Ghys Victor

Generating in-game level layouts with wave function collapse

Graduation work 2021-2022

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT

This paper aims to find an answer to our research question “How can we use Wave Function Collapse to automatically generate hotel layouts?”. And tries to find out if it is possible to procedurally create playable, interesting and logical layouts.

In this paper two models of wave function collapse are used for generating layouts and compared which each other.

INTRODUCTION

In the course game project we made a game called Boo-Ya!. In this game the player is a ghost in a hotel and scares the people. There was only one level in the game which limits the replayability. This paper aims to find out how to use wave function collapse (WFC) to procedurally generate more hotel level layouts for the game. This would facilitate the job of the level designer requiring him to only make one level. And it would also create an almost infinite number of distinct levels for the player to play in and explore.

Looking at previous work done in wave function collapse, it is definitely possible to generate new levels. But this research wants to find out if it is also possible to make the algorithm meet higher level requirements. Like, are the layouts the algorithm produces also as playable, interesting and logical as the example level that it was given?

RELATED WORK

1. WAVE FUNCTION COLLAPSE

Wave function collapse was introduced by Maxim Gumin [1]. He used the algorithm for texture synthesis, which is creating a bigger texture from a small sample. His approach for this problem was loosely inspired by quantum mechanics.

In short, every pixel is put in super position meaning it has every colour of the example texture at once. Then a random pixel is picked and given one of the colours. Now the collapse part of the algorithm starts, and it goes over all the pixels to check them with constraints that were generated on the basis of the example texture. This ensures that all the invalid possibilities around the newly selected pixel are removed. Then the colour of the next pixel is set, but now the pixels are not chosen randomly anymore but with a least entropy heuristic. Meaning the pixels with the least amount of possibilities will get a fixed value first.

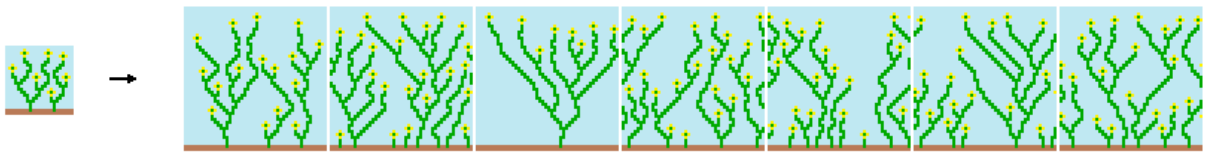


Figure 1 : Example of Wave Function Collapse by M. Gumin [1]

2. MODEL SYNTHESIS

Gumin has based his algorithm on the Model synthesis of P. Merrell [2]. Merrells idea was to instead of doing synthesis on a 2D texture, do it on a 3D model. In model synthesis the world is split in a 3D grid of cells that function as slots for his predefined model pieces. The input model is then created out of these model pieces or building blocks.

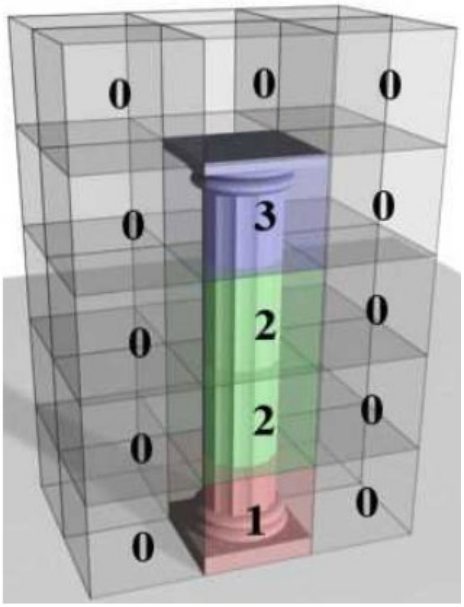


Figure 2 : Model Synthesis by P. Merrell [2]

Gumin has build upon model synthesis by generalising the adjacency constraint to work with an NxN pattern of tiles instead of individual tiles. By varying N, the output can be made to look more like the input or less. He also introduced the lowest entropy heuristic that removes the directional bias in generated results.

Merrell also produced a method of incrementally modifying the model in parts to reduce the failure rate.

3. BUILDING LAYOUTS

Another paper by Merrell et al. [3] researched the generation of residential building layouts. In this paper the algorithm was given a set of high-level requirements. This were things such as the number of bedrooms, number of bathrooms and approximate footage. From these requirements layouts of a residential building were generated with a Bayesian network trained on real-world data. After the generation of a layout a Metropolis algorithm was used to optimise it. For this it made use of moves like sliding and snapping walls and swapping rooms to optimise the cost function. The cost function combines requirements like accessibility, correct dimensions and near-convex rooms to quantify the desirability of a layout.

Figure 3 : Building Layouts by Merrell et al. [3]

4. OTHER USES OF WAVE FUNCTION COLLAPSE

As described by Karth et al. [4], WFC has because of its abstract chunks of content many exiting applications. So was Joseph Parker one of the first to use WFC in a game engine Unity 3D [5]. He later was also one of the first to use WFC for level generation in his game Proc Skater [6]. Another developer that has done significant work with WFC is Oskar Stålberg. He was among the first to start generalising WFC by extending it with other tile shapes, 3D meshes, performance optimizations and adding backtracking. He also made an interactive demo to show how WFC works under the hood [7]. His works include the games Townscaper and Bad North which use WFC to generate buildings and islands accordingly.

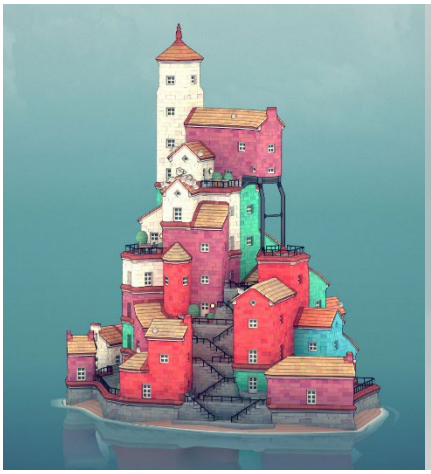


Figure 4 : Townscaper

Figure 5 : Bad North

WFC attracted some attention of several indie and hobby game developers via social media [8]. WFC also incrementally solves the problem very human-like. Gumin says that when he sees humans draw, they often use the minimal entropy heuristic themselves. Which explains why the algorithm is so enjoyable to watch.

