Designing and Implementing a browser RTS

João Pedro Lopes Ferreira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Pedro Alexandre Simões dos Santos

Examination Committee

Chairperson: Nuno João Neves Mamede
Supervisor: Prof. Pedro Alexandre Simões dos Santos
Member of the Committee: João Miguel De Sousa de Assis Dias

June 2019

Acknowledgments

I would like to thank my parents and my sister for every support and care they gave since I was born, my thesis coordinator for having an immeasurable patience with the development of this project and supporting me until it's conclusion, my closest colleagues who gave me the support to reach this point of my academic studies, and my friends who are part of my life which shared some of the good and bad moments of it. To all of them a big Thank You.

Abstract

The video game genre that is recognized as Real-Time Strategy (RTS) had many stages of evolution since the beginning of it's creation. As the technology evolved, this type of game can nowadays be played in different types of devices. There is still few RTS games that can be played with more than one different device.

This document presents the solution to design and create a RTS Game that is playable on both personal computers and mobile devices. The game conceived on this project uses HTML5 and JavaScript as programming languages and is playable in multiplayer mode. Multiple players compete between themselves in 1 versus 1 game matches. This game also contains the 4X elements present in the traditional RTS games ("eXplore, eXpand, eXploit and eXterminate").

Keywords

RTS, Multiplayer game, Phaser.io, Client-Server networking, Cross-platform.

Resumo

O género de video jogos que é conhecido como Estratégia em tempo real (RTS) teve vários estados de evolução desde o início da sua criação. À media que a tecnologia foi evoluindo este tipo de jogo pode hoje em dia ser jogado em diferentes tipos de dispositivos. Existem ainda poucos jogos RTS que podem ser jogados em mais que um dispositivo.

Este documento apresenta a solução em desenhar e criar um jogo RTS que é jogável tanto em computadores pessoais como em dispositivos móveis. Este jogo usa as linguagens de programação HTML5 e Javascript e é jogável em modo multi-jogador. Múltiplos jogadores competem entre si em jogos de modo 1 contra 1. Posteriormente este jogo contém os elementos 4X presentes nos jogos RTS tradicionais (”eXplore, eXpand, eXploit and eXterminate”).

Palavras Chave

RTS, Jogo multijogador, Comunicação cliente-servidor, Phaser.io, Multiplataforma.

Contents

List of Figures

List of Tables

Listings

Acronyms

APIs	Application Program Interfaces
CSS	Cascading Style Sheets
CPU	Central Processing Unit
DHTML	Dynamic HyperText Markup Language
DOM	Document Object Model
ESA	Entertainment Software Association
FPS	Frames Per Second
GIMP	GNU Image Manipulation Program
HTML	HyperText Markup Language
RPS	Rock-Paper-Scissors
RTS	Real-Time Strategy
TCP	Transport Control Protocol
UDP	User Datagram Protocol
W3C	World Wide Web Consortium
WHATWG	Web Hypertext Application Technology Working Group
XTML	eXtensible Hypertext Markup Language

1

Introduction

Contents

This is a war universe. War all the time. That is its nature. There may be other universes based on all sorts of other principles, but ours seems to be based on war and games. - William Burrows [1]

The term Real-Time Strategy (RTS) was introduced by BYTE magazine in 1982 [2, pg 3, 124], however it's Brett Speny that is credited with the creation of this term with his game Dune II [3].

Strategy games challenge the player to achieve victory, by planning and using different strategies that are used against one or more opponents [4, pg 419]. This genre distinguishes strategy games from puzzle games that call for planning without the existence of conflict, and from construction and management simulations that doesn't have influence against opponents. Strategy games usually have as a key goal defeating all enemy forces, so the majority of strategy games are military games with some differences between each other. The fiction of these type of games can be of medieval, futuristic times or even the present day. Victory is obtained by superior planning and taking the best possible actions, where the elements of chance or luck are not determinant. Other challenges such as tactical, logistical, economic and exploration challenges can also be present.

This type of games falls in two main sub-genres: Turn-Based Strategy and Real-Time Strategy. In a Turn-Based Strategy players can consider what type of actions they want to do in the game, considering the benefits between each available actions and choosing the most optimal ones during their turns. The player doesn't have to worry about the time that is passing during the consideration of it's actions. Real-Time Strategy Games who were developed after the Turn-based genre, add time to the game for pressuring the players to choose the optimal actions at a fast pace [4, pg 420]. Also since both players actions are happening at the same time, players must be aware of the what their opponents are doing and think of strategies that can counter the opponents actions. For this reason the players must always be aware of what is happening during the game.

1.1 Motivation

The RTS Video Game genre can be played in both personal computers and mobile platforms. In 2015 Entertainment Software Association (ESA) released a study about the state of industry and video game industry in the USA [5], during the 2014 year. This study reports that 62% of the population plays video games on personal computers, 35% on their smartphones and 31% on wireless devices. Also 37.7% of the computer games played were of the Strategy genre. Moreover 54% of the gamers played at least a multiplayer game once per week.

According to the Independent there is at least one mobile device for each living person on planet [6]. These devices have powerful graphical capabilities, that permit these types of devices to run video games. Those games can be installed and downloaded on these devices, by application stores (App Store, Google Play) or be played in a mobile browser.

The market of the mobile devices games were initially dominated by casual games (games that aren't very challenging in difficulty). However with the hardware evolution of these devices and the increase in complexity and graphics of the games developed in them, the players that were playing these games on mobile devices became more experiencing in gaming. As result they wanted to play other type of games that were more challenging and competitive, and if possible play those games with their friends or other people.

1.2 Objectives

The main goal present in this thesis is the creation of a RTS game, using HTML5 and JavaScript languages, that is capable of being cross-platform (working in personal computers and mobile devices). The game must contain the traditional 4X elements present in this game genre ("eXplore, eXpand, eXploit and eXterminate") and being able to be playable in multiplayer mode.

1.3 Contributions

To be able to accomplish the main goal of this thesis User Tests were effectuated, with a chosen **Focus Group** made of people with heterogeneous characteristics. These tests were performed to observe the game experience that is being proportionate for the participants of these type of tests, with the goal to improve it and check if the desired objectives purposed at the start of this project were accomplished. Three methods were used to gather the results made by those tests such as **Observation**, **Questionnaire** and **Server Text Logs**. Those methods would be applied in two sets of test experiments, corresponding to the Alpha and Beta prototype made for this thesis project.

1.4 Structure of this document

This document is divided in three main parts: The first one describes the **Related Work** of this project, being divided on three sections: **RTS game elements**, **RTS History and Examples Technologies Used**. Those are referent to chapter 2 of this document.

The RTS game elements section describes the main elements present in the majority of RTS games such as Theme, Units, Structures, Resources, Fog of War, Player Actions, Interface, Concept Map, 4X elements and Player archetype. The RTS History and Examples section refers some games that are part of the history of the RTS games, describing the characteristics of each one.

The Technologies used section informs the technologies that were researched and used in this game such as HTML5 and JavaScript languages. There is also a presentation of the chosen HTML5 game framework for the development of this project called *Phaser.io*, being referred the main features of it and how it will benefit the development of this project. Then a comparison between this framework and other game frameworks is shown, to observe the advantages and disadvantages between them and why *Phaser.io* was the preferred choice.

The second part of this document is about the Solution of this project on creating and designing a RTS game. This part is composed on 4 sections: the **Game Design** and **Game Architecture** sections of chapter 3, the **Implementation** section on chapter 4 and the **User Tests** written on chapter 5.

The first section of chapter 3 consists on the Game Design applied to this project, explaining the elements that are part of it such as the objectives and features of this game. There is also a discussion about the elements that are found in-game such as: Units, Structures, Monsters and others.

The second section of chapter 3 consists on the Game Architecture that is part of this project, describing the technical limitations that this project has. Then there is a reference on the functionality of the Game Server and the game framework used, explaining how they contributed to the development of this project. A File Structure is also presented, being referent to the project's organization. There is also an explanation of the role that each file on that structure has, and how it contributes to the project functionality.

The chapter 4 of this document presents some of the Implementation tasks that were part of this project. This chapter starts indicating the development process that was enfolded in this project, by listing some of the most important tasks that were part of it. Then there is the indication of some tutorials that were completed, and how they helped to being more familiarized with the chosen game's framework and with making a RTS game. Following that indication there is a description on some of the most complex implementations of game events that were made during the project's development. The chapter ends with a list of some problems that occurred during the game development, discussing the causes and consequences that they had for this project.

The chapter 5 presents the Project results made by the User Tests effectuated. This chapter refers the experiments made to observe if the project attained the initial objectives purposed. The results of those experiments helped to determine which design decisions and game functions could be improved.

The last chapter corresponds to the Conclusion of the work effectuated since the beginning of the project, with a reflection of the tasks performed on the project's development that were elaborated on the other chapters written in this document. There is also a discussion on what could be the tasks that could be applied in the future for this project.

2

Related Work

Contents

This chapter starts to refer the game elements that are present in the majority of the RTS games.

2.1 RTS game elements

2.1.1 Theme

The theme of a strategy game is usually derived from the primary activity that takes place on it. While there can be differences among RTS games, most of them fall into different categories of games of conquest, trade or exploration. Although in most cases all those three categories are mixed together in a certain quantity.

The majority of RTS games are usually games of military conquest. The main objective of those games is to generate a military force to defeat the opponent, often by eliminating their troops and destroy their structures. Exploration and trade are also relevant to this goal. Exploration is conducted to gain information about the surroundings, in order to plan how to attack or defend against the opponent forces or to search for available resources. Trade is often the conversion of resources into units, research technology or conversion on other resources.

2.1.2 Units

Since the player acts as a military commander of a faction or nation in RTS games, he must deliver orders to his troops for being able to build an organized army to defeat his opponent. Those troops are called units. There are various types of units in a RTS game, and some of them are not specialized in combat. The basic archetypes of units present in a typical RTS game are [7]:

- **Worker unit**: A basic and economical unit that is dedicated to gather resources and construct new structures.
- **Melee unit**: The most basic military type of unit available that possesses a melee weapon and some armour. This type of unit is very common in medieval settings, but not much in present day or futuristic settings.
- **Ranged unit**: Another basic military unit that is capable of fighting enemies from afar, by using a ranged weapon such as a bow, a gun or a plasma weapon depending on the setting the game is insert.
- **Siege unit**: Most seen in present/futuristic settings, this unit has slow movement across the map and has expensive production costs. However it possesses high firepower and defence.

- **Mounted unit**: Most seen in medieval settings, this unit has high movement speed on the map. One example of a type of this unit is a cavalier in a horse. These types of units have better attack and defence than the basic military units.
- **Artillery unit**: Long-range equivalent to the siege unit, the artillery unit can't hit anything that is close to it and is usually slow and poor armoured. To compensate those drawbacks this unit can inflict damage in an area, instead of being limited to damage one enemy. They have long range and high attack power and are effective in destroying enemy structures.
- **Naval unit**: A type of unit that usually can only travel by the sea. There are various subtypes of this unit: it can be a boat who fishes resources only available in certain areas on the map or a Galley who can fire cannon balls.
- **Air unit**: A type of unit that usually can only travel in the air. There are various subtypes of this unit: it can be a helicopter whose only function is to transport a set of limited ground moving units, to another location on the map. Another possible type is a fighter plane, who is very proficient at attacking enemy air units.

2.1.3 Structures

A crucial part of all RTS games are structures. They are needed for the player being able to grow in military power. Those structures are capable to produce units, store collectible resources or research new technology. The set of structures that a player has in one location is usually defined as his military base. The basic archetypes for this base are [7]:

- **Command/Construction Center**: Either constructs structures by itself or produces worker units for it. Sometimes this structure offers upgrades and researches to other structures or abilities.
- **Resource Structure**: This structure stores resources collected by the worker units. The Command Center can also act as this kind of structure.
- **House**: There are various names for this structure in the different RTS games, but the purpose that it offers is generally the same. The House structure increases the number of overall units (Arbitrary Headcount Limit) or generally called population limit, that can be on the map. Some units have different population costs to stay on a map. For instance a worker unit generally increments the population count to one unity, while a siege unity may increase to two or more population units.
- **Barracks**: A structure that is responsible to produce basic infantry units and sometimes researches infantry unit-specific upgrades.

- **Tech-structure**: A structure whose main concern is to advance the Tech Tree and upgrade the weapons and armor units.
- **The Stable/Vehicle Factory**: A structure that can produce siege and artillery units and sometimes upgrades those units.
- **Naval structure**: A structure that can produce naval units and repair them. Sometimes it can also research naval unit-specific upgrades.
- **Aircraft Factory**: A structure that builds, upgrades and sometimes rearms air units.
- **Defence Turrets**: A small structure with a mounted weapon. The type of that weapon can be part of an anti-infantry, anti-armor or anti-air structure.

2.1.4 Resources

To be able to construct structures or produce units the player has to spent the currency of the RTS games, which is obtained by harvesting resources around the map. The type of resources that can be obtained depends of the setting that the game is inserted in. If the game takes part on medieval times the resources can be gold, wood, stone or iron while if takes part on a futuristic time it can be minerals or gas. Also the quantity of the type of resources that exists in the game is a design choice by the game's developers.

The resources are usually gathered by worker units and as the military units evolve in their strength, the quantity of resources to produce them will be more expensive than the previous units. Another use for the resources for a RTS game is a *tech-tree*. The *tech-tree* or technology tree is a tool developers use to control the pace of the game, by limiting and rationalizing the spread of technology [4, pg 438]. The *tech-tree* enables players to obtain more powerful units or technologies that can increase their offensive and defensive power.

2.1.5 Fog of War

In a RTS game the game scenario is presented in a map that reflects the world of the game. At the start of a RTS game the majority of the map is covered in black clouds, and the only terrain visible to the player is where the units and structures are standing. These black clouds are called Fog of War. This element limits the vision that the player has in the map. As the player moves units or builds structures into the hidden areas of the map, those areas become revealed. If the player's units or structures are eliminated in a portion of the map that area will be covered in fog of war. If an area of the game map is covered by those black clouds, the player will not be able to see any kind of actions into it, but the layout of the terrain remains visible.

It's useful for the player at the beginning of the game to send an unit to travel trough the map. The purpose of this action is for searching the military base location of the enemy, and locations with additional resources so the player can expand it's economy.

2.1.6 Player Actions

As referred before, the role of the player in a RTS game is commanding his units to lead his nation to victory over other nations. To do so the player has to give orders to his units and structures, which can be called Player Actions. An order is typically delivered by two-step process: First the player selects an unit or a structure that will receive the order; Then the player issues the order, by clicking somewhere in the landscape, enemy unit or a button with a possible action that the unit or a structure can do.

There are numerous actions that are different from unit to unit or structure to structure. For example a worker unit is the only type of unit who can harvest resources, but it doesn't possess abilities that special military units have. However there are a group of actions that almost any unit has, such as moving from one location to another, attack an enemy, stop moving or patrol an area. The actions for structures are usually producing units or developing new technologies.

2.1.7 Interface

The interface for a RTS game is how the information from the game is presented to the player, in a way to make them understand and react for what is happening in it. Often in RTS games the user needs to keep track of various types of information at the same time. The easiest approach of breaking that information is with the use of windows and buttons. Each window keeps track of different information and allows the user to take different actions, by clicking on the corresponding buttons for them. Most RTS games have three basic windows to display the game general information:

1. The **game window** is where all the terrain units and structures are displayed. This window allows the player to view the events of the game and give orders to their units and structures. This window usually shows just a portion of the game world. The player resources are presented in the top border of this window.

2. The **status window** that allows the player to view additional information and actions that an unit or a structure have. Also information such as health points, offensive and defensive power of units or structures are represented in this window.

3. The **mini-map** is a small representation of the entire playing field or map. Usually the mini-map is a direct top-down of the entire game world that reminiscences the two-dimensional maps. In this map the units and structures are represented in small dots and squares. The player can click on a

location of that map, for the game window to quickly display the playing field that on the location it was clicked.

Bellow is an image of a game interface that is part of a well known RTS game, which is called *Starcraft 2*[1]. The **Square 1** represents the game window. The **Square 2** has the player resources which are *minerals*, *vespene gas* and unit population by order. On the **Square 3** is located the mini-map. On the **Square 4** there is the information about a selected unit or structure which can include health points, attack and defence points. Finally on the **Square 5** there are the possible actions available for the unit or structure selected. The end part of the right side of the image is covered by a Fog of War, since it was not explored yet by the player.

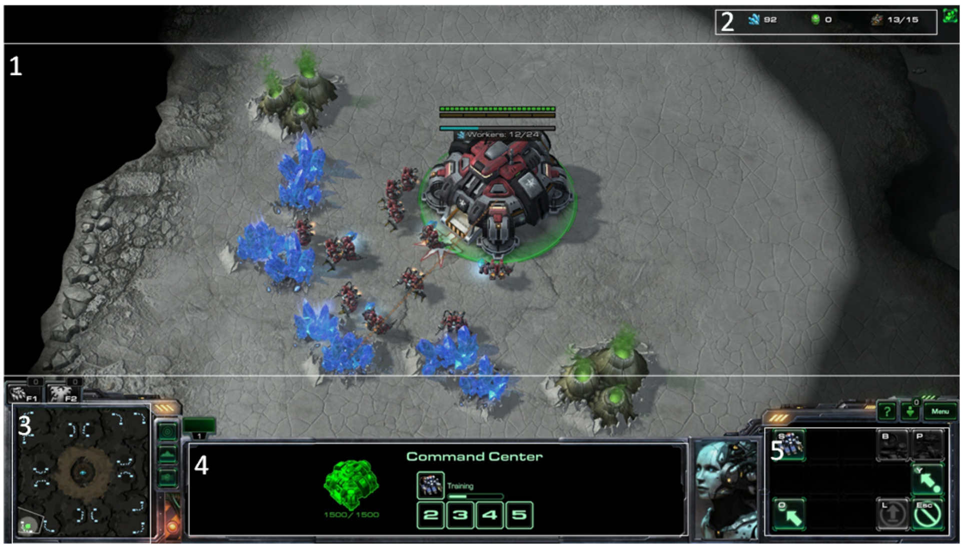


Figure 2.1: Starcraft 2 game interface

2.1.8 4X

4X is a genre of strategy-based video and board games, which players control an army and "eXplore, eXpand, eXploit, and eXterminate" with it. The term was first used by Alan Emrich in September 1993 with his preview of Master of Orion for Computer Gaming World. [8, pg 92-93]

4X computer games are noted for their complex gameplay. The player starts with poor resources and structures, with the goal to establish an empire that can fight against other nations. Elements such as economy or technology development also place a great deal to achieve victory. These games can take a long time to finish since the amount of micromanagement needed to develop a military army increases as the game flows. The phases of this game usually overlap with each other in different ways and weights, depending on how the game is designed.

[1] `https://starcraft2.com/en-us/`

The meaning of the 4x elements of this genre are: *Explore* means that players send units across a map to reveal surrounding territories and enemies. *Expand* means that players conquer new territory by creating new structures on it or sometimes by extending the influence of existing structures. *Exploit* means that players extract and use resources to improve their nation. *Exterminate* means attacking and eliminating other players, mostly trough military battles.

RTS generic diagram

Bellow is a generic RTS diagram that can be used to see how the majority of the RTS games are composed, with the information that was referred above.

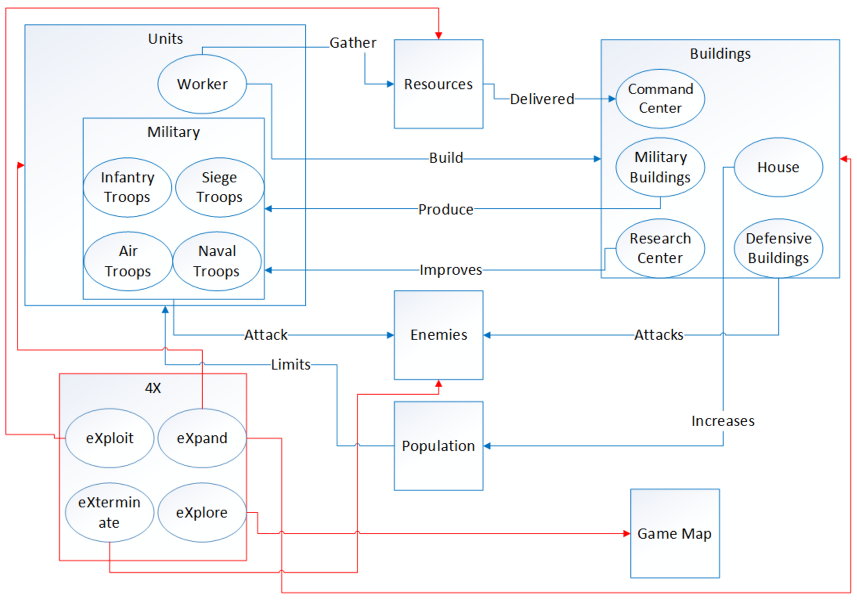


Figure 2.2: Generic RTS diagram

2.1.9 Player archetypes

Since the video game players have different necessities, preferences and expectations towards games, it's important to define the archetypes of players that are more interested in the RTS game style. To refer whose player archetypes are more common in this type of games, one model that can be used is BRAINHEX model [9, pg 102]. This model defines seven types of players associated with different styles of playing that stimulates the brain, and consequently generates different types of pleasure. The BRAINHEX player types associated to the RTS genre are:

1. **Conquerer**: A player of this type enjoys winning and defeating very difficult enemies. The harder the challenges he faces, the more the player enjoys obtaining victory. The dominant activity is defeating enemies. This player type is related to the RTS genre because players compete against each other to achieve victory, by destroying the opponent's army. Also there are players who are

more difficult to defeat than others, based on their game experience in this genre. When these type of players are defeated the victor feels more satisfied about it, than defeating a player with less game experience.

2. **Mentor**: A player of this type enjoys finding solutions to problematic situations. The player enjoys even further the more complex the situation is and the possibility of defining and experimenting different strategies. The dominant activity of this type of player is to solve problems. This type of player is related to the RTS genre, because the players have to plan and execute different strategies to defeat different types of players, such as aggressive or defensive players. They also have to perform those strategies faster and better than their opponents.

2.2 RTS History and Examples

This section descibes some of the most influential RTS that exist, with some of them being taken as a influence to the development of this project.

2.2.1 Stonkers

Stonkers is one of the first RTS games to be released. This game is playable on the ZX Spectrum 48K platform and the publisher was Imagine Software that released it in 1983. The game designer and programmer was John Gibson, and the graphics designer was Paul Lindale[2].

Stonkers is a single-player game that can be played using keyboard or joystick. The role of the game is to control sixteen military units, against the same quantity of units from the computer enemy side. There are four types of army types at the player disposal: infantry, artillery, tank, and supply-truck units [10]. These units are chosen to be part of the sixteen player's army. Combat units consume supplies over time, so the player has to use supply-units to replenish them. Information about the events in the game is displayed in a ticker tape on the bottom of the screen. The game had serious bugs that made the game crash. Even so it was awarded the title ”Best Wargame” by CRASH in 1984 [11], with one of the reviewers saying: ‘The game appealed to me much more than most of the other wargames due to its higher quality of graphics, large scale and simple controls’.

