Modeling and Generation of Playable Scenarios Based on Societies of Agents
Bernardo Nunes Correia
1 2 9 0
UNIVERSIDADE D COIMBRA

1 2  9 0

UNIVERSIDADE D
COIMBRA

Bernardo Nunes Correia

Modeling and Generation of Playable Scenarios Based on Societies of Agents

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems advised by Professor Licínio Roque and presented to Faculty of Sciences and Technology / Department of Informatics Engineering.

September 2021

Faculty of Sciences and Technology

Department of Informatics Engineering

Modeling and Generation of Playable Scenarios Based on Societies of Agents

Bernardo Nunes Correia

Dissertation in the context of the Master in Informatics Engineering, Specialization in Intelligent Systems advised by Prof. Licínio Roque and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering..

September 2021

UNIVERSIDADE D
COIMBRA

This page is intentionally left blank.

Abstract

Procedural content generation in video games consists of algorithmic approaches to generate game content autonomously or semi-autonomously. It has been used for several years as a way to diminish the authorial burden of artists and designers, to assist them in the creation of the content, to diminish the amount of content needed to be stored in memory, and to enable the possibility of games that do not need to end. Several methods have been developed, each one with its advantages and disadvantages. Agent-based methods generate game artifacts with the help of, often very simplistic, AI agents that independently make decisions that affect the end result in some way. This work aims at creating a platform that utilizes complex adaptive systems of these agents to model game scenarios. Two proof of concept game scenarios were created. One of them used the famous Conway's Game of Life and the other an adaptation of the arcade game "Bomberman". A graphical user interface was developed in order to give users a way to view, interact with, and edit the simulation. A Procedural content generation-based co-creation tool was also developed to further aid the user. The tool uses the Wave Function Collapse algorithm to propagate the pattern style of a selected area to the rest of the simulation grid. The developed architecture is successful in giving the user the control needed to incentivize exploration of the developed game scenarios. Such a platform could be used as a base for the testing and exploration of PCG approaches.

Keywords

Procedural Content Generation, Game Content, Agent-based PCG Methods, Modeling of Game Scenarios, Complex Adaptive Systems, Wave Function Collapse.

This page is intentionally left blank.

Resumo

Geração procedimental de conteúdo em videojogos consiste em abordagens algorítmicas à geração de conteúdo de jogos autónoma ou semiautónoma. É utilizada à vários anos como uma forma de diminuir o conteúdo que artistas e designers têm de criar, para os assistir na criação de conteúdo, para diminuir a quantidade de conteúdo que é necessário guardar em memória, e para permitir a possibilidade de jogos que não necessitam de terminar. Vários métodos têm sido desenvolvidos, cada um com as suas vantagens e desvantagens. Métodos baseados em agentes geram conteúdo com a ajuda de, normalmente simples, agentes IA independentes que tomam decisões que afetam o resultado final de alguma maneira. Este trabalho procura criar uma plataforma que utilize sistemas complexos adaptativos de agentes para modelar cenários de jogo. Dois cenários de jogo foram utilizados como prova de conceito. Um deles utiliza o famoso "Game of Life" de Conway, e o segundo uma adaptação do jogo de arcada "Bomberman". Uma interface gráfica do utilizador foi desenvolvida de modo a dar aos utilizadores uma forma de ver, interagir com, e editar a simulação. Uma ferramenta de cocriação baseada em geração procedimental de conteúdo foi também desenvolvida para auxiliar o utilizador na edição da simulação. A ferramenta usa o algoritmo "Wave Function Collapse" para propagar os padrões de uma área selecionada para o resto da grelha de simulação. A arquitetura desenvolvida teve sucesso em dar aos seus utilizadores o controlo necessário para incentivar a exploração dos cenários de jogo desenvolvidos. Tal plataforma poderá ser usada como uma base para o teste e exploração de abordagens de geração procedimental de conteúdo.

Palavras-Chave

Geração Procedimental de Conteúdo, Conteúdo de Jogo, Agentes IA, Métodos de Geração Procedimental de Conteúdo Baseados em Agentes, Modelação de Cenários de Jogo, Sistemas Complexos Adaptativos, Wave Funtion Collapse.

This page is intentionally left blank.

Contents

This page is intentionally left blank.

List of Figures

This page is intentionally left blank.

List of Tables

This page is intentionally left blank.

Chapter 1

Introduction

1.1 Context and Motivation

Procedural content generation has been used as early as the 1980s in video game production. Several methods for the autonomous generation of game content have been created through the years, as a way to diminish the load of content that artists and designers need to create, to assist them in the creation of the content, to diminish the amount of content needed to be stored in memory, and to enable the possibility of games that do not need to end. Some of these methods utilize AI agents as components in the generation process. These agents are usually quite simple and act autonomously from one another, the aim of these kinds of methods being the use of their stochastic nature as a way to generate variety in the content. The usage of more complex agents that interact and coexist with each other for such processes is unexplored. The main goal of this dissertation's project is the creation of a platform that utilizes complex adaptive systems of these agents to model game scenarios. The platform should allow the creation and editing of these scenarios by a user and allow for the testing and exploration of PCG approaches.

1.2 Goals and Contributions

Several important goals can be outlined for the proposed project:

- **Modeling of game scenarios as a complex adaptive system.** First is the modeling of the game scenario as a complex adaptive system, composed of several agent components, with their emergent behavior forming the game mechanics. This "Society of Agents" is implemented as a discrete bi-dimensional grid.
- **Modeling of the agents as components of the game system.** The agents must also be modeled - their behavioral logic, what kind of input they receive, how they interact with the environment they are a part of, and how they are visually represented to the user.
- **Support the training of Machine Learning agents.** The developed platform will be used by Diogo Alves [Alves, 2021], whose dissertation project pertains to the modeling of synthetic players as testing devices for generative game content. For such collaboration to be possible, the system had to be developed with this integration in mind.

- **Modeling of an editing system and GUI.** To make the platform usable by human designers that require it to test and explore PCG approaches, a way to edit the society of agents needs to be implemented.
- **Modeling of a co-creation editorial tool.** A co-creation, PCG-based tool to aid the user in the editing process by trying to replicate what already exists throughout the rest of the complex adaptive system. The algorithm chosen to be at the base of this tool was the Wave Function Collapse.

A platform that utilizes a simulation of an agent society to model game scenarios was created and evaluated by analyzing the behavior of five voluntary participants who were asked to operate it while following a list of tasks. All the participants showed interest in exploring the capabilities of the simulation beyond the requirements of the tasks. This is a good indication that the platform was successful enough in giving control of the simulation to the users to incentivize exploration of the developed game scenarios.

1.3 Methodology

A Design Science Research-based methodology was used in the development of the proposed project. The Design Science Research process is composed of five stages. It starts with the definition of the problem, alongside a list of the main goals to achieve and requirements for a design proposal. The second step of the process, the design proposal, will have as an artifact the first architecture of the system. What follows is the development phase, where a proof of concept based on the defined architecture is the resulting artifact. This proof of concept is assessed in the evaluation phase, where performance measures based on the goals that were defined in the first phase are used to generate information that will help to readjust the process if need be. The final phase concludes the process with the dissertation that will be written about the obtained results. As can be seen in figure x, this is not a linear process, each step of the process not only informs the next one but also the previous ones. Information gathered in the development and evaluation phases may change the design proposal in a new iteration, or even redefine the original problem that is being addressed. The process of development will inform the system's architecture as much as the second informs the first. It's "evolutionary prototyping", where we will learn more about the problem as we develop prototypes.

The two tools used in the development process of this project were the Unity game engine [Haas, 2014] and GitHub. Unity is used to develop and run video games, being one of the main leading game engines in game developing circles, either for 2D or 3D projects. It supports C#. GitHub was used as a repository for the project, allowing for version control.

1.4 Document Structure

What follows in this document is a gathering of information about most of the current methods used to generate game content, what contents are being generated, and ways to evaluate the output or the methods. After discussing the state of the art, the design proposal will be defined in the third chapter by detailing the objectives, requirements, methodology, and the platform's architecture. Chapter Prototype Implementation deals with the development process, starting from the initial prototype to the final edition of the platform, documenting the decisions made at each step. The results of some tests with

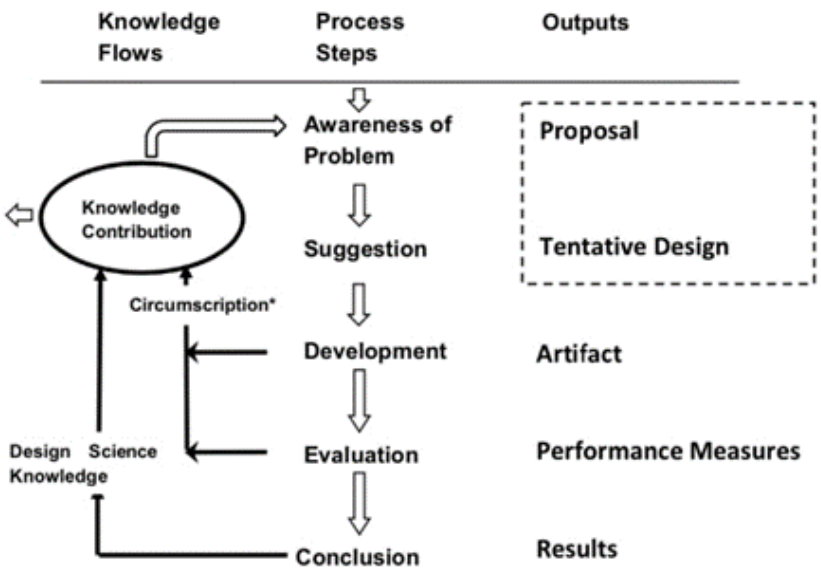


Figure 1.1: The Design Science Research process. Image obtained from Vaishnavi et al. [2019].

human users will be discussed in chapter Evaluations, with the final chapter reaching some conclusions about the contributions of the project and some future work.

This page is intentionally left blank.

Chapter 2

State of Art

This section of the document is devoted to clarifying what procedural content generation is in the field of video games, its influence through the years, and the reasons why it is used. Two recently proposed taxonomies of procedural content generation are explored, giving us two different perspectives on how to classify the possible approaches. Lastly, most of the current methods used to generate game content, what contents are being generated, and ways to evaluate the output or the methods are analyzed.

2.1 What is PCG in Games

Procedural content generation (more often abbreviated as PCG) can be understood as the automatic creation of game content through some algorithm, or computer procedure, with no or limited human input. It's rather difficult to reach a concrete definition of what PCG is and is not, since even when talking about what constitutes "game content" not everybody agrees [Togelius et al., 2011].

Game content refers to most of what is contained in a game: levels, maps, game rules, textures, stories, items, quests, music, weapons, vehicles, characters, etc. However, neither game engines nor non-player character (NPC) behavior are currently considered game content in most academic definitions of PCG, being outside of what's expected to be generated. The research in applying artificial intelligence (AI) methods to character behavior is much more vast than that in PCG. This narrowing of the definition of content is a way to set apart the field of PCG from the more "mainstream" where AI is used to define or learn how to play a game [Shaker et al., 2016] [Yannakakis and Togelius, 2018].

The generation of scenarios and artifacts is not an exclusive research field to video game content. However, what sets apart the generation of images, music, text, and other types of art from the particular case of PCG is that games need to be playable.as a Constraints and affordances of game design need to be taken into account [Shaker et al., 2016]. While generating "mostly correct" images is an acceptable outcome in other domains, if a PCG software generates levels that cannot be finished, it lacks any utility.

From an AI point of view, PCG problems can be seen as AI problems that have as solutions artifacts that respect certain constraints, such as being playable, and/or that optimize some set of metrics [Yannakakis and Togelius, 2018]. Many common AI methods such as evolutionary algorithms and neural networks are used in PCG as well as some more specialized ones, such as formal grammars and cellular automata.

2.2 A Little Bit of History

The limited storage of video game hardware in the early 1980s was one of the main driving forces for the creation of PCG techniques. Seeds were used in collaboration with deterministic methods as a way to always generate the same content. Storing a seed number and the generative method was a more efficient way of using the storage available than storing the whole map that would be generated. An early game that used such a technique was *Elite* [Braben and Bell, 1984], a space trading simulator by Acornsoft released in 1984, that would store the seed numbers used to procedurally generate 8 galaxies with 256 each [Shaker et al., 2016]. But even before Elite, there was 1980's *Rogue* [Wichman and Toy, 1980], a dungeon-crawling game that would generate new and unique levels every time a new game was started or the player lost. The uniqueness of Rouge would have such an impact that new genres of games inspired by its PCG mechanics would later be established, the so-called "roguelikes" and "roguelites".

Meanwhile, in the academia side of things, PCG has seen an explosive growth of interest within the second half of the 2000s that has kept momentum until nowadays [Yannakakis and Togelius, 2018]. Georgios N. Yannakakis and Julian Togelius, for example, are some of the most prolific researchers in the field, and they will be included in many of the references found throughout this text.

While in academia PCG is booming, in the industry it is currently not being explored to the same extent. While many AAA games still use PCG methods in some of their less critical aspects, they refrain from giving it center stage. PCG brings some uncertainty with it - uncertainty that big companies tend to shy away from. Of course, there are notable exceptions: the *Diablo* series [Blizzard North, 1997] of action role-playing hack-and-slash video games where maps, type, number, and placement of items and monsters are procedurally generated; *Spore* [Maxis, 2008], in which the creatures created by the player are animated with procedural animation techniques; Valve's *Left 4 Dead* [Valve Corporation, 2008] [Booth, 2009], which procedurally adapts the spawning of enemies in response to the player's behavior, allowing for a dynamic experience; and the *Civilization* series [Firaxis Games, 2016] of turn-based strategy games with randomly generated maps.

However, it is in the independent scene where we mostly see developers willing to put PCG at the forefront. *Minecraft* [Mojang, 2011], probably the most well-known example, is a 3D sandbox game with a focus on letting the player explore and modify a vast and diverse procedural generated world, with *Terraria* [Re-Logic, 2011] as its 2D counterpart with a bigger focus on combat and on allowing the player to customize their ability set with a vast assortment of items procedurally spread through the map. *Dwarf Fortress* [Adams, 2006], although not as "mainstream", is one of the most impressive uses of PCG, with entirely unique worlds of complex and interacting elements generated, where we can see the simulated lives of entire races coexisting through generations, as they live, die, modify the world and construct their fortresses. A tradeoff for all the focus in PCG that is apparent as soon as one lays eyes on *Dwarf Fortress* is its lack of visual appeal. The game has text-based graphics that for the ones unfamiliar with it can be quite hard to follow, although the strong community that has been built around the game has produced many custom tile-sets that permit a friendlier gameplay experience. In 2012 it was selected among other games to be featured in the NY Museum of Modern Art. *Dwarf Fortress* development started in 2002 and is still today continuously updated by Tarn and Zach Adams, with more interacting elements being added each time. In a similar vein as *Dwarf Fortress*, *Caves of Cud* [Freehold Games, 2015] uses PCG to generate a complex and dynamic world where the player can adventure in a way not unlike in pen-and-paper role-playing games,

such as Dungeons & Dragons. *Caves of Cud* belongs to the genre of games coined after the already mentioned *Rogue*, the "roguelikes", being that it presents "permadeath" - after dying the player cannot return to the same world, having to start from a newly generated one, with a new character. This type of game has become quite popular within the indies, utilizing its PCG-centric nature to give the player a seemingly infinite number of challenges.
Other (but not all) notable roguelikes through the years are: the award-winning 2D platformer *Spelunky* [Yu and Hull, 2008][Yu, 2016], that sees the player as an Indiana Jones-like explorer searching for treasure in an ever-varying cave system, and its recently released sequel, *Spelunky 2* [Yu and Mossmouth, 2020]; the indie darling from 2011, *The Binding of Isaac* [McMillen and Himsl, 2011] and its 2014 remake, *The Binding of Isaac: Rebirth* [McMillen and Nicalis, 2014]; the 2016's bullet hell gun-centric *Enter the Gungeon* [Doge Roll, 2016]; the turn-based *Darkest Dungeon* [Red Hook Studios, 2016], in which the player is tasked with managing the psychological state a team of adventurers through the procedurally generated dungeons; 2019's *Slay the Spire* [Mega Crit Games, 2019], where the player progresses through a spire full of procedurally generated encounters with enemies, fighting them through a collectible card-based system, with the player building a different deck of powerful moves through each attempt to reach the end; and lastly, one of the most popular games of 2020, Supergiant Games' *Hades* [Supergiant Games, 2020]. Supergiant Games, while having a small team of 12 core members, was able to build an experience capable of keeping players entertained for copious amounts of hours by using PCG techniques to generate a practically infinite number of variations of room sequences, enemy combinations, powerup choices given to the player, and unique encounters found through each attempt to escape greek hell. Probably the biggest innovation that sets *Hades* apart from all the roguelikes that came before is its approach to and focus on the story. Being that each player will have a different level of success, and the attempt that will finally set the main character free isn't known a priori, the story and interactions with the other mythological characters that inhabit the world of *Hades* adapts to each player's unique experience. For this effect, over 1 million unique lines of dialog were recorded and are selected to be shown to the player in function of what relevant events or combination of events happened in their latest attempt, or if a certain condition that enables the progression of the story arc of one particular character was met. No matter if the player reaches the end or not, they will always have some progression in the interactions with the other characters, allowing the story to evolve in a way that is unique for each player's experience.

2.3 Why use PCG

Both indie and AAA studios have problems with lengthy and costly development [Togelius et al., 2010][Barreto and Roque, 2014]. Humans are expensive and slow, and with the rising standards put on video games, every year, bigger and bigger teams of developers and artists are required to create more and more content with increasing costs that won't necessarily translate into value to the player. The number of person-months that go into the development of successful commercial games has been increasing more or less constantly [Shaker et al., 2016]. With the standard becoming teams of hundreds working for several years building mountains of content, game productions often risk becoming unprofitable and unaffordable to most developers, driving most of the smaller studios out of the AAA market. In turn, this can lead to a risk-averse industry, recycling of proven formulas with minor improvements, and less diversity in the game marketplace. Independent teams, with smaller budgets and number of members, are more capable to take riskier approaches, but competing with the bigger AAA teams with more technical and/or artistic expertise is no

easy task [Barreto and Roque, 2014]. The designer Will Wright in his talk "The Future of Content" at the 2015 Game Developers Conference, where he reveals the first prototypes of *Spore*[Maxis, 2008], argues that the best way to combat this impending cultural stagnation of video games is through procedural content generation [Shaker et al., 2016][Yannakakis and Togelius, 2018]. In the budget of most AAA games, art and content production constitute around 40% of a game's cost [Yannakakis and Togelius, 2018]. Rather than the programmers, artists and designers end up being the most expensive employees necessary in the game-making process. Will Wright argues that replacing some of their repetitive work with algorithms would give companies a competitive advantage and allow them to achieve the same quality faster and cheaper [Shaker et al., 2016][Yannakakis and Togelius, 2018]. Replacing all artists and designers is far from being a desirable goal, of course, since they are still needed to drive creativity and can do things that the current technology is far from achieving. Probably a more appealing advantage that PCG brings for artists and designers is its ability to empower their work process, instead of "stealing" it. Intelligent design tools (IDTs) embedded with content generation capabilities can augment their creative process on top of streamlining their workflow. Humans tend to imitate one another, including themselves. The input of an algorithm that follows the imposed constraints and yet generates some valid but unexpected solution can stimulate the designer's creativity [Shaker et al., 2016]. PCG can then provide a way to democratize game production by offering reliable and accessible ways to make better games in less time, being that anyone, from AAA studios, small teams, or even hobbyists, may become capable of building content-rich games, with smaller teams [Yannakakis and Togelius, 2018].
PCG also allows the creation of new types of games that would not be possible otherwise. Beyond having mechanics that employ PCG methods, the ability to generate content at the same rhythm that it is consumed opens the possibility for games with no end. Combining PCG with some type of player modeling also allows for the existence of player-adaptive games, where the newly generated content is tailored to the player and their unique tastes, enabling the game to adapt itself to the analyzed behavior. This would enhance the game experience, keeping the difficulty just right no matter the skill of who is playing, or maximizing the desired effects on the player, like joy, tension, or in the case of an educational game, its learning effects [Shaker et al., 2016]. Through the process of creating software that can generate some type of content, there exists an added bonus of our understanding of the design process we do "manually" becoming more clear. By better understanding the advantages and disadvantages of a particular process we can better use it and build better PCG methods, resulting in an iterative process of improvement [Shaker et al., 2016][Yannakakis and Togelius, 2018].

2.4 PCG Taxonomies

As there are diverse ways to look at PCG, there are also many methods and generation problems, and by having a structure to organize them we can better see how each of them differs and what they have in common. There are also many possible taxonomies, and analyzing how they differ can also teach us many things. Different taxonomies focus on different aspects of the field and give us perspectives from which we can gain a better understanding of where and how PCG is applied and what it can do. Two recently proposed taxonomies will be presented. The first one, of Togelius et al.[Togelius et al., 2010], defines various dimensions where each method or solution should be lying somewhere in between the two extremes. The second one, from Carveirinha et al.[Craveirinha et al., 2016], analyzes each approach considering three actors - the human designer, the computational algorithm, and the human player - and what role they fulfill in each approach. We will see

how each structure organizes the space of possibilities and what we can gain from them.

2.4.1 Togelius, Yannakakis, Stanley and Browne (2016)

Togelius et al. approach the challenge of organizing the PCG field by first defining three ways in which we can observe a content generation problem: the content that is being created, the properties of the PCG methods applied, and what role the PCG algorithm took. Then, they established 7 dimensions along which each problem can be placed. This taxonomy has seen several revisions over the years, the one described next can be found in [Shaker et al., 2016].

Two of the seven dimensions relate to the generated game content:

- Online vs. Offline
  - PCG techniques that generate content online do it as the player is playing the game, allowing for an endless stream of variation that could make a game infinitely replayable. The content being generated as it is being played can also allow for player-adaptive games, with variations to the content made on the fly. The already mentioned *Left 4 Dead* [Valve Corporation, 2008] [Booth, 2009] is a good example of this.
  - Offline generation of content is that which occurs during development - before shipping - or before each play session. This is usually done for more complex content that cannot be generated at the same rate that is played. Interestingly, some games provide content editors with PCG elements that allow the creation and sharing of custom content by the players, such as in *LittleBigPlanet* [Media Molecule, 2008] and *Spore* [Maxis, 2008]. These too would be considered offline exemples, since the content is being generated before being played.
- Necessary vs. Optional
  - Necessary content is that without which it would not be possible to play the game. Such content must always be correct since it will necessarily be presented to the player and they won't be able to avoid it. A required item or the structure of a level would be considered necessary content.
  - Optional content can be ignored or easily replaced by other content. It is seen as auxiliary. As such, it doesn't have the same quality level pressure as the necessary content, since the player can sidestep it.

Regarding the generation methods, they can be placed somewhere between the extremes of 3 dimensions:

- Non-Controllable vs. Controllable
  - There are many ways by which we can control a PCG method, some ways offer more control over what will be generated, to the designer, than others. In the original version of this taxonomy, this dimension was described as "Random seeds versus parameter vectors". Random seeds offer very little control, applying no additional restriction to the generation, only the guarantee that if we use the same seed, the same thing would be generated. A method truly non-controllable would require 0 input from the designer, except turning it on, of course.

  - Many methods give more control to the designer by requiring the input of a vector of parameters, allowing the control of the output along a number of dimensions of constraints.

- Stochastic vs. Deterministic

  - Stochastic PCG doesn't allow for the recreation of previously generated content. The closer a method is to this extreme, the more variation we get in outcomes between different runs of the algorithm with the same parameters. The diversity and expressivity come at the cost of control, and by its nature, the effect of randomness can only be seen after the fact.
  - Deterministic PCG stands at the other extreme of the spectrum. When we generate a new world in *Minecraft* [Mojang, 2011] using the same number seed we can expect that the same one will be generated every time. A totally deterministic algorithm can be seen as a way to compress data.

- Constructive vs. Generate-and-test

  - Constructive methods generate the content in one pass, correcting if necessary during the process of generation. We will see many examples of this type of method in Section 2.5 As we will also see, they are many times also incorporated into the generate-and-test framework.
  - Generate-and-test algorithms on the other hand have a second phase, where the generated content is tested and evaluated, and, if not acceptable, generated once again. A very popular framework of this kind that we will analyze in Section 2.5.1 is the search-based method.

In this taxonomy, the possible roles that PCG can take in the design process are organized in 2 dimensions:

- Autonomous vs. Mixed-Initiative

  - In automatic or autonomous generation the only input that human designers have is the initial parameters. Does not consider the designer in the creative process beyond that.
  - Mixed-Initiative PCG allows both the designer and the algorithm some level of initiative that varies on a case by case basis, a mixed authorship. It offers a type of AI-assisted game design. It's very hard for both the human designer and the computer to have the exact same level of initiative. In the cases in which the creative initiative comes from the designer, we have computer-aided design (CAD), where the PCG is facilitating their creative process, evaluating if the design goes against some constraint and presenting alternatives, much like a word processor that corrects and suggests better suited words to a writer. When the computer has the creative initiative, it generates the content but relies on the designer to evaluate it, guiding the generative process. In these cases we are dealing with interactive evolutionary computation (IEC) - especially useful when the evaluation is subjective, unknown, or too hard to be mathematically formulated. Since mixed-initiative requires interaction with the designer, this type of PCG is exclusively performed offline. Many of the Intelligent design tools (IDTs) previously mentioned in the last section offer some kind of initiative from the software.

(a) Computer-aided design: Humans have the idea, the computer supports their creative process

(b) Interactive evolution: The computer creates content, humans guide it to create content they prefer

Figure 2.1: Two types of mixed-initiative design. Image reproduced from [Shaker et al., 2016] chapter 11.

- Experience-Agnostic vs. Experience-Driven
  - Experience-Agnostic PCG, much like autonomous PCG that does not take into account the designer, does not consider the player's behavior during the generation of new content.
  - Experience-Driven PCG (EDPCG) [Yannakakis G. N and. Togelius, 2015], on the other hand, relates to the already mentioned player-adaptive games. The objective of EDPCG is to optimize the player's experience via the adaptation of the experienced content. Player experience is the collection of affective patterns elicited, cognitive processes emerged and behavioral traits observed during gameplay [Yannakakis and Togelius, 2018][Yannakakis and Paiva, 2014]. For an EDPCG approach it is necessary to find a way to map the generated content's quality to the patterns of player experience. Because of this need to be in interaction with the player, this PCG role can only be fulfilled in an online context. The example of *Left 4 Dead* [Valve Corporation, 2008] [Booth, 2009] has already been discussed, and another good one is that of *Galactic Arms Race* [Hastings et al., 2009], where new weapons presented to the player are evolved based on which ones they have fired the most, turning this aspect of the game into an evolutionary process in which the player is the one who, through their behavior, decides which of the individuals from the population are the fittest. Shaker et al. [Shaker et al., 2013d][Shaker et al., 2013c][Shaker et al., 2010b] applied EDPCG in *Infinite Mario Bros* where they considered multiple features to characterize the player experience, including subjective self-reports, objective measures such as the player's head movements, and gameplay features registered in logs. The player experience measurements were then used as a component of the fitness function for grammatical evolution of the game levels' structure.

2.4.2 Craveirinha, Barreto and Roque (2016)

While Togelius et al. had 2 dimensions to describe the role of the PCG algorithm, Craveirinha et al.'s taxonomy [Craveirinha et al., 2016] also considers the human designer and human player and how each of these three actors involved in the process fulfills the roles of generators and/or evaluators of the game content. Generation includes all processes with the goal of creation, recreation or iteration of a game content solution, while evaluation refers to any procedures or acts that determine the value of any generated solution, in a way that guides the creative generation process in subsequent iterations.

