Instant Architecture in Minecraft using Box-Split Grammars

Markus Eger
meger@cpp.edu
Cal Poly Pomona

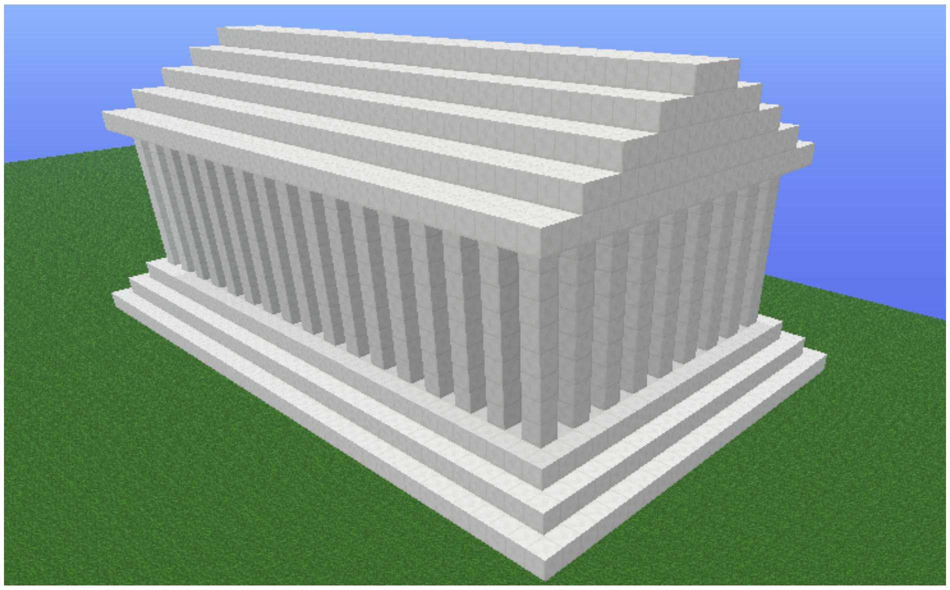


Figure 1: The Parthenon in Minecraft, generated using Box-Split Grammars

ABSTRACT

In this paper, we present a formalism we call Box-Split Grammars for the procedural modeling of structures in Minecraft and similar environments. Our grammars are based on previous work on split grammars and box grammars, where rules define how a given box, labeled with a non-terminal symbol, can be split into smaller pieces, and how subsequent rules are to be applied. We represent grammar rules as ordinary, well-structured python functions, allowing the integration into existing systems, and demonstrate their utility by recreating variations of ancient Greek temples using a few simple grammar rules.

CCS CONCEPTS

• **Applied computing** → *Computer-aided design*; • **Computing methodologies** → *Shape modeling*; • **Theory of computation** → *Grammars and context-free languages.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FDG '22, September 5–8, 2022, Athens, Greece
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9795-7/22/09…$15.00
https://doi.org/10.1145/3555858.3555865

KEYWORDS

procedural content generation, shape grammars, minecraft

ACM Reference Format:
Markus Eger. 2022. Instant Architecture in Minecraft using Box-Split Grammars. In *FDG '22: Proceedings of the 17th International Conference on the Foundations of Digital Games (FDG '22), September 5–8, 2022, Athens, Greece.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3555858.3555865

1 INTRODUCTION

Having computer programs that can produce a wide variety of different artifacts with minimal human input is one of the key tenets of Procedural Content Generation. The techniques that are employed to achieve this are as varied as the possible target output: Everything from planning to generate narratives, constraint solving to generate levels, Markov models to generate music, to Neural Networks to generate video renderings has been used. In this paper, we focus on the use of formal grammars to define structures within a Minecraft world. While grammars are often used to parse input, such as computer code, they have a rich history in a generative context as well, and have been used to generate everything from plant models [11] to text [3]. Our own main goal is to provide a convenient and expressive framework to generate 3D structures within the world of Minecraft.

The main contribution of this paper is twofold: First, we present a grammar model based on existing Box Grammars and Split Grammars, which we call Box-Split Grammars. This model provides a way to compactly define geometry using (context-free) grammars, while allowing the use of stochastic evaluation and rule constraints to generate a variety of geometry. Second, existing grammar models are typically built in a domain-specific framework, while our implementation is written, and fully integrates with, Python, even expressing grammar rules as Python functions. This makes it straightforward to integrate our grammar with other systems, and only delegating part of the content generation to the grammar formalism. To demonstrate the capabilities of our system, and its integration with another system, we present several examples of generated structures inside Minecraft, using MCEdit as the development platform/backend. Before we discuss our own grammar formalism in more detail, we will briefly discuss relevant related work.

2 BACKGROUND AND RELATED WORK

The use of grammars to non-text applications, in particular shapes, goes back to Stiny and Gips [14]. In their *Shape Grammars*, grammar rules consist of a mapping of a geometric substructure (a "shape") to other geometry. Starting with a basic shape, grammar evaluation proceeds by replacing substructures according to these rules. An actual implementation therefore has to perform geometric queries to determine which rules apply in each step, taking any necessary transformations such as rotation or scaling into account, making these grammars challenging to implement, as well as to define. More recent Shape Grammar developments have therefore eschewed such geometric rules in favor of labeled non-terminal symbols that replaced with multiple parts, which are, in turn, labeled as non-terminal symbols again, or replaced with a terminal symbol, i.e. actual geometry. This modern variation of Shape Grammars originated with work by Wonka et al. [17] which they called *Split Grammars*, as the principal operation is replacing a larger (geometric) scope into smaller pieces and applying subsequent rules to them. However, in order to be able to produce arbitrary 3D-structures, this first attempt used two separate grammars, one to define the geometry (which may use geometric queries) and another, which they call *Control Grammar* to propagate attributes, such as style information. Subsequent work by Müller et al. [9] addressed this by formally defining the "split" operation that lends its name to these grammars, and established the CGA Shape system. The generation of arbitrary 3D structures is handled by applying transformations to the scopes, as well as by allowing a variety of complex split- and geometric query-operations. CGA Shape has since been integrated into the ArcGIS CityEngine [8], and is widely used in many application areas, including Archeology [10], and city planning [5]. However, the focus of the system is on precise forward modeling of arbitrary 3D structures, while our system is targeted at providing more generative power, in a more restricted voxel-based environment.

