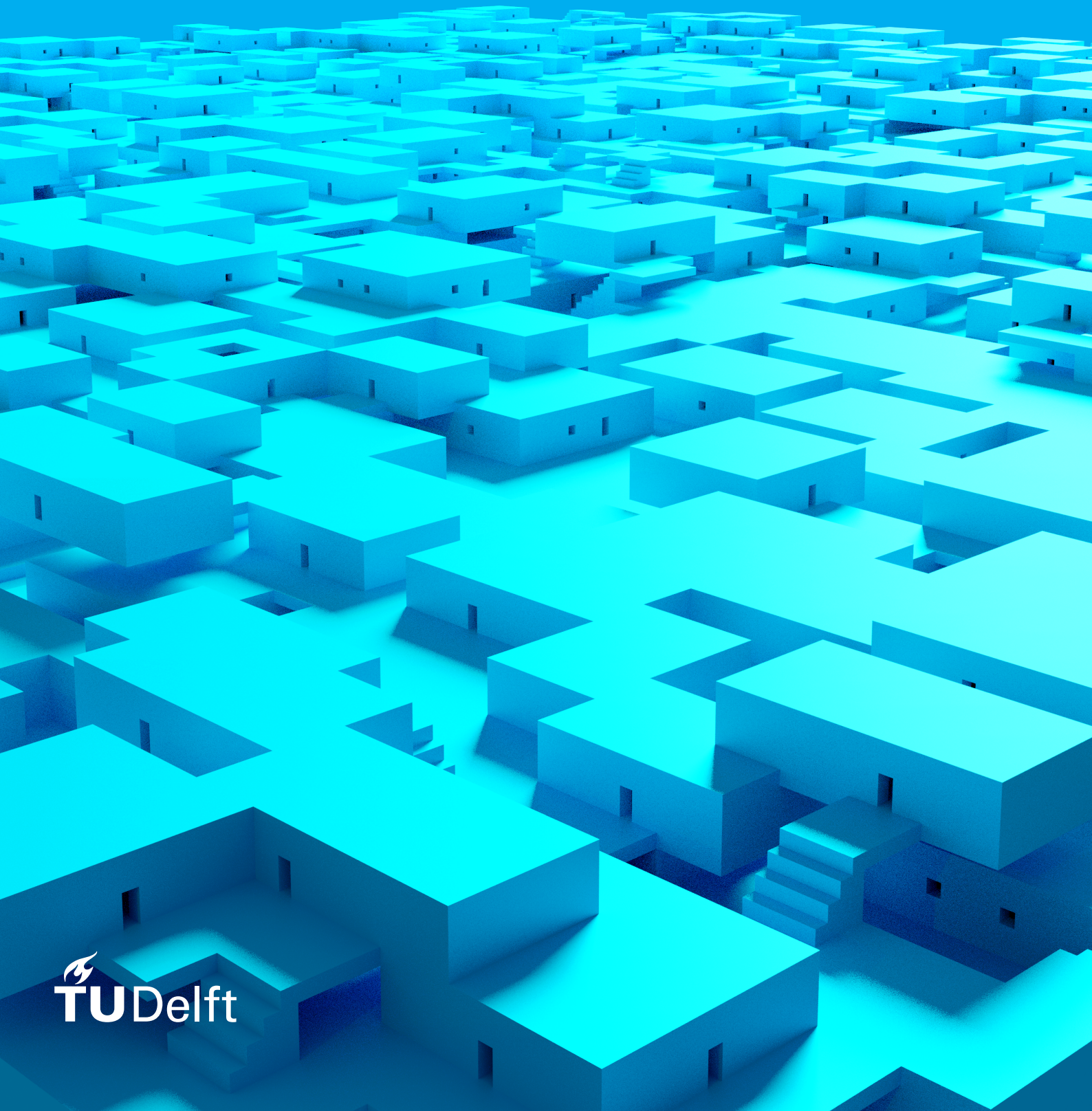


Architectural Profiles

Procedural Content Generation using Tile-based Architectural Profiles
Levi van Aanholt



Architectural Profiles

Procedural Content Generation using Tile-based Architectural Profiles

by

Levi van Aanholt

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday August 28, 2020 at 12:00.

Under guidance of:



Computer Graphics and Visualisation
Department of Computer Science
Faculty of EEMCS
Delft University of Technology

Student number:	4084101
Project duration:	November 1, 2018 – August, 2020
Thesis committee:	Dr. ir. R. Bidarra, TU Delft, supervisor
	Dr. ir. K. Hildebrandt, TU Delft
	Dr. ir. E. Visser, TU Delft

An electronic version of this thesis is available at <http://repository.tudelft.nl/>.

Architectural Profiles

Procedural Content Generation using Tile-based Architectural Profiles

Abstract

Procedural content generation (PCG) for architecture is widely used in a variety of digital media, most notably in games. However, such methods are often limited in their expressive range, and require considerable technical knowledge to create non-trivial architectural structures. We present a novel tile-based PCG approach for generating architecture, that proposes the use of *architectural profiles*, a declarative characterization of architectural typology, within a generic tile solving framework. An architectural profile consists of a set of tiles, representing atomic architectural building blocks, and a set of declarative constraints and rules, specifying which conditions a tile configuration has to satisfy to be valid. These conditions are translated into logic constraints, and used by a tile solver to control tile placement in a bottom-up manner. Eventually, each valid model output by the solver is a representative instance of its architectural profile. We describe an implementation of this approach with Answer Set Programming, using an off-the-shelf constraint solver for model generation. We performed an expressive range analysis, and concluded that our declarative method is quite controllable and can be steered over a broad range of architectural structures, regarding density and repetitiveness. Due to this expressive range and control, our tile-based method is very suitable for the customized development of urban environments for games. We also explore the adaptability of architectural profiles by application on pre-existing terrains.

Preface

Before studying Computer Science(CS) I acquired a bachelor in Architecture (2011-2014) at the TU Delft. I loved all the conceptual knowledge and methodologies that were taught there. But discovered my indifference to become a practitioner, because working as an architect seemed a little vague to me.

There were a few courses in the second year using visual programming in a geometric editor(a software plugin called Grasshopper used in a 3D modelling system called Rhinoceros) and I had affinity with these. I pursued this interest further by doing a software minor at CS. Surprisingly to me, I found connections with my interests regarding architecture. It showed me that the architecture I am most interested in is an overlap between methodologies and concepts from architecture and Computer Science. I feel this is an interesting intersection, as both of these fields have such different approaches. Architecture has a long persistent tradition that is ubiquitous when designing. Computer Science, on the other hand, focusses on innovation, ideas and paradigms follow each other in quick succession.

After I finished my bachelor degree in Architecture, I started a new study at CS. I have chosen to pursue my interest from the direction of CS with insights into the architectural design process from the architecture degree. Computer Science was a new adventure that included adjusting to a more logical and technical discourse. There were also amazing sidesteps such as an unforgettable Game jam organized by the professor of Computer Graphics dr. Elmar Eisemann and an exciting study trip to Japan and South Korea organized by the EWI study association CH.

In this thesis I have made my first step in doing research in the field of Procedural content generation. This research took over a year of intensive work. It wouldn't have been possible without my thesis supervisor dr. ir. Rafael Bidarra, who had the patience to help steer this project and gave me the freedom to research this topic at my pace. While also giving valuable insight into the research process, the inner workings of academics and suggested many improvements to the approach. Also I won't ever look the same at dice again. I want to thank dr. Adam Smith as well, an expert in ASP programming that gave valuable advise regarding logic programming.

I also want to thank Hao Ming for being a friend and colleague throughout the Computer Science study and the thesis process. Elvira, a good friend and a talented fine artist. My parents, Lionel and Sylvia, who supported me. My kind-hearted grandparents, Sjoerd and Marie, who both, wistfully, passed away during this research. And lastly, mechanical keyboards, whose clicks and clacks filled many nights of thesis work.

*Levi van Aanholt
Delft,*

Contents

1	Introduction	7
1.1	Architectural representation	7
1.1.1	Architectural Typology	7
1.1.2	Design methodology	8
1.2	Representing architecture for procedural generation	9
1.2.1	Top-down/ bottom-up techniques	9
1.3	Thesis Structure	10
2	Declarative procedural generation of architecture with semantic architectural profiles	13
2.1	Related Work	13
2.1.1	Tile Solving	13
2.1.2	Architectural Semantics in PCG	14
2.2	Approach	14
2.2.1	General Tile Solving Framework	14
2.2.2	Architectural Profile	15
2.3	Evocative Examples	17
2.3.1	Examples	18
3	Evaluating Comprehensiveness with an Expressive Range Analysis	21
3.1	Metrics	21
3.1.1	Density	21
3.1.2	Repetitiveness	21
3.2	Model generation	21
3.3	Exploring the expressive range	23
3.3.1	Lower left	23
3.3.2	Upper left	23
3.3.3	Lower right	23
3.3.4	Upper right	23
3.4	Implementation	23
4	Adaptable Architectural Profiles with an Iterative Solver	25
4.1	Related Work	25
4.1.1	Settlement generation and reconstruction	25
4.1.2	Generative Design in Minecraft Competition	26
4.2	Our pipeline	26
4.3	Terrain analyser	26
4.3.1	Discretization of the terrain	28
4.3.2	Categorization of the terrain	28
4.3.3	Assigning profiles to the terrain	29
4.4	Iterative AP Shape Solver	29
4.4.1	Preliminaries	29
4.4.2	Instigator loop	31
4.4.3	Iterative Shape Solver	31
4.4.4	Create Shape	32
4.4.5	Place Tile	33
4.5	Removing Unnecessary Shapes	33
4.6	(+) Combining settlements and the terrain	36
4.6.1	Terrain adjustments	36
4.6.2	Building Materials	36

4.7	Implementation	37
4.8	Results	37
4.8.1	Flat terrain	37
4.9	Results on GDMC terrains 2020	38
4.9.1	Map 1	39
4.9.2	Map 2	40
4.9.3	Map 3	40
5	Conclusion	43
5.1	Future Work.	43
	Bibliography	45

1

Introduction

Procedural content generation (PCG) automates, or aids, in the creation of content through computational means. It is used in a variety of digital media, ranging from music, movies and games. PCG helps make production more cost-effective, limiting dependence on hand-made content. It also allows for novel content that would be infeasible to create by hand. For example in the area of video games. It is regularly employed as the main selling point for video games, for example when the game spaces are generated procedurally. Space exploration games, such as *Elite Dangerous*, feature billions of planets. Survival games, such as *Minecraft*[24], feature a different sandbox world every time you play.

The different spatial domains (such as natural terrains and architectural structures), of a game world require different approaches[30]. Natural terrain can be approximated with noise functions, yielding convincing results. Architecture, however, requires a different strategy. The main reason is that architecture needs to evoke that it is designed by humans or at least looks sensible from a human perspective.

PCG approaches that create architecture are numerous and varied. *Comprehensive* techniques, such as grammar approaches[21], offer fine grained control, but are work intensive to create. *Declarative* techniques on the other hand make the technique easily tunable. They may offer designer interfaces such as sketching[22] or inputting images[23]. These are easy to use but not comprehensive, limited by the training set or the user interface. Finding a new PCG technique for architecture that is both declarative and comprehensive is beneficial to designers of digital media such as games. Therefore we will investigate this. In section 1.1 we will draw from architectural academics to sketch a theoretical foundation in which our PCG technique can be placed. Henceforth in section 1.2 we investigate a practical computational representation that forms the starting point of our PCG technique.

1.1. Architectural representation

We will identify the following two terms that stem from the domain of architectural academics. *Architectural typology*, how to differentiate architectural designs. And *design methodology*, that explains the emergence of a design.

1.1.1. Architectural Typology

An architectural design is perceived by *architectural analysis*: analysing its physical and functional attributes. These consist of many metrics which include footprint size, identifying routes, functionality of the rooms and uncovering a design grid. These metrics may indicate the adherence to architectural rules, that require adherence for the design to make sense. *Architectural typology* is the classification of architecture based on these metrics. It is a useful term to describe our goals for a comprehensive technique. An architectural design often fulfils many architectural rules. Some of which are quite obvious; that all rooms need to be reachable and that a building is enclosed. More specific are contextual parameters that greatly influence the design. Cultural parameters for example, in Victorian England, country houses belonging to the rich were designed to create separate routes for the (wealthy) owners and the (poor) servants. Environmentally (sensible) houses in areas with hot climates don't have windows in the south, in direct sunlight, to avoid the heat. Technological advancements can have a great influence on architectural design. Reinforced concrete allow for open floor



Figure 1.1: International examples of architectural structures that show a wide variety in both density and organization of building placement.

Upper left: Skyscrapers positioned in an organized grid structure along straight roads.

Lower left: A low density residential area, consisting of houses placed along organized roads.

Lower right: The coastline of Oia, a city in Greece. Buildings are densely packed together connected with adventurous walkways.

Upper right: The, now demolished, Kowloon Walled City in Hong Kong features high density and chaotic placement of buildings, where buildings are even built on top of others. An interconnected network of pathways connect them.

plans. Elevators allow for high-rise. Contemporary buildings take novel shapes, benefiting from advancements in computational means to generate support structures[25].

For our approach we want to investigate a representation of architectural typology that spans a few extensive, quantifiable metrics. In figure 1.1¹ we show wide ranging architectural structures that form an interesting space in architectural typology, regarding density and the repetitiveness of the building placements. A technique that can represent this space, we argue, is comprehensive.

1.1.2. Design methodology

Design methodologies explain how designs emerge. It is used within a design process to act as a steering mechanism to make consistent design decisions. Interestingly, the design methodology used by architects is often implicit[39]. Designers learn by doing and act from experience. A designer may reflect on design decisions as intuition, instead of being able to reconstruct a procedural explanation. This is common with contemporary architecture, where architects both fulfil complex design parameters and also create a building that are recognizably theirs. On the other hand, a design methodology may be explicit. Then the process is composed of clear instructions. The famous "A pattern language" by Christopher Alexander, suggests architectural design patterns to inform social and humane designs. Even more explicit design practices were common in Classical architecture, which follows strict rules. These are often based on lofty ideals, sometimes even exhaustively documented (such as in the four books of architecture by Palladio). Recent research even analysed classical rules by translating them to a grammar[6]. Given a method is explicit enough (either through exact procedure, or approximative with a complete training set[32]), it becomes computable and translates into a PCG technique. This allows the computer to perform the act of design instead of a human.

¹Images originate from pixabay.com

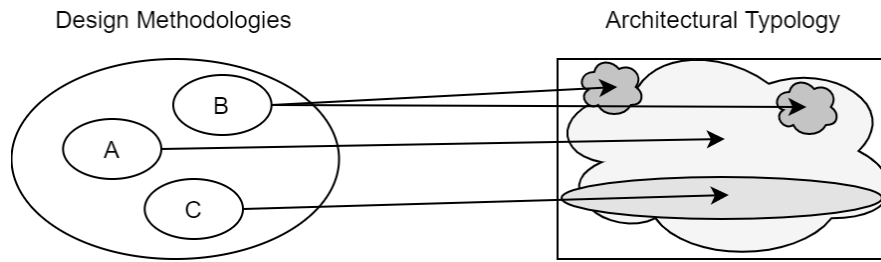


Figure 1.2: We are interested in finding an architectural PCG technique that is declarative and comprehensive like method A, able to create designs over the whole space. Method B is able to create a large variety of designs but not the designs inbetween. And Method C can only create designs within a limited range.

Conceptually, we identify the usage of an architectural PCG technique as a substitute for the manual design process. Figure 1.2 shows what we want the technique to achieve. Our technique should be composed of a procedural design methodology that is declaratively configured by the user and is tunable to a wide range of designs, meaning it is comprehensive.

