INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




A framework for real time MMO game content generation

TIAGO JOAQUIM DIAS ALVES
Outubro de 2015

POLITÉCNICO
DO PORTO

A framework for real time massively multiplayer online game content generation

Tiago Joaquim Dias Alves

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática, Área de Especialização em
Sistemas Gráficos e Multimédia

Orientador:
Jorge Manuel Neves Coelho

Júri:
Nome do presidente do júri, Categoria, Escola
Nome do primeiro vogal, Categoria, Escola
Nome do segundo vogal, Categoria, Escola

Porto, Outubro de 2015

Acknowledgements

There were several people who helped me along this academic path so I could achieve my goals. Therefore I want to use this section to show my gratitude to the people that directly and indirectly had an important role during this dissertation.

During this process I realized that there are several challenges I had to overcome. Having an idea is not enough. I learned that I have to define achievable goals and do the necessary research as early as possible. Also having a good time management of my personal and professional life was critical in order to accomplish this before the time deadline.

The first and obvious acknowledgement goes to Jorge Coelho. He gave me a continued support over the development of this dissertation guiding me with his critical analysis and reviews. He helped me in several aspects from objectives definition, phase planning, writing and more. His participation improved the quality of this dissertation and I am grateful for it.

I want also to thank my Friends for the support and for understanding my limited availability during the last months.

Finally I want to conclude this section with an acknowledgement that is pending for years. This one is for the person that encouraged me to follow the academic path. The one that left his country in order to financially support me. The one that always told me to do what I believe to be the best for me. Thank you Dad.

Resumo

Com o aumento de plataformas móveis disponíveis no mercado e com o constante incremento na sua capacidade computacional, a possibilidade de executar aplicações e em especial jogos com elevados requisitos de desempenho aumentou consideravelmente. O mercado dos videojogos tem assim um cada vez maior número de potenciais clientes. Em especial, o mercado de jogos *massive multiplayer online* (MMO) tem-se tornado muito atractivo para as empresas de desenvolvimento de jogos. Estes jogos suportam uma elevada quantidade de jogadores em simultâneo que podem estar a executar o jogo em diferentes plataformas e distribuídos por um "mundo" de jogo extenso. Para incentivar a exploração desse "mundo", distribuem-se de forma inteligente pontos de interesse que podem ser explorados pelo jogador. Esta abordagem leva a um esforço substancial no planeamento e construção desses mundos, gastando tempo e recursos durante a fase de desenvolvimento. Isto representa um problema para as empresas de desenvolvimento de jogos, e em alguns casos, é impraticável suportar tais custos para equipas *indie*.

Nesta tese é apresentada uma abordagem para a criação de mundos para jogos MMO. Estudam-se vários jogos MMO que são casos de sucesso de modo a identificar propriedades comuns nos seus mundos. O objectivo é criar uma framework flexível capaz de gerar mundos com estruturas que respeitam conjuntos de regras definidas por *game designers*. Para que seja possível usar a abordagem aqui apresentada em várias aplicações diferentes, foram desenvolvidos dois módulos principais. O primeiro, chamado *rule-based-map-generator*, contém a lógica e operações necessárias para a criação de mundos. O segundo, chamado *blocker*, é um *wrapper* à volta do módulo *rule-based-map-generator* que gere as comunicações entre servidor e clientes.

De uma forma resumida, o objectivo geral é disponibilizar uma framework para facilitar a geração de mundos para jogos MMO, o que normalmente é um processo bastante demorado e aumenta significativamente o custo de produção, através de uma abordagem semi-automática combinando os benefícios de *procedural content generation* (PCG) com conteúdo gráfico gerado manualmente.

Abstract

The constant raise of mobile devices computational power enables the execution of complex applications and games in such devices. The raise in their popularity considerably increases the number of potential costumers for the videogame industry. In particular, the massive multiplayer online (MMO) games market is becoming more attractive. These games are known for supporting a large number of players simultaneously ideally using different devices in extensive game worlds that are the game's environment. These game worlds typically have points of interest that may be visited by the players and which are distributed in an intelligent manner in order to make the game addictive. Due to their complexity, a lot of effort is necessary to design game worlds increasing the development costs. This is a problem for game companies and is commonly impracticable for indie development teams.

In this thesis we present an approach to enable the easy creation of game worlds for MMO games. We studied several successful MMO games to identify common properties in order to create a flexible framework capable of generating worlds with structures based on a set of game design rules. In order to use this approach in different application contexts, two main modules were developed. The first one named *rule-based-map-generator* handles all the world generation logic and operations. The second module named *blocker* is a wrapper around *rule-based-map-generator* that handles network communication between the game server and its clients.

The main goal is to create an open framework to ease the generation of game worlds using predefined rules in a semi-automatic approach joining the benefits of procedural generation with manual art work creation.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

As a result of several innovations in electronic devices, interactive entertainment has almost tripled in size over the past decade [28]. Smartphones have increased mobile gaming to unprecedented values and improved broadband speeds and infrastructure have pushed online gaming forward.

A recent analysis of the global games market by segment [28] predicts that by 2015, free-to-play massive multiplayer online (MMO) games will have a share part of $8.7 billions of a total of $74.2 billions in the game industry. There are several types of MMO games and most of them are known for supporting a large number of players, simultaneously interacting with each other in extensive game worlds.

Normally, MMO games encourage players to explore the unknown in order to evolve. To be additive, the distribution of points of interest to the players around the game world must be done in an intelligent manner. For example, if it takes too long for a player to reach some point of interest, he may feel a lack of reward for the time he spent, feel bored and leave the game. The same may happen if the player finds all the needed points of interest near him without having to explore the world. Another key factor for this type of games is that the player must feel that each new place he reaches is somehow unique. If he does not, his desire for exploration will decrease.

This means that a lot of effort is necessary to design those game worlds taking a lot of time and money during the game development process. This represents a problem for game companies and in some cases it is very difficult for indie development teams to support such costs.

An alternative to create large game worlds with manual artwork is the use of procedural generation where content is produced algorithmically [1, 12, 3]. Although this process can provide infinite content, different playthrough experiences and increase the game replay value, it can also make the game predictable and illogical, degrading game experience. We believe

that by mixing content created by game designers with procedural generation techniques provided in an open framework that eases content production, it is possible to join the best of both worlds with fruitful results.

The contribution of the work described in this thesis is to provide a framework capable of generating worlds with structures that obey a previously configured set of rules. The aim of the framework is to be very flexible so it comprises several needs found in current MMO games. Besides providing a reasonable set of rules, the framework handles all the generation logic expanding the world when necessary in a seamless way. To the best of our knowledge there is no other similar framework available.

The terms world and map usually describe the same area in a MMO game. Thus, for the rest of this dissertation these two terms – world and map – will be used without distinction. The same applies to the terms rule and constraint which describe restrictions on the content generation.

The remaining of this thesis is organized as follows, in chapter 2 we introduce all the concepts relevant in the context of this thesis. Then in chapter 3 we describe our approach to generate worlds based on game design rules. In chapter 4 we present implementation details. In chapter 5 we present a case study where the framework is used to generate a world for a simple game inspired by a real example. Finally in chapter 6 we conclude this thesis and identify strengths and weaknesses of our approach. We also present possible extensions.

Chapter 2

Preliminaries

In this chapter we introduce all the concepts that are relevant in the context of this thesis. We introduce massive multiplayer online (MMO) games, study several MMO games and the role that procedural generation has in those games. We also introduce typical procedural generation techniques for content production, focus on the disadvantages of the existing approaches and then explain some technical details of our framework.

2.1 Massive multiplayer online games

A massive multiplayer online game, also known as MMO, is a video game where a large number of players interact with each other in a persistent world. In these games it is important to have the players absorbed in the world and to do so, points of interest for the player must be distributed in an intelligent manner.

Some MMO games are capable of generating their worlds. By other words, there are games that can produce world content for the players, following patterns and obeying game design rules. We chose to study several of these MMO games in order to understand common properties of world generation that was proven successful by keeping the players engaged and willing to keep exploring the game world. The goal is to produce a general purpose open framework to ease the development task.

Manually designing worlds capable of having simultaneously a large number of players can be very costly and time consuming. That is why having a way of procedurally generating world content that respects rules defined by the game designers can bring a lot of benefit for the game development teams and consequently for the game players.

2.2 Procedural content generation in gaming context

Procedural content generation, from now on abbreviated as PCG, consists in the creation of content algorithmically with limited or indirect user input. In other words, PCG refers to computer software capable of creating game content on its own, or together with the help of human players or game designers. By content we mean most of what is contained in a game: levels, maps, textures, music etc.

The generated content may coexist with authored content since the two are not mutually exclusive. This concept has been used in gaming context for a long time. For example, in the early eighties, game developers used PCG to deal with the limited storage capabilities that home computers had at that time. Also, PCG is used as a way to provide infinite new experiences to the players as it is the case with recent games like Minecraft [7].

The following are examples of PCG:

- Software tool capable of creating dungeons for adventure games
- Game engine that populates game worlds with vegetation
- Graphical design tool that allows a user to design maps for strategy games while continuously evaluating the designed maps for its gameplay properties and suggesting improvements to the maps to make them better balanced and more interesting.

2.2.1 Why use procedural content generation

There are different reasons to use PCG during game development. Removing or even reducing the need of having game designers generating game content seems an obvious reason to use PCG. The number of man-months needed to develop a successful commercial game have increased ever since computer games were invented and it is now common for a game to be developed by hundreds of people over a period of a year or more. This leads to less diversity in the games marketplace because fewer game companies and indie developers can afford to develop a game.

Reducing the game designers work using algorithms allows games to be produced faster and cheaper while supposedly preserving quality. Also intelligent design tools that make use of PCG can help small teams without the resources of large companies, and indie developers, releasing them from worrying about the game content design that can be very costly, and therefore allowing them to focus on other game development details.

Another reason for using PCG is that it might help game developers to be more creative. Algorithmic approaches for generating content can offer unexpected but valid solutions for the requirements in mind, that game designers could have never though if they were manually creating the content, bringing new ideas during the game development and therefore improving the game quality.

Finally, PCG methods can drastically reduce game packages size. In a mobile gaming context, this can be very helpful since mobile devices are limited in storage capabilities when compared to home computers. A game with a small package size can reach a wider audience and therefore be more profitable.

A deep analysis of PCG in game development can be found in [23].

2.2.2 Games that make use of procedural content generation

In the early eighties, home computers had limited capabilities resulting in a limited amount of space available for storing data. This pushed forward the use of PCG in game development because game designers were unable to statically deploy the amount of data needed for the games.

A notable example is the 1980's Rogue game [30], where the player is a hero traveling randomly generated dungeons. Rogue served as inspiration for so many games that a sub-genre called roguelike emerged [26]. Roguelike games are characterized by the exploration and discovery of randomly generated worlds.

Another classical example is Elite that overtake the storage limitations using PCG to generate 8 different galaxies with 256 planets each.

More recently, procedural generation has been brought to the spotlight by several cases of success. Here we present some games that make use of procedural generation that we found relevant for this thesis.

2.2.2.1 Minecraft

Most notable case of using PCG successfully might be Minecraft [7]. Minecraft is a open world game that has no specific goals for the player to accomplish besides exploration and construction. It is a game about breaking and placing blocks and finding landscapes.

At the beginning, the player is placed in a world with procedurally generated landscapes like the one present in figure 2.1. This initial world is built with cubes which can be destroyed and placed elsewhere.

Figure 2.1: Minecraft world example

As the player explores new directions, more wold content is generated on the fly without limitations. Theoretically Minecraft generates infinite worlds and, however there is a lot of repetition, that does not have a negative influence in this particular game [17].

2.2.2.2 Don't Starve

Another case of success is Don't Starve [14]. Don't Starve is a single-player survival game with a randomly generated open world like the one present in figure 2.2. Only the area immediately around the player is shown and more is generated as the player explores the world.

The main goal in Don't Starve is to survive. Unlike other similar games, players are not able to construct and modify the world. Don't Starve distributes resources in an intelligent manner in order to challenge players to find those resources [24].

2.2.2.3 Rust

Rust [27] is a survival multiplayer game played in an open world. In order to survive, players must gather resources from the environment, such as wood and rocks, and craft tools by mixing those resources.

Players interact with each other if they are playing in the same server. Each server has its own world that was procedurally generated (see figure 2.3) based on a seed value given when the instance of the game is started.

Figure 2.2: Don't starve world example

Rust's worlds are divided into large, geographically distinct areas: desert in the South, forest in the center and a snowy one in the North. Each area has its own animals and resources. These animals and resources disappear when consumed by a player and have a timespan to respawn. Since resources are consumed and take time to reappear, players have the need to explore the world to search for more resources.

2.2.2.4 Civilization V

Civilization V [18] is another game that uses procedural generation. Here the players are able to pick the world shape by selecting which generation algorithm to use and setting some generation variables. Every world is different, however the structure is the same every time depending on which algorithm was chosen.

In Civilization V, worlds are always structured with hexagonal grids as it can be seen in figure 2.4 where the world structure is highlighted. More about hexagonal grids in gaming context can be found in [9].

In game, the player leads a civilization searching for different achievements in the scope of research, exploration, and expansion.

