FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Tile-Based Procedural Terrain Generation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Dominik Scholz
Registration Number 01527434

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Dipl.-Ing. Bernhard Steiner, BSc

Vienna, 8th January, 2019

Dominik Scholz	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Dominik Scholz
Kapellerfelderstraße 63, 2201 Gerasdorf, Österreich

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 8. Jänner 2019

Dominik Scholz

Acknowledgements

First and foremost I want to thank Bernhard Steiner for his valuable input during research and writing. I also want to thank my parents for giving me the opportunity to aim for an academic degree. Last but not least I want to thank all my colleagues, friends and family members who supported me along my journey.

Kurzfassung

Procedural Content Generation ist in den letzten Jahren ein immer wichtigeres Werkzeug für die Erstellung virtueller Erlebnisse geworden. Es ermöglicht Designern große und detaillierte virtuelle Welten zu schaffen. Herkömmliche Anwendungsgebiete in denen Procedural Content Generation verwendet wird, wie zum Beispiel der künstlichen Erstellung von Terrain, benutzen hoch spezialisierte Algorithmen. Diese Algorithmen können nur von Programmieren oder technisch versierten Designern benutzt werden, da die Parametrisierung ein technisches Verständnis voraussetzt. Mit einem neuen Procedural Generation Ansatz namens Model Synthesis ist es möglich unterschiedliche Arten von Inhalten, wie Texturen oder 3D-Modelle, mit einem generischen Algorithmus zu erstellen ohne domänenspezifisches Wissen von Grammatiken oder logischen Ausdrücken vorauszusetzen. In dieser Arbeit wird eine Demonstrations-Anwendung erstellt, um zu zeigen wie diese neue algorithmische Vorgehensweise dazu benutzt werden kann endloses Terrain zu generieren. Ein häufig benutzter Model Synthesis Algorithmus namens Wave Function Collapse wird in ein System integriert, welches die komplette Funktionalität für den Einsatz dieser Technologie in einem realen Anwendungsfall bereitstellt. Das System beginnt mit dem Verarbeiten der Eingabe: quadratische Blöcke welche 3D-Geometrie beinhalten. Bei der Terrain Erstellung werden diese in einem Gitter so platziert, dass die mit ihren Nachbarn zusammenpassen. Um festzustellen welche Blöcke nebeneinander passen analysiert das entwickelte System die Blöcke anhand ihrer Geometrie und bildet daraus Nachbarschaftsbedingungen. Der Terrain Generator benutzt dann diese Information um mit Hilfe des Wave Function Collapse Algorithmus kleine Stücke an Terrain zu erstellen. Unser entwickeltes System verbindet anschließend diese Stücke um beliebig großes Terrain, sowohl vor als auch zur Laufzeit, zu generieren.

Abstract

In recent years, procedural content generation became a valuable tool to create virtual experiences. In a multitude of computer graphics applications, it aids designers to generate vast, detailed worlds. Previous methods for procedural content generation, for example, terrain generation, only cover certain distinct domains where highly specialized algorithms are used. These specialized algorithms can only be configured by programmers or artists with a technical background. With a new approach to procedural generation, called Model Synthesis, it is possible to generate a variety of content, like textures or models with a general purpose algorithm that can be configured by users without domain-specific knowledge of grammars or logical expressions. In this thesis, a proof-of-concept implementation is created to show how this new algorithmic approach can be utilized to generate infinite terrains. A widely used Model Synthesis algorithm named Wave Function Collapse is incorporated into a system providing the complete pipeline needed for the use in a production environment. This pipeline starts with processing of the input: three-dimensional blocks of geometry called tiles. In the terrain generation, they are placed in a lattice such that their geometry matches with the neighboring ones. To create this matching information, the developed system analyzes the geometry of the tiles to generate neighborhood constraints. The terrain generator then uses this information to solve small patches of terrain with instances of the Wave Function Collapse algorithm. Merging these patches to an arbitrary large terrain that can be extended on runtime is also achieved with the developed system.

Contents

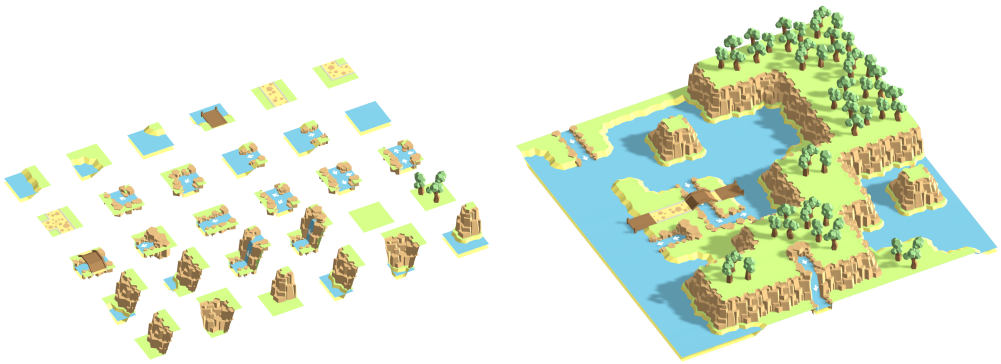
CHAPTER 1

Introduction

Procedural content generation (PCG), meaning the automatic creation of game content like levels or items by algorithms, has been in use since the early 1980s. Back then it was used to save disk space, for example, by generating textures with algorithms [STN16]. After disk space got larger, PCG became widely unused. In the last decade however, it got a lot of traction and various games made PCG popular again. PCG is used frequently among smaller game development studios because it is capable of matching the amounts of content larger studios with greater workforce are creating manually. These so-called indie game studios are also often the leading force for new experimental features in games. In the case of PCG, for example, games like Minecraft[1] or No Man's Sky[2] made the use of run-time procedural terrain generation popular.

In this thesis, an algorithm named *Wave Function Collapse* (WFC) [KS17], of a recently developed class of PCG algorithms called Model Synthesis is utilized. These algorithms work by specifying input tiles (mostly two or three dimensional) and their adjacency constraints and then create a consistent model out of it (see Figure 1.1). Adjacency constraints are constraints which specify which tiles can be placed next to each other. The output model has to satisfy all adjacency constraints of every tile. Generating levels with such tiles is a technique used since the 80s to draw a scene efficiently [Ayc16]. Today however, tiles are often not only 2D sprites but 3D blocks of geometry. The major benefit of the new approach is that it does not need a complicated parameterization to create the wanted results. Model Synthesis algorithms use rather intuitive inputs - as the name suggests *3D-models* - to generate similar output, hence the name *synthesis*. This leads to greater control for designers in comparison to current PCG algorithms.

[1] `https://minecraft.net/` Accessed: 17.12.2018
[2] `https://www.nomanssky.com/` Accessed: 17.12.2018



(a) (b)

Figure 1.1: A set of input tiles (a) is used to generate an output model (b).

1.1 Problem Statement and Aim of the Work

This thesis aims to create a proof-of-concept application that displays the advantages of using a Model Synthesis-based procedural generation approach. For this the WFC algorithm is adapted and extended. An overview of the WFC algorithm in comparison with our system is shown in Figure 1.2.

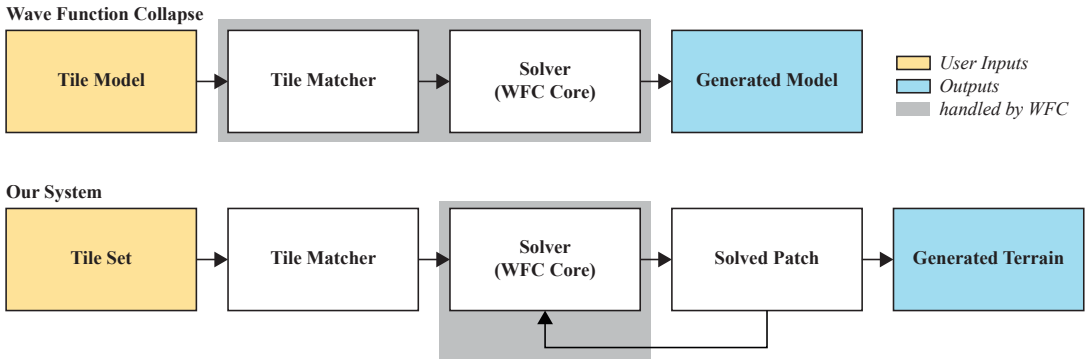


Figure 1.2: Visualization of the steps of the WFC algorithm (top) in comparison with our system (bottom).

As can be seen in the figure, WFC takes a tile-model as input. This model is composed of different tiles that are placed in a 3D lattice. The same tile can occur at multiple positions. The number of occurrences as well as the placement of the neighboring tiles is used to determine the matching information by the *Tile Matcher*. Afterwards WFC solves a rectangular area with matching tiles that resemble the input tile-model. A more in-depth explanation of the WFC algorithm is given in Section 2.2.

Our system does not use a tile-model but a tile-set as input. This means, that the tiles are supplied as a set of different meshes without taking spatial or quantitative information into account. The matching information is then purely calculated on the geometry of the tiles. This is done because the additional generation of a sample model, although useful for the universality of the algorithm, creates an unnecessary overhead for the user. To achieve this geometric matching the tile matcher of the WFC was replaced by our own algorithm described in Chapter 3.

The other important contribution of our system is the generation of arbitrarily large terrains directly usable for interactive applications. This means that the WFC solver is not just used once to generate a model but instead generates patches of terrain, called chunks, in such a way that they fit together to form a seamless terrain surface. In combination with a seeded pseudo-random generator this also leads to the possibility of continuously generating new terrain on run-time in a deterministic fashion such that given the same seed the same terrain is generated. This terrain generator, the solutions to propagating the matching constraints over chunks and the generation in a deterministic fashion are explained in Chapter 4.

