Proceedings of the Nineteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2023)

Observer Rules for Box-Split Grammars

Nicholas Baron, Markus Eger
Cal Poly Pomona
njbaron@cpp.edu, meger@cpp.edu

Abstract

Grammars are well-suited for the generation of structured content, such as text. Some specialized grammars, such as Shape Grammars, can even be used to generate 3D structures inside a game world like Minecraft. However, the top-down nature of grammars present limitations when it comes to modeling structures that should be connected to or utilize given geometry. In this paper, we describe an extension to an existing grammar model, called Box-Split Grammars, that extends it with the ability to observe existing geometry during the generation process, in order to incorporate it propertly into the generated structures. This modification also requires the addition of back-tracking in order to handle states in which certain geometry was (not) observed. We demonstrate the utility of this extension by showing how it can be used to place support structures for bridges and tunnels in a way that fits within an existing landscape.

Introduction

Procedural Content Generation (PCG) is widely used in the games industry, and extensively studied in academia. The idea that a computer program can produce variations of structures is enticing, as it allows greater flexibility when developing games. There are numerous approaches that are being used, from logic programming to define constraints on desirable output (Nelson and Smith 2016), to Neural Networks that generate images or text by mimicry (Creswell et al. 2018). Just as numerous as the approaches are the potential applications for PCG. More commonly used artifacts that are generated include plants and trees (Lindenmayer, Prusinkiewicz et al. 1990), levels (Van Der Linden, Lopes, and Bidarra 2013), or character- and item-names, sometimes even including their backstory (Adams 2019). However, there has also been interest in generating other elements of games, including 3D models, or even entire games (Cook 2022). Our interest lies with the generation of structures in the world of Minecraft. This domain is of particular interest, as a competition to procedurally generate entire settlements has been running for several years now (Salge et al. 2018), and we strive to develop approaches that may be useful within this competition and beyond. The goal of the settlement generation competition is to generate an entire settlement in a given Minecraft world, which is then evaluated by human judges. One of the evaluation criteria is how well the generated settlement fits within the given world, requiring incorporating existing features of the geography in the generation process.

Copyright © 2023, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Even within this narrow space, many approaches have been proposed and used, each with its own benefits and drawbacks. As some of the judges noted previously, many competitors rely on placing pre-defined templated buildings (Salge et al. 2022), and use a balance of ad-hoc coding and intelligent behavior to lay out the settlement. Some approaches are based on agents, where the simulated lives of agents define the settlement-layout (Iramanesh and Kreminski 2021), or Answer Set Programming (van Aanholt and Bidarra 2020) by defining constraints and using an external solver to produce the actual layout. However, different components within a settlement each have their own requirements, and we therefore believe that developing principled approaches that adequately address the challenges of each individual component is essential. In other words, while, for example, a cellular automata-based approach may be well-suited to generate “natural-looking” structures (Johnson, Yannakakis, and Togelius 2010), including parks, it would not be as useful to generate structured architecture. Our focus is on the latter. Architecture is often marked by a self-similarity and rigidity that makes it an ideal candidate to be described by a rule-system. In particular, Shape Grammars provide such a rule system and have, in fact, been used to describe architecture in a variety of domains (Hohmann et al. 2010). In a previous article, we have presented a Shape Grammar framework based on the notion of boxes with a split-operation to refine a larger structure into smaller components(Eger 2022). However, while such an approach has the benefit of simplicity and efficiency, there are certain forms it can not express. For example, our Box-Split Grammars are ideal to describe how to place e.g., a castle in a fixed and empty location, and would allow many variations of this castle to be generated, they assume that the target location is empty, and all geometry is placed by the grammar system itself. In the context of a larger generator, this assumption will not always hold, and for certain types of structures, such as bridges, it would actually be essential to be able to detect existing geometry and/or the lack thereof. This inherent limitation of grammars has been noted previ-

ously (Krecklau and Kobbelt 2011), but we present a novel approach that addresses it directly within the grammar formalism itself.