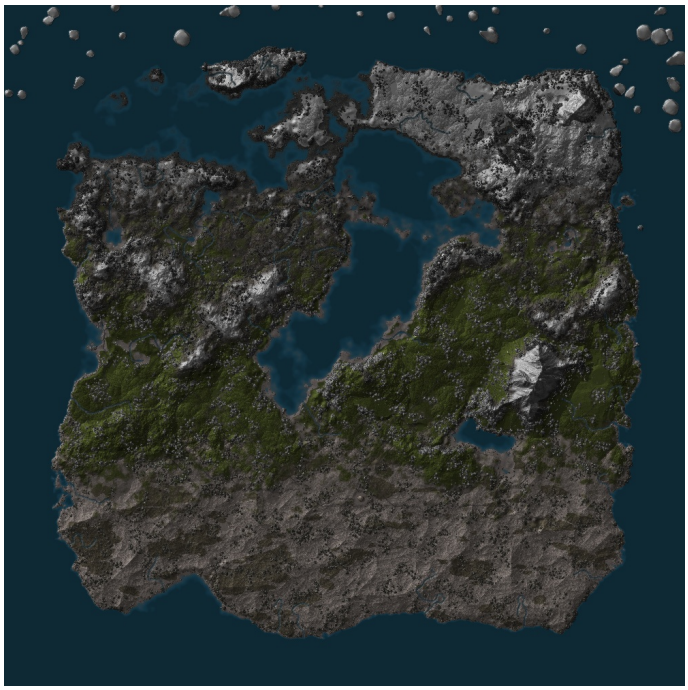

Figure 2.3: Rust world example from top prespective

Figure 2.4: Civilization world example

2.2.2.5 Pioneers

Pioneers [8] is a turn-based exploration RPG. A turn-based RPG is a type of role playing game where the players face battles that consist of turns. In these turns, the player can command their characters to perform various actions to defeat the opponents.

In Pioneers, the player leads a group of travelers in an adventure in a procedurally generated world (see figure 2.5). In this world, players search for temples and tribes solving puzzles along the way and gathering resources to survive through the year's seasons.

The player can only see the world near him. This encourages the player to explore new parts of the world.

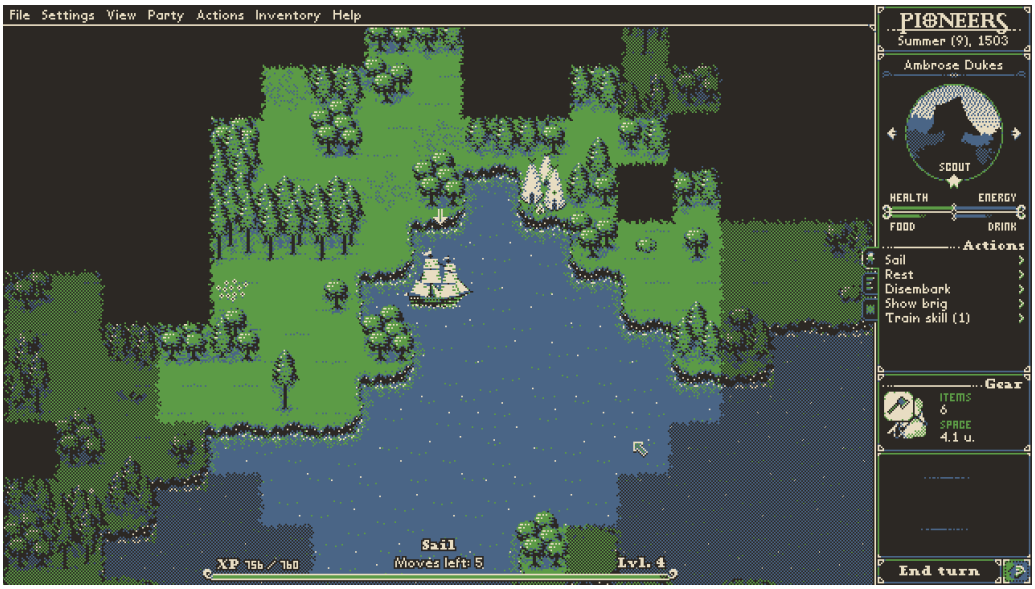

Figure 2.5: Pioneers world example

2.2.3 Common goals

Taking an overall look over these games we can identify several common goals. This kind of games try to encourage players to explore the unknown in order to get something that might be valuable. Also it is important that the valuable resources are distributed in an intelligent manner. For example, if it takes too long for a player to reach some resource, he may feel a lack of reward over the time he spent, feel bored and leave the game. In contrast if the player finds all the needed resources near him without having to explore the world, world generation is useless.

Another key factor is that the player must feel that each new place he reaches is somehow unique. If he does not, his desire of exploration will decrease.

2.3 Technologies

In this section we present the technologies used in the framework implementation.

2.3.1 JavaScript

The language used for the framework implementation was *JavaScript*. *JavaScript* is a lightweight, interpreted, object-oriented language with first-class functions [19]. Having first-class functions means that the language supports passing functions as arguments to other functions, returning functions as the value from other functions, assigning them to variables and storing them in data structures.

Although *JavaScript* popularity came from being the scripting language used in web browsers, it is also used in many non-browser environments such as *node.js* (see 2.3.2).

JavaScript is a dynamic scripting language supporting prototype based object construction, meaning that once an object is created it can be used as a prototype for creating similar objects. In *JavaScript*, objects are created programmatically by attaching methods and properties to otherwise empty objects at run time, as opposed to the syntactic class definitions common in compiled languages like *C++* and *Java*.

2.3.2 Node.js

The goal is to provide a reliable framework that will gracefully handle a massive number of network requests. Since the application will not do CPU intensive computations and most of the time will be spent waiting for I/O network operations, it was concluded that it was useful the use of a non-blocking I/O platform.

The choice relies on using *Node.js* [29] since we aim to have a data-intensive real-time framework and take advantage of *Node.js* event-driven and non-blocking I/O model (see figure 2.6). Another benefit of *Node.js* is its ability to be cross-platform meaning that the framework can be deployed and used by many different platforms, and thus, reach a larger number of developers.

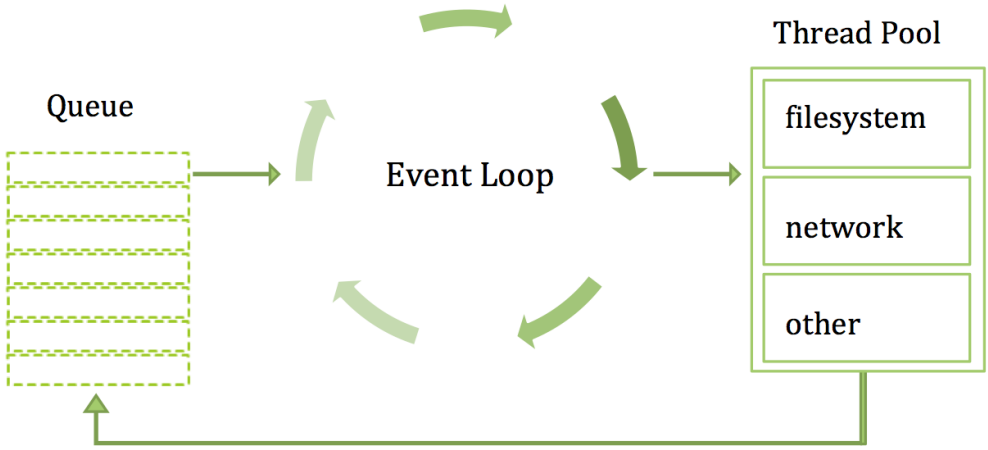

Figure 2.6: Node.js event loop simplified

2.3.3 npm

Due to the choice of *Node.js* and since it is our goal to provide a framework that can be easily used by the game development community we choose to use *npm* [21] and its registry.

npm is the package manager for *node.js* modules and its registry is a public collection of packages of open-source code. Using *npm* we made the modules available as dependencies for the game development community. This way, our project can easily be used with little effort using the *npm* client tool. For example, using *npm* client tool, developers can install each of our modules by running the following commands:

- `npm install blocker`
- `npm install rule-based-map-generator`

2.3.4 git

We provide a solution that is ready to be used for real case gaming applications and that is flexible enough to comprise a large set of different needs. The framework was planned in such a way that it should be easy to add new features. So we chose to use *Git* [10] as the revision control system for all modules and *GitHub* [11] service to host the repositories.

GitHub is a web-based *Git* repository hosting service that provides useful collaboration features such as bug tracking, feature requests, code reviews and so on. *GitHub* allows developers that are interested in the project, to add

more functionality, improve existing functionality, report issues and fix bugs if necessary, through a controlled environment using well defined processes.

2.3.5 JSON

JavaScript Object Notation (JSON), originally specified by Douglas Crockford, is an open standard format to transmit data objects consisting of attribute-value pairs, using human-readable text [16].

Despite the name referencing the *JavaScript* programming language, *JSON* is a language independent data format and is currently used by many different languages. The available data types are:

Number double-precision floating-point format in JavaScript

String double-quoted Unicode with backslash escaping

Boolean either the values true or false

Array an ordered sequence of values

Object an unordered collection of key-value pairs where the keys are Strings

null an empty value, using the word *null*

An example of a *JSON* definition can be seen in listing A.2.

Chapter 3

Proposed approach to content generation

Here we describe our framework for the open development of content in MMO games.

3.1 Analysis of Procedural Generation Techniques

In this section we focus on the topics of Procedural Generation Techniques that can be useful for the framework.

In [12] a six-layered taxonomy for procedural generation of game content is introduced: *bits*, *space*, *systems*, *scenarios*, *design* and *derived*. The authors of the survey represent this layers as a pyramid in which layers closer to the top may be built with elements from the layers at the bottom. Our area of application is in the *scenarios* layer and more precisely in the *levels* section. *Levels* consist of the playable game space and are of extreme importance in the game world design.

The games presented in the previous chapter use techniques such as pseudo-random number generators (PRNG). These generators are algorithms capable of generating sequences of numbers that although not being truly random, are still a good approximation. Because those generated sequences are based on an initial seed value, all sequences can be reproduced. The same seed will generate the same sequence which can be useful in some contexts, for example, Rust map generator uses a seed for the map creation. This allows servers to have the same map when sharing the same seed. In contrast with a truly random sequence, a pseudo-random sequence has a periodicity meaning it will eventually repeat itself. This occurs when the algorithm

uses a seed it has previously used along the sequence. In our framework we introduce pseudo-random number generators for this exact purpose.

More techniques often used are midpoint displacement algorithms [15] and Perlin Noise [22]. Both techniques are used to generate height maps in order to create realistic looking terrains and landscapes. Other common techniques are Simulation of Complex Systems techniques such as Cellular Automata [4] and Agent-based Simulation [5], Image Filtering (IF) and Spatial algorithms (SA). Further details can be found in [12] and [23].

3.2 Content generation concepts

We propose the decomposition of each world in a set of blocks that can be connected. We define these notions formally in this section.

Definition 3.2.1 (World Template)
A World Template $\mathcal{W}_T$ *is a set* $\{\mathcal{I}_d, \mathcal{N}_{sides}, \mathcal{S}_b, \mathcal{S}_c, \mathcal{C}_w\}$ *where* $\mathcal{N}_{sides} \in \{4, 6\}$ *is the number of sides of all block units in the world,* $\mathcal{S}_b$ *is the set of blocks,* $\mathcal{S}_c$ *is the set of connectors and* $\mathcal{C}_w$ *are the world based constraints.*

Definition 3.2.2 (Block)
A block $\mathcal{B}$ *is a world unit with the set of properties* $\{\mathcal{I}_d, \mathcal{L}, \mathcal{S}, \mathcal{C}_b\}$ *where* $\mathcal{I}_d$ *is a unique identifier,* $\mathcal{L}$ *is a list of classes,* $\mathcal{S}$ *is a set of pairs* (s_i, c_i) *where side* s_i *has connector* c_i *and* $\mathcal{C}_b$ *are the block based constraints.*

Definition 3.2.3 (Connector)
A connector $\mathcal{C}$ *defines the compatibility between blocks and is defined by* $\mathcal{I}_d, \mathcal{T}, \mathcal{B}_i, \mathcal{B}_c$ *where* $\mathcal{I}_d$ *is the connector unique identifier,* $\mathcal{T}$ *is its type which is one of bl (blacklist) or wl (whitelist),* $\mathcal{B}_i$ *is a set of block identifiers and* $\mathcal{B}_c$ *is a set of block classes.*

Example 3.2.1
A connector $\mathcal{C}_1 = \{c_1, wl, \{\mathcal{B}_1, \mathcal{B}_2\}, \{\}\}$ *means that the side of the connected block can only be connected with a block* $\mathcal{B}_1$ *or* $\mathcal{B}_2$*. A connector* $\mathcal{C}_2 = \{c_2, bl, \{\mathcal{B}_3\}, \{\}\}$ *means that the side of the connected block can be connected with any block except for* $\mathcal{B}_3$*.*

We also had the need to define a function for pseudo-random generation. This function enables the production of a random list of numbers based on a seed which can be replicated and is our base for the creation and reproduction of content.

Definition 3.2.4 (Random matrix generator)
Given a seed s, and a pair of Cartesian coordinates $P_{os} = \{x, y\}$, the generation of the next pseudo-random is given by, P where

$$\mathcal{P}(s, x, y) = strip_5(sin(s * x * y)) \tag{3.1}$$

where sin is the standard trigonometric function for sin calculation and $strip_5$ is a function that removes the first 5 digits from the generated number.

Intuitively, to produce a number for a world position, the generator uses the seed, the abscissa and the ordinate, multiplying these values altogether, and then applying the sine trigonometric function to the result of this multiplication. Then it throws away the first 5 digits of the decimal part of the result. Finally, the produced number is the decimal part of the result.

3.3 World instance generation

The generation of a world instance is based on an initial configuration where the programmer defines a *World template*, its *Blocks* and associated *Connectors* and a set of *Constraints*. These constraints may be of one of the following 2 categories: world based and block based.

World based constraints are the ones related with the map as a whole and defines, for example, the initial map state and how much it can expand. We propose the following world based constraints:

- Initial map size
- Horizontal and vertical boundaries
- Map center

On the other hand, as the name denotes, block based constraints are the ones related with the blocks and have effect on the selection process. For example, block based constraints may forbid a block selection for a given world position if certain conditions apply. We propose the following block based constraints:

- Blacklist connectors
- Whitelist connectors
- Maximum occupation

- Maximum occupation by percentage
- Minimum distance to other blocks

World generation stops when a given number of block units, previously defined by the game designer, is available to all the game players. When a player moves to a new area, the generation process resumes. Also the world instance is evaluated on a periodic base and blocks with an associated lifespan may expire and be unavailable after a given evaluation.

