Politecnico di Milano
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Development of a framework for user-based validation of FPS-oriented level design research

Supervisor: Professor Daniele LOIACONO

Final thesis by:
Marco BALLABIO Matr. 857169

Academic year 2017/2018

Thanks

I would like to thank Professor Daniele Loiacono for his help during the six months that led to the completion of this work.

Thank also to my whole family and to my colleague and friend Luca, essential companion in this journey.

Finally, I would like to thank Politecnico di Milano itself, of which I can call myself a proud student.

Marco Ballabio

Abstract

Level design plays a key role in the development of a video game, since it allows to transform the *game design* in the actual *gameplay* that the final user is going to experience. Nevertheless, we are still far from a scientific approach to the subject, with a complete lack of a shared terminology and almost no experimental validation for the most used techniques. Even if the video game industry doesn't acknowledge this problem, in the last years the academic environments have shown an increasing interest towards this subject.
We analyzed the main breakthroughs made in *level design* research applied to the genre of *First Person Shooters*, devoting particular attention to the ones that try to assist the design process by employing *Search-Based Procedural Content Generation* combined with *evolutionary algorithms*. We noticed that in most cases researchers recur to *validation* via *artificial agents*, since human play-test session are too time-consuming and allow to collect only a limited amount of data. Unfortunately, this solution decreases the scientific soundness of the obtained results, since the behavior of an AI, no matter how advanced, is different form the one of a human player. To solve this problem, we developed an *open-source framework* capable of deploying online experiments to collect data from real users via a browser FPS game. We also explored a novel approach to procedural generation of contents, developing a tool that uses *Graph Theory* to displace spawn-points and resources in a procedurally generated maps. Finally, we used our framework to validate this tool by means of an online data-collection campaign.

Sintesi

Il *Level design* gioca un ruolo chiave nello sviluppo di un videogioco, dal momento che permette di trasformare il *game design* nell'effettiva esperienza di *gameplay* che verrà sperimentata dall'utente finale. Nonostante ciò, siamo ancora lontani da un approccio scientifico verso la materia, a casusa della completa mancanza di un vocabolario condiviso e della quasi totale assenza di validazione sperimentale per le tecniche più comuni. Anche se l'industria tende ad ignorare questo problema, negli ultimi anni gli ambienti accademici hanno mostrato un crescente interesse verso questo campo.
Abbiamo analizzato le principali scoperte fatte nel campo del *level design* applicato al genere dei *First Person Shooter*, riservando particolare attenzione ai casi dove si usa la *generazione procedurali di contenuti* tramite *algoritmi evolutivi* per assistere il processo di design. Abbiamo notato che nella maggior parte dei casi i ricercatori ricorrono alla *validazione* tramite *agenti artificiali*, dal momento che organizzare sessioni di test con giocatori umani richiede molto tempo e permette di raccogliere una quantità limitata di dati. Dal momento che il comportamento di un'IA, per quanto avanzata, è molto diverso da quello di un giocatore umano, questa soluzione va a diminuire la solidità scientifica dei risultati ottenuti. Per ovviare a questo problema, abbiamo sviluppato un *framework open-source* per la raccolta di dati online da utenti reali, tramite un gioco FPS usufruibile da browser.
Abbiamo anche tentato un nuovo approccio alla generazione procedurale di contenuti, sviluppando uno strumento che utilizza la *Teoria dei Grafi* per disporre punti di respawn e risorse di vario tipo in una mappa generata proceduralmente. Abbiamo infine utilizzato il nostro framework per validare questo strumento tramite una raccolta di dati online.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Video games are really complex products, but it is rather simple to point out three elements that mainly influence their commercial success: *visuals*, **gameplay**[1] and *narrative*. If in the past visual improvements were impressively fast, with games looking more and more realistic from year to year, lately this progress consistently slowed down, leaving narrative and gameplay as the main selling points. Since video games are interactive products, the latter is the one that influences user experience the most. Gameplay is defined by a set of rules commonly referred as *game design*. To the player, game design is presented in a tangible way via *level design*, which consists in the creation of the worlds where the game takes place. This is a critical component, since an inadequate level design can easily compromise the whole experience.

One of the most successful video game genre is the one of **First Person Shooters**, that thanks to its first player perspective allows the user to experiment a complete immersion in the game world. From the very beginning, this has required a close attention to level design, that underwent a constant evolution, up to the stable situation of the recent years. Furthermore, the ever-increasing success of competitive multiplayer added a new level of complexity to the creation of maps, that need to support different game modes, play styles and interactions, allowing a fun and challenging gameplay to arise naturally.

[1] The specific way in which players interact with the game.

1.1 Motivations and purpose

Despite the importance of level design for the FPS genre, the video game industry has never attempted a scientific approach to this field. Consequently, game design is a rather abstract discipline, with no common vocabulary or well-defined standards, but rather based on the experience of who is working in this field from many years. This affects also the related literature, that is confined to listing the most used patterns and conventions, without focusing on why they work.

In the last years, instead, academic environments started to address this discipline with increasing interest. The researches performed in this field revolve around the identification and definition of design patterns and design techniques, with a deep analysis of how and why they work, and the creation of novel approaches to level design. Some of these techniques try to automate the design process by employing *procedural generation*, often combined with *evolutionary algorithms*.

These methods have achieved interesting results, but they are often criticized for employing matches played by *artificial agents* as a *validation method*. Since the behavior of an AI, no matter how advanced, is way less complex than the one of a human player, this approach impairs the scientific value of the final results. Nevertheless, researchers are forced to use AI to fasten up their experiment, because organizing play-test sessions requires too much time and resources.

The aim of this thesis is to solve this problem, by introducing an open-source *framework* capable of deploying online experiments to collect data from real users via a browser FPS game. Using this framework, we also explore the possibility of using *Graph Theory* to populate procedural generated maps with spawn-points and various kinds of resources.

1.2 Synopsis

The contents of the thesis are the followings:

In the second chapter we describe the state of the art of first person shooters, both in academic research and in commercial games, paying attention on how level design practices and procedural content generation are applied in this genre. We also list some examples of how Graph Theory is employed in video games.

In the third chapter we present the framework that we have developed, analyzing its features and its overall structure.

In the fourth chapter we present a tool that uses Graph Theory to place resources in procedurally generated levels, with an overview of the theory and the assumptions behind it.

In the fifth chapter we describe an experiment performed with our framework aimed at validating the framework itself and the tool described in the previous chapter.

In the concluding chapter we evaluate the obtained results and we analyze the potential future developments of this work.

Chapter 2

State of the art

In this chapter we analyze the current state of *Level Design* and of its common practices, both in academic and in professional environments, with attention to the genre of *First Person Shooters* (or *FPS*).

We then talk about *Procedural Content Generation* (or *PCG*), focusing on how it allows to enrich and ease the design process.

After that, we give an overview of the First Person Shooter genre, analyzing its features, history and evolution, devoting special attention to the games that mostly contributed to the definition of this genre.

Finally, we analyze how Graph Theory has been in used in Video Games during the years.

2.1 Level Design Theory

Level Design is a game development discipline focused on the creation of video game levels.

Today, the level designer is a well-defined and fundamental figure in the development of a game, but it was not always so. In the early days of the video game industry, it was a widespread practice to assign the development of levels to members of the team with other roles, usually programmers. Apart from the limited number of team members and the low budget, this was because there were no tools such as *level editors*[1], that allowed the level designer to work on a level without being involved with code.

The level designer has a really significant role in the development of a good game, since he is responsible for the creation of the world and for how

[1] A level editor is a software used to design levels, maps and virtual worlds for a video game.

the player interacts with it. The level designer takes an idea, which is the game design, and makes it tangible. Despite the importance of this role, after all this years it has not been established a common ground or a set of standards yet, instead, level design is often considered as a form of art, based on heuristics, observation, previous solutions and personal sensibility.

In addition to gameplay, the game designer must consider the visual appearance of the level and the technological limitations of the *game engine*[2], combining all this elements in a harmonic way.

One of the core components of level design is the **"level flow"**. For single player games it translates into the series of actions and movements that the player needs to perform to complete a level. A proficient level design practice is to guide the player in a transparent way, by directing his attention towards the path he needs to follow. This can be achieved in diverse ways. Power ups and items can be used as breadcrumbs to suggest the right direction in a one-way fashion, since they disappear once picked up. Lighting, illumination and distinctly colored objects are another common approach to this problem. A brilliant example of this is *Mirror's Edge*[3], which uses a really clear color code, with red interactive objects in an otherwise white world, to guide the player through its fast-paced levels. There are also even more inventive solutions, like the dynamic flock of birds in *Half Life 2*[4], used to catch the player attention or to warn him of incoming dangers[**?**]. Finally, sounds and architectures are other elements that can be used to guide the player. In the academic environment, a lot of researchers have analyzed the effectiveness of this kind of solutions: Alotto[**?**] considers how architecture influences the decisions of the player, whereas Hoeg[**?**] also considers the effect of sounds, objects and illumination, with the last being the focus of Brownmiller's[**?**] work.

In multiplayer games the level flow is defined by how the players interact with each other and with the environment. Because of this, the control of the level designer is less direct and is exercised almost exclusively by modeling the map. Considering FPSes, the level flow changes depending on how much an area is attractive for a player. The more an area is easy to navigate or offers tactical advantage, such as covers, resources or high ground, the more players will be comfortable moving in it. This doesn't mean that all areas need to be designed like this, since zones with a "bad" flow but an attractive reward, such as a powerful weapon, force the player to evaluate risks and

[2] A game engine is a software framework designed for the creation and development of video games.
[3] Digital Illusions CE, 2008.
[4] Valve, 2004.

benefits, making the gameplay more engaging. The conformation of the map and the positioning of interesting resources are used to obtain what Güttler et al.[**?**] define as "points of collisions", i.e. zones of the map were the majority of the fights are bound to happen.

Moving back to academic research, Güttler et al. have also noticed how aesthetic design loses importance in a multiplayer context. Other researches are instead focused on finding **patterns** in the design of multiplayer maps: Larsen[**?**] analyzes three really different multiplayer games, *Unreal Tournament 2004*[5], *Day of Defeat: Source*[6] and *Battlefield 1942*[7], identifying shared patterns and measuring their effect on gameplay, suggesting some guidelines on how to use them, whereas Hullet and Whitehead identify some patterns for single player games[**?**], many of whom are compatible with a multiplayer setting, with Hullett also proving cause-effect relationships for some of this patterns by confronting hypnotized results with the ones observed on a sample of real players[**?**]. Despite these experimental results contributing to a formalization of level design, we are still far from a structured scientific approach to the subject.

2.2 Procedural Content Generation

Procedural Content Generation refers to a family of algorithms used to create data and content in an automatic fashion. In game development it is commonly used to generate weapons, objects, maps and levels, but it is also employed for producing textures, models, animations, music and dialogues.

The first popular game to use this technique was *Rogue*[8], an ASCII dungeon exploration game released in 1980, where the rooms, hallways, monsters, and treasures the player was going to find were generated in a pseudo-random fashion at each playthrough. Besides providing a huge replay value to a game, PCG allowed to overcome the strict memory limitations of the early computers. Many games used pseudo-random generators with predefined *seed values* to create very large game worlds that appeared to be premade. For instance, the space exploration and trading game *Elite*[9] contained only eight galaxies, each one with 256 solar systems, of the possible 282 trillion the code was able to generate, since the publisher was afraid that such an high number could cause disbelief in the players. Another

[5]Epic Games, 2004.
[6]Valve, 2005.
[7]DICE, 2002.
[8]Michael Toy, Glenn Wichman, 1980.
[9]David Braben, Ian Bell, 1984.

example is the open world action role-playing game *The Elder Scrolls II: Daggerfall*[10], which game world has the same size of Great Britain.