1.2. Representing architecture for procedural generation

We want to find a PCG technique that generates architecture. First we need a computational structure to represent architecture. Many of these can be found in previous work. *Split grammars*[43] can model building exteriors. The faces of the facade are created by splitting a starting surface according to the grammar and additional split rules are applied to add more detail to the facade. *Shape grammars*, notably CGA [21], improve expressivity of split grammars. The technique starts with mass models. These can then be placed as specified by the building envelope. From this point onward the mass model is given a detailed facade by using grammar operations on its faces. This grammar expresses rules to add more building details inward by using the grammars production rules, that reference the placement of geometry or geometric operations. Recently there has been promising evidence for using *learned techniques*, using generative deep learning on building plans[3]. A major caveat is that a large corpus needs to exist, which limits the applicability.

Another approach uses *tiles*. PCG techniques that use tiles build on a representation of computational problems called Wang Tiles[40]. Model synthesis cuts a 3D model up in tiles and creates new models by placing these tiles while preserving local neighbourhoods[20] that itself builds upon work in synthesis techniques for textures[4]. Recently a technique called WaveFunctionCollapse[10] has been popular in the PCG community, inspiring multiple game concepts and academic work[12]. Particularly noteworthy is that small input to the generator yields a wide variety of output. The input are a number of tile images(same sized textures), of which the border pixels of some tiles coincide with each other. Then and only then can they be encountered adjacently in the solution.

1.2.1. Top-down/ bottom-up techniques

We qualify the aforementioned techniques in one of two categories, top-down and bottom-up.

Top-down in this qualification means that given more information the amount of conceivable generated models increase. Grammar techniques fall in this category. When more rules are added to the grammar the models increase in variety and detail. A caveat of using grammars is that creating them by hand is known to be work intensive and requires expert knowledge, which is not declarative. Techniques exist that make grammars easier to use and adapt to different situations[29]. Furthermore, inverse modelling accepts a pre-existing grammar and a target model and returns a model similar to the target produced by the grammar. Probabilistic inference can be used to optimize a grammar production to look like the target[35]. Even more accessible for designers is when they only need to input an exemplar model, and a grammar is derived from it automatically, for example, a grammar can be formed from model pieces by cutting it at symmetric areas[2].

Bottom-up inverts the generation process. Now information needs to be added to constrain the output to generate only models that fit the requirements. When unconstrained, the amount of conceivable models increase, varying widely in suitability.

Techniques featuring tiles use the bottom-up approach. As an exploration we implemented the aforementioned WaveFunctionCollapse and created a small tile set shown in figure 1.3. Although the tiles repre-

sent architectural elements, the output does not make sense from an architectural point of view. Because the input only consists of tiles, there is no room to specify these rules declaratively. There have been attempts outside of academia to use WFC directly for results that look architecturally sensible. The approach is then to grow a large tile set over time, by hand, making sure that the relationship between the tiles as defined by the borders remains sensible, which is work intensive. Declarative approaches for generating architecture with tiles do not yet exist. Therefore we will investigate this approach further.

The bottom-up approach fits better with the goals for our comprehensive and declarative procedural technique. It better resembles our conceptual goal to create a procedural substitution for the manual design process. Given a design task, a designer creates based on a design method, which can result in a wide range of designs. The design task requires adherence to contextual parameters that constrain and steer the designer. Similarly in our technique, the starting point is that the generator is comprehensive, unconstrained it contains all models. Given more specificity in the form of declarative constraints, only models with specific characteristics are generated. We will investigate a bottom-up PCG approach using tiles.

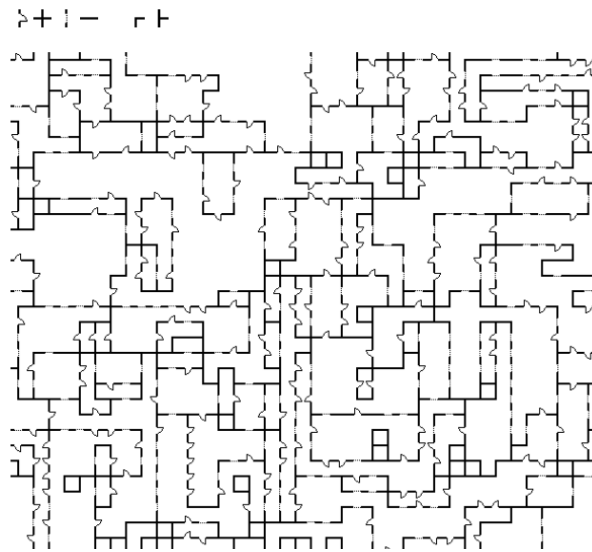


Figure 1.3: A two dimensional WFC result created by our own WFC implementation. At the top the tiles are shown, we created tiles that represent elements in an architectural floor plan. If a tile is not fully rotational symmetric, the rotations are also used as tiles. The solution shows a rich distribution of tiles but lacks semantics of architecture. Rooms are often recursive, rooms occur in rooms. Doors and windows are spread all over the place and have no clear route. WFC offers the possibility to tweak the probability of the occurrence of a certain tile. In this experiment these were kept the same for all tiles.

1.3. Thesis Structure

We will investigate a bottom-up, procedural technique for architecture that uses tiles. If this technique is both declarative and comprehensive, then it is an adequate representation of architectural typology, as shown in figure 1.4. The technique is declarative if it is easily controllable by the user. This will be discussed in chapter 2. The technique is comprehensive if it can be fine-tuned over a large variety of designs, in other words, it covers a large space within architectural typology. We will evaluate comprehensiveness with an Expressive Range analysis[31], discussed in chapter 3. Furthermore we will explore the adaptability to contextual parameters by evaluating the technique on a variety of pre-existing terrains, discussed in chapter 4. In summary, our research question is:

- What computational representation of architectural typology based on tiles is:
 - declarative (chapter 2),
 - comprehensive (chapter 3),
 - and adaptable (chapter 4)?

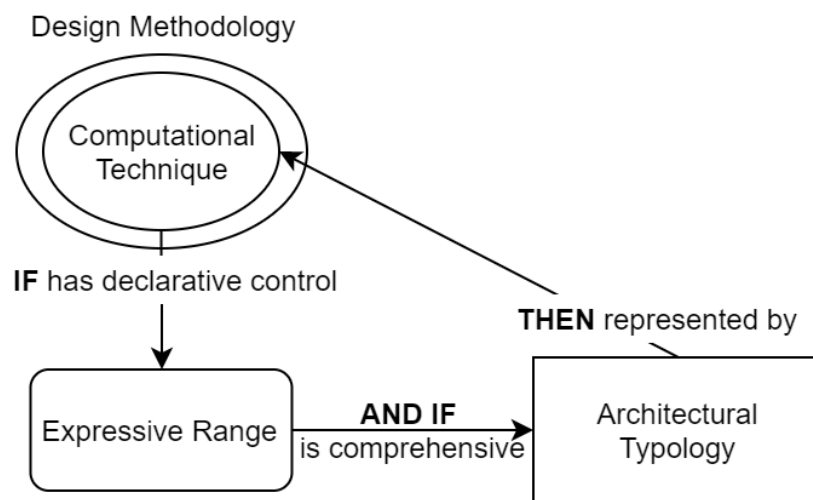


Figure 1.4: We search for a procedural technique that automates the human design process. If this technique has declarative control and is comprehensive then it is a valuable representation of architectural typology.

2

Declarative procedural generation of architecture with semantic architectural profiles

Procedural content generation (PCG) for architecture is increasingly used in a variety of digital media, allowing for the cost-effective creation of urban spaces for ads, movies and games, and reducing the volume of hand-made content needed [30].

Architectural typology categorises buildings based on their physical and functional properties, and this characterization can be applied to PCG for architecture. We say a PCG technique is *declarative* when the generator can be easily steered to generate models of the intended typology [37]; and it is *comprehensive* if it is capable of generating a variety of typologically distinct models [38]. Many current PCG approaches use grammars, which are highly configurable and fast, but are difficult to write and apply, thus lacking in declarative control [28]. Other approaches are data-driven, which makes them easier to use, but their expressive range is dependent on the existence of good training sets, potentially compromising comprehensiveness.

In this chapter we present a novel tile-based PCG approach¹ for generating architecture that is declarative. Our method does not rely on grammars or a training set, but on the definition and use of *architectural profiles*, a semantic characterisation of architectural typology suitable for use within a generic tile solving framework (Section 2.2). This declarative approach is easily configurable (Section 2.3).

2.1. Related Work

Previous research results fall into two categories: tile solving methods and procedural modelling for architecture.

2.1.1. Tile Solving

Tile solving deals with the challenge of filling a space with tiles given a tile set, under a variety of neighborhood constraints for each tile, e.g. allowing only a tile subset at its adjacent positions. Wang Tiles approaches [40] formalize this problem. A common data-driven PCG approach with tiles is tile synthesis, consisting of a tile retrieval step and a tile solving step. This approach achieved successes in texture synthesis [4, 16] and in 3D model synthesis [18]; the latter takes a 3D object as input, splits it into tiles using a grid, and proceeds to do tile solving with a greedy algorithm. Tiles can also be obtained using offset statistics [44]. Model synthesis was later generalized by using Markov Random Fields [45], which allow for statistical constraints over larger tile groups. WaveFunctionCollapse (WFC) [10] is another tile solving method, which omits a tile retrieval step. It has been popular in the PCG community and has also inspired valuable research work [13, 27]. In particular, WFC can be solved using Answer set Programming (ASP) [12], a declarative language for computationally complex problems. This approach is promising because by using ASP it is exact and extendable.

¹The contents of this chapter have been published at IEEE COG 2020 [1] and have been co-written with Rafa Bidarra. Special thanks go to Adam Smith who gave valuable advice regarding the ASP implementation.

2.1.2. Architectural Semantics in PCG

Architecture can be understood as the meaningful organization of space, accomplished by purposefully placing construction elements. PCG techniques model architecture by connecting architectural semantics with computational means. Shape grammars with split rules [21] model architecture as a hierarchic application of refinements to a base shape. This approach successfully models building envelopes and classic architecture [6] but does not easily capture the overall building semantics. To create believable buildings, multiple distinct generators can be orchestrated [37], or a building layout can be optimized using a trained network [19]. Grammars representing diverse architecture can be blended by adding labels [15], creating a large variety of new mixtures. Moreover, labeling shapes as public and private space [11] results in evocative spatial layouts within a hierarchical generation process.

2.2. Approach

We describe our novel tile-based approach for the generation of architectural structures, that proposes the notion of *architectural profile* and its use within a generic tile solving framework. An architectural profile is a declarative characterisation of a given architectural typology, and consists of a set of tiles (as atomic architectural building blocks) and a variety of declarative constraints and rules (as validity conditions for tile configuration).

Before describing in detail the components of the architectural profile, we first introduce the basic elements of a tile solving framework.

2.2.1. General Tile Solving Framework

The essential elements of a general framework for tile solving can be summarized around the notion of Tile Constraint Profile, as follows:

$$\text{TileConstraintProfile} = (T, A, R), \quad (2.1)$$

where T is the set of available tiles, A a set of adjacency conditions between tiles, and R a set of validity constraints. Figure 2.1 illustrates the global approach: given a Profile and the available input space, the tile solver incrementally ‘fills that space’ with tiles that comply to all conditions and constraints expressed, yielding a valid output model.

For this, we consider the input space subdivided in equally sized cells c , each cell susceptible to hold one and only one tile. The goal of the solving process is to assign one tile to each cell without violating the profile conditions and constraints, in A and R .

Tiles

T is a set of 3-dimensional tiles, each of which has a bounding box of the size of a (space) cell c . Some examples of tiles are given in Figure 2.2, fitting a cell size of $5 \times 5 \times 4$. For convenience (and unlike Wang tiles), we will consider that a tile may be rotated (by multiples of) 90° around the vertical axis (Z). This allows for a strong reduction in the size of the tile set T , required for generating moderately complex structures.

Adjacency conditions

A is the set of Adjacency Conditions, i.e. hard constraints denoting each pair of tiles (t_1, t_2) that may be assigned to adjacent cells, along some direction d ; this constraint is indicated as $t_1 \xrightarrow{d} t_2$. An adjacency condition is, by definition, always bidirectional, i.e. $t_1 \xrightarrow{d} t_2 \iff t_2 \xrightarrow{-d} t_1$. Moreover, an adjacency condition $t_1 \xrightarrow{d} t_2$ also holds for all 90° rotations (mentioned above) equally applied to the two tiles. In short, during the solving process, two tiles can only be placed adjacent to each other if they comply to some adjacency condition of A .

Validity Constraints

R is the set of Validity Constraints, i.e. profile rules that additionally constrain the solving process, by specifying explicit restrictions on tile placement. Constraints can be either hard constraints, which have to be satisfied by all tiles in the output Model, or soft constraints, each of which poses a penalty cost when violated.

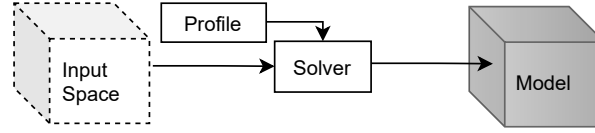


Figure 2.1: An input space and a profile are input to the solver yielding an output model.

An example of a hard validity constraint would be to require the output to follow a valid Sudoku pattern, i.e. for any row, column or sub grid in the Model, no two cells may contain the same tile. An example of a soft validity constraint is optimizing for a uniform distribution of tiles in the Model. The penalty cost can be described as the variance of the occurrences of the tiles in the Model.

Summarizing, the purpose of the whole solving process for a Tile Constraint Profile (T, A, R) consists of assigning a tile from T to each and every cell of the input Space, yielding an output Model such that (i) every tile satisfies the adjacency conditions in A , (ii) it satisfies all hard validity constraints in R , and (iii) it minimizes the cost penalty of all soft validity constraints in R .

2.2.2. Architectural Profile

An Architectural Profile is a semantic specification tailored to declaratively constrain a tile solver for architectural generation. The specification for an architectural profile applies the general framework of the previous subsection to an architectural context: tiles represent architectural elements, each with its own semantics, and are combined to create architectural structures called *shapes*, which in turn can be further combined. The essential solving process of architectural profiles will, therefore, focus on the placement of shapes in the input space, rather than on tile placement. An architectural profile is defined as:

$$\text{ArchitecturalProfile} = (T_a, A, S, A_S, R_S), \quad (2.2)$$

where T_a is the set of tiles with architectural semantics, representing architectural building elements and their function, and A is a set of adjacency conditions between tiles. S is the set of profile shapes, built with tiles of T_a , A_S is a set of adjacency conditions between shapes of S , and R_S is a set of architectural validity constraints on shapes of S .

Tiles

In an architectural profile each tile has some architectural meaning, or semantics e.g. walls, windows and doors. Moreover, semantics on tiles can be captured in the form of labels. For the examples in this paper, we will make use of two tile labels to indicate that a tile is meant for building *interior* or *exterior* (or possibly both). This tile architectural semantics is illustrated in Figure 2.2, where each tile corresponds to an architectural element, and has one or more labels. Enriched with this semantics, tiles can be grouped and filtered, defining *types* which will prove convenient for describing shapes.

Adjacency conditions

Semantics is also defined on the adjacencies between tiles, expressing the meaning of how tiles may relate to each other. This can define, for example, an allowed navigation direction from one tile to the other (*traversal* condition), or whether a tile supports construction above it (*construction* condition). These meaningful adjacencies can be defined between a tile and a whole type of tiles, yielding so-called *typed adjacency conditions*: $t \xrightarrow{d} \text{type} \in A_{\text{type}} \subseteq A$. Figure 2.3 exemplifies the two typed adjacency conditions mentioned above, used throughout the examples in this paper. As before, typed adjacency conditions define in which direction(s) they hold. Typed adjacency conditions are useful to specify which building elements can be connected within a shape.

Finally, we define a typed adjacency condition as an ‘entrance’, when it involves an entrance tile t (e.g. a ‘door’) and (tiles of) some *type* (e.g. ‘traversal’), along the entrance direction d ; it is denoted as $t \xrightarrow{d,e} \text{type}$. Entrances defined in this way are useful as the entry point for shapes to be connected together, thus allowing for the definition of shape connectivity.