One key factor of massive worlds is how resources are distributed. For example, if gold is a highly valuable resource and it is found only in gold mines, it makes sense that gold mines are not common in the world in order to encourage exploration. To enable this type of scenarios, game designers are able to set two rules per block type; an occupation percentage and/or a maximum number of appearance in the world. Also, it is common the case where the player needs two or more different type of resources to craft some tool as it happens in games such as Rust and Don't Starve. In this case, it would make sense to avoid having these two types of resources near one another.

Our solution is to optionally set, for each block, a minimum distance to other blocks. Connectors are created and assigned to a block side describing the compatibility with other blocks. The same connector can be used by different sides and blocks. Each block must have at least one connector and can have a total of n connectors (the same as the number of sides which is 4 or 6) where each side can only have one connector.

We divide the world generation in 3 different phases. The first one runs only once at the very start of the world generation to create the initially visible part of the world. A high level description of this algorithm is presented in Algorithm 1. The second one runs multiple times to generate more world for a given world position and is triggered, for example, when a player moves to new locations. A high level description of this algorithm is presented in Algorithm 2. The third phase runs to invalidate blocks due to timeouts or other constraints. A high level description of this algorithm is presented in Algorithm 3.

Example 3.3.1 (World instance generation)
Given a World Template $\mathcal{W}_T = \{sW, 4, \mathcal{S}_b, \mathcal{S}_c\}$*, where the set of connectors are:*

- $\{c_1, wl, \{B_1\}\}$
- $\{c_2, wl, \{B_2\}\}$

Algorithm 1 Initial instance world creation

Let the PRNG sequence be $\mathcal{S}$.
Let the world template be $\mathcal{W}_T$.
Let the set of blocks be $\mathcal{B}$.
Let the set of connectors be $\mathcal{C}$.

procedure CREATE WORLD($\mathcal{S}, \mathcal{W}_T, \mathcal{B}, \mathcal{C}$)
new_world_instance ← *instantiate_world*($\mathcal{S}, \mathcal{W}_T, \mathcal{B}, \mathcal{C}$)
▷ Create an empty world instance
selected_blocks ← *select_initial_blocks*(*new_world_instance*)
▷ Select suitable blocks based on constraints
new_world_instance ← *update*(*new_world_instance*, *selected_blocks*)
▷ Update world instance with suitable blocks
return *new_world_instance*
end procedure

Algorithm 2 Generate map in location

Let the world instance be $\mathcal{W}_i$.
Let the location be $\mathcal{L}$.
The world generation for a location is given by:

procedure GENERATE MAP IN LOCATION($\mathcal{W}_i, \mathcal{L}$)
selected_blocks ← *select_blocks*($\mathcal{W}_i, \mathcal{L}$)
▷ Select suitable blocks for given location based on constraints
new_world_instance ← *update*($\mathcal{W}_i$, *selected_blocks*)
▷ Update world instance with suitable blocks
return *new_world_instance*
end procedure

Algorithm 3 Invalidate map

Let the world instance be $\mathcal{W}_i$.
The world evaluation is given by:

procedure INVALIDATE MAP($\mathcal{W}_i$)
invalid_blocks ← *get_invalid_blocks*($\mathcal{W}_i$)
▷ Evaluate map to get invalid world blocks
new_world_instance ← *remove_invalid_blocks*($\mathcal{W}_i$, *invalid_blocks*)
▷ Remove invalid blocks from the world instance
return *new_world_instance*
end procedure

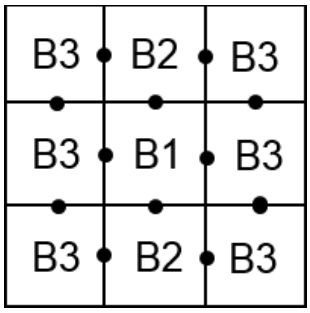

Figure 3.1: Possible result

- $\{c_3, wl, \{B_3\}\}$
- $\{c_4, wl, \{B_1, B_2, B_3\}\}$

and the set of blocks are:

- $\{B_1,\ d_1, L_1,\ \{(top, c_2),\ (right, c_3),\ (bottom, c_2),\ (left, c_3)\}\}$
- $\{B_2,\ d_2, L_2,\ \{(top, c_1),\ (right, c_3),\ (bottom, c_1),\ (left, c_3)\}\}$
- $\{B_3,\ d_3, L_3,\ \{(top, c_4),\ (right, c_4),\ (bottom, c_4),\ (left, c_4)\}\}$
- $\{B_4,\ d_4, L_4,\ \{(top, c_2),\ (right, c_3),\ (bottom, c_2),\ (left, c_3)\}\}$

one possible output of our algorithm is presented in figure 3.1.

For easy understanding the example above only defines 3 blocks and their compatibility using 4 whitelist connectors. No world or block based constraints were taken in account for the output and only the first 3x3 map units are represented. We can see that left and right sides of $\mathcal{B}_1$ are only compatible with $\mathcal{B}_3$ blocks and that top and bottom sides are only compatible with $\mathcal{B}_2$. Left and right sides of $\mathcal{B}_2$ are only compatible with $\mathcal{B}_3$ blocks and that top and bottom sides are only compatible with $\mathcal{B}_1$. Finally, all $\mathcal{B}_3$ sides have the same connector ($\mathcal{C}_4$) and so they are compatible with any of the 3 blocks, $\mathcal{B}_1$, $\mathcal{B}_2$ and $\mathcal{B}_3$.

3.4 Block selection

A fundamental part of the world generation is the block selection process. This process occurs several times during the world generation and its purpose is to choose which block best fits a given position in the world.

The block selection process makes a decision based on the block based constraints and the world state in order to select a suitable block for the world position in question. It consists in applying several rules sequentially to a set of candidates (blocks of the world instance), filtering the ones that are valid matches for the world position.

Rules are applied sequentially while there are candidates remaining. If all block candidates are excluded during the process, the world position is set as void. Thus, the world position is a completely empty space in the world if there is no suitable candidate for it.

If only one candidate remains after applying all the rules, it will be selected for that position and if there are more candidates, one will be randomly picked using a pseudo random number generator along with a seed passed in the world instance creation.

A high level description of this algorithm is presented in Algorithm 4. This algorithm assumes the existence of several functions. They are:

get_candidates - returns a list of block candidates for a given world instance.

count - returns the number of elements of a given list.

apply_rile_filter - applies a rule filter to a given list of candidates and returns a new list of candidates.

select_pseudo_random - returns a candidate from a given list using a pseudo random number generator.

3.4.1 Rules

The rules for block selection are:

Blacklist/Whitelist Filters a list of block candidates based on its connectors and the connectors of the direct neighbours for the world position in question.

Maximum occupation Filters a list of blocks by invalidating the candidates which reached the **amount** limit of occupation (defined as a block constraint) in the current map state.

Maximum occupation by percentage Filters a list of blocks by invalidating the candidates which reached the **ratio** limit of occupation (defined as a block constraint) in the current map state.

Minimum distance Filters a list of blocks based on its distance to other blocks in the current map state. It invalidates candidates that does not respect the minimum distance defined as a block constraint.

Algorithm 4 Block selection

Let the world instance be $\mathcal{W}_i$.
Let the location be $\mathcal{L}$.
Let the set of rule filters be $\mathcal{F}$.
Let the pseudo random number generator be $\mathcal{P}$.
The block selection process is given by:

procedure SELECT BLOCK($\mathcal{W}_i, \mathcal{L}, \mathcal{F}, \mathcal{P}$)
  $candidates \leftarrow get\_candidates(\mathcal{W}_i)$
  $remaining\_candidates \leftarrow count(candidates)$
  $remaining\_filters \leftarrow count(\mathcal{F})$
  $filter\_index \leftarrow 0$
  while $remaining\_candidates > 0 \wedge remaining\_filters > 0$ **do**
    $filter \leftarrow \mathcal{F}[filter\_index]$ ▷ Get rule filter
    $candidates \leftarrow apply\_rule\_filter(filter, candidates, \mathcal{W}_i, \mathcal{L})$
    $remaining\_candidates \leftarrow count(candidates)$
    $filter\_index \leftarrow filter\_index + 1$ ▷ Point to next rule filter
    $remaining\_filters \leftarrow remaining\_filters - 1$
  end while
  if $remaining\_candidates = 1$ **then**
    return $candidates[0]$ ▷ Return first and only candidate remaining
  end if
  if $remaining\_candidates > 1$ **then**
    return $select\_pseudo\_random(candidates, \mathcal{P})$
  end if
  return *null* ▷ there is no suitable candidate
end procedure

Chapter 4

Implementation

In this chapter we present implementation details of the framework.

4.1 Project overview

Our project consists of 3 different standalone modules: *rule-based-map-generator*, *blocker* and *random-matrix*. Although each module can be used as a unique dependency, *blocker* has *rule-based-map-generator* as a dependency as well as *rule-based-map-generator* has *random-matrix* as one of its dependencies. This is described by 4.1.

The idea of making *rule-based-map-generator* logic completely independent (and therefore an isolated module that can be used by other projects) of the server logic provided by *blocker* aimed to fulfill different needs described by the following case scenarios:

1. the world is generated on the client side with no need of network communications

2. the world is generated on server side but the game developer will choose how to transmit the world to the clients

3. the world is generated on server side and sent to clients through socket communication

For case 1 and 2, the game developer should use *rule-based-map-generator* module. For case 3, the game developer should use the *blocker* module.

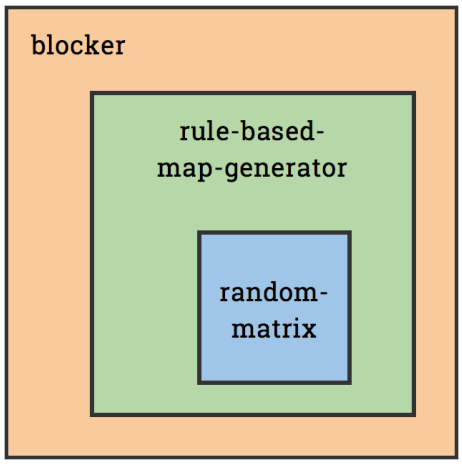

Figure 4.1: Project overview

4.2 Rule based map generator

This module holds all map generation logic and exposes all necessary methods to create world instances by means of an API described in section 4.2.6. Since there are several map structures, being square and hexagonal grids the most common ones, a strategy pattern was used for the generation process. This way each world instance holds the family of algorithms with the behavior needed for its map structure.

So far, the only strategy implemented is for square grid maps. We decided to center our efforts on the development of this strategy because it is one of the most common map structures and its map representations are easier to understand when compared to hex grid maps, and thus facilitates the explanation of the rules in the following sections.

Since the beginning a special attention was given to the project architecture related to the generation behavior in order to easily add or replace generation strategies (see section 4.2.4).

4.2.1 Constraints

Here we present and explain the constraints available for world generation. We divide the constraints in two main groups: *world based* and *block based*. Examples of their application are presented along with the description.

4.2.1.1 World based constraints

World based constraints are the ones related with the map as a whole. The available constraints are:

Initial map size This constraint is mandatory. A positive integer must be given to set the constraint. It must be defined an initial map size that represents both its initial width and height.

Horizontal and vertical boundaries This constraint is optional. To set this constraint a positive integer is necessary for each boundary: vertical and horizontal. When boundaries are defined the map can grow until it reaches them. Otherwise the map can grow with no limitations.

Map center This constraint is optional. An existing block identifier must be given to set the constraint. If this constraint is set, the map position *(0,0)* will hold the given block, otherwise any block may occupy the map center.

4.2.1.2 Block based constraints

As the name denotes, block based constraints are the ones related with the world blocks and have effect on the selection process. The available constraints are:

Blacklist/Whitelist This constraint is mandatory for each defined connector and will have effect on the block neighbours selection. A connector must have a list of block identifiers and a type, being **blacklist** and **whitelist** the available ones. If some connector's type is **whitelist**, every block side that has this connector attached will only allow as its neighbours, blocks included in the list. Otherwise if some connector's type is **blacklist**, every block side that has this connector attached, will deny blocks included in the list as its neighbours.

Example 4.2.1 *Given a world with initial size of 4 and three blocks, B1, B2 and B3, and two connectors, ALLOW_B2 and DENY_B2, where ALLOW_B2 is a whitelist connector, DENY_B2 is a blacklist connector, and both connectors have B2 block in their list of blocks. Bottom and upper sides of B1 have connector ALLOW_B2 attached. Left and right sides of B1 have connector DENY_B2 attached.*

The result present in figure 4.2 is an example of an initial map generated with these constraints. We can see that upper and bottom sides of B1 blocks

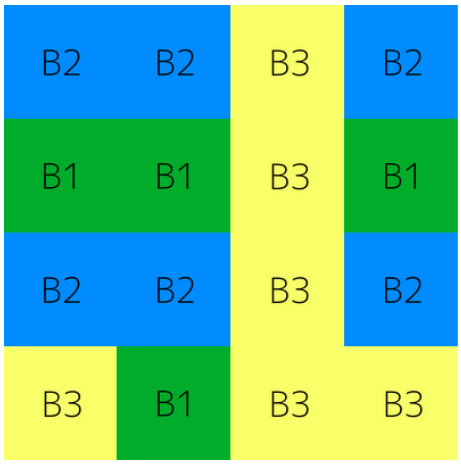

Figure 4.2: Map with blacklist and whitelist constraint

have always B2 blocks as neighbours resulting of the application of the whitelist connector constraint. On the other hand, left and right sides of B1 blocks have B1 and B3 blocks as neighbours because of the blacklist connector.

Maximum occupation This constraint is optional and a positive integer must be given to set the constraint. This constraint is set per block and will limit the number of times a given block can appear in the whole world.

