Hierarchical Semantic Wave Function Collapse

Master's Thesis

Shaad Alaka

Hierarchical Semantic Wave Function Collapse

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Shaad Alaka
born in Schiedam, the Netherlands

TU Delft

Computer Graphics and Visualization Group
Department of Intelligent Systems
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Hierarchical Semantic Wave Function Collapse

Author: Shaad Alaka
Student id: 4287851

Abstract

Virtual worlds are rapidly increasing in size, enabled by advancements in computing technology, which puts a large burden on designers creating them. Procedural Content Generation can help alleviate this burden, though lacks precise control, to the detriment of designer intent. Some PCG algorithms, such as Wave Function Collapse (WFC) - known for generating tile-based content adhering to certain constraints, are able to induce this control through mixed-initiative editing opportunities, combining the efforts of humans and machines while capturing designer intent. However, stock WFC operates on a flat set of tiles with all their semantics blurred together, which unnecessarily strains designers with additional cognitive load when working with such detailed tiles. We therefore propose Hierarchical Semantic Wave Function Collapse (HSWFC), a generalized approach to WFC that augments the tileset with a new type of tile, the meta-tile, which represents semantic traits, and then organizes the tileset into a hierarchy akin to a taxonomy induced by such semantic representations: the meta-tree. A cell once collapsed to any tile high up in the meta-tree (such as ”forest” or ”village”), can further collapse to concrete tiles at the bottom (such as “tree” or “wall”). We investigate how this extension in data organization affects the original algorithm, and explore several novel editing facilities, including e.g. sketching with semantic tiles, controlling tile distributions, regenerating areas that represent specific semantics, and more. A prototypical HSWFC-driven tile-editor was developed and evaluated through a user study, confirming that such an editor indeed reduces cognitive load compared to its stock WFC counterpart, and that the newly enabled features are highly valued by environment designers.

Thesis Committee:

Chair: Prof. Dr. E. Eisemann, Faculty EEMCS, TU Delft
University supervisor: Dr. ir. R. Bidarra, Faculty EEMCS, TU Delft
Committee Member: Dr. A. Katsifodimos, Faculty EEMCS, TU Delft

Preface

Dear reader, before you lies the culmination of my research efforts, driven by the urge to push the frontiers of virtual world design, further narrowing the barrier between mind and manifestation. This is merely a small step on a long journey however...

The core idea of this work sprang into existence during a brainstorm session with a good friend of mine, Arjen: we were wondering how we could design aesthetically pleasing levels for a hypothetical first-person game quickly with just the two of us together, without the need for a dedicated artist. Eventually we landed upon the idea of extending wave function collapse such that it would support localized restriction of the kinds of allowed tiles based on (semantic) categories. While initially met with disapproval due to its deceiving simplicity, the idea caught on after being fleshed out, and even resulted into a paper, and a presentation at the procedural generation workshop of the FDG 2023 conference, in Lisbon. I am incredibly excited to see how researchers and game developers will use the knowledge presented here for their own work.

The past year was a tough year for me. It was a year of change, a year of breaking through, but also a year of sacrifices. Juggling four occupations (thesis, work, Invertigo development, doing board at SoSalsa) at once is not exactly the best thing for your personal bonds with other people, but as a result I came to value those who stuck with me more than ever before. The bond that I strengthened the most however, is the bond with myself, through many moments of introspection and self-reflection. It was a year of growth, for I have grown more as a person in the past year than in any other year of my life, in no small amount thanks to the people who supported me through this journey, some of which absolutely require explicit mention;

I would like to thank Rafa, not only for being a great supervisor, but also for being an amazing friend - you were there for me when I was as lost as a man could be, even before I had started working on my thesis. I would also like to thank all the wonderful people at SoSalsa for bringing in some fun into my life - in particular I would like to thank my fellow ex-board colleagues from B17 just for being a bunch of amazing human beings with whom it was an absolute pleasure to serve the association. My thanks also extend to my family - which also includes Alan, Daan, Ella, and others - for having my back when I struggled the most, and for supporting me with love and care. Finally, I would also like to thank my long-time close friends Niels, Max, Jop, Arjen and Justin - each of you enriching my life in your own beautiful ways.

Shaad Alaka
Delft, the Netherlands
November 10, 2023

Contents

Chapter 1

Introduction

As applications and hardware grow in power, demand rises for more complex models of reality in the form of virtual environments, which are more effortful to create for a designer. It is therefore attractive to explore methods that can reduce this effort, while remaining true to the designer's intent through sufficient control. For a long time now, the use of Procedural Content Generation (PCG) techniques has been one of the answers to this problem (e.g. in No Man's Sky[1], Terragen[2], Star Citizen[3], online D&D map generators[4], Minecraft[5], Unreal Engine[6]). While very convenient for generating large volumes of output, a common pain point of such techniques is the lack of control. There are several known ways of improving this: adding (more) parameters [1, 2, 3], combining several algorithms, each one responsible for a specific part of the PCG [4, 5] or simply adding interactivity [6, 7, 8]. This last method in particular is interesting for one specific algorithm that goes by the name of Wave Function Collapse (WFC).

Since its inception in 2016, WFC has become a popular algorithm for procedural generation of textures, environments, objects and other content that can be represented by a grid [9]. The attractiveness of WFC comes from its generic nature and uniform building blocks consisting of some entities that are allocated to nodes on a graph, and constraints that are tied to these entities, indicating how they can be placed on this graph in relation to one another; it is essentially a constraint solver [9, 10]. Because of this generic nature, several variants have since then been proposed, extending its functionality and convenience [11, 12, 13, 14, 15]. In this work, we are interested in the variant that operates on a grid with cells, with the entities being tiles, also known as the 'simple tiled model' [9]. Adding interactivity to this variant of WFC is attractive because the algorithm has obvious entry points for manual input, while also having a ubiquitous user interface for this added interactivity, namely the equivalent of painting on a canvas.

Exposing such entry points for interactivity to the user is often done via an editor application of some sort, which brings us to the realm of mixed-initiative editing: a back-and-forth between man and machine. Such editors have proven to be very convenient, as they can effectively support designers who wish to express their intent, while amplifying their creative freedom [16]. Recent approaches at using WFC in this fashion have shown great promise and versatility, for instance the miWFC prototype[7] [6]. The uniformity of WFC, however, while very nice for the algorithm's simplicity, is also responsible for some limitations with

[1] https://www.nomanssky.com/
[2] https://planetside.co.uk/terragen-overview/
[3] https://robertsspaceindustries.com/star-citizen
[4] https://azgaar.github.io/Fantasy-Map-Generator/
[5] https://www.minecraft.net/en-us
[6] https://www.unrealengine.com/en-US
[7] https://github.com/ThijmenL98/miWFC

respect to the expressive power of an editor powered by it. These limitations cause the designer to exert cognitive load in order to overcome them, which manifests itself in two distinct ways:

- The designer always has to consider everything at the most detailed level, since WFC only offers a single level of detail, namely the flat set of tiles it receives as input.
- The designer has to think about satisfying constraints from existing tiles on the grid while placing new tiles (i.e. has to pick the exact tile that fits onto the already existing and surrounding tiles).

Both phenomena clearly demand cognitive load; in other words, without these there would be less effort spent on using the editor and memorizing a variety of information (e.g. specific constraints, or additional detail level considerations) that is not relevant to the intent of the designer.

Hypothesis 1. *The cognitive load induced by the aforementioned inhibitions is significant enough for a designer to notice and thus be negatively affected by it.*

There is evidence that shows that increased cognitive load results into reduced productivity [17, 18], which may in turn result into reduced creativity as well [19]. The problem mostly stems from the fact that humans tend to reason in semantics [20], while the algorithm does not have any discernment of semantics built into it: the tiles may contain several semantic concepts that are blurred together, but there is no built in support for a structure that can keep these untangled. Because of these entangled semantics, the tiles also become complicated in terms of constraints. One can already sense this problem before even verifying it, just try to describe such a complicated tile: 'a sandy piece of road on a patch of grass that makes a turn to the right, starting from the top'. To the base WFC algorithm however, it is 'just another tile'.

Hypothesis 2. *Being able to disentangle semantics, so that designers can utilize them separately, will eliminate both aforementioned sources of cognitive load.*

This may seem like a leap of thought, but the basic idea here is that the semantics of a tile often create the necessity for constraints; being able to separately reason about the semantics, also implies being able to separately reason about the constraints. Using this reasoning, one idea could be to have semantic abstractions that group tiles together, and then allowing the designer to locally constrain the kinds of tiles that can exist at a particular cell on the grid with these semantic abstractions. For instance, the semantic abstraction for 'grass' would encompass all tiles that contain grass on them. In turn, due to the presence of numerous tiles in this abstraction, the constraints are not as specific anymore, which removes the need to very specifically think about the constraints of a single tile. The same holds true for 'road'; no longer is it necessary to think about all the twists and turns that the roads on a tile with a piece of road on it can take, they are all encompassed by the semantic abstraction for 'road'.

While the core idea is already sufficient for eliminating both sources of cognitive load, an important opportunity is missed in relating these semantic abstractions to one another. Consider that humans tend to organize semantics hierarchically, which makes such a representation a natural fit [20]. Moreover, the idea of 'level of detail' hints towards some kind of hierarchy as well. The breadth of such a hierarchy covers the semantic abstractions, while the depth corresponds to variation in level of detail.

The concept of level of detail can then be attributed to this hierarchical structure by relating these abstractions to one another through relations that indicate the representation of some semantic trait. For instance, a 'village' abstraction could represent the idea that 'house', 'road' and 'grass' could belong in a village. Furthermore, 'house' could be an abstraction as well, representing the idea that 'wall' and 'floor' belong in a house. Besides just relating the abstractions, the hierarchy also allows for quantifying the level of detail instead of just having a qualitative loosely coupled approach; it gives the ability to formally specify that some semantic abstraction is higher level than another, which is useful for a variety of uses, such as targeting all abstractions that fall beneath some higher level abstraction.

Having an algorithm that is capable of understanding hierarchical semantics opens many doors to many novel editing facilities in a mixed-initiative editor that would be powered by it: as the algorithm can now distinguish semantics, the editor can now offer tools to the designer that operate on these semantics. This has high potential to further reduce cognitive load in previously unsuspected ways.

1.1 Research Questions

In this work, we present the research and results that come forth from trying to answer the following research question and its sub-questions, that correspond to the hypotheses presented earlier:

Research Question: How can we reduce the cognitive load required for working with a mixed-initiative environment editor powered by an algorithm that is based on WFC?

- How can we enhance the WFC algorithm with hierarchical semantic abstractions?
- What novel features does a WFC version enhanced with hierarchical semantics enable that may reduce cognitive load?
- What kind of other editing facilities can we implement that may reduce cognitive effort required?
- How does the experienced amount of cognitive load differ between an editor with the novel features enabled by the enhanced WFC algorithm and an editor driven by stock WFC without these features?

Looking ahead already, the newly designed algorithm will be referred to as Hierarchical Semantic Wave Function Collapse (HSWFC) throughout the rest of the work.

1.2 Methodology

The general flow of this thesis consists of three parts: one for conceiving the core algorithm, one for designing the new editing facilities and building an editor that implements them in a user-friendly manner, and one for performing an evaluation of this editor with a user study. Each step builds upon the previous one; the aim is to eventually be able to answer all the questions posed in the previous section.

Exploration phase - how to modify WFC?

The first thing required is to have an idea of how to modify the algorithm in order for it to support hierarchical semantics. This requires a formal definition and specification of the algorithm, but for exploration purposes an implementation is deemed to be useful as well. To this end, an implementation of the algorithm is created as a driving part of a prototypical editor, written in Python using `pygame`[8] and `numpy`[9]. Such a basic editor allows for empirical confirmation and studying of the functionality and workings of the algorithm based on our speculations, without having to resort to constructing complicated proofs and derivations. Instead, a rich debugging suite can be built into the editor to see precisely how the algorithm behaves in specific scenarios. It also allows for a quick iteration cycle that would show immediately interesting results, in no small part thanks to the ease of use of Python and the broad toolkit that `pygame` and `numpy` provide together.

Real world use-case implementation

After being satisfied with the official specification of the algorithm and how its implementation is functioning, the next step is to officially specify the editing facilities that we will consider for this work, and use these as requirements for building an editor. Such an editor is no longer prototypical and crude in usage, but is much more suitable for designers to use. The new editor is built as a web-application, for two reasons mainly:

- Web browsers are ubiquitous, thus making your application web-based has large implications for accessibility. Furthermore, no software installation would be required at all for users.
- Many sophisticated front-end frameworks exist, and User Interface (UI) specification is quite mature with support from `CSS4` and `HTML5`.

In the end, this also allows for quick iterations on the UI design, which enables experimentation and finding the optimal layout, in preparation for the next phase.

[8] https://www.pygame.org/news
[9] https://numpy.org/

User study

Finally, the user-friendly editor has to be evaluated, to see whether the new HSWFC algorithm powering it is truly able to reduce cognitive load via the new editing facilities when compared to an editor that is driven by its stock counterpart without such features. We opt for blind A/B testing in order to recognize this difference. To recreate the experience of using an editor that does not benefit from HSWFC functionality and make it as close as possible to the HSWFC editor, a modified version of the web-editor is conceived.

Then we commence with the user study, where NASA TLX is used as a measuring guideline for cognitive load. Users are given equivalent tasks in both groups, with the only difference being the editor. Besides just measuring cognitive load, we also want to qualitatively assess the resulting outputs of the tasks, and see whether there are differences between the two groups in those. Lastly, we also opt for asking some additional questions about phenomena that come up during use of a WFC-driven tile editor, and some specific questions that target the HSWFC-enabled features, in order to get some insight into the overall approval of the new functionality and opinions about the intricacies that come with using WFC as a backbone.

Chapter 2

Related work

In 2016, Maxim Gumin unleashed the WFC algorithm, publishing a repository containing his initial implementation [9]. Since then, WFC has had a profound impact on technical artists and game developers, getting adopted, adapted and used in commercially published and upcoming projects (Caves of Qud, Townscaper, Matrix Awakens), and in research. The repository has become a hub of anything related to WFC, linking to research, derived works, alternative implementations, etc. [10, 12, 11, 14, 21, 22, 15]. Years before, Paul Merrel had published the conceptually identical Model Synthesis algorithm, though it did not catch on as much as WFC did, possibly due to its lower accessibility, main 3D focus with a rather broad scope, and computing requirements at the time [23].

2.1 The basic WFC algorithm

For completeness, we provide here a generic description of how WFC operates, to set the context for this work. WFC accepts as input (T,A), where T is a set of tiles, and A is a set of adjacency constraints over pairs of tiles in T. As output, it produces a finite grid where every cell has an assigned tile. Tiles may be placed into cells, and each cell keeps track of which tiles is it still allowed to contain, given the constraints induced by its neighbouring cells. Initially, each cell of this grid is empty and can potentially contain any tile (hence the analogy with 'wave function'). As the algorithm progresses (see Algorithm 1), the following three steps are iterated:

1. A cell is chosen to be 'collapsed': for this some heuristic can be used, e.g. the cell allowing the least amount of potential tiles (i.e. lowest entropy);
2. A tile is chosen to collapse that cell into: for this, one of the tiles is chosen among those allowed for that cell, possibly with non-uniform choosing probabilities among the choices;
3. The collapse propagates its effects to the neighbouring cells, disallowing tiles on them that are no longer allowed to be adjacent to the origin cell. This cycle repeats with the neighbours of the neighbours, etc, also known as the propagation wave. When no more changes occur to the allowed set of tiles from cell to cell, propagation ceases.

This cycle repeats until either all cells have been collapsed, producing a grid of tiles that satisfies the constraints in A or, alternatively, when a conflict is reached. The latter happens if a cell ends up without any allowed tile choices after a propagation wave hits it, rendering the current grid instance unsolvable.

Algorithm 1 Basic WFC algorithm

```
Initialize algorithm (building tile and constraint tables)
repeat
    Choose next cell to collapse
    Choose which tile to collapse it into
    Collapse and propagate constraints
until Each grid cell hosts a tile, or a conflict occurred
```

As noted before, in this algorithm, both the cell to collapse and the tile to collapse it into could potentially be chosen by a human. This fact provides the basis for an interactive, mixed-initiative WFC editor, in which the user may directly select, on the output grid, which cell(s) to collapse into some selected tile. In this setting, WFC can automatically validate the action, and further propagate any changes across the output grid.

2.2 Mixed Initiative Editing and WFC

Adding mixed-initiative interactivity to WFC to make the generation process more spatially controllable has been proposed with miWFC, an interactive editor that allows you to place/overwrite tiles with a brush, create snapshots, regenerate marked parts, and spatially alter the tile probabilities [6]. In addition, other proposals of WFC-based interactive editors have been made, various links are available on the WFC Github repository [9]. Other types of mixed-initiative interactivity have been proposed for WFC that do not involve spatial control. Karth and Smith [24], for example, propose to allow the designer to intuitively adapt constraints by providing positive and negative examples of tile combinations; to fulfil them, a back-and-forth process progresses towards the generated result. There are also applications of mixed-initiative WFC interactivity within a game. In Townscaper, for example, Oskar Stålberg already shows the concept of letting users interactively collapse specific parts of the grid, while letting WFC choose the appropriate tile for the context [25]. This last application was a core inspiration for this work. Adam Newgas explored the concept of being able to edit WFC grids interactively and built a proof of concept editor, while also briefly touching upon how this can be implemented using 'driven WFC', as used in Townscaper [26] [27]. All these approaches demonstrate the flexibility of WFC when it comes to involving humans in the generative process.

2.3 Using PCG for environment design

There are several tools on the market right now that are used for environment design and involve some form of PCG. For instance Terragen[1], World Machine[2] and World Creator[3] are all similar tools for world design in artwork/movies that have support for generating

[1] https://planetside.co.uk/
[2] https://www.world-machine.com/
[3] https://www.world-creator.com/gallery.phtml

terrain procedurally, given relevant parameters. Bryce 3D[4], an older environment rendering and 3D art application, has support for procedurally generated heightmapped terrain, which could then be edited by users with a brush, or with procedural filters. Townscaper, as mentioned before is able to procedurally generate towns, given user input on where to do this. Unreal Engine 4 can be combined with Houdini to create tools that are able to procedurally generate entire cityscapes, or provide 3D assets that are modular and procedural[5]. Unreal Engine 5 has built in support for procedural and conditional placement of props via its node system, which implies that it can also be conditioned on user input in order to facilitate mixed-initiative editing[6]. DeBroglie by Adam "BorisTheBrave" Newgas is a C# toolkit for procedural environment design powered by WFC at its core, but with many extensions such as additional constraints and additional grid types, and it has integration with Unity 3D via Tessara, a WFC toolkit [28, 13]. Similar WFC-powered toolkits exist for Unreal Engine as well. Paul Merrel recently invented a method for procedurally generating polygonal shapes that are similar to some example shape(s) based on grammar extraction, which could be used to generate a large and varied environment that looks similar to a small example [29]. While much focus in this paragraph has been on exterior environments, there also exists work on procedural generation of interior environment [30], or toolkits such as Dungeon Architect[7], both with plugins for Unity 3D. Several video games exist that utilize PCG in real-time for both exterior and interior environments, such as No Man's Sky, Minecraft, Dwarf Fortress, Caves of Qud, Spelunky, Terarria, the Diablo series, Valheim, and many more[8].

2.4 Designer intent, semantics and PCG

Being able to more accurately capture designer intent is desirable because it will speed up the overall design process. This has been done so far in a variety of ways. In one such example, an interactive world editor [16] was augmented to capture design semantics, introducing semantic constraints such as e.g. preserving line of sight between world entities [31]. In another project, the designer can input a rough design specification that guides a procedural generator for room layouts [32]. There have also been efforts at using PCG for architectural structures, via 3D building blocks and multiple types of semantic constraints among these that are enforced during generation (e.g. construction, traversability, occurrence) [33].

2.5 PCG and hierarchies

There have also been some efforts at incorporating hierarchies in PCG in general [34], e.g. by using a hierarchy of rules [35]. In particular, one approach that comes close to the idea behind HSWFC is rule-based layout solving [36], which also uses classes of objects in

[4] http://www.bryce3d.co.uk/bryce-3d/
[5] https://forums.unrealengine.com/t/inside-unreal-procedural-tools-with-houdini
[6] https://docs.unrealengine.com/5.2/en-US/procedural-content-generation-overview/
[7] https://dungeonarchitect.dev/
[8] https://en.wikipedia.org/wiki/Category:Video_games_using_procedural_generation

order to populate areas marked with those class constraints, though it does not allow for a hierarchy that goes deeper than one level. In another example, a 2D game map consisting of tiles is divided into chunks and then clustered, in order to find high-level tiles, which are similar in spirit to the meta-tiles presented in this work, though again only representing a single level of depth [37].