CHAPTER 2

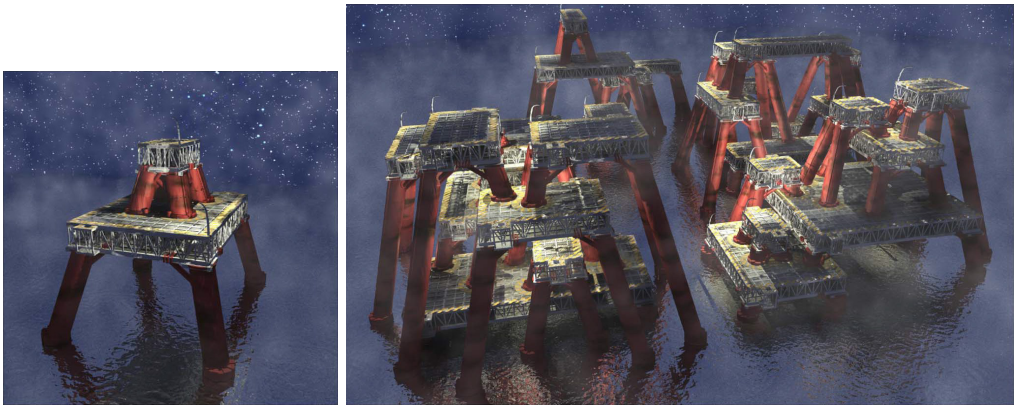
State of the Art

In contrast to established methods for procedural generation like *L-Systems* [Pru86] or generation of geometry by using *Marching Cubes* [LC87] on *Perlin Noise* [Per85] the following methods do not depend on domain-specific knowledge of certain grammar rules or noise parametrization.

The general approach in the two main methods shown in this section is to provide a set of input objects and constraints of how these input objects should be arranged in relation to each other. This means that only the input, for example, in the form of geometric building blocks, has to be provided.

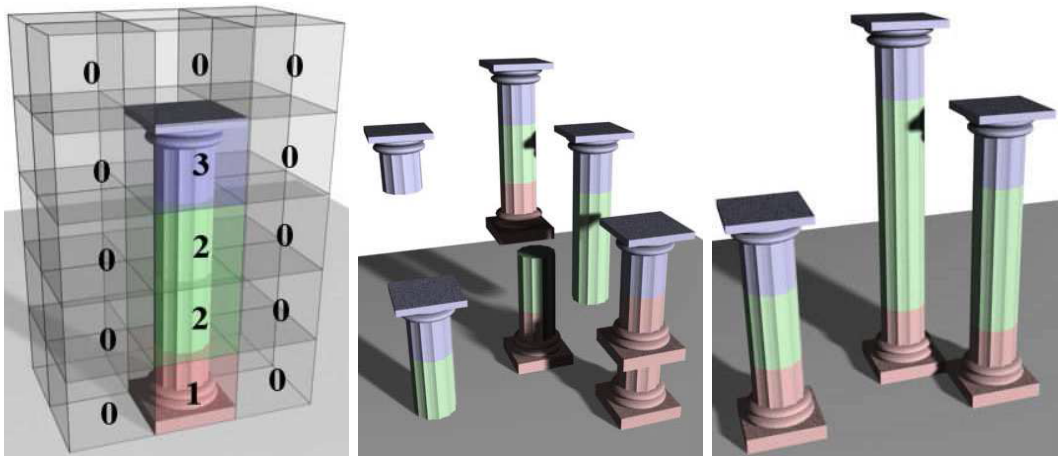
2.1 Model Synthesis

The first of these methods is *Example-Based Model Synthesis* [Mer07]. This method uses an input model and generates an arbitrary large output model (see Figure 2.1). In a very similar fashion, this is also done in the heavily studied field of texture synthesis hence the name model synthesis. The input model is split into *model pieces* using a regular 3D grid. Model pieces with the same geometry are treated as one logical model piece. For example, in Figure 2.2a there are four different model pieces: air (0), the bottom piece of the pillar (1), the center piece (2) and the top piece (3). Now that the basic building blocks of this method are established, one could try to arrange them in a 3D grid. However, without further knowledge of the structure, one has to place them randomly which would result in an unwanted outcome like in Figure 2.2b. To get a visually consistent result with the input model the local neighborhood of every block has to be analyzed. For every piece, the neighboring pieces in all six directions are observed. The output model is only allowed to put pieces next to others if the same arrangement is also present in the input model. If this constraint is fulfilled for the whole output model it is called *consistent* (see Figure 2.2c).



(a) (b)

Figure 2.1: A simple input model (a) can generate a complex output model (b).
Source: [MM11]



(a) (b) (c)

Figure 2.2: A pillar is split into basic model pieces (a). They can be arranged randomly (b) or placed with regard to the neighborhood constraints (c)
Source: [Mer07]

To generate the output a 3D grid is generated where every cell is initially marked as unassigned. Every cell can only contain one label. The cells are then assigned labels corresponding to the model pieces. To place the labels in a way that satisfies the consistency constraints a transition function is defined. This function takes two labeled cells as input and returns zero if they are adjacent to each other and violate the consistency constraints. If the cells are adjacent to each other and satisfy the constraints or if they are not adjacent the function returns one. A model is formally consistent if for every cell pair the transition function evaluates to one. The transition function is automatically

generated from the input model so it returns one if and only if two labels are also adjacent in the input model.

Now the grid is populated using a set of possible labels for every cell of the model M called $C(M)$. This set is also called the possibility space. At first, $C(M)$ holds all labels for every cell. Then for one cell a starting label is chosen. This could reduce the number of possible labels for the adjacent cells that satisfy the transition function. Therefore these cells have to be updated in $C(M)$. If one of these cells changed, its possible labels in $C(M)$ and also its adjacent cells have to be updated (see Figure 2.3). This whole process repeats until no cell has to be updated anymore. Then a next cell where the label is not yet decided is chosen and it is assigned one of its possible labels from $C(M)$. This process runs as long as $C(M)$ contains only one label for every cell.

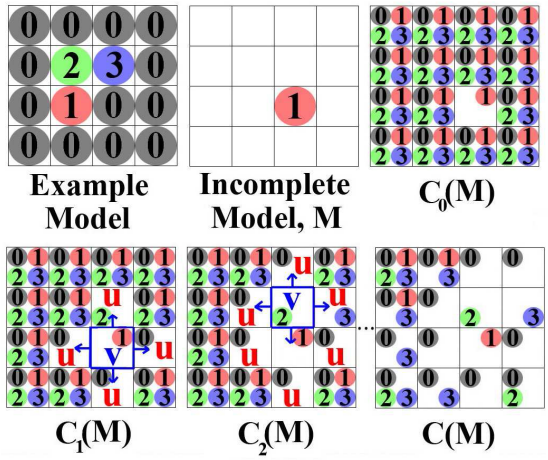


Figure 2.3: The example model defines constraints to how $C(M)$ is updated when one cell in M is assigned a label.
Source: [Mer07]

This process however has a drawback. For one iteration step, it has to execute a global update which can lead to having to update every cell of the model. In this method deciding if the model has a consistent solution when setting one cell to a fixed label is NP-complete as shown by Merrell [Mer07]. To avoid this the before mentioned algorithm is adapted to start with a fully consistent model containing the same label in every cell. This is often the "empty space" label as it is allowed to be adjacent to itself in every direction. Then a small subregion B is chosen where $C(M)$ is calculated. When all labels for B are decided, B is shifted one unit and the process repeats (see Figure 2.4). This works similar to filter kernels. If the subregion calculation was applied to every possible position for B, the model is finished.

It is however not guaranteed that the algorithm calculating the subregion exits successfully. A seemingly allowed label placement can lead to arrangements where unassigned positions in the possibility set are empty. This means that the calculated subregion has

conflicting constraints and is therefore inconsistent. In this situation, the subregion is discarded and the next subregion calculation, with an offset to the previous subregion, is started. Merrell states that conflicts are less likely to occur when using small subregions. Additionally, every cell position is recalculated multiple times because neighboring subregion calculations overlap each other. Therefore, in the case of a conflicting subregion, it can be dropped without restarting the synthesis process for the whole model.

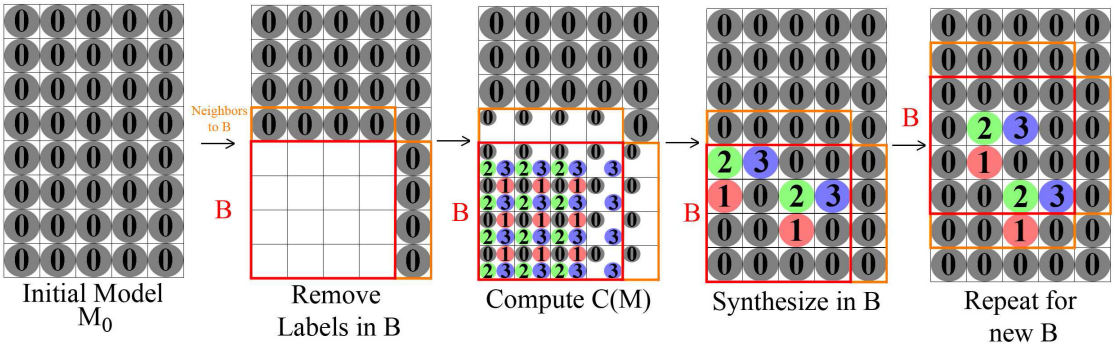


Figure 2.4: A small subregion B is used to modify the model M so it stays consistent all the time but without searching the whole global space in every iteration.
Source: [Mer07]

2.1.1 Additional Constraints

Example-based model synthesis, as well as continuous model synthesis, incorporate only one type of constraint, the adjacency constraint. This leads to globally uniform distribution patterns. In *Constraint-Based Model Synthesis* [MM09], different constraints used in CAD modeling [Aul99] are incorporated, which allow the generation of visually varying output on a global scale as well as giving the user more creative control to specify the output model (see Figure 2.5). The constraints used in constraint-based model synthesis apart from the adjacency constraint are:

- **Dimensional constraints:** Constraining the size in every dimension by a specific range
- **Algebraic constraints:** Constraining properties by mathematical functions, for example, the width of an object has to be at least twice as long as its height.
- **Connectivity constraints:** Certain parts of objects have to be connected (like road networks).
- **Incidence constraints:** Constraining the number of vertices of a face to allow for more complex objects.
- **Large-scale constraints:** Constraining where parts of the model are allowed to be placed.

These constraints are evaluated at different points in the synthesis algorithm which is adapted from continuous model synthesis by changing the probabilities of the possibility set. The target of constraint-based model synthesis is to create not only larger models but also ones with a higher visual variety than the input. This however means, that the information to generate this increased variety has to be supplied additionally to the input model. So other than for the adjacency constraints the advanced constraints cannot be inferred from the input model itself. Therefore the user has to provide these constraints.

Figure 2.5: Large-scale constraints leading to the model being laid out in the form of letters. Also dimensional constraints can be seen, for example, in the letter G, some parts are more horizontally laid out.
Source: [MM09]

2.2 Wave Function Collapse

A more recent method for applying constraints to level generation is an algorithm influenced by model synthesis [MM08] as well as a texture synthesis algorithm [Har00]. It is called *Wave Function Collapse* (WFC) and was created by Maxim Gumin who also provides a reference implementation in C#. Still, exploring this algorithm is quite difficult as there is no paper from the original author. However, Karth and Smith [KS17] analyzed the algorithm as well as the rapidly growing indie-game developer community around this algorithm. This part of the report is therefore based on this analysis.