In this paper, we present an extension to our existing grammar framework that incorporates observations of existing Minecraft blocks, and allows rules to change their behavior based on this information. Our contribution is twofold: First, we describe how our grammar formalism can be extended to observe its surroundings by expanding our existing constraint-formalism. However, our existing constraint-implementation simply failed when there was no rule whose constraints were satisfied, which was an adequate response for the previous applications of rule constraints. The true versatility of observer constraints, though, comes from having observer rules deep in the grammar structure and being able to choose different branches. The second part of our contribution is therefore the extension of our existing grammar system with backtracking in case of failed rules. By adding backtracking, we take our grammar from a purely forward deriving system towards a more computational approach. We demonstrate the relevance of our extensions using bridges as a guiding example, and show tunnels as another direct application of the presented technique.

Background and Related Work

While there are many approaches to Procedural Content Generation, as mentioned above, our approach is based on Shape Grammars, which were first described by Stiny and Gips (Stiny and Gips 1971). Their approach, which itself is based on formal grammars as described by Chomsky (Chomsky 1959), details Shape Grammars that consist of rules, where each rule maps a shape on its left side to a more complex shape on its right side. To use such a system in a generative manner, one would start with an initial (simple) shape, and apply rules by finding sub-shapes that correspond to the left side of some rule, and replace that sub-shape with the right side of the rule. The actual implementation of such a system therefore requires the ability to find arbitrary sub-shapes within a larger structure. More recent advances in the field have done away with the need for such geometric queries by explicitly labeling the shapes or 3D-structures that are produced, similar to how Graph Grammars operate (Ehrig, Pfender, and Schneider 1973). Wonka et al. describe *Split Grammars* where rules are applied to labeled "scopes", which are split into smaller sub-scopes, to which further rules can then be applied (Wonka et al. 2003). Müller et al. have then improved upon this system by establishing a formal definition of the "split"-operation (Müller et al. 2006). The resulting system is widely used for everything from city planning (Halatsch, Kunze, and Schmitt 2008) to archaeology (Piccoli 2013). However, the focus of this system is to allow designers (or archaeologists) to precisely define new or existing structures, rather than providing a way to generate a wide variety of generated structures.

Nevertheless, a related approach by Hohmann et al. (Hohmann et al. 2010) has been used for generative purposes. Most importantly, further extensions of this work have established how grammar rules can be represented as functions in the Generative Modeling Language (Havemann 2005), and have also coined the term (Thaller et al. 2013) *Box Grammars*. Our own approach, detailed in a previous publication, builds on these ideas and combines their relative advantages into *Box-Split Grammars*, which represent grammar rules as simple Python functions.

Box-Split Grammars

As mentioned above, our approach is based on our work on Box-Split Grammars, which are a recursive subdivision scheme on axis-aligned boxes. In our system, grammar rules are represented as python functions that operate on a so-called *scope*, which contains a box in a given environment, such as Minecraft, and a label. Rules can split a scope into smaller sub-scopes and label them, which will lead to further rules being applied to them, or fill them with geometry (including "air"). A split grammar rule such as

$$A \rightarrow X\ Y\ X\ [\text{split}(x, [1, 2, 1])]$$

could be written in python as

```
@rule
def A():
    with split(Direction.X, [1,2,1]):
        X(), Y(), X()
```

This rule represents taking a scope which was assigned the non-terminal symbol `A`, splitting it along the x-axis into three scopes, of sizes 1, 2 and 1, and assigning them the labels X, Y, and X in this order. The decorator `@rule` plays an important role, as multiple rules could exist for the same non-terminal symbol. The decorator registers each possible rule for a non-terminal symbol, and upon applying a rule, selects one of the applicable rules at random. Our system allows controlling the relative probabilities of rules with parameters to the `@rule`-decorator, but more importantly, it also allows the restriction of when a rule is applicable. In prior work, we have used this to restrict rules to only apply to matching geometry. For example, to ensure that a box that is being split by our rule has size 4 to begin with (i.e., the sum of the sizes of the resulting sub-scopes), we can add a *constraint* to the decorator:

```
@rule(constraint=Direction.X == 4)
def A():
    with split(Direction.X, [1,2,1]):
        X(), Y(), X()
```

