Wauthier Tristan

Using Wave Function Collapse algorithm for 2D and 3D level generation

Graduation work 2022-2023

Digital Arts and Entertainment

Howest.be

TABLE OF CONTENTS

ABSTRACT

Wave Function Collapse (WFC) is a greedy algorithm that introduces a new way to procedurally generate both 2D and 3D worlds. One of the big advantages of using WFC is the customizability available, because of the tile set input required and possibly additional constraints and options.

In this paper, a clear and cut way to implement 2D Wave Function Collapse to generate game levels is documented and extended into 3D. Extending to 3D requires minimal changes to the algorithm. Multiple extra options to customize the result into a desired state are introduced. Additional constraints are here used to generate an island level. The impact of each of these options and additional constraints on execution time is then measured and evaluated.

From the experiments, it is concluded that while extra adjacency constraints increase execution time, they can be used for optimizations purposes as well.

KEYWORDS

Wave Function Collapse – Procedural Content/ Map Generation – Computer Games – Constraint Solving

INTRODUCTION

Procedural Content Generation (PCG) offers game development teams the opportunity to spend fewer resources on content generation. PCG has been used to create all kinds of content ranging from sound to 3D models and quests [1] [2] [3]. It is getting more and more popular due to the growing size of games and the content within them. Using Procedural content generators requires an upfront cost to implement, but if implemented correctly can save huge amounts of development time and resources. PCG can be used both during development and runtime to provide a different experience each time a game is played [4] [5].

PCG has also been used for a long time in level design. Designing good levels is an iterative and resource-consuming process, therefore many developers decide to use PCG if possible. Many games like Minecraft, No Mans's Sky, and the entire genre of Roguelikes draw their main appeal from their random world generation and the aspect of exploring the unknown. In games like these PCG contributes a lot towards the game's replayability. [6] [7] [8]

The Wave Function Collapse (from here on referred to as WFC) algorithm is a greedy, computational algorithm that is seeing more and more popularity due to its ease of use. It is loosely based on the wave function quantum mechanics, where a particle is in a superposition until it is observed. This means a particle can be in any possible state until it is observed. [9]

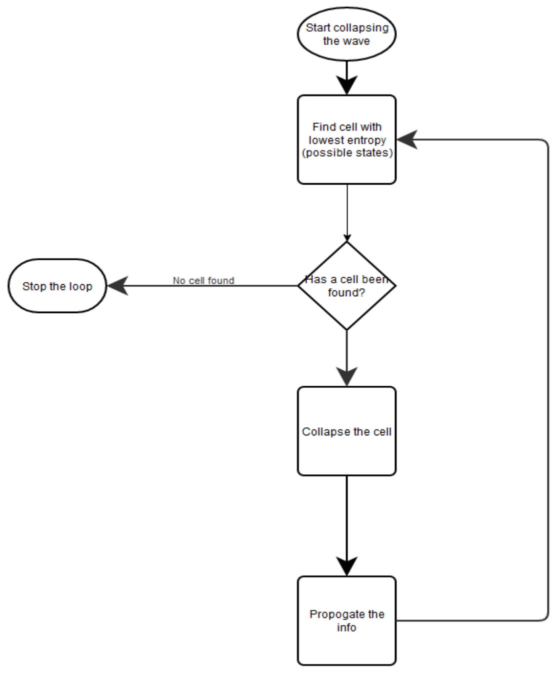


Figure 1: Flowchart representation of WFC

In WFC this is represented by initializing the level as a grid filled with cells, wherein each cell can be any possible state from a list of input tiles. The algorithm goes through the list each iteration finding the cell with the lowest entropy. Entropy represents the amount of chaos within a cell, which is represented by its number of possible states. Once this cell has been found it gets observed, causing it to collapse to its final state/tile. This information is then propagated to the neighboring cells, where the possible states of those cells are then updated based on adjacency constraints and propagated, and so on. These adjacency constraints are conditions for possible states to align and there can be as many added on top of each other as desired. Once all cells have been notified of the changes, the algorithm moves on to the next iteration and keeps iterating until all cells have collapsed, giving us our result. [9] [10] [11]

The Wave Function Collapse algorithm is greedy. WFC looks for local possible and well-fitting solutions, instead of considering the entire map. It also does not reconsider made decisions at any time, leading to possible failure states. [12]

This paper will investigate the WFC and how it can create 2D and 3D levels based on input tiles. First, a 2D version of the algorithm will be implemented and it will then be investigated how this implementation can be adapted to work for 3D levels. Afterward, additional adjacency constraints will be added to the 3D algorithm and their impact on the overall performance will be calculated.

This is the research question this paper will focus on:

- **What are the necessary modifications to the 2D wave function collapse algorithm to 3D space and what is the impact of adding extra adjacency constraints?**

RELATED WORK

Procedural Content Generation (PCG) focuses on the use of algorithms and software to automatically generate game content. It aims to create high-quality content for games, without the need for developers or artists to manually create it. PCG makes it a great toolset to reduce the time and resources spent on content generation. However, it comes with an upfront cost to implement and requires a strong understanding of game design, data structures, and algorithms. If one would use machine learning for PCG, large datasets would also be required.

Figure 2: Screenshot of Elite (1984)

PCG has multiple types of applications in games, like level creation, character creation, and quest generation. This also means that it can be used by teams in different scenarios. In Elite (1984), PCG was used to dynamically generate levels to create huge universes for the player to explore, without using too much memory. It is often used to generate 3D levels at runtime, offering more game replayability. It can also be combined with AI systems to create a dynamic difficulty system, making it change game parameters based on the detected skill level of the player. [13] [14] [15]

One of the main challenges of PCG is not just creating the system but retaining some form of control of it. It is easy to generate a 3D level, but the difficulty lies in making sure the level fits the game. This is where Wave Function Collapse (WFC) comes into play. WFC uses tiles as input, meaning that tiles can easily be enabled or disabled to fit a game/ level design choice. It is also possible to add constraints to the model, further enabling customizability (e.g., a sand tile cannot be directly connected to a stone tile). [16]

1. MAXIM GUMIN

WFC was originally invented by Maxim Gumin, who created it as an algorithm for texture synthesis. The way his original algorithm worked was:

1. Read the sample texture and count the detected patterns.
2. Create the output pixel array for the texture and initialize it with all the patterns and booleans representing whether a pattern has been discarded yet.
3. Collapse the wave, by repeating the following steps.
    a. Observation: Find the wave element with the lowest entropy (if none was found, break the cycle as this means the wave has fully collapsed). Collapse this element based on the patterns that have not been discarded yet.
    b. Propagation: relay the info of the collapsed wave element to its neighbors. Discard patterns that can no longer fit and propagate this info to those neighboring cells. Repeat this until all cells that need to discard patterns have done so.
4. The wave should now either be fully observed, which gives a correct output image, or end in a contradictory state. In the first case return the output pixel array, in the second case return nothing.

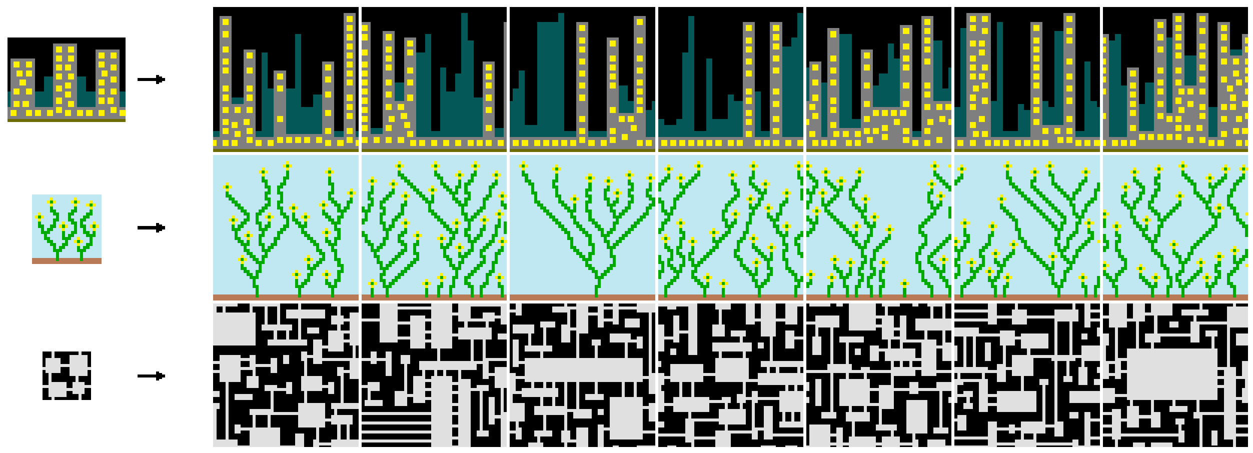


Figure 3: Gumin's implementation of WFC for texture synthesis

He then also introduced a 'Simple Tiled Version' in which the algorithm does not recognize patterns itself but uses input tiled patterns. The biggest change this has on the algorithm is that the propagation stage now is adjacency constraint propagation. This means that the algorithm needs predefined rulesets to detect whether tiles can fit together or not, an example could be edge color detection. The implementation of this paper will be based on Gumin's Simple Tiled Model [17].

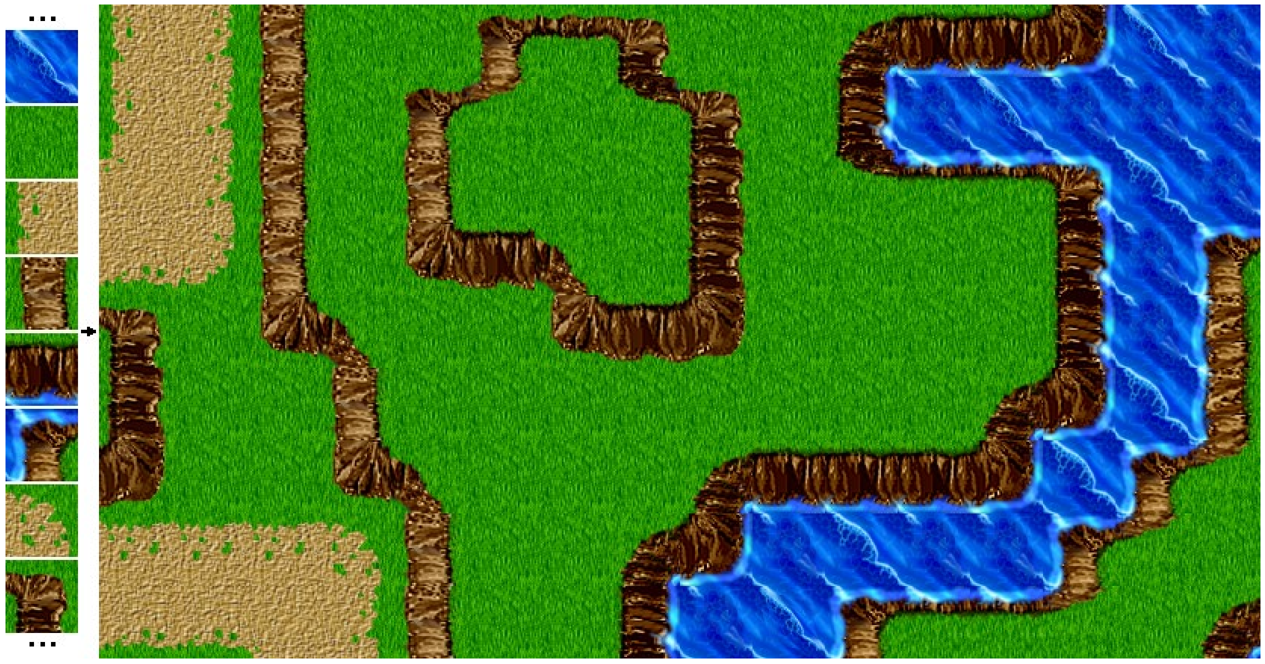


Figure 4: An example of Gumin's Simple Tiled Model of WFC

2. OSKAR STÅLBERG

2.1. 2D

When people discuss WFC, Oskar Stålberg is often mentioned. Stålberg is known for his "*Wave*" demo, which showcases how 2D WFC works when generating a 2D image. This demo showcases how the algorithm thinks and uses color detection to see if tiles can fit next to each other. This is also the implementation that this paper's 2D implementation is based on [18].

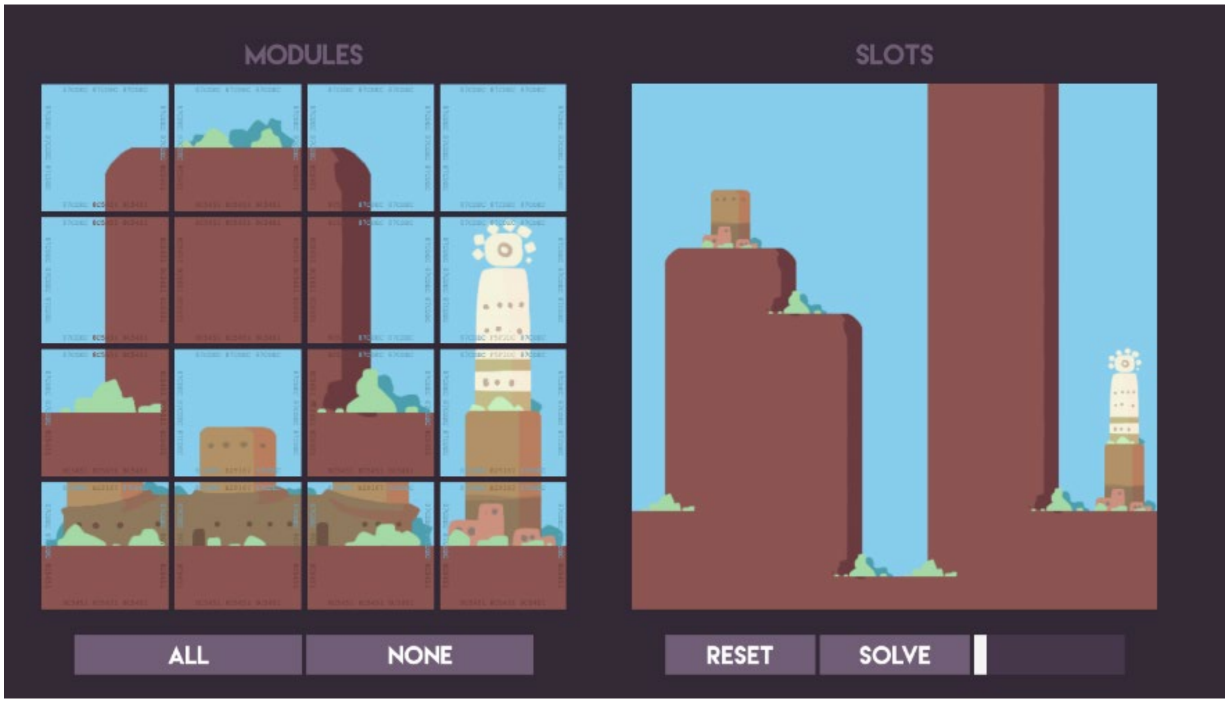


Figure 5: Wave by Oskar Stålberg

2.2. 3D

Stålberg also expanded WFC to work in 3D spaces and showcases this in the games Bad North and Townscaper. These games contributed vastly towards WFC and made it more known. He also proposed the idea of calculating the adjacency constraint when importing the 3d tiles from the modeling software to the game engine. The edges on all faces are scanned and compared to the other tiles to know which can fit together. The implementation this paper provides is also based on this principle.