2.2.2 Dune II

Dune II was released in December 1992, being developed by Westwood Studios and released by Virgin Interactive. This game was produced by Brett Sperry, being on the science fiction novel by Frank Herbert that also has the same name.

[2]`http://www.worldofspectrum.org/infoseekid.cgi?id=0004913`

In this game players would choose one of the three races available and each of them had their advantages and disadvantages. The players would construct a base that allow them to deploy a harvester. This unit can gather spice, being the only resource that was present in the game [12, pg 141]. By gathering this resource, players could construct other structures that would allow the production of military units. These units would be used to attack and defeat the opponent by destroying all of their structures and military forces, or to defend one's own structures from destruction to avoid defeat.

Though not every feature was unique and not being the first real-time strategy game to be released, the combination of a Fog of War in the map, managing a military force through mouse clicking, an economic model of resource-gathering and base-building would be used as a template in many RTS successors [13].

2.2.3 Warcraft: Orcs & Humans

Blizzard Entertainment released *Warcraft: Orcs and Humans*[3], produced by Bill Roper and Patrick Wyatt. This game is inserted into a fantasy setting, having 2 available races to be played: Humans and Orcs. *Warcraft* had various features comparing to the RTS predecessor games, such as two available resources to collect (gold and wood) instead of one and the map has wild monsters that can be encountered that may harm the player troops [14]. However some of those monsters can be used as part of the player's troops, by summoning them with certain units spells [15, pg 31]. The design of the military units was also different from other games, containing melee and ranged infantry units, and spellcasters with consumable magical abilities.

This game also had a campaign mode where the objectives to complete a mission, were not restricted to destroying the enemy troops and base. Those objectives could be rescuing friendly units from a enemy camp, rescuing and rebuilding besieged towns or kill the Orc's chief daughter. *Warcraft: Orcs and Humans* also allowed for two players to compete in multiplayer contests, by modem or local networks.

2.2.4 Command & Conquer

Westwood Studios released *Command & Conquer* in 1995, properly known has *Tiberian Dawn* [16]. It was created by Joseph Bostic and Brett W. Sperry. Set in an alternate history of modern day, the game tells the story of a World War between two factions: The Global Defence Initiative of the United Nations, and Brotherhood of Nod which is a cult-like militant organization.

The factions of this game have differences in their strengths and weaknesses. The Global Defence Initiative had stronger units but they were expensive to produce. While the Brotherhood of Nod had cheaper and faster units but they were weaker comparing to the other race. These differences leads the

[3]http://us.blizzard.com/en-us/games/legacy/

player to adapt different strategies when playing with each nation [17, pg 352–354]. The differences in the playability between the two factions added to the appeal of *Command & Conquer*.

It was the first RTS game that introduced certain features that were followed in the next games of this series. The features included full motion video cutscenes with a notable ensemble cast to progress the story, as opposed to digitally in-game rendered cutscenes [18]. Another feature is the inclusion of a side bar at the right side of the screen, where the player could build structures and produce units without clicking in any unit or structure to build them. The majority of the RTS games didn't had this feature, where those orders could only be make clicking on the units or structures, and selecting them at the bottom-right of the screen.

2.3 Technologies used

As it was seen in chapter one, the majority of RTS games were released only to the computer platform. This can be explained due to a couple of reasons, such as the computer mouse being a vital element in a RTS game, because the actions that the player can do. Some of those actions such as moving a unit to a specified location or a structure being constructed, are done by clicking and moving the computer mouse. These actions with the use of the mouse hardware are very efficient time-wise, which benefits the speed of the actions that a player can do in a RTS game.

Another important reason is the presence of a keyboard, which can further increase the speed of the actions that a player can order to it's nation during the game. One example is using the selecting key shortcuts that are designated to specific keys on that hardware to apply those actions (for instance using the 'C' button to order a selected worker unit to build a command center). The RTS games are less played in other game devices, because their game controllers or touch inputs cannot perform game actions with the speed and precision of having a mouse and keyboard. Nowadays RTS games can also be played in other gaming platforms such as mobile devices. This section starts by referring those type of devices.

2.3.1 Mobile platforms

In the past few years mobile devices such as smartphones and tablets have become more popular, and users could use some of the features in those devices that they were capable of doing them in personal computers. Some of those features include seeing videos, playing games or search content in the Internet. Mobile browsers can access the Internet via a cellular telephone service provider or via wireless network.

To access to the Internet mobile devices use mobile browsers. A mobile browser is similar to other web browsers, that work on personal computers. The main difference is that they are designed for use on a mobile device such as a mobile phone, tablet or PDA. Mobile browsers are optimized to display web content more effectively to small screens and to enable touch events. Mobile browser software must be small and efficient, to accommodate the low memory capacity and low-bandwidth of wireless handheld devices [19].

For this thesis since the main goal present in it was to create a RTS game that is cross-platform, working for both personal computers and mobile devices, it is important to explain this concept and know what are the advantages and disadvantages of it. Cross-platform (or sometimes called multi-platform) when related to the field of computer software, refers to the interoperability between different computing platforms (or operating systems). For the advantages and disadvantages, there are [20] [21]:

Advantages:

- **Single unified codebase**: By having a single and unified codebase it reduces code maintenance tasks, such as version and branching strategy and the number of source repositories that are needed to administer. The time taken to debug, fix, test and deploy defects is also reduced.
- **Greater Reach**: The greater the number of platforms it can target, the greater the potential audience and therefore the number of potential customers. Also players can play a cross-platform game together with different devices.
- **Cost Effectiveness**: Since the game works on multiple devices, it is not needed to develop each game application to each device, saving time and money.
- **Development tools**: There are a great deal of cross-platform game development devices, and engines which are easy to learn and to use. A cross-platform game development engine also assists to render good efficiency and high quality animation and graphics, thus developing an effective gaming environment.

Disadvantages:

- **Performance**: The game performance must be optimal in both devices which may be difficult to approach it, because the hardware on both devices is not the same.
- **Device resolution**: Personal computers and mobile devices have different screen resolution devices. Even mobile devices have it different on their own.
- **In-game Commands**: Each device has different approaches to input commands in a game. Personal computers use the mouse and keyboard, while mobile devices uses touch inputs.
- **Connection Speed**: The speed of the web service in mobile devices is slower than the personal computers, meaning that web content may be slower to get in the mobile devices.

2.3.2 HTML

HyperText Markup Language (HTML) is a computer language that allows the creation of websites. It was developed by World Wide Web Consortium (W3C)[4] and Web Hypertext Application Technology Working Group (WHATWG)[5], with the first version being released in 1993. Along with Cascading Style Sheets (CSS) and JavaScript, HTML is part of a triad of technologies that web developers use to create web content [22]. A markup language is a type of language that annotates text that can be manipulated by a computer, and is written in a way to distinguish it from other text [23]. The meaning of ”marking up” is from paper manuscripts, that were traditionally written with a blue pencil on authors manuscripts. HTML has pre-defined presentation semantics, that specifies how HTML language is represented in a way that can be readable and understandable by humans.

HTML documents are made with HTML elements. These elements are composed with a start tag and end tag, that are written with angle brackets and the context between them. With the use of these tags it's possible to manipulate various elements that can be part of a web page, such as a title, a header, a paragraph, an image, a video and many others [24]. Each tag has an appropriate name to it's definition. For instance the tag <button>defines a button where the user can click on it.

HTML can embed scripts written in languages, one of them being JavaScript, that performs functions on web pages that HTML alone can't do, such as adding a mouse over event or adding dynamic behaviours on a script [25]. HTML markup can also refer the browser to describe the look and layout of text and other material that is rendered on the screen, such as the color, font and the size of a text [26].

The latest HTML version was the fifth major version of the hypertext markup language specification called HTML5, and it was released on 28 October 2014 by W3C [27]. The core aims of HTML5 were to improve the language with support for the latest multimedia, keeping it easily readable by humans and understood by computers and devices. HTML5 also defines a HTML syntax that is compatible with HTML4 and eXtensible Hypertext Markup Language (XTML) version 1 documents that are published on the web [28].

HTML5 contains many differences from his predecessor, one of them was the inclusion of new HTML elements such as video and audio for multimedia content. It also introduced Application Program Interfaces (APIs) that are useful in creating web pages, and can be used together with the new elements in this version. There are APIs that prompt the user such as *alert()*, *confirm()* and *protomp()*; for printing the document with *print()* and many others. Many features of HTML5 had in consideration at being able to run in mobile platforms such as smartphones or tablets, which makes a potential candidate for cross-platform mobile applications.

[4] https://www.w3.org/
[5] https://whatwg.org/

2.3.3 JavaScript

JavaScript is a dynamic and interpreted programming language, that is commonly used to make web pages interactive [22, pg 1]. It first appeared in May 23 1995 developed by Brendam Eich, and it has been standardized in the ECMAScript language specification [22, pg 2].

This language is commonly used with HTML to add client-side behavior to web pages a.k.a Dynamic HyperText Markup Language (DHTML). With the help of the Document Object Model (DOM) [29] it's possible to add interactive events to a web page, linking scripts to a HTML page in the <script>tag. Web browsers usually create "host objects" to represent this kind of model, that can be manipulated to dynamically generate web pages. Some of the features that JavaScript offers to web pages are [30] [25]: Different types of mouse effects that can be triggered, like a mouse click or hovering a picture. Javascript is able to play audio or video content. This language can work on web pages, even when they are offline; This language can load content in a web page, without reloading the entire page.Finally JavaScript can change HTML styles (CSS).

Since JavaScript code can run locally in a user's browser, the browser can respond to user actions quickly, making an application more responsive. Also as it's one of the few languages that most popular browsers share support for [31], JavaScript has become a target language for many frameworks available for those browsers [32].

2.3.4 Advantages of using HTML5 to make games

One of the main advantages of making games in HTML5, is that the player doesn't have to install any kind of additional software or applications to play them. There is only the need of a web browser that support this language and all the modern ones already do [33]. Also it's possible to create games that can be playable in more that one device, meaning that they are cross-platform games. The graphics, sounds and animations of those games work in the same way for any compatible platform.

Another advantage of using HTML5 is that there are many JavaScript game engines that can be used with HTML, which game developers can use. These engines makes developing 2D games easier and allows developers to use advance multimedia elements of HTML5 in their games.

Other advantage is that it's not needed to separate the code for a different platform. It's possible to calculate the size of the game window in HTML5, by calculating the screen dimensions of the device that is accessing the browsers that contains the game. Also HTML5 can make any update in a program and those modifications are automatically configured in all the particular configurations online. This is useful because when the player wants to access the game another time, it doesn't have to install or download any kind of plug-in. This behaviour leads to not decrease his motivation for playing the game and not needing to have adding extensions or downloading plug-ins. This was the main reason of the downfall of the Flash Player based games, because when the player attempted to play games using this technology, the player needed to have plug-ins installed. Otherwise he would have to install them to play Flash based games.

Another feature in HTML5, is that it can be operational offline. With this feature some HTML games can be played offline, without web connection. This can be done trough the *cache manifest*.

2.3.5 HTML5 Game Framework

A HTML5 game framework is the solution that was approached, to be able to develop a cross-platform game in this language. The framework can make a game support cross-platform issues, solving the screen resolution, provide multiplayer options, load sprites, sounds, animations and many other useful features to build a game in HTML5. For solving the problems that making a cross-platform RTS game had, various HTML5 game engines were researched to see which one would be best of use in this project. After seeing the features of each framework and testing some available demos of them, the framework that was considered to be more useful for this project was *Phaser.io* or simply called *Phaser*[6].

Phaser is a open source HTML5 game framework. It uses a custom build of *Pixi.js*[7] for WebGL and Canvas rendering across desktop and mobile web browsers. Games can be compiled to iOS, Android and desktop apps via 3rd party tools like Cocoon, Cordova and Electron [34]. This framework is compatible with any browser that supports Canvas[8] or webGL[9]. Some of the features that this framework offers are:

- **WebGL & Canvas**: *Phaser* uses both a Canvas and WebGL render internally and can swap between them based on browser support. This allows for faster rendering across Desktop and Mobile.
- **Sprites**: *Phaser* can use sprites in-game and apply various functions with them, such as position, rotation, scale, animation, collision, click and drag events or paint them.
- **Input**: With the call *Phaser.Pointer*, this function recognizes a mouse click or a screen touch in the same way. And there are other mouse, keyboard or multi-touch functions available.
- **Device Scaling**: *Phaser* has a built-in Scale Manager which allows to scale any game to fit any size of the screen, being possible to control aspect ratios, minimum and maximum scales.
- **Mobile Browser**: *Phaser* was built specifically for Mobile web browsers, while working on desktops as well.

[6]http://www.phaser.io
[7]http://www.pixijs.com/
[8]https://caniuse.com/#feat=canvas
[9]https://caniuse.com/#feat=webgl

The main reasons that *Phaser* was the chosen framework for this project were the numerous examples available [35] for it, and the active and helpful community present in it [36], that were greater than any other framework that was researched. *Phaser* is also one of the most game frameworks stared in GitHub [37]. Furthermore the feature of being specially designed for working in mobile devices, was important for the game of this project being able to run with a good performance in mobile devices. Bellow is a comparison table with some of the HTML game frameworks that were researched, with every framework being available for free and supporting cross-platform gaming:

Table 2.1: HTML Game Frameworks comparison

Engine	Rendering	Physics	Animation models	Special Features
Construct2	WebGl or canvas	WebGl, Box2D	Keyframing	Scene editor, debugger. A* pathfinding
Impact.js	WebGl, canvas or DOM	CWebGl, Box2D	Keyframing, CSS	Level editor, debugger
Phaser.io	WebGl or canvas	Arcade, Ninja, P2	Keyframing, CS6/CC, Flash, texture packer	Particle System
Pixi.js	WebGl or canvas	Physics.JS	Keyframing	Blending, bitmaps, sprite sheet
Quintus	DOM or canvas	2D module	Keyframing, CSS	Component model
Crafty.js	DOM or canvas	Gravity component	Keyframes, tweenings	Sprite map

2.4 Discussion

As it was seen in this section RTS have a lot of game elements that are different from other games. Also those elements can change between games of this genre, such as the Theme, Units, Structures, Interface and others. This was proved by observing the games present on the *RTS History and Examples* section. The features present in these games influenced some of the features present in this project, which will be referred in the Game Design section. It could also be observed, that mobile platforms are a potential candidate for being used for a cross-platform game with personal computers, and there are great advantages in developing a cross-platform game. The approach used for this problem was to develop a game in the web, where both platforms can run it in a web browser. For that HTML and JavaScript languages were used to develop the project's game.

To solve the differences that mobile platforms have compared to the computer platforms, such as the screen resolution of the game or the usage of touch events instead of a mouse and keyboard, an HTML framework is used which is *Phaser*. This framework was specifically made for developing games in mobile platforms, while being possible to develop in computer games as well with the same code. *Phaser* also has a substantial amount of resources that a web programmer can use to help the development of the game. The following chapter will relate the Game Design that was conceived in *Planetary Conquest* and the Game Architecture of it as well.

3 Game Design and Architecture

Contents

In this chapter there are two main components of this project that are going to be referred. The **Game Design** and the **Game Architecture** that is part of *Planetary Conquest*.

3.1 Game Design

After researching many RTS games and HTML5 game frameworks, the game designed and implemented for this thesis is named *Planetary Conquest*. Some of the elements that belong to this game such as existing Structures and Units detailed attribute data, and an extensive explanation about the Technical Design which was made for some of the elements discussed in this section are referred in the Annexes section of this document. Game Design is the process of coordinating the evolution of the design of a game [38] [39]. The goal of it is to create a gaming experience to the users, working on the requirements to achieve the game objectives. In this section is written a list of every requirement that is part of the game design and explained in detail.

This section starts with the explanation of the existing **Races and Objectives** present in *Planetary Conquest*, then refers the visual **Perspective** of the game elements shown to the Players. The explanation of the *Planetary Conquest* **Features**, **Units**, **Structures**, **Monsters**, **Upgrades** and **Super Powers** that are part of the game, is also presented.

Following the explanation of those elements, there is a discussion on how the **4X** elements common to a RTS game, are present in *Planetary Conquest*. Afterwards, the game **Progression** that occurs during a game match of *Planetary Conquest* is written. Then a figure is presented, being the **Concept Map** of *Planetary Conquest* containing each element of the game referred in the previous subsections of this chapter, and showing the relations between each element. Finally another figure is presented, being the **Interface** that is shown to the players when playing the game, indicating every aspect that is part of it.

3.1.1 Races and Objectives

There are two available races to be played: the **Humans** and the **Orghz** who are vile green creatures. The main objective of this game is to destroy every unit and structure that the enemy race disposes. By doing that the winning race conquers the planet.

3.1.2 Perspective

The game window of this RTS game is in a top down view with 2.5D perspective or called isometric projection. With this type of projection, it is possible to represent shapes close to a three-dimensions group.

However since only a side of a game object is shown per time frame it is referred as 2.5 perspective, since it is an illusion of three dimensions depth [40].

For being able to have isometric projection the shapes to be seen in isometric view, are placed on a three-dimensional space. The X and Z axis are inclined to the horizontal plane at an angle of 30 degrees, while the Y axis remains stationary and perpendicular to the plane [41].

In the game screen it is shown various game objects, such as Units, Structures, Monsters, mountains, waters, and others placed on the game map. However only a portion of the entirety of the game objects is shown on the game window. To access to other sections of the map, the player has to drag the game cursor to a position it chooses to see it or the player can click on a position of the mini-map, which is a small representation of the entire game map.

For the game art used in this project, since it was very difficult to find isometric sprites within the theme of SCI-FI or space, and it was quite demanding for drawing each one of them, it was used for this project a collection of spritesheets belonging to a game of the *Warcraft* universe, called *Warcraft II: Tides of Darkness*[1]. That game has 2 races: the **Orcs** and the **Humans**, which correspond to the **Orghz** and the **Human** races of this project respectively. The spritesheets were found on the website of the group **The Sprite Resource** [42]. The methods used for those assets to be placed and animated in the game are going to be explained in the **Implementation** chapter.

3.1.3 Features

While at it's basis *Planetary Conquest* works like a typical RTS game, there are some differences comparing to the majority of other games of this type such as:

- **Monster encampments**: Symmetrical spread across the map, there are different types of monster groups which can be interacted with the players. There are three available options for interacting with them such as **Trade**, **Recruit** or **Pillage**.
- **Unique Upgrades**: Each type of military unit has an unique upgrade assigned to it, which can reflect it's race behaviour. Also there is an unique upgrade for each race and for it's Tower structure. Every single unique upgrade is different from each other.
- **Hydroxygen resource**: Of the 3 resources present in this game, Hydroxygen has a completely different usage between each race. Orghz use it for the production and upgrades of units or structures. Humans use Hydrorxygen to breath, consuming this resource every 6 seconds of playing time.
- **Ancient Resource Deposits**: While there are resource deposits where the players can extract resources from them, that are symmetrical spread across the map, there is one special resource

[1]http://us.blizzard.com/en-us/games/legacy/

deposit for each resource. Those resource deposits are situated in the middle of the map, and when extracted they give to the player twice the amount of resources when gathered, as opposed to the normal quantity of every other resource deposit.

- **Power Structure**: Each race also has a Power Structure. The function of this structure is to give a temporary power boost to units which can be reactivated overtime. There are two Super Powers, one for each race whose effects are different from each other.

Some of the games referred in the section of RTS History and Examples, inspired the features present in *Planetary Conquest*.

The **Monster encampments** feature was inspired by a game of the *Warcraft* universe called *Warcraft III: Reign of Chaos*, which also contained monster encampments where monsters could be killed for winning gold or for the player's hero to win experience or items. The **Unique Upgrades** feature was inspired by *Warcraft: Orcs & Humans*. Most of the units present in that game have at least an unique upgrade available for them, with *Planetary Conquest* having also one available for each military unit. The **Ancient Resource Deposits** feature was inspired by a game of the same developers as *Command & Conquer*, which was *Command & Conquer: Red Alert 2*. That game only has one currency which is cash and the player had to extract it from small chunks of gold spread in certain locations of the map. However few zones contained diamonds instead, which each extraction valuing twice the amount of a gold extraction. Finally the **Power Structure** feature was also inspired in *Command & Conquer: Red Alert 2*. After the construction of certain structures in that game, a repeated timer would start. When it ended the structure could activate a phenom when the player wished to, such as teleporting units from one location to another or launching a nuclear missile to a designated location. If the building was destroyed the phenom could not be activated until the player constructed that structure again, resetting the timer to it's initial duration. Furthermore only one type of that structure could be built at the same time, not being possible to activate multiple phenoms at a short pace of time.

3.1.4 Economy

There are 3 types of resource present in *Planetary Conquest*. They are **Crystals**, **Nitrogen Liquid** and **Hydroxygen Gas**. These resources can be extracted from Crystal, Nitrogen and Hydroxygen resource deposits respectively. There are 6 clusters of 3 different types of resource deposits spread symmetrically across the map and one special cluster which contains the **Ancient Resource Deposits**. Each resource deposit holds 10000 quantity of resources of one type, which can be extracted by 10 quantity each time a peon unit makes an extraction from a resource gathering. The Ancient Deposit Resources hold the same quantity of resources, however for each extraction the peon will transport 20 quantity of the resource extracted instead. Also the consumption of the **Hydroxygen Gas** is different from the two races. The **Orghz** spends that resource together with the other two mentioned to produce units and structures, and research new technology. The humans only use the *Crystals* and *Nitrogen* to improve it's military power, with the *Hydroxygen* only being spent on the breathing of the Human units. The latter resource mentioned is spent for this behaviour over a repeating 6 seconds timer, being more spent with the more population player accumulates. If the quantity of that resource is not enough for the breathing of some units, they will take damage equal to a percentage of their maximum health.

3.1.5 Units

The units present in *Planetary Conquest* are the following:

- **Peon unit**: The only non-military unit present in *Planetary Conquest*. While it is the weakest in combat purposes, it is vital to the growth of the player's army. This unit can gather resources from Resource Deposits by extracting them, then they are delivered to the closest Command Center. It can construct any structure available, such as defensive, unit production or research upgrades structures. Every structure can also be repaired, recovering the structure's health points to it's maximum value at the cost of Crystal resources. Only one peon unit can construct a structure at a time, without any sort of help of other peons. However more that one type of this unit can repair the same structure at the same time.
- **Melee infantry unit**: The military unit that the Player can afford for the lowest resource costs, which can attack other units or structures at a short distance. Their names are **Orghz Warrior** and **Human Soldier**.
- **Ranged infantry unit**: The second type of infantry unit that can be produced. This type of unit is slightly weaker in combat stats than the melee infantry, but it can attack from distance. Their names are **Orghz Butcher** and **Human Archer**.
- **Mounted land units**: This type of units are similar to the infantry units, however they are mounted by horses making them faster than the previous units mentioned. Mounted units also have increased combat stats compared to the previous ones. Their names are **Orghz Shadow Knight** which has ranged attack type and **Human Cavalier** which has melee attack type.
- **Flying unit**: The last type of military units which can travel on any map terrain, even if it is occupied by rocks, water, lava or structures. Only units that have ranged attack type or that are also flying units can attack this type of unit. Their names are **Orghz Wyvern** which has melee attack type and **Human Eagle Knight** which has ranged attack type.