Example 4.2.2 *Given a world with initial size of 4 and three blocks, B1, B2 and B3, where B1 has a maximum occupation of 5 and B2 has a maximum occupation of 2. Then, the result present in figure 4.3 is an example of an initial map generated with these constraints. We can see that there is 5 B1 blocks and 2 B2 blocks of a total of 16 blocks that compose the initial map. On the other hand there is 9 B3 blocks because no constraint was defined for it.*

Maximum occupation by percentage This constraint is optional. An integer between 1 and 99 inclusive must be given to set the constraint. This constraint is similar to the previous one but the limit represents a ratio between current block occupation and current map size.

Example 4.2.3 *Given a world with initial size of 4 and three blocks, B1, B2 and B3, where B1 has a maximum occupation percentage of 50 and B2 has a maximum occupation percentage of 30. Then, the result present in figure 4.4*

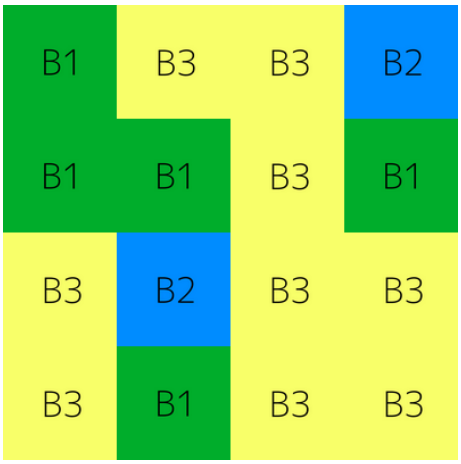

Figure 4.3: Map with maximum occupation constraint

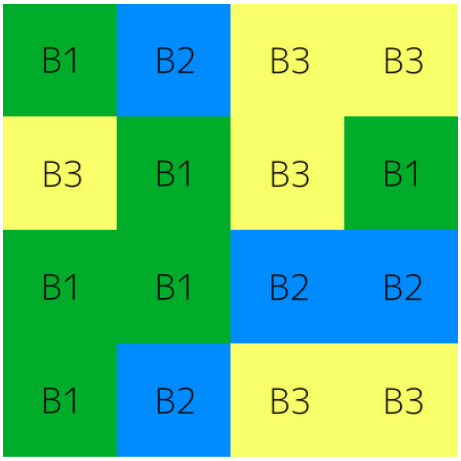

Figure 4.4: Map with maximum occupation by percentage constraint

is an example of an initial map generated with these constraints. We can see that there is 6 B1 blocks out of 16 blocks which makes an occupation percentage of 37.5% and 4 B2 blocks out of 16 blocks which makes an occupation percentage of 25%, both values below the limits defined.

Minimum distance to other blocks This constraint is optional. To properly set it, a list of key-value pairs must be given, being the key an existing block identifier and the value a positive integer that represents the minimum distance to that block.

Example 4.2.4 *Given a world with initial size of 6 and four blocks, B1, B2, B3 and B4, where B1 has a minimum distance of 3 to other B1 blocks and*

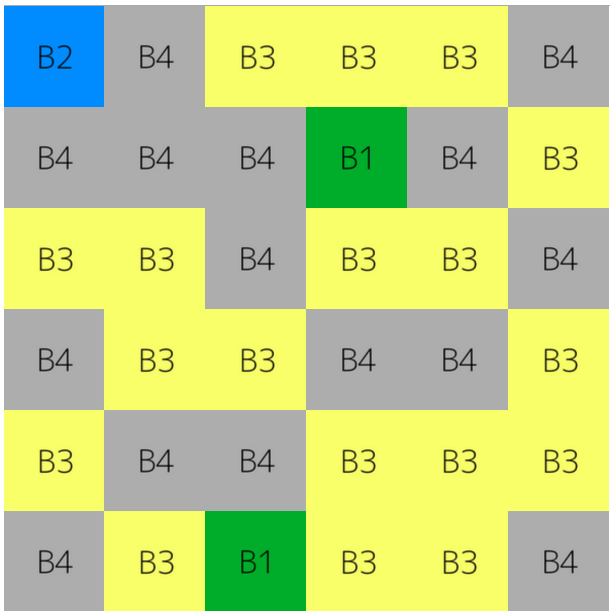

Figure 4.5: Map with minimum distances constraint

a distance of 2 to B2 blocks. Also B2 has a minimum distance of 1 to B3 blocks. Then, the result present in figure 4.5 is an example of an initial map generated with these constraints. We can see that there is two B1 blocks in the map far away of each other by 3 positions and B1 is far from B2 block by at least 2 positions while the B2 block is apart of B3 blocks by at least 1 position.

4.2.2 Block selection

The block based constraints defined in 4.2.1.2 are used along with the current map state to determine which block fits a given map position. It is called the block selection process and it consists in applying several rules sequentially to a set of candidates (blocks of the world instance), filtering the ones that are valid matches for that world position.

There are rules for all block based constraints and they are applied sequentially while there are candidates remaining. If all block candidates are excluded during the process, the world position is set as void. Thus, being an empty position due to the non existence of candidates.

If only one candidate remains after applying all the rules, it will be selected for that position and if more than one candidate are valid for the position, one will be randomly picked using a pseudo random number generator (see

4.4) along with a seed (see 4.2.5) passed in the world instance creation. See listing A.4.

Although there is a defined order in our implementation for which rules are applied, the block selection process is not dependent of the order. This means that the output by the end of the process will be the same regardless of the rules order.

4.2.3 Rules

There are several rules applied during the block selection process and all of them are stateless. By other words, there is no record of previous iterations and each iteration is handled based entirely on information that is passed to it.

All rules receive a list of block candidates and several information of the world state. Below we describe each rule in more detail.

4.2.3.1 Blacklist/Whitelist

This rule is used to filter a list of block candidates based on its connectors and the connectors of the direct neighbours for the world position being calculated. For example, if the world position in question is *(4,4)*, this rule receives the blocks that belong to the positions *(4,5)*, *(5,4)*, *(4,3)* and *(3,4)*. The rule only allows candidates that are compatible with the neighbours.

4.2.3.2 Maximum occupation

This rule filters a list of blocks by invalidating the candidates which reached the maximum **amount** of occupation (defined as a block constraint) in the current map state. It receives an object containing information about the occupation of all block types of the world instance. The rule allows blocks that don't have defined the block constraint *maximum occupation* and the ones that its current occupation don't exceed the constraint defined.

4.2.3.3 Maximum occupation by percentage

This rule filters a list of blocks by invalidating the candidates which reached the maximum **ratio** of occupation (defined as a block constraint) in the current map state. It receives an object containing information about the occupation in percentage of all block types of the world instance. The rule allows blocks that don't have defined the block constraint *maximum occupation by percentage* and the ones that its current occupation in percentage don't exceed the constraint defined.

4.2.3.4 Minimum distance

This rule is used to filter a list of blocks based on its distance to other blocks in the current map state. It invalidates candidates that do not respect the minimum distance defined as a block constraint. It receives the world position in question and a function to retrieve a part of the map. The rule allows blocks that do not violate minimum distances to other blocks around the world position in question.

4.2.4 Strategies

In order to allow the generation of different map structures such as square grid maps and hexagonal grid maps, we used a strategy pattern for the generation process. In this section we start by describing the interface that each strategy must obey so it is valid for the process generation. Then we present our strategy to produce square grid maps, problems found during its development and our approach to solve those problems.

4.2.4.1 Interface

To provide an effortless way to add different logic and behaviour for world generation we used the strategy design pattern. We defined an interface with four different functions that each strategy must respect. Those functions are described next.

- **init(worldConstraints, blocks)**
  This function is called once in the world instance lifetime. Its purpose is to initialize the world instance by passing its world constraints and blocks. It is also responsible for the first map generation.

- **getAtPosition(x, y)**
  Used to retrieve a block for a given position composed by two arguments. x is the abscissa and y the ordinate.

- **getPartialMap(minX, minY, maxX, maxY)**
  This function is used to retrieve a specific part of the map. This can trigger a map generation or not, whether the requested part of the map, was already fully generated. The portion boundaries are passed as arguments. minX is the bottom/left abscissa, minY is the bottom/left ordinate, maxX is the upper/right abscissa and maxY is the upper-/right ordinate.

- **getMap()**
  This function is used to retrieve the current map and does not trigger the generation process. It returns the map as it is in that moment. No arguments are needed.

4.2.4.2 Square grid

Square grids are the most common grids used in games, primarily because they are easy to use. To reference a certain square of the grid it is enough a pair of Cartesian coordinates (x, y) and to reference a certain edge of a square it is needed some unique identifier (ex: letter). Each square has 4 direct neighbours, one per side.

4.2.4.3 Advantages and disadvantages when using a square grid

The movement directions available for the player in a game is not directly related to the number of sides each map tile has. For example, a map composed by tiles with four sides may allow a player to move diagonally from tile to tile, besides the typical vertical and horizontal movements.

If a certain game that uses a square grid allows the player to move in diagonal directions, the player will take longer to reach the next block when moving diagonally than when moving vertically or horizontally. This happens because the square centers are not all at the same distance.

There are other grid representations that do not have this problem. For example, hexagonal grids solve this problem because the centers of all hexagons are at the same distance. However, in a hexagonal grid, the player can move directly from some hexagon center to an adjacent hexagon center using 1 of 6 possible directions while in a square grid the player can move directly to another adjacent square using 1 of 8 possible directions.

4.2.4.4 Performance concerns

Although a square grid map can be represented with only one matrix structure, using only one to manage the map expansions will cause the following problems:

Unnecessary expansion A matrix can grow in two ways. It can grow vertically by adding more rows and it can grow horizontally by adding more columns. In a 100x100 map, if we expand the map 1 unit both vertically and horizontally, we will be adding more 101x101 - 100x100 = 201 blocks, when in fact it is only necessary to expand the boundary where the player is moving to.

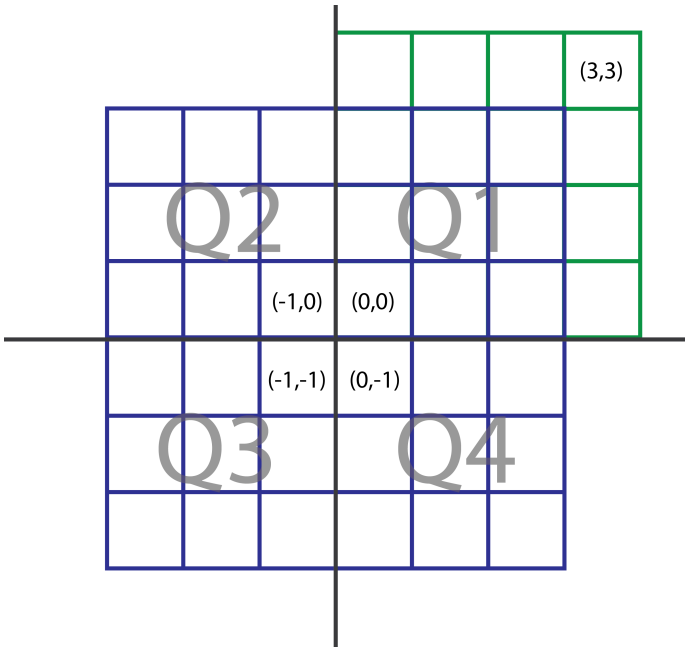

Figure 4.6: Expansion using four matrices combined

Slower operations If we want to expand a matrix to the left or downwards, besides adding a row or column we will also have to shift all matrix elements. This can be time costly making the expansion process slower.

4.2.4.5 Our approach

To avoid the issues enumerated in section 4.2.4.4 we followed an approach using 4 matrices combined.

We used one matrix for each map Quadrant. The first element of Quadrant 1 matrix is the map center while the first elements of the other matrices are its neighbours. This means that Quadrant 2 matrix is horizontally inverted, Quadrant 4 matrix is vertically inverted and Quadrant 3 is both horizontally and vertically inverted. This way only a quadrant expands at a time and always by adding more columns or rows, making the expansion operation faster. For example, in a 100x100 map where every Quadrant is 50x50, expanding one of the Quadrants by 1 unit both vertically and horizontally will add more 51x51 - 50x50 = 101 blocks, instead of 201 blocks if we used only 1 matrix.

Figure 4.6 presents an example of a map expansion using this approach, caused by a partial map request that goes from position (1, 1) to position (3, 3). The blue lines represent the initial map while the green lines represent

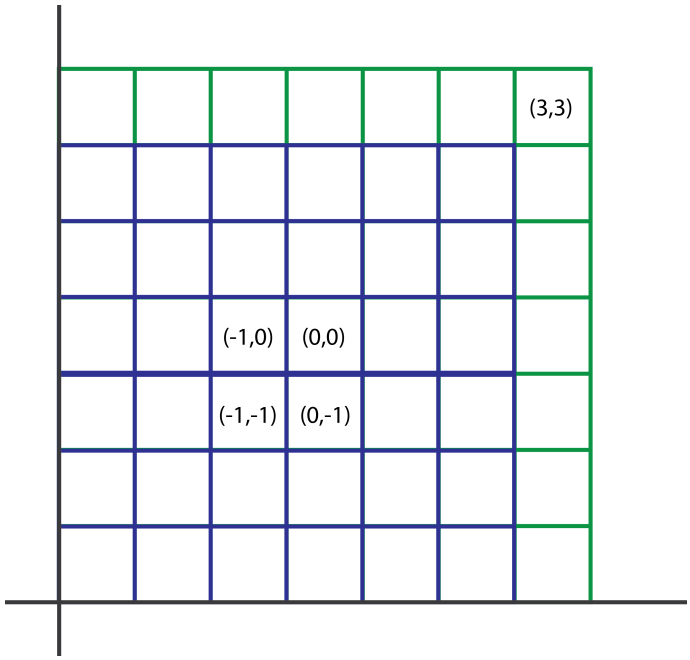

Figure 4.7: Expansion using a single matrix

new map positions generated by the expansion.