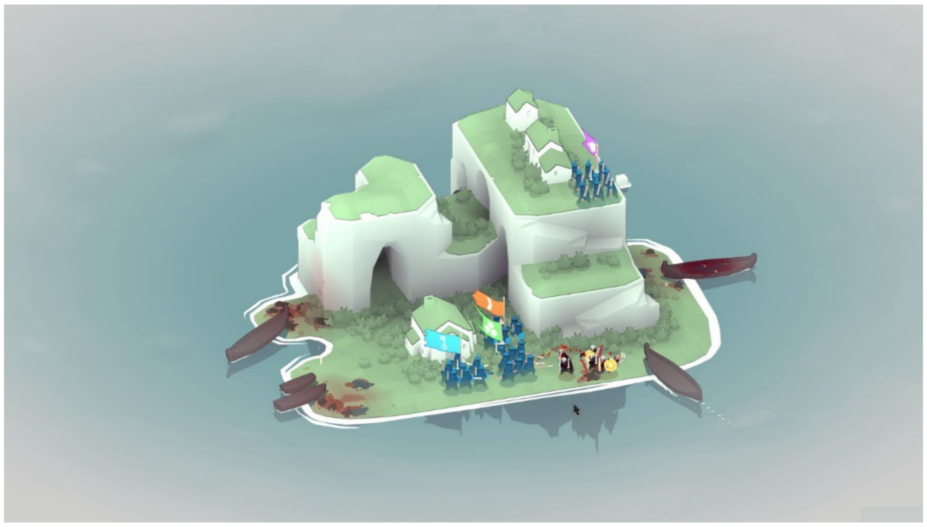


Figure 6: Screenshot of Bad North from the steam page, showcasing a WFC-generated island.

3. BORIS THE BRAVE

When people talk about Tiles 3D Wave Function Collapse, Boris the Brave is often mentioned. He created Tessera [19], a Unity add-on that procedurally generates 3D levels using Wave Function Collapse. He is also the creator of DeBroglie [20], a C# library and command line program that generates 2D tile maps using WFC. On this page [21] he gives a lot of advice on implementing and testing WFC and possible constraints to add.

WEIGHTED ENTROPY

Boris explains that the process of picking a cell is of great importance. Choosing the cell with the least entropy is preferred as it minimizes risks later. If the algorithm would wait too long with collapsing some cells, the algorithm could run into a contradiction.

For calculating the entropy there are two options. If all tiles have an equal chance of being collapsed into, it suffices to use the number of remaining modules as the entropy value. If some tiles have a bigger chance of being chosen than others, we need to sum over all the remaining modules and choose the cell that minimizes:

$$Entropy = -\Sigma p_i log(p_i)$$

In this formula, p_i represents the probability of a module within the current cell. We sum all these and take the inverse as the entropy for the current cell. [22]

TILE SHAPE

Tiles in WFC don't necessarily have to be square (2D) or cubes (3D). Boris mentions that it is possible to use WFC with hex grids or more unusual surfaces. One possible example of an unusual surface is a sphere.

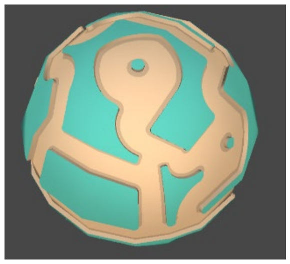


Figure 7: Example of a WFC generated sphere map.

He also notes that a practical extra constraint would be to allow some tiles to span across multiple tiles. This is because WFC doesn't usually generate large structures by itself, making large maps look automatically generated. Balancing normal WFC and forced big structures can clean up the look of a level. [22]

FIXED TILES

Being sure of certain tiles when generating a map is often desired. This might be needed for fixed-level entry and exit points, drawing boundaries, or letting WFC fill in a pre-designed level. Fixed tiles are added before the algorithm starts collapsing the wave. [21]

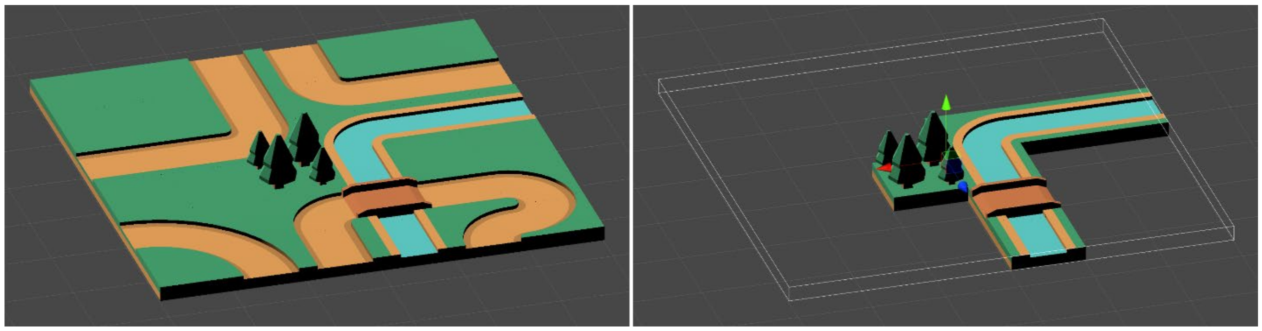


Figure 8: Level with fixed tiles

PATH CONSTRAINT

When generating rooms or very densely populated areas it is very valuable to add a path constraint. This constraint forces the algorithm to look for a path connection between 2 points on the map. For example, when generating rooms, it can add doors so rooms are connected and for generating wood it can create a nice walkable path. [21]

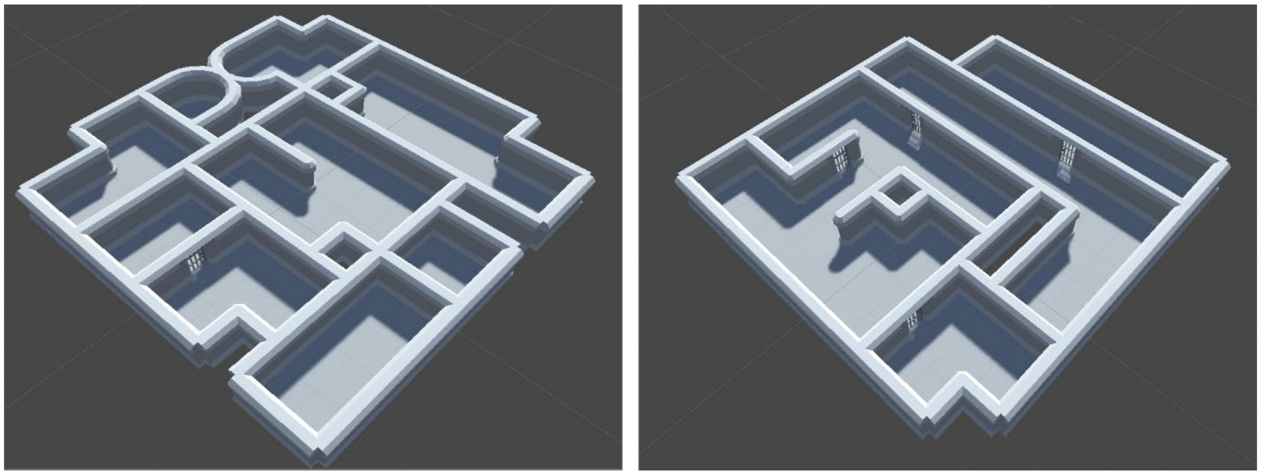


Figure 9: Generated level without path constraint (left) and one generated with (right)

4. MARIAN 42

Marian 42 created an infinite procedurally generated world using 3D Wave Function Collapse. When walking through the level, new chunks are generated at runtime. To be able to do this he added multiple extra constraints. [23]

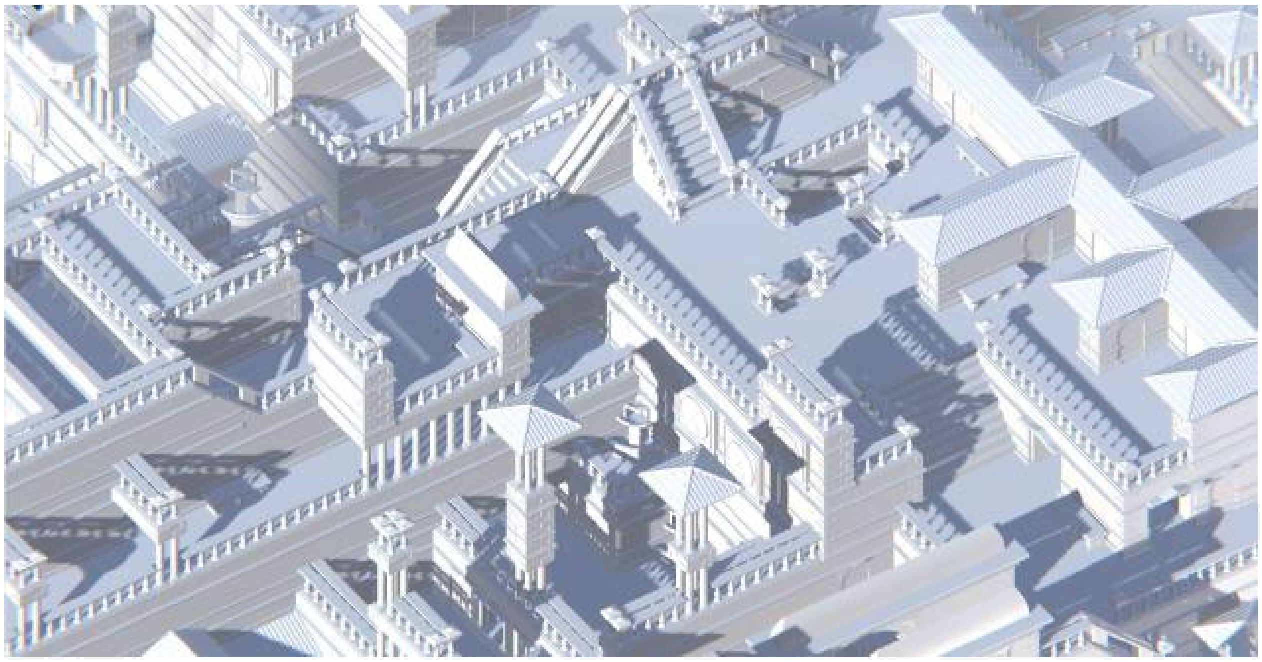


Figure 10: Top-down view of a chunk by Marian42

SPECIAL TILES

Marian 42 added two special tiles to their tile set. He added an empty tile and a solid tile. The empty tile would be added whenever the algorithm would need to place a block of air and a solid tile for occluded tiles.

BOUNDARY CONSTRAINTS

The behavior of WFC at the edge of maps is not clearly defined. If left untouched it could end up with holes in the floor or missing roofs at the top of the map. Boundary constraints can prevent these unwanted behaviors. The two boundary constraints Marian42 added, were assuring all the faces at the top of the map have an empty connector and all bottom faces have a solid connector. This is easy to use in finite maps, as the map size is known at the start and thus, all top and bottom cells can first be assigned. For infinite maps, however, it becomes a lot more difficult. This falls out of the scope of this paper.

ERROR STATES

WFC can reach a contradiction while collapsing cells. A contradiction happens when a cell needs to get collapsed, but no possible states are possible because of the neighboring collapsed cells. For finite worlds, it is sufficient to restart the collapsing of the wave. For infinite worlds, backtracking would need to be implemented.

CASE STUDY

1. INTRODUCTION

This chapter will give an overview of the entire implementation of the algorithm. It provides a step-by-step guide to reproduce this implementation of 2D and 3D Wave Function Collapse in Unity. The 2D tiles were made in Adobe Photoshop and the 3D tiles were made in Autodesk 3DS Max.

Additional customizability options will also be added to the 3D algorithm and explained. The hypothesis is that these extra constraints will all increase the execution time substantially. This paper hypothesizes that, while not adjacency constraints will influence the execution time equally, the execution time will at least double, when using all these constraints together.