The attributes that are part of an unit are:

- **Population value**: The amount of population points an unit will occupy to the total population a player can have. When an unit is produced the player's current population will be incremented by that value.
- **Health Points**: The unit's Health Points represent their life total. When the unit's live points are 0 or lower, the unit will die and the player's current population will be decremented by the unit's population value.
- **Damage**: The amount of damage an unit can inflict to an unit, monster or structure.
- **Armor**: The amount of damage that an unit can reduce from incoming attacks.
- **Range**: The maximum distance which an unit can attack an enemy. Melee units have 1 range while ranged units have 2 range. Each unity of range is equivalent to 50 pixels of distance between the unit and it's attacking target.
- **Speed**: The amount of pixels an unit can travel per second.
- **Movement Type**: If the unit can fly over occupied terrain having flying movement or it walks on unoccupied map terrain having ground movement.
- **Unique, Race, Weapon and Armor Upgrade**: If the game object has the type of that upgrade researched or not.

3.1.6 Structures

The structures present in Planetary Conquest are the following:

- **Command Center**: This structure is the main center of the player's economy growth. It produces peon units and receives resources gathered by them, giving the possibility for the player to improve it's army. This structure also offers the research of two unique upgrades, one being a racial upgrade the other being the Tower structure upgrade.
- **Barracks**: This structure is responsible for the production of infantry units, both melee and ranged. The unique upgrades for these units are also researched here.
- **House**: Structure that increases the total population points a player can have, which is 10 points increased. The player cannot have more that 50 total population points.
- **Tower**: Defensive structure that can attack from a distance other enemy units or structures.
- **Factory**: This structure produces mounted units and offers the research of their unique upgrades.

- **Research Building**: A structure where is only possible the research of upgrades. There are three types: Melee weapon upgrade, Ranged weapon upgrade and Armor weapon upgrade. The first two will only affect units with the respective attack type, while the last one will affect all units. There are three levels for each upgrade that can be researched.

- **Flying Nest**: The structures **Orghz Wyvern Nest** and **Human Eagle Nest** permit the production of flying units and the research of their unique upgrades.

- **Power Structure**: The only structure that cannot produce any unit or research any upgrade. However after a certain period of time, this strcture provides to the selected player's units a temporary power boost. The functionality of this structure will be explained in detail in the subsection **Super Power**.

Every structure contain as attributes **Health Points** and **Armor Points**. The Tower Structure also contains **Damage**, **Range** and **Unique Upgrade**. Each upgrade has the same behaviour as what was written on the unit's attributes section.

3.1.7 Monsters

One of the features of *Planetary Conquest* is the existence of multiple Monster encampments spread across the map. There are 6 encampments on the map, grouped with 3 monsters of the same type. The type of those monsters can be **Ogres** which are a melee infantry unit, **Archers** which are a ranged infantry unit and **Gnomish Airplanes** which are flying ranged units.

To interact with a monster encampment the player must send at least one unit to the location where the monsters are. To do so the player with it's selected units clicks on the monster it wishes to interact, with the unit moving to it and stopping when it's close of the monster encampment. There are 3 options available for the player to choose:

- **Trade**: The player trades a quantity of a requested resource for another type of resource. The quantity and the type of resources to trade are already specified. For example an Ogre encampment requests 200 quantity of Crystals and gives 300 quantity of Hydroxygen. The amount of resources that the player gives and receives is always the same, regardless the encampment monster's type. However the type of each resource for each trade is different from each other.

- **Recruit**: The player pays a certain amount of specified resources to the monsters, for them to give to the player 3 monster recruits of their type to the player's army. The amount of resources requested varies with the military power of each monster. The monsters received by the player are treated as units.

- **Pillage**: If the player opts to choose this option the monsters of that encampment will attack the nearby units of the player. The 3 monsters that are part of that encampment will be enemies of the player, trying to kill it's nearby units. However for each monster killed the player will win a determinate amount of resources that can be seen before choosing this option.

Each singular monster is part of a monster encampment having an hidden monster group ID, referring which encampment the monster is part of. For each encampment the player can only choose one of the 3 available options. Once selected no more options are available for that specified group. Also unless the Pillage option is selected, it is impossible to attack these monster units. Although monster units can be killed it is possible for other players that are not enemies with them, to interact with a monster encampment if at least one of those 3 monster are still alive. The monster attributes include almost every attribute an unit has with the same behaviour. The attributes they don't have are the **Race Upgrade**, **Unique Upgrade** and **Weapon and Armor Upgrade**. The recruited monsters since they will count as units have the previous missing attributes referred.

3.1.8 Upgrades

Has seen earlier there are two types of structures responsible for researching the upgrades of the race, tower structure and units. Those are the structures that produce units and the Research Building. The structures that produce units are responsible for the research of unique upgrades. There are various different purposes for each researched unique upgrade such as reducing the maximum value of the armor points of an enemy unit or monster by one, when the player's unit attacks (Orghz Warrior), increasing the tower's armor and health points (Human Tower) or decreasing the time for constructing structures and researching upgrades (Human race upgrade). The Research Buildings will only upgrade the damage or the armor an unit has. Every upgrade can also affect the monsters recruited by the player. The type of these monsters is reflected on the player's army (melee infantry, ranged infantry, flying unit). For instance if an Orghz player finished the research of the Orghz Warrior (melee infantry) unique upgrade and has Ogre units previously recruited, those will also benefit from that upgrade and other Ogres to be recruited will have the same scenario applied.

3.1.9 Super Power

By constructing the Power Building, the player can activate a temporary power boost to their selected units. When the player decides to construct a Power Building, the other player will be warned that it's opponent is constructing that building. When the construction is finished both players will see a timer representing the time that is remaining, until the player who constructed the Power Building can activate it's Super Power. It takes 4 minutes to activate the Super Power after construction. Only one copy of the Power Building can stay constructed for each player. Afterwards when the Super Power is ready the player can click on the Super Power button, and every unit that is currently being selected will have a temporary power boost during 30 seconds. After those 30 seconds if the Power Building is still constructed and active, the preparation timer will refresh again for 4 minutes.

If the Power Building is destroyed before the Super Power is active the timer will be inactive, and the player cannot activate it again until it reconstructs the Power Building again. However if the Super Power is active, even if the Power Building is destroyed it is still going to be active until the end of it's duration. The 2 different Super Powers for each race are **Brute Force** for Orghz which increases the affected units attack by 3 points and **Indomitable Spirit** for Humans which adds a Shield, to protect those units from enemy attacks. The shield points are 35. More discussion about those Super Powers is found in the **Technical Design** subsection on the annexes of this document.

3.1.10 4X

Planetary Conquest has the 4X elements present on it. The player can *Expand* by constructing new structures. The player can *Exploit* by extracting resource locations or interacting with monsters and it can *Exterminate* by killing enemy troops or monsters present on the map. The player can *Explore* the game map to gather resources in other Resource Deposits or to interact with monster encampments. The location for those interests is already revealed to the players, since there is no **Fog of War**. The reason for not having Fog of War implemented will be explained in the section **What went wrong** of the **Implementation** chapter.

3.1.11 Progression

In this section of the document it's going to be discussed the progression which is a ”Process of developing gradually towards a more advanced state” [43] that takes place on this game. There are two types of progression which are in the view of the game and in the view of the player.

In the **game progression** two players will compete against each other. One will be controlling the Orghz race, the other will be controlling the Human race. A 1 vs 1 game match has a duration about 15 minutes. The game ends when the enemy units and structures are destroyed. In the map there are resources that the player can harvest to improve it's race. There are also structures to produce units, research technology that can increase the strength of those units, defensive structures able to damage enemy units and structures and a Power structure temporary increasing some units military power. Monsters encampments are also present on the map that can be interacted by the player's units, that may help the player to further increase the strength of it's race.

In the **player progression** the players will take some non optimal actions when they are playing for the first time, because of their inexperience in playing the game. As they play further they will gain more knowledge and they will take optimal actions more often. Another way to improve their game knowledge is to compete against stronger opponents, learning their strategies for using or adapting to them in other matches. The main concerns that will be present to the player will be finding monsters to interact, extracting resources from Resource Deposit locations, producing units, constructing structures, researching upgrades and fighting against it's opponent. This actions are done for the player being able to win the game match. As the game progresses the player will start to feel more powerful, as with the gather of resources and construction of structures will permit to research new kinds of technology upgrades and produce new kinds of troops.

3.1.12 Planetary Conquest Concept Map

Bellow there is a concept map of *Planetary Conquest*, based on the generic RTS diagram presented in the Related work section.

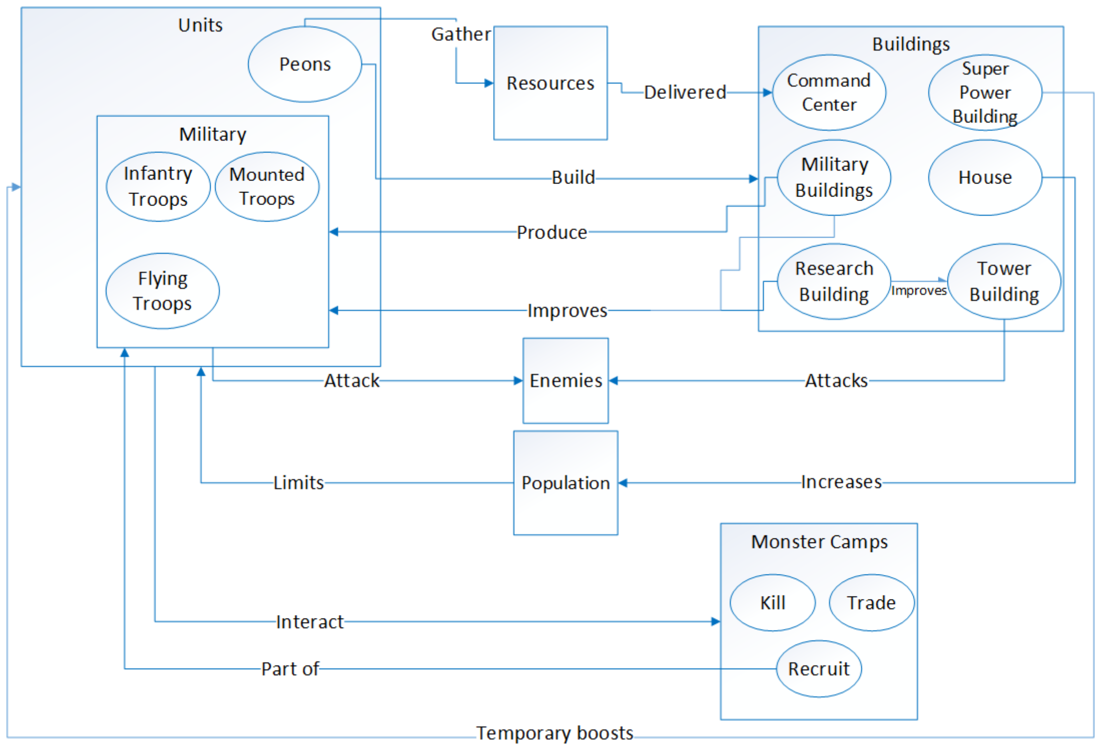
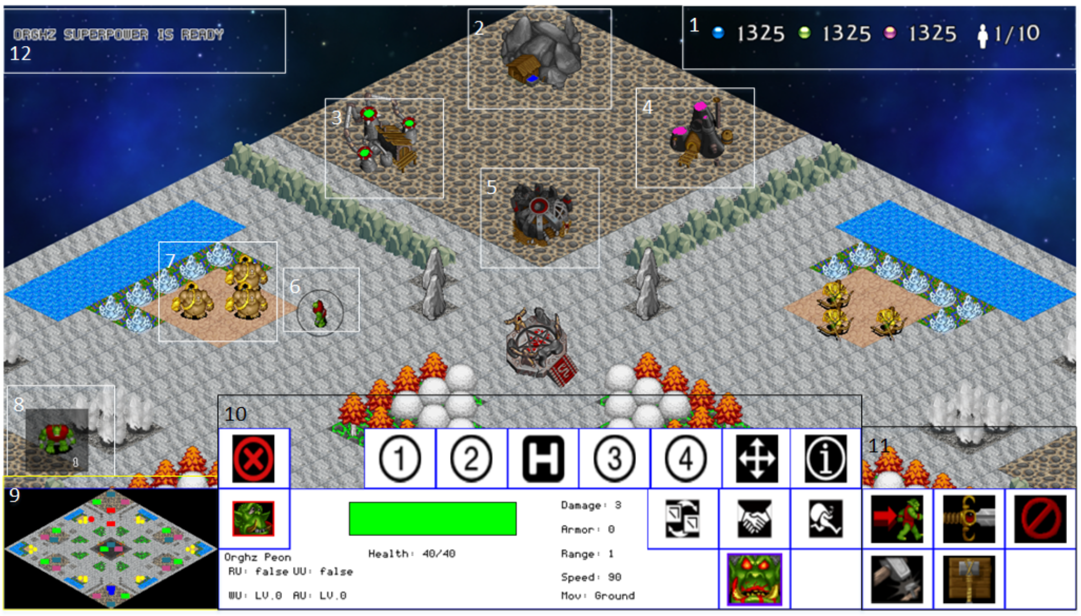


Figure 3.1: Planetary Conquest Concept Map

3.1.13 Interface

Bellow is a figure containing the interface of *Planetary Conquest* for the mobile devices. Referring what each rectangle in underlying: In the **1 Rectangle** is the types of resources used in this game being *Crystals*, *Nitrogen* and *Hydroxygen* by order; with the current and max population being the last icon represented. The **2,3 and 4 Rectangle** represent the resource deposits from the same order of resources described earlier. The **5 Rectangle** shows the Command Center of the Orghz race. The **6 rectangle** shows a Orghz Peon interacting with a group of Ogre monsters represented by the **7 Rectangle**. The **8 Rectangle** shows the amount of peons that are not working or moving for the player's race. The **9 Rectangle** shows the mini-map, which is a small scaled representation of the main game map. The **10 Rectangle** is the Information Bar containing all the attributes that belong to the game object selected, and on the right side it can be seen on the upper bar the monster options (Trade, Recruit and Pillage) to negotiate with the monsters that are being interacted. Bellow of that bar is the button to activate the Super Power of the player's race on all the units selected. The **11 Rectangle** represents the command bar, showing the buttons that permit to order specific commands for the units that are selected by the player. For what can be ordered on those buttons: on three buttons on the upper bar, the first is for moving the unit, the second for attacking an enemy target, and the third for stopping the unit's movement and behaviour. Regarding the second line those buttons are exclusive to peon units. The first is for repairing a structure and the second one to shows the structures that are available to be constructed. The **12 Rectangle** is where it is presented the timer of the Super Powers for the player's race which can be seen by both players, reporting the state of it (timer countdown for activation, ready for activation and timer countdown to end activation). The Computer Interface is identical to the one presented, except for the bar that is right up the information bar. This bar is only usable by mobile devices to conduct certain commands that are difficult to do on these types of devices such as: deselecting game objects selected, choosing a number as an Hotkey to quickly access a game object selected, moving the game camera and enable to see the information regarding the action performed on a button. This types of behaviour can also be done on computer devices, but are more easily to do so with the help of the mouse and the keyboard.



Orghz Peon
RU: false UU: false
WU: LV.0 AU: LV.0
Health: 40/40
Damage: 3
Armor: 0
Range: 1
Speed: 90
Mov: Ground

Figure 3.2: Planetary Conquest Interface for Computer platforms

3.2 Architecture Design

The Architecture Design of *Planetary Conquest* is discussed in this section. The **Technical Limitations and Requirements** of this project are analyzed. The role that a **Client-Server Networking** has, for being possible to create a multiplayer RTS game is presented. Then there is presented the game framework used **Phaser.io**, describing how the game project is organized and what it contributes to the functionality of the game.

The game architecture of the game *Freewee* [44] was used as reference, for the documentation present in this section. That game has many similarities to *Planetary Conquest* such as being programmed used HTML5, Node.js, Socket.io and *Phaser.io*.

3.2.1 Technical Limitations

Starting with the Technical Limitations and Requirements that are present in this project, some of those are:

- **Server**: A Game Server is needed for this project with a computer device running it, for other players in computer platforms or mobile devices to access it. The player will have access to the server as a client, in order to be able to play the game. Also the server has an important role in maintaining the clients synchronized and being authoritative to the events that unfold in the game matches.

- **Socket.io**: This library enables real-time, bidirectional and event-based communication between the browser and the server[2].
- **Node.js**: A bidirectional RPC library[3] using *Primus.io*[4] as a network layer. It is designed to build scalable network applications.
- **Express**: Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile. applications[5].
- **Phaser.io**: A 2D game framework for making HTML5 games for desktop and mobile web browsers, supporting Canvas and WebGL rendering.
- **Game Device**: A personal computer or mobile device is needed to play the game.
- **Internet Browser**: An Internet browser is needed to be installed on the gaming platform device chosen by the player, in order to access to the server's domain to be able to play the game.

User Requirements

This is an analysis of what the target users of *Planetary Conquest* are expected to do and desire of this game:

1. Users need to have a playable device that can support the use of a internet browser, such as a personal computer or mobile device.
2. Users must know how to access to the game's domain in order to be able to play with other players.
3. Users must be able to issue input commands to the game they are playing, in order to send order commands to the server and to interact with their opponents.
4. Users want a game that has faster response time to their input commands. Also the interface to support the execution to those commands must be easy to understand and not difficult to navigate. This has to be applied on any game device chosen by the player.

3.2.2 Client-Server Networking

To be able to connect the players to the game, a server is needed. The server will have all the information about the game match, simulate some events belonging to it and have complete authority about what it is enfolded between the players actions.

[2] https://socket.io/docs/
[3] https://nodejs.org/en/
[4] https://github.com/primus/primus
[5] https://expressjs.com/

The server will check if the players are connected to the game match and broadcast the information that is being made by the players actions. This is possible to do so because the players are connected to the server as clients. However each action is supervised by the server game's simulation, in order to check if it is valid or not.

The clients will be responsible to take the orders issued by the players from their gaming platform input commands, then the server will receive those commands. After checking if the actions requested by the clients are valid, the server will broadcast the results of it's simulation to every client connected to the game match.

In short the server is in charge of providing data and services to one or more clients. In the context of game development, the most common scenario is when two or more clients connect to the same server. The server will keep track of the game as well as the distributed players. [45].

Some advantages on using the communication method are:

- **Authoritative**: With the server being completely authoritative over the actions that enfold during the game, the players do not have to worry about game cheating or having different game scenarios between the game machines of each player.
- **Less calculations for clients**: With the server simulating the game events that are made by the players, plus being exclusive responsible for some heavy computing algorithms, it is possible to save processing time from the client's game device.

However there are also disadvantages is this method:

- **Communication takes longer to propagate**: When a client wants to apply a command issued by the player, it needs to send the request of that command to the server, then the server needs to check if the command is valid. If that command is valid, the server will broadcast the response of that command to all connected clients. It takes some time for the players to see the command sent being effectuated by the game, comparing to the communication used on the Peer-to-Peer model, since this model directly relays the commands for client to client without the server being a mediator over them.
- **More code complexity**: While it is useful that the server is the entity responsible for running the game logic code, with the clients being responsible for the animation and rendering parts of the game, it can be complex to divide the game code efficiently between the Client and the Server side.

Now it will be presented some of the functionality included in the Game Server implemented and used in this thesis. There is an explanation on how the server can accept connections and communicate with Game Clients and how it enfolds it's authority over the events of a game match between two players.

Server

There is one tutorial [46] that helped in the implementation of*Planetary Conquest* game server. That tutorial explain how to setup a multiplayer game, programming the server and client module, and making those two interact with Socket.io. The client is written in Javascript using *Phaser* as a game framework. The server is also written in Javascript using Node.js and express module. The client and the server communicate by using Socket.io.

Bellow is a simplified figure, representing the communications between the Server and the clients:

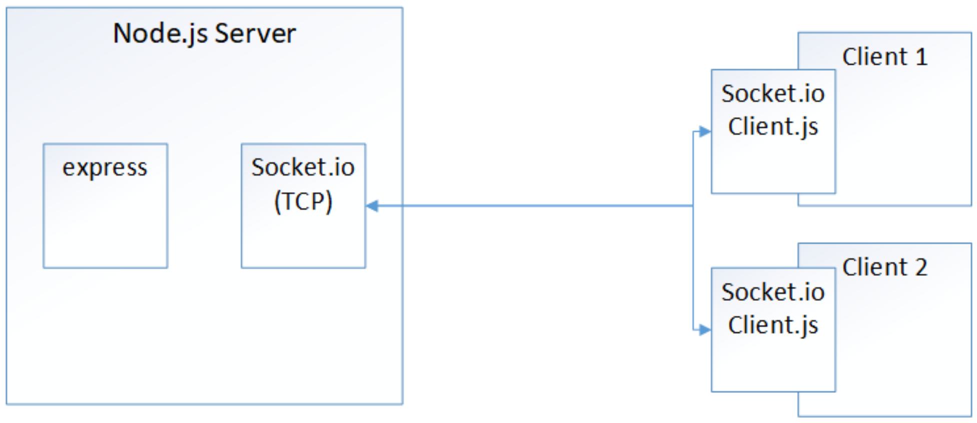


Figure 3.3: Simplified Game Networking

The server will need to use Node.js modules, since they are required to use.

Listing 3.1: Server Initiation code

```
//requiring Express module
var express = require('express');
//creating an instance of express, to start an Express application
var app = express();

//creating a server, for being able to: communicating with client players,
//keeping game information, creating sockets and being authoritative
//over client requests
var server = require('http').Server(app);

//requiring socket.io for listening client connections, to the created server
var io = require('socket.io').listen(server);
```

Express is the model used for serving files to clients, making these files accessible to them. *app* is a new instance of *Express* and will combine with HTTP in order to have a HTTP server. With this type of server it is possible to satisfy client requests over HTTP and other protocols. Then the *server* variable will combine with *socket.io* and it will listen to clients connections.

The type of connection will be Transport Control Protocol (TCP), which guarantees the delivery of the messages data in the correct order. If a package is lost the target application can notify the sender application, and any missing packets are sent again until the entire message is received. However there are problems with this connections, as the reply of a message can take some time compared with the other commonly used protocol User Datagram Protocol (UDP). Still since TCP guarantees that every message is sent to the clients in it's entirety, this situation makes this protocol ideal for the communication of a multiplayer RTS game.