We can see that for the expansion to be successful, only the matrix relative to the first map quadrant was affected, expanding in 1 row and 1 column, and therefore increasing the map with 7 more blocks. Figure 4.7 presents the same expansion using a single matrix to manage the map expansions. We can see that expanding 1 column and 1 row would increase the map with 13 more blocks, which is $\approx 85\%$ more.

4.2.4.6 Generation process

The generation process is triggered when a part of the map is requested for the first time. The generation process is composed by 4 phases, one for each quadrant. Each phase may or may not occur depending whether or not the requested part of the map occupies the quadrant that phase is responsible for. The phases order is the following:

1. Quadrant 1 (the upper right quadrant)
2. Quadrant 2 (the upper left quadrant)
3. Quadrant 3 (the bottom left quadrant)
4. Quadrant 4 (the bottom right quadrant)

The generation process stops when all map positions from the requested part of the map went through the block selection process.

All phases go through each row and column, selecting the suitable block for each position if not already selected by a previous generation. All phases start with the rows/columns closer to the map center and consequently end with the rows/columns farther from the map center.

To better understand this process we will use an example that occupies the first and fourth quadrants.

Example 4.2.5 *For a given part of the map that goes from (2, -3) to (4, 2), a short description of each phase is the following:*

Phase 1 *generates map from all positions in its quadrant, this means from (2, 0) to (4, 2). It starts with the row (y = 0), going horizontally from left to right, selecting a block for each position from (2, 0) to (4, 0). Then it goes through the column (x = 2), going vertically and up, selecting a block for each position from (2, 1) to (2, 2). Then this sequence continues until all map positions have gone through the selection process.*

Phase 2 *does no work since the requested part of the map does not occupy its quadrant (2).*

Phase 3 *does no work since the requested part of the map does not occupy its quadrant (3).*

Phase 4 *generates map from all positions in its quadrant, this means from (2, -3) to (4, -1). It starts with the row (y = -1), going horizontally from left to right, selecting a block for each position from (2, -1) to (4, -1). Then it goes through the column (x = 2), going vertically and up, selecting a block for each position from (2, -2) to (2, -3). Then this sequence continues until all map positions have gone through the selection process.*

Figure 4.8 illustrates the generation process for this example showing the requested map part in cause and the order of each position in the block selection process. Each map position has a number that represents its order. Only phase 1 and 4 are described because there was no work in phases 2 and 3.

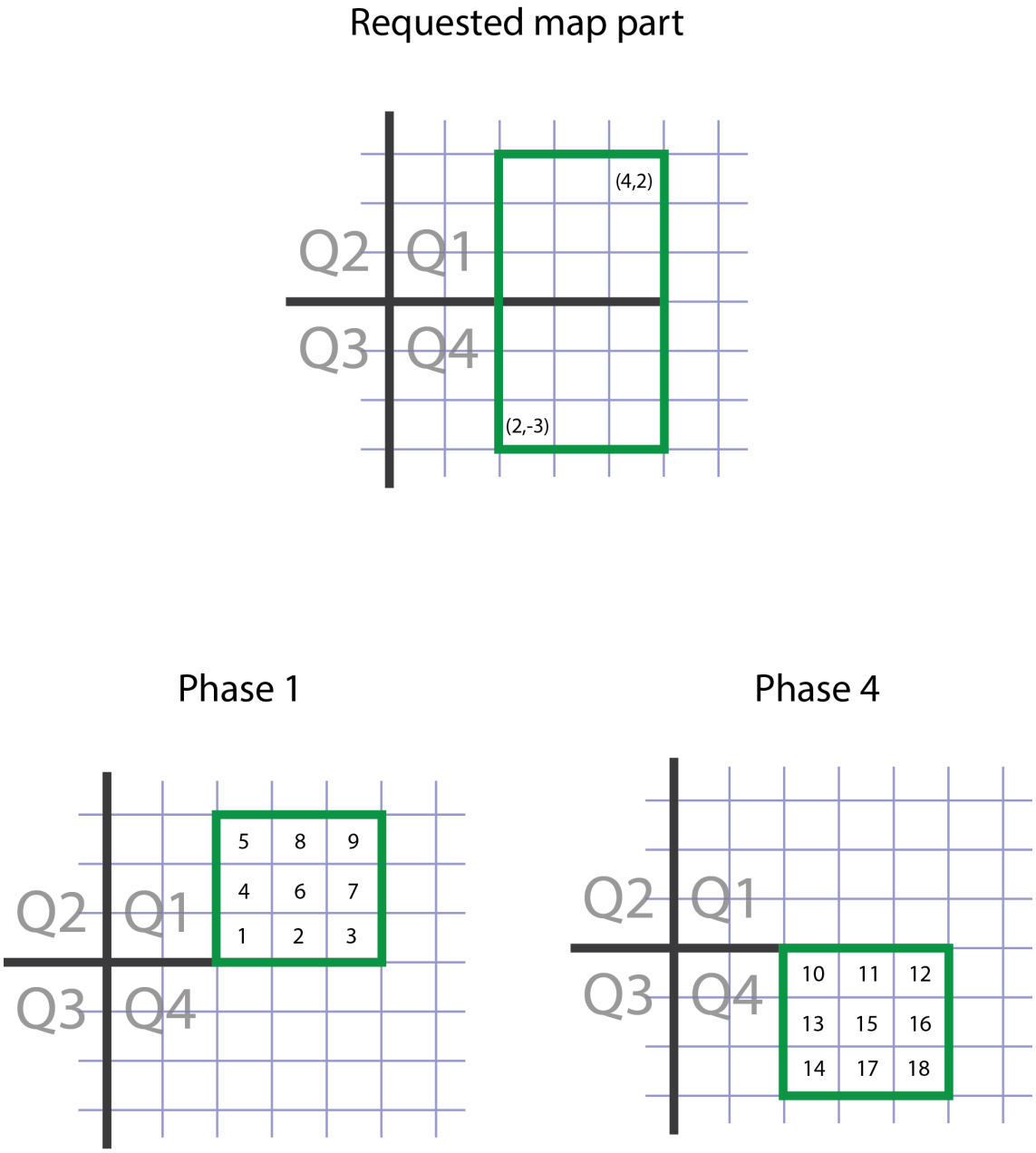

Figure 4.8: Example 4.2.5 generation illustration

4.2.5 World seed

During the block selection process for a certain world position, when all rules were applied and there is a list of 2 or more candidates to that position, the framework needs to know which candidate to use. It could pick randomly one block of the remaining list but we wanted the framework to be able to reproduce the same map every time the user wants to.

We followed a similar approach to well known games like Minecraft and Rust that generate their maps based on a seed value using pseudo random number generators. However we were not able to use a regular PRNG. A regular PRNG generates sequences of numbers based only on a seed. If we generated the map at once and respected an order it wouldn't be a problem, but that's not the case because here maps can grow to different directions depending on players's movement in the world. So we needed a PRNG that produces numbers based on a seed and a position (Cartesian coordinates). We developed a module called *random-matrix* described in section 4.4, that provides a customized PRNG used in the world generation.

4.2.6 API

rule-based-map-generator module exposes the following API to enable the creation of world instances:

- **createBlockFactory(numberOfSides)**
  This function returns a function that is used to create block instances for a given map structure. It receives one argument, numberOfSides, that will identify which strategy will be used. The returned function is:
  - **createBlockInstance (id, classes, constraints)**
    This function creates a block instance. It receives as arguments the block identifier (id), a list of classes (classes) and a set of block constraints (constraints).
- **createConnectorInstance(id, type, blockIds, blockClasses)**
  This function creates a connector instance. It receives as arguments the connector identifier (id), its type that can be blacklist or whitelist (type) and a list of classes (blockClasses).
- **createWorldInstance(numberOfSides, constraints, blocks)**
  This function creates a world instance. It receives as arguments the number of sides each block has (numberOfSides), that will determine

```
1 var worldConfiguration = require('./path/to/world.json');
2 var worldParser = require('rule-based-map-generator').
    worldParser;
3
4 var worldInstance = worldParser.parse(worldConfiguration);
```

Listing 4.1: World parser example

the world map structure, a set of world constraints and a list of block instances.

- **parser**
  Provides a world configuration parser to ease world instances creation. See 4.2.7.

4.2.7 World configuration parser

To ease the world instance creation, *rule-based-map-generator* API provides a world configuration parser. This parser has a single function called *parse* that receives one argument. This argument is a literal object with the world, blocks and connectors definitions. One can simply define the world in a JSON file, like the example in listing A.2, and use it to easily create a world instance. The listing 4.1 shows how to use the *rule-based-map-generator* API to parse a world configuration defined in a JSON file. This can be useful to game designers, that with or without a GUI tool, can create and modify this JSON files, easily defining the game world.

4.2.8 World instance

World instances expose the following methods to interact with the game world:

- **start()**
  This method starts the world instance triggering the first world generation for the initial map size defined in the world constraints.

- **getPartialMap(minX, minY, maxX, maxY)**
  This methods returns the map portion for the boundaries passed as arguments, triggering world generation if necessary.

- **getMap()**
  This methods returns the whole current map state.

- **hasStarted()**
  Returns a boolean indicating whether or not the world has been started.

4.2.9 Block instance

Block instances expose the following methods:

- **getId()**
  Returns the block identifier.
- **setSideConnector(side, connector)**
  This methods receives a block side identifier and a connector instance. It sets the given connector to the given side of the block instance.
- **getSideConnector(side)**
  Returns the connector attached to a given side passed as argument.
- **getMaxOccupation()**
  Returns the block constraint that defines its maximum occupation in the world.
- **getMaxOccupationPercentage()**
  Returns the block constraint that defines its maximum occupation ratio in the world.
- **getMinimumDistancesToOtherBlocks()**
  Returns the block constraint that defines the minimum distances to the other blocks in the world.
- **getNumberOfSides()**
  Returns the number of sides of the block instance.

4.2.10 Connector instance

Connector instances expose the following methods:

- **getId()**
  Returns the connector identifier.
- **getType()**
  Returns the connector type that can be on of the following: *blacklist* or *whitelist*.

- **getBlockIds()**
  Returns the set of block identifiers that the connector allows/denies.
- **getBlockClasses()**
  Returns the set of block classes that the connector allows/denies.