An alternative approach to Shape Grammars, that simplifies the implementation and also makes grammar rules easier to define, operates only on axis-aligned boxes. This approach traces its roots to work by Hohmann et al. [7], that describes grammars that were inspired by CGA Shape, but rather than adding more and more complex shapes and rules, their approach distills them down to the simplest possible form: Each scope is an axis-aligned box labeled with a Non-Terminal Symbol, and a rule splits a box into smaller boxes, each again with its own label. Most interestingly, grammar rules are expressed as functions in the Generative Modeling Language [6], allowing them to interact nicely with the language's other facilities for procedural modeling. While having access to a wider variety of control structures can be helpful, by using function calls grammars become deterministic in their evaluation. While our approach also uses a similar approach of axis-aligned boxes and an encoding of grammar rules into an imperative, function-based style, we also provide stochastic evaluation and rule constraints, allowing users to define true options in their grammar rules. Finally, while not directly relevant to our work, Thaller et al. [15] expanded upon this same approach by allowing the use of arbitrary convex polyhedra. In the same article they also coined the term *Box Grammars* when discussing the basis of their work, leading us to name our approach *Box-Split Grammar* as a combination of the two lines of work discussed so far: We combine the non-deterministic, flexible rules used by Split Grammars with a procedural, programming-language based approach based on Box Grammars. While the individual pieces we use were present in different approaches, their combination is novel and particularly applicable to a voxel-based environment.

Finally, before we describe our approach in detail, we briefly want to discuss our chosen backend. While our grammar system is output-agnostic, Minecraft presents an attractive first target. Interest in procedural generation in Minecraft has given rise to the generative design in Minecraft [13] settlement generation competition, where a wide variety of approaches have shown promise, including Answer Set Programming [16] and cellular automata [4]. Participation in the competition has seen a steady increase every year [12], indicating sustained interest. Our grammar-based approach is aimed at providing a novel way to define procedural content, with an eye on the ease of integration with other approaches.

3 BOX-SPLIT GRAMMARS IN MINECRAFT

As described in the previous section, our approach can be seen as combining appealing features from *Split Grammar* with those of *Box Grammars*, resulting in what we call *Box-Split Grammars*. On a high level, our grammars take a box, assign it a label, and non-deterministically apply an applicable rule to the box, which may split it into smaller boxes, assigning labels to each.

3.1 Boxes

Much like the constituting components of a formal grammar are non-terminal and terminal *symbols*, our Box-Split Grammars represent them as boxes. Each such box is axis-aligned, and has integer size in each dimension, and therefore corresponds to the set of voxels contained in a rectangular hexahedron. In other words, each box is defined by an origin voxel (o_x, o_y, o_z) and three integer sizes $(\delta_x, \delta_y, \delta_z)$, and spans all voxels (x, y, z) with $(o_x \leq x < o_x + \delta_x) \wedge (o_y \leq y < o_y + \delta_y) \wedge (o_z \leq z < o_z + \delta_z)$. However, each of our boxes additionally stores a *local orientation*, which means the voxel coordinates (x, y, z) mentioned above may not refer to the global x-, y- and z-dimension in that order, but rather any permutation thereof. For example, our grammar may use a box with the global Minecraft origin $(0, 1, 2)$ and size $(3, 4, 5)$, but treat it as a logical box with origin $(2, 0, 1)$ and size $(5, 3, 4)$. This allows grammar rules to use consistent directional information, even if the resulting (sub)structure should be rotated within the world. The translation from local to global coordinates is handled automatically, and we will discuss how the user can control this reorientation below.

As with each grammar, our grammars also need a *start symbol*, which is a designated non-terminal symbol, and accordingly also a box that is attached to said non-terminal symbol. In the Minecraft editor, one way to obtain such a start symbol box is for the user to select it in the game world, and pass it to our grammar. With a start symbol in hand, we can now apply grammar rules.

3.2 Split Rules

While formal grammars for text-applications can simply define that a non-terminal ought to be replaced with a sequence of terminal- and non-terminal symbols, our Box-Split Grammars additionally need to specify what happens to the box. The box corresponding to the non-terminal symbol represents the space associated with that symbol, and in order to replace it, a grammar rule has to specify how the box is split into smaller boxes, which are then associated with the terminal- and non-terminal symbols at the right side of the rule. A (split) rule may be defined as:

$$N \rightarrow A\ B\ C\ [\text{split}(x, [1, 2, 1])]$$

The semantics of this rule are to take a box associated with the non-terminal N, and split it into three pieces along the local x-dimension, one with size 1, one with size 2 and another with size 1. The first of these boxes is then associated with the non-terminal A, the second with B and the third with C, each of which may in turn have their own rules for how to further split the resulting box (perhaps in a different dimension), or replace it with a terminal symbol (which will fill the entire box with a concrete block).