Chapter 3

HSWFC - Algorithm Design

The core problem of stock WFC in the context of this work is the fact that it is agnostic to semantics, which makes it cumbersome for designers to express their intent. As shown in Chapter 1, semantic discernment across the tiles would benefit greatly from being conducted through a hierarchical structure. Therefore, two new abstractions will be introduced first: meta-tiles, as a solution to the semantic blind spot of stock WFC; and the meta-tree, to be able to intuitively structure the tiles into a hierarchy. Together, these abstractions then steer the design of the HSWFC algorithm.

3.1 Meta-tiles

Tiles in stock WFC are the immediate manifestation of the semantic traits they carry; they are concrete and final. In HSWFC, we wish to obtain an abstraction that is able to represent these semantic traits, in order to be used by a designer to constrain parts of the grid by them. This description spawns several requirements:

- Since the abstraction has the ability to constrain parts of the grid, it must somehow be able propagate these changes.
- The designer needs a way to mark areas on the grid with the semantic abstraction.

Within WFC, tiles provide both of these pieces of functionality without having to reinvent the wheel. Therefore, we wish to introduce a new kind of tile:

> **Definition 1.** *A meta-tile is an abstract tile that is able to represent semantic traits that are present in other tiles.*

For brevity, we will use the following notation: given meta-tile T_{MT} that represents any number of semantic traits present in some tile T_X, we write $T_{MT} \xrightarrow[\text{Repr}]{} T_X$. Furthermore, we shall call such a tile T_{MT} an ancestor of T_X, and T_X a descendant of T_{MT}.

The intended use of a meta-tile is to facilitate the grouping of tiles sharing specific semantic traits, so that these traits are available to the environment designer in isolation. The details of which semantic traits exist and how they are structured, depends on the model of reality that is being emulated and the context of application.

A meta-tile represents semantic traits, but is not a manifestation of these; it is abstract and ephemeral in nature, and therefore relies on other tiles for the manifestation to occur. Concrete tiles from stock WFC do not exhibit this reliance, by virtue of being a manifestation of

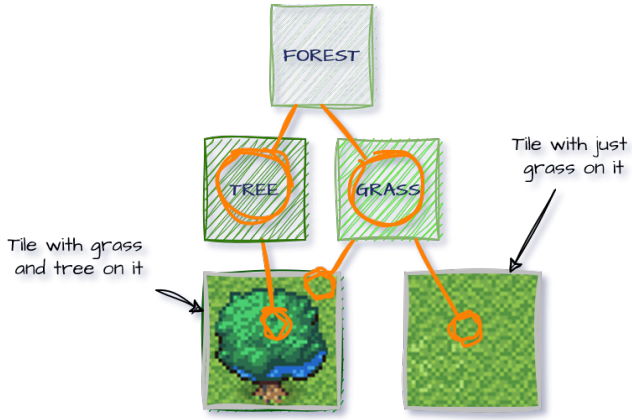


Figure 3.1: Meta-tiles T_{FOREST}, T_{TREE} and T_{GRASS} represent the semantic traits that are present in other tiles, while the remaining terminal tiles are the manifestation of these traits as concrete instances. Since both T_{GRASS} and T_{TREE} are elements that could appear in a forest in this context, this idea or 'trait' can be represented by T_{FOREST}.

their semantic traits. Therefore, in order to distinguish them from meta-tiles, we shall refer to these tiles as 'terminal tiles'.

> **Definition 2.** *A terminal tile is a tile that solely represents its own semantic traits, and is the manifestation of these.*

Meta-tiles have several properties;

> **Property 1.** *Given meta-tiles T_A, T_B, and T_C, where $T_A \xrightarrow[Repr]{} T_B$ and $T_B \xrightarrow[Repr]{} T_C$, it follows that $T_A \xrightarrow[Repr]{} T_C$.*

This property establishes the notion of transitivity. For instance, in Figure 3.1 T_{FOREST} is able to transitively represent the semantic traits present in the terminal tiles, due to representing the semantic trait of 'forest' in T_{GRASS} and T_{TREE}, which in turn represent 'grass' and 'tree' in the terminal tiles, like links in a chain.

This has important implications for the next property:

> **Property 2.** *A meta-tile behaves as the superposition of all tiles that carry the semantic traits represented by it.*

An area consisting of meta-tiles indicates a desire to constrain that area to specific semantic traits, hence all tiles that exhibit these should have the potential to appear in this area, and all tiles that do not exhibit these traits should be excluded from appearing. Together with

Property 1, it establishes that a meta-tile such as T_{FOREST} in Figure 3.1 is in superposition of T_{GRASS}, T_{TREE} and both terminal tiles.

This idea of having a tile appear in place of a meta-tile leads us to the following property:

> **Property 3.** *Given meta-tile T_{MT}, and tile T_X with $T_{MT} \xrightarrow[Repr]{} T_X$:*
>
> - *T_{MT} can be substituted by T_X without loss of semantic representation of T_X.*
> - *T_X can be substituted by T_{MT} without loss of any semantic representation.*

This is best illustrated through some examples, using the configuration of tiles as presented in Figure 3.1:

Example 1: Given meta-tile T_{FOREST} , it can potentially be substituted by either T_{TREE} or T_{GRASS}. If we choose T_{TREE} for the substitution, then we are (obviously) still representing the semantic trait that T_{TREE} was representing. Note that this is a narrowing behaviour; we do lose the representation of the semantic trait of T_{GRASS} when going from T_{FOREST} to T_{TREE}.

Example 2: Given meta-tile T_{GRASS}, it can be substituted by T_{FOREST} without losing the ability to represent the semantic trait that T_{GRASS} was representing, due to transitivity (Property 1).

If we look at the whole body of the representational relationships that tiles carry between each other, a graph-like structure emerges naturally. The next section will describe the limitations we need to impose on this structure in order for it to be useful.

3.2 The Meta-tree

The overall structure that arises from the representational hierarchy between tiles is called the meta-tree (see Figure 3.2). Let us assume that the tiles function as nodes in this graph, and that the edges carry the meaning of *directly* representing semantic traits of another tile. The first thing to notice is that these edges are directed by definition of semantic representation, which is a uni-directional relationship.

Now also consider the following example; given meta-tiles T_{LAND} and T_{WATER}, and $T_{LAND} \xrightarrow[\text{Repr}]{}$ T_{WATER} (e.g. a lake), do we allow $T_{WATER} \xrightarrow[\text{Repr}]{} T_{LAND}$ (e.g. an island)? While this example has some merit, it is highly impractical for the algorithm design, as it introduces an extra layer of complexity into the graph without necessarily being a main point: the presence of cycles. The intended shape of the structure, as explained in Chapter 1, is that of a hierarchy, and cycles do not fit within this idea. Therefore, the following holds;

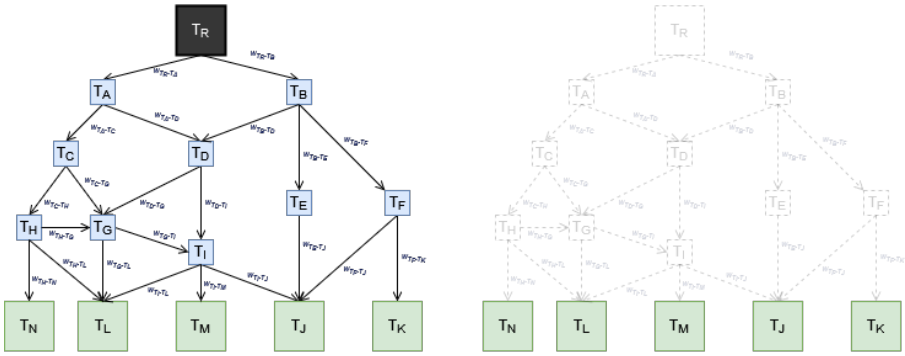


Figure 3.2: On the left, a generic meta-tree is shown. T_{ROOT} (abbreviated to T_R) is shown in black, and the terminal tiles are shown in green, both are enlarged. The blue tiles are meta-tiles other than T_{ROOT}. On the right, a comparison is made with a non-hierarchical WFC tileset, which would only consist of the terminal tiles.

> **Property 4.** *Given meta-tiles T_A and T_B with $T_A \xrightarrow[Repr]{} T_B$, it is not possible to also have $T_B \xrightarrow[Repr]{} T_A$.*

This is a conscious choice that has been made for this work; we encourage investigating the use cases of cyclic/recursive semantic representations in future work.

The consequence of this property is that the graph takes on the shape of a connected Directed Acyclic Graph (DAG) [38]. Such graphs consist of nodes that only have outgoing edges (sources), nodes that only have incoming edges (sinks), and nodes that have both. Starting at any node in such a graph and simply following the outgoing edges will guarantee that you arrive at a sink eventually [39]. There is a direct correspondence between sinks and terminal tiles, since terminal tiles do not represent any semantic traits of other tiles.

Most meta-tiles will represent some semantic traits in some other tiles, and have their semantic traits be represented by other meta-tiles. However, it is not useful to extend this graph infinitely, as we only have as many nodes as the amount of tiles that have been defined in the input, and a DAG must have at least one source, or otherwise abstain from the acyclic property [38]. Therefore, there must exist at least one meta-tile that does not have its semantic traits represented by another meta-tile. This is where we make another conscious design choice; we consider this to be the only source in the DAG:

> **Definition 3.** *T_{ROOT} is the only tile in the meta-tree for which no tile T_X exists such that $T_X \xrightarrow[Repr]{} T_{ROOT}$.*

This definition forces $T_{ROOT} \xrightarrow[\text{Repr}]{} T_X$ for each tile T_X, which follows from Property 1 and the fact there is a single source (see Proof A.0.1).

> **Property 5.** *Given the set of all tiles in the input T, $T_{ROOT} \xrightarrow[Repr]{} T_X$ for each $T_X \in T$.*

Then, from Properties 2 and 5, it follows that:

> **Property 6.** *T_{ROOT} can be considered to be in superposition of all tiles in the input.*

The last thing to direct our attention towards are the weights on the edges in the meta-tree: they correspond to the degree of representation of some semantic trait for some tile. Below is an example of how one could interpret this weight in relation to other weights.

Example: Given meta-tile T_{CORNER} representing the semantic trait of 'being a corner' for several terminal tiles that resemble corner pieces of a building. Now perhaps there is one particular corner T_X that we find more 'corner-esque' than any of the other corners. In other words, T_{CORNER} represents the 'corner-ness' of T_X better than it does for the other corner pieces. This is when we would make w_{T_{CORNER},T_X} larger than the other weights w_{T_{CORNER},T_Y} where $T_{CORNER} \xrightarrow[\text{Repr}]{} T_Y$.

3.3 Algorithm Design

At a high level, HSWFC operates nearly in the same way as WFC, as described in Chapter 2; there is still a loop that chooses the next cell to collapse into some tile, after which a propagation wave follows. At this level there is only one core difference, which is the termination condition; each cell on the grid must host a terminal tile, rather than just a tile, in order to obtain a concrete output. Conceptually, this is the same condition that stock WFC has, except now we need to make the distinction more specific due to the presence of meta-tiles. Once we dig deeper into the sub-components, many other differences start to reveal themselves.

3.3.1 Choice of next cell

In stock WFC, a cell can only collapse when it is empty. Upon doing so, it has reached its final state immediately and can no longer be chosen again for a collapse operation. Of the entire set of tiles that is available to that cell, only a single tile will be chosen to occupy it. In HSWFC this is no longer the case, because meta-tiles are in superposition of multiple other tiles by definition, which we can formalize in the following manner:

Definition 4. *The superposition state of a tile T_{MT} is equivalent to the set of all tiles T where $T_{MT} \xrightarrow[Repr]{} T$.*

Note that the "superposition state" of a terminal tile is just itself, due to Definition 2. When a cell collapses to a meta-tile T_{MT}, it collapses to a superposition of tiles, which is determined by the intersection of the previous set of allowed tiles at that cell (influences from earlier propagation waves), and the superposition state of T_{MT}. Because the cell is still in a superposition, it can potentially be chosen again to collapse, which implies that HSWFC is capable of collapsing cells that already have an occupant, essentially replacing the occupant; a notable difference with stock WFC only being capable of collapsing empty cells. The meta-tile essentially acts as a proxy for the superposition state that is currently active at that cell, which will be used in Section 3.3.5 to facilitate cell resetting.

3.3.2 Choice of tile

In HSWFC, the set of tiles that can be chosen during a collapse for some cell C is considerably smaller than in WFC. Given that C hosts meta-tile T_{MT}, then HSWFC can only legally choose a tile T_X for C for which $T_{MT} \xrightarrow[\text{Repr}]{} T_X$ holds, because T_{MT} is only in superposition of such tiles, thus only then the representation of the semantic traits in T_X get *preserved* upon substitution, as shown in Property 3.

More strictly, a deliberate choice is made to make HSWFC limit this to each tile T_D that is *directly* represented by T_{MT} during automated generation. This is done in order to be able to use the weights w_{T_{MT},T_D} in the meta-tree as an additional input for inferring information about the distribution of the available tile choices; For instance, a cell with T_{FOREST} will collapse into either T_{GRASS} or T_{TREE}. If T_{FOREST} represents the 'belongs in a forest' trait better in T_{TREE} than in T_{GRASS}, then it would make intuitive sense that there would be more trees than grass tiles in this forest, hence the weights affect the choosing probability. This idea also implies that cells always collapse in a step-wise manner; one meta-tile at a time until a terminal tile is reached, ensuring that all meta-tiles in the path from the starting meta-tile until the terminal tile were visited in the meta-tree.

3.3.3 Initialization

In HSWFC, grid cells can be initialized to T_{ROOT}, since according to Property 6 this tile carries the same meaning that the empty cells in WFC have. This choice is made for implementation consistency and unification reasons; resetting or initializing cells no longer has to involve setting them to a special state where they have no occupant. Instead, the cell state that represents the superposition of all tiles has a direct correspondence to T_{ROOT}, thus we can now assume that every cell in a grid will be filled with a tile at all times.

3.3.4 HSWFC input processing

As shown in Section 2.1, the input to stock WFC consists of the pair (A,T), where T is the set of tiles, and A is the set of adjacency constraints over T. For HSWFC, T now also contains meta-tiles, and an additional input term E is required that consists of the direct representation of semantic traits among the tiles as a set of (T_A,T_B) tile pairs and an associated weight w_{T_A,T_B}, where $T_A \xrightarrow[\text{Repr}]{} T_B$ directly. Thus HSWFC requires as input the triple (A,T,E), where T and E together make up the meta-tree. This difference in input is also shown in Figure 3.2.

Some notation will be used for dealing with adjacency constraints:

- A represents the set of all adjacency constraints in the input.
- A_{NAME} corresponds to the set of tiles that can be adjacent to T_{NAME}.
- $A(T_A,T_B)$ represents an adjacency constraint between T_A and T_B.

Meta-tile constraint inference

Adjacency constraints in HSWFC are positively defined, just as in stock WFC. Therefore, not defining any constraints for meta-tiles will result into them not being able to be adjacent to any tile, which makes them unusable. Providing them manually in the input is one option, but it could be much more convenient to infer them from the meta-tree.

Because of Property 2, we can establish that a meta-tile T_{MT} can infer its constraint set as:

$$A_{MT} = \bigcup \left\{ A_X \;\middle|\; T_{MT} \xrightarrow[\text{Repr}]{} T_X \right\}$$

This can be realized using a bottom-up approach, starting at the terminal tiles, see Algorithm 2. The constraints of higher level tiles progressively get updated with the constraints of lower level tiles as the algorithm moves up the meta-tree. Note that self-adjacency is not

Algorithm 2 HSWFC meta-tile constraint inference

```
1: Q ← queue initialized with all terminal tiles
2: while Q has entries do
3:     T_CURRENT ← dequeue next entry in Q
4:     for each T_MT such that T_MT ⟶(Repr) T_CURRENT do
5:         Add adjacency constraints A(T, T_MT) for tiles T ∈ A_CURRENT
6:         if T_CURRENT is self-adjacent then
7:             Update adjacency constraints so that T_MT is self-adjacent
8:         Enqueue T_MT if not already enqueued in Q
```

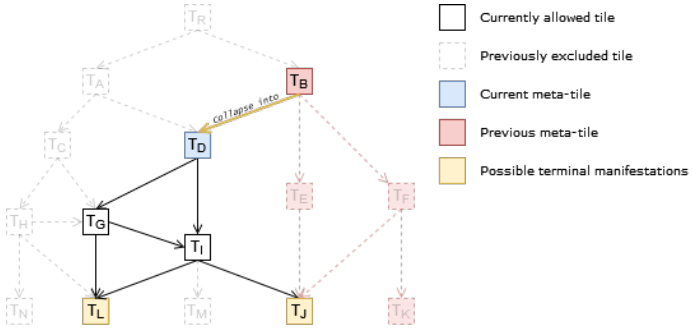


Figure 3.3: This figure utilizes the meta-tree representation to depict the state change of a cell C hosting meta-tile T_B, which collapsed into meta-tile T_D. Note that T_M got excluded by an earlier propagation wave, while the other tiles got excluded by an earlier collapse from T_{ROOT} (abbreviated to T_R) to T_B. Also note how the exclusion of T_M affects the superposition induced by T_D, which now only involves T_L and T_J.

considered in the union displayed above; for self-adjacency of T_{MT} there needs to exist only one tile T_X with $T_{MT} \xrightarrow[\text{Repr}]{} T_X$ that can also be self-adjacent.

Meta-tile constraints

The fact that meta-tiles are actually tiles, provides the possibility to specify constraints that involve meta-tiles in the input. There are two distinct types of specifications here:

- Between meta-tiles and terminal tiles
- Among meta-tiles

Inspired by Property 2, we can interpret the presence of a meta-tile T_{MT} in an adjacency constraint to carry the meaning of a wildcard; Given adjacency constraint $A(T_{MT}, T_{OTHER})$ where T_{MT} is a meta-tile and T_{OTHER} is a terminal tile, each terminal tile T_{MTERM} for which $T_{MT} \xrightarrow[\text{Repr}]{} T_{MTERM}$ will generate an adjacency constraint $A(T_{MTERM}, T_{OTHER})$.

If T_{OTHER} is a meta-tile instead, $A(T_A, T_B)$ will be added for tile pairs that satisfy:

$$(T_A, T_B) \in \left\{ T_A \;\middle|\; T_{MT} \xrightarrow[\text{Repr}]{} T_A \right\} \times \left\{ T_B \;\middle|\; T_{OTHER} \xrightarrow[\text{Repr}]{} T_B \right\}$$

In essence, all such adjacency constraints that involve a meta-tile get flattened to the terminal tiles (and as a result, stripped from the meta-tiles), which must happen before meta-tile constraint inference (since we are changing the constraints of terminal tiles).

3.3.5 Propagation

Propagation occurs in a nearly identical fashion as in WFC. However, a particularly intricate and non-obvious consequence of the design of meta-tiles that affects propagation, follows from Property 2. When a tile gets excluded from a cell due to earlier propagation waves (as explained in Section 3.3.1), it can no longer be part of the superposition state of a meta-tile. As tiles get excluded from this superposition state, the associated constraint sets can also no longer be taken into account. Since terminal tiles alone can fully determine the constraint set of a meta-tile (shown in Section 3.3.4), only taking the constraint sets of terminal tiles into account during propagation ensures correctness, see Figure 3.3 for further clarification and an example. So in essence, this is exactly how propagation works in stock WFC, where we also only look at terminal tiles, except in stock WFC those are the only kind of tiles available in the input. In addition, whereas in stock WFC superpositions only get propagated at a later stage in the propagation wave, in HSWFC this often happens immediately after collapsing to a meta-tile due to their superposition states, see Figure 3.4.

A special case can occur however that requires extra care: there exists the possibility for a still allowed meta-tile T_{MT} at some cell C that each terminal tile T_X for which $T_{MT} \xrightarrow[\text{Repr}]{} T_X$ holds gets excluded. In this case, T_{MT} is no longer a viable choice for C, as choosing it will guarantee that C will never hold a terminal tile for the current grid instance. Therefore, T_{MT} needs to be excluded from C to ensure that HSWFC can terminate. See Algorithm 3.