As computer hardware advanced and CDs become more and more capacious, procedural generation of game worlds was generally put aside, since it could not compete with the level of detail that hand-crafted worlds were able to achieve.

However, in the last years, with the players' expectations and the production value of video games constantly increasing, procedural generation made a comeback as a way to automate the development process and reduce costs. Many *middleware* tools, such as *SpeedTree*[11] and *World Machine*[12], are used to produce various kind of content, like terrain and natural or artificial environments.

Many modern *AAA*[13] games use procedural generation: in *Borderlands*[14] a procedural algorithm is responsible for the generation of guns and other pieces of equipment, with over a million unique combinations; in *Left 4 Dead*[15] an artificial intelligence is used to constantly make the players feel under threat, by dynamically changing the music, spawning waves of enemies and changing the accessible paths of the level; in *Spore*[16] *procedural animation* is employed to determine how the creatures created by the player move.

Nowadays, PCG is widely used by *independent* developers, that, lacking the high budgets of AAA games, try to obtain engaging and unusual gameplay using unconventional means. The most famous example is *Minecraft*[17], a sandbox survival game which worlds, composed exclusively by cubes, are generated automatically. Currently, the most extreme form of procedural generation is the one found in *No Man's Sky*[18], a space exploration game, where space stations, star-ships, planets, trees, resources, buildings, animals, weapons and even missions are generated procedurally. Following in the footstep of their forefather, many roguelike games still use PCG, like *The Binding of Isaac*[19].

[10] Bethesda Softworks, 1996.
[11] IDV, Inc.
[12] World Machine Software, LLC.
[13] Video games produced and distributed by a major publisher, typically having high development and marketing budgets.
[14] Gearbox Software, 2009.
[15] Valve, 2008.
[16] Maxis, 2008.
[17] Mojang, 2011.
[18] Hello Games, 2016.
[19] Edmund McMillen, 2011.

All the algorithms used by these games and middleware are designed to be as fast as possible, since they need to generate the content in real time. In the last years researchers have nevertheless tried to explore new paradigms, creating more complex procedural generation techniques, that allow for a tighter control on the output. Being one of the problems of PCG the lack of an assured minimum quality on the produced content, the academic environment has focused not only on more advanced generation algorithms, but also on techniques to evaluate the output itself in an *automatic* fashion. In this field, Togelius et al.[**?**] defined *Search-Based Procedural Content Generation*, a particular kind of *Generate-And-Test*[20] algorithm, where the generated content, instead of being just accepted or discarded, is evaluated assigning a suitability *score* obtained from a *fitness function*, used to select the best candidates for the next iterations.

2.3 Procedural Content Generation for FPS maps

We have really few examples of commercial FPSes that use PCG to generate their maps: with the exception of *Soldier of Fortune II: Double Helix*[21], that employs these techniques to generate whole missions, the few other cases we have are all roguelikes with a FPS gameplay, like *STRAFE*[22].

Despite the total lack of FPSes using procedural generation to obtain multiplayer maps, researchers have proved that search-based procedural content generation can be an useful tool in this field. This method has been applied for the first time by Cardamone et al.[**?**], who tried to understand which kind of *deathmatch*[23] maps created the most enjoyable gameplay possible. To achieve this, the authors generate maps for *Cube 2: Sauerbraten*[24] by maximizing a fitness function computed on *fight time* data collected from *simulations*[25], with the fight time being the time between the start of a fight and the death of one of the two contenders. The choice of this fitness func-

[20] Algorithms with both a generation and an evaluation component, that depending on some criterion, decide to keep the current result or to generate a new one.

[21] Raven Software, 2002.

[22] Pixel Titans, 2017.

[23] A widely used multiplayer game mode where the goal of each player is to kill as many other players as possible until a certain end condition is reached, commonly being a kill limit or a time limit.

[24] Wouter van Oortmerssen, 2004

[25] In the field of search-based procedural generation, fitness function based on simulation are computed on the data collected from a match between artificial agents in the map at issue. They differ from *direct* and *interactive* functions, that evaluate, respectively, the generated content and the interaction with a real player.

tion is based on the consideration that a long fight is correlated with the presence of interesting features in the map, such as escape or flanking routes, hideouts and well positioned resources.

Stucchi[**?**], yet remaining in the same field, attempted a completely different use of procedural generation, by producing balanced maps for player with different weapons or different levels of skill. For doing so, he generates procedural maps via evolutionary algorithms, evaluating them with a fitness function based on simulation that computes the entropy of kills. Starting from a situation where one of the two players has a significant advantage, Stucchi is able to prove that changes in the map structure allow to achieve a significant balance increase.

Arnaboldi[**?**] combined these two approaches, creating a framework that automatically produces maps using a genetic process like the one of Cardamone. In Arnaboldi's work, however, the fitness function is way more complex, since it considers a high number of gameplay metrics, and the AI of the *bots*[26] is closer to the one of a human player, thanks to a series of adjustments made to the stock *Cube 2* one. These improvements significantly increase the scientific accuracy and the overall quality of the output, allowing to identify and analyze some recurring patterns and their relationship with the statistics gathered during the simulations.

Ølsted et al.[**?**] moved the focus of their research from deathmacth to squad game modes with specific objectives, sustaining not only that the maps generated by Cardamone et al. are not suitable for this kind of gameplay, but also that they do not conform to what they define as *The Good Engagement* (or *TGE*) rules, i.e. a set of rules that a FPS should satisfy to support and encourage interesting player choices, from which an engaging gameplay should emerge naturally. By analyzing the *Search & Destroy*[27] mode of games like *Counter Strike*[28] and *Call of Duty*[29], they defined a process to generate suitable maps: starting from a grid, some nodes are selected and connected among them, the result is then optimized to satisfy the TGE rules and finally rooms, resources, objects and spawn points are added, as can be seen in figure 2.1. Opting for an *interactive* approach, the fitness function used to guide the evolution of these maps is computed on the binary appreciation feedback expressed by real users, since the authors consider bot behavior too different form the one of real users.

[26] The artificial players of a video game.
[27] A multiplayer game mode where players, divided in two teams, have to eliminate the enemy team or detonate a bomb in their base.
[28] Valve Software, 2000
[29] Infinity Ward, 2003

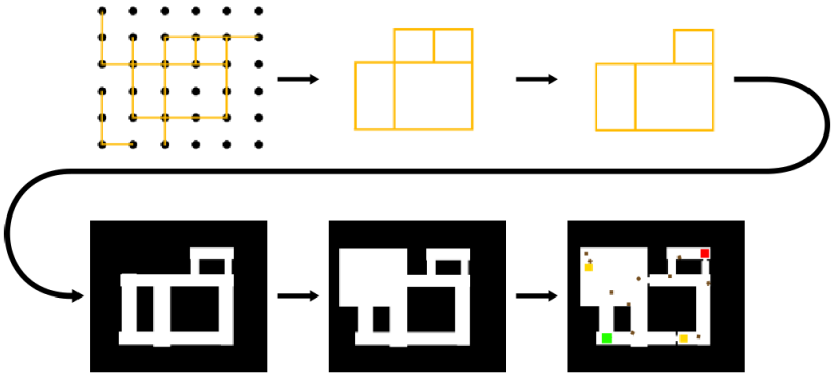


Figure 2.1: Visual representation of Ølsted et al.[**?**] generative process.

A completely different approach from the ones listed above is the one of Anand and Wong[**?**], who employed search-based procedural generation to create *online*, automatically and rapidly multiplayer maps for the *Capture and Hold*[30] game mode, without compromising the quality of the final output. To achieve this, they employ a genetic approach, which fitness function is evaluated directly on the topology of the map, considering four different factors: the connectivity between regions, the number of points of collision, the balancing in the positioning of control points and spawn points. With no need to simulate matches, this process can be completed in a matter of seconds. The algorithm starts by generating three maps, that are then evolved by mutation. To obtain the initial maps, Anand and Wong populate a grid with random tiles, they clean it of undesired artifacts and they identify regions within it, that are then populated with strategic points, resources, spawn points and covers. Despite its good results, this approach is not sound enough on a scientific stand point, since it directly depends on the validity of the selected topological metrics and, as we have seen, it is still not clear which the good elements of a level are.

Finally, Cachia et al.[**?**] extended search-based procedural generation to multi-level maps, generating the ground floor with one of the methods defined by Cardamone and employing a random digger for the first floor. The final result can be seen in figure 2.2. Their algorithm also positions

[30] A multiplayer game mode where players, divided in two teams, fight for the control of some strategic areas. The score of each team increases over time proportionally to the number of controlled points until one of the two teams reaches a given limit, winning the game.

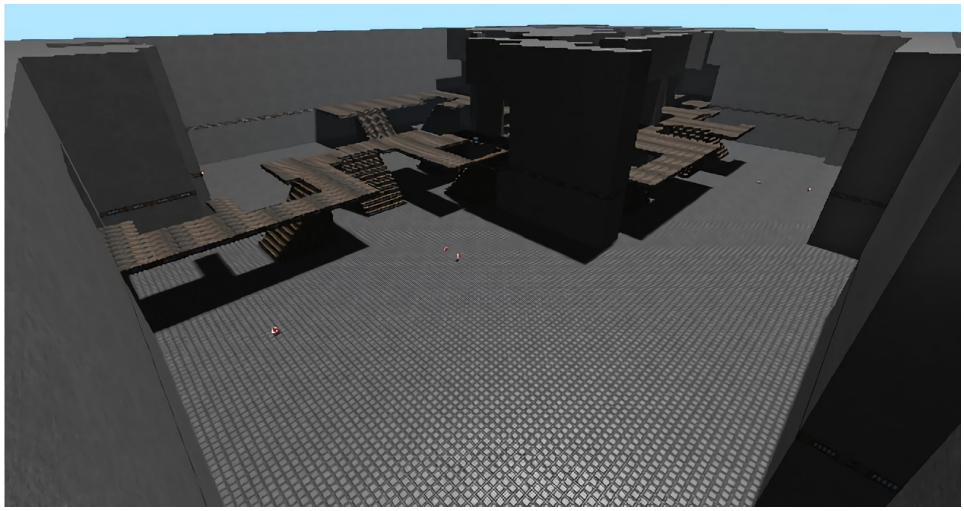


Figure 2.2: One of the maps evolved by Cachia's et al.[**?**] algorithm.

spawn points and resources through a topological fitness function, which entails the same problems described for Anand and Wong's approach.

2.4 History of Level Design in FPSes

First Person Shooter are a video game genre which main features are the first-person player perspective and a weapon-based combat gameplay. During the years this genre has been consonantly evolving, as new elements started to emerge from the very aggressive and fast-paced gameplay of the first games of this kind, like *Doom*[31]. Games like *Half-Life*[32] highlighted the importance of the story and of the setting, games like *Deus Ex*[33] introduced role-play and stealth mechanics and games like *Quake III: Arena*[34] and *Unreal Tournament*[35] moved the focus from single player to multiplayer. Finally, Modern Military Shooters introduced a slower and more realistic gameplay and radically changed the setting from fictional conflicts to contemporary ones.