To make our grammars actually usable in practice, we represent each rule as a short python function with a predefined structure. The rule described above could be written in python as follows:

```
@rule
def N():
    with split(Dimension.X, [1,2,1]):
        A(), B(), C()
```

Each rule consists of a decorator that registers it with our system, a name, and the desired split operation. Then, a non-terminal symbol is assigned to each resulting box. Note that, as we will describe below, while this is written as an ordinary function call, the `@rule`-decorator actually intercepts the call, allowing each non-terminal symbol to be associated with multiple rules.

While we can already describe a variety of different structures using this basic split operator, it will result in an error if we attempt to split a box into pieces that do not completely partition it, i.e. that either leave parts of the box unassigned, or try to assign boxes of a larger size than we have, as would be the case if the box associated with N did not have size exactly equal to 4 in its x-Dimension. In order to give additional flexibility to grammar authors, and similar to Hohmann et al. [7], we also allow splitting into *relative* sizes, for example to cut a box into two pieces, where one is twice the size of the other. Relative sizes are indicated by negative numbers and can be freely mixed with positive/absolute ones. For example, `split(Dimension.X, [-1,2,-1])` would split a box into three pieces in the local x-dimension where the center piece is 2 units long, and the other two pieces equally divide the remainder of the box. A split may only contain relative sizes, in which case the entire scope is allocated proportionally, it may only contain relative sizes, which must add up to the total size, or a combination of the two. In this last case, the absolute sizes are allocated first, and the remainder is allocated proportionally.

However, since our boxes can only have integer-sizes, we may need to round the resulting box sizes. We support three rounding modes: `TRUNCATE` (the default), will ignore any remainder, `BEGINNING` will start assigning the remainder to (relatively sized) boxes at the beginning of the list, while `END` will assign the remainder starting from the end of the list. Figure 2 shows several grammar rules that split a given scope along its X-axis. Each split contains a combination of absolute and relative sizes, demonstrating how the scopes are distributed among these scopes. Particularly note that `split3` divides the block into three pieces of sizes -2, 2, and -3, with negative numbers indicating relative sizes as indicated above. For our given scope of width 10, first the piece with absolute size 2 will be allocated, and then the remaining 8 blocks will be distributed between the relatively sized scopes. As 8 is not evenly divisible by 5, this leaves 3 blocks unallocated. If our original scope had width 12, the same split would have allocated 4, 2, and 6 blocks to its three sub-scopes instead, covering the entire width.

Note that another potential rounding mode, which creates another box with any remainder is problematic, because it may or may not create an additional box, making it uncertain if an additional non-terminal symbol is needed to consume this additional scope. We provide an alternative way to address the case where the desired behavior is to separate the remainder using rule constraints, described below.

3.3 Coordinate Reorientation

As described above, each box keeps track of its own, local coordinate system, and splits are performed along these local dimensions. The purpose of these local coordinate systems is to facilitate rule-reuse. For example, a wall of a castle may require crenellations at the top, which can be defined as a split along the direction of the wall, alternating between placing a stone wall, and keeping an empty space. However, some walls will run along the (global) x-dimension, while others will run along the (global) z-dimension (y is the global "up"-direction), which would require separate rules. Similarly, if a grammar is to place an entire castle, we might want the "front" of the castle to be whichever side is wider, in order to be able to comfortably place a gate and guard towers. For either case, having local coordinate systems allows the grammar to be written independently of the rotation of the (sub)structure. We therefore provide an operator `reorient` in addition to `split`. For example, a grammar rule may exchange the x- and z-dimensions as follows:

```
@rule
def N():
    with reorient(x=Dimension.Z,
                  z=Dimension.X):
        M()
```

```
@rule
def split1():
    with split(Dimension.X, [1, 1, -1]):
        fill(), void(), fill(MARBLE)

@rule
def split2():
    with split(Dimension.X, [1, -1, 2]):
        fill(), void(), fill(MARBLE)

@rule
def split3():
    with split(Dimension.X, [-2, 2, -3]):
        fill(), void(), fill(MARBLE)

@rule
def split4():
    with split(Dimension.X, [-1, -1, -3]):
        fill(), void(), fill(MARBLE)
```

(a) Four different split rules with absolute and relative sizes.

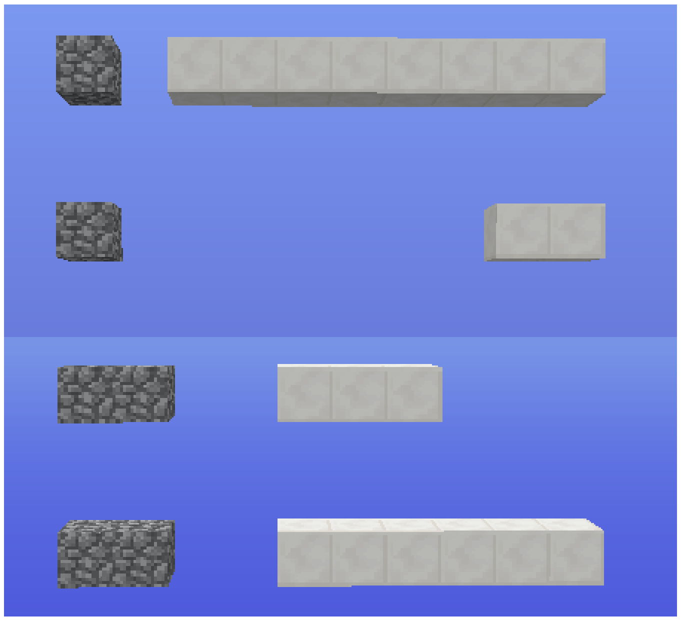

(b) The result of applying the split rules to 10 blocks in one row.

Figure 2: Correspondence between split rules and the resulting blocks.