When a rule is to be applied to a scope, the constraint of each possible option are checked, and the random selection will be performed only between the option whose constraints are satisfied. Note that this may lead to "dead-ends" in rule application, if there is no applicable rule for a given non-terminal symbol. Our previous system regards such situations as errors on behalf of the grammar-author and exits with an error message. For our present work, we expand upon this constraint-mechanism to be able to conditionally apply rules in situations depending on which prior geometry is already present in the world. However, this may lead to cases where an *observation* fails, and in such cases it can be beneficial to back-track to earlier rule choices, to potentially find a derivation that does not lead to such an error state. In the next section, we will detail how we added these two capabilities.

Approach

Procedural content generation done by a decompositional system like a grammar has a tendency to lose information that would make the later steps more accurate. In our case, grammar evaluation starts with selecting a box in a world, which is simply labeled with the start symbol, losing all information about what else may already have been inside parts of that box. While this may be useful when building a house, as we certainly wouldn't want the floor to follow hilly terrain, and instead rather "carve out" a foundation for the building, it is undesirable in other instances, where geometry would follow or utilize existing terrain.

Our approach to reintroducing said information into our grammar-based generation system, is to allow grammar rules to *observe* the world, beyond the grammar system's own labeled scopes. Such an observation is expressed as a constraint on the applicability of a grammar rule, allowing us to express what must be present in the world for the rule to be applicable. A side effect of this addition is the possibility of rule failure. Importantly, a failure of one rule that does not necessarily fail the whole grammar application, but may simply imply that a different observation (or non-observation) may be applicable. We address this by adding backtracking to possible alternate expansions. In this section, we will describe the observer constraints and backtracking mechanism in more detail.

Observer Constraints

As described above, our rules can have constraints placed on them that control when they are applicable. Our new contribution adds a special type of constraint called *Observer constraints*, that take existing geometry (e.g., produced by the world generator) into account when determining the applicability of a rule. Observer constraints are placed on a particular option for a grammar rule and can distinguish between which material(s) are present in the box associated with the scope the rule is to be applied to. The simplest distinction that can be made is whether the box is empty or full of a particular, given block type. However, as a convenience to the end user, we also provide an additional nonempty distinction, to determine if there is at least one block of a particular type contained within a given scope/box. Observer constraints integrate with the existing constraint system and can be freely mixed and matched, for example, allowing constraining over the dimensions of a selected box as well as its content. Since a scope in Box-Split grammars will always contain its descendants, preventing grammar "self-intersections," observer constraints currently only work on the preexisting world data to avoid the need to consider the evaluation order of the grammar. This new feature introduces the need to recover from application failure in grammar evaluation, as a different (observer or other) constraint could potentially be satisfied in a different evaluation branch.

In our grammar system, the `@rule` annotation optionally accepts a `constraint` parameter, which accepts constraint expressions, which were previously only expressions to involving the dimensions of the containing

```
@rule(constraint=
    Observation.FULL(BlockType.SOLID))
def column_vertical_fill():
    # either the current box
    #   is already solid
    skip()


@rule(constraint=Constraint.ELSE)
def column_vertical_fill():
    # or we need to go
    #    down another level
    with split(Dimension.Y, [-1, 1]):
        column_vertical_fill()
        fill()
```

Figure 1: Python code to generate a column until solid blocks are found

scope. Using the same parameter, we now provide a new `Observation` class of constraints. We support three types of `Observation` constraints, covering a range of use cases: `FULL`, `NONEMPTY`, and `EMPTY`. Each observation constraint takes a `BlockType` parameter to filter which blocks should be considered by the observation. While our main application is Minecraft, the `BlockType` constitutes an abstraction that permits portability of the system and thus the grammar to other application domains. It also allows grouping of blocks into larger "types". We have mapped, for example, all blocks that are "solid" in Minecraft to a `SOLID` block type, to avoid having to specify all individual possible block types.

