Aalborg University Copenhagen

Semester: 10th semester

Title: Expanding Wave Function Collapse with Growing Grids for Procedural Content Generation

Aalborg University Copenhagen

Frederikskaj 12,

DK-2450 Copenhagen SV

Semester Coordinator:

Secretary:

Project Period: 1. feb 2019 – 28. may 2019

Semester Theme: Procedural Content Generation

Supervisor(s): George Palamas

Project group no.: ?

Members: Jonas Aksel Billeskov & Tobias Nordvig Møller

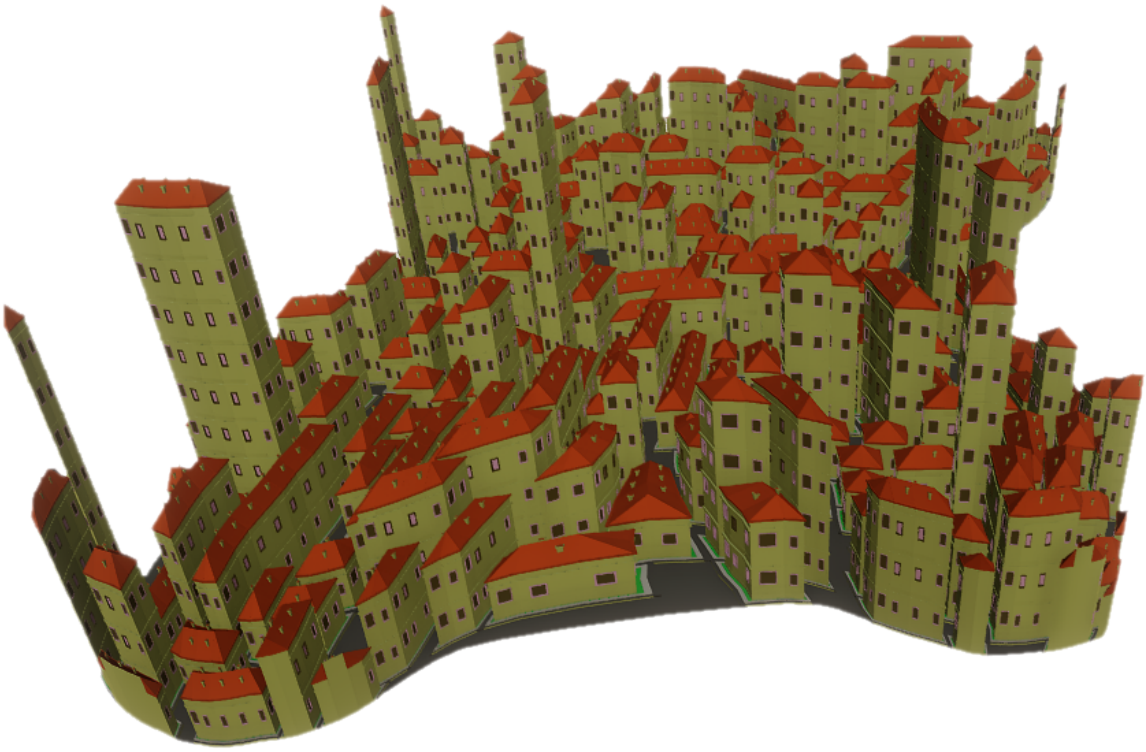
Abstract:

This project combines Wave Function Collapse and Growing Grids to create an improved procedural content generator that creates diverse output. A solution for traversal and feature distribution was also implemented. The results were evaluated by measuring the difficulty of navigating maps created. Another evaluation measured the recognizability of maps created with the system. A visual inspection was also done to showcase the capabilities. The results showed that navigation was significantly harder in maps created with the system compared to maps created with the original WFC algorithm. The results from the second test showed that it was slightly easier for the evaluation participants to recognize the maps created with the system compared to maps created with the original WFC algorithm.

Expanding Wave Function Collapse with Growing Grids for Procedural Content Generation

Aalborg University Copenhagen. May 2019

Tobias Nordvig Møller and Jonas Aksel Billeskov



Contents

1 Abstract

This project combines Wave Function Collapse and Growing Grids to create an improved procedural content generator that creates diverse output. A solution for traversal and feature distribution was also implemented. The results were evaluated by measuring the difficulty of navigating maps created. Another evaluation measured the recognizability of maps created with the system. A visual inspection was also done to showcase the capabilities. The results showed that navigation was significantly harder in maps created with the system compared to maps created with the original WFC algorithm. The results from the second test showed that it was slightly easier for the evaluation participants to recognize the maps created with the system compared to maps created with the original WFC algorithm.

2 Introduction

Procedural content generation (PCG) is the method for algorithmically creating assets, with popular installments including terrains, levels and items. Using PCG in video games as opposed to manually creating assets, has the benefit of allowing developers to produce larger amounts of content, and the possibility of games with endless content [1]. Some of the most exciting games in recent history have used PCG to create content, such as an entire universe in No Man's Sky. A novel procedural content generator is the constraint solver Wave Function Collapse (WFC). It is inspired by quantum mechanics' wave functions, where an unobserved state allows for all states to be possible, and observations constraints the possibilities. In WFC the states are a set of models created, where each model has a set of constraints, which limits what models can be placed next to it in the grid. A grid of cells is initialized, with each cell being unobserved and thus could contain all models. Slowly the grid is filled with models, constraining neighbouring cells until the entire grid is collapsed into an observed state, with only one model being possible in each cell. WFC is unique from many other PCG algorithms because it's a constraint solving algorithm. This makes the output from WFC unique and the process of building worlds with WFC is very satisfying. WFC is also interesting because there are no limitations as to what the models could be. It can be used for terrains, with models being hills, trees and rocks or be used for cities, with models being houses, roads or parks. In addition to this, you can place models, such as roads or buildings before running the algorithm and it will fill the grid around it. This makes it possible to handcraft parts of the world and let the WFC algorithm handle the rest. This project aims to expand upon WFC, by introducing irregular quadrilateral grids. These will be generated with Growing Grid (GG), which is a space filling algorithm designed to take a binary input image and distribute a structured grid accordingly. Using this algorithm in combination with WFC, any shape can be used to create diverse and interesting video game content.

3 Analysis

The analysis chapter will describe general PCG and different algorithms used in this field. It will describe state of the art PCG algorithms used in video games and how they might relate to this project. Following this, the chapter will go in depth with the WFC algorithm and state of the art applications of WFC. In relation to WFC the chapter will describe methods to expand upon WFC, with irregular grids, marching squares and ensuring traversability.

3.1 What is Procedural Content Generation?

PCG is a method of generating content using an algorithm as opposed to making the content manually. It is often used in relation to video games, computer graphics and motion pictures. Procedural implies that there is a procedure or algorithm needed to generate the content and while there exist many recognized methods for doing so, the algorithm can also be custom made to fit specific design requirements. The algorithm will often run once to generate an output, but can also run continuously to keep generating content [2]. The content in PCG is defined as levels, maps, game rules, textures, stories, items, quests, music, weapons, vehicles, characters, etc. [3]. Content is thus most of the items, which it takes to develop a video game and a video game can then in theory be completely procedurally made [4]. Generation implies that the algorithm results in an output of some sorts. In the case of PCG this would be some content, such as a map or an item. PCG can be used to generate content from scratch, to modify content or to combine content in a way to generate something new. The algorithm often involves some random modifier to ensure the output is different every time the algorithm is run. This modifier can be for example choosing a random seed to initialize the generation, user or designer input or a given game state. The algorithm can range from simple to complex as for example giving items random attributes or generating a universe with realistic planets. Even a simple algorithm such as random attributes for items can create millions of possible items, to for example add an aspect to a game of finding rare items with the perfect attributes. PCG have played a role in game design even before digital games [5] and many of the algorithms used today was not originally intended for video games. Some of the early work that is still used today is algorithms such as L-systems, which was made for describing plant growth [6], or Perlin noise, originally made for movies [7]. Two distinct approaches to PCG exist is online and offline generation [8]. Online generation means that the content is generated while using the media; while offline means that the content has been generated beforehand and could have been polished by developers. Both approaches have their pros and cons. Offline generation takes more memory for all the created content and can be more time consuming. It is however, a lot easier to control the output and remove bugs before the consumers see the content. Online creation makes it possible to generate a lot more content, but the algorithm needs to guarantee playable content. Another important distinction is the difference in PCG use, with three

categories; core, partial framing and decorational [8]. "Core" PCG media is when a core part of the media is procedurally generated, examples of this could be games such as No Man's Sky[9] or The Flame in the Flood[10]. In these games the planets and the river respectively plays an important role in the core gameplay. "Partial framing" is when parts of the media are made with PCG, examples of this could be the weapons in Borderlands[11], where the weapons are very central but the core gameplay is a FPS shooter. "Decorational" is when PCG is only used for decorating handmade content, alot of videogames use this to some degree. PCG algorithms can be categorized in six different categories [1]. Pseudo random number generators, generative grammars, image filters, spatial algorithms, modeling, simulation of complex systems and finally artificial intelligence where constraint satisfaction algorithms such as WFC are assigned.

3.2 Why use Procedural Content Generation in video games

Before getting into how PCG is used it can be useful to understand why it is used and for what cases it makes sense. As described later, PCG is used widely in video games but to accomplish very different goals. It is also important to consider what pitfalls and issues procedural generation can have. A common reason to use PCG is its ability to help the level designers and game artists create assets to avoid spending valuable work hours creating content with small variations. In the 2016 GDC talk "Forging the River in the Flame in The Flood" Damian Isla describes how procedural generation can be attractive to smaller game studies as a way to amplify their small art teams [12]. The book Procedural Generation in Games also emphasizes this as one of the main reasons for procedural generation but also mentions how procedural generation can be the foundation of a new genre of games with infinite content that never ends [3]. An exciting possibility in procedural generation is generating content based on player preferences [13][14]. If the game levels were generated with the players favorite types of gameplay in mind, these elements could be enhanced and improve the player experience. Implementing features such as this requires a high level of control over the PCG algorithms and requires the algorithm to run online. PCG can also allow for certain aesthetics or designs, which may not have been possible to achieve by manually creating such assets. This is for example seen when using Perlin noise for wavy terrains. Using PCG might even reveal designs, which were not considered initially and could influence the concepting of a production. Using PCG for decorational purposes is used in many different videogames and allows for improved details in areas such as backdrops and textures.

3.3 Procedural Content Generation in Video Games

This section takes a closer look into video games that uses PCG in novel and interesting ways. It will explore how PCG is used right now in the industry

and what tools game designers are using to model their procedural content to fit their vision and requirements.

PCG in video games have a long history going back to games like the original Rogue from 1980 that had the entire genre of video games Roguelikes, named after itself [15]. Rogue was a fantasy game where the player had to explore a procedurally generated dungeon that was different each time. This procedural system was implemented to keep the game fresh and exciting for multiple playthroughs as the game featured permanent deaths and therefore a whole lot of starting the game over.

Dead Cells

Dead Cells is one of the newer and more popular examples of a Roguelike. You are playing as an undead warrior trying to escape an infected island and in the end, kill the evil king. Dead Cells uses PCG for all its levels and some of its weapons. The interesting part about Dead Cells' use of PCG is how restricted it is. The level designer Sebastien Benard describes how in Dead Cells they only had PCG change very few things about their levels while handcrafting rooms [16]. Every room does however, have a list of things the PCG algorithm is allowed to change. They call this approach a hybrid between PCG and handcrafted levels. This approach was used because the developers felt that a more unrestricted use of PCG would decrease the quality of level design more than it would increase the replay ability and gameplay. Their algorithms are also focused on the difficulty of the game, where they control the power of the weapons you pick up, the number of enemies you encounter and the damage the enemies output. This makes the game challenging even on later playthroughs. The game has completion time for the entire game around 60 hours, which is a long game given a small studio and short development time. This is largely due to PCG as handcrafting that amount of content with the given circumstances would not be possible.

Figure 1: Procedurally generated dungeon in Dead Cells [17]

The Flame in the Flood

New video games in the RogueLike genre are still being developed and one of the more interesting titles in recent years is The Flame in the Flood. The game is developed by the small development team, the Molases Flood. Their game stays true to the original RogueLike features like permanent deaths and generated levels but instead of dungeons, The Flame in the Flood procedurally generates a river that the player navigates downstream. The river is generated by using gradient descent to place islands in the river. The game also uses flow calculations based on the width of the river and the islands to generate dynamic rapids and streams [12]. The Flame in the Flood highlights how PCG can be integrated into the core game mechanics of a video game. The procedural generation system allows control over how many rapids and flowing obstacles that are seamlessly integrated in the rivers. These tools lets the developers control the difficulty of the game and thereby creating more content that keeps being challenging for players.

Figure 2: Procedurally generated river in The Flame in the Flood [10]

Borderlands

An example of using procedural generation as a partial framing of the gameplay can be seen in the Borderlands series by GearBox games. Borderlands is a classic looter shooter that takes places in handcrafted levels but the weapons are mostly procedurally generated. This allows for the same levels to be replayed to get better gear and for trading between players to coordinate strengths and weaknesses in your team. In Borderlands, the millions of differently generated weapons each have a purpose because even small differences between weapons can be important when facing tough enemies. This is again, an example of how difficulty control can be implemented as part of the PCG. The procedural generation in Borderlands also creates an understanding between the players and the procedural generation, by categorizing the procedural content. Borderlands

have 12 different manufacturers of weapons, where each manufacturer has their own special strengths and weaknesses. The Dahl manufacturers' weapons have a high recoil reduction and low accuracy, for example. This system makes it possible for the players to familiarize themselves with the generated weapons and prefer one manufacturer over the other.

Figure 3: Procedurally generated weapons from the Jakobs manufacturer [11]

No Man's Sky

One of the most ambitious video games using procedural generation in recent years is No Man's Sky. No Man's Sky is a space exploration game by Hello Games where an entire universe is procedural generated. No Man's Sky received mixed reviews upon release and the game never really justified why 18,446,744,073,709,551,616 different but strikingly similar planets is better than a few handcrafted ones. IGN reviewer Dan Stapleton says "The promise of a limitless universe ended up working against it when I lost faith that it had meaning-full things to show me" [18]. This was however also the reason why the game received attention in the early days of development. The promise of infinite content and exploration was a big appeal for players around the world. No Man's Sky used many different novel techniques to create their world such as L-systems for vegetation, Warped Perlin noise for better landscapes and circular coordinate systems to enable their planets gravity to work as intended. No Man's Sky also controls parts of the difficulty with their procedural generation. They control how many resources are on each planet, the amount of enemies and the power of the spaceships and enemies.

Figure 4: One of the many procedurally generated planets in No Man's Sky [9]

Discussion of PCG in video games

All these games show how procedural generation can be used in different ways and how it can be used to achieve specific goals for game design and visuals. Dead Cells' combination of handcrafted rooms and PCG, The Flame in the Flood with their unique use of PCG, Borderlands with millions of weapons that are still recognizable and meaningfully different from each other, and No Man's Sky with their impressive scale and their use of procedural generation that allows for novel gameplay and narratives. Some concerns related to procedural generation can also be raised when exploring these games. Does No Man's Sky's almost infinite amount of planets make more sense than, for example, 200 partly handcrafted planets? Does Dead Cells really benefit from procedural generation when it's so restricted? These questions are somewhat similar as they both relate to the range of expressions and the diversity of the output the procedural generation can create. How much diversity does Dead Cells restrictions remove from the output? Does No Man's Sky's planets differentiate themselves enough from each other that they are all entertaining to explore? Another thing that are common across the games are how they use procedural generation to have greater range of difficulties in their game and in by that, creating more content for their players.

3.4 Wave Function Collapse

Using procedural content generators to create content in video games makes basis for interesting questions. It forces a developer to set up requirements for design and aesthetics before choosing or developing an algorithm. A lot of different algorithms are used in game development right now, where each has its pros and cons. A unique and novel algorithm is WFC, which stands out because of its modularity and expandability.

What is Wave Function Collapse?