Then there are files that will be accessed to the clients:

Listing 3.2: Directories to be used

```
//load some directories, for clients to use
app.use('/css', express.static(__dirname + '/css'));
app.use('/js', express.static(__dirname + '/js'));
app.use('/images', express.static(__dirname + '/images'));

//using index.html as the root page
app.get('/', function (req, res) {
    res.sendFile(__dirname + '/index.html');
});

//server is listening to the given port
server.listen(process.env.PORT || 80, function () {
    console.log('Listening on ' + server.address().port);
});
```

The first three lines of code, permit the clients to use files needed to the game functionality that are not directly accessed. Then the server will listen to the designated port (port 80) where the clients will access to, with the file *index.html* serving as the root page.

The clients will communicate with the server, emitting commands made by the players input with the use of *socket.io.js* library.

Lock-step method

To guarantee that the clients are synchronized with the server and the messages sent by the server are received to all the clients at the same time without major latency problems, an architecture known as *lock-step* networking model [47, pg 382] was used.

In this model both players start with the same game state. When the player issues a command to an unit or structure it possesses in-game, the client of the player's gaming platform will send the command to the server instead of being executed immediately. The server after checking if the command is valid or not for execution, will then send the same command to the connected players, with arguments to be executed on the client side functions. Once the players receive the command they will execute it at the same time, ensuring that the game on both machines stay synchronized.

The server will achieve this behavior by running it's own game timer at 10 clock ticks per second (100 ms). The player's client also runs it's own game timer at the same pace as the server. Those ticks are also recorded by these two entities.

Each time the client timer ends it will send to the server it's own game tick, being updated to it's corresponding socket on the server side. When a command request is sent by a client, it will be stored in a command array that the server has.

Each time the server game timer ends it will send the response of the requested commands to the clients. Those commands have specified the current game tick on them. The clients will execute the commands response corresponding to the game tick received.

Since the server needs to execute the response to the commands for all the players at the same time, it will need to wait for the commands sent from all the player clients to arrive before stepping ahead to the next game tick, which is why it's called *lock-step*. If there is a least one client whose game tick is lower than the server's game tick, which could be caused by latency issues, the commands stored by the server will not be send for the connected clients. The server has to wait another timer cycle to check if the clients who had their game tick lower than the server's game tick are now valid, in order to send the commands response to all the connected clients.

Authoritative Server

The *Lock-step* method demonstrated how it was possible to have the clients synchronized with the server, and having the server messages be relayed to all the clients at the same time. However there is still the emergence to guarantee that no types of cheating happens. To do so the server needs to be *authorative* over the events of the game match played by the clients. With the server simulating the game match state, the clients do not have to worry about being cheated or receiving incorrect messages about the events of the game. One example of a server using it's authority is in unit vs unit combat. The order for these units to combat is made by the clients input and the animation of those attacks as well. However the calculations of damage on that combat are simulated by the server only. The server sends information to the client about the amount of health an unit has lost by enemy attacks and when it has to die.

In summary the game state of a game match is managed by the server alone. Clients send their actions to the server. The server updates the game state periodically and then sends the new game state back to clients who just render it on the screen [48]. There were some measures taken for the server to be able to be authoritative to the game state such as calculating the damage done by the combats, simulating the movement of units and monsters or having timers to the construction of structures and production of units. Those details will be discussed in the next chapter **Implementation**.

To end this subsection a diagram is presented below, with the relations between the clients and the server in order to setup a *Planetary Conquest* game match.

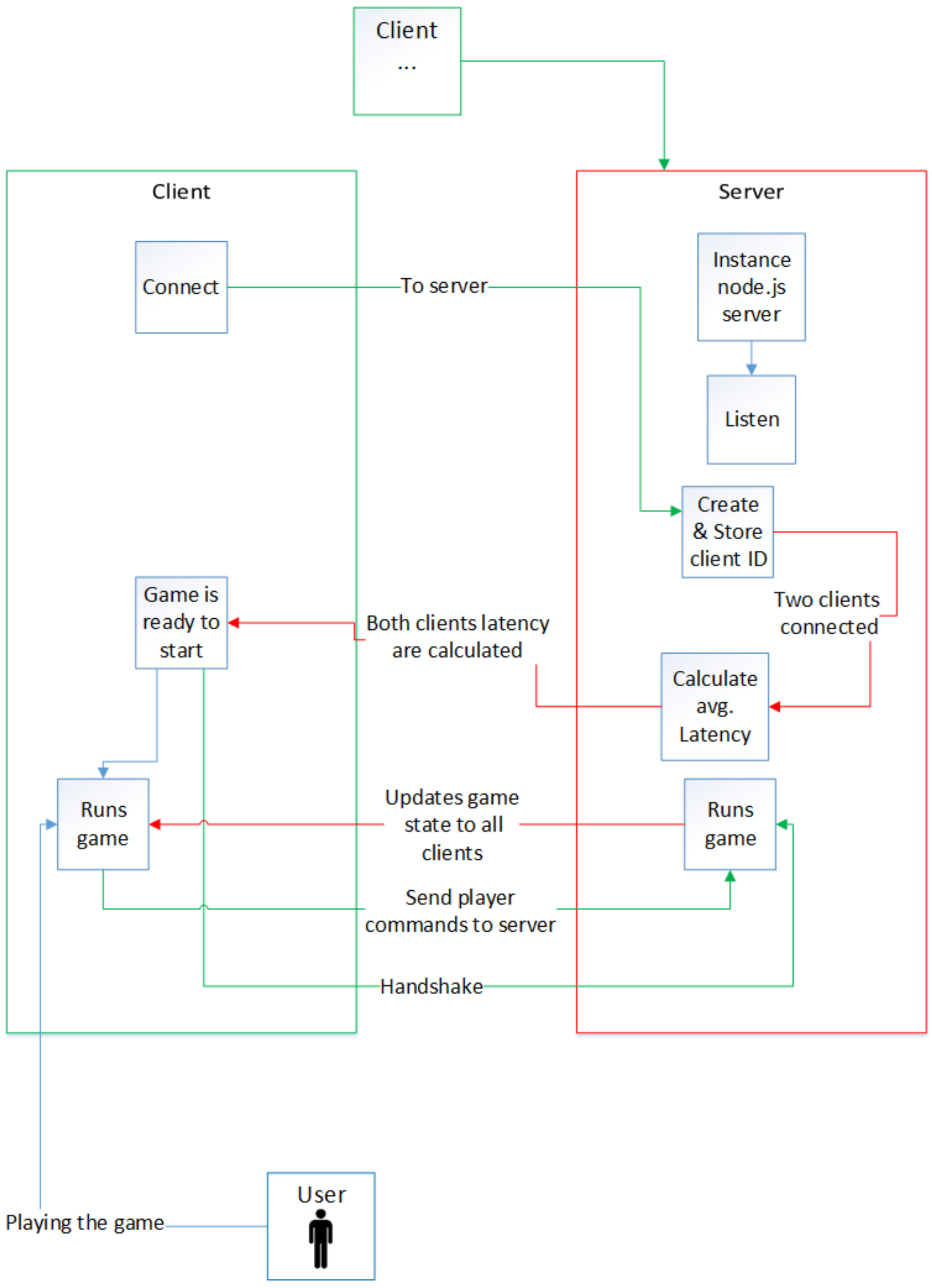


Figure 3.4: Client-Server game communication

3.2.3 Phaser.io

In this subsection there is a discussion about the architecture of the game framework *Phaser* used in *Planetary Conquest*. To understand how this project is organized, an explanation of the files which were loaded in the root file of this project *index.html* is made bellow.

Starting with the libraries used, this project uses *socket.io.js* for socket network communication and *phaser.js* to be able to use the functions available to the *Phaser* game framework. There are other libraries used such as *phaser-plugin-isometric.min.js* that loads the plugin responsible for isometric projection of the game assets and the use of isometric operations. *phaser-kinetic-scrolling-plugin.js* is used for scrolling the game map on the mobile platform. *easystar-0.2.1.min.js* is used for the *A-Star* movement of units and monsters and *HealthBar.standalone.js* is used for creating bars for representing the health of an unit or the percentage of a technology research.

Then there are loaded 4 files: *MonsterStruct_Client.js*, *UnitStruct_Client.js*, *BuildingStruct_Client.js* and *UpgradeStruct_Client.js*. Those files refer to detailed information about each game element which are Units, Structures, Monsters or Upgrades respectively. The same information for these elements is used for the Clients and the Server. That information can include a game object health points, development time, production resource costs and others.

After those 4 there are loaded the *Phaser* game states that are part of this project. *Phaser* supports this method allowing different parts of the game to be included in these files. When one part of a state functionality is completed, it will request the start of a following one. There are four states present in this project:

1. *game.js* The first state of this project that starts by creating a *Phaser* game instance with a size, and render mode specified. Then it adds the other three states to the project game states.

2. *boot.js* The second state that is responsible for the initialization of the game screen, with the size depending on the game device used. It is also responsible for other types of initialization such as the type of physics to be used or enabling input events.

3. *load.js* The third state responsible for loading to the game various assets used such as images, spritesheets, text fonts or JSON files containing information about the game map.

4. *play.js* The last state, which is the game manager of *Planetary Conquest*. This state has various functions such as initializing the interface, respond to client inputs or receive and treat the command responses sent by the server.

Then there are three classes containing the interactive game elements present in this game: *unit.js* *structure.js*, *monsters.js* for Units, Structures and Monsters respectively. The first two elements are programmed as Prefabs, which means game objects created by these classes will be reusable. When one of these game objects reaches 0 or less health points, instead of being destroyed these objects are put in a state where they are not rendered neither have their behaviour processed. However they still keep the information initiated by their creation and are still present in the game device memory. When the player produces another type of these game objects such a new unit or structure, if there are available prefabs that are inactive one of those will be active again, with the information of the the new game object produced. With this approach it is possible to save memory and processing time when creating new units or structures. The reason for monsters not being implemented as prefabs is because there is no other way to have more monsters than those that are spawned at the beginning of the game, having no utility for them being counted as prefabs.

Finally there is the *client.js* file, that is responsible for sending the command inputs of the player to the server and relay the server messages to the game manager. Bellow is a figure with the order of *Phaser* states in *Planetary Conquest*:

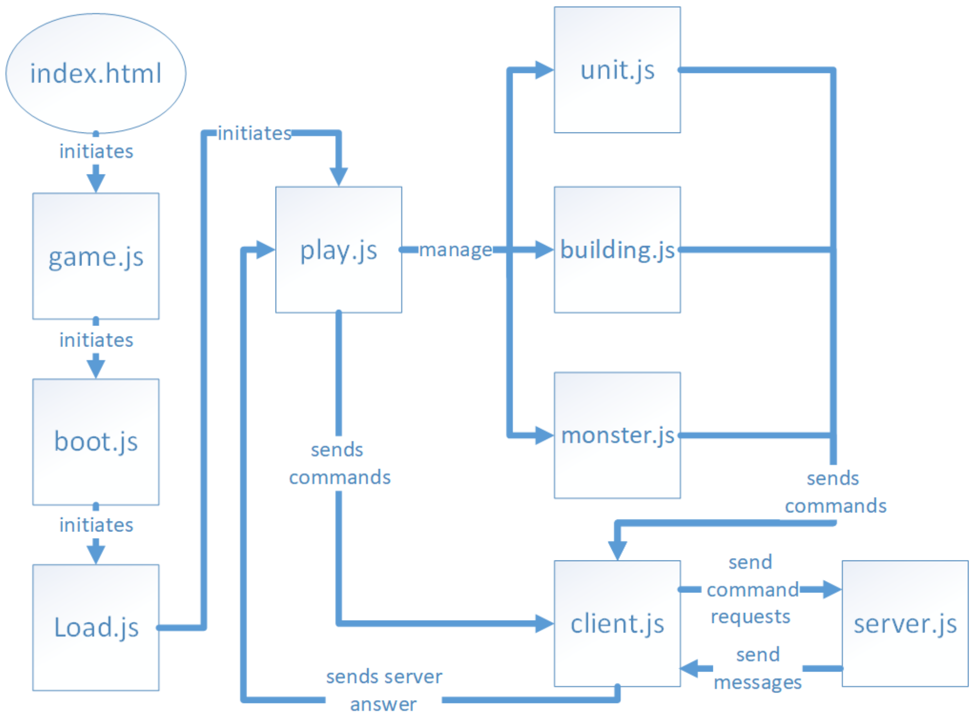
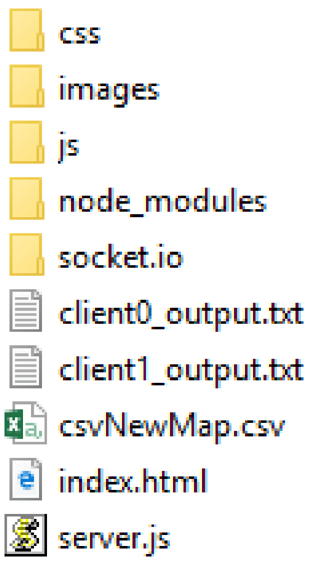


Figure 3.5: State diagram of Planetary Conquest

3.2.4 File Structure

The file structure that is part of *Planetary Conquest* is the following:



css
images
js
node_modules
socket.io
client0_output.txt
client1_output.txt
csvNewMap.csv
index.html
server.js

Figure 3.6: Planetary Conquest file structure

The **css** folder contains the *style.css*. The **images** folder contains various assets used in this game such as text fonts, icons and spritesheets of units or monsters. It also contains images of structures and immovable map objects and projectile animations. The **js** folder includes three folders: The first is the **libs** folder containing some libraries used in this game. The second **prefabs** contains the scripts for the prefabs that exist in this game (units, structures), the monsters file and the scripts that have attribute information about the attributes of the three interactive game objects and upgrades available to them. The final folder is the **states** containing the client.js file, and the file for the game states mentioned for this project. The **node_modules** contains some of the libraries to be loaded on the server for supporting it's functionality. The last folder **socket.io** permits the use of socket functions to be able to setup a networking multiplayer platform.

The two *.txt* output files are used as log files, containing various information about the state of the game matches. The *.csv* file is used to load information about the game map, such as the immovable game objects that are part of it and their position on the map. The *index.html* loads the scripts that will be used in *Planetary Conquest* on the client side and **server.js** which has the server information.

Phaser.io Game Logic

In this subsection it is discussed some of the functions that are usually part of the *Phaser* game states. Referring to the game states present in *Planetary Conquest* those are [49]:

- *preload()*: This function is called at the beginning of a state file if needed and it's used to load game assets to the project. This function is being used on the **load.js** state.
- *create()*: This function is called after *preload()*, being used for the initialization off attributes of a

state and enabling some events for the game such as physics or enabling input support. This function is being used on all the game states present in this project besides *game.js*.

- *update()*: After *create()* this function is called periodically at every game frame. It is useful to update variables or to change conditions trough the duration of the game. This function is being used on the game state *play.js* and in Unit, Monster and Structure classes.
- *render()*: While it is not being used for game playing since all game objects are automatically updated, this method is called after game renderer and plugins have rendered the game objects. It is possible to do final post-processing style effects in this function or apply debug functions to see information about *Phaser elements* in-game such as: camera position, game size, game objects physic containers and others.

It is presented a diagram resuming the use of the *Phaser* game state logic.

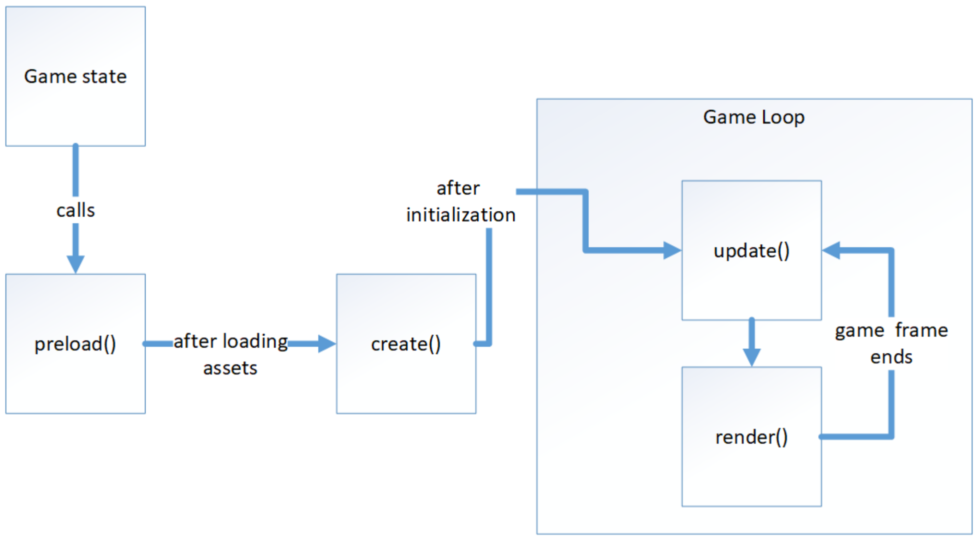


Figure 3.7: Diagram of Phaser Game state logic

3.3 Discussion

This marks the end of Chapter 3 where there was a discussion the on the **Game Design** and **Game Architecture** used in *Planetary Conquest* .

The **Game Design** section presented many in-game elements that are part of it common to other RTS games, the progression made by the players, the interface and some others.

For the section of **Game Architecture**, there was a presentation of the Technical Limitations this project it has, how the implementation of a game Server and a game Client for each player made it possible to have a multiplayer networking game and how the project was organized with the use of the game framework *Phaser.io*.

The next chapter is about the **Implementation** of this project, detailing how some functions that permitted the functionally of some of the most important game events were implemented on the project files discussed in this chapter.

4 Implementation

Contents

The **Implementation** process that was part of the *Planetary Conquest* game is written in this section. This section starts with the **Development Process** that was enfolded on this project and how some of the most important tasks were implemented. Then refers **What went wrong** during the development of the project, explaining the problems that occurred and what were the causes for those problems.

4.1 Development Process

There were various tasks that had to be implemented on this project in order to develop a RTS multiplayer game. Some of those were crucial to the creation of it, which were: **Spritesheets**, **Game Map**, **Phaser Game states**, **Interface**, **Isometric projection**, **Prefabs**, **A-Star movement**, **Spatial Hashing**, **Client-Server communication**, **Server authoritative** and **Device orientation**. Before starting the explanation of some of these tasks that were implemented, the **Starting Tutorials** that helped to understand the game framework that was used are described, and what was their importance in order to know how to create a HTML RTS multiplayer game.

4.1.1 Starting Tutorials

To know more about the game framework *Phaser.io* that was chosen to help creating *Planetary Conquest*, there were some tutorials found in the internet that helped to understand how this framework could benefit in creating a RTS multiplayer game.

The first one used can be found on the *Phaser* website [50]. The framework version used in this tutorial was *Phaser* 2, which also coincides with the version that this project uses. For most of the development time the revision used was the 2.6.2 - *Kore Springs*, since it was the last official build published by the developers of this framework that contained the last update of the *Phaser* API docs. Much later during the development cycle it was used the 2.9.2 version made by the Community Edition[1] of this framework. This new revision corrected some warnings that were appearing on the console of the programming tools, that are available for the web browsers used during the project's development. This tutorial started to introduce the importance on knowing how to use a web server to implement a multiplayer game, and then showing a simple *Hello World* example to being setup on a webpage. Then it presented a simple game tutorial which explained the basic steps to create a basic game using this framework [51].

Following that tutorial, it was studied some video tutorials on how to make a HTML5 game using *Phaser*. One was *TapTapTaxi*[2]. This tutorial was helpful to understand some techniques that are made in game developing such as: Prefab creation, Animation and Spritesheets use, loading assets and

[1] https://github.com/photonstorm/phaser-ce/tree/v2.9.2
[2] https://www.codecaptain.io/courses/html5-game-development-in-phaserjs

sprite placement. The game made by this tutorial also used isometric sprites and could work on personal computers and mobile platforms. Another one was the *Spaceship Game* created by Zenva Course[3], that explained other set of techniques such as creation of particle effects, physics events like collisions or velocity movement, camera functionality, *Phaser* states configuration and moving background animation.

With these tutorials it was learned on how to create a HTML game in Phaser and knowing how to use various techniques that are applied to game developing. However there was still missing some information about creating a RTS game and being multiplayer as well. Fortunately there was a book that could satisfy both needs, written by Aditya Ravi Shankar called ***Pro HTML5 Games*** [47] [52]. This book explained how it was possible to program a multiplayer RTS game written in HTML. The book referred topics such as creating game elements, interface, map creation, producing units and structures, A-Star movement, unit collision and avoidance, Fog of War and how to setup the game to be played with multiple players. Also it explained how to implement the *Lock-step* method referred in the previous chapter. This book is a very appropriate reference for some of the tasks implemented in *Planetary Conquest*.

Finally there were two websites who were useful to solve some technical doubts surged during development. One was **HTML5 Game Devs** [36] which had the largest community about the development of HTML5 games, and has *Phaser* as the principal game framework used. The other website was the API documents that were available for the *Phaser* 2 version [4], where information about the Classes that *Phaser* has at is disposal is documented.

With all these tutorials plus the one referred at chapter 3 about creating a multiplayer game in *Phaser* [46] it was acquired sufficient knowledge about this game framework. Thus the implementation of this project could start. Concluding the explanation of tutorials used for this project, there is the explanation of the most important implementations made in this project and what they contributed to the development of it.

4.1.2 Spritesheets

After completing the tutorials indicated in the subsection above simple test of loading a game map was made. The map consisted of a 30x30 rectangle with one color and an image of an unit placed on it. The initial strategy for completing the tasks needed for this project was implementing every single task that could be run in just one client such as the game states, game objects, interface and game map. After that it would be implemented the server and programming the communication that would be made with the players clients.

One of the first tasks implemented was how to load a spritesheet that could be used on a game sprite and how it could be animated. A sprite consists of a set of coordinates and a texture that is rendered

[3]https://www.udemy.com/phaser-game-development/
[4]https://phaser.io/docs/2.6.2/index

to the game canvas, that can also have additional properties such as physics motion, input handling, events, animations and others[5].

As said in chapter 3 in the subsection **Perspective** it was quite difficult to create or find sci-fi image assets, so the spritesheets that could be found on the group **The Sprite Resource** [42] were adequate to use in this project. After deciding the game image assets that would be used, it was needed to turn them in spritesheets that could be loaded into the *Phaser* framework and animated by it properly. The way that the spritesheets on the game website were presented, they could not be recognized as valid by *Phaser*. The issue was the existence of different space sizes between each image frame of the spritesheet, and they needed to be spaced with the same horizontal and vertical space for being able to be recognized by the *Phaser* framework. Each frame of a sprite needed to have the same width and height from each other. The solution to solve those problems was the following:

1. First those images would be downloaded and placed on a separated folder. Then each frame of a spritesheet would be separated using the program *ShoeBox*[6] from Addobe Air, which is a program that the user can drag and drop images to apply operations for sprites, bitmaps, animations an others. One option available is on the Sprites tab which is **Extract Sprites**, that was used to remove every image frame that was contained in every spritesheet downloaded. By applying this method every frame that was present on the spritesheets was separated in one single image.