Shapes

The central concept of an architectural profile is a *shape*, defined as a *connected* cluster of tiles, all sharing a common label (e.g. *interior*). The term ‘connected’ here has the common recursive definition (i.e. t_1 is

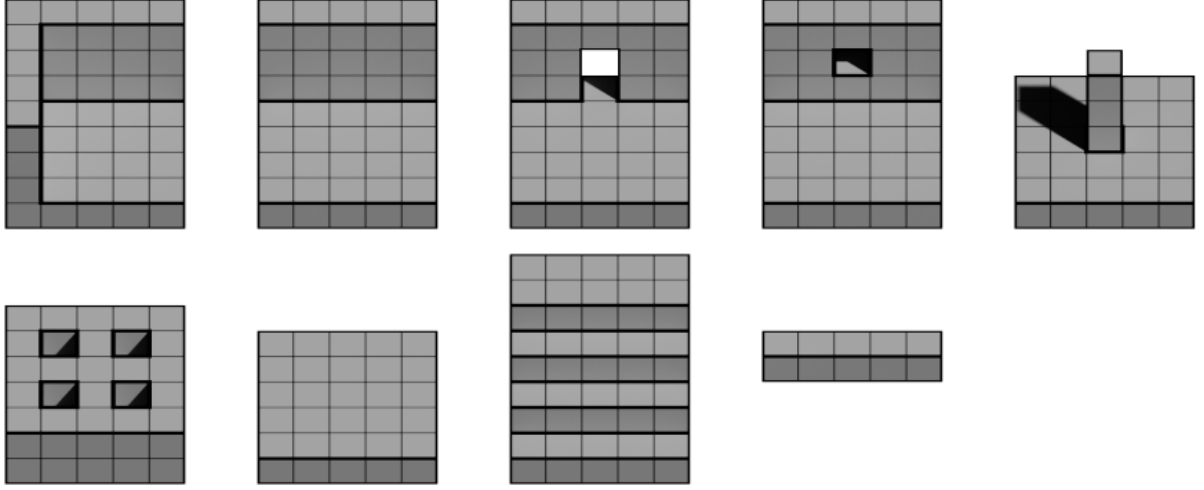


Figure 2.2: Tiles used in the profiles showcased in this paper. From left to right, first row (interior tiles): corner, wall, door, window, interior area; second row (exterior tiles): roof, street, stairs, landing, void tile. Stairs and landing tiles can be both interior and exterior.

connected to t_2 if either t_1 is adjacent to t_2 or t_1 is adjacent to some tile t_i that is connected to t_2), but with the condition that these adjacencies are of the same type (e.g. *traversal*), and are not entrances. All tiles that are connected in this way belong to the same shape. This adjacency type is, therefore, essential in the shape definition.

Because we are interested in the generation of architecture, shapes can be seen as the larger structures (e.g. streets or buildings), and the tiles as the basic elements composing that structure (see Figure 2.4).

As an architectural structure, a shape can be expected to assume a variety of sizes. Depending on the shape and on its intended architectural use, some way has to be provided to indicate the valid range(s) for a shape's size. For simplicity, we will assume here two simple bounding boxes, indicating the shape's minimum and maximum sizes, respectively.

Summarizing, in order to generate a shape, a cluster of connected tiles has to be assembled, such that (i) they all share the same specific label, (ii) their adjacencies are of the same type, and (iii) the tile cluster fits within the given size range.

Shape adjacency conditions

Similar to adjacency conditions defined between tiles, we define adjacency conditions between shapes to constrain their placement. Adjacency conditions between tiles described local placement relations between architectural elements such as walls, ceilings and stairs. Adjacency conditions between shapes describe placement relations between architectural structures. For example, a building and its connection(s) with the street or with other buildings, illustrated by figure 2.5.

We say that two shapes, $shape_a, shape_b \in Model$, are adjacent over a given adjacency *type* when they include two adjacent tiles, t_a, t_b , such that $t_a \xrightarrow{*} t_b \in A_{type}$, where the $*$ symbol stands for any direction. An adjacency condition between the two shapes is then indicated as $Shape_a \xrightarrow{*} Shape_b \in A_{S,type}$; and, because tile adjacencies are symmetric relations, so are shape adjacencies. We will, therefore, simply indicate a shape adjacency condition as $Shape_a \xleftrightarrow{*} Shape_b \in A_{S,type}$.

All adjacent shapes in the model have to satisfy the shape adjacency conditions.

Architectural validity constraints

Architectural validity constraints are important to express generic conditions that hold for most structures. In addition, they help steering the generation process towards the domain of plausible and feasible architectural models. For the examples in this paper, we define the following three rules, controlling traversability, gravity and occurrence.

Traversability This constraint aims at ensuring that the model is traversable, i.e. that all shapes in it are accessible, via adjacencies of type *traversal*. Without this, some architectural structure might never be reachable from the remaining ones. In formal terms, this means that the graph defined by the tile adjacency conditions of type *traversal* is connected.

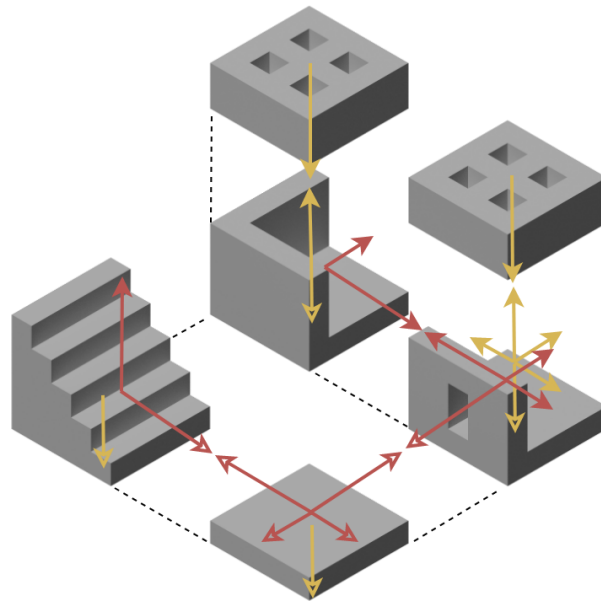


Figure 2.3: Semantics on tile adjacency conditions, in two types: *traversal* (in red) and *construction* (in yellow). The interior here, composed of a corner tile and a door tile, allows for *construction* above them, as indicated by the upward arrows, which are matched by the roof tiles above, with downward arrows. The door tile, allowing for a *traversal* entrance adjacency condition (towards the exterior), matches with the street tile, providing access to the interior.

Gravity In most architectural structures, each construction always stands or leans on some previous structure, possibly including the ground, rather than just floating above it. The gravity constraint aims at ensuring the connection of each shape to the ground, which can be itself regarded as a shape on its own. In formal terms, this requires that all graphs defined by the tile adjacency conditions of type *construction* are connected with the ground.

Occurrence The occurrence constraint allows for the specification of a desired density for a given shape in the model. In this way, one can control, for example, the density of house placing, ranging from, say, a rural village to a dense social neighborhood. It also allows to prioritize the occurrence of some shapes over others.

Summarizing, the complete solving process for an architectural profile (Eq. 2.2) consists of successively generating shape instances, and ‘attaching’ them to the model, by finding for them a location and orientation that (i) adheres to all shape adjacency conditions, and (ii) fulfills all validity constraints defined in the profile.

2.3. Evocative Examples

In this section, we present several examples of architectural profiles that are illustrative of the power of the concept. For this, we first define a simple architectural profile that forms the basis for all subsequent variants.

Our Base Profile uses the tile set T_S shown in Figure 2.2, and includes the Traversability and Gravity validity rules for all building constructions. The adjacency conditions for this Base Profile A are obtained from one example model, by registering all tile adjacencies found between its tiles.

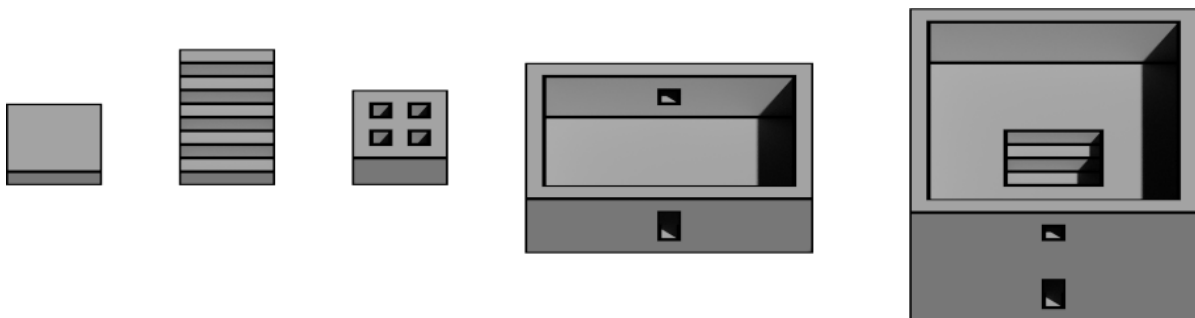


Figure 2.4: Shapes used in the profiles showcased in this paper: (from left to right) street, stairs, roof, one-story house, two-story house.

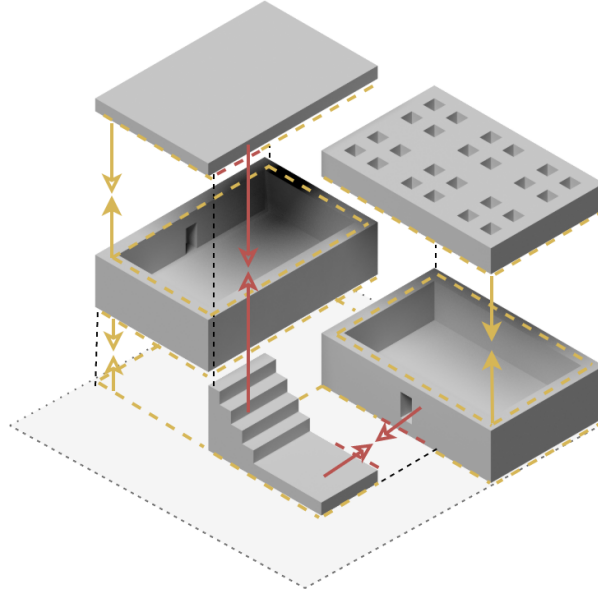


Figure 2.5: Shape adjacencies between shapes, indicated by the arrows, traversal in red and construction in yellow. Each shape adjacency can only exist if there is a corresponding shape adjacency condition defined in the profile.

The Base Profile employs the same shapes shown in Figure 2.4. Additionally, other derived profiles also use the ‘high-rise’ and the ‘monolith’ shapes, which represent larger buildings. Following the shape definition in the previous section, a shape is defined by specifying a label, an adjacency type and a bounding box. For example, a small one-story house can be defined with [traversal, interior, (3,2-3,1)], while other bigger one-story buildings can use larger values for the bounding box. In these profiles, Shape Adjacencies have a type, either traversal or construction which is indicated by τ and κ .

2.3.1. Examples

Table 2.1 presents four examples of architectural profiles, all of them extending the Base Profile above. For each of these profiles, we depict one model output by the solver for it.

One-story Profile

the street, one-story house and roof shapes are used to create a flat composition that is only traversed horizontally. The street is connected with shape adjacencies to itself and the one-story shape through the traversal. One-story shape is connected with the street, the roof and the ground through construction.

One-story Stacked Profile

the inclusion of a shape that allows for vertical traversal can dramatically change the generated models. We modify the [One-story] profile to become the [One-story] [stacked] profile. The stairs shape is added and it is connected using shape adjacencies with the street and itself through traversal, and with the ground through construction. Also a one-story house is connected to itself through construction. This results in the buildings being stacked on top of each other. Because stairs exist, walkways are being generated above the ground to connect the stacked buildings with each other.

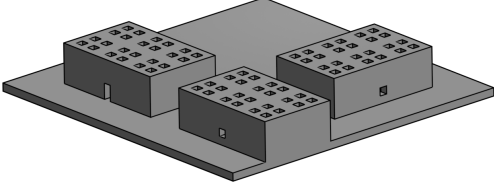
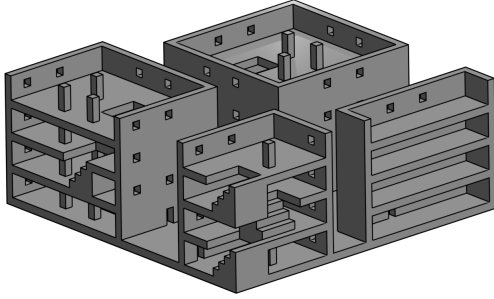
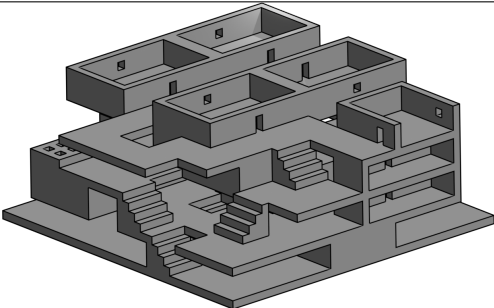
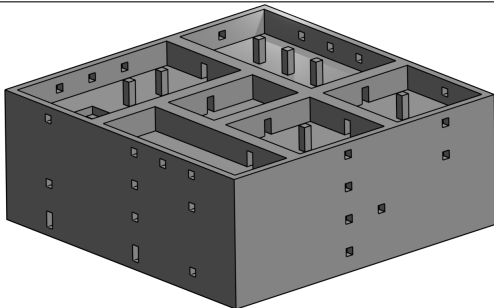
High-rise Profile

the [High-rise] profile is a modification of [One-story] profile, substituting the one-story shape for the much larger high-rise shape. This modification makes buildings multi-storied and increases the building volume.

Monolith Profile

in the [Monolith] profile, the monolith shape is used that takes all the volume of the input space. This results in the model consisting of one large building.

Table 2.1: Evocative examples

 <p>[One-story]</p>	 <p>[High-rise]</p>
 <p>[One-story][stacked]</p>	 <p>[Monolith]</p>

<p>[One-story]</p> $\begin{aligned} S &= \{ \\ &\text{street}, 1\text{story}, \text{roof} \} \\ S_A &= \{ \\ &\text{street} \xleftrightarrow{\tau,*} 1\text{story}, \\ &\text{street} \xleftrightarrow{\tau,*} \text{street}, \\ &\text{roof} \xleftrightarrow{\kappa,*} 1\text{story}, \\ &\text{street} \xleftrightarrow{\kappa,*} \text{ground}, \\ &1\text{story} \xleftrightarrow{\kappa,*} \text{ground}, \\ &\} \end{aligned}$	<p>[One-story][stacked]</p> $\begin{aligned} S &= ([One-story]).S + \\ &\{ \text{stairs} \} \\ S_A &= \{ \\ &([One-story]).S_A + \{ \\ &\text{stairs} \xleftrightarrow{\tau,*} \text{street}, \\ &\text{stairs} \xleftrightarrow{\tau,*} \text{stairs}, \\ &\text{stairs} \xleftrightarrow{\tau,*} \text{ground}, \\ &1\text{story} \xleftrightarrow{\kappa,*} 1\text{story}, \\ &\text{street} \xleftrightarrow{\kappa,*} 1\text{story} \\ &\} \end{aligned}$	<p>[High-rise]</p> $\begin{aligned} S &= \{ \\ &\text{street}, \text{highrise}, \text{roof} \} \\ S_A &= \{ \\ &\text{street} \xleftrightarrow{\tau,*} \text{highrise}, \\ &\text{street} \xleftrightarrow{\tau,*} \text{street}, \\ &\text{roof} \xleftrightarrow{\kappa,*} \text{highrise}, \\ &\text{street} \xleftrightarrow{\kappa,*} \text{ground}, \\ &\text{skyscraper} \xleftrightarrow{\kappa,*} \text{ground} \\ &\} \end{aligned}$	<p>[Monolith]</p> $\begin{aligned} S &= \{ \\ &\text{street}, \text{monolith}, \text{roof} \} \\ S_A &= \{ \\ &\text{street} \xleftrightarrow{\tau,*} \text{street}, \\ &\text{street} \xleftrightarrow{\tau,*} \text{monolith}, \\ &\text{roof} \xleftrightarrow{\kappa,*} \text{monolith}, \\ &\text{street} \xleftrightarrow{\kappa,*} \text{ground}, \\ &\text{monolith} \xleftrightarrow{\kappa,*} \text{ground} \\ &\} \end{aligned}$
---	--	--	---

3

Evaluating Comprehensiveness with an Expressive Range Analysis

We evaluate our technique by analysing the expressive range [31]. This evaluation technique analyses the variety a generator can produce in relation to the configuration effort. It has been used to analyse many PCG techniques such as a road generator [36] and several learned generators of 2D game levels [33].