In Figure 2.2, each cell represents the possibilities for a specific combination of role and

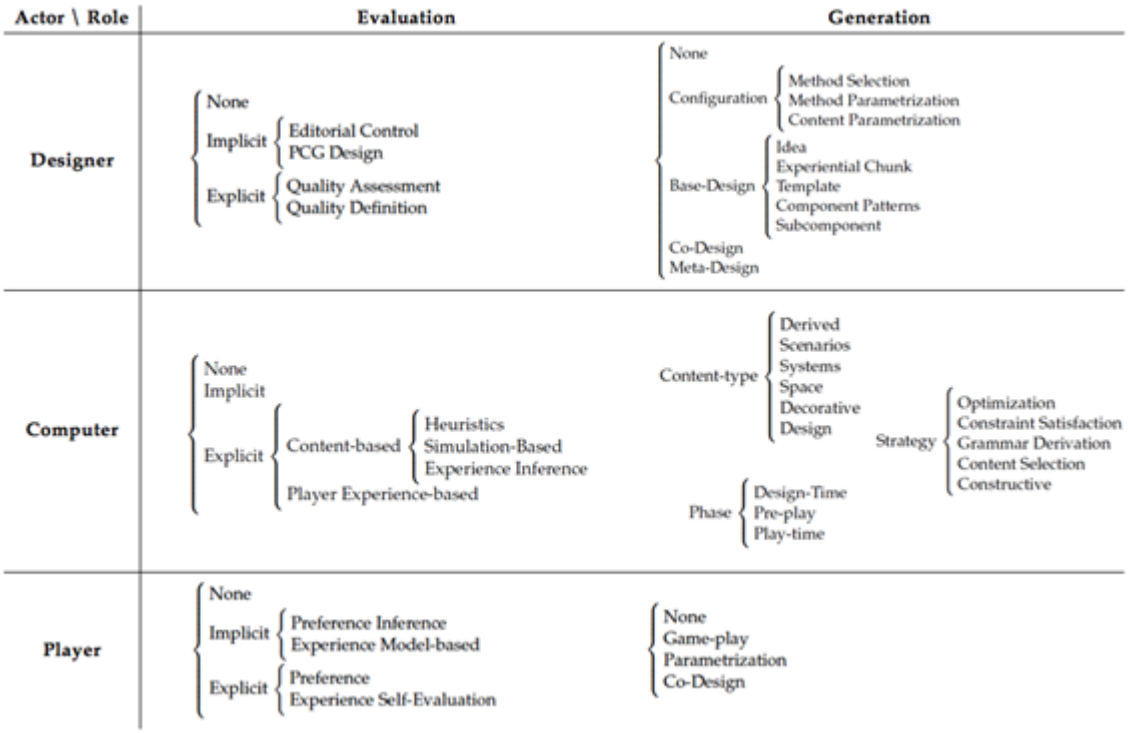


Figure 2.2: Taxonomy used to clarify actors' roles in PCG methods. Image reproduced from [Craveirinha et al., 2016].

actor that, unless otherwise stated, are not mutually-exclusive. The cell of computer generation is unique as it is further subdivided into three sub-taxonomies.

Human Designer Evaluation

Regarding evaluation, both human players and designers can impact this process implicitly or explicitly, or not at all.
The designer may indirectly affect how solutions are evaluated in the process of Designing the PCG system, by coding its procedures in a way that implicitly promotes certain qualities in generated content and demotes others. However, the most common form of implicit designer evaluation is Editorial Control - when using PCG tools (such as the already mentioned IDTs) in the development phase designers will often repeat procedural processes to obtain the best results. Explicit evaluation by the designers, on the other hand, refers to Quality Assessment or Quality Definition. In Quality Assessment, the most direct approach, the designer is attributing quality to generated solutions presented to them by the PCG methods, either by selecting which solutions to pursue or which quality value is attributed. In Quality Definition the designers do not attribute quality themselves, but define how the PCG system will automatically do it - they "express their intentions through the design of fitness functions" [Machado et al., 2014] [Machado and Amaro, 2013].

Human Player Evaluation

When players take part in the evaluation, we are most likely referring to player adaptive games, where every action with the finished video-game artifact affects how generated con-

tent is evaluated. Implicit evaluation by the player may come as Preference-inference, as seen in the previously mentioned *Galactic Arms Race* [Hastings et al., 2009] where player action or behavior betrays their liking a particular quality in generated content, or Experience Model-based, where an experience model that maps behavior to some experiential concept is used as the basis for new content, like in the previously mentioned *Left 4 Dead* [Valve Corporation, 2008] [Booth, 2009], which uses a director AI that controls game events and seeks to modulate player tension, stress, pacing, conflict, and difficulty. Methods that use explicit player evaluation usually do so through questionnaires, where they evaluate the quality of the content by attributing a score or comparing them - Preference - or they evaluate their own player experience, but not in terms of what they want or prefer in the game object - Experience Self-evaluation.

Computer Evaluation

Evaluation can also be done automatically and autonomously by the computational agent. This can occur implicitly by feeding some bias in the generation process, but it is more often an explicit process, where the algorithm enacts some formal act of content quality judgment, either Content-based or Player Experience-based. Content-based evaluation takes into consideration the intrinsic aspects of the generated content, using Heuristics, Simulation, and/or Experience Inference. Heuristics map the functional relationship between intrinsic features and content quality, typically based on some theoretical model. Simulation-based evaluation employs artificial agents to "play" the game so that gameplay data can be extracted and used to infer the quality of the solution. Experience Inference is used when there is a mapping between structural content components and predicted affective states of the player, using these predictions as a basis for evaluating the content before it is played. Player Experience-based evaluations judge the content based on the player experience data instead of the intrinsic qualities of the solution.

Human Designer Generation

In the generation process, each actor can impact the generated solution in several unique ways.
The Designer can have four major types of generation impact. None is the easiest to understand - the computational system creates with absolute autonomy from the designer and without using any designer-authored content.
The Designer can impact generation through Configuration, meaning customization of the operation of a pre-programmed PCG method. This can occur by Method Selection, for when a PCG tool offers different alternatives to its generation algorithm; Method Parameterization, when procedural aspects can be configured in the generator; and Content Parametrization, when we impose metrics on the output content.
Designers can also impact PCG generations through the human-authored Base Design that is available. This Base Design can be an Idea, a high-level concept used as a basis for the system to generate solutions, as seen in *ANGELINA* [Cook and Colton, 2014], which requires a thematic input from a human agent that is then incorporated into the generation system which abides by it as a theme. Other Base Design cases are Experiential Chunks, large building blocks designed by human authors that are used to form the content; Templates, generalized forms of experimental chunks with blank spaces for the computer to fill in; Component patterns, human-designed components too small to greatly impact the player's experience, unlike experiential chunks.
Co-design corresponds to the mixed-initiative already talked about in the first taxonomy, in

which the PCG method gives both computational and human actors some level of initiative in the creative act.
Lastly, Designers can also impact the generation through Meta-design, meaning the cases where the designers create their own custom-made PCG algorithm. This category serves to distinguish between these cases and the ones where general-purpose ready-made methods are used.

Human Player Generation

The player can also have an impact in the generation process, not just the evaluation. Just like evaluation, this occurs via the interaction with the video-game. Game-play refers to cases where the in-game interactions are taken in account by the PCG to select which type of content to generate. These actions do not evaluate other content, they rather serve as a signal to which the system replies with dynamic consequences, like in the interactive drama *Facade* [Mateas and Stern, 2003], or the procedural quest generation system of *Skyrim* [Bethesda Game Studios, 2011].
Much like the Designer's Content Parametrization, there are also cases in which the game allows the player to Parameterize what will be generated, such as the map generation of the *Civilization* games [Firaxis Games, 2016] and *Dwarf Fortress* [Adams, 2006].
There may also be cases in which the player can Co-design with a computational agent inside the game, although this is less common.

Computer Generation

Computer Generation is where the bulk of PCG occurs. This taxonomy identifies both the generation Strategy used, the generated Content-type, and the Phase when generation occurs.
Strategy is the "how" of the computational agent generative process, the algorithm family used. Optimization methods can be seen as a search for the solution that best fits some criteria and are analogous to the Generate-and-test methods of the first taxonomy. Constraint Satisfaction methods are based on the declarative specification of the properties and constraints that define the content to be generated, such as the solver-based methods that will be talked about in Section 2.5.2 Grammar Derivation methods generate content through expansion of grammars and will be further explored in Section 2.5.3 Content Selection refers to simply sampling content from a database and presenting it to the player. Lastly, Constructive generators assemble previously constructed building blocks to create new content.
Content-Type is the "what" of the process. Derived content is all content that is a side-product of the game world, like news and broadcasts about game-events. Scenarios are content that describe a way an event will unfold, like story, level and quest sequences, and puzzles. System Design is the modeling of real-life phenomena, such as ecosystems and entity behaviour. Space content is the environment in which gameplay takes place. Decorative contents are all the elementary units of game content that the player usually does not interact directly with. Finally, Design content is generated by the likes of *ANGELINA* [Cook and Colton, 2014], and refers to prototypes of or fully fledged video games, a composite whole.
The Phase of the computational generation in this taxonomy is derived from the Online vs Offline dimension of the first one, but it has three possibilities instead of the two. Playtime is the same as the Online generation - it occurs at the same time that the game is being generated. Offline generation, however, in this taxonomy is divided into Design-

time, during the development cycle, and Pre-play, in a released game but before the players gameplay experience starts.

2.5 PCG Methods

As we have seen, PCG can be accomplished by several algorithmic means. Many of the methods used are also common AI methods, while others are inspired by other fields, such as theoretical computer science or even biology [Yannakakis and Togelius, 2018]. There is no superior method, there are only methods better suited to different PCG problems. Each one has its drawbacks and advantages that affect their speed, reliability, controllability, expressivity, and believability, and tradeoffs are always necessary [Shaker et al., 2016]. Occasionally, two or more of these types of methods are used together in order to take advantage of the better qualities of each one. Next, we will explore some of these types of methods in more detail.

2.5.1 Search-Based PCG Methods

Intensively investigated in the last few years [Togelius et al., 2010], the search-based approach to PCG operates under the assumption that within a space of possible solutions exists a good enough one, and that through iterations and modifications of a solution or population of possible solutions from that space, keeping the good changes and leaving behind the bad ones, eventually we will find a suitable one. Such a process can be compared to that of real life evolution of species or even the design process that we use for creating anything through iteration.

This type of approach requires a search algorithm that will move the process forward, a way to represent the content that is being evolved/generated, and an evaluation function (also called fitness function) to assess the quality of an artefact [Shaker et al., 2016] [Yannakakis and Togelius, 2018].

Search Algorithms used in Search-Based PCG Methods

As a search algorithm, search-based methods often use an evolutionary algorithm as their engine. Of course, some cases require more sophisticated approaches that take into account restrictions and constraints or are built for a specific representation of the generated content[Shaker et al., 2016].
Evolutionary algorithms are some of the best-researched search algorithms - they take inspiration from Darwinian evolution by dividing the generation process into selection and reproduction phases, creating a selective pressure that incentivizes solutions to evolve in the direction that leads to a better fitness score from the evaluation function.
Simple evolutionary algorithms have the offspring be just copies of the selected individuals from the previous generation with some type of mutation applied, while genetic algorithms are a particular type of evolutionary algorithm that adds recombination while still having some chance of mutation as a way to preserve the qualities that made the selected individuals fit[Shaker et al., 2016].
In the more complex cases just one evolution function won't be enough, and just adding their results to obtain the final fitness score mostly leads to one of them being optimised at the cost of the others. For these cases the best solutions can be obtained with a multiobjective evolutionary algorithm. Since often evaluation functions work against each other, the

use of these algorithms results in individuals with unique combinations of strengths[Shaker et al., 2016].
Non evolutionary algorithms can also be used. Stochastic search/optimization algorithms such as swarm intelligence algorithms (particle swarm optimization and ant colony optimization are some examples) can be used for the same purpose. Sometimes evolutionary algorithms are overkill and an exhaustive search algorithm can be a better fit, if we have the time or the search space is small enough. If finding good solutions is easy and we want to focus on diversity of output, random search may be better suited for the initial design job. Of course, these algorithms still need a representation for the content and an evaluation function[Shaker et al., 2016].

Content Representation in Search-Based PCG Methods

The choice of content representation affects not only the efficiency of the generation algorithm but also the search space that we can explore, which will influence the type of content that can possibly be generated. The structure that will represent the content is the genotype, which we can later use as instructions to create the final piece of content, the phenotype.
How directly or indirectly the phenotype is obtained from the genotype is one of our main decisions regarding representation. As an example [Shaker et al., 2016], a level from *Super Mario Bros.*[Miyamoto and Tezuka, 1980] may be represented directly by a 2D grid of variables that translate directly to the blocks of the phenotype; a little less directly by a list of positions and properties of different game entities and structures; a little more indirectly by a list of already made reusable patterns; indirectly by a list of desirable properties of the level, like number of enemies, coins and endless pits; and completely indirect with a random number seed. While it isn't very logical to evolve a random seed, all other examples are valid representations to generate a level. This choice of how directly the genotype correlates to the phenotype also has consequences to the locality and dimensionality of the search space. A more direct representation leads to bigger dimensionality which could significantly affect the efficiency of the search - curse of dimensionality - while also increasing the locality of the solutions - neighboring solutions have much in common with each other. Indirect representations have less dimensions and give us smaller search spaces, easier to explore, but with a lesser expressive range compared to more direct approaches.

Representation naturally has a big impact on the appearance of the generated content. Figure 2.3 [Ashlock et al., 2011] shows four different maze-like maps generated with similar evolutionary algorithms that use distinct ways of representing the genotype of the solutions [Shaker et al., 2016].
Top-left uses a negative representation, where blank spaces are added to a matrix field with ones according to the genome, where each loci of the gene indicates the position of a room and its dimensions. With a big population and an evaluation function that properly defines what we are looking for, solutions with connecting rooms start to be generated. Sparse initialization - in the initial population of random solutions, the probability of a tile being filled is 0.2 instead of 0.5 - is used to find solutions with a connected map easier.
Top-right uses a binary representation in conjunction with required content. It contains some rooms specified by the designer in a configuration file. The first loci of a gene specifies the position of these rooms and the rest determine if each tile is filled or not.
Bottom-left is the positive representation, in which walls are added to a blank matrix. These walls' position, direction and length are specified in the genotype.
Bottom-right uses a binary representation with symmetry. The genotype indicates for each tile of one of the quadrants of the map if it is filled or not. This quadrant is then replicated

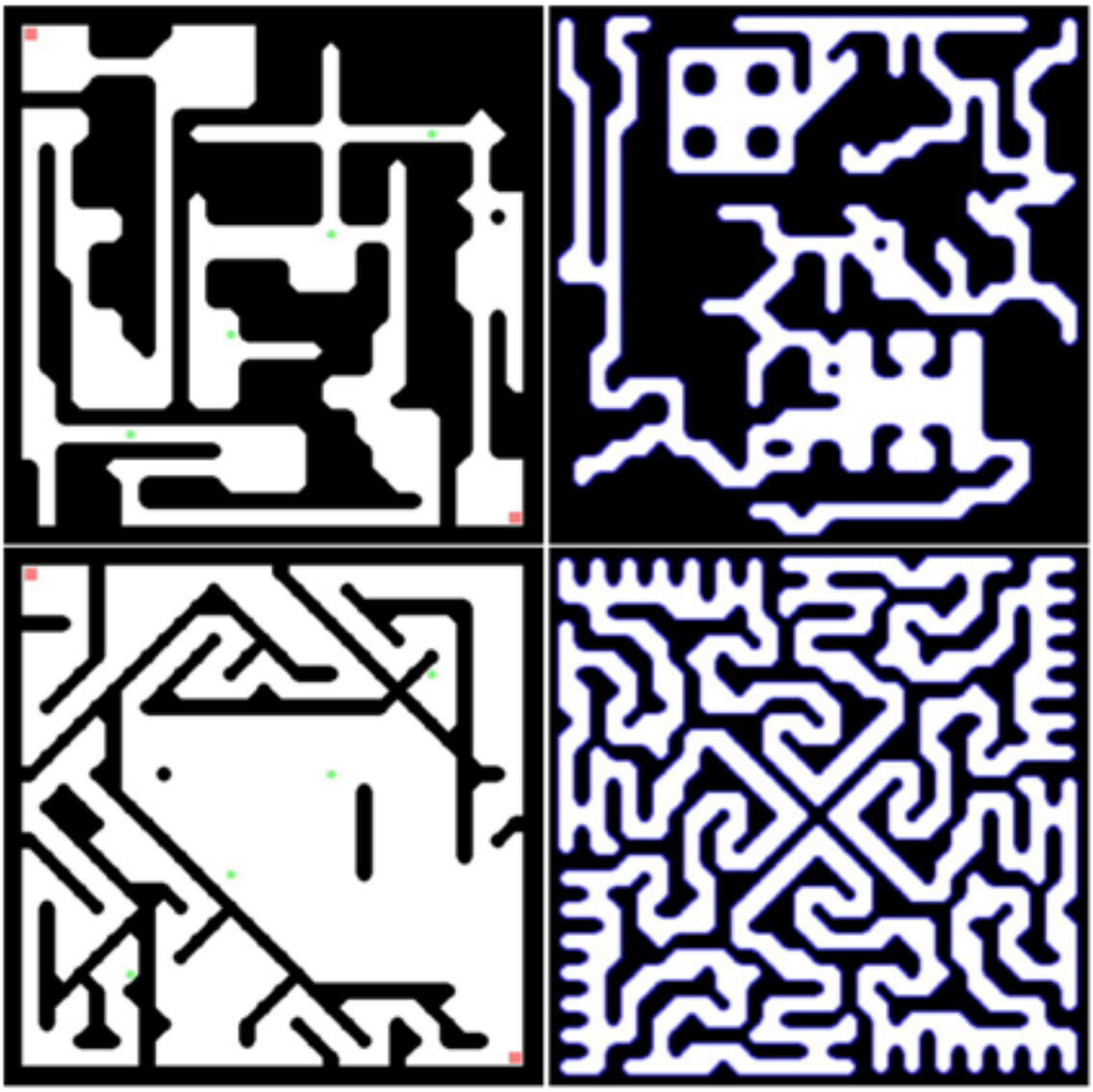


Figure 2.3: Maps generated using four different representations. Image reproduced from [Ashlock et al., 2011].

in the other 3 in a symmetrical fashion, giving it a distinct look.

Evaluation Functions in Search-Based PCG Methods

The definition of an evaluation function is many times the hardest part of the creation of one of these methods. If we do not have a good evaluation function, we will not find good content. They have to model some type of quality that we want to find on the artifact. Some qualities are easier and more straightforward to model, like guaranteeing that the rooms in a map are all connected, while others are a bit more ambiguous, like playability, regularity, or fun. It's a hard task to formalize and give a score to something as subjective as "fun" [Shaker et al., 2016]. In some studies, fun is calculated using the player's performance [Togelius et al., 2006][Togelius et al., 2007a], while others question the player directly with self-reports [Shaker et al., 2010a], and others determine fun through the patterns of alternating difficulty found in the generated content [Sorenson and Pasquier,

2010]. It can be argued that to understand what humans find fun one must think like a human, which AI is far from achieving - AI-complete. However, with the right heuristics, we can go very far [Yannakakis and Togelius, 2018].
Evaluation functions can be divided into three classes:

- Direct, which map extracted features from the generated content into a quality value. They are quick to compute and easy to implement, however, in some cases, it's difficult to find a direct evaluation for some qualities of game content. These types of evaluation functions can be further divided into Theory-driven and Data-driven. Theory-driven are guided by qualitative theories of player experience, like in the already mentioned [Togelius et al., 2006][Togelius et al., 2007a], where Togelius et al. evolve racecar tracks with an evaluation function based on a combination of theoretical studies of what makes a game fun and the intuition of the developers. In the case of Data-driven direct evaluation functions, the mapping between features and fitness can also be contingent on a model of the playing style, preferences, or affective state of players - personalizing the experience. In [Shaker et al., 2012b][Shaker et al., 2013b], Shaker et al. train neural networks to predict the player's experience and use that prediction as a basis for the evaluation function.

- Simulation-based evaluation uses AI agents to play the generated content, using the generated statistics to calculate the value of quality. The aspect of the content that is being evaluated should be reflected in the type of agent used. If we are evaluating if a generated level can be completed, we should use an agent trained solely on completing levels. In [Togelius et al., 2007b], an agent is trained to drive like a player and used to playtest newly generated car tracks before the player ever sees them, and in [Cardamone et al., 2011b], first-person shooter maps are scored based on how long matches between two AI agents last. Simulation-based evaluation can further be divided into two other types: Static, in which it is assumed that the agent behavior stays the same throughout the simulation, and Dynamic where the agent adapts itself, leading to the possibility to evaluate learnability.

- Interactive evaluation functions take a "human in the loop" approach. Data collection for this type of evaluation can be done either implicitly or explicitly. Implicit data collection does not query the players directly but instead interprets their behavior observed within the game to assess their preferences or affective state. One good example is that of Galactic Arms Race [Hastings et al., 2009], where the current population of solutions is the three weapons that the player has in their possession at any time, and the fitness of these weapons increases with their frequency of use. Whenever a new weapon has to be generated, their genotype is formed from recombination of the genotypes of the weapons in the player's inventory, with their probability of being selected correlated with their frequency of use. Such type of data collection has the risk of being noisy, or to be based on wrong assumptions about what in the gameplay reflects the enjoyment of the player. Explicit data collection, on the other hand, uses direct self-reports from the player, and if not well integrated it has the risk of disturbing the play session. However, they have the benefit of being more reliable and accurate to the player's experience. In [Cardamone et al., 2011a] users play and evaluate race tracks according to their preferences, which leads to later generations of tracks being better suited to them, increasing the player's satisfaction. Hybrid methods of data collection are also used as a way to mitigate the problems of both approaches [Shaker et al., 2013a][Martinez and Yannakakis, 2011][Yannakakis and Hallam, 2007].

When to use Search-Based PCG Methods

Search-based PCG methods are quite versatile - evolutionary computation can be used to generate practically any type of content with the right representation and with the ability to incorporate any type of objective and restriction with evaluation functions[Yannakakis and Togelius, 2018].
However, this versatility in many cases is acquired through a trade-off with efficiency. They can be quite slow in the more extreme and complex cases since they work through the evaluation of multiple candidates. If we have a time limit, it is not guaranteed that a suitable solution will be found by the end, so they aren't the best choice for a critical generation unless we are certain of the efficiency of the particular instance of the method that we are using. Many other methods can be better suited in some cases and for some particular type of content, but the search-based beats all others in versatility[Yannakakis and Togelius, 2018].
It's not rare to find examples of other methods being combined with search-based approaches, as we will see. In many cases, the evolutionary algorithms are used as a way to search the parameter space of the algorithms of these other methods[Shaker et al., 2016].

2.5.2 Solver-Based PCG Methods

Solver-based methods, much like search-based methods, see the generation of content as a search in a space of possible solutions. But instead of generating and testing iteratively several solutions, they find a solution in one pass, using constraint solvers, much like the ones used in logic programming. Programs written in a logic programming language are a set of sentences in logical form, expressing facts and rules about some problem domain [Wikipedia contributors, 2021].
Instead of defining a representation of the content and an evaluation function separately, in solver-based methods the logic of the content domain and the constraints on the properties of what we want to generate are defined in conjunction [Smith and Mateas, 2011a].
The logic of the content domain is its structures and mechanics. Structures describe the environment where the gameplay will take place, like for example a grid composed of various tiles. Mechanics, on the other hand define what can be done in that environment - how the game works - like, "the player starts on tile X", "the player can move to agencent tiles", or "the player can pick up item Y" [Shaker et al., 2016]. Codifying this logic in computational logic we can use it to guide the generation.
Constraints are what we use to define which properties related to the game logic that we want the generated content to exhibit. Things like "There must be a sequence of actions that the player can take to complete the level", or "the start and end of the level must be on opposite sides of the map" [Shaker et al., 2016].
Logic and Constraints can be understood as the equivalent in search-based methods as the representation and the evaluation function, only more explicit and symbolic. They are passed on to a constraint solver that tries to solve the given logic problem, finding an instance of content that conforms to the logic and satisfies all constraints. These solvers work by repeatedly eliminating parts of the search space until only variable solutions are left [Yannakakis and Togelius, 2018]. While if we stopped a search-based algorithm in the middle of the process we would, although not completely optimized, still get a solution, if we did the same with a solver-based approach we would only have partial solutions.

One of the major logic programming approaches is Answer Set Programming (ASP)[Smith and Mateas, 2011a][Smith and Mateas, 2011b], which programs are expressed in a language called AnsProlog and are processed by an ASP solver. The specifics of this language are

beyond the scope of this text, however it should be mentioned that ASP is one of the most used approaches in solver-based PCG research and applications. One example of a game using ASP to generate levels is Refraction [Smith et al., 2012][Butler et al., 2013] Figure 2.4, an educational game. The use of hard constraints defined with ASP lands itself well for the generation of logic puzzles. Many other languages can be used for PCG [Horswill and Foged, 2012], but answer set programming is well established with reliable existing tools [Shaker et al., 2016], making it a good general-purpose choice for programming logic and constraint-based PCG systems.

Figure 2.4: Refraction level generated with ASP. Image reproduced from [Smith et al., 2012] [Butler et al., 2013].

Solver-based methods, depending on how many constraints need to be satisfied, can find a viable solution in little time. While the worst-case complexity is often shockingly high, these algorithms can be very fast in practice [Yannakakis and Togelius, 2018]. They are ideal for generation problems that can be formalised and encoded in the language of the constraint solver - those that operate on hard logical constraints. Smith and Whitehead's *Tanagra* [Smith et al., 2011a] is a mixed-initiative design tool to build levels for platformer games. It uses a constraint solver that takes into account a number of constraints not only about what makes a level solvable, but also aesthetic considerations, like the "rhythm" of the level.
El-Nasr et al. [El-Nasr et al., 2009][El-Nasr, 2005] demonstrated constraint optimization to procedurally generate configuration of lights in order to enhance the player experience.

The search for the right constraints that result in the space of viable solutions that we are envisioning is many times an iterative process led by the designer. Some research has been done on the combination of search-base methods and solver-based methods, by using an

evolutionary algorithm to find the parameters of and answer set program [Togelius et al., 2012].

2.5.3 Grammar-Based PCG Methods

Formal grammars are fundamental structures in computer science and they have many uses in PCG too. They are composed of a set of production rules that describe how to form strings according to an associated syntax. Production rules define how a string may be rewritten - a pattern on the original string may be replaced by another one, forming a new string that complies with the syntax. Grammars can be generative because by the virtue of describing these structures, they also describe how to generate them. Any type of content that can be represented by a string of values can be generated by a grammar-based PCG method. Particularly, content that has self-similar nature - structures found on a micro-level repeat themselves on a macro-level - is the best suited for this type of generator [Shaker et al., 2016].
Plants and vegetation are one of the most usual cases in which grammar-based methods are used, even in AAA games - they are a common component of the environment of many games and usually don't affect the gameplay. Because, in the majority of the cases, they need to exist in large numbers and with a decent level of variety as not to look "copy-pasted", they are a prime target for PCG, and grammar-based methods have proven to be the best solution. *SpeedTree* [Interactive Data Visualization, Inc., 2014], as an example, is a widely used middleware in AAA games for the procedural generation of vegetation. One of the best ways to generate a tree or a bush is with one particular type of formal grammar called an L-system [Shaker et al., 2016]. L-systems have a peculiarity of the rewriting of the strings being made in parallel. Interpreting the generated strings has instructions for a turtle in turtle graphics, interesting organic forms can be created [Yannakakis and Togelius, 2018]. Aristid Lindenmayer introduced them in 1968 as a way to module the growing patterns of organic systems [Lindenmayer., 1968]. Bracketed L-systems are an extension to the regular L-systems that allows for the storing of states, meaning that the position and orientation of a turtle at any given step of the creation process could be stored and later recuperated, allowing for ramified structures, like trees. Other extensions to L-systems exist, like non-deterministic L-systems that allow for greater diversity, or context-sensitive L-systems that can create more complex patterns. Many plant generators in video games are based on L-systems or other formal grammars - for more examples, Prusinkiewicz and Lindenmayer's *The Algorithmic Beauty of Plants* [Prusinkiewicz and Lindenmayer, 1990] presents several organic structures generated with L-systems, Figure 2.5.

But grammar-based PCG Methods can be used beyond vegetation. They have also been used to generate game levels. Game levels, however, are not a singular construction, but a combination of two interacting structures: a mission(sequence of tasks that the player has to undertake) and a space - both have been generated using graph grammars [Dormans, 2010]. Graph grammars work in a similar way to string grammars - the production rules, Figure 2.6 define how a subsection of a graph may be replaced by a different pattern, Figure 2.7. While string could work to describe linear missions, graphs can have divergent and convergent paths, allowing for missions with optional tasks or tasks that can be done in arbitrary order, Figure 2.8. The space can also be generated through graph grammars - not the geometry but the topology. Missions have to take into account flow, pacing, and causality, while the generation of space has to focus on connectivity, distance, and signposting. Methods for generating whole levels need to generate each structure - the space and the mission - in a way that strengthens these individual qualities and make sure that they are interrelated and work well together [Shaker et al., 2016].

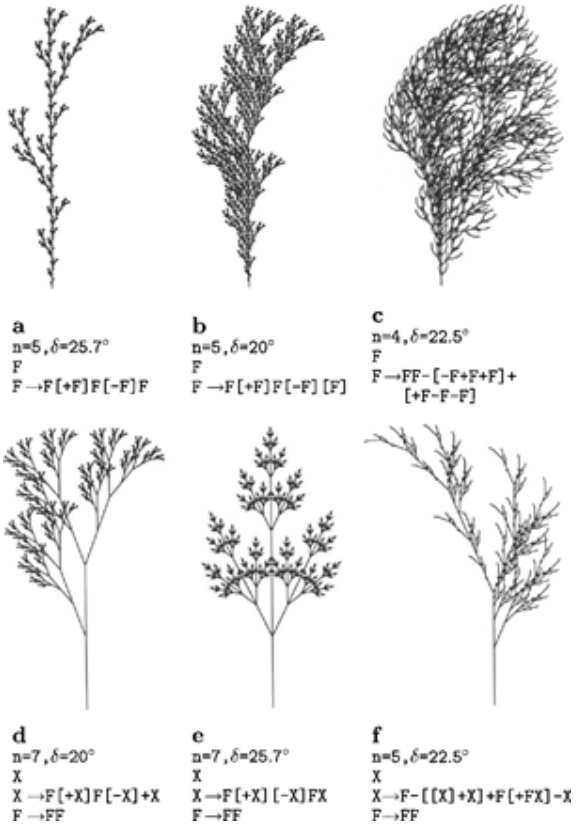


Figure 2.5: Examples of plants generated with L-systems. Although the example is in 2D, the same logic of using the generated strings as construction instructions can be also applied for 3D. Image reproduced from [Prusinkiewicz and Lindenmayer, 1990]

Figure 2.6: A graph grammar rule. Image reproduced from [Shaker et al., 2016] chapter 5.