In addition to level generation, WFC has also been applied to other kinds of content. So came M. O'Leary with a pretty unusual idea. He created a poetry generator using WFC [9]. It mashes up sentence fragments from a given corpus to produce poems with fixed metric forms. O'Leary treats syllables as the basic unit, so each tile is a sequence of syllables. He then enters these in a 1-dimentional WFC sequence, with some extra long-distance constraints such as rhyme and meter.

CASE STUDY

1. INTRODUCTION

First, I will explain the wave function collapse algorithm. Next, I will show how it is more specifically used to generate hotel layouts.

2. THE ALGORITHM

The wave function collapse algorithm exists of five steps:

- Get the patterns form sample
- Build propagator
- Observe
- Propagate
- Output the observations

The explanation of the algorithm will be using one of Gumin's sample image files as a running example (Fig. 6). It will go over and explain two models of WFC, the simple tiled model and the overlapping model. These are two slightly different ways to use the WFC algorithm. First the overlapping model will be explained in depth. Afterwards the differences with the simple tiled model will be discussed.

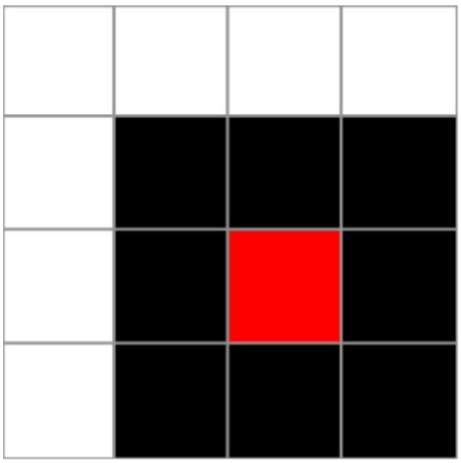


Figure 6: Sample image [4]

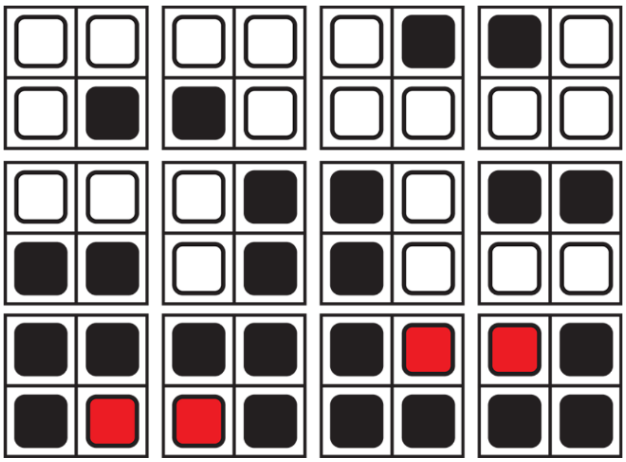


Figure 7: Derived patterns [4]

2.1 GET PATTERNS FROM SAMPLE

As first step, the overlapping model starts with deriving patterns from the sample image. A pattern here is a unique combination of input tiles. It can be seen as a sub-image of the original. Patterns can also be rotated and reflected to get more patterns and increase the chance of finding fitting patterns.

In the overlapping model a sample size of NxN is used. Fig. 7 shows all the twelve unique patterns derived from the sample with a sample size of 2x2. Every derived pattern maps to a section in the input, or a rotated or reflected version of it.

2.2 BUILD PROPAGATOR

As second step, the sampled patterns are taken and checked which patterns can be next to another. This relationship is stored in a data structure called the propagator. It holds the constraint relationship in every direction for every tile. Calculating this upfront improves performance significantly.

For the overlapping model, the propagator contains the pre-calculated answers to whether the union between two patterns match when placed near the other with a particular x, y offset.

With 2x2 patterns there are five ways in which two patterns can overlap, see Fig. 8. In Fig. 9, an example of what the propagator looks like for one pattern can be seen. With in the middle the pattern itself and next to it the patterns that are allowed next to it in each direction.

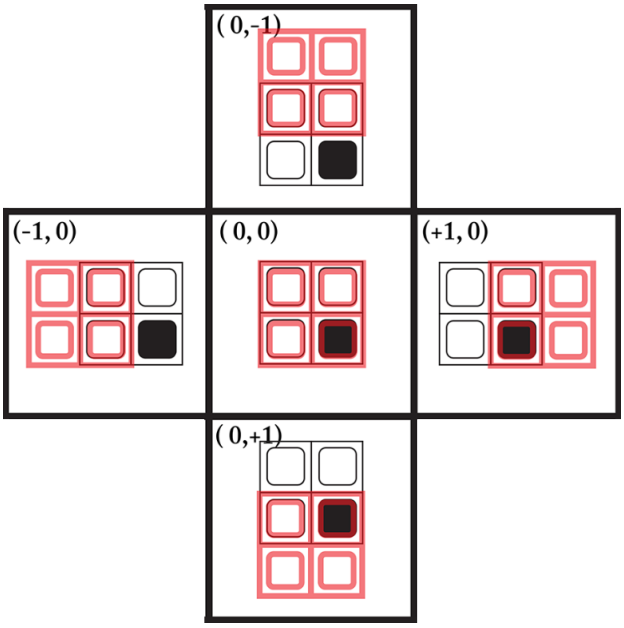


Figure 8: The 5 ways of overlapping two 2x2 patterns together [4]