WFC is a constraint based procedural algorithm that is inspired and named after the concept wave function collapse from quantum physics. In quantum physics wave function collapse is the idea that the unobserved state of a particle can be anything. As soon as the particle is observed, the possibilities disappear and the wave function collapses. The same idea is the backbone of the procedural algorithm. WFC can be implemented with two different models, the tiled model and the overlapping model. In this report, the focus will entirely on the tiled model. The WFC algorithm initializes a grid, where each cell in the grid will be defined as a slot. The WFC algorithm's tiled model variant occupies each slot in the grid with a module. A module contains information about the 3D model and the constraints for the module's neighbours. In the complete unobserved state, each slot has the possibility to be filed by every module possible. The modules have lists of possible neighbours they allow next to them in the grid. This means that if one slot is collapsed down to a single possible module, the neighbouring slots will also be restricted in possible modules because of the neighbouring collapsed slot. This constraint of possible modules then spreads to neighbouring slots, also constraining them in possible modules. WFC can be extended to high dimensions but the most common is two and three dimensions. For a two dimensional system, four neighbours is needed for a square grid and for three dimensions six neighbours. WFC can also be used in different grid shapes, square being the most common and the shape used in this report. Hexagon grids with six neighbours are also popular. Designing the set of modules that can fill out the grid can be a hard task, as they need to fit each other and allow some flexibility in possible neighbours. This area will be explained further later in the report. The algorithm for WFC described in the following section is from the original WFC GitHub by Maxim Gumin [19]. This version is modified to fit the tiled model and this project's implementation.

1. Create an array with the dimensions of the output, this array will be described as the grid but can also be described as the wave. Each element in the array contains a slot, each slot contains a list of possible modules, this list will be described as the possibility space.

2. Initialize the grid in a completely unobserved state, i.e. with all modules added to all slots' possibility spaces.

3. Repeat the following steps:

    (a) Observation:

        i. Find the slot with the smallest possibility space. If multiple slots are tied, select one of those at random. If all slots only have one module left in their possibility space or all modules only have zero modules left. Break the cycle and go to step 4.

ii. Collapse this slot's possibility space at random down to a single module. This is done by removing all but one module from the possibility space.

(b) Propagation: propagate the information gained on the previous observation step.

4. By now all the slots possibility space either contains exactly one module or they all contain exactly zero modules. If all slots contain zero modules, the algorithm has hit a contradiction and the result should be discarded. If the slots all contain one module, the algorithm is completed and the result can be returned.

A module contains four lists of modules that are allowed neighbours; a North, West, South, East list. For two modules to be placed next to each other they both need to have the other module in the respective neighbour list. If module one contains module two in its north list of allowed neighbours, module two needs to contain module one on its south list of allowed neighbours. If this is not the case the algorithm will most likely hit a contradiction.

A possibility space can never become larger; it can only decrease in size. When the algorithm starts propagating through the grid, possible modules for a slot can only be removed. As they are removed they start a chain reaction that reduces the size of neighbouring possibility spaces.

The propagation step propagates through the whole grid and checks if changes made in the observation step affect the possibility space of the neighbouring slots. A possibility space contains all modules that are allowed to be placed in the slot based on the module's allowed neighbours. There are two distinct approaches to this. One approach being an inwards approach and another being an outwards approach. In the inwards approach each slot checks its four neighbouring slots possibility spaces and derives what modules that are possible in the slot itself. This is done by looping through all modules in all four neighbouring slots possibility spaces. If these four possibility spaces have an identical module on their opposite neighbour list, the module can stay in the center slots possibility space. The outwards approach removes modules in the neighbouring slots possibility spaces instead, based on its own possibility space allowed neighbours.

The propagation step is the most expensive part of the algorithm and is recursive and dependent on the size of the grid. In the simplest approach it propagates through each single slot in the grid each time a change in a possibility space happens. It is not enough to only to loop through the grid once per propagation step, as a reduction in a possibility space caused by the loop can require new reductions in the neighbouring slots possibility space.

Step four is the last step when all possibility spaces have been reduced to their minimum size. If only one module is left in every slot's possibility space, the algorithm succeeded. This means that a full grid has been generated and all modules fit together with its neighbours. If zero modules are left in the possibility spaces the algorithm failed and hit a contradiction. Contradiction

happens when the algorithm works itself into a corner where some constraints in the module sets make no modules available for another slots possibility space. This can be caused by demanding module sets. In this report we define a demanding module set as a set where the modules have very small amount of neighbours. This could be the case if the visual style of the output should be very controlled or a specialized sub-module set with few transitions to the main set is added in.

It is perfectly fine to do a pre-collapse of a slot. If a specific module is needed on a specific slot because of design considerations or such, this can be done after step 2. Simply remove all other modules from the given slots possibility space and begin the algorithm with the propagation step. The algorithm will then build around the given module and works like if it had been collapsed randomly in step 3.a.ii. Pre-collapsing makes it possible to handcraft areas of output generated with WFC. This can be useful when designing dungeons for video games or similar content. For example, if a boss encounter needed multiple modules put together in a specific way to form an arena in the center of a dungeon; this could easily be done with WFC. Just pre-collapse the arena modules before running the algorithm and let the algorithm form the dungeon around the arena.

Previous Applications of Wave Function Collapse

WFC was initially developed as an algorithm for generating 2D procedural art by Maxim Gumin [19] (See Fig. 5). The algorithm received attention from animations of the algorithm solving different problems. The animations showed a natural way to draw procedural art and were enjoyable to watch, which pushed the popularity of the algorithm. The algorithm was later adapted for other scenarios such as 3D procedural art and game map generation.

Figure 5: Example of 2D procedural art using the overlapping model by Maxim Gumin [19]

WFC was quickly adapted to work in three dimensions and the possibilities of tilesets changed from sets of pixels to sets of 3D models. Maxim Gumin made an application with voxel shapes to show the capabilities of working in three dimensions [20].

Marian Kleineberg created an infinite world in three dimensions using WFC [21]. The world can extend infinitely in any direction the player chooses to go in. The tilesets consists of around 100 different modules, with custom constraints for each of them. The result of this is a fully traversable 3D city with stairs, houses and bridges.

The first use of WFC in video games was in the game ProcSkater developed by Joseph Parker and Ryan Jones at a game jam [22]. The algorithm was used for level generation and given a certain set of constraints, the generated levels were ensured to be traversable. Joseph Parker went on to use the WFC algorithm to create plateaus in the game Swapland and creating platforms in the game Bug with a Gun.

Game developer Oskar Stålberg popularized the algorithm further by exploring working with WFC in three dimensions, non-square shapes and creating beautiful tilesets. Oskar Stålberg demonstrates how working with WFC can be used to generate procedural spheres with custom tilesets, generate buildings extended infinitely into the air and generating small islands. The ladder is used in the real-time tactics game Bad North developed by Plausible Concept (See Fig. 6).

Figure 6: Example of 3D level generated by WFC in the game Bad North [23]

Andy Wallace created the game Maureens' Chaotic Dungeon at the Global Game Jam 2019, which utilizes an interesting feature of WFC [24]. The level is generated using WFC and on runtime you can regenerate parts of the level. This core gameplay mechanic allows the player to move around the infinite world and change the level using the existing constraints of the WFC.

WFC introduces a wide array of uses to generate levels, whether it is in 2D, 3D or to manipulate existing levels on runtime. The algorithm is versatile in its use and can be used with endless amounts of module sets to create exactly the aesthetic needed. This makes the algorithm interesting to work with and tweak as modifications to the algorithm can create basis for something completely different, but with the existing qualities of the original WFC algorithm. Given

its versatility, it is still seen that all the games mentioned use WFC with regular grids and evenly sized slots. There is big room for improvement in this field as the only requirement for WFC to work is that each slot has an equal amount of edges, and does not necessarily need to have a regular shape.

3.5 Irregular Quadrilateral Grids

One of the attributes of WFC is that it is organized in a regular grid, and depending on the module set, it may end up being very apparent in the final product. A certain module will always look the same given that all slots are the same size and shape and this limits the diversity of WFC. One way to combat this is to create an extensive module set, where even if the module will look the same, its occurrence in the final product will be low. But the modules will still be organized in a regular grid and will look thereafter. This is especially a problem if you want to utilize WFC to create natural looking products such as terrains and caves, where regular grids are a rarity. The following section will describe some methods that were discovered to create irregular quadrilateral grids.

Noise

The simplest method that was found to create a structured irregular quadrilateral grid was using noise. Every point in the grid would be distorted by a small amount to create some randomness and irregularities in the grid. The method worked well and did what it was supposed to, but the grid was still visible, since the center of each slot had not moved and was still in the same position. Using this approach was also limited since there was a maximum amount of noise you could apply to the points. If the noise became too high the points would overlap into neighbouring slots and the grid would be destroyed. The method could be salvaged by for example applying limits on the amount of noise based on the neighbours' noise. However it was quickly found that if this method should work it would require many modifications and the outcome might become uncontrollable.

Growing grid

Growing Grid (GG) is a self-organizing network developed by Bernd Fritzke in 1995 [25]. The algorithm takes in a 2D binary image as an input and organizes a grid to best cover the shape. GG is an expansion on the Growing Neural Gas also developed by Bernd Fritzke [26]. The intended purpose of the algorithm is as a space filling algorithm that makes it easier to use it in computer graphics than Growing Neural Gas as grids are very common in this field. The algorithm starts out with a 2x2 grid and goes through two phases known as "the growth phase" and "the fine tuning phase". The growth phase of the algorithm is set in epochs that iteratively chooses a random point in the distribution and drags the nearest point in the grid towards it. The other points will be dragged towards the selected distribution point too, penalized by their distance in the grid to the closest point. This process keeps looping until an arbitrary epoch size is reached. When this epoch size is reached, the algorithm will insert a new row or column in the grid. The row or column will be placed between the two points with the highest distance between them. Because of the distance between the closest point and the other points in the grid is calculated by a distance in the grid and not a Euclidean distance, the grid will relax itself in the distribution. When a defined max size of the grid is reached, the algorithm will enter the "fine-tuning phase." This phase is almost identical to the "growth-phase", where the only difference is that there will no longer be added rows and columns to the grid. When the algorithm has run for one final epoch, it terminates and returns the points from the grid. The growth phase is the main phase of the algorithm and can be split into eight steps according to an explanation of by Bernd Fritzke [27]. These steps will be explained further in the implementation where examples from the Unity3D C# implementation can also be seen.

The input image thus decides how you want to use the algorithm. Using a simple shape, such as a circle as an input image will leave you with an approximation of a circular grid (see fig. 7).

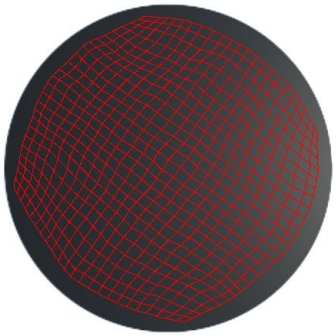


Figure 7: Circle input for growing grid

Using something more complex, such as a heightmap can let you control the density of the grid i.e. size of each slot, while also being able to control the shape of the final grid (see fig. 8).

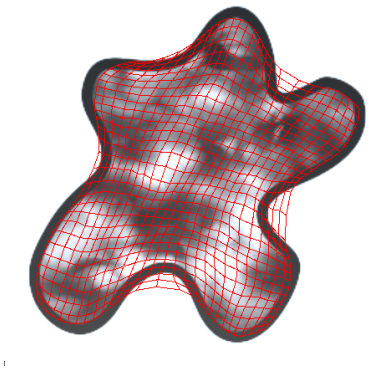


Figure 8: Shape with gradient input for growing grid

The algorithm is not very intuitive, as there are many random variables in the algorithm and it takes some playing around with it to understand how it works. Some inputs create stretched grids and large slots, while others create even distributions. In addition to this, the same input does not necessarily ensure the same output. However the algorithm will always ensure a structured grid, which allows WFC to work without modifications to the algorithm.

Alternative space filling algorithm

Oskar Stålberg made an algorithm, which given a 2D shape will fill it with an irregular quadrilateral grid. A demonstration of this was released on his Twitter profile, where the algorithm was used to generate cities with WFC (see fig. 9) [28]. However there was no documentation on the algorithm, so the following explanation is based on a short talk with Oskar and deductions from the demonstrations.

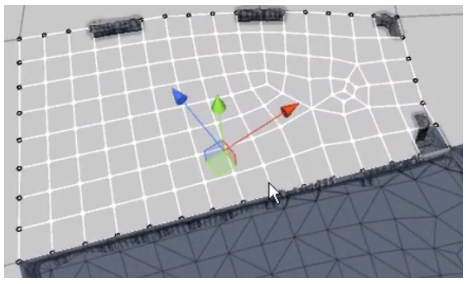


Figure 9: Alternative space filling algorithm [28]

The algorithm will take the corners of the input shape and invert those inwards to create the corner squares of the grid. It will thereafter create squares following the edge of the input shape between the corners to create the edges of the grid. This is repeated until no longer possible and the remaining area is triangulated to create an even amount of triangles, which are combined in pairs to create quads. All slots are then subdivided if needed to create a uniform size of slots.

The algorithm has a lot of potential for map generation given the strict constraints it is able to solve and seemingly no constraints to what kind of input it takes. The algorithm will allow a designer to create maps in any kind of shape and size and combinations of shapes. Given the grid structure of the output it will allows WFC to work seamlessly and other operations, which works on grid. For example, in the demonstrations we see how the edges of each grid are used to generate roads and roadside buildings, while the inside of the grid is used to fill out the buildings.

Discussion of irregular quadrilateral grids

Using irregular quadrilateral grids in combination with WFC can be used as an extension to WFC as it can be combined to work seamlessly, but remove some of the regularity which is present in WFC. Using noise to break the regularity of the grid could be useful, but the grid would overall always be square. Noise could however be used on top of another system to create further randomness in the final grid. Using the GG algorithm lets a designer define the shape of the grid as well as the density of the slots. In addition to this the GG algorithm can be adjusted using the predetermined variables and working with different inputs can create interesting results. This makes it a versatile algorithm to use and will be able to create diverse and interesting grids. A downside to the GG algorithm is that the grid might end up different given the same input and the grid will always be a loose fit to the input shape. Oskar Stålbergs algorithm is very precise and reliable in its output based on input and would let a designer get exactly what was asked for. However the algorithm lacked natural shapes and most cells in the grid looks regular. For these reasons it was chosen to implement the GG algorithm to create irregular quadrilateral grids in this project.

3.6 Marching squares

Designing a module set requires planning as the module should fit together and have constraints, which avoids contradictions. Inspiration can be drawn from an algorithm named marching cubes. Marching cubes is an algorithm originally developed for processing three dimensional medical data such as CT and MR scans[29]. In this project the two dimensional version called marching squares will be utilized. Marching squares is a two dimensional algorithm that can be used to calculate vertices' positions for contouring in computer graphics [30]. In this project the marching squares system will be used mostly as an inspiration

to formulate a system of modules that fit together. Marching squares could also have been used to implement flexible module shapes. In marching squares, a square is defined by its four corners. Each corner can be either solid or not. As squares in a larger grid of squares will share corner points between each other, the values can then be interpolated to create smooth contours for the solid area in the grid. This technique is commonly used to create smooth output from binary input produced by other procedural algorithms. Video games use this for many different purposes, such as caves or destructible environment.

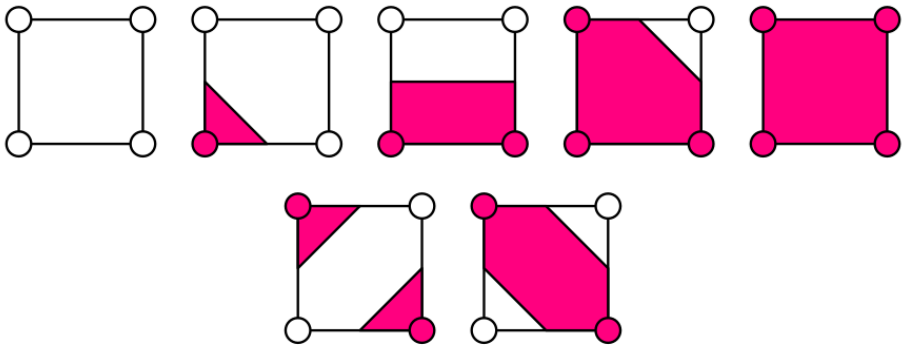


Figure 10: Marching squares corners
[31]

By defining each square by its corners and as seen in figure 10, there are six different possible squares if each square can be rotated freely. Both squares in the second row are defined by the same corners and are ambiguous, if both are implemented that totals in seven different squares. Only five of the seven squares need rotation as both completely solid and completely non solid is the same regardless of rotation. The two diagonal squares in the second row only need one rotation. That totals to 18 different squares for a complete set. This set of squares is because of its relation to two dimensional games, known as the "marching square tileset". This tileset can be expanded further into The Blob tileset and reduced into the Micro-blob tileset. The Micro-blob tileset coined by BorisTheBrave [32], is almost identical to the marching squares tileset. The only difference is that both ambiguous diagonal tiles are removed. This means that there are only five unique tiles remaining for the tileset and a total of 14 tiles including rotated tiles. This makes creating working tilesets for WFC a simpler task as the number of unique tiles are low and neighbourhood constraints can also be defined according to this system.