Algorithm 3 HSWFC propagation

```
1:  procedure PROPAGATE(C : Cell)
2:      Q ← queue initialized with collapsed cells
3:      while Q has entries do
4:          O ← dequeue next cell in Q
5:          for all neighbouring cells N of O do      ▷ Omitting direction and bounds check
6:              CUR ← allowed tiles at O
7:              PRE ← allowed tiles at N
8:              ADJ ← tiles allowed to be adjacent to terminal tiles in CUR
9:              POST ← PRE ∩ ADJ
10:
11:             for all T_MT in POST do          ▷ As described at the end of Section 3.3.5
12:                 if no terminal T_X with T_MT ⟶(Repr) T_X exists that is allowed in POST then
13:                     Remove T_MT from POST
14:
15:             Update grid/entropy at N with POST
16:
17:             if POST changed compared to PRE then
18:                 Enqueue N if not already enqueued in Q
```

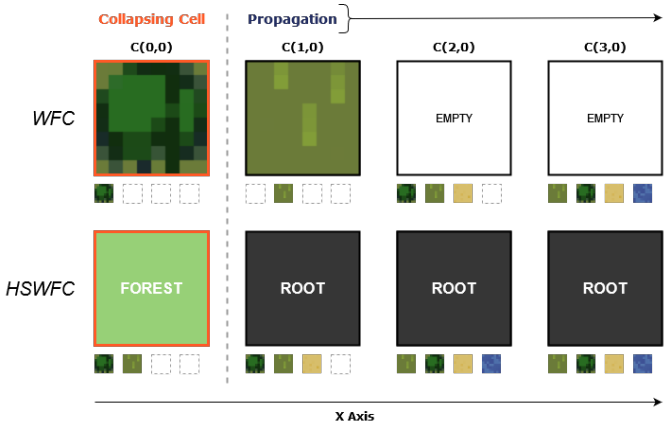


Figure 3.4: A collapse and propagation shown on a single axis, with the example tileset of Appendix B. The core difference in propagation between WFC and HSWFC; a cell may collapse to a superposition of tiles, instead of a single tile. Note how WFC also propagates superpositions, but further away from the collapsing cell (from $C(2,0)$ onward). Also notice how cell $C(1,0)$ for HSWFC hosts T_{ROOT} with T_{WATER} excluded from its superposition state.

Resetting cells (a.k.a. "uncollapsing")

Being able to universally reset cells back to their superposition state is highly desirable, because it allows designers to undo the effects of collapsing in a targetted manner, even more so in HSWFC, where there may be many intermediate superposition states. We can translate the concept of a "superposition state" to stock WFC as well: in that case, there is only one superposition state, namely the empty cell that can still host all possible tiles. The notion of resetting cells is not new to WFC, as projects such as Townscaper and miWFC clearly show the capability to do this. However, currently there is no (publicly available) formal definition or algorithmic framework that explains how exactly cells can be reset while maintaining grid consistency.

The difficulty in resetting a cell mostly stems from the fact that it is not just the collapsed cell that needs to be reset; all the cells that were affected by the propagation of the collapse will also need to have this influence undone somehow, which is information that is dissolved within the state of the cell together with numerous other propagation influences from other cells. One option is to also store the attribution of the removal of tile choices; for each removed tile choice, you would also store the cell responsible for it. This can quickly result in overhead from the explosion in the amount of data stored for the grid and from the fact that propagation cycles occur very rapidly.

Hence we opted for investigating a state-less solution. We found that such state-less methods should at least consist of the following steps:

1. Cells that are marked for uncollapsing are reset to their superposition state.
2. Some algorithm is ran to determine which other cells were affected by the collapses of the cells in the previous step. These are reset to their superposition states as well.

Algorithm 4 HSWFC depropagation

```
1:  Q ← queue initialized with uncollapsed cells
2:  P ← {}
3:  while Q has entries do
4:      O ← dequeue next cell in Q
5:      for all neighbouring cells N of O do        ▷ Omitting direction and bounds check
6:          PRE ← allowed tiles at N
7:          POST ← superposition state of current tile on N
8:          if POST changed compared to PRE then
9:              Enqueue N if not already enqueued in Q
10:         else
11:             Add N to P                          ▷ This is how we obtain the bordering context
12: PROPAGATE(P)
```

3. The constraining effect of the existing context on the grid should then be reintroduced, by propagating all the cells that may affect the cells that have been reset.

Finding the optimal algorithm for step 2 is an open problem. The trivial solution is to reset all cells on the grid to their superposition state, which completely undoes the effects of propagation everywhere. Then, simply propagate each cell again. Naturally, this approach is not very efficient, though most propagation waves will not travel beyond a single cell due to a lack of change in allowed tiles.

A slightly better solution is to perform something that we will refer to as "depropagation", after an uncollapse, which is shown in Algorithm 4. This is a propagation algorithm that resets cells to their superposition state, and only stops if there are no allowed tiles from cell to cell. Then, all cells bordering this body of reset cells are queued for propagation.

This works due to the nature of propagation; the bordering cells already contain all the constraining information from the cells beyond the bordering cells from previous propagation waves, hence all they have to do is simply propagate this back into the cells that have been reset. In practice, this still often ends up resetting a much larger area compared to the ground-truth of using state to keep track, though this is the solution that we will use in this work.

3.4 Generality of HSWFC

The approach presented in this chapter is general, in the sense that it can be applied to other variants of WFC as well, with minimal adjustments. The main part that needs context-specific adjustments is the constraint-inference, since constraint specifications often vary between the different models of WFC. The simple-tiled model was chosen in this work to minimize distraction from the main contribution, which is that of semantic tile hierarchies.

As an example, we can examine the overlapping model's compatibility with hierarchical semantics: in this case the primary elements are patterns that consist of tiles, instead of

just tiles, and adjacency constraints work by overlapping these patterns. Note that because of this, adjacency constraints are implicitly defined by pattern compatibility. A meta-tile would have the same function as with the simple-tiled model: it would be a way for the designer to limit the scope of a region to the semantic concept that the meta-tile represents. Thus, a meta-tile acts as a scoped wildcard, much like how empty space in the original overlapping-model version of WFC gives the ability to represent any pattern given that it fits onto its context. Investigating what other editing facilities and benefits HSWFC brings to the overlapping model and other variations (e.g. graph-based) could be a piece of future work with high potential. For instance, what about using meta-tiles in the patterns themselves? It would allow for the definition of "loose" patterns that would not need an exact tile match.

Chapter 4

Editing Facilities

With the specification that was provided in Chapter 3, significant editing functionality can be realized. The matter of fact remains that with stock WFC, we are restricted to only instantiating terminal tiles on the grid cells. With HSWFC and meta-tiles however, a lot more becomes possible.

In order to clearly show what each editing facility brings to the user, several example use cases will be presented in their respective sections. These example cases will use the hierarchical tileset presented in Appendix B. Take note that this is just one example tileset, with the specific context of an outdoors environment. This is purely done so that examples of usage can be provided, in order to give an idea of the utility that HSWFC has. Many other hierarchical tilesets can be made for this algorithm, which may generate countless other example uses cases for the editor facilities presented here.

4.1 Meta-tile painting

The most evident functionality that comes from having a meta-tree at our disposal is the ability for a user to paint with meta-tiles. This allows a user to paint with the semantics that the hierarchical tileset provides. This feature also brings some other interesting emergent functionality when combined with uncollapsing (also see Section 4.2): painting with T_{ROOT} becomes the equivalent of an eraser.

Example 1: See Figure 4.1(a). A user wants a certain region of the grid to be populated with grass and trees. To achieve this, the user paints over this region with T_{FOREST}. Then, letting the algorithm proceed will ensure that every cell hosting T_{FOREST} will eventually collapse into either T_{GRASS} or T_{TREE}.

> *Without HSWFC, the user would have manually placed the T_{TREE} tiles first, being mindful of the manner in which they distribute the trees, and then fill the remaining cells with T_{GRASS}.*

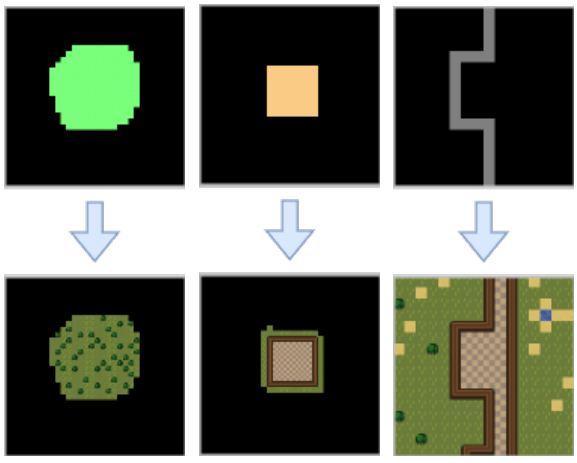
Example 2: See Figure 4.1(b). A user wants to put together a house in some area on the grid. To achieve this, the user just needs to paint a region with T_{HOUSE}, and then let the algorithm resume.

> *Without HSWFC, the user can still get away with placing T_{FLOOR}, as the constraints of this tile force the presence of either T_{FLOOR} or descendants of T_{WALL}, but this is counter-intuitive; the user desires to paint a house, not a floor. In addition, once inner walls and other complex house-interior structures are present in the tileset, this is no longer a viable strategy.*

Example 3: See Figure 4.1(c). A user wants to have a wall on the grid in a particular shape, perhaps for navigability purposes. For this, the user can use T_{WALL} and paint the wall in the desired shape, then leave it up to the algorithm to collapse the painted wall into the appropriate corners and straight pieces.

> *Without HSWFC, the user is required to spend time finding the right tile in order to construct a valid wall.*

In addition to these examples, all of them can be used simultaneously in a sketch, which can then be saved as some kind of design blueprint that can be generated multiple times to create variations. While not investigated in this work, this can extend to creating heterogeneous brush templates that consist of compositions of certain meta-tiles, e.g. a building with a certain shape, a room with certain contents, or a park, etc.



(a) (b) (c)

Figure 4.1: Three meta-tile painting example use cases, with at the top the painted meta-tiles and at the bottom the result after resuming the algorithm. (a) Painting forest with T_{FOREST}. (b) Painting a house with T_{HOUSE}. (c) Painting a wall of a specific shape with T_{WALL}.

4.2 Tile overwriting

The ability to replace a tile with another tile generally is regarded as standard functionality of a tile editor, yet overwriting tiles in WFC is rather tricky, because tiles that are currently occupying cells may have produced propagation waves from earlier collapses that may have affected other cells. With the depropagation algorithm shown in Section 3.3.5, overwriting becomes possible in a generic way, by first uncollapsing to T_{ROOT} and depropagating, and then collapsing to the designated tile as usual.

Naturally, while uncollapsing to T_{ROOT}, the propagation of the surrounding tiles after depropagation may immediately disallow certain tiles on the cells that were intended to be overwritten, which may make it impossible for the full painted region to be filled with the intended tile. Unfortunately, this is a limitation that is inherent to the nature of WFC and constraint propagation. There are two approaches for dealing with this:

- **Conservative:** Cells that cannot host the requested tile after uncollapsing to T_{ROOT}, will collapse normally from the first ancestor of the requested tile that is still allowed.
- **Destructive:** The surrounding cells are cleared in order to allow the requested tile to occupy all the painted cells.

In this work, we will only cover the first option, as the second option requires solving the non-trivial problem of knowing which cells *may also* have affected some other cell. It would be interesting to know which of the two is preferred by users, once a viable solution exists to implement the second approach.

Example 1: See Figure 4.2(a). A user has generated the full grid, but wishes to paint a village in some area that is currently filled with grass and trees. To achieve this, the user simply selects $T_{VILLAGE}$ and paints over the area that should become a village.

> *Without HSWFC, this operation would be only be possible by emptying the grid, painting all the rest of the grid as it was (this process could be automated in the editor), and then leaving the area designated for the village for the end.*

Example 2: See Figure 4.2(b). A user painted a big patch of water, but wishes to have an island with some houses on it. To achieve this, the user can simply select T_{HOUSE}, and paint with it on the water. Since T_{WATER} can only be adjacent to T_{SAND} (see Appendix B), the cells at the border of the painted area will collapse to T_{SAND} immediately, upon which T_{GRASS} follows, as the wall tiles represented by T_{HOUSE} can only be adjacent to T_{GRASS}. In other words, little islands with beaches are naturally formed to accommodate the houses.

> *As above, without HSWFC the grid would have to be emptied, painting back the rest of the environment, and then manually put together the island and the house similar to the example outlined in example 3 of Section 4.1.*

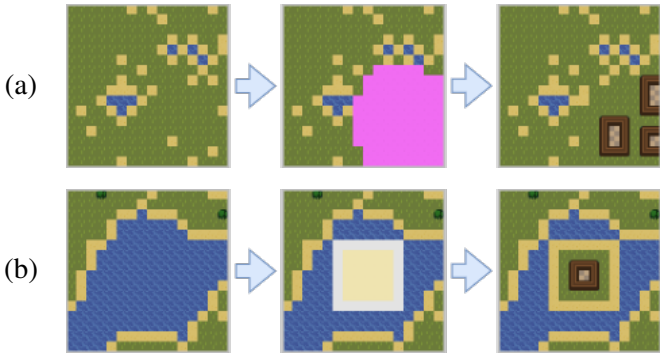


Figure 4.2: Two meta-tile overwriting examples. In (a), a village is added to an already generated environment using $T_{VILLAGE}$. In (b), a house is painted in the middle of a river with T_{HOUSE}. Note how the outer tiles in (b) could not collapse further down into T_{HOUSE} but instead collapsed into T_{LAND}, because T_{HOUSE} does not have any descendants that can be adjacent to T_{WATER}.

4.3 Dynamic tile distribution tweaking

The edges in the meta-tree between a meta-tile T_{MT} and each directly descending tile T_X, carry numerical weights w_{T_{MT},T_X} that give an indication of how well the semantic trait is represented by T_{MT} for some particular tile T_X. As explained in Section 3.3.2, HSWFC uses these weights to influence the probability that some T_X gets chosen after a collapse of a cell hosting T_{MT}, by deriving a (non-uniform) discrete probability distribution $P(T_X)$ from these weights. Tweaking the weights will result into a different tile distribution, which provides an opportunity to add interactivity. Note that the novelty here is that instead of having one big distribution with all terminal tiles in it, we now have many smaller distributions that can be independently tweaked: one for each meta-tile.

Example 1: See Figure 4.3(a). A user wishes to have a patch of forest on the grid that is slightly more dense than usual, and has therefore painted a region with T_{FOREST} as a first step. Instead of having to manually add additional trees after letting this region collapse, the user can temporarily override $w_{T_{FOREST},T_{TREE}}$ to be higher, and then let the algorithm resume.

> *Without HSWFC, there would be no $w_{T_{FOREST},T_{TREE}}$ to tweak; the distribution would have to be emulated manually.*

Example 2: See Figure 4.3(b). The user can control approximately how large the houses become on average from painting with T_{HOUSE}, by altering the balance between T_{FLOOR} and T_{WALL} through weight tweaking. This will also carry over to $T_{VILLAGE}$, since $T_{VILLAGE} \xrightarrow[\text{Repr}]{} T_{HOUSE}$.

> *Without HSWFC, the user would have to adjust the collapsing probabilities of floor and wall pieces in the same distribution as all other tiles, which greatly reduces controllability.*

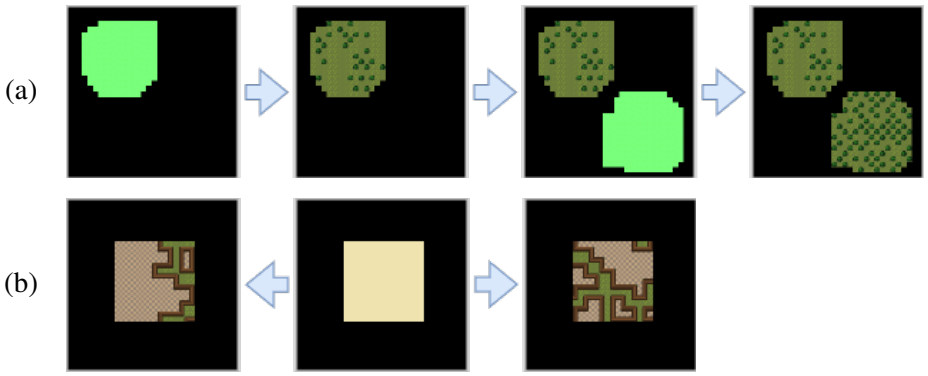


Figure 4.3: Two meta-tile distribution tweaking examples. In (a), two forests are painted with T_{FOREST} with a larger $w_{T_{FOREST},T_{TREE}}$ weight for the second forest. In (b), two houses are painted with T_{HOUSE}, with increased $w_{T_{HOUSE},T_{FLOOR}}$ on the left, and increased $w_{T_{HOUSE},T_{WALL}}$ on the right.

4.4 Collapse path selection

Since HSWFC chooses tiles in a step-wise manner through the hierarchical structure imposed by the meta-tree, there is an opportunity for recording the evolution of a cell based on this path from some meta-tile T_{MT} (often T_{ROOT}, but need not be), up until the current tile. This path through the meta-tree that a cell has walked, known as a 'collapse path', essentially shows which ancestors of the current tile the cell collapsed into in the past. This allows for a new type of tool where, given some meta-tile, cells that have this meta-tile in their collapse path can be selected, at which point an operation can be executed on these cells, such as resetting their state back to that meta-tile, or to T_{ROOT}, or overwriting them with another meta-tile, etc. The ability to regenerate specific regions can even be combined with dynamic probability tweaking (as shown in Section 4.3) to regenerate the areas with a different tile distribution. This could also potentially be used to create selective brushes that operate on queries that make use of collapse paths; we propose to investigate the full range of possibilities that this facility brings in future work.

Example 1: See Figure 4.4. A user wishes to regenerate all cells that were painted over with $T_{VILLAGE}$, in order to have larger houses. To achieve this, the user simply

resets all cells with tiles that had $T_{VILLAGE}$ as ancestor, tweaks the probability of T_{FLOOR} to be more frequent, and then resumes generation.

> *Without HSWFC, a compound operation such as this one would involve tedious manual work, similar to the previous examples.*

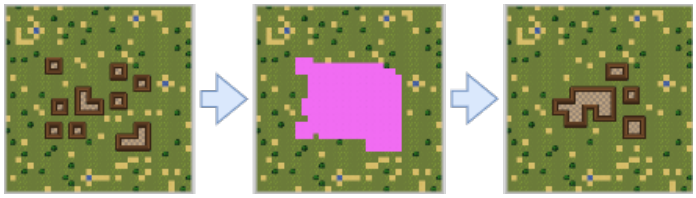


Figure 4.4: The collapse path regeneration example, as described in section 4.4. Note how the regenerated village has larger houses on average.

Chapter 5

Implementation

Two implementations were made of HSWFC, both for the simple tiled model:

- A Python implementation for quick core algorithm iteration
- A Quasar web-app implementation for public deployment and user tests

The implementation of the core algorithm is nearly identical between the two versions, hence it is covered first, after which the most important aspects of both editors are explored in more detail.

An experimental version of the Python implementation was also created to demonstrate the adaptation for the overlapping model, as shown in the HSWFC paper [40].

5.1 Common core HSWFC algorithm

The core implementation of HSWFC was done with contiguous boolean arrays in mind, to keep memory fragmentation minimal and to facilitate vector operations or Single Instruction Multiple Data (SIMD). Therefore, nearly all of the data structures that are used for computations are n-dimensional boolean arrays. Both implementations share this form, although implemented with different libraries. The implementation consists of four distinct parts: the data structures, input processing, the update loop, and the core algorithm itself.

5.1.1 Data structures

At the heart of the implementation lie the data structures used, which are of two types: the data structures for the meta-tree, and the data structures for computations on the grid.

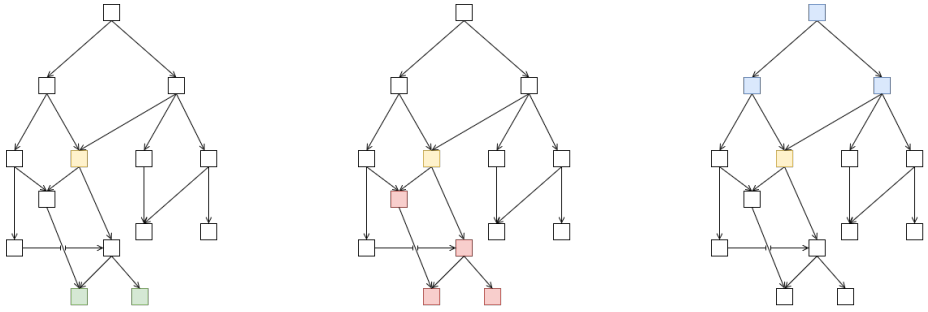
Meta-tree representation

The meta-tree was implemented using a `MetaNode` class that has references to its direct descendants (subtypes) and references to its direct ancestors (archetypes). The meta-tree structure emerges through traversal of these references. The references are represented by a `MetaLink` class, containing information about the edge, such as the from-node, the to-node, and the choosing probability as explained in Sections 3.3.2 and 4.3. The `MetaNode` class defines several methods that are important for convenient retrieval of certain parts of the meta-tree, such as getting the terminal tiles, the subtree (equivalent to the superposition state from Definition 4), or the ancestors (see Figure 5.1). Most of these methods are implemented using recursion.

MetaNode		
-name	:	string
-index	:	int
-archetypes	:	Set<MetaLink>
-subtypes	:	Set<MetaLink>
+leaves()	:	Set<MetaNode>
+subtree()	:	Set<MetaNode>
+ancestors()	:	Set<MetaNode>
+getProbability(nodeTo : MetaNode)	:	float

MetaLink		
-nodeFrom	:	MetaNode
-nodeTo	:	MetaNode
-weight	:	float

Figure 5.1: UML diagram of the MetaNode and MetaLink classes. Getters have been omitted for brevity.