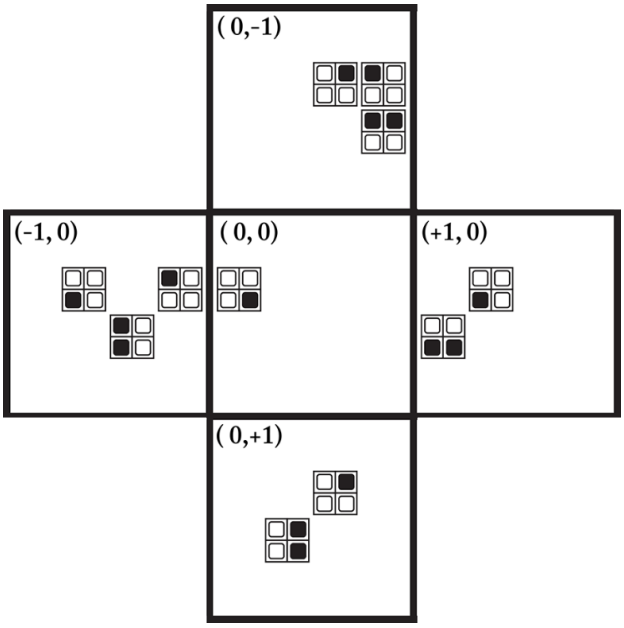


Figure 9: Part of propagator [4]

2.3 OBSERVE

From now on the incremental generation process starts. This is the same for both the models. During this process, the algorithm keeps track of which patterns/tiles can possibly exist on every location. And keeps this in a data structure called wave (in loose reference to a quantum wave function) that holds an array of Boolean values as long as the number of patterns for every position on the grid. At the start the wave is set in a completely unobserved state, meaning all Boolean coefficients are true and that every pattern/tile can exist on every location. Over the course of the algorithm options of patterns/tiles will be removed and the domains of the wave will shrink. This until each location only has one possibility left.

In the third step it searches for the location on the gid with the lowest nonzero entropy. The location with the lowest entropy is the location with the fewest possible patterns/tiles that can still be assigned to it. Once the location with the lowest nonzero entropy is found, it is assigned a random pattern/tile from its possibilities. The random assignment is influenced by the frequency in which the patterns are present in the sample image. This is done to make sure patterns are as frequent in the output image as in the input sample image. After the location has been assigned a pattern/tile, it is added to a stack to indicate that its neighbours need to be updated.

2.4 PROPAGATE

In the fourth step, the algorithm goes over the locations on the stack of which the neighbours need to be updated and update them. For this, a constraint solving method called Arc Consistency is used. This method “ensures that a value only appears in a domain of a variable if there exists a valid value in the domain of related variables such that constraints over those variables could be satisfied”[4]. Or more simply said it ensures that a pattern only appears in the possibilities of a position if there exists a valid pattern in the possibilities of a neighbour such that the constraints in the propagator are satisfied. This algorithm is also known in the field of graphics as flood fill.

The algorithm starts with the first location on the stack and checks its neighbours whether they need to be updated. For this the propagator build in step 2.2 is used. If they do need to be updated, they are placed on the stack and will be updated the next iteration. The algorithm will end if there are no locations left on the stack that need to be updated.

After the propagation, some locations are updated and have lost some possibilities thus reducing the entropy. Next the observe step is done again, picking the least nonzero entropy location and assigning it. Next, the changes are propagated. This continues until there is no more entropy in the system, i.e., all locations have a single pattern/tile assigned to it.

2.5 OUTPUT THE OBSERVATIONS

As last step, the observations are outputted. For the overlapping model this is done by picking the lower left tile in each of the single pattern possibilities for each location.

There can also be taken advantage of the side-effect of each cell having an array of potential patterns/tiles and output a partially finished image after each cycle of observation and propagation. This allows for enticing visualizations. On Fig. 10 two of the possible outcomes of the generation are visible.

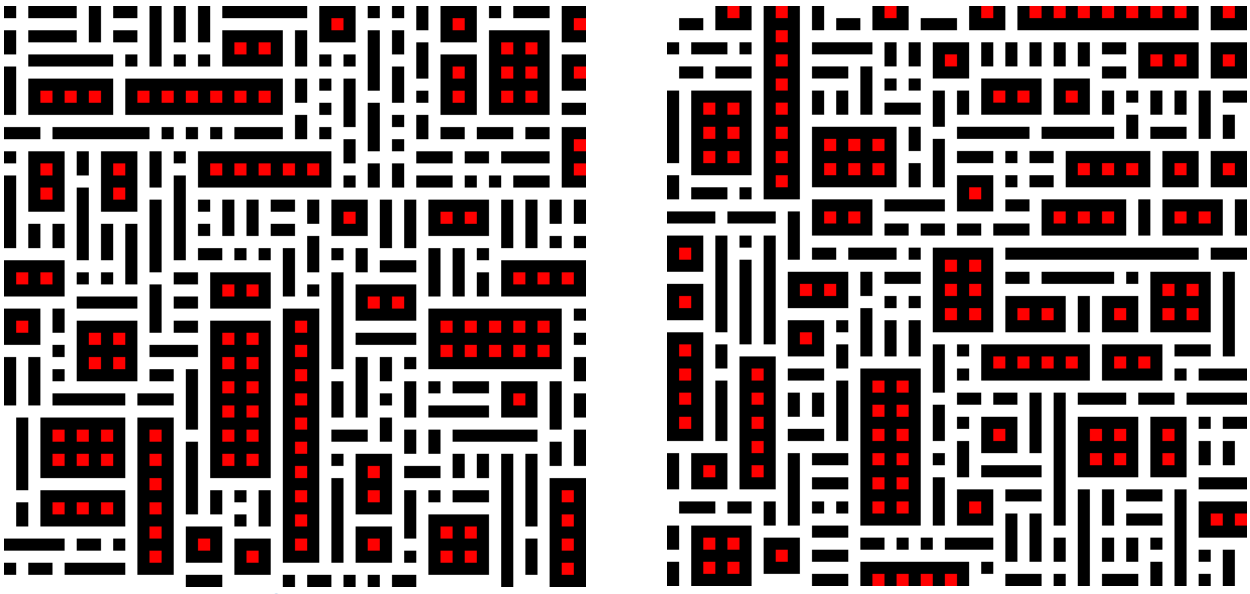


Figure 10: 2 outcomes of generation

SIMPLE TILED MODEL

The simple tiled model is a different way to use the WFC algorithm. It differs form the earlier discussed overlapping model in three steps:

- Get the patterns form sample
- Build propagator
- Output the observations

The simple tiled model is like an overlapping model with a sampling size of NxM = 1x2. But in this model, patterns of tiles are not used, but the tiles themselves. And instead of sampling from an example, constraint relationships are defined explicitly. So, there can be read in directly from a json file which tile can be next to another, which makes the propagator step far simpler. The get the patterns from sample step is also easier, because the different kind of tiles are also stored in the json file and can be read in.