3.7 Traversability

An important aspect of making a video game level is making sure that the player is able to reach the goal of the level from the starting position [33]. Making sure that the level allows the player to traverse through the level all the way to the goal is an easy task when designing a level by hand, but when the level generation is procedural it will require some rules. Procedurally generated levels may end trapping the player in the starting position or locking off the goal of the level. This would make the level unplayable and thus not fit for a game. In many games with procedurally generated levels there is no set objective, such as Minecraft or No Man's Sky, which nullifies this problem. However in games such as Diablo, Path of Exiles and Dead Cells, there is a need for the player to get from point A to point B, which makes creating a traversable path between the points a concern. The solution for this will often be specific to the underlying procedural generation algorithm and a specific set of rules can not be set for all games. The following sections will explain some of the different techniques used to guarantee traversable levels in PCG.

Random Walk Cave Generation

One of the simplest methods of guaranteeing a traversable level is using the Random Walk algorithm [34]. The Random Walk algorithm works by having a grid of solid cells to start off with. The next step is to choose a starting cell and changing it into a non-solid cell. Now the algorithm proceeds to take a step in a random direction and if it is a solid cell it will change it into a non-solid cell, until the amount of non-solid cells meets the requirements. The Random Walk algorithm thus guarantees that the traversable part of the level is contained within a single mesh and every point of the mesh can be reached from any other point of the mesh. Given the simple nature of the algorithm it is not possible to set certain constraints, such as an entrance and exit, since you are not certain that the algorithm will reach the exit. Objectives such as enemies, resources or an exit could however be placed after the algorithm has carved out a traversable level.

Caves in Minecraft

The caves in Minecraft are made based on an algorithm similar to the Random Walk algorithm. The algorithm is called Perlin Worms and carves out caves out of a solid with a metaphorical worm moving around [35]. The direction of the worm is determined by a 3D Perlin noise and changes after the worm has moved a set length. Compared to using 3D Perlin noise to generate caves, this algorithm ensures that each cave created by one worm is connected all the way through, where 3D Perlin noise often leaves unconnected chunks of caves.

Rooms in Path of Exiles

The rooms in Path of Exiles are structured using graph theory combined with branching [36]. The levels in Path of Exiles consist of a set of square rooms organized such that you can exit and enter all rooms, and there is a path from the entrance to the exit. The first step to making a level is placing the entrance and the exit rooms. The next step is populating a grid with different rooms from a library of rooms. These rooms have different sizes and shapes and are assigned a weight, determining the difficulty of the room. All rooms in the grid are placed to be traversable, such that the entire level can be navigated. Now using a shortest path algorithm a route is determined from the entrance to the exit based on length and difficulty of the rooms. All rooms not on the route are now removed and the final step is to branch out from the path and place rooms neighbouring the route. The algorithm ensures that the player experiences minimal backtracking, while not having a straightforward path to the exit.

Chiseling

Chiseling is an algorithm introduced by Boris the Brave and is used to ensure a path between two or more points [37]. The algorithm expects a grid of nodes and while it is originally implemented in 2D it also works in 3D. The algorithm works by first placing the points, which should be connected with a path and then populating the rest of the nodes with traversable objects. Now the algorithm iteratively removes the traversable objects at random while ensuring that a path between the points is still intact. This is ensured using a depth first search, which allows removal of an object if all the placed points have been visited. This is repeated until no more nodes can be removed, without breaking the path. The algorithm thus ensures a path between predetermined points, which could for example be entrance and exit. It produces a semi random output, but unlike some other algorithms it allows predetermined constraints.

Discussion of traversability implementations

While there are many ways to ensure a traversal level, not all of them are suited to be added on top of an existing system. Many of the algorithms are specifically designed to fit the underlying procedural generation algorithm. This also means that some algorithms have constraints, which fit the theme of the game, such as in Minecraft, where the algorithm does not set a start or an end to the caves, but rather ensures aesthetically pleasing caves and a minimal amount of small closed caves. Given that WFC is a constraint solver it makes sense to using an algorithm, which allows constraints in the path generation. This makes chiseling an excellent candidate since it constraints as many points as you wish and works well in a grid. Given that you know the size of the grid and placement of keypoints you can run the chiseling algorithm before running

the WFC algorithm. This ensures that you get an organic looking level with a traversable path between keypoints.

3.8 Related Works

A common research topic in the PCG area is the topic of control vs. variety. Gillian Smith and Jim Whitehead names the controllable interval of a procedural algorithm "The expressive range"[38]. Shaker et. al. explores this concept with the use of Super Mario levels[39]. A related challenge when working with and researching procedural generation is how to evaluate the output of an algorithm. Horn et. al. creates a benchmark test for evaluation of procedural algorithms [40]. Cameron Browne and Frederic Maire defines metrics for automatic game design that will interest human players [41].

Isaac Karth and Adam M. Smith formulate WFC as a constraint solving algorithm and describes how it has affected areas such as generative art and procedural content for games. [42].

Hugo Scurti and Clark Verbrugge explore how paths can be generated in WFC levels. Their modification allows for input that gets transformed into paths in their generated levels. Their modification in based on the overlap model of WFC [43].

Wulff-Jensen et. al. describes a way to generate greyscale heightmaps for video game levels based on real world satellites images using a DCGAN network [44]. By using these heightmaps in combination with the irregular grids and feature distribution discussed in this paper, game content based on the real world could be created with WFC.

Ian Horswill and Leif Foged proposes a method to create procedural levels for video games using constrained paths [45]. By adding paths with constrains to their levels they can control the placement of things like power-ups before challenging boss battles or not misplace keys behind the doors they are supposed to unlock. One of their concepts involve calculating how many hit points each enemy will on average damage the player and they can therefore create levels with more advanced topology and still balance the many different paths through the levels similarly. It seems like it could be possible to combine these concepts with WFC to create better balance in the output.

Adrian et, al. describes a method to control the difficulty of procedurally generated games using a difficulty curve [46]. They use a fitness function that compares the difficulty of the level to the designed difficulty curve and chooses the best candidate. They have created a racing game, where the difficulty is measured based on how many empty tiles and obstacles there are in a segment of the level. In additions to this they conclude that the fitness function is not dependent on the game type or content and can be applied to most games.

Biggs et. al. finds that navigation in video games can be related to subtle patterns in the level design and tries to use this in procedural generation for their own game [47]. They evaluate the project by having test participants walk around their generated levels to see if they could use the patterns for navigation. Creating irregular quadrilateral grids for WFC with GG could possibly create

levels with unique features that could work in a similar way as the discussed patterns.

Daniel R. Montello shows that navigating urban fields with parallel and orthogonal streets are easier than oblique streets[48]. In this project, using the combined WFC, GG and chiseling could likely produce output with non-orthogonal pathways. This could suggest a relation between navigation difficulty and irregular grids.

4 Final problem statement

Based on the analysis, the requirements for this project can be set and combined into the final problem statement. We have identified key factors for PCG success in the video game industry, explored the strength and weaknesses of WFC and explored the PCG research area.

Final Problem Statement:
How can Wave Function Collapse be expanded using Growing Grid to create recognizable output with diverse difficulty?

5 Implementation

The implementation chapter will discuss how the different elements of this project were developed. The elements are, WFC, the different module sets, optimizing the algorithm, feature distribution and GG.

5.1 Implementation of Wave Function Collapse

The WFC algorithm is implemented in Unity3D and is written in C#. The implementation can be run with different module sets, which are all created in Cinema4D. Most of the Unity implementation of WFC is contained the script called GridManager. The GridManager script is then placed on a null object in the scene. This chapter will follow the same structure as the WFC algorithm overview in the analysis. The implementation allows for both two and three dimensional output. The implementation below will be described in three dimensions.

Step 1: Initialize the grid

Initializing the grid happens in a nested for-loop that takes in four points for each slot, where some slots share points. These points are assigned from the GG algorithm that runs before the WFC algorithm. A root position is also assigned to each point based on its position in the loop (see fig. 11).

```
public void InitializeSlots()
{
    int index = 0;
    for (int i = 0; i < gridX; i++)
    {
        for (int k = 0; k < gridY; k++)
        {
            for (int j = 0; j < gridZ; j++)
            {
                slot tmpSlot = new slot(pointgrid[i,k,j], pointgrid[i+1, k, j], pointgrid[i, k, j+1], pointgrid[i+1, k, j+1]);
                grid[i, k, j] = tmpSlot;
                if (enableChiseling) grid[i, k, j].isPath = true;
                grid[i, k, j].index = index;
                grid[i, k, j].pos = new Vector3(i, k, j);
                index++;
                mL.randomPool.Add(index);
            }
        }
    }
```

Figure 11: The loop initializing new slots with points from the Growing Grid algorithm

After the slots have been initialized each slot is assigned its six neighbours based on grid positions. There is no border padding as a slot can have less than six neighbours and is therefore only assigned its neighbours if they exist.

Step 2: Initialize the grid in unobserved state

Each slot has a list of all modules in a module set added to its own possibility space. The list of modules is retrieved from a scriptable object list that makes it easy to add and remove modules from the scriptable object in the Unity Editor. If the slot has been selected for a path by the chiseling, instead of adding the whole list of modules, only a single floor module is added. This results in the slot being precollapsed into a floor module. The modules contain the 3D model they eventually need to instantiate when the algorithm finishes and they contain six lists with the neighbouring modules they are allowed to have (see fig. 12).

```
//initialze grid
for (int i = 0; i < gridX; i++)
{
    for (int k = 0; k < gridY; k++)
    {
        for (int j = 0; j < gridZ; j++)
        {
            grid[i, k, j].posibilitySpace.Clear();
            if(grid[i, k, j].isPath)
                grid[i, k, j].posibilitySpace.Add(floorGO);
            else
                grid[i, k, j].posibilitySpace.AddRange(modules);
        }
    }
}
```

Figure 12: The loop loading the slots possibility spaces with modules

Step 3: Repeat the following

Steps 3.a and 3.b are an iterative process, which will run as long as any slot contain more than one module or contains zero modules in their possibility space.

Step 3.a: Observation

The first part of the observation step is a straightforward check of what slot have the fewest modules left in their possibility space. The second part is a collapse of the possibility space of the slot with the smallest possibility space in the grid. This part is located on the slots own script and collapses the possibility space down to a single random remaining module. In the current implementation, it only removes a single random module from the possibility space per collapse call. It does so to allow a better control over the process while in combination with the improved way of propagates does not decrease the computational speed significantly. Different approaches to the collapse step were tried, such as collapsing a random slot with the largest possibility space. This resulted in an increased number of contradictions when running the WFC

algorithm. One explanation for this could be that it is harder for the algorithm to work itself systematic forward through the grid. Maxim Gumin original explanation for the use of smallest possibility space was "I noticed that when humans draw something they often follow the minimal entropy heuristic themselves. That's why the algorithm is so enjoyable to watch "[19]. But maybe he also stumbled upon the approach creating the least contradictions. When there is only a single module left in a possibility space, that module's 3D model gets instantiated at the slots position.

Step 3.b: Propagation

As described in the algorithm overview, this implementation opts for an inwards propagation instead of an outwards propagation (see fig. 13). This part of the code is also located in the slots own script. The idea is that each slot runs through its neighbours' possibility spaces and checks those modules' opposite neighbours for duplicates. A slot finds the allowed neighbours to the north by checking the neighbours allowed south of the slot located north of it. One list is created for each neighbour and those lists are then cross referenced for duplicates. So only if a module exists on all neighbour lists, the module gets to stay in the possibility space. The propagation is done in a loop through the entire grid and keeps looping until there are no more changes to any slots possibility spaces. As soon as there is a reduction in a single possibility space, another propagation loop is called. When there is no more reductions in possibility spaces, step three restarts again until all slots possibility spaces size are one or zero.

```
for (int i = 0; i < neighbours.Length; i++)
{
    if (neighbours[i] != null)
    {
        for (int k = 0; k < neighbours[i].posibilitySpace.Count; k++)
        {
            if (i == 0) neighbour0Pos.AddRange(neighbours[i].posibilitySpace[k].GetComponent<Modulescript>().neighbourSouth);
            if (i == 1) neighbour1Pos.AddRange(neighbours[i].posibilitySpace[k].GetComponent<Modulescript>().neighbourWest);
            if (i == 2) neighbour2Pos.AddRange(neighbours[i].posibilitySpace[k].GetComponent<Modulescript>().neighbourNorth);
            if (i == 3) neighbour3Pos.AddRange(neighbours[i].posibilitySpace[k].GetComponent<Modulescript>().neighbourEast);
            if (i == 4) neighbour4Pos.AddRange(neighbours[i].posibilitySpace[k].GetComponent<Modulescript>().neighbourUp);
            if (i == 5) neighbour5Pos.AddRange(neighbours[i].posibilitySpace[k].GetComponent<Modulescript>().neighbourDown);
        }
    }
}
```

Figure 13: The calculation of the slots possibility space.

Step 4: Cleanup

When step three is done every slot's possibility space size is either one or zero. If the size is one, the algorithm succeeded and every slot should have instantiated a single 3D model in its position. If the size is zero, a contradiction

has happened and the algorithm failed. If there is no contradiction failcase, Unity will usually crash and the application has to be restarted. But adding a simple method to check for zero sized possibility spaces and calling it in the propagation step solves this. When the content generation is done, this project does a few things that are not integral pieces of the WFC algorithm but is almost always going to be useful in any real application. First, a recalculation of the normals of each instantiated model happens. If this is not done, and the meshes have been changed, it can result in very strange looking output. The mesh is also made static so a navmesh can be calculated, this allows NPC movement and simplified player movement on the generated map. These things are only needed if the WFC algorithm is going to be used for online generation as opposed to offline pre generated content. The implementation is setup to use coroutines. Using this allows for the WFC algorithm to be spread out over a suitable amount of frames and thereby prevent the process from crashing instantly. It also makes debugging easier as it is possible to follow the collapsing and propagation process visually when using coroutines.

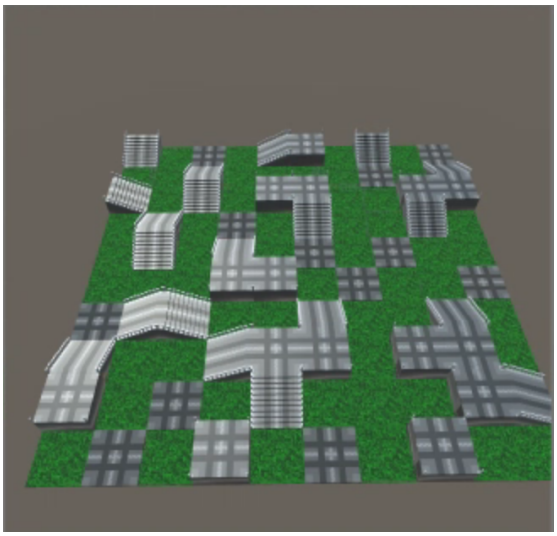


Figure 14: The first level created with Wave Function Collapse.