(a) (b) (c)

Figure 5.2: The parts of the meta-tree returned by the methods (a) `leaves`, (b) `subtree` and (c) `ancestors`, for the tile marked in yellow.

Note that the `leaves` method returns the terminal descendants of the meta-tile it was called on. The actual meta-tree itself is built up during input reading, by letting `MetaNodes` reference each other accordingly. To make the implementation efficient, much of the information obtained from the methods such as `subtree` and `ancestors` can be stored into boolean arrays, where each entry represents a `MetaNode` index. The recursive method is only used during input processing in that case, to get the initial values for storage.

Grid representation

Most of the structures used for computations are represented as multidimensional boolean arrays. The most important one of these is the choices array: this array holds the allowed choices for each node index per cell on the grid and therefore has three dimensions, see Figure 5.3. Along with this, there are several arrays with the same dimensions of the grid, that are intended for storing relevant data that is used in various parts of the application, such as the index of the tile that is currently occupying a cell (integer), the entropy of a cell (floating point) or whether a cell was painted on (boolean).

There is one more 3D array with the same structure as the allowed choices, and this is the collapse path array. Instead of having a `True` value for every tile that is still allowed, it just marks the tiles that the cell collapsed before to as `True`, and is maintained accordingly when cells get overwritten, or when their state gets reset by snapshot loading operations.

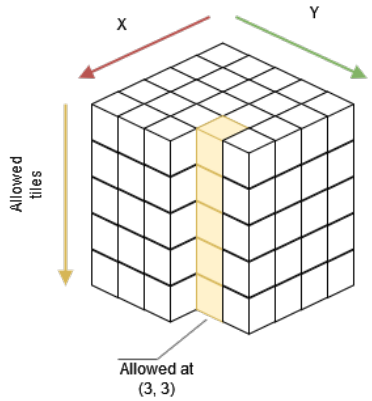


Figure 5.3: The 3D array that holds the allowed choices.

The adjacency constraints are stored as boolean matrices (one for each cardinal direction), with the tile index on both axes. As mentioned above, the subtree, ancestors and leaves per meta-tile are also stored as matrices, structured in the same way as the adjacency matrices.

All of the data structures are stored in a `GridState` object, which represents the current state of the grid. This state can be snapshot and used for undo/redo operations, or saving/loading of the grid.

5.1.2 Input processing

The actual input format can vary per implementation, but the way it is processed is more or less the same. The input must at least provide a sequence of tiles, where each tile has at least the following properties:

- `identifier`, preferably a name
- `image`, the graphical representation of the tile

- `children`, a collection of pairs of tile identifier and weight. If empty, the tile is considered to be terminal.
- `adjacency constraints`:
  - The adjacency constraints are specified per cardinal direction.
  - Adjacency constraints are specified as a collection of pairs of tile identifiers.

Meta-tree construction

From the tile identifiers in the `children` collection, the meta-tree can be built. Every meta-tree starts with T_{ROOT}, which is considered to be the first tile in the sequence. For each tile, a `MetaNode` instance is created first with the tile data, and tile indices are assigned based on the sequence order, with T_{ROOT} always receiving index 0. Then, starting at T_{ROOT}, `MetaLink` instances are created and added to both the `subtypes` field of the `MetaNode` for T_{ROOT}, and the `archetypes` field of `MetaNodes` corresponding to the tile identifiers in the `children` field of T_{ROOT}. This process is then repeated for the subtypes, all the way until the terminal tiles, that have empty `children` fields.

Parsing adjacency constraints

Only the simple case will be described here, where the adjacency constraints are global (for the more advanced and experimental case please refer to Appendix D). In this case, all the adjacency constraints are aggregated into a single set per cardinal direction, and for each of these an NxN boolean matrix is initialized to `False`, where N is equal to the amount of tiles. Then, the sets of adjacency constraints are iterated upon, and for the tile indices of each pair, the appropriate position in the matrix is marked with `True`.

5.1.3 Core algorithm

The core algorithm is driven by two pairs of operations.

Collapsing and propagating

Cells in HSWFC can collapse given that they contain a meta-tile. Upon collapsing a cell, the new tile to choose is determined through weighted random choice among the still allowed direct descendants of the former meta-tile occupying the cell, based on the weights of the edges, or through user choice. It is important to note that users are not limited to the choice of a direct descendant; this only applies to automatic generation (see Section 3.3.2). Once the new tile has been chosen, a number of grid data update operations must take place:

- Setting the new current tile index for the cell
- Restricting the allowed tile choices for the cell by the subtree of the new tile
- Updating the entropy of a cell based on the updated allowed tile choices

- Marking the tile index in the collapse path array

After collapsing, propagation occurs, which is the most computationally heavy operation of the algorithm [10]. In this work, executing a propagation cycle was implemented through vectorized boolean operations. Given origin cell C_O that propagates its constraints over to neighbouring cell C_N, and propagation direction D, we proceed as follows:

1. Take the vector with allowed choices of C_O from the choices array.
2. Determine the tiles that can be adjacent to C_O by selecting the columns (via broadcast *AND* if the library supports it) with the indices of the allowed terminal tiles (see Section 3.3.5) from the adjacency matrix that corresponds to direction D.
3. Combine all of the selected columns by performing element-wise *OR* between the vectors representing them.
4. Use the newly obtained vector to determine what tiles should be disallowed at C_N, by taking the vector with allowed choices of C_N and performing element-wise *AND* between the two vectors.

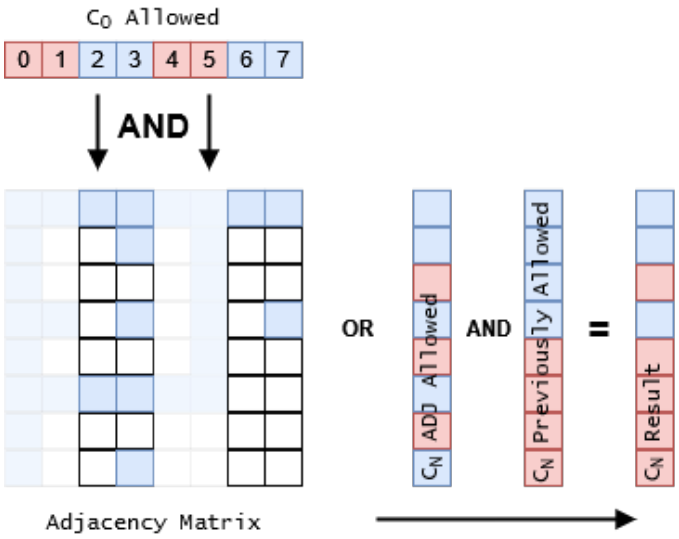


Figure 5.4: A depiction of the operations that occur during a propagation cycle.

This sequence of operations is also depicted in Figure 5.4. The benefit of this approach is twofold: the code footprint is relatively small, because it mostly relies on existing implementations of performing matrix operations. Because of that, it is also quite performant without having to do much additional work, given that the underlying data layout facilitates caching of the values, and given that SIMD support is present in the linear algebra libraries used.

As stated in Section 3.3.5, a final step is required to ensure correct behaviour: meta-tiles that cannot lead to terminal tiles must be removed. This is implemented by performing an element-wise *AND* between the vector obtained at step 4 above and the stored `leaves` of each meta-tile, and then checking whether any tile remains.

Once a final vector of allowed choices is obtained for C_N, it is set on the choices array, and the entropy is recalculated and stored in the entropy matrix.

Uncollapsing and depropagating

The base implementation of uncollapsing and depropagation is nearly identical, with two core differences:

- Cells that are uncollapsed have their allowed choices reset to the stored `subtree` of the new tile they host.
- Depropagation works by solely resetting neighbouring cells to their stored `subtree` like the uncollapse, until there is no difference, after which a propagation occurs from all neighbours that did not change anymore from depropagation, see Algorithm 4.

Note that cells can only uncollapse to tiles that are ancestors of the currently hosted tile, similar to how cells can only collapse to tiles that are descendants of the hosted tile.

5.1.4 Update Loop - fusing automated generation with user interaction

The core loop of the algorithm utilizes a work queue that is constantly probed for new (un)collapse tasks via the update loop of the application. Such a task may have cell locations associated with it, or not; in the latter case, the locations are determined in hindsight through the least entropy heuristic. Depending on the task, the appropriate collapse/uncollapse method is called on the cells that are targeted, which happens sequentially in order to have the most up-to-date information about the grid and its entropy. A task may also have a tile associated with it, in which case the intended cell would collapse/uncollapse (or both, in case of an overwrite) to the given tile, if possible. Typically, tasks that come from user interaction have both cell locations and tile choice specified beforehand. The philosophy used with respect to the tasks is to execute a user request to our best ability: depending on how the newly requested tiles relate to the currently hosted tiles at a cells, the appropriate set of operations will be chosen (collapse/propagate, uncollapse/depropagate, or first uncollapse/depropagate and then collapse/propagate).

Tasks are queued by the application either through user interaction, or through automated generation requests. Tasks that involve user interaction receive priority and are thus placed in the front of the queue, which keeps the application responsive. The tasks that have no predetermined locations come from automated generation, in which case we are forced to propagate immediately after a collapse due to the heterogeneous and non-contiguous nature of cell/tile selection. For brush strokes however, there are two options on how to proceed with propagation of the collapsed cells:

- Wait with the propagation until all cells have been collapsed.
- Propagate immediately after collapsing a cell.

The first method can save a lot of performance, because it avoids propagation in the area covered by the brush, which is very noticeable for large brush strokes. However, it also allows incorrect adjacency behaviour for the placed tiles by allowing all tiles in the brush stroke to be placed, which is especially relevant for tiles that do not allow self-adjacency, such as T_{TREE} (see Appendix B). In addition, if a contradiction is reached due to one of the cell collapses, it is impossible to preserve the propagation effects of the other cells, essentially requiring the application to undo the entire brush stroke. This implementation is used in the Python prototype for experimental purposes.

The current web-app uses a hybrid approach: for regular collapsing, the first method is used, just like in the Python app. For overwriting (which involves an uncollapse to T_{ROOT} followed by a collapse to the designated tile), the cells are propagated separately. This was done because it was much more likely for an overwrite brush stroke to partially contradict, and because the propagation waves were usually considerably smaller than the propagation waves that resulted from collapse-painting cells on an "empty" (filled with T_{ROOT}) grid, so just rolling back the parts that contradicted and executing the rest of the brush stroke as usual would lead to better satisfaction of the user request.

Uncollapsing

Uncollapsing can only happen via user requests, and depropagation only happens after all the cells from a brush stroke have been uncollapsed. This is done because the current depropagation algorithm would otherwise constantly undo its own work after propagating the marked cells again (see Section 3.3.5 and Algorithm 4); namely the depropagation algorithm is "rather generous" with how many cells it resets, and the propagation then restricts these cells again, causing the next cell of the brush stroke to reset the cells that have just been propagated over by the previous one, and so on.

Choosing the next cell

The choice of next cell for automated generation does not need to only be influenced by the minimum entropy heuristic. In fact, to facilitate user interaction, it is useful to combine minimum entropy together with spatial priority for locations that have been painted on together with their propagated surroundings. That way, when the user paints something, the generator will try to fully solve the painted area first. This information can be kept track of on a separate boolean matrix. Another experimental idea involved using the minimum amount of hops from T_{ROOT} to any other meta-tile as a kind of hierarchical depth, and prioritize cells that hosted tiles with a low depth number, which makes the automated generation prioritize cells that still host high level tiles, applying more of a breadth-first solving methodology rather than a depth-first one.

5.2 Python pygame editor

The first prototypical editor was made to be able to quickly evaluate the inner workings of the HSWFC algorithm. It is built in the `pygame`[1] engine, making use of hardware texture splatting to create the entire UI, which makes it very fast and responsive. For instance, when the visual representation of the grid needs to be updated, we are just coloring pixels on a texture and then scaling it up for display.

Interaction is captured through specific keybinds, and by tracking the mouse position on the texture that displays the current output. Information that requires spatial highlighting on the grid is shown by overlaying this texture with another transparent texture of the same dimensions, e.g. to show where the cursor and how large the brush currently is, to mark cells visually while debugging methods, to show cell influences, and to visualize recorded propagation waves.

The algorithm was implemented mostly using `numpy`, which had the data structures and operations necessary for the implementation described in Section 5.1, thanks to the `ndarray` class. Furthermore, `numpy` harnesses SIMD for its vector/matrix operations, which gives a good performance boost when using these arrays for computations.

5.2.1 Debugging Suite

In order to gain quick and deep insights about how HSWFC operates, an extensive debugging suite was built into the editor. This did sacrifice performance somewhat, but performance was not a core objective yet at this stage of research.

Extensive Logging

A variety of information is sent to the terminal output, among other things:

- Pretty printed matrices for adjacency constraints and other boolean matrices
- Other parts of the input: the meta-tree, tiles
- Debug information about a cell on middle-clicking, consisting of the hosted tile, allowed tile choices, current entropy value, tiles influenced, and tiles influenced by.

Cell influences

As explained in Section 3.3.5, recording cell influences is rather inefficient with regards to data storage and also adds additional storing operations into the propagation loop which is called extensively, reducing performance. In addition, a large amount of bookkeeping is required when uncollapsing/resetting of cells is also taken into account. However, having a basic version of storing this information is extremely useful for debugging, as it shows exactly what the consequences are of collapsing some cell to a tile. The Python editor stores this information both ways, for some cell C, we store:

[1] `https://www.pygame.org/`

- All cells that were influenced by C through propagation from its collapse (blinking (red))
- All cells that influenced C somehow via their own propagation waves (blinking (green))

Note that it is possible for a cell to be influenced by and to have influenced C, upon which it becomes yellow. A cell on the grid can be selected for displaying this information by clicking it with the middle mouse button. See Figure 5.5 for an example of this.

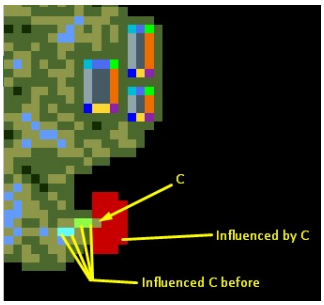


Figure 5.5: A screenshot of the cell influence visualization feature. Cells influenced by the selected cell C are marked red; the cells that have influenced C in the past are marked green.

Tile choices tooltip

A hover tooltip that shows the available tile choices at a cell can be summoned with a hotkey. This debugging feature also synergizes with the cell influences feature: when a cell C is selected, hovering over the areas that C influenced (the red areas) will show at each cell which tiles were excluded at that point in the propagation wave, which is very useful for debugging tileset constraints and propagation. See Figure 5.6.

Figure 5.6: A screenshot of the hover tooltip, showing allowed tiles. Since a tile was selected and we are hovering over its influence area, we can see which tiles were excluded at that point (displayed as a marking on the tile across the full height of the tooltip).

Entropy panel

Next to the grid output, another panel is shown that displays the entropy of the grid. Displaying the entropy in this manner is very useful for debugging cell states, in particular for debugging the consistency of the uncollapse-depropagation algorithm, as it gives an immediate overview of the cell states. See Figure 5.7.

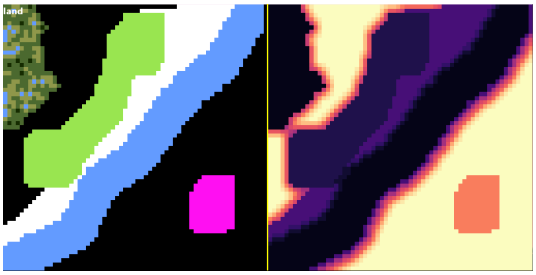
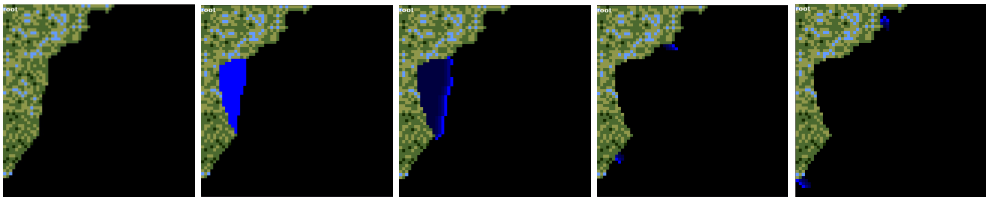


Figure 5.7: A screenshot showcasing the entropy panel on the right for the grid on the left. Bright colors indicate high entropy, while dark colors indicate low entropy. Terminal tiles have the lowest entropy thus appear black, by virtue of being fully determined.

(De)propagation wave recording

In order to effectively develop the depropagation algorithm, it was necessary to understand how the propagation and depropagation waves travelled over the grid. Therefore, a recording feature is implemented, that can be toggled. It will show animated propagation waves in red and depropagation waves in blue. We opted for recording instead of showing it real-time, because it was easier to process the animation, it wouldn't require slowing the algorithm and lastly because it would allow for viewing the propagation wave multiple times. To make it extra clear where the wave is, the previously colored cells do not turn off their highlighting immediately, but rather maintain some opacity, creating a trail effect.

Thanks to this feature the issue that was described in Section 5.1.4 was found, which first manifested itself as a tremendously slow uncollapsing operation. From the recording it became clear that doing uncollapsing per cell would constantly undo the work done by the previous cell. In addition, this feature also allowed studying the behaviour of depropagation in general, and finding that the current algorithm is not optimal. See Figure 5.8.



(a) (b) (c) (d) (e)

Figure 5.8: A series of screenshots that show a depropagation wave recording that is being played back. (a) shows the former state of the grid, and (b) shows the area that was uncollapsed to T_{ROOT}.

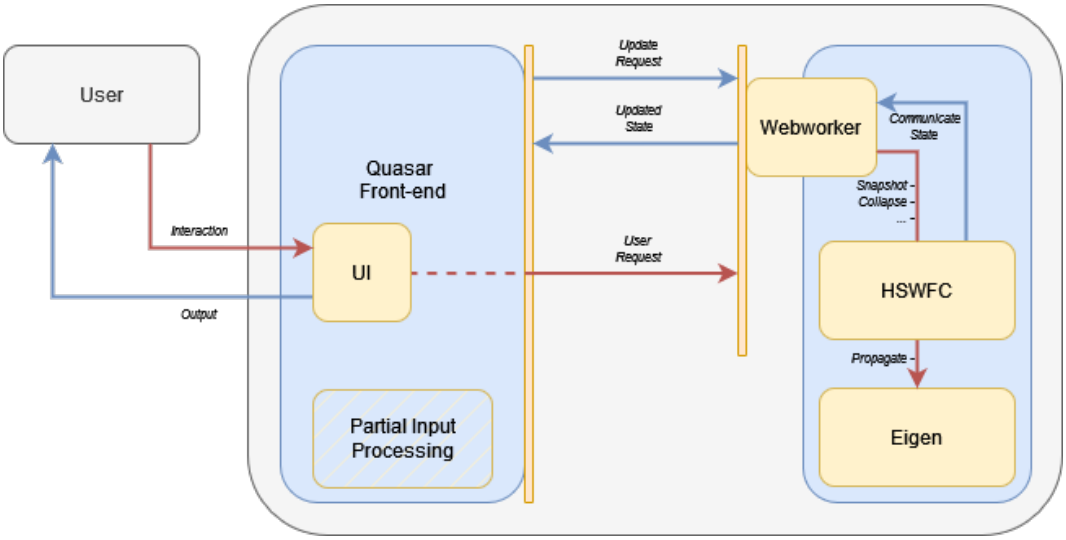


Figure 5.9: A system overview of the web editor.

5.3 Quasar web editor

In this section we will describe the implementation of the Quasar-based web environment editor. Aside from an environment editor, an input editor was also conceived in order to allow for the quick construction and iteration of hierarchical tilesets for HSWFC, but building this involved developing some highly experimental algorithms and features which were only tangentially related to the research question but could be quite significant for future work, hence this portion of the implementation can be found in Appendix D.

Before diving into the components that make up the environment editor, it would be nice to have an overview of the whole system, as visually shown in Figure 5.9. In essence, we have a front-end that is driven by Quasar, aided by a web-worker which allows for background computations that rely on webassembly to speed up the propagation of the algorithm. Dealing with data on the Javascript side is done using `mathjs`[2], which is a mathematics/computation and matrix algebra npm module, while the webassembly part uses `Eigen`[3], a highly efficient C++ library for doing numerical computations that involve matrices and tensors. These choices were made to mirror the choice of using `numpy`[4] in the Python prototype, which allowed for data modifications using bit masks and bit arrays, and indexing schemes that are easy to understand for general purpose data retrieval. Both the front-end and back-end make use of `mathjs` in order to handle the data that HSWFC produces. The web-worker has a message handler in the front, which redirects calls to the HSWFC algorithm, which resides in a separate module. This module makes calls to `Eigen` for propagations. Note that the front-end currently also does a part of the input processing, but this is planned to be moved to the web-worker as well.