[31] ID Software, 1993.
[32] Valve Software, 1998.
[33] Ion Storm, 2000.
[34] id Software, 1999.
[35] Epic Games, 1999.

2.4.1 Level Design evolution in FPSes

The evolution of the mechanics of this genre was supported by a constant refinement of level design and of the used tools.

Before 1992: The first FPSes

The origin of the FPS genre goes back to the beginning of video games themselves. In 1973, Steve Colley developed *Maze War*, a simple black-and-white multiplayer game set in a tile-based maze, where players would search for other players' avatars, killing them to earn points. Around the same time, Jim Bowery created *Spasim*, a simple first-person space flight simulator. Both games were never released, instead, we must wait the beginning of the 80's to see the first products available to the public. Heavily inspired by Colley and Bowerey's work, these games had almost no level design, but during the years they started to become a little more complex, with tangled sci-fi structures taking over mazes.

1992: Wolfenstein 3D defines a new genre

When *Wolfenstein 3D*[36] was released in 1992, it changed the genre forever, thanks to its fast gameplay and its light game engine, that allowed to target an audience as wide as possible. Wolfenstein 3D took up the exploitative approach of the period, with its levels full of items, weapons and secret rooms. The level design was still simple, because the technical limitations of the engine and of its *tile-based*[37] *top-down*[38] level editor allowed to create only flat levels, with no real floor or ceiling and walls always placed at a right angle.

1993 - 1995: Doom and its legacy

A year later, id Software released *Doom*, a milestone in the history of the genre. The game engine of Doom was capable of many innovations: sections with floor and ceiling with variable height, elevators, non-orthogonal walls, interactive elements, lightning, even dynamic, textures for horizontal

[36] id Software, 1992.
[37] The map consists in a grid composed by squared cells, or *tiles*, of equal size.
[38] In *top-down* games and editors, the game world is seen from above.

surfaces and even a simple *skybox*[39], all of this without losing the speed that characterized Wolfenstein. The developers took advantage as much as possible of the capabilities of the engine, achieving a level design way more complex than what had been seen before. In addition, Doom provided the users with *WADs*, a tool to create levels and mod the game, and it featured cooperative and competitive multiplayer, via LAN or dial-in connections, that rapidly gathered a massive user base.

The impact of Doom on the genre was so strong that in the following years the market was flooded with its clones. All these games, as well as Doom, had still some technical limits, in their engines, that were not completely 3D, in their level editors, that were still top-down and did not allow for the design of more complex levels, and in their gameplay, that still faced limited movements.

1996 - 2000: A constant evolution

In the next years, many games continued what Doom started, bringing constant improvements to the genre. *Duke Nukem 3D*[40] set aside the sci-fi settings of its predecessors, switching to real locations, inspired by the ones of Los Angeles. This was possible thanks to its *2.5D*[41] engine *Build*[42], that provided a *What You See Is What You Get*[43] level editor. Furthermore, Build allowed to apply *scripts* to certain elements of the map, resulting in a more interactive environment.

Released in 1996, *Quake*[44] was one of the first and most successful FPSes with a real 3D engine, that allowed an incredible jump forward in terms of realism, level design and interactivity. Quake had also a rich multiplayer, with a lot of maps, game modes and special features, like clans and modding. Two years later, *Unreal*[45] brought a new improvement in terms of realism

[39] A method of creating backgrounds that represent scenery in the distance, making the game world look bigger than it really is.

[40] 3D Realms, 1996.

[41] A *2.5D* engine renders a world with a two dimensional geometry in a way that looks three-dimensional. An additional height component can be introduced, allowing to render different ceiling and floor height. This rendering technique limits the movements of the camera only to the horizontal plane. Doom and many similar games employed this technology.

[42] Developed by Ken Silverman in 1995.

[43] In computer science, *What You See Is What You Get* denotes a particular kind of editors where there is no difference form what you see during editing and the final output.

[44] id Software, 1996.

[45] Epic Games, 1998.

and level design, thanks to its engine capable of displaying huge outdoors settings.

In those years many new sub-genres started to spawn, obtained by emphasizing certain features of the previous games or by borrowing mechanics from other genres. *Quake III Arena*[46] and *Unreal Tournament*[47] were some of the first and most successful multiplayer games ever released, that required a new approach to level design, completely focused on the creation of competitive maps. Games like *Deus Ex* introduced tactical and RPG elements, with the possibility of reaching an objective in multiple ways, thanks to a level design that exalted the freedom left to player. Finally, *Half-Life* changed forever the approach of this genre to storytelling and to level design: the game puts great emphasis on the story, that is narrated from the eyes of the player, without cut-scenes, and introduces the possibility of moving freely between areas, with no interruption. The game also set new standards with its challenging enemy AI, capable of taking advantage of the terrain and coordinating flanking maneuvers.

2001: The rise of console shooters

In 2001, *Halo: Combat Evolved*[48] revolutionized the genre, introducing some of the mechanics on which modern FPSes are based, as the limited amount of weapons that the player can carry and the regeneration of health over time. This slower and more strategic approach to gameplay matched perfectly with consoles and their twin-stick-controllers, that were not suitable for the extremely fast paced action of the past. Over time, this new approach to gameplay overshadowed almost completely all the others, thanks to the diffusion of consoles and to the increase of production costs, that made PC exclusives economically disadvantageous. From the standpoint of level design, this change required to increase the complexity of the levels and the addition of strategically placed covers.

Today: A time of stagnation

Starting from its origins, level design undergone a radical evolution, but in the last years it has shown no significant improvements. This could be due to the considerable risk associated with modern projects or to the lack of

[46] id Software, 1999.
[47] Epic Games, 1999.
[48] Bungie, 2001.

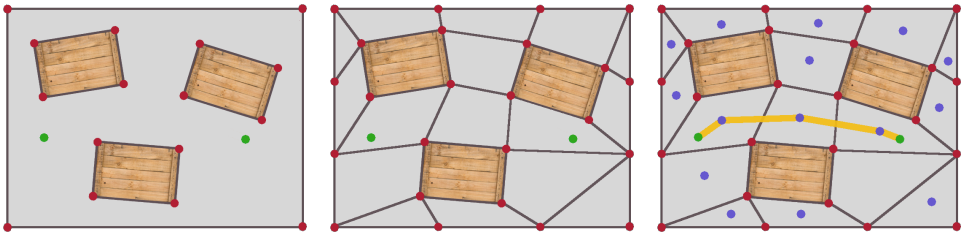


Figure 2.3: A common pathfinding process.

suitable instruments.

2.5 Graph Theory in video games

Graph Theory has always been used in video games, usually as a useful tool to perform *pathfinding* for artificial agents. These techniques revolve around the creation of a *navigation mesh*, i.e. a representation of the walkable areas of a level using non-overlapping polygons obtained by removing the shapes of obstacles from the considered surface. The result is then used to generate a graph, selecting as nodes the vertices or the centers or the centers of edges of the obtained polygons, depending on the kind of movement that must be achieved. This graph can be pre-generated or computed at run-time, if the technique is applied to dynamic environments. Finally, an algorithm like $A*$ is employed to find the shortest path between two points. This process can be seen in figure 2.3

Graphs are also used in procedural generation, as an effective tool to model landmasses and roads.

2.6 Summary

In this chapter we analyzed the current state of level design for first person shooters and how this field has been explored in academic research. We then introduced Procedural Content Generation and we observed how some studies have proved that it is a suitable method to produce maps for FPS games. We also took a brief look at the history of first person shooters, analyzing how level design evolved over time. Finally, we depicted some of the most common uses of Graph Theory in video games.

Chapter 3

The research framework

In this chapter we describe the *framework* that we have developed to perform user-based online validation of procedurally generated maps for multiplayer Firsts Person Shooters. After a quick overview, we present the map formats that the framework supports and we extensively analyze its structure, its components and its features.

3.1 Description of the framework

We designed our framework with the objective of providing a valid alternative to the games currently employed as a validation tool in this research field. All the available options, like *Cube 2: Sauerbraten*, are powerful tools to perform validation via artificial agents, but they are not suitable for user-based validation. A data-collection campaign based on these games requires to download the game or to take part in real-life play-test sessions, but these options discourage potential participants because they are significantly time-consuming. For this reason, we decided to develop a *Unity*[1] framework that is as light as possible, with a WebGL build weighting less than 10MB that can be played using any browser.

Since the purpose of this tool is to be used in research, we decided to support most map representation formats used in previous works and we designed our framework to be as modular, expansible and configurable as possible.

[1]Unity Technologies, 2005. *Unity* is a game development environment that includes a game editor and a game engine. Currently, it is the most used game development tool.

3.2 Map representation

Maps are structured as grids of orthogonal *tiles* and are internally represented by matrices of characters, where each cell corresponds to a specific *tile*. Depending on the character it contains, a cell can represent a wall, a floor or an object on the floor. If a cell corresponds to a wall tile we say it is *filled*, if it corresponds to a floor tile we say it is *empty*. The framework supports multi-level maps, which are represented by a list of matrices, with each matrix corresponding to a level.

The framework allows to represent maps in two more formats, that are converted to the internal one when provided as input.

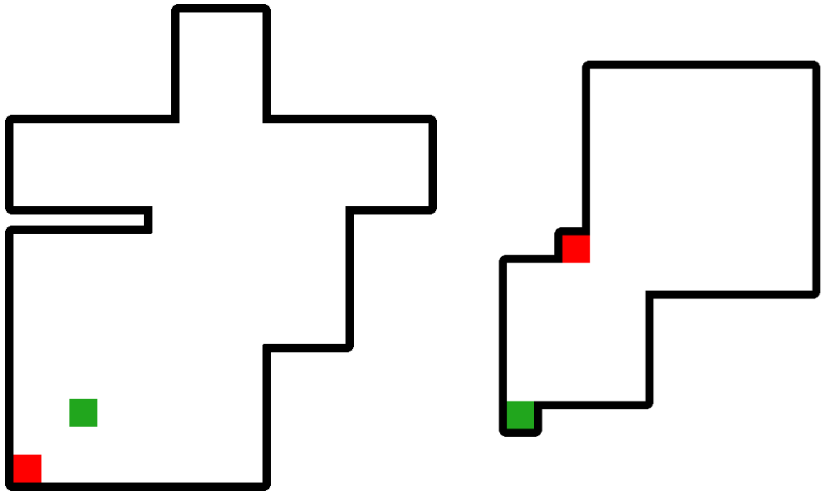
3.2.1 Text representation

The text representation allows to encode the map as a text, of which each line corresponds to a row of the internal matrix representation and each character corresponds to a cell. For multi-level maps, the format is the same, with the exception of blank lines used to separate the floors, that are encoded from the lower one to the higher one.

3.2.2 All-Black representation

Our All-Black representation is an extended version of the one defined by Cardamone et al.[**?**], to which we have added the support for objects and multi-level maps. In their work, the All-Black representation encodes the empty areas of an otherwise filled map, consisting in square rooms and corridors of fixed width. Rooms are defined by $\langle x, y, s \rangle$ triples, where x and y define the coordinates of the center of the room and s defines its width. Corridors are rectangular areas with a fixed width of 3 cells and are defined by $\langle x, y, l \rangle$ triples, where x and y define the point in which the corridor starts and l defines its length. l also provides the direction of the corridor: if l is positive the corridor extends along the x-axis, otherwise it extends along the y-axis. With respect to this representation, we changed the encoding of the rooms by considering x and y as the coordinates of the corner closer to the origin; this change allows to remove any ambiguity deriving from the position of the center of a room of even width.