5.2 Optimizing the propagation step

The propagation step is the most expensive part of the WFC algorithm and its running time is dependent on the size of the grid. Optimizing the propagation step is appealing because the most basic approach that was implemented was ineffective, as it recursively recalculated every slot in the grid multiple times per propagation step. Each time a change in a slots' possibility space was made, another recalculation of the entire grid of slots were made to see if any changes should be applied because of this. To optimize this, an approach inspired by the

grass fire algorithm is now implemented. It is given from the WFC algorithm that a possibility space can only change if, it is being changed directly in the observation step or if its neighbouring slots possibility spaces change. Because of this, it is possible to know exactly what slots that can change without recalculating the entire grid. Note that, it is possible to know if a possibility space can change but not possible to know if it will change without calculations, as a smaller possibility space in neighbouring slots not necessarily decreases the original possibility spaces size. Each slot now contains a Boolean that holds information about whether it should be recalculated or not. This Boolean is true if a change in any of its neighbouring slots possibility spaces have happened since its own last recalculation. If a slot gets recalculated, it sets the neighbouring slots Boolean to true. The propagation steps still needs to check the whole grid for this Boolean and the exit condition, which means that the number of loops through the grid remains the same. But it no longer needs to recalculate every slot in the grid. The practical benefit from this change was not documented but the effect made it possible to calculate grids many times larger than originally without crashing Unity.

5.3 Contradictions and attempts at reinforcement learning

Contradictions rarely happen in WFC but can become more common if the module set is demanding. In this project's effort to create traversable maps with WFC a set of modules was added to the original set of micro-blob modules. This created contradictions in a large amount of the test runs that crashed the system. The common approach to solving this problem is implementing a backtracking mechanism that stores a list of the latest actions the WFC algorithm has executed. If a contradiction is then reached, the algorithm will backtrack an arbitrary amount of steps and start over from that point. In an early effort to explore the possibility of using reinforcement learning to fine-tune maps according to a fitness function, a system was implemented to un-collapse slots. This report defines un-collapsing as the action of adding removed modules back into the slots possibility space. This restarts the WFC algorithm in the un-collapsed slots and possibly creates a different final result. In the attempt at fitting the generated maps to a given fitness function, a sub system was defined. A complete map was generated, saved and measured against a fitness function. If the error was above a threshold, a randomly placed subgrid of 4x4 slots would be un-collapsed and re-collapsed. If the fitness of the map with the recollapsed area was higher than the original map, the new map would be saved, replacing the original map. If not, the re-collapsed map would be discarded and the system would return to the original map before the un-collapse happened. If the error was still above the threshold the process would be repeated on a new randomly placed subgrid not equal to an earlier subgrid. If the whole map would be iterated and the error was still above the threshold, the process would start over. The idea was that this slowly would tune the map to a fitness function, and worked somewhat, it just took extremely long time; way too long to ever

imagine it being implemented in an online application. The idea was discarded after this but the uncollapsing subgrid is still used to handle contradictions. If a contradiction happens, a sub-grid of 4x4 slots with the source of the contradiction in the center of the sub-grid will un-collapse and re-collapse. If it does not solve the contradiction the process will repeat until the contradiction is solved.

5.4 Feature distribution

Controlling the distribution of modules in WFC without compromising the core algorithm is challenging as the constraint based neighbourhood system calculates the possible modules on a local neighbourhood and does not calculate any global data. Common solutions to distribution in WFC are weighting the modules differently when collapsing the slot. An example could be to make floor module have a higher chance of being selected than a wall module when they both are possible. The approach has the issue of not using the weights when the algorithm collapses modules, when only one module is available for the slot. The weights only affect the selection when the algorithm is done iterating and multiple possible modules are available for all slots. This means that if your module set is demanding, the weighting will have very little effect. The weights will then not control the overall distribution of modules, but rather the distribution of the randomly collapsed modules, which may not be a large quantity of the output.

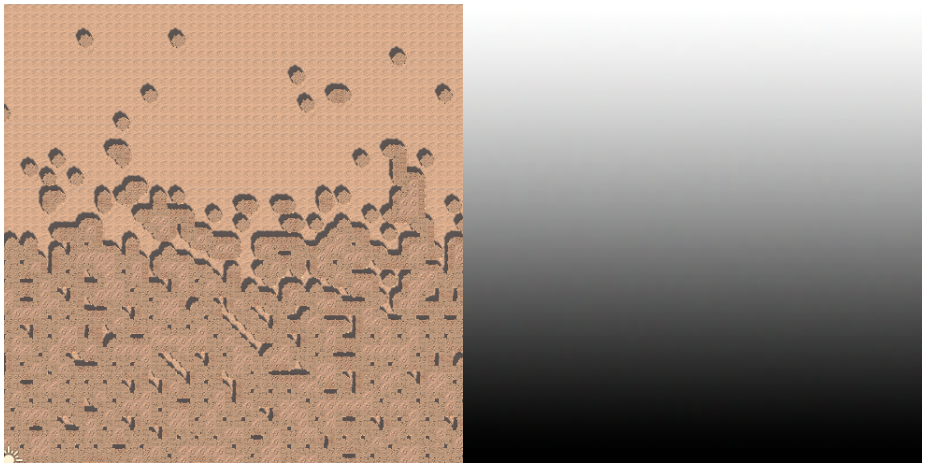


Figure 15: Gradient input image and output from WFC with weighted feature distribution

It also makes transitions difficult as a system can not be affected by weights if the transition between two weights is not an allowed neighbour. I.e. if a solid module is placed and the weights dictates a floor which is not allowed to

neighbour it, the system will have no incentive to place a module that makes a transition possible for the following slots. By utilizing the grammar system this project assigns weights based on their neighbourhood grammar code. This means that instead of weighting tiles directly, the tiles are weighted based on their grammar code. This means that if the weight of a specific slot dictates that a floor should be placed, but it is not allowed due to the neighbouring slots, it will place a module, with a grammar code, which allows a floor to be placed next to it. This system will allow the WFC to better transition into the modules that are dictated by the weights to be placed. When the WFC was run without this system and weights, it was seen that the modules placed would cluster together and not allow the weights to work. I.e. if a solid module is placed there would be a cluster of walls and corners around it and the WFC would have a hard time breaking out of it and transitioning to a different type of tile, such as a floor. The system was implemented to take in a binary image, which would control the density of the module placed. To control this a density factor would have to be calculated for each module, such that it could be matched to the input image. The density factor was calculated based on each module's grammar code and how many of its neighbours are allowed to be floor tiles. For example a floor module is allowed to have four floor modules as neighbours and thus it will have a low density factor while a solid module is allowed to have 0 floor tiles as neighbours and will then have a high density factor. In between are the corner module, which is allowed two floor modules and the wall module, which is allowed one floor module. For each floor module allowed, 0.25 is added to a variable, giving for example a floor module a value of one and a wall module a value of 0.25 (see fig. 16).

```
for (int i = 0; i < modules.list.Count; i++)
{
    modules.list[i].GetComponent<Modulescript>().weightModifier = 0;
    for (int j = 0; j < modules.list[i].GetComponent<Modulescript>().neighbours.Length - 2; j++)
    {
        if (modules.list[i].GetComponent<Modulescript>().neighbours[j] == 0) modules.list[i].GetComponent<Modulescript>().weightModifier += 0.25f;
    }

}
```

Figure 16: Code snippet for calculating density factor

Next when collapsing a slot the density factor is taking into account in choosing, which module is collapsed. This is done by assigning a weight to each module depending on the value from the input image and the density factor. The weight is calculated by taking a linear interpolation between (1 - the density factor) and the density factor by the value form the input image. This is then lifted to the power of eight to give us the final weight score (see fig. 17). Given an input value of 0.8 will then result in a floor tile having a weight of

$$((1 - 0.8) * (1 - 1) + (0.8 * 1))^8 = 0.16777$$

and a wall tile having a weight of

$$((1 - 0.8) * (1 - 0.25) + (0.8 * 0.25))^8 = 0.00023$$

While an input value of 0.2 will give a floor tile a weight of

$$((1 - 0.25) * (1 - 1) + (0.25 * 1))^8 = 0.00002$$

and a wall tile having a weight of

$$((1 - 0.2) * (1 - 0.25) + (0.2 * 0.25))^8 = 0.03186$$

```
for (int i = 0; i < posibilitySpace.Count; i++)
{
    posibilitySpace[i].GetComponent<Modulescript>().weight =
    Mathf.Pow(Mathf.Lerp(1 - posibilitySpace[i].GetComponent<Modulescript>().weightModifier,
    posibilitySpace[i].GetComponent<Modulescript>().weightModifier, noiseWeight), 8);
}
```

Figure 17: Code snippet for calculating weight when collapsing a slot

In the micro-blob tileset there exists four different density factors and these can be plotted to see the correlation between input values, density factor and the calculated weight (see fig. 18).

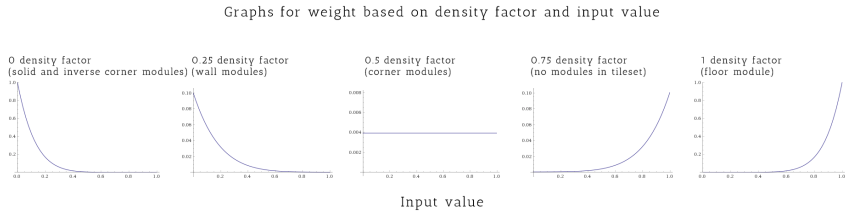


Figure 18: Calculation of weights plotted for each density factor

Each module in the possibility space then has a weight and the module to collapse is chosen with a weighted random selection (see fig. 19).

```
float weightSum = 0;
int moduleToCollapse = 0;
for (int i = 0; i < posibilitySpace.Count; i++)
{
    weightSum += posibilitySpace[i].GetComponent<Modulescript>().weight;
}

float random = Random.Range(0, weightSum);

for (int i = 0; i < posibilitySpace.Count; i++)
{
    if (random < posibilitySpace[i].GetComponent<Modulescript>().weight)
    {
        moduleToCollapse = i;
        break;
    }
    random -= posibilitySpace[i].GetComponent<Modulescript>().weight;
}

GameObject tmpModule = posibilitySpace[moduleToCollapse];
posibilitySpace.Clear();
posibilitySpace.Add(tmpModule);
```

Figure 19: Code snippet for weighted random selection

The feature distribution can be used to create open and closed spaces as it was done it the examples presented in this section. However the features can be expanded to be much more than this. The feature could for example be enemies or power-ups for the player, which could be used to control the difficulty of the level.

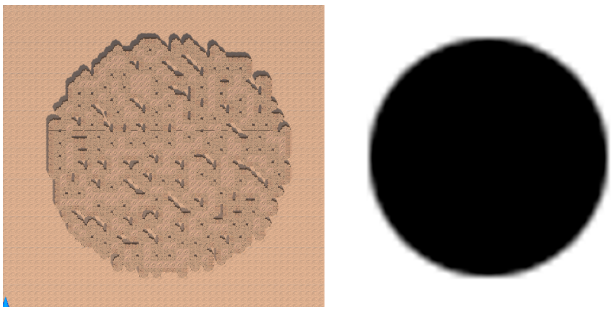


Figure 20: Circle input image and output from WFC with weighted feature distribution

5.5 Implementation of Growing Grid

The implementation of the GG algorithm was done following the description from Bernd Fritzke [27] [25] In addition to this, the source code for the Java implementation on the demonstration website was also used. All implementation was done in Unity3D and written in C#. The implementation was done in two dimensions but it could be adapted to more.

The first step of the algorithm is to initialize the different values of the algorithm. The distribution is created based on an input images pixels grey scale values. The starting size of the grid was mostly set to 2x2 but can be anything lower than the max grid size. The position of the points is chosen at random from the distribution. Four other variables are also set, the max number of points, the neighbourhood range, the epoch size and the learning rate. The second step is to select a random point from the distribution. The second step is also the start of an iterative process, which goes through step 2-8. The third step finds the closest point in the grid to the random point based on Euclidean distance (see fig. 21).

```
public Point CalculateWinner()
{
    randomPoint = distributionPoints[Random.Range(0, distributionPoints.Count)];
    Point tmpPoint = new Point(new Vector3(0,0,0),0,0);
    float tmpDistance=10000f;
    float distance=0;

    for (int i = 0; i < xSize; i++)
    {
        for (int k = 0; k < ySize; k++)
        {
            distance = Vector3.Distance(points[i, k].position, randomPoint.position);
            if (distance< tmpDistance)
            {
                tmpPoint = points[i, k];
                tmpDistance = distance;
            }
        }
    }
    return tmpPoint;
}
```

Figure 21: Code snippet for calculating the closest point

The fourth and fifth step increment a local counter on the winner and a global time counter. These values will be used later. The sixth step adapts each point by changing its position in the direction towards the random point

depending on learning rate and adaption strength (see fig. 22).

```
public void Adapt()
{
    for (int i = 0; i < xSize; i++)
    {
        for (int k = 0; k < ySize; k++)
        {
            Vector3 direction = randomPoint.position - points[i, k].position;
            points[i, k].position += learningRate*AdaptationStrength(winningPoint, points[i, k]) * direction;
        }
    }
}
```

Figure 22: Code snippet for adapting each point

The learning rate is a modifier to the distance changed, which ranges from 0 – 0.1. The adaption strength is calculated for each point based on the distance between the point and the winning point in terms of their index in the grid and the neighbourhood range (see fig. 23). This means that the closest point in the grid to the random point will move towards the random point as well as a certain neighbourhood around the winning point.

```
public float AdaptationStrength(Point point1, Point point2)
{
    float d = Distance(point1, point2);
    return Mathf.Exp(-d * d / (2 * neighbourhoodRange * neighbourhoodRange));
}
```

Figure 23: Code snippet for calculating adaption strength

```
public float Distance(Point vector1, Point vector2)
{
    return Mathf.Abs(vector1.xIndex - vector2.xIndex) + Mathf.Abs(vector1.yIndex - vector2.yIndex);
}
```

Figure 24: Code snippet for calculating the distance between points

The seventh step is executed if the global time counter has reached a chosen epoch size. If not the algorithm continues with from step two again. If the epoch size has been reached you find the point, which has been the winning point most frequently (see fig. 25).

```
public Point FindPointWithMostWins()
{
    int tmpCount=0;
    Point tmpPoint = new Point(new Vector3(0, 0, 0),0,0);

        for (int i = 0; i < xSize; i++)
        {
            for (int k = 0; k < ySize; k++)
            {
                if (points[i,k].counter > tmpCount)
                {
                    tmpPoint = points[i, k];
                    tmpCount = points[i, k].counter;
                }
            }
        }
    return tmpPoint;
}
```

Figure 25: Code snippet for finding the points with most wins

Next you find the most distant neighbour to the most winning point in Euclidean distance. If the two points are in the same row, a new column will be inserted and if they are in the same column, a new row will be inserted. The new row or column will be between the two points' rows or columns and the points will be interpolated between their direct neighbours.

If a row or column is inserted all local counters will be reset and the global time counter will be reset as well. The eighth step checks if the network size has reached the desired network size, which is initialized in total amount of points in the grid. If it has not the algorithm returns to step two and if it has the algorithm goes out of the growth phase and into the fine tune phase. The fine tune phase is basically running the algorithm from step 2-6 for one more epoch. This allows the algorithm to smooth out the most recently added row or column and relax the grid. This phase can also be used to gradually reduce learning rate and can be adjusted in length according to need. After the fine tuning phase the GG algorithm is finished and you will be left with an irregular quadrilateral grid, which is distributed according to the input image.

After the original algorithm was implemented there were added a few modifications. The first modification was instead of having a binary image as input, a grayscale image was used where the grayscale value would be used to control the distribution. The values were split into four intervals, where a higher value would increase the number of occurrences in the distribution and thus a more dense distribution in the final grid. This modification would be used to be able to better control the density of the final grid, through the input image. This allowed input images such as heightmaps and gradients to be used. Using the

final grid with WFC showed that at times the grid would be flipped upside down. This was fixed using the crossproduct of the first square in the grid. The grid would then be flipped if this was the case. The final modification was the length of each epoch, which was set to a fixed size instead of being dependent on the size of the grid as it is in the original algorithm. It was found that the first couple of iterations needed longer time to run and the last iterations needed less time to run. Combining WFC and GG will produce maps on an irregular grid, which we will define as twisted maps as opposed to maps created on a regular grid, which will be defined as straight maps.