Generally, Maxim Gumin designed the algorithm for the task of texture synthesis, although his code does not enforce two-dimensionality. It was also applied to one-dimensional as well as higher, like three-dimensional, synthesis applications and is therefore heavily used in level generation [KS17].

The name wave function collapse hints at the integral part that was already explored by model synthesis. Namely, that a possibility space is created and at some point one possible state of a cell is chosen. This is called *observing* - also hinting at quantum mechanics - in Maxim Gumin's source code.

Then the incremental process of reducing the possibility set, called *wave* in Gumin's code, by deciding values for cells and then propagating their constraints starts.

2.2.1 Algorithm

Karth and Smith identified four distinct steps in the algorithm [KS17]. These are:

1. Extracting local patterns from the input
2. Aggregate these patterns to an index for fast constraint lookup
3. Incrementally generating the output by eliminating states in the possibility set
4. Generating the output from the fixed assignment set

These patterns resemble the adjacency constraints in model synthesis. To identify them a N^D kernel is used to sample the input, where N is a chosen size and D is the dimensionality of the input. Optionally, rotations or mirroring of these patterns can be taken into account. This means that, for example, only one configuration of a corner has to be in the input and the algorithm is still able to form an enclosed shape. After that, an index is calculated which contains the adjacency constraints meaning that it returns whether two patterns can be placed next to each other given a specific offset. This is more versatile than in model synthesis where the transition function only differs between adjacent and non-adjacent labels.

It is also possible to completely skip the pattern building process so that there is no consistent input from which patterns are generated but users directly provide the patterns and their adjacency constraints. This is employed in most practical applications in the case of tile sets. Directly providing tiles however, leads to a significant difference to model synthesis. In model synthesis it is always possible to generate a consistent model because the input model is already the simplest example of a consistent model matching the adjacency constraints. Provided tiles could be unable to generate a consistent model if their constraints lead to unmatchable configurations.

If the WFC algorithm runs into situations where no assignment is possible for a cell, it restarts its whole generation process. This means that it does not employ backtracking. Karth and Smith [KS17] showed that this decision was taken because in practice conflicts happen very rarely.

WFC chooses which cell it assigns a pattern next by selecting the cell with the lowest possible assignments. This heuristic is commonly known as *minimum remaining values* heuristic in constraint solving [KS17]. When the algorithm selects a cell, it chooses a random sample of the possible patterns for this cell, weighted by their frequency, in the input. This leads to outputs with similar pattern distributions as the input. Exactly like in model synthesis, after assigning a pattern to a cell, the possibility set/wave is updated

to contain only possible patterns given the current final assignment. As noted by [KS17] this constraint propagation proceeds like a flood-fill algorithm.

Like in the naive version of the model synthesis algorithm this constraint updating is done in the full global space.

CHAPTER 3

Tile Matcher

In this chapter the structure of the tile-matcher will be explained. The Tile Matcher is the component that analyzes the tiles given as input and creates matching information from them. Tiles are the basic units that get combined to form the terrain. The matching information defines the connectivity constraints for the tiles for the application of the WFC solver. The connection constraints state whether two tiles are allowed to be placed next to each other. This placement can also include the tile's rotation. An example of these constraints can be seen in Figure 3.1. Depicted is a 2D tile with four edges, each having one of two possible states (orange or green). If both adjacent edges match (having the same color in this example) the placement is considered valid. The output of this stage is a dictionary, giving the information needed to respond with a result for every possible input, consisting of the tiles from the tile-set T and their faces F, to the matching function m:

$$m((t_1, f_1), (t_2, f_2)) = \begin{cases} 1, & \text{if side } f_1 \text{ of } t_1 \text{ may be placed next to } t_2 \text{ on side } f_2 \\ 0, & \text{if side } f_1 \text{ of } t_1 \text{ may not be placed next to } t_2 \text{ on side } f_2 \end{cases}$$
$$t_1, t_2 \in T \quad f_1, f_2 \in F \tag{3.1}$$

Usually, as described in Chapter 2, Model Synthesis algorithms like WFC create their matching information by analyzing an example model given as input. This example model is constructed of tiles allowing the same tile to occur multiple times. The frequency of a tile as well as it's neighborhood relation is then used as input for the WFC solver. The problem with this approach is that with an increasing number of tiles the sample model gets more and more complex since every combination of allowed neighborhood arrangements has to be included. Eventually, the sample model has to cover amounts of cases the designer cannot check manually. This results in a high possibility of tile pairs

Figure 3.1: Two tiles placed next to each other. On the left, the placement results in an invalid configuration due to the different colors along the connecting edge. On the right, the neighborhood constraints match.

that are intended to be valid, not being contained in the sample model and therefore not getting chosen in the model synthesis algorithm.

Our improved tile-matcher takes a different approach for creating these constraints. It uses a tile-set as input that has no prior information about neighboring tiles or their frequency. Instead, the matching information is extracted from the geometric properties of the tiles without requiring additional information. The algorithms considers two sides of a tile to be matching when the geometry on the faces is similar. This includes vertices as well as the edges and the face normals of mesh-faces orthogonal to the tile-face.

3.1 Matching

Tiles are cubic pieces containing 3D geometry consisting of vertices, edges and faces. For simplicity, these tiles are placed in a 2D lattice although an extension to feature a 3D placement would be easily possible. Therefore matching information only needs to be calculated on the faces of a cube representing the boundary of the lattice cell in which the tile is placed. This means that matching information is not calculated for the top and bottom face of the tile. This leads to four faces of the tile, looking in the cardinal directions along the x- and z-axis, where neighboring tiles can connect. The tiles can be visualized as squares (viewed from the top) with each edge having assigned matching information as depicted in Figure 3.2.

To decide which tiles are allowed to be placed next to each other, matching information is calculated for every considered face of the tile's bounding box. This matching information serves as a decoupling mechanism because deciding if two tiles can be placed next to each other is reduced to a comparison of the matching information instead of pre-computing a binary decision for every tile pair. Therefore the complexity of finding the complete matching information for a set of tiles is $O(n)$ with n being the amount of matching information of the whole tile-set. In our case, one tile has four faces that are considered for matching. Given a tile-set with t tiles, this means that $n = 4 * t$.

Furthermore, four different rotational states have to be considered per tile (0, 90, 180 and 270 degrees). For this however, no additional information is needed because the

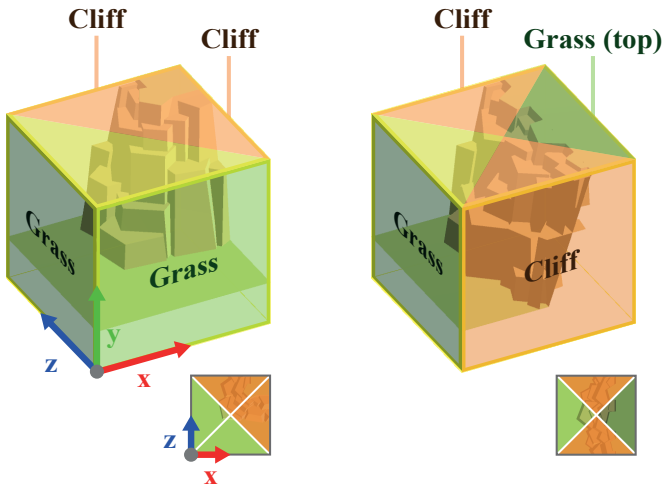


Figure 3.2: Two tiles with example matching information for the four sides in x/z direction.

calculated matching information is rotational invariant. The rotated faces can be accessed by offsetting the face number when retrieving the face. For example, the face showing in the positive x-direction in the un-rotated tile is the face showing in negative x-direction when rotated 180 degrees.

3.1.1 Symmetry

Faces are categorized as either symmetrical or asymmetrical ones. This distinction is needed because placing of asymmetric faces next to each other requires one of the faces to have the geometry mirrored. The information is encoded in the matching information and used in the matching process. In the symmetric case, matching is trivial. The neighboring face only has to have the same matching information. In the other case however, a distinction has to be made. Asymmetrical faces get a winding direction assigned to them that is either clockwise or counterclockwise. Rotating the tile retains the same winding direction for the face. The rest of the matching information is calculated in a such a way, that it results in the same data for mirrored faces except their winding direction. How this is achieved is later described in Section 3.2.3. When choosing possible neighbors of a tile on a asymmetric face, the other tile has to have a face with the opposing winding direction and otherwise same matching information. If this is true, then the two tiles are allowed to be placed next to each other (see Figure 3.3).

3.2 Matching Information

The matching information is the per-face data of the tiles bounding box to decide which other tile-faces are allowed to be placed next to it. The matching information consists of

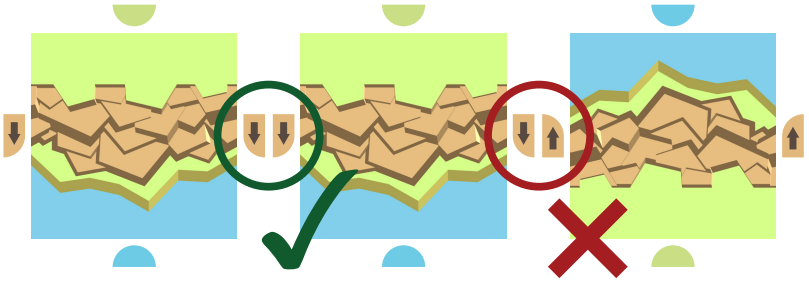


Figure 3.3: Connection example with asymmetrical faces, if the winding does not match the placement is invalid (right).

two parts, a hash as well as directional information. This hash is called face-hash and is used to represent the geometric information of the mesh on a side of a tile as one numeric value. For this, all edges, as well as the faces of the mesh, are considered. If both, the start and the end vertex of an edge lie within a small distance ϵ of the face plane, it is added to the list of edges that contribute to that faces hash as shown in Figure 3.4. When all edges are processed and either ignored or assigned to one or multiple faces, the hash of the tile-faces can be calculated. First, the hashes of each edge are calculated and then combined to create the hash of the tile-face.