2. WAVE FUNCTION COLLAPSE GENERAL ALGORITHM

We start WFC by generating a grid of a given size with empty cells. These cells hold all possible modules they can, meaning it is in a superposition. A module represents a possible state of a cell. This grid represents the "wave" that will get collapsed. We start the algorithm and thus start collapsing the wave. This algorithm loops over the following steps:

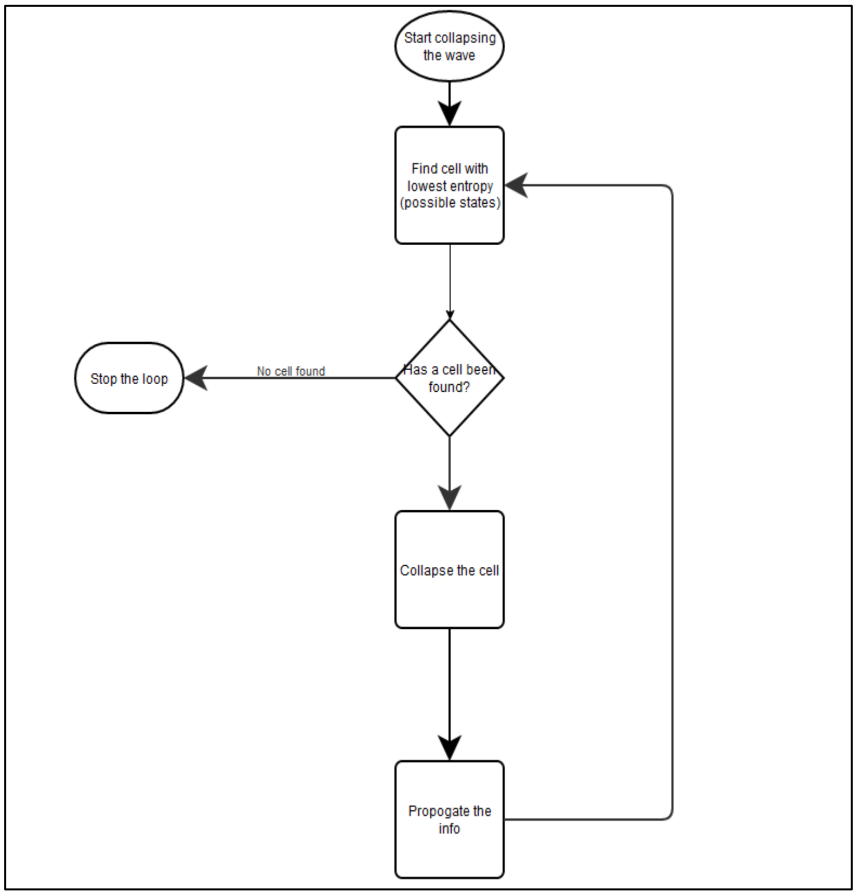


Figure 11: Flowchart representation of WFC

1. Observation: Start by looking for the cell with the lowest entropy. During this search, add some randomness using Unity's built-in random number generator when multiple cells with the same entropy are found. This assures we get a truly random cell in this case. Collapse the cell to a random module left in its possible modules.
2. Propagation: Relay the info of the collapsed cell to its neighbors. Discard modules that can no longer fit and propagate this info to those neighboring cells. Repeat this until all cells that need to discard modules have done so.
3. Search for the cell with the lowest entropy. Check if one has been found.
   a. Cell was found; repeat the loop from step 1.
   b. No cell was found; stop the loop as the wave has collapsed.

3. 2D WAVE FUNCTION COLLAPSE

To get WFC to work in a truly 2D environment (using sprites), a consistent way to check adjacency constraints is required. Therefore, edge color detection is often chosen [18] [9].

3.1. TILES

For the input in 2D WFC, sprites are used. A common and consistent way to check adjacency for sprites is edge color detection. Within Unity, it is very easy to get the color of a given pixel coordinate of a texture.

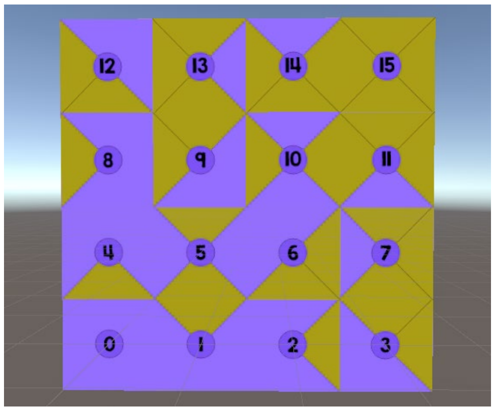


Figure 12: Used Wang tiles.

Wang tiles are advised [9] to use as input when implementing the algorithm. A Wang tile is a square with zero to four colored edges. If preferred each tile can be numbered to get a better overview of the generated map. The usage of Wang tiles when creating 2D WFC has multiple advantages:

- Wang tiles are simple and easy to create. The time investment in creating them is minimal.
- A set of 16 Wang tiles cover all possible states of a Wang tile. Thus, there is no need to use any kind of rotation or reflection on the tile.
- This set can create huge output maps without breaking visual coherence if the algorithm works correctly. If the output looks wrong, that means there is an error in the algorithm. [24] [25]

To get an idea of how the output would look using game tiles, simple road tiles can be used. These provide no explicit advantages or disadvantages but can just give an idea of how the generation of a 2D map with real game tile would look.

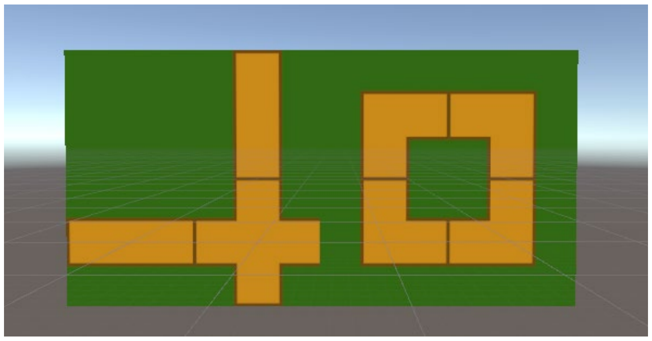


Figure 13: Used road tiles.

3.2. WAVE AND CELL REPRESENTATION

A cell in the wave is represented with the WFCCell2D class. This class contains a list of the possible modules (sprites) the cell can collapse into, a boolean that represents whether the cell has collapsed, and its collapsed state (which is null until it gets collapsed). The class also contains two public methods `GetEntropy()` and `CollapseCell()`. These methods are explained in later paragraphs.

The wave itself is represented by a 2D array of size [x, y], where x and y are integers given by the user, of WFCCell2D objects. This array gets generated and filled with objects when the algorithm starts. We give each cell object a new List of sprite objects coming from the given tiles.

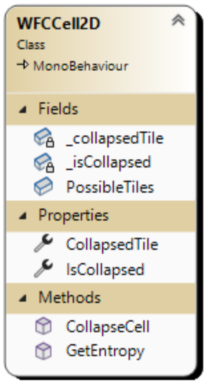


Figure 14: WFCCell2D Class diagram

3.3. OBSERVATION STAGE

The observation stage starts by looking for the cell with the lowest entropy. This is rather straightforward because the collapsed state is completely randomly chosen when collapsing the cell. The entropy of a cell is just the number of modules left in the list of possible modules.

To find the cell with the lowest entropy in the wave, the `GetLowestEntropyCell()` function is used. This function loops over all the cells in the wave and looks for the one with the lowest entropy. Collapsed cells are ignored and there is randomness introduced when multiple cells would have the same entropy. If no cells are found, this function returns a vector with negative indices.

Once the cell with the lowest entropy is found, it can get collapsed. This is done by generating a random number in the range of 0 and the number of modules left -1. Afterward, the sprite at this index is fetched and saved in the collapsed Tile object. A Sprite Renderer component is added to the game object to render the collapsed tile. After this, all that is left to be done is setting the collapsed boolean to true and emptying the list of the possible modules. This marks the end of the observation stage.

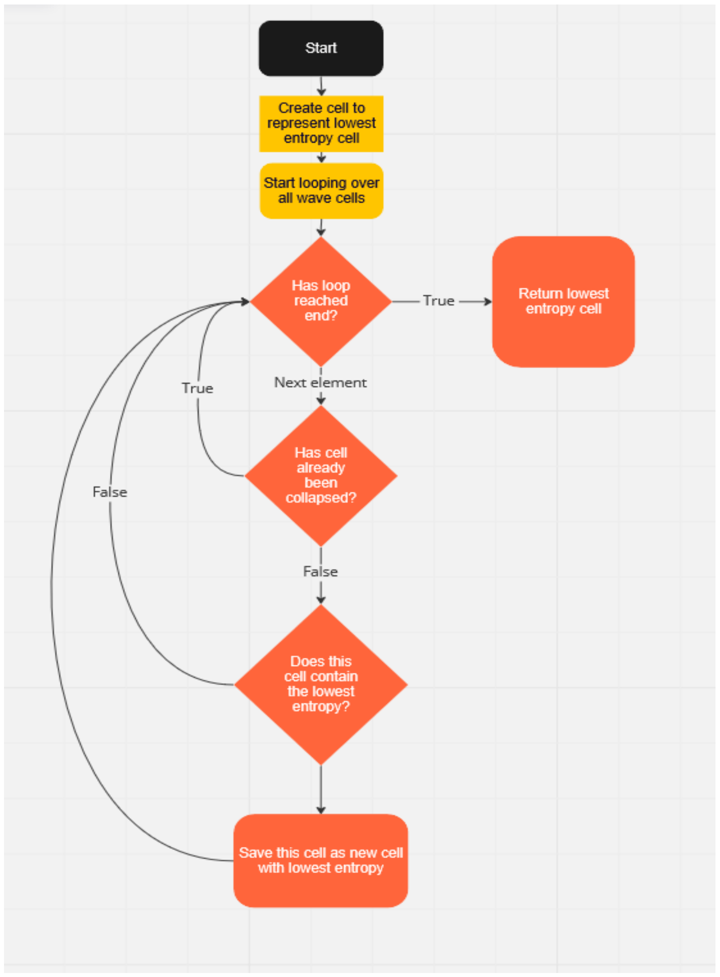

Start
Create cell to represent lowest entropy cell
Start looping over all wave cells
Has loop reached end?
True
Return lowest entropy cell
Next element
True
Has cell already been collapsed?
False
False
Does this cell contain the lowest entropy?
Save this cell as new cell with lowest entropy

Figure 15: Flowchart representing the GetLowestEntropyCell function.

3.4. PROPAGATION STAGE

In the propagation stage, a stack of cells is created to keep track of all cells that have been changed due to the collapsing of the original cell. To start this stage of the algorithm, the cell that collapsed in the observation stage is added to the stack.

The algorithm then starts popping the cells from the stack. Each popped cell gets processed.

1. Check if the cell has already collapsed. If so, ignore this cell as a collapsed cell is not in a superposition anymore.
2. Get the neighbors of this cell and start looping over them.
3. For each neighbor:
    a. Compare all the possible tiles from both cells using adjacency constraints. In this implementation edge color detection is used to check whether tiles can fit together.
    b. Remove impossible states from the list of possible tiles in the neighbor.
    c. If at least 1 possible tile was removed from this neighbor, add it to the stack.
4. Check if the stack is not empty.
    a. If the stack is not empty, pop the next cell from the stack and restart this loop.
    b. If the stack is empty, terminate the function and restart the WFC loop.

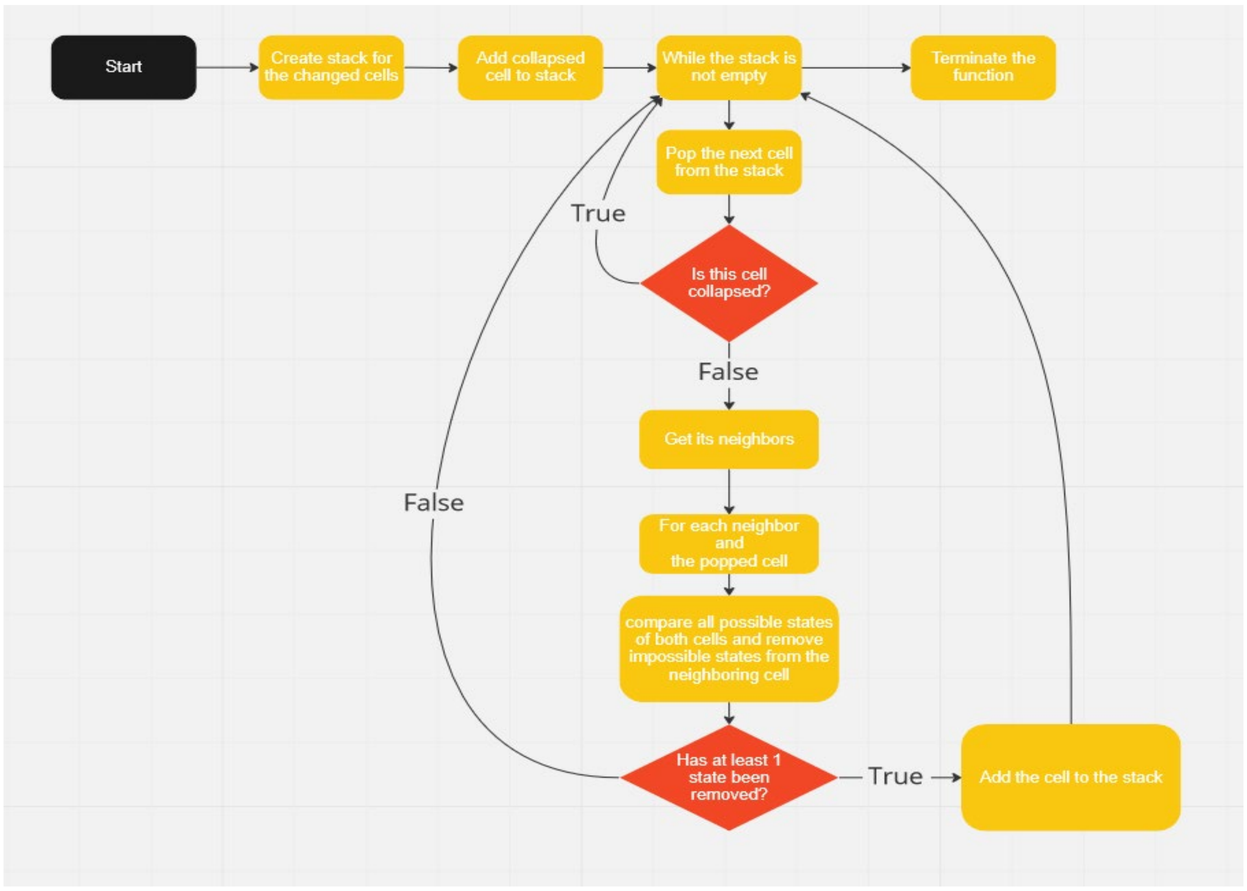

Start
Create stack for the changed cells
Add collapsed cell to stack
While the stack is not empty
Terminate the function
Pop the next cell from the stack
True
Is this cell collapsed?
False
Get its neighbors
False
For each neighbor and the popped cell
compare all possible states of both cells and remove impossible states from the neighboring cell
Has at least 1 state been removed?
True
Add the cell to the stack

Figure 16: Flowchart representing the propagation stage.

3.5. RESULT

With both the observation and propagation stages correctly implemented, a 2D map can now be generated. The image (Figure 17: WFC 2D result) below shows an example of 2 generated maps using Wang tiles and 2 generated maps using the road tiles seen before.