3.1. Metrics

For our purposes, we define two metrics, density and repetitiveness, and evaluate how the output of various profiles falls within these metrics¹.

3.1.1. Density

Density is a common metric in architecture, indicating the ratio between interior and exterior space. We classify interior space as that of shapes that form an enclosed space, and we count the tiles in these enclosed shapes (*interior*). Exterior space then are (the tiles of) all remaining shapes (*exterior*). The density is given by $density = interior / (interior + exterior)$

3.1.2. Repetitiveness

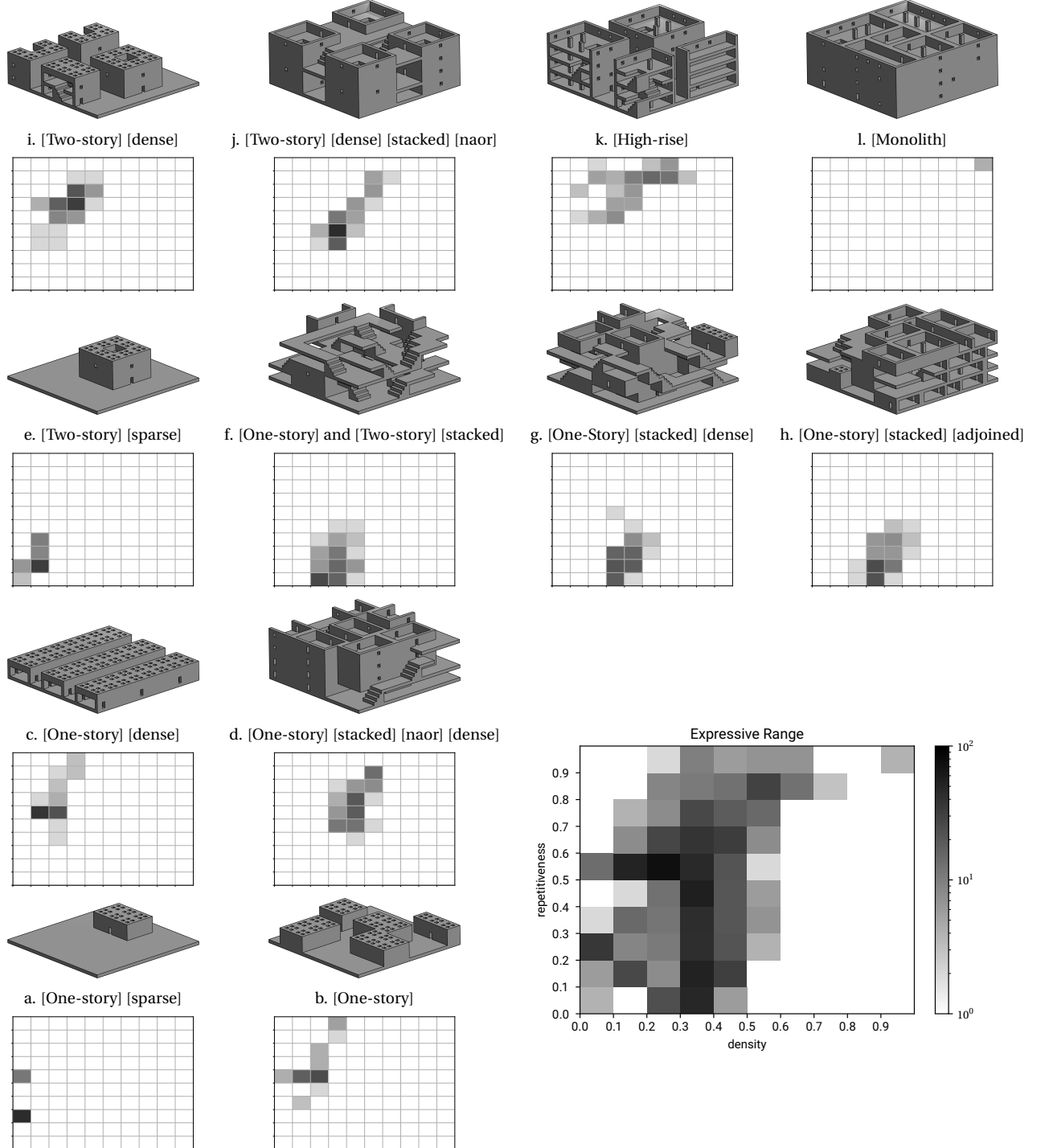
The repetitiveness metric indicates the prevalence of some patterns in a model. We focus only on the repetitiveness that occurs through the placement of buildings, i.e. regarding interior tiles. First the repetitiveness of all relevant tiles is calculated individually. We define the repetitiveness of an individual tile with a specific rotation as the ratio between r , the highest amount of times it is repeated a specific distance apart and r_t , the total occurrence of that rotated tile in the model. The repetitiveness for the entire model is calculated as the weighted average of the individual repetitiveness for all (interior) tiles, where each weight is the normalized occurrence of the corresponding rotated tile in the model.

3.2. Model generation

We construct twelve profiles for the analysis. A maximum of 100 model samples are generated per profile. Each model is of size $8 \times 8 \times 4$ cells. We use a concise naming scheme, previously used in the evocative examples section, to describe the different profiles. This convention indicates the building shapes used and additional other designations that alter the profile. The [stacked] designation means that buildings are allowed to stack on top of each other and the stairs shape is added to allow for vertical traversal. When [sparse] or its opposite [dense] is used, an occurrence rule is used on the building shapes in the profile, [sparse] indicating one building shape in the Model space in total, and [dense] indicating at least one building shape in every equal quadrant of the Space. [adjoined] means that separate building shapes can directly be connected door to door. Lastly, [naor] stands for no access on roof, meaning the top of buildings is never walkable (which they otherwise are).

¹The contents of this chapter have been published at IEEE COG 2020 [1] and have been co-written with Rafa Bidarra.

Table 3.1: Expressive range analysis of twelve architectural profiles: above each bin plot histogram, one of its sample models is shown. Lower right corner accumulates all expressive range results (color indicates the amount of models in a bin).



3.3. Exploring the expressive range

Our metrics result in a two dimensional evaluation space with noteworthy properties shown in Table 3.1. We now discuss each quadrant of this space.

3.3.1. Lower left

This quadrant indicates models that are varied and are sparsely built. The corner point is the empty model, no density and therefore no possibility for patterns. Profile *a* and *e*, that can only build on the ground and are sparsely built, naturally reside in this space. Also many profiles that stack buildings on top of each other are placed in this quadrant, *f*, *g* and *h*. These profiles are varied due to the existence of a lot of possibilities to stack buildings in a disordered way. To connect these freely placed buildings, a large amount of walkways are placed, limiting the density.

3.3.2. Upper left

This quadrant contains repetitive sparsely built models. Occupied by profile *b*, *c* and *i*. No models exist for the corner point, because our repetitiveness metric is zero if the density is zero.

3.3.3. Lower right

This quadrant has dense models with little repetitiveness. In the profiles that we have included in this investigation, one cannot find such models; these would need more tiles than the present limited tile set of 10 tiles, which does not provide a large enough variety of combinations.

3.3.4. Upper right

This quadrant is dense and highly repetitive. This is occupied by models from high-rise(*k*) and monolith(*l*) profiles, as these fill much space with repetitive buildings elements. The repetitiveness here stems predominantly from vertical placement patterns.

Looking closer at the impact of small changes between profiles, the following observations can be made on the changes in the expressive range.

Profiles *d* relative to *g* show the impact of the [naor] designation to disallow traversal on the top of buildings. *d*'s Profile is more constrained and a subset of *g*'s Profile, but the samples of *g* mostly stay out of the area covered by *d*'s samples instead of covering the whole space evenly. This suggests for profile *g* that model distribution is biased towards low repetitiveness. The declarative control added by [naor] in profile *d* is useful to create higher repetitive models.

The sample distributions for profiles *b* and *c* are roughly the same, although *c* is a subset of *b*. At the same time the lower repetitive models in *a* are not covered in *b*. This suggests that *b*'s profile model distribution is biased to be dense.

3.4. Implementation

We have used Answer Set Programming to implement architectural profiles. This implementation consists of a dynamic part, that converts the input space and an architectural profile to ASP, and a static part, consisting of fixed logic written in ASP to constrain the Model to architectural profile adherence (described in Section 2.2). These two parts are combined and given to an ASP solver, in our case, the off-the-shelf solver clingo [7]. The solver then outputs a model that adheres to the architectural profile. For every new model generated the solver is restarted with a new random seed.

Solving an architectural profile is a complex problem, as it is generally as hard as tile solving (which is proven to be NP-hard[20]). Evidence of this complexity is also given in Figure 3.1, which shows an exponential run-time growth with Model size. To make solving an architectural profile computationally feasible we solve shapes and profiles separately. First, given the shape definitions in a profile, a finite number of shape instantiations are generated and stored. Subsequently these are made available in the profile solver to build models with.

Additionally, we keep low the amount of unique shapes used in the profile solver: two shape instantiations per shape definition. Our ASP implementation only makes use of hard constraints (no soft constraints were used). Finally, the current implementation is not optimal, and can be improved by addressing all performance bottlenecks, exploring further use of solving heuristics in clingo, and making use of multi-shot solving within clingo [8].

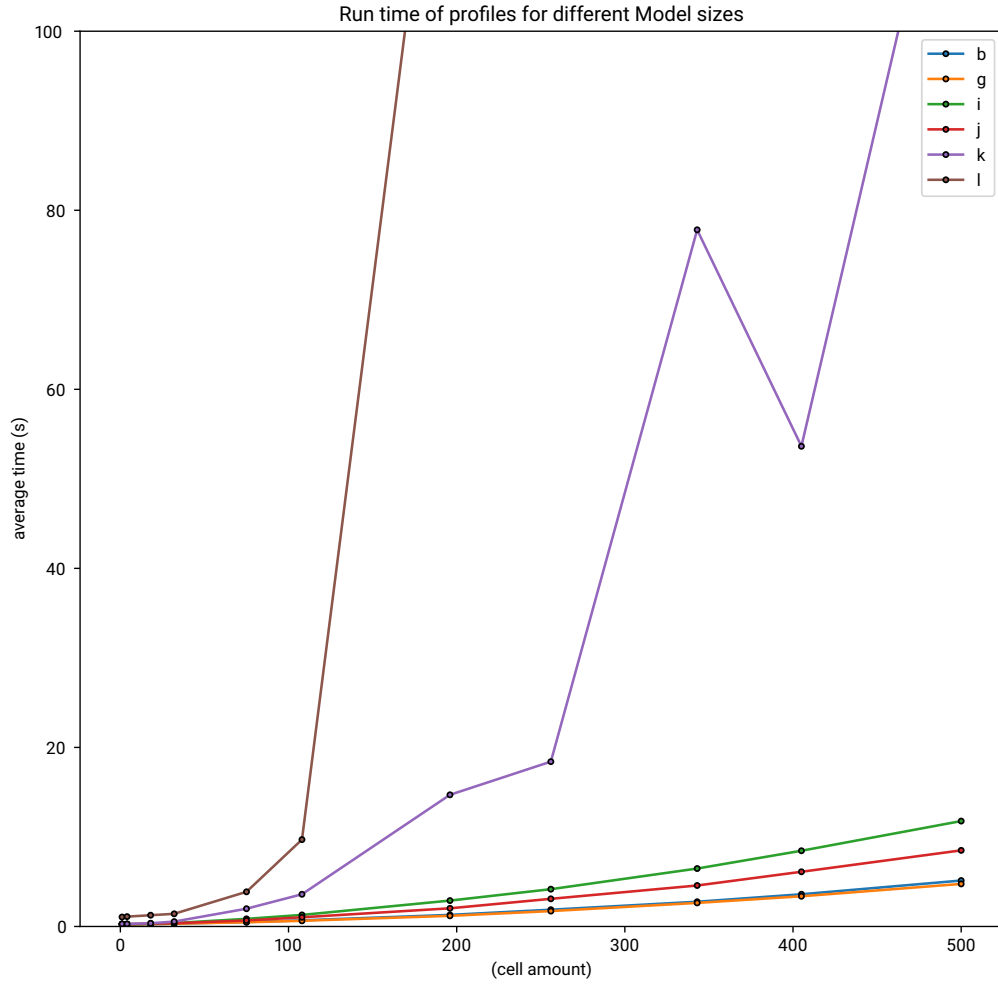


Figure 3.1: Runtime analysis of the implementation. The different colored plots correspond to different profiles from Table 3.1. Every data point is the average runtime of 10 model generations. The size of the shapes in the profile has a large impact on runtime: profiles that place bigger shapes, such as high-rise shapes and monolith shapes in profile *k* and *l*, take much longer to solve than profiles with small one-story buildings, such as profile *b*. Interestingly at 343 cells, profile *k* is slower than at 405 cells. We think this is due to the height of the model size and profile *k* (that creates high-rise buildings) being disproportionally effected by the height for its solving speed, the former is $7 \times 7 \times 7$ cells, higher than the latter, $9 \times 9 \times 5$ cells.

4

Adaptable Architectural Profiles with an Iterative Solver

In chapter 3 an implementation for generating architectural profiles(AP) was discussed. This was used to evaluate the expressive range. The implementation, written in answer set programming, became exponentially slower with an increase in input size and shape size. The technique also does not take the terrain into account, always generating on a flat surface.

We think the usefulness of architectural profiles can be further extended when we increase its applicability. Specifically interesting would be the technique working well in the challenge of Settlement generation. Settlement generation is a PCG problem that involves building settlements on a pre-existing landscape. The focus, different from city generation, is that settlements adapt to the terrain, their placement looks believable given the terrain. The terrain may be big, so the technique must handle larger input sizes. The terrain may be hilly or partially uninhabitable and consist of different biomes, asking for a different type of settlement. Incidentally, a current settlement generation competition called Generative Design in Minecraft Competition (GDMC)[26] challenges contestants to write a settlement generator for an arbitrary Minecraft map. We believe that if we do well in settlement generation, then our approach increases the applicability of architectural profiles.

Therefore we discuss a new approach that applies architectural profiles to the challenge of settlement generation. Our approach consists of a terrain analyser (section 4.3), that converts the terrain into a tiled representation on which architectural shapes can be placed. Then the iterative AP solver greedily places architectural shapes on the tiled representation (section 4.4), where tiles placed to form shapes are constrained by WaveFunctionCollapse [10] (WFC).

4.1. Related Work

Relevant work to our approach are Settlement generation techniques either used to create new settlements or reconstruct historic ones. It is also relevant to look at previous results of the GDMC competition.

4.1.1. Settlement generation and reconstruction

Settlement generation is a PCG domain where settlements are fittingly placed on a pre-existing terrain. It involves multiple PCG areas that are combined, such as road and parcel generation, exterior building generation and interior building generation.