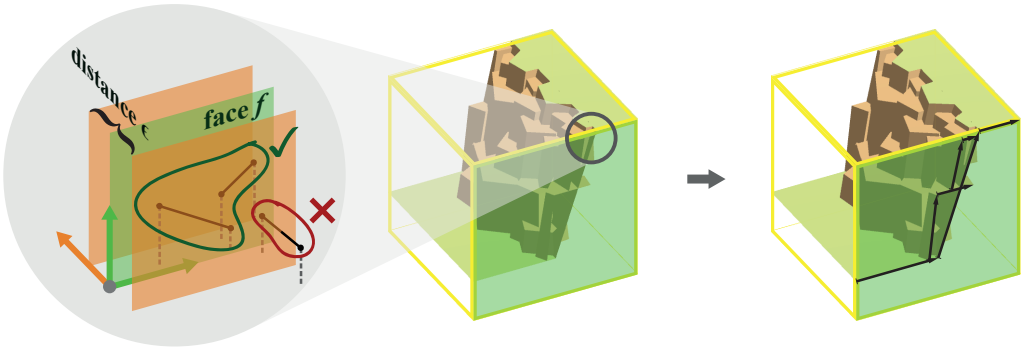


Figure 3.4: The edges along the selected face (green) are marked (black arrows)

3.2.1 Edge Hash

To calculate the hash of an edge its vertices are first transformed from the 3-dimensional coordinates inside the tile into a 2-dimensional space spanned by the respective tile-face as seen in Figure 3.5. This is done by projecting each edge to the current tile-face. After that, the four floating point numbers (x- and y-values for both vertices, v_1 and v_2 describing the edge $\overline{v_1 v_2}$) need to be quantized. This is done with a user-specified quantization-grid size. The primary requirement of the edge hash $h_{\overline{v_1 v_2}}$ is that two similar

but different inputs should lead to different hashes. Also, different edges should not lead to the same hashes. Therefore the simple addition of the quantized values is not sufficient as this would lead to a large number of collisions. To ensure a large spread while still remaining easy to compute, a hash calculation inspired by Kernighan and Ritchie [KR17] as shown in Equation 3.2 was chosen.

$$h_{\overline{v_1 v_2}} = (((((x_1 * p) \oplus y_1) * p) \oplus x_2) * p) \oplus y_2 \qquad (3.2)$$
$$p \text{ is prime}$$

This approach however introduces an unwanted order dependency of the values, meaning that an edge $\overline{v_1 v_2}$ between two example points v_1 and v_2 will not result in the same hash as the inverse edge $\overline{v_2 v_1}$. To calculate the same hash for either case the point with the lowest sum of x- and y-coordinates is chosen as the first point.

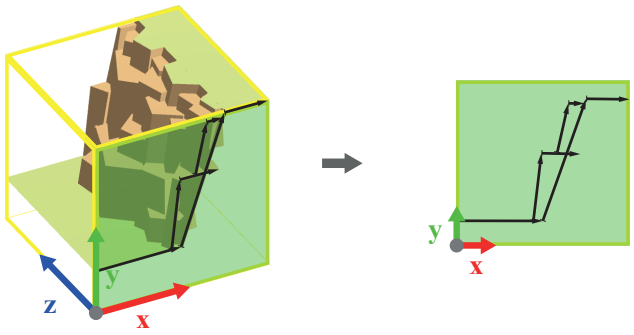


Figure 3.5: The edges are transformed into the face-space.

3.2.2 Face Hash

The face hash is calculated as the mere sum of all of the face's edge hashes. This is done because the order of the edges is not guaranteed to be the same among faces with matching geometry. To improve the distinctiveness of the outputs, the result is multiplied with a prime and XOR-ed with the number of edges of the face. This is used to not only rely on the distinctiveness of the edge hash. A face could have the same geometry as another face in a visual sense but with different amounts of edges. Even if the value calculated from the hashes is naive, like just taking the sum, for example, the multiplication of the face hash with the number of edges still guarantees a different face hash.

3.2.3 Mirror Hash

As already mentioned in Section 3.1.1, a face with asymmetrical geometry seen from the vertical mirror axis needs a face with the mirrored geometry next to it. This requirement is illustrated in Figure 3.6. To encode this in a hash, the faces are split into two categories. Faces that are symmetrical and ones that are asymmetrical. To categorize the faces, the face hash is calculated two times per face. Once seen from the front and once from the back. If both hashes are the same, the face is considered symmetrical. If not they are labeled as asymmetrical and get an additional winding direction assigned to them that is either clockwise or counterclockwise. The choice is made by comparing the two calculated hashes with each other. Depending on whether the smaller numerical hash was the mirrored hash or the normal one the direction is chosen. This choice is arbitrary as long as it is deterministic and results in the same assignment for every tile. For the hash value of the face also the smaller one of the two calculated hashes is chosen.

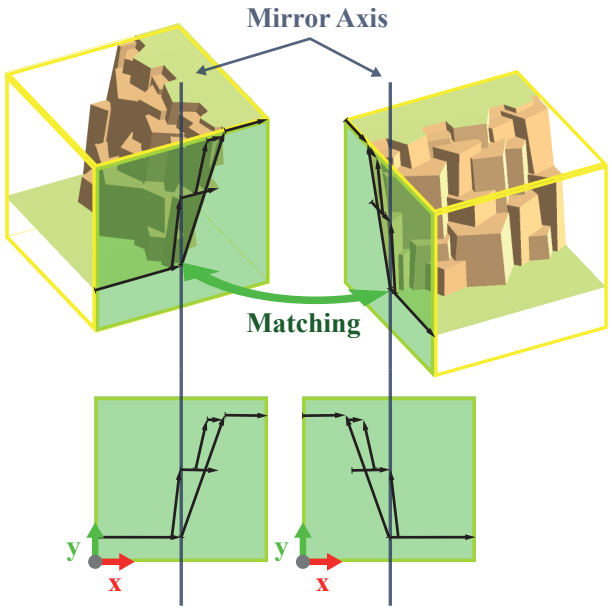


Figure 3.6: For two faces to match the opposite face has to have the mirrored geometry.

3.3 Propagator Construction

The matching information has to be translated into WFCs constraint representation which is called *propagator* in Karth and Smith [KS17]. This propagator is a three-dimensional array where the first dimension represents the direction, the second one being the tile who's matching is described and the third one is a list of all matching tiles.

To create the propagator, a dictionary, indexed with the matching information, is created. The dictionary contains lists stating which tiles fit the given matching information. As WFC has no sense of rotation, tiles have to be added multiple times, once for each of the four rotation configurations. A tile in the propagator is therefore referenced with its tile-index built by combining the tile-number n, which is the original order in the input tile-set, and the rotation configuration (between 0 and 3) r as $n * 4 + r$.

For every tile, all of the four face's matching information is retrieved. If the face is a symmetrical face, the face-hash is simply looked up in the dictionary and the list of fitting tiles is entered in the propagator. If the face is an asymmetrical one, the dictionary is looked up for tile-indices with the same face-hash but an opposite winding direction.

With this information, the WFC is ready to solve an instance of the matching problem.

CHAPTER 4

Terrain Generation

With the tile-matcher, the initial information to apply the WFC algorithm is given. In its unmodified version, the algorithm allows filling of a rectangular lattice of arbitrary size. For this thesis, the WFC algorithm, as well as the terrain generation in general, was modified to fulfill properties that make it useable for terrain generation in video games and other interactive media. These adaptions include the generation of infinitely large terrains, the seeded pseudo-randomness of the generation allowing for a deterministic result across invocations and the possibility of generation at run-time. Additionally, the generator also allows changing tile probabilities for the WFC algorithm such that the terrain has a higher visual variety and is not just a connected collection of uniformly distributed tiles. Given these requirements, the algorithm has to have the following properties:

Infinite Terrain

Infinite large terrain means that if a certain position of the terrain is requested, it is possible to serve the requested area. This requested area has to adhere to the connection constraints of the bordering area that is either already generated or not. High emphasis is put on the case where the bordering area is not already generated because in that case, it should be possible to create the requested terrain with minimal generation of neighboring terrain outside the requested area.

Seeded Pseudo-Randomness

The generation of the procedural terrain is randomized with a deterministic random generator that can be seeded. With that, the same terrain can be regenerated on multiple executions of the program, so the terrain information does not have to be stored on the disk. Also in cases where multiple machines have to render the same terrain, for example, in a multi-player video-game, the networking code only has to exchange the random seed

once at the connection. After that, there is sufficient information to generate the terrain identically on both machines.

Run-Time Generation

Generating the terrain at run-time should be possible. This means that the terrain has to support adding unexplored areas without seams and also that generating those new areas is done without interrupting the display of the terrain. Ideally, the generation of new terrain is as fast as the maximum possible camera speed such that terrain is generated without stuttering when the camera traverses it.1

Customization

The terrain generation should also offer methods of modifying or configuring the generation process to create a higher variety of possible landscapes. Particularly, it should be possible to change the probability for certain tiles to be placed based on their location.

4.1 Chunks

A chunk is a square region of neighboring tiles as can be seen in Figure 4.1. All chunks consist of the same number of tiles. However, this number can be calibrated to optimize the performance. It is used to subdivide the terrain into larger components than tiles and is the unit the infinite terrain generation works on.

Chunks also play a critical role in the proposed run-time terrain generation. In this use-case, a certain number of chunks are held in memory. When the point of reference (the player or the camera in most cases) moves, new chunks in direction of the movement are generated while the in-memory chunks that have the greatest distance to the player get unloaded.

In the implemented generator one chunk corresponds to one complete solution of a WFC instance. Generating one such chunk is a trivial task as it just runs the WFC algorithm on a small confined area.

4.2 Constraint Propagation over Chunk Borders

Tile constraints, meaning the constraints of only using matching tiles as neighboring tiles, have to be satisfied throughout the whole terrain. While satisfying these constraints within a chunk is already completely handled by the WFC algorithm, keeping and propagating them over multiple chunks is a more complicated endeavor that is not solved by traditional algorithms. Therefore a new method for ensuring common edges between chunks was developed.

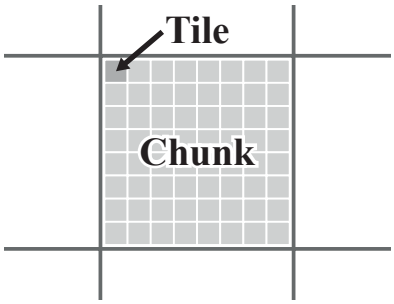


Figure 4.1: Chunk with 8×8 tiles

Tiles that are on the edges of a chunk have to match the adjacent tiles of the bordering chunks, as seen in Figure 4.2. In the special case of the four tiles in the corners of a chunk, these even have to consider two tile constraints from two different chunks.