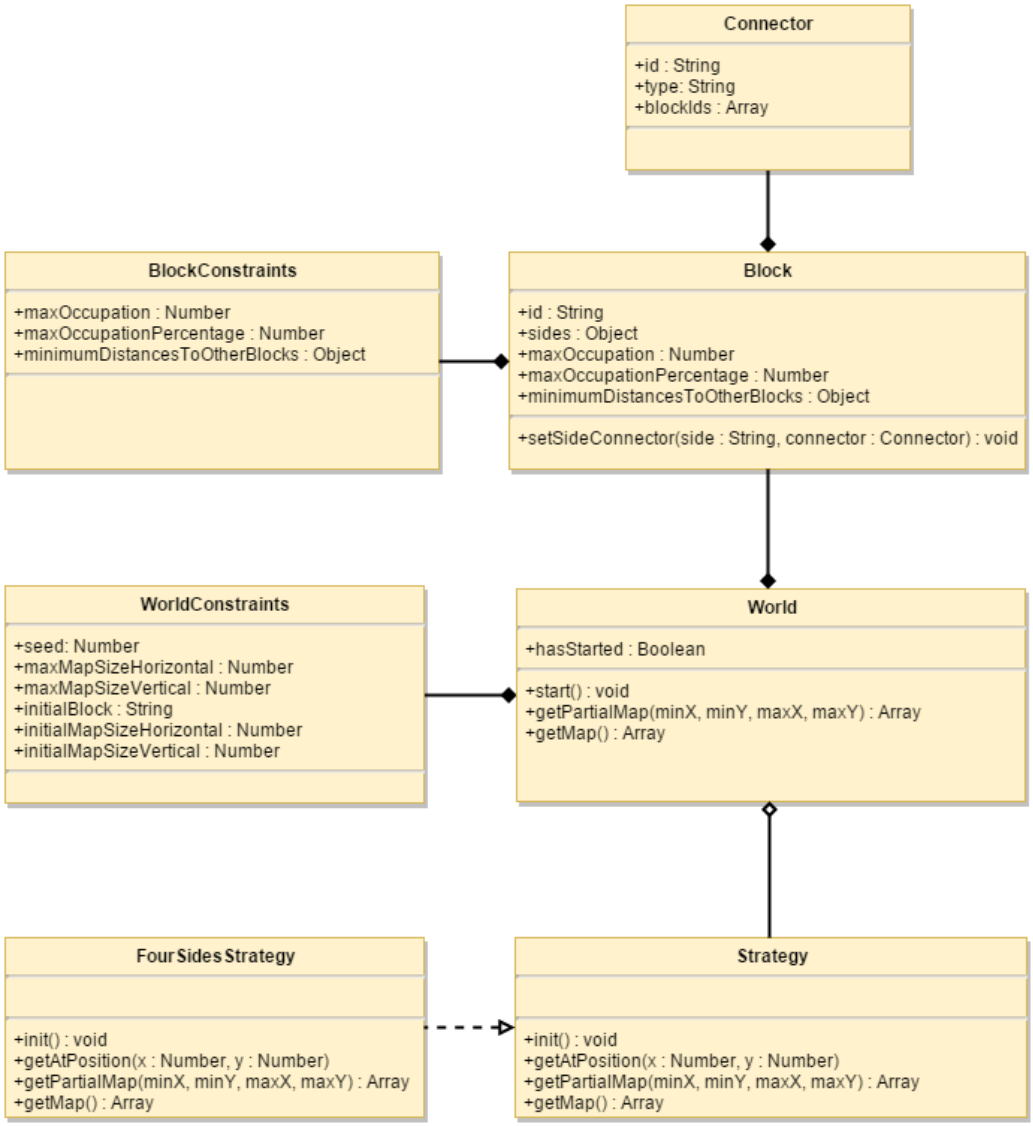

Figure 4.9: Domain Model

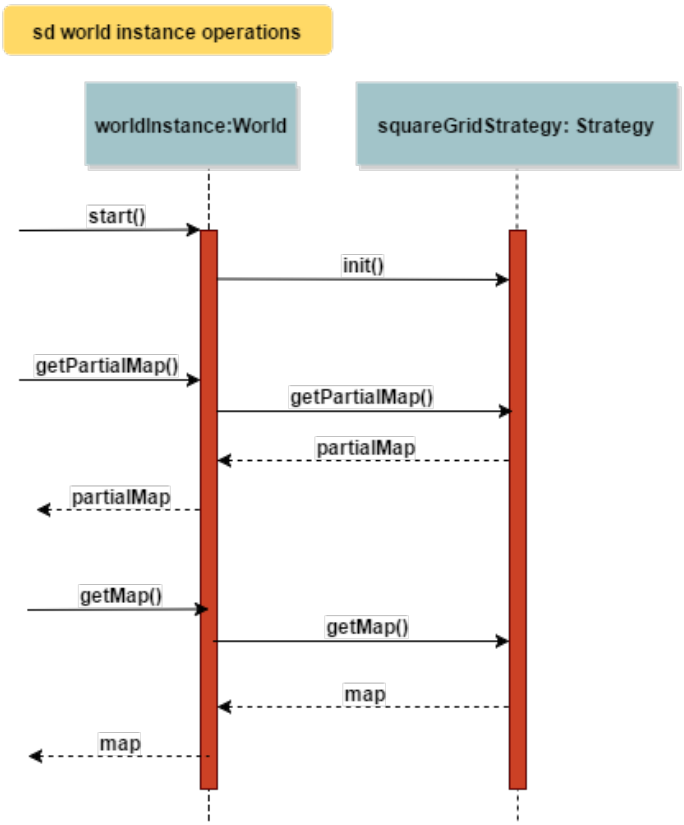

Figure 4.10: Sequence diagram: world instance operations

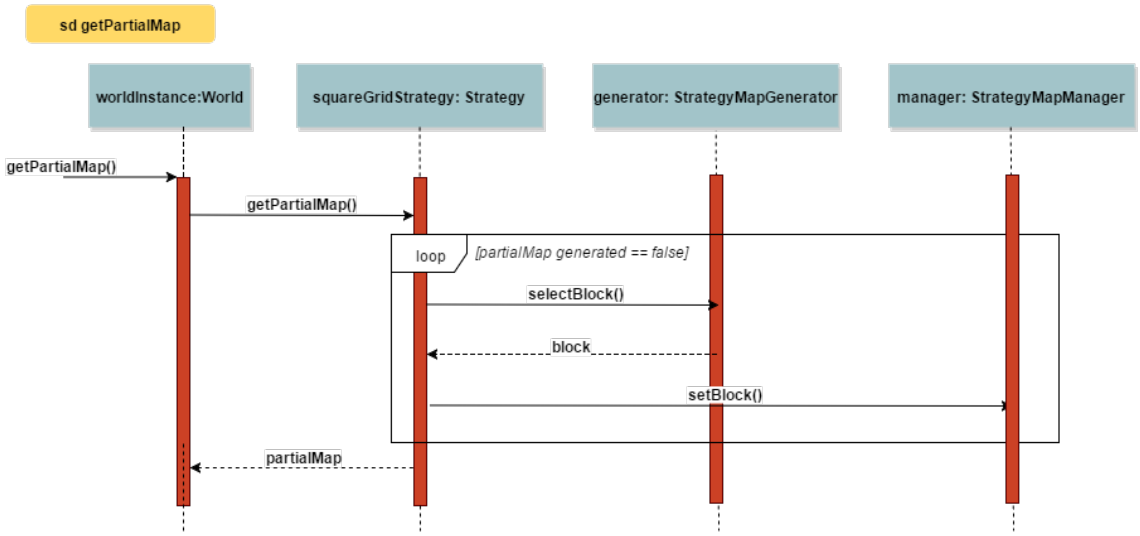

Figure 4.11: Sequence diagram: get partial map

4.3 Blocker

Blocker is a wrapper around *rule-based-map-generator* that handles real time communication between web clients and the server using *WebSocket*s [20].

We wanted our framework to be able to open communication sessions between the clients and the server using *WebSocket*s, so both can trade messages without having to poll each other, and therefore reducing the network payload and decreasing request times. We could use browser native *WebSocket*s implementations directly, but since it is a recent technology and not all web browsers support it we chose to use a cross platform library called Socket.IO [25].

Socket.IO is an event based bi-directional communication library for web applications. It falls back to other transports when the client does not support *WebSocket*s and exposes a single API that handles all logic necessary to deal with different browsers implementations. *Socket.IO* has two libraries: one for client side and the other for the server side.

4.3.1 Rooms

In a multiplayer game context, we define *room* as a virtual channel where players can join and interact with each other in the same game world. Sometimes these *rooms* are simply called *servers* by the players but in fact, a single server may provide different *rooms* where players can connect to.

Blocker allows one to have multiple rooms simultaneously. Each room is running on a different port and has its own world instance. Each player can choose then what room to connect to and share the world with other players that are in the same room.

4.3.2 Events

Once a client connection has been established, both the client and the server can initiate data transmissions using a set of events. These events allow the clients to request and receive map portions, and allows the server to broadcast map changes to the clients. Those events are described below:

PARTIAL_MAP This event is used by the client to request a part of the map. The client sends with this event the following map boundaries:

minX bottom/left abscissa

minY bottom/left ordinate

maxX upper/right abscissa

maxY upper/right ordinate

MAP This event is used by the client to request the whole map. Clients subscribe a listener to this event that will receive a data structure that represents the map.

MAP_UPDATED When the map changes, this event is broadcast to all clients in the room to notify them of those changes.

4.3.3 API

blocker module exposes the following API:

- **createServer(port, worldInstance)**
  This function creates a server instance on a given port serving the given world instance. It receives two mandatory arguments: *port* where the server will be running on, and *worldInstance* that is a world instance used by the players that will connect to that server.

- **generator**
  This is a reference to *rule-based-map-generator* API to allow the creating of world instances. It provides the functions described in 4.2.6.

4.3.4 Blocker server instance

blocker server instances expose the following methods:

- **start()**
  This method starts the server instance by creating a HTTP server listening on a given port for events defined in 4.3.2.

- **close()**
  This method closes the HTTP server making it unavailable for future client requests.

- **setPreProcessor(fn)**
  This method can be used to define a middleware function that will be executed for each received event. The given middleware function receives the event identifier and the data sent by the client as arguments. This allows the developer to add custom behaviour to his server. For example, if clients send their world position when requesting a partial map, this middleware should transcribe the world position to a map boundaries object, the data structure that *PARTIAL_MAP* event is waiting for.

- **setPostPrecessor(fn)**
  This method can be used to define a middleware function that will be executed for each event emitted to the client. The given middleware function receives the event identifier and the data that will be sent to the client. This allows the developer to add custom behaviour to his application. For example, there may be the case where the developer wants to sent also other game related information along with the world information.

4.3.5 Usage example

Here we present a simple example of how to use *blocker*. We start by showing the server side where we create a world instance and start a server on a given port. Then, we show how to communicate with the server from the client side using the *Socket.IO* library.

4.3.5.1 Server side

The listing 4.2 presents an example of how to start a server with a world instance. For a better understanding of the example we use the World parser (see 4.2.7) to create it.

```
1 var worldConfiguration = require('./path/to/world.json');
2 var blocker = require('blocker');
3
4 var port = 8000;
5 var parser = blocker.generator.parser;
6 var worldInstance = parser.parse(worldConfiguration);
7
8 var server = blocker.createServer(port, worldInstance);
```

Listing 4.2: Server side usage example of *blocker*

In line 1 and 2 we load a world configuration defined in JSON format (an example can be seen in A.2) and the *blocker* module, respectively. In lines 5 and 6 we create the world instance from the previously loaded configuration. Finally we start the server on port 8000 for the world instance we created, in line 8. From this moment on, clients can connect to the server using *WebSocket* communication.

4.3.5.2 Client side

For the client side example we use *Socket.IO* client library exposed in the global context with the variable *io*. The listing 4.3 shows how to open a connection with a server running at *localhost* on port 8000.

```
1 var path = 'http://localhost:8000';
2 var socket = io.connect(path);
```

Listing 4.3: Client side usage example of *blocker* - Open a connection

To request data from the server, whether is a part of the map or the whole map, we can emit any of the available events (see 4.3.2). The listing 4.4 shows how to request a part of the map (from line 2 to line 7) and how to request the whole map in line 10. When requesting a part of the map, its boundaries are sent to the server along with the event *PARTIAL_MAP*.

```
1  // requesting server for a specific part of the map
2  socket.emit('PARTIAL_MAP', {
3      minX: -2,
4      minY: -1,
5      maxX: 3,
6      maxY: 4
7  });
8
9  // requesting server for the all map
10 socket.emit('MAP');
```

Listing 4.4: Client side usage example of *blocker* - Emitting to server

The listing 4.5 shows how to handle server events on the client side by registering listeners functions.

From line 2 to line 4 we register a listener function for *PARTIAL_MAP* event. When server sends the *PARTIAL_MAP* event, *onPartialMapEvent* function will be executed with an argument that will contain a part of the map.

From line 7 to 8 we register a listener function for *MAP* event. When server sends the *MAP* event, *onMapEvent* function will be executed with an argument that contain the data representing the whole map.

```
1  // listening for a server event with a specific part of the
       map
2  socket.on('PARTIAL_MAP', function onPartialMapEvent (
       partialMap) {
3      // use partial map
4  });
5
6  // listening for a server server with the whole map
7  socket.on('MAP', function onMapEvent (map) {
8      // use map
9  });
10
11 // listening for a map change
12 socket.on('MAP_UPDATED', function onMapUpdated (map) {
13     // use map
14 });
```

Listing 4.5: Client side usage example of *blocker* - Listening to server events

Finally from line 12 to 14, we register a listener function for *MAP_UPDATED* event. When a players in the same map triggers some map changes, the server emits this event to the other players notifying a map change. In this example, the *onMapUpdated* will be executed with an argument that contains the data representing the whole map after the changes.

The map data sent by the server respects the same format for all events whether its a part of the map or the whole map, and a brief example can be seen in listing A.3.

4.4 random-matrix

Here we present and describe the module *random-matrix*. We start by describing its implementation, show how to use the module and present testing results to prove its efficiency.

4.4.1 Implementation

The implementation of the *random-matrix* relies on a simple approach (see A.1) that does not have impact on the project performance and still transmits a sense of randomness. Each number is generated based on the seed, the position abscissa and the position ordinate. We multiply abscissa, ordinate

and seed values altogether. Then we apply sine trigonometric function and throw away the first 5 digits of the decimal part. The final result is the decimal part of the resulting number. To avoid repetitions in near position like (1, 2), (2, 1), (-2, -1) and (-1, -2) we start by adding a fixed amount to abscissa value and another fixed but different amount to ordinate value.

4.4.2 Usage

This module works as a function that accepts only one argument being the *seed*. It returns a number generator object that contains a single function to use called *rand*. This *rand* function accepts two arguments that represent the map position, x and y, and returns a pseudo random number between 0 and 1.

The listing 4.6 shows how to use *random-matrix*. It creates three pseudo random number generators. The first and second PRNG have different seeds while the third one has the same seed as the first one. The example presents comparison instructions that return `true` (from line 10 to 17) to demonstrate the module behavior.

Line 10 shows that the same generator returns the same output for a given position in every call. However, the same is not true for different positions as it is shown in line 11.

Line 14 shows that two generators using different seeds outputs different results for the same position. Although that was the case with the seeds used for the example (123 and 456), it is not guaranteed because there is an extremely small chance of two different seeds producing the same value for the same position due to the behaviour of random generation.

Finally line 17 shows that using the same seed, despite using two different generators, returns the same output for the same position.

4.4.3 Randomness

Since we have our own PRNG implementation, it is essential that our solution is capable of generating numbers with a very good approximation of random numbers without having impact on performance.

We concluded that our implementation satisfied those requirements by doing performance tests where we analyzed the execution time and the number of occurrences when generating a big amount of random numbers. For example, generating random numbers between (-50, -50) and (50, 50) which amounts to 10000 numbers, in a *Intel(R) Core(TM) i7 Q720 @ 1.60 GHz CPU*, had an execution time of $\approx$ 40ms. The results are presented in table 4.1. The column *Interval* presents 10 possible intervals for the output that

```
1  var randomMatrix = require('random-matrix');
2  var seed1 = 123;
3  var seed2 = 456;
4  var seed3 = seed1;
5
6  var generator1 = randomMatrix(seed1);
7  var generator2 = randomMatrix(seed2);
8  var generator3 = randomMatrix(seed3);
9
10 generator1.rand(0, 0) === generator1.rand(0, 0);
11 generator1.rand(1, 2) !== generator1.rand(2, 1);
12
13 // different generators with different seeds, produce
      different values
14 generator1.rand(0, 0) !== generator2.rand(0, 0);
15
16 // different generators with same seed, produce same values
17 generator1.rand(1, 1) === generator3.rand(1, 1);
```

Listing 4.6: Usage example of *random-matrix*

can go between *0* and *1*. Each interval has a range of *0.1*. For example, a given generated number *0.24567* goes into interval *between 0.2 and 0.3* while *0.5* goes into interval *between 0.4 and 0.5*. The column *Occurrences* shows how many generated numbers the respective interval had. The column *Percentage* presents the ratio between the number of occurrences for the respective interval and the total of generated numbers (10000).

We concluded that these values satisfy our requirements.