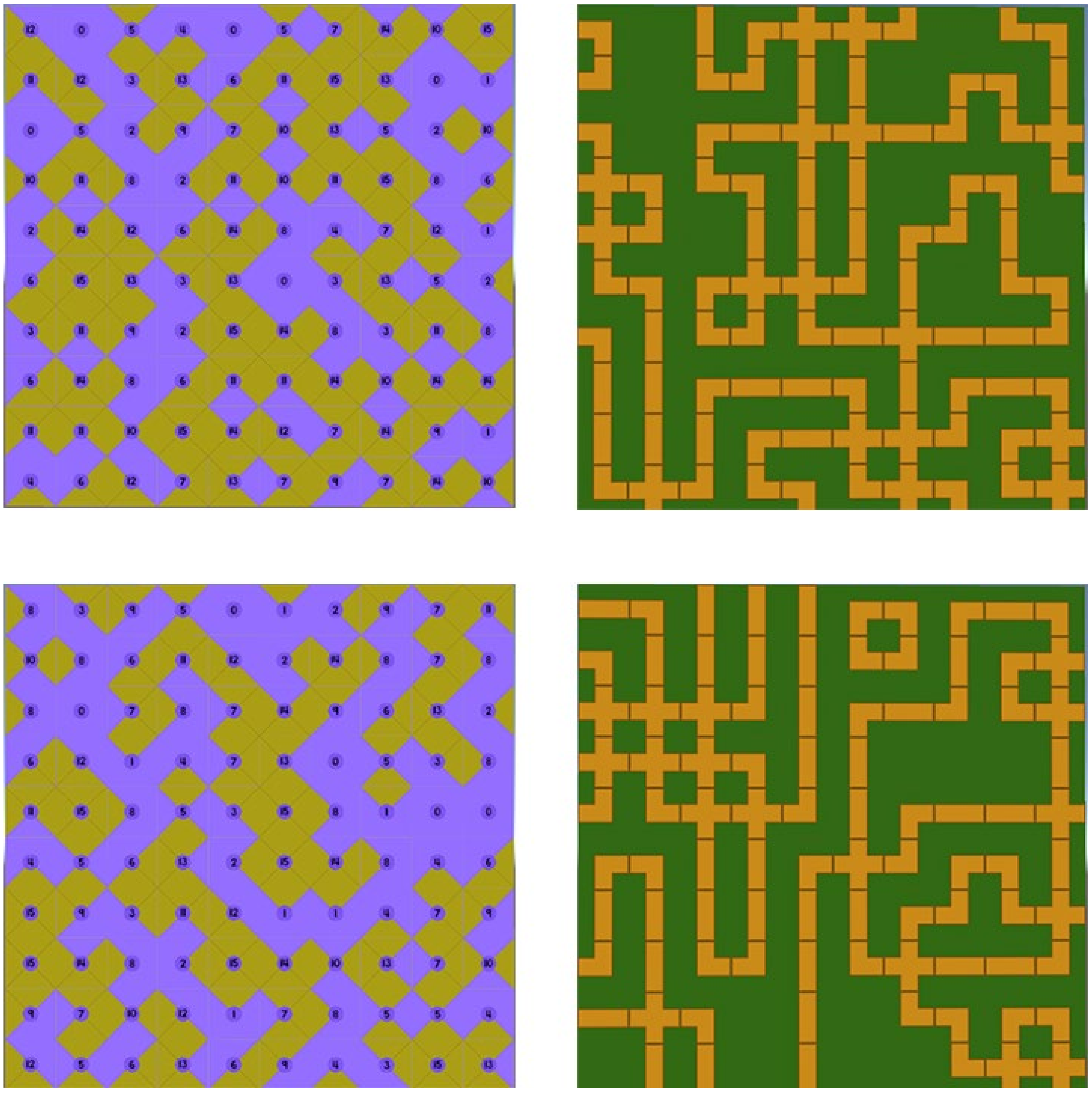


Figure 17: WFC 2D result

4. 3D WAVE FUNCTION COLLAPSE

Extending the Wave function collapse to the third dimension requires the algorithm to get extended as well. While minimal changes are required to the algorithm loop itself, the adjacency constraints pose the biggest challenge.

1. INPUT

For 3D WFC, 3D models are used as input. Wave function collapse in 3D has two major pros:

1. You can easily control the ambiance and art style of the generated level depending on the input tiles.
2. The tiles can have any shape possible and be used in the 3D algorithm with minimal change to the structure of the algorithm. Other examples include hexagonal tiles, and triangle tiles [26].

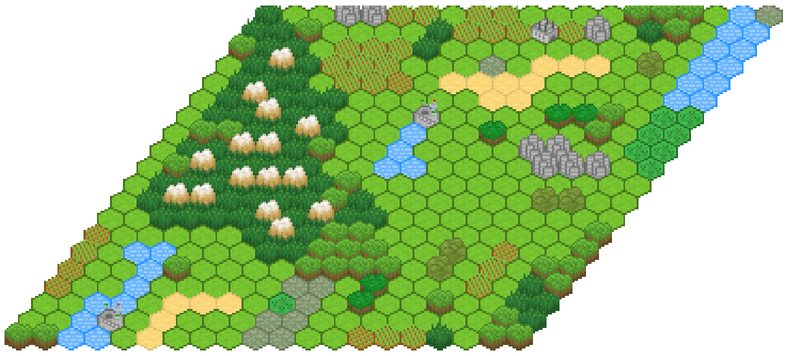


Figure 18: Example of a hexagonal grid

Most implementations of WFC in 3D use box-shaped modules as input, as these are the most basic and easy to implement. It is important to assure that the bounding box of all tiles is the same size. For this implementation, square tiles are used, with a bounding box of 2m x 2m x 2m. With these parameters, tiles can be created. These tiles should all be saved in 1 prefab, but each tile should be a child object of the main prefab (Figure 19: Used tile set). There should also be an empty tile that represents air and a solid tile for obstructed cells. This helps prevent unwanted edge cases and artifacts [23].

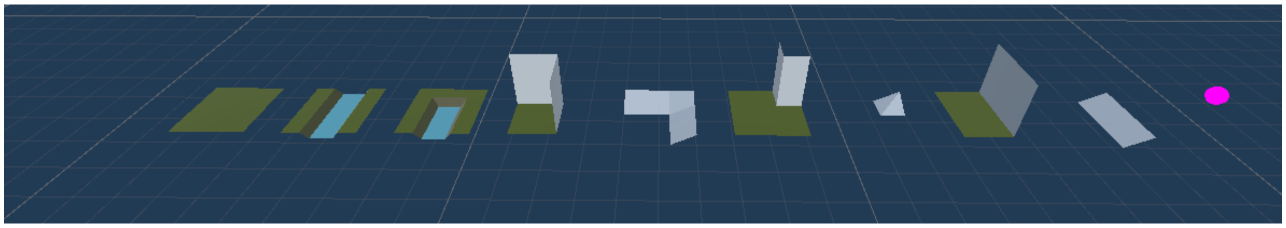


Figure 19: Used tile set prefab.

2. SETTING UP ADJACENCY CONSTRAINTS

While adjacency constraints were straightforward in 2D WFC, extending to 3D means completely changing the way they work. Instead of edge color detection, face edge alignment is the main adjacency constraint used. As the name suggests, connected faces on neighboring cells will have their edges compared, to see how they fit together. It proves very useful to set up an edge atlas while setting up the adjacency constraint (Figure 20: Example edge atlas).

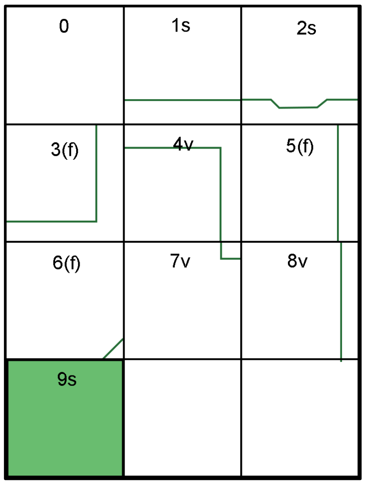


Figure 20: Example edge atlas

An easy way to represent the edge of a face is by adding a socket to each tile face (6 in total). A socket contains this information:

- An integer used as an identifier for the edge itself.
- For horizontal faces:
    - A boolean representing a symmetric face.
    - A boolean representing a flipped face.
- For vertical faces:
    - An integer representing the rotation index (0-3)
    - A boolean representing whether the face is rotationally invariant.

To assign the sockets on all the tiles, we follow this flowchart (Figure 21: Flowchart on how to assign):

1. Loop over all tiles.
2. For each face on the current tile.
    a. Check if this face already exists.
        i. If so, copy the data and move to the next face.
    b. Give it the next possible socket ID.
    c. Check whether it’s a vertical or horizontal face.
        i. Horizontal: Is the face symmetrical?
            1. Yes: Set symmetric boolean to true.
            2. No: Set flipped boolean to true if this face is the flipped version of socket Id.
        ii. Vertical: Does rotating this face change the edge alignment?
            1. Yes: Set invariant boolean to true.
            2. No: Set the correct rotation index.

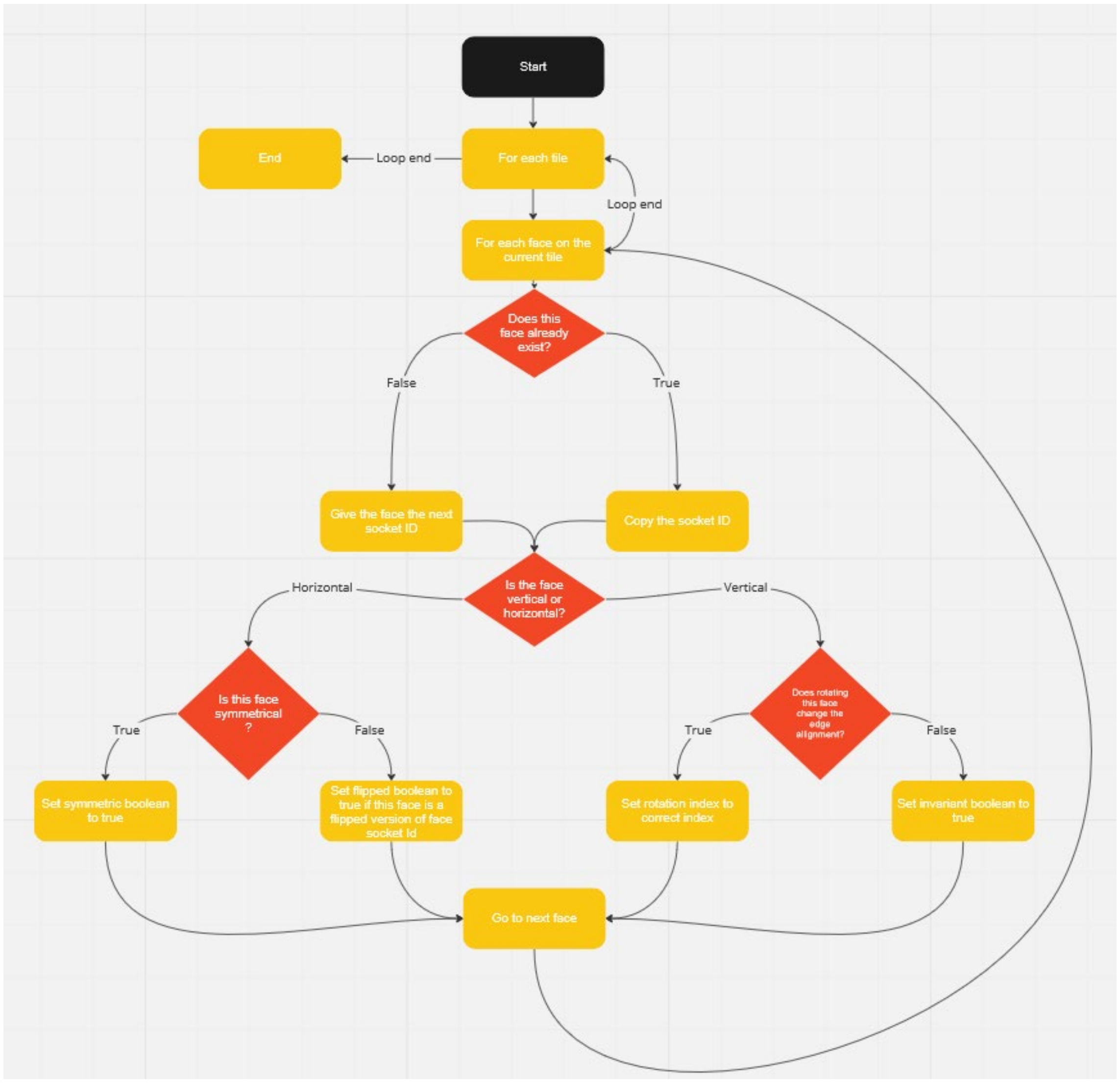

Start
End
Loop end
For each tile
Loop end
For each face on the current tile
Does this face already exist?
False
True
Give the face the next socket ID
Copy the socket ID
Is the face vertical or horizontal?
Horizontal
Vertical
Is this face symmetrical ?
True
False
Does rotating this face change the edge alignment?
True
False
Set symmetric boolean to true
Set flipped boolean to true if this face is a flipped version of face socket Id
Set rotation index to correct index
Set invariant boolean to true
Go to next face

Figure 21: Flowchart on how to assign sockets.

Let’s take the tile below as an example (Figure 22: Example tile with correctly assigned sockets.). To be clear, within Unity the red arrow represents the X-axis, the green arrow represents the Y-axis, and the blue arrow represents the Z-axis.

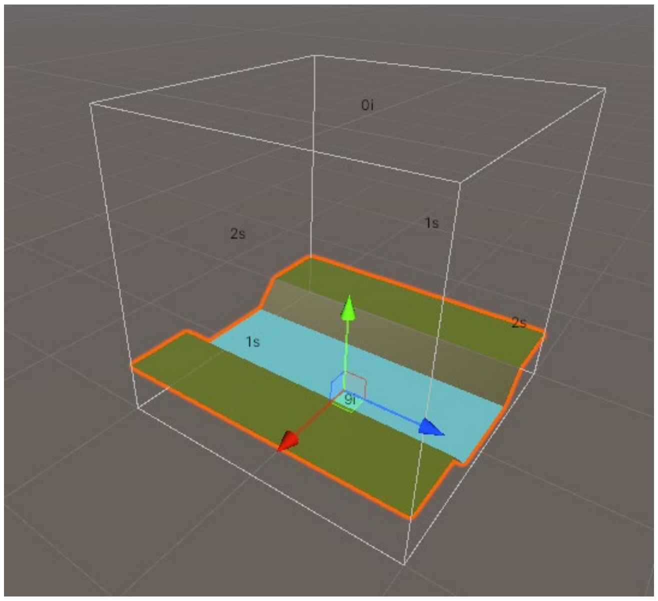


Figure 22: Example tile with correctly assigned sockets.