For allowing the encoding of objects, we added a third kind of triple, $\langle x, y, o \rangle$, that uses x and y to denote the coordinates of the tile that hosts the objects and o to denote the object itself, encoded as a character. In our representation, first we store the triples representing the rooms, then



(a) Map represented by ⟨5, 5, 9⟩ ⟨10, 10, 7⟩ ⟨15, 25, 3⟩ | ⟨5, 15, 15⟩ ⟨11, 15, −7⟩ | ⟨5, 5, s⟩ ⟨7, 7, d⟩.

(b) Map represented by ⟨1, 2, 5⟩ ⟨4, 6, 8⟩ | ⟨5, 6, −10⟩ ⟨10, 15, −6⟩ | ⟨3, 7, s⟩ ⟨1, 1, d⟩.

Figure 3.1: Two simple maps with their All-Black representation.

the triples representing the corridors and finally the triples representing the objects. These groups are separated by the special character "|" and can have any number of elements, with the exception of the one denoting the rooms, that must have at least one triple.

We have also extended the All-Black representation including the one defined by Cachia et al.[**?**], that allows to encode maps generated with a random digger algorithm, i.e. an algorithm that randomly moves in a filled map emptying all the cells it crosses (for more details see subsubsection 3.3.3). The map is encoded by a quintuple $\langle f, l, r, v, s \rangle$, where f encodes the probability of moving forwards, l encodes the probability of turning left, r encodes the probability of turning right, v encodes the probability of jumping to a visited cell and s encodes the probability of placing a flight of stairs, if in a multi-level setting. With respect to the representation defined by Cachia et al., we added the possibility of encoding objects, which group of triples is separated by the digger quintuple by the special character "|".

Multi-level maps are represented by encoding the floors from the lower one to the higher one using one of the two single-level All-Black formats that we have defined. The encoding of different floors are separated using the double special characters "||".

Figure 3.1 shows two maps with their All-Black representation.

3.3 Framework structure

The framework collects data by assigning to the users *matches* to play. A match is defined by the *game mode* and by the *map type*, which in turn is defined by the *map topology* and by the *map appearance*. The *map topology* defines how the map is going to *be* and depends on the algorithm used to generate it, whereas the *map appearance* defines how the map is going to *look* and depends on how the map is assembled. This implies that the map type defines a whole array of procedurally generated maps that share the same topology and appearance. Therefore, when referring to a match we are considering a specific game-mode played in a procedurally generated map. If needed, it is possible to use a pre-generated map instead of generating a new one, by providing it as input using one of the supported formats. In this case the *map topology* defines how to interpret the input, that is then displayed considering the *map appearance*.

A match is defined by combining different modular *Manager* objects, each of which controls a different aspect of the match. To assure their interchangeability, most modules are defined by their own abstract class.

3.3.1 The Game Manager

The *Game Manager* is the module responsible for the overall behavior of a match. Each game mode consists in a different implementation of the *Game Manager*. It leans on the *Map Manager* for the generation and the assembly of the map and on the *Spawn Point Manager* for the spawn of entities. The *Game Manager* controls the life-cycle of the match, that can be divided in the following phases:

- *Setup*: all the modules are initialized.
- *Generation*: the *Map Manager* generates or imports the map and assembles it.
- *Ready*: the *Game Manager* displays a countdown announcing the start of the game.
- *Play*: the *Game Manager* handles the game while the *Experiment Manager* logs the actions of the player, if needed. This phase continues until an end condition is satisfied.
- *Score*: the *Game Manager* stops the game and displays the final score.

3.3.2 The Map Manager

The *Map Manager* controls the generation, the import and the assembly of the map and the displacement of objects inside it. It leans on the *Map Generator* for the generation, on the *Map Assembler* for the *assembly*[2] and on the *Object Displacer* for the *displacement*[3], whereas it performs the import itself. If the map is provided in input as a text file, the *Map Generator* is not called, whereas it is used to perform decoding if the map is provided in All-Black format.

The framework provides three different implementations of the *Map Manager*.

Single-Level Map Manager

The *Single-Level Map Manager* is used for any kind of single level map. It can generate maps, import them from file or decode them from All-Black format.

Multi-Level Map Manager

The *Multi-Level Map Manager* is used for any kind of map that has more than one floor. It can generate multi-level maps or import them from file, but it cannot perform All-Black decoding. In addition to the standard modules, it employs a *Stairs Generator* to position flight of stairs to connect the different floors.

Multi-level maps are obtained by using at least one generator to produce the desired number of floors. Since this allows to combine different kind of generators, we were able to obtain maps similar to the ones evolved by Cachia et al.[**?**] (see figure 3.2), as well as maps with a more complex and interesting layout than the ones obtained by previous works (see figure 3.3).

All-Black Multi-Level Map Manager

The *All-Black Multi-Level Map Manager* is used to decode multi-level maps saved in All-Black format. If no stairs are found among the objects, it employs the *Stairs Generator* to position them.

[2]With *assembly* we mean the operation of creating a 3D model of the map starting from its matrix representation.

[3]With *displacement* we mean the operation of placing the 3D models of the objects in the assembled map, according to their position defined by the *Map Generator* trough a *positioning* algorithm.

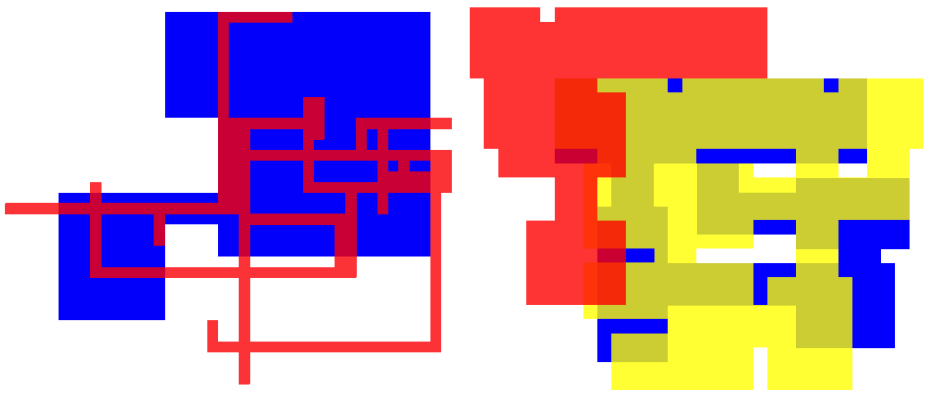


Figure 3.2: Multilevel map with a divisive floor (in blue) and a digger floor (in red).

Figure 3.3: Multilevel map with three divisive floors.

3.3.3 The Map Generator

The *Map Generator* controls the generation of the map. Each implementation of the *Map Generator* defines a different *map topology* depending on the used generation algorithm and on how its parametric setting are tuned. Some of these settings are shared by all the implementations, whereas some of them are implementation-specific.

The shared settings are used to define the size of the map and its encoding, to define the objects and to impose some constraints on their positioning:

- *Width*: the number of rows of the matrix that represents the map.
- *Height*: the number of columns of the matrix that represents the map.
- *ObjectToObjectDistance*: the minimum number of cells that must separate two objects.
- *ObjectToWallDistance*: the minimum number of cells that must separate an object and a wall.
- *BorderSize*: the width of the border placed all around the map once it has been generated, expressed in number of cells.

- *RoomChar*: the character used to represent a clear cell where the player can walk.
- *WallChar*: the character used to represent a filled cell where the player cannot walk.
- *MapObjects*: a list of the objects that must be placed in the map.

The objects contained in *MapObjects* can represent spawn points, resources or decoration. They have the following properties:

- *ObjectChar*: the character used to represent the object.
- *NumObjPerMap*: the number of objects of that kind that must be placed in the map.
- *PlaceAnywhere*: if this value is set to true, the restriction on the distance from the walls is ignored.
- *PositioningMode*: the algorithm used to position the object in the map.

The framework provides three different algorithms to position the objects inside the map:

- *Rain*: positions the objects selecting random cells from the ones that are empty and satisfy the *ObjectToWallDistance* constraint.
- *Rain Shared*: positions the objects selecting random cells from the ones that are empty and satisfy the *ObjectToWallDistance* constraint and the *ObjectToObjectDistance* constraint on the objects that have been placed using *Rain Shared*.
- *Rain Distanced*: positions the objects selecting random cells from the ones that are empty and satisfy the *ObjectToWallDistance* constraint and the *ObjectToObjectDistance* constraint on the objects with the same *ObjectChar*.

All of the following implementations of the *Map Generator* are deterministic, since they require a *seed* value as input that constrains the output to a specific map.

Cellular Generator

The *Cellular Generator* employs a parametric *cellular automaton*[4] to generate a natural looking map.

The algorithm starts by filling some tiles of the map selected at random, then it applies the cellular automaton for a certain number of generations and finally it performs some refinements (for more details, see algorithm 1). The resulting topology depends on the following parameters:

- *RandomFillPercent*: the percentage of tiles that are randomly filled during the initialization of the algorithm. High values promote narrow spaces, small values promote wide open areas.
- *SmoothingInteration*: the number of generations the cellular automaton is ran for. High values penalize small features and make the walls smoother.
- *NeighbourTileLimitLow*: the minimum number of neighbors a cell must have to became filled. Its value must be lesser or equal than the one of *NeighbourTileLimitHigh*. The map becomes noisier the more they diverge.
- *NeighbourTileLimitHigh*: the maximum number of neighbors a cell must have to became empty.
- *WallThresholdSize*: the minimum number of cells that an isolated filled region must include to not be deleted. High values penalize small filled regions.
- *RoomThresholdSize*: the minimum number of cells that an isolated void region must include to not be deleted. High values penalize small empty regions.
- *PassageWidth*: the width of a passage connecting two different areas, expressed in number of cells.

Figure 3.4 shows how these parameters influence the topology of a map.

The *Cellular Generator* can perform import and export using the text representation.

[4] A *cellular automaton* consists of a grid of cells, each in one of a finite number of states, such as on and off. For each cell, a set of cells called its neighborhood is defined, usually composed by the ones that share at least one vertex with it (referred as *8-neighbors*). Given the current state of the grid, a new generation is created, according to some fixed rule that determines the new state of each cell depending on the current state of the cell and of its neighbors.

Divisive Generator

The *Divisive Generator* employs a *binary space partitioning algorithm* to generate a man-made looking map.

The algorithm starts by obtaining partitions of the map by recursively dividing it in two sides of random size along one of the axes, then it selects some of these partitions as rooms and finally it connects them with corridors (for more details, see algorithm 2). The resulting topology depends on the following parameters:

- *RoomDivideProbability*: probability of a partition being divided again. High values promote small rooms.
- *MapRoomPercentage*: minimum percentage of tiles of the map that must be empty. High values promote close rooms separated by walls, low values promote distant rooms connected by corridors.
- *DivideLowerBound*: minimum division point expressed as percentage of the dimension of the room.
- *DivideUpperBound*: maximum division point expressed as percentage of the dimension of the room.
- *MinimumRoomDimension*: minimum width expressed in number of cells that a partition must have to be divided again. High values promote large rooms.
- *MinimumDepth*: the minimum number of recursive divisions that each partition must have experienced.
- *PassageWidth*: the width expressed in number of cells of the corridors connecting the rooms.
- *MaxRandomPassages*: the number of additional corridors to place, if possible, once that all the rooms are connected.