A terrain with hills can be inhabited with small villages[5]. An iterative process is used to create settlement seeds on the landscape and road connections based on closeness to water or terrain elevation. The land around the settlement seeds is divided into parcels which are subsequently covered with houses using a shape grammar that adapts to slopes. Building further on settlement seeds, changes in technological and architectural periods can be taken into account during the growth of the settlement[42]. An agent based city generator can adapt based on resources focussing on the creation of a road network[34]. Agent based systems simulate the development of modern urban cities[41] or the evolution of historic fortified cities[17]. Similar to settlement generation is the reconstruction of historical habitation. Geographic maps are combined with Procedural techniques to create three dimensional reconstructions. The city of Pompei[21], informal South

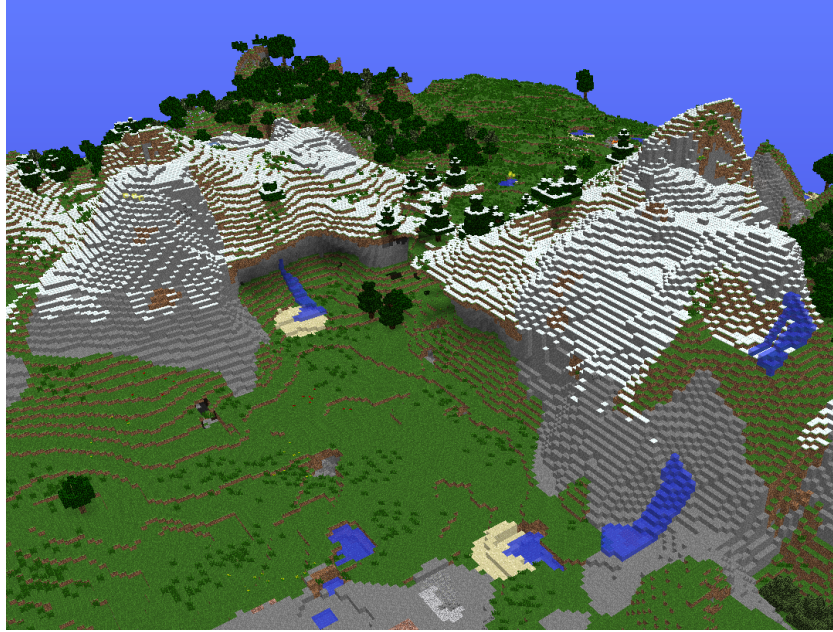


Figure 4.1: An example of a minecraft terrain. This biome consists of mountainous terrain, lakes and is snowy.

African settlements[9] and ancient Greek settlements in Central Zagori[14] have been reconstructed procedurally.

4.1.2. Generative Design in Minecraft Competition

GDMC is an annual competition[26] that challenges contestants to write a procedure to create functional and evocative settlements on Minecraft maps, for example depicted in figure 4.1. The maps are not disclosed before hand, so the procedure needs to take into account many features of Minecraft terrains. Evaluation of the performance is done by a number of experts from different fields: architecture, PCG research and game design. Qualitative metrics are used based on adaptability, functionality, evocativeness and aesthetics. The 2018 entry that won the competition created settlements that were well embedded on the terrain. Building locations are evaluated on random positions and placed on the most adequate. Subsequently the built structures are connected with roads using the A* algorithm for path finding. Although all entries were inventive and surprising, a big improvement between the winner and other entries was adaptability with the terrain. Some other entries deformed the terrain too much to the point that it became unrecognizable. Some built too much or too high, which seemed out of place in a natural environment.

4.2. Our pipeline

We will now discuss our approach to apply architectural profiles to settlement generation. Our pipeline is shown in figure 4.2. This is a variation on our approach discussed in chapter 2, figure 2.1 where an input space and only one profile served as input for the solver. The solver consists of the terrain analyser and the iterative AP solver. The terrain analyser returns an analysis of the terrain which is then used by the Iterative AP solver to place settlements on the terrain. After the settlements are generated, a number of post-processing components are used to improve them, and henceforth, combine them with the pre-existing terrain.

4.3. Terrain analyser

The terrain analyser analyses the terrain to instruct how architectural profiles are to be employed to create adapted settlements. The analysis consists of (i) discretization of the terrain to a cell-based representation, (ii) categorization of the terrain based on its biome and relative position, (iii) and using this categorization to assign profiles to areas. Which decides what kinds of structures are placed in certain areas.

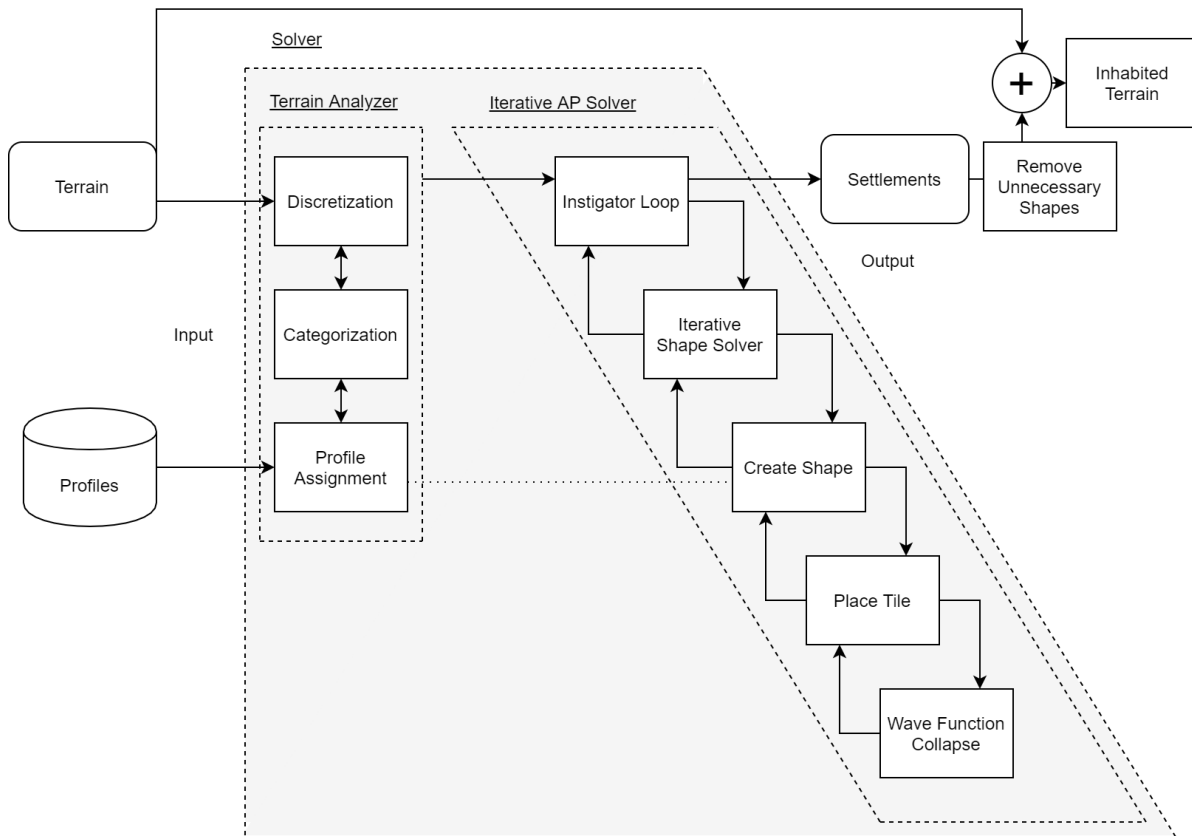
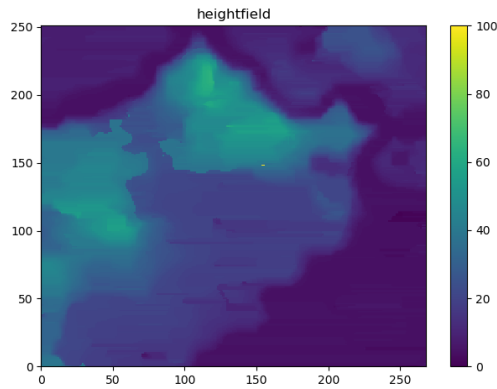


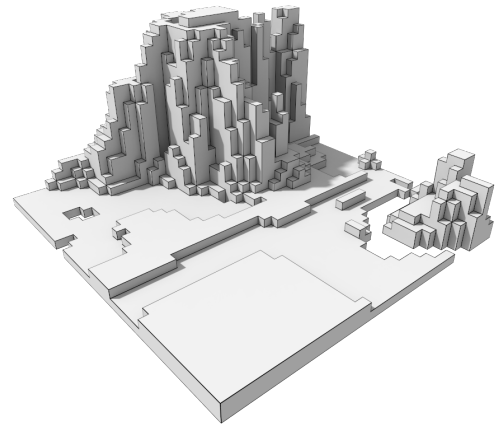
Figure 4.2: Our pipeline for the iterative approach is the following. We take as input a terrain and a number of profiles which serve as parameters for the solver. The solver consists of, the terrain analyzer and the iterative AP solver. The terrain analyzer discretizes the terrain, categorizes the terrain in areas and assigns the architectural profiles to these areas on the terrain. The iterative AP solver places appropriate shapes on the discretized terrain which form settlements. Remove unnecessary shapes henceforth improves the settlements by removing greedy artefacts. Lastly, the original terrain and the settlements are combined to create the inhabited terrain.

4.3.1. Discretization of the terrain

Minecraft terrain is succinctly described as a three dimensional voxel map. Each voxel is called a Minecraft *block*. Each block is one cubic meter and hundreds of different types exist. Block types have a wide range of gameplay consequences, they can form crafting materials and some, like lava, damage the player. In our discussion we focus on the following blocks: water, air, tree and ground(which include dirt, stone and sand). We divide the terrain in cells, so it can serve as input for a tile-based architectural profile solver. We create a *height map* of the terrain, an example shown in figure 4.3a, by noting the highest block that denotes the terrain surface, consisting of water and ground blocks, ignoring blocks that are not part of the surface, like trees and grass. From this height map we create a discretization of the terrain shown in figure 4.3b. The terrain is converted into cells, and all cells that are below the height map are assigned the *ground tile*. This will allow for construction in the generation phase. If water blocks are found on the surface, then we want to avoid building there. To specify this, we fill that entire column of cells with void tiles. Which indicates that no construction can take place there.



(a) Heightmap

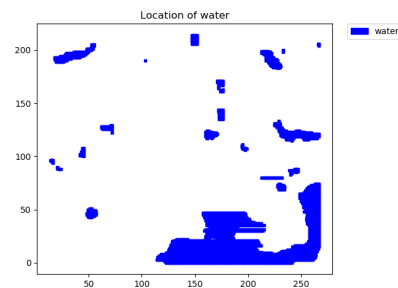


(b) Discretized terrain with filled with ground tiles.

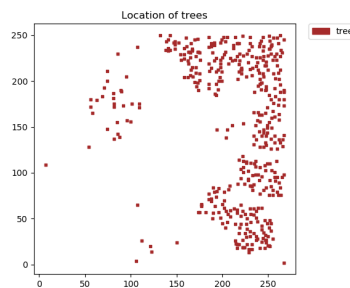
4.3.2. Categorization of the terrain

In the previous step the buildable areas are identified by using the height map. The height map contains for every position the Minecraft block value found on the surface and the height of the block. In the categorization step the heightmap will be used to identify the properties of the buildable areas on the terrain.

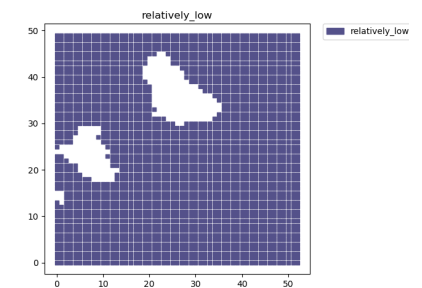
- **Relatively high/Relatively low** indicates whether a buildable area is relatively high or low compared to others. This can be used to make a distinction between settlements in a valley and settlements on a hill.
- **Woods** indicates whether the area is forest like, meaning it is occupied by a significant amount of trees.
- **Close to water** indicates whether the area is close to a large body of water.



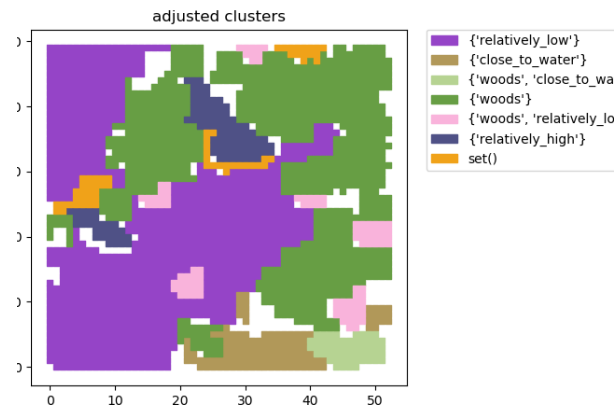
(a) Water



(b) Location of trees



(c) The terrain that is relatively low. The terrain that does not overlap with these points is considered relatively high.



(a) Categorization of the terrain in clusters with specific properties. These areas will be used to assign profiles to the terrain.

This results in a two-dimensional categorization map of the terrain. An example is shown in figure 4.5a.

4.3.3. Assigning profiles to the terrain

The assignment of profiles is done by performing the following algorithm on the categorization map. A random spot is chosen on the map and the properties are registered. Through a search all connected spots are found that have these same properties. When these spots are exhausted the procedure is done again until all spots are covered. This yields a set of unique property combinations. We randomly choose a profile for each unique property combination and henceforth apply this profile on all spots identified with that property combination. We recognize that using a specification instead, that would allow declarative control over these assignments is beneficial, this is future work.

4.4. Iterative AP Shape Solver

The terrain analysis returns a space of cells and a profile assignment for each cell. From this information, the Iterative Shape Solver can build appropriate settlements following the assigned profiles at appropriate places. Henceforth we introduce a novel iterative approach to generate settlements with tile-based architectural profiles. This technique, unlike the exact solver discussed in chapter 2, is heuristic-based, and employs a greedy strategy. This approach is more appropriate for settlement generation, because it allows for iterative generation, which means it can handle a larger input, can more easily adapt to different terrains and can incorporate multiple profiles.

In figure 4.2 the hierarchical process of the Iterative AP solver is shown. These consist of four hierarchical subcomponents that are tasked with generating on different architectural scale levels, ranging from settlements to placing tiles. The instigator loop (subsection 4.4.2) places settlements, deciding on the starting point of its placement. The Iterative Shape solver (subsection 4.4) places shapes. The place Tile procedure (subsection 4.4.5) places tiles constrained by the shape the placement belongs to. Tile placement is further constrained by WFC that indexes what tiles are still available for placement.

We will first explain notions used throughout the Iterative AP solver in subsection 4.4.1, as well as our usage of WFC for placing tiles. After that we will explain the Iterative AP solver procedure.

4.4.1. Preliminaries

Our approach involves two tile solving techniques. Architectural profiles that constrain clusters of tiles called shapes. And the WaveFunctionCollapse (WFC) algorithm[10], that heuristically fills a space with tiles while satisfying the adjacency conditions. We will discuss our usage of WFC. We also discuss attach points, which are necessary to iteratively place new shapes on pre-existing shapes. Related to this is the notion of closed shapes, which we will explain. We allow profiles to work on mountainous terrain by adding ground tiles and ground shapes. Lastly we discuss a new distinction of primary and secondary shapes.

Using Wave Function Collapse

WaveFunctionCollapse[10] is a greedy tile solving heuristic[12], represented as a *Tile constraint profile*, it consists of tiles, adjacency conditions and optional weights for tile occurrence. WFC creates from the input tiles, unique $N \times N \times N$ patterns which are put in an indexing structure. During the solving process WFC uses the notion of states, the state of a cell are the possible pattern placements at the current iteration for that cell. It indicates what patterns can be placed in a cell without breaking adjacency conditions. The goal of the algorithm is to have exactly one state at every cell. At the start, all cells have all states. Then, iteratively, WFC picks a cell with the least amount of states and randomly assigns one of the states in the cell, removing all others. If a cell has only one state, then the associated tile is placed in the cell, as there are no other possibilities left. Then this information propagates to all other cells. Which means that neighbouring cells potentially lose states due to adherence to the adjacency conditions. This greedy algorithm finishes if all cells have one state, which automatically means that every cell has a tile placement. If there is a cell that has no state, then the algorithm has failed. It can be restarted for another attempt.