[2] `https://mathjs.org/`
[3] `https://eigen.tuxfamily.org/`
[4] `https://numpy.org/`

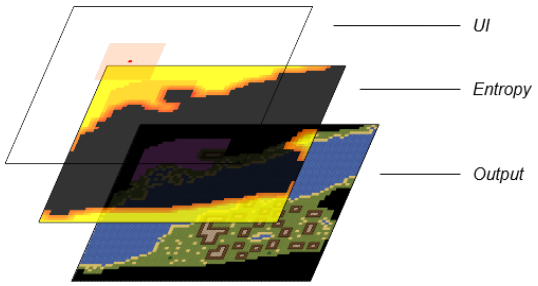


Figure 5.10: How the canvases are layered on top of each other. User interactions are caught in the top layer, which also handles highlighting.

In the following sections we solely discuss the Quasar environment editor, with the Quasar input editor being discussed in Appendix D. The specifics of the optimizations implemented such as the webworker, the use of webassembly and `Eigen`, and the solution we use for finding the minimum entropy on a grid quickly can be found in Appendix E.

5.3.1 The environment editor

This is the web-app equivalent of the Python-based editor (see Figure 5.11). It mostly has similar functionality, made more accessible due to the usage of Quasar components. Most of the UI elements have tooltips, and everything has a polished animated finish for most interactions that comes from Quasar itself.

Canvas

HTML5 canvases are an important part of what makes the web app tick. Most of the image data that involves tiles or the grid makes heavy use of canvases. For instance, meta-tile images are generated from a color using a canvas, and any element that the user can draw on is powered by multiple canvases that work together to provide user feedback and accept user input. Much like the textures in `pygame`, images can be projected onto canvases, and their internal image data can be used by other canvases in order to facilitate functionality such as image downloads, or image copying for previewing purposes in e.g. snapshots.

In the middle of the UI resides the canvas that the user draws on (see Figure 5.10). This canvas is covered by two other canvases, one for showing an overlay of the entropy if enabled, and another one for giving user feedback. The user feedback includes:

- Location and size of brush (red)
- Cells that are about to be drawn (yellow)
- Cells that are queued to be processed by the algorithm (pulsating blue)
- Cells that have contradicted (pulsating red)

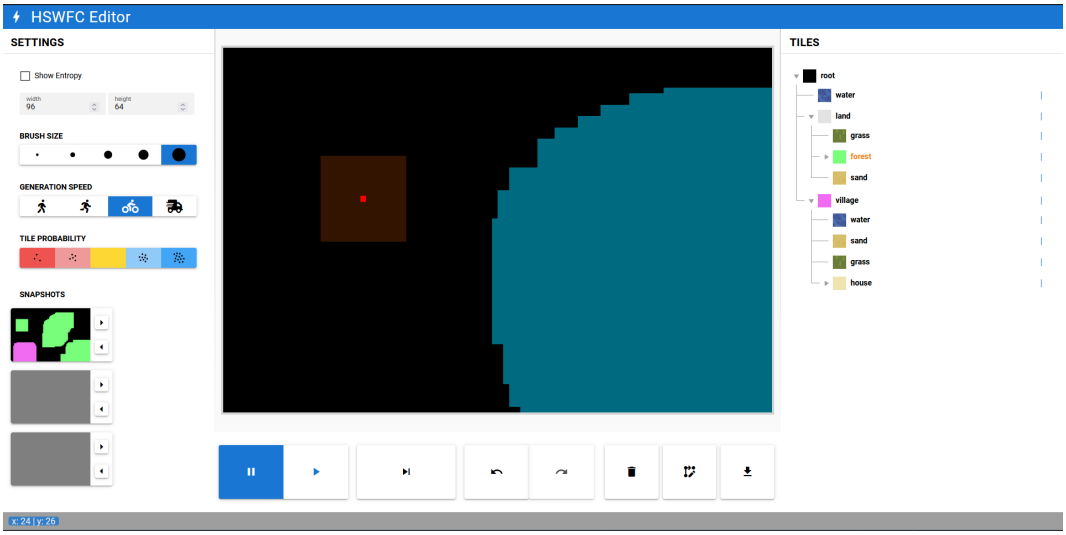


Figure 5.11: The interface used for the environment editor during the user study. In the middle we see the canvas, with some cells that are enqueued to be processed. At the right, the hierarchy of tiles that can be selected for painting are shown as a tree. Below the canvas we find some controls for the generation process. To the left, we see settings such as 'brush size' and 'generation speed', and at the bottom left we can see the snapshotting feature.

This highlighting helps the user a lot with understanding what is going to happen, especially during large painting operations. The contradiction highlight only shows when auto-undo on contradiction is disabled, an implemented setting not available on the Graphical User Interface (GUI) yet.

The actual pixel-size of the bottom canvas is determined by the size of the tiles and the dimensions of the HSWFC grid. Virtual cells are created by rounding the mouse position on the canvas. The mouse location on the canvas is obtained via HTML element offsets, and then rounded appropriately to ensure that the pointer properly translates its coordinates to indices of the cell under it.

Meta-tree

The meta-tree is represented by a Quasar tree component[5], which can also be seen in Figure 5.11. Each tree node in the component corresponds to a tile T_B along with its incoming edge from direct ancestor T_A, so that the user may tweak the weight w_{T_A,T_B} by simply selecting T_B. For T_{ROOT} there is no incoming edge, thus no weight to tweak. Mapping a DAG to a tree requires duplication of the nodes, hence the same tiles may appear in multiple places depending on the structure of the meta-tree, even though they are in fact the same tile. This makes sense for tweaking the weights, as such node duplicates will have different incoming edges with different weights. A user may select a tile in this tree to paint on the canvas with, and expand/contract parts of the tree to show/hide tiles as they wish.

[5] https://quasar.dev/vue-components/tree/

Controls

Under the canvas (see Figure 5.11) are buttons for: starting/pausing auto-generation, stepping, undo/redo, clearing the grid, resetting an area via collapse path editing and downloading the current canvas as an image. Several of these controls interact with other areas of the UI: the amount of cells that are queued in a step is equivalent to the square of the generation speed setting in order to allow controlling the step size, and the collapse path editing button requires selecting a node in the tile tree.

Options

On the left are options for tweaking properties such as the brush size, the generation speed, or the canvas size. For more information about the workings of the generation speed option, see Section E.0.1.

Snapshots

The bottom left contains some widgets that allow the user to snapshot what is currently on the grid. Clicking the arrows will either save or load the snapshot hosted by the widget, which shows as an image that is copied from the canvas on save. This image can be enlarged and brought into focus by clicking on it. Snapshots are useful for generating variants of some meta-tile configuration, by returning to the snapshot until the result is satisfactory.

There was an additional feature that made use of snapshotting that got scrapped from the implementation for the user study due to its added complexity and distraction from the main point. It allowed the user to click a button and generate a different continuation of what is currently on the grid for each snapshot slot, providing multiple "futures" to choose from.

Note that the undo/redo stack is implemented using the same mechanism as snapshots, except they get stored in a stack and are managed accordingly by the undo/redo buttons.

Info Bar

The info bar at the bottom is able to show some useful debugging information, such as the position of the mouse cursor on the grid, which tiles are still allowed at the grid cell under the mouse cursor, the entropy value, and which tile is currently occupying that cell.

Chapter 6

Evaluation

In order to assess whether the editing facilities described in Chapter 4 truly reduce cognitive load when employed in an editor, a user study was conducted. For this work, only a single user test was carried out.

6.1 Method

The core idea, as mentioned in the introduction, is to conduct a blind A/B test with the hypothesis that there will be a significant difference in the measurements of cognitive load between group A and group B, favorable for the group that will be using the editor that has the HSWFC feature set. Naturally, that means that another version of the editor is required that does not have this feature set. It is possible to get the stock WFC behaviour via HSWFC, namely by not having any other meta-tile than T_{ROOT} in the meta-tree. This allowed for building an alternative editor that is as close as possible to the HSWFC editor, but with a feature set limited by capabilities of stock WFC[1,2].

In the experiment, we wish to measure cognitive load, and figure out how the user engages with the tools, so we need to give the user a task, with some rationale:

- The task should be complex enough so that a user from group B can potentially use the full suite of features described in Chapter 4 if they would want to.
- The task should not be too lengthy, in order to maximize user participation.
- Some parts of the task should be left open to interpretation, in order to allow the user to explore the editor on their own terms.

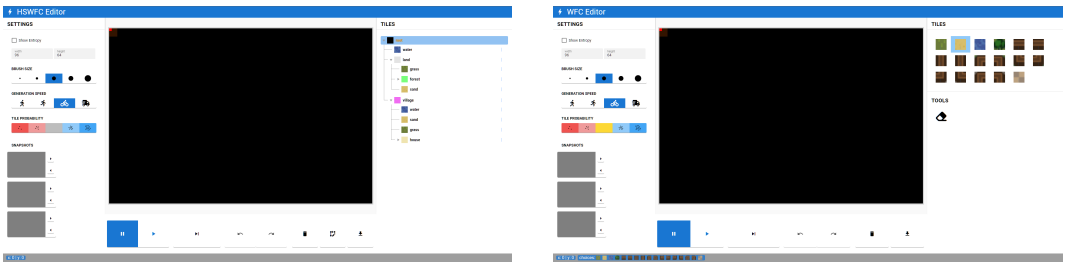


Figure 6.1: On the left the HSWFC editor is shown for group B, on the right the modified editor that is powered by an imitation of stock WFC for group A. Note how T_{ROOT} is portrayed as an eraser in the group A editor.

[1] Editor A: https://archer6621.github.io/hswfc-editor-a/
[2] Editor B: https://archer6621.github.io/hswfc-editor-b/

ID	Question
Q1	I felt that the editor was able to capture my intent
Q1	I felt like I had the freedom to tweak things easily
Q3	I understood why certain tiles could not be placed in certain locations
HSWFC Q1	I found using the hierarchy for selecting a tile to paint with to be intuitive
HSWFC Q2	Using a single situation-dependent brush for painting and erasing felt intuitive
HSWFC Q3	Painting with meta tiles gave me the results I expected
HSWFC Q4	Adjusting the meta-tile probabilities had the results I expected
HSWFC Q5	I found using the regeneration tool useful for creating variations of my design

Table 6.1: Additional questions asked to the groups about the editor. The first three questions were asked to both groups, the remaining questions only to group B.

In the end, the task required the user to create a little world with two villages adjacent to a body of water, some forest with some constraints and properties. They were given the role of "level designer" for some game under development, receiving a task from the "producer" to create a game world, in order to instigate a sense of purpose. After creating the world, they were free to make some variants of it according to their own creative touch, in order to explore the editor a bit more and get a more solidified opinion on it.

For this experiment, the tileset presented in Appendix B was used, as it was appropriately detailed for the task described above.

The division of groups was as follows:

Group A: uses the WFC-imitating editor.

Group B: uses the editor with the full HSWFC feature set described in Chapter 4.

After being done with the tasks, the users had to fill in a modified NASA TLX survey on a 7-point scale, which is used to measure the overall cognitive load. The survey was modified, suppressing the 'physical effort' rubric, as it was deemed irrelevant for the task. In hindsight, this rubric could have been kept to measure strain from clicking too much.

Aside from just measuring cognitive load, some additional questions were asked that focused a bit more on the user experience of the editor itself for both editors (see Table 6.1), and in the case of the HSWFC editor, how intuitive the new editor facilities were and whether there were significant differences between their usability. The environments that users sketched were collected as well, mostly to give users that participated a sense of objective, and also to find qualitative differences between environments that were created among the two groups. For details regarding the survey, please see Appendix C.

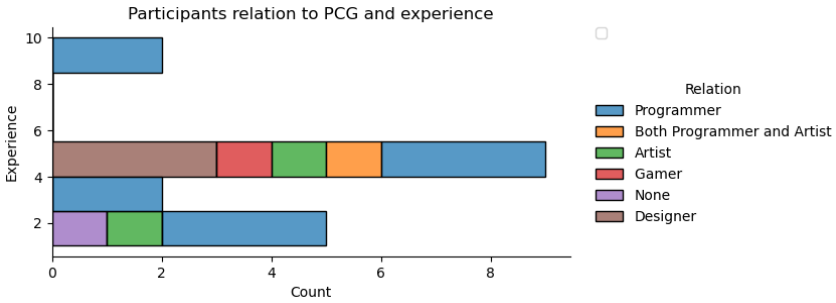


Figure 6.2: The distribution of the participants' relation to PCG and experience.

6.2 Results

A total of 9 survey responses were collected per group. While the web app provided mobile support, all participants used a desktop. Before the user study starts, some questions were asked to gain some insights into the relation and experience of the participants with PCG, which can be viewed in Figure 6.2. Many of the participants were programmers, though the ratio of programmers and artists/designers was approximately equal between the two groups. Regarding demographics, we could empirically establish from the names and from knowing many of the participants that they were predominantly male, with age varying between 20 to 50 years old.

The resulting aggregation of the scores for the NASA TLX rubrics can be inspected in Figure 6.3. Along with that, the statistical significance of these results were evaluated using Student's t-test [41] and can be found in Table 6.2, obtained using the independent t-test implemented by `scipy`[3], which tests for the null hypothesis; whether the 2 independent collections of samples (in our case, group A and group B) have identical expected values. Rejection of this hypothesis means that the two groups are potentially different. All the

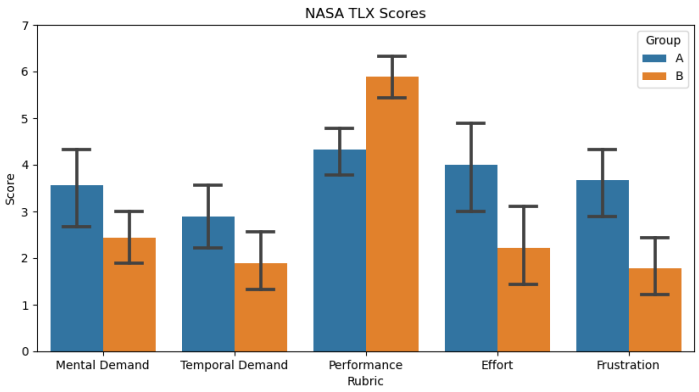


Figure 6.3: The NASA TLX results aggregated into a bar plot.

[3] https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ttest_ind.html

Rubric	t(16) statistic	p value	Significant
Mental Demand	2.08514	0.05343	Doubt
Temporal Demand	2.01246	0.06132	Doubt
Performance	-4.00000	0.00103	Yes
Effort	2.60412	0.01918	Yes
Frustration	3.45218	0.00328	Yes

Table 6.2: The results of performing the independent t-test.

metrics with p-values lower than 0.05 can be assumed to be significantly different between the two groups: these are 'Performance', 'Effort' and 'Frustration'.

Overall, the NASA TLX results show that group B felt significantly more successful in their performance, had to exert significantly less effort, and experienced significantly less frustration while using the HSWFC driven editor. Temporal demand and mental demand show hints of being improved as well, but the pool of participants in the user study was not large enough to be conclusive about the significance of these differences.

Aside from NASA TLX, there were also some questions that were more specifically aimed at the usability of the two editors. The results for these can be found in Figure 6.4, along with statistical significance of the differences in Table 6.3. It appears that none of these questions had significant differences in their answers among the two groups. Interestingly, even though the HSWFC driven editor did not specifically focus on making it easier to understand why certain tiles can be placed, Q3 still shows a difference that is bordering on significant. In addition, Q2 scored exceptionally high for the HSWFC editor, with a low variance, which means that participants in group B almost unanimously had the feeling that they were able to tweak their creation easily.

As for the HSWFC specific questions, it seems that `HSWFC Q2` had the most divided opinions, pointing out that not everyone finds using meta-tiles for erasing intuitive. People were least divided over `HSWFC Q3` and `HSWFC Q5`. `HSWFC Q1` had the highest average score, meaning that participants found the hierarchy to be quite intuitive to use. Participants were least enthusiastic about the probability tweaking, potentially because of the somewhat inconvenient UI for it. Overall, the scores were favorable.

Analyzing the images of the first task that users created in Figure 6.5, a distinct difference appears to be that users from group *A* were much less likely to draw a river than users from group *B*, even though a river was not explicitly mentioned in the task (see Appendix C).

Question	t(16) statistic	p value	Significant
Q1	-1.20605	0.24533	Unlikely
Q2	-1.76505	0.09663	Doubt
Q3	-1.91156	0.07401	Doubt

Table 6.3: The results of performing the independent t-test for the common editor questions.

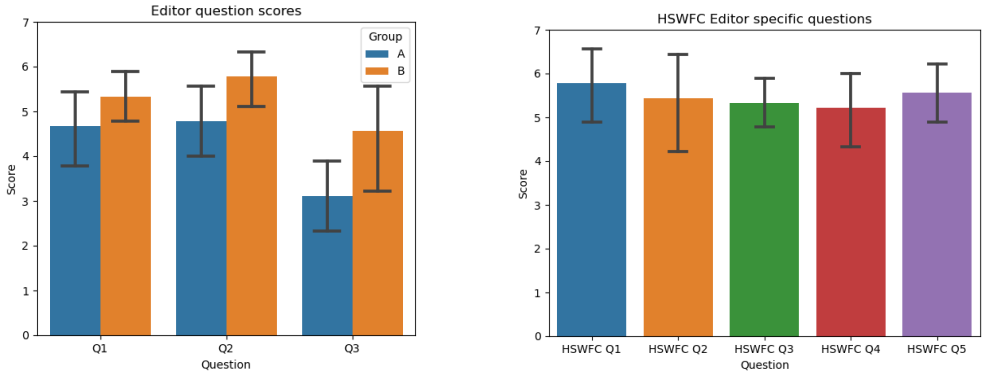


Figure 6.4: The resulting scores for the editor questions aggregated into two bar plots. On the left are common editor questions asked to both groups, on the right are questions that were specifically asked to group B, tailored at the HSWFC feature set.

Another visible difference is that the users from group *A* preferred to draw relatively few houses, that were much larger and less detailed on average, while group *B* was able to paint more houses per village in general.

Overall, group *B* was able to get closer to the objectives that were given in the task than group *A*.

Finally, going over some of the open feedback that users have given, these were the most common themes and most interesting points raised:

A/B: User feedback for when something goes wrong would be nice, currently it seems as if just nothing happens because of auto-undoing.

A/B: Users found the snapshotting feature useful.

A: Trees were problematic to place due to them not being allowed to be adjacent.

A: Making a building with an irregular shape was quite hard.

B: The regeneration tool targets all tiles, it would be nicer if one could use the brush to mark specific regions instead.

B: The root tile acting as eraser was not intuitive.

B: Probability tweaking was a bit arduous.

Interestingly, group *A* was more likely to leave open feedback than group *B*, though most of the group *A* feedback was about how hard it was to place trees.

Figure 6.5: User creations from the first task. Top batch corresponds to group A, the bottom one to group B.

6.3 Discussion

The most important results are the NASA TLX survey outcomes and their significance, because they directly relate to the research question that we are trying to answer in this chapter:

Question: *How does the experienced amount of cognitive load differ between an editor driven by HSWFC with the novel facilities and an editor driven by stock WFC without these facilities?*

With the results presented in the previous section, we can now answer this question. These results show that an editor driven by HSWFC with the facilities presented in Chapter 4 generates significantly less cognitive load than an editor driven by stock WFC without these facilities. It seems that a lot of the excess cognitive load generated by the stock WFC editor that group *A* experienced comes from frustration and effort. A caveat here is that some of the excess effort and frustration may be exacerbated in the stock WFC editor due to the fact that at the time of the user study, the editor did not try to collapse as much of a brush stroke as possible yet, and instead rejected the entire stroke if it caused a contradiction. This affected both editors, but the lack of meta-tiles might have made this more apparent in group *A*, which explains the numerous complaints about the trees.

There were also differences in the mean values of the temporal and mental demand metrics of the NASA TLX survey between the two groups, with group A having higher mean values than group B, though the significance of these differences is doubtful, but only barely upon inspecting the significance values in Table 6.2. A study on the correlation between the NASA TLX dimensions shows that temporal and mental demand are strongly positively correlated to effort and frustration, so we can assume that the new editing facilities have a positive effect on temporal and mental demand as well [42]. Noteworthy to mention is that there was no time limit on the tasks to induce temporal demand, though we did give an estimated amount of time it would take in the pitch (around 15 minutes), hence temporal demand may still be self-imposed from users not willing to spend too much time on the study.

The fact that there was such a large difference in the performance rubric (see Figure 6.3) also reflected in the fact that users were able to get closer to the objective in group *B* than in group *A*. It therefore seems that editor *B* made users be more successful at accomplishing their tasks. One interesting result was the lack of rivers in group *A*. Intuitively, a river made sense for the request we had in the task. Drawing a river is not more difficult between the two editor versions, so perhaps participants were distracted by other usability issues, or were more conservative with their brush strokes in group *A* because of the effort involved with fixing things again after overwriting.