The rest of the algorithm is the same for the simple tiled model as with the overlapping model. With the key difference being it works with tiles and not patterns. Except the output the observations step is also a bit simplified. Here the only tile that remains a possibility for each location is picked of course.

3. WFC FOR GENERATING HOTEL LAYOUTS

TILES

The hotel layout has four distinct types of rooms: a normal room, a hallway, a bathroom and elevator room. Around the outside there is an outer wall. All these rooms consist of tiles and each room has its own tiles with own aesthetic. There are six kinds of tiles: floor, wall, inner corner, outer corner are the simple tiles. The doors and windows each consist out of two pieces, a left and a right piece. They all have the same dimensions, so they fit on the grid. The tile set is completed to make sure that in whatever position with whatever neighbours there will always be a tile that fits. Important to note although the tiles are 3d models, they are arranged on a 2d grid plane in the x and z directions of unity.

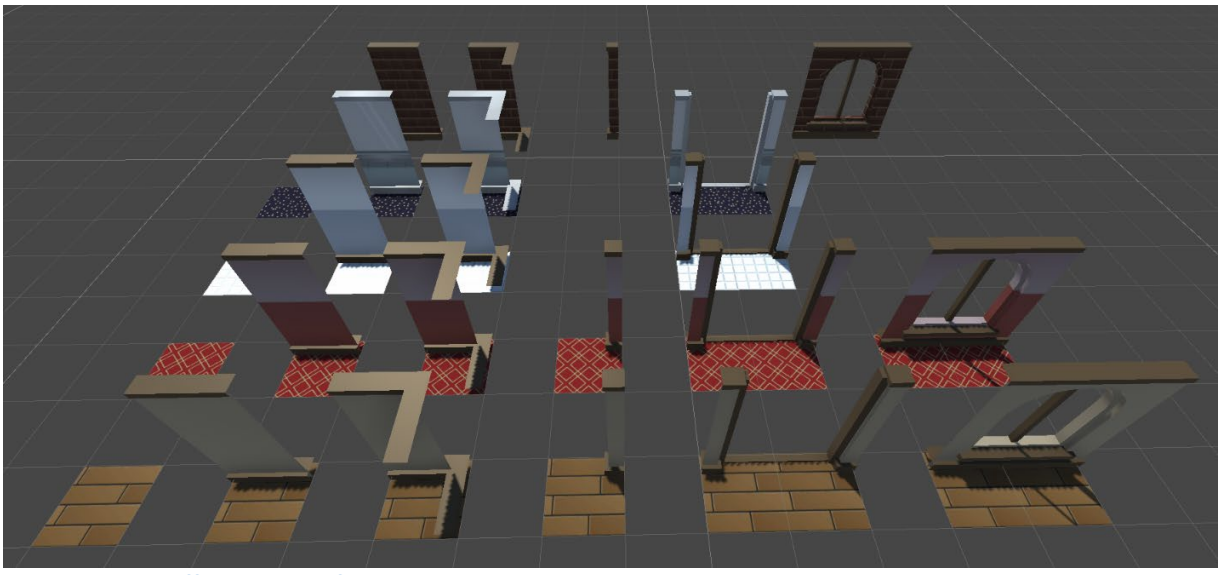


Figure 11: The different kind of tiles

USING THE ALGORITHM

In the overlapping model the example image is substituted for an example arrangement of tiles. And this is then used instead for deriving patterns with a chosen size from. The patterns can also be rotated and reflected to get more patterns and increase the chance of finding fitting patterns. The size of the patterns used to sample from the example can be adjusted. If it is increased, it will make the output resemble the input more closely. But the downside is that the patterns are also more difficult to fit together. This reduces our success rate.

The WFC algorithm can not guarantee that it always succeeds in finding a solution. Sometimes a contradiction can occur that makes the algorithm fail. This can happen because it starts off by saying every pattern/tile can be at every place. Each time a pattern/tile is locked in, the possibilities around it that are not valid anymore are removed. This can lead to some positions that have no possibilities left. If this happens, the simplest thing to do is to start over again, pick a new random seed and try again. Another option is to add backtracking, meaning that the algorithm goes back to the last pattern/tile that it assigned, invalidate it and choose another one.

The WFC algorithm also uses a weight heuristic. This can be used to tell the algorithm to prefer certain tiles more than others. In the overlapping model the weights are the frequency of how many times a pattern occurs in the input sample. To improve the output some of the weights of patterns can be set to a custom amount. For example, if a dense level is wanted, the weight of the pattern with all empty tiles can be reduced. Or if there are too much big open rooms, the weight of the pattern with all room floor tiles can be reduced. In the simple tiling model, the weight data is also explicitly defined and can be read in.

Because the WFC algorithm is a constraint algorithm that keeps track of all the pattern/tile possibilities for each location, it can be used to extend a level. This is when a level designer starts a level and asks the algorithm to finish it. The extension of a level can be done fairly easy with WFC. It is done by setting every pattern/tile from the input to extend as the only possibility for that position before the first observe step. And then propagating these changes.

Upon testing, a problem often occurred on the borders of our output, there were often rooms that were open, see Fig. 11. This is not desired because every generated layout should have an outside wall around it. As solution the same thing as the level extension can be done. Setting the empty tile/pattern as only possibilities on the border positions before the first observe. And then propagating these changes. This solved the problem because in the sample/constraints file only the outside wall can be next to the empty tile/pattern.