Interval	Occurrences	Percentage
between 0 and 0.1	991	9.91%
between 0.1 and 0.2	972	9.72%
between 0.2 and 0.3	977	9.77%
between 0.3 and 0.4	985	9.85%
between 0.4 and 0.5	1019	10.19%
between 0.5 and 0.6	996	9.96%
between 0.6 and 0.7	996	9.96%
between 0.7 and 0.8	1032	10.32%
between 0.8 and 0.9	1021	10.21%
between 0.9 and 1	1011	10.11%

Table 4.1: Random numbers generation results

4.5 Code quality tool

Dynamic and loosely-typed languages like JavaScript are especially prone to developer errors. A way to be warned about potential problems in the code is to use a code linting tool.

Code linting is a type of static analysis frequently used to detect problematic patterns and code that does not respect style guidelines.

We used ESLint [31] as our code linting tool with our custom set of linting rules. Besides being warned about potential errors, we think that it is very important to have a set of strict guidelines that must be respected by who contributes to the project code base. Imposing a code style helps the developers reading others code, which is a must in an open source project.

4.6 Unit tests

One of our main goals is to provide a reliable framework to the game development community where everyone is welcome to contribute and take this project farther and in other directions. Said that, it is crucial to have a good suite of unit tests to give confidence to whoever uses the project. Not only to whom uses the framework but also to whom develops it in order to be notified when some new development accidentally breaks already implemented behaviour.

All modules use *Mocha* [13] as the test runner and *Chai* [2] as the assertion library to create a good test development environment. Unit tests in *Mocha* run serially allowing accurate reports and *Chai* has several interfaces that improves the developer flexibility when testing functionality behaviors. An example of unit tests for a module of this project can be seen in listing A.5.

All modules have a suite of unit tests with a coverage $\geq 90\%$, which brings a good level of confidence during next developments.

4.7 Project repositories

All implementation can be found on *GitHub* service (see 2.3.4). Each module has its own *Git* repository and can be found at:

rule-based-map-generator
`https://github.com/letiagoalves/rule-based-map-generator`

random-matrix
`https://github.com/letiagoalves/random-matrix`

blocker

`https://github.com/letiagoalves/blocker`

Chapter 5

Case study: A simple game

As a proof of concept we will use our framework to generate a desirable world in real time for an open-world survival MMO game called H1Z1 [6]. In H1Z1 players are dropped into the world (see figure 5.1) with nothing but the clothes they wear and a flashlight, and must explore, search and collect food, water and weapons to protect themselves. Having weapons and ammunition is crucial to survive from other players attacks and also from zombies that wander in the world.

The world we are going to create is mainly made of forests and has several points of interest spread over where players can find the means needed to survive. These points of interest include for example, water wells where players can gather water, police stations where weapons and ammunition can be found and buildings such as motels, houses and churches that may or may not have resources inside. There are also spawning locations for vehicles where players can pick them up in order to travel faster and gas stations, to refill the vehicles.

The way these points of interest are spread over the world is what makes the gameplay enjoyable and the players eager to explore new areas, immersing in the game experience. For example, it is important for the game experience that food resources are distant from police stations, where ammunition can be found. The same goes for water wells so the players don't find a part of the world where they can collect both water and food, and consequently, turn the level easier to survive. Also, it is important that gas stations are very limited and distributed in a balanced way, since players will need to refuel a vehicle very occasionally and we want them to go out of fuel, to bring hardness to the gameplay.

Since this is a survival game, these resource locations should be in short number so they are hard to find, being most of the world, areas without any items (such as forests).

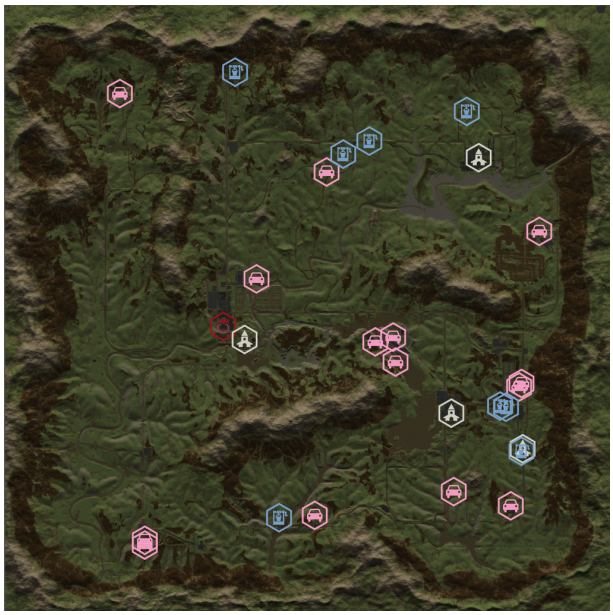

Figure 5.1: H1Z1 map from top prespective

5.1 World definition

Considering the world we want to create, we believe that the following characteristics can produce an interesting map for this case study:

- Initial map size of 10x10 units that correspond to a grid of 100 blocks
- Maximum map size of 40x40 units that correspond to a grid of 1600 blocks
- Players are spawned in the center of the world
- Vehicles are spawned in the center of the world (so a new player can pick up a vehicle in the beginning of the game)
- Car spawning locations are apart from other car spawn locations by at least 10 map units and apart from gas stations by at least 7 map units
- Gas station are apart from other gas station by at least 5 units
- Water wells are apart from other water wells by at least 4 map units and apart from police stations by at least 4 map units
- Police stations are apart from other police stations by at least 14 map units

- Buildings should be apart from other buildings by at least 5 map units
- Water wells should be in short number so they should be no more than 3% of the whole map
- Gas stations should be in short number so they should be no more than 1% of the whole map

5.1.1 Blocks and connectors

For this world the following set of blocks and connectors were defined:

5.1.1.1 Blocks

FOREST Represented as a green block and its sides have the *ALLOW_ALL* connector

VEHICLES_SPAWN Represented as a yellow block and its sides have the *ALLOW_FOREST* connector

GAS_STATION Represented as an orange block and its sides have the *ALLOW_FOREST* connector

WATER_WELL Represented as a blue block and its sides have the *ALLOW_FOREST* connector

POLICE_STATION Represented as a gray block and its sides have the *ALLOW_FOREST* connector

BUILDING Represented as a red block and its sides have the *ALLOW_FOREST* connector. This could be split in several blocks such as house, motel and church, but since all of those are in fact buildings where the player can hide and search for resources, we can use only one block to simplify this proof of concept.

5.1.1.2 Connectors

We defined two whitelist connectors (table 5.1). The first one identified as *ALLOW_ALL* connects with all blocks defined previously. The second one identified as *ALLOW_FOREST* connects only to *FOREST* blocks.

ID	Type	Blocks
ALLOW_ALL	whitelist	All
ALLOW_FOREST	whitelist	FOREST

Table 5.1: Case study connectors

5.1.1.3 World constraints

Here we present the world based constraints we defined:

Initial size 10 units

Horizontal and vertical boundaries 40 units

Map center VEHICLES_SPAWN block

5.1.1.4 Block constraints

Here we present the constraints applied to each block.
For VEHICLES_SPAWN block we defined the following rules:

Minimum distance

- 10 units from other VEHICLES_SPAWN blocks
- 7 units from GAS_STATION blocks

For GAS_STATION block we defined the following rules:

Maximum occupation percentage 1

Minimum distance

- 5 units from GAS_STATION blocks

For WATER_WELL block we defined the following rules:

Maximum occupation percentage 3

Minimum distance

- 4 units from other WATER_WELL blocks
- 6 units from BUILDING blocks

For POLICE_STATION block we defined the following rules:

Minimum distance

- 14 units from other POLICE_STATION blocks

For BUILDING block we defined the following rules:

Minimum distance

- 5 units away from other BUILDING blocks

5.2 Step by step world generation

For this proof of concept, two different players moving in different directions are used to trigger world generation. The world is generated around each player by 1 unit.

The two players will meet in the end. Both players will start from position *(0, 0)*. The first player will move horizontally to position *(-10, 0)* while the second player will move also horizontally but to position *(10, 0)* instead. Then the first player will move vertically to position *(-10, 10)* while the second player will move to position *(10, 10)*. Then both players will move horizontally 10 units until they meet at position *(0, 10)*.

5.2.1 Initial world generation

The framework generated the first portion of the world with an initial size of 10x10 units. As we can see in figure 5.2, there is a vehicle spawning zone in the world center, represented in yellow. Near this block there is a building and a water well that are 6 units apart from each other obeying to the constraint that was defined. There is also a police station to the right of the world center.

5.2.2 World growth

The first world expansion occurred when the first player reached position *(-5, 0)* as we can see in figure 5.3. The same happened when the second player reached position *(5, 0)* as figure 5.4 shows.

While the players were moving the world grew. We can see in figure 5.5 that the first player ran into two water wells, and a gas station before reaching position *(-10, 10)*. By the time the players met in position *(0, 10)*,

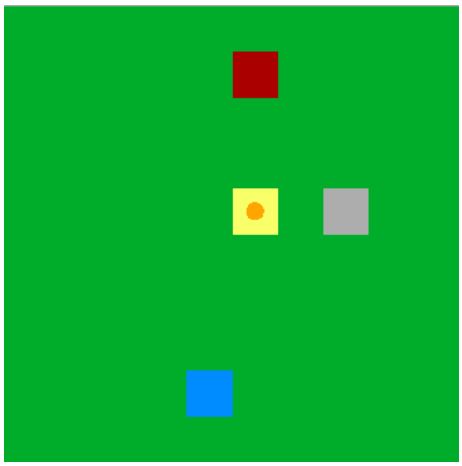

Figure 5.2: First step of world generation

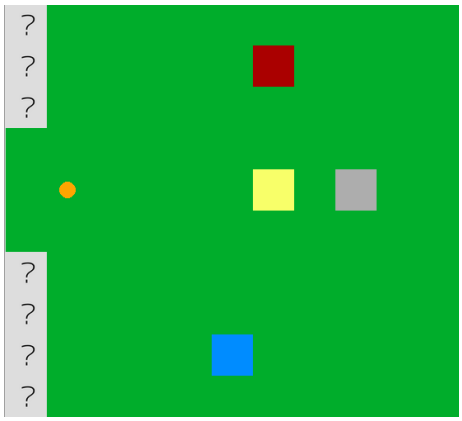

Figure 5.3: First expansion caused by first player's movement

the resulting map was what we can see in figure 5.6. The players ran into more water wells, buildings and vehicle spawning locations along the way, and all of these points of interest locations obey to the previously defined constraints.

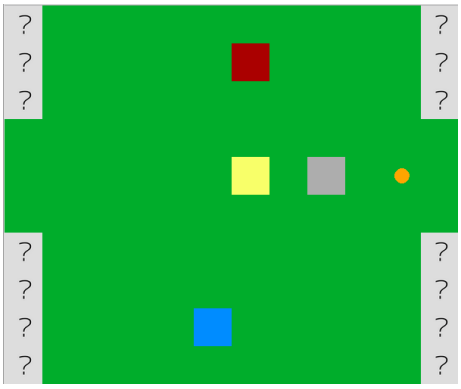

Figure 5.4: Second expansion caused by second player's movement

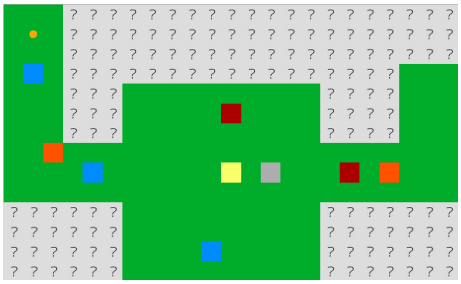

Figure 5.5: Expansion (1)

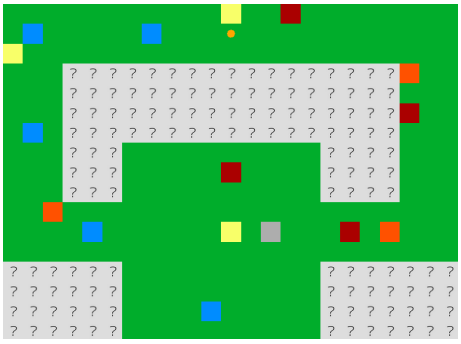

Figure 5.6: Expansion (2)

Chapter 6

Conclusions

We started by studying several popular massive multiplayer online games that make use of procedural generation to generate worlds. Our approach assembles several characteristics found on those games in an open framework that enables easy development. Firstly our framework is able to expand worlds on the fly as players move in it in the same way we can see in Minecraft and Don't Starve. It can also generate blocks in the middle of nowhere which is useful for games that spawn players through the whole world. Secondly the same way Don't Starve and Rust distributes game resources and points of interest in an engaging way for the players, we allow the game designers to set several rules that are enforced by the generation process, in order to keep the generated worlds engaging without having to manually create them. Thirdly we noticed that there are different map structures adopted, being the most used square grids and hexagonal grids. For example, while Minecraft worlds are made of a bunch of 3D Blocks, Civilization V uses hexagonal grids to represent maps. We designed our framework in a way that is easy for the developers to add or replace world generation strategies, so it can support different map structures.

After identifying the common properties between those games, we started to think about how we wanted our framework to be used by the game developers and what would be the main goals we would try to accomplish. Because we want to reduce development effort when developing massive multiplayer online game worlds, we decided that our solution should be a standalone module that can easily be used by any developer. One main feature of the framework should be the possibility of application to different contexts, such as, a game server, where players open connections with the framework to request portions of the map or even the whole map. Another main feature is that it could be a part of the game package and used by the client device without the need of network communications for world generation. We also

wanted that it could be used as a tool for game designers so they could generate maps and save the final result (map structure) to be used latter as static maps. Other main goal was to have a flexible framework so it could be used in very different gaming contexts. By other words, it should allow game designers to have the power to decide how the world should be structured so it is engaging for the players, without having to manually create it.