2. As it can be seen in the example of the Orc Peon spritesheet[7] there are frames for the movement, walking, mining/attacking, return resources and death animations that can be used to animate a sprite. However only 5 directions were present (North, Northeast, East, Southeast and South) and a unit or monster sprite would have 8. The 3 remaining directions were Southwest, West and Northwest. By using the program GNU Image Manipulation Program (GIMP)[8], which is as it's name implies an image manipulator program, it was possible to invert a spritesheet image horizontally, then saving to another image and repeat the process done on *ShoeBox* for this new image. There were extracted the sprite frames of the 3 directions that were missing.

3. Then those extracted image frames were ordered by number to be used in *Phaser*, first the 5 directions described for moving and attacking from the original spritesheet. Those would be followed by the other 3 directions for the same purposes from the inverted spritesheet and finally the dead animation.

4. After ordering the frames, Gimp would be used again to finally have an appropriate spritesheet image. To do that a tutorial was used [53]. Afterwards all the spritesheet frame layers are aligned

[5]https://phaser.io/docs/2.6.2/Phaser.Sprite.html
[6]https://renderhjs.net/shoebox/
[7]https://www.spriters-resource.com/pc_computer/warcraft2/sheet/29480/
[8]https://www.gimp.org/

to the center, then the layers are resized to have the same height and width without deforming the frame and finally combining all the layers to a single spritesheet, using the plugin *Fuse Layers*. Each line of the spritesheet contained 5 frames. This method would be concluded with the spritesheet being saved in a single *.png* file.

5. With the spritesheet image created, it would be loaded in the *Phaser* game load state - *load.js*, giving a key name to the spritesheet and the width and height that was common to each frame. This was done in order for *Phaser* to load the frames that are part of an animation with the exact dimensions that they have.

6. These spritesheets are used to their corresponding prefab unit, specifying the frames that are part of an animation of a prefab such as moving to a given direction, attacking or dying. There was also the need to designate the amount of time that an animation would take from start to finish and looping that animation when it was ordered to play. Flying units and monsters have one particular detail, which is having their moving animation looping even when they are still in the same position without moving, to give the impression that they are flying.

This solution was used in every unit and monster spritesheet. There was one more course of action for the monsters spritesheet, which was having three sets of colours. One for each race Orghz and Human (red and blue) and one for the Monsters (yellow). To do that one color was already applied to each monster, the other two would be applied using an image available in the Sprite Resource website[9], that had 8 army colors at disposal. The RGB code of those colors was used to replaced one set of colors to another.

4.1.3 Game Map

With the creation of spritesheets and the animation of their corresponding prefabs, the next task was to load the game map to this project.

To be able to create a game map image the program **Tiled**[10] was used, which is a 2D map editor both for orthogonal and isometric perspective. The first map created was a simple 30x30 tiles map, with the same ground tile used that was by 64x32 pixels, with this measure giving an isometric aspect view for the tiles used. Then as the project progressed there were added more tiles, some with different terrain tiles such as water, magma or flora which led to the final game map. The dimension of that map was 45 by 45 tiles, with each tile still being by 64x32 pixels with isometric orientation. The game map would be loaded on the game load state - *load.js* as an image, then it would be placed in the game manager state - play.js with it's sprite anchor set to (0.5,0). An anchor of a sprite is the point of reference when placing

[9] https://www.spriters-resource.com/pc_computer/warcraft2/sheet/60022/
[10] https://www.mapeditor.org/

it in the game world[11] . Each tile position would be referenced to a matrix being also 45x45, where there could be one of two possible numbers: 0 specifying a non-occupied position, and 1 specifying an occupied position. This matrix is present in the server and in the client side, with the server being responsible for changing the matrix positions values on both sides.

Now a game map could be loaded to use and the game objects that would be over it as well. To be able to place the starting immovable objects that were part of the map such as rocks, mountains or trees, it was done the following:

1. First in Tiled each special tile that would have a immovable game object over it, had a field called *Personalized Property* with a name to reference that object. For example the white trees immovable object would have their name being W.
2. Then after those names being designated for every special tile the map would be exported to a .csv file. That file would have 45 lines and columns, with each element being separated by a comma containing either the number 0 or the names specified earlier for each special tile.
3. Then using Microsoft Office Excel for loading the .csv file it was created a Pivot Table in order to have three columns with the positions and names of each special tile. The three new columns were X, Y and Value; for the line, column and name that each special tile had respectively. The resulting table would be saved in a .csv to be used on the server side. On the client side it would be used in .JSON format, using a free converter found on the internet[12].
4. Afterwards it was coded in the server's initialization code a 45x45 matrix (45 arrays each with with 45 positions, into a single array with 45 positions), having their value at 0 for each non-occupied position and 1 for occupied positions. On the client side was also used the same method and it was loaded each image corresponding to the game element it represented.

The starting immovable game objects of the game map could be changed in position, type and quantity, by changing those values on the .csv files.

4.1.4 Isometric plugin

Since one of the main objectives for this project was to have a RTS game with isometric projection, it was researched how could it be possible to implement it. After some research there was found a plugin compatible with *Phaser* [54]. This is a comprehensive axonometric plugin for *Phaser*, which provides an API for handling axonometric projection of assets in 3D space to the screen[13]. With the use of this plugin it would be possible to use a set of operations that would not be possible using what

[11]https://phaser.io/examples/v2/sprites/anchor
[12]https://www.csvjson.com/csv2json
[13]https://github.com/lewster32/phaser-plugin-isometric/

Phaser offers, regarding orthogonal projection. Some of those were physics operations such as velocity, collision or sprite container. There were also options for depth sorting the game objects present on the game map. The position of the game objects would be using isometric coordinated, with the sprites having an isometric view on a grid shaped diamond map.

There is a tutorial that setups a small *Phaser* game example using this plugin [54]. The tutorial explains how this plugin can be loaded in the game project, how to add objects that are recognized as isometric (*isoSprite*) and adding physics, collision and overlap events to them. It was possible to initially add these operation to *Planetary Conquest* project as well. One that had to be removed afterwards was collisions, since it provoked serious problems regarding some actions that enfolded during a game match. Those problems will be discussed in the section **What went wrong** of this chapter.

With this plugin each object prefab and the other immovable game object would be instantiated as a *isoSprite*, instead of a sprite object for being able to use some of the operations described above. However there was a problem using this method with a large number of game objects instantiated this way. It cost several processing power to the game devices, when drawing these objects during each frame. Even with only applying this plugin to the prefab objects it would still happen. There was no other solution to revert to the use of normal sprite objects, instead of *isoSprite*. With the use of this type of objects there was a change on how the velocity of the units and monsters would be calculated and it is going to be explained as well on the section **What went wrong** of this chapter.

4.1.5 A-Star movement

In order to have intelligent movement made by units and monsters by walking from a start to a end position, using the shortest path possible and without colliding on any immovable object it was needed a pathfinding method. There are algorithms used for pathfinding search, one of those is **Dijkstra's Algorithm** created by Edsger Dijkstra [55]. This algorithm permitted to find the shortest way possible from a node to another, by checking every single node and route cost that belonged to a graph. As it can be seen in the figure bellow, it was determined the shorted distance possible between the two green nodes, by calculating which path would have the shortest total cost. That cost was the sum of each route that was part of the shortest path [56].

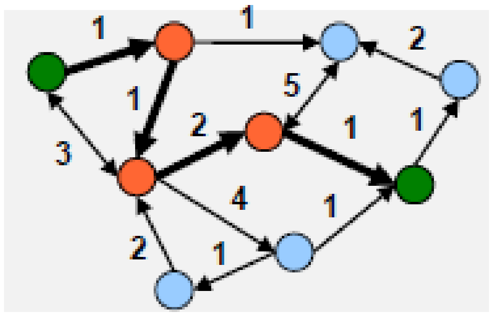


Figure 4.1: Dijkstra shortest path calculation - IST AIG Class

The problem with this algorithm is exactly what was written above, on how is performed the shortest path search. The algorithm needs to check every single node and route costs in order to calculate the best path possible. Fortunately there is an optimized version of this algorithm, that was used for calculating the paths that were part of the movement made by the units and monsters of this project. That algorithm is denominated **A-Star**.

This algorithm is faster than the previous one presented, since it will travel the node graph to find the shortest path possible between two nodes without the necessity of visiting all nodes of the graph. To do so it can be represented an evaluation function:

$$f(n) = g(n) + h(n) \tag{4.1}$$

The objective of this function is to have the lowest **f** value possible, which will indicate the shortest path possible for one node to another. The parameter **g** means the movement cost to move from the starting node to a given square on the grid, following the path generated to get there. The parameter **h** is the estimated movement cost to move from that given square on the grid to the final destination, which is referred as an heuristic. With the use of an heuristic it would be faster to calculate the shortest path possible.

There are two commonly used heuristics that can be used with this algorithm in grid maps. One is the *Manhatan* which calculates the shortest distance of two nodes, with the restriction of not using diagonal movement between each node. The formula is given by:

$$h(n) = |goal.x - n.x| + |goal.y - n.y| \tag{4.2}$$

The other one is the *Euclidean* which also calculates the shortest distance possible, but it can move diagonally, with the formula being the following:

$$h(n) = \sqrt{(goal.x – n.x)2 + (goal.y – n.y)2} \tag{4.3}$$

The following figure shows the use of these two heuristics, with the green line being *Euclidean* distance (6 nodes). The other 3 are different applications of the *Manhathan* heuristic, having the same distance between them (12 nodes).

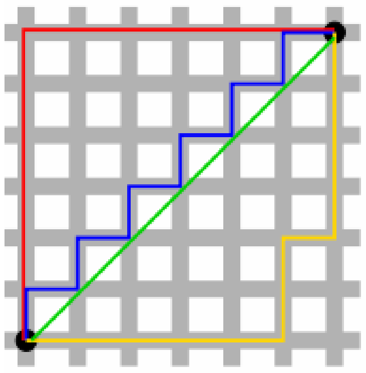


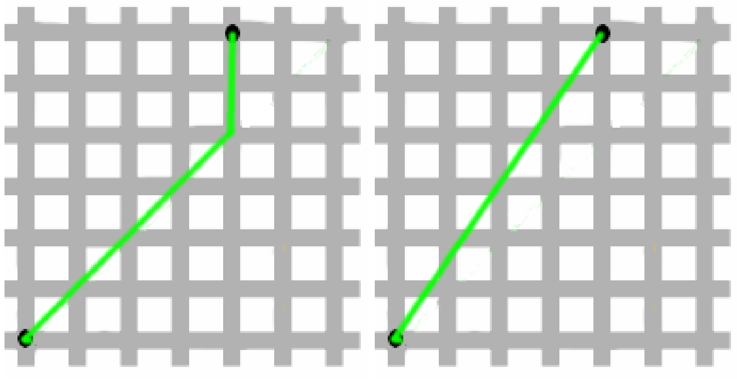
Figure 4.2: A-Star distance heuristics - - IST AIG Class

The *Euclidean* distance would be appropriate to be used in this project, since units have intercardinal movement directions (NW,NE,SW,SE) and with this type of movement it was possible to be found the shortest distance possible. For using the A-star algorithm a tutorial is available on the internet [57] by the same company TIZEN, that made the tutorial presented before about the use of the isometric plugin for *Phaser*. This tutorial explained how it was possible to give movement to a game object using the A-star algorithm. It was used a library called **easystar.js**[14], which permits the use of A-star algorithm already implemented in this library.

There are some set of options offered by this library, such as enabling corner cutting that are part of occupied nodes for the calculated path, permit the path to move trough diagonal nodes or not, setting which numbers of a grid matrix are acceptable as walking nodes, and others. With the use of this library calculating the shortest distance between two nodes, with the options of corner cutting disabled and the diagonal movement enabled, it was possible to have A-star algorithm used in this project with a *Euclidean* distance heuristic.

Looking at the distance heuristic used there is still a situation that could be optimized, being that the units and monsters have to walk on all the nodes calculated which might not be the shortest distance possible. The following 2 figures show an example of this possibility. The first one is the path calculated by A-Star with the *Euclidean* heuristic from one node to another represented by black nodes, with the path being the green lines Figure 4.3(a). The second one is the same situation but with a method called **smooth path** applied to the calculated path, reducing the distance from one node to another Figure 4.3(b).

[14]https://github.com/prettymuchbryce/easystarjs



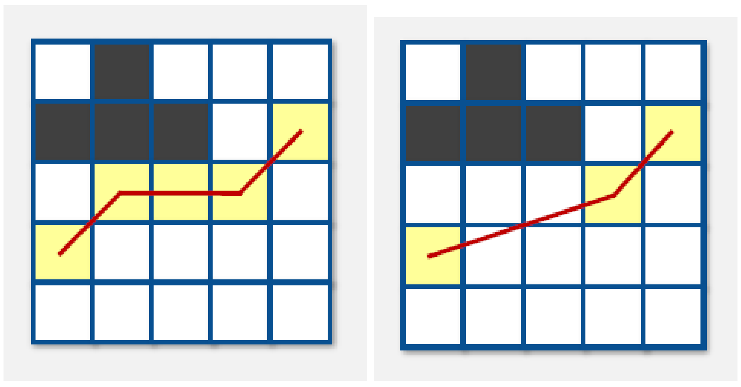
(a) Without smooth path **(b)** With smooth path

Figure 4.3: Differences between smooth path application - IST AIG Class

To apply this type of behaviour to the A-star path calculated, the smoothing path method had to be implemented. On the *server.js* file the function *smoothPathServer* was responsible to apply the behaviour of this technique. The idea of this function was retrieved by this reference [58], using the method **Straight-line Smoothing**. This method receives a path already calculated by A-star method and checks in the following order:

```
Given a path P=[p1, p2, ..., pn]
•i=0
•Repeat until i <n-2
•For every 3 waypoints pi, pi+1, pi+2 in the path
•If walkable(pi, pi+1) and walkable(pi+1, pi+2) then delete pi+1
•Else move to next waypoint
•i= i+1
```

This means that the function will check if the 3 first waypoints are walkable, determining if the small paths with (pi, pi+1) and (pi+1, pi+2) nodes do not have any obstacle between them. To do so it was defined as ”walkable” if the two nodes compared would not have their adjacent nodes with an occupied or out of map bounds position. If the distance between those two nodes was ”walkable”, then the second node of those two would be cut from the path calculated previously. This would be repeated until this method reached the final node. Unfortunately this method still has a problem, the nodes that have adjacent positions which are occupied will count for the final path calculated. There are cases where this could be avoided. One example is the two figures bellow: the first Figure 4.4(a) being the current solution and the other without having the issue mentioned above Figure 4.4(b).



(a) Current smooth path **(b)** Proposed smooth path

Figure 4.4: Differences between straight-line smooth path application - IST AIG Class

To solve this problems one solution could be the use of raycasts emitted on the units and monsters position, to check if those hit an unmovable object during movement. However it couldn't be used on the server prediction simulations. Another solution would be adding sample points between each node (pi+1, pi+2), and check with the units width if it would occur a collision with an unoccupied tile. This was also not employed since *Phaser* physics could not be run on the server side. The solution being used in this project would still return the path of the first figure.

For the server to be able to maintain authority of the movement performed by the units and monsters of the game match, it would create a simulation of that movement. To do so it was calculated the amount of time that it would take from a moving game object to reach from a node to the next one of the path previously calculated. This would continue until the last node would be reached. With this the server had a simulation that would permit it to calculate the position of a moving game object at a given time.

The movement also had to be synchronous between the clients and the server, to do that there were two behaviours implemented.

The first one would be the clients sending the position of the units belonging to both clients and monsters, that were moving on it's game side to the server. The server would check the current position of a moving game object resulted by it's simulation, with the position received by the client. If there would be a substantial distance between the two of them (it was used 10 pixels at minimum to be checked), the server would have to change the game object's position on the client side. To do that the server calculated the future position where the unit would be, when the client received the server message. That calculation would take in account the average latency that was calculated for that client before the beginning of the game.

The second implementation performed would be the arrange of the *Phaser* physics time, during each update cycle of the unit and monsters prefab of each client. The alteration of the *Phaser* physics time was needed in order to calculate the velocity of a sprite, having in account the time that each update cycle would take. The following operation was added and executed each update cycle, when a game object was moving:

Listing 4.1: Physics elapsed calculation

```
var deltaTime = (this.game.time.elapsedMS) / (1000 / this.game.time.desiredFps);
this.game.time.physicsElapsed = (1 / this.game.time.desiredFps) * deltaTime;
```

The first line calculated an approximation of the delta time of the current game frame, which would use the time elapsed of the previous frame (*this.game.time.elapsedMS*, measured in milliseconds), divided by the amount of milliseconds that was supposed to be equal to the time that each frame would take (1000/60). Then the physics update delta (which is (1 / *this.game.time.desiredFps*) at the start of the game), would be changed by the second line of the formula above. This would mean that if suddenly a client would be having 30 Frames Per Second (FPS) during some time, the update cycle would move the unit twice the amount of pixels that what it was expected to be with a 60 FPS update cycle.

4.1.6 Spatial Hashing

At some point of the project's development it was found a problem for some functions that could be called by units, monsters or towers. Those functions involved checking which nearby game objects would be close to the game objects in question. This verification would contribute to look for nearby allies after receiving an enemy attack, targeting an enemy unit after killing the previous enemy target or determining if it is possible to build a structure that would not overlap on units or monsters. The type of issue found on the starting implementation of these functions would be performance, since the starting solution used would be checking every game object on the map to satisfy what was requested by the game objects that called those functions.

To tackle this problem it was needed some sort of data structure that would store the location of the game objects, and that it could be easily accessed to, in order to do some operations such as: Checking nearby game objects that are close of a game object or updating their position on the data structure.

One option that could solve this problem was the use of **Quadtree**. A quadtree is a collection of a trees used to efficiently store data of points on a two-dimensional space. In this tree each node has at most four children [59]. The methods used to build a tree of this type were the following:

1. Dividing the current two dimensional space into four boxes.
2. If a box contains one or more points in it, a child object is created. On that child object it is stored the two dimensional space of the box where this child is contained.
3. If a box does not contain any points, a child would not be created on it.

4. The previous steps are recursed for each of the children.

This method would be seem to be appropriate to resolve the problem quoted on this task. Quadtree permitted to store units, monsters or structures that could have their position updated and accessed to. However this method posed a problem. It was demanding for the Central Processing Unit (CPU) of the game device to rebuild the tree each time there was a change on a game object position that would oblige this behaviour.

The other method that came to be the solution to this problem and was used in this project is **Spacial Hashing**. This method is a process by which a 3D or 2D domain space is projected into a 1D hash table. To help solve this problem there is a tutorial available on the internet [60]. In this project instead of an hash it would be used an array that would be instantiated by the following code:

Listing 4.2: Spatial Hash instantiated

```
//the cell size that is part of each row and column of an hash
this.hashCellSize = this.mapSize * this.tileSize / 8;

//for units
this.hashArray = new Array();

//for structures
this.hashBuildingArray = new Array();

//the number of tile coulmns and rows are the same, 45
this.hashColumns = Math.floor((this.mapSize * this.tileSize) / this.hashCellSize);
this.hashRows = Math.floor((this.mapSize * this.tileSize) / this.hashCellSize);

this.hashWidth = Math.floor((this.mapSize * this.tileSize) / this.hashCellSize);
```

The size of each cell that would be part of an hash would be 1 of a 64th of the map size, which means that the hashArray would have 64 positions, from 8 hash lines and columns. The values used for this type of calculations are all according to isometric coordinates. Each position would contain another array (or usually called buckets by this method) that would contain the game objects inserted on them. There would be one for units and monsters together (hashArray) and one for structures (hashBuildingArray).

When one of these 3 objects would be created to the game world, it would be calculated which buckets they would be part of the corresponding hash array. Since these objects are sprites with width and height dimensions, just inserting them in one bucket to their corresponding position would not be enough. There was the necessity to see which buckets would be overlaped by the sprite bounds. For that by checking the physics container of a game object sprite, which is a rectangle with the width and height of it with it's center on the sprite middle point. Then it is only needed to check the buckets that the 4 vertex of the rectangle are part of. The following figure will show an example of this:

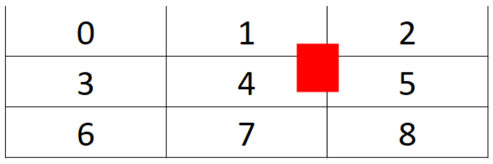


Figure 4.5: Example of Hash spacing application

As it can be seen in this picture it is an example of a bucket list with 9 buckets (from 0 to 8). The red rectangle is corresponding to a game object sprite. If it would only be accounted the middle position of it, the game object would only be inserted into the bucket 4. However the sprite also overlaps with the 1, 2 and 5 buckets. Thus those other buckets also have to be accounted for.

The functions responsible to store the game objects mentioned into their respective hash array, would be called when one of these objects would be created to the game world or when an unit or monster would move. The update of the sprites position would also cause to change the buckets where they were inserted. One important factor to be aware of is that a game object container should not be bigger that the size of a bucket, otherwise the calculations performed would not be valid.

With the Spatial Hashing method it was possible to perform some functions that required to check nearby game objects, without having to check all the ones that were created to the game world. Instead it would be checking those who were close to the game object in question.

4.1.7 Device orientation

This task was done for the mobile platforms such as smartphones, iPad or tablets. For these types of device it would be needed to scale the game size, according to the screen size of the game device and adjust the game size to the game device landscape (horizontal) position.

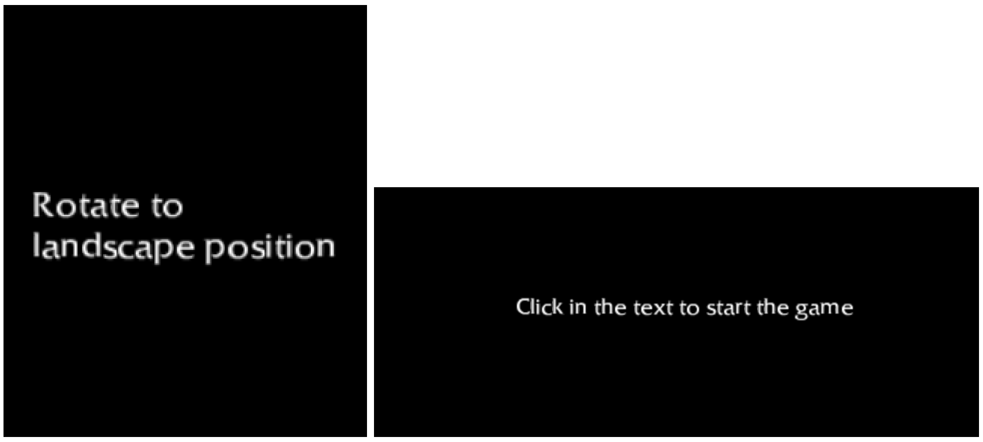
There were a set of operations that have to be implemented, in order to have the appropriate game size for the mobile platforms. Those operations are present on *boot.js* file and were performed in the following order:

1. First it would be needed to insert a starting screen after the player's client loaded this project's webpage. It was inserted a starting text, requesting to the player to click on the text to start the game.