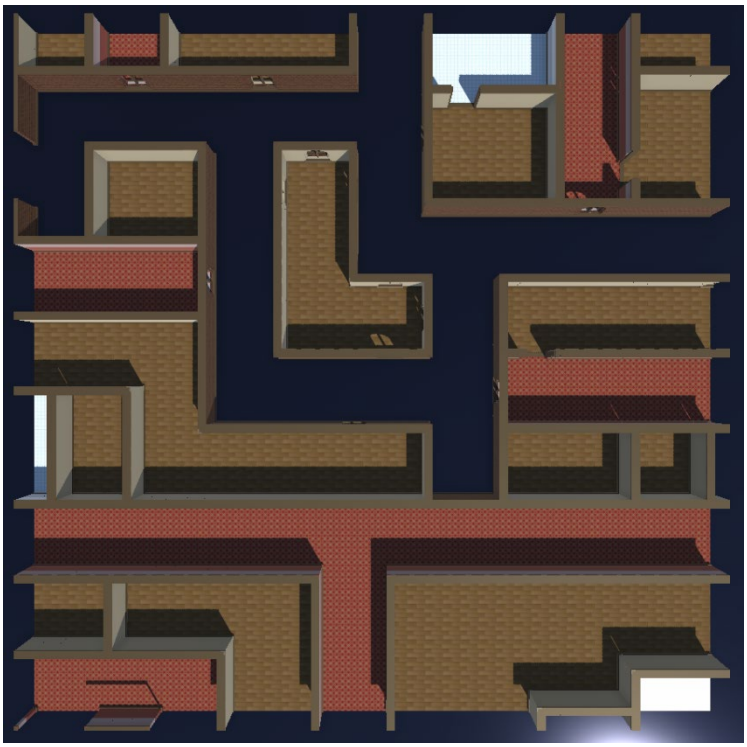
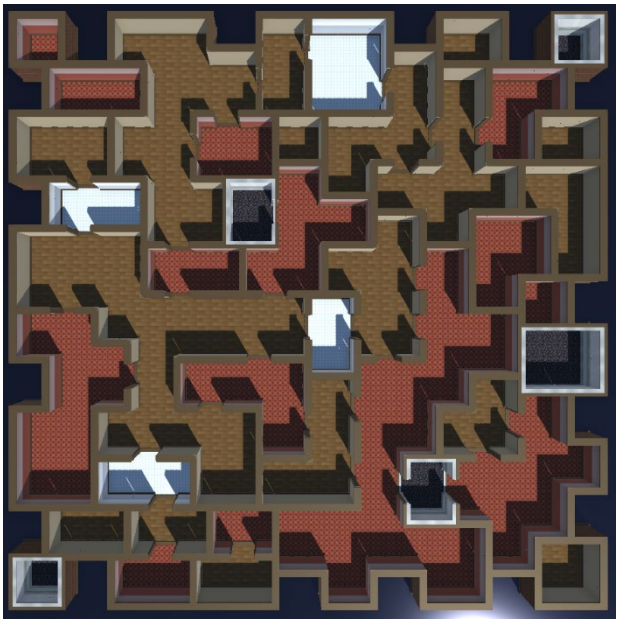
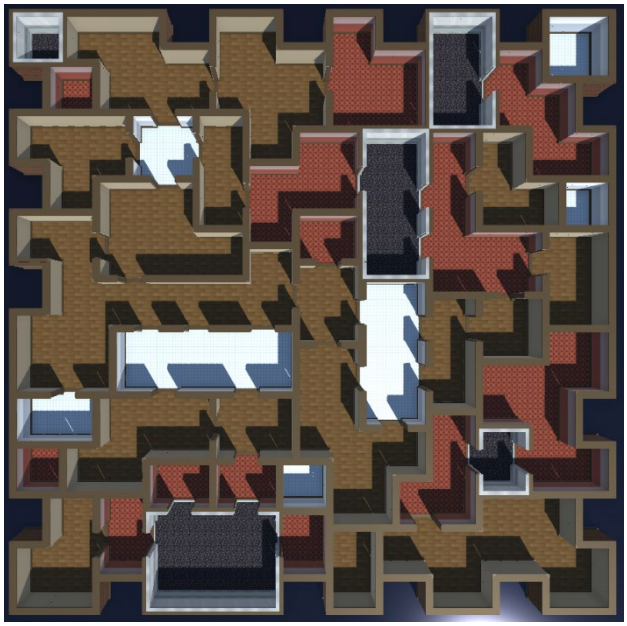


Figure 11: Example of open rooms at border problem

To improve the output of the overlapping model symmetry can be used. Here the patterns are reflected and rotated to create more variations of them. It means that not every pattern in the output maps directly to a place on the input but can be a reflected or rotated version of it. Using symmetry also greatly improves the success rate of the algorithm, because the increased number of patterns better fit together.