The non-terminal M will then operate on a box that is the same as the box that corresponded to N, except that its x and z-dimension are flipped. `reorient` will automatically determine any missing dimension from its parameters: If two dimensions are given, the third (local) dimension is set to whichever global dimension remains unassigned. If only one dimension is given, it either switches two dimensions (i.e. it would have been enough to only pass `x=Dimension.Z` in the example above), or a local dimension would be reassigned to the global dimension it already represents, in which case no other dimensions are changed either. Some (sub)structures may require to be built in alignment with the global coordinate system, which a user can specify with `Dimension.WORLD_X`, `Dimension.WORLD_Y`, or `Dimension.WORLD_Z`. Finally, the special values `Dimension.SMALLEST` and `Dimension.LARGEST` can be used to align a local dimension with the dimension in which the box has its smallest or largest extent. Note that dimensions that are explicitly assigned take precedence over these special operators, so that e.g. `reorient(x=Dimension.LARGEST, y=Dimension.Y)` will never set the local x-dimension to the previous local y dimension.

A common pattern is to split a box and then reorient all of its pieces, we also allow to specify the reorientation directly as additional parameters on the `split`-function in the same way.

3.4 Iteration

One advantage of expressing grammar rules as code is that we can make use of control flow statements to guide the generation process. A particularly common scenario is the application of a repeating pattern of structures along an axis, such as columns lining a temple, windows along the front of a building, or crenelations on top of a castle. Our grammar system includes the capability to let a `split` repeat as often as it can fit along the given dimension by setting the `repeat` parameter to true. Such a split returns an object that can be iterated over with a `while` loop and which will continuously set the current scope. Figure 3 shows an example of how such a repeating split can be applied to scopes of different sizes.

3.5 Probabilistic Rules

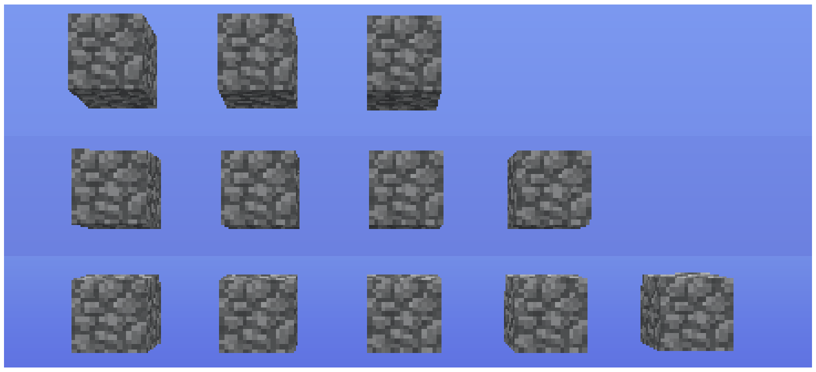
As previously mentioned, our encoding of grammar rules as python functions invokes the appearance that we are calling these functions deterministically, which would deter from one of the core ideas of our grammars, which is to generate a *variety* of structures. However, what the `@rule` decorator actually does is to extract the name of the function and register it in a dictionary, as one of the possible ways to resolve a non-terminal symbol. It then replaces the function with one that performs a lookup in this dictionary, choosing a rule at random from among all that are applicable (this also allows us to place constraints on when a rule is considered "applicable", see below). Grammar authors can control this probabilistic behavior by specifying (relative) probabilities for the different rules corresponding to the same non-terminal symbol:

```
@rule(probability=4)
def N():
    # rule content

@rule(probability=1)
def N():
    # other rule content
```

Whenever the non-terminal `N` is assigned to a box, by calling `N()`, one of these two rules will be chosen at random, where the

```
@rule
def iteration():
    pieces = split(Dimension.X, [1,1], repeat=True)
    while pieces:
        fill(), void()
```

(a) A split that iterates a fill/void pattern over the entire width of a scope.



(b) The result of applying the iteration to differently sized scopes.

Figure 3: The effect of applying an iteration.

first rule has a probability to be chosen that is four times higher then the probability of the second rule (i.e. 80% vs. 20%). If the `probability`-parameter is omitted, it defaults to 1, i.e. rules are *always* stochastic, with a default of using the same probability for each option. Note that probabilities are automatically normalized between all applicable options, which interacts nicely with rule constraints, which we will describe next.

3.6 Rule Constraints

The nature of our integer-sized boxes makes it such that not every rule can be applied to every arbitrary box and still produce a reasonable split. For example, to place crenellations by alternating stone blocks and empty space, the total length of the box must be odd, if both ends should begin with an empty space. Our grammar implementation allows authors to place *constraints* on rules that restrict when a rule is applicable to a box associated with the corresponding non-terminal symbol. These constraints are expressed as arithmetic comparisons involving the dimensions of the box. For example, limiting a rule's applicability to cases where the a box has even length in its local x-dimension, can written as follows:

```
@rule(constraint=(Dimension.X%2 == 0))
def N():
    # other rule content
```

When another rule refers to `N()` in its body, the constraints of each potential rule are evaluated against the actual box the rule should be applied to, and a random rule is chosen among all applicable rules. As mentioned above, this rule choice also takes any probabilities that may be present into account, normalizing them as needed. Constraints may make use of any of the `Dimension`-variables described above, including `Dimension.LARGEST` and `Dimension.SMALLEST`. The special value `Constraints.ELSE` can be used to create a constraint that matches if and only if the constraint of no other option for a particular rule does.

Rule constraints have several applications, one being the aforementioned enforcement of divisibility. Additionally, they allow grammar authors to specify more or less detail depending on the available space/resolution, making it able to scale structures arbitrarily and showing as much detail as possible at each scale. Structures also often permit different substructures depending on their overall size. A large castle may make use of more complex towers than a smaller one might be able to fit.