Adams [Adams, 2002] uses graph grammars to generate the topology of FPS (first-person shooter) levels, resulting in mazes of interconnected rooms, Figure 2.9. Through a search algorithm, at each step, the production rule that is best suited for the parameters given by the designer is chosen. The choice of what production rules to include in the set to comply with the requirements of the level to be generated is done in an ad-hoc manner. New sets of production rules have to be created for different games.

Dormans in [Dormans, 2010][Dormans and Bakkes, 2011] uses a graph grammar not to structure the topology of the level, but the mission first. A shape grammar, Figure 2.10 is used to generate the space. Shape grammars function as any other formal grammar that we have talked about. Their alphabet is constituted by symbols like "walls" and "empty space", and a non-terminal symbol "connection". The generated mission is used in conjunction with the shape grammar to generate the space. To do this, rules associated with the terminal nodes of the mission grammar were added to the shape grammar. In each step of the generation of the space, the system looks for the next task in the mission to be

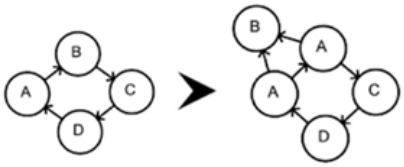


Figure 2.7: Example of the graph grammar rule in Figure 2.6 applied on a graph. Image reproduced from [Shaker et al., 2016] chapter 5.

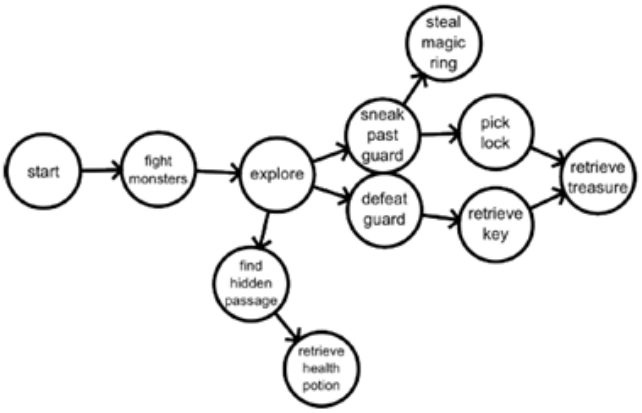


Figure 2.8: Example of a mission graph with two paths. Image reproduced from [Shaker et al., 2016] chapter 5.

implemented and finds the rules that are associated with it, selecting one at random, based on their relative weights. The usage of this mission grammar allows for some gameplay-based control in the generation of the level.

Van der Linden et al. [Van der Linden et al., 2013] strengthens the gameplay-based control of the designer by using gameplay grammars, Figure 2.11. Gameplay grammars are another type of graph grammar that uses nodes that represent the possible actions that the player can do. By changing the production rules of these grammars, the design has control over what sequences of actions will be required of the player. The gameplay graph is used to map the game space and with a second procedural method generate the geometry, using the intended gameplay to restrict the space generation.

Search-based methods can also be used in combination with grammars. An evolutionary algorithm could be used to search for the best set of production rules, or in the case of Grammatical Evolution (GE) [O'Neill and Ryan, 2001] search for the best order of production rules to apply. GE's are based on Genetic Programming (GP), using as a genotype a vector of integers. Each solution is synthesized using a context-free grammar, by reading each of the integers in the genotype and using them to select the production rule from the possibles for the current symbols. It is possible for an individual to not have enough genes to complete the process, in those cases we either can declare it invalid and give it a negative fitness, or restart from the beginning of the genotype until there are no non-terminal symbols [Shaker et al., 2016].
Shaker et al. [Shaker et al., 2012a] use GE to generate levels for *Infinite Mario Bros*, Figure 2.12. Levels are generated by positioning premade chunks in a bidimensional map. Each chunk has several properties that affect its positioning on the map, width, height, number and type of enemies, etc. The grammar used has these chunks in consideration and also has

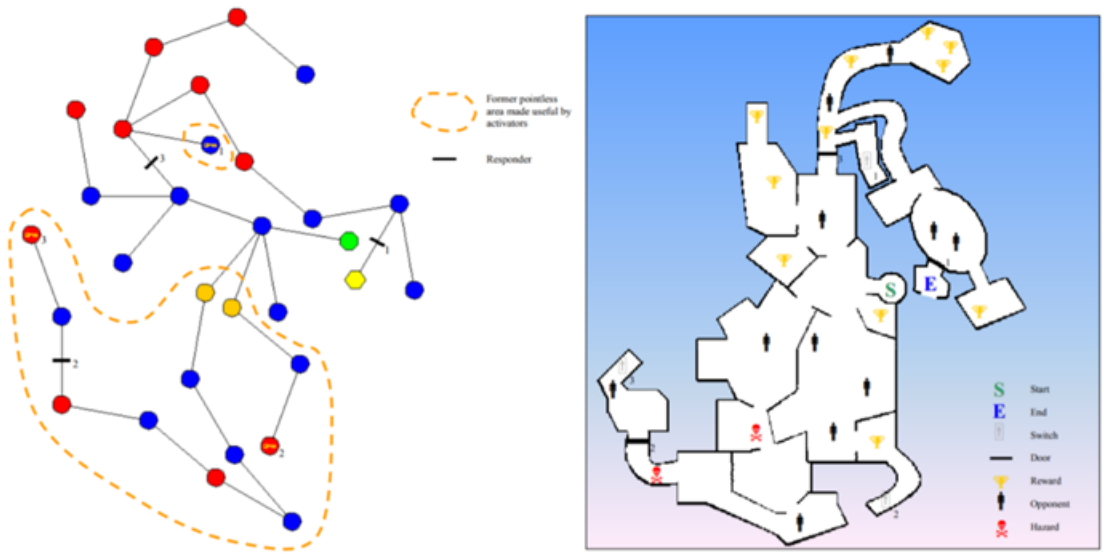


Figure 2.9: Topology graph generated by Adam's system and a possible two-dimensional map for it. Image reproduced from [Adams, 2002].

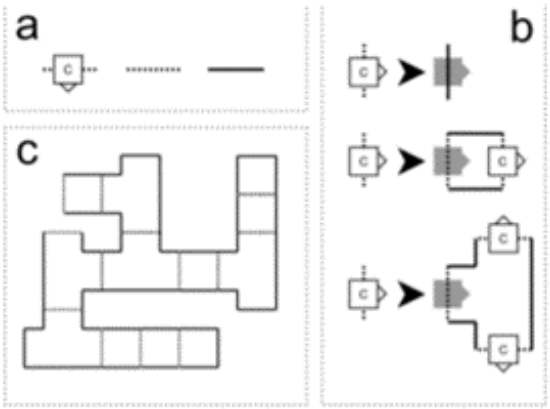


Figure 2.10: Example of a shape grammar. a) alphabet, b) rules, c) possible output. Image reproduced from [Dormans, 2010].

production rules that determine the values of their parameters. The synthesized solution is a list of chunks that constitute the level and their parameters. The overlaps between chunks are resolved by giving priority values to each chunk type and allowing some specific pairings, like hills of different heights, to overlap. The evaluation function is the weighted sum of the number of chunks used and the number of conflicts with overlap detected.

Beyond game levels and plants, some other applications of grammar-based PCG methods are the rock generator *SpeedRock* [Dart et al., 2011], Abela et al.'s [Abela et al., 2015] generators of aquatic flora for underwater environments, and Mark et al.'s [Mark et al., 2015] procedural generation of 3D cave systems.

Figure 2.11: a) example of gameplay graph , b) Level layout generated from a). Image reproduced from [Van der Linden et al., 2013].

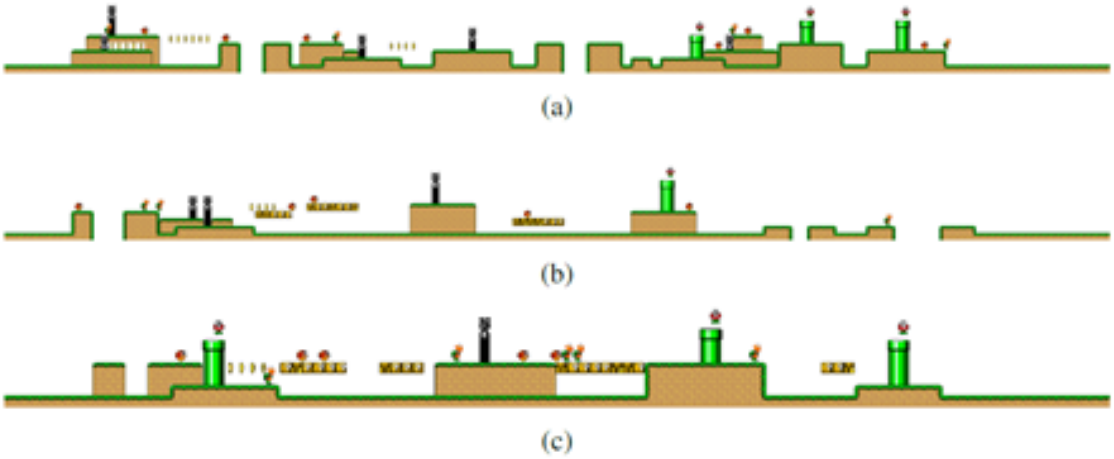


Figure 2.12: examples of levels generated for *Infinite Mario Bros.* with GE by Shaker et al. Images reproduced from [Shaker et al., 2012a].

2.5.4 Cellular Automata-Based PCG Methods

Cellular Automaton (CA) is a discrete model of computation widely studied in computer science, physics, complexity science, and biology. It can be used to computationally model biological and physical phenomena such as growth, development, and emergence [Yannakakis and Togelius, 2018]. Cellular automata can be seen as a set of cells placed on a grid (typically of two dimensions) that can be in one of many states. These states change through a number of discrete time steps according to a set of rules regarding the current state of the cell and the state of its neighbors.
There are two main methods to define the neighborhood of a cell: the von Neumann neighborhoods and the Moore neighborhoods, Figure 2.13. The number of possible configurations for a neighborhood equals the number of possible states for a cell to the power of the number of cells in the neighborhood. For a cellular automaton with two states and a Moore neighborhood of size 2, each neighborhood would have 33,554,432 possible configurations [Shaker et al., 2016]. For small neighborhoods, it is possible to define the transition rules as a table that maps each configuration of the neighborhood to a new state. But for cases where the number of possible configurations is too big, like in the example, the proportion of states within the neighborhood is used, as is "if more than 50% of the neighboring cells are in state 1, then in the next time step switch to state 1".

Cellular automata were introduced in the 1940s by Stanislaw Ulam and John von Neumann

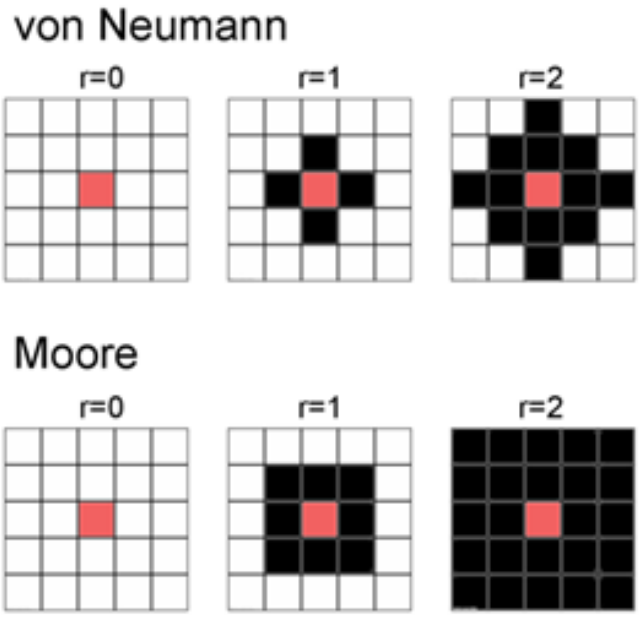


Figure 2.13: the von Neumann and Moore neighbourhoods for sizes 0, 1 and 2.

[von Neumann., 1951][von Neumann., 1966], but it was only 30 years later that they were used for more than research with Conway's *Game of Life* [Games, 1970]. The *Game of Life* has been classified as a zero-player game since it requires no input from the player - apart from the definition of the initial state [Shaker et al., 2016]. It is probably the most popular application of Cellular Automata. Much of the fun of the game comes from observing how complex mechanics can develop from such simple rules depending on the initial state, as cells are "born", "live" and "die". It is Turing complete, being able to simulate self-replicant machines or any other Turing machine [Zucconi., 2020]. Through the years multiple configurations of cells that result in unique behavior have been found and registered. This list of configurations as well as a playable version of the *Game of Life* can be found in https://playgameoflife.com/.

In video games, cellular automata are extensively used to model environmental systems. In [Forsyth., 2002] Tom Forsyth demonstrates how cellular automata-based methods allow for the simulation of a wide range of real-world effects, such as air and water pressure and flow, heat, and the spread of fire, smoke, and explosions with realistic effects. The usage of more organic and realistic models of such effects give way to a more dynamic game world for the player to experiment with and more inventive puzzles.

Penelope Sweetser in [Sweetser and Wiles., 2005] and later on her thesis [Sweetser., 2005], argues that the two ways of designing game environments - hand-crafting events through scripting, and the use of general rules that result in emergent behavior - are too extremes of the same continuum, neither one being ideal, and the use of both is the best solution - using hard-coded for story and game objectives, and emergence for interactions and behaviors outside those hand-crafted boundaries. Through the creation of more flexible game worlds that offer more freedom and control, Sweetser believes that player enjoyment could be improved. A great and recent example of this is the indie game *Noita* [Nolla Games, 2020] [Loginova, 2019], Figure 2.14. *Noita* started as a falling sand simulator and rapidly grew into a game in which, as its tagline says, "every pixel is simulated". It's a complex cellular automaton, in which each pixel follows simple rules that are associated with its material. Some materials are combustible and can be ignited by a neighboring pixel that is on fire,

others are different types of gases and liquids with different pressures, densities, and effects on the player and environment, such as acid that will corrode most other pixels. *Noita* gives the player the challenge of reaching the end of a world procedurally generated in which each pixel follows these simple sets of rules that can result in complex behavior which the player can take advantage of or, if they are unlucky, will result in their death.

Figure 2.14: Screenshot from *Noita*, where each pixel follows a simple set of rules dependent on its material. Image reproduced from [Loginova, 2019].

Cellular automata have also seen use in the terrain generation process. More specifically, in [Olsen., 2004] Jacob Olsen uses cellular automata-based methods to simulate thermal and hydraulic erosion on the initial base terrain.

These methods can also be used to generate level structure. Johnson et al. [Johnson et al., 2010] uses cellular automata to generate infinite cave-like dungeons, Figure 2.15. Since the cave system is infinite, stretching endlessly in every direction, it must be generated in real time. The game presents one screen at a time to the player, which offers time to generate the next few screens when the player exits to another room. Each cell of a 50x50 grid can either be empty or have a rock. The process starts by sprinkling randomly rocks through the grid as the initial state of the cellular automaton, and then running two time steps with the rule "if a cell has at least five rocks as neighbors, then it becomes a rock". The ending result is an organic, cave-like structure with pockets of empty space where the player can roam. Since each screen has to connect to the four other that surround it, if the end result has no connections to its neighbours, a "tunnel" of empty cells is excavated between the largest pockets of empty space in each screen and another two iterations of the cellular automaton are run to regain the organic look. The generation is extremely fast and can generate a screen and its eight neighbors in less than a millisecond on a modern computer [Yannakakis and Togelius, 2018].

A very similar process was used for the indie game *Galak-Z: The Dimensional* [17-Bit, 2015]. As is stated in their 2015 Game Developers Conference presentation [Aikman., 2015], the layout of the level is defined by a variation of a Hilbert curve, and the individual rooms are generated with the necessary entrances and exits in mind with a cellular automaton, to achieve the right organic and natural look, Figure 2.16. Various rule sets were tested to get the one that best generated the type of level the team was after. Is of note that

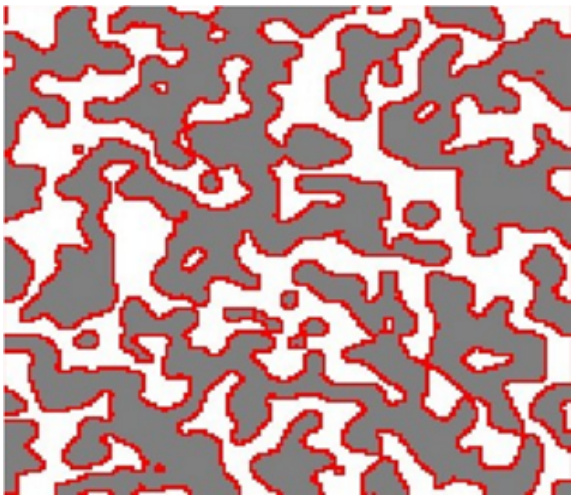


Figure 2.15: A 3x3 base grid map generated with CA. The results are an organic cave-like level. Image reproduced from [Johnson et al., 2010].

after the process, some post-processing is needed to remove some obstructing pixels, or to widen the exits.

Cellular automata-based PCG methods are quite quick and are best used for content that requires a more organic look, or for modeling systems with emergent properties. They have the advantage of requiring a small number of parameters and being relatively easy to implement. However, it is hard to know what the impact of one parameter will be in the final result since each one affects multiple characteristics of the output and emergent properties are hard to predict. This lack of control also affects gameplay, since it can't guarantee playability or a path that will allow for the completion of the level, for example. Any additional gameplay characteristic has to be guaranteed with the combination of other methods, used in post-processing and/or through a search-based approach [Yannakakis and Togelius, 2018].

2.5.5 Fractals and Noise-Based PCG Methods

Fractals and noise algorithms are frequently used in many PCG applications and are probably what most people think of when procedural generation is mentioned.
One of their biggest uses is the generation of textures, which, since the value of each pixel has a calculation behind it, are scale-invariant and can have infinite resolution. Noise is useful whenever small variations need to be added to a surface, like in skyboxes where clouds are white-colored noise, or dust settled on ground and wall textures, or to implement fire, plasma, or skin and fur coloration, among many other examples [Ebert. and Kaufmann, 2003][Shaker et al., 2016]. One of the most famous gradient noises is the Perlin noise [Perlin, 1985], implemented for the first time by Ken Perlin in 1982 as a way to generate CGI that was less "machine-like" for the movie *Tron*.
Any content that can be represented by a two-dimensional matrix of real numbers can be generated by these methods [Yannakakis and Togelius, 2018]. For texture generation, intensity maps can be generated, with the values corresponding directly with the luminosity of each pixel. For terrain generation, the same matrix can be called heightmap, and each value now represents the height of the associated point in the terrain. With this same representation, any method that can be used to generate textures can also generate terrain

Figure 2.16: A level of *Galak-Z* where we can see the Hilbert curve that is used as a basis for the connections between each individual room. Image reproduced from [Aikman., 2015].

or any other content that can use a two-dimensional matrix as its basis.

Real terrain and many other types of content have variations at various scales (or frequencies). Much like we saw with grammar-based methods in Section 2.5.3, the kind of variation that we see on a small scale is the same as the one we see on the bigger scales - mountains have peaks and valleys through their inclinations, and valleys have smaller hills and ravines within them. This self-similarity is the basis for fractals and it can be produced by methods that are based directly on fractal math or that produce similar effects by simpler means [Shaker et al., 2016]. An easy way to produce such effects is to take a single-scale random method and apply it multiple times, decreasing the scale each time. For the example of terrain, first, we would generate the big features - the mountains and valleys - and then we would add smaller hills and more subtle details as we decrease the scale. We can use noise in this way too, by scaling each noise layer by the inverse of their frequency [Shaker et al., 2016]. In [Olsen., 2004] (already mentioned in Section 2.5.4) multiple combinations of gradient noise are used to generate the base terrain that is later eroded with cellular automata-based methods.
Fractals are commonly used in real-time map generation [Ebert. and Kaufmann, 2003]. Musgrave et al. [Musgrave et al., 1989] grouped all methods for fractal terrain generation into five categories of technical approaches, all of which can be seen as implementations for the general concept of fractional Brownian motion (fBm) [Musgrave et al., 1989]. fBm can be understood as terrain being generated by multiple random walks following specific statistical properties. Since taking these million of random walks would be too computationally expensive, similar end results are achieved with various techniques [Shaker et al., 2016]. Midpoint displacement [Belhadj., 2007] is one of these techniques. It is used for the generation of two-dimensional landscapes and works by subdividing repeatedly a line. It's an iterative process in which the midpoint of the line is found and moved a random amount up or down, creating two new lines to which the same process will be applied, only

with a smaller range of random numbers, Figure 2.17.

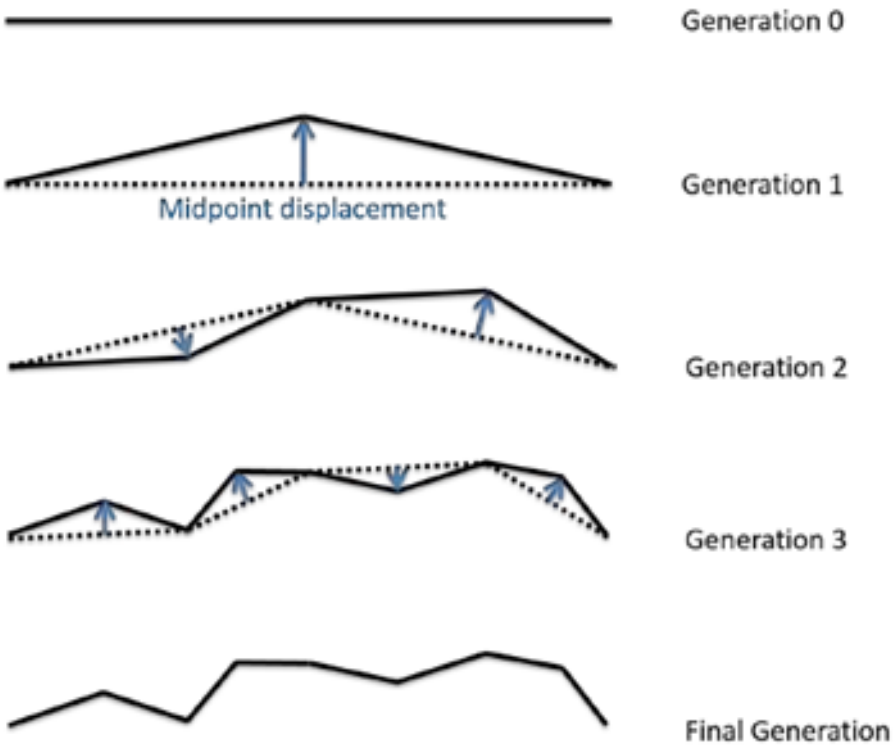


Figure 2.17: The midpoint displacement fractal process. Image reproduced from [Yannakakis and Togelius, 2018].

Midpoint displacement can be extended to another dimension (to generate two-dimensional heightmaps) with the Diamond-Square algorithm [Yannakakis and Togelius, 2018]. It is also sometimes called cloud fractal or plasma fractal for its regular use to generate such effects. The process behind the diamond-square algorithm encompasses two distinct steps that are applied to a heightmap: the diamond and the square steps. The process is initiated with the assignment of random values to the four corners of the heightmap, followed by the diamond step in which the middle point of the square is assigned the average value of the corners plus a random value. The square step consists of finding the four cells in between the corners and assigning them the average of the value of the two corners surrounding them and the middle point plus a random value. By the end of the square step, four new squares have been formed and the process can repeat itself in each of them, using a smaller range of random numbers, Figure 2.18.

For terrain, there are representations other than heightmaps that allow for more complex results, such as voxel grids. A clear example of this representation is *Minecraft* [Mojang, 2011], that applies 3D perlin noise on a voxel grid to generate structures that would be impossible with other representations, such as caves and overhanging cliffs. The distinct look of each biome in a *Minecraft* world is achieved by using different combinations of multiple noises in combination with other PCG methods to make each region unique [notch, 2011].

Much like cellular automata-based PCG methods, fractal and noise-based methods are fast, easy to use, but lack controllability [Yannakakis and Togelius, 2018]. A good way to regain some of that design and gameplay control is through search-based methods. Frade et al. [Frade et al., 2008][Frade et al., 2010][Frade et al., 2012] developed the concept of Genetic

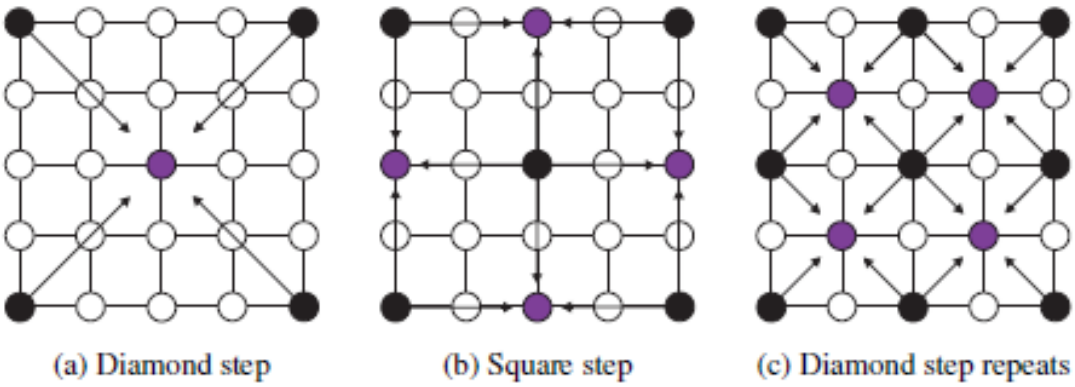


Figure 2.18: The diamond-square algorithm. Image obtained from [Shaker et al., 2016].

Terrain Programming (GTP), where an expression tree is used as the representation. These expression trees are evolved via genetic programming - a method for generating executable programs with evolutionary computation. In GTP the function set for the expression trees includes arithmetical, trigonometric, exponential, and logarithmic functions, and the terminal set includes the x and y coordinates, functions dependent on the distance to the center of the map, and several noise functions, including Perlin. Each expression tree is an indirect and compact representation of the map that allows for infinite scalability. To obtain the phenotype - the map - the algorithm iterates each point of the heightmap, querying the expression tree using the x and y coordinates as input, Figure 2.19. Various evaluation functions were tested for the system, including interactive ones in which users selected the terrains that they wanted to be used to generate the next generation, and more direct evaluation functions that, for example, promoted accessibility by rewarding smooth surfaces for the player to drive on. To avoid completely flat maps, the accessibility metric was counterbalanced by others that promoted some obstacles for the player[Shaker et al., 2016].