EXPERIMENTS & RESULTS & DISCUSSION

SIMPLE TILED MODEL

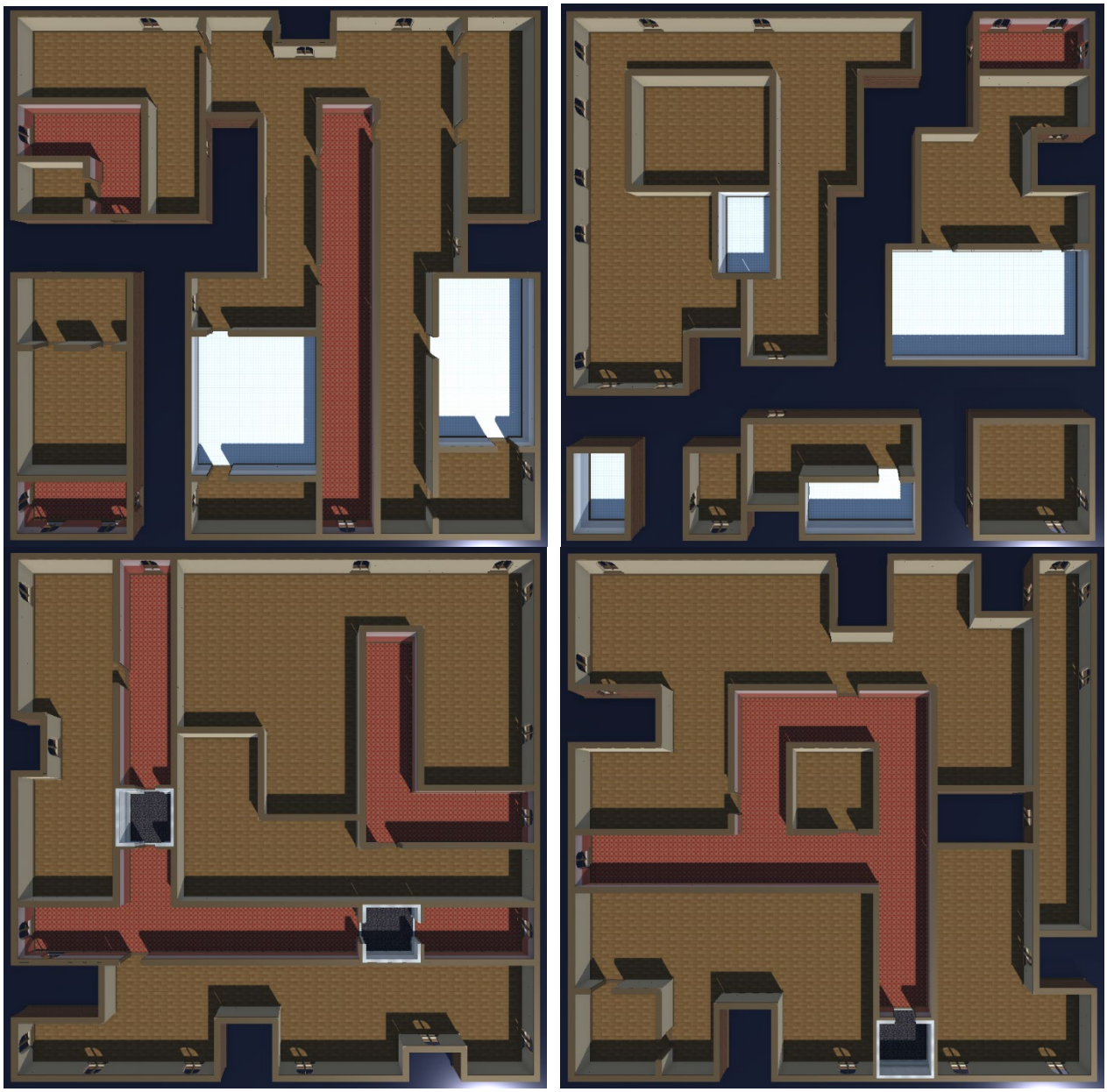



Figures 12 & 13: Results of simple tiled model

It is apparent that due to the simple tiled model only looking at the neighbour constraints, it lacks a higher-level understanding of what a hotel layout should look like. This is especially noticeable when looking at how the rooms are connected. Some rooms have no doors while others have too many of them. Also, rooms are far from square as would be expected from them and go all over the place. There are even some rooms that become an island and don't connect to the rest of the layout.

However, the biggest problem is that the simple tiled model does not know the concept of a hallway. A hallway should be long and narrow, and it should connect all the rooms with each other. But this model just treats them the same as normal rooms. The same issue with elevator rooms. Preferably there would be only one elevator that is a small square room with one door to a hallway. But there is no way for the algorithm to know this. In general, the layouts of the simple tiled model look unnatural, chaotic and not logical.

OVERLAPPING MODEL



Figures 14, 15, 16 & 17: Results of overlapping model

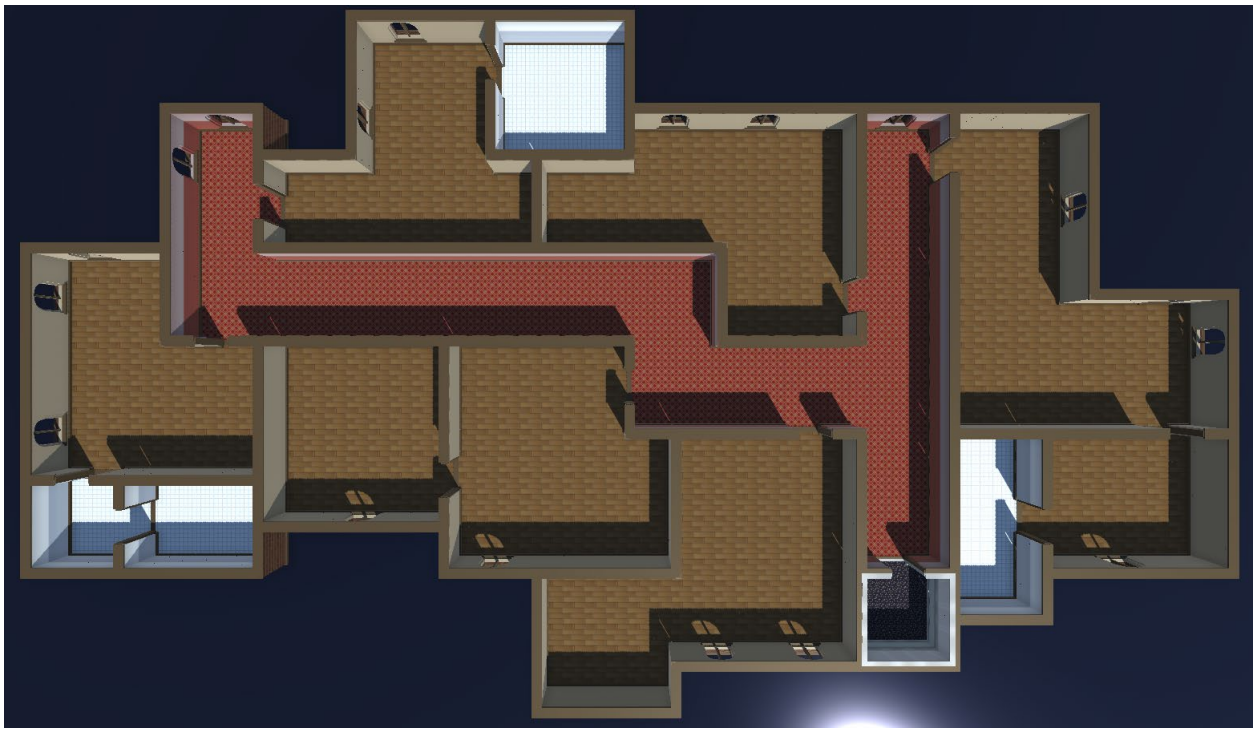


Figure 18: The sample layout for the overlapping model, original designed layout of the game Boo-Ya!