WFC uses states and state propagation to place tiles without breaking adjacency conditions. In our iterative approach, WFC is used as a sub procedure, that constrains the placement of tiles based on adjacency conditions. It does, however, not have knowledge of shapes, which are essential to generate *Architectural profile* models. These are handled by separate, higher level components. We use WFC during tile placements to help develop a shape. Therefore we use a pattern size of $1 \times 1 \times 1$ for our WFC usage. Meaning that a state pattern is a single tile. If the pattern size was greater, then it would be necessary for WFC to take shape constraints into account, which we have not implemented here. Employing bigger patterns, and handling this overhead in terms of shape constraints is future work.

We also make an addition to our WFC adaptation, that may also be an improvement to the algorithm itself. We use the concept of *releasing* cells, when they are given an incorrect tile assignment. Releasing a collection of cells means they are reset, signifying that they are given all initial states back. Then they propagate this change to neighbouring cells, which may receive additional states accordingly. This is an alternative to restarting the entire algorithm, if there is a failure, which would be too costly in our situation.

Lastly, we limit the WFC global state propagation in our system. Instead of propagating to all cells, we only do it a limited amount of times. This means that propagation stays local. The reason is because state propagation, a recursive procedure, is very costly. And in settlement generation, we do not necessarily want to fill all cells with shapes. When a small village of one-story houses is created for example, the cells upwards in the sky do not need to be part of the computation because no shape will ever reach it.

Attach Points

The Iterative Shape Solver is centered on iteratively places shapes on attach points. *Attach points* are provided by existing shapes that have typed adjacency conditions pointing outside of itself, a concept which is previously explained in the approach of architectural profiles in section 2.2. There is an attach point on the outward adjacency condition of an existing shape, for every profile shape that is allowed to be connected, based on the shape adjacency conditions. Every iteration of the iterative shape solver, an attach point is chosen and an attempt is made to place the new shape. Attach points are ephemeral, they can only be used once, to ensure the technique terminates.

Closed shape

A shape is *closed* when it does not expose a typed adjacency condition(that is not an entrance) to outside, without being connected with another shape. An example of an open shape, is a room without a roof. In this case the top is not covered although it should be. In other words, the tiles of the room shape have typed adjacency conditions(in this case construction) pointing upward, but there is no roof shape, or other room shape that is placed on them, meaning the shape is not closed. To create sensible structures, all shapes need to be closed in the model. These outward adjacency conditions are represented as attach points, therefore the iterative AP solver aims to close all shapes, by placing shapes on attach points until there are no attach points left.

Ground shapes

An important feature of the approach is that the solver can handle a pre-existing terrain surface, which may be hilly or mountainous. Therefore the input space of cells given to the solver can be one of the following:

- Unassigned, denoting buildable areas

- Assigned the ground tile, to support construction
- Assigned the void tile, denoting not buildable areas

For every input cell that is assigned the ground tile, a ground shape is registered at that location. The ground shape is a new shape that is added to all profiles used in the iterative solver. The ground shape has a single entrance adjacency condition upwards (z-direction), of type construction. This means that all shapes in the profile with a shape adjacency condition with the ground, can attach on the ground below. These initial attach points create the start of the placement of shapes on the terrain.

Distinction between Primary and secondary shapes

We will use the notion of *primary shapes*, those that are leading and *secondary shapes*, those that are supporting. We use it to distinguish shapes that have a purpose on their own, like buildings. And shapes that only make sense in relation to primary shapes, such as roads, which only make sense if they lead to primary shapes. We will use this notion later to explain the removal of unnecessary shapes, that do not lead anywhere, in section 4.5.

4.4.2. Instigator loop

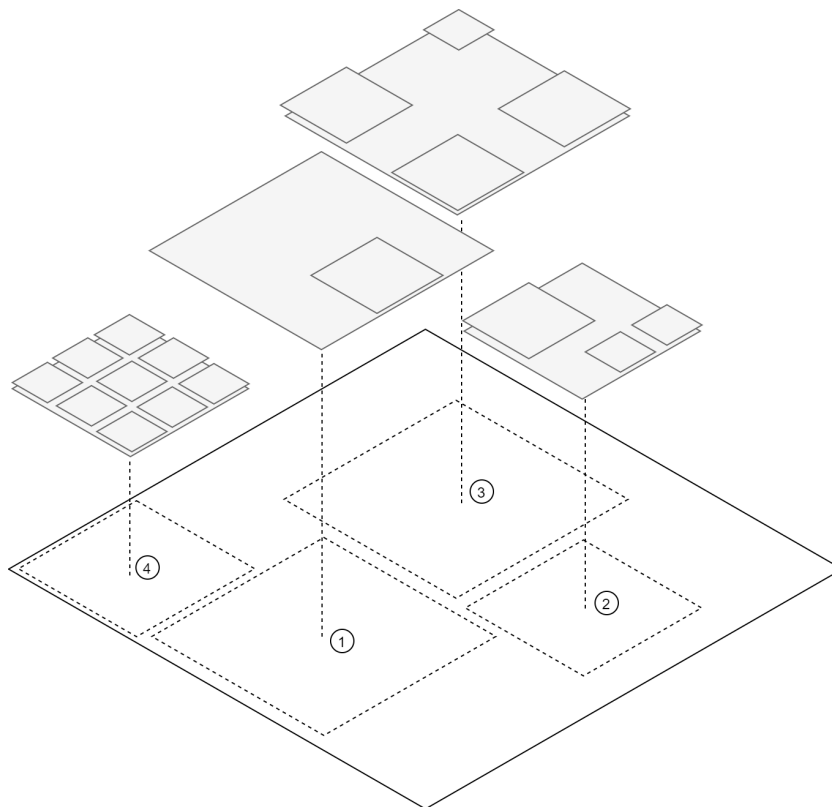


Figure 4.6: The largest scale of the approach is the terrain with all profile assignment areas. Iteratively a starting point is chosen and the iterative Shape solver where the iterative AP solver creates a settlement. In this figure the iterative Shape solver has been called four times.

The main method of the solver, procedure 1, figure 4.6, accepts the cells and profile assignments from the terrain analyser. The method is a loop, where every iteration a cell is fed to the `IterativeShapeSolver` method, that has not yet been visited. Because `IterativeShapeSolver` is called multiple times on different separate locations in the terrain, the technique handles disjoint settlements on the terrain.

4.4.3. Iterative Shape Solver

The Iterative Shape Solver is given the task to grow a settlement from a starting position shown in procedure 2, and visualized in figure 4.7. All attach points at the starting position are registered. Then weighted randomization is used to choose an attach point. The weights are set with the shape *occurrence* defined in the

Algorithm 1 *MainLoop(cells, profileAssignments)*

```

visited  $\leftarrow \{\}$ , shapes  $\leftarrow []$ 
while visited  $\subset$  cells do
  startPoint  $\leftarrow$  chooseRandom(cells – visited)
  newShapes, newlyVisited  $\leftarrow$  IterativePlaceSolver(startPoint, cells, profileAssignments)
  visited  $+$  newlyVisited; shapes  $+$  newShapes
end while
return shapes

```

profile. A high occurrence will mean a high weight, resulting in that particular shape probabilistically occurring more often in the model. Then **Create Shape**, procedure 3, is called that will return a new shape placed at that position if successful. Success means that the returned shape is placed and the attach points are updated with the existence of this new shape. If the shape creation was not successful, the shape is released, by calling **releaseShape**. This means that the cells of the shape are released and this change is propagated for all neighbouring cells. The released cells obtain all initial states and surrounding cells potentially obtain additional states from the propagation. However, cells that are already part of other shapes, are exempted from this propagation. These are not allowed to change their state any more. This ensures that existing shapes are preserved. To highlight the greedy attributes of this method, in figure 4.8, an characteristic example is shown of how shape placements at earlier iterations greatly influence placements in the future.

Algorithm 2 *IterativeShapeSolver(startPoint, cells, profileAssignments)*

```

visited  $\leftarrow \{startPoint\}$ 
shapes  $\leftarrow []$ 
attachPoints  $\leftarrow$  findAttachPoint( $\{startPoint\}$ , profileAssignments)
while |attachPoints|  $> 0$  do
  currentAttachPoint  $\leftarrow$  chooseAttachPoint(attachPoints)
  profile  $\leftarrow$  profileAssignments[currentAttachPoint]
  newShape, success  $\leftarrow$  createShape(attachPoint, cells, profile)
  attachPoints – attachPoint
  if success then
    shapes  $+$  newShape
    attachPoints  $+$  findAttachPoints(surrounding(newShape), assignments)
    attachPoints – AttachPointTowardsPositions(newShape.positions)
    visited  $+$  newShape.positions
  else
    releaseShape(newShape, cells)
  end if
end while
return shapes, visited

```

4.4.4. Create Shape

Create Shape is ordered to develop a certain shape at a position. Starting from the initial cell position, adjacent cells are selected to develop the shape. A shape, as previously defined (section 2.2), consists of a label, a typed adjacency and a minimal/maximum bounding box. In this procedure we will constrain based on the bounding box. The shape must stay within the constraints of the bounding box. The procedure greedily steers towards this by filtering out tiles that would break this constraint. This filtered set of tiles is then sent to the **placeTile** procedure. It will receive back whether one of these tiles was placed successfully. If not, the procedure returns that the shape failed to develop. The developing shape is done, when it cannot grow any further. Meaning it does not have the typed adjacency, as defined for the shape, outwards. If the shape is done, but smaller than the minimal bounding box, it has failed to develop. In contrast, if it is not done, but reached the maximal bounding box, it failed as well. Some other typical examples of shape failures are shown in figure 4.9. If the shape has gravity constraints then it is checked whether the shape does not float in the air by looking at the neighbouring shapes and whether there is support, as discussed in architectural profiles.

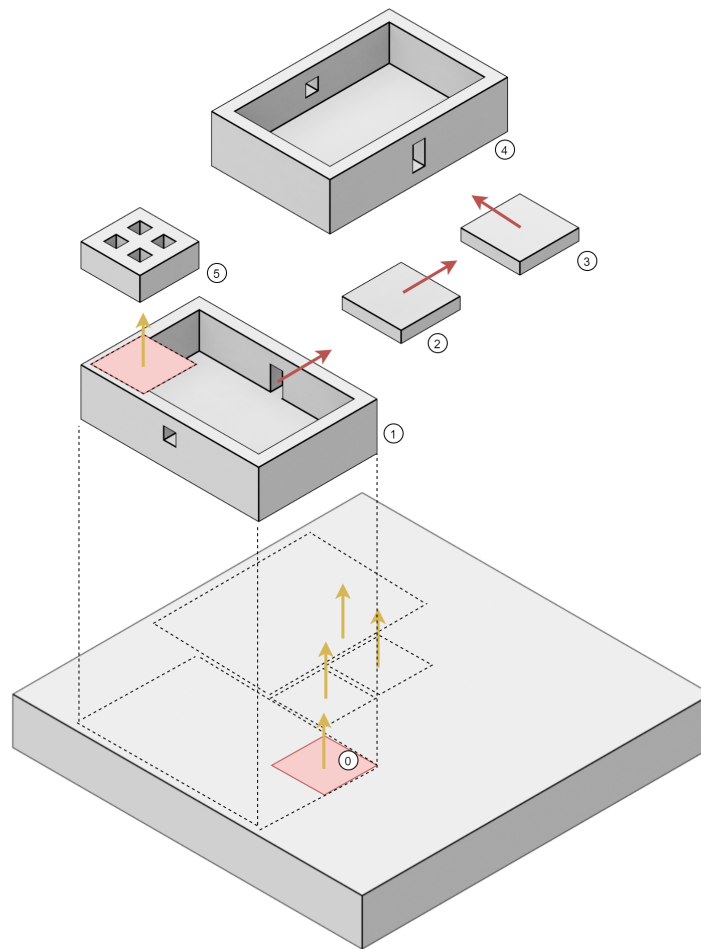


Figure 4.7: The first 5 steps of typical iterative placement of shapes. At the start a ground tile (0) is chosen which exposes an attach point. On this construction attach point a shape is placed defined by the shape adjacencies and the occurrences for potential shapes defined in the profile. Here a room shape is chosen (1). The room shape exposes a new traversal attach point, a door to the outside. A street shape is placed on this attach point (2). In the same way another street shape (3) and a room shape is attached (4). The room shapes also create construction attach points, a roof shape is placed on top of it (5). Given the profile, rooms or street shapes can also be placed on top of rooms, but sensibly only if the top is reached through an traversal attach point, to enforce that all shapes can be reached.

Then and only then the **gravity** method returns true.

4.4.5. Place Tile

Lastly in this iterative procedure we select a tile that further develops the shape, visualized in figure 4.10. The tile that can be placed are constrained by (i) the existing states of the corresponding cell (provided by WFC), (ii) whether cells have the shape label and (iii) whether the chosen tile connects through the associated typed adjacency with the developing shape. If there is a tile that fits all these requirements, it will be placed by using WFC with *WaveFunctionCollapseStep*. The call will place the tile and propagate the change to other cells meaning they potentially lose states. If there are no satisfactory tiles, the procedure returns back that it failed.

4.5. Removing Unnecessary Shapes

The iterative AP solver creates structures iteratively in a greedy manner, as previously highlighted in figure 4.8. This approach makes it possible that buildings are stacked on top of each other, while still maintaining traversability between all shapes. The top of a previously created building can be reached through a walkway

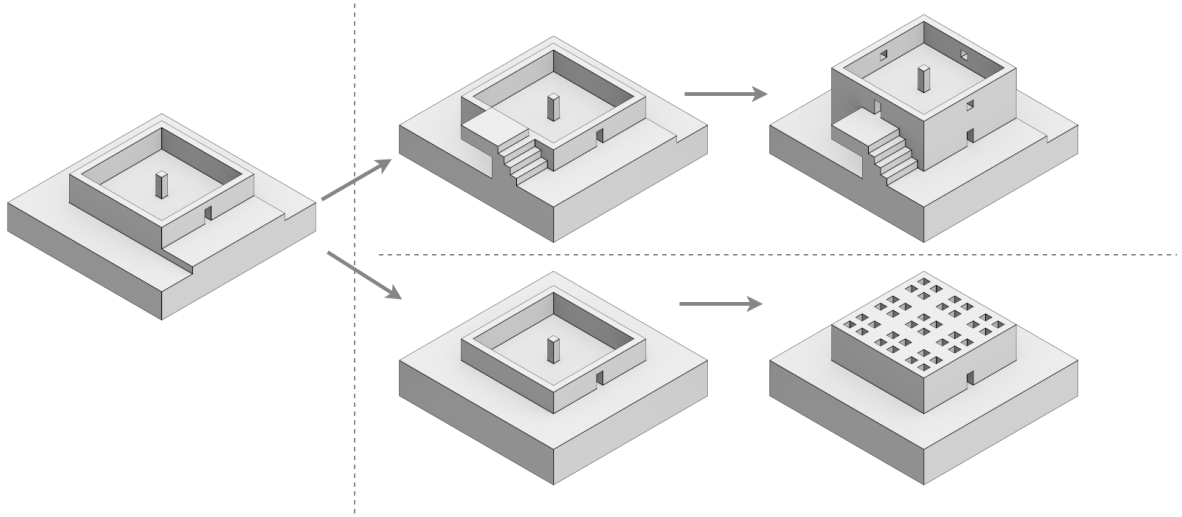


Figure 4.8: The first model to the left is shown on the crossroads of two possible continuations of its development. At the top shapes are added that make it possible to create a new building on top. On the bottom, instead, the building is surrounded with street shapes, therefore the possibility is lost for a building to be placed on top of it. Instead roof tiles are placed to finish the shape.