Figure 3.5 shows how these parameters influence the topology of a map.

The *Divisive Generator* can perform both import and export using the text representation, whereas the All-Black format is used only for export. The latter matches perfectly with this generator, since both are based on the concept of rooms and corridors.

Digger Generator

The *Digger Generator* employs a simple algorithm to generate a man-made looking map.

The algorithm is iterative and its state is defined by the current cell and by the current direction, that together with a randomly selected action determine the next cell that the algorithm is going to visit. Starting from the central cell of a filled map, at each iteration the algorithm empties the current cell and randomly decides if moving forward, turning left, turning right, jumping to a random visited cell or placing a flight of stairs, if controlled by a *Multi-Level Generator*. The algorithm stops when a certain percentage of cells has been emptied. The resulting topology depends on the following parameters:

- *ForwardProbability*: probability of moving forward in the next iteration. High values promote long corridors.
- *LeftProbability*: probability of moving leftward in the next iteration. High values promote wide open areas.
- *RightProbability*: probability of moving rightward in the next iteration. High values promote wide open areas.
- *VisitedProbability*: probability of jumping to a visited cell in the next iteration. High values promote a more complex topology.
- *StairProbability*: probability of placing a flight of stairs.
- *RoomPercentage*: percentage of tiles of the map that must be empty.

Figure 3.6 shows how these parameters influence the topology of a map.

The *Digger Generator* can perform both import and export using the text representation, whereas its own All-Black format is used only for import.

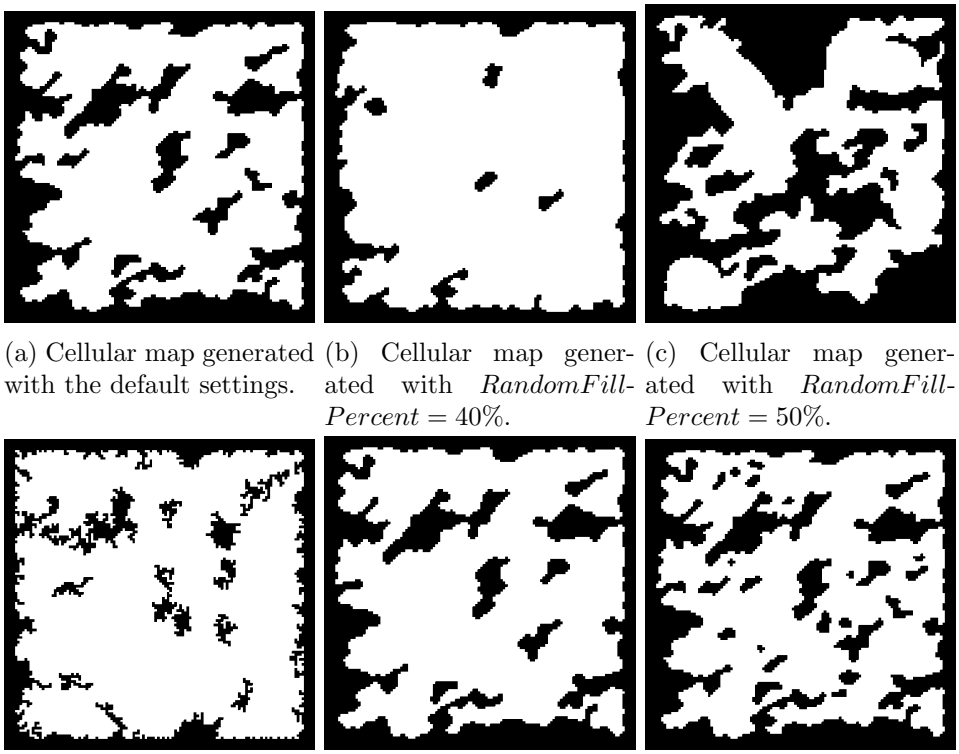
All-Black Generator

This simple generator parses inputs expressed in All-Black format, extracting rooms and corridors. If no objects are specified, it adds them to the map.

Algorithm 1: Cellular generation algorithm.

```
for every cell in the map do
    empty the current cell;
end
while percentage of filled cells < RandomFillPercent do
    select a random cell;
    fill the selected cell;
end
for generation from 0 to SmoothingIterations do
    for every cell in the map do
        count the 8-neighbors of the cell;
        if 8-neighbors count > NeighbourTileLimitLow then
            mark the current cell as filled for the next generation;
        end
        if 8-neighbors count < NeighbourTileLimitHigh then
            mark the current cell as empty for the next generation;
        end
    end
    update the map to the next generation;
end
for every isolated region of empty cells do
    if #cells in the region < RoomThresholdSize then
        fill all the cells in the region;
    end
end
for every isolated region of filled cells do
    if #cells in the region < WallThresholdSize then
        empty all the cells in the region;
    end
end
connect all the regions composed by empty cells;
place the objects;
```

This algorithm is a modified version of the one proposed by Sebastian Lague[**?**].



(a) Cellular map generated with the default settings.

(b) Cellular map generated with $RandomFillPercent = 40\%$.

(c) Cellular map generated with $RandomFillPercent = 50\%$.

(d) Cellular map generated with $SmoothingIterations = 0$.

(e) Cellular map generated with $SmoothingIterations = 3$.

(f) Cellular map generated with $WallThresholdSize = 5$.

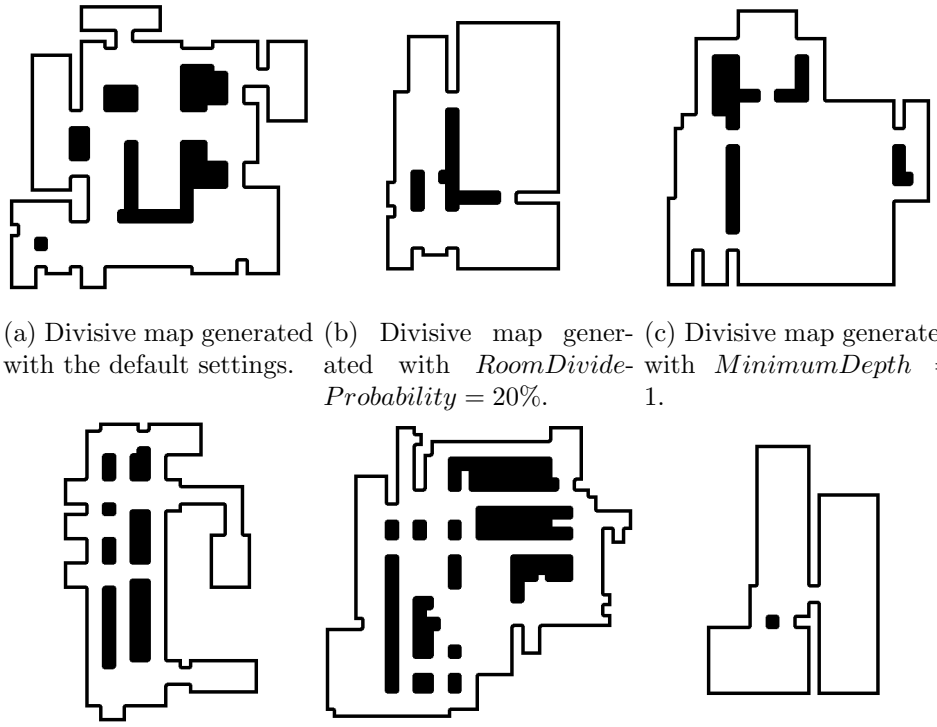
Figure 3.4: Six maps generated by the Cellular Generator using "*ANotSoRandomSeed*" as seed, but different settings.

By default, the Cellular Generator has *RandomFillPercent* set to 45%, *SmoothingIterations* set to 2, *NeighbourTileLimitHigh* set to 4, *NeighbourTileLimitLow* set to 4, *WallThresholdSize* set to 40 and *RoomThresholdSize* set to 100.

Algorithm 2: Divisive generation algorithm.

```
for every cell in the map do
    fill the current cell;
end
initialize the partitions list;
DivideRoom(map, 0);
while percentage of empty tiles < MapRoomPercentage do
    extract a partition from the partitions list at random;
    make the partition a room;
    empty the tiles in the room;
end
connect the rooms;
while all the rooms are not directly connected and #placed
 additional corridors < MaxRandomPassages do
    add an additional corridor between two rooms selected at
     random;
end
place the objects;
```

```
Function DivideRoom(section, depth) is
    if (true with probability roomDivideProbability and partition
     width > minimumDividableRoomDimension and partition
     heigth > minimumDividableRoomDimension) or depth <
     minimumDepth then
        if previous division was horizontal then
            perform a random vertical division between
             divideLowerBound and divideUpperBound;
        else
            perform a random horizontal division between
             divideLowerBound and divideUpperBound;
        end
        DivdeRoom(first sub-section, depth + 1);
        DivdeRoom(second sub-section, depth + 1);
    else
        add the partition to the partitions list;
    end
end
```



(a) Divisive map generated with the default settings.

(b) Divisive map generated with $RoomDivideProbability = 20\%$.

(c) Divisive map generated with $MinimumDepth = 1$.

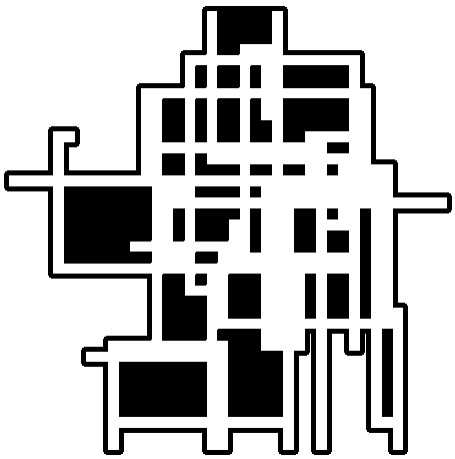
(d) Divisive map generated with $MinimumDepth = 8$.

(e) Divisive map generated with $MinimumRoomDimension = 1$.

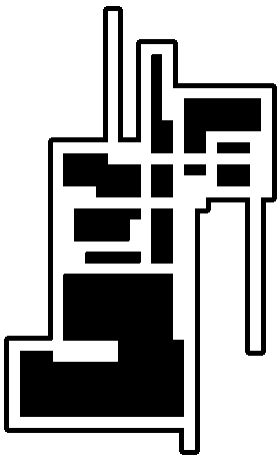
(f) Divisive map generated with $MinimumRoomDimension = 7$.

Figure 3.5: Six maps generated by the Divisive Generator using "*AModeratelyRandomSeed*" as seed, but different settings.

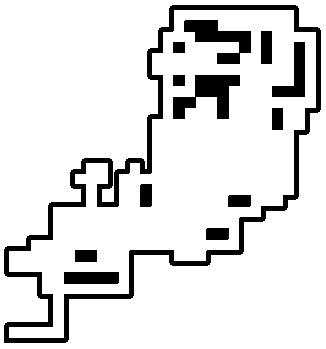
By default, the Cellular Generator has *RoomDivideProbability* set to 80%, *MapRoomPercentage* set to 90%, *DivideLowerBound* set to 10%, *DivideUpperBound* set to 90%, *MinimumRoomDimension* set to 3, *MinimumDepth* set to 4, *PassageWidth* set to 3 and *MaxRandomPassages* set to 12.