5.6 Blend shapes in a WFC environment

Combining WFC and GG to create twisted maps required a way to skew each single module to fit into the grid slot. This was for this report accomplished by using blend shapes. Blend shapes is a way to pre-bake poses into a mesh from the 3D modeling software, in this case Cinema4D. Unity natively accepts blend shapes and has an editor and a scripting interface for utilizing them. Another useful property of blend shapes is that they can be combined. A mesh can be 50% one pose and 50 % another pose. Each vertices in the mesh is interpolated a given amount dependent on each poses influence. Both Unity and Cinema4D also allows for poses to be extrapolated allowing the shapes to shoot past the initial pose. Blend shapes is mostly used for character facial animations where different expressions needs to be blended between fluently. In this project blend shapes can be utilized in the simplest way by assigning each module's mesh eight different blend shape poses. Each pose consist of the one of the square mesh sides being compressed down to a length of zero. I.e. one corner being moved into the same position as another corner on the same axis. By having eight of these, each corner of the mesh's x and z axis can be controlled by a single blend shape pose each. One corner in its original position would be seen on two different poses both at 0% influence, one for each axis. This pose setup lets the corner positions of the slots in the grid be used as input to the blend shapes. Because the calculation scale of the grid is one, a grid corner twisted 0.2 towards the center of the slot in the x-axis would increase the pose influence for that corners x-axis with 20%. This calculation is a bit simplified as the position of the corners exists in world space. This means that each corner needs to be calculated based on what corner it is. This approach to blend shapes allows every needed shape of the module to be calculated based on the shared corner positions of the slots. Another approach to solving this problem is by using the earlier described marching squares algorithm. In the earlier chapter, the algorithm simply inspired the way the set of modules was designed but the original purpose of the algorithm was to solve issues very similar to this. Marching squares could be used to calculate new positions of the module meshes vertices in a similar way to blend shapes but without having to pre-bake anything in the modeling software which is labor intensive and would not really be scalable to larger projects with larger sets of modules.

The Cave Module set

The first module set that was created with another purpose than purely testing if the WFC algorithm worked was the cave module set (see fig. 27). The cave module set contains five different modules. Floor, Solid, Corner, Inverse Corner and Wall (see fig. 26). The three rotation dependent modules (Corner, Inverse Corner and Wall) are also added to the system with their rotations. This totals to 14 different modules. There are three different grammar codes for the modules. Zero means no mass above floor level, one means half wall above floor level and two means solid wall. Each of the modules also defines if its rotation is inverse and if it is symmetrical to enable the grammar to work on the modules that are rotated. An example could be that a Floors grammar is 0,0,0,0 because it contains no solid mass above ground level. Another example could be that the Corners grammar is 1,0,0,1. This grammar system is inspired by the same marching squares system that inspired what modules that was needed in the set. The concept is to define every slot by its four corners instead of its center point. Defining a corner as "on" or "off" can then be used to create a continuous boundary between two defined spaces. This has a number of benefits that makes it easier to create procedural assets that connects well together as well as having a binary grammar that defines each possibility.

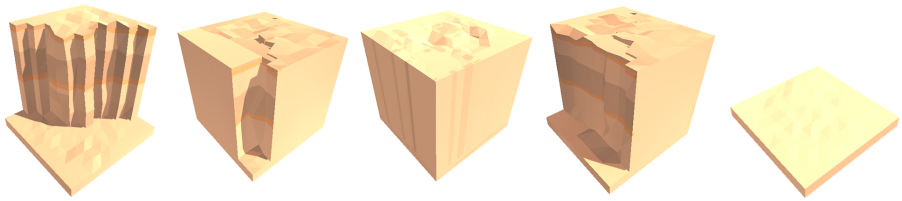


Figure 26: All five modules used for the project. Adding up to 14 modules with rotations.

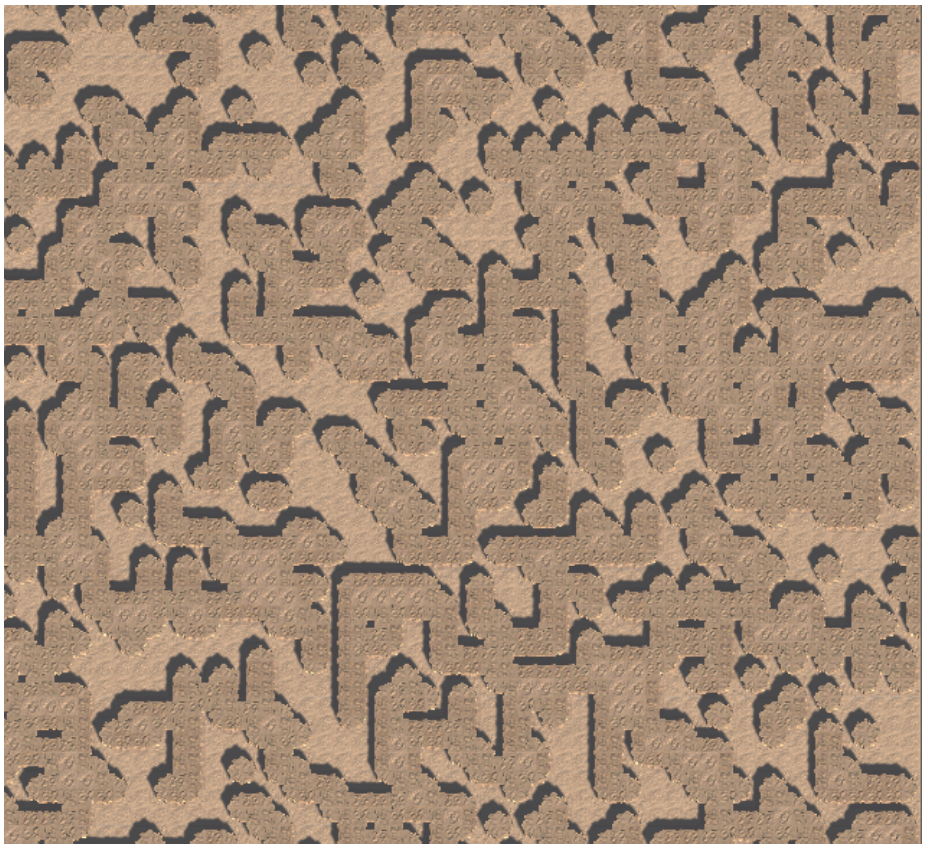


Figure 27: A 50 by 50 grid generated from the cave module set

The hand drawn module set

For the purpose of exploring the systems it was interesting to explore how a two dimensional module set would work with GG and feature distributions. For this purpose a hand drawn two dimensional module set was made. The module set was hand drawn with black ink and processed with Adobe Photoshop. The module set is designed after the marching squares system just like the cave module set, it does however, include the diagonal module which was not included in the cave set. The module set was hand drawn with black ink. Creating the module set in an analogue way created some challenges as the tiles needed to fit perfectly together to create any sense of continuity across modules. To create continuity between modules such as the chimney entering the module in the top and exiting in the bottom in the solid module. The continuity part needs to be centered on the vertical axis so the part can be flipped and still work. If

the module set was complete and every module was unique this would not be needed and the continuity parts could be spread unevenly across the edges. As all the modules are in this set are flipped around the vertical axis the continuity such as the walls does not need to be centered on the edges.

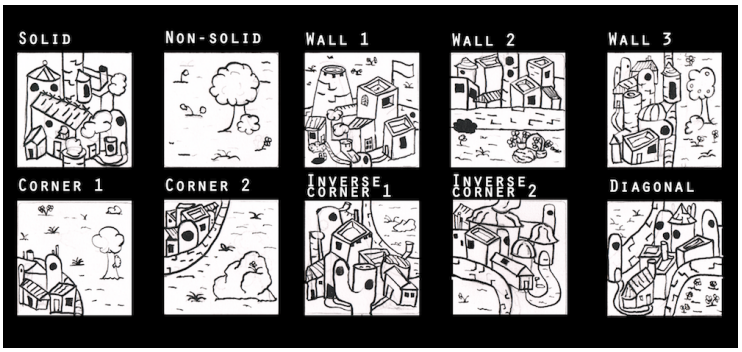


Figure 28: All ten modules for the hand drawn module set

A two dimensional module set requires more unique modules than a 2.5 dimensional module set. When creating a 2.5 dimensional module set, only five different modules are needed as the wall, corner and inverse corner can be rotated and flipped to complete the total 15 modules for a complete micro blob set. When creating a two dimensional module set that uses a standard horizontal perspective line, only the modules that can be flipped around the vertical axis can be flipped without messing up the modules. The modules that are flipped around the vertical axis still works as their perspective is still intact. This can be seen in fig. 29 where three different modules are needed for the wall module and two different corner and inverse corner modules. These considerations are not needed for two dimensional module sets without perspective such as flat patterns or maps.

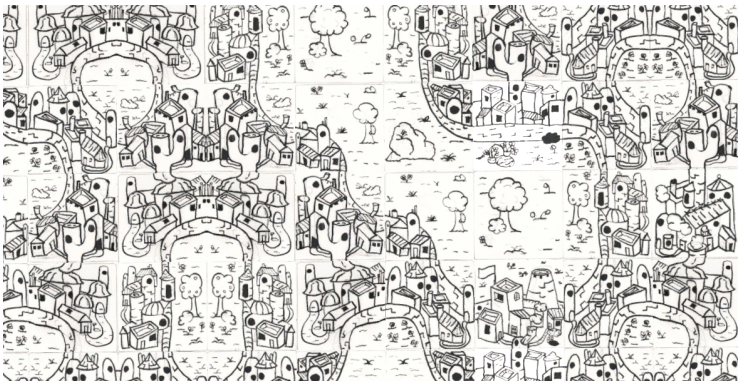


Figure 29: Example of output from the hand drawn module set

The three dimensional moduleset

To further evaluate the system and to gain a better understanding of the visual outcomes, a three dimensional module set was created. In contrast to the cave module set, this set was designed to work in 3D grids. Because of the "natural" setting of the cave set, a concern about how the twisting and skewing of man-made locations would look was raised. This module sets aim was to explore this concern. The module set consists of seven unique modules (see fig. 31).

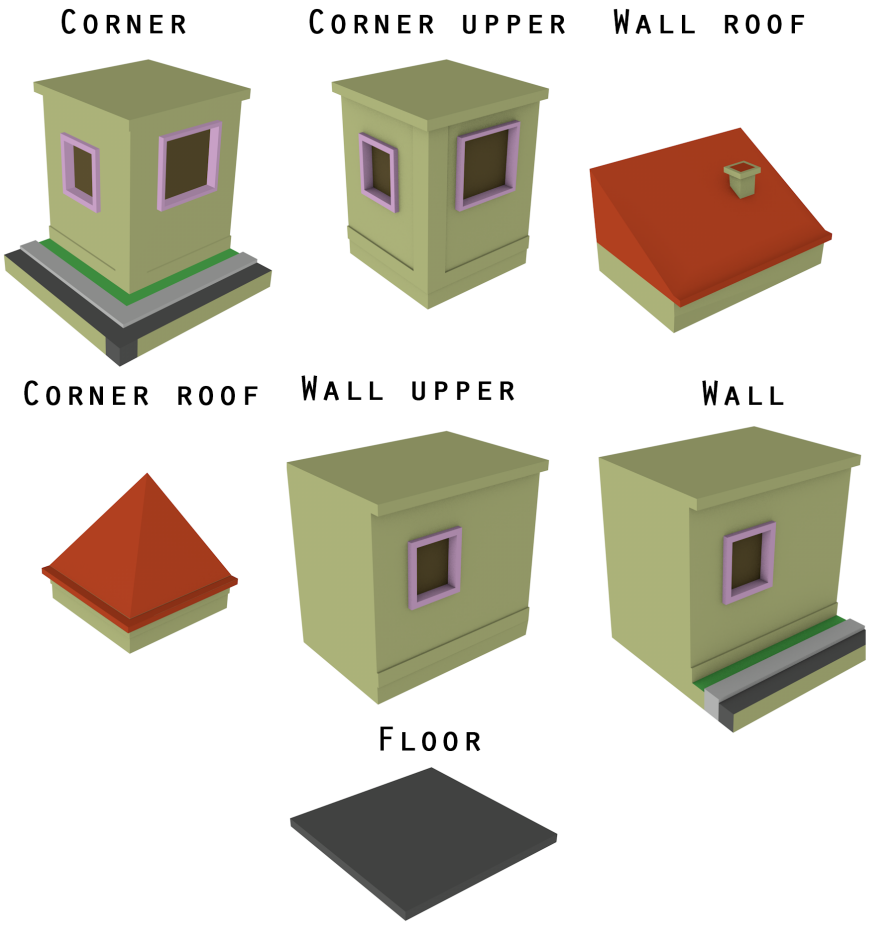


Figure 30: Each unique module in the three dimensional set

The module set is, like the other sets, based on the marching squares system. However, there are some omissions compared to the other sets. The non-solid

module from the other sets is not present in this set as the buildings should be rectangular like real buildings allowing windows in every room. If non-rectangular blocks of buildings was desirable, a solid module could easily be added to the set as it does not need a model. This is because the solid module would never be visible in a three dimensional system as it is always encapsulated. The inverse corner is also removed for the similar reasons, adding in an inverse corner could create L shaped houses which are not desirable.

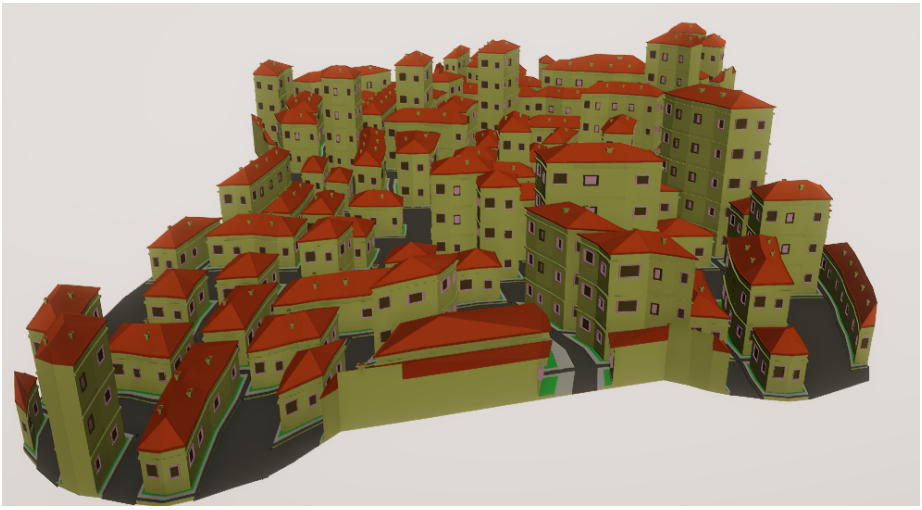


Figure 31: Example output with the 3D module set.

6 Evaluation

The evaluation chapter is split into three parts; two evaluations and a third part with visual inspection of the generated output. The first evaluation focus on the difficulty of the generated output. The second evaluation focuses on the recognizability of the generation output. In the visual inspection section we explore how the combined system could be used and test the capabilities of the system. Evaluation procedural generated content can be difficult, even more so when the product is more of a method or technique than an actual video game or system.

6.1 First Evaluation

The first evaluation of this project will focus on the difference between maps generated using WFC and GG on various shapes and maps generated with only WFC. The first evaluation will test participants' ability to navigate these different maps. The hypothesis of the evaluation is that, twisting the maps with

WFC and GG will increase the time it takes for the participants to navigate the maps. The hypothesis is based on the work by Daniel R. Montello that suggests navigating in parallel and orthogonal paths are easier than navigation in map with twisted paths [48]. Based on this research, we expect the test participants to have a lower completion time in the non-twisted maps and a higher completion time in the twisted maps created with WFC and GG.

First evaluation setup

To evaluate how well a map can be navigated, participants were asked to follow a route through a map and when they reached the end of the route, the route would disappear and they were asked to find their way back to the start. Participants played a first person controller using WASD to move around and the mouse to look around. Participants were timed on their trip out and back and their distance travelled were calculated out and back. For this task six different maps were generated. The maps were generated with GG on six different shapes, which can be seen in fig. 32.