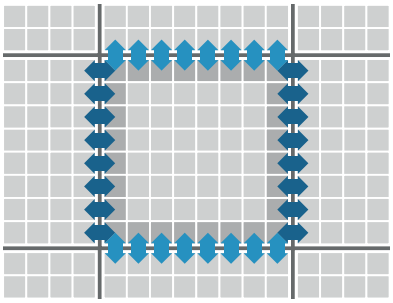
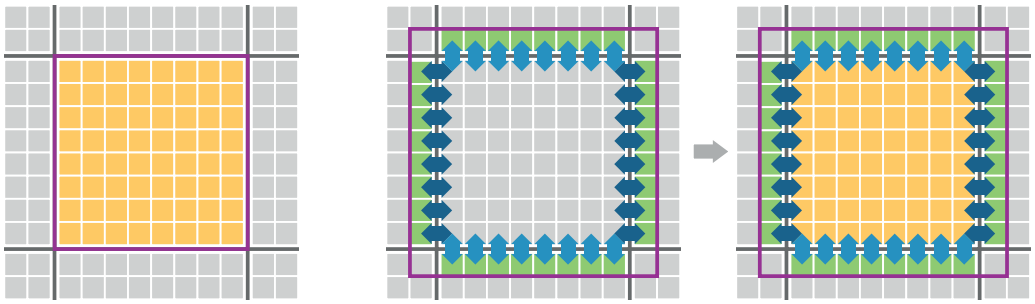


Figure 4.2: Border tiles of a chunk (dark gray) have to match the neighboring tiles (blue arrows)

4.2.1 Constraint Injection

To solve a stand-alone chunk, a WFC instance can be created that solves an area with the same size as the chunk. To inject neighbor constraints into the chunk generation, the WFC instance has to solve an area one tile larger on every side (see purple border in Figure 4.3). Normally the WFC algorithm starts with an *observe* step as explained in 2.2.1. To inject the constraints however, the adjacent tiles from the neighboring chunks are inserted into WFC's *wave* array after which the *propagate* step of the WFC algorithm is executed. Only after that, the usual loop of alternating *observe* and *propagate* is started. The reason for this is evident. First, the possibility set on the positions of the injected tiles are set to the value of the injected tiles as seen in Figure 4.3b (green tiles). Then the propagation is executed, which updates the possibility set in such a way that the inner cells - the positions of the actual tiles of the chunk (depicted in yellow in Figure

4.3b) - are now restricted to only allow tiles that match the neighboring constraints. After this, the WFC algorithm can proceed normally. However, when transforming the WFC solution and spawning the tiles of the chunk the added border of one tile per side has to be ignored.



(a) (b)

Figure 4.3: Difference between normal WFC solving of a chunk (a) and with constraint injection of neighbors (b).

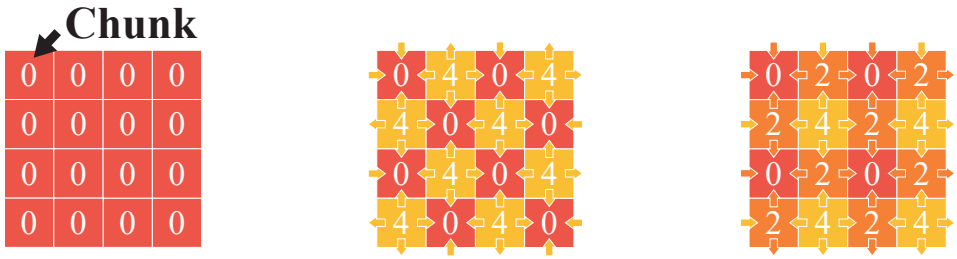
4.2.2 Chunk Layout

To resolve the neighborhood constraints across chunk boundaries, the chunks are not generated independently from each other. Every chunk relies on up to four of its neighbors to be generated. This hierarchical structure of chunk generation can be seen in Figure 4.4. Depicted are three different layouts. The number inside the chunk denotes how many other chunks a chunk depends on. Figure 4.4a depicts the naive approach of generating every chunk on its own without integrating neighboring chunks. These chunks are called class-0 chunks because they rely on zero other chunks. The naming scheme was chosen to directly denote on how many neighboring chunks a chunk depends on for generation. The naive version results in completely random chunk borders where the tiles do not match at all.

The chunk layout in Figure 4.4b is more sophisticated. It alternates such that every other chunk depends on all four of its neighbors (hence it is called class-4 chunk), much like a checkerboard. This means that if the generation of a chunk at a chunk-position with a class-4 chunk is requested, the four adjacent chunks, which are class-0 chunks, have to be generated at first. After that, the neighboring tiles of the recently generated class-0 chunks are incorporated into the generation of the original class-4 chunk. However, this chunk layout configuration has a significant drawback. The corner tile of a class-4 chunk has to solve two adjacency constraints from two different class-0 chunks at the same time.

Generally, tile-sets are non-complete. This means that if the four neighbor tiles of a tile-position are filled with random tiles from the tile-set, there is not always a tile in the set that can be placed at the tile-position and satisfies the adjacency constraints of the

neighbor tiles. In the case of a class-4 chunk, this means that already two of the four neighbors of a tile are defined. This restricts the number of possible tile choices for the corner place of the class-4 chunk drastically. With this chunk layout, the corner problem is only solvable if there exists a tile for every possible corner constraint combination that has the matching constraints on two adjacent tile-faces.



(a) The naive layout consisting only of class-0 chunks.

(b) Checkerboard layout with class-4 chunks.

(c) A refined version of the hierarchical chunk generation layout.

Figure 4.4: Different chunk class layouts.

To reduce this problem, the final chunk layout depicted in Figure 4.4c was chosen. It has three different classes of chunks that are generated hierarchically. First, the class-0 chunks are generated at every position with even x- and y-coordinates of the terrain plane. Then the chunks adjacent to these chunks are generated. These class-2 chunks are chunks that have one even and one odd coordinate value, meaning that they are either horizontally or vertically connected to two class-0 chunks. After these class-2 chunks are generated with the constraints of their class-0 chunk neighbors, the remaining chunks in the layout - the positions with odd x- and y-coordinates - are populated. These chunks are class-4 chunks like in the previous layout.

With this approach, the earlier mentioned problem with the choice of the corner tile is reduced from every second chunk to every fourth chunk and does not rely on class-0 chunks anymore. This is important because class-0 chunks are used as stable reference points for the deterministic terrain generation. The problem is not eliminated with this layout though. However it allows employing hierarchical backtracking that was not previously possible.

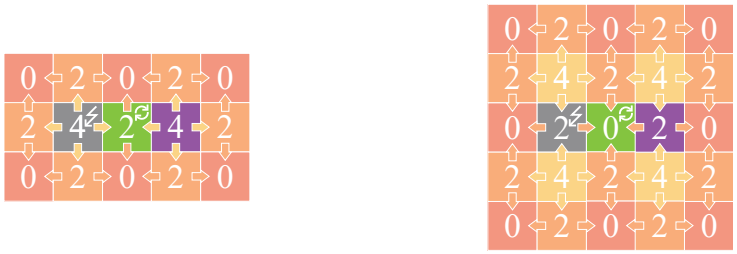
4.2.3 Chunk Generation Backtracking

If a chunk fails to be generated, a multitude of strategies can be employed to recover the consistency of the terrain. A chunk-fail is detected if the WFC algorithm encounters a position where none of the possible tiles can be placed to resolve all bordering neighborhood constraints. As the WFC algorithm itself does not employ backtracking the simplest reason for failure is just the order of tile and position choices that let the WFC algorithm run into a dead-end. Therefore the first recovery technique is to restart

the WFC algorithm with a different seed value leading to a completely different order of tile choices. To sustain the deterministic property of the whole algorithm, this new seed is also generated using the original seed value. This process of regenerating the chunk is done in a loop until either a valid solution is found or a predefined limit of retries is reached. In the latter case, hierarchical chunk generation backtracking is utilized. As a shortcut, chunk generation backtracking can also be applied if the WFC algorithm directly fails after the neighboring tile injection in the first step. Such a failure implies that there exists no possible tile to resolve the corners of the chunk. The backtracking of chunk generation works by selecting one of the neighboring chunks and regenerating it. Afterwards, the WFC algorithm is applied to the current chunk again. If it still fails, another neighbor is chosen.

4.2.4 Chunk Regeneration

If the chunk generation backtracking selects a chunk to be regenerated, it is not possible to generate it with the same neighbor constraints as before, because that could invalidate the consistency of the terrain on the sides without neighbor constraints. Regeneration only happens to lower chunk classes. Applied to the chunk class layout described in section 4.2.2, this means that if a class-4 chunk fails all neighboring class-2 chunks are possible candidates for regeneration but in the seldom case that a class-2 chunk fails only the class-0 and not the class-4 chunks are regenerated. Both cases, depicted in Figure 4.5, have to consider neighbors that are not hierarchically dependent (purple chunks in the image).



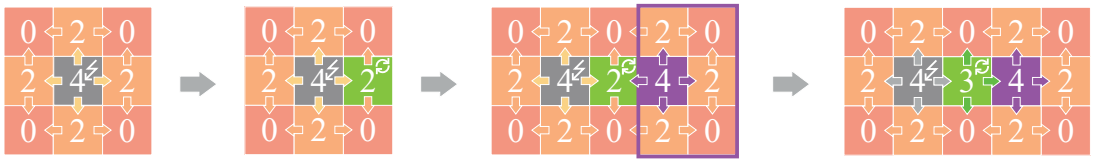
(a) (b)

Figure 4.5: The two types of chunk regeneration. Regeneration of a class-2 chunk (a) and a class-0 chunk(b). Depicted is the failed chunk (gray) the regenerated chunk bordering it (green) and the chunk that has to be generated before the regeneration chunk can be regenerated (purple).

Regeneration of a class-2 chunk

A class-2 regeneration is triggered by a failed class-4 chunk (as seen in Figure 4.6). In this case, up to three of its borders have to match the neighboring chunks and one has to be populated randomly. Trivially the two borders that connect to the class-0 chunks

have to match again just like in the initial hierarchical generation process. The border that connects to the chunk that triggered the regeneration (gray) has to get a new set of tiles as well. This is necessary because otherwise, the regeneration would not have any impact other than randomizing the inner tiles and would not help at solving the problem. The remaining border, the one on the opposite side of the triggering chunk (purple), has to match the neighboring chunk as well. This is another class-4 chunk, so it is downstream in the generation hierarchy. Choosing random tiles for the border tiles of the regenerated class-2 chunk (green) adjacent to the class-4 chunk (purple) would require regeneration of this class-4 chunk. That again could trigger a whole generation backtracking process. Without keeping the neighborhood constraints on the opposing border, the terrain generation as a whole would not suffice the target properties anymore. The result would be a global influence of the regeneration sub-system. Therefore one would have to generate the whole terrain, to make sure that one chunk does not trigger a chain of regeneration that completely changes the landscape. Obviously, generating the whole terrain is not possible when generating an infinite terrain. Keeping the opposing constraints intact results in the regeneration being contained as a phenomenon with local implication. With this approach, already generated chunks may be changed by the regeneration, but its maximum impact is limited to one tile in every direction. Therefore if an area is requested from the terrain generation, it just has to generate a slightly larger area around the border that is smaller or equal to two chunks thick. Although the regeneration has an impact radius of one chunk (straight as well as diagonally), two chunks are required because otherwise the information on how to constraint the opposing side would be lacking.