2. The Figure 4.6(b) shows the starting text that is visualized for the computer platforms and mobile platforms that are in landscape (horizontal) position.

3. If a mobile platform would be in portrait (vertical) position, then the text would change as it can be seen in the Figure 4.6(a), requesting the player to rotate the mobile platform to the landscape position.
4. When the player would click on the text with the valid orientation requested, it was started an operation to set the game to fullscreen mode. This operation had to be done in order for the browser border to disappear, and there was the need of an input from the player since this operation can only be enabled this way by *Phaser*.
5. Depending on the game device screen size, there were different scales applied to the game size. The values used were the result of numerous tests applied on bigger and small game devices screen.
6. When all of these steps were applied, the client would send a message to the server reporting that it was ready to start the game. When there were two clients with these conditions connected with the server, the latter would sent a message back to the clients in order to load the necessary assets, for being able to start the game match.

The implementation of these operations were influenced by these two tutorials [61] [62].



(a) Portrait position **(b)** Landscape position

Figure 4.6: Different starting screen texts

4.1.8 Server authoritative

It was already discussed a type of authority that the server unfold on the game state on this chapter, which is the simulation of the units and monsters movement. In this subsection there is a discussion about some other elements that are part of a game match, on how the server could unfold authority to them and send the resulting data to the players clients.

- The server has some information regarding each client, stored in a property of their sockets. That information is kept on *socket.player* which is a struct with various attributes about the result of player actions. Some of those attributes are the player's clientID (for differentiation between the other client), number of population, total resources, state of upgrades, number of structures, super power being active and others.
- The server is responsible for operating all the results of the monster options and see if some of them such as **Trade** or **Recruit** can be made or not.
- Production of units and structures are also done by the server. To do so after checking if they are valid to be made or not, the server creates a timer with the duration equivalent to the production duration. The functions for units and structures are *unitTimer* and *structureTimer* respectively. Those game objects would only be spawn on the client side, when on the server side this timer's duration would be completed.
- If there is a production of an unit that cannot be made because it would exceed the maximum population available, the server is going to store that production order on the array *productionsTo-Process*, which is an attribute of *socket.player*. When an unit dies or a house is produced, it is going to be checked if it is possible for the unit to be produced now on the function *checkPendin-gResearches*.
- There are also timers for the research of upgrades, the duration of preparation and activation of superPowers and an interval of time to repair structures.
- The combat sequences are all simulated by the client. There are four different types of combat between game objects: unit vs unit or monster, monster vs unit, unit vs structure and structure vs game object (the 3 types referred). The functions responsible for these sequences simulate the entire combat between these game objects. It is taken in account their attributes, the distance be-tween each of them, their projectile's speed if they have ranged attack type, the effects of upgrades or super power that are present on them and some others behaviours.
- The decrement of hydroxygen resource of the Human race is also calculated by the server within a time interval. It also damages the Human's units when there is no Hydroxygen available and it prioritizes dealing damage to non-peon units first and peons secondly, in order to not prejudice the gathering of resources made by the Human player.

In the list above there were many example of the server's authority over the game state. Unfor-tunately there is a situation where the server could not be authoritative which is the start of attacks between moving objects. The reason for this is since there was no *Phaser* physics implemented on the server, because *Phaser* can only operate on the client side, it would be difficult to put the server with the responsibility of starting attacks even with it's A-star simulation. When collision avoidance and collide events between units and monster were tried to be implemented, it was even harder to make this authority possible. The solution was that the client of the unit would tell the server that when it arrived close to it's enemy, the combat could start. For monsters it would be decided by the client of their current enemy target.

4.1.9 Performance optimization

In order to had better performance for this project for being able to play this game smoothly either in computer devices and mobile platforms, some types of optimization were done for *Planetary Conquest*. There are some examples of optimization already discussed in this document such as the use of prefabs or the creation of spritesheets. Some other optimization tasks performed were the following:

- **Render option** In *game.js* file the *Phaser* game that was initially instantiated could have two render modes to be chosen which are *Phaser.WEBGL* and *Phaser.CANVAS*. The first would render the game using WebGL and the second would render using canvas. While WebGL is generally faster that canvas, which would be the case on the computer platforms, there were two problems with this approach. The first one is that WebGL performance tends to vary in some devices and mobile devices performed poorly with this graphic API. The second one would be when there was a substantial quantity of prefab objects being processed on the gaming platforms, the performance of the game would be worse on WebGL that on Canvas regardless the device used. Since there were not any 3D graphics being used on this project, WebGL was discarded for the use of Canvas on every game platform.
- **Other prefabs** There were already prefabs used for Units and Structures, since they would be reused often which increased the game's performance. Other game objects that were also used in this game would be used as prefabs as well, such as the projectiles of each unit or tower structure. The upper health bars (health bars that would be seen when the player would hover the cursor on a unit, monster or structure sprite), were also used to save performance with only one of those being created on the game manager. When the player would click on each game object sprite, the upper health bar would just change it's shape and position since only one of these objects could be hovered by the game cursor at a time. Also it was applied not only on the prefabs, but also on the other game objects present on the map the *autoCull* property. This operation checked the sprite bounds against the World Camera bounds every frame, not rendering the sprite if it was not intersecting the camera bounds. Since there were numerous objects present on the map, specially the immovable starting objects it was beneficial to the game's performance.
- **Text usage** The function that was used in some tutorials that were researched for inputting text

on *Phaser* was *game.add.text*. This operation would add a string of text to the game world, with a specified string and letter style (font, fill, alignment) that could be chosen within the styles library that *Phaser* disposes. However using this function would be harmful to the game's performance [63]. The solution would be using Bitmap fonts instead, adding text with the function *game.add.bitmapText*. This function needed a bitmap font in order to place the string requested into the game world, so it was created two bitmap fonts using the website application *Littera*[15]. Those fonts were loaded in the *load.js* file and placed on the images folder.

- **Dead animations** Instead of using particle systems to animate the defeat of the game objects, it was used dead animations that were found on the assets researched. Also those dead animations would be used as prefabs as well.

- **Reduce image assets size** To reduce the memory used by the game devices when storing the assets loaded, it was used the website application TinyPNG[16], in order to compress these images to reduce their file size.

- **Phaser.Image** *Phaser.Sprite* is what is often used to display textures rendered to the game world. These can contain operations such as physics or animations. However *Phaser* offers a similar method of displaying a texture, that does not need the use of physics or animations. Those are *Phaser.Image*[17]. The advantage of using this latter method instead of sprites without the conditions referred, is since it has less properties it ends being lighter on the memory allocated for this object, increasing the game's performance as well.

- **Normal sprites** The use of the isometric plugin on game objects had to be abandoned on Units, Structures and Monsters creation, since it was prejudicial to the game's performance. Instead normal sprites would be used because of this issue. However without the use of isometric plugin on sprites, it was not possible to use isometric physics for the movement of units and monsters. The solution for this problem was still using the plugin, not for sprites in general but just for 2 functions. One was *game.iso.unproject(2DPoint, 3DPoint)*, which would change a set of Cartesian coordinates (2DPoint) to isometric coordinates (3DPoint). The other was game.iso.projectXY(3DPoint, 2DPoint) which would do the opposite explained. With the use of these two functions it was possible to have isometric map positions for units, monsters and structures; and velocity movement using isometric coordinates for the first two game objects, without any of these three objects being a *isoSprite*.

[15]`http://kvazars.com/littera/`
[16]`https://tinypng.com/`
[17]`https://phaser.io/docs/2.6.2/Phaser.Image.html`

4.2 What went wrong

This section discusses what went wrong during the development of this project, indicating some tasks that could not be implemented and what were the approaches made to try to tackle those problems.

1. **Collisions and Object avoidance** The problem which was given much time to solve, but still could not be accomplished was Collision and Avoidance with moving game objects. The starting approach for this problem can be found on this tutorial [64]. To avoid collision with units or monsters with each other, these game objects had an ahead vector, which is a vector with a given magnitude that would point to the direction those game objects were heading. If those objects were moving, on each update frame it would be checked if their ahead vector was colliding with another game object. The object would change their trajectory for some moment, by rotating their ahead vector by a given angle. After the change of trajectory would be made the game object would resume it's movement, going to it's path end position.

   While this approached had favorable results to some extend, when it was tested with small groups of units such as 2 units in a formation avoiding colliding with other 2 units; there were some problems when tested with more units. One of those would be that units could not avoid collisions, when there were bigger unit formations groups trying to avoid colliding with each other, some of those would even get stuck with other units, since their velocity vector would have the opposite direction of the objects they were colliding. Another one would be when those units would be fighting against each other, sometimes they would push other units to attack their desired enemy target. In addition collision and avoidance behaviours would be exclusive done on the clients, since the server could not access to the *Phaser* physics engine.

   Even with either cutting collision events or avoidance movement with these game objects, there were still problems that would persist and it was prejudicial to the game performance, when there were a substantial amount of units present on the game map. There was no other choice to refrain from having collision and avoidance events.

2. **Fog of War** This was another problem that had substantial time to be solved, however unlike the first one presented above there were attainable results for this task. Still it was not possible to add it to this project. It was influenced by the solution that can be found in this website [65]. To have the Fog of War be placed on the game map, it would be used two *bitMapData*[18] objects that would cover the entire map. One *bitMapData* would be all covered in black color representing the area of the map that was not explored yet by the player. The second one would be painted in gray with having some transparency (by modifying the alpha value of this *bitMapData*), and it would represent the areas already explored by the player, that were not being visible at the moment since

[18]https://phaser.io/docs/2.6.2/Phaser.BitmapData.html

there was no player's unit or structure nearby those areas. The first *bitMapData* would be in front of the second one.

With the *bitMapData* objects created the Fog of War could be changed. Every time an unit or structure would spawn or an unit would move, the Fog of War would be dissipated from the map. The approach used to solve this problem was having during each update frame, check if the Fog of War had to be changed or not and whose *bitMapData* would suffer a change.

The first *bitMapData* would only be changed if the player did not explored that map region yet. What would be added here was a circle, which center corresponded to the unit or structure position and that circle would uncover a portion of the map. This was possible by using 2 canvas operations of the *bitMapData* for the attribute *globalCompositeOperation*, which were *destination-out* and *source-over*. The attribute *globalCompositeOperation* permits the use of composing operation to canvas, when drawing new shapes on it [66].

The first operation *destination-out* will permit that when a new shape is drawn on top of the canvas, it will remove any content that is overlapping with the new shape drawn. In the case of the Fog of War a circle would be drawn on a center of an unit or structure, removing the part of the *bitMapData* that intersected this shape. This operation was used both on the first and second bitmaps when an unit would walk to a location of the map that was unexplored.

The second operation *source-over* would be used after the first one, indicating for the canvas to return to it's default setting. Each time a shape would be drawn on top of the canvas that shape would be placed on top of it, without any other alterations. This was used on the second *BitMapData*, to allocate a new circle each time a location of the map that was already explored could not being seen, since player's units and structures were not occupied that location.

While this method worked on this project, the biggest issue was the impact on the game's performance it was having. There were two major problems with this implementation. One was having to draw multiple circles over time at least in the second *bitMapData*, so at some time within the game match, there would be a lot of memory being allocated for multiple circles created by the player units. The second one would be, instead of having multiple circles allocated, the second *BitMapData* would be cleared and redrawn on each frame, however this consumed a lot of processor power from the game device, resulting in massive frame drops. Even with the second *BitMapData* being discarded, with the first the only one remaining it would still be prejudicial to the game's performance, especially on mobile devices.

There were also another approach tried which was the use of **alpha masks** as it can be seen in this example [67]. However this mask could only be applied to a sprite, removing the content that was not inside of the shape used as a mask. This would not work for the bitMapData in fog of war. If that method would be applied it would draw black circles on the game objects positions, revealing the rest of the map and removing the rest of the bitMapData content. There is another method which is the opposite of this one called **inverted masks** that could do the desired Fog of War behaviour. However inverted masks are not implemented on the *Phaser* framework version used for this project. Thus for these problems fog of war was not implemented.

3. **Quantity of units** If there were a substantial quantity of units present on the map, the game would have some performance problems especially on mobile devices. To solve it the maximum population that a player could have is 50, for reducing the amount of units a player can have.

4. **Starting implementation approach** Before the server was implemented in this project, the starting approach used for developing *Planetary Conquest* was implementing the entire game functionality first on the client, then on the server. This was heavily influenced on how it was done on the HTML RTS Book [47]. The reason this was a big mistake is because it unnecessarily increased the implementation work of this project, since not only the server behaviour needed to be added but also some components of the client had to be removed. If those components were not removed, the client would also have authority over some events of the game, which had to be exclusive to the server. If the server was implemented first by implementing the functions for the game events on this side first-hand, then the implementation part on the client side secondly, it would save a lot of work time.

5. **Project module** Being influenced by the tutorials before the game development, the modules that were added after the starting game states were the game manager - *play.js* and the interactive game objects that are used on the game - *unit.js*, *monster.js* and *building.js*. The problem with this approach was that there were so much scripting on these classes that was difficult to organize them, also taking quite some time to debug some game events when they were not working the way they should. If some of the information of those scripts would be divided in other files such as interface, common unit and monster behaviour, combat events and others; it would facilitate which project's components had to be on each class and the debug process would be easier to be solved.

4.3 Discussion

This chapter presented the implementation that was part of *Planetary Conquest*. It was shown the most important tasks that were part of the project development, the detailed information of the implementation of some of them and what were the tasks that could not be implemented reporting the causes of it.

The next chapter shows how the Game Design and the Implementation factors contributed to fulfill the main objectives of this thesis, by observing the **User Tests** performed.

5 User Tests

Contents

This chapter presents the **User Tests** performed with the Focus Group of this project and the results that were obtained with an analysis of them.

It was imperative to perform User Tests on *Planetary Conquest*, for being able to observe the game experience that is generated by the users playing the game's project. These tests will help to identify the components that are contributing more to the game experience and try to find an equilibrium on parameters that would be analyzed [9, pg 253]. An example of a parameter analyzed is that if a resource that can be gathered in this game, is being spent way more often by the players than the other two resources available.

To explain what was done during these tests a discussion is presented about the **Conditions** that these test had. These include the types of people that were chosen as a Focus Group, where it was performed, what it would be tested, how it was tested and how the results were gathered.

Then there is a discussion about the **Results of User Tests** that were preformed during two prototypes available during development: The **Alpha prototype** and the **Beta prototype**. On each of them is referred what was the state of the game project when they were tested and what were the elements that were important to be checked.

This chapter ends with a **Discussion** of the tests performed, making an overview of what was concluded with the results obtained.

5.1 Conditions

This section starts to describe the **Focus Group** that participated in the User Tests performed for this project. It will be discussed what was tested on the sections **Alpha Prototype** and **Beta Prototype**.

5.1.1 Focus Group

There were 20 users that were part of the Focus group gathered, having participated in 13 user tests. Each person had different characteristics between each other such as age, job, education and gaming experience with RTS and computer and mobile games in general. It was important to have a group of people with heterogeneity characteristics, in order to increase having different gaming experience within the tests performed.

5.1.2 Where it was tested

The Alpha User tests were mostly performed in the developers house, since the participants were already accustomed with it, not having any problems or anxiety to that place. However the Beta User tests were mostly performed on the **Instituto Superior Técnico - Campus Taguspark** during the MOJO

2019 event organized by the Games Laboratory [68]. This event consists on a showcase of games developed by students of IST, and the people around the campus could experience those games. **Planetary Conquest** was on of the games showcased during that event.

5.1.3 How it was tested

Each test was performed by two participants, one being the Orghz player and the other being the Human player. Before the players started the test, it was explained why these tests were being performed and what they would do in these tests: The participants would be playing for the maximum of 15 to 20 minutes (Alpha and Beta prototype limit time) for trying to defeat their opponent. The reason for a time limit is because it could be excessive to play for more than that time, since the concentration of the players could be decreased resulting in a poorer gaming experience. Also with limiting the time of each test performed, there would be an observation on the preferred actions that the players were executing during the game, in order to be able to achieve victory.

Before commencing the tests 5 minutes were used to explain the contents that could be found in game. There was also an explanation on the basic actions that the players could do such as gathering resources with peon units, constructing a structure, choosing monster options and attacking an enemy unit.

Afterwards the tests would start. The participants could ask some doubts about the game functionality they could not understand. However they were not allowed to think aloud, since it could be said the strategy they were thinking which would be recognized by their opponents.

For analyzing each test it were applied 3 methods: **Observation**, **Questionnaires** and **Server Log Files**. The first test method **Observation** consisted on looking what the players were doing on each game device screen and taking notes about some events during the experiment. Those events include their visible emotions, if a game event was not behaving properly, if there were problems with the usability of the interface and some others.

The second test method was the application of **Questionnaires**, which are a set of questions that would be answered by the participants at the end of the User tests performed. The questionnaire contains 4 sections: The first one consisted on questions about the characteristics of the tester, such as age, job, studies and the experience of playing RTS and computer and mobile games. The second one consisted on closed questions about the game elements of this project, such as the race played, the opinion about the usage of each resource available, the monster options, resource costs of units and structures and others. The third one is about the game experience, where there are questions about how focused the player was during their actions, how quickly the player could thought about the actions it wished to perform, the player's opinion about the game interface and controls, the feelings the player felt when it was playing the game and if the game posed some challenge to their gaming capabilities. The questions present in this section were influenced by these references [69] [70]. The fourth one consisted on two open questions: asking if any element of the game was not understood completely, with the other question being if there was any element of the game that should have behaved differently. There were two questionnaires made, one for the **Alpha Prototype** and one for the **Beta Prototype**. The draft for the Alpha and Beta prototype respectively can be accessed by these links [71] [72].

The last test method used was **Server Log Files**, that consisted on the recording of some events that were part of a the game match between 2 user testers, being written in text files. This method was used to record some game events such as unit and structure production or combat results, that were difficult to measure just by using the former two test methods. What was recorded on each file will be discussed on the results of both prototype tests.

5.2 Alpha Prototype

This was the first prototype to be tested. This prototype only enabled the users to construct the first four structures (Command Center, Barracks, House and Tower). Also this prototype did not contained any upgrade available for any unit or race, ancient resource deposits were not available at the middle of the map, Human *Hydroxygen* timer was not functional and the monster Gnomish Airplane was not present on the map. Furthermore since the game design was at early stages of development, each type of units and structures available had identical resource costs and attribute values regardless the race chosen (for example the Orghz infantry unit had the exact same military power and resource costs that the Human military infantry). The reason for keeping this project with few features available, was for observing what would be the most important actions that the users would do with limited actions available. Since it was the first set of tests being performed, it would be useful for the users to learn how to play the game in these conditions. Every player would start with a peon unit and Command Center respective to their race, with 150 quantity for each of the 3 resources. At this experiment the production of infantry units would need *Crystal* and *Nitrogen* resources, while the ranged units would need *Crystal* and *Hydroxygen*.

The expected results for these tests were: The users knowing how to perform some basic game events that were fundamental to build a military army. Some of those basic events include producing units and structures, gathering resources across the map, use the monster options to their advantage and being able to easily and quickly navigate over the game's interface for effectuating the action they desired. There were 4 tests effectuated with 7 participants, all of them using computer platforms and being played on the Google Chrome web browser. The conditions used were exactly the ones stated on the subsection **How it was tested** of this chapter. Bellow there is a description on the results obtained with the three test methods used for the user experiments.

Observation

The first emotion that was noticed by some of the participants of this set of tests, was that they were a little anxious at the start of the experiments, with the major cause being the 15 minute time constrain for the duration of the tests for defeating their opponent. However they began to be more relaxed after seeing their military power increased by the actions they were performing during the game match. There was some help required by the participants, in some aspects of the game that were not understood completely during the pre-testing phase. Fortunately few doubts were asked by them, which were also quickly solved. Other factors observed included that the players were not much attentive to what their opponent would be doing, being much more concentrated on the actions they were doing. Also some game control options were rarely used, which were: the button used for selecting idle peons and using keyboard shortcuts to bind a key number to a formation of units.

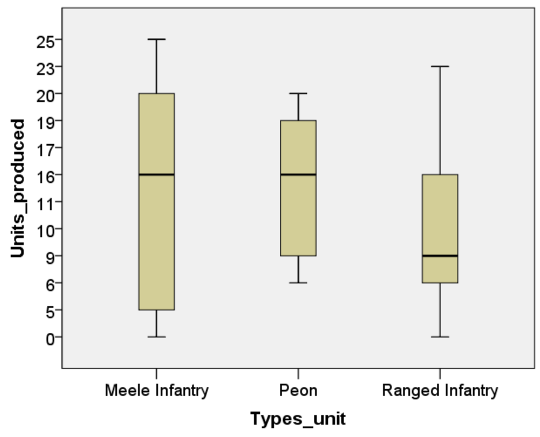
Questionnaires

The link to the responses resume, effectuated by user participants can be found here [73]. In the first section of the questionnaire it can be seen that the users differed on the experience in playing RTS games, but most of them were at least experienced in playing computer or mobile games. About the second section referent to the actions performed in-game, most users found that the *Crystal* resource was much more important to use than the other two resources and the map size was not adequate to construct numerous structures. Also from the 3 available monster options the *Recruit* option was quite used and useful to the users, while the other two options, in particular the *Pillage* option not so much. For the third section it could be observed that most players were focused on the actions they were performed, although some of them were distracted during their actions. Most of the players knew the actions they wanted to perform during the experiments and knew how to operate with the game's interface for the execution of those actions. Regarding their feelings the users liked to play the game and felt that it posed some challenged to their game capabilities. For the fourth section, there were some players that did not understand completely the behaviour of some game elements. Also they thought that the game needed to include Fog of War behaviour and the interaction with the monsters could be differently made.

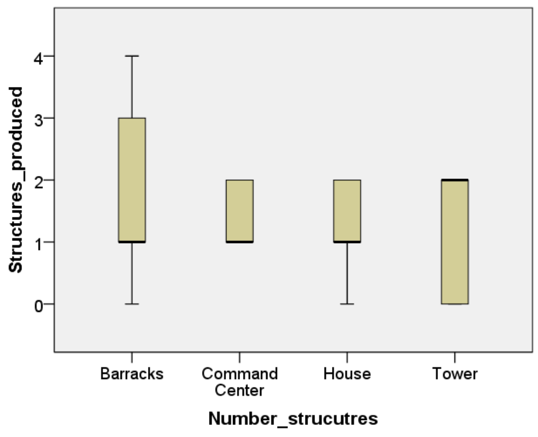
Server text logs

By observing the data including on the server text logs, the conclusions were the following: Regarding the **Units produced** per game, the average of peons and Meele infantry units produced were close from each other being 13.5 and 13 respectively, however there were cases where the military infantry were slightly higher than the peons produced and on others slightly lower. The number of average ranged units produced was 6.8 being lower than the other 2 units, but it was still substantially used. Regarding the **Structures produced**, the average number of Command Centers and Barracks constructed were the same being 1.1, and the House was the structures with most constructions having an average of 1.6.