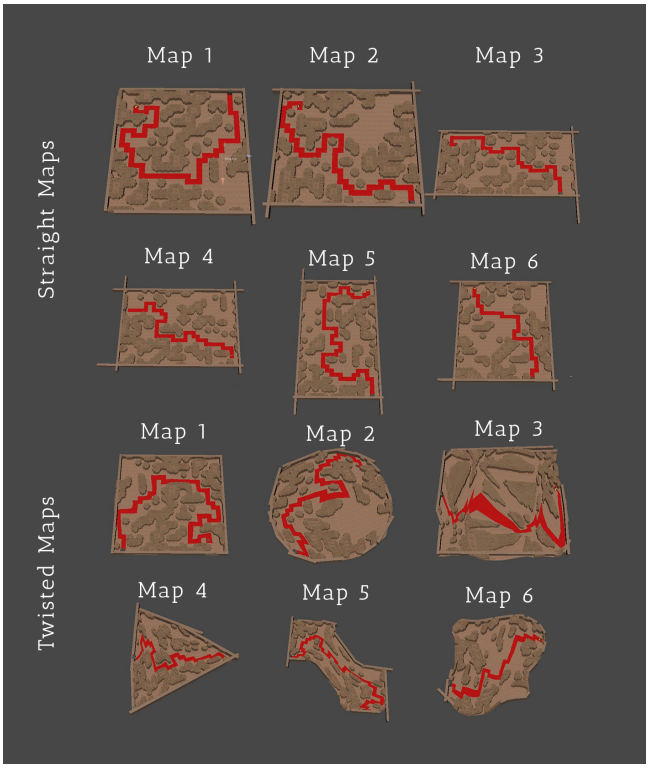


Figure 32: Maps used for the first evaluation

All modules were placed without any weighting and with chiseling implemented to create a route between two corners of the map. These six maps then had the GG features removed to generate additional six maps, but with the same distribution of modules. This created a total of 12 maps for the evaluation. Participants of the evaluation would play a total of six of these maps, where they would play each of the maps but have it randomly be either twisted with GG or without. Every participant played the maps in the same order, to account for learning how to control the character. The end of the labyrinth was marked with a chest filled with gold the player would pick up when arriving at the end. The start was marked with a closed-off entrance door to the labyrinth. The player was also informed that they were a poor treasure hunter looking to prove themselves in the labyrinth. All this was done to give the participants some motivation and some associations with other games or experiences. The evaluation was done online, using Unity's webGL build and results were returned through automated email. The participants would therefore evaluate without supervision from us. The evaluation took approximately 5-10 minutes to complete all six labyrinths. The evaluation was deployed through social media.

First evaluation results

The results from this evaluation are the time for each participant to return from the chest at end of the path (see appendix 1 for raw results). This time will have a baseline time subtracted from it, to be able to evaluate across different maps. This baseline is calculated by following the route exactly both forward and back and taking the mean of the two. This setup will yield two sets of times for each map, one time for all participants who got the twisted map and one time for all the participants who got the straight map. These will together yield two larger sets of data, one set for all six twisted maps and one set for all six straight maps. The evaluation had 44 participants, which produced 141 individual times for twisted maps and 123 individual times for the straight maps. The reason the numbers are not equal is that participants were given either twisted or straight maps at random. To test whether the data is normally distributed the two sets of data was tested for normality using the Shapiro-Wilk test with a significance level of 5%. The null hypothesis H0 and alternative hypothesis Ha are as follows for both sets of data.

- H0: The variable from which the sample was extracted follows a Normal distribution.
- Ha: The variable from which the sample was extracted does not follow a Normal distribution.

The data set for twisted maps returned the following values and QQ-plot:

W:	0.616
p-value:	< 0.0001
alpha:	0.05

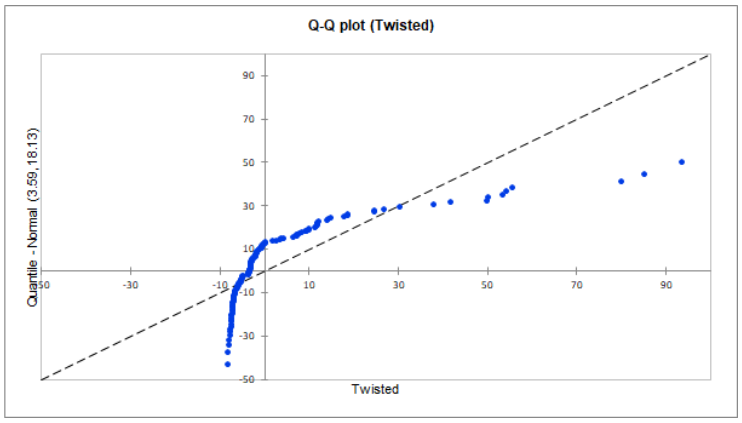


Figure 33: QQ-plot for the twisted maps data set

As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha.
The data set for straight maps returned the following values and QQ-plot:

W:	0.602
p-value:	< 0.0001
alpha:	0.05

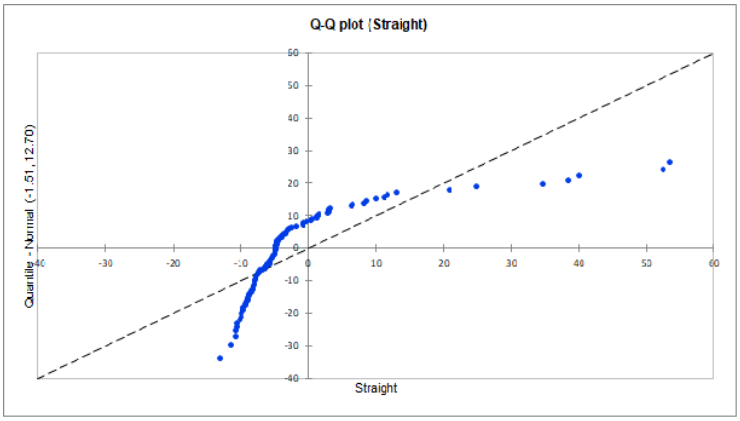


Figure 34: QQ-plot for the straight maps data set

As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha. Neither of the samples thus follows a normal distribution and to test if the distributions are significantly different, a non-parametric test has to be performed. For this a two-sample Kolmogorov–Smirnov test was run. The null hypothesis H0 and alternative hypothesis Ha are as follows:

- H0: The two samples follow the same distribution.
- Ha: The distributions of the two samples are different.

The test returned the following values:

D:	0.292
p-value:	< 0.0001
alpha:	0.05

As the computed p-value is lower than the significance level alpha=0.05, one should reject the null hypothesis H0, and accept the alternative hypothesis Ha. This shows that the two distributions are significantly different from each other.

The two distributions returned the following measures:

Variable	n	Min	Max	Mean	Median	Std. deviation
Twisted	141	-8.180	93.600	3.594	-3.01	18.199
Straight	123	-13.080	61.880	-1.512	-5.02	12.755

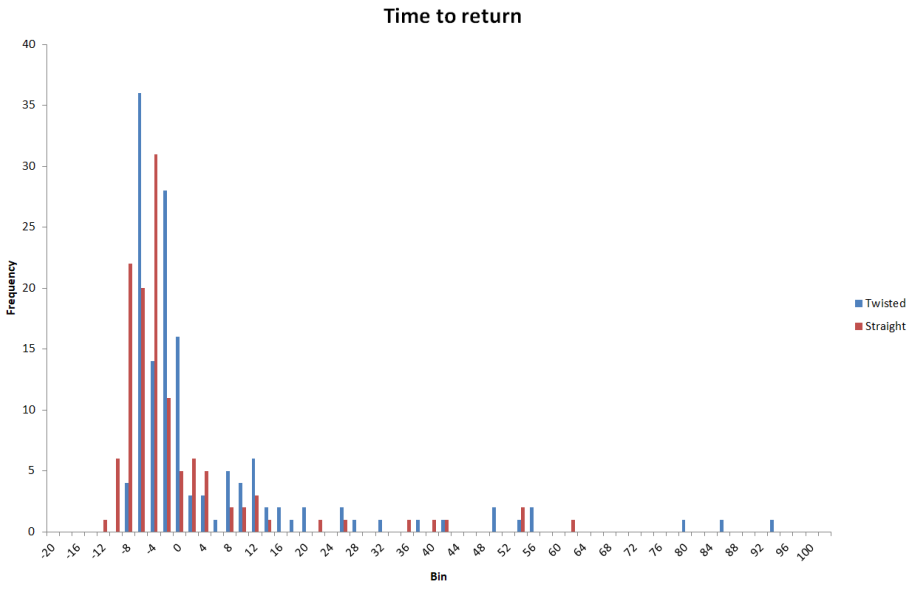


Figure 35: Histogram for time to return with both distributions

Discussion of first evaluation results

The results showed that the two distributions were significantly different from each other and that when playing straight maps, participants were faster

compared to the baseline, than when playing the twisted maps. There is a greater difference between the means than between the medians, which suggests some outliers in the dataset. This can also be seen in the higher standard deviation of the twisted dataset, suggesting a greater tendency to get lost, when playing the twisted maps. This can also be seen in the maximum value of each data set, where the twisted dataset has a maximum value of 93.6 and the straight dataset has a maximum value of 61.89. Disturbing the regularity of the grid thus makes navigation more difficult and increases the tendency to lose spatial orientation. These results are in line with the work by Daniel R. Montello, where participants show greater navigational skills, when in orthogonal surroundings [48]. The results also confirm the hypothesis proposed in this evaluation.

6.2 Second Evaluation

An important aspect of diversity in PCG is the ability for the user to recognize and distinguish the generated content. The second evaluation attempts to explore how the twisted generated maps and the straight WFC maps differentiate in this aspect. This evaluation placed the evaluation participants in a map generated with the three dimensional module set for one minute. After the minute was up, they were shown a picture of six different maps and had to point out which map they were placed in. There were three different maps with a twisted and a straight version each, totaling six maps. Each participant played three maps and never a duplicate of the same map in each version. The participants had one minute to investigate the map before being stopped and having the picture shown. The picture showed all six generated map from a perspective view (see fig. 36).

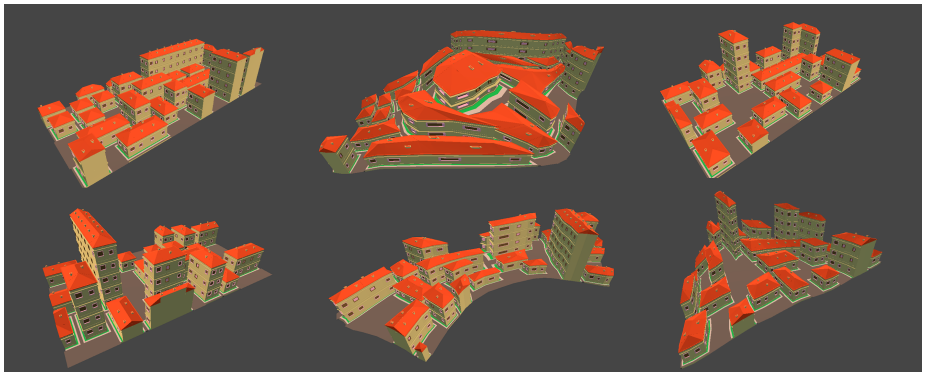


Figure 36: The six maps the participants had to match with the map they played.

This second evaluation had some similarities to the first evaluation, as the participants were asked to walk around a generated map and the players were

trying to gain some sense of the map they were placed in. However, this time around the evaluation was done under the supervision of us, this made it possible to observe the evaluation and to be able to ask the participants questions. For each map the participants were asked about their confidence in their choice, their overview over the map and if they used any specific features to match the pictures to the maps. After the participants had played their three maps, they were asked for general techniques they had used and what map they thought was the hardest to match in hindsight. The evaluation had ten participants. Biggs et. al suggests players use patterns to orientate themselves in procedurally generated maps [47]. However, the use of the patterns had no significant impact on their measured ability to navigate. The use of patterns could be similar to the use of twisted maps, as the twisted maps contain unique versions of the modules that could be noticeable. The hypothesis of the evaluation is: When twisting the maps with WFC and GG participants will have increased ability to match the maps to the pictures.

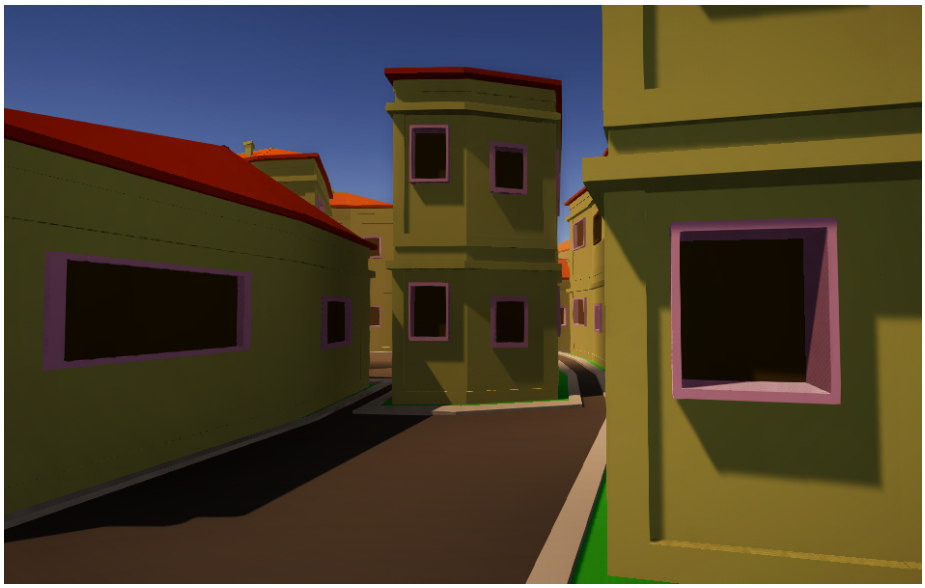


Figure 37: Example of the evaluation participants perspective.

Second evaluation results

The results from the evaluation consists of the participants' guesses for matching the pictures to the maps, their answers to the questions after each map and the final two concluding questions(see appendix 2 for raw results).

X	Correct Guesses
Straight	60%
Twisted	80%

The participants guesses shows that it was slightly easier for them to match the maps in the twisted versions. The self assessment answers following each questions in figure 6.2 shows similar results.

X	Mean confidence	Med confidence	Mean overview	Med overview
Straight	7.66	8	6.4	6
Twisted	7.93	9	7.1	7

The answers to the Question "Did you base your choice on a specific feature of the map?" Shows that in 10 out of the 15 played twisted maps, the participant used the shape of the buildings in some way. Indicated by keywords such as "curved","flat","thick" and "triangle". 19 of the total 30 answers mentioned the height of the buildings indicated with the words "tower" and "height" or mentioning the number of floors. Of the 19 answers 10 were in straight maps and nine in twisted maps. In the question "What techniques did you use to investigate the map?" asked after the participants had played three maps, eight participants mentioned the height of the buildings. Three participants mentioned the shape of the buildings. Three participants mentioned the corners of the maps. In the question "Which map would you rate the most difficult to match to the picture?" six participants said a straight map and four participants a twisted. Additionally one participant mentioned a straight map being the easiest and one participant mentioned a twisted map being the easiest.

Discussion of second evaluation results

The evaluation results showed that the test participants were slightly better at matching the twisted maps than the straight maps. This suggests that twisting the maps actually does improve the players' ability to recognize the procedurally generated maps and is in line with the hypothesis. Twisted maps also increased participants' self-asserted confidence and overview over the map. This seems in line with the work of Biggs et. al. that suggested players would use patterns in procedural generated maps to orientate themselves, even though it did not improve their ability to navigate them [47]. Many of the evaluation participants mentioned the height of the buildings as a specific thing used when trying to orientate themselves. This could be seen as an indication that the height of the buildings was a more powerful tool for orientation than using the shape of the buildings. A majority of the participants playing twisted maps did mention the shape of the buildings. This together with the self assessment results and matching ability could indicate that the reconizability increases when using the diverse maps created by WFC and GG.

6.3 Evaluation through visual inspection

This section will evaluate the project through a visual inspection, showcasing what the different algorithms of the project are capable of achieving. The purpose of this section is to visually present different use cases of the algorithms and discuss advantages and disadvantages of this approach to PCG. The different algorithms, which will be presented is GG, Chiseling and Feature Distribution. The algorithms will be tested on the same shapes.

Visual inspection

Using GG on a contained shape works well and the final grid will take on the shape of the input (see fig. 38).

Figure 38: Growing Grid on various shapes

However when using GG on multiple shapes at once the space in between will often become stretched (see fig. 39). If controlled this might not be an issue, but it will often produce stretched models, when WFC is run on the grid (see fig. 40).