Consider the rules in figure 1 which generate a support column, e.g. for a bridge, down to ground level. There are two cases when this rule is applied to some scope. The first case is that the `Observation.FULL(BlockType.SOLID)` is fulfilled, that is, the box associated with the scope contains only solid blocks and is thus ground. The `skip()` tells the evaluator to do nothing with the scope, effectively removing the scope from consideration. The second case happens if the aforementioned `Observation` cannot be fulfilled (i.e. there is at least one non-solid block in the box). The `split()` breaks the box into two sub-scopes, a 1-height box atop the other box, which contains the remainder of the box (a size of -1 indicates a size corresponding to the remainder of the box). The lower box is associated with the `column_vertical_fill` non-terminal symbol, causing the rule to applied recursively, while the upper one is filled with the default material. The result of these two rules, when applied to a location where a column ought to be placed, is that the box containing the column is filled one block at a time, from the top down, until the remaining box is completely filled with solid blocks, i.e. the ground has been found.

Note that this process may fail if *no* solid blocks are found at all, as the `ELSE`-case will eventually attempt to split an empty box. Rather than requiring the user to specify all such constraints, we also added the capability to backtrack to a

higher level rule if rule application fails, as we will describe now.

Backtracking

Since individual rules can fail even though the whole grammar could still be successfully evaluated if a different branch was taken, the grammar evaluator needs to backtrack on any failure to take an alternate expansion. We handle this by maintaining a stack of rule applications, together with their associated rule applications. If a failure occurs, we pop items from this stack until a rule application with an alternative is found, which is then applied. In this case, all information regarding the failed expansion is forgotten and expansion continues as usual. If no alternate is found ere the root grammar rule is reached, the whole evaluation is considered to have failed, as the environment selected cannot satisfy the grammar's constraints.

Each invocation of a rule maps to an entry on the evaluation stack. Each entry on this stack carries a set of possible alternates along with the box that the rule is being applied to. Each possible alternate consists of a Python function and any optional parameters passed into the `@rule` annotation. When executing a rule, each sub-scope it creates is pushed onto the stack, with child rules always selecting the current top for expansion. When a rule is successfully applied, the corresponding scope is removed, which should be the top scope. This makes successive rules apply to different boxes, while also allowing the implementation to store any additional data it may require (e.g., the set of possible alternates).

When a rule is called, all of its known alternates are checked against the scope, with their constraints (if present) being evaluated. If the constraint was not provided, or it is satisfied, that alternate is added to the set of possible ones for the top scope. Once all alternates have been checked in this manner, there are three different cases that could happen:

- If the set of applicable rule options is empty, meaning that no option can be applied to this box, the rule fails to apply.
- If the set contains exactly one item, that item is selected, and the corresponding rule option applied.
- If there are multiple items, one is selected by random choice (with probabilities controllable by an optional parameter to the `@rule`-decorator). The selected alternate is removed from the set before it is applied to the top box.

If the same scope is encountered again during backtracking, the set of alternates of the previous instance is used as the initial set, instead of rerunning the constraint evaluation again. Since an arbitrarily deep child of this box may fail to apply, the aforementioned set rules are run again if a failed expansion is reported, without the rule option that had just been applied (since that alternative was removed from the list). This leads to backtracking behavior in the expansion of the grammar rules: If a child fails, a rule will explore all possible alternatives, before, itself, reporting a failure to its parent rule.

Figure 2 shows how this process may play out for the rules shown in figure 1. The case in which the rule is applicable is labeled as the "happy path", whereas the case where no solid block is present is labeled as the "sad path". The upper, "happy" path shows how conventional function calling in Python is used to evaluate each box, with stack frames corresponding to the evaluation stack. In particular, the happy path shown is one where the vertical fill terminates due to finding some ground underneath it. This results in one-by-one stack popping, as each `@rule(ELSE)` needs to execute its second part. Similarly, the "sad" path mimics the error handling procedure of Python's `try-catch` blocks. Here, when no ground can be found to start the column fill due to the final unit box being not full, all boxes in that column are popped off. This is due to constraint reevaluation not finding a possible alternate, due to both failing (`FULL` due to its precondition and `ELSE` being the one currently failing).