The answers to the questions that were not part of the NASA TLX survey tend to show that users are in favor of the new editing facilities that HSWFC is able to bring to an editor, especially for hierarchical tilesets. There was a fear that presenting the tileset in this manner may be counter-intuitive, but the high score for question `HSWFC Q1` potentially debunks that. Since the enhanced feature set is the core difference between the two editors towards the users, we can definitely say that the feature set established in Chapter 4 is a good starting point for building an environment editor that is powered by HSWFC. The recommendation would be to further advance the feature set and implement some of the open feedback that was given with regards to its usability.

An important observation from the open feedback in both groups that was that a substantial amount of the users seem to prefer a destructive approach over a conforming approach. This can also be gathered from `Q3`, as the users that gave a low score here for HSWFC were the users that (according to the open feedback) discovered T_{TREE} and were not able to effectively place it. In other words, it is likely that users care more about exerting their will, rather than having the constraints be fulfilled. There are two distinct areas relating to this that are worth looking into for future testing and implementation:

- Local adjacency overriding: this is already possible to some extent using the implementation presented in Section D.0.2, but its intuitiveness has not been verified through user studies yet. Potentially, users may prefer doing this locally/spatially, rather than through the meta-tree abstraction.
- Destructive overwriting, as briefly described in Section 4.2. This is a complex operation that might have unintended effects, as it may be necessary to remove surrounding

tiles to exert the user's will.

Finally, one limitation of this user study is that the participants do not resemble the target audience for an environment editor such as this one very well. This type of application is mostly aimed at environment designers for a variety of use cases, the largest one being environment/level design in game development. Among the participants were merely three designers, and it is not clear whether they are environment/level designers. Even though attempts were made at reaching this target audience for the user study by contacting game developer companies, posting on slack channels and mailing lists, participation was unfortunately very sparse. Several lessons were learnt from this user study for conducting future ones, that may help with performing a user study with a more fitting audience:

- Contact potential participants directly as much as possible alongside posting numerous pitches and announcements on channels frequented by the target audience and contacting numerous companies where such an audience may be employed.
- Put much effort into the onboarding pitch, make sure that it comes across as a very low investment with high reward. Test it on friends to see whether it catches their attention immediately, prior to posting or broadcasting it further.
- Keep the study concise; leave a tutorial out of the survey as it may scare off participants or give them the impression that the user study is too lengthy. Instead, delegate this to the UI via tool-tips, to a video, or to a separate "manual" document.
- Keep the algorithm description in the survey as simple and relatable as possible (e.g. anecdotes), while also keeping it concise; this information is not supposed to take the spotlight, and may even be left out if it does not add any value to the user study.
- Provide a clear deadline for the user study, otherwise potential participants may leave it on the side and forget about it. Also make sure to send at least one reminder close to the deadline.

Chapter 7

Conclusions and future work

This chapter gives an overview of the project's contributions, concluding the thesis and providing several interesting directions and recommendations for future work.

7.1 Conclusion

In this work we wish to answer the following question: ***"How can we reduce the cognitive effort required for working with a mixed-initiative environment editor powered by an algorithm that is based on WFC?"***. This question can be answered in a piece-wise manner, by providing answers to the sub-questions;

Q: *"How can we enhance the WFC algorithm with hierarchical semantic abstractions?"*

We have shown that we can do this through the introduction of a new type of tile, called a meta-tile, which, as a result of being a tile, has a representation on the grid and has (inferred) adjacency constraints to other tiles. A meta-tile relates to other tiles by representing the semantic traits they may carry, and the set of interrelated meta-tiles together form a hierarchical structure that we call the meta-tree. The new generative algorithm that is proposed results from the fusing of WFC with hierarchical semantics and is called Hierarchical Semantic Wave Function Collapse, or in short, HSWFC.

Q: *"What novel features does HSWFC enable that may reduce cognitive load?"*

Harnessing the power of HSWFC, we have shown how it can expand the editing features of a tile editor, with a focus on reducing cognitive load:

- Painting with semantic traits through meta-tiles (Section 4.1).
- Hierarchical tile overwriting (Section 4.2).
- Distribution tweaking of the tiles that cells hosting meta-tiles can collapse into (Section 4.3).
- Editing facilities that operate on collapse paths of cells, such as targetted overwriting (Section 4.4).
- Scoped adjacency constraint specification with overrides and inheritance (Appendix D).

And perhaps much more, as these are merely initial findings of the potential that HSWFC unlocks for mixed-initiative editing.

Q: *"What kind of other editing facilities can we implement that may reduce cognitive effort required?"*

We have shown that novelty did not only have to come from new editing facilities directly made possible by HSWFC. Instead, features such as snapshotting, an undo/redo stack (for both see Section 5.3.1), clear highlighting (Section 5.3.1), and an investigation into clarifying "uncollapse" operations (Section 3.3.5) have shown to be valuable additions. In many cases, these features also exhibited synergy with the HSWFC specific features, enabling some of the new workflows that HSWFC editing facilities bring, such as hierarchical tile overwriting, use of collapse paths for overwrite operations, generating several variants from a snapshotted meta-tile template, and more.

Q: *"How does the experienced amount of cognitive load differ between an editor with the novel features enabled by HSWFC and an editor driven by stock WFC without these features?"*

When combining the new editing facilities with general convenience features unrelated to HSWFC, we have shown that the implemented combined package can significantly reduce the overall cognitive load that users experience while building a world in a user-friendly 2D tile editor that employs them (see Chapter 6). In particular, the performance of users was visibly better in the produced outputs of the HSWFC-driven editor when compared to an editor using stock WFC (without the editing facilities seen in Chapter 4). This was also backed up by the performance scores that users gave themselves in the NASA TLX survey, while keeping in mind that some bias was introduced by certain choices in algorithm implementation that hindered the stock WFC editor more than the HSWFC editor.

The method applied in this work as described in Section 1.2 has proven to be effective, as building a lightweight editor for rapid algorithm iteration allowed for combining it with the implementation of prototypical versions of the editing facilities, which helped assessing their potential early on in the process. Furthermore, the choice of web-app for the user-friendly version of the editor saved a lot of effort spent on unrelated subjects such as UI implementation, as the frameworks and tools available in the web domain are highly mature and convenient to use. The user study, while effective, required some more polish with regards to survey design, pitching and dissemination, and making sure that the algorithmic implementation is mature enough to prevent obvious biases from occurring between the two versions. Several points have been made in Section 6.3 for improving this methodology for future work involving user studies.

7.2 Contributions

This work provides the following contributions:

- The HSWFC algorithm, a modified version of WFC that is able to express semantics in a hierarchical fashion (Chapter 3).
- A blueprint for an efficient implementation of HSWFC, using vectorized bit operations, (Chapter 5).
- A Python-based tile editor driven by HSWFC with a rich debugging suite for rapid iteration, development and experimentation on the algorithm, with the source published and documented[1] (Section 5.2).
- A web-based tile editor driven by HSWFC for conducting public experiments that involve usability and usefulness, efficiently implemented using state of the art technology, with the source published[2], and a live demo available[3], together with pre-made tilesets in order to be able to get started quickly with exploring the tool[4] (Section 5.3).
- A modification of the C++ codebase of `eigen-js` that exposes parts of `Eigen` intended for vectorized boolean operations using tensors, with published source[5] (Appendix E).
- A user study that verifies the usefulness of the HSWFC-enabled features in a tile editor for environment design, along with rigorous recommendations on how to carry out such a user study for future iterations (Chapter 6).

The web-based editor and the `eigen-js` fork will both eventually receive a refactor and documentation as well, so that future research and extensions can be carried out without too much effort.

7.3 Future work

HSWFC has opened many doors for future work. We will consider four core directions: input handling, user interaction, algorithm, and constraints.

7.3.1 Input Handling

The experimental input editor shown in Appendix D was not evaluated yet. Performing further user studies to evaluate the usefulness of this input specification workflow is another step towards making both WFC and HSWFC more accessible for projects that require controllable PCG.

Other than that, it is also worth to lean more towards experimenting with the hierarchical inputs, in order to find out what kind of hierarchical representation suits designers best. This

[1] https://github.com/Archer6621/HSWFC-editor-pygame
[2] https://github.com/Archer6621/HSWFC-Editor
[3] https://archer6621.github.io/hswfc-editor-dev/
[4] https://drive.google.com/drive/folders/10MOZ0KVQR_QdlP9mcx1YNXkd-tyDSA7A?usp=sharing
[5] https://github.com/Archer6621/eigen-js

can be done by trying either many more hierarchical tilesets or a very extensive one, and then evaluating the designer experience.

As to input specification, some geographical data also consists of zones and layers. It can be interesting to derive a meta-tree from this input source, which allows one to use chunks of this geographic data as snapshots that can be manipulated accordingly then. Moreover, the rich representation of semantics that HSWFC can provide could prove to be interesting in combination with machine learning. For instance, estimating a meta-tree and thus deriving a HSWFC tileset from an input image would vastly reduce the effort required to build such a tileset.

7.3.2 User Interaction

One interesting addition to the feature set would be the ability to use complex heterogeneous brushes that paint pre-made templates of meta-tiles, e.g. a house with its walls in a certain shape and a door in a specific spot, or a template for a park that has the same spatial division but generates differently due to its meta-tiles. Maybe these templates can generate under certain meta-tiles as well, or in particular under the meta-tile that has all of the tiles in the template as direct descendants. One recent work is already exploring this facet of stock WFC; we believe that HSWFC can be used to enrich the idea further [15].

Additionally, it would be useful to investigate having boundary conditions for brush strokes with meta-tiles; currently, if some semantic concept is painted with a meta-tile brush, it might be possible that the semantics will extend uncontrollably outside of the brush stroke (onto surrounding T_{ROOT} tiles). In some cases, this may be undesirable, e.g. when a user wishes to paint a house of a very specific size. Allowing brush strokes to come with boundary conditions would make it possible to restrict the semantics to the stroked area only, without having to manually restrict (and subsequently unrestrict) the surrounding cells to prevent the overflow.

Lastly, in Section 3.3.5 there already was mention of a destructive method instead of the current conservative one for tile overwriting that might be preferred by users in certain circumstances. Being able to conceive an algorithm capable of doing this and then evaluating it with users will make it possible to figure out the nuances with respect to the use cases of the two methods, and whether one is more favorable by default than the other.

7.3.3 Algorithm

Besides user-focused research, there are many facets to explore regarding the HSWFC algorithm itself. In particular, the current way of choosing a next tile in the meta-tree involves a simple weighted choice of the direct descendants. What if we were to include contextual information as well? Pushing this idea further, what if every meta-tile would receive its own procedural noise generator that takes into account spatial and temporal aspects of the context around the the target cell on the grid? This would give meta-tiles another dimension to help approach their semantic intent, allowing them to generate with far more interesting distributions. For instance, a meta-tile that represents a shoreline can attempt to orient wa-

ter and sand generation in the brush strokes towards existing land mass, or a meta-tile that represents ”land” can generate large patches of either grassy areas or desert areas instead of high frequency noise as it would do currently.

More specific to the core of the algorithm, it would be nice to further investigate ”depropagation” and ”uncollapsing”: currently, depropagation resets many more cells than it really needs to. Correctly being able to estimate the scope of influence of placement of a tile onto a cell will help with making this operation more optimal. The feasibility of doing this might be interesting to study.

It is also worthwhile to more specifically investigate how the hierarchical semantics in HSWFC generalize to other forms of WFC, such as the overlapping model, graph-based WFC[11], or extending the algorithm to a 3D setting with voxels.

7.3.4 Constraints

This work did not put much emphasis on the exploration of new possibilities with regards to constraints in HSWFC, even though there much potential, in particular with regards to collapse paths (see Section 4.4). The introduction of this data structure allows us to identify semantic groupings of tiles in the output, e.g. all T_{HOUSE}, or all T_{FOREST}. One may be able to use connected component analysis to identify these groupings as separate instances of the semantic concept that the meta-tile represents. This allows for the specification of higher order constraints, just to name a few examples for the tileset used in this work:

- A village may contain only up to 10 houses.
- A house must have at least one door.
- A desert may contain only one body of water (oasis).
- Houses in a village must all be connected via a road.

There exists some work already on the specification of constraints that go beyond mere adjacency, e.g. in `DeBroglie` [28, 13] path constraints are featured, showing that WFC is not bound to adjacency constraints only. This can be a nice starting point for exploring the definition of more complex constraints between semantic instances.

Furthermore, such constraints could also be coupled with the constraint inheritance/overriding mechanic shown in Appendix D, to scope such higher-order constraints to specific meta-tiles. Since this constraint inheritance/overriding mechanic has not been fully completed yet and poses limitations in its current form, finding a way around these limitations is also another step towards making this a generic constraint specification paradigm for HSWFC.

Finally, it is currently not clear how generic the concept of constraint inference used by meta-tiles is, and how well it can be applied to more complicated constraints than just positive adjacency constraints. Finding this out will be instrumental to including more advanced constraints such as the ones presented above.

Appendix A

Proofs and examples

A.0.1 Given any meta-tree, $T_{ROOT} \xrightarrow[\textbf{Repr}]{} T$ for all T.

Proof. If there would exist a tile T_X for which $T_{ROOT} \xrightarrow[\text{Repr}]{} T_X$ does not hold, it means that there exists another tile T_Y where $T_Y \xrightarrow[\text{Repr}]{} T_X$, and not $T_{ROOT} \xrightarrow[\text{Repr}]{} T_Y$. This last requirement forces the presence of a source T_Z, where $T_Z \xrightarrow[\text{Repr}]{} T_Y$. However, the meta-tree is a connected DAG, and T_{ROOT} is the only source in that DAG, hence T_Z must be equivalent to T_{ROOT}, therefore $T_{ROOT} \xrightarrow[\text{Repr}]{} T_Y$ and from Property 1 it follows that $T_{ROOT} \xrightarrow[\text{Repr}]{} T_X$. □

A.0.2 Contextual constraints problem case

This example will show why having context-dependent adjacency constraints for the same tile can and will lead to contradictions. The input used for this example is given by the meta-tree in Figure A.1, and has two meta-tiles that represent the semantic trait of "forest", but only in one of them, trees may be self-adjacent.

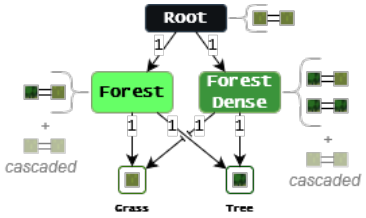
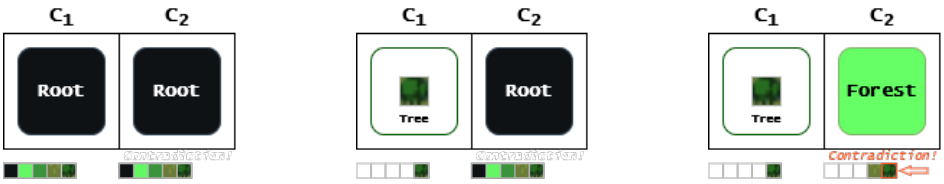


Figure A.1: Example meta-tree, with adjacency constraints next to corresponding meta-tile. Note how T_{ROOT} cascades its constraints downwards.

The problematic case is demonstrated by the sequence of events shown in Figure A.2. The problem occurs because we did not know beforehand which of T_{FOREST} or $T_{FOREST-DENSE}$ would be chosen for C_2, hence the choice of T_{TREE} for C_1 did not have any effect on C_2.



(a) Initial state (b) C_1 chooses T_{TREE} (c) C_2 chooses T_{FOREST}

Figure A.2: The problematic case with using the cascaded adjacency matrices per context for the same tile. When C_2 chooses T_{TREE} after (c), the grid will contradict, because in the context of T_{FOREST}, T_{TREE} cannot be self-adjacent.

Appendix B

Tileset

Recall that an input for HSWFC consists of the triple (T, A, E), where T is the set of tiles, A is the set of adjacency constraints over T, and E is the set of edges in the meta-tree over T as triples $(T_A, T_B, w_{T_A - T_B})$, where $w_{T_A - T_B}$ represents the weight. In figure B.1, T and E are shown in the form of a meta-tree. In Section B.1, the adjacency constraints specified in the input are shown. For brevity, directionality is indicated with: [U]p, [D]own, [L]eft, [R]ight, [S]ymmetric (all four directions).

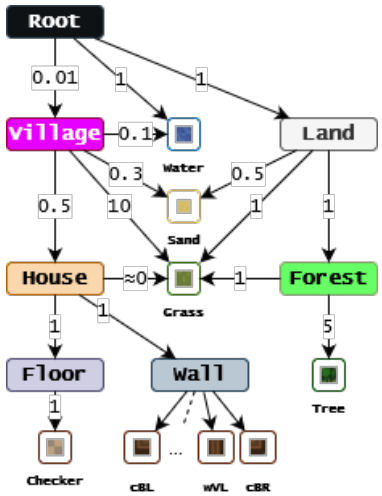


Figure B.1: The meta-tree that will be used throughout this work. Meta-tiles are shown as rounded rectangles with their name in them. The rounded squares correspond to terminal tiles. The numbers on the edges correspond to the weights of the semantic representations. The wall terminals have been abbreviated and are only partially shown, all of their edges carry weight 1.

B.1 Adjacency constraints and (terminal) tile list

Constraints are defined for both participants in a constraint pair for clarity, so that looking up one tile in this section will give you a full overview of its constraints. They are listed per tile, prefixed by the direction they apply to. There are no meta-tile constraints in this HSWFC tileset, hence they are omitted from this list. Figure B.1 gives an overview of the meta-tiles that are present.

ADJ T_{GRASS}

- S T_{GRASS}
- S T_{TREE}
- S T_{SAND}
- U $T_{WALL\_HORIZONTAL\_BOTTOM}$
- U $T_{CORNER\_BOTTOM\_LEFT}$
- U $T_{CORNER\_BOTTOM\_RIGHT}$
- D $T_{WALL\_HORIZONTAL\_TOP}$
- D $T_{CORNER\_TOP\_LEFT}$
- D $T_{CORNER\_TOP\_RIGHT}$
- L $T_{WALL\_VERTICAL\_RIGHT}$
- L $T_{CORNER\_TOP\_RIGHT}$
- L $T_{CORNER\_BOTTOM\_RIGHT}$
- R $T_{WALL\_VERTICAL\_LEFT}$
- R $T_{CORNER\_TOP\_LEFT}$
- R $T_{CORNER\_BOTTOM\_LEFT}$

ADJ T_{WATER}

- S T_{WATER}
- S T_{SAND}

ADJ T_{TREE}

- S T_{GRASS}

ADJ T_{SAND}

- S T_{WATER}
- S T_{GRASS}
- S T_{SAND}

ADJ $T_{CORNER\_TOP\_LEFT}$

- U T_{GRASS}
- D $T_{WALL\_VERTICAL\_LEFT}$
- L T_{GRASS}
- R $T_{WALL\_HORIZONTAL\_TOP}$

ADJ $T_{CORNER\_TOP\_RIGHT}$

- U T_{GRASS}
- D $T_{WALL\_VERTICAL\_RIGHT}$
- L $T_{WALL\_HORIZONTAL\_TOP}$
- R T_{GRASS}

ADJ $T_{CORNER\_BOTTOM\_LEFT}$

- U $T_{WALL\_VERTICAL\_LEFT}$
- D T_{GRASS}
- L T_{GRASS}
- R $T_{WALL\_HORIZONTAL\_BOTTOM}$

ADJ $T_{CORNER\_BOTTOM\_RIGHT}$

- U $T_{WALL\_VERTICAL\_RIGHT}$
- D T_{GRASS}
- L $T_{WALL\_HORIZONTAL\_BOTTOM}$
- R T_{GRASS}

ADJ $T_{WALL\_VERTICAL\_LEFT}$

- U $T_{WALL\_VERTICAL\_LEFT}$
- U $T_{CORNER\_TOP\_LEFT}$
- D $T_{WALL\_VERTICAL\_LEFT}$
- D $T_{CORNER\_BOTTOM\_LEFT}$
- L T_{GRASS}
- R $T_{CHECKER}$

ADJ $T_{WALL\_HORIZONTAL\_TOP}$

- U T_{GRASS}
- D $T_{CHECKER}$
- L $T_{WALL\_HORIZONTAL\_TOP}$
- L $T_{CORNER\_TOP\_LEFT}$
- R $T_{WALL\_HORIZONTAL\_TOP}$
- R $T_{CORNER\_TOP\_RIGHT}$

ADJ $T_{WALL\_VERTICAL\_RIGHT}$

- U $T_{WALL\_VERTICAL\_RIGHT}$
- U $T_{CORNER\_TOP\_RIGHT}$
- D $T_{WALL\_VERTICAL\_RIGHT}$
- D $T_{CORNER\_BOTTOM\_RIGHT}$
- L $T_{CHECKER}$
- R T_{GRASS}

ADJ $T_{WALL\_HORIZONTAL\_BOTTOM}$

- U $T_{CHECKER}$
- D T_{GRASS}
- L $T_{WALL\_HORIZONTAL\_BOTTOM}$
- L $T_{CORNER\_BOTTOM\_LEFT}$
- R $T_{WALL\_HORIZONTAL\_BOTTOM}$
- R $T_{CORNER\_BOTTOM\_RIGHT}$

ADJ $T_{CHECKER}$

- U $T_{WALL\_HORIZONTAL\_TOP}$
- D $T_{WALL\_HORIZONTAL\_BOTTOM}$
- L $T_{WALL\_VERTICAL\_LEFT}$
- R $T_{WALL\_VERTICAL\_RIGHT}$

Appendix C

User Study

This is a description of the user study that was performed. Two groups were made, A and B; group A worked with the stock WFC editor, and group B worked with the HSWFC editor. For group B, some additional questions were asked pertaining to HSWFC functionality.