Algorithm 3 *createShape(attachPoint, cells, profile)*

```

position, shapeCriteria  $\leftarrow$  attachPoint
shape  $\leftarrow$  {}
candidates  $\leftarrow$  {position}
while  $\neg$ done(structure) do
    cell  $\leftarrow$  chooseCell(candidates, shapeCriteria)
    tile, success  $\leftarrow$  placeTile(cell, shapeCriteria, cells, profile)
    shape.add(tile)
    candidates  $\leftarrow$  getCandidatePositions(shape)
    if  $\neg$ success then
        return shape, false
    end if
end while
return shape, gravity(shape, cells)

```

Algorithm 4 *placeTile(cell, shapeCriteria, cells, profile)*

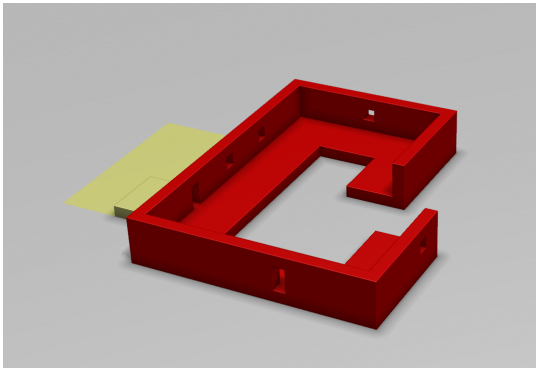
```

possibleTiles  $\leftarrow$  shapeCriteria.tiles
possibleTiles  $\cap$  = neighborAdjacencies(cell, cells, shapeCriteria.adjacencyType)
tile, success  $\leftarrow$  WaveFunctionCollapseStep(cell, possibleTiles, cells)
return tile, success

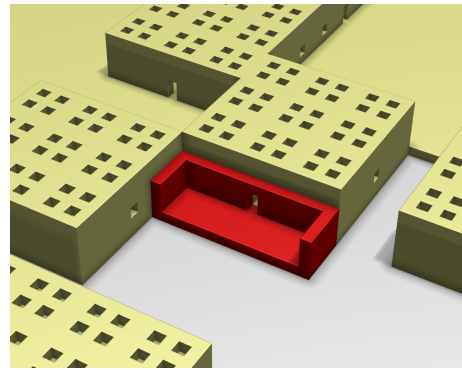
```

created later in the solving process and now the possibility arises for a building to be placed there. But this procedure can also lead to walkways that have dead ends, because an anticipated destination, a building, was never created. Failing to create a destination results in unnatural artefacts of our greedy procedure, such as stairs shapes that end in the air, and the aforementioned walkways that create dead ends. To circumvent such unnatural artefacts we create the following post-processing step after the solver returns settlements, described in procedure 5 and visualized in figure 4.11. We identify shapes that are primary, for example buildings, and secondary, walkways for example. For every traversability rule we make sure that secondary shapes only exist if they are part of the reachability. Otherwise, the shape is removed, which means that *releaseShape* is called on that shape.

The removal of secondary shapes might cause persisting shapes to lose their closed shape status, shown in figure 4.11, stage three. Therefore after the removal process, attach points are reintroduced at all positions



(a) A shape, that has failed to develop because our architectural profile, due to the tile set used, does not support non rectangular or convex shapes.



(b) A shape that has failed because if fully developed would have obstructed the window of an adjacent shape.

Figure 4.9: Typical ways shape development fails.

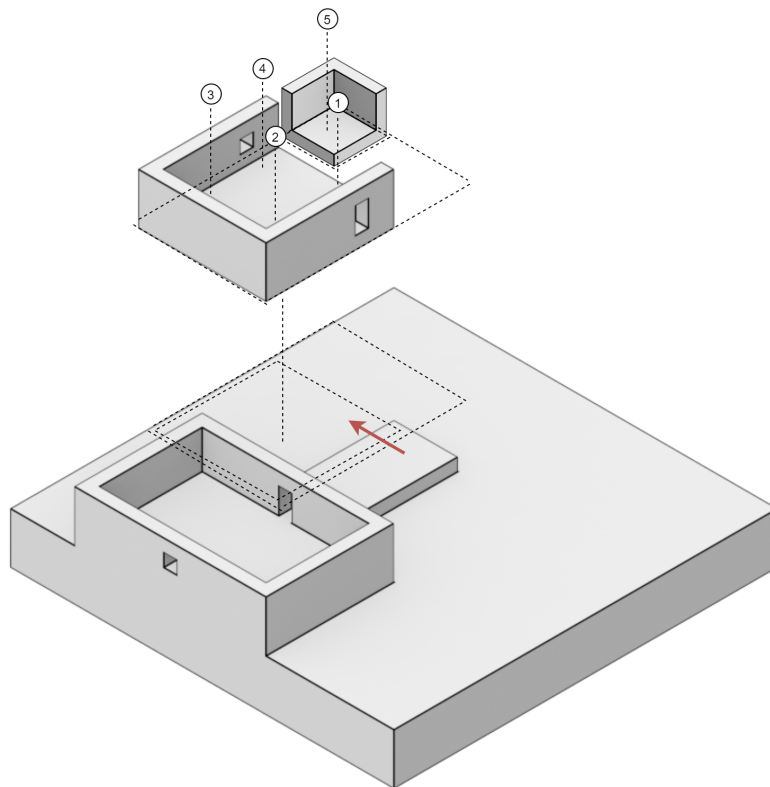


Figure 4.10: On the smallest scale, individual tiles are placed to fulfil the creation of a shape. Here a room shape, placed in step 4 in figure 4.7, is in the process of being created, and has partially been filled in by iteratively placing tiles. A door tile is placed(1), connecting the attach point with the new shape. The current iteration is at (5), where a decision is made to place a corner tile. The corner tile adheres both the shape constraints and the WFC states and is therefore placed next.

with outward typed adjacency conditions of persisting shapes that were previously closed. Shapes are then placed there, that may not break the traversability rules defined in the profile. In the profiles that we have defined, the result is that roof shapes are placed at these positions, to close persisting shapes.

Algorithm 5 *RemoveUnnecessaryShapes*(*adjacencyType*, *primaryShapes*, *secondaryShapes*)

```

destinationShapes ← {}
for shape in primaryShapes do
    destinationShapes+ = neighborShapeAdjacencies(shape, shapes, adjacencyType) ∩ secondaryShapes
end for
paths ← {}
for a, b in (destinationShapes × destinationShapes) do
    if there exists no transitive route in paths between a and b then
        paths+ = shortestPath(a, b, secondaryShapes)
    end if
end for
shapesToRemove ← secondaryShapes − shapesInPaths(paths)
for shape in shapesToRemove do
    releaseShape(shape)
end for

```

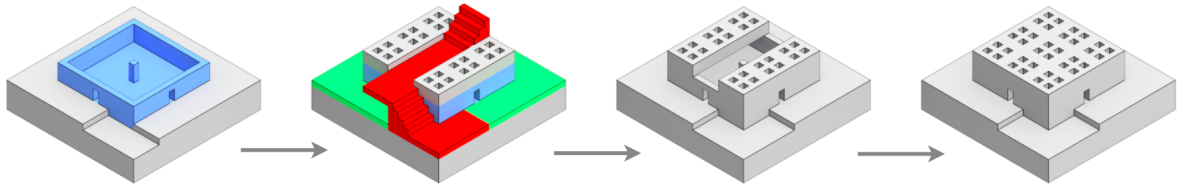


Figure 4.11: These four stages highlight the removal of unnecessary shapes. At the first stage the primary shape is shown in blue and if it is a secondary shape the color is grey. In the next stage additional shapes are added to the model, roofing and a path that goes on the building. But this (red) path, consisting of secondary shapes, does not go anywhere. Unlike the secondary (green) path, that does connect two entrances of a primary shape. The third stage shows the model after **RemoveUnnecessaryShapes** has been run, the (red) path that went on the building is gone, because it had no destination. In the last stage, roof shapes are added to close the shape. Shapes, that are placed after removing unnecessary shapes may not be subject of any types part of a traversability rule, which roof shapes are not.

4.6. (+) Combining settlements and the terrain

We have discussed the terrain analyser and iterative AP solver, that output settlements that are to be placed on the terrain. The remaining task is to combine settlements, based on a discretized, cell representation with the original terrain. We use the following two steps. The terrain is changed where settlements are placed. And the building materials for the settlements are decided by their local availability in the area they are placed. A result is shown in figure 4.12.

4.6.1. Terrain adjustments

We raise the terrain when we build something on top of it. We raise the terrain with the ground material we found most often at that position. When settlements are placed in the terrain, terrain is removed where it obstructs the settlements (by removing the overlap of the terrain with the settlement containing cells). Otherwise we leave the terrain unaltered.

4.6.2. Building Materials

A special tile-set is used, for which each voxel refers to, a class of building material. Voxels may refer to, foundation, building construction, window, door, stair step or some special items, like lighting fixture. These voxels are then replaced with the materials, represented as minecraft block types, that fit the location. Meaning that in an area where stone is mostly found, the building material is stone and will be used for creating the buildings. If the ground is mostly mud, the foundation on which the buildings are placed will be made of mud.



Figure 4.12: A settlement generated with our technique. The terrain is raised where the settlement is placed to support its construction. Dirt is used, because the ground is composed of dirt at that location.

4.7. Implementation

We implemented the iterative AP solver in python 3. We limit state propagation of WFC to the default stack limit in python, keeping it smaller than 1000. To obtain Minecraft maps, and show the results, we use a Minecraft python framework called MCEdit¹, specifically, the modified version for the GDMC competition². MCEdit allows users to run their own *filter*, an external program that may read and modify the terrain. We use a filter that calls our program as a subprocess.

4.8. Results

We will now show results obtained from our technique. First we will show a variety of profiles on a flat terrain. Then we will show results obtained from running our technique on GDMC maps.

4.8.1. Flat terrain

We have used two different profiles, and created a model of each, of size model size $50 \times 50 \times 5$, with the iterative AP solver. In the models we have replaced all roof tiles with street tiles, to make the visuals easier to parse. First the model of the obtained using the One-story houses profile is shown in figure 4.13. Buildings are relatively small and allowed to stack on top of each other. The model seems to reflect the profile while, showing high variety in building placement (low repetitiveness) and a steady amount of houses throughout the model. Next, a profile model with Larger two-story houses is shown in figure 4.14. We can observe that only a part of the input space is filled with shapes in the latter. This is a direct consequence of the greedy tile placement method to develop shapes. Larger building shapes fail to develop often, as larger shapes have a larger chance of failing due to the exploratory nature of shape development. In their place, smaller shapes, such as street shapes are placed, because their succeed rate is higher. These street shapes are later removed in the RemoveUnnecessaryShapes process, because many of them are unnecessary.

¹<https://www.mcedit.net/>

²<https://github.com/mcgreentn/GDMC>

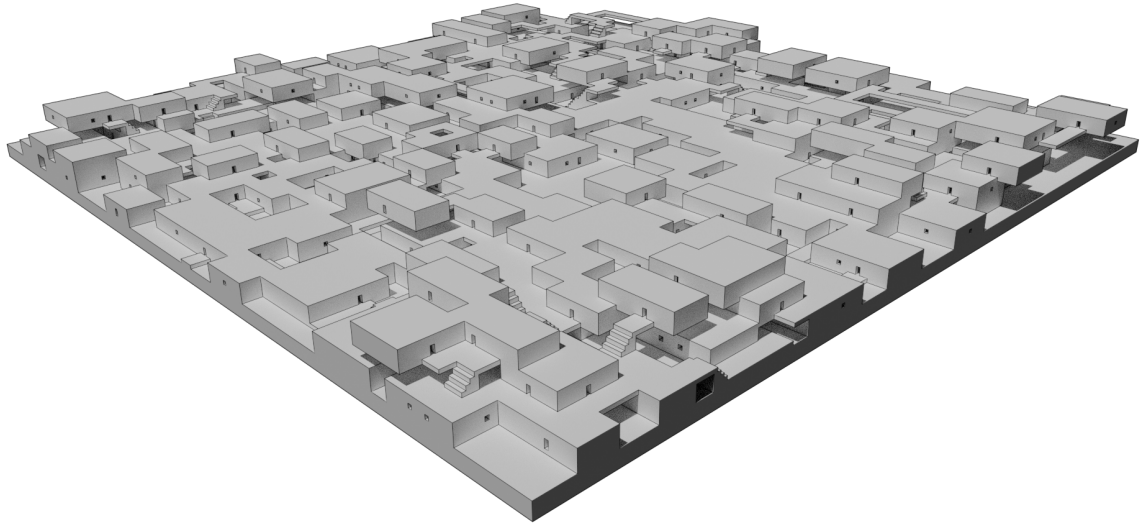


Figure 4.13: [One-story][stacked]

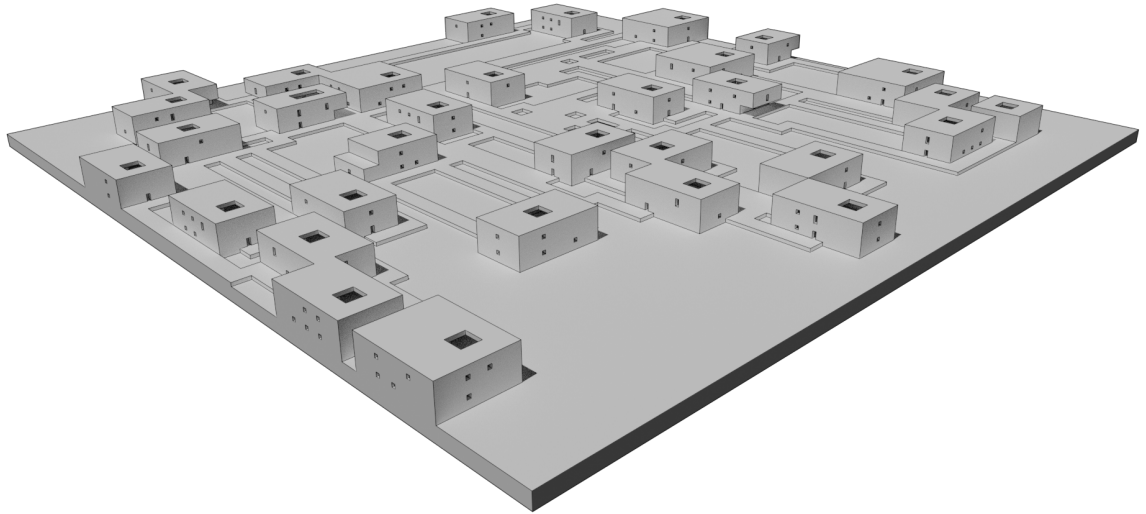


Figure 4.14: [Two-story]

4.9. Results on GDMC terrains 2020

GDMC shares the terrains used in previous competitions. These will be used to show how our generator performs on the task of settlement generation³. A GDMC map is roughly of size $256 \times 256 \times 256$ voxels. With our cell size of $5 \times 5 \times 4$ this roughly translates in $50 \times 50 \times 50$ cells as input for our iterative AP solver. We will await expert opinion of our technique output by entering the GDMC competition and only show the results here on the provided minecraft maps.

³We have made a submission for GDMC 2020 at an earlier stage of the implementation of our technique. The implementation described here is an improvement of our submission, solving bugs and adding an implementation for Removing unnecessary shapes.

4.9.1. Map 1

Terrain with woods, sand terrain and a number of mountains. Figures 4.15, 4.16, 4.17 and 4.18 show results. Settlements are placed both in the valley and on the mountain.



Figure 4.15



Figure 4.16



Figure 4.17



Figure 4.18

4.9.2. Map 2

Terrain with two small islands. Figure 4.19 shows a result, a single solitary house, which seems to be made out of glass.