For another example of this backtracking algorithm, consider building a garden fence that respects the underlying geography. The grammar could split the length of the fence into 1-block wide columns, traversing each downwards. At the first evaluation, a column could fail its terminal rule and need to use its recursive one. This "local" failure would occur repeatedly until either the terminal rule succeeds or the evaluator tries to create a box with 0 height. In the first case, this success would propagate up the column, finishing the evaluation of the recursive case like a chain of dominoes. However, in the latter case, since a box with 0 in any dimension is considered invalid as it cannot contain any blocks, the whole column fails, leading to a "global" failure due to the fence grammar not permitting any breaks.

Results

Observer rules extend the existing grammar formalism significantly, by allowing the incorporation of existing geometry, which is highly relevant in a number of applications. Real-world architecture often incorporates the existing landscape to place support structures, or even determine the placement of buildings, such as a castle on top of a hill or in another location that is easy to defend. Out of the myriad of structures that could benefit from observation rules, we focused on bridges due to their simplicity and direct need to conform to surroundings.

The placement of support structures is the most direct application of our observer rules. The supports of a bridge must reach to the ground and, depending on designer preference, stop somewhere at or below ground level. It is uncommon, especially in procedurally generated worlds like Minecraft, to find perfectly flat terrain, let alone have it conveniently placed at the desired bridge location. Our grammar extension addresses this by allowing us to build the supports recursively, terminating at the first full box of solid blocks. We can also gracefully handle cases where a bridge cannot be built, due to missing environmental constraints, or even the lack of need for a bridge.

Bridge Rules

For our demonstration, we consider the case where we want to cross a given span with a bridge. It is wasteful, or at least unnecessary, to delve supports beyond what the player can see. Thus, in the rules to generate the bridge, a support terminates early if there is a solid block underneath it.