The overlapping model is a small improvement over the simple tiled model. But the results can vary a lot ranging from ok to terrible. The best results were gotten with a sample size of three and all symmetries. A sample size of at least three is needed to make sure the hallways (four tiles wide) are understanded correctly by the algorithm. Otherwise, they can become too wide. A sample size of four leads to too many patterns that do not fit together. Symmetry helps a lot, if not used the output always looks like the patterns of the input sample but arranged worse. The symmetry brings some of the wanted variety in the output that makes the layout interesting.

The layout looks a lot more logical than the simple tiled model. Most of the rooms are square like.

One problem is that often not every room type is used. The normal room is always present, but the elevator, hallway and bathroom types can be absent. In the simple tiled model this was not the case. But the overlapping model has a harder time to get a good mix of room types. A simple fix can be to use the extend level functionality to force types of rooms to be at certain places. For example, if just one elevator floor tile is placed in the input to extend. It is guaranteed that the elevator and hallway room types are present, because an elevator can only have a door with the hallway like in the sample layout Fig. 18. Instead of a level designer doing this manually, a small procedural algorithm can randomly place an elevator and bathroom floor tile. By doing this it is then guaranteed that every room type will be present.

The problem of rooms existing with no doors to other rooms or no connection to the rest of the layout still exists. It is also not always guaranteed that every room is accessible form the hallway. This problem could be solved with an algorithm that runs after WFC and finds rooms that have no connections and either makes a connection or deletes the room. But ideally the WFC should be modified to only make connected layouts.

Using the overlapping model to extend levels started by level designers could facilitate their work. But the results are not consistent and good enough to be used directly in game.

PERFORMANCE

The only thing that is measured is the core algorithm loop (observe and propagation steps), steps get patterns from sample, build propagator and output observations are not measured.

Looking at the performance graph Fig. 19, it is directly visible that the overlapping model a lot slower is then the simple tiled model. It also scales a lot worse. This is mostly because the overlapping model has a lot more patterns than the simple tiled model has tiles. Making the propagation take significantly longer.

It is also visible that in the simple tiling model the failure rate increases with the output size. The algorithm was tested 10 times. At the right end of the graph, it is visible that for an output size of 200 it failed 9 times. So almost every time the algorithm had to restart. It is here that adding backtracking can make a significant difference on the execution time.

In the overlapping model no failures were measured. This is because with the symmetry a reflected version of each pattern is created that makes it so that for each situation the reflected pattern version will always fit.

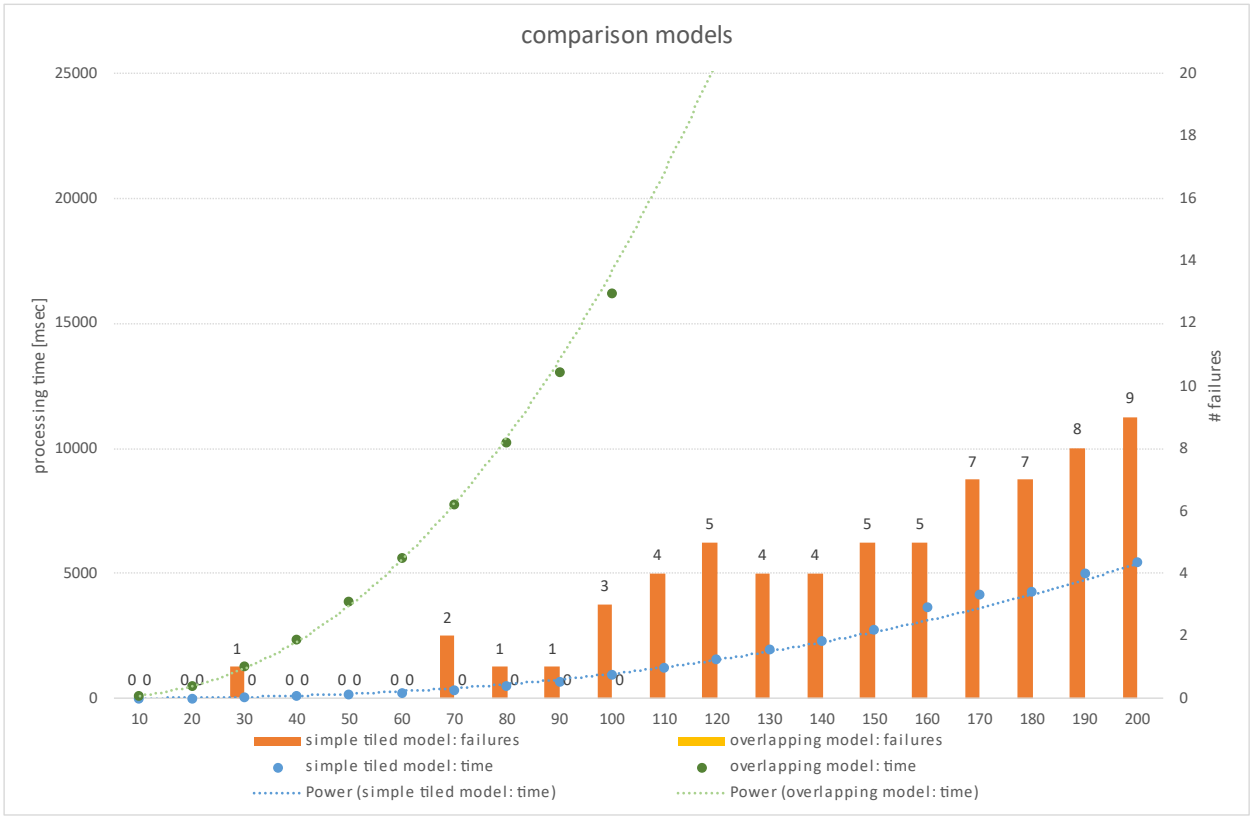


Figure 19: Performance of the WFC algorithm

CONCLUSION & FUTURE WORK

One thing that could be interesting to add to WFC are extra heuristics. So does Oskar Stålberg add a navigation heuristic to his implementation of WFC, see Fig. 20. He adds data to every tile about how it is navigable and keeps track of a navigable network when generating. When observing he then prefers to choose tiles that expand this navigable network. This could also improve our output and make sure every layout is playable. It could theoretically fix the biggest issue that is still present. It can make sure every room is accessible, fixing rooms without doors and isolated rooms without connection to the rest of the layout. Heuristics could also be used to encourage variation in room types or encourage long hallways.