C.1 Tasks

Users were given two tasks; the first one involved following some loose specification of requirements and wishes for a game environment/world in a game studio setting, creating a design for it. The second task was more about playing around with the result of that, creating variations of the design. The main intention of the second task was to give users ample opportunity to use all the tools that were given to them, so that their opinion would be somewhat more solidified for the editor/tooling-specific questions that followed. For the NASA TLX survey however it was better to focus on just one task and capture the experience of performing it. Below you can find both task descriptions:

T1 - Creating a game world: You are the environment designer of Dragon Bane, a top-down role-playing game where the player character may traverse the world to find and complete quests in search of eternal fame and glory. You have been asked by your producer to create environment concepts for one of the outdoor areas of the game, with the deadline being today.

In this game, the player cannot pass through water, can get quests and gear in towns, and the quests often take place in the forests and the wilds. Areas in the game, such as the one you will design, are connected to each other at their borders.

The gameplay designer and the lore wizard did have some more specific requirements for this particular area though:

- There should be two villages that are clearly separated
- One of the villages should be surrounded by forest as far as possible
- One of the villages should have a considerably larger house in the center of it with an interesting shape
- There should be an additional but very dense forest somewhere in the area
- The villages should be connected via a sandy path
- Both villages should be adjacent to some body of water

Besides these minimum requirements (which can be interpreted fairly loosely), they trust that you will be able to fill up the rest of the environment with interesting features, as long as the points above are not violated. You can use the description given earlier to do this in a way that makes sense for the game.

Make sure to make a snapshot of your final result, and keep the window open, as there will be a second task.

T2 - Making variations: With conceptual design, it is quite usual to make several variations and iterations of some basic idea, as this allows you to home in on a final direction that both you and your clients are happy with. In this case, you have been asked to make at least one or more variants of the landscape that you have created before.

In fact, you have been given the freedom to violate the rules that were given before for these variants, with the idea of using your previous work as a starting point and experimenting from that point onward with the functionality that the editor provides.

C.2 Questions

All non-open questions used a 7-point scale. The survey for the non-HSWFC version of the editor omits the HSWFC-specific questions.

NASA TLX

The NASA TLX questions were shown after **T1**.

- Mental Demand - The task was mentally demanding
- Temporal Demand - The pace of the task felt hurried or rushed
- Performance - I felt successful in accomplishing the task
- Effort - I felt that I had to work hard to achieve my desired level of performance in the task
- Frustration - I felt insecure, discouraged, stressed and/or annoyed with the editor while performing the task

General WFC questions

Shown after **T2**.

- I felt that the editor was able to capture my intent
- I felt like I had the freedom to tweak things easily
- I understood why certain tiles could not be placed in certain locations

HSWFC specific questions

Shown after **T2**, given that the HSWFC-editor was used.

- I found using the hierarchy for selecting a tile to paint with to be intuitive
- Using a single situation-dependent brush for painting and erasing felt intuitive
- Painting with meta tiles gave me the results I expected
- Adjusting the meta-tile probabilities had the results I expected
- I found using the regeneration tool useful for creating variations of my design

Open questions

- (After making some variations) Can you briefly describe your train of thought for coming up with the variant(s)?
- (At the end) Any particular feedback about the hierarchical tile set, the single situation-dependent brush, the regeneration tool, or elaboration on any of the scores given above?

Appendix D

Input Editor

In this Appendix we will cover a piece of implementation that was tangentially related to the research question. The input editor allows for the specification of adjacency constraints that are *scoped* under a particular meta-tile via meta-tree induced inheritance and overriding, allowing context-specific behaviour for the "same" tile. This is achieved by cloning the tile and changing the meta-tree relations and adjacency constraints such that the appropriate clone is used in the appropriate situation transparently, without the user noticing. The algorithm presented here for doing so is still work in progress and has limitations. The aim of this Appendix is to share the work that has been done so far for potential future work for intuitively providing input to HSWFC.

D.0.1 The input editor

The input editor is meant as an effort to reduce the labor that is required to modify or create a suitable input for HSWFC. It allows the user to create a hierarchical tileset from scratch, adding/modifying/removing tiles in the process. The design of the input editor guided a significant extension to the input processing part of HSWFC which allows for a form of constraint inheritance and overriding, which will be described in Section D.0.2. The input editor is solely implemented in the web app.

Hierarchical input editing

The core idea that drove the design of the input editor came from a wish to unify the interface for designing the meta-tree and the interface for specifying the adjacency constraints into one cohesive user experience. This was achieved by providing every meta-tile in the meta-tree has with its own "layer", represented by a HTML5 canvas; we shall refer to these as meta-layers. Editing a meta-tile then involves painting or erasing tiles from its meta-layer. Given that the meta-layer contains at least one instance of some tile, that tile will become a direct descendant of the corresponding meta-tile, with the weight of the corresponding edge leading to this tile determined by the amount of instances of that tile on the meta-layer. Adjacency constraints are specified by painting tiles adjacent to one another, similar to how stock WFC can infer adjacency constraints from providing image examples as input. Some tiles cannot be painted on a meta-layer, as it would violate Property 4:

- The meta-tile of the meta-layer itself.
- Ancestors of the meta-tile of the meta-layer.

Terminal tiles do not have layers associated with them, because they cannot have any descendants. Which layer is active can be selected by the user on a tile tree that is very similar

Figure D.1: The input editor of the HSWFC web editor, showing the meta-layer for a concave wall. Note how all non-wall tiles are transient, they are there to indicate the context in which these walls are allowed to be placed.

to the one from the environment editor, except in this case that tree is used for both tile selection (single click) and layer selection (double click).Whenever any meta-layer is modified, the meta-tree is rebuilt, which involves parsing the layer information. Since T_{ROOT} is the starting point of every meta-tree, there is always already a T_{ROOT} meta-tile (invisible) and corresponding meta-layer present on initialization.

Manipulating the tile collection

At the bottom right, users may find the current collection of tiles they may use. It contains both the terminal tiles as well as the meta-tiles defined so far. Additional tiles can be added via a dialog that can be summoned by clicking the plus (see Figure D.2). Note that in the algorithm we purely distinguish between terminal and meta-tiles by looking at whether they have any descendants. However, it was clearer to distinguish the two in the UI, as they serve different purposes, especially in the input editor.

The user can also choose to edit a tile, which can be done through the same dialog.

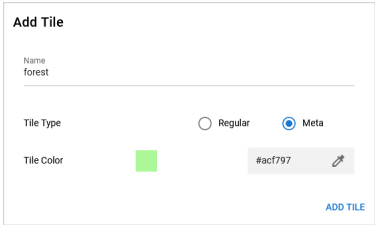


Figure D.2: The add tile dialog.

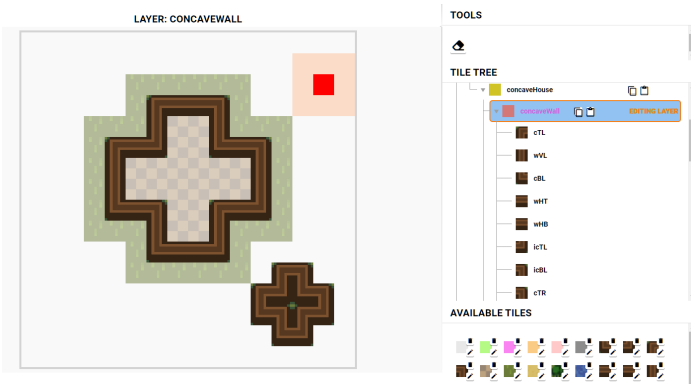


Figure D.3: The meta-layer for $T_{CONCAVE-WALL}$, which has transient T_{GRASS} tiles on it to specify adjacency constraints for the wall pieces.

Removing a tile from the collection is a more complicated operation, as a number of things need to happen:

- All tiles need to be re-indexed.
- All tiles on the canvases of the layers need to be mapped to this new indexing (including the copy buffer, see Section D.0.1).
- If it was a meta-tile, its associated meta-layer should be deleted.

Transient tiles

Sometimes, the tileset designer would like to purely indicate an adjacency constraint between two tiles, without any further consequences such as making either tiles descendants of the meta-tile currently being edited, or affecting the tile distributions of the aforementioned tiles. A prime example of the former case is shown in Figure D.3, where the designer wishes to specify that wall pieces can be adjacent to T_{GRASS} in some way, without making T_{GRASS} a descendant of T_{WALL}, so that painting a cell with T_{WALL} does not result into a substitution by T_{GRASS} upon collapsing the cell. An example of the latter case would be that the designer may want to indicate that T_{SAND} is can be adjacent to T_{GRASS} in each direction, without affecting their ratios within the tile distribution of the meta-tile that is currently being edited.

To facilitate these use cases, a tile instance may be painted over again to be made transient (visually indicated with transparency), which has two effects:

- The tile instance will not be considered for the process of checking whether it is a descendant of the meta-tile currently being edited.
- The tile instance is not counted towards the weight of the edge leading to it.

The wording of the first point is quite specific. As briefly touched upon in Section D.0.1, without transient tiles, determining whether a meta-tile T_{MT} has descendants is done in the following manner: given some tile T, as long as there exists any instance of T on the meta-layer of T_{MT}, T will be considered to be a direct descendant of T_{MT}. When a tile instance is made transient however, it is disregarded completely from this existence check. Hence, if all instances of T are made transient on the meta-layer of T_{MT}, then T is no longer regarded as a direct descendant, because to the input reading algorithm it will be as if there are no instances of T on the meta-layer of T_{MT}. This also means that the weight $w_{T_{MT},T}$ is determined by the amount of non-transient instances of T on the meta-layer of T_{MT}.

Transient tiles are essentially contextual mark-up: their sole purpose is to indicate, via adjacency constraints, the context in which other tiles on the meta-layer can exist on the grid; they are essentially "ghosts", in that they do not try to interfere with the other mechanics of input specification, such as descendancy or weight.

Tileset import/export

Hierarchical tilesets that are created in the input editor can be exported and imported. The export contains the meta-layers (with serialized `mathjs` matrices), and the set of tiles. In the current web-app implementation there is also an abstraction for nodes that is exported, but this is redundant. Importing just rebuilds the meta-tree according to the data found in the export. This allows users to continue working on tilesets in between sessions, because normally all data is kept in memory and lost upon refresh. It also allows for sharing specific/tailored tilesets with users that participate in a user study.

Loading tiles from an atlas image

For convenience, one can also input a tile atlas image (see Figure D.4 for an example). Given the correct dimensions, the web app will cut the tile atlas into tiles that are named according to the file name of the atlas, trivially eliminating duplicates by inserting the `base64` encodings of the images into a set. In the future, automatically putting the tiles in the atlas on a canvas and deriving the constraints in this manner may also be supported. This will further reduce the effort needed to create a HSWFC tileset, but it first requires supporting the specification of canvas dimensions per meta-layer instead of globally. Also note that any image exports downloaded from the environment editor can also be loaded in this manner, though chances are that not all constraints or tiles are present in every output. Finally, only lossless formats (such as `.png` are supported for now).

Copying and pasting meta-layers

Tile contents of a meta-layer can be copied, and pasted onto other layers. This allows for rapid editing, as a user can create a descendant of a meta-tile, and then just paste the contents of the ancestor into it for specific overrides and modifications. This buffer is cloned, to prevent accidental editing by reference.

Figure D.4: Example of a tile atlas with tiles that are 32x32 pixels. The input editor can parse this and add the tiles for further constraint specification. Tile atlas obtained from: https://opengameart.org/content/just-some-32x32-tiles.

D.0.2 Advanced input processing for input editor

While the design presented in Section D.0.1 succeeds at fusing the user interaction for meta-tree building and adjacency specification together, it is not necessarily intuitive; the main problem is that adjacency constraints are defined globally in HSWFC. This can potentially scatter the definition of adjacency constraints for some tile over a number of meta-layers, which harms overview. One solution would be to simply aggregate the constraints for a tile and display this somewhere on the UI. Much more interesting however would be if we could incorporate the meta-tree structure into adjacency specification in a meaningful way, as this would actually harness the expressiveness that HSWFC provides. This section will describe an implementation that attempts to do this by introducing constraint inheritance and overriding that is guided by the meta-tree.

Constraint cascading

In this new input processing scheme, the adjacency constraints are no longer collected globally: after meta-tree construction, the meta-layer canvas associated with each meta-tile is scanned for adjacency constraints, and these are stored per meta-tile. This leads to many adjacency matrices, one set (of the four cardinal directions) for each meta-tile.

Then, for each terminal tile T_X, all the possible unique paths from T_{ROOT} to T_X are collected, and for each of these paths the set of adjacency constraints is accumulated with the adjacency constraints of the next meta-tile as the path is traversed downwards, creating a cascade (see Figure D.5(a)), all the way up until T_X itself. The cascades are stored in a dictionary per terminal tile, keyed by the paths that lead to them.

Constraint inheritance and overriding based on cascades

It is possible that, given a terminal tile T_X and adjacency cascades A_X^1 and $A_X T^2$, that $A_X^1 \neq A_X^2$. This means that somewhere along the paths that lead to A_X^1 and A_X^2, there was

a difference in adjacency specification. But it is also possible that there is no difference between the cascades, and thus that $A_X^1 = A_X^2$. In order to find out which cascades share the same adjacency constraints, they can be clustered such that each cluster contains cascades that are equivalent. If this clustering process results into a single cluster for T_X, it means that T_X has globally homogeneous adjacency constraints. However, if there is more than one cluster, it means that T_X has different adjacency constraints depending on the path taken in the meta-tree.

In HSWFC, when a tile is chosen for a cell, a propagation wave occurs according to what the adjacency constraints of the tile allow for, just like in stock WFC. Thus, the first intuition is to simply allow the different adjacency specifications for T_X to coexist, and then use the one that is applicable for the current context of the cell that is being propagated, the context being the path taken in the meta-tree. This would work with contexts that are static and predetermined on the grid, because we then know beforehand how to constrain T_X appropriately.

In our case however, the contexts are not static; a collapse path through the meta-tree is determined dynamically by the meta-tiles that randomly get chosen, or even painted by the user. Why this is a problem is rather complicated and best illustrated through an example, which can be found in Section A.0.2 of Appendix A.

Due to the ramifications of this example, we fundamentally cannot have different contextual constraints for the same tiles in HSWFC. We can however, have different tiles with different constraints. This is the basis for an idea that involves tile cloning: terminal tiles that have multiple paths in the meta-tree that lead to different cascades, can be duplicated and given the appropriate adjacency constraints based on these paths. In the worst case, this can

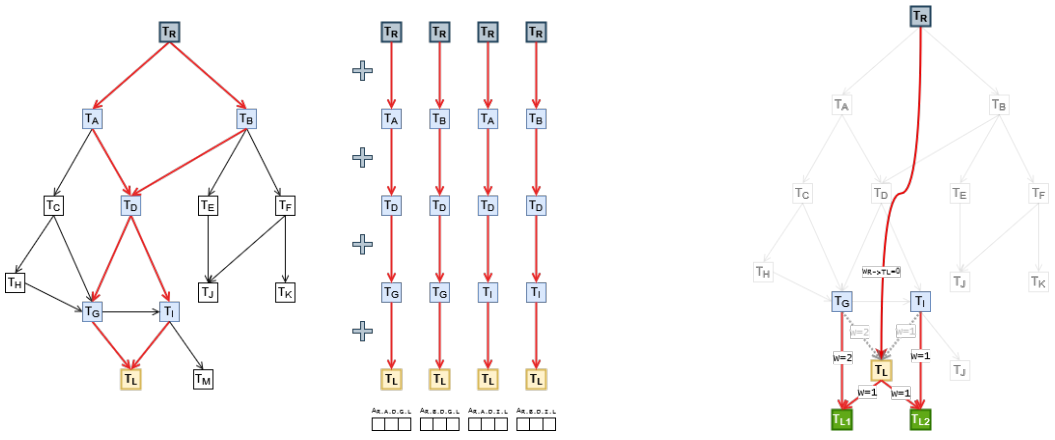


Figure D.5: (a) shows how adjacency constraints cascade downwards across all possible paths for T_L (with the paths separately laid out on the right side to convey this), accumulating constraints as the paths are iterated downwards. (b) visualizes the tile splitting algorithm for T_L, for a difference in its adjacency constraint specification between T_G and T_I.

increase terminal tile count up to the amount of unique paths from T_{ROOT} to the terminal tiles. In practice however, most of these paths lead to the same set of adjacency constraints. To employ this idea, we perform a tile-splitting procedure for any tile that ends up with more than one cascade cluster, consisting of the following steps:

1. For each cluster, a new tile is created and indexed that is an exact clone of T_L.
2. Each clone will receive the adjacency constraint vector that corresponds to the adjacency cascade of its cluster.
3. T_L itself gets connected to its clones with equal distribution of weights and disconnected from its direct ancestors. Instead, it gets connected to T_{ROOT} with a zero-weight edge.
4. The clones will be connected to the former direct ancestors of T_L with the same weights, so that they are constrained to the meta-tile that caused the difference in adjacency constraints,

The edits to the meta-tree that are made in this part of the input processing are not reflected in the tree UI, which causes a discrepancy in the meta-tree between front-end and algorithm back-end. This is not a problem though, because the identifiers of the tiles that need to be exposed to the user are still the same. The user does not need to see the cloned tiles, as the intention of this feature is allowing contextual adjacency overrides.

The connection to T_{ROOT} is made in order to ensure that T_L is still somehow connected to the meta-tree. While zero-weight edges normally would cause the algorithm to halt, in this case we have guaranteed that there is always an alternate path, namely through the cascade paths that formed the clones. This allows users to paint with T_L as usual, without noticing that T_L has now transformed into a meta-tile that collapses further into its clones, while it also being correctly bounded by the meta-tiles that caused the difference in constraint specification.

This feature interacts with some other parts of the input processing implementation in interesting ways:

Transient tiles: Tiles that are fully transient on some meta-layer are not actually part of the meta-tree at that specific point anymore. At the same time, they may generate adjacency constraints further downstream. If we cascade adjacency constraints downwards according to the paths of the meta-tree with these fully transient tiles pruned, that downstream part will be ignored; simply put, the tile that we were painting the context for will have the proper adjacency constraints, but the transient tiles we used to paint this context will not receive these, which causes loss of bidirectionality in the constraints. To prevent this, edges that resulted from fully transient tiles are only removed from the meta-tree after enumeration of all the possible paths, and their adjacency specification is added to the first ancestor that contains at least one non-transient instance of this tile.

Meta-tile adjacency constraints: Adjacency constraints between meta-tiles and terminal tiles, or even among meta-tiles, have the effect of a wildcard (as described in Section 3.3.4). The process of transferring the adjacency constraints to the terminal tiles must happen after the splitting process, as it would allow for scoping the wildcard effect under a particular meta-tile and only applying it to the cloned tiles that are descendants of this meta-tile. Not doing this will cause unexpected behaviour, as the augmented adjacency constraints will extend to all the clones of a tile, which does not follow the idea of having different adjacency constraint contexts per path.

The limitation of the current iteration of this algorithm is that it only works for differences in adjacency specification that occur in direct ancestors. This is mostly due to the last step in the procedure. Connecting the cloned tiles to the direct ancestors of T_L will not account for correct scoping if the difference in adjacency specification occurred further upstream in the meta-tree. The solution to this problem is complex, and will therefore be relegated to future work.

Appendix E

Web App Optimizations

In this appendix we will cover some interesting optimizations that were implemented for the Quasar-based HSWFC editor web application. The aim of these is to guarantee enough performance for carrying out the user study presented in Appendix C with the tileset presented in Appendix B such that any significant biases in cognitive load and usability scores caused by performance issues are avoided.

E.0.1 Web-worker for background computation

As soon as the first prototype of the web app editor was up and running, it became clear that the computations of the algorithm should not be happening on the same thread as the front-end, as it was starving the thread, making user interaction sluggish and unresponsive.

The natural solution to this in modern web apps is to use something called a web-worker, which essentially is another browser process that can be spun up and communicated with via a messaging system. This feature is universally supported among current versions of the large browsers that are available[1]. Usually messages that represent actions are sent from the front-end to the worker, though communication can be initiated bidirectionally; if something special happens on the worker that is running the algorithm (e.g. a contradiction), this can be communicated to the front-end as well.