2.5.6 Machine Learning-Based PCG Methods

An emerging direction in PCG research is to train generators on already existing content to produce similar results. Such techniques have been used for a long time outside of video games with deep neural networks that produce images, text, and music based on the examples that are given to them [Yannakakis and Togelius, 2018]. However, the generation of game content brings with it additional challenges that producing images and text don't. Games can only be experienced by playing them, and if the generated level impedes that, then no matter what good qualities it has, it isn't a good level. Text, images, and music can have some imperfections and still retain some quality, but game content needs to ensure playability to be valid. The same methods used outside of the video game context may not be suitable for it [Yannakakis and Togelius, 2018].
Despite all the additional limitations, machine learning-based PCG methods are still an active research area [Summerville et al., 2017].
Markov models can be used for the generation of content that can be represented as one or two-dimensional discrete structures. Dahlskog et al. [Dahlskog and Togelius., 2013][Dahlskog et al., 2014] used n-grams - a particular type of Markov model usually used for text prediction [Yannakakis and Togelius, 2018] - to generate *Super Mario Bros.* [Miyamoto and Tezuka, 1980] levels. The levels were segmented into slices and the n-grams,

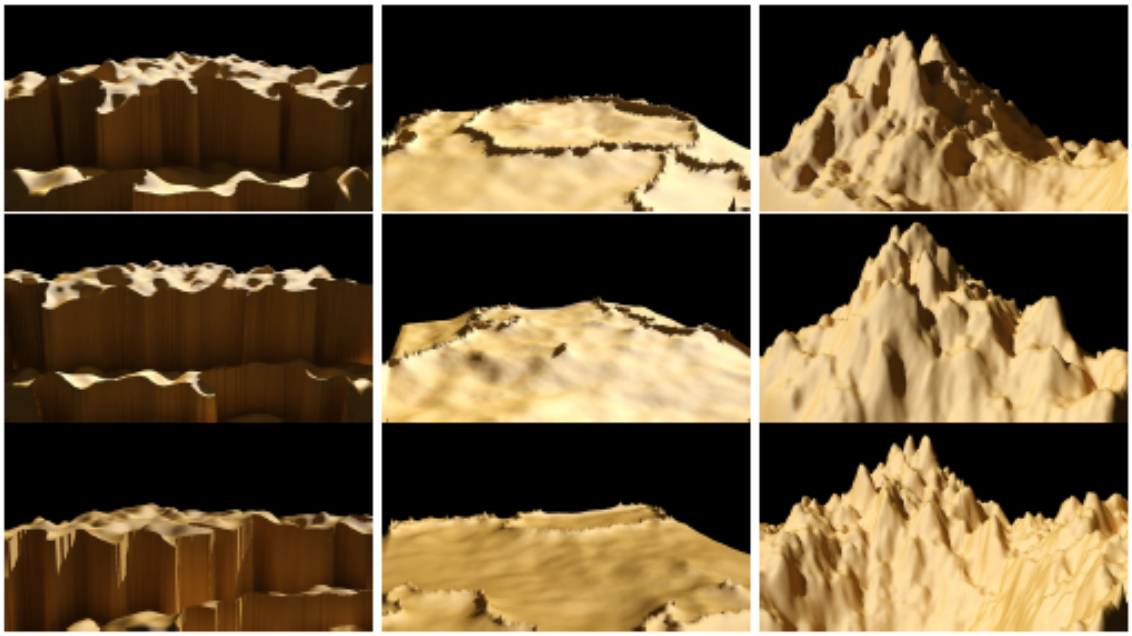
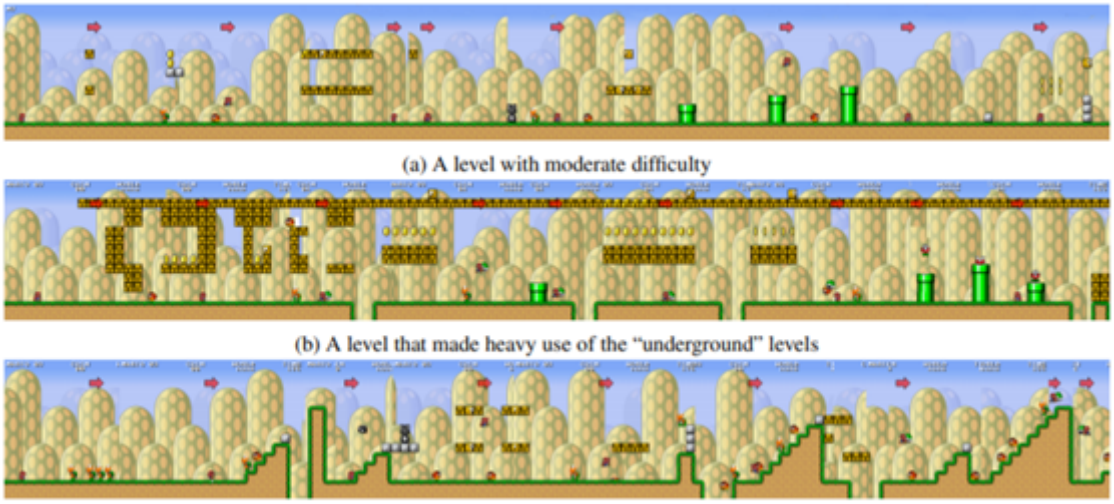


Figure 2.19: Terrain generated by GTP, using different evaluation functions. Image reproduced from [Frade et al., 2008].

instead of predicting words, were trained to generate sequences of these slices based on the patterns in the examples given. Summerville et al. [Summerville et al., 2015] use Monte Carlo Tree Search (MCTS) to guide the Markov Chain generation of the *Super Mario Bros.* [Miyamoto and Tezuka, 1980] levels, in what they call Markov Chain Monte Carlo Tree Search (MCMCTS). MTCS allows for the use of designer authored evaluation functions to guide the search and prune selections that lead to unplayable or otherwise undesirable levels, Figure 2.20.

Neural networks are also highly useful for machine learning-based PCG [Yannakakis and Togelius, 2018]. Hoover et al. [Hoover et al., 2015] generate levels for *Super Mario Bros.* [Miyamoto and Tezuka, 1980] through a representation called "functional scaffolding for musical competition", originally designed to procedurally compose music by generating accompaniment "voices" and "instruments". The game levels are seen through a music metaphor, with each type of tile representing a "voice" or "instrument" that are used to "compose" the generated levels. This work proves the utility of drawing inspiration from works of other fields.
Machine learning can also be used for generating content that doesn't have the constraint of having to be playable, like Summerville et al.'s [Summerville and Mateas., 2016] *Mystical Tutor*, a *Magic: The Gathering* Design assistant that generates inspirational raw material for card designers [Yannakakis and Togelius, 2018]. Since the generated content will only be used as inspirational material, it can have some flaws and still be useful.

Machine learning is also used in other PCG methods, such as search-based. Liapis et al. [Liapis et al., 2013] use a deep learning autoencoder as an evaluation function that evolves over time. Togelius et al. [Togelius et al., 2007a] generate race tracks personalized for each player, by having a neural network trained to drive like a player play through the generated tracks to determine if they are fit or not. In *Galactic Arms Race* [Hastings et al., 2009], use Compositional Pattern Producing Networks (CPPNs) as a representation for the evolved weapons. The CPPNs start simple, with few layers, and become more complex over the



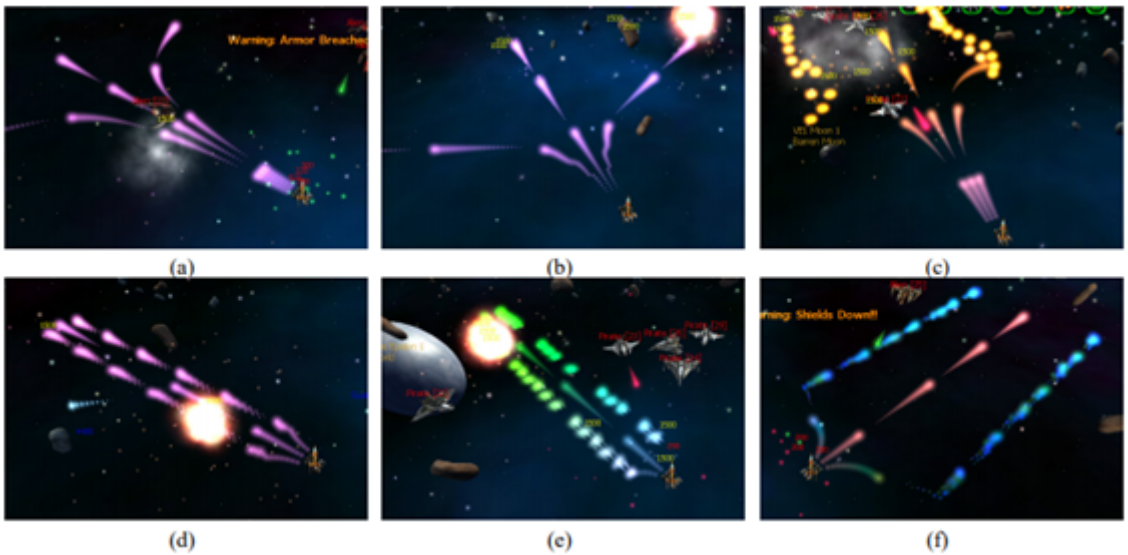
(a) A level with moderate difficulty

(b) A level that made heavy use of the "underground" levels

(c) A hard level with a high density of enemies

Figure 2.20: Three levels generated by Summerville et al.'s MCMCTS. Image reproduced from [Summerville et al., 2015].

generations. In each frame of animation, each particle inputs its position relative to the player into the CPPN and obtains its new velocity and color in return, Figure 2.21.



(a) (b) (c)

(d) (e) (f)

Figure 2.21: Examples of evolved weapons represented by CPPNs in Galactic Arms Race. Image reproduced from [Hastings et al., 2009].

2.5.7 Agent-Based PCG Methods

Game content can also be generated through the actions of one or multiple AI agents. These agents usually follow a simple set of rules related to their immediate vicinity rather than the content to be generated as a whole.
A good example of a simple implementation with just one agent can be found in [Shaker et al., 2016]. The objective is to generate the structure of rooms and corridors of a dungeon

level by allowing an agent to roam stochastically through the space "digging" the dungeon out. The look of the generated level is highly connected to the agent's behavior. Two agent approaches are detailed for this implementation:

- A highly stochastic "blind" method, in which the agent starts in a random location and direction, having an increasing chance to change direction as it moves forward. As it walks, it "diggs" the tiles it moves through, forming the corridors of the dungeon, having an increasing chance to "dig" a room of varying dimensions instead of a single tile. Since the agent does not take into consideration the already created structure, the resulting levels have intersecting rooms and corridors, Figure 2.22.

Figure 2.22: Example of a run of the "blind" agent. Image reproduced from [Shaker et al., 2016] chapter 3.

- A "look ahead" approach is developed with the shortcomings of the "blind" agent in mind. The agent considers all possible rooms it can "dig", choosing one that won't intersect with other rooms or corridors. If no room can be built, then the agent picks a direction and distance that won't collide with the existing topology and builds a corridor. If none of the possible actions are executable, the agent stops and the generation process ends. In the worst-case scenario, the agent only explores part of the existing space, Figure 2.23.

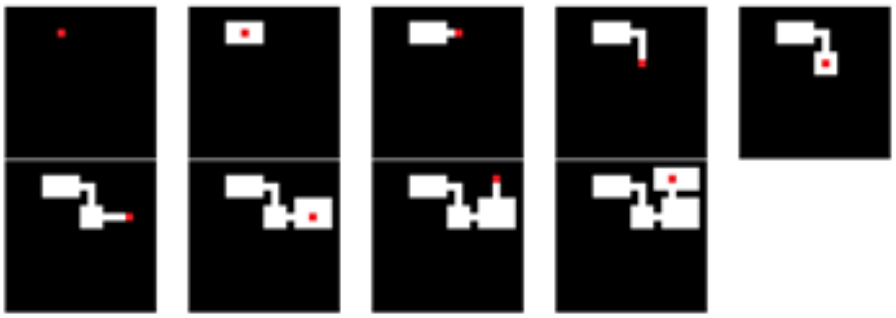


Figure 2.23: Example of a run of the "look ahead" agent. Image reproduced from [Shaker et al., 2016] chapter 3.

In the cases where more than one agent is used, the objective is to cover a larger area more efficiently and/or to have specific types of agents, each dedicated to one type of modification of the content. However, these agents work individually and don't have connections with each other - only interacting with each other indirectly, through the

changes they make on the content. Lechner et al. [Lechner et al., 2003b][Lechner et al., 2003a] utilize multiple agents to build road networks that connect separated areas of a city. These agents, called "road developers", are divided into several types with specific behaviors, including "extenders", that search for unconnected areas, and "connectors" that add connections between roads with long travel times.

A more complex example is that of Doran and Parberry [Doran and Parberry, 2010], which use six different types of agents for landscape generation. Every agent has a sense of the environment that surrounds it and the capability of modifying it, and each type of agent focuses on a specific aspect of the terrain, multiple of them working simultaneously to simulate natural phenomena.
Coastline agents are the first ones to act, by elevating points on the map above sea level. Each agent is assigned a small part of the map where they roam and modify the terrain until they run out of actions. Locality is preserved since the reduced number of actions for each agent limits them to their immediate surroundings. Increasing the number of actions decreases the number of agents, decreasing the detail of the coastline.
Smoothing agents eliminate rapid elevation changes generated by the coastline agents by moving around a starting point and modifying the height of arbitrary points to the weighted average of its neighbors.
Beach agents form beaches by traversing the shoreline in random directions and lowering points surrounding it if they are under a predefined height.
Mountain agents roam regions of the map above a certain altitude, raising mountain peaks and foothills.
Hill agents work similarly to mountain agents, only at lower altitudes.
River agents are the last ones to modify the landscape. They define two random points, one at the coast and another on a mountain, and move from the first to the second following the mountain's gradient, determining the river's path. They come back to the coast through the path, "digging" the river along the way.
The behavior of each type of agent can be controlled by the designers through parameters such as the number of actions they can take, the range of heights the beach agents can assign to the points they affect or even the shape of the mountains raised by the mountain agents. By controlling the behavior of the agents, the designers have direct control of the characteristics that the generated landscape will possess. Random seeds may also be utilized to set up the agents' parameters, Figure 2.24.

AI agents can also be incorporated in search-based approaches. Kerssemakers et al. [Kerssemakers et al., 2012] consider content generators another type of content that can be generated. They represent a procedural level generator as a set of agents that build levels, and by evolving this set of agents with an evolutionary algorithm they can optimize it. In [Kerssemakers et al., 2012] they apply this procedural procedural level generator generator (PPLGG) to *Super Mario Bros.* [Miyamoto and Tezuka, 1980] levels, Figure 2.25. Two types of evaluations are done to the generated content generators: a simulated-based one to determine if the generated levels are playable, and an interactive one, in which the user is presented a sample of 10 levels for each generator and picks which ones carry on to the next generation. Each content generator is composed of 14 to 24 agents, each one with parameters that determine how they move, for how long, what triggers their action, and what changes their action applies on the level.

Agent-based PCG Methods have been mostly used for level and terrain generation. The stochastic nature of the AI agents leads to the generation of content with a less structured feel to it, which can be ideal if we are aiming for something more organic and realistic. However, this also makes the process less controllable, and reduces the predictability of the generated content, with appearance and playability being hard to guess without trial

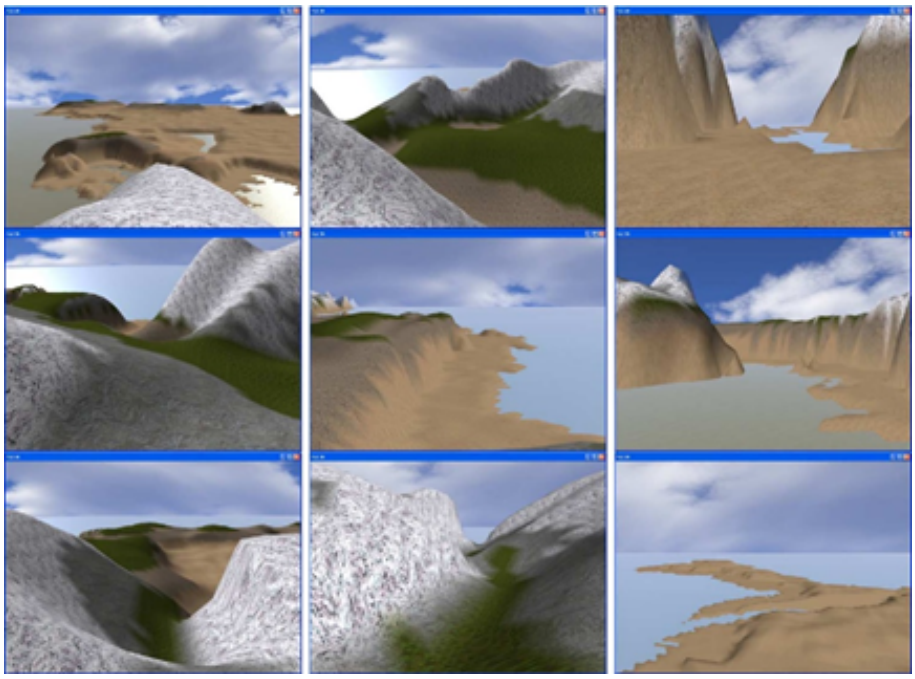


Figure 2.24: Agent generated terrains. Image reproduced from [Doran and Parberry, 2010].

and error [Shaker et al., 2016].

2.5.8 WFC-Based PCG Methods

Wave Function Collapse (WFC) by Maxim Gumin [Gumin, 2016] is a local constraint-based procedural generation algorithm utilized to generate bitmap patterns using an input sample bitmap pattern [Gumin, 2016][Gaisbauer, 2021]. It is a non-backtracking, greedy search algorithm capable of using a small number of constraints to generate large consistent output [Sandhu, 2019]. The output is generated in the style of given examples by ensuring every local window of the output occurs somewhere in the input [Karth, 2017]. Although being a very flexible algorithm, default WFC's generated output can be hard to control, since it doesn't have any global structure and only ensures that the output is locally similar to the input [Gumin, 2016].

WFC is relatively recent and has been getting more attention. It's been used primarily as a way to generate game maps, as seen in Caves of Cud [Freehold Games, 2015] and Bad North [Plausible Concept, 2018] [Boris, 2020]. The first game adds variety and greater control to the map generation, by subdividing it into different areas and then running WFC with specific settings on subsets of the map. The second game is a good example of the capabilities of WFC in 3D grids. Bad North controls the biome of the generated Island by limiting the tiles that the WFC algorithm can use [Boris, 2020].

The WFC algorithm shares its name with a process found in quantum mechanics that describes the way a system can evolve according to quantic laws. The algorithm shares little in common with how this quantic process works, nonetheless, the core idea is similar. A superposition of various probable outcomes is reduced to one by, in the quantum process case, being observed, and in the PCG version by the choice of the algorithm. By choosing one of the possible outcomes - one of the possible tiles in a specific point of the map – the group of possible tiles that can exist on each surrounding point is also changed.

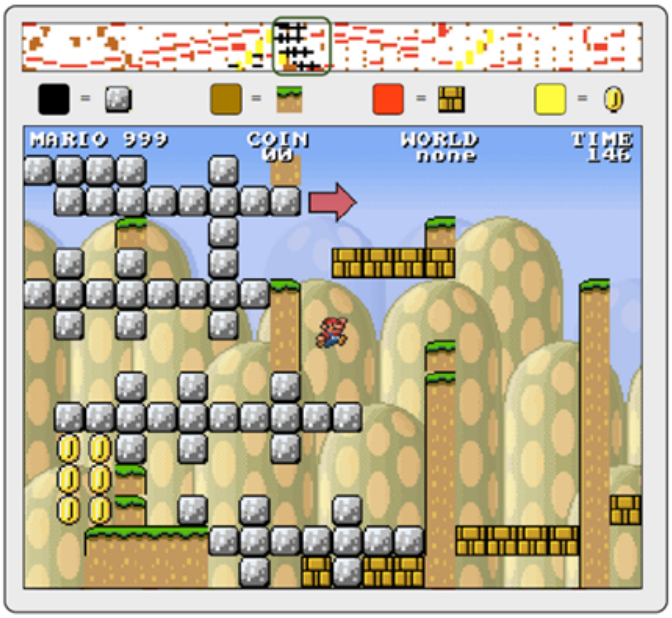


Figure 2.25: A level generated by Kerssemakers et al. PPLGG. Image reproduced from [Kerssemakers et al., 2012].

Figure 2.26: A WFC example by Gumin [Gumin, 2016]. On the left is the input given to the algorithm and on the right is the generated output.

The information is spread through the map like a "wave" [Sandhu, 2019], and the process repeats until all the points of the map are "collapsed" – only have one possible outcome.

The PCG WFC algorithm starts with the extraction of NxN patterns from the given input. The collection of available patterns may be augmented through optional processes of rotation and mirroring. Possible neighbors for each pattern are determined through the search of compatible borders, or directly from the positioning in the input.
The output is an array with predefined dimensions, referred to as "wave". The elements of the wave represent an NxN region of the output. They are superpositions of patterns. Each element has a boolean for each of the NxN patterns, indicating if they are forbidden in that region.
The wave is initialized in a completely unobserved state, i.e. with all the patterns being allowed for all regions.
The main loop of the algorithm starts by checking if the wave is fully collapsed, i.e. the output is ready. If there are still superpositions to collapse, the one with the least entropy is found and collapsed, i.e. one of the possible patterns is selected as the only allowed one at that region. Following this choice, the information is propagated to all neighboring

Figure 2.27: Examples of maps generated by WFC in Bad North [Plausible Concept, 2018]. Picture found in [Boris, 2020].

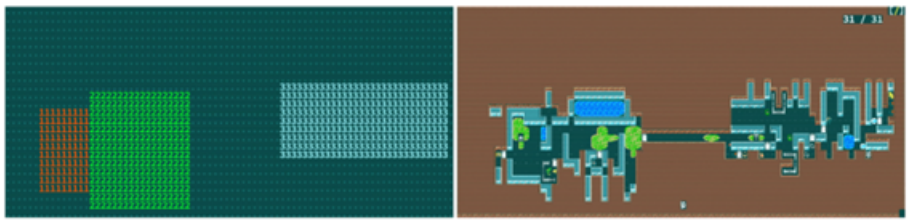


Figure 2.28: Example of map generated by WFC in Caves of Cud [Freehold Games, 2015]. On the left, we see the subareas of the map. For each of them, the WFC algorithm will be run with a different setting. On the right is the final map. Picture found in [Boris, 2020].

regions that recalculate which patterns are still allowed in them.
At the end of the process, an output locally similar to the given input is returned [Gumin, 2016].

WFC is a fairly recent algorithm, nonetheless, some works that expand on this base have since been published. Isaac Karth and Adam M Smith have a more in-depth academic discussion about WFC in [Karth, 2017].
Sandu et al. describe and analyze some technical implementations of integrating generalized design constraints into the WFC algorithm [Sandhu, 2019]. They experiment with: Weighted Choice for the tiles at the collapsing step of the main loop, to achieve an even better resemblance with the input; Adding a non-local design constraint that allows designers to define dependencies between pairs of tiles or items on the map, minimum and maximum distances between them, and the possibility of a number limit for each; Weight Recalculation gives the designers the tools to give tiles new weights in certain areas of the output; and Area Propagation, which takes away tile choices to constrict the generative space, giving designers a way to create subareas.
Gaisbauer, in his doctoral thesis [Gaisbauer, 2021], creates immersive populated cities for virtual reality, using WFC as the main tool for generating the video game levels. WFC is used in this case as a mixed-initiative tool, allowing for the design of a lot of similar levels quicker. To this end, two main improvements were made to the base algorithm, Pattern Weights and Freeze Patterns – very similar to the same enhancements seen in Sandu et

al. 's work [Sandhu, 2019]. Gaisbauer's project is another great example of WFC being capable of working with more than tilemaps, with 3D game objects being used in this particular case.

2.6 Evaluation of PCG Methods

Evaluating content generators is an important and difficult task. They must be judged on their ability to achieve the desired goals of the designer, however doing this can be tricky. It's easy to build a generator, but hard to build one that produces valuable and novel content [Shaker et al., 2016].
The stochastic nature of both the algorithms at the heat of the PCG methods and of the players are hard to control and difficult the evaluation. Content quality might be affected by the non-deterministic behaviors of the algorithm in an unpredicted way, and users respond to content in different ways depending on their personalities, gameplay aims, styles, and goals [Yannakakis and Togelius, 2018].
By evaluating a content generator, we gain a better understanding of its capabilities that we wouldn't have from seeing individual instances of their output; we can make guarantees about the generated content; turns iterating upon it easier, since we understand better its strengths and weaknesses; and makes the comparison with other generators possible [Shaker et al., 2016].
There are two main approaches to evaluation: top-down and bottom-up. However, a hybrid approach involving both can provide a more holistic evaluation and understanding [Shaker et al., 2016].
Top-down methods, like the visualization of expressive measures, have designers directly observe the properties of the content generator. They involve the computation of meaningful metrics and of ways to visualize them. Metric design is an important step that will affect the quality of the evaluation and is usually done in an ad-hoc manner [Yannakakis and Togelius, 2018]. Expressive range visualization is performed by choosing metrics by which the content can be evaluated and using them as axes to define the space of possible content. A large number of instances of content are generated and evaluated according to the metrics and plotted on a heatmap, which can reveal biases in the generator [Shaker et al., 2016]. Generating multiple heatmaps for different parameter configurations can show us how controllable the generator is. Smith et al. [Smith et al., 2011b], use expressive range visualization to evaluate their rhythm-based level generator, *Launchpad*. Using varying rhythm types as input for the generator, they can better understand how leniency and linearity are affected by these changes, Figure 2.29.

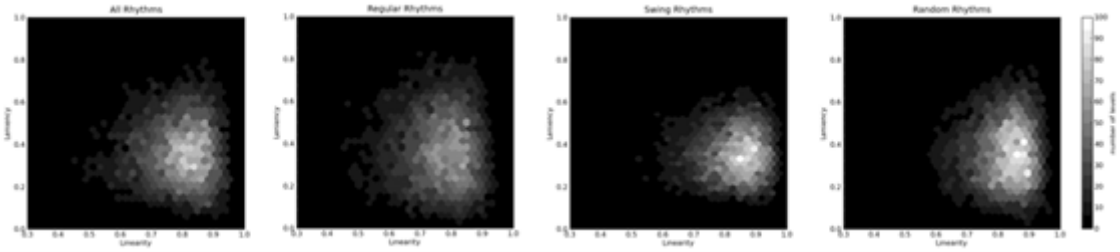


Figure 2.29: Expressive range of *Launchpad* levels using varying rhythm types. From the left to the right: all rhythms, only regular-type rhythms, only swing-type rhythms, only random-type rhythms. Image reproduced from [Smith et al., 2011b].

Bottom-up methods, a more indirect approach to evaluation, consist of measuring some-

thing about the content from reports given either by human players or AI agents as feedback [Yannakakis and Togelius, 2018][Shaker et al., 2016]. Playtesting is especially useful for the evaluation of aspects of content that cannot be measured objectively. User studies can involve a small number of dedicated players that play several instances of the content, or, in a crowd-sourced approach, provide large quantities of data to machine learn content evaluation function [Yannakakis and Togelius, 2018]. Boyang et al. [Li et al., 2012], for example, generate narratives with the help of script-like narrative knowledge learned from crowdsourcing, taking advantage of the lifetime of experience explaining stories of several people.
An alternative to using human players is using AI agents to playtest the content. It is cheaper, quicker, and can be done in much bigger numbers than human playtesting. The AI simulates the content, testing the potential of the generator across several metrics defined by the designer, and returning values about its quality [Yannakakis and Togelius, 2018].

2.7 Chapter Summary

To gain a better understanding of what procedural content generation is in the field of video games, in this chapter several topics were approached.

First, a definition of what procedural generation, as well as an explanation of what game content can entail, were given. Procedural content generation was also contextualized for the field of video game content, as it brings a unique challenge for the designers since the generated content needs to be playable.

Next, a brief history of PCG within the videogame field was detailed. From its first usage in the early 1980s as a way to combat storage limitations to modern times, where it is used for less critical aspects in AAA games, and as the main tool that empowers independent developers and small studios.

A more in-depth look at why PCG is used is had in the third subsection. As a way to combat the increasing number of required person-months that go into the development of successful commercial games, democratizing the production of games by offering reliable and accessible ways to make better games in less time. This enables independent studios and even hobbyists to be able to compete in the market and offers a way to combat the cultural stagnation of the field. PCG can also be used as a tool to empower and streamline the work process of designers and artists. It also allows for the creation of new types of games, such as ones that use it to generate never-ending content, or that in conjunction with some type of player modeling create a player-adaptive experience.

In this subsection, two recent PCG Taxonomies are analyzed, as a way to understand the field from two different perspectives, contextualizing the many PCG methods that are explored in the following subsection.

Togelius et al.'s taxonomy establishes 7 dimensions along which each PCG problem can be placed. Two of them relate to the generated game content: Online vs. Offline and Necessary vs. Optional. Three of the 7 dimensions regard the generation methods: Non-Controllable vs. Controllable, Stochastic vs. Deterministic, and Constructive vs. Generate-and-test. The last two dimensions characterize the possible roles that PCG can take in the design process: Autonomous vs. Mixed-Initiative and Experience-Agnostic vs. Experience-Driven.

Craveirinha et al.'s taxonomy takes the approach of describing each PCG solution as the joint effort of three actors: The human designer, the computer, and the human player. Each

of the actors can fulfill the role of the generator and/or evaluator of the game content. The taxonomy goes further into detail about the possibilities for each actor within each role.

Procedural content generation can be achieved by many means. Some solutions are better at answering a particular type of PCG problem, and sometimes two or more are used complementary. This subsection is dedicated to exploring some of the more noteworthy ones.

Search-based PCG methods operate under the assumption that within a space of possible solutions exists a good enough one, and that through iterations and modifications of a solution or population of possible solutions from that space, keeping the good changes and leaving behind the bad ones, eventually we will find a suitable one. In this sub-sub section, search algorithms, representation of the solutions, their evaluation, and when this type of method is useful are discussed.

Solver-based PCG methods also see the generation of content as a search in a space of possible solutions. However, they find solutions in one pass, instead of iteratively, using constraint solvers borrowing from the ones used in logic programming. A great focus is given to one of the better established of these approaches, Answer Set Programming.

Grammar-based PCG methods use formal grammars, fundamental structures in computer science, as their basis. Usually, formal grammars are composed of a set of production rules that describe how to form strings according to an associated syntax. In the case of generating game content, these strings are later mapped into the type of content needed. One of the more common uses is to use the generated string as a set of instructions to "draw" an organic structure, such as a tree. Instead of strings, it is also possible to have formal grammars of graphs or even shapes, useful for the generation of level structures or the space where they take place.

Cellular automata-based PCG methods, much like their name indicates, use cellular automata, a discrete model of computation also studied in computer science, as well as physics, complexity science, and biology. Its main use is the modeling of biological and physical phenomena. It is comprised of a grid of cells that can be in one of many states and the rules by which these cell's states change over time, according to the states of their neighboring cells. In the interest of generating game content, the same process can be applied to simulate complex and dynamic systems and the generation of organic-looking environments.