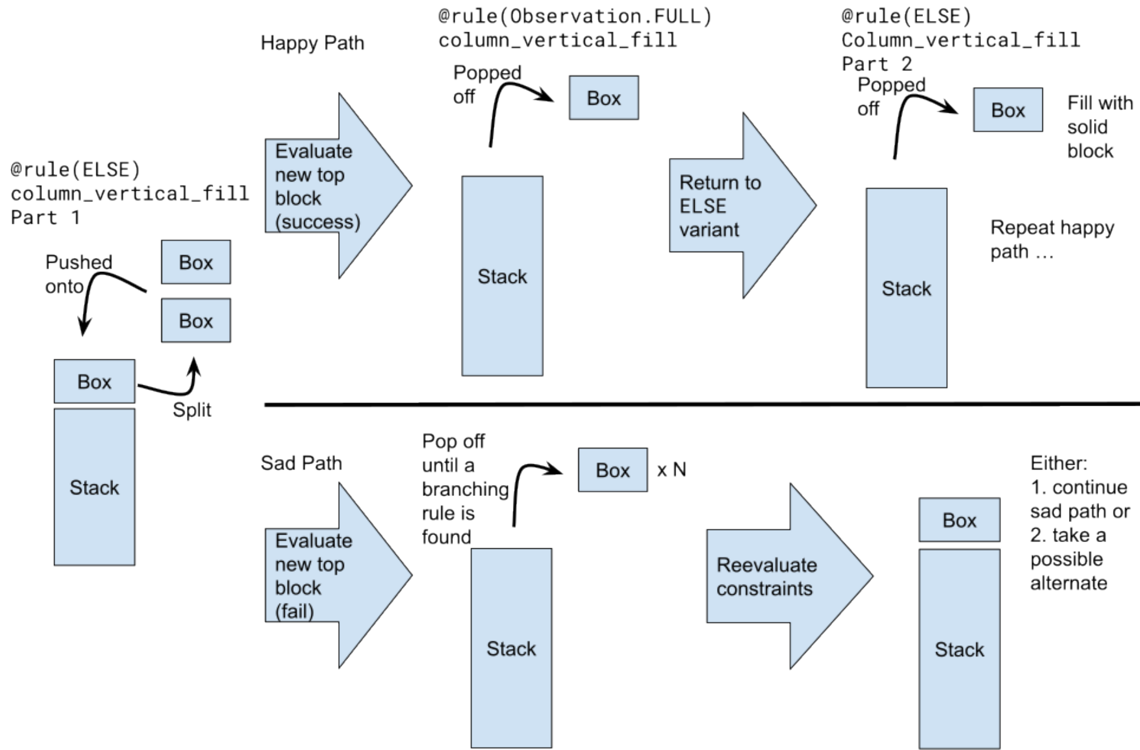

Figure 2: Grammar Evaluation Handling a Possible Failure

Bridge Scenarios

Figures 3 shows what was possible with our previous system without observer rules, while 4 contrasts this with output using our newly introduced observer rules. The non-observing grammar in figure 3 simply drills through the obstacle to place support structures, as it can not determine what already exists in the landscape, while the observing version in figure 4 allows the pillar to not destroy the model hill. The bridge in figure 4 uses the rules defined in figure 1 to generate its supports. Instead of simply generating the support from one monolithic block, figure 1 breaks the support into a chain of sub-boxes from top to bottom, with the lowest one either equal in height or the largest of the whole chain. The boxes are evaluated top down until the next box is already filled with solid blocks. If the next box is not filled with solid blocks, the large bottom box is split into a 1-height box at the top and the remainder becomes the new bottom.

This ability to generate bridge supports along a given landscape is directly applicable to scenarios encountered in actual Minecraft-worlds. Figure 5 uses the observing grammar to generate a bridge with varying column heights over a valley within a real Minecraft world. The screenshot was taken in-game, showing that this system can fit in an already established end to end game or level design system.

Alternatively, observer rules can also serve as a simple binary test for the suitability of column-placement. In figure 6 a foot path across a pond can be seen. The pond, as generated by the game, contains several islets/land areas inside it, upon which a column can be placed. When generating the footpath, the grammar attempts to place columns evenly along the length. The rule placing a column contains an observation constraint, though, and only actually places the column if it would rest on solid ground.

Tunnel Scenarios

Another, somewhat related, structure that can be generated in a geometry-dependent way using our approach is a tunnel. The grammar used for figures 7 and 8 adds sidewalls and a roof only as far as needed on each end. This allows the tunnel entrances to fit more naturally into their surroundings without requiring manual editing after generation.

To gain the precision required for the tunnels, both the wall and roof volumes are split into strips perpendicular to the longest axis. These strips are then tested for having any solid blocks in them, with such strips being filled. Strips that do not contain solid blocks, would be located outside the mountain the tunnel is dug through, and therefore nothing needs to be placed in them.

Conclusion and Future Work

In this paper, we have shown how our existing rule-based grammar system can be extended to include *observer rules*. Unlike our existing approach, such observer rules allow the grammar to take already existing geometry, such as blocks that are already filled in our Minecraft world, into account, and incorporate them in deciding how rules are being ap-