The worker and the front-end do not share the same state, so some of this state needs to be communicated to the other side through messages, which can be done for any serializable data types. The worker runs the algorithm and will therefore obtain the result data; if we wish to display this, it has to be sent to the front-end. Likewise, importing the input happens in the front-end, so if we want this information to be available to the algorithm, it has to be sent to the worker.

Sending too much information too often can cause a bottleneck to occur on both sides, where the threads get starved from either sending or receiving too many messages. More specifically, if the front-end "spams" the worker, it can actually reduce the solving performance of the algorithm, because either the worker thread is too busy with handling incoming messages, or the request to collapse a cell is taking longer than the interval at which the front-end sends messages. The latter situation can escalate fairly quickly if the sending interval is on average shorter in duration than the ability of the worker to complete its update cycle, causing it to lag incrementally further behind with handling the en-queued cell collapse requests. Therefore, instead of sending collapse requests at regular intervals, the front-end waits for an acknowledgement from the worker that the collapse request was handled properly, and only then sends the next collapse request. This ensures that when the

[1] https://caniuse.com/webworkers

algorithm spends a long time on the update cycle, the front-end does not continue spamming the worker with collapse requests, causing the downwards spiral as described above.

While this solved one issue, it created another one: there exists a balance between how many messages are sent, and how many collapse requests there are per message. Initially, this latter value was not implemented, so it was just 1. The algorithm became particularly sluggish because of this, but the thread did not seem to be saturated; instead, it was waiting due to the latency incurred by messaging. Therefore, a "generation speed" parameter was conceived, which delegates the responsibility of choosing the balance between thread utilization and responsiveness to the user, and also gives them the opportunity to speed up the algorithm to quickly solve an instance, at the cost of UI performance.

The generation speed parameter (also shown on the GUI in Figure 5.11) controls how many cells the HSWFC instance within the worker will look for to collapse upon receiving an update message, before completing its update cycle and handing control back over to the front-end. Contrary to intuition, setting this value very high still affects smoothness of the user interaction, because the algorithm is busy for much longer periods of time, and newly painted cells will have to wait for the previous update cycle to finish before they can be processed.

Finally, the amount of data sent from the worker to the front-end can also potentially starve the front-end thread. This actually has more to do with data representation, than saturating the messaging bandwidth: if only the entire grid is sent from worker to front-end, the front-end does not know what has changed, so it is forced to repaint the entire canvas. While this is error-proof due to its simplicity, it quickly starts to impact UI performance, especially with larger grids. The solution to this was to also send change lists of cells that have been modified, which is kept track of by HSWFC per update cycle and returned at the end of each cycle, accompanied with the occasional full redraw in case of operations that involve manipulating the full grid state, such as snapshot loading, grid resetting, undoing, etc.

E.0.2 Webassembly and Eigen C++

After deploying an initial implementation of the back-end fully using `mathjs`, and creating more complex tilesets that started to use more tiles and more complex constraints that would create larger propagation waves, it became clear that using `mathjs` in the backend would not be scalable, and would in fact interfere with the responsiveness that was required for the user study. We want to minimize the bias introduced by technical defects, such as low responsiveness and bugs, thus a different solution had to be found.

The problem with `mathjs` lies in both the sluggishness of interpreted Javascript, and the data representation that `mathjs` used: a matrix in `mathjs` is represented by an array of arrays. In Javascript and many other languages, arrays are typically represented as object instances. In general, one can assume that newly instanced objects get a memory allocation that starts at a random point in the available memory, which means that the locality of the data in an array of arrays is pretty much all over the place. When the CPU fetches data from memory, surrounding pieces of information in the memory get cached, and the proximity

and frequency of access determine the level of caching that occurs. This hardware optimization, if utilized properly, can be responsible for tremendous increases in performance, and can potentially drive algorithm design [43]. Likewise, if completely ignored due to a bad data layout, much of this potential performance is lost. This can partially be remedied by using a different representation of the data, but this would require constant casting back and forth between the two representations in order to be able to use the handy indexing and methods of `mathjs`, which would negate the performance benefits.

Using webassembly was the first idea that came up for improving performance in general, as it had been established for some years now, and it offered the opportunity to run heavy computations very efficiently in the browser, much more than `Javascript`. Since webassembly is fairly mature at this point, many tools exist that are able to compile code from high performance languages such as Rust and C++ into webassembly. There is a requirement in our case though: we are looking for a high performance matrix library that preferably has support for boolean operations and multidimensional arrays. After some searching, `Eigen` was stumbled upon. It appeared to perform on par or better than counterparts with the same feature set that were available for public use, such as `xtensor` or `Blaze`[2,3,4]. Better yet, a webassembly port existed already as npm module, named `eigen-js`[5].

One problem however was that the `eigen-js` module was not immediately sufficient for our needs. It only exported a very small subset of functionality of `Eigen`, which mostly involved matrix operations. It appeared that `Eigen` did have support for multidimensional arrays in the form of tensors, as a community contribution. Tensors were specifically contributed to `Eigen` for machine learning purposes, but they appeared to be a great fit for our use case, allowing uniform computations across dimensions: single numbers, vectors and matrices are still represented as tensors, which means that the same operations can be used on all of them, keeping code complexity low. Furthermore, tensor operations are lazily evaluated, giving the compiler the opportunity to optimize the resulting code based on the expression tree[6].

The initial experimental implementation involved using `Eigen` just for the matrix computations, making use of the npm module untouched. It involved casting data back-and-forth between `mathjs` and `Eigen` during run-time, replacing the bit operations with adding and multiplications of ones and zeroes. This was already moderately faster, providing the green-light for a more advanced implementation. The goal was to be able to fully perform the propagation operation on the webassembly side of the code base. This required forking and modifying the npm module [7] to include tensor support and bit operations, which involved writing C++ code, compiling it to webassembly using Emscripten [8], and then building the new npm module for usage within HSWFC.

[2] https://romanpoya.medium.com/a-look-at-the-performance-of-expression-templates-in-c-eigen-vs-blaze-vs-
[3] https://bertrandbev.github.io/eigen-js/#/benchmark
[4] https://eigen.tuxfamily.org/index.php?title=Benchmark
[5] https://www.npmjs.com/package/eigen
[6] https://eigen.tuxfamily.org/dox/TopicLazyEvaluation.html
[7] Modified `eigen-js` npm module: https://github.com/Archer6621/eigen-js
[8] Emscripten: https://emscripten.org/

The final implementation mirrors Section 5.1.3 using the `Eigen` API . The additional benefit here is that both `Eigen` and the LLVM backend for webassembly used by `Emscripten` support SIMD as of recently[9], so it can benefit from the same speed-up from vector operations as `numpy` does. Data casting costs between `Eigen` and the rest of the web app are minimized by using the same dimension order for the indices as in the backend, and avoid casting the choices array to `mathjs` altogether, essentially sending array buffers around between the front-end and back-end and then creating methods for accessing them conveniently according tot the indexing scheme that `Eigen` uses.

In the end, a speed-up of approximately 3-4 times was achieved with this, and that is without even moving the full algorithm to the backend or enabling SIMD for the `Emscripten` compilation yet, so potential for further performance increases is still there as low-hanging fruit. In any case, this speed up was sufficient for proceeding with the user study with the tileset shown in Appendeix B.

While the data locality problem is partially solved, the current solution is still not ideal. The raw data is stored linearly for the 3D choices array, which means that we can only choose one dimension to be laid out contiguously. The natural choice would be the depth, corresponding to the tile choices that are still allowed for some cell, as the propagation algorithm operates on two cells (each consisting of an array of allowed choices) per iteration. Potentially more speed gains can be realized if a custom layout was chosen that also guarantees some locality for the two other axes, e.g. by using a blocked memory layout.

E.0.3 Efficiently getting the minimum entropy

Another bottleneck that started to occur early on is the retrieval of the cell with mimimum entropy, which is used for automatic generation. It seemed that finding the minimum entropy almost doubled the execution time for a $128x128$ grid compared to a naive search method that simply searches for the next empty cell in iteration order. Initially, finding the minimum entropy was implemented as a search on a `mathjs` matrix that kept the entropy values for the grid in memory . This was problematic for two reasons:

- Finding the minimum required searching the full grid.
- `mathjs` matrices store their data all over the place in memory.

The ramifications of this quickly became apparent after some targetted benchmarks, which showed that as the canvas size increased, finding the cell with minimum entropy was overtaking all other parts during automatic generation, which impacted UI responsiveness (see Table E.1).

[9] SIMD support:
- https://github.com/WebAssembly/\gls{simd}/blob/main/proposals/\gls{simd}/ImplementationStatus.md
- http://eigen.tuxfamily.org/index.php?title=FAQ#Which_\gls{simd}_instruction_sets_are_supported_by_Eigen.3F
- https://emscripten.org/docs/porting/\gls{simd}.html

Grid Size	Mode	Time	Slowdown
32 x 32	Scanline Search	2.0s	n/a
128 x 128	Scanline Search	110.0s	3.44x
32 x 32	Minimum Entropy Search	2.5s	n/a
128 x 128	Minimum Entropy Search	210.0s	5.25x

Table E.1: Quick benchmarks revealing that finding the minimum entropy value was significantly slowing down the algorithm. Scanline just finds the next empty cell in iteration order. The slowdown column reveals how much slower the algorithm gets per cell compared to the 32x32 case for that mode.

Sorted set buffer based on skip lists

This solution, which is the one that the algorithm is currently using, is to use a sorted set as an entropy value buffer. A sorted set is a collection of keys and values that is ordered by the values, with the keys being unique. The underlying data-structures are a hash table for the keys (referencing the values), and a skip list for the values (referencing the keys), which enforces the ordering (see Figure E.1 for an overview of the skip list data structure). In our case, the keys are instances that correspond to the grid cells, and the values are their entropy. As a result, setting an entropy value for some cell will always overwrite the existing value if that cell was in the buffer, otherwise it will be added. Sorted sets implemented with skip lists are quite performant because insertions and deletions are both $O(log(n))$, and getting the key with smallest value is simply $O(1)$ [44, 45].

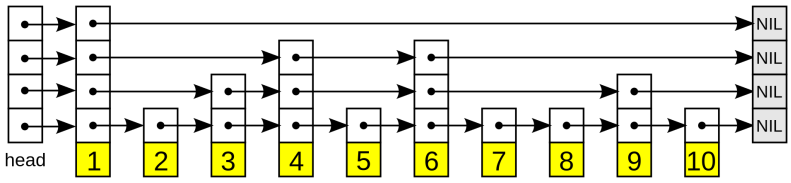


Figure E.1: The skip list data structure.

Furthermore, our sorted set approach does not store the full grid, but cells are only added as their entropy changes, and removed if fully collapsed to a terminal tile.

In practice, the buffer completely eliminated the bottleneck, which is strange considering the fact that skip lists have even worse data locality than the `mathjs` matrices (see Table E.2). One reason could be that the immediate neighbours that are discovered during propagation are often candidates for next entropy minima, which means that searching where to insert the next node in the skip list barely ever gets close to the worst case time complexity of $O(log(n))$. A second advantage over the minimum quad-tree is that the entropy buffer is ordered, and therefore can retrieve multiple minima without having to search, which may be useful for multi-threaded WFC implementations.

The only remaining difficulty that arose at times was when checkpoints/snapshots were loaded, or when a cell collapsed without causing a propagation wave, which can lead to

the buffer running dry. In order to deal with these scenarios in a universal way, the old algorithm for searching the grid is simply used to find a minimum entropy cell.

Grid Size	Mode	Time
128 x 128	Minimum Entropy Search	210.0s
128 x 128	Scanline Search	110.0s
128 x 128	Minimum Entropy Quad Tree	75.0s
128 x 128	Minimum Entropy Sorted Set	48.0s

Table E.2: Benchmarks of a variety of methods for finding the minimum entropy. These benchmarks were done using a subset of the tileset from Appendix B, and before accelerating the propagation with webassembly. The sorted set shows significant performance benefits for larger grid sizes.

Acronyms

DAG Directed Acyclic Graph. 13, 40, 55

GUI Graphical User Interface. 40, 70

HSWFC Hierarchical Semantic Wave Function Collapse. 3, 5, 8, 10, 14–16, 18–28, 31, 35, 38, 40, 42, 43, 45–48, 50–54, 56, 58–62, 64–66, 69–71

PCG Procedural Content Generation. 1, 7, 8, 44, 52

SIMD Single Instruction Multiple Data. 28, 32, 35, 72

UI User Interface. 4, 35, 39, 41, 45, 49, 51, 62, 65, 67, 70, 72

WFC Wave Function Collapse. 1–3, 5–8, 10, 13–16, 18–22, 24, 42, 43, 47, 50–54, 58, 61, 66, 73

Bibliography

[1] George Kelly and Hugh McCabe. A survey of procedural techniques for city generation. *The ITB Journal*, 7(2):5, 2006.

[2] Michael Cook, Simon Colton, Jeremy Gow, and Gillian Smith. General analytical techniques for parameter-based procedural content generators. In *2019 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2019.

[3] Sam Earle, Maria Edwards, Ahmed Khalifa, Philip Bontrager, and Julian Togelius. Learning controllable content generators. In *2021 IEEE Conference on Games (CoG)*, pages 1–9. IEEE, 2021.

[4] Michael Beukman, Manuel Fokam, Marcel Kruger, Guy Axelrod, Muhammad Nasir, Branden Ingram, Benjamin Rosman, and Steven James. Hierarchically composing level generators for the creation of complex structures. *arXiv preprint arXiv:2302.01561*, 2023.

[5] Roland Van Der Linden, Ricardo Lopes, and Rafael Bidarra. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1):78–89, 2013.

[6] Thijmen SL Langendam and Rafael Bidarra. miwfc - designer empowerment through mixed-initiative wave function collapse. In *Proceedings of the 17th International Conference on the Foundations of Digital Games*, FDG '22, New York, NY, USA, 2022. Association for Computing Machinery.

[7] Ruben Smelik, Tim Tutenel, Klaas Jan De Kraker, and Rafael Bidarra. Integrating procedural generation and manual editing of virtual worlds. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, pages 1–8, 2010.

[8] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Müller, and Eugene Zhang. Interactive procedural street modeling. In *ACM SIGGRAPH 2008 papers*, pages 1–10. 2008.

[9] Maxim Gumin. Wave Function Collapse Algorithm, September 2016. Github repository at: https://github.com/mxgmn/WaveFunctionCollapse.

[10] Isaac Karth and Adam M. Smith. Wavefunctioncollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, New York, NY, USA, 2017. Association for Computing Machinery.

[11] Hwanhee Kim, Seongtaek Lee, Hyundong Lee, Teasung Hahn, and Shinjin Kang. Automatic generation of game content using a graph-based wave function collapse algorithm. *Proceeding of IEEE Conference on Games*, 1(1):1–4, 08 2019.

[12] Isaac Karth and Adam Smith. WaveFunctionCollapse: Content generation via constraint solving and machine learning. *IEEE Transactions on Games*, PP:1–1, 05 2021.

[13] Adam Newgas. Tessera: A practical system for extended wavefunctioncollapse. *The 18th Foundations of Digital Games*, 2023.

[14] Tobias Nordvig Møller, Jonas Billeskov, and George Palamas. Expanding wave function collapse with growing grids for procedural map generation. In *Proceedings of the 15th International Conference on the Foundations of Digital Games*, FDG '20, New York, NY, USA, 2020. Association for Computing Machinery.

[15] Michael Beukman, Branden Ingram, Ireton Liu, and Benjamin Rosman. Hierarchical wavefunction collapse. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 19, pages 23–33, 2023.

[16] Ruben M Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. A declarative approach to procedural modeling of virtual worlds. *Computers & Graphics*, 35(2):352–363, 2011.

[17] John Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2):257–285, 1988.

[18] Edith Galy, Magali Cariou, and Claudine Mélan. What is the relationship between mental workload factors and cognitive load types? *International journal of psychophysiology*, 83(3):269–275, 2012.

[19] Christine S Lee and David J Therriault. The cognitive underpinnings of creative thought: A latent variable analysis exploring the roles of intelligence and working memory in three creative thinking processes. *Intelligence*, 41(5):306–320, 2013.

[20] Ceyuan Yang, Yujun Shen, and Bolei Zhou. Semantic hierarchy emerges in deep generative representations for scene synthesis. *International Journal of Computer Vision*, 129:1451–1466, 2021.

[21] Alain Lioret, Nicolas Ruche, Etienne Gibiat, and Cédric Chopin. Gan applied to wave function collapse for procedural map generation. In *ACM SIGGRAPH 2022 Posters*, SIGGRAPH '22, New York, NY, USA, 2022. Association for Computing Machinery.

[22] Yuhe Nie, Shaoming Zheng, Zhan Zhuang, and Xuan Song. Extend wave function collapse to large-scale content generation. *arXiv preprint arXiv:2308.07307*, 2023.

[23] Paul Merrell and Dinesh Manocha. Model synthesis: A general procedural modeling algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 17(6):715–728, 2010.

[24] Isaac Karth and Adam M. Smith. Addressing the fundamental tension of pcgml with discriminative learning. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, FDG '19, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Oskar Stålberg. Townscaper. Raw Fury, Steam, Epic Games Store, GOG, Nintendo Switch, xBox, App Store, Google Play, 2021. Browser version: https://oskarstalberg.com/Townscaper/.

[26] Adam Newgas. Editable wfc, April 2022. Article can be viewed at: https://www.boristhebrave.com/2022/04/25/editable-wfc/.

[27] Adam Newgas. Driven wavefunctioncollapse, June 2021. Article can be viewed at: https://www.boristhebrave.com/2021/06/06/driven-wavefunctioncollapse/.

[28] Adam Newgas. Constraints, September 2018. Article can be viewed at: https://boristhebrave.github.io/DeBroglie/articles/constraints.html.

[29] Paul Merrell. Example-based procedural modeling using graph grammars. *ACM Transactions on Graphics (TOG)*, 42(4):1–16, 2023.

[30] Jonas Freiknecht and Wolfgang Effelsberg. Procedural generation of multistory buildings with interior. *IEEE Transactions on Games*, 12(3):323–336, 2020.

[31] Ruben Smelik, Krzysztof Galka, Klaas Jan de Kraker, Frido Kuijper, and Rafael Bidarra. Semantic constraints for procedural generation of virtual worlds. In *Proceedings of the 2nd International Workshop on Procedural Content Generation in Games*, PCGames '11, New York, NY, USA, 2011. Association for Computing Machinery.

[32] Konstantinos Sfikas, Antonios Liapis, and Georgios N. Yannakakis. A general-purpose expressive algorithm for room-based environments. In *Proceedings of the 17th International Conference on the Foundations of Digital Games*, FDG '22, New York, NY, USA, 2022. Association for Computing Machinery.

[33] Levi van Aanholt and Rafael Bidarra. Declarative procedural generation of architecture with semantic architectural profiles. In *2020 IEEE Conference on Games (CoG)*, pages 351–358. IEEE, 2020.

[34] Jassin Kessing, Tim Tutenel, and Rafael Bidarra. Designing semantic game worlds. In *Proceedings of PCG 2012 - Workshop on Procedural Content Generation, co-located with the Seventh International Conference on the Foundations of Digital Games*, pages 40–48, Raleigh, NC, may 2012. ACM.

[35] Peter Kan, Andrija Kurtic, Mohamed Radwan, and Jorge M Loaiciga Rodriguez. Automatic interior design in augmented reality based on hierarchical tree of procedural rules. *Electronics*, 10(3):245, 2021.

[36] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik, and Klaas Jan de Kraker. Rule-based layout solving and its application to procedural interior generation. In *Proceedings of the CASA'09 Workshop on 3D Advanced Media in Gaming and Simulation*, pages 15–24, Amsterdam, The Netherlands, jun 2009. Utrecht University.

[37] Sam Snodgrass and Santiago Ontanon. A hierarchical mdmc approach to 2d video game map generation. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 11(1):205–211, Jun. 2021.

[38] Jonathan L Gross and Jay Yellen. *Handbook of graph theory*. CRC press, 2003.

[39] Frank Harary, Robert Z. (Robert Zane) Norman, and Dorwin Cartwright. *Structural models : an introduction to the theory of directed graphs*. Wiley, 1965.

[40] Shaad Alaka and Rafael Bidarra. Hierarchical semantic wave function collapse. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, pages 1–10, 2023.

[41] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[42] Christopher Nikulin, Gabriela Lopez, Eduardo Piñonez, Luis Gonzalez, and Pia Zapata. Nasa-tlx for predictability and measurability of instructional design models: case study in design methods. *Educational Technology Research and Development*, 67:467–493, 2019.

[43] Markus Kowarschik and Christian Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. *Algorithms for memory hierarchies: advanced lectures*, pages 213–232, 2003.

[44] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[45] Kurt Mehlhorn, Peter Sanders, and Peter Sanders. *Algorithms and data structures: The basic toolbox*, volume 55. Springer, 2008.