3.7 Minecraft Editor Integration

While we developed our grammar system with Minecraft as an application in mind, the actual implementation is agnostic to the game. Only when we actually need to place concrete blocks in the world, does the system need any knowledge of the target domain. For this, our system requires an adapter for each concrete output domain that performs the actual block placement. In our current implementation for Minecraft, this adapter takes an entire box, and fills every voxel contained in that box with a given material. The second function this adapter needs to provide is a way to construct the initial box from domain-specific information.

We want to note that the implementation we chose makes our grammar system very flexible in how it can be used. Instead of using a domain-specific file format to encode our grammar rules, they are written directly as python functions, which allows users to use the full power of the language to create richer structures or simplifying their grammars in a practical way. At the same time, grammars defined formally can still be expressed in a straightforward way. What makes the system particularly attractive, though, is that it can be integrated seamlessly with other systems. A level generator may create biomes and landscape, but then uses a grammar to define the outlines of cities by defining a box in which the city shall be contained. That same grammar may then use rules to divide the city into districts, blocks, and buildings, but calls another generator to model an organic structure such as a park inside an appropriate box, instead of using concrete (Minecraft) terminal symbols itself.

In order to more fully demonstrate the capabilities of our system, we will now discuss several concrete examples of grammar rules and generated structures.

4 RESULTS

What makes grammars attractive for the procedural modeling of buildings is that architecture itself is often modular to begin with, and can be expressed compactly as grammar rules. As a demonstration of our Box-Split Grammars we will show how a variety of ancient Greek temples can be modeled using the grammars, and how this allows generating variations of these temples. Greek temples generally consist of a central chamber (the *naos*) with surrounding columns, and different styles are named depending on the number of placement of these columns: The (Greek) number of columns across the front determines is used as a prefix, indicating whether a temple has two (*distyle*), four (*tetrastyle*), six (*hexastyle*) or eight (*octastyle*) columns across its front. Additionally, the placement of columns can be indicated with an additional prefix, which includes columns between the side walls (*in antis*), columns in front of the temple (*prostyle*), columns in front and behind the temple (*amphi-prostyle*), or columns surrounding the temple (one row: *peripteral*, two rows: *dipteral*), among others [2]. These definitions can be combined to describe a concrete temple layout. For example, the Temple of Athena Nike in Athens is an amphi-prostyle tetrastyle temple, consisting of an inner chamber with four columns in the front and four behind the temple. The Parthenon, on the other hand, is built in the peripetral octastyle, with eight columns along the front side, with columns all around its perimeter.

To demonstrate the capabilities of our Box-Split Grammars, we modeled different variations of temples. As a base-line, consider an amphi-prostyle tetrastyle temple: The floor plan of such a temple can be split into three parts across its depth: There is a front row of columns, the naos, and a back row of columns. Either of the two rows of columns consist of a split into 7 parts across the width of the temple: Four columns, with three gaps in between them. The naos itself can be split in the depth dimension, with (from back to front) a back wall, a center part, and a front part which defines the entrance of the temple. For our purposes, we will have a simple, empty naos, and the entrance will consist of two columns. Figure 4 shows how such a temple floor plan can be modeled as grammar rules, and the resulting structure. Of course, an actual temple consists of more than just the floor plan, but our grammar rules are agnostic to the height of the temple, and we can create a temple of any height by selecting an appropriately sized box. We can integrate this floor plan into a larger grammar that includes a base and a roof, and obtain a full temple, as shown in figure 5.

As mentioned above, a key appeal of grammars is their modularity and correspondence to the structure of the modeled buildings. For our temple, we have just stated that we would add "a roof", but there are multiple different roof options. Figure 6 shows a comparison of four different ways of topping our temple. What makes our grammars appealing for procedural content generation is that the rule for each of these options (or others) can each be called `roof` in code, which will result in a roof type being chosen at random for each generated temple, producing variety of output if desired.

On the other hand, as described above, Greek temples are named for the number of columns that are present across their front, so it would also be desirable to be able to produce a variety across this dimension easily. We will describe two approaches how this can be achieved with our grammars, and their relative trade-offs. First, if it is desired that the aforementioned styles with 2, 4, 6, or 8 columns (and not, say, a temple with 26 columns) should ever be generated, but we want to automatically determine which number of columns fits evenly into the selected box, we can use rule constraints. For example, to limit distyle temples to boxes which can be evenly split into three equal pieces, we can use the following rule constraint:

```
@rule(constraint=Dimension.X%3 == 0)
def columns():
    with split(Dimension.X, [-1,-1,-1]):
        fill(), void(), fill()
```

Tetrastyle temples would correspondingly use `Dimension.X%7 == 0`, and similarly for other column counts. The advantage of this approach is that it uses relative sizes, allowing structures to be scaled up arbitrarily (e.g. by making a temple twice as big, which will keep the same proportions). On the other hand, if we have a box of width e.g. 77, it would permit either a tetrastyle temple with column width 11, or a hexastyle temple with column width 7, and the grammar will correctly choose a random *applicable* layout. The drawback of this approach is that it requires the enumeration of all options and the definition of the correct constraints.