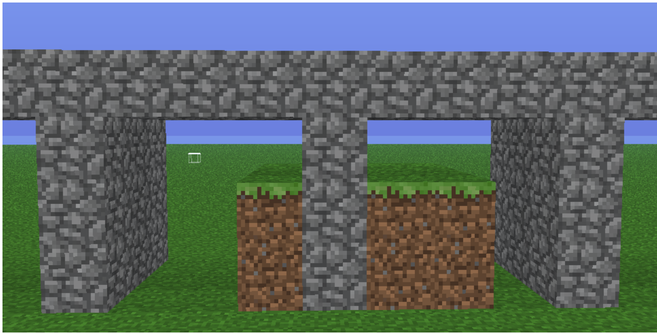

Figure 3: A bridge without observation rules crossing a hill

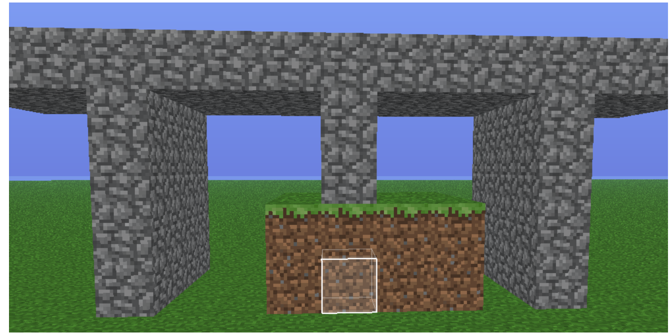

Figure 4: A bridge with observation rules crossing a hill

Figure 5: A bridge over a “naturally occurring” valley in-game. Without observer rules, the support columns would all terminate at the same height.

Figure 6: A footpath over a pond. Observer rules can be used to only place support columns when there is solid ground, and avoid placing them on water.

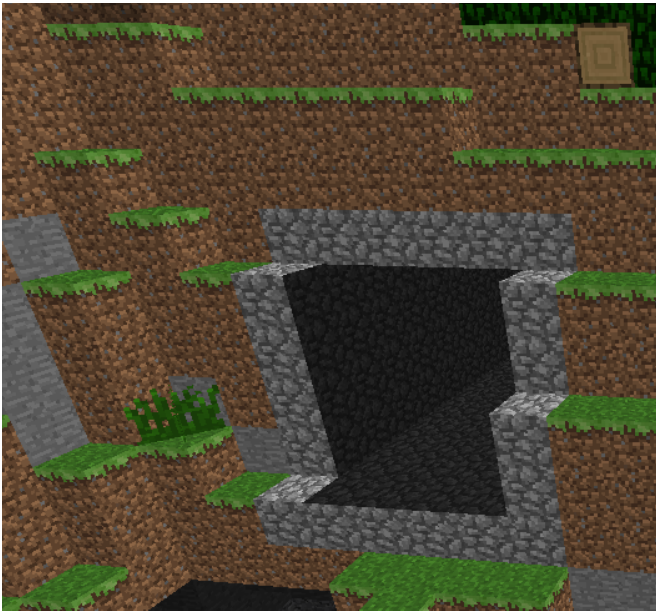

Figure 7: One end of a grammar-generated tunnel. Without observer rules, the tunnel walls would terminate in one straight edge, rather than following the outline of the mountain side.

Figure 8: The other end of the same tunnel. The ceiling is currently modeled as one slab, but could also be split into multiple pieces that terminate at different lengths.

plied to scopes. We have also shown that back-tracking is beneficial for such cases, in order to explore different rule options, some of which may fail in the presence (or absence) of existing blocks. There are a myriad of structures that benefit from being able to incorporate existing geometry, and this capability is particularly important for being able to use grammars to generate content within an existing world. As a standalone example that benefits from observation rules, we have shown how bridges, in particular, can be defined in a way such that their supports follow the existing relief of the existing landscape.

Our goal with this extension to our existing system is to make it applicable to the challenge of populating Minecraft worlds with larger-scale content, including entire settlements, or even groups of settlements. Our system is well suited for integration with other generators, but its strengths are the generation of artificial structures, which often utilize the existing landscape, just as our bridges do. In future work, we want to expand the scale of generation, and integrate it into such a larger system. As our generator only needs a box in which to evaluate the grammar, and can use the existing geometry to inform the evaluation, existing generators in the competition could utilize the grammar-formalism for the generation of buildings rather than relying on templated buildings. Observation rules can then be used to ensure the proper orientation and placement of doors, and otherwise connect the building with the settlement. We are also currently investigating the generation of road networks in order to actually connect our bridges with something.

While we use Minecraft as our main application domain, our system is actually agnostic to the concrete output. In our previous publication, we have, for example, demonstrated its applicability to generating tile patterns in PNG files. The extension presented in this article, likewise, is applicable to other domains, and we are still exploring what observation rules for other backends might be used for.

References

Adams, T. 2019. Emergent narrative in dwarf fortress. In *Procedural Storytelling in Game Design*, 149–158. AK Peters/CRC Press.