(a) Chunk generation fails (gray).

(b) Regeneration chunk is selected (green).

(c) Additional chunks are generated to get all needed constraints (purple).

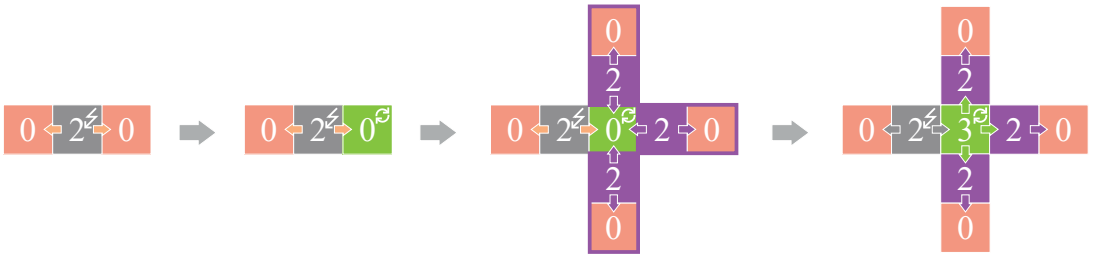
(d) Regeneration chunk is generated and failed chunk is retried.

Figure 4.6: Regeneration steps of a class-2 chunk.

Regeneration of a class-0 chunk

A much rarer case is the regeneration of a class-0 chunk (see Figure 4.7). This is needed if the hierarchical generation already fails at the second step while generating a class-2 chunk between two class-0 chunks. In this case, one of the class-0 chunks has to be selected. Afterwards, the remaining adjacent class-2 chunks have to be generated. Those themselves require the generation of additional class-0 chunks on their opposing border of the regenerated class-0 chunk. With this information, the regenerating class-0 chunk

can now satisfy the connection constraints of three of its borders, all but the border to the chunk that triggered the regeneration.



(a) Chunk generation fails (gray).

(b) Regeneration chunk is selected (green).

(c) Additional chunks are generated to get all needed constraints (purple).

(d) Regeneration chunk is generated and failed chunk is retried.

Figure 4.7: Regeneration steps of a class-0 chunk.

4.3 Tile Probabilities

The WFC algorithm allows defining different weights for each tile. This weight defines the probability of choosing the tile when multiple tiles are possible for a certain position. In the initial version of model synthesis [Mer07] as well as in the original application of the WFC algorithm [KS17], these probabilities would have been proportional to the number of occurrences in the sample model. Because the tile matching does not need a sample model, it would be inconvenient to add it only to calculate the tile weights. Therefore the choice of defining tile probabilities is given to the user. It is possible to assign different probabilities for every tile.

Even with different tile probabilities, the distribution of the tiles over the terrain is still constant. To break with that uniformity, an arbitrary amount of probability sets can be defined. Each probability set contains a probability value for every tile. At chunk generation, one probability set and therefore the specific probability of every tile of the tile-set can be selected. How this selection happens can be adapted. It can be done with a simple pseudo-random generator or more sophisticated methods like noise functions.

CHAPTER 5

Results

The tile-matcher, as well as the terrain-generation system, were implemented using the Unity[1] game engine (2018.2.13f1). Additionally, multiple tile-sets were created to ensure the soundness of the approach and the correct handling of edge cases even with larger tile-sets. The prototype offers its user a straightforward approach for integrating tile-sets and generating a region of terrain with adjustable parameters like the chunk size. A sample terrain with a complex tile-set can be seen in Figure 5.1.

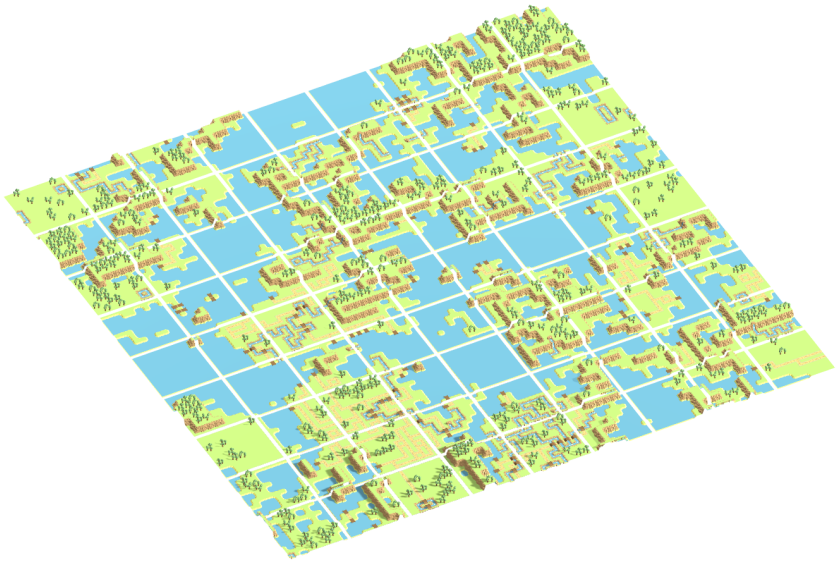


Figure 5.1: Large terrain generated from a complex tile-set with visible chunk borders.

[1] https://unity3d.com/

5.1 User Workflow

The user workflow is the sequence in which the user steps through the different subsystems. An overview graphic of the workflow is shown in Figure 5.2. At first, the user has to supply a 3D-model of the tile-set. If the 3D-modeling software supports exporting to file formats that support multiple sub-models in a single file, the imported *game object* can be directly plugged into the tile-matcher *component* in Unity's *inspector*. Otherwise, the user has to import the models separately and group them into a parent object that is then supplied to the tile-matcher.

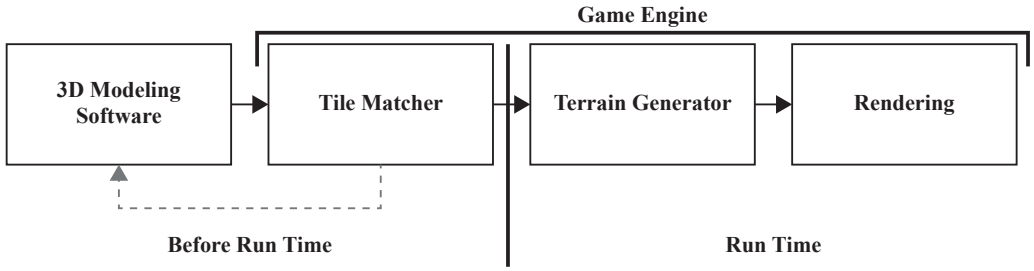


Figure 5.2: The User Workflow.

Afterwards, the tile-matcher can be run on the previously imported tile-set. The tile-set is then displayed to the user showing the connection constraints as colored spheres on the sides of the tile. The directional constraints (see Section 3.2.3) are displayed as spheres with a smaller cube next to it showing the clock direction. With this information, the user can easily verify visually if all the constraints are correctly generated as seen in Figure 5.3. In the case of a mismatch, the user can adjust parameters of the tile-matcher like the quantization amount or the face distance ϵ (see Section 3.2). If the matching error persists, the user possibly created unaligned geometry. In such a case, the tiles geometry has to be corrected in the 3D-modeling software.

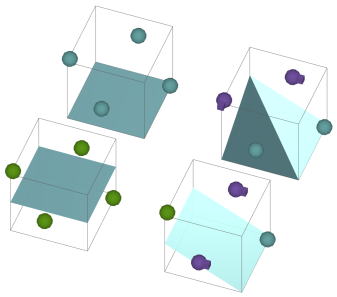


Figure 5.3: Display of a simple tile-set consisting of four tiles. Constraints are shown as spheres on the sides.

If the matching is successful, the terrain generation can be configured. The configuration includes the selection of the tile probabilities (see Section 4.3). To give the user a straight-forward interface to enter these probabilities they are arranged in a table. The cells of the table contain the different weights/probabilities of the tiles. The tiles are represented as the rows of the table. The tile names for the labeling of the rows are extracted from the tiles mesh representation as sub-model. If the sub-models do not contain names, generic names with ascending numbers in the same order as displayed in the before mentioned tile-matching overview are given. The columns in the table are the different *biomes*. These biomes are groupings of a specific set of probabilities for every tile in the tile-set. The number of biomes can be configured. The user has to define a probability for every tile-to-biome combination hence a table was chosen as the representation. Later in the terrain-generation, a random biome is chosen for each chunk.

The other configuration options for the terrain-generator are the chunk size and the number of generation-retries in case of a failed generation. As every chunk has a square shape, the number of tiles for one side can be entered. The retries are necessary to avoid an endless loop in cases where the WFC algorithm leads to an invalid result. As explained in Section 2.2 the WFC algorithm works probabilistic and encounters these states sometimes.

Now everything is ready for the terrain-generation step. This step can happen "offline" or "online"/on runtime. The first case is used in the workflow of the designer for experimenting with generation parameters and observing the overall look of the terrain-generation inside a sample region. The runtime terrain-generation is used in real-world applications. In video games, for example, the generation would happen continuously as the player moves across the terrain. There are two parameters needed to generate the terrain in a deterministic fashion. One is the position of the chunks that should be generated and the other is the seed - the input to reliably get the same outputs from the pseudo-random generator. For the prototype implementation, the position and size of a rectangular area have to be defined to run the terrain-generator in. Also, the seed can be changed to generate different terrains segments of the same size without changing the tile-probabilities.

In a production environment, the tile-matcher could be decoupled entirely into an offline-invoked application that is not contained in the client application. In such a case the matching, as well as the configurations of the terrain-generator, would be loaded from a file and directly fed into the terrain-generator. This would then generate chunks of terrain depending on the requested location.