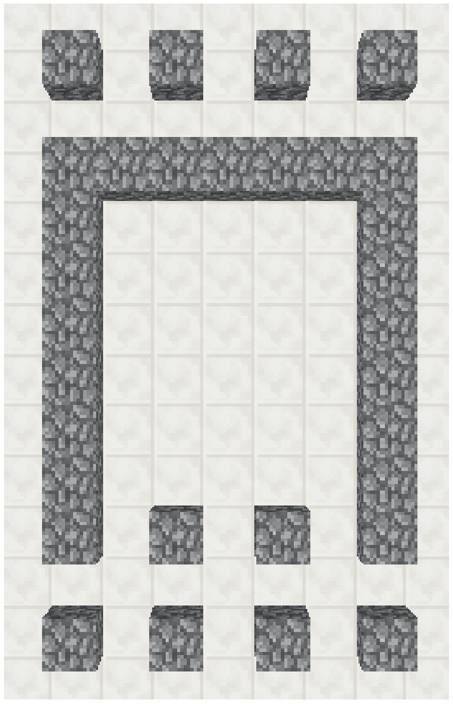
The second way to model arbitrary temple-widths is through the use of iterative rules. We could replace our different column-rules with the following single rule that first splits off the left delimiter column, and then alternatingly places an empty space and a column:

```
@rule(constraint=Dimension.X%2 == 1)
def columns():
    with split(Dimension.X, [1,-1]):
        fill()
        itercolumns()

@rule
def itercolumns():
    items = split(Dimension.X, [1,1],
                  repeat=True)
    while items:
        void(), fill()
```

The advantage of this approach is that this rule covers all cases, and even includes styles that may not actually have been used in ancient Greece, such as a temple with 3 columns. If this is undesireable, additional constraints may be placed on the rule. Note that the rule already has a constraint to ensure that the width of the box is odd, which is necessary to ensure placement of columns on

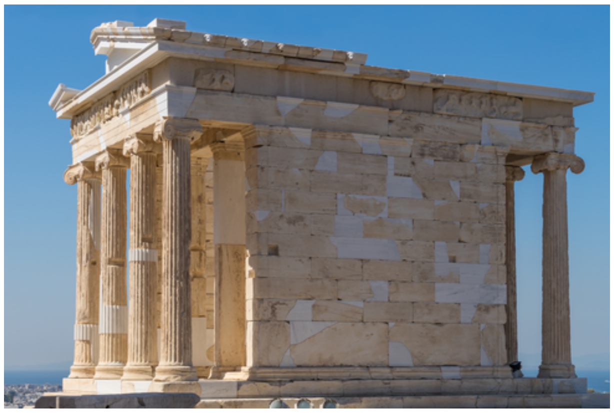
```
@rule
def columns():
    with split(Dimension.X,[1,1,1,1,1,1,1]):
        fill(), void(), fill(),void()
        fill(), void(), fill()

@rule
def chamber():
    with split(Dimension.X, [1, -1, 1]):
        fill(), void(), fill()

@rule
def naos():
    with split(Dimension.Z, [1, -1, 1]):
        columns(), chamber(), fill()

@rule
def floorplan():
    with split(Dimension.Z,
               [1, 1, -1, 1, 1]):
        columns(),void(),naos(),void(),columns()
```

(a) Grammar rules.

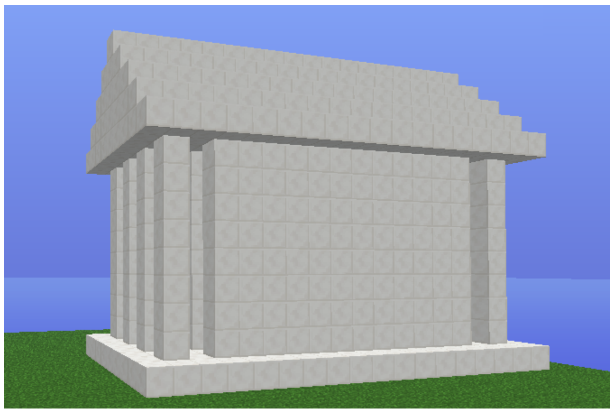


(b) The generated floor plan.

Figure 4: An amphi-prostyle tetrastyle temple floor plan expressed as a Box-Split Grammar



(a) The Temple of Athena Nike (picture is in the public domain).



(b) A generated version of the temple using Box-Split Grammars.

Figure 5: Side-by-side comparison of a real temple and a generated version.

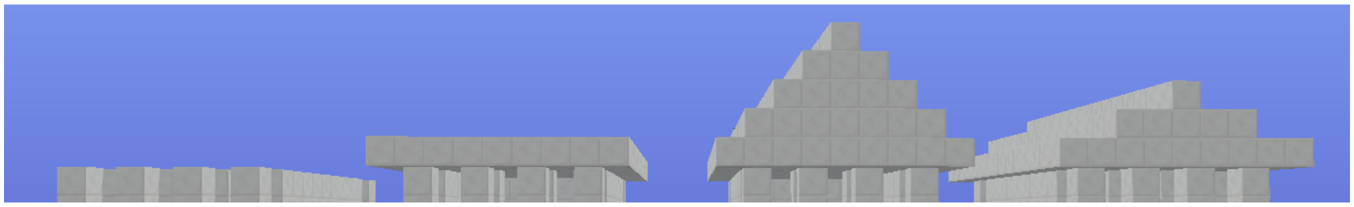


Figure 6: Different roof options for our temple: No roof, flat top, steep, moderate (from left to right).