Fractal and noise algorithms are probably the most well known examples of PCG, much because of their frequent usage. They are used in the generation of scale-invariant textures – since behind the value of each pixel there is a calculation, their resolution is infinite. The main utility of these textures is to add variation to surfaces or even to simulate dynamic effects. Terrain generation is another good example of the usefulness of these algorithms, since the generated textures can be mapped into height maps.

Machine learning-based PCG methods are an emerging direction in PCG research. Content generators are trained on already existing content to produce similar results. Such techniques have been used for a long time outside of video games. However, a big challenge reveals itself when implementing them for the generation of game content: the final result must be playable. Beyond generation, machine learning is also used in other PCG methods as a way to represent and evaluate content.

The actions of one or multiple computational agents can be used in the content generation process. With agents, we refer to computer entities that exist within an environment, receive information from, and can manipulate it. They usually follow a simple set of

rules related to their immediate vicinity rather than the content to be generated as a whole. Agents are mainly used in terrain generation. Their stochastic nature leads to the generation of content with a less structured feel to it.

Wave function collapse is a local constraint-based procedural generation algorithm utilized to generate bitmap patterns using an input sample bitmap pattern. The output generated is in the style of given examples. This recent algorithm has mainly been used as a way to generate game maps. Its consistency allows for a large output based on some examples. It also shows potential as a mixed-initiative tool for designers.

In the last subsection of this chapter, the topic of the evaluation of content generators is overviewed. Thanks to the stochastic nature of both the algorithms and of the players, evaluation is a difficult but important task. Two approaches to evaluation are discussed: Top-down methods, which have designers directly observe the properties of the content generator; and Bottom-up methods, which consist of measuring something about the content from reports given either by human players or AI agents as feedback.

This page is intentionally left blank.

Chapter 3

Design Proposal

The proposed platform defines an architecture with the objective of supporting the creation of game scenarios for the testing and exploration of PCG approaches. It uses a simulation of an agent society to model these game scenarios. These computational agents are components that receive information from their environment and can affect it through actions dictated by their internal rules. To define an agent is to define what it receives as information, its internal state and rules, and what actions it can execute. Their world is a discrete bi-dimensional grid and the patterns that emerge from their interactions with each other is what constitutes the mechanics of the modeled game scenario.

The architecture of the platform is divided into three main components: setup, update, and visualization. Each of them interacts with the agent grid at the center of the system in a specific way and is associated with the steps present in the loop of the simulation. The setup component initializes the grid, the update component determines how the update cycle of the agents on the grid is realized, and the visualization component is responsible for how the agents are represented to the user. From the initial design stage of the architecture, it was decided to make this division of the process in these three components, as it allows for different combinations of initial setup, update loop, and visualization of the same game scenario. This modulation of the simulation process allows for greater control of the user and an easier development cycle.

A fourth component exists to coordinate the execution of the three principal components. Although it does not affect the grid directly, it is in this main script that defines which versions of the setup, update, and visualization components are used and when. In the final version of the platform, this script controls the state of the system, allowing the user the possibility of stopping the simulation and editing the grid through the same user interface used to visualize the running simulation.

Instead of using the setup component to generate the first iteration of the grid, or in addition to it, the platform allows the user to edit the composition of the agent society at any moment of the simulation. The editor works in a similar manner to a simple drawing software, using the grid as a canvas and the different types of agents as different colored inks. Additionally, two editorial tools are available to the user, the selection tool and a co-creation tool that uses the wave function collapse algorithm to fill the grid in the same style as the area that the user selected.

Two case studies were used as benchmarks during the development, Conway's Game of Life [Games, 1970] and a turn-based adaptation of the arcade game Bomberman [Hudson Soft, 1983].

3.1 Objectives and Requirements

Before development can start, a list of objectives and requirements has been elaborated. Through it, a first architecture of the system can be defined and only then developed. With this intent several goals were outlined for the proposed project:

- **Modeling of game scenarios as a complex adaptive system**. In [Holland, 2012] John Holland describes complex adaptive systems as intricate signal/boundary hierarchies. Boundaries create delimitations of components (or systems) - be they physical or not - and within them can exist more boundaries that represent subcomponents (or sub-systems) of the whole, much like a society is composed of individuals that themselves contain subcomponents. Interactions between these boundaries are made through signals that are sent from one boundary to another in the same level of hierarchy or the one immediately above or below. The basic idea for the complex adaptive system that the platform uses started from this paradigm of boundaries and signals and evolved into a more simplified version. The final version models this system as a grid of agent components and signals as modifications that can occur or be effected on the grid.
- **Modeling of the agents as components of the game system**. As was established in the previous goal, in John Holland's model, agents themselves can be seen as subsystems within the bigger system of the game scenario, also delimited by a boundary that houses components that communicate with each other through signals. In the final version, the behavioral logic within these agents can be seen as the modeling of the previously referred subsystems.
- **Support the training of Machine Learning agents.** The developed platform would be used by Diogo Alves [Alves, 2021], whose dissertation project pertained to the modeling of synthetic players as testing devices for generative game content. For such collaboration to be possible, the system was developed with this integration in mind, namely by enabling agents to have a sensing interface over part of the grid and accommodating agent actions on it.
- **Modeling of an editing system and GUI**. This platform is meant to be used by human designers and, ideally, to be useful for them. The objective is to support the creation of game scenarios. For this, an editing system of the complex adaptive system that is used to model the game scenario had to be modeled, along with a GUI.
- **Modeling of a co-creation editorial tool**. Related to the last objective, a co-creation tool to aid the human designer was also designed. This tool enabled testing of the wave function collapse algorithm as a proof-of-concept operation to propagate design styles over the game space. It fills the grid in the same style (using the same patterns) as the area that the user selected. With this tool, a designer can quickly generate spaces to be tested with synthetic players.

From the goals stated, several project requirements can be formulated:

- **Model of the game world**. The structure that will contain all other structures needs to be defined for the complex system to exist.
- **Model of the agents**. There needs to be a way to define various types of agents and how they may interact with the grid and each other. Agents differ from each

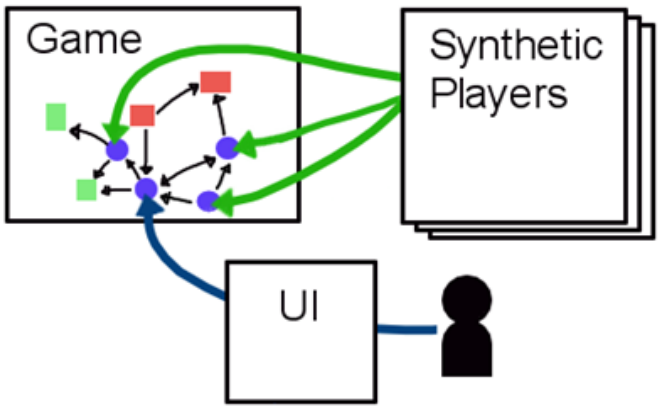


Figure 3.1: A representation of the overall system components and interactions. The game is a complex adaptive system, having as some of its components agents that can be controlled both by synthetic players and by human players using a GUI. The interactions between the components within the complex system result in emerging patterns that characterize the game scenario.

other by the way they interpret information and what type of actions they perform. With this in mind, a sensing method is necessary for the agents to get information from their environment, as well as a method for the agents to act upon that same environment.

- **Model of a setup system**. This refers to the component of the system responsible for initializing the game world. There needs to be a starting point for the simulation.
- **Model of a visualization system**. To be usable by humans, the platform requires a visual interface component. The visualization system translates the agent grid state in the code into a visual representation drawn on the screen and updates it when needed.
- **Model of an update system**. For the simulation to develop there needs to be a component that determines how the update cycle of the agents on the grid occurs. The complex system at the core of the platform is composed of multiple agents that need to be coordinated. There is no central component that determines exactly what happens in the simulation - the emergent behavior generated by the interactions of the game agents is what drives the simulation forward. These interactions occur by updating the states of the agents. The update component is responsible for synchronizing these updates on each update cycle.
- **Model of a coordination system**. While the update system coordinates an update cycle, there needs to exist a system component above it that coordinates it along with the setup and visualization system. The coordination system organizes the order of operations, determining when the agent grid is initialized/reinitialized, when a new update cycle may start/continue, and when the visual representation is updated.
- **Model of a user and machine learning interface**. For synthetic players to be able to control agents, the system must be able to handle player-controlled agents. This also means that human players shall be able to control agents. For that to be possible, player-controlled agents must also be modeled - agents capable of detecting user input and acting upon it. The update system must also be able to handle the existence of these types of agents.

- **Model of an editing/control interface**. The platform must allow the user to edit the composition of the agent society at any moment of the simulation. For this to be possible, there needs to be an interface that allows the editing of the agent grid and the control of the flow of the simulation. Basic editing of the game world involves the adding and removing of agents from it. A GUI to aid the user in these tasks must also be implemented.
- **Modeling of selection and WFC-based editing tools**. Along with the basic editing referenced in the previous requirement, a WFC-based co-creation editorial tool was defined as one of the objectives of the project. In order to implement such a tool, an additional selection tool is also required.

3.2 Proposed System Architecture

The following subsection describes each of the components of the base proposed architecture. The editor system was developed after the base platform had been prototyped with the game engine Unity Haas [2014]. As such, it will be introduced in a later subsection of this chapter - *subsection 3.5*.

3.2.1 Game Agent Model

The agent is the smallest unit of the system. For this project, they are given the name of game agents, and from now on they will be referred to as such. Everything in the game scenario can be represented by a game agent. They can be used to represent a dynamic actor within the scenario, a static object, or even a unit of space. This generalization requires a base class broad enough to allow polymorphism or specialization for all these possibilities.

An abstract class called ***GameAgent*** was created in Unity with the programing language C#. This class has the following attributes:

- *typeName* - an identifying string for the GameAgent type;
- *position* - a vector with two integers, representing the coordinates of the GameAgent on the agent grid;
- *relative_sensors* - a list of vectors with two integers, representing the positions relative to the GameAgent's positions to be used as input;
- *constan_sensors* - a list of vectors with two integers, representing the fixed positions of the agent grid to be used as input;
- *states* - a list of integers of variable length that represent the internal states of the GameAgent;
- *exists* - a boolean that indicates whether the GameAgent is in the grid;
- *colliderTypes* - a list of strings indicating what other types of GameAgent the agent in question cannot coexist in the same agent grid position with;
- *updateInterface* - a reference to the update component in usage;
- *creator* - a reference to the GameAgent that created the GameAgent in question;

The relative_sensors and constant_sensors attributes represent the positions on the agent grid that the GameAgent will use as input when updating. The GameAgent will search in these positions for what other GameAgents are located in them, using their typeName or states as variables in its updating rules.
The states attribute may also be used as an input for the GameAgent's update rules and/or changed according to them.
The exists attribute is used as a measure of redundancy to guarantee that a GameAgent removed from the grid isn't accidently updated.
The colliderTypes attribute is used by movement methods that will shortly be detailed.
The updateInterface is used by specific methods to affect the update component. This gives the GameAgent the possibility of affecting the update cycle.
The creator attribute was implemented mainly as a necessity for the training of synthetic players by Diogo Alves Alves [2021], although it could give more possibilities to the GameAgent that could be explored in future work.

The GameAgent abstract class has the following methods:

- *UpdateAgent*
  - It receives as parameters a reference to the Grid (g), an integer (step_stage), and a pseudo-random number generator (prng);
  - Each class that inherits GameAgent must implement this method. In it, an action to be performed is chosen according to the input and update rules of the agent. As input, the update rules may use the state of the agent grid g as seen in the GameAgent's sensors, or the GameAgent's states. An action may be modifying the agent grid g by creating, modifying or deleting other GameAgents or itself. The usage of a pseudo-random number generator allows for the results to be replicable if the same number seed is used.
- *Epitaph*
  - It receives as parameters a reference to the Grid (g), an integer (step_stage), and a pseudo-random number generator (prng);
  - Each class that inherits GameAgent must implement this method. It is executed on the elimination of the Agent from the agent grid g. The elimination of some GameAgents may affect the grid beyond just their removal.
- *GetSensors*
  - It receives as parameters a reference to the Grid (g);
  - Returns a list of all GameAgents contained in the sensor positions on the agent grid g as referenced in the relative_sensores and constant_sensors attributes. For relative_sensors, if the position is outside the dimensions of the agent grid, it will wrap around as if each extreme of the game world is connected with its opposite.
- *PutAgentOnGrid*
  - It receives as parameters a reference to the Grid (g), a vector with two integers (newAgentPos), and a GameAgent (newAgent);
  - Adds the given newAgent to the agent grid g on the given position newAgentPos, returning true if it was successful and false otherwise.
- *RemoveAgentOffGrid*

  - It receives as parameters a reference to the Grid (g), and a GameAgent (agentToRemove);
  - Removes the given agentToRemove from its position on the agent grid g without turning its exists attribute to false.

- *EliminateAgent*
  - It receives as parameters a reference to the Grid (g), and a GameAgent (agentToEliminate);
  - Eliminates the given agentToEliminate by calling the RemoveAgentOffGrid function and turning its exists attribute to false.

- *MoveAgent*
  - It receives as parameters a reference to the Grid (g), a vector with two integers (newAgentPos), and a GameAgent (agentToMove);
  - Moves the given agentToMove from its position on the agent grid g to the given position newAgentPos. Returns true if the operation was successful and false otherwise.

The UpdateAgent method has a parameter step_stage relevant to a characteristic of the update component, and will be discussed further when describing its model later. It allows for the existence of update cycles with several steps, where GameAgents may be called to update again, with a different set of update rules.
The RemoveAgentOffGrid method does not change the exists attribute of the GameAgent to false because it is also used in the MoveAgent method, removing the GameAgent from its previous location. The EliminateAgent is the method that must be called to effectively eliminate the GameAgent from the simulation.

3.2.2 Game Agent Player Model

For GameAgents that are meant to receive player input - be it a human or a synthetic player - an abstract class that inherits GameAgent called GameAgentPlayer was created. The detection of human player input is implemented with the help of coroutines.

In addition to the ones of GameAgent, ***GameAgentPlayer*** has the following additional attributes:

- *updated* - A boolean that indicates whether the GameAgentPlayer has finished its update cycle;
- *input* - A reference to the KeyCode last inputted by the player.

In addition to the ones of GameAgent, ***GameAgentPlayer*** has the following additional methods:

- *Logic*
  - It receives as parameters a reference to the Grid (g), an integer (step_stage), and a pseudo-random number generator (prng);

  - Each class that inherits GameAgentPlayer must implement this method. It's an IEnumerator function used for the coroutine of the GameAgentPlayer logic. Will start WaitForKeyDown coroutines whenever the update rules require human player input. Will finish by turning the updated attribute to true.

- *WaitForKeyDown*
  - It receives as parameters an array of KeyCodes (codes);
  - It's an IEnumerator function used for coroutines waiting for the player's input. All the given codes are checked. If the player presses a key corresponding to one of the given codes, the input attribute is updated and the function will finish.

GameAgentPlayer also overrides the UpdateAgent method of GameAgent. When called, and only if the GameAgentPlayer has updated at true, will start the Logic coroutine. The update logic that would have been in UpdateAgent is transferred into Logic.

3.2.3 Grid Model

The agent grid is the container of all the GameAgents on the simulation and also represents the state of the complex system at any given moment. The class Grid was created to contain both this agent grid and a grid of Unity's GameObjects that are used to represent the agent grid visually in this implementation of the proposed architecture.

The ***Grid*** class has the following attributes:

- *agentGrid* - A matrix of lists of GameAgents;
- *objectGrid* - A matrix of GameObjects;
- *container* - GameObject parent to all the GameObjects contained in objectGrid;
- *width* - Width of both agentGrid and objectGrid;
- *height* - Height of both agentGrid and objectGrid;
- *cellSize* - Cell size used for the objectGrid;
- *agentTypes*- A array of strings containing all the types of GameAgents on the simulation;
- *updated* - A boolean that indicates if the objectGrid may be updated with new information from the agentGrid;
- *simOver* - A boolean that indicates if the simulation has finished.

The agentGrid attribute is a matrix of lists of GameAgents to allow the existence of multiple GameAgents in a single position. Each position of the matrix may have an empty list or a list with one or more GameAgents. Updates to the agentGrid are handled by the update component.
The objectGrid can be understood as the visual representation of the agentGrid. Each GameObject represents one cell of the agentGrid, since only one GameAgent is visually represented for each cell. Priorities in representation alongside updates to the objectGrid are handled by the visualization component. The container attribute exists for ease of positioning of the objectGrid on the screen.

The agentTypes attribute contains the same strings stored in the GameAgent's typeName attribute.
The update component utilizes the update attribute of the Grid to coordinate with the visualization component.
The simOver attribute may be used by the update component for game scenarios that have an end condition.

The ***Grid*** class has the following methods:

- *Grid*
  - It receives as parameters an integer (width), an integer (height), a float (cellSize), a matrix of lists of GameAgents (agentGrid), and a array of strings (agentTypes);
  - The Grid constructor. It initiates the objectGrid.
- *GetWorldPosition*
  - It receives as parameters an integer (x) and an integer (y);
  - It maps and returns the game world position equivalent to the given x and y on the grid.
- *GetXY*
  - It receives as parameters a vector with three floats (worldPosition);
  - It maps and returns the grid position equivalent to the given game world position.
- *PosInGrid*
  - It receives as parameters an integer (x) and an integer (y);
  - Returns true if the given x and y coordinates are within the boundaries of the Grid.
- *ConvertAgentGrid*
  - Returns a matrix of lists of integers. This matrix is a conversion of the agentGrid in which each GameAgent is replaced with the integer corresponding to the index of its typeName attribute in the agentTypes array of strings.
- *GetAgentTypeInt*
  - It receives as parameters a string(type);
  - Returns the index of type in the agentTypes array of strings.
- *DeleteContainer*
  - It destroys all of the objectGrid GameObjects and the container.

The GetWorldPosition, GetXY, and PosInGrid methods are used to convert positions between the screen space and the agentGrid.
The ConvertAgentGrid method was created to facilitate the integration of synthetic players Alves [2021], but was also useful as a debug tool.

3.2.4 Setup Component Model

All three main components of the system - setup, update and visualization - are implemented in Unity as interfaces. Interfaces are a set of methods and attributes that a target class must implement. Several variations of these components were implemented - more about this in the subsections about the proofs of concept 3.3 and 3.4. Having these classes inherit from Interfaces allows for easier development using the Unity editor - more about this in the sub-subsection3.2.7, about the simulation coordinator component model.

The setup Interface ***ISetup*** was created to characterize a setup component of the system. These components determine how the agent grid is initialized before the start of the simulation.

A class that inherits from ISetup must implement the following methods:

- *SetupGrid*
  - It receives as parameters a pseudo-random number generator (prng), an integer (width), and an integer (height);
  - It sets up a new Grid object, ready for the start of the simulation, and returns it. It is responsible for the creation and dimensions of the Grid, as well as the initial distribution of agents in the agentGrid. The usage of a pseudo-random number generator allows for the results to be replicable if the same number seed is used.
- *ReturnSet*
  - It returns a string with the name of the "set" to which the setup component belongs.
- *ReturnName*
  - It returns a string with the name of the setup component

The ReturnSet and ReturnName methods allow for better organization in the final prototype developed for this project, where the user is free to choose between a combination of several components. Specific IUpdate and IVisualize Interfaces may be required in order to work with the same types of agents - these interfaces belong to the same "set".

3.2.5 Update Component Model

The update Interface ***IUpdate*** was created to characterize an update component of the system. These components determine how the game agents are updated in an update cycle, moving the simulation forward.

A class that inherits from IUpdate must implement the following methods:

- *SetupSimulation*
  - It receives as parameters a reference to the Grid (g) and a pseudo-random number generator (prng);

  - It initialises possible variables that may be used during a simulation. These variables may be dependent on the contents of the agentGrid. In the case it is required to restart the simulation, this method is called once again. The pseudo-random number generator prng is used when randomization is required in this step.

- *UpdateGrid*
  - It receives as parameters a reference to the Grid (g), a pseudo-random number generator (prng), a float (delay), and a boolean (debug);
  - It describes how the game agents of the agent grid are updated. It is responsible for the order in which these updates occur, as well as how to handle game agents that require player input. It also controls when the attribute updated of the Grid g is turned to true, indicating that the visualization component may update the objectGrid. The usage of a pseudo-random number generator allows for the results to be replicable if the same number seed is used. With human visualization in mind, the method will only initiate a new update cycle once delay time has passed since the last one. The boolean debug indicates if the start of a new update cycle is controlled by the press of the mouse button.

- *AgentCall*
  - It receives as parameters a reference to the Grid (g), a pseudo-random number generator (prng), and a GameAgent (agent);
  - It is used as a way for the GameAgents to communicate with the update component, affecting the flow of the simulation.

- *ReturnSet*
  - It returns an array of strings with the names of the "sets" to which the update component belongs.

An update component may or may not be designed for specific types of agent combinations. Since on the C# code all the GameAgents on the agent grid are treated as their abstract class, there is no need for specifying agent types. This allows an update component to belong to several "sets" - to work with several types of ISetup and IVisualize Interfaces.

3.2.6 Visualization Component Model

The visualization Interface ***IVisualize*** was created to characterize a visualization component of the system. These components determine how the objectGrid attribute of the Grid is updated according to the state of its agentGrid attribute.

A class that inherits from IVisualize must implement the following methods:

- *VisualizeGrid*
  - It receives as parameters a reference to the Grid (g);
  - It describes how the GameObjects contained in the objectGrid attribute of the Grid g are updated in order to indicate what game agent types are positioned in that same position on the agentGrid. The SpriteRenderer components of the GameObjects contained in the objectGrid are modified for this effect. This method is also responsible for defining priorities of representation for different agent types.

- *ReturnSet*
  - It returns a string with the name of the "set" to which the visualization component belongs.

Specific IUpdate and ISetup Interfaces may be required in order to work with the same types of agents - these interfaces belong to the same "set".

3.2.7 Simulation Coordinator Component Model

The simulation coordinator component is the main script of the system, and the one responsible for connecting all other components. A base model of the script will be presented now, while the final version which incorporates the PCG Scenario Editor will be presented in its own subsection 3.5.

The main script contains the following attributes:

- *seed* - It's the integer used to initiate the pseudo-random number generator;
- *useRandomSeed* - A boolean that determines if the seed or a random value will be used as the seed used to initiate the pseudo-random number generator;
- *numberOfEpisodes* - An integer indicating the number of times the simulation will be run. Default value 1;
- *episodeNumber* - An integer indicating the number of times the simulation has been run;
- *prng* - A reference to the pseudo-random number generator;
- *grid* - A reference to the Grid object used in the simulation;
- *SetupInterface* - A reference to a setup component;
- *UpdateInterface* - A reference to a update component;
- *VisualizeInterface* - A reference to a visualization component;

The numberOfEpisodes and episodeNumber attributes were mainly used for the purposes of supporting the training of Machine Learning GameAgents.
The modularizing of the system into several components and the **SetupInterface**, **UpdateInterface** and **VisualizeInterface** attributes allow for easy substitution of components of the same type from the Unity editor.

The main script contains the following methods:

- *Start*
  - It initializes the Interface attributes, initializes the pseudo-random number generator, uses the SetupInterface to generate the grid, sets up the simulation with the UpdateInterface and updates the visuals according with the initial state of the agent grid with the VisualizeInterface.
- *Update*

  - It updates the grid with the UpdateInterface, and determines if the VisualizeGrid method of the VisualizeInterface should be called. If the simulation is over (the simOverattribute of the grid is true) and there still is a number of episodes to run the simulation, it will restart the grid and the simulation.

The Start method is called automatically by Unity on the frame when the script is enabled just before the Update method is called the first time - Update method that is called every frame after that.

3.3 Design of the Game of Life Proof of Concept

The initial prototype used Conway's Game of Life [Games, 1970] as a case study as a means to define the first iteration of the architecture of the system. The Game of Life is one of the most famous examples of a Cellular Automata, as seen in section 2.5.4. The basic idea behind the complex adaptive system model at the core of the platform can be seen as a type of cellular automata - a discrete two dimensional grid of elements that change states according with what's around them. Aiming to replicate the simple three-rules-based Conway's Game of Life seemed like an excellent choice to guarantee that the proposed architecture of the system was behaving as it should.

Conway's Game of Life takes place in a two dimensional orthogonal grid of cells that can be in one of two possible states: alive or dead. The state of each of these cells from one time step to the next depends on their Moore neighborhood of size 1 - their closest eight neighbours. What happens to each cell at each time step can be condensed into three rules:

- Any alive cell with two or three neighbours survives
- Any dead cell with three live neighbours become an alive cell
- All other cells become or stay dead.

The whole system can be seen as the representation of a population, with the first rule representing a sustainable pattern, the second rule representing reproduction, and the third rule death by underpopulation or overpopulation.

3.3.1 LifeAgent

For a simple Conway's Game of Life recreation, a single agent class LifeAgent was developed. This class implements a constructor method:

- *LifeAgent*
  - It receives as parameters a list of integers (states), an integer (x), and an integer (y);
  - It initializes the GameAgent attributes - states, position, typeName, relative_sensors, and constant_sensors.

This GameAgent class uses a list of two integers to represent its internal state - one for the number of alive neighbors and another to represent if itself is alive or dead. The LifeAgent

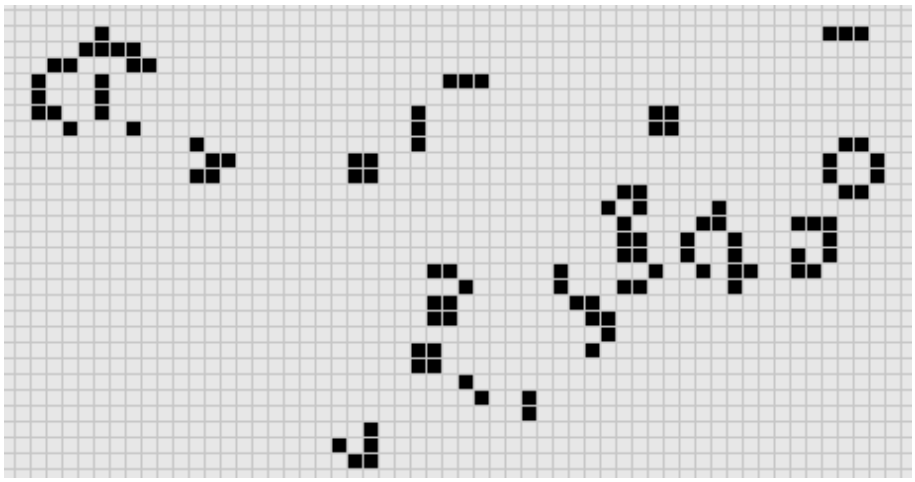


Figure 3.2: An example of a time step in Conway's Game of Life [Games, 1970], where the black cells represent the alive state, and white cells the state of dead cells.

class does not use any constant_sensors but has a relative_sensors list with eight positions - one for each neighbor.

LifeAgent implements its own version of the UpdateAgent method. It will be touched upon in greater detail when the update component developed for this game scenario is discussed, but the usage of the UpdateAgent's step_stage parameter is necessary. In each update cycle, UpdateAgent is called twice for each LifeAgent. The first time, step_stage will be equal to 0 and the LifeAgent's states attribute will be modified according to how many of the agent's neighbours have a states attribute that indicates that they are alive. The second time, step_stage will be equal to 1 and the LifeAgents dead or alive status will be modified if need be according to the three rules of Conway's Game of Life by modifying its states attribute.

3.3.2 Game of Life update component

The update component developed for this game scenario, GameOfLifeUpdate, starts its UpdateGrid method by creating a list of all the agents contained in the agent grid and shuffling it, guaranteeing that in each update cycle the order of the agents is always different.

The nature of Conway's Game of Life requires that each update cycle be done in two steps - one to register the current state of the grid, and another to act upon it. Each GameAgent is updated one at a time, not in parallel. If a LiveAgent updated its dead or alive status on the same step that it determines how many of its neighbours are alive, its not-yet-updated neighbour will see a version of the grid different from the one at the beginning of the update cycle. Conway's Game of Life requires its elements to be updated at the same time - the proposed platform runs an asynchronous simulation. To represent this game scenario, the "investigation" and "enactment" phases must be separated.

The list of all the agents created at the beginning of the method is run through, and each GameAgent's UpdateAgent method is called with step_stage at 0. When this cycle ends, a new one begins, running through the same agent list once again, this time calling the UpdateAgent methods with step_stage at 1.

3.3.3 Game of Life setup component

Two setup components were developed for this game scenario, GameOfLifeSetup and GameOfLifeSetup_Glider. GameOfLifeSetup simply initializes an agent grid with a LifeAgent in each position, with an equal possibility of being initialized with the status of alive or dead.
GameOfLifeSetup_Glider was developed to test if one of the most famous emerging structures of Conway's Game of Life, the glider, behaves correctly. Five specific cells of the agent grid are initialized with LifeAgents with the status alive, while all the rest are dead.

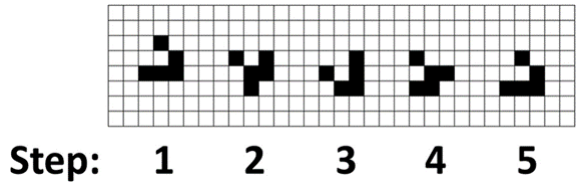


Figure 3.3: A glider pattern in Conway's Game of Life [Games, 1970]. Through the time steps, this structure cycles through four different configurations, having the emergent behaviour of looking like it's moving through the grid.