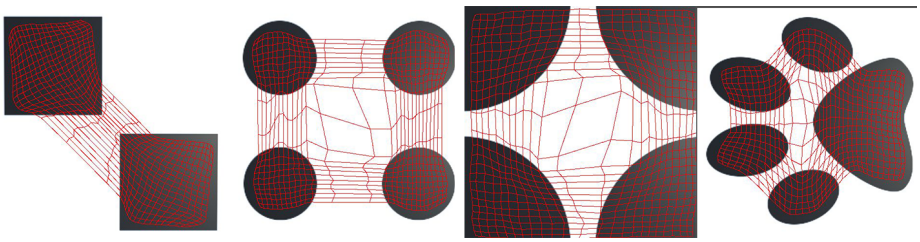


Figure 39: Growing Grid on various shapes

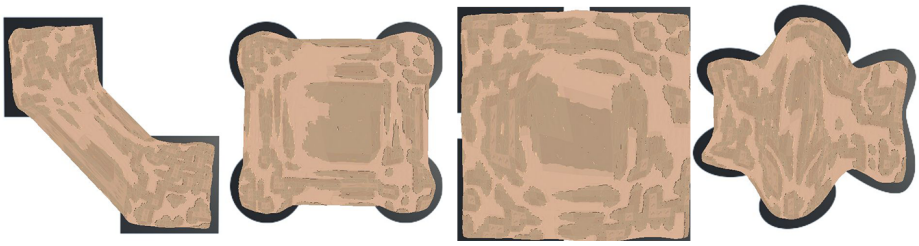


Figure 40: Growing Grid and Wave Function Collapse on various shapes

When running WFC on contained shapes there will be no stretched models in the output, as the sizes of each cell are homogeneous (see fig. 41).

Figure 41: Growing Grid and Wave Function Collapse on various shapes

The feature distribution will be inspected using two different inputs (see fig. 42). The lines input shows how an input will be twisted when used together with GG (see fig. 43). The edge input shows how some inputs can be implemented regardless of the shape of the grid (see fig. 44).

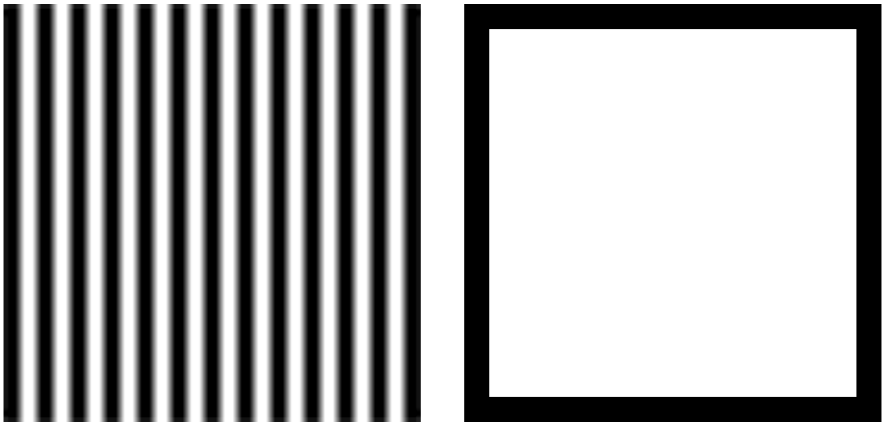


Figure 42: Inputs for feature distribution

Knowing how the input is transformed when applied to a twisted grid is important when working with the two algorithms together. General inputs such as an edge or a line through the entire image will work well, while specific shapes might get skewed.

Figure 43: Growing Grid, Wave Function Collapse and feature distribution on various shapes with lines input

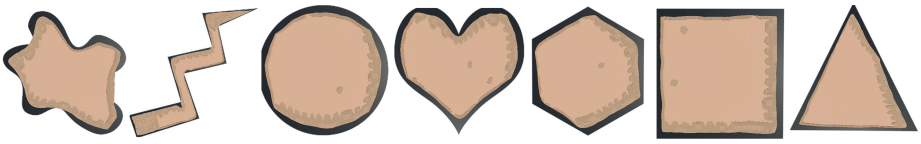


Figure 44: Growing Grid, Wave Function Collapse and feature distribution on various shapes with edge input

Chiseling is applied to all the shapes with the line input for feature distribution to show how the path is generated before everything else (see fig. 45). It is set to go from one corner to the opposite corner in the following examples. The feature distribution will continue to work outside of the path regardless of the shape of the path.

Figure 45: Growing Grid, Wave Function Collapse, feature distribution and chiseling on various shapes

The following example is an attempt at producing an arena, which could be a specific use case of the system. The inputs for GG and feature distribution can be seen in fig. 46.

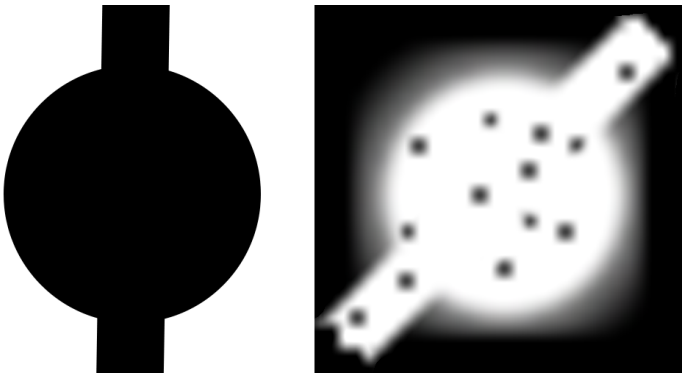


Figure 46: Input for Growing Grid and feature distribution for creating an arena

The arena shows how a distribution input can be used with general shapes, such as an edge and a line through the map to create the map. In addition to this random dots can be placed around the map, where the position of these are not important, but rather to create a certain aesthetic in the map.

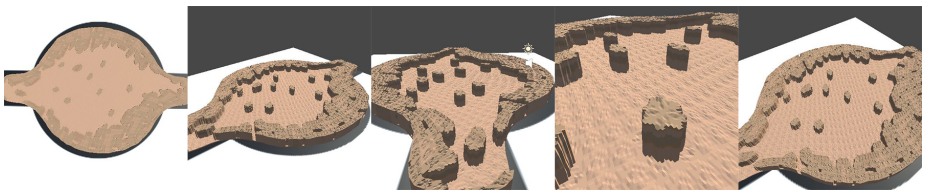


Figure 47: Attempt at producing an arena

7 Discussion

The aim of this project was to utilize multiple state of the art systems for PCG and combining them into a single system. Important factors in game design, such as diversity, recognizability, traversability and being controllable was chosen as the main focus for the algorithm. The final algorithm proved to work well, where evaluation showed that the main focus of the algorithm was achieved. However the results show only how the system compares to a different version of the WFC algorithm and not how it compares to other PCG algorithms. Because of this the steps taken to ensure the requirements to the algorithm is a step in the right direction, but with no comparison to other PCG algorithms, it is hard to predict the usefulness of the system. Using GG and the transformation of the grid as a metric in both tests was interesting, but to further solidify the results, a metric for the transformation would be needed. It is difficult to measure the transformation of the grid, however it would be interesting to match the result to a metric of how much the grid

was transformed. Some of the systems implemented were not evaluated as the output of them is certain, such as chiseling and feature distribution, which both ensures a certain output. Both could be evaluated as a designer tool and if a designer would be able to use them when making a map. Evaluating the project as a designer tool was considered throughout the process. The goal of the algorithm is to be used for a video game by a game designer and thus the usability of the algorithm is important. It was chosen to perform the current evaluation, because it was decided that the functionality of the algorithm was more important to evaluate, than the usability.

8 Conclusion

This report attempted to explore a novel PCG algorithm WFC. By analyzing state of the art applications of PCG and WFC in the video game industry, diversity, difficulty control and traversability was selected as three areas where WFC could be improved, to increase its chances of use in future game projects. To improve on these areas, WFC was combined with the GG algorithm because of its ability to create diverse output. Traversability was solved with the use of chiseling and a solution to control feature distribution was also implemented. To evaluate how the difficulty could be adjusted an evaluation was done with 44 evaluation participants navigating mazes created with a combination of all three algorithms. The results from the first evaluation show that participants navigated the straight maps faster than the twisted maps, which suggests that the difficulty of a possible game could be adjusted by disturbing the regularity of the grid. The second evaluation suggests that by using the proposed system, maps can become more recognizable. Adjusting the difficulty and ensuring recognizable maps are two of the important factors mentioned in the analysis, which both can be achieved by using GG in combination with WFC. The visual inspection shows how diverse output can be created using this algorithm. This indicates that our efforts to create diverse and meaningful content was successful and could be expanded for future use in video games.

9 Bibliography

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

[2] Julian Togelius, Emil Kastbjerg, David Schedl, and Georgios N. Yannakakis. What is procedural content generation?: Mario on the borderline. In *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games*, PCGames '11, pages 3:1–3:6, New York, NY, USA, 2011. ACM.

[3] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content*

Generation in Games: A Textbook and an Overview of Current Research. Springer, 2016.

[4] Julian Togelius, Alex J. Champandard, Pier Luca Lanzi, Michael Mateas, Ana Paiva, Mike Preuss, and Kenneth O. Stanley. Procedural Content Generation: Goals, Challenges and Actionable Steps. In Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 61–75. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2013.

[5] Gillian Smith. An analog history of procedural content generation. In *FDG*, 2015.

[6] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of theoretical biology*, 18:280–99, 04 1968.

[7] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[8] Gillian Smith. Understanding procedural content generation: A design-centric analysis of the role of pcg in games. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '14, pages 917–926, New York, NY, USA, 2014. ACM.

[9] Hello Games. No man's sky. https://www.nomanssky.com, 2016.

[10] The Molasses Flood. The flame in the flood. http://www.themolassesflood.com/the-flame-in-the-flood, 2016.

[11] Gearbox software. Borderlands. https://borderlands.com/en-US/, 2009.

[12] Damian Isla. Forging the river in the flame in the flood. https://www.youtube.com/watch?v=6N56YpHCHBM, 2016. [Online; accessed 22-may-2019].

[13] Noor Shaker, Georgios Yannakakis, and Julian Togelius. Towards automatic personalized content generation for platform games. In *Proceedings of the Sixth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'10, pages 63–68. AAAI Press, 2010.

[14] Noor Shaker, Georgios N. Yannakakis, and Julian Togelius. Towards player-driven procedural content generation. In *Proceedings of the 9th Conference on Computing Frontiers*, CF '12, pages 237–240, New York, NY, USA, 2012. ACM.

[15] Michael C. Toy and Kenneth C. R. C. Arnold. A guide to the dungeons of doom. `https://docs.freebsd.org/44doc/usd/30.rogue/paper.pdf`. [Online; accessed 22-april-2019].

[16] Sebastien Benard. Building the level design of a procedurally generated metroidvania: a hybrid approach. https://www.gamasutra.com/blogs/SebastienBENARD/20170329/294642/Building_the_Level_Design_of_a_procedurally_generated_Metroidvania_a_hybrid_approach.php, 2017. [Online; accessed 22-May-2019].

[17] Motion Twin. Dead cells. https://dead-cells.com, 2017.

[18] Dan Stapleton. No man's sky review. https://www.youtube.com/watch?v=bcjA74emEJI, 2016. [Online; accessed 25-may-2019].

[19] Maxim Gumin. Wavefunctioncollapse. https://github.com/mxgmn/WaveFunctionCollapse/blob/master/README.md, 2016. [Online; accessed 22-March-2018].

[20] Maxim Gumin. basic3dwfc. https://bitbucket.org/mxgmn/basic3dwfc/overview, 2016. [Online; accessed 22-March-2019].

[21] Marian Kleineberg. marian42wavefunctioncollapse. https://github.com/marian42/wavefunctioncollapse, 2018. [Online; accessed 22-March-2019].

[22] Joseph Parker and Ryan Jones. procskater. https://arcadia-clojure.itch.io/proc-skater-2016, 2016. [Online; accessed 22-March-2018].

[23] Plausible Concept. Bad north. https://www.badnorth.com/, 2019. [Online; accessed 22-March-2018].

[24] Andy Wallace. Maureens' chaotic dungeon. http://andymakesgames.tumblr.com/post/182363131350/global-game-jam-2019-maureens-chaotic-dungeon, 2019. [Online; accessed 22-March-2018].

[25] Bernd Fritzke. Growing grid — a self-organizing network with constant neighborhood range and adaptation strength. *Neural Processing Letters*, 2(5):9–13, Sep 1995.

[26] Bernd Fritzke. A growing neural gas network learns topologies. In *Advances in neural information processing systems*, pages 625–632, 1995.

[27] Bernd Fritzke. Growing grid. https://demogng.de/JavaPaper/node24.html#SECTION00730000000000000000, 1997. [Online; accessed 22-May-2019].

[28] Oskar Stålberg. Marching squares on irregular quadrilateral grids. https://twitter.com/OskSta/status/1120036868966502401, 2019. [Online; accessed 25-may-2019].

[29] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.

[30] Chien.-Chang Ho, Fu-Che Wu, Bing-Yu Chen, Yung-Yu Chuang, and Ming Ouhyoung. Cubical marching squares: Adaptive feature preserving surface extraction from volume data. *Computer Graphics Forum*, 24(3):537–545, 2005.

[31] CatlikeCoding. marching squares, a unity c tutorial. `https://catlikecoding.com/unity/tutorials/marching-squares/`, 2019. [Online; accessed 9-May-2019].

[32] Boris the Brave. Tileset roundup. `https://www.boristhebrave.com/2013/07/14/tileset-roundup/`, 2013. [Online; accessed 22-april-2019].

[33] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna I. Esparcia-Alcazar, Chi-Keong Goh, Juan J. Merelo, Ferrante Neri, Mike Preuß, Julian Togelius, and Georgios N. Yannakakis, editors, *Applications of Evolutionary Computation*, pages 141–150, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[34] Jessica Baron. Procedural dungeon generation analysis and adaptation. pages 168–171, 04 2017.

[35] Jason Bevins. Perlin worms. http://libnoise.sourceforge.net/examples/worms/, 2005. [Online; accessed 25-may-2019].

[36] Path of Exiles. Path of exile - random level generation presentation. https://www.youtube.com/watch?v=GcM9Ynfzll0, 2011. [Online; accessed 25-may-2019].

[37] Boris the Brave. Chiseled random paths. `https://github.com/BorisTheBrave/chiseled-random-paths`, 2018. [Online; accessed 22-april-2019].

[38] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, pages 4:1–4:7, New York, NY, USA, 2010. ACM.

[39] N. Shaker, M. Nicolau, G. N. Yannakakis, J. Togelius, and M. O'Neill. Evolving levels for super mario bros using grammatical evolution. In *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 304–311, Sep. 2012.

[40] Britton Horn, Steve Dahlskog, Noor Shaker, Gillian Smith, and Julian Togelius. A comparative evaluation of procedural level generators in the mario ai framework. 2014.

[41] Cameron Browne and Frederic Maire. Evolutionary game design. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2:1 – 16, 04 2010.

[42] Isaac Karth and Adam M. Smith. Wavefunctioncollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, pages 68:1–68:10, New York, NY, USA, 2017. ACM.

[43] Hugo Scurti and Clark Verbrugge. Generating paths with WFC. *CoRR*, abs/1808.04317, 2018.

[44] Andreas Wulff-Jensen, Niclas Nerup Rant, Tobias Nordvig Møller, and Jonas Aksel Billeskov. Deep convolutional generative adversarial network for procedural 3d landscape generation based on dem. In Anthony L. Brooks, Eva Brooks, and Nikolas Vidakis, editors, *Interactivity, Game Creation, Design, Learning, and Innovation*, pages 85–94, Cham, 2018. Springer International Publishing.

[45] Ian Horswill and Leif Foged. Fast procedural level population with playability constraints. In *Proceedings of the Eighth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AIIDE'12, pages 20–25. AAAI Press, 2012.

[46] D. H. Adrian and S. C. Ana Luisa. An approach to level design using procedural content generation and difficulty curves. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8, Aug 2013.