After all the analysis, we studied which technologies would better fit our purpose. MMO game servers need to gracefully handle a massive number of network requests. We aim to have a data-intensive real-time framework, and since it would not do CPU intensive computations and most of the time will be spent waiting for I/O network operations, we realized that we could benefit of using a non-blocking I/O platform and chose Node.js.

This platform has an event driven and non-blocking I/O model that brings a lot of benefits for projects like these. Instead of executing operations in parallel and doing thread and shared state management, Node.Js has an asynchronous philosophy centered in its event loop. An event driven architecture like Node.Js can lead to more scalable applications.

6.1 Our approach

In order to have a framework that could be used in different contexts (described above) we realized that we should create two modules. One that we named *rule-based-map-generator* that holds all world generation logic and another that we named *blocker*, which has the first module as a dependency and handles the real time network communication between the clients and the server.

rule-based-map-generator exposes an API that enables the creation of world instances. World instances hold the current map state, the set of constraints provided by the game designer, the generation strategy used for the map structure and the set of map blocks. A block represents a potential piece of the map that is a candidate to be placed in each map position during generation process. A world instance also exposes public methods to retrieve map portions, the block positioned in a given position or the whole map state.

When a world instance is created, the generation process starts generating a map with the initial size passed in the world constraints. If the world constraints also define an initial block, that initial block will be allocated in the map center, otherwise any block could be placed there with equal probabilities. Then, the generation process is resumed every time a portion of the map is requested for the first time. During the generation process,

each position of the requested map portion that is empty passes through a block selection process. The selection process consists in filtering which of the block candidates satisfy the defined rules. Each rule has the power to discard the candidates that do not satisfy the block based constraints defined by the game designer. For example, if the game designer defined that a given block has a maximum occupation in the world of 5 units and that limit was already reached, the rule for maximum occupation will discard that candidate during the generation process. After all candidates passed through all rules it can happen one of three scenarios: if no candidate passed no block will be placed in that position, if only one candidate remains it will be selected for that position and if more than one candidate are valid for the position, one will be randomly picked using a pseudo random number generator along with a seed passed in the world instance creation. The seed is used in order to be able to replicate maps. Unless the game designer defines world limits using the world constraints, the generation process can occur everywhere in the world with no particular order.

Blocker provides an easy way to create servers responsible to transmit map data to clients in real time. It uses *rule-based-map-generator* module as a dependency and handles network communications between the servers and the clients. Although at the moment it only supports *WebSocket* communication using *SocketIO* libraries, it should be useful to extend it with other protocols. Each server has one world instance and runs on a given TCP port. The user starts a server with a call to *blocker* API where it passes the world instance, created using *rule-based-map-generator* API, and the port it should use. This way the game developer can then create several *rooms* (see 4.3.1), each one with its own world, where players can interact. Players, when connected to a room, can request portions of the map and the whole map. When map generation is triggered by a player, an event is broadcast to notify the other players in the same map that the map has changed.

6.2 Strengths

We developed a framework that can be easily used by game developers and game designers to generate worlds based on previously defined rules. Because we used the *Node.Js* platform, its package manager and *npm* registry, our modules can be quickly used by installing them as dependencies using *npm* tool (see section 2.3.3). If the developer just wants to generate the map and handle the result, he can use *rule-based-map-generator* module to create world instances programmatically. On the other hand if the developer wants a server serving a given world instance to a group of players he can use *blocker*

module.

From the beginning, our focus was to provide a framework to help the development of massive multiplayer online game worlds, but the framework is not confined to that context at all. The framework is simply an approach to connect blocks with certain characteristics following rules previously defined, and so it can be used in other contexts. For example, we can add a new strategy to plan a restaurant hall arrangement where tables and self serving zones must be apart by some distances and have maximum occupations.

6.3 Weaknesses and possible extensions

Although we believe that the rules we created from the worlds properties we identified in several MMO games makes our approach flexible and ready to be used in real case applications, we identified some limitations.

Firstly our approach assumes that all blocks have the same size and each position represents an entire block. This can be a problem in some cases where blocks have different sizes.

Secondly *blocker* only allows *WebSocket* communication between the server and the clients. Although other protocols can be implemented, the current version of *blocker* can be used only for web based games. Anyway, game developers can use *rule-based-map-generator* module directly and choose how to communicate with the game clients.

Lastly using the same seed does not ensure that the worlds generated in runtime will always have the same structure. Although the initial world will be the same, the rest of the world may not be. This will always depend on the order that world parts are requested, and consequently, where generation occurs in the world. However it is possible to replicate worlds using the seed if the world is generated as a whole in a single step.

Bibliography

[1] D.M.D. Carli, F. Bevilacqua, C.T. Pozzer, and M.C. d'Ornellas. A survey of procedural content generation techniques suitable to game development. In *Games and Digital Entertainment, Brazilian Symposium on*, pages 26–35, 2011. 1

[2] ChaiJS. Chai assertion library. World Wide Web, 2015. `http://chaijs.com/`. 47

[3] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. Interactive procedural street modeling. *ACM Trans. Graph.*, 27(3):103:1–103:10, August 2008. 1

[4] Bastien Chopard and Michel Droz. *Cellular automata modeling of physical systems*. Cambridge University Press, Cambridge, England, New York, 1998. 14

[5] Paul Davidsson. Multi agent based simulation: Beyond social simulation. In *Proceedings of the Second International Workshop on Multi-Agent-Based Simulation-Revised and Additional Papers*, MABS '00, pages 97–107, London, UK, UK, 2001. Springer-Verlag. 14

[6] Daybreak. What is h1z1. World Wide Web, 2015. `https://www.h1z1.com/what-is-h1z1`. 49

[7] Sean C. Duncan. Minecraft, beyond construction and survival. *Well Played*, 1(1):1–22, January 2011. 4, 5

[8] EIGENLENK. Pioneers. World Wide Web, 2013. `http://www.pioneersgame.com/`. 9

[9] Red Blob Games. Hexagonal grids. World Wide Web, 2013. `http://www.redblobgames.com/grids/hexagons/`. 7

[10] git. git. World Wide Web, 2015. `https://git-scm.com/`. 11

[11] Inc GitHub. Where software is built. World Wide Web, 2015. https://github.com/. 11

[12] Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, February 2013. 1, 13, 14

[13] TJ Holowaychuk. mocha. World Wide Web, 2015. http://mochajs.org/. 47

[14] Klei Entertainment Inc. Don't starve. World Wide Web, 2013. http://www.dontstarvegame.com/. 6

[15] Jonathan Jilesen, Jim Kuo, and Fue-Sang Lien. Three-dimensional midpoint displacement algorithm for the generation of fractal porous media. *Comput. Geosci.*, 46:164–173, September 2012. 14

[16] JSON.org. Introducing json. World Wide Web, 2015. http://www.json.org/. 12

[17] Jaz McDougall. Minecraft. PC Gamer, 12 2011. http://www.pcgamer.com/minecraft-review/. 6

[18] Sid Meier. Civilization v. World Wide Web, 2010. http://playrust.com/. 7

[19] Mozilla. About javascript. World Wide Web, 2015. https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript. 10

[20] Mozilla. Websockets. World Wide Web, 2015. https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API/. 40

[21] Inc. npm. npm is the package manager for javascript. World Wide Web, 2015. https://www.npmjs.com/. 11

[22] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985. 14

[23] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2015. 5, 14

[24] Marty Sliva. Don't starve review. IGN Entertainment, Inc., 5 2013. IGN Entertainment, Inc. 6

[25] Socket.IO. Socket.io. World Wide Web, 2015. http://socket.io/. 40

[26] Ks Sponsors. A brief history of the roguelike. World Wide Web, 09 2013. http://killscreendaily.com/articles/brief-history-roguelike/. 5

[27] Facepunch Studios. Rust. World Wide Web, 2013. http://playrust.com/. 6

[28] SUPERDATA. Playable media is the next big thing in global games market. World Wide Web, 2015. https://www.superdataresearch.com/blog/global-games-market-2015/. 1

[29] Stefan Tilkov and Steve Vinoski. Node.js: Using javascript to build high-performance network programs. *IEEE Internet Computing*, 14(6):80–83, November 2010. 10

[30] Glenn R. Wichman. A brief history of "rogue". World Wide Web, 1997. http://www.wichman.org/roguehistory.html. 5

[31] Nicholas C. Zakas. The pluggable linting utility for javascript and jsx. World Wide Web, 2015. http://eslint.org/. 47

Appendix A

Code

A.1 random-matrix

```
1  function rand(x, y) {
2      var value;
3      var randomValue;
4
5      // assertions
6      if (!isInteger(x)) {
7          throw new Error('x must be an integer {Number}');
8      }
9
10     if (!isInteger(y)) {
11         throw new Error('y must be an integer {Number}');
12     }
13
14     // in order to make (1,2) a different result than (2,1)
15     x += X_ADDITIVE;
16     y += Y_ADDITIVE;
17
18     value = seed * x * y;
19     randomValue = Math.sin(value) * THROW_AWAY_DIGITS;
20
21     // just decimal part
22     return randomValue - Math.floor(randomValue);
23 }
```

Listing A.1: *random-matrix* implementation

A.2 Example of a world definition using JSON

```
{
    "seed": 1337,
    "numberOfSides": 4,
    "initialBlock": "B1",
    "initialMapSize": 10,
    "blocks": [{
        "id": "B1",
        "connectors": {
            "UP": "ALLOW_ALL",
            "RIGHT": "ALLOW_ALL",
            "BOTTOM": "ALLOW_ALL",
            "LEFT": "ALLOW_ALL"
        },
        "constraints": {
            "maxOccupation": 10
        }
    }, {
        "id": "B2",
        "connectors": {
            "UP": "ALLOW_ALL",
            "RIGHT": "ALLOW_ALL",
            "BOTTOM": "ALLOW_ALL",
            "LEFT": "ALLOW_ALL"
        },
        "constraints": {
            "minimumDistance": {
                "B1": 2
            }
        }
    }],
    "connectors": [{
        "id": "ALLOW_ALL",
        "type": "whitelist",
        "blockIds": ["B1", "B2", "B3"]
    }]
}
```

Listing A.2: Parser JSON example

A.3 Example of a map structure

```
[
    [
        { "x": -2, "y": 2, "block": "b3" },
        { "x": -1, "y": 2, "block": "b1" },
        { "x": 0, "y": 2, "block": "b2" },
        { "x": 1, "y": 2, "block": "b1" },
        { "x": 2, "y": 2, "block": "b3" }
    ],
    [
        { "x": -2, "y": 1, "block": "b2" },
        { "x": -1, "y": 1, "block": "b1" },
        { "x": 0, "y": 1, "block": "b3" },
        { "x": 1, "y": 1, "block": "b3" },
        { "x": 2, "y": 1, "block": "b3" }
    ],
    [
        { "x": -2, "y": 0, "block": "b1" },
        { "x": -1, "y": 0, "block": "b1" },
        { "x": 0, "y": 0, "block": "b3" },
        { "x": 1, "y": 0, "block": "b2" },
        { "x": 2, "y": 0, "block": "b3" }
    ],
    [
        { "x": -2, "y": -1, "block": "b3" },
        { "x": -1, "y": -1, "block": "b2" },
        { "x": 0, "y": -1, "block": "b3" },
        { "x": 1, "y": -1, "block": "b3" },
        { "x": 2, "y": -1, "block": "b2" }
    ],
    [
        { "x": -2, "y": -2, "block": "b2" },
        { "x": -1, "y": -2, "block": "b2" },
        { "x": 0, "y": -2, "block": "b3" },
        { "x": 1, "y": -2, "block": "b2" },
        { "x": 2, "y": -2, "block": "b3" }
    ]
]
```

Listing A.3: Map structure example

A.4 Block selection

```
1  /**
2   * @function selectBlock
3   * @description Returns a suitable and valid Block for a
        World position
4   */
5  function selectBlock(neighbours, blocksMap, mapStatus,
       position, getPartialMapFn, mapBounds, randomValue) {
6      var candidates = Object.keys(blocksMap);
7      var stackOfRules = [
8          useBlackAndWhiteListsRule
9          useMaxOccupationRule
10         useMaxOccupationPercentageRule
11         useMinimumDistanceRule
12     ];
13     var randomItemPosition;
14
15     stackOfRules.some(function applyAgainstCandidates(rule) {
16         candidates = rule(candidates, neighbours, blocksMap,
               mapStatus, position, getPartialMapFn, mapBounds,
               randomValue);
17         // bail when there is no candidates to avoid
               unnecessary work
18         return candidates.length === 0;
19     });
20
21     if (candidates.length === 1) {
22         return candidates[0];
23     }
24
25     if (candidates.length > 1) {
26         randomItemPosition = Math.floor(randomValue *
               candidates.length);
27         // because if randomValue === 1, array position will
               be outside bounds
28         randomItemPosition = Math.min(randomItemPosition,
               candidates.length - 1);
29         return candidates[randomItemPosition];
30     }
31
32     // there is no candidates
33     return null;
34 }
```

Listing A.4: Block selection

A.5 Example of a unit test using Mocha and ChaiJS

```
1  var expect = require('chai').expect;
2  var MapManager = require('./../../../../src/strategies/four-
       sides/map-manager.js');
3
4  describe('map-manager.js', function () {
5
6      describe('expandMap', function () {
7
8          describe('q1', function () {
9              var victim;
10
11             beforeEach(function () {
12                 victim = new MapManager(4);
13             });
14
15             it('should expand upwards', function () {
16                 var mapBounds;
17
18                 victim.expandMap(0, 0, 1, 2);
19                 mapBounds = victim.getWrappedBounds();
20
21                 expect(mapBounds.minX).to.be.equal(-2);
22                 expect(mapBounds.minY).to.be.equal(-2);
23                 expect(mapBounds.maxX).to.be.equal(1);
24                 expect(mapBounds.maxY).to.be.equal(2);
25             });
26
27             it('should expand to the right', function () {
28                 /* ... */
29             });
30
31         });
32
33     });
34
35 });
```

Listing A.5: Unit test example