When assigning face sockets, we first need to look at the edge atlas (Figure 20: Example edge atlas) and look whether an identical edge is already in it, if it isn’t, we add it.

For the face on the positive X- axis, we see that the edge is a straight horizontal line on the lower part of the face. Within the edge atlas, the edge has been found with an id of “1” and since this edge face is symmetric, the “s” identifier is added. This process is identical to the negative X-axis.

To assign the positive Z-axis, the edge is again analyzed. The edge in this case is a horizontal line with a dent in it. When searching it up in the edge atlas, it is apparent that it is already in there, so the correct socket id is assigned, being 2. Since the edge is also symmetric, the “s” identifier is again added. Once again this is added for the negative Z-axis.

When assigning the positive Y-axis, we see that it has no edges on the face boundary. This means that it must be connected to either an air tile or a solid tile. Since the top of the mesh should be visible when standing next to it, the air socket id, which is 0, is assigned. For the negative Y-axis, the process is the same, except that the bottom of the tile should never be visible, meaning it must connect to a solid block and hence get the socket id of 9. Both Y faces are invariant as rotating them would not change any possible adjacencies.

This forms the basis adjacency constraint system for the 3D Wave Function algorithm. Many other adjacency constraints can be added on top of this basis constraint. Some examples will be given and used later.

3. HOW DO WE REPRESENT THE INPUT TILESET IN UNITY?

Before we can represent the entire tile set, a single tile needs to be represented. This is where modules come in. A module contains the name of the tile, an object holding all the tile data (explained in the next heading), and a reference to the tile prefab. There should be at least as many modules as used tiles when generating a 3D map.

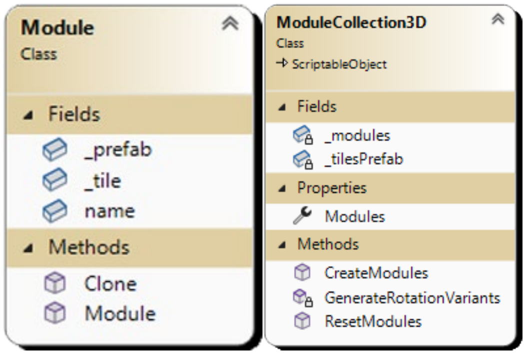


Figure 23: Module and module collection class diagrams

This implementation uses a scriptable object as the input for 3D WFC. This scriptable object is called Module collection 3D. The module collection contains a reference to a prefab containing all the tiles in separate objects and a list containing all the models it generates. Its most important methods are the `CreateModules()` and `GenerateRotationVariants()` methods.

The generate modules function transforms the tile prefab into a list of modules. First, it starts by looping over all the child transforms in the prefab and discarding the inactive transforms. This is an easy way to disable/enable prefabs from being used in map generation. Secondly, both vertical faces are checked. If they are both rotationally invariant, a module with the name of the mesh + "_i" is added. This identifier string indicates that this prefab is completely invariant. If one or both face(s) are rotationally variant, four modules are generated. For this the generate rotation variants function is used. Here four modules are generated, one for each rotation, and the sockets are swapped according to the current rotation. The name for each of these modules is just the mesh name plus "_x", where x is the current rotation index. Finally, the generated module(s) are added to the list of modules. If the last child element has been processed, the function is finished, and the modules are ready to be used in the algorithm.

4. CONVERTING FROM 2D TO 3D

As mentioned before, minimal changes are needed to the overall algorithm to extend WFC to 3D. Most of the changes just involve adding the third dimension to objects (e.g., generated map and WFC cell).

Observation stage

While getting the lowest entropy cell and choosing which module to collapse a cell to remain the same, the instantiating of the chosen module differs slightly. Instead of instantiating a sprite, the prefab in the chosen module is instantiated as a child object to the current cell. This assures that the 3D model is spawned at the correct location. We then locally rotate the 3D model around the Y-axis by x * 90 degrees, where x is the rotation index of the negative Y-face, so the 3D model aligns with the neighbors [11].

Propagation stage

The entire main loop of the propagation stays the same. The only changes that are made are in the comparison of neighboring cells. The parts that do the color detection can be entirely discarded and replaced with the new adjacency constraint, which is making sure aligned sockets can fit.

For sockets to fit together, multiple criteria must be met:

- The socket IDs must be the same on both sockets.
- For horizontal sockets both sockets should be symmetrical, or one socket should be flipped and the other not.
- For vertical sockets both sockets should either be rotationally invariant or both sockets should have the same rotation index.

To serve as an example, let’s have a look at these two modules (Figure 24: Modules that fit together without rotating.). These can fit together horizontally. This means that a normal 2 socket is needed to fit it.

These modules (Figure 25: Modules that do not fit together when placed next to each other without rotation.) however cannot fit together as they have different socket IDs, 2 and 1.

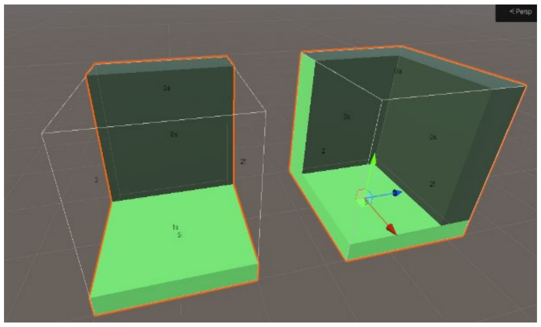


Figure 24: Modules that fit together without rotating.

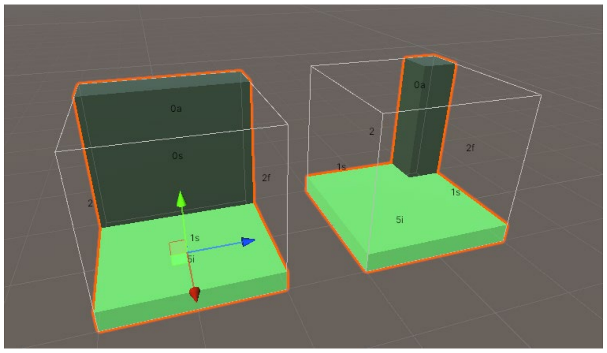


Figure 25: Modules that do not fit together when placed next to each other without rotation.

5. RESULT

Once all these changes have been implemented, the generation of a 3D map should now work. Running the generator with the simple tile (Figure 19: Used tile set) set gets the results seen in the image below.

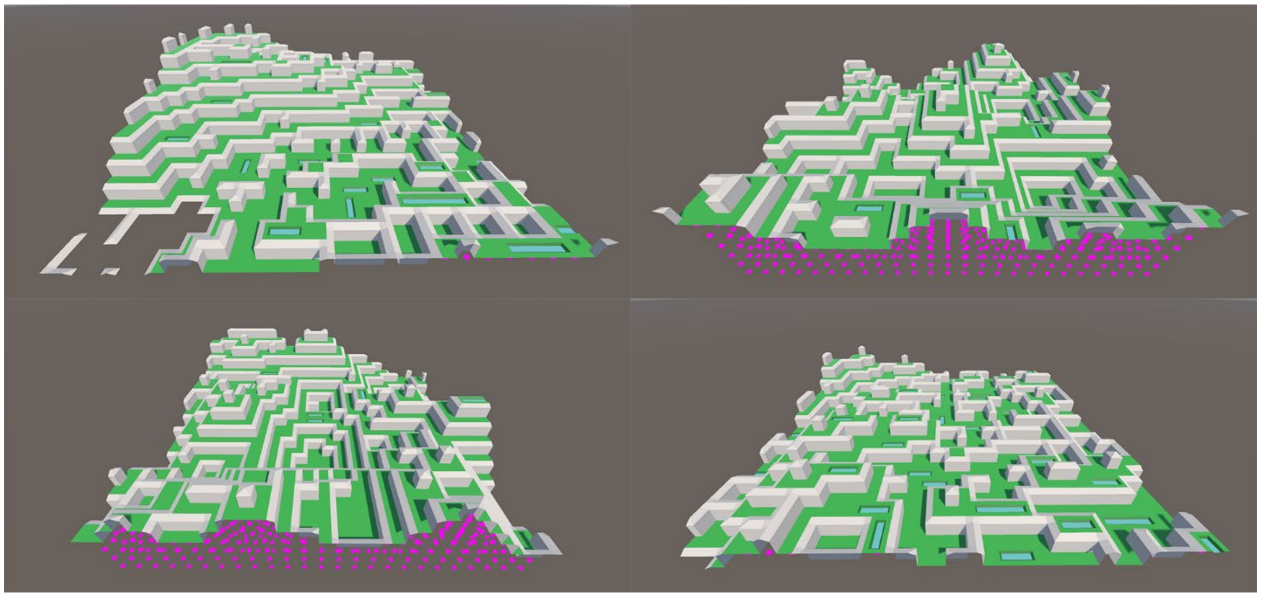


Figure 26: Example of 3D output without additional constraints

As you can see the 3D algorithm generates a nice-looking terrain. This basic implementation can generate many kinds of layouts basic on the simple set of 9 tiles (excluding the empty and solid tiles). If more customizability or different kinds of results would be wanted, extra tiles could be added, or additional adjacency constraints could be added.

5. WAYS TO FURTHER CONTROL THE RESULT

As mentioned before, the WFC algorithm relies on the given adjacency constraints to create a coherent and usable level. Therefore, rather than just creating extra tiles to customize the level generator; it is also advisable to add extra adjacency constraints. In this chapter, additional constraints will be added to transform the output from giving a more mountain-like result, to an island-looking result.

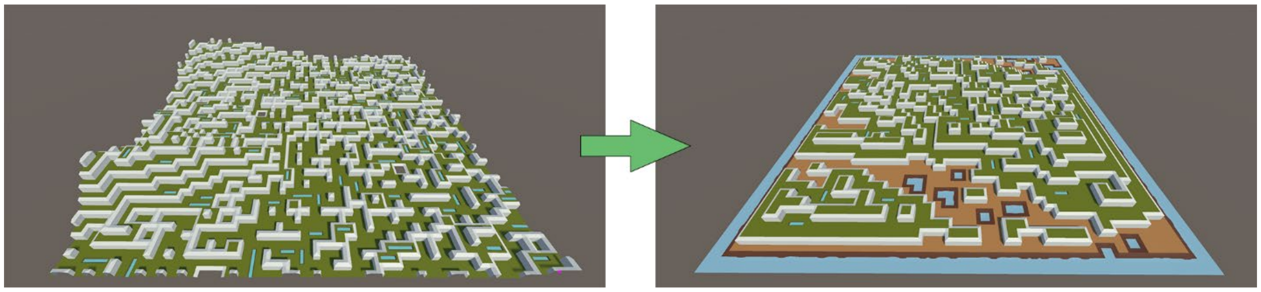


Figure 27: 3D WFC result before and after adding additional constraints.

5.1. SOLID FLOOR

One of the first constraints added is a solid floor, which was inspired by Kobe De Vrijsen [27]. For this, all cells with y index 0 are collapsed to the solid tile before the wave collapse is initiated. This assured that all cells with y index 1 have a solid bottom face. This prevents the possibility of holes in the result and gives the below result. Note: This outcome was also possible without this constraint, but now it is guaranteed instead of being based on chance.

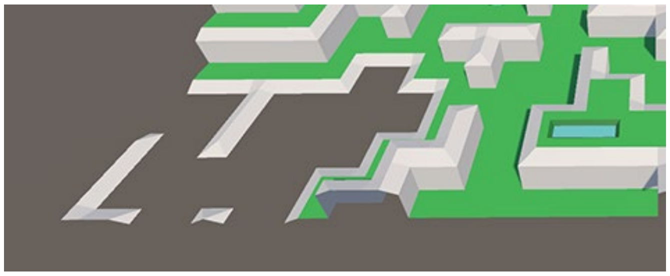


Figure 28: Example of a hole in a generated map

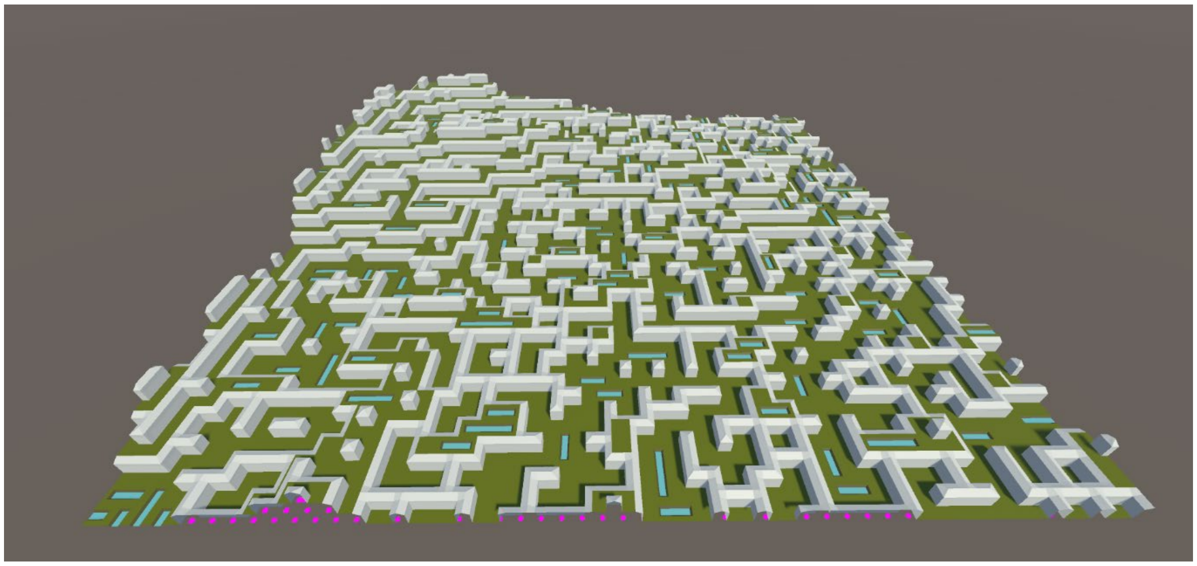


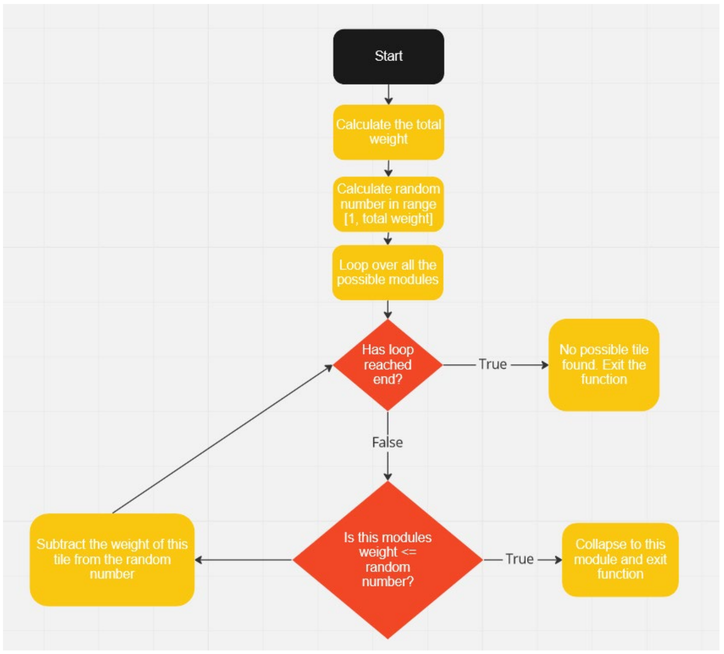
Figure 29: Result after using a solid floor.