Chomsky, N. 1959. On certain formal properties of grammars. *Information and control*, 2(2): 137–167.

Cook, M. 2022. Puck: A Slow and Personal Automated Game Designer. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 18, 232–239.

Creswell, A.; White, T.; Dumoulin, V.; Arulkumaran, K.; Sengupta, B.; and Bharath, A. A. 2018. Generative adversarial networks: An overview. *IEEE signal processing magazine*, 35(1): 53–65.

Eger, M. 2022. Instant Architecture in Minecraft using Box-Split Grammars. *Proceedings of Foundations of Digital Games (FDG '22).*

Ehrig, H.; Pfender, M.; and Schneider, H. J. 1973. Graph-grammars: An algebraic approach. In *14th Annual symposium on switching and automata theory (swat 1973)*, 167–180. IEEE.

Halatsch, J.; Kunze, A.; and Schmitt, G. 2008. Using shape grammars for master planning. In *Design Computing and Cognition'08*, 655–673. Springer.

Havemann, S. 2005. *Generative mesh modeling*. Ph.D. thesis, Braunschweig University of Technology, Germany.

Hohmann, B.; Havemann, S.; Krispel, U.; and Fellner, D. 2010. A GML shape grammar for semantically enriched 3D building models. *Computers & Graphics*, 34(4): 322–334. Procedural Methods in Computer Graphics Illustrative Visualization.

Iramanesh, A.; and Kreminski, M. 2021. AgentCraft: An Agent-Based Minecraft Settlement Generator. In *Proceedings of the AIIDE Workshop on Experimental AI in Games*, 1–6.

Johnson, L.; Yannakakis, G. N.; and Togelius, J. 2010. Cellular automata for real-time generation of infinite cave levels. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, 1–4.

Krecklau, L.; and Kobbelt, L. 2011. Procedural modeling of interconnected structures. In *Computer Graphics Forum*, volume 30, 335–344. Wiley Online Library.

Lindenmayer, A.; Prusinkiewicz, P.; et al. 1990. *The algorithmic beauty of plants*, volume 1. New York: Springer-Verlag.

Müller, P.; Wonka, P.; Haegler, S.; Ulmer, A.; and Van Gool, L. 2006. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*, 614–623.

Nelson, M. J.; and Smith, A. M. 2016. ASP with applications to mazes and levels. In *Procedural Content Generation in Games*, 143–157. Springer.

Piccoli, C. 2013. CityEngine for Archaeology. In *Proceedings of the Mini Conference 3D GIS for Mapping the via Appia, Amsterdam, The Netherlands*, volume 19.

Salge, C.; Aranha, C.; Brightmoore, A.; Butler, S.; De Moura Canaan, R.; Cook, M.; Green, M.; Fischer, H.; Guckelsberger, C.; Hadley, J.; et al. 2022. Impressions of the GDMC AI Settlement Generation Challenge in Minecraft. In *Proceedings of the 17th International Conference on the Foundations of Digital Games*, 1–16.

Salge, C.; Green, M. C.; Canaan, R.; and Togelius, J. 2018. Generative Design in Minecraft (GDMC), Settlement Generation Competition. *CoRR*, abs/1803.09853.

Stiny, G.; and Gips, J. 1971. Shape grammars and the generative specification of painting and sculpture. In *IFIP congress (2)*, volume 2, 125–135.

Thaller, W.; Krispel, U.; Zmugg, R.; Havemann, S.; and Fellner, D. W. 2013. Shape grammars on convex polyhedra. *Computers & Graphics*, 37(6): 707–717.

van Aanholt, L.; and Bidarra, R. 2020. Declarative procedural generation of architecture with semantic architectural profiles. In *2020 IEEE Conference on Games (CoG)*, 351–358.

Van Der Linden, R.; Lopes, R.; and Bidarra, R. 2013. Procedural generation of dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(1): 78–89.

Wonka, P.; Wimmer, M.; Sillion, F.; and Ribarsky, W. 2003. Instant architecture. *ACM Transactions on Graphics (TOG)*, 22(3): 669–677.