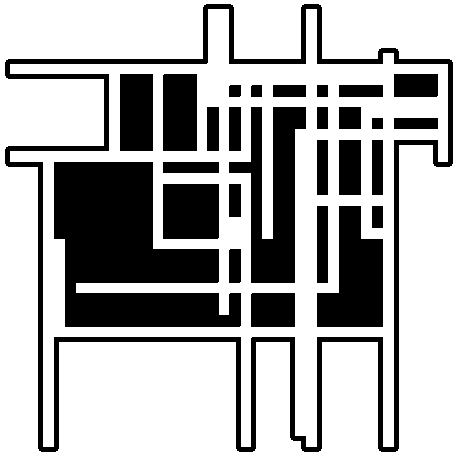
(a) Digger map generated with the default settings.



(b) Digger map generated with *RoomPercentage* = 20%.



(c) Digger map generated with *ForwardProbability* = 60%, *RightProbability* = 19% and $LeftwardProbability = 19\%$.



(d) Digger map generated with *ForwardProbability* = 96%, *RightProbability* = 1% and $LeftwardProbability = 1\%$.

Figure 3.6: Four maps generated by the Digger Generator using "*AFairlyRandomSeed*" as seed, but different settings.

By default, the Digger Generator has *ForwardProbability* set to 90%, *LeftProbability* set to 4%, *RightProbability* set to 4%, *VisitedProbability* set to 2%, *StairProbability* set to 0% and *RoomPercentage* set to 50%.

3.3.4 The Stairs Generator

The *Stairs Generator* places stairs in the map after having analyzed it to find possible positions, but if stairs have already been placed by the *Map Generator* (this happens with the *Digger Generator*), it just validates them.

3.3.5 The Map Assembler

The *Map Assembler* controls the assembly of the map. Each different implementation of the Map Assembler corresponds to a different *map appearance*.

Mesh Assembler

The *Mesh Assembler* produces a 3D model of the map using an implementation of the *marching squares algorithm*[5] to generate three meshes: one for the floor, one for the walls and one for the ceiling. As it can be seen in figure 3.8a, the result is a natural-looking environment.

Prefab Assembler

The *Prefab Assembler* produces a 3D model of the map by associating to each tile a specific 3D model, or *prefab*, depending on the value of the tile and of its 8-neighbors (see figure 3.7). Figure 3.8c shows a map assembled with this algorithm.

Multi-Level Prefab Assembler

Like the *Prefab Assembler*, the *Multi-Level Prefab Assembler* produces a 3D model of the map by combining prefabs, but it employs additional logic to menage the overlap of multiple floors. Figure 3.8d shows a map assembled with this algorithm.

3.3.6 The Spawn Point Manager

The *Spawn Point Manager* contains a list of all the spawn points displaced in the map, that is populated at the end of the *Generation phase* by the *Game Manager*. When the *Game Manager* needs to spawn an entity, the *Spawn Point Manager* provides a random spawn point from the ones that have not been used in a certain amount of time. If no spawn point meets this condition, the extraction is performed from the complete pool.

[5] *Marching squares* is a computer graphics algorithm that generates contours for a *two-dimensional scalar field*, i.e. a rectangular array of individual numerical values.

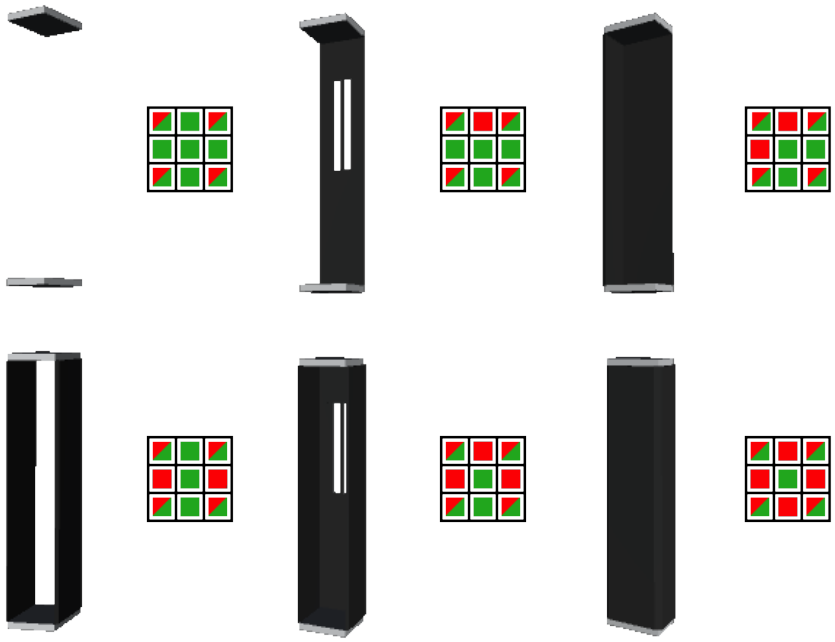
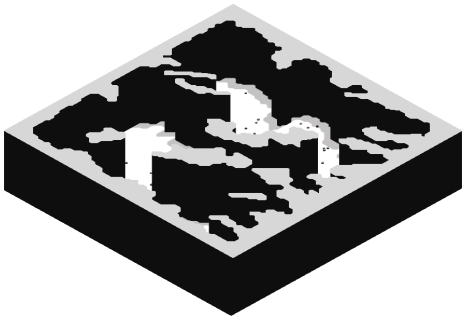
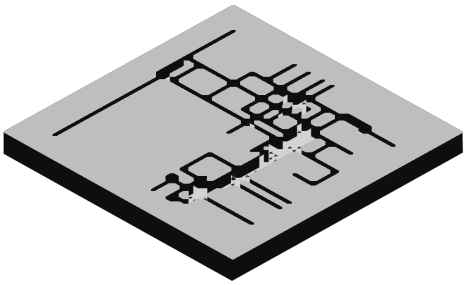


Figure 3.7: Some prefabs and the masks they are associated to.

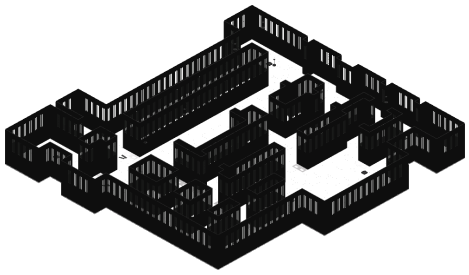
Each model refers to the central cell of the corresponding mask. Green denotes empty cells, red denotes filled cell, half-green and half-red denotes cells that are ignored by the mask. The masks can be rotated to obtain all the possible configurations.



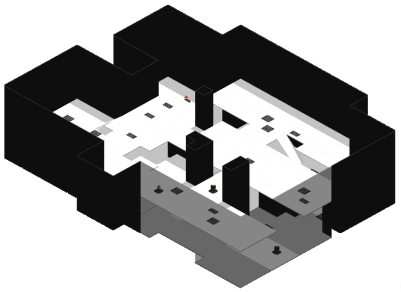
(a) Cellular map assembled with the *Mesh Assembler*



(b) Digger map assembled with the *Mesh Assembler*



(c) Divisive map assembled with the *Prefab Assembler*



(d) Multi-level map assembled with the *Prefab Assembler*

Figure 3.8: Some possible combinations of generators and assemblers.

3.3.7 The Object Displacer

The *Object Displacer* associates a character that represents neither a wall or a clear cell to the corresponding object, displacing it at the coordinates defined by its position in the map matrix. During this process, it populates a dictionary containing all the objects in the map divided by category, that is used by the *Game Manager* to populate the list of spawn points used by the *Spawn Point Manager*.

3.3.8 The Experiment Manager

The *Experiment Manager* is a stand alone module that allows to create and menage the experiments used to perform user-based validation. Once that an experiment has been defined, the *Experiment Manager* automatically assigns to the users the matches to play and collects the desired information.

Experiment definition

An experiment is defined by a *tutorial*, a list of *studies* and a *survey*.

The *tutorial* is optional and consists in a match with a simple objective used to explain the commands to the user.

The *studies* are not optional and each one of them consists in a list of *cases*. Each case contains a pool of maps and a single game mode, that is used to play the maps in the pool. The maps in the pool are the object of validation, whereas the game mode is the employed validation method.

The *survey* is optional and consists in a list of multiple-choice questions that are presented to the player at the end of the experiment.

All these elements can be easily customized, as well as the number of test cases that an user has to play in a single experiment session, defined by the parametric variable *CasesPerUsers*. The *Experiment Manager* also allows to diagonally flip the maps, which is an useful method to avoid the rise of a bias due to memorization when the player is presented with different versions of the same map.

Experiment management

Once that the experiment has been defined, it is ready to be played by the users.

Each time that a user participates in the experiment, the *Experiment Manager* selects the least played case of the least played study, in a round-

robin fashion. This allows to have equally distributed data for each study and is possible thanks to the completion tracking provided by the *Experiment Manager* itself. Then, for each case that the user is going to play, a pre-generated map is extracted from the pool and presented to the player as a match of the game mode specified by the case. In a complete experiment, the player will consecutively play the tutorial, one ore more matches and finally answer the survey.

The experiment can be performed *offline* or *online*. In the former, the computed data and the experiment completion are stored locally, whereas in the latter, they are stored on a server. If the experiment is provided via an executable, then it is possible to configure it as *offline*, *online* or both, with the completion that is stored on a server and the computed data that is stored both locally and remotely. If the experiment is provided via a web build playable via browser, the only supported configuration is the *online* one.

Logging

By default, the *Experiment Manager* produces a complete log of each match, saving the following information:

- *MapInfo*: this field contains general information about the map featured in the match, as its name, its dimension, the size of its tiles and if it has been flipped.
- *GameInfo*: this field contains general information about the match, as the experiment name, the game mode and the duration.
- *SpawnLogs*: this field contains a list of all the spawn events. Each entry contains a timestamp, the coordinates of the spawn point and the name of the spawned entity.
- *PositionLogs*: this field contains a discretized list of the positions occupied by the player during the match, acquired with a given frequency. Each entry contains a timestamp, the coordinates of the player and the direction he is facing expressed in degrees.
- *ShotLogs*: this field contains a list of all the shots fired by the player. Each entry contains the same fields of the *PositionLogs*, plus the identifier of the firing weapon, the number of projectiles in its magazine and its total available ammunition.

- *ReloadLogs*: this field contains a list of all the reloadings performed by the player. Each entry contains a timestamp, the identifier of the weapon that is being reloaded, the number of projectiles in its magazine and its total available ammunition, both before the reloading.

- *HitLogs*: this field contains a list of all the shots that hitted an entity. Each entry contains a timestamp, the coordinates of the hitted entity, the name of the hitted entity, the name of the hitter entity and the caused damage.

- *KillLogs*: this field contains a list of all the killings. Each entry contains a timestamp, the coordinates of the killed entity, the name of the killed entity and the name of the killer entity.

It is possible to customize the *Experiment Manager* to have it compute and save specific metrics in a different log. Moreover, if the experiment includes a survey, the answers of the user are saved in a dedicated log.

Data retrieval

The framework provides a simple interface for downloading the logs stored on the server. Since it is possible to set a limit on the dimension of logs which causes them to be split in multiple parts, the framework automatically performs merging and signals incomplete logs.

3.4 Entities

The *entities* are the *agents* that take part in a match. All the entities share some common features, but are further characterized by their specific implementation; these features are:

- *TotalHealth*: the maximum number of health points of the entity, i.e. the quantity of damage the entity can receive before being destroyed.

- *Guns*: the guns associated to the entity.

The framework includes three different kind of agents:

- *Player*: the entity that is controlled by the user. It can walk, jump, aim, deal and receive damage and pick resources.

