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

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 map. 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 usato il nostro framework per validare questo strumento tramite una raccolta di dati online.

Contents

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 to 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 requites 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 lead to greater innovation in the field and to the ones that are used to perform academic research in this field.

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 budget, this was because there were no tools such *level editors*[1], that allowed the level designer to work on a level without being involved with code.

---

[1] A level editor is a software used to design levels, maps and virtual worlds for a video game. An individual involved with the creation of game levels is a level designer.

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 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 game play, 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 the 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 cover, resources or high ground, the more players will be comfortable moving in it. This doesn't mean that all

---

[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.

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 benefits, making the game play 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 pre-defined *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

---

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

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 example is the open world action role-playing game *The Elder Scrolls II: Daggerfall*[10], which game world has the same size as 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, as *SpeedTree*[11] and *World Machine*[12], are used to produce 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

---

[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.

The Binding of Isaac[19].

All the algorithms used by these games and middleware are design 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

---

[19]Edmund McMillen, 2011.

[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

9

and the death of one of the two contenders. The choice of this fitness function is based on the consideration that a long fight is correlated to 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 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 simulation.

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 satisfy what they define as *The Good Engagement* (or *TGE*) rules, 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 ap-

---

generated content and the interaction with a real player.

[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.

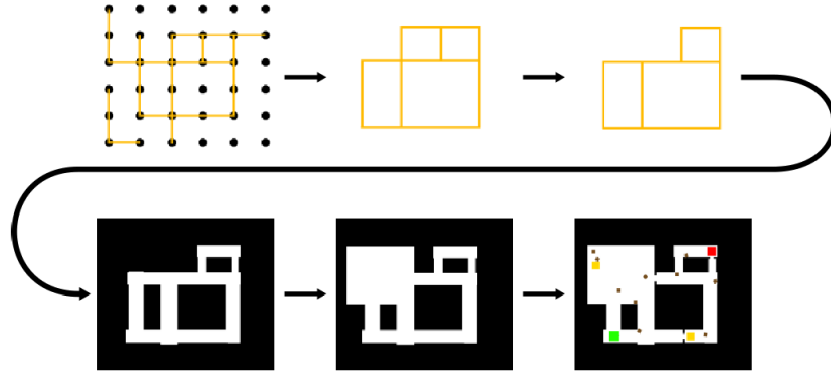[28]Valve Software, 2000

[29]Infinity Ward, 2003

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

preciation feedback expressed by real users, since the authors consider bot behavior too different form the one of real users.

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 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

---

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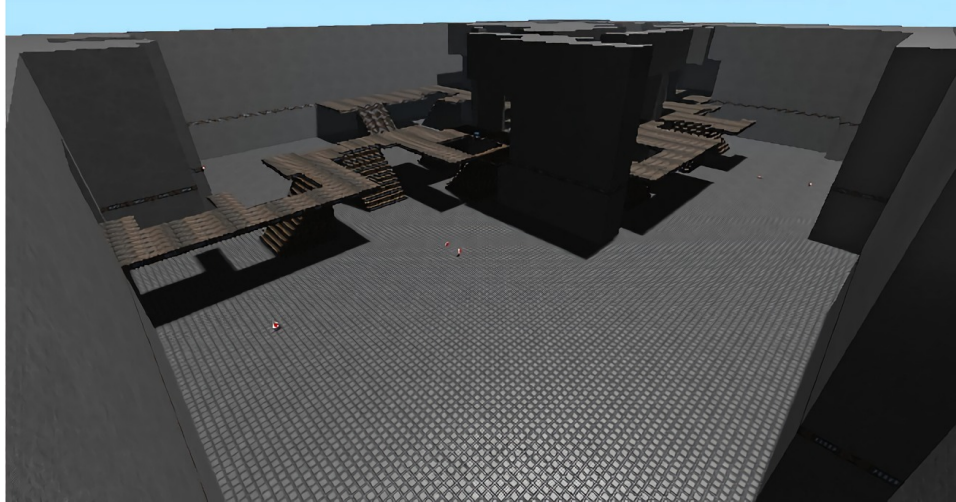
Figure 2.2: One of the maps evolved by Cachia's et al.[?] algorithm.