5.2. WEIGHTED ENTROPY AND CHOICE

It often could prove useful to be able to provide how often we want certain tiles to appear compared to others. For example, imagine if a certain game required low meadows with a hint of verticality for its first level, and in contrast, it needed a level with a lot of verticality and big mountains. In this scenario, one approach could be to create two different tile sets, one for each level. According to Boris The Brave [22], This can be achieved by using weighted entropy. Sandhu et al. [16] add that while weighted entropy does achieve some control, it does not fully represent the weight of a tile. Instead, they recommend using weighted choice. This implementation will implement and use both.

Weighted choice can get added rather easily. Each tile gets an integer number to represent its weight. Then when the cell is getting collapsed, we take this weight into account, to determine its final state. Taking the weight into account means these steps [28]:

1. Calculate the total weight of the remaining tiles.
2. Get a random number in the range [1, total weight]
3. Loop over all the modules
    a. If the random number is smaller or equal to the module's weight
        i. Collapse the cell to this module and exit the function.
    b. Subtract the weight of this tile from the random number and continue the loop.


Start
Calculate the total weight
Calculate random number in range [1, total weight]
Loop over all the possible modules
Has loop reached end?
True
No possible tile found. Exit the function
False
Is this modules weight <= random number?
True
Collapse to this module and exit function
Subtract the weight of this tile from the random number

Once this has been implemented and all the tiles have been assigned their correct weight, the desired output can be generated.

Weighted entropy is also not that difficult to add. We change the way entropy is calculated by using this formula [22]:

$$Entropy = -\Sigma p_i log(p_i)$$

This formula loops over all the modules left in the current cell and multiplies the weight of the module with the natural logarithm of that weight. These are then all summed and inverted. These are very heavy calculations, so it is important to store this value, to not recalculate it each time the entropy is needed. The entropy then of course needs to be recalculated each time the remaining module collection is changed.

Note: finding the perfect balance between tile weights can be very challenging.

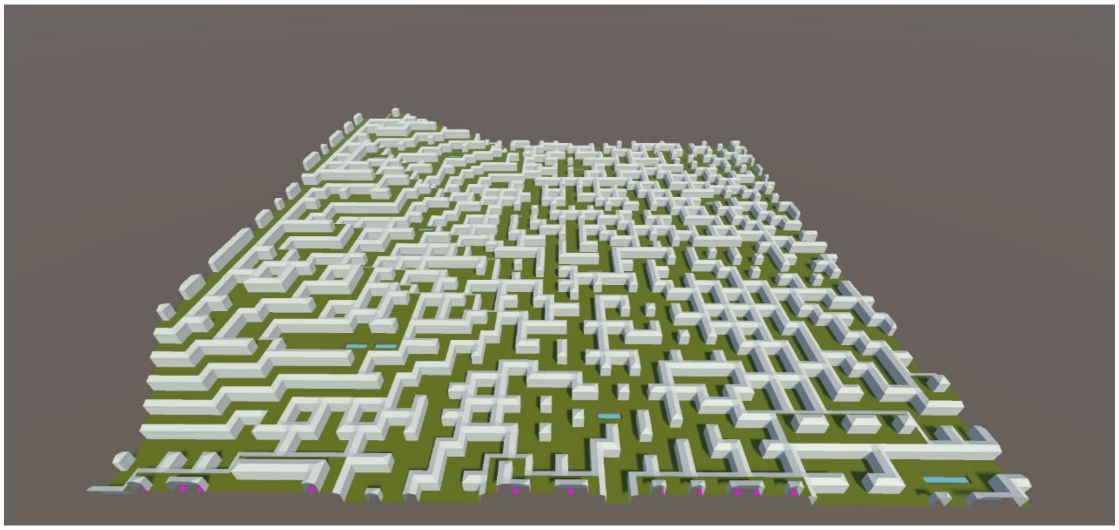


Figure 30: Example generated map with rock tiles with high weights

5.3. MATERIAL ADJACENCY

When generating a level, having multiple types of material can make the level more diverse and interesting. However, creating additional tiles or each material takes up a lot of time. Another approach would be to re-use tiles that are already available and just assign them a different material and have a way to tell the algorithm when certain materials can align. This forms the basis of material adjacency.

The image below showcases all the tiles used for material adjacency. There are grass tiles, sand tiles, and water tiles. The sand tiles are just recoloring of grass tiles and the water tiles are unique.

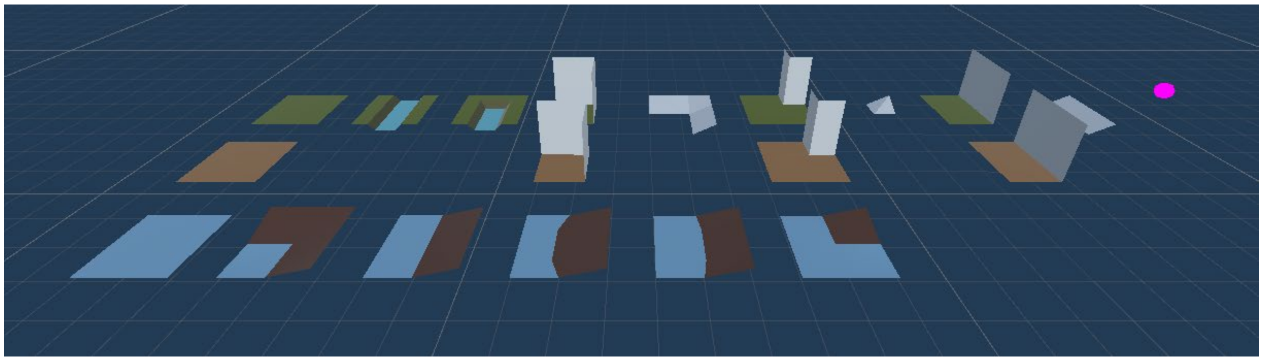


Figure 31: Tiles with the new materials

To start generating with material adjacency, changes must be made to our tile data and the propagation stage.

For the tile data, we first add a field representing the tile's material called compatible materials. This material represents the material seen on the bottom face, should multiple materials be present on one tile. Then, we also add a list of materials to each horizontal face. These are then filled with all the materials that can fit that face.

For the propagation stage, we just add a check when comparing the faces of the current cell with the neighbor's faces. If the neighbor's compatible materials contain the current cell face's material or vice versa, they are compatible.

With just these small changes, material adjacency is added, and an inherent level can be regenerated into a level with coherent diversity.

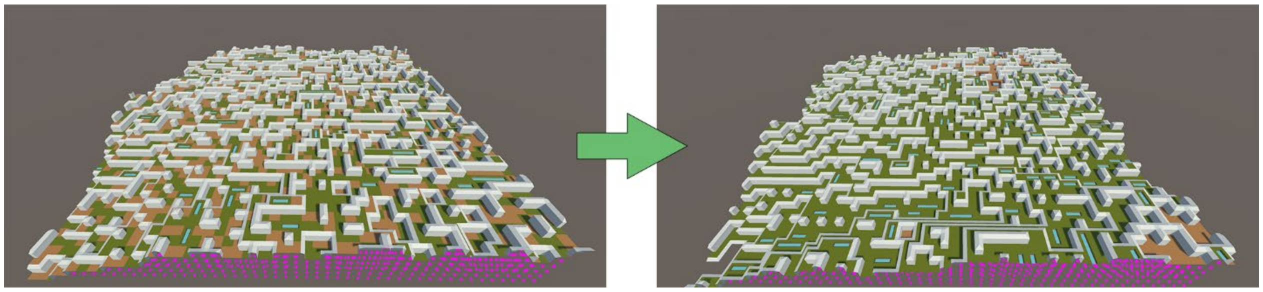


Figure 32: Before and after material adjacency

5.4. EXCLUDING NEIGHBORS

See the image below (Figure 33: Sample output of plain 3D WFC with notes). In the image, many walls stick together. Sometimes 4 outer corners are placed next to each other, creating a pillar structure. For the direction this implementation is headed, these are not wanted. In general, there are too many walls following each other vertically, so it would also be preferred if walls don't follow each other vertically if there are next to one another.

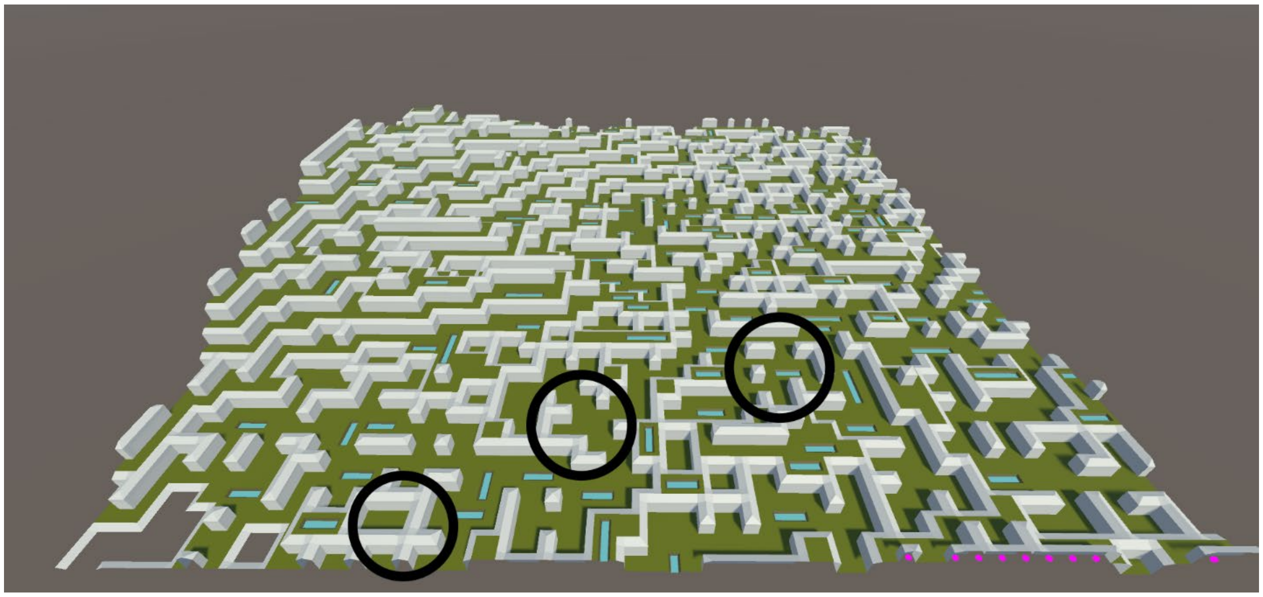


Figure 33: Sample output of plain 3D WFC with notes

These goals can be achieved by adding excluded neighbors adjacency [29]. This is done by adding a list of excluded neighbors to each face of a tile. Then, in the propagation stage, we just check whether one face contains the other in its excluded list. If it does, we ignore this tile and continue to the next face. Afterward, it's only a matter of assigning excluded neighbors to the tiles to get a result that is preferred. The transformation of the map generation from this adjacency constraint can be seen below. (Figure 34: Before and after excluded)

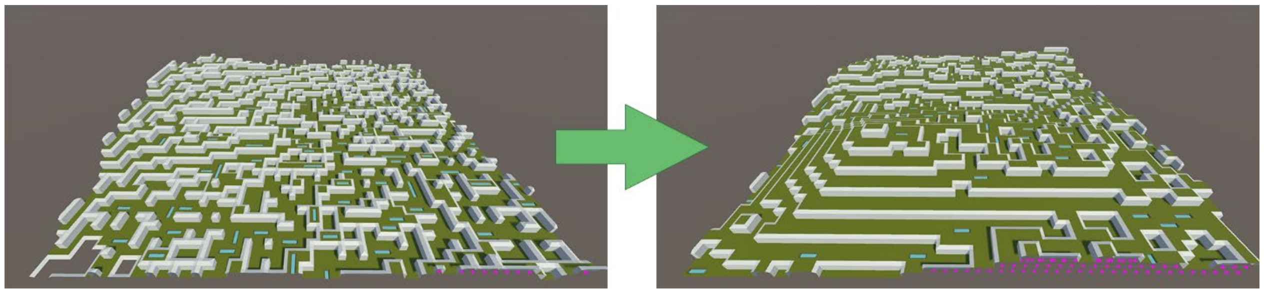


Figure 34: Before and after excluded neighbors.

EXPERIMENTS & RESULTS

In this paper, the impact of using additional adjacency constraints on the execution time is analyzed. We measured the execution time in milliseconds. For all experiments, the execution time is defined as the initialization of the wave, the execution of the wave, and any cleanup of algorithm objects. To avoid any hardware interference or dependencies, all the tests were run on the same computer, while it was only running said tests. No other programs were being used during tests.

Each of the constraints was put in a different test case for a total of six cases. Only a maximum of 1 extra constraint was used per case, except for case six.

1. No extra constraints are used.
2. Assuring solid floor.
3. Enabled material adjacency.
4. Enabled weighted choice and entropy.
5. Enabled excluded neighbors.
6. Enabled all extra constraints.

Each test case was run twelve times. The minimal and maximal measured time from each case was discarded. This entire experiment was run for three different map sizes: 5.000 cells, 25.000 cells, and 45.500 cells. Running these three experiments gave the results discussed in the next paragraphs of this chapter.