Figure 3.9: Different ready-to-use target entities provided by the framework.

From the first to the third are simple targets, from the fourth to the sixth are targets equipped with two opposing laser guns, from the seventh to the ninth are"*core*" targets equipped with an increasing number of radial laser guns. The size of each target is proportional to its *TotalHealth*.

- *Opponent*: this implementation is similar to the one of the *Player*, but in the current version of the framework it has no active logic, beside the one that controls its health.
- *Target*: this simple entity rotates in place. Besides receiving damage, it can harm the player thanks to the laser gun it can be equipped it. Figure 3.11 shows different kind of targets.

3.5 Weapons

The framework allows to easily define any kind of fire arm with the extension of a common parametric structure, that characterizes the basic behavior of a gun with the following variables:

- *Damage*: the damage inflicted by a single projectile.
- *Dispersion*: the aperture of the cone-shaped projectile spread expressed in degree.
- *ProjectilePerShot*: the number of projectile emitted with one shot.
- *InfiniteAmmo*: tells if the gun has infinite ammunition.
- *ChargerSize*: the capacity of the gun magazine.
- *MaximumAmmo*: the maximum quantity of ammunition that can be carried for a specific gun.
- *ReloadTime*: the amount of time needed to reload the gun.
- *CooldownTime*: the amount of time needed after a shot to fire again.

- *AimEnabled*: tells if the gun allows the player to aim.
- *Zoom*: the zoom provided by the scope when aiming.

This parametric approach allows to use the framework as a tool for user-based validation of procedurally generated weapons, that is another research field that has been explored in recent years [**?**].

The framework comes with three different class of weapons already implemented.

Raycast guns

Raycast guns are weapons which projectiles have no *time of flight*, but instantly hit the target once shot. The only additional parameter that this class introduces is *Range*, which can be used to limit the reach of the weapon.

There are three weapons of this class that the player can use:

- *Assault Rifle*: a medium range weapon with a high fire rate, a capacious magazine and no dispersion which shots single medium damage projectiles. Figure 3.10a shows its model and table 3.1 shows its parametric configuration.
- *Shotgun*: a short range weapon with a slow fire rate, a small magazine and high dispersion which shots multiple low damage projectiles. Figure 3.10b shows its model and table 3.1 shows its parametric configuration.
- *Sniper Rifle*: a long range weapon with a slow fire rate, a small magazine and no dispersion which shots single high damage projectiles. It is equipped with a scope. Figure 3.10d shows its model and table 3.1 shows its parametric configuration.

Projectile guns

Projectile guns are weapons that shoot projectiles with a limited flight speed. This class introduces two additional parameters:

- *ProjectileLifetime*: if the projectile does not hit anything after this amount of time, it is destroyed.
- *ProjectileSpeed*: the speed of the projectile.

The only weapon of this class that the player can use is the *Rocket Launcher*, a long range weapon with a slow fire rate, a small magazine and no dispersion which shots explosive projectiles. The projectiles of this weapon are slow and explode on impact, dealing an high damage that decreases radially from the center of the explosion. Figure 3.10c shows its model and table 3.1 shows its parametric configuration.

Laser guns

Laser guns are a separate class of weapons, since they are not based on the structure extended by the previous two. Laser guns emit a continuous ray that deals damage over time to everything it touches. Their only configurable parameter is *DPS* (*damage per second*), i.e. the damage the the gun deals in a second when continuously hitting a target.

3.6 Objects

Beyond *decorations*, that are simple 3D models with no logic used to graphically enrich the map, the framework provides *spawners*, i.e. objects that spawn a resource that can be collected by the entities. Once that the resource is collected, it disappears for an interval of time defined by the parametric variable *Cooldown*. The framework comes with two different *spawners*:

- *Health pack Spawner*: it spawns health packs, that partially restore the health of the entity. The healed amount of health is defined by *RestoredHealth*.

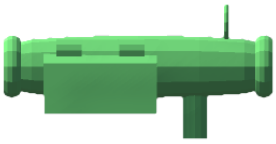
- *Ammunition Spawner*: it spawns ammunition crates, that supply the entity with ammunition. *SuppliedGuns* defines which guns the crates can supply, whereas *AmmoAmounts* defines how many ammunition are provided to the entity for each supplied weapon.

3.7 Game modes

The framework comes with three different game modes, that have been designed to highlight specific aspects of a multiplayer FPS. Each game mode is defined by a different implementation of the *Game Manager*.

(a) The Assault Rifle.

(b) The Shotgun.



(c) The Rocket Launcher.

(d) The Sniper Rifle.

Figure 3.10: The weapons that the player can use.

	Assault Rifle	Shotgun	Rocket Launcher	Sniper Rifle
Damage	15	20	120	75
Dispersion	0	7.5	0	0
ProjectilesPerShot	1	5	1	1
InfiniteAmmo	false	false	false	false
ChargerSize	32	3	2	5
MaximumAmmo	120	24	16	30
ReloadTime	1	1	1	1
CooldownTime	0.1	0.75	0.75	0.5
AimEnabled	false	false	false	true
Zoom	1	1	1	3
LimitRange	false	true	-	false
Range	-	100	-	-
ProjectileLifeTime	-	-	10	-
ProjectileSpeed	-	-	50	-

Table 3.1: Parametric configuration of the four weapons available to the player.

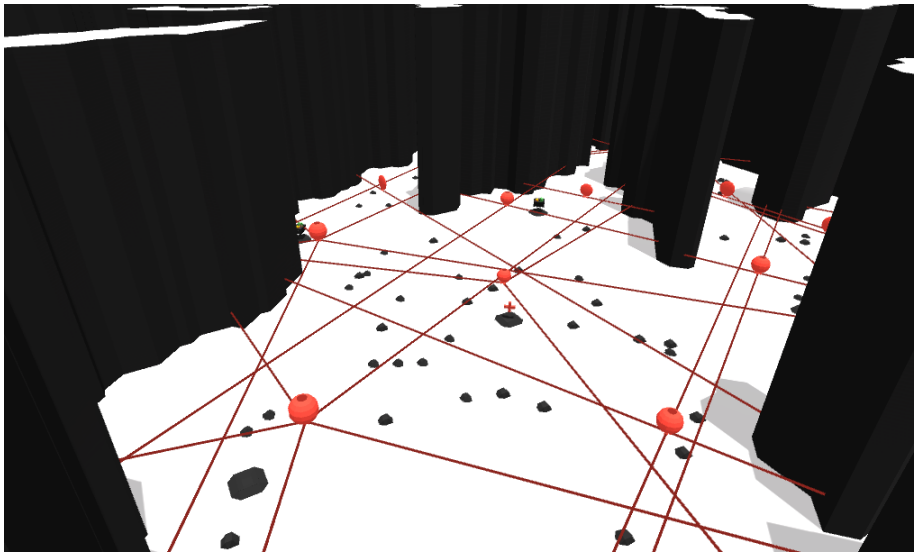


Figure 3.11: Targets equipped with laser guns in the final wave of a *Target Rush* match.

3.7.1 Duel

The *Duel* game mode is a classic *deathmatch* redistricted to two entities, with one of the two being the player. Each time that an entity eliminates the other one, it scores one point, whereas it loses one if it destroys itself by accident. When an entity has been eliminated, it *respawns*[6] at a random spawn point. At the end of the match, which is marked by a time limit, the winner is the contender who has scored the highest number of points.

Of the game modes that the framework provides, this one is the most complete, because it contains all the dynamics that characterize a multiplayer FPS match, and the most important, since it is the one that is usually used to perform validation in this research field.

3.7.2 Target Rush

In the *Target Rush* game mode the player faces increasingly difficult waves of enemies, trying to obtain a score as high as possible before the end of the game, that is triggered by the player death or by the countdown hitting zero. The player earns points and additional time when he destroys an

[6]In video games, *respawn* denotes the reappearing in a specific location, called *spawn point*, of an entity which has been eliminated.

enemy or he completes a wave. The number of waves is parametric, as well as the content of each one. By default, this mode has twenty waves and uses targets as enemies, that start as harmless but became more and more numerous and dangerous with each wave (see figure 3.11).

This game mode has been designed to force the player to explore the map, through the research of enemies, health packs and ammunition, that quickly become indispensable as the match progresses.

3.7.3 Target Hunt

In the *Target Hunt* game mode the player needs to find and eliminate a series of enemies in a given amount of time. The enemies that the player is going to face are stored in a parametric list that is read circularly and are spawned one at a time, as soon as the previous one has been eliminated. To each enemy is assigned a score.

This game mode has been designed to force the player to search a specific objective in the map.

3.8 Summary

In this chapter we analyzed the framework that we have developed to perform user-based validation, focusing on its structure, its components and its parametric nature.

Chapter 4

The graph-based tool

In this chapter we describe the tool that we have developed to perform analysis and populating of pre-generated maps using *Graph Theory*. After a quick overview, we introduce the analysis capabilities of this tool and then we present how we have employed them, together with Graph Theory, to strategically place resources in pre-generated maps.

4.1 Description of the tool

This tool generates different kind of graphs, starting from the text and the All-Black representation of a map, that are used to perform various analysis and manipulation operations.

The use of All-Black format is convenient, because it provides by default a logical division of the map in different areas and it allows our tool to be applied by other researchers, since as we have seen the All-Black format is widely used in this field. We have used this tool to position resources in a pre-generated map, but, for instance, it could be used to address the identification and definition of design patterns from an unfamiliar perspective or for *direct evaluation* in Search Based PCG.

We developed this tool in *Python* because if offers many solid Graph Theory libraries, as *NetworkX*, that is the one we have used.

4.2 Analysis of pre-generated maps

The analysis is performed by generating different kind of graphs, each one used to highlight a different feature of the map in question.

4.2.1 Outlines graph

The *outlines graph* is generated starting from the All-Black representation of a map and is obtained by associating a node to every vertex of every room and corridor and by connecting the non-adjacent ones that belong of the same outline. This graph has a single kind of node (*vertex node*), which contains the coordinates of the tile it represents, which are used to position the node when the graph is visualized. Figure 4.1b shows an example of this graph.

This graph can be used to visualize the structures which compose the map.

4.2.2 Reachability graphs

Our tool can generate various kinds of *reachability graphs* that represent various ways in which an entity can navigate a map. In these graphs a *node* represents a position that an entity can reach, whereas an *edge* indicates a viable path from a position to another.

Tiles graph

The *tiles graph* is generated starting from the text representation of a map and is obtained by associating a node to each empty tile and by connecting each node to its corresponding 8-neighbors. The horizontal and vertical edges have cost 1, whereas the diagonal ones have cost $\sqrt{2}$. This graph has a single kind of node (*tile node*), which contains the coordinates of the tile it represents, which are used to position the node when the graph is visualized. Figure 4.1c shows an example of this graph.

This graph can be used to find the minimum distance that separates two cells, along with the shortest path that connects them.

Rooms graph

The *rooms graph* is generated starting from the All-Black representation of a map and is obtained by associating a node to each room and corridor and by connecting nodes which corresponding rooms or corridors overlap with each other, using as weight the Euclidean distance of their central tile. This graph has a single kind of node (*structure node*), used to represent both rooms and corridors, which contains the coordinates of the closest and furthest vertex of the structure from the origin. When visualized, each node is positioned

on the coordinates of the central tile of the structure it represents. Figure 4.1d shows an example of this graph.