The average constructed Towers per game was equivalent to 0.83. For the **Monster options** only the recruit option was chosen 4 times for the Ogres and 1 times for the Archer. Looking at the **Resources Gathered** per game, Crystals were by far the most resource obtained with an average of 2970.8, with the nitrogen falling much shorter being 195.8 and 33.3 respectively. Finally the average duration of each game match was 12 minutes. Bellow there are presented boxplot graphs regarding the results of those elements, on the Beta prototype tests.

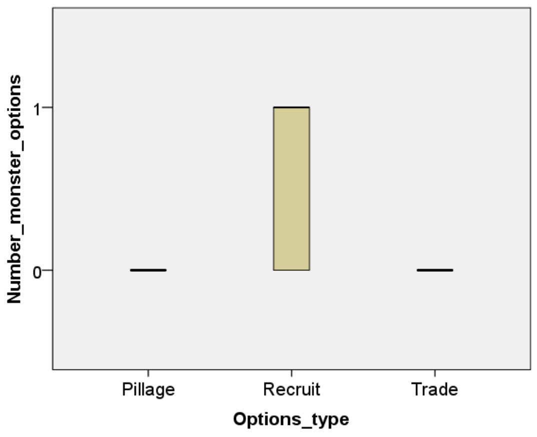


(a) Number of units produced

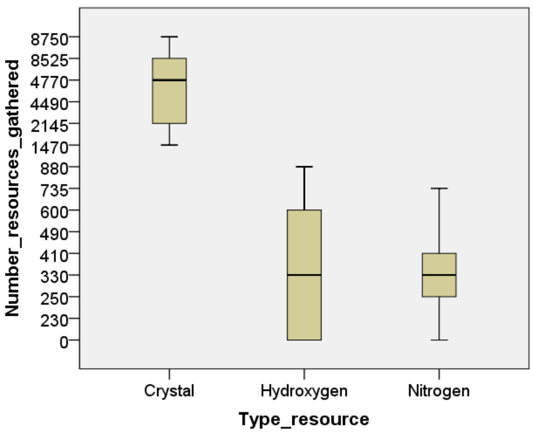


(b) Number of structures produced

Figure 5.1: Number of game objects produced (Alpha)



(a) Monster options chosen



(b) Resources Gathered

Figure 5.2: Monster options chosen and resources gathered (Alpha)

Discussion of results

There were some conclusions obtained with the results of these tests. For starters most of the expected results were met. The players could effectuate most of the actions they desired with easiness for trying to defeat their opponent, and there weren't many difficulties presented for the players to apply those actions. However there were some issues that were revised by observing the methods used for these tests. One that suffered alterations were the monster options, since the only option chosen on these experiment was **Recruit**. This option had it's resource cost increased during the design process of these options. The other issue solved was the map space. Instead of changing the map dimensions it was redone how the structures were placed: At the time a 2x2 structure size would check 4x4 positions, for being able to be certain that the adjacent tile positions of that structure would never intersect with occupied positions. This approach was abandoned and now each time a 2x2 structure would be produced, it would check the corresponding 2x2 tile positions that were tested to be placed upon.

5.3 Beta Prototype

For the second and final prototype of this project, all the elements discussed in the **Game Design** section such as Units, Structures, Upgrades and Monsters were implemented, with the values corresponding to what it was conceived for the design process. The usage of the Super Power, the placement of Ancient Resource Deposits and the existence of the Hydroxygen Timer for the Human race were also functional. The results expected in these tests would be the production of some of the new units and structures added, and the use of upgrades, Super Power and monster options for the Gnomish Airplane. The conditions were the same as the previous tests made in the **Alpha prototype**, with a difference: The time for the experiments would be a maximum of 20 minutes instead of 15. The time limit was extended in order for the participants to feel less anxious about this timer. There were tests performed in all the platforms: computer devices and mobile platforms (Android Smartphone and Tablets). Most of the tests performed were on Android Tablet devices. The controls of the mobile platforms were explained before the start of each experiment. There were 15 participants for this set of tests, 2 which participated in the Alpha prototype and 13 who participated during the MOJO 2019 event.

Observation

Since the majority of the tests was performed on mobile platforms, it was important to observe how the users could operate on the mobile interface of *Planetary Conquest*. While the users had some doubts at the start of the experiments, they were able to quickly grasp the functionality of mobile controls. Also most of the users understood how to make the basic actions of this game quickly, and was able to form a military army for trying to defeat their opponent's army.

Regarding the new elements added to this prototype, most of them were used during the experiment of these tests. The elements used were: Gnomish Airplane recruitment, Ancient Resource Deposits extracted, Super Power activation and some unique upgrades such as Race, Tower structure, Melee and Ranged infantry. However the use of these new elements was not common in every test performed for this prototype, with the players opting to produce the peon and infantry units and constructing the four structures that were also available on the Alpha prototype. Furthermore the production of air units by the Orghz and Human structures and the construction of the Research Building structure, were no performed in any of these tests.

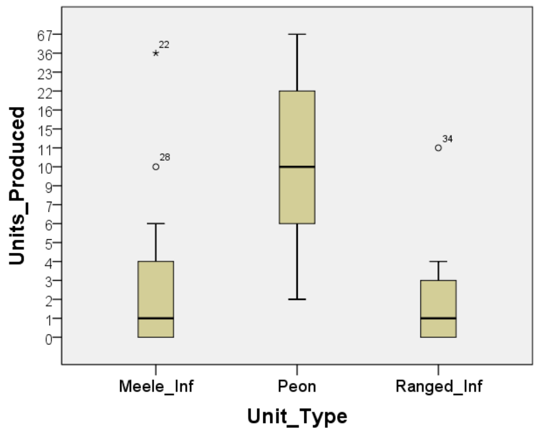
Questionnaires

The link to the responses resume effectuated by user participants can be found here [74]. On the first section it can be observed that the users seem to be more experienced in playing RTS games, and other Computer and Mobile games than the previous tests. On the second section it is observed that the resource *Crystal* is still the most useful. However the other two resources were also well received. The map space available to construct structures was more appraised than on the Alpha structure. The *Recruit* monster option was still way more useful than the other two options. On the third section the players were more focused on their action than their opponent's. Also the users presented a little more difficulty operating with the mobile interface. Still the players generally enjoyed playing *Planetary Conquest* and thought this game challenged their gaming abilities. Finally in the fourth section, most of the game elements were understood, only with one user reporting that it did not understood the aspect of some unmovable objects. Regarding the elements of the game that should behave differently there were some suggestions, such as the increase of the map size, selection of units, map tiles that could affect the units behaviour and changing some elements regarding the mobile interface.

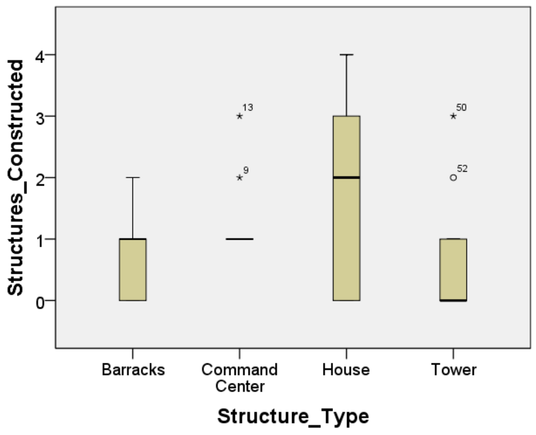
Server Text Logs

By observing the data including on the server text logs, the conclusions were the following: Regarding the **Units produced** per game, the average of peons produce was of 16.625, being by a large margin the most produced unit during these experiments. The average of Melee and Ranged Infantry was of 4 and 1.87 respectively, much lower than what was obtained on the Alpha prototype tests. Regarding the **Structures produced** the average of Command Centers produced was of 1.1 and the Barracks structure of 0.68, again shorter on what was obtained during the Alpha tests. The House structure was the one being more constructed with an average number of 1.75. The Tower structure had an average of 0.68. Referring the **Monster options** the **Recruit** options was chosen 8 times, twice the amount of the previous tests with Ogre and Archer monsters being both recruited by 3 times, and Gnomish Airplane two times. The **Trade** and **Pillage** options were only chosen once. Observing the **Resources Spent** for each race, **Crystals** was still the most used resource with an average quantity of 1229 and 1187.5 for the Orghz and Human race respectively per game. The **Nitrogen** had an average of 480 and 268, and the **Hydroxygen** resource was spent much less on the Orghz race with an average number of 318.75, than on the Human race with 924.75 respectively. Most of the **Hydroxygen** spent for the Human race was on the **Hydroxygen timer** behaviour, with some of it being used on the Monster options. Bellow there are presented boxplot graphs regarding the results of those elements, on the Beta prototype tests.

Discussion of results

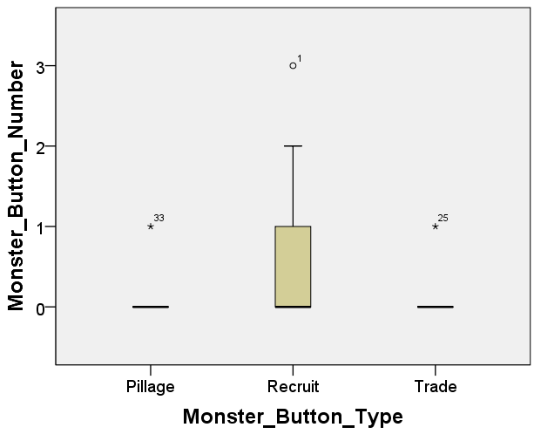


(a) Number of units produced

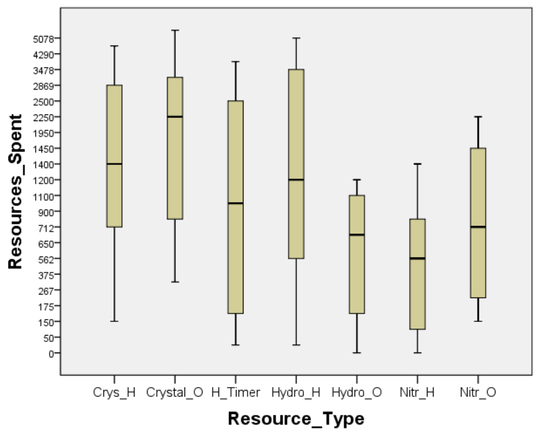


(b) Number of structures produced

Figure 5.3: Number of game objects produced (Beta)



(a) Monster options chosen



(b) Resources Spent

Figure 5.4: Monster options chosen and resources spent (Beta)

The results obtained for these tests mostly corresponded to what it was expected. *Planetary Conquest* could be played on both Personal Computers and Mobile Devices. Also the users did not presented severe difficulties on understanding the mobile interface of this game, and knew how to navigate with it in order to perform the actions they desired. Some of the features added in this prototype were not used, such as Orghz and Human Flying Units production or the Upgrades available on the Research Building Structure. Still most of them could be observed even if sporadically. If there was the possibility on having the Users that participated on these tests, being available to perform more sets of tests using this prototype, maybe they would choose the new features present on it more often, than the basic units and structures available.

5.4 Discussion

This chapter presented the User tests performed for *Planetary Conquest*.There was a description on the utility that those test would bring for this project. There was a presentation on the **Focus Group** that was gathered for this project, describing their characteristics and what were the **Conditions** that would be present on these types of test, in order to maximize the utility of them. As for the set of tests performed, the **Alpha prototype** tests had satisfactory results since the expected results matched in most part with the results obtained by the test experiments. With the different types of test methods to analysis the results obtained, it was possible to correct some of the complains received by the users feedback. On the second set of tests **Beta prototype** the results were also satisfactory, since **Planetary Conquest** could be played on the Computer platforms and Mobile devices. Also the majority of the features was used at least once, and they proved they usefulness for helping the users to have a stronger military army.

Following this chapter is the final chapter of this document **Conclusion**, that describes a global reflection of the work that enfolded the entire project's development, what objectives were accomplished and what would be the future work to be applied on *Planetary Conquest*.

6
Conclusion

Contents

This chapter discusses the **Conclusions** about the work performed for this thesis project, then it refers additional features that could be added on the section **Future Work**.

6.1 Conclusions

Reviewing the goal of this thesis: Designing and Creating a RTS game, using HTLM5 and Javascript programming languages, that can be played on both personal computers and mobile devices. Also the game has the traditional 4X elements present in this game genre and being played in multiplayer mode. By reading this document it could be seen that the main objective was accomplished.

For attaining the main objectives, the first task to be performed was a research of what is a RTS game and the elements that were part of it such as Units, Structures, Resources, 4X, interface and others. There was also a presentation on some of the most important games that are part of the history of the RTS genre, with some of them inspiring an amount of design decisions that are part of *Planetary Conquest*. Then a study was presented, regarding the benefits that could be found using the programming languages purposed and being able to be played on multiple devices. The game framework used *Phaser.io* was also shown, referring the main benefits available on it and concluding that it was possible to create a game with the conditions initially stated at the start of this thesis.

To create and design a RTS game various tasks had to be applied, since making a RTS involved various academic strands in order to create and implement a game of this magnitude. There was the game design that was conceived for this game, carefully designing each aspect that is part of it such as Units, Structures, Economy, Upgrades, 4X, Super Power and others. Then a presentation was made regarding the Architecture that is part of *Planetary Conquest*, referring the Technical Limitations present on it and how it is structured the communications between the Game Server and the Game clients. There was also a representation on how this project was organized with the use of the *Phaser* framework. For the implementation of this project there were implemented several tasks on it, discussing in this document those who contributed the most to fulfill the main objectives present in this thesis. There was also a reference on the tasks what could not be implemented, giving the reasons for it.

For being able to observe if the objectives of this thesis were accomplished, User Tests were conducted with the Focus Group chosen for this project. They were done in order to maximize the game experience of *Planetary Conquest*, see the usage of some elements that were part of this project's gameplay such as the balance between the Units, Structures and Monster attributes. Also to observe if the interface could be understood by the participant players and apply the game actions they desired with the help of the game controls and what were their emotions when they were playing the game. From the two sets of tests performed, it was observed on the **Alpha prototype** that it was possible to play a multiplayer RTS game with the main objectives present on this thesis on computer platforms, with the users understanding which actions they had to perform in order to defeat their opponents. From the **Beta prototype** it could be observed that *Planetary Conquest* can be played using both proposed devices for this project, and the majority of the introduced features on this prototype was used at some degree.

In spite of some features not being implemented such as Fog of War and Collision Avoidance between moving game objects, it was possible to create a multiplayer RTS game that can be played on multiple game platforms, with the simple use of a web browser.

6.2 Future Work

For the future work purposed for this project it would be appropriate to review some of the errors made on it, observe what it can be done to the tasks that could not be implemented and implement some other tasks that could enhance the game experience. Some of those tasks include adding game sounds, adding the ability to have 2 vs 2 player matches and having the game server capable of having authority over multiple game matches being played at the same time.

Bibliography

[1] W. Burrows, *In Grand Street 37*. W W Norton & Co Inc, 1991.

[2] A. Sartori-Angus, "Cosmic conquest," *BYTE*, 1982.

[3] D. Kosak, "Top ten real-time games of all the time," 2004, http://web.archive.org/web/20100616031405/http://archive.gamespy.com/top10/february04/rts.

[4] E. Adams, *In Fundamentals of Game Design*. New Riders, 2010.

[5] "Essencial facts about the computer and video game industry," 2015, http://www.theesa.com/wp-content/uploads/2015/04/ESA-Essential-Facts-2015.pdf.

[6] Z. D. Boren, "There are officially more mobile devices than people in the world," 2014, http://www.independent.co.uk/life-style/gadgets-and-tech/news/there-are-officially-more-mobile-devices-than-people-in-the-world-9780518.html.

[7] "Real-time strategy," http://tvtropes.org/pmwiki/pmwiki.php/Main/RealTimeStrategy.

[8] A. Emerich, "Microprose's strategic space opera is rated xxxx," 1993, http://www.cgwmuseum.org/galleries/issues/cgw_110.pdf.

[9] P. S. Carlos Martinho and R. Prada, *Design e Desenvolvimento de Jogos*. FCA, 2014.

[10] C. Driver, "Stonkers review," 2008, https://www.retrogamer.net/retro_games80/stonkers/.

[11] "Best wargame," 1984, http://www.crashonline.org.uk/12/awards.htm.

[12] "Dune 2 - the building of a dynasty," https://archive.org/details/msdos_Dune_2_-_The_Building_of_a_Dynasty_1992.

[13] B. Bates, *Game Developer's Market Guide*. Thomson Course Technology, 2003.

[14] "Warcraft review," http://www.csoon.com/issue2/WARCRAFT.HTM.

[15] *Warcraft: Orcs & Humans game manual*. Blizzard Entertainment, 1994.

[16] “Command & conquer,” http://www.giantbomb.com/command-conquer/3025-98/.

[17] M. E. Circulus, “Command & conquer review,” 1995, http://www.cgwmuseum.org/galleries/issues/cgw_137.pdf.

[18] R. Whitman, “Command & conquer remake,” 2018, https://www.extremetech.com/gaming/280629-original-command-conquer-developers-remastering-the-classic-90s-games.

[19] C. M. Jo Rabin, “Mobile web best practices 1.0,” W3C, 2008, https://www.w3.org/TR/mobile-bp/#requirements.

[20] D. Burford, “Cross-platform mobile development vs native development,” 2014, http://www.codeproject.com/Tips/816977/Cross-platform-Mobile-Development-vs-Native-Develo.

[21] P. N, “Cross platform mobile application development - advantages and disadvantages,” 2015, http://ezinearticles.com/?Cross-Platform-Mobile-Application-Development---Advantages-and-Disadvantages&id=9036883.

[22] D. Flannagan, *JavaScript - The definitive guide*, 6th ed. O’Reilly Media, 2011.

[23] J. Kyrnin, “What are markup languages,” http://webdesign.about.com/od/htmlxhtmltutorials/p/what-are-markup-languages.htm.

[24] “Html element reference,” http://www.w3schools.com/tags/.

[25] “Javascript in html,” http://www.simplehtmlguide.com/javascript.php.

[26] “Css definition,” http://www.simplehtmlguide.com/whatiscss.php.

[27] P. Bright, “Html5 specification finalized, squabbling over specs continues,” 2014, http://arstechnica.com/information-technology/2014/10/html5-specification-finalized-squabbling-over-who-writes-the-specs-continues/.

[28] S. P. Anne van Kesteren, “Html5 differences from html4,” 2011, http://www.w3.org/TR/2011/WD-html5-diff-20110405/.

[29] “Javascript html domain object model,” http://www.w3schools.com/js/js_htmldom.asp.

[30] “What javascript can do for you,” https://www.w3.org/community/webed/wiki/What_can_you_do_with_JavaScript.

[31] “Javascript compability table,” http://kangax.github.io/compat-table/es5/.

[32] N. Hamilton, “The a-z of programming languages: Javascript,” 2008, http://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/.

[33] K. Nasir, "Developing a 2d game? here's why html5 is the best choice," https://www.htmlgoodies.com/html5/other/developing-a-2d-game-heres-why-html5-is-the-best-choice.html.

[34] "Phaser.io github," https://github.com/photonstorm/phaser/tree/v2.6.2.

[35] "Phaser examples site," http://www.phaser.io/examples/v2/input/follow-mousex.

[36] "Html5 game devs," http://www.html5gamedevs.com/.

[37] "Javascript game engines," 2014, https://github.com/showcases/javascript-game-engines.

[38] Bateman and Boon, "Game design definition," 2006.

[39] P. Santos, "Game design and development," 2015, https://fenix.tecnico.ulisboa.pt/downloadFile/563568428717813/intro_game_development2015_handouts.pdf.

[40] T. editors of of Encyclopaedia Britannica, "Isometric drawing," https://www.britannica.com/topic/isometric-drawing.

[41] A. Kovalenko, "Isometric projection," 2017, https://medium.com/gravitdesigner/designers-guide-to-isometric-projection-6bfd66934fc7.

[42] "Sprite sheets assets," https://www.spriters-resource.com/pc_computer/warcraft2/.

[43] R. Prada, "Progression in games," 2016, https://fenix.tecnico.ulisboa.pt/downloadFile/845043405448407/Progression.pdf.

[44] "Freewee game architecture," https://github.com/christabella/freewee#system-design.

[45] R. Silveira, "Multiplayer game programming," 2015, https://hub.packtpub.com/getting-started-multiplayer-game-programming/.

[46] J. Renaux, "Phaser multiplayer tutorial," 2017, https://www.dynetisgames.com/2017/03/06/how-to-make-a-multiplayer-online-game-with-phaser-socket-io-and-node-js/.

[47] A. R. Shankar, *lock-step method*, 2nd ed. Apress, 2017.

[48] G. Gambetta, "Authoritative server," https://www.gabrielgambetta.com/client-server-game-architecture.html.

[49] "Phaser state class documentation," https://phaser.io/docs/2.6.2/Phaser.State.html#render.

[50] "Phaser 2 tutorial," https://phaser.io/tutorials/getting-started-phaser2.

[51] "Phaser tutorial," http://phaser.io/tutorials/making-your-first-phaser-2-game.

[52] A. R. Shankar, "Pro html5 games - book," 2015, https://www.adityaravishankar.com/pro-html5-games-first-edition/.

[53] M. W. Johnson, "Spritesheet creation using gimp," 2012, http://imagine.kicbak.com/blog/?p=114.

[54] "Phaser isometric plugin," http://rotates.org/phaser/iso/.

[55] C. Kaitila, "A-star pathfinding for html5," 2013, http://buildnewgames.com/astar/.

[56] J. M. Dias, "Pathfinding part 1," 2015, https://fenix.tecnico.ulisboa.pt/downloadFile/1970943312265443/Pathfinding%20Part%201.pdf.

[57] "A-star phaser example," https://developer.tizen.org/community/tip-tech/using-easystar.js-implement-pathfinding-tizen-game-projects.

[58] J. M. Dias, "Pathfinding part 2," 2015, https://fenix.tecnico.ulisboa.pt/downloadFile/563568428715746/Pathfinding%20Part%202.pdf.

[59] A. Kamath, "Quadtree explanation," https://www.geeksforgeeks.org/quad-tree/.

[60] "Spatial hashing tutorial," 2009, https://conkerjo.wordpress.com/2009/06/13/spatial-hashing-implementation-for-fast-2d-collisions/.