Also part of the user workflow is the correct adjustment of the chunk size to ensure an optimal terrain-generation and rendering performance. This has to be done because the best chunk size depends on the combination of all the constraints of a tile-set. The constraints vary per tile-set and therefore also the best chunk size choice.

5.2 Performance

The following performance benchmarks were created on an Acer Aspire V3-772G-747a151.12TMakk notebook with an Intel Core i7-4702MQ CPU, 16 GB RAM and a dedicated NVIDIA GeForce GTX 760M GPU running Windows 10 (64-Bit). For every test instance, 100 iterations with ascending seed values were run to create a sound estimation of the underlying average performance.

As test tile-set, a large set with 33 tiles was chosen. With the four possible rotations of every tile, this results in a propagator (see Section 3.3) with 132 different tile indices. In total it contains 10 different constraint values (see Figure 5.4).

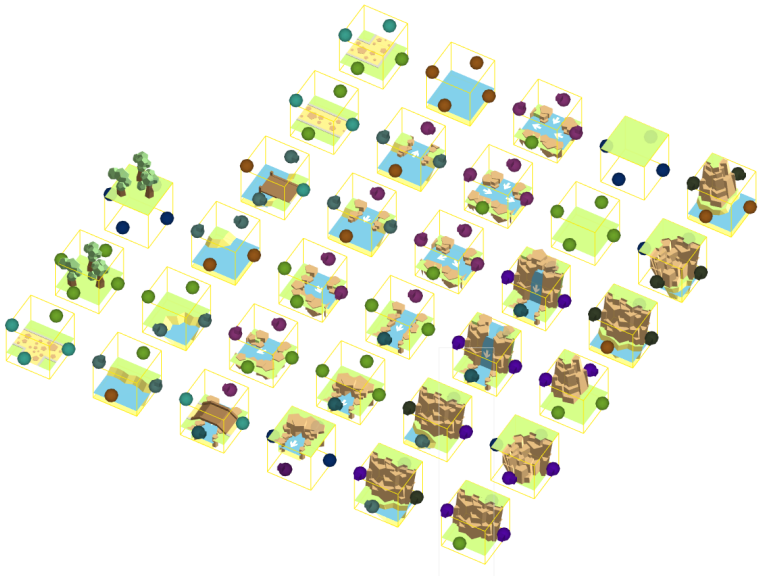


Figure 5.4: A large tile-set with 10 unique constraints.

5.2.1 Run-Time Speed Requirement

The performance of the terrain generation is critical, as the target is to employ the algorithm in computer games on run-time. The performance goal is to generate an area of chunks around the player called frontier f in at least the same speed v as the player moves across the terrain. This ensures that the player can walk infinitely in any direction and never "catch up" with the generator. The calculation depends on the size of the terrain as well as the size of a tile in relation to the player's speed. Solving it is always possible by changing how fast the player is in relation to the tile size. To give an example of how to calculate this speed a back-of-the-envelope calculation for one of the test instance's generation speed is given:

The test instance of 9 chunks with 225 tiles per chunk ran with 0.79 seconds as median time. This means that $9 * 225 = 2025$ tiles were generated in that time leading to a

generation speed of around 2563 tiles/second or one tile in 0.39 milliseconds. In the simplest case, a 3×3 chunk area around the player is sustained continuously. This results in a frontier of up to 5 chunks if the player moves diagonally. In such a case the generation time g_f of the frontier is $225 * 5 * 0.00039 \approx 0.44$ seconds. He traveled diagonally, therefore the distance until the new frontier has to finish calculation is ≈ 21.2 tile lengths. Therefore the player's speed has to remain under 48 tiles per second. This means that if the player travels with a speed of, for example, 6 m/s the tiles are allowed to be as small as 12.5×12.5 cm in virtual units.

This shows that even with a five-year-old notebook, the generation speed is fast enough to be used in a commercial application. In cases where a tile is, for example, set to 4×4 m, the frontier, as well as the player's maximum speed, can be increased by multiple orders of magnitude.

5.2.2 Chunk Size

The chunk size - the number of tiles a chunk contains - is a crucial factor for the performance of the terrain-generator. To evaluate in what way the chunk size affects the performance multiple benchmark instances were run containing the same number of tiles in each case but grouped in differently sized chunks.

Figure 5.5 shows a comparison of four different instances, every one of them generating 2025 tiles. The diagram shows that the median generation time, as well as the variance, decreases as the chunks size increases. This leads to the observation that for this specific tile-set the regeneration of the larger chunks in case of an internal chunk generation error outweighs the complexity of performing the regeneration of neighboring chunks if chunk generation backtracking is triggered (see Section 4.2.3).

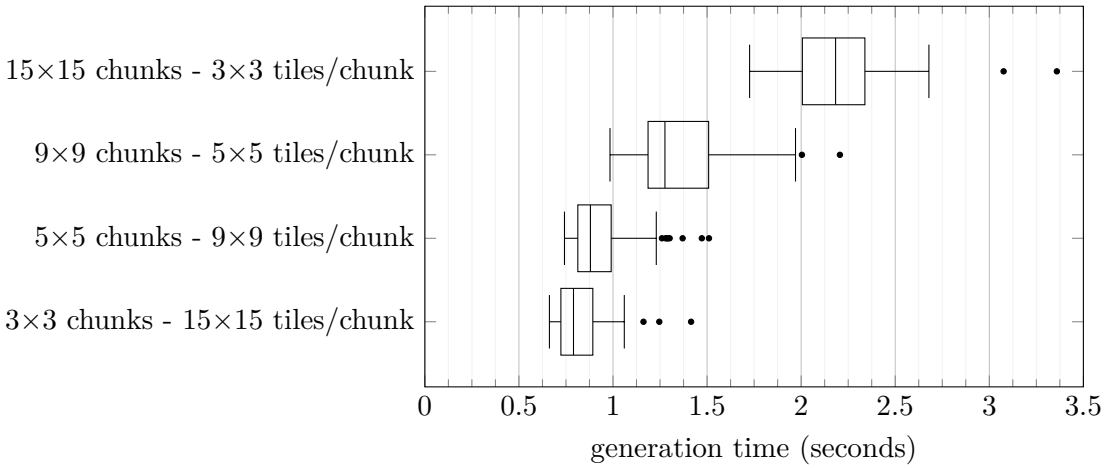


Figure 5.5: Generation time of 2025 tiles in all three cases while varying the chunk size to chunk amount ratio.

5.2.3 Scaling

Performance benchmarks were run to show in which way scaling of the generated terrain affects the generation time. The hypothesis was that scaling the terrain results in a linear increase in time needed to generate it. The benchmarking was done using chunks with a size of 9×9 tiles with an increasing amount of chunks. The terrain size was enlarged for every instance by increasing the number of chunks in each of the two axes by two chunks. This was done to remain at an odd number of chunks so no unnecessary chunks would be generated due to the chunk generation layout (see Section 4.2.2).

Figure 5.6 shows the statistical results of this benchmark. In the diagram, the blue line, depicting the median generation time of an instance (y-axis), and the red line showing the number of generated tiles per instance correlate. Both x-axis-attributes, the number of tiles and the generation time were scaled so that they start with the origin at zero and that the last data-point matches. An even better view of the correlation is given in Figure 5.7. Here the two values are plotted against each other. The linear correlation is clearly shown. As can be seen, the problem scales in a perfectly linear fashion concerning the output tile size. Therefore the assumed hypothesis was proven. Additionally, this also implies that the assumption of the chunk regeneration backtracking only having a local implication was true (see Section 4.2.3). Otherwise, the generation time would have increased faster than the number of tiles.

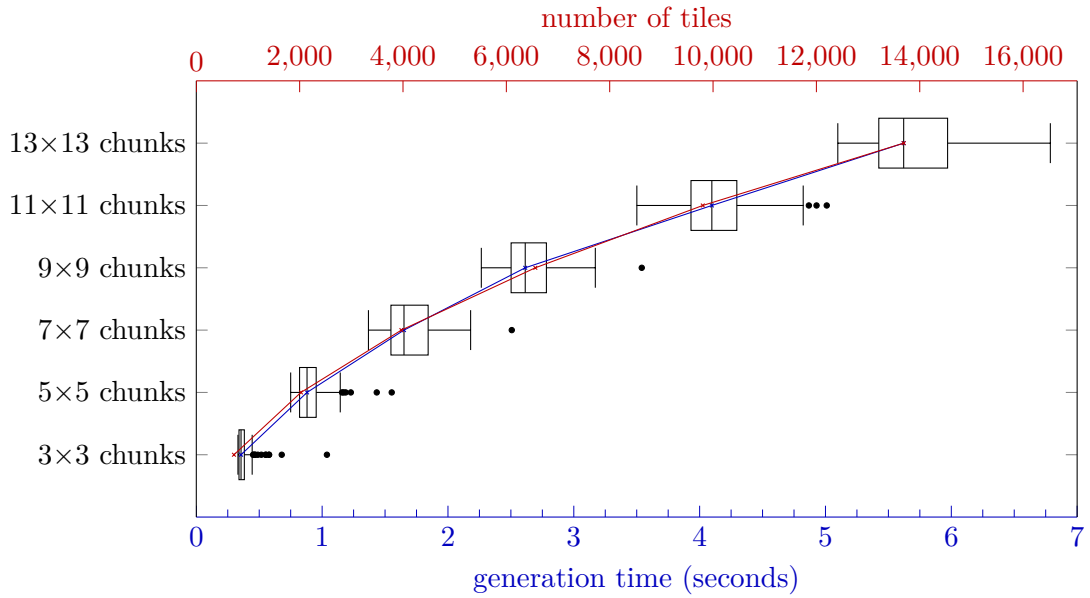


Figure 5.6: Generation times of increasingly large terrain regions using chunks consisting of 9×9 tiles.