This graph can be used to analyze the topology of a map, in order to find loops, choke points, central areas and other kind of structures.

Rooms and resources graph

The *rooms and resources graph* is an extension of the room graph, which also include resources as nodes, that are connected to the nodes corresponding to the rooms and corridors which contain them. In addition to the structure node inherited form the rooms graph, this graph has a node to represent resources (*resource node*), which contains the coordinates of the resource, which are used to visualize the node, and the character associated to the resource. Figure 4.1e shows an example of this graph.

4.2.3 Visibility graph

The *visibility graph* is generated starting from the text representation of a map and is obtained by associating a node to each empty tile and by connecting each node to all the tiles that are visible from that node. For two tiles to be respectively visible, it must be possible to connect them with a line without crossing any filled tile. This graph has a single kind of node (*visibility node*), which contains the coordinates of the tile it represents, which are used to position the node when the graph is visualized, and its *degree centrality*, i.e. the number of edges incident to that node.

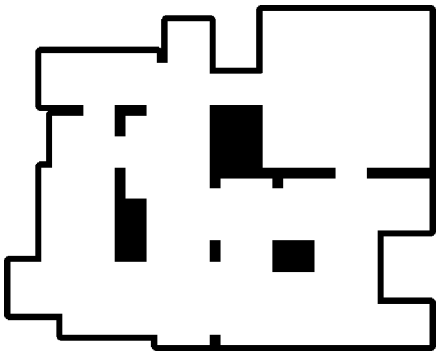
To make this graph easier to read by the user, the tool associates a color to the nodes, which ranges from blue, for the one with the minimum visibility, to red, for the one with the maximum visibility. This can be seen in figure 4.1f.

This graph can be used to analyze which areas of the map are more exposed and which ones are more repaired.

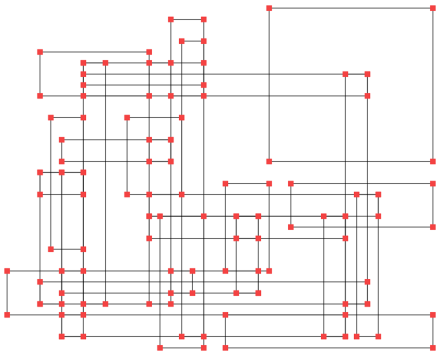
4.2.4 Interesting metrics

Considering the graphs, in particular the ones with rooms and corridors as nodes, the following metrics defined by Graph Theory provide interesting information about the layout of a map:

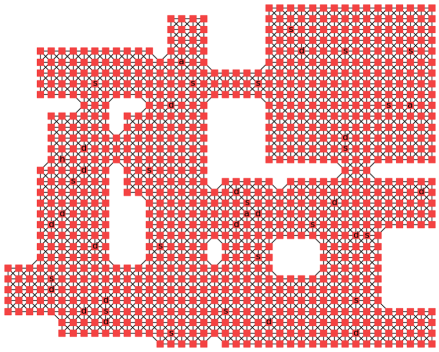
- *Degree centrality*: defined for a node, it is the number of edges that the node has. If the node represents a room, it measures how many entrance or exits the room has.



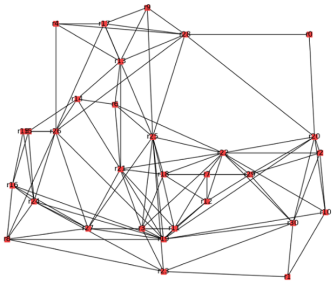
(a) The map.



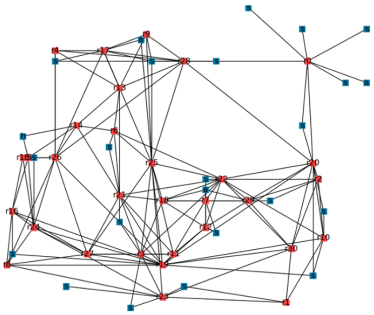
(b) The outlines graph of the map.



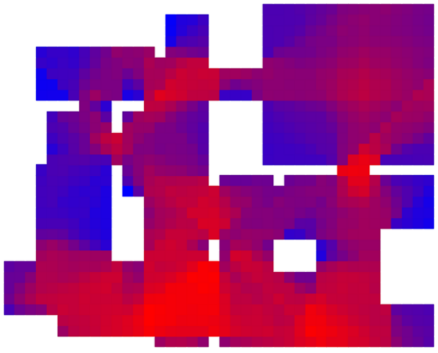
(c) The tiles graph of the map.



(d) The rooms graph of the map.



(e) The rooms and resources graph of the map.



(f) The visibility graph of the map.

Figure 4.1: A map and all the graphs that the tool can generate from it.

- *Normalized degree centrality*: defined for a node, it is obtained by normalizing the degree centrality, associating 0 to the node with the minimum centrality and 1 to the node with the maximum centrality.

- *Closeness centrality*: defined for a node, it measures its centrality in the graph, computed as the sum of the lengths of the shortest paths between the node and all other nodes in the graph. If the node represents a room, it measures how central the room is.

- *Betweenness centrality*: defined for a node, it measures its centrality in the graph, computed as the number of shortest paths connecting the nodes in the graph that pass through the node. If the node represents a room, it measures how central the room is.

- *Connectivity*: defined for a graph, it is the minimum number of elements (nodes or edges) that need to be removed to disconnect the remaining nodes from each other. If the graph represents a map, it measures the existence of isolated areas.

- *Eccentricity*: defined for a node, it is the maximum distance from the node to all other nodes in the graph. If the node represents a room, it measured how isolated the room is.

- *Diameter*: defined for a graph, it is the maximum eccentricity of its nodes. If the graph represents a map, it measures the size of the map.

- *Radius*: defined for a graph, it is the minimum eccentricity of its nodes. If the graph represents a map, it measures how distanced the rooms are from each other.

- *Periphery*: defined for a graph, it is the set of nodes with eccentricity equal to the diameter. If the graph represents a map, it defines its peripheral areas.

- *Center*: defined for a graph, it is the set of nodes with eccentricity equal to the radius. If the graph represents a map, it defines its central areas.

- *Density*: defined for a graph, it ranges from 0 to 1, going from a graph without edges to a complete graph. If the graph represents a map, it measures how complex it is.

4.3 Populating of pre-generated maps

We have defined multiple heuristics to populate a map with spawn points and resources using the metrics that can be extracted from a graph. These heuristic are a mathematical transposition of rules and patterns concerning resource placement that we have extracted from the work of Tim Schäfer[**?**], who has performed an in depth analysis of multiplayer 1vs1 maps for *Quake 2*[1].

4.3.1 FPS map analysis

The balance of a deathmatch game radically changes each time that a player is killed. If the game has more than two players, the player who won the fight does not gain any strategic advantage, since he still has the other players to face, whereas the defeated player is put at considerable disadvantage, because on death he loses all the weapons and ammunition that he collected. In a 1vs1 match, a kill has an even stronger influence, since the surviving player has more weapons and ammunition and gains the complete control of the map, that comes with the chance of scoring another easy kill, as soon as the other player respawns, or of searching for additional equipment. Schäfer refers to the surviving player as **up-player** and to the defeated one as **down-player**.

To obtain a multiplayer map that is interesting and fun to play it is important to consider the up-player vs down-player dynamic both when defining the map layout and when positioning resources.

The spawn points, i.e. the locations where the down-player reappears, should be positioned in areas that are of low interest for the up-player and that are easy to leave. Obviously, central hubs and dead ends are a bad choice, whereas rooms with 2 or 3 exits are usually the best option.

For what concerns the resources, they must be placed considering both the up-player vs down-player dynamic and the characteristics of the resource itself. It is important to place the right amount of resources on the map, because too many would eliminate the need for exploration, whereas too few would further disadvantage the down player. It is also important not to place too many powerful items in the same area or in boring spots, since the risk to obtain them should always be proportional to the strategical advantage they allow to achieve. It is important to consider that a powerful resource is interesting for both players, so it often acts as a *point of collision*. The resources usually are of five kinds: health pack, armor, power-up,

[1] Id Software, 1997

ammunition and weapon.

The health packs are placed in zones that are safe or not too dangerous. They have no use for the down-player, that respawns with full health, but they can be useful for the up-player, if he has been damaged during the fight, whereas they always come in handy during a fight or when one of the contenders disengages.

Armor, which is a second health that is consumed before the main one, is usually placed in spots that are aimed both at the down-player and at the up-player: objects that provide a small quantity of armor should be easy to achieve, whereas the ones that provide full armor should be placed in dangerous areas.

Power-ups grant temporary advantages to the player who collects them, like invisibility or increased damage, and are placed in locations difficult to reach.

The position of a weapon and of its ammunition depends on the weapon itself. We can divide the weapons in three categories: weak, medium and strong. Weak weapons are of a certain interest for the down-player, if he has not collected any other weapon yet, and of no interest for the up-player, so they are placed near spawn-points or in gaps where no other weapon is available, together with their ammunition. Medium weapons are of high interest for the down-player, since he needs to get one of them as soon as possible if he wants to face the up-player, so they are placed in areas that are easy to reach and the same goes for their ammunition. Finally, the strong weapons should be placed in areas that are strategically disadvantageous, like dead ends or vertically dominated areas, or difficult to reach. If a weapon is very contextual, i.e. it is useful in very few situations, it is usually placed in an area that allows to take advantage of its features, whereas a weapon that is strong in almost any situation is usually placed in an area where it cannot be used optimally (e.g. a rocket launcher in a small room).

4.3.2 Spawn points placement

We have defined four heuristics for the placement of spawn points.

Low risk heuristic

We defined an heuristic that places spawn points in the rooms that have the minimum number of entrances but are not dead ends and are distant enough from each other. Inside the room, the spawn point is placed on the tile that has the lowest visibility but is not adjacent to the wall. In this

way spawn points are placed in passageways that are sheltered and easy to leave. At each iteration, we select a structure from the *rooms and resources graph* using the following heuristic, given G the rooms and corridors graph, $S \subset G$ the subset of structure nodes, $R \subset G$ the subset of resource nodes and n the number of spawn points that must be added to the map:

$$room = \arg\min_{s \in S} \left\{ w_1 \times \left\{ \begin{array}{ll} 1 & \text{if } \deg(s) = 1 \\ \dfrac{\deg(s) - \min_{s' \in S} \deg(s')}{\max_{s' \in S} \deg(s') - \min_{s' \in S} \deg(s')} & \text{if } \deg(s) \neq 1 \end{array} \right\} \right.$$
$$+ w_2 \times \min_{n \in G} \left\{ \begin{array}{ll} 1 & \text{if } n \in S \\ \dfrac{\text{shortest path length}(r, n)}{\text{diameter}(G)} & \text{if } n \in R \end{array} \right\}$$
$$\left. + w_3 \times \frac{|r \in R \,/\, r \in \text{neighborhood}(s)|}{n} \right\}$$

The first component of the equation, which is weighted by w_1, penalizes rooms depending on the number of passages they have, the second component, which is weighted by w_1, penalizes rooms close to a spawn point, and the third component, which is weighted by w_3, penalizes rooms that already contain one or more spawn points. All the three components are normalized in the interval $[0, 1]$ so that their influence on the final result is proportional to their weight. We experimentally found that the values that provide the best results are:

$w_1 = 1, w_2 = 0.5, w_3 = -2$

High risk heuristic

Uniform heuristic

Random heuristic

4.3.3 Ammunition placement

4.3.4 Health packs placement

4.4 Summary