3.3.4 Game of Life visualization component

A simple visualization component, GameOfLifeVisualize, was developed to allow the visualization of the Game of Life scenario. Its VisualizeGrid method cycles through every GameAgent in the agentGrid and changes the color of the corresponding GameObject to black if alive and to white if dead.

3.3.5 Variant of the Game of Life Proof of Concept with two GameAgent types instead of one

After successfully developing a prototype capable of replicating Conway's Game of Life, a new variant of the scenario was created with two different GameAgent classes instead of just one to demonstrate that the platform worked with multiple types of GameAgent, allowing for greater variety moving forward.

The main difference of this variant is the aforementioned existence of two types of GameAgent, LifeAgentAlive and LifeAgentDead. They work mostly in the exact same way as LifeAgent, only their states attribute only has one integer, indicating how many of their neighbours are alive. GameAgents dead or alive statues are no longer characterized by the states attribute, but by the type of agent they are. This requires the usage of the GameAgents typeName attribute to identify them. In the second step of the update cycle, the change from dead cell to alive cell and vice versa is done through the agent eliminating itself and creating a new GameAgent of the other type on its position.
The update component used is the same one, since it only sees GameAgents as that abstract class. New setup and visualization components were created, since these depend on the type of GameAgent used, however their operations are essentially the same as the components already described, only using two types of GameAgent to represent dead and alive cells instead of only one type with an internal state that dictates it.

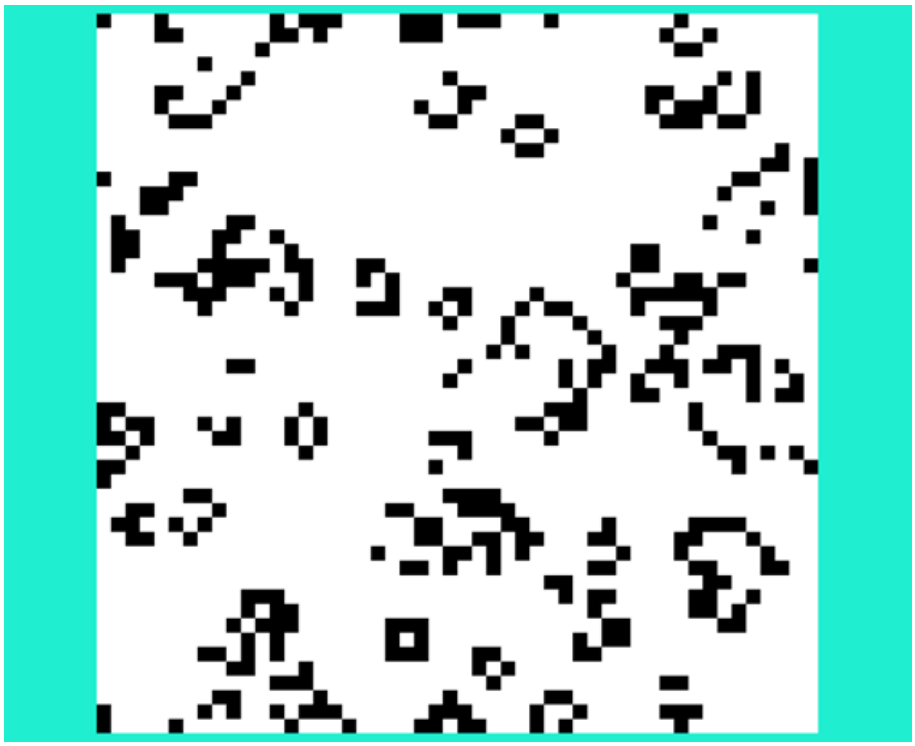


Figure 3.4: Conway's Game of Life [Games, 1970] was replicated successfully on the platform.

3.4 Design of the Bomberman Proof of Concept

After Conway's Game of Life, Bomberman Hudson Soft [1983], was chosen as a case study. Bomberman is a classic arcade game with simple rules that involve many more moving pieces than the Game of Life, being a good step forward to demonstrate the capabilities of the architecture to enable concurrent changes in the agent space composing the grid and other interacting elements, enabling us to represent a game with run-time generated components.
In Bomberman, the player controls the titular main character, moving within a world akin to a grid of indestructible walls, destructible blocks or other items, and empty spaces where the player can move. The player may place a bomb on the map that explodes in the shape of a cross, eliminating the destructible items, other bomberman controlled by other players or the computer and even the player itself. The objective of the game is to be the last bomberman standing. Bomberman games can also include power ups that give temporary or permanent benefits, but for the version of the game that we choose to replicate it sufficed to demo that simpler one just described.
For the implementation of this case study there needed to be a way for a player to control an agent; an end state for the simulation; the implementation of several types of GameAgents: AgentPlayerBomberman, AgentBomberman, AgentBomb, AgentFire, AgentStrongWall, and AgentWeakWall; and the implementation of setup, update and visualization components to support it all.

Figure 3.5: Example of a Bomberman Hudson Soft [1983] game in action.

3.4.1 Bomberman GameAgents

Starting with the easiest types - the static walls. AgentWeakWall and AgentStrongWall are virtually identical, except for the typeName attribute. Their UpdateAgent method is empty, since they don't affect the simulation in any way aside from being an obstacle that the bombermen agents cannot pass through.

AgentFire is a more complex GameAgent type. They represent the explosion of a bomb in a cell of the grid. It's a GameAgent with built in limited time of existence, its states attribute a single integer representing the number of updates until it expires. It only has one sensor - in its exact position. In the UpdateAgent method, the AgentFire checks its sensores and detects if there are GameAgents which typeNames are contained in its list of flammable types - AgentWeekWall, AgentBomberman, and AgentPlayerBomberman - and eliminate them. At the end of the UpdateAgent method it checks if the states integer has reached zero, and if yes, eliminates itself.

AgentBomb has a states attribute with two integers, the first, much like the AgentFire, dictates how many updates it has before it has to eliminate itself, and the second is the size of the explosion. Its UpdateAgent method only reduces its countdown until explosion, and when it reaches zero the AgentBomb eliminates itself. Its Epithaph method that is called when it is eliminated from the agent grid consists in inserting into the agent grid AgentFires in the pattern of a cross. The pattern is actually a little more complicated than that, with the "arms" of the explosion being blocked by walls.

AgentBomberman is the agent type with the most complex set of update rules, Its states attribute is composed of two integers - one representing the cooldown between bombs and the number of updates until another can be placed on the agent grid. This GameAgent makes use of the colliderTypes attribute, having on it the typeNames of the AgentWeek-

Wall, AgentStrongWall, AgentBomb, AgentPlayerBomberman, and the AgentBomberman itself. Basically, a bomberman cannot pass through walls, bombs or other bombermans, including the player. It has as relative_sensors the von Neumann neighborhood of size one.
Its UpdateAgent method has as a priority to determine if the agent is in danger, in which case it moves in the opposite direction if possible. The secondary priority is to determine if it's possible to place a bomb near an enemy or a breakable wall, otherwise it defaults to walk in a random, not obstructed direction.
The Epitaph method uses the AgentCall method of the updateInterface, sending the own AgentBomberman as a parameter.

AgentPlayerBomberman inherits from the GameAgentPlayer abstract class. Unlike AgentBomberman, it does not make use of the states nor the sensors attributes, since the player is the one that will be the brains of the operation. It, however, uses the same colliderTypes list. The Logic method awaits for the player to press one of the arrow keys or the spacebar through a coroutine. After an input is received, the corresponding action will be performed - movement in one of the cardinal directions for the arrow keys, and planting a bomb for the spacebar.
Just like the AgentBomberman, the Epitaph method of the AgentPlayerBomberman uses the AgentCall method of the updateInterface, sending itself as a parameter.

3.4.2 Bomberman update component

The update component for this game scenario, BombermanUpdate, has the additional challenge of handling GameAgentPlayers. It starts its UpdateGrid method by creating a list of all the agents contained in the agent grid and shuffling it, guaranteeing that in each update cycle the order of the agents is always different. The list of agents is run through, updating each GameAgent. If the updated GameAgent is a GameAgentPlayer, the update cycle is put on pause until this particular GameAgentPlayer's updated attribute is true, meaning that the user input was received, the agent had the opportunity of finishing its update routine, and the update cycle may proceed as usual. At the end of each update cycle, and as soon as a GameAgentPlayer is to be updated, the Grid attribute updated is changed to true, meaning that the visuals will be updated. It would be incomprehensible if the visuals were updated as soon as each GameAgent is updated, as such this only occurs when the cycle has ended and all the GameAgents are updated - the simulation has advanced one time step - or when the player needs to know the current state of the system in order to give an input to a GameAgentPlayer.
The method SetupSimulation searches the initial agentGrid of the Grid for how many bombermen exist in the simulation - either AgentPlayerBomberman or AgentBomberman - to determine the number of players in the simulation.
The method AgentCall that the AgentPlayerBomberman and the AgentBomberman call when they are eliminated from the agent grid, determines if the simulation may end because the current number of players in the simulation has decreased past a predefined threshold.

3.4.3 Bomberman setup component

Two setup components were developed for this game scenario. BombermanSetup fills a percentage of cells of the agent grid with AgentWeakWall, another percentage with AgentStrongWall, and lastly spreads a number of AgentBomberman and one AgentPlayerBomberman through the grid.

BombermanSetup2 opts for a more classical look, recreating the style of map from the original game and placing three AgentBomberman and one AgentPlayerBomberman in each corner.

3.4.4 Bomberman visualization component

BombermanVisualize was developed to allow the visualization of the Bomberman game scenario. Its VisualizeGrid method cycles through every GameAgent in the agentGrid and changes the color of the corresponding GameObject to the one associated with the GameAgent typeName. Black for AgentStrongWall, grey for AgentWeakWall, yellow for AgentFire, red for AgentBomb, cyan for AgentBomberman and dark blue for AgentPlayerBomberman.
Since several GameAgents can exist in the same agent grid position, BombermanVisualize also has a priorityList, an Array of strings, each corresponding to a GameAgent typeName. The lower the index higher the priority, meaning that will be the one chosen to be represented visually when competing with lower priority GameAgents.

Figure 3.6: Bomberman Hudson Soft [1983] replicated successfully on the platform using the random setup.

3.4.5 Exploring variations to the Bomberman base scenario

With the objective of exploring more possibilities with the platform, the Bomberman game scenario was modified with the addition of some new types of GameAgents - AgentBush and AgentBushman - as well as some modifications to existing GameAgent types.

AgentBush generates new instances of AgentBush around itself, blocks the passage of bombermans and is highly inflammable - meaning it has a high chance of spreading AgentFire to adjacent AgentBushes if an AgentFire is in its position on the agent grid. Its states attribute contains two integers, one indicates how many update cycles must pass before the AgentBush can spread, and the other indicates how many update cycles have to pass until the AgentBrush can spread. In its UpdateAgent method the number of update cycles until the next spread is decreased until it reaches zero, at which point a random empty adjacent cell is chosen and a new AgentBush is inserted in it.

AgentBushman behaves much like an AgentBomberman, only instead of placing bombs on

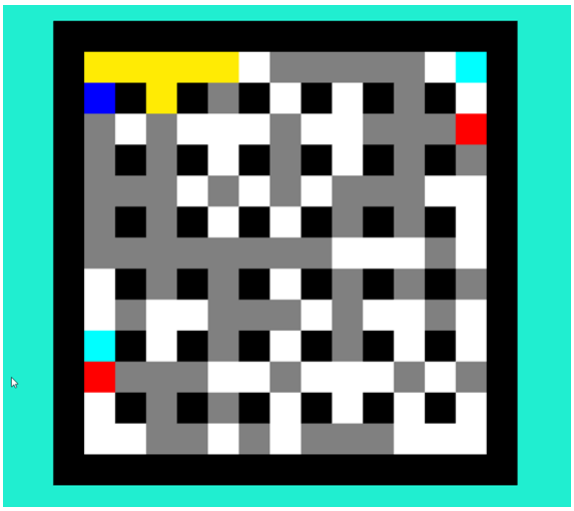


Figure 3.7: Bomberman Hudson Soft [1983] replicated successfully on the platform using the classic setup.

the agent grid, it places AgentBushes. The UpdateAgent method, like in the case of the AgentBomberman, prioritizes keeping the AgentBushman safe, and if no danger is detected placing AgentBushes whenever possible, with walking in a random direction as the default.

AgentFire's UpdateAgent method was modified so that if next to the AgentFire there exists an AgentBush there is a 50

AgentPlayerBomberman was modified to also accept the key "B" as an input to place an AgentBush in addition to all the other already existing options.

The random setup component, BombermanSetup was modified to also fill a percentage of cells of the agent grid with AgentBush and to also spreads a number of AgentBushman through the agent grid

The visualization component was updated for the representation of the new GameAgent types - green for the AgentBush and purple for the AgentBushman.

3.5 Designing a PCG Scenario Editor

Finally, the editor system prototype was designed and developed in Unity after the base platform had been established to be working properly, so that we could better understand which user actions would be useful and interesting to develop. The editor enables the experimentation with diverse PCG agent scenarios. The user is able not only to generate and edit the Grid, but also to control the flow of the simulation - to stop it, accelerate or decelerate, or even control when a new update cycle occurs with the click of the left mouse button, allowing for more detailed step-by-step analysis.
The editor's GUI allows the user to choose the game scenario, Game of Life or Bomberman, and which setup to choose, including a blank one, and to generate a Grid with the given dimensions. The user may stop the simulation at any time and edit the grid. One can modify the Grid by picking an Agent on the GUI and "painting agents" with it, as if it were a brush of a drawing software and the grid the canvas. Two more editorial tools are available, the Selection Tool, which facilitates the modification of large portions of the grid, and the Wave Function Collapse based tool, which can be used as a way to propagate

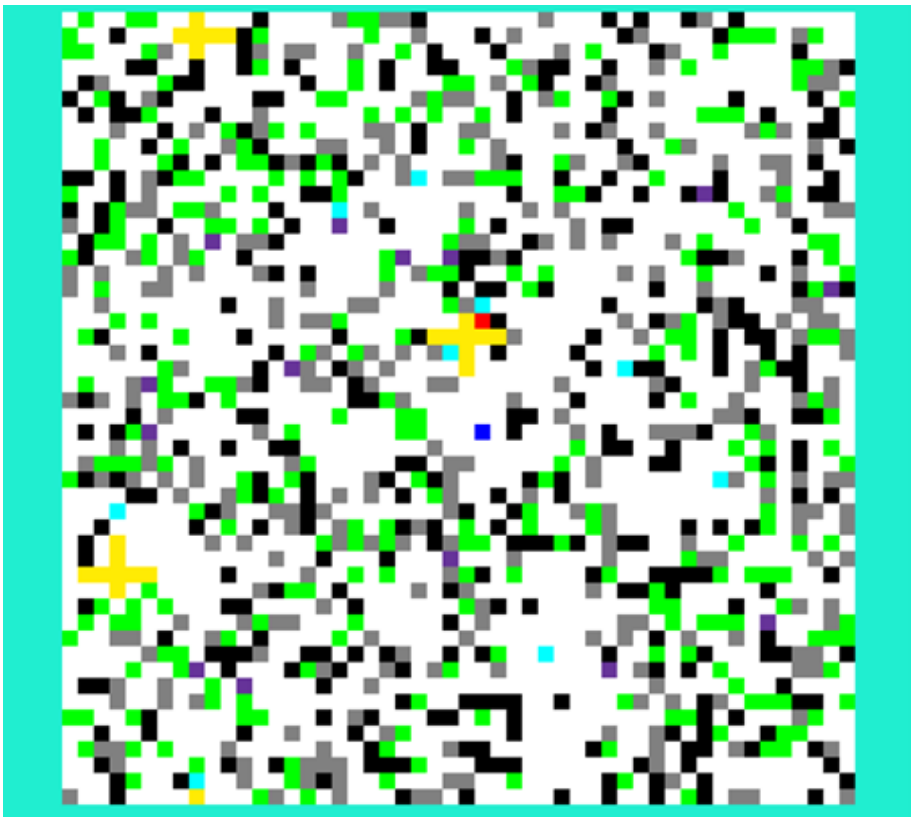


Figure 3.8: Modified Bomberman running on the platform, using the random setup.

scenario style.

This editor interface is a modification of the simulation coordinator component talked about in subsection 3.5. It still is the main script of the system, and the one responsible for connecting all other components.

The GUI is organized as such: roughly two thirds of the screen present the visualization of the grid, while the rest on the far left present the user controls over the simulation. These controls are divided into three subsections: Generate, Edit and Play.
The **Generate** subsection contains two dropdown menus - one for the game scenario selection and another for the setup selection - two input fields for the dimensions of the grid, and a button to generate it.
The **Edit** section is further divided into another three subsections, each one representing one of the tools available to the user. The first Edit subsection contains a list of toggles of all the GameAgent types of the current game scenario and an "Erase" option. When one of these toggles is selected, the user may press the left mouse button on the grid and "paint" in the agent type as if it was a canvas. The second Edit subsection is dedicated to the Selection tool, containing one toggle that is connected to the same toggle group of the agent toggles - meaning that they are mutually exclusive. When it is selected, the user can click on two positions of the grd to form a selection area and select one of the agent types or the eraser to fill it or clear it. The third Edit subsection is the Wave Function Collapse tool, containing a slider for the pattern size used on the WFC algorithm, a couple of toggles for optional processes of rotation and mirroring, and a button to apply. The WFC tool can only be applied when a bid enough area is selected with the Selection tool. The area within the selection is used as input for the WFC algorithm and the tool propagates the style to the rest of the grid.

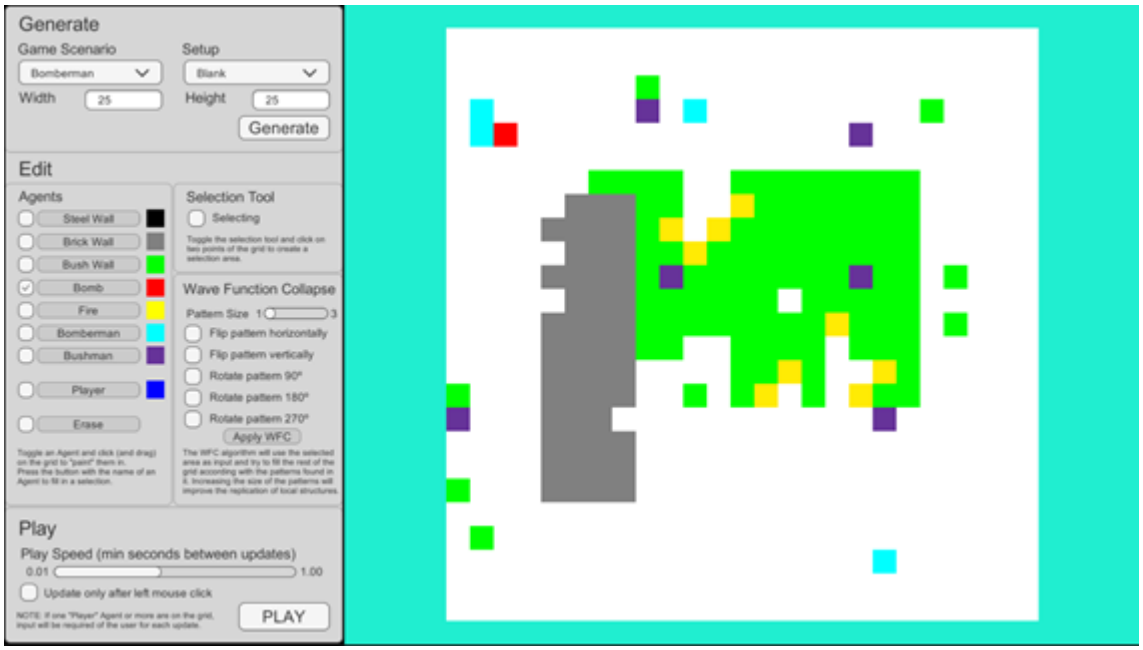


Figure 3.9: The final editor, being used to run a costume simulation of the Bomberman game scenario.

Lastly, the **Play** subsection contains a slider for the simulation play speed, a toggle for the debug mode and a play/edit button to change the state of the system.

The script contains the following attributes:

- *grid* - A reference to the Grid object used in the simulation;
- *gameScenarios* - An array of strings containing the names of the game scenarios;
- *agentTypes* - An Array of Arrays of strings, containing the typeNames of the agents of each game scenario;
- *setups* - An Array of Arrays of strings, containing the names of the setup types for each game scenario;
- *width* - Default width of the Grid;
- *height* - Default height of the Grid;
- *widthMin* - Minimum width of the Grid;
- *heightMin* - Minimum height of the Grid;
- *widthMax*- Maximum width of the Grid;
- *heightMax* - Maximum height of the Grid;
- *prng* - A reference to the pseudo-random number generator;
- *SetupInterface* - A reference to a setup component;
- *UpdateInterface* - A reference to a update component;
- *VisualizeInterface* - A reference to a visualization component;

- *WFCInterface* - A reference to the Interface that implements WFC;
- *selection* - A reference to the partially transparent GameObject that acts as the visualizer of the selection tool;
- *selectionPosition1* - The coordinates of the first point selected with the selection tool;
- *selectionPosition2* - The coordinates of the second point selected with the selection tool;
- *selectingStage* - A integer which value determines the selection stage: at 0 the selection tool is deactivated; at 1 one point of the grid hás been selected; at 2 two points of the grid have been selected;
- *editing* - A boolean that indicates if the simulation is running or if the platform is in the editing state;
- and several references to GUI components.

The main script contains the following methods:

- *Start*
  - It initializes the Interface attributes, initializes the pseudo-random number generator, puts the grid on a default empty state, updates the visuals according with the initial state of the agent grid with the VisualizeInterface, and handles all the necessary work to correctly initialize al the UI components.
- *Update*
  - If the system is in the "play" state, it updates the grid with the correct UpdateInterface for the game scenario selected, and determines if the VisualizeGrid method of the VisualizeInterface should be called. If the system is in the "edit" state, it checks if any key has been pressed and if the mouse is positioned inside the grid, calling the EditGrid function to handle the modifications of the grid.
- *EditGrid*
  - If the left mouse was clicked inside the grid and one of the agent toggles is selected, that GameAgent type is inserted. If the selection tool is selected, it modifies the selection area accordingly.
- *InsertOnGrid*
  - Used to insert new GameAgents into the Grid.
- *GenerateGrid*
  - Called when the "Generate" button is pressed. Generates the Grid according to the specifications made by the user.
- *ResetupGrid*
  - Used by GenerateGrid to handle the new setup of the Grid.
- *WFC*

  - Called when the "Apply WFC" button is pressed. Prepares the input for it to be used by the WFC Interface, and modifies the Grid with the resulting output.

- *SetupGrid*
  - Receives a matrix of integers and uses it to set up the Grid with the GameAgents corresponding with the integers. Used, for example, by the WFC method when applying the output of the WFC Interface output.

- *PlayStop*
  - Called when the play/edit button is pressed. It changes the state of the system.

- *SetupMainCamera*
  - Modifies the camera in such a way that the whole Grid is always visible.

- *ToogleSelect*
  - Called when the selection tool is activated or deactivated.

- *ToggleButtons*
  - Togles a list of buttons interactable.

- *ToggleToggles*
  - Toggles a list of toggles interactable.

- *ScenarioDropdownItemSelected*
  - Called when a new game scenario is selected, changing the contents of the Setup dropdown menu.

- *ToggleWFC*
  - Used to toggle the WFC button enabled or disabled according with if the selection area is big enough to allow for the usage of the WFC tool.

3.5.1 Developing a WFC tool

As was mentioned, the WFC algorithm, like the setup, update and visualize components, was implemented as an Interface in Unity. This interface only has one method that a target class must implement, WFC. This method receives as parameters a matrix of integers representing the input, a matrix of integers that will be used as output, a integer representing the pattern size to be used, the coordinates indicated where in the output the input is positioned, and several booleans representing the optional processes of rotation and mirroring that can be applied to the patterns.
A class WaveFunctionCollapse that inherits the interface was created. Before going into detail about its methods, an auxiliary class Pattern was created to streamline the process.

Pattern contains two attribute, an integer matrix called grid that represents the pattern and an integer pattern_size. The grid is always a square, so pattern_size represents both width and height. Pattern contains a couple of methods to help in its usage:

- *GetSubGrid*

  - Receives as parameters one cardinal direction;
  - It returns a subsection of its grid on the border corresponding to the direction given.

- *IsNeighbour*
  - Receives as parameters another Pattern and the direction that Pattern is relative to it;
  - Determines if both Patterns overlap by comparing the related subsections.

- *Equals*
  - Receives another Pattern;
  - Returns true if they have the same grid.

The WaveFunctionCollapse class contains the following methods:

- *WFC*
  - Implementation of the Interface method;
  - Starts by running through the input grid, registering the Patterns found on one list and their frequencies on another. Checks if any of the optional processes of rotation and mirroring are required, and if so applies them to copies of each of the Patterns on the list and adds them to it. Calculates the relative frequency of each Pattern and creates a list of possible neighbours for each pattern for every cardinal direction. Creates a Matrix of lists of integers that represents for each position of the output the possible Patterns that can exist, with the integers representing the index of the Pattern on the pattern list. The input is added to this output matrix by eliminating all other Patterns on the positions occupied by the input, signifying that only the Patterns already present in the input on those positions are allowed. With those preparations, the main loop of the algorithm can begin. Until a maximum number of iterations equal to the number of elements on the output grid is reached, the algorithm checks if the grid has collapsed (meaning there is only one Pattern for each position and the method can finish) or if there has occurred a collision (if there are any spots where there can't exist any of the Patterns). Following that, the position with the least entropy is calculated and one of the possible Patterns is chosen to be the only one allowed. To end the main loop, the information resulting from this decision is propagated.

- *Propagate*
  - Receives as parameters the output grid, the point that was collapsed and the lists of neighbours;
  - The propagation step of the WFC algorithm. It is responsible for determining which neighbours are affected by a change of possible Patterns in one point of the grid. If the possible Patterns for one of the neighbours is changed, the same process occurs to its neighbours.

- *RecalculatePossibilities*
  - Receives as parameters the output grid, the point that was collapsed and the lists of neighbours;

  - Used by the Propagate method. With the help of the lists of possible neighbours of each Pattern, it determines which ones are still possible in the requested position.

- *CheckGridCollapsed*
  - Receives a parameter the output grids;
  - Checks if all points of the output grid have only one Pattern associated.

- *CheckCollision*
  - Receives a parameter the output grids;
  - Checks if any of the points of the output grid doesn't have a single possible Pattern.

- *SelectPossibleValue*
  - Receives as parameters a list of possible Patterns and their relative frequencies;
  - Using the relative frequencies of the Patterns in the input grid, it picks one from a list of possibilities.

- *CheckGridCollapsed*
  - Receives as parameters a list of possible Patterns and their relative frequencies;
  - Calculates the entropy of one point of the output grid.

- *FindLeastEntropy*
  - Receives as parameters the output grid and their relative frequencies of every Pattern;
  - Finds the point of the output grid with least entropy.

The WFC algorithm was successfully implemented. The pattern size is a very influential parameter. At value 1 it makes the algorithm very locally focused, while respecting the relative frequency of each pattern, which in this case is only one tile. At value 3 it has a better capability of capturing the more global structures or relative relations/positions present in the input, with any value above showing little difference. The optional processes of rotation and mirroring allow for an easy addition of extra variety, however they do lengthen the process duration, since they increase the number of possible Patterns.

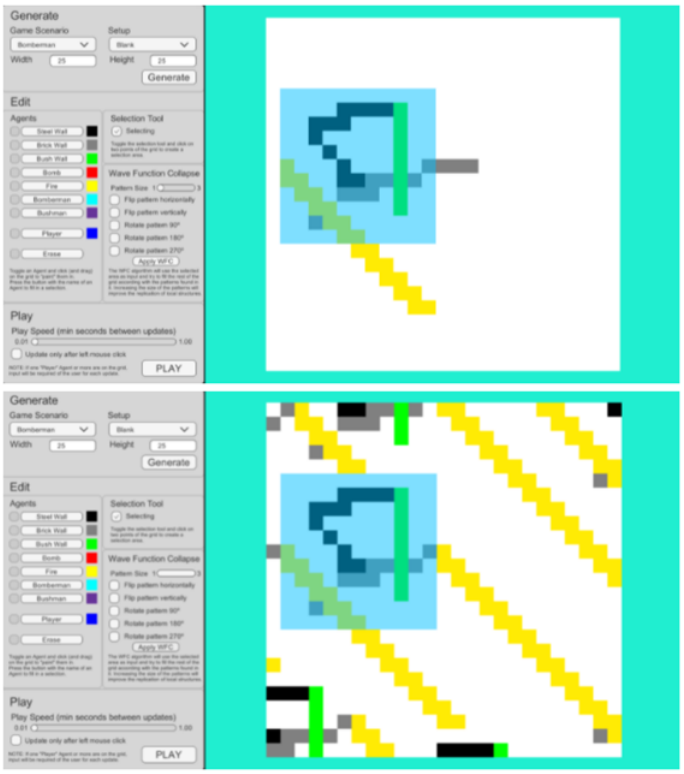


Figure 3.10: Example of the usage of the WFC tool in the final editor.

