are five of them water, sand, grass, rocks and snow, and they layer in this order with water being in the lowest points and the snow in the highest. As a first step of rendering we smoothen the height map to negate big differences in height which may have been created by the algorithm. For each type of terrain we generate a bitmap by randomly modifying HSL parameters of the initial colour, it allows us to achieve better looking, fully parameterized texture with reasonably fast processing time. After the map is covered with texture and the height map is ready we apply shading which depends on the height of the specific pixel. Because we do not generate a height points for each pixel as that would be computationally too expensive, we linearly interpolate values for the missing height points. We used a very simple shading model, where we determine the colour of a pixel basing on the difference in height between this and the previous pixel towards Sun.



The rendering engine generates graphics on several canvases. It creates the board grid, the ownership grid, the fog of war (canvas which covers the regions which cannot be seen by the player), the channels, the platforms and the resources. The main class is the Renderer, which receives all the calls and then delegates them to the specific specialized class. There are several drawer classes which extend the Drawer class. The drawer class includes methods which are used to calculate appropriate board coordinates of field neighbours and to translate board coordinates to pixel coordinates. All the bitmaps are loaded in the Renderer and the constructors of the subclasses are not triggered until they finish loading. All the graphics generated in the drawer classes are cached. When the objects drawn on a canvas change, then the cache is updated.



One of the reasons to use Javascript for development was ability to use the same code on the client and server side. Logic engine is a perfect example of this use case, by writing portable code we were able to avoid necessity to write the same code twice. We run the same engine on both sides, what not only makes coding easier, but also guarantees that situation of the synchronization of game state were caused by either poor connection or user's modifications, not by differences of implementation. On the server side we can recognize two types of objects, those common to all connected users, games and sessions and those which exist exclusively for one of them. Each running game has its own game manager which controls it and is aware of existing players and their physical location, and its own instance of the logic engine which runs the mechanics and is updated with user interactions by informations from the event bus. Construction of the communicator is almost the same as on the client side, except that it is responsible for multiple users grouped into channels of interest, instead of a single user and the server. The Games Server manages players and games they have joined, creates new games and makes sure that there always exists a game that someone can join. It is also responsible for transmitting occurring events to the engine and to all the users.

External resources used during development of the game

Node.js modules

express	http://expressjs.com/
connect	http://www.senchalabs.org/connect/
socket.io	http://socket.io/
coffeescript	http://coffeescript.org/
hbs	https://github.com/donpark/hbs
handlebars	http://handlebarsjs.com/
backbone	http://backbonejs.org/
underscore	http://underscorejs.org/
LESS	http://lesscss.org/
everyauth	http://everyauth.com/
docco	http://jashkenas.github.com/docco/
mongoose	https://github.com/LearnBoost/mongoose
promised-io	https://github.com/kriszyp/promised-io
request	https://github.com/mikeal/request

Javascript libraries

keyboard.js	http://robertwhurst.github.com/KeyboardJS/
modernizr	http://modernizr.com/
sha512	http://jssha.sourceforge.net/
color.js	https://github.com/brehaut/color-js
Zynga Scroller	https://github.com/zynga/scroller
Easel.js	http://www.createjs.com/#!/EaselJS
Handlebars.js	http://handlebarsjs.com/
jQuery nanoscoller	http://jamesflorentino.com/jquery.nanoscroller/
jQuery	http://jquery.com/
handlebars.js	http://handlebarsjs.com/
backbone.js	http://backbonejs.org/
underscore.js	http://underscorejs.org/

Media resources

Unknown Horizons (GPLv3)	http://www.unknown-horizons.org/
Graphics for Battle of Wesnoth (CC0)	http://opengameart.org/content/old-stone-buildings

Summary

First of all we designed purely event driven game logic, furthermore the whole application is event driven, hence blocking occurs rarely. Due to use of events we were able to decouple our architecture.

This means that we can turn off a specific enhancement (like more detailed graphics) without breaking any other functionalities. Because one of the more advanced features of graphics were causing significant performance drops, we decided to turn off parts responsible for water and weather animation. One more advanced feature of the game that we decided to keep is the fog of war, which we deem essential for any RTS. To improve in game experience players can use a dedicated chat to communicate with other players. They can also track their statistics and see how they compare with their friends. To achieve that we require login authorization from one of the social networks, currently supported are: Facebook and Google+.

Due to time constraints we were not able to deliver on all our goals. First thing we scraped was deployment to multiple servers and usage of redis db as a session store. Secondly, we decide to get rid of 2.5D graphics. Even though we were able to convert graphics to that form we had quickly discovered that projecting mouse clicks would be too difficult and cumbersome to debug. Further on we realised that implementing team matches and trading within the game would be too time consuming, hence we decided to focus on core game functionality. As Lucas was working on more advanced graphic features, he had to resign from weather conditions and division into water and land in the playable area. We also aimed to introduce different platform types, however, we first wanted to make sure that everything is working correctly and decided to stay with only two types of platforms. The most surprising fact was that highlighting object under a cursor was very resource intensive and we would have to implement our own event propagation in order to deal with this issue.

We have learned multitude of things during this project. Most importantly we understood how to handle asynchronicity in programs. We had to handle multiple events of the same kind firing at the same time as well as synchronising many event callbacks within one function. We achieved that by using promises which we think is the best approach to dealing with unpredictable time of a function call. In order to generate decent maps we had to learn basics of 2D graphics (shading). Even though due to advancements in hardware those techniques are not commonly used anymore, as more efficient GPU versions exist, we found it beneficial to learn them. Last but not least we learnt client server synchronization and communicating events among multiple players.

What we have not done was to adopt Minimal Viable Product (MVP) ideology, where the most important goal is to deliver working product even though it might not have all the desired features. It would have allowed us to test early and have feedback on developed features earlier. When we had looked at this project from perspective of working product we have realised that we approached this assignment more as a research topic and not as goal oriented task.

All in all we find this project very successful, however it would benefit from more target oriented approach where delivering working product would have been of utter most priority.