[61] E. Feronato, "Lock orientation in your html5 game," 2015, https://www.emanueleferonato.com/2015/04/23/how-to-lock-orientation-in-your-html5-responsive-game-using-phaser/.

[62] ——, "Scale game screen on mobile devices," 2015, https://www.emanueleferonato.com/2015/03/25/quick-tip-how-to-scale-your-html5-endless-runner-game-to-play-it-on-mobile-devices/.

[63] "Difference between text and bitmaptext," 2018, http://www.html5gamedevs.com/topic/38100-text-vs-bitmaptext-performance-the-conclusion.

[64] F. Bevilacqua, "Collision avoidance between game objects," 2013, https://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-collision-avoidance--gamedev-7777.

[65] M. Kahn, "Fog of war approach," https://codepen.io/zyklus/pen/prvnb.

[66] "Canvas composition operations," https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D/globalCompositeOperation.

[67] "Alpha mask," https://phaser.io/examples/v2/bitmapdata/alpha-mask.

[68] "Mojo 2019 event," https://tecnico.ulisboa.pt/pt/eventos/mojo-2019-montra-de-jogos-do-tecnico-12a-edicao/.

[69] d. K. Y. . P. K. IJsselsteijn, W.A., "The game experience questionnaire," 2013, https://pure.tue.nl/ws/files/21666907/Game_Experience_Questionnaire_English.pdf.

[70] M. J. S. Bob G. Witmer, “Measuring presence in virtualenvironments: A presencequestionnaire,” 1998, https://nil.cs.uno.edu/publications/papers/witmer1998measuring.pdf.

[71] “Alpha prototype questionnaire,” https://docs.google.com/document/d/1BOh_4aY8QFqSLuXPo6-R5sYeU_aeNYM7J-ciPmTHfko/edit?usp=sharing.

[72] “Beta prototype questionnaire,” https://docs.google.com/document/d/1gYgAZIMiZmYwdEIE0cSiavg1N604mbEvPPbOzytU7mo/edit?usp=sharing.

[73] “Alpha prototype responses,” https://docs.google.com/forms/d/1ofN1DikaQmGLsyWfFWHZ63B7oIkIIYwqYrIxeIByfk
viewanalytics.

[74] “Beta prototype responses,” https://docs.google.com/forms/d/15iiZJgXc68zeXVoRbZrLTcoW9dvHhQIIK9kv4JGcdH
viewanalytics.

[75] D. Silverman, “Map design,” 2012, https://gamedevelopment.tutsplus.com/tutorials/starcraft-ii-level-design-introduction-and-melee-maps--gamedev-3304.

[76] T. Doll, “Multiplayer map design,” 2015, https://waywardstrategy.com/2015/06/07/time-as-a-resource-part-2-multiplayer-map-design/.

A Annexes

A.1 RPS

One design pattern, present in the creation of each Unit and Monster is the Rock-Paper-Scissors (RPS) [9, page 203]. This is a well known game where is played between two players and each of them picks one of the available choices 'Rock', 'Paper' or 'Scissors'. The choices are revealed simultaneously, if the choices are the same for both players it's a tie, otherwise it wins the player that picked rock over scissors, paper over rock or scissors over paper.

This model is used in many games to guarantee that there is a symmetry between the available strategies in a game, and none of them is a dominant strategy. This is important because if there would be a dominant strategy all the players would exploit it, and the game would become tedious and boring.

However the normal model of RPS for RTS games is still not sufficient. In RTS games the player usually sends some of it's units to spy theirs opponents, and see which kind of units they are producing. With this information they produce other type of units that counters the opponent current units. The problem with this approach is that the players will only care of micromanagment (component of efficient management of produced units) over macromanagment (component of building an efficient global strategy, such as army composition, large manoeuvres, expansion, relations between economy investment and military units) [9, page 204]. So variation of a model of RPS is needed, that increases the interest of RTS games using this model. Of the existing models, one appropriate for a RTS game is the **RPS diffuse model** [9, page 206].

In the diffuse model the advantages and the disadvantages of each unit usually aren't direct or absolute (for instance a spearman infantry unit deal double damage versus a mounted knight unit), but various parameters such as speed, type and quantity of damage, defence, health points, special powers, terrain effects and others are different for each game object. It's up to the player to use for his benefit the advantages that each game object offers. If the player can play tactically well, he can have a type of game object win against other type of game object that is stronger.

In order to balance the Units, Monsters and Tower structures using this model, it was made various combat tests between these elements and tweaking some of their combat values. Also in the user tests performed there was retrieved information results of the test experience from the server, to observe if any of these game objects had balance issues or not.

A.2 Unit Combat Formula

In order to calculate the strength of an Unit, Monster and the Tower Structure it was used a formula for it. The strength calculated would represent the combat value of those game objects.

After numerous theoretical tests and changes on some values, the formula is the following:

MH = Maximum Health; AP = Armor Points; AP = Attack Points; AS = Attack Speed; AR = Attack Range; MS = Movement Speed; CV = Combat Value

$$MH \times (1 + AP/6) \times AP \times AS \times (AR \times 50 + MS) \times 0.1 = CV \tag{A.1}$$

Analyzing some values on the formula presented starting by Armor Points, the value indicated was the best possible approximation to calculate an unit survival, together with Maximum health points. The attack range value is multiplied by 50 because of the unit's range points that can go from 1 to 3, each point is equivalent to 50 pixels on the game screen. Finally for the sum of the attack range and movement speed of an unit, since the value is usually much greater that any of the other attributes, the sum is reduced to 10% on this formula.

The calculations for the Tower structure are identical tp the Unit and Monster calculations. However the movement speed is set to 0 since structures do not move.

Unit and monster combat value cannot be directly compared to structure value (structure does not have movement speed and can be repaired), also some unit upgrades are not being included in the formula, although there are some calculations with the effects of each update at the end of the Annexes.

Orhgz Warrior and Wyvern have slight better combat value than Human Soldier and Eagle Knight, Human Wizard and Cavalier have slight better combat value than Butcher and Shadow Knight.

There was also conducted multiple tests between the different types of each race, to observe if some sort of unbalance would exist between them. The results are bellow the Information Values subsection of each game element.

A.3 Combat Value Tables

Acronyms:

N - Name

HP - Health Points

ATK - Attack Points

ARM - Armor

ATR - Attack Range

MSP - Movement Speed

MOT - Movement Type (G - Ground, F - Flying)

RC - Resources Cost (C - Crystal, N - Nitrogen, H - Hydroxygen)

TRS - Total Resources Spent

PV - Population Value

RT - Research Time (in seconds)

CV - Unit Combat Value

CV/(T*P) - Unit Combat Value / (Total Resources Spent * Population Value)

A.3.1 Orghz Units and Tower

Table A.1: Orghz Units and Tower attributes

Name	HP	ATK	ARM	ATR	MSP	MOT	PV	RC	RT	CV	CV/(T*P)
Peon	40	3	0	1	90	G	1	50 C	10	1680	33.6
Warrior	70	7	0	1	100	G	1	100 C, 50 N	14	7350	49
Butcher	70	6	0	2	95	G	1	125 N, 50 H	14	8190	46.8
Shadow Knight	155	14	0	2	115	G	2	250 C, 200 H	24	46655	51.83
Wyvern	190	17	2	1	115	F	2	325 N, 325 H	29	71060	54.66
Tower	110	11	1	2	n/a	n/a	n/a	175 C, 100 H	35	14116.6	51.3

Total resources for 1 copy of every unit (minus tower) :
450C + 500 N + 575 H = 1525, Production Time Total: 91s

A.3.2 Human Units and Tower

Table A.2: Human Units and Tower attributes

Name	HP	ATK	ARM	ATR	MSP	MOT	PV	RC	RT	CV	CV/(T*P)
Peon	40	3	0	1	90	G	1	50 C	10	1680	33.6
Soldier	80	5	1	1	100	G	1	100 C,	15	7233.33	72.133
Wizard	55	8	0	2	95	G	1	125 N	15	8470	67.76
Cavalier	155	15	1	1	115	G	2	250 C, 125N	25	47600	73.23
Eagle Knight	155	18	1	2	115	F	2	200 N, 250 H	30	69982.5	77.75
Tower	120	10	1	2	n/a	n/a	n/a	200 C	35	14000	70

Total resources for 1 copy of every unit (minus tower) : 550 C + 500 N = 1050
Multiplying by 1.5 (hypothetically counting Hydroxygen resource) : 1575, Production Time Total: 95s

A.3.3 Monster Units

Acronyms:

TRA - Trade (1st resource for 2nd resource)

REC - Recruit (recruit 3 types of monster)

PIL - Pillage (when killed each monster drops resources)

CV * 3 / (REC * PV) - Combat value * 3 recruit units / (Recruit resources * Population Value)

Table A.3: Monster attributes

Name	HP	ATK	ARM	ATR	MSP	MOT	PV	RC	CV	CV * 3 / (REC * PV)
Ogre	90	8	0	1	90	G	1	500 C	12180	73.08
Archer	75	8	1	2	95	G	1	300 N, 200 H	11700	70.2
Gnomish Airplane	175	17	2	2	105	F	2	350 C, 550 N, 650 H	81316.6	78.69

Total resources to recruit one group of 3 elements, of each unit type: 850 C + 850 N + 850 H = 2550

Counterparts (Monster receives unique upgrade of race unit):

Ogre: Warrior or Soldier; **Archer**: Butcher or Wizard; **Gnomish Airplane**: Wyvern or Eagle Knight

Each of the monster has better combat value than their unit counterparts, since the monsters recruited are limited.

A.3.4 Orghz Strucutres

CT - Construction Time (in seconds)

Table A.4: Orghz Structures attributes

Name	HP	AR	RC	CT
Command Center	400	3	275 C, 200 H	40
Barracks	300	2	200C, 125 N	35
House	200	0	100 C	25
Tower	100	1	175 C, 100 H	35
Factory	300	2	225 C, 150 N	35
Research Building	250	1	225 N, 100 H	30
Wyvern Nest	250	2	175 C, 175 N, 175 H	30
Power Building	300	1	200 N, 250 H	35

Resources Total: 1150 C + 875 N + 825 H = 2850, Constructing Time Total: 265s

A.3.5 Human Structures

Table A.5: Human Structures attributes

Name	HP	AR	RC	CT
Command Center	400	3	300 C, 100 N	40
Barracks	300	2	150 C, 50 N	35
House	200	0	100 C	25
Tower	120	2	200 C	35
Factory	300	2	150 C , 100 N	35
Research Building	250	1	50 C, 150 N	30
Wyvern Nest	250	2	150 C, 250 N	30
Power Building	300	1	100 C, 200 N	35

Resources Total: 1200 C + 850 N = 2050, Constructing Time Total: 265s
(With Hydroxygen approximation) 1025 + 2050 = 3075

A.3.6 Orghz Weapons and Armor Upgrade

Acronyms:

UT - Upgrade Type

LV - Level

RC - Resource Costs (C - Crystal, N - nitrogen, H - Hydroxygen)

RT - Research Time (in seconds)

Table A.6: Orghz Weapons and Armor Upgrades attributes

UT	LV	RC	CT
Melee Weapon	1/2/3	200 / 300 / 375 C, 150 / 250 / 325 N	45 / 60 / 75
Ranged Weapon	1/2/3	250 / 350 / 425 N, 100 / 200 / 275 C	45 / 60 /75
Armor	1/2/3	150 / 250/ 350 C, 150 / 250 / 325 N	75 / 90 / 105

Total Resource spent for every upgrade at maximum level: 1725C + 1750N + 1725H = 5200, Production Time Total: 540s

A.3.7 Human Weapons and Armor Upgrade

Table A.7: Human Weapons and Armor Upgrades attributes

UT	LV	RC	CT
Melee Weapon	1/2/3	150 / 250 / 300 C, 50 / 125 / 200 N	40 / 55 / 70
Ranged Weapon	1/2/3	150 / 225 / 300 N, 50 / 125 / 200 C	40 / 55 / 70
Armor	1/2/3	150 / 250/ 350 C, 150 / 250 / 325 N	70 / 85/ 100

Total Resource spent for every upgrade at maximum level: 1800C + 1800N = 3600, Production Time Total: 585s

(With Hydroxygen approximation) 1800 + 3600 = 5400

A.3.8 Orghz Unique Upgrades and Super Power

Race skill: Wall Breaker - Units deal 2 more damage to buildings.

Tower: Precision Shot - Increase tower attack range by 1.

Orghz Warrior: Brute Force - Reduce enemy target units or monsters Armor to 1 when attacking them, for 2 seconds.

Orghz Butcher: Poison Corrosion - Poisons enemy unit or monster dealing 2 damage per 2 seconds, up to 3 stacks. A stack is lost per poison damage and increasing 1 damage per stack. Each Butcher attack adds a poison stack to the enemy unit. Ex: (In 6 seconds without being attacked by a Orghz Butcher, damage will be 4 - ¿ 3 - ¿ 2, being 9 damage in total).

Orghz Shadow Knight: Soul Drain - Recovers 2 HP per Shadow Knight attack if this unit is injured against units or monsters. If this unit has full health points it deals plus 2 damage against enemy units or monsters instead.

Orghz Wyvern: Infernal Armor - Deals 2 damage every time an unit, monster or tower attacks Orghz Wyvern.

Orghz Super Power: BloodLust - Increases affected units attack points by 3.

A.3.9 Human Unique Upgrades and Super Power

Race skill: Improved Development Tools- Construction time of structures and research time of all upgrades is 20% faster.

Tower: Fortified Plating - Increases Tower's armor by 1 and HP by 40.

Human Soldier: Shield Bash - Decreases attack points of an enemy unit or monster by 1 for 2 seconds.

Human Wizard: Bind Shackle - Decreases movement speed of an enemy unit or monster by 20% for 2 seconds.

Human Cavalier: Equestrian training - Increases Cavalier's Movement Speed by 10 and armor by 1.

Human Eagle: Electric Current - Deals 3 damage per second to an enemy unit, monster or building, ending at 4 seconds. This effect is applied by Eagle Knight's attack.

Human Super Power: Indomitable Spirit - Places a 35 HP shield for affected units. Each time an affected unit is hit, the shield is decremented by the value of enemy object attack points. If the remaining shield points are lesser than the damage inflicted by an attack, the difference is afflicted to the unit's health points. The unit's armor points are not evaluated, to reduce the attack damage while this super power is active.

Upgrades are not cumulative, for instance Brute Force can only reduce the armor of an enemy by 1, even if 2 or more Orghz Warriors are attacking the same enemy object.

A.3.10 Unique Upgrades Resource Costs and Combat Value Increases

Acronyms:

UT - Upgrade Type

RC - Resource Cost (C - Crystal, N - Nitrogen, H - hydroxygen)

RT - Research Time (in seconds)

AFF - Affected units or tower

OCV - Old combat value (before upgrade)

NCV - New combat value (after upgrade)

- For Orghz

Table A.8: Orghz unique upgrades attributes

UT	RC	RT	AFF	OCV	NCV
Wall Breaker	100 C, 150 N	45	Every Orghz Unit	n/a	n/a
Precision Shot	100 C, 150 H	45	Tower	14341.46	21513.96
Brute Force	150 C, 100 N	50	Warrior	7350	9912
Poison Corrosion	150 N, 100 H	50	Butcher	8190	10920
Soul Drain	150 C, 100 H	55	Shadow Knight	46655	53320
Infernal Armor	150 N, 200 H	60	Wyvern	71060	88825

Total resources spent: 600 C + 600 N + 600 H = 1800

- For Human

Table A.9: Human unique upgrades attributes

UT	RC	RT	AFF	OCV	NCV
Improved Development Tools	100 C, 150 N	40	Every Human Structure	n/a	n/a
Fortified Plating	100 C, 150 H	40	Tower	14000	21333
Shield Bash	150 C, 100 N	45	Soldier	7233.33	9262.2
Bind Shackle	150 N, 100 H	45	Wizard	8470	10518.75
Equestrian Training	150 C, 100 H	50	Cavalier	47600	57600
Hammer of Judgment	150 N, 200 H	55	Eagle Knight	69982.5	80720

Total resources spent for every upgrade: 900 C + 850 N = 1750, Production Time Total: 275s

Total resources spent by the Orghz for one copy of every Unit, Structure and Upgrade: 11300

Total resources spent by the Human for one copy of every Unit, Structure and Upgrade (with Hydroxygen approximation): 12750

A.4 Technical Design

In this section of the Annexes chapter, there is a discussion on some technical details about decisions made for some of the elements discussed in the Game Design section, on the chapter 3 of this document.

Race Design

To provide an engaging experience to the players when playing, there are differences between the two races.

The first one is in the resource consumption between each race. The Orghz race use the 3 resources available to increase their military power. The Human race uses 2 (Crystals and Nitroxygen) for military power increase and Hydroxygen is not spent on any production or research, instead being consumed overtime. The amount of Hydroxygen spent each 6 seconds depends on the Human's population amount. The difference between the resource consumption of these 2 races adds depth to the playability of **Planetary Conquest**, as the two races have different means of spending their resources.

The total resource cost for various productions and researches for the Orghz race is more expensive than of the Human race. However the average spending cost of the resources Crystal and Nitrogen for the Orghz on this matter, is almost identical than the Human resources spent. Then the consumption of the Hydroxygen resource for the Orghz race as the game progresses, also tends to be similar to the Human Hydroxygen spent.

With this behaviour while the resources are spent differently between the two races, each of them is important to their race development.

Units

Then there are differences between the units. While both races have 4 military units with the same 4 different types available, there are different in their military attributes, resource production cost, production time and in 2 military types their attack type.

Starting with the units military power, Orghz military infantry and Flying units have slightly better combat value, than the Human units of this type. Human ranged infantry and mounted units have the same situation described than the Ogre units of the same type. Even if their Unit Combat Value is quite similar, most of their attributes are different between each other.

Another difference to add heterogeneity to the units of the two races, is the attack type of the mounted and flying units. Orghz have ranged and melee type respectively, while Humans have melee and ranged.

Also Orghz military units take one less second to be produced than Human units.

For the resource costs some Orghz units need 2 or 3 types of resource to produce an unit type, while the Human units of the same military type costs 1 or 2 types of resources.

Structures

For the differences between the races structures, there is the resource costs of each structure, different military attributes for the Tower structure and different construction time. The last one is applied for the Human race, which takes 20% less time, when it's Race Upgrade is researched.

Monsters

To add variety to the monster encampments there are 3 types of monsters that the player can interact to. Also to add more complexity each player can only choose one of 3 available options, when interacting with those monsters.

For those 3 options the resources to be spent for each of them is completely different between each type of monster. Each monster will ask for a different resource type in the Trade option, also providing to the player a different resource type between them. The resources requested between each monsters are identical in number, the same situation is applied for the resources given.

For the recruit cost, the total quantity of resources for the Orcs and Archer is identical, since they are both infantry units. To differentiate them the resource types for recruiting these monsters are different between each other, with the quantity of resource types also being different. For the Gnomish Airplanes the total quantity requested is more expensive, since they have superior military power than the former two monsters.

Discussing the final option Pillage, the 3 types of monster when killed will give to the player 3 types of different resource, referent to their monster type. The Gnomish Airplane monsters will give the double amount of resource quantity to the player comparing to the former two monster types, because they have superior military power, rewarding the player in a greater extend for defeating them.

Upgrades

While the weapon and the armor upgrades are identical in terms of military improvement between the two races, the race and unique upgrades are completely different between each other. The thinking point when designing each of them was having the Orghz with an aggressive and competitive mindset. Meaning that the Orghz unique upgrades would strength an unit, to be slightly more powerful against the Human unit of the same type. The Human upgrades have an adaptive approach for an unit type, to it's military counter type. For example the Human Cavalier upgrade increases it's movement speed. This permits this units to escape it's counter unit which is flying units, since the Cavalier cannot attack it, with more easiness.

Since the Human race is more technology advanced as the Orghz race, their upgrades take 5 seconds less to be researched.

Super Power

Each race has it's own Super Power. The effects of each other are completely different. The Orghz Super Power is the **Brute Force**. It increases the attack of every affected unit by 3 points. This super power reflects the nature of the Orghz, competitive and agressive.

The Human Super Power is **Indomitable Spirit**. It gives to each affected unit a 35 hp shield, that can only be decremented by the enemies damage. However the armor points do not enter in equation, when the shield points are decremented. The reason for it is for not having dominant strategies when choosing the units to be affected by the Super Power. Giving the case of the Human Super Power, units with the most armor points such as Cavalier would be preferred because they have more starting armor points that any other Human military unit, making the number of possible choices for the units to be affected by this Super Power be reduced.

Another factor that was taken in account when designing the Super Powers, was to have the same increased amount of military power for every type of unit regardless of their attributes, for the same purposed discussed before. This decision helped to avoid dominant strategies from happening.

Map

Starting by referring the map size,the dimensions are of 2880 per 1440 pixels containing 45 to 45 tiles. The size of the map is appropriate for the players to explore it, and to conquest territory to gather more resources or to interact with monster encampments.

The bases location are at the top and bottom of the map, for Orghz and Human respectively. Those bases are surrounded by mountains except a small part serving as an entrance, to make the player's military base difficult to attack.

Each base also starts with one resource deposit for each available resource, however the player in order to improve it's economy growth may explore other locations with resource deposits, constructing expansion bases [75]. As the article cited explains for **Expansion Bases**, there are two types of them: Natural Expansions that are generally adjacent or close to one of the starting bases; and Unnatural Expansions which are not close to the player's starting base.

In the map designed for this game, there are also these type of bases. There are 2 Natural Expansion bases close to each player's starting base symmetrical between each other, however on each side of the map each natural expansion of a player side is also close of each other. With this layout players may be likely to engage each other when trying to construct Natural Expansion Bases.

For Unnatural Expansions there is only one, which is located at the middle of the map surrounded by flora and lava, containing Ancient Resource Deposits for each resource available. This zone of the map may encourage confrontation between the players, in order for them to be able to extract the resources present in these deposits, since the quantity of resources extracted is twice as much as the normal extraction of resources, further increasing the player's economy.

There are many non-movable objects such as rocks, mountains, trees, lava or water spread across the map. Some of these are positioned closed from each other creating narrow passages, where player's units might have to travel on them. These passages are called **Choke Points**, that can force the players to avoid passing by them or to use them to their advantage [76]. A good example of the later is the entrance of the player's starting base. The player can construct at the entrance structures such as Houses or Towers occupying the entire entrance zone, forcing the opponent to either destroy those structures to be able to attack the player's base or to use Flying units to walk over them. Either way this strategy will delay the opponent's attack to the player base.

The monster encampments locations are also spread symmetrically between each other. The weaker monster encampment types Ogres and Archer are close to the player's starting base, however the last monster type Gnomish Airplane not only are situated at the horizontal edges of the map, but are also closer to the Natural Expansions of each player, which could be another factor to promote confrontation between the players.

The existence of various symmetrical points of interest across the map such as the monster encampments and resource deposits, is to give equal conditions for each player to use them, in order to grow their military power and conquering the planet.