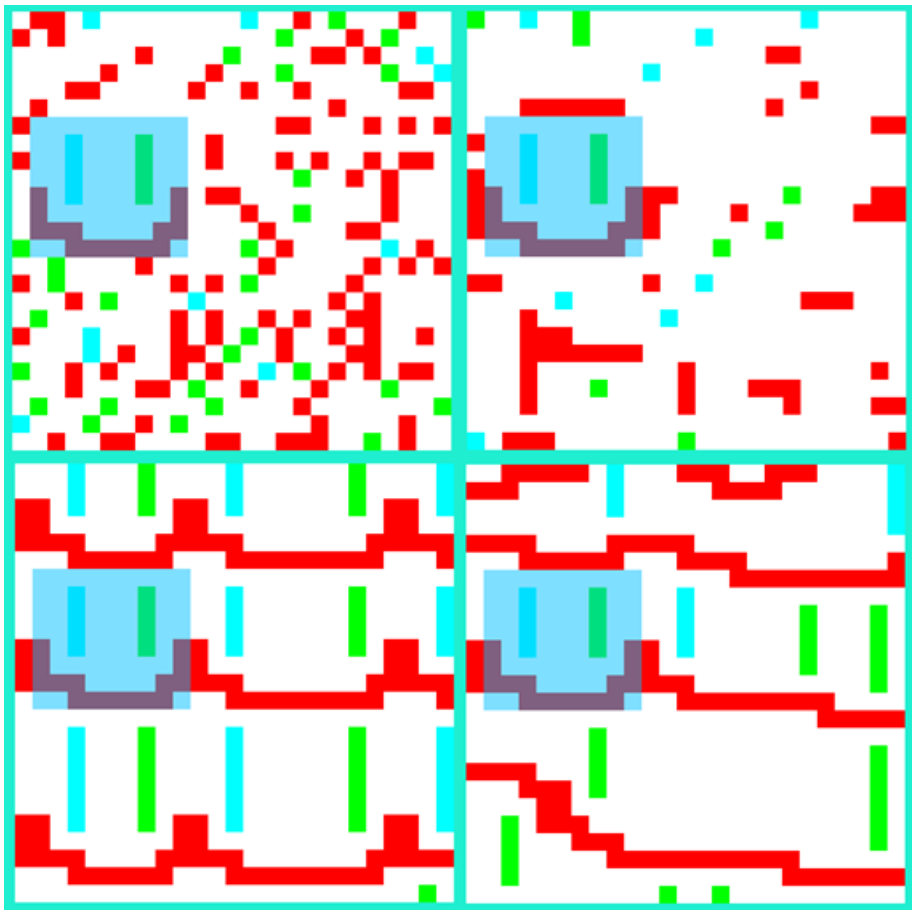


Figure 3.11: Example of the differences between pattern size values in the WFC tool. Top left has a pattern size value of 1; top right has a pattern size value of 2; bottom left has a pattern size of 3; and bottom right has a pattern size value of 3 with the optional processes of mirroring active.

Chapter 4

Prototype Implementation

This section will relate the development process, starting from the initial prototype to the final edition of the platform, documenting the decisions made at each step.

The platform was developed in the Unity Haas [2014] game engine in C#, although the proposed architecture isn't exclusive to it. As was already stated, one of the main objectives of its development was the support of the training of Machine Learning agents done by Diogo Alves Alves [2021], whose dissertation project pertains to the modeling of synthetic players as testing devices for generative game content. As such, the development process started with the creation of a Unity Project with a 2D template and the creation of a GitHub repository where all the parties involved could access the latest developed prototypes of the platform.

Development was structured in "sprints" of one to two weeks. Each sprint aimed at furthering the development of the platform, expanding it in some new way or revamping what was already there, all with the results of previous sprints in mind. In total there were 9 sprints with an average of 50 hours each. At the end of each sprint the GitHub repository was updated. There were weekly reunions with my supervisor where the state of the project and what we should be aiming for was discussed, helping keep the right objectives in mind, which contributed to the decisions that will be presented in this chapter.

4.0.1 Sprint #1: Game of Life Prototype

The first sprint objective was the creation of a prototype of the system that could successfully recreate Conway's Game of Life Games [1970]. The similarities of the proposed architecture with a Cellular Automata and the simplicity of the proof of concept were the driving factors of this decision.

A first version of the abstract class GameAgent was created, as well as the Grid class. To recreate Conway's Game of Life, the class LifeAgent, which inherited from the GameAgent class, was created, with the internal logic of the basic rules of the Game of Life. A temporary main script that did the operations later reserved for the interface components was created to set up, update and show the simulation on screen.

From this sprint, it was demonstrated that the proposed architecture was capable of successfully recreate the Game of Life and that the GameAgent would be able to take and analyze positions of the grid and use them in its update routine. The visualization of the simulation on-screen was also successful, meaning that the usage of Unity's GameObjects was a viable way of doing so.

The implementation of the main script revealed the necessity of turning the setup, update, and visualization processes modular since for different game scenarios there would be different necessities, while also existing some overlap.

A mistake in the implementation also revealed the necessity of sometimes having update cycles with several steps: the Game of Life simulation was working, but not replicating the patterns of the original version correctly. This occurred because after counting the number of alive neighbors the agent would immediately update its state, giving the yet-to-update agents the wrong version of the grid as input.

4.0.2 Sprint #2: Game of Life Prototype with 2 Types of Agents

In the second sprint, the errors found in the implementation of the prototype made in the previous sprint were corrected and a new version of the Game of Life proof of concept with multiple types of GameAgents was implemented.

The two GameAgent classes, LifeAgentAlive and LifeAgentDead were implemented and the Setup, Update and Visualize Interfaces were created. Two setup components and two visualization components were created - one of each for each of the versions of the Game of Life proof of concept - in addition to one update component which works for both.

The GameAgents and the update loop of the simulation were modified to allow for multiple steps for each update cycle, resulting in a faithful recreation of Conway's Game of Life. A setup component with a hard-coded "glider" structure was used to verify this.

As a result of this sprint, it was demonstrated that the simulation worked with multiple types of GameAgent, allowing for greater variety moving forward. The GameAgents were also proven to be able to modify the Grid successfully, instead of just changing their own states, being able to do such things as eliminating and creating new GameAgents.

4.0.3 Sprint #3: Implementing More Complex Agents

Sprint three was dedicated to pushing what the GameAgents were capable of doing forward and revamping some obsolete code in preparation for the Bomberman proof of concept.

A test class, AgentRandomWalk, was created to demonstrate GameAgent's ability to traverse the Grid from consecutive positions. As its name indicates, the agent in each update cycle would choose between keeping moving forward or choosing a new direction, keeping in its states attribute a reference to the direction it is facing.
Methods used by the AgentRandomWalk to move from one position of the Grid to another were later added to the GameAgent abstract class, being accessible for all who inherited it.

Another test class, AgentPlayerWalk, was created to figure out a way to give a human player control of a GameAgent. Coroutines was the solution found, and the AgentPlayerWalk class was reformatted into the abstract class, GameAgentPlayer.

From this sprint, it was demonstrated that the proposed architecture allowed for agents with more complex behaviors, including accepting input from a player.

4.0.4 Sprint #4: Basic Bomberman Prototype

In the fourth sprint, the objective of creating a prototype for a Bomberman proof of concept was tackled. Trying to replicate the game would demonstrate if the Platform would be able to handle a more complex simulation with the addition of a player-controlled agent.

The basic GameAgents AgentBomb, AgentFire, AgentStrongWall, and AgentWeakWall were developed and tested. A setup component that randomly places wall agents through the grid was created, as well as a visualization component to give each GameAgent type a distinct color. The AgentPlayerBomberman, which inherited from GameAgentPlayer, was created, which allowed a player to move through the generated map and place bombs. An update component capable of handling the existence of a GameAgent which requires the input of a user was created and tested.

This basic version of Bomberman without enemies allowed for the testing and refining of the systems architecture.

4.0.5 Sprint #5: Bomberman with Additional Agent Types

The fifth sprint built upon the prototype from the forth sprint by adding a GameAgent capable of behaving like a Bomberman enemy and some variations to the base game.

The AgentBomberman was created. It had the same actions at its disposal as the AgentPlayerBomberman, however, it is not controlled by a player, meaning that all the logic of what action to make when had to be implemented, proving if the architecture was capable of such a thing.

With enemies, the simulation needed an end condition to be a faithful recreation of the original game. As such, the update component was modified to keep track of how many players are still active, and the GameAgent class was modified to be able to contact the update component, in this particular case informing of the elimination of the agent.

To explore new possibilities of the platform, additional agents were created to expand upon the base of the Bomberman proof of concept. Since the AgentBomberman main influence upon the grid is destroying things, a new GameAgent AgentBushman would create bushes that expand across available space. This means, of course, that AgentBush was also created. In order to explore more interesting interactions, the AgentFire was modified to spread if in contact with AgentBrush.

4.0.6 Sprint #6: Support of Machine Learning Agent Training

The sixth sprint focused on preparing the platform to be used by Diogo Alves [drca], whose dissertation project pertains to the modeling of synthetic players as testing devices for generative game content. The system had to allow the training of machine learning-controlled agents.

In the eyes of the system, these types of agents would not be different from GameAgentPlayers, meaning that the update cycle would be halted until a decision was made by the synthetic player. The usage of the Bomberman game scenario would provide a win condition for the synthetic player, and the additional methods were implemented to facilitate the tracking of the agent's fitness. It would be necessary to run the simulation several times to train an agent, for this the ability to restart it once it has finished was added, controlling

the maximum number of update cycles in each run of the simulation and how many times it would be run. Lastly, all the steps of the system process that used randomness were altered to use the same pseudo-random number generator, to guarantee that, if a specific seed was used once again, the same results would show, in case there was a need to analyze one seed in particular.

4.0.7 Sprint #7: Enabling Editing of the Grid

In the seventh sprint, a way to edit the Grid was created. By treating it like a canvas where the GameAgents are paints of different colors a user can easily create simple scenarios in a familiar way.

A prototype of the editing system was created, allowing the user to stop the simulation at any type and click on the Grid to add an agent of a selected type to the specific position. The prototype was far from being easy to use, since each agent was mapped to a key on the keyboard, and with the great amount of GameAgent types of the Bomberman game scenario, one could get easily lost. However, the editing system was successful in allowing the user to edit the simulation at any time step, making changes on the fly as seen fit.

A selection tool was also developed to allow for bigger edits. Picking two positions of the Grid would create a selection area, visualized by a semi-transparent GameObject. By pressing a key corresponding to a GameAgent, the area would be filled.

4.0.8 Sprint #8: Including Wave Function Collapse

The eighth sprint was solely focused on the Wave Function Collapse algorithm and how to integrate it into the platform. With the editor prototype implemented in the last sprint, there was the opportunity of adding a tool that could assist the human designer in an intelligent way, a co-creation tool. The algorithm chosen as the base of this tool was one capable of receiving an input and recreating its style by analyzing its patterns and placing them in ways that make local sense, the Wave Function Collapse.

A Wave Function Collapse method and auxiliary methods were implemented directly on the editor system script. Several optional processes of rotation and mirroring, as well as the ability to handle several pattern sizes, were added to the base Wave Function Collapse algorithm, giving the user better control of the output given by the tool. A big change added was the integration of the given input into the output, since the objective was for the tool to "complete" the Grid around the selected area.
The already existing selection tool was revamped to return the selected area of the Grid as an input capable of being parsed by the Wave Function Collapse method.

The tool does its job successfully, apart from some bugs that were later corrected. However, as the size of the selected area and the grid grow, its efficiency declines. The bigger the input, the bigger the number of possible patterns, the bigger the size of the patterns, the more complex it is to check for compatibility. The bigger the output, the bigger the number of cells to propagate information to. Enabling the optional processes of rotation and mirroring also increases the number of possible Patterns to consider. That said, the results for the smaller grid sizes show that the Wave Function Collapse algorithm as a mechanic has potential as an editorial tool if optimized, being an easy way of propagating a style through the grid.

4.0.9 Sprint #9: Designing a GUI

In the final sprint, the objective was to turn the platform more user-friendly by designing a graphical user interface. This was achieved through the usage of Unity's built-in GUI elements, reserving a third of the screen to the simulation and adding a panel with generation, editing, and play controls.

The main script was adapted to work with all the new GUI elements correctly, and access to all the Interface components created through the development process was established, enabling the usage of the two game scenarios, Game of Life and Bomberman. The Wave Function Collapse code was moved into a new class to ease development. For extra control, a customizable delay was added to the update component.

At the end of the last development sprint, all that was left was to evaluate the final prototype.

This page is intentionally left blank.

Chapter 5

Evaluations

5.1 Goals

The proposed platform is designed to support and aid in the creation and editing of playable scenarios based on the model of a society of agents. To gain some understanding of how useful and usable the developed prototype was, how it was interpreted and explored by users, five voluntary participants were asked to operate the platform. After initially trying to perform a given list of tasks. These tasks start specific and become more and more open to interpretation, allowing for the assessment of how users understand what and how they can operate the platform, and if they could build upon that understanding. There was also the question of how they would interpret the co-creativity brought in by the WFC agent generation tool and what users would do with it.

5.2 Protocol

The tests would start with a short introduction, contextualizing what the platform and its objectives are. The participants would then be asked to perform the following sequence of tasks:

- Generate a grid for the game scenario "Game of Life" with the "Random" setup; the simulation and change the update delay to 1 second. Activate the debug mode; the simulation, add new Alive Agents to it and unpause it. Do the same thing with the selection tool; the simulation and apply the WFC tool to some area of the grid; a game of Bomberman with the "Classic" setup and play; to determine the behavior of the AgentBushman and AgentBush; the platform for a minute however you see fit.

On average, a test took 15 minutes. During these tests, the participants were encouraged to voice their thought processes. All of the sessions where screen recorded and notes of relevant behavior were taken during the tests for future analysis.

5.3 Participants

Since these tests were done during the COVID-19 pandemic, it is relevant to note that safety measures were taken. The hardware used was disinfected in between tests and all

the involved wore face masks.

All the participants were students of the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra. However, not all of them had a background in Informatics Engineering or Intelligent Systems, not being entirely familiar with some of the concepts related to the platform.

5.4 First Observations Report

Participant 1 was able to successfully operate the platform through the first two tasks, having difficulty in applying agents to a selected area in task 3. Despite this, the participant showed great interest in using the editorial capabilities of the platform to experiment with the Game of Life game scenario, "drawing" patterns of agents to explore how they would evolve through the steps of the simulation. There were some problems with task 4 while using the WFC tool: The size of the grid, 100x100, and of the patterns used by the algorithm, 4, made it so the process would take more time than acceptable. Faced with this setback, the platform was rebooted and task 4 was tried once again with a smaller grid size, 50x50, this time successfully, figure 5.1. After all the enemy bomberman died during task 5, the participant added more with the aid of the editing system. In task 6, the participant found out what the agents did by first placing them on the grid and observing and later by adding an agent player to interact directly with them. Participant 1 ended the test by using the WFC tool to create a Bomberman map and placing an AgentPlayerBomberman to play, figure 5.2.

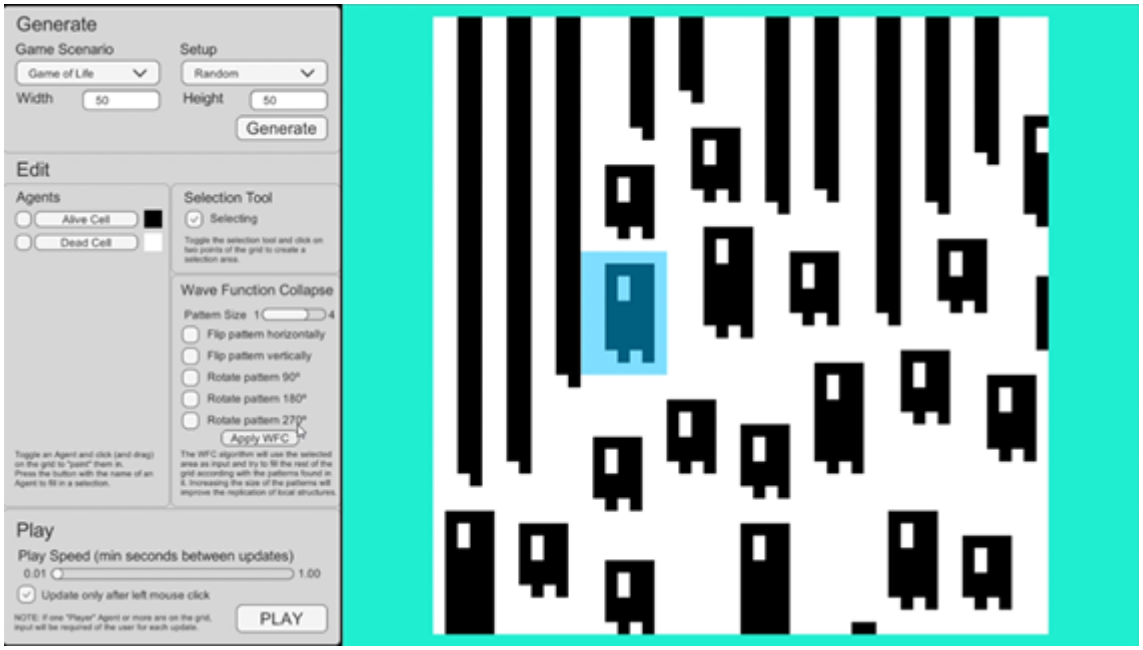


Figure 5.1: Participant 1's usage of the WFC tool.

Participant 2, much like the first one, was able to operate the platform successfully during the first two tasks. The same problems with the selection and WFC tools repeated themselves in the same manner and the platform had to be restarted once again. For the fourth task, the participant chose to use the WFC tool on a repeating pattern that they noticed while running the simulation, propagating through the rest of the grid. While playing the classic bomberman simulation, they also placed new enemies once the initial

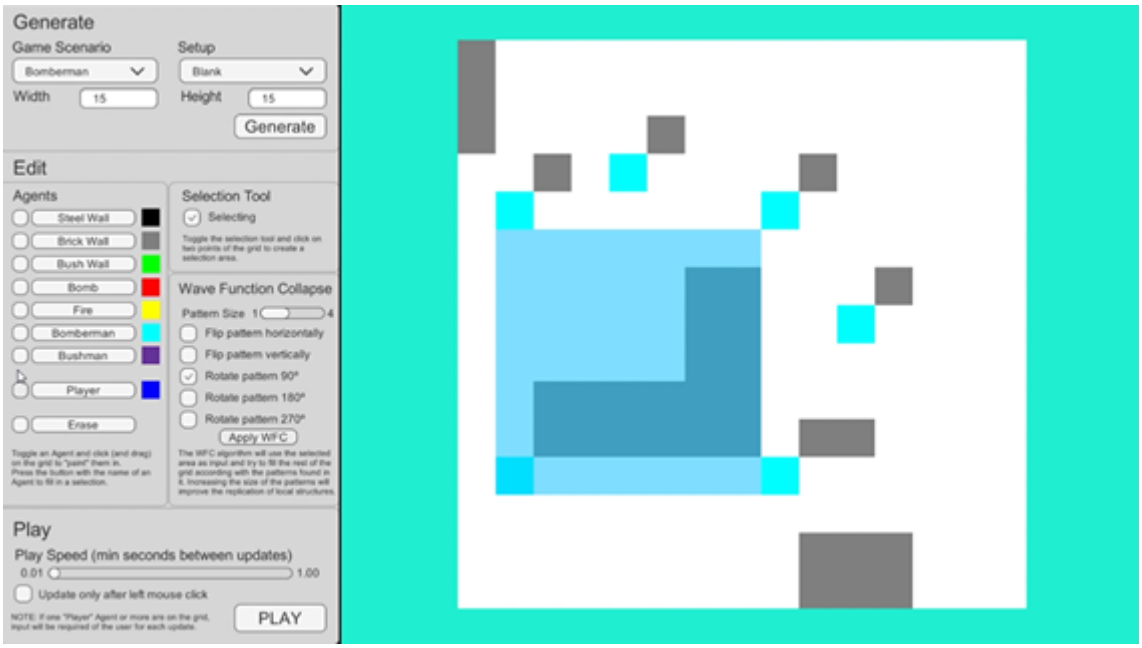


Figure 5.2: Participant 1's WFC generated Bomberman map.

ones had died. The participant also started to show interest in how the game scenario was modeled through a society of agents and tried to figure out the logic of each agent, which they continued in the sixth task by interacting with the mystery agents with an AgentPlayerBomberman. For the last task, the participant chose to explore the Game of Life game scenario further, by creating patterns of alive cells with all the editorial tools and studying the resulting simulation.

5.5 First Insights and Updates

With insights gained from the first two explorations, changes were made to the GUI to make the selection tool less misleading - when an area of the grid is selected, toggling an Agent on would fill the area with it instead of leaving the selection mode. When both participants tried to use the selection tool this was the sequence of actions that seemed logical to them, this change allows for that possibility.

Also, at this point in the sequence of tests, some problems with the WFC tool were registered. Whenever the last three participants would use it in such a way that would necessitate the restart of the platform, they were discouraged as to make the tests more efficient.

5.6 Second Observations Report

Participant 3 showed great interest in exploring the Game of Life simulation, creating complex structures, and analysing how they developed. They noted that it would be interesting to give the user a way to go back in the simulation. The participant explored the WFC tool by trying out several pattern sizes, noting that it would be useful to have a previsualization window that shows how the optional processes of mirroring and rotation

of the tool affected a pattern. During task 5, the participant tried out several grid sizes for the Bomberman "Classic" setup, playing several times. In order to determine the behavior of the AgentBushman and AgentBush, the participant opted to restart on a blank grid, scattering every type of agent through it and analyzing the results. Once the behavior of these agents as well as how AgentBush behaves when in contact with AgentFire, the participant had some fun with the simulation of the spreading fire, creating scenarios full of AgentBushes and placing bombs with the AgentPlayerBomberman. In the last task, the participant chose to create a Bomberman map, using the WFC tool to generate weak and strong walls through the grid, and then positioning AgentBombermens and AgentBushmens by hand alongside one AgentPlayerBomberman and started playing, figure 5.4.

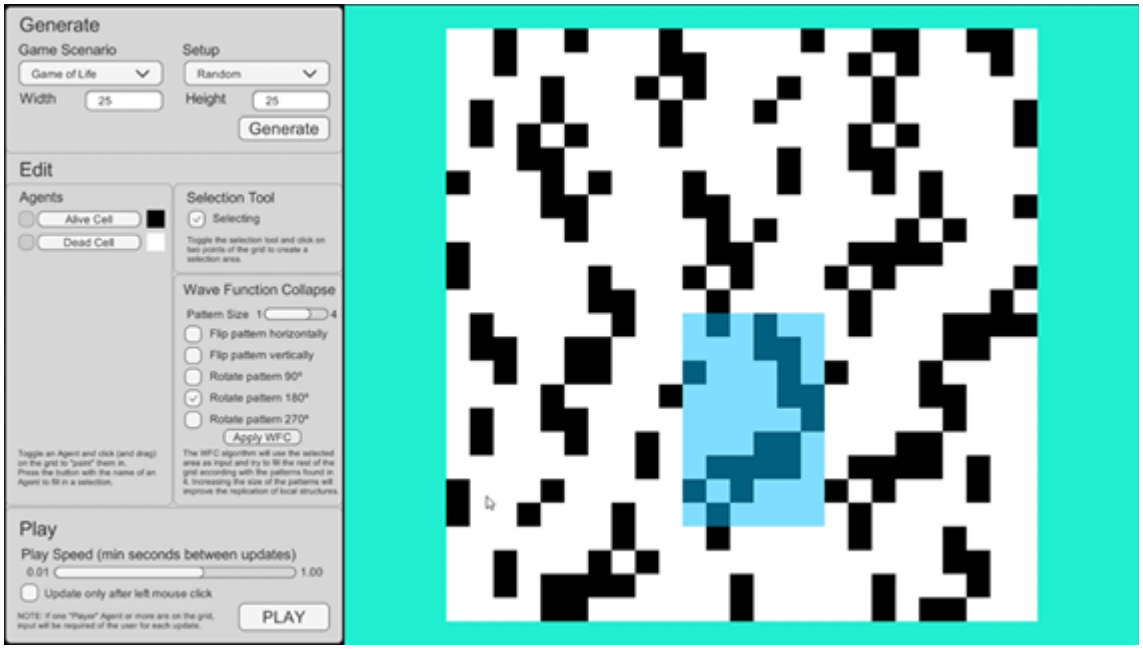


Figure 5.3: Participant 3's usage of the WFC tool.

Participant 4 was able to complete the first four tasks related to the Game of Life game scenario with some occasional confusion about the GUI layout. Once the participant understood that the editor could be used as one would a drawing software, some exploration of the possibilities of the Game of Life was made. The participant wasn't familiar with video games, although there were no major problems in controlling the AgentPlayerBomberman during the fifth task. For the sixth task, the participant placed the two types of agents on a blank canvas and ran the simulation. No additional agent types were added to test possible interactions. Lastly, the participant explored the Game of Life game scenario a little more, using the selection and WFC tools and experimenting with different grid dimensions. The participant had some closing remarks about the GUI: the subsections could be segmented in a more perceptible way and the selection of what agent type to use with the selection tool activated is still quite confusing.

Participant 5 completed the four tasks related to the Game of Life game scenario with ease. During the fifth task, the participant added new bomberman agents once the original ones were destroyed with the aid of the editing system. To explore the behavior of AgentBushman and AgentBush, the participant generated a new empty grid and created a complex structure with several of the Bomberman game scenario GamAgent types, figure x. For the last task, the participant tried to explore a little more of every aspect of the

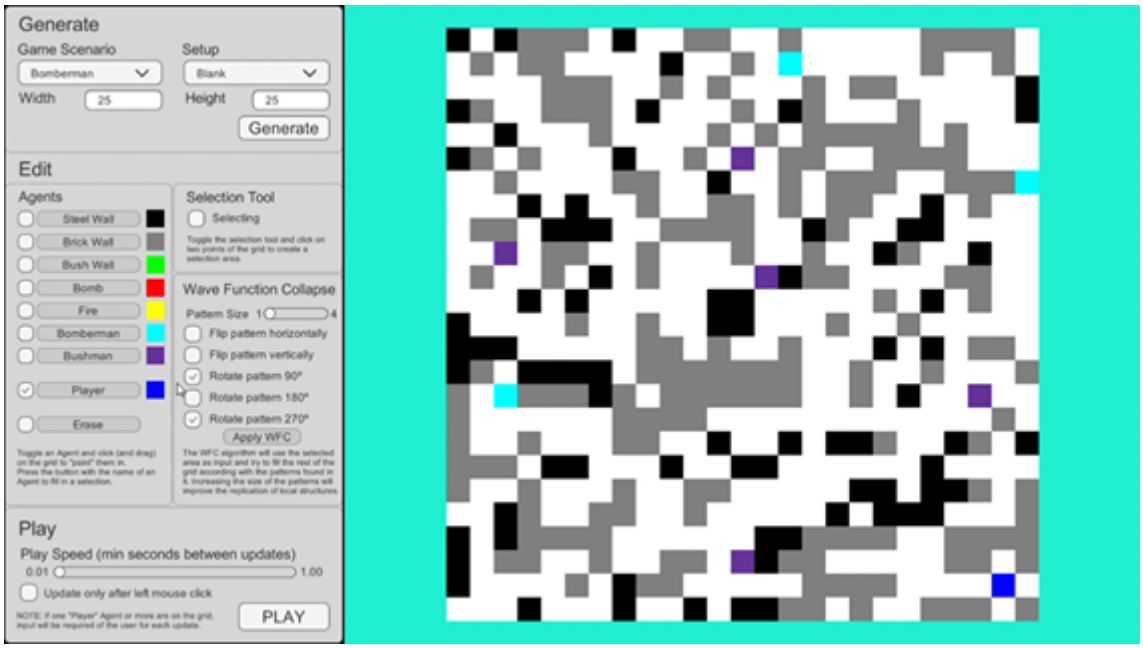


Figure 5.4: Bomberman map created by Participant 3.

platform, experimenting with all the editor tools in both game scenarios. The session concluded with the participant elaborating upon some of the inconveniences within the editor system: the selection tool does not work by dragging the mouse across the grid, as one has come to expect from such tool, and the same complaints presented by participant 4 about the selection of agent types to fill the selected area.

5.7 Issues and Insights

From the reports of the tests, some recurrent observations were made that can inform the design of the system.

The WFC, when used in the wrong conditions, is very inefficient, halting the whole system for minutes on end. During the tests, this seemed to happen mainly in the Bomberman game scenario, with a possible explanation being that for the Game of Life scenario there are fewer possible patterns since there are only two types of agents. The size of the grid, the size of the pattern, and the size of the input area also negatively affect performance, since they respectively increase the number of positions to calculate a pattern for, increase the complexity of the pattern operations, and the number of possible patterns. The usage of the optional processes of mirroring and rotation naturally have the same effect, since each one has the potential to double the number of possible patterns.