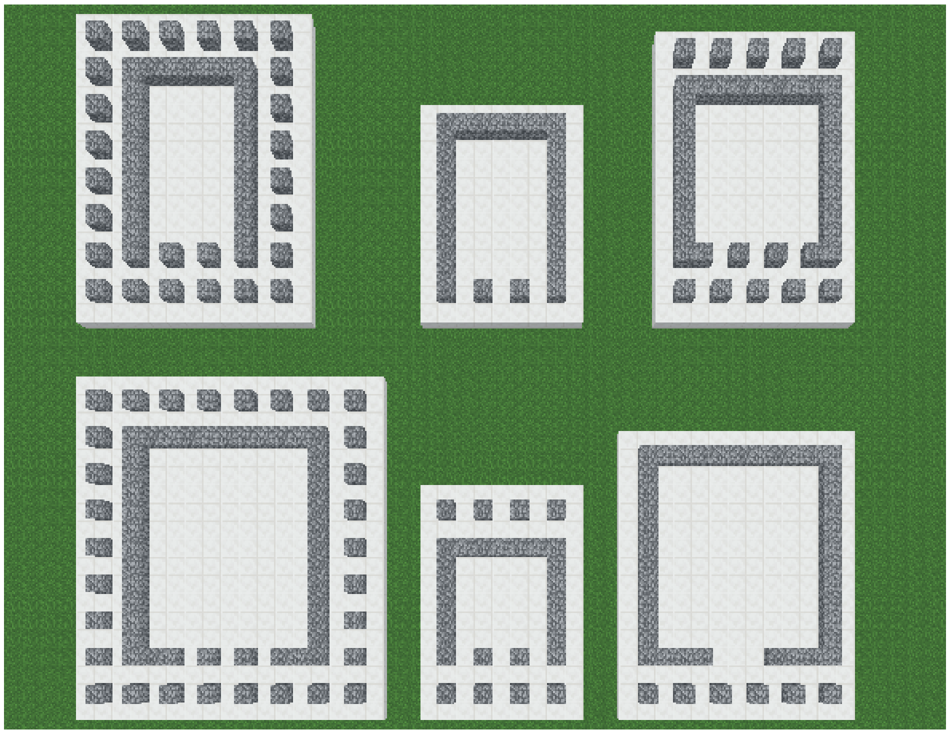


Figure 7: Some floor plans generated by a simple temple grammar, from left to right: peripteral hexastyle, distyle in antis, hexastyle amphi-prostyle (top row), peripteral octastyle, tetrastyle amphi-prostyle, hexastyle prostyle (bottom row). Note that the entrance of the naos also has multiple variations.

both ends. The drawback of this rule is that columns are always of width 1, which makes it harder to scale the temple.

Finally, addressing different column placement, such prostyle, amphi-prostyle and peripteral only require minor modifications: Columns are always present at the front of the temple, and we can have the naos either cover the entire rest of the scope, or split it to have columns behind or around it (note: columns at the side of the naos can be achieved with the same `columns` rules as above, by `reorienting` the scope). Once again, we place constraints on the rule to ensure that the naos has a reasonable width and depth, and define multiple rules with the same name. If we put all these modules together, we end up with a grammar of 14 rules that can generate a wide variety of temple floor plans, as show in figure 7

Note that all the user does is select an arbitrarily sized box, and the grammar randomly generates a temple that fits within that scope. To generate the replication of the Parthenon shown in figure 1 we limited the grammar to the rules corresponding to a peripteral octastyle to control what was being generated.

For this article, we have limited our demonstration to ancient Greek temples, but we believe this class of structures serves as a good demonstration of the broader capabilities of our system. Figure 8 shows a castle outline as another example we have been working on. We have also implemented a second backend, which uses pixels as the basic building block (with the z-axis representing occlusion) and writes its result to an image file. Figure 9 shows an example of a repeating spiral-pattern produced using this backend. Adapting the

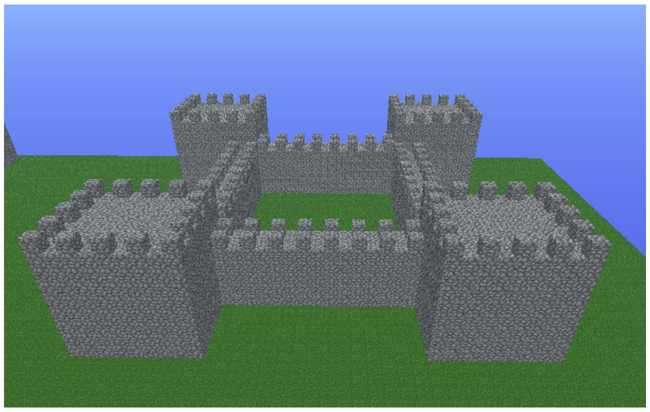


Figure 8: The outline of a castle generated by a grammar.

system for image output required only to provide means to create an initial bounding box and set pixels in it, totaling to about 35 lines of python, most of which were boilerplate function signatures and similar. As stated above, a key feature of our approach is that it integrates nicely within a larger ecosystem, making it possible to apply in its areas of strength while mitigating some of its limitations, which we will discuss in the next section.

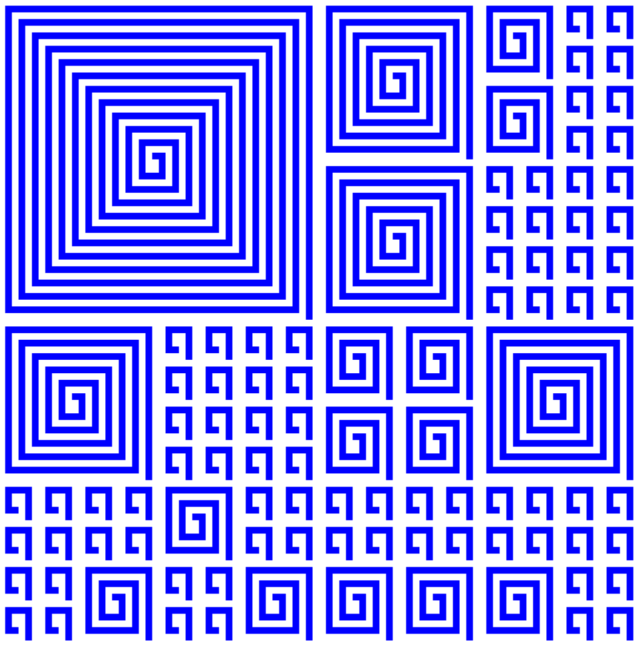


Figure 9: A spiral pattern produced with grammars using an Image-backend.