4.9.3. Map 3

Snowy mountainous terrain with a valley by the lake. Figure 4.20, 4.21, 4.22 and 4.23 show results. In the valley near the water there is a connected settlement of larger buildings. On the mountain there is a settlement of smaller buildings, and some big buildings spread around the space.



Figure 4.19

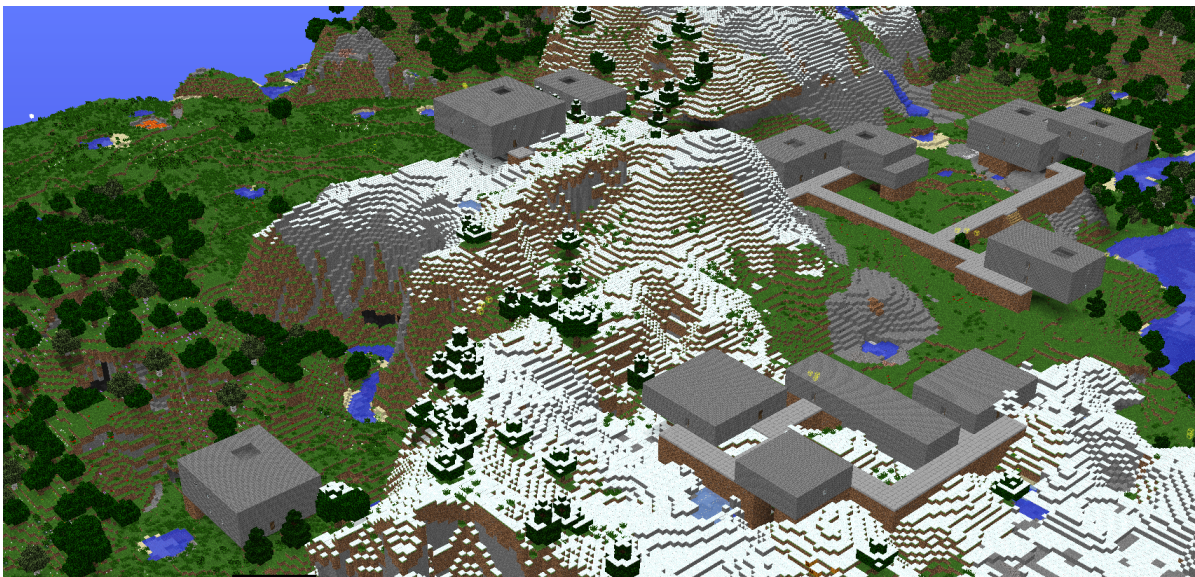


Figure 4.20



Figure 4.21



Figure 4.22



Figure 4.23

5

Conclusion

Table 5.1: Differences of the two solver solutions presented in this work to generate models of architectural profiles (AP).

	Exact AP Solver (chapter 3)	Greedy Iterative AP Solver (chapter 4)
Purpose	Expressive Range Analysis	Settlement Generation
Solver Backend	ASP	WFC
Time Complexity	NP	P
Size of Input	Small	Large
Builds on existing terrain	No	Yes

We present a novel tile-based PCG method for generating architecture. The method is centered on the notion of *architectural profiles*, a semantic specification that declaratively characterizes architectural typology. By combining a set of tiles, meaningful adjacency conditions among them, and a variety of validity constraints, architectural profiles offer a powerful vocabulary for steering a 3D tile solver towards the generation of many creative architectural structures. We have discussed two approaches that solve architectural profiles, summarized in table 5.1. From our expressive range analysis, we have shown comprehensiveness. This method is tune-able to generate a large variety of architectural structures over significant ranges of density and repetitiveness. We have also shown architectural profiles are adaptable to pre-existing terrains. We have explored its application on settlement generation in Minecraft[26] and have obtained promising results. We believe this method has a large potential for generating urban environments within a game context.

5.1. Future Work

Interesting future work includes (i) automating the creation of an architectural profile by using machine learning on example data, (ii) recursively defining architectural shapes within profiles, further expanding the expressive range of the method, and (iii) the development of a designer GUI for allowing the interactive configuration of architectural profiles. Regarding its application to settlement generation, (iv) we envision a declarative specification to fine-tune the generation of settlements, including architectural profile assignments to help steer towards narrative-rich settlements. And lastly, (v) improving the reliability and speed of the iterative AP solver by using machine learning to aid in making more informed decisions at each solving step.

Bibliography

- [1] Levi van Aanholt and Rafael Bidarra. Declarative procedural generation of architecture with semantic architectural profiles. In *Proceedings of CoG 2020 - IEEE Conference on Games*. IEEE, aug 2020. URL <http://graphics.tudelft.nl/Publications-new/2020/AB20>.
- [2] Martin Bokeloh, Michael Wand, and Hans-Peter Seidel. A connection between partial symmetry and inverse procedural modeling. *ACM Trans. Graph.*, 29(4), July 2010. ISSN 0730-0301. doi: 10.1145/1778765.1778841. URL <https://doi.org/10.1145/1778765.1778841>.
- [3] Stanislas Chaillou. Archigan: a generative stack for apartment building design, 2019. URL <https://devblogs.nvidia.com/archigan-generative-stack-apartment-building-design/>.
- [4] Michael F Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 287–294, New York, NY, USA, 2003. ACM. ISBN 1-58113-709-5. doi: 10.1145/1201775.882265. URL <http://doi.acm.org/10.1145/1201775.882265>.
- [5] Arnaud Emilien, Adrien Bernhardt, Adrien Peytavie, Marie-Paule Cani, and Eric Galin. Procedural generation of villages on arbitrary terrains. *Vis. Comput.*, 28(6–8):809–818, June 2012. ISSN 0178-2789. doi: 10.1007/s00371-012-0699-7. URL <https://doi.org/10.1007/s00371-012-0699-7>.
- [6] Pedro Filipe Coutinho Cabral D'Oliveira Quaresma. A detail shape grammar. using alberti's column system rules to evaluate the longitudinal elevation of the nave of sant'andrea church generation. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 32:1–13, 04 2018. doi: 10.1017/S0890060417000646.
- [7] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The Potsdam answer set solving collection. *AI Commun.*, 24(2):107–124, April 2011. ISSN 0921-7126.
- [8] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot ASP solving with clingo. *CoRR*, abs/1705.09811, 2017. URL <http://arxiv.org/abs/1705.09811>.
- [9] Kevin R. Glass, Chantelle Morkel, and Shaun D. Bangay. Duplicating road patterns in south african informal settlements using procedural techniques. In *Proceedings of the 4th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*, AFRIGRAPH '06, page 161–169, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595932887. doi: 10.1145/1108590.1108616. URL <https://doi.org/10.1145/1108590.1108616>.
- [10] Maxin Gumin. WaveFunctionCollapse, 2016. URL <https://github.com/mxgmn/WaveFunctionCollapse/>.
- [11] H. Hua. A bi-directional procedural model for architectural design. *Computer Graphics Forum*, 36(8):219–231, 2017. doi: 10.1111/cgf.13074. URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13074>.
- [12] 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. ISBN 978-1-4503-5319-9. doi: 10.1145/3102071.3110566. URL <http://doi.acm.org/10.1145/3102071.3110566>.
- [13] 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. ISBN 9781450372176. doi: 10.1145/3337722.3341845. URL <https://doi.org/10.1145/3337722.3341845>.

- [14] D. Kitsakis, E. Tsiliakou, T. Labropoulos, and E. Dimopoulou. Procedural 3d Modelling for Traditional Settlements. The Case Study of Central Zagori. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 42W3:369–376, February 2017. doi: 10.5194/isprs-archives-XLII-2-W3-369-2017.
- [15] Stefan Lienhard, Cheryl Lau, Pascal Müller, Peter Wonka, and Mark Pauly. Design transformations for rule-based procedural modeling. *Comput. Graph. Forum*, 36(2):39–48, May 2017. ISSN 0167-7055. doi: 10.1111/cgf.13105. URL <https://doi.org/10.1111/cgf.13105>.
- [16] Aidong Lu, David S. Ebert, Wei Qiao, Martin Kraus, and Benjamin Mora. Volume illustration using Wang cubes. *ACM Trans. Graph.*, 26(2), June 2007. ISSN 0730-0301. doi: 10.1145/1243980.1243985. URL <http://doi.acm.org/10.1145/1243980.1243985>.
- [17] Albert Mas, Ignacio Martin, and Gustavo Patow. Simulating the evolution of ancient fortified cities. *Computer Graphics Forum*, 39(1):650–671, 2020. doi: 10.1111/cgf.13897. URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13897>.
- [18] Paul Merrell. Example-based model synthesis. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, page 105–112, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936288. doi: 10.1145/1230100.1230119. URL <https://doi.org/10.1145/1230100.1230119>.
- [19] Paul Merrell, Eric Schkufza, and Vladlen Koltun. Computer-generated residential building layouts. *ACM Trans. Graph.*, 29(6):181:1–181:12, December 2010. ISSN 0730-0301. doi: 10.1145/1882261.1866203. URL <http://doi.acm.org/10.1145/1882261.1866203>.
- [20] Paul C Merrell. *Model synthesis*. PhD thesis, University of North Carolina at Chapel Hill, 2009.
- [21] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Trans. Graph.*, 25(3):614–623, July 2006. ISSN 0730-0301. doi: 10.1145/1141911.1141931. URL <http://doi.acm.org/10.1145/1141911.1141931>.
- [22] Gen Nishida, Ignacio Garcia-Dorado, Daniel G. Aliaga, Bedrich Benes, and Adrien Bousseau. Interactive sketching of urban procedural models. *ACM Trans. Graph.*, 35(4):130:1–130:11, July 2016. ISSN 0730-0301. doi: 10.1145/2897824.2925951. URL <http://doi.acm.org/10.1145/2897824.2925951>.
- [23] Gen Nishida, Adrien Bousseau, and Daniel G. Aliaga. Procedural modeling of a building from a single image. *Computer Graphics Forum*, 37:415–429, 2018.
- [24] Markus Persson. *Minecraft*, 2011.
- [25] Helmut Pottmann, Michael Eigensatz, Amir Vaxman, and Johannes Wallner. Architectural geometry. *Computers Graphics*, 47:145 – 164, 2015. ISSN 0097-8493. doi: <https://doi.org/10.1016/j.cag.2014.11.002>. URL <http://www.sciencedirect.com/science/article/pii/S009784931400140X>.
- [26] Christoph Salge, Michael Cerny Green, Rodrigo Canaan, Filip Skwarski, Rafael Fritsch, Adrian Brightmoore, Shaofang Ye, Changxing Cao, and Julian Togelius. The AI settlement generation challenge in minecraft. *KI - Künstliche Intelligenz*, 34, 2020. URL <https://doi.org/10.1007/s13218-020-00635-0>.
- [27] Arunpreet Sandhu, Zeyuan Chen, and Joshua McCoy. Enhancing Wave Function Collapse with design-level constraints. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, FDG '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372176. doi: 10.1145/3337722.3337752. URL <https://doi.org/10.1145/3337722.3337752>.
- [28] Pedro Silva, Pascal Mueller, Rafael Bidarra, and Antonio Coelho. Node-based shape grammar representation and editing. In *Proceedings of PCG 2013 - Workshop on Procedural Content Generation for Games, co-located with the Eighth International Conference on the Foundations of Digital Games*, Chania, Crete, Greece, may 2013. URL <http://graphics.tudelft.nl/Publications-new/2013/SMBC13a>.

- [29] Pedro Silva, Elmar Eiseemann, Rafael Bidarra, and Antonio Coelho. Procedural content graphs for urban modeling. *International Journal of Computer Games Technology*, 2015 (808904), jun 2015. URL <http://graphics.tudelft.nl/Publications-new/2015/SEBC15>. <http://www.hindawi.com/journals/ijcgt/2015/808904/>.
- [30] Ruben M. Smelik, Tim Tutenel, Rafael Bidarra, and Bedřich Beneš. A survey on procedural modeling for virtual worlds. *Computer Graphics Forum*, 33(6):31–50, 2014. doi: 10.1111/cgf.12276.
- [31] Gillian Smith and Jim Whitehead. Analyzing the expressive range of a level generator. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*. ACM, 2010.
- [32] Nizam Onur Sönmez. A review of the use of examples for automating architectural design tasks. *Computer-Aided Design*, 96:13 – 30, 2018. ISSN 0010-4485. doi: <https://doi.org/10.1016/j.cad.2017.10.005>. URL <http://www.sciencedirect.com/science/article/pii/S0010448517301781>.
- [33] S. Snodgrass and S. Ontañón. Learning to generate video game maps using markov models. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(4):410–422, Dec 2017. ISSN 1943-0698. doi: 10.1109/TCIAIG.2016.2623560.
- [34] Asiah Song and Jim Whitehead. Townsim: agent-based city evolution for naturalistic road network generation. *Proceedings of the 14th International Conference on the Foundations of Digital Games*, 2019.
- [35] Jerry O. Talton, Yu Lou, Steve Lesser, Jared Duke, Radomír Měch, and Vladlen Koltun. Metropolis procedural modeling. *ACM Trans. Graph.*, 30(2), April 2011. ISSN 0730-0301. doi: 10.1145/1944846.1944851. URL <https://doi.org/10.1145/1944846.1944851>.
- [36] Edward Teng and Rafael Bidarra. A semantic approach to patch-based procedural generation of urban road networks. In *Proceedings of PCG 2017 - Workshop on Procedural Content Generation for Games, co-located with the Twelfth International Conference on the Foundations of Digital Games*, 2017. URL <http://graphics.tudelft.nl/Publications-new/2017/TB17>.
- [37] T. Tutenel, R. M. Smelik, R. Lopes, K. J. de Kraker, and R. Bidarra. Generating consistent buildings: A semantic approach for integrating procedural techniques. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):274–288, Sep. 2011. ISSN 1943-068X. doi: 10.1109/TCIAIG.2011.2162842.
- [38] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik, and Klaas Jan de Kraker. The role of semantics in games and simulations. *ACM Computers in Entertainment*, 6:1–35, 2008. doi: <http://doi.acm.org/10.1145/1461999.1462009>.
- [39] Elise van Dooren, Els Boshuizen, Jeroen van Merriënboer, Thijs Asselbergs, and Machiel van Dorst. Making explicit in design education: generic elements in the design process. *International Journal of Technology and Design Education*, 24(1):53–71, Feb 2014. ISSN 1573-1804. doi: 10.1007/s10798-013-9246-8. URL <https://doi.org/10.1007/s10798-013-9246-8>.
- [40] H. Wang. Proving theorems by pattern recognition — ii. *The Bell System Technical Journal*, 40(1):1–41, Jan 1961. ISSN 0005-8580. doi: 10.1002/j.1538-7305.1961.tb03975.x.
- [41] Basil Weber, Pascal Müller, Peter Wonka, and Markus Gross. Interactive geometric simulation of 4d cities. *Computer Graphics Forum*, 28(2):481–492, 2009. doi: 10.1111/j.1467-8659.2009.01387.x. URL <https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01387.x>.
- [42] B. Williams and C. J. Headleand. A time-line approach for the generation of simulated settlements. In *2017 International Conference on Cyberworlds (CW)*, pages 134–141, 2017.
- [43] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Trans. Graph.*, 22(3):669–677, July 2003. ISSN 0730-0301. doi: 10.1145/882262.882324. URL <http://doi.acm.org/10.1145/882262.882324>.
- [44] X. Wu, C. Li, M. Wand, K. Hildebrandt, S. Jansen, and H. Seidel. 3d model retargeting using offset statistics. In *2014 2nd International Conference on 3D Vision*, volume 1, pages 353–360, Dec 2014. doi: 10.1109/3DV.2014.74.

-
- [45] Yi-Ting Yeh, Katherine Breeden, Lingfeng Yang, Matthew Fisher, and Pat Hanrahan. Synthesis of tiled patterns using factor graphs. *ACM Trans. Graph.*, 32(1), February 2013. ISSN 0730-0301. doi: 10.1145/2421636.2421639. URL <https://doi.org/10.1145/2421636.2421639>.