The selection tool needs revamping. Several of the participants had moments of confusion using it. This seemed to occur for two main reasons: the way the selection area is defined and the way to fill the area with a specific agent. In most applications with such a tool, the creation of the selection area is done through the dragging of the mouse from one extreme to another. Instead, the selection tool of the platform requires two clicks on the grid to define the two extreme points of the area, which could have led to this confusion. The way one fills the selected area with a specific agent type can also be quite confusing to users not familiar with the platform since it uses a different GUI element than the toggle used

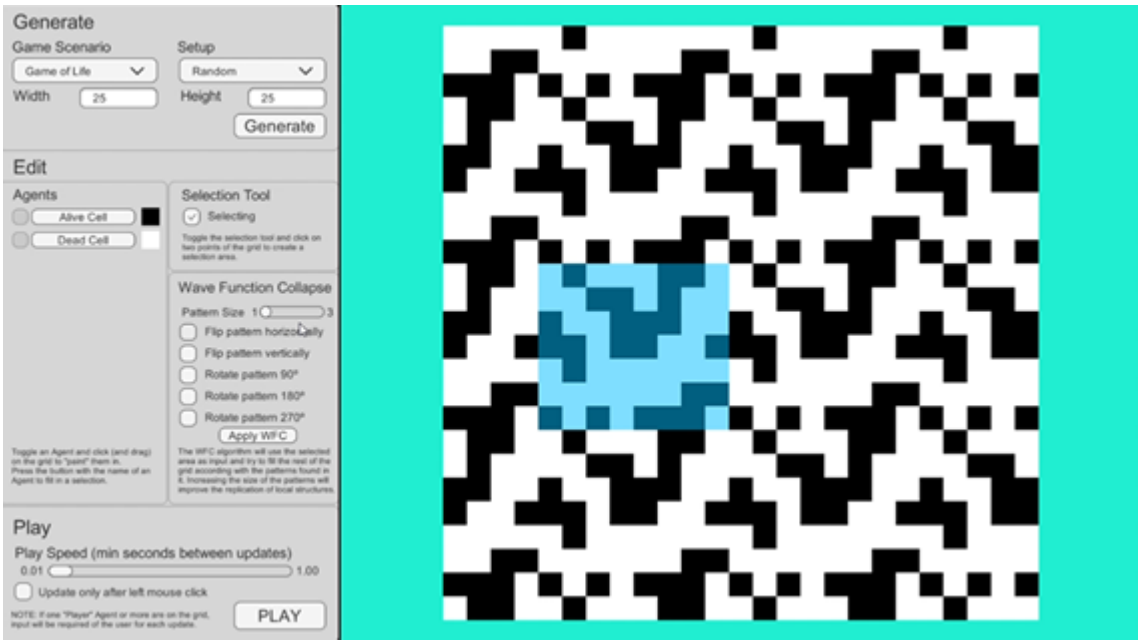


Figure 5.5: Participant 4's usage of the WFC tool.

to select the agent type while "drawing" normally.

Some comments about the structure of the GUI not being ideal were made. The main problem being that thanks to the panels used to group each of the editorial tools' components, it is not certain if they belong to the "Edit" subsection of the GUI. Restructuring of these panels or the usage of color to symbolize each of the subsections could be a possible solution.

The speed change slider could be better. As it stands, the slider controls the delay between each update cycle, with values varying from 0.01 to 1 second. This is counterintuitive, since as the slider increases the speed of the stimulation decreases.

All the participants showed interest in exploring the capabilities of the simulation of at least one of the game scenarios, as was shown in section 5.4 and section 5.6. This is a good indication that the platform was successful enough in giving control of the simulation to the users as to incentivize exploration of the developed game scenarios.

When asked to use the platform however they would see fit, all of the participants engaged with the WFC tool, despite its setbacks with grids of larger size. This demonstrates that the Wave Function Collapse algorithm as the basis for a co-creation tool can not only be usable, but useful in the creation process.

5.8 Evolution Proposals

From the obtained results some evolution proposals of where to take the platform forward can be made.

One of the most interesting suggestions given by the participants was that of participant 2's ability to go back in the simulation. This could be implemented by keeping a history of a limited number of previous agentGrid states. That would give the user the possibility of trying some alternative setup and comparing the results or going back to a point in the

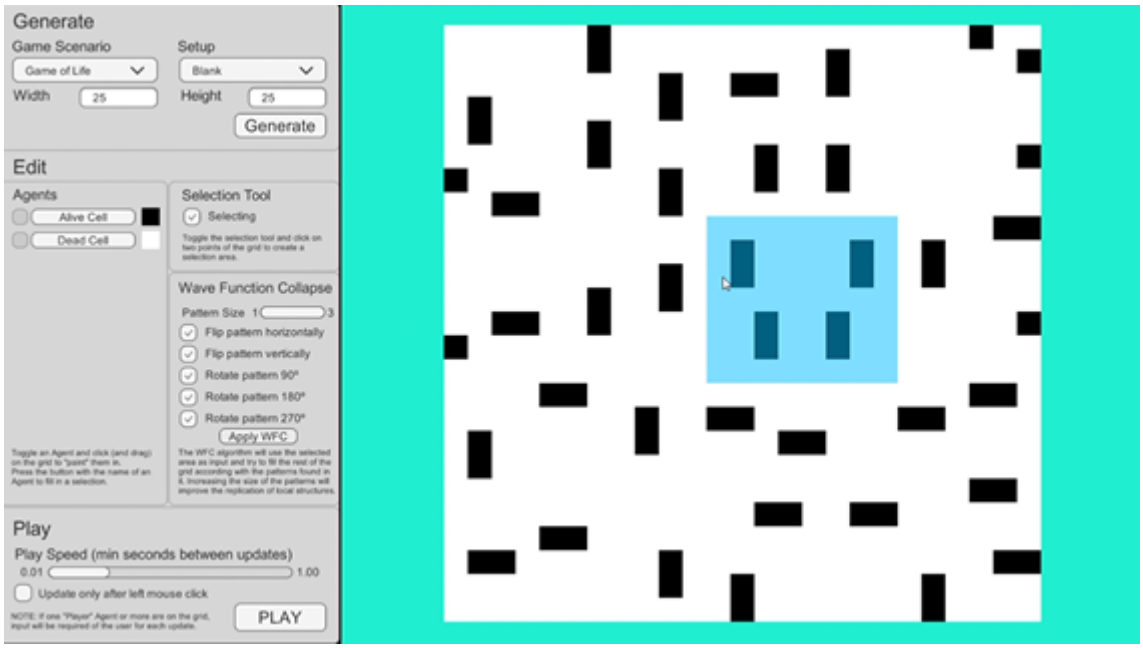


Figure 5.6: Participant 5's usage of the WFC tool.

simulation before something unwanted had happened. It could also enable the possibility of replaying simulations without having to redo them.

Allowing the editing of different layers of the agent grid is another possible evolution of the platform. Since the architecture allows for several agents to occupy the same place, it would make sense to be able to edit several of them in one of the positions of the grid. This could be achieved through the implementation of a layer system.

A final and particularly ambitious idea is that of an agent editor - a tool that would allow the user to create new agents, defining their internal rules, input sensors, output actions, and visual representation. If the editor can be thought of as a drawing software where the grid is the canvas and the agent types the different colored brushes, this would be akin to giving the user the ability to create its own new brush.

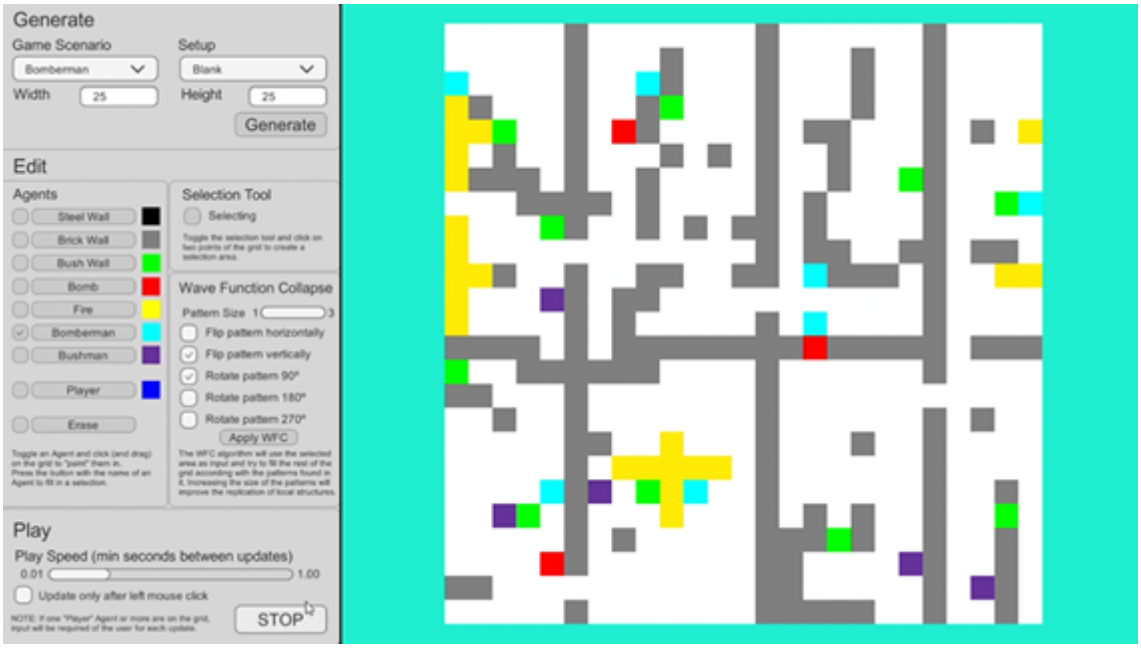


Figure 5.7: Participant 5's Bomberman map.

Chapter 6

Conclusion and Future Work

6.1 Contributions

The main objective of this dissertation was the creation of a platform that utilizes a simulation of an agent society to model game scenarios for the testing and exploration of PCG approaches. With the conclusion of this work, the following contributions were made:

- An architecture capable of modeling game scenarios as complex adaptive systems was developed;
- A model of game agent used as a way to define various types of agents and how they may interact with the complex adaptive system and each other;
- A model of game agent controllable by a player, either synthetic or human, and usable in the developed architecture;
- Modeling of an editing system for complex adaptive systems;
- Modeling of a Wave Function Collapse-based co-creation tool to be used in the editing system.

6.2 Future Work

From the development process and the analysis of its results, several ideas of how to move the architecture of the platform further were formed, pointing at some future work to undertake:

- Creation of a modular way to define an agent's behavior, allowing the creation of new agents on the fly by a user or through some PCG algorithm;
- Creation of a system that allows the editing of several layers of the complex adaptive system, not only the basic distribution of agent types per cell;
- Modeling of agents capable of evolving their own behavioral rules throughout their lifetime, by learning successful action sequences or recognizing conditions for acting;
- Evolving the WFC-based co-creativity tool to make it more efficient and usable in broader cases, for instance by enabling selection and painting of selected areas and formats.

6.3 Closing Remarks

This work allowed me to explore the world of PCG in games, deepening my knowledge of how game content is generated and for what purpose. Learning about and exploring the PCG state-of-art made me aware of tons of techniques and possibilities not yet explored. Getting to know the advantages and disadvantages of each one, and how their combination can result in something greater than the sum of its parts was one of the most fascinating things that I learned outside of any subject of the course. I will keep using the knowledge I gained here in all future projects - even if just as a hobbyist game developer, I have learned how PCG can empower me.
Another aspect of this project that influenced me for the better was that of its management. I had the opportunity to apply the methodologies I had studied in my course into a big project - at least the biggest I have tackled until now. The importance of having a development process established from the beginning was made clear to me in practice and not just theory.
Developing the platform and its architecture was one of the best experiences in my learning journey. This is a project that I genuinely care about and want to see evolve and where it can reach.

References

Wikipedia contributors. Logic programming. *Wikipedia, The Free Encyclopedia. Retrieved 00:48, January 9, 2021*, 2021. URL `https://en.wikipedia.org/w/index.php?title=Logic_programming&oldid=996484651`.

17-Bit. *Galak-Z: The Dimensional*. 17-Bit, Sony Interactive Entertainment, 2015.

R. Abela, A. Liapis, and G. N. Yannakakis. A constructive approach for the generation of underwater environments. *Proceedings of the FDG workshop on Procedural Content Generation in Games*, 2015.

D. Adams. Automatic generation of dungeons for computer games. *B.Sc. thesis, University of Sheffield, UK*, 2002.

T. Adams. *Dwarf Fortress*. Bay 12 Games, 2006.

Z. Aikman. Galak-z: Forever: Building space-dungeons organically. *Game Developers Conference*, 2015.

D. Alves. Modeling of synthetic players as an instrument for testing generative content. *Master' Dissertation, Universidade de Coimbra, 30041-531 Coimbra*, 2021. URL `http://estagios.dei.uc.pt/cursos/mei/ano-lectivo-2020-2021/propostas-com-alunos/?idestagio=3925`.

D. Ashlock, C. Lee, and C. McGuinness. Search-based procedural generation of maze-like levels. *IEEE Transactions on Computational Intelligence and AI in Games 3(3)*, page 260–273, 2011.

N. Barreto and L. Roque. A survey of procedural content generation tools in video game creature design. 2014.

F. Belhadj. Terrain modeling: a constrained fractal model. *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, page 197–204, 2007.

Bethesda Game Studios. *The Elder Scrolls V: Skyrim*. Bethesda Softworks, 2011.

Blizzard North. *Diablo*. Blizzard Entertainment, Ubisoft and Electronic Arts, 1997.

M. Booth. The ai systems of left 4 dead. *Keynote, Fifth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 2009.

Boris. Wave function collapse tips and tricks. *BorisTheBrave.Com, 2020-02-08*, 2020. URL `https://www.boristhebrave.com/2020/02/08/wave-function-collapse-tips-and-tricks/`.

D. Braben and I. Bell. *Elite*. Acornsoft, Firebird and Imagineer, 1984.

E. Butler, A. M. Smith, Y.-E. Liu, and Z. Popovic. A mixed-initiative tool for designing level progressions in games. *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, page 377–386, 2013.

L. Cardamone, D. Loiacono, and P. Lanzi. Interactive evolution for the procedural generation of tracks in a high-end racing game. *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, page 395–402, 2011a.

L. Cardamone, G. N. Yannakakis, J. Togelius, and P. L. Lanzi. Evolving interesting maps for a first person shooter. *In Applications of Evolutionary Computation*, page 63–72, 2011b.

M. Cook and S. Colton. Ludus ex machina: Building a 3d game designer that competes alongside humans. *In Proceedings of the Fifth International Conference on Computational Creativity (ICCC2014). International Association for Computational Creativity.*, 2014.

R. Craveirinha, N. Barreto, and L. Roque. Towards a taxonomy for the clarification of pcg actors' roles. 2016.

S. Dahlskog and J. Togelius. Patterns as objectives for level generation. *In Proceedings of the International Conference on the Foundations of Digital Games.*, 2013.

S. Dahlskog, J. Togelius, and M. J. Nelson. Linear levels through n-grams. *Proceedings of the 18th International Academic MindTrek Conference: Media Business, Management, Content Services.*, page 200–206, 2014.

I. M. Dart, G. D. Rossi, and J. Togelius. Speedrock: procedural rocks through grammars and evolution. *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, 2011.

Doge Roll. *Enter the Gungeon.* Doge Roll, 2016.

J. Doran and I. Parberry. Controlled procedural terrain generation using software agents. *IEEE Transactions on Computational Intelligence and AI in Games 2(2)*, page 111–119, 2010.

J. Dormans. Adventures in level design: generating missions and spaces for action adventure games. *In: PCG'10: Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 1 –8, 2010.

J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *IEEE Transactions on Computational Intelligence and AI in Games, 3(3)*, page 216–228, 2011.

D. S. Ebert. and M. Kaufmann. Texturing modeling: a procedural approach. 2003.

M. S. El-Nasr. Intelligent lighting for game environments. *Journal of Game Development*, 2005.

M. S. El-Nasr, A. Vasilakos, C. Rao, and J. Zupko. Dynamic intelligent lighting for directing visual attention in interactive 3-d scenes. *Computational Intelligence and AI in Games, IEEE Transactions on, 1(2)*, page 145–153, 2009.

Firaxis Games. *Civilization VI.* 2K Games Aspyr, 2016.

T. Forsyth. Cellular automata for physical modelling. *Game Programming Gems, 3:200–214*, 2002.

M. Frade, F. de Vega, and C. Cotta. Modelling video games' landscapes by means of genetic terrain programming: A new approach for improving users' experience. *Applications of Evolutionary Computing*, page 485–490, 2008.

M. Frade, F. de Vega, and C. Cotta. Evolution of artificial terrains for video games based on accessibility. *Applications of Evolutionary Computation*, pages 90–99, 2010.

M. Frade, F. de Vega, and C. Cotta. Automatic evolution of programs for procedural generation of terrains for video games. *Soft Computing 16(11)*, page 1893–1914, 2012.

Freehold Games. *Caves of Cud*. Freehold Games, 2015.

W. Gaisbauer. Creating immersive populated cities for virtual reality. *Universität Wien, Vienna.*, 2021.

M. Games. The fantastic combinations of john conway's new solitaire game "life" by martin gardner. *Scientific American, 223*, page 120–123, 1970.

M. Gumin. Wave Function Collapse Algorithm, 9 2016. URL `https://github.com/mxgmn/WaveFunctionCollapse`.

J. K. Haas. A history of the unity game engine. 2014.

E. J. Hastings, R. K. Guha, and K. O. Stanley. Evolving content in the galactic arms race video game. *In IEEE Symposium on Computational Intelligence and Games*, page 241–248, 2009.

J. H. Holland. *Signals and Boundaries: Building Blocks for Complex Adaptive Systems*. The MIT Press., 2012.

A. K. Hoover, J. Togelius, and G. N. Yannakakis. Composing video game levels with music metaphors through functional scaffolding. *First Computational Creativity and Games Workshop, ICCC.*, 2015.

I. Horswill and L. Foged. Fast procedural level population with playability constraints. *Proceedings of the Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, page 20–25, 2012.

K. Hudson Soft. *Bomberman*. Nintendo, Hudson Soft, Sega, Konami, 1983.

Interactive Data Visualization, Inc. *SpeedTree*. 2014.

L. Johnson, G. N. Yannakakis, and J. Togelius. Cellular automata for realtime generation of infinite cave levels. *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 2010.

S.-A. Karth, I. Wavefunctioncollapse is constraint solving in the wild. *Proceedings of the International Conference on the Foundations of Digital Games - FDG '17, article no.: 68*, pages 1–10, 2017.

M. Kerssemakers, J. Tuxen, J. Togelius, and G. Yannakakis. A procedural procedural level generator generator. *Proceedings of the IEEE Conference on Computational Intelligence and Games*, page 335–341, 2012.

T. Lechner, P. Ren, B. Watson, C. Brozefski, and U. Wilenski. Procedural modeling of urban land use. *ACM SIGGRAPH 2006 Research posters*, page 135, 2003a.

T. Lechner, B. Watson, and U. Wilensky. Procedural city modeling. *Proceedings of the 1st Midwestern Graphics Conference*, 2003b.

B. Li, S. Lee-Urban, D. S. Appling, and M. O. Riedl. Crowdsourcing narrative intelligence. *Advances in Cognitive Systems, 2(1)*, 2012.

A. Liapis, H. P. Martınez, J. Togelius, and G. N. Yannakakis. Transforming exploratory creativity with delenox. *Proceedings of the Fourth International Conference on Computational Creativity*, page 56–63, 2013.

A. Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology, 18(3)*, page 280–299, 1968.

D. Loginova. Noita: a game based on falling sand simulation. 2019. URL `https://80.lv/articles/noita-a-game-based-on-falling-sand-simulation/`.

P. Machado and H. Amaro. Fitness functions for ant colony paintings. *In Proceedings of the fourth International Conference on Computational Creativity (ICCC)*, 2013.

P. Machado, T. Martins, H. Amaro, and P. H. Abreu. Evolutionary and biologically inspired music, sound, art and design. *Third European Conference, EvoMUSART*, 2014.

B. Mark, T. Berechet, T. Mahlmann, and J. Togelius. Procedural generation of 3d caves for games on the gpu. *Proceedings of the Conference on the Foundations of Digital Games (FDG)*, 2015.

H. Martinez and G. Yannakakis. Mining multimodal sequential patterns: A case study on affect detection. *Proceedings of the 13th International Conference in Multimodal Interaction. ACM*, 2011.

M. Mateas and A. Stern. Facade: An experiment in building a fully-realized interactive drama. *In Game Developers Conference*, 2003.

Maxis. *Spore*. Electronic Arts, 2008.

E. McMillen and F. Himsl. *The Binding of Isaac*. Independent, 2011.

E. McMillen and Nicalis. *The Binding of Isaac: Rebirth*. Nicalis, 2014.

Media Molecule. *Little Big Planet*. Sony Computer Entertainment Europe, 2008.

Mega Crit Games. *Slay the Spire*. Mega Crit Games, 2019.

S. Miyamoto and T. Tezuka. *Super Mario Bros.* Nintendo, 1980.

Mojang. *Minecraft*. Mojang and Microsoft Studios, 2011.

F. Musgrave, C. Kolb, and R. Mace. The synthesis and rendering of eroded fractal terrains. *Proceedings of SIGGRAPH 1989*, page 41–50, 1989.

Nolla Games. *Noita*. Nolla Games, 2020.

notch. Terrain generation part 1. *Tumblr,March 9, 2011*, 2011. URL `https://notch.tumblr.com/post/3746989361/terrain-generation-part-1`.

J. Olsen. Realtime procedural terrain generation. 2004.

M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation 5(4)*, page 349–358, 2001.

K. Perlin. An image synthesizer. *ACM SIGGRAPH Computer Graphics, 19(3)*, page 287–296, 1985.

O. Plausible Concept, Stålberg. *Bad North.* Independent, 2018.

P. Prusinkiewicz and A. Lindenmayer. he algorithmic beauty of plants. 1990.

Re-Logic. *Terraria.* Re-Logic, 2011.

Red Hook Studios. *Darkest Dungeon.* Red Hook Studios, 2016.

C. Z. M. J. Sandhu, A. Enhancing wave function collapse with design-level constraints. *Proceedings of the 14th International Conference on the Foundations of Digital Games - FDG'19, article no.: 17*, pages 1–9, 2019.

N. Shaker, J. Togelius, and G. Yannakakis. Towards automatic personalized content generation for platform games. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2010a.

N. Shaker, J. Togelius, and G. Yannakakis. Towards automatic personalized content generation for platform games. *Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference*, page 63–68, 2010b.

N. Shaker, M. Nicolau, G. Yannakakis, J. Togelius, and M. O'Neill. Evolving levels for super mario bros. using grammatical evolution. *Proceedings of the IEEE Conference on Computational Intelligence and Games*, page 304–311, 2012a.

N. Shaker, G. Yannakakis, J. Togelius, M. Nicolau, and M. ONeill. Fusing visual and behavioral cues for modeling user experience in games. *IEEE Transactions on Systems Man, and Cybernetics 43(6)*, page 1519–1531, 2012b.

N. Shaker, S. Asteriadis, G. N. Yannakakis, and K. Karpouzis. Fusing visual and behavioral cues for modeling user experience in games. *Cybernetics, IEEE Transactions on, 43(6)*, page 1519–1531, 2013a.

N. Shaker, S. Asteriadis, G. N. Yannakakis, and K. Karpouzis. Fusing visual and behavioral cues for modeling user experience in games. *Cybernetics, IEEE Transactions on, 43(6)*, page 1519–1531, 2013b.

N. Shaker, S. Asteridadis, K. Karpouzis, and G. Yannakakis. Fusing visual and behavioral cues for modeling user experience in games. *IEEE Transactions on Cybernetics 43(6)*, page 1519– 1531, 2013c.

N. Shaker, G. Yannakakis, and J. Togelius. Crowdsourcing the aesthetics of platform games. *IEEE Transactions on Computational Intelligence and AI in Games 5(3)*, page 276–290, 2013d.

N. Shaker, J. Togelius, and M. J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, 2016.

A. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games 3(3)*, page 187–200, 2011a.

A. M. Smith and M. Mateas. Answer set programming for procedural content generation: A design space approach. *Computational Intelligence and AI in Games, IEEE Transactions on, 3(3)*, page 187–200, 2011b.

A. M. Smith, E. Andersen, M. Mateas, and Z. Popovic. A case study of expressively constrainable level design automation tools for a puzzle game. *Proceedings of the International Conference on the Foundations of Digital Games*, page 156–163, 2012.

G. Smith, J. Whitehead, and M. Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *Computational Intelligence and AI in Games, IEEE Transactions on, 3(3)*, page 201–215, 2011a.

G. Smith, J. Whitehead, M. Mateas, M. Treanor, J. March, and M. Cha. Launchpad: A rhythmbased level generator for 2-d platformers. *IEEE Transactions on Computational Intelligence and AI in Games 3(1)*, page 1–16, 2011b.

N. Sorenson and P. Pasquier. The evolution of fun: Automatic level design through challenge modeling. *Proceedings of the First International Conference on Computational Creativity (ICCCX)*, page 258–267, 2010.

A. J. Summerville and M. Mateas. Mystical tutor: A magic: The gathering design assistant via denoising sequence-to-sequence learning. *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference.*, 2016.

A. J. Summerville, S. Philip, and M. Mateas. Mcmcts pcg 4 smb: Monte carlo tree search to guide platformer level generation. *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference.*, 2015.

A. J. Summerville, S. Snodgrass, M. Guzdial, C. Holmgârd, A. K. Hoover, A. Isaksen, A. Nealen, and J. Togelius. Procedural content generation via machine learning (pcgml). 2017.

Supergiant Games. *Hades*. Supergiant Games, 2020.

P. Sweetser. An emergent approach to game design – development and play. *Thesis. The University of Queensland*, 2005.

P. Sweetser and J. Wiles. Scripting versus emergence: issues for game developers and players in game environment design. *International Journal of Intelligent Games and Simulations, 4(1)*, pages 1–9, 2005.

J. Togelius, R. Nardi, and S. Lucas. Making racing fun through player modeling and track evolution. *Proceedings of the SAB'06 Workshop on Adaptive Approaches for Optimizing Player Satisfaction in Computer and Physical Games*, 2006.

J. Togelius, R. Nardi, and S. Lucas. Towards automatic personalised content creation for racing games. *IEEE Symposium on Computational Intelligence and Games, 2007. CIG 2007*, page 252–259, 2007a.

J. Togelius, R. D. Nardi, and S. M. Lucas. Towards automatic personalised content creation for racing games. *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium*, page 252–259, 2007b.

J. Togelius, G. Yannakakis, K. Stanley, and C. Browne. Search-based procedural content generation. *Applications of Evolutionary Computation*, page 141–150, 2010.

J. Togelius, E. Kastbjerg, D. Schedl, and G. Yannakakis. What is procedural content generation?: Mario on the borderline. *Proceedings of the 2nd Workshop on Procedural Content Generation in Games*, 2011.

J. Togelius, T. Justinussen, and A. Hartzen. Compositional procedural content generation. *FDG Workshop on Procedural Content Generation (PCG)*, 2012.

V. Vaishnavi, W. Kuechler, and S. Petter. Design science research in information systems. *January 20, 2004 (created in 2004 and updated until 2015 by Vaishnavi, V. and Kuechler, W.); last updated (by Vaishnavi, V. and Petter, S.), June 30, 2019.*, 2019. URL `http://www.desrist.org/design-research-in-information-systems/`.

Valve Corporation. *Left 4 Dead*. Valve Corporation, 2008.

R. Van der Linden, R. Lopes, and R. Bidarra. Designing procedurally generated levels. *Proceedings of the the 2nd AIIDE Workshop on Artificial Intelligence in the Game Design Process*, page 41–47, 2013.

J. von Neumann. The general and logical theory of automata. *Cerebral Mechanisms in Behavior*, page 1–2, 1951.

J. von Neumann. Theory of self-reproducing automata. *University of Illinois Press, Champaign, IL, USA*, 1966.

G. Wichman and M. Toy. *Rouge*. A.I. Design, 1980.

G. Yannakakis and J. Hallam. Entertainment modeling in physical play through physiology beyond heart-rate. *Affective Computing and Intelligent Interaction*, page 254–265, 2007.

G. Yannakakis and J. Togelius. Artificial inteligence and games. *Artificial Intelligence and Games*, pages 1–337, 2018.

G. N. Yannakakis and A. Paiva. Emotion in games. *Handbook on Affective Computing*, page 459–471, 2014.

J. Yannakakis G. N and. Togelius. Experience-driven procedural content generation (extended abstract). *In Proceedings of the 2015 International Conference on Affective Computing and Intelligent Interaction*, 2015.

D. Yu. Spelunky. boss fight books. 2016.

D. Yu and A. Hull. *Spelunky*. Independent, 2008.

D. Yu and Mossmouth. *Spelunky 2*. Mossmouth, 2020.

A. Zucconi. Let's build a computer in conway's game of life [video].youtube. 2020. URL `https://www.youtube.com/watch?v=Kk2MH9O4pXY&feature=youtu.be`.