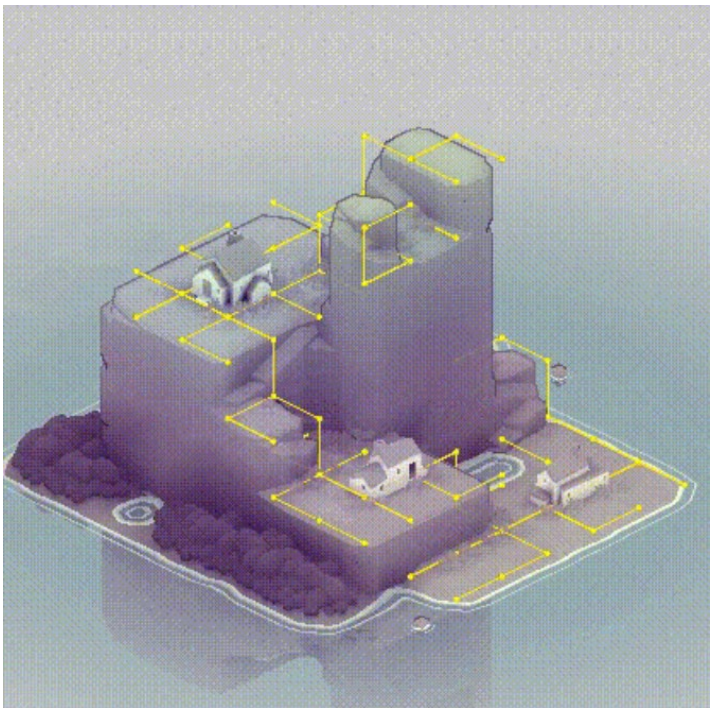


Figure 20: Navigation heuristic in bad north [10]

Working with a json file to define neighbour constraints in the simple tiled model is tedious. One small mistake in it can lead to the algorithm not working. It is also annoying to add new tiles, because then constraints of other tiles will need to be changed too. A solution to make it less tedious is to automatically generate the constraints. This can be done by finding the outer edges of the 3D models of the tiles and hashing them. And if an edge of another tile has the same hash, then the tiles can be next to each other.

Adding backtracking as earlier discussed could also improve the algorithm a lot. Taking a step back and trying to place a different pattern/tile could significantly improve performance over globally restarting the algorithm when a contradiction is found.

The conclusion of the research is that the layouts created by WFC are not game ready. And require more processing or extra heuristics (like the navigation heuristic) for them to become useful. WFC works great for filling a space with tiles following neighbour constraints. But if there are more requirements that go further than just any allowed mix of the tiles that are given, WFC will struggle to meet these requirements.

BIBLIOGRAPHY

[1] Gumin Maxim, “mxgmn/WaveFunctionCollapse: Bitmap & tilemap generation from a single example with the help of ideas from quantum mechanics,” 2016. https://github.com/mxgmn/WaveFunctionCollapse (accessed Dec. 17, 2021).

[2] P. Merrell, “Example-Based Model Synthesis.”

[3] P. Merrell, E. Schkufza, and V. Koltun, “Computer-Generated Residential Building Layouts.”

[4] I. Karth and A. M. Smith, “Wave Function Collapse is constraint solving in the wild,” in *ACM International Conference Proceeding Series*, Aug. 2017, vol. Part F130151. doi: 10.1145/3102071.3110566.

[5] “unity-wave-function-collapse by selfsame.” https://selfsame.itch.io/unitywfc (accessed Dec. 17, 2021).

[6] “Proc Skater 2016 by arcadia-clojure.” https://arcadia-clojure.itch.io/proc-skater-2016 (accessed Dec. 17, 2021).

[7] “Wave - by Oskar Stålberg.” https://oskarstalberg.com/game/wave/wave.html (accessed Dec. 17, 2021).

[8] “Maxim Gumin on Twitter: ‘Procedural generation from a single example by wave function collapse https://t.co/NcRn5iIOFz https://t.co/VOZwtSvIvO’ / Twitter.” https://twitter.com/ExUtumno/status/781833475884277760 (accessed Dec. 17, 2021).

[9] O’Leary Martin, “mewo2/oisin: Oisín: Wave Function Collapse for poetry.” https://github.com/mewo2/oisin (accessed Dec. 20, 2021).

[10] “Oskar Stålberg on Twitter: ‘A somewhat major breakthrough in my procedural generation: a heuristic for deliberately generating navigable levels https://t.co/qoO3rElw0W’ / Twitter.” https://twitter.com/OskSta/status/917405214638006273 (accessed Jan. 24, 2022).

APPENDICES

THE RAW DATA FOR THE COMPARISON MODELS GRAPH FIG. 19.

output size	simple tiled model: time	simple tiled model: failures	overlapping model: time	overlapping model: failures
10	4,625	0	98,125	0
20	19,625	0	520,125	0
30	52,125	1	1287,625	0
40	95,375	0	2369,75	0
50	159,625	0	3896,125	0
60	236,5	0	5626,375	0
70	356,125	2	7804,875	0
80	498,875	1	10266,75	0
90	684,25	1	13066	0
100	946,625	3	16257,75	0
110	1221,625	4		
120	1596,25	5		
130	1989,125	4		
140	2343,25	4		
150	2789,25	5		
160	3653,25	5		
170	4197,25	7		
180	4314,5	7		
190	5024,875	8		
200	5466,5	9		