THE IMPACT OF ALL EXTRA CONSTRAINTS

First, it is useful to compare the execution time of 3D Wave Function Collapse without extra adjacency constraints and with extra adjacency constraints. This indicates the overall added load of adjacency constraints on the algorithm. Running these tests gives us the following graph:

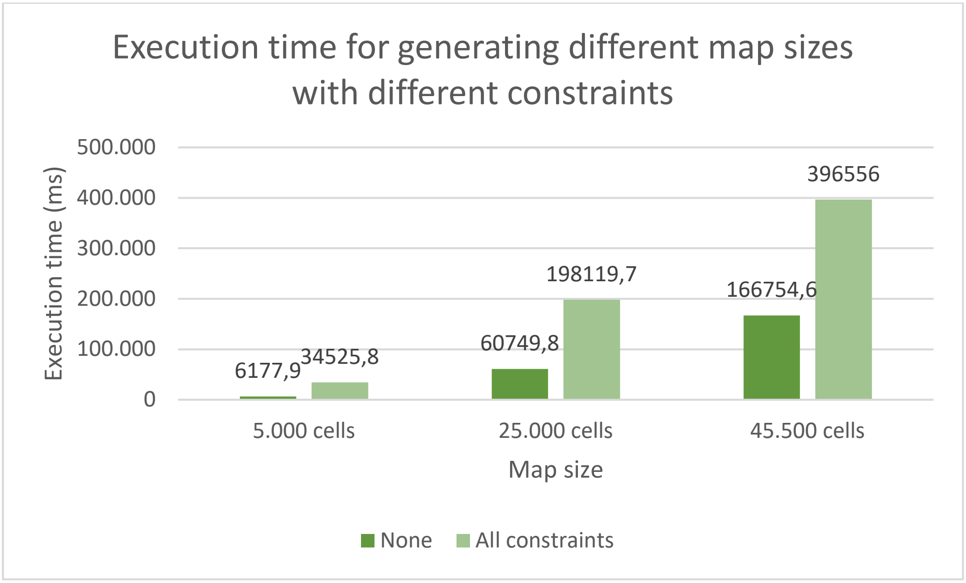


Figure 35: Clustered column graph comparing execution time for different map sizes with and without all the extra constraints.

From this graph, it is obvious that a load of extra adjacency constraints cannot at all be neglected when implementing them. For a map of 5.000 cells, the execution time increases by 471%. While there is still a big increase for a map of

45.500 cells, it is substantially less. When generating a large map, the increase is (only) 160%. For this reason, this increase was also tested for other map sizes. When running the algorithm for maps of other sizes, we get this chart:

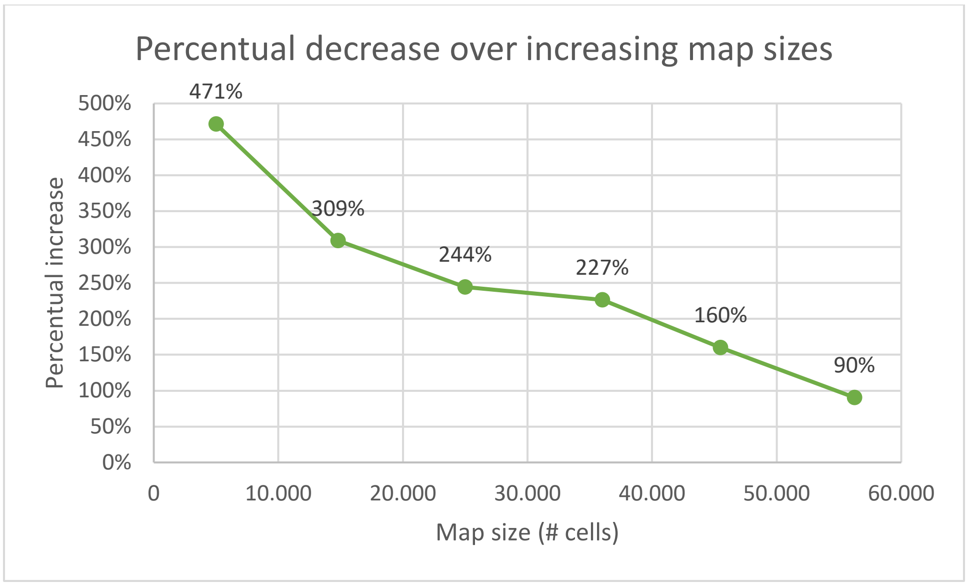


Figure 36: Scatter chart representing the decrease in percentual execution time increase over larger map sizes.

As you can see, there is a downward trend for the percentual increase regarding increasing map sizes. This indicates that the relative load added from adjacency constraints falls off over larger maps. This opens the possibility to look at adjacency constraints as a way of optimizing 3D Wave Function Collapse. It may also be possible that when implementing additional constraints, it is imperative to have a target map size in mind.

THE IMPACT OF INDIVIDUAL CONSTRAINTS

The impact of each individual adjacency constraint will also individually be tested. As seen before they add a substantial load to the algorithm, so it is very useful to evaluate each constraint separately.

1. SOLID FLOOR

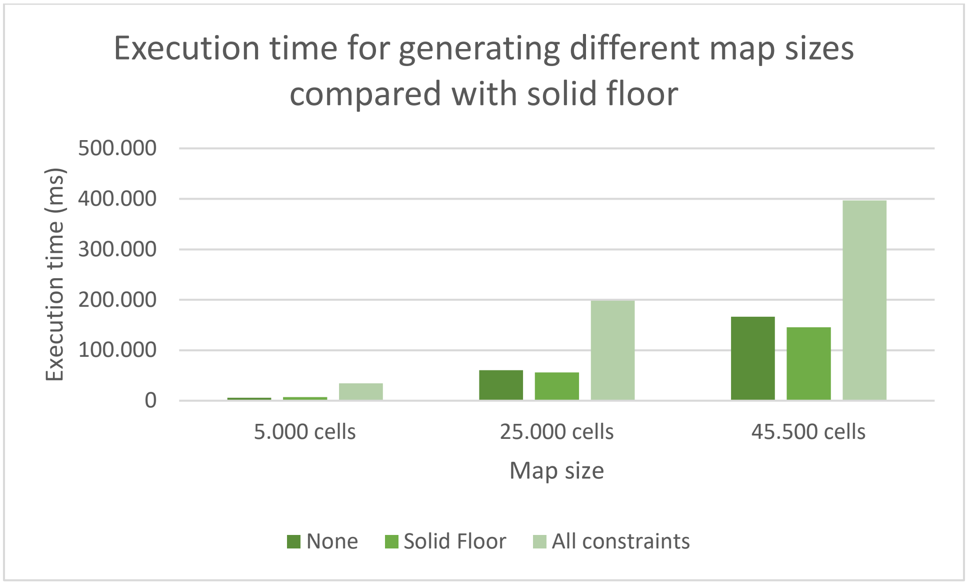


Figure 37: Clustered column graph comparing execution time for different map sizes with and without the solid floor constraint enabled.

Testing the solid floor constraint gives rather interesting results. While it increases the execution time for a map of 5.000 cells, it speeds up the map generation for larger maps. These results support the possibility of using adjacency constraints as a way of optimizing generation time.

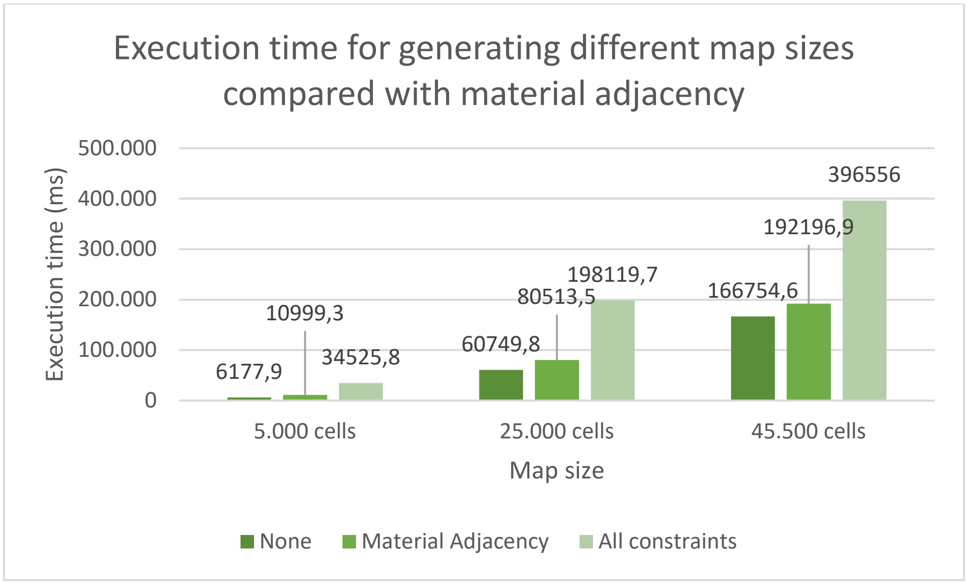


Figure 38: Clustered column graph comparing execution time for different map sizes with and without material adjacency enabled.

When using material adjacency, execution is increased. This was to be suspected as material adjacency adds two condition checks for every cell face comparison. The percentage at which execution time is increased also seems to decrease as map sizes increases. For maps of 5.000 cells, 25.000 cells, and 45.500 cells, there are percentual increases in execution times of 78%, 33%, and 15% respectively. This also indicates that material adjacency becomes more efficient as larger maps are generated. This could be because possible modules get discarded earlier than they would without material adjacency.

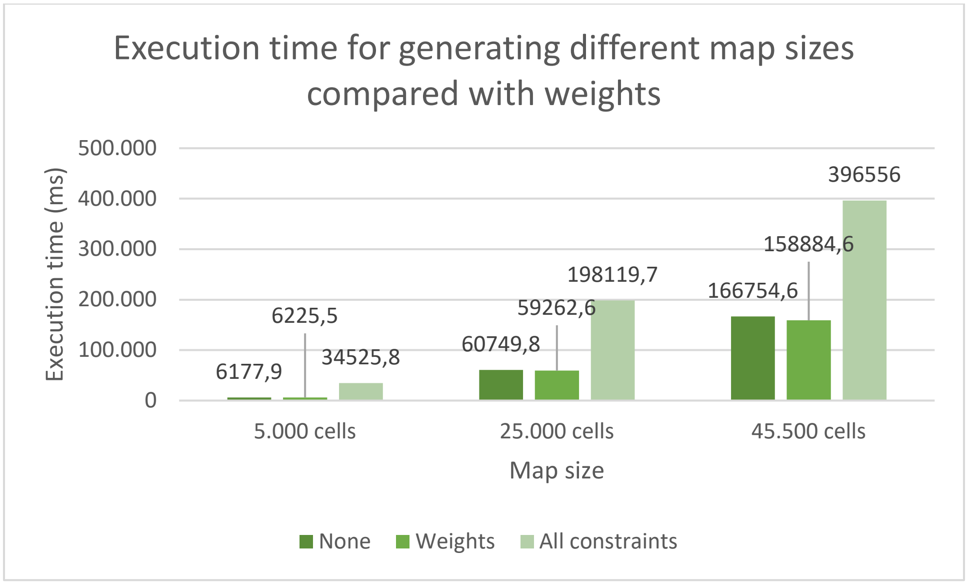


Figure 39: Clustered column graph comparing execution time for different map sizes with and without weighted entropy and choice enabled.

While weighted entropy and choice involve some heavy calculations, the impact on execution time seems almost negligible. It is only for a map of 5.000 cells that the execution time is increased, being by 0,77%. For a map of 25.000 cells and a map of 45.500 cells, there are decreases of 2,45% and 4,72% respectively. These decreases might be explained that since weighted choice and entropy promote some tiles over others, tiles that have small weights are discarded earlier on in the collapsing of the wave.

4. EXCLUDING NEIGHBORS

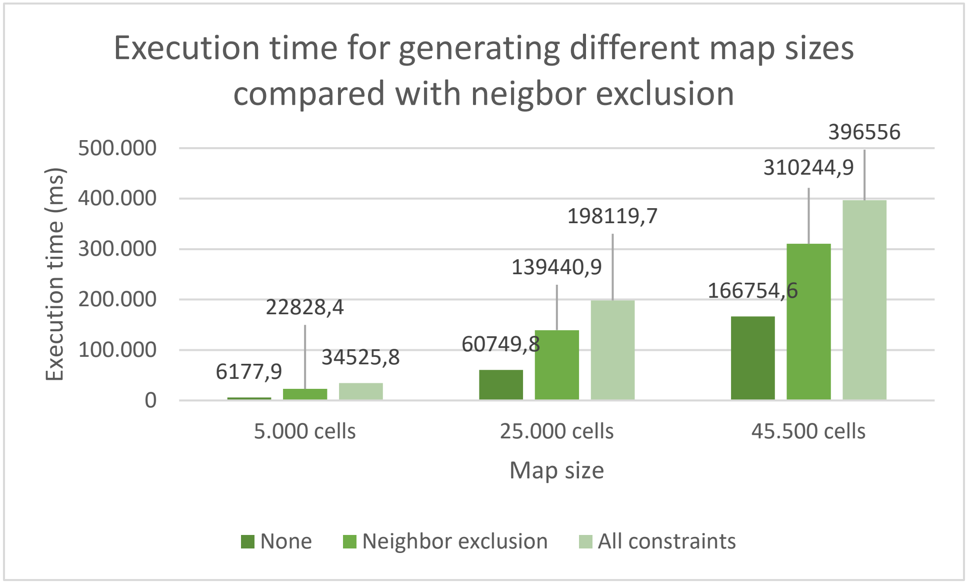


Figure 40: Clustered column graph comparing execution time for different map sizes with and without material adjacency enabled.

As the other constraints have little to no increase in execution time or even decrease, it is only fair to assume that excluding neighbors is the big contributor to the increase from no extra constraints to all extra constraints. This statement is most definitely supported by the tests. For the test of a 5.000 cell map, there is an increase of 270%, which makes it by large the biggest bottleneck on performance. This bottleneck decreases for larger maps, where it becomes an increase of 130% and 86% for maps of 25.000 and 45.500 cells respectively.

TIME COMPLEXITY

To measure the time complexity, the algorithm is run 5 times for a consistent map size. In this case the map size for each case was 45.000 cells (or 70 x 10 x 65 cells). The 5 different cases were:

1. 27 input modules.
2. 50 input modules.
3. 75 input modules.
4. 101 input modules.
5. 123 input modules.

Each case was run twelve times, where the minimum and maximum values were discarded for correctness. From the ten remaining values the average was calculated and then graphed. This resulted in the graph below (Figure 41: Time complexity for a map of 45.500 cells.).