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 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 radical 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 deathmatch 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 didn't 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 to 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 weren't suitable for the extremely fast paced action of the past. Over time, this new gameplay overshadowed almost completely all the other approaches , 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.
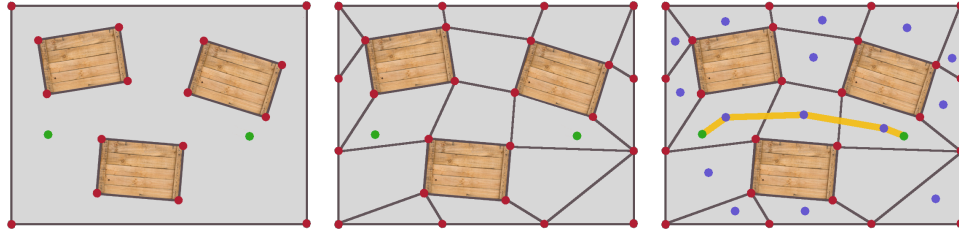[47]Epic Games, 1999.
[48]Bungie, 2001.

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 often 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 shooter games 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 FPS maps. 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

Features of the research framework

In this chapter we describe the *framework* that we have developed to perform user-based online validation for researches in procedural content generation of multiplayer levels for Firsts Person Shooters. In the first section we give an overview of the framework, of its features and of its components, analyzing them one by one in the following sections.

3.1 Overview

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 framework that is as light as possible, with a WebGL build weighting less than 10MB that can be played using any browser. The framework was developed with Unity.

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

3.2   The 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 in 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.

3.2.1   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.

3.2.2 The Spawn Point Manager

The *Spawn Point Manager* contains a list of all the spawn points displaced on the map, that is populated during 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 made from the complete pool.

3.2.3 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¿[1], on the *Object Displacer* for the *displacement*[2], whereas it performs the import itself. If the map is provided as input, the *Map Generator* is not called.

Maps are represented as matrices of characters, where each character corresponds to a tile. Depending on the character, a tile can represent a wall, a room or an object inside a room. If a tile corresponds to a wall we say it is *filled*, if it corresponds to a room we say it is *empty*.

3.2.4 The Map Generator

The *Map Generator* controls the generation of the map. Each implementation of the *Map Generator* defines a different topology depending on the used generation algorithm and on how its parametric setting are tuned. Some of these setting 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 width of the matrix that represents the map.

- *Height*, the height of the matrix that represents the map.

---

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

[2] 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.

- *ObjectToObjectDistance*, the minimum number of cells that must separate two objects.

- *ObjectToWallDistance*, the minimum number of cells that must separate an objects 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 can not 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*.

We now analyze the available implementations of the *Map Generator*.

Cellular generation

The *Cellular Generator* employs a parametric *cellular automaton*[3] to generate a natural looking map. The algorithm (see Algorithm 1) depends on the following parameters:

- *RandomFillPercent*, the percentage of tiles that are randomly filled during the initialization of the algorithm.

- *SmoothingInteration*, the number of generations the cellular automaton is ran for.

- *NeighbourTileLimitLow*, the minimum number of neighbors a cell must have to became filled.

- *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.

- *RoomThresholdSize*, the minimum number of cells that an isolated void region must include to not be deleted.

- *PassageWidth*, the width of a passage connecting two different areas, expressed in number of cells.

---

[3]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 cells 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 itself and of the cells in its neighborhood. Our implementation is

Algorithm 1: Cellular generation algorithm

1 **for** *every cell in the map* **do**
2 | empty the current cell;
3 **end**
4 **while** *the percentage of filled cells is less than RandomFillPercent*
 do
5 | select a random cell;
6 | fill the selected cell;
7 **end**
8 **for** *#SmoothingInteration iterations* **do**
9 | **for** *every cell in the map* **do**
10 | | count the 8-neighbors of the cell;
11 | | **if** *the cell has more than NeighbourTileLimitLow neighbors*
 then
12 | | | fill the current cell;
13 | | **end**
14 | | **if** *the cell has less than NeighbourTileLimitHigh neighbors*
 then
15 | | | empty the current cell;
16 | | **end**
17 | **end**
18 **end**
19 **for** *every connected region of empty cells* **do**
20 | **if** *the number of cells in the region is less than*
 RoomThresholdSize **then**
21 | | fill all the cells in the region;
22 | **end**
23 **end**
24 **for** *every connected region of filled cells* **do**
25 | **if** *the number of cells in the region is less than*
 WallThresholdSize **then**
26 | | empty all the cells in the region;
27 | **end**
28 **end**
29 connect all the regions composed by filled cells;
30 place the objects;

Divisive generation

Random digger generation

Multi-level generation

3.2.5   The Map Assembler

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

3.2.6   The Object Displacer

The *Object Displacer* associates a character that represents neither a wall or a clear cell to the corresponding object, displacing it in the correct position. 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   Map representation

3.4   Weapons and objects

3.5   Game modes

3.6   Logging

3.7   Experiment management