5 CONCLUSION AND FUTURE WORK

We have presented a formulation of shape grammars on axis-aligned boxes of integer-sizes that is suitable for the procedural modeling of structures in an environment such as Minecraft. Our grammars are expressed as python functions, and can use `split` and `reorient` operations to divide a given box into smaller boxes that are then filled with materials. Grammar rules are chosen non-deterministically with controllable probabilities, and can have constraints placed on them. Our system is designed to be used within a larger generator, allowing the use of any given box, or to use a box that is deemed a "terminal symbol" by the grammar with another generator. To demonstrate our system, we have modeled variations of Greek temples in the system, mapping different building styles to grammar rules which allow arbitrary recombination. Our goal was a demonstration of a variety of features of our grammar system as opposed to a fully faithful replication of all aspects of ancient Greek temples. Upon publication of this article, we will also make our system available on github. An aspect of our system that we have yet to study further stems from its representation of grammars as python functions. On one hand, each formal grammar rule can translated to its python equivalent in a straightforward manner, which would allow a sort of *meta-generation* of grammar rules using some other generator. On the other hand, the functions that make up our grammar "play nicely" with ordinary control flow constructs (the `repeat`-option of the `split` operation is one example of this), and we would like to explore what other possibilities this integration opens up in future work. However, as Alexander noted, "A city is not a tree" [1], indicating that a simple, context-free grammar model may not be sufficient to model all the intricacies of a modern (or ancient) city-design. In particular, cities are often designed in an interconnected way, rather than purely top-down. We may be able to determine which lots to place individual buildings in, but we then, subsequently, want to align doors or windows or floors across lots. Relatedly, we assume that we are starting with an empty box, but in many practical applications (and surely in Minecraft), there often is existing geometry that should be integrated into our building designs. Both of these challenges are related to providing *context* to rules, which – as the name implies – context-free grammars are not sufficiently powerful for. In future work, we want to explore how to integrate such existing blocks, whether they come from the level, or were produced by other rules, into the generation of structures. An example application would be the generation of roads within a town, which has to take into account elevation, and where bridges would be suitable.

REFERENCES

[1] Christopher Alexander. 2013. *A city is not a tree.* Routledge.
[2] Sir Banister Fletcher. 1961. *A History of Architecture on the Comparative Method.* [London]: University of London, Athione Press.
[3] Kate Compton, Ben Kybartas, and Michael Mateas. 2015. Tracery: an author-focused generative text tool. In *International Conference on Interactive Digital Storytelling*. Springer, 154–161.
[4] Michael Cerny Green, Christoph Salge, and Julian Togelius. 2019. Organic building generation in minecraft. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*. 1–7.
[5] Jan Halatsch, Antje Kunze, and Gerhard Schmitt. 2008. Using shape grammars for master planning. In *Design Computing and Cognition'08*. Springer, 655–673.
[6] Sven Havemann and Dieter W Fellner. 2005. Generative mesh modeling. (2005).
[7] Bernhard Hohmann, Sven Havemann, Ulrich Krispel, and Dieter Fellner. 2010. A GML shape grammar for semantically enriched 3D building models. *Computers & Graphics* 34, 4 (2010), 322–334. https://doi.org/10.1016/j.cag.2010.05.007 Procedural Methods in Computer Graphics Illustrative Visualization.
[8] Guangyin Jia and Kaiju Liao. 2017. 3D modeling based on CityEngine. In *AIP Conference Proceedings*, Vol. 1820. AIP Publishing LLC, 050001.
[9] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural modeling of buildings. In *ACM SIGGRAPH 2006 Papers*. 614–623.
[10] Chiara Piccoli. 2013. CityEngine for Archaeology. In *Proceedings of the Mini Conference 3D GIS for Mapping the via Appia, Amsterdam, The Netherlands*, Vol. 19.
[11] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. 1990. Graphical modeling using L-systems. In *The Algorithmic Beauty of Plants*. Springer, 1–50.
[12] Christoph Salge, Claus Aranha, Adrian Brightmoore, Sean Butler, Rodrigo Canaan, Michael Cook, Michael Cerny Green, Hagen Fischer, Christian Guckelsberger, Jupiter Hadley, et al. 2021. Impressions of the GDMC AI Settlement Generation Challenge in Minecraft. *arXiv preprint arXiv:2108.02955* (2021).
[13] Christoph Salge, Michael Cerny Green, Rodgrigo Canaan, and Julian Togelius. 2018. Generative design in minecraft (gdmc) settlement generation competition. In *Proceedings of the 13th International Conference on the Foundations of Digital Games*. 1–10.
[14] George Stiny and James Gips. 1971. Shape grammars and the generative specification of painting and sculpture.. In *IFIP congress (2)*, Vol. 2. 125–135.
[15] Wolfgang Thaller, Ulrich Krispel, René Zmugg, Sven Havemann, and Dieter W Fellner. 2013. Shape grammars on convex polyhedra. *Computers & Graphics* 37, 6 (2013), 707–717.
[16] Levi van Aanholt and Rafael Bidarra. 2020. Declarative procedural generation of architecture with semantic architectural profiles. In *2020 IEEE Conference on Games (CoG)*. 351–358. https://doi.org/10.1109/CoG47356.2020.9231561
[17] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. 2003. Instant architecture. *ACM Transactions on Graphics (TOG)* 22, 3 (2003), 669–677.