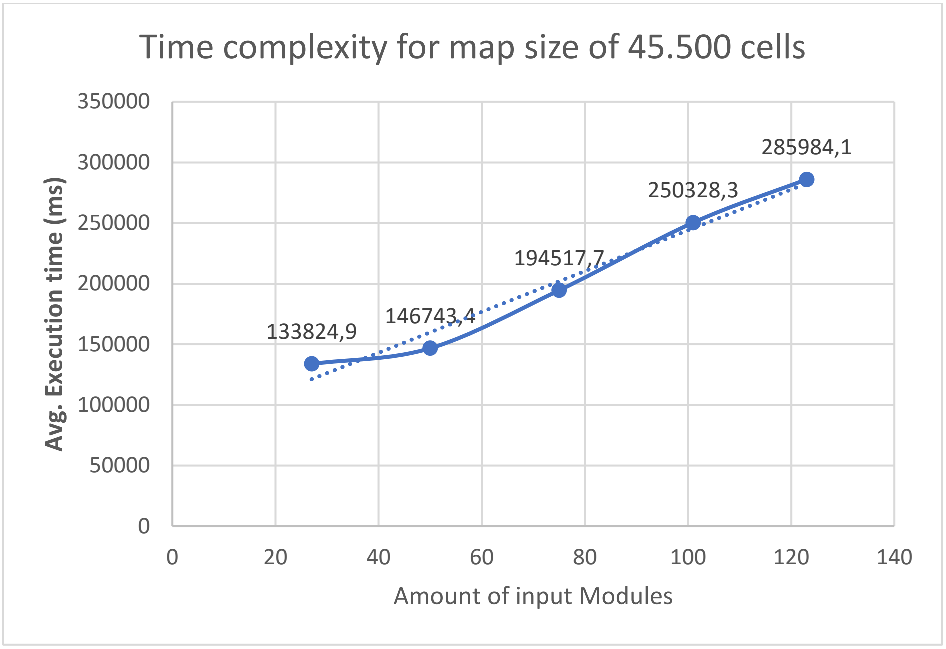


Figure 41: Time complexity for a map of 45.500 cells.

From this data it can be derived that the Wave Function Collapse algorithm probably has an average case time complexity of nlog(n). Do note that this data was empirically gathered and if the algorithm was properly processed, the concluded time complexity could differ.

This time complexity once again reinforces the need for optimizations. This algorithm get much heavier when adding additional modules, adjacency constraints or increasing desired map size.

DISCUSSION

During this research, 2D Wave function collapse was implemented and extended to 3D Wave function collapse. Implementing the 2D algorithm was rather straightforward. However, as Chloe Sun mentioned [9], it is very useful to use Wang edge tiles as a tile set when implementing the algorithm. The algorithm itself can be properly implemented if understood, for this, it was very useful to first schedule the algorithm in a flowchart and then translate it into code.

This 2D implementation focused on edge color detection as the main adjacency constraint. It is however equally possible to add other types of constraints, to further customize it. This fell out of scope for this paper but is worth mentioning, nevertheless. Cheng D. et al [30] and Sandhu et al. [16] gave some great examples of the possibility of 2D constraints.

Extending the 2D algorithm to 3D generally was less complicated than expected. The biggest challenge came in finding a correct way to represent tiles and their adjacency constraints in 3D. The solution for this was based on the implementations of Marian42 [23] and Martin Donald [11]. They both provided a solid system for representing tiles and using edge alignment as the main adjacency constraint. There however is one aspect that is different in this implementation, the list of possible modules for a cell. While in their implementations each face has its list of possible states for the cell, in this implementation there is only one list for the entire cell. There do not seem to be any repercussions from doing this.

When generating the list of modules, it is rather easy to generate rotational variants (if needed), as Martin D. mentioned [11]. This eliminates some manual labor of having to export each rotated variant of a tile mesh and assign all the adjacency constraint info.

While implementing 3D WFC, one big mistake was made. At first, a simple tile set, which did not support verticality well, was used. This was a crucial mistake that resulted in wasted time and unnecessary debugging. If the tile set is not expansive enough, the algorithm will often fail.

AbeTusk [31] mentioned that one of the best things one can do when optimizing 3D WFC is to look for possible optimizations. This is because the algorithm is very heavy and takes a long time to collapse a wave. This statement is fully supported by this implementation. While the execution time for plain 3D WFC and small maps is (relatively) low, adding additional constraints and increasing map size quickly increases.

As seen by Sandhu et al. [16] and Boris the Brave, one of the big strengths of WFC comes in the form of adjacency constraints. The four constraints used in this implementation are each a good example of how even just two or three extra adjacency constraints can completely change a result and give more control to the user. It can become very chaotic, however, by combining these constraints and coding them. This is another takeaway from this paper; When implementing additional adjacency constraints, structure them well in code and, if possible, try to set up a modular system for adding and removing adjacency constraints. It is also important to keep performance in mind. As seen with the excluded neighbor constraint, one single constraint can triple the execution time. Added customizability also comes with additional time necessities, best example by weighted entropy and weighted choice. If the weights are not balanced well, the level can become very bland or repetitive.

As mentioned before, Boris the Brave [22] uses weighted entropy to represent tile weights, while Sandhu et al. [16] advocate using weighted choice. This implementation found that using a combination of both gives the most control over the result. This might take away some of the charms of having the levels be generated more randomly, but if desired can be a powerful tool.

This paper gives clear instructions on how to implement 2D and 3D Wave Function Collapse, opening the possibility for game programmers to create a scalable tool, that could be used across different projects without having to be adapted (much). Given that some adjacency constraints seem to have decreased performance time, even for larger maps, the idea of using adjacency constraints to optimize WFC is possible. It does remain apparent that every possible optimization goes a long way for this algorithm as it is very heavy and, in most cases, unrealistic to use at runtime.

CONCLUSION & FUTURE WORK

CONCLUSIONS

When talking about procedural content generation in video games, there are many options available. Wave Function Collapse offers a way to procedurally generate 2D and 3D levels based on input tiles. The biggest challenge in converting 2D Wave Function Collapse to 3D is representing the input and adjacency constraints in a clear and manageable system. This can be done by using modules, that contains all necessary data for the adjacency constraint system. Ideally, these modules are generated from a prefab containing all the separate tiles. This collection of modules can then generate rotational variants if desired, eliminating the manual labor of having to create and assign each rotational variant by hand. Other than this, the changes to the observation and propagation are minimal.

Additional adjacency constraints can be very powerful tools to tailor generated maps to desired results but can come at a heavy price. They can increase the execution time of the algorithm by very substantial amounts and can increase the failure rate as well. They can also make code very chaotic and difficult to manage, if not structured nicely. Balancing (different) constraints can also prove to be a challenge and, if done incorrectly, can make generated maps bland or repetitive.

FUTURE WORK

Adjacency constraints are not equally slowing down performance and some can even improve it. This indicates that there is a possibility to use additional adjacency constraints as ways to improve execution time. If this is indeed possible, Wave Function Collapse could become more mainstream and often used.

3D WFC could benefit from an automatic way to assign sockets for modules. The process of doing this by hand is very tedious and prone to error. Errors due to wrong constraint assignments are very annoying and difficult to find. For these reasons finding a way to have an algorithm do this, would go a long way toward maintaining larger tile sets.

A big part of creating the desired result with 3D WFC is iterating different adjacency constraint settings and combinations. For this, a modular system to add, handle or change adjacency constraints could be very powerful in keeping one implementation untouched for multiple projects and creating a kind of sandbox to experiment with. This could also help lessen the clutter of having all the different constraints coded directly into the algorithm.

There is always the possibility of contradictions with WFC. As generated map becomes larger, the chance of having the algorithm fail increases. For this, it could be very useful to have an efficient way of handling these. For most cases, it could be sufficient to just try to regenerate a new map. Finding a way to handle contradictions, however, could open many new use cases of WFC in gaming and procedural content generation.

Finally, as mentioned before, as many optimizations must be found and used as possible. Execution time is very slow for WFC, even compared to other map generation algorithms. If it is possible to lessen execution time significantly, the possibility of using WFC to create an endless world at runtime opens (as demonstrated by Marian42 [29]). For this to work, however, there also needs to be a consistent and efficient way to handle contradictions.

BIBLIOGRAPHY

[1] O. Veneri, C. S. Gros and C. S. Natkin, *Procedural Audio for Game using GAF,* 2008.

[2] P. Müller, G. Zeng, P. Wonka and L. V. Gool, *Image-based procedural modeling of facades,* vol. 26, ACM PUB27 New York, NY, USA , 2007.

[3] E. S. D. Lima, B. Feijó and A. L. Furtado, *Procedural Generation of Quests for Games Using Genetic Algorithms and Automated Planning,* 2019.

[4] A. Amato, *Procedural content generation in the game industry,* Springer International Publishing, 2017, pp. 15-50.

[5] G. Smith, *The Future of Procedural Content Generation in Games,* 2014.

[6] GameInformer, *A Behind-The-Scenes Tour Of No Man's Sky's Technology - YouTube,* 2014.

[7] G. Coleman, *Roguelike Definition & History Explained, from Rogue to Hades,* 2020.

[8] J. Freiknecht and W. Effelsberg, *A Survey on the Procedural Generation of Virtual Worlds,* vol. 1, Multidisciplinary Digital Publishing Institute, 2017.

[9] C. Sun, *Implementation of Wave Function Collapse Algorithm in Houdini for 3D Content Generation,* 2020.

[10] R. Heaton, *The Wavefunction Collapse Algorithm explained very clearly,* 2018.

[11] D. Martin, *Superpositions, Sudoku, the Wave Function Collapse algorithm. - YouTube,* 2020.

[12] S. A. Curtis, *The classification of greedy algorithms,* vol. 49, 2003, pp. 125-157.

[13] J. B. Rasmus, J. A. Lennar, R. L.-L. Marcus, P. Johnsson and T. Svensson, *An Exploration of Procedural Content Generation for Top-Down Level Design,* 2018.

[14] R. Stangl, *Procedural Content Generation: Techniques and Applications,* 2017.

[15] G. Smith, *Procedural Content Generation An Overview,* 2015, pp. 501-518.

[16] A. Sandhu, Z. Chen and J. McCoy, *Enhancing wave function collapse with design-level constraints,* Association for Computing Machinery, 2019.

[17] M. Gumin, *mxgmn/WaveFunctionCollapse: Bitmap & tilemap generation from a single example with the help of ideas from quantum mechanics,* 2016.

[18] O. Stålberg, *Wave - by Oskar Stålberg.*

[19] BorrisTheBrave, *Tessera Documentation.*

[20] BorrisTheBrave, *DeBroglie Documentation.*

[21] BorisTheBrave, *Wave Function Collapse tips and tricks – BorisTheBrave.Com.*

[22] BorisTheBrave, *Wave Function Collapse Explained – BorisTheBrave.Com,* 2020.

[23] Marian42, *Infinite procedurally generated city with the Wave Function Collapse algorithm | Marian's Blog,* 2019.

[24] T. van Loenhout, *Wang Tiles and Cubes: Aperiodic Sets,* 2021.

[25] E. Jeandel and M. Rao, *An aperiodic set of 11 Wang tiles,* 2021, pp. 1-3.

[26] O. Stålberg and S. G. Arena, *SGC21- Oskar Stålberg - Beyond Townscapers - YouTube,* 2021.

[27] K. D. Vrijsen, *How can we get better and more varied scenes using different types of tile shapes using wave function collapse,* 2021.

[28] J. Castello, *How to Generate Weighted Random Numbers,* 2016.

[29] Marian42, *marian42/wavefunctioncollapse: Walk through an infinite, procedurally generated city,* 2019.

[30] D. Cheng, H. Han and G. Fei, *Automatic Generation of Game Levels Based on Controllable Wave Function Collapse Algorithm,* vol. 12523 LNCS, Springer Science and Business Media Deutschland GmbH, 2020, pp. 37-50.

[31] Abetusk, *Lessons Learned from Implementing "Wave Function Collapse" — fxhash,* 2022.

[32] F. Rossi, P. V. Beek and T. Walsh, “Constraint Programming,” 2006.

[33] L. Westfalen, *Procedural Generation of Buildings with Wave Function Collapse and Marching Cubes,* 2022.

[34] O. Stålberg and Konsoll, *Konsoll 2021: Oskar Stålberg - The Story of Townscaper - YouTube,* 2022.

[35] O. Stålberg and B. games, *EPC2018 - Oskar Stalberg - Wave Function Collapse in Bad North - YouTube,* 2018.

[36] H. Scurti and C. Verbrugge, *Generating Paths with WFC,* 2018.

[37] OddMax, *GitHub - oddmax/unity-wave-function-collapse-3d: Implementation of wave function collapse approach for Unity in 3d space,* 2022.

[38] Q. E. Morris, *Modifying Wave Function Collapse for more Complex Use in Modifying Wave Function Collapse for more Complex Use in Game Generation and Design Game Generation and Design,* 2021.

[39] T. N. Møller, J. Billeskov and G. Palamas, *Expanding Wave Function Collapse with Growing Grids for Procedural Map Generation,* Association for Computing Machinery, 2020.

[40] H. Kim, T. Hahn, S. Kim and S. Kang, *Graph based wave function collapse algorithm for procedural content generation in games,* vol. E103D, Institute of Electronics, Information and Communication, Engineers, IEICE, 2020, pp. 1901-1910.

[41] H. Kim, S. Lee, H. Lee, T. Hahn and S. Kang, *Automatic generation of game content using a graph-based wave function collapse algorithm,* Vols. 2019-August, IEEE Computer Society, 2019.

[42] T. Kesler, *Procedural Generation with Prefabs: Wave Function Collapse | Medium,* 2022.

[43] I. Karth and A. M. Smith, *Wave Function Collapse is constraint solving in the wild,* vol. Part F130151, Association for Computing Machinery, 2017.

[44] I. Karth and A. M. Smith, *Addressing the fundamental tension of PCGML with discriminative learning,* Association for Computing Machinery, 2019.

[45] A. Karth, I. Smith, A. Marshall, I. Karth and A. M. Smith, *WaveFunctionCollapse: Content Generation via Constraint Solving and Machine Learning,* Institute of Electrical and Electronics Engineers Inc., 2021.

[46] B. Horn, S. Dahlskog, N. Shaker, G. Smith and J. Togelius, *A comparative evaluation of procedural level generators in the Mario AI framework,* 2014.

[47] W. Hamilton, *Procedural Generation of Three-Dimensional Game Levels with Interior Architecture,* 2019.

[48] Brian, *Wave Function Collapse — UpRoom Games,* 2021.

TABLE OF FIGURES

APPENDICES