[47] Michael Biggs, Ute Fischer, and Michael Nitsche. Supporting wayfinding through patterns within procedurally generated virtual environments. *Proceedings of Sandbox 2008: An ACM SIGGRAPH Videogame Symposium, Sandbox'08*, 09 0002.

[48] Daniel R. Montello. Spatial orientation and the angularity of urban routes: A field study. *Environment and Behavior*, 23(1):47–69, 1991.

10 Appendix

1. Raw test results from evaluation 1

	Map played	Trip out	Trip back	Distance out	Distance back	Map played	Trip out	Trip back	Distance out	Distance back	Map played	Trip out	Trip back	Distance out	Distance back	Map played	Trip out	Trip back	Distance out	Distance back	Map played	Trip out	Trip back	Distance out	Distance back
Participant 1	1	37.10	17.66	113.28	73.39	3	14.66	13.18	58.33	53.63	5	18.70	22.76	70.28	87.93	8	15.74	15.24	64.89	62.33	10	19.26	21.82	80.65	82.50	11	9.66	9.94	40.24	40.95
Participant 2	1	21.86	17.62	91.74	73.71	3	17.00	17.14	71.23	70.78	6	15.90	24.08	67.02	98.40	7	9.92	18.68	40.69	64.85	10	20.74	32.82	87.50	130.59	11	9.46	9.78	39.28	40.22
Participant 3	2	20.40	36.96	83.57	139.28	3	14.56	44.98	58.20	153.36	6	15.06	59.34	62.72	227.23	8	13.10	23.64	54.46	83.03	10	18.46	22.44	77.42	92.65	12	12.20	57.54	51.08	215.68
Participant 4	1	20.28	52.04	80.07	171.89	3	17.76	15.36	73.91	55.81	5	19.78	51.32	65.21	144.17	7	13.40	91.46	36.14	240.75	10	22.06	20.98	80.78	88.48	12	12.80	12.28	51.48	51.67
Participant 5	2	28.76	20.02	82.22	82.74	3	14.24	13.96	55.60	56.56	5	16.20	62.70	64.71	244.51	8	13.26	15.02	53.38	55.46	9	15.90	14.52	63.22	56.59	12	23.84	12.72	76.67	53.20
Participant 6	1	26.20	17.62	96.98	73.50	3	12.82	13.30	53.40	54.79	5	14.46	29.20	61.28	117.37	8	14.74	14.58	60.19	60.78	10	19.96	21.68	83.25	88.61	11	10.28	10.08	42.85	41.78
Participant 7	1	18.44	18.64	76.30	77.18	3	13.82	13.48	54.53	55.28	6	15.58	15.84	64.75	66.15	8	12.98	14.42	53.23	58.28	9	15.14	14.78	62.50	60.35	12	12.86	14.12	53.66	56.92
Participant 8	2	21.66	19.56	89.01	80.69	3	13.48	12.90	55.31	53.45	6	16.84	16.08	69.15	67.25	8	12.72	13.08	53.09	54.36	10	17.74	18.96	73.73	75.74	11	15.92	9.60	54.78	40.24
Participant 9	1	17.58	17.92	72.27	74.50	3	12.92	12.16	50.87	49.30	6	15.34	16.06	62.77	66.37	8	12.52	16.62	52.18	66.87	9	15.46	12.94	61.64	52.85	11	9.44	10.10	38.86	40.48
Participant 10	1	17.04	17.98	70.42	74.45	3	11.94	12.16	48.81	49.37	5	14.50	32.84	60.55	126.64	7	8.70	10.84	34.33	43.74	9	12.64	12.86	52.23	52.00	11	9.70	11.38	39.96	46.15
Participant 11	1	24.62	18.36	100.96	75.96	3	17.50	13.36	72.50	55.32	6	19.88	16.12	76.80	66.90	7	11.46	7.92	45.08	31.19	9	19.48	12.80	77.59	51.84	12	14.66	12.96	61.13	53.85
Participant 12	2	32.64	49.34	122.40	185.42	3	12.04	13.18	49.53	54.79	6	27.00	74.22	97.01	149.38	8	15.46	12.68	54.30	53.32	9	20.40	19.84	71.06	68.91	11	10.10	12.22	41.01	49.12
Participant 13	1	20.96	18.72	84.63	77.83	3	13.08	19.24	54.34	62.58	5	17.36	32.50	68.69	128.73	7	13.82	20.56	46.63	63.96	9	14.98	14.34	61.04	52.86	12	13.08	14.40	53.53	59.70
Participant 14	2	20.12	19.88	83.53	84.04	3	13.14	13.14	55.10	53.17	6	15.42	16.08	63.80	67.43	8	13.86	13.88	57.66	58.39	9	14.64	13.26	60.79	51.97	11	9.82	11.14	40.06	42.63
Participant 15	2	26.12	25.20	109.13	102.50	3	27.98	13.16	78.02	54.52	5	15.92	39.58	67.64	154.64	8	14.68	12.82	52.97	53.53	10	18.84	30.96	79.53	124.44	11	10.02	17.18	41.64	64.95
Participant 16	2	21.62	19.38	89.81	80.95	3	12.52	12.32	50.71	51.29	5	15.18	15.38	62.51	63.04	8	13.46	18.86	55.90	73.91	10	17.22	19.02	72.25	78.86	11	10.84	9.46	42.01	39.36
Participant 17	1	44.16	23.22	92.30	81.90	3	22.92	13.58	59.61	52.48	5	21.98	24.34	66.13	89.83	8	25.36	12.62	52.16	52.46	9	20.24	15.72	65.60	63.07	12	16.44	15.08	55.65	60.73
Participant 18	1	23.02	19.30	94.59	79.91	3	18.52	18.48	76.25	76.00	6	21.02	16.40	76.09	68.17	8	20.50	13.64	72.85	55.66	9	18.66	13.12	70.69	53.91	11	13.08	27.84	52.07	105.67
Participant 19	1	24.24	39.50	99.37	125.02	3	19.02	18.16	73.49	75.56	5	18.70	18.72	77.29	75.40	8	14.48	15.14	60.33	63.12	9	16.88	12.62	62.23	52.01	11	9.74	9.66	40.86	40.05
Participant 20	1	21.90	32.86	86.47	124.27	3	14.70	12.24	59.22	50.73	6	23.20	16.86	78.54	68.79	7	11.52	7.96	36.88	30.48	9	15.90	12.60	62.09	51.64	12	19.48	13.28	52.79	55.70
Participant 21	2	27.06	30.02	96.21	113.86	3	23.30	13.54	58.09	55.69	5	24.52	18.00	71.88	68.86	8	28.00	14.64	65.24	60.95	10	20.72	21.66	84.20	84.29	11	9.94	9.88	41.29	41.04
Participant 22	1	47.18	37.20	100.02	112.38	3	52.72	17.66	112.35	63.91	6	16.04	73.34	63.94	208.64	7	14.42	21.60	37.97	53.76	9	17.36	14.62	63.59	55.66	11	9.94	98.14	41.00	284.13
Participant 23	1	24.02	22.24	83.32	81.88	3	18.08	24.02	57.83	65.43	5	26.90	28.14	82.76	96.02	7	13.76	11.18	37.40	36.90	9	18.52	19.80	64.05	69.62	12	15.86	20.60	60.27	69.18
Participant 24	1	44.30	80.68	104.35	190.54	3	40.82	73.70	74.15	153.74	5	27.04	21.16	64.61	68.89	7	10.80	105.18	32.67	218.32	9	21.78	39.00	62.88	88.22	12	15.10	16.70	52.87	58.28
Participant 25	1	46.62	42.92	114.79	109.32	4	26.16	59.04	88.97	153.33	6	15.48	16.00	64.28	63.02	7	9.02	23.52	34.54	68.22	10	18.80	24.02	79.08	91.83	12	12.54	18.66	52.53	58.30
Participant 26	1	23.88	36.46	99.81	143.19	4	18.76	13.74	79.01	57.36	5	17.00	32.72	72.64	122.48	8	16.44	13.12	69.29	53.67	10	22.60	19.82	94.89	82.65	12	14.42	12.50	60.57	52.18
Participant 27	1	20.38	19.10	83.49	78.33	4	14.72	14.72	61.16	61.18	5	17.86	23.46	75.05	95.31	7	9.52	9.98	33.90	38.98	9	15.06	12.88	61.93	52.80	11	9.94	9.78	41.18	40.47
Participant 28	1	19.40	18.44	78.75	75.97	4	20.98	15.78	80.09	62.83	5	17.70	30.96	70.27	120.62	8	16.02	17.64	63.94	71.89	9	15.42	15.38	61.99	59.41	11	9.82	9.90	40.51	41.45
Participant 29	1	22.20	18.44	91.08	76.02	4	17.28	36.06	70.56	142.73	6	15.44	15.36	64.69	64.22	7	8.64	8.54	34.80	34.51	9	15.00	13.46	61.83	54.75	12	13.26	18.10	56.09	74.32
Participant 30	2	25.14	28.82	105.35	108.81	4	19.26	16.38	80.21	68.66	6	17.44	32.06	72.49	128.23	7	9.98	9.74	40.70	37.00	9	15.58	15.52	64.11	62.75	12	13.64	12.48	56.84	52.07
Participant 31	1	27.70	18.16	97.89	75.20	4	21.08	18.52	83.02	75.88	5	20.66	30.24	80.76	120.98	8	17.70	17.04	72.18	65.78	10	24.00	24.48	98.13	96.07	11	12.92	10.04	50.51	41.83
Participant 32	1	17.44	79.44	72.31	288.89	4	19.66	14.28	67.05	59.53	5	23.22	15.92	72.57	63.78	8	17.70	13.00	56.49	53.57	10	247.08	20.86	179.96	86.54	11	17.18	10.20	48.97	42.32
Participant 33	1	17.54	17.86	71.42	73.23	4	18.02	14.46	62.05	59.75	6	17.12	22.16	62.22	87.65	7	9.12	8.24	35.79	32.80	10	158.34	19.26	160.81	77.80	11	12.00	10.14	40.97	41.47
Participant 34	1	18.94	22.44	78.65	92.30	4	16.22	30.68	66.05	125.33	6	16.76	16.86	69.00	69.50	7	8.40	11.44	33.19	42.74	10	19.68	20.70	82.73	86.29	12	13.24	12.62	55.15	53.33
Participant 35	2	30.62	90.28	99.05	324.26	4	23.08	19.56	71.96	67.30	6	26.96	27.34	84.92	92.86	8	24.38	16.06	73.91	65.39	10	26.90	20.56	110.04	80.35	12	13.26	14.60	55.80	58.49
Participant 36	1	22.86	19.02	72.04	77.75	4	17.38	14.64	68.67	59.60	6	16.42	15.60	64.95	64.70	7	7.88	7.90	31.21	32.03	10	17.76	16.72	74.24	66.27	12	12.74	12.86	52.29	54.11
Participant 37	1	19.78	18.60	81.78	77.10	4	15.66	15.16	64.81	62.48	5	15.28	21.08	63.97	85.65	8	13.56	15.68	54.75	64.08	9	15.20	70.44	62.55	270.86	11	10.48	11.42	43.68	45.88
Participant 38	2	25.44	20.58	92.32	85.35	4	20.18	17.16	79.23	66.94	5	16.98	70.72	65.23	218.10	8	15.14	14.30	58.75	58.77	10	19.42	22.64	81.76	88.85	12	13.16	14.74	53.39	57.89
Participant 39	2	39.78	41.38	99.05	134.75	4	49.40	23.64	68.23	87.63	5	26.42	45.58	81.63	139.70	8	18.60	12.68	58.46	53.26	9	21.66	34.40	68.47	94.05	12	17.32	16.66	53.21	56.53
Participant 40	2	27.52	38.50	98.23	152.50	4	21.42	14.74	74.86	61.16	5	16.08	15.68	65.17	65.94	8	13.12	12.86	53.53	54.03	10	18.86	32.82	77.53	131.53	12	13.28	12.86	55.03	53.86
Participant 41	1	25.52	31.66	106.05	115.40	4	21.14	19.92	83.71	78.49	6	24.26	15.72	71.42	65.98	8	16.24	14.26	59.73	59.61	10	21.50	18.40	90.48	77.29	11	10.08	10.04	42.33	40.95
Participant 42	1	42.18	24.00	116.86	94.44	4	25.52	18.12	95.45	73.15	6	19.20	20.48	77.32	84.06	8	18.06	15.00	74.23	60.55	9	17.64	15.22	71.23	59.98	12	16.78	13.80	68.92	57.50
Participant 43	1	24.82	22.72	76.48	90.90	4	27.06	16.60	66.75	66.61	5	27.52	28.14	66.77	97.82	8	24.42	45.48	57.10	145.62	10	19.34	20.90	80.91	84.17	11	9.64	12.72	40.39	44.68
Participant 44	1	23.68	19.84	91.16	81.11	4	16.48	16.16	60.09	66.52	5	19.76	58.88	80.92	243.02	7	9.00	9.44	34.97	38.26	9	17.82	15.12	67.58	59.28	12	18.50	25.78	68.02	102.72

2. Raw test results from evaluation two translated form danish

Test Participant	Map played	Which map on the picture did you play?	How confident are you in your choice? 1-10	Rate your overview of the map? 1-10	Did you base your choice on a specific feature of the map?	What techniques did you use to investigate the map?	Which level would you rate the most difficult to match to the picture?
participant 1	1	1	9	7	The clocktower, the wide roads and their placement	the shape of the map, the height of the buildings	3, easiest being number 5
	3	3	3	2	the tick roads		
	5	5	10	8	The corner tower, the big spaces, the size of the buildings		
participant 2	2	2	8	6	the two wide houses	Height of the buildings, size of the buildings	the first map I played
	4	5	7	7	the skewed buildings and the high house in the middle		
	6	6	9	9	the big thick house in the middle		
participant 3	1	3	7	3	tried walking a bit around and hold the houses	height of building, triangle house	map 1
	3	1	10	10	big block and high houses, low houses in the other corner		
	5	6	10	10	the gables was triangles		
participant 4	2	3	4	8	One of the houses was higher than the others, but it was the same way on the other pictures	First with high houses and their relationship between eachother. Afterwards when the building got skewed I looked building fronts and shapes.	map 4
	4	5	6	10	5 floor house. triangle tower and sharp corner		
	6	6	7	6	High building with flat rear, no open spaces and only narrow pathways		
participant 5	1	1	10	8	The corners, height of buildings in the corners. the big open space	looked in the corner. overview over the borders. high buildings and their placement. stayed away from dense areas.	map number 2 I played. the first I played was the easiest
	3	6	9	6	The corners and open spaces, high long buildings		
	5	5	9	8	it seemed divided, tried to look at the sides.		
participant 6	2	5	10	5	The little 1x1 house in the corner	Searched for the corners in the map borders. high buildings	Number 2 map (4)
	4	4	8	3	The walls wasnt equal height or straight. three high building + 1 halvheight		
	6	6	10	6	2 high buildings to the right and one long at the side and the high building in the middle		
participant 7	1	1	9	6	Høje bygninger i den ene ende og lave bygninger i den anden. Højt tårn i midten splitter mappet. Mest de høje hus.	Looked up to see the height of the buildings. localized high buildings and the middle. walk the same path to create a mental image.	number 2 (map 3)
	3	3	5	5	long and alittle high houses in the top. the houses was standing very close to eachother making it hard to get a overview. confusing and very alike.		
	5	5	10	9	Curved map. only one way at the the side of the high building down at the right. big space down to the left. different heights gave a good overview. bigger diversity and curves gave overview.		
participant 8	2	3	6	5	one 4 floor building and a open space(knew what map it was but could not see it on the picture, would have guessed 5 but it was twisted.)	looked for things that stood out. big buildings, strange shapes and how many floors. also skewed windows	number 3 (map 6)
	4	4	9	6	the thin high building		
	6	6	4	3	generic map, low diversity, small streets.		
participant 9	1	1	10	10	what buildings was at the borders, the sequence of buildings at the borders.	tried to find the corners. if you can reconize a corner you can reconize the map	number 2 (map 3)
	3	3	8	10	the big flat long high building and the long building		
	5	5	10	10	the little 1x1 building out in the corner		
participant 10	2	2	7	5	height of buildings.	no, looked at the houses around the highest	number 3(map 6)
	4	4	8	4	height of buildings		
	6	6	2	2	no		