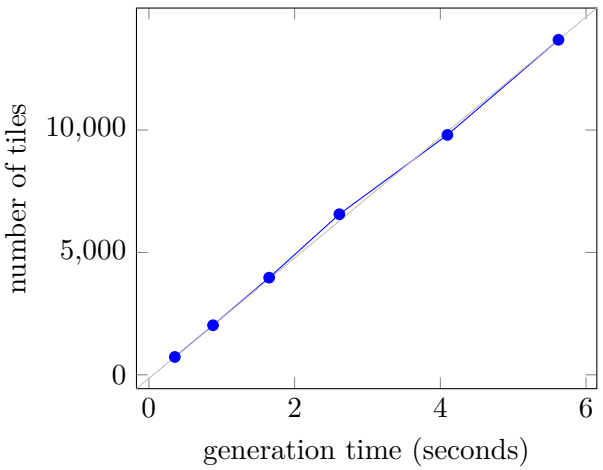


Figure 5.7: Generation time versus amount of tiles, same data as in Figure 5.6

5.2.4 Rendering

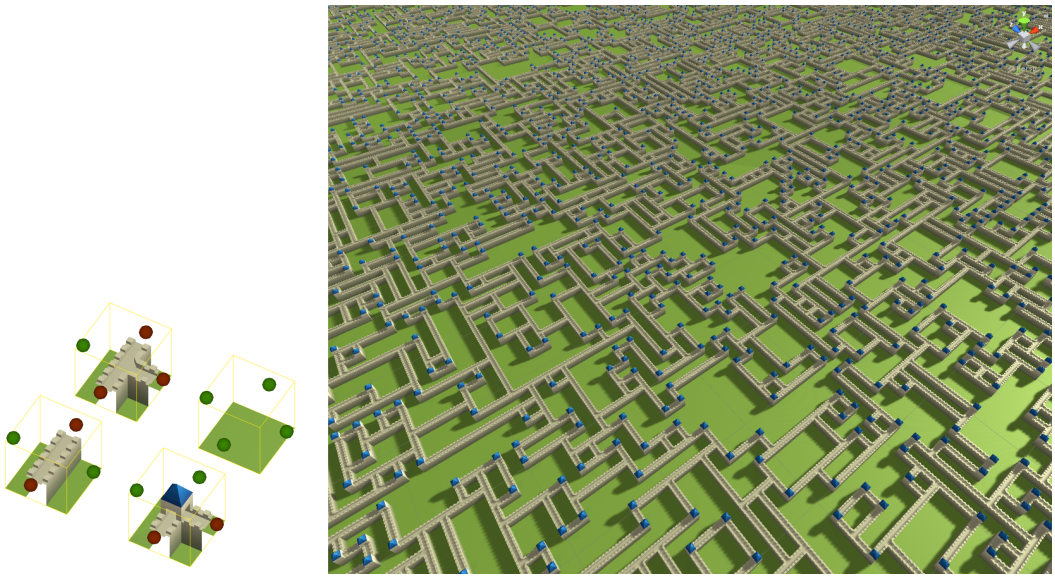
Rendering the terrain in real-time, although not the central aspect of this thesis, was also a target. Two different rendering modes have been implemented.

The first is GPU instanced rendering of the tiles. In this mode, the mesh of each tile in the tile-set only resides in the GPU's memory once. Then at each draw call, a list of positions for every tile is submitted to decide at which positions the tile should be drawn. With this mode it was possible to render 9 801 tiles, which amounts to 1.7 million triangles and 3.2 million vertices, using a top-down view, while staying at around 60 FPS and 190 MB memory usage.

The second mode works by grouping all meshes of the tiles contained in a chunk into one large mesh, effectively merging the geometry. Instancing cannot be used in this case, because every chunk mesh only exists once on the terrain. This also means that this approach uses more video memory on the GPU as the needed memory scales linearly with the size of the rendered terrain. Still, the performance was increased by a large margin. The rendering of 136 161 tiles, consisting of 22.9 million triangles and 41.6 million vertices in total, was possible with almost 200 FPS and 2350 MB memory usage.

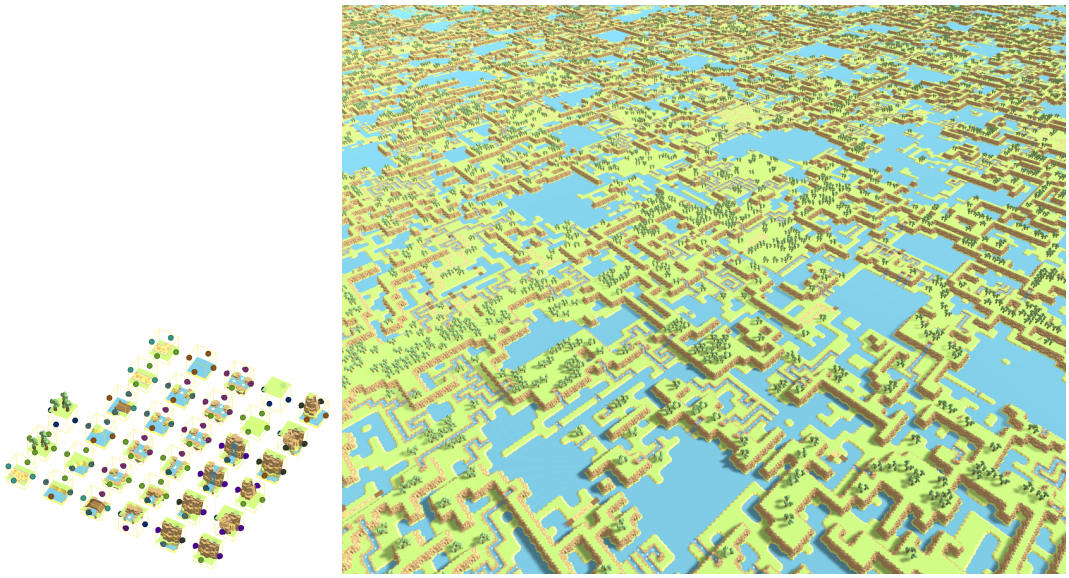
5.3 Test Tile-Sets

A multitude of tile-sets was created to evaluate the soundness of the tile-matcher. An example set with tiles that only contain one edge per tile-face is shown in Figure 3.2. More complex tile-sets, as well as terrains generated with them, are displayed in Figures 5.8 and 5.9. Figure 5.8 consists of a low number of tiles and features symmetric constraints only. Figure 5.9 on the other hand, shows the tile-set with the highest complexity also used for the performance benchmarks.



(a) (b)

Figure 5.8: A simple tile-set consisting of four tiles with symmetric constraints (a) is used to generate output terrain (b).



(a) (b)

Figure 5.9: A complex tile-set with 33 tiles and 10 different constraints (a) is used to generate output terrain (b).

CHAPTER 6

Conclusion

A concrete application of the WFC algorithm for the use in interactive media was shown. For this, the process of generating matching information with respect to the geometry of tiles in a tile-set was automated. This automation was integrated with a visual overview for the user to enable rapid verification of the generated constraints. With the matching information, the consistent mapping of mesh information to a numerical presentation for neighborhood constraints was achieved. The need to distinguish between different types of tile-sides (symmetric and asymmetric) was detected. Correct discovery and subsequent handling of these cases were implemented in the tile-matcher.

A terrain generator that works by supplying the generated matching information to an adapted WFC algorithm was developed. This terrain generator features the generation of infinite terrain in a deterministic fashion using a seeded pseudo-random generator. The generator's use in a run-time environment was evaluated and a simple system for the continues generation of terrain patches during the locomotion of the point of view was proposed. For this to work, the terrain was subdivided into larger units called chunks. Passing the correct neighborhood constraints on the chunk borders was solved through constraint injection of chunks from already generated ones. A hierarchical generation layout was proposed to decide which chunks depend on each other. It was shown, that the terrain generator runs in linear time concerning the requested amount of output tiles. This assertion holds, even though the complex problem of unsolvable constrain chunk borders and the resolution thereof requires a sophisticated strategy to avoid dead-ends in the generation. This strategy was developed and proven to only have an impact on the surrounding neighborhood of the affected chunk.

In conclusion, a complete solution from the matching of tiles from a tile-set to the generation of terrain with them was developed in such a way, that it can be directly integrated by developers and accordingly configured.

6.1 Future Work

The research of the topic of this thesis led to some interesting advancements and alternative approaches that are described below.

6.1.1 Proto-Terrain for Chunk Matching

An alternative strategy to solve the matching issue at the chunk borders was conceived. Instead of relying on a hierarchical layout and injecting the constraints to the generations of chunks in lower levels the passing could be eliminated. Pseudo-random noise functions with the desired dimensionality could be used to get a proto-terrain where the tiles are chosen much like in the marching cubes approach [LC87]. Afterwards, this simplistic proto-terrain that already has a consistent matching would be refined chunk-wise like in the original model synthesis [Mer07] approach with the chunk being the replacement-kernel. This would need more user supplied information as to how specific tiles from a tile-set would be used to represent the sampled noise function which is the reason this approach was not developed further. However, it is expected that this would lead to further decreases in the generation time.

6.1.2 Probability Functions

Instead of constant values for the probabilities of a tile per biome, parameterized noise functions like Perlin noise [Per85] could be used to define probabilities changing per tile. This would increase the visual fidelity even further and would hide the edges of biomes better. An extensive rewrite of the WFC algorithm would be needed for this change because the sum of the probabilities are calculated only per instance and would have to be changed to be calculated for every tile position. Noise functions could also be used to select the biomes instead of the current implementation using a random selection. This change however, is minor and could be implemented directly.

List of Figures

Bibliography

[Aul99] Holly K Ault. Using geometric constraints to capture design intent. *Journal for Geometry and Graphics*, 3(1):39–45, 1999.

[Ayc16] John Aycock. *Retrogame Archeology: Exploring Old Computer Games*. Springer International Publishing, 2016.

[Har00] Paul Harrison. A non-hierarchical procedure for re-synthesis of complex textures. In *In WSCG'2001 Conference proceedings*, pages 190—-197. Citeseer, 2000.

[KR17] Brian Kernighan and Dennis M Ritchie. *The C programming language*. Prentice hall, 2017.

[KS17] Isaac Karth and Adam M. Smith. Wavefunctioncollapse is constraint solving in the wild. In *Proceedings of the 12th International Conference on the Foundations of Digital Games*, FDG '17, pages 68:1–68:10, New York, NY, USA, 2017. ACM.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA, 1987. ACM.

[Mer07] Paul Merrell. Example-based model synthesis. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 105–112, New York, NY, USA, 2007. ACM.

[MM08] Paul Merrell and Dinesh Manocha. Continuous model synthesis. *ACM Trans. Graph.*, 27(5):158:1–158:7, December 2008.

[MM09] Paul Merrell and Dinesh Manocha. Constraint-based model synthesis. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, pages 101–111, New York, NY, USA, 2009. ACM.

[MM11] Paul Merrell and Dinesh Manocha. Model synthesis: A general procedural modeling algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 17(6):715–728, June 2011.

[Per85] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[Pru86] P Prusinkiewicz. Graphical applications of l-systems. In *Proceedings on Graphics Interface '86/Vision Interface '86*, pages 247–253, Toronto, Ont., Canada, Canada, 1986. Canadian Information Processing Society.

[STN16] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games*. Springer Publishing Company, Incorporated, 1st edition, 2016.