OBSERVER RULES FOR BOX-SPLIT GRAMMARS

A Thesis
Presented to the
Faculty of
California State Polytechnic University, Pomona

In Partial Fulfillment
Of the Requirements for the Degree
Masters in Science
In
Computer Science

By
Nicholas Baron
2022
SIGNATURE PAGE

THESIS: OBSERVER RULES FOR BOX-SPLIT GRAMMARS

AUTHOR: Nicholas Baron

DATE SUBMITTED: Fall 2022

Department of Computer Science

Markus Eger
Thesis Committee Chair
Professor of Computer Science

David Johannsen
Professor of Computer Science

Adam Summerville
Professor of Computer Science
ABSTRACT

Procedural generation of computer simulated worlds has had a long history of various approaches. Box-split grammars are one of these various approaches. However, these grammars are currently not sensitive of their surroundings, leaving the resulting structures out-of-place.

By introducing rules that can observe the environment around a particular instantiation of the box-split grammar, a grammar can modify its output to fit its surroundings more appropriately. This new type of rule can be integrated into previous work that used a constraint system. The aforementioned constraint system uses Python as both its language of implementation and usage, allowing for portability and ease of teaching. These results are transferable to other video games due to a platform-agnostic design. An example ruleset is used to test and demonstrate the abilities of the new system in contrast to previous work. Furthermore, an abstract semantic model of such a system is detailed to allow for comparisons in a formal context.

Additional research is possible in testing the system with other grammars, allowing more types of shapes as basic units, and connecting multiple grammars together.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGNATURE PAGE</td>
<td>ii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Background</td>
<td>2</td>
</tr>
<tr>
<td>Procedural Content Generation</td>
<td>2</td>
</tr>
<tr>
<td>Shape Grammars</td>
<td>4</td>
</tr>
<tr>
<td>Generative Design in Minecraft Content (GDMC)</td>
<td>5</td>
</tr>
<tr>
<td>Our Approach</td>
<td>6</td>
</tr>
<tr>
<td>Alternative Approaches</td>
<td>6</td>
</tr>
<tr>
<td>Formal Definition</td>
<td>8</td>
</tr>
<tr>
<td>Terminal Actions</td>
<td>9</td>
</tr>
<tr>
<td>Nonterminals Actions</td>
<td>10</td>
</tr>
<tr>
<td>Rule Constraints</td>
<td>13</td>
</tr>
<tr>
<td>Problem Description</td>
<td>14</td>
</tr>
<tr>
<td>Methodology</td>
<td>15</td>
</tr>
<tr>
<td>Implementation</td>
<td>16</td>
</tr>
<tr>
<td>Results</td>
<td>19</td>
</tr>
<tr>
<td>Generated Examples</td>
<td>19</td>
</tr>
<tr>
<td>Conclusion</td>
<td>26</td>
</tr>
<tr>
<td>Future Work</td>
<td>26</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Formal definition high-level outline</td>
<td>9</td>
</tr>
<tr>
<td>Figure 2</td>
<td>The four kinds of atomic grammar actions</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3</td>
<td>The three split types</td>
<td>11</td>
</tr>
<tr>
<td>Figure 4</td>
<td>An example rule for constructing bridge supports</td>
<td>17</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Diagram of the happy and sad paths of column_vertical_fill</td>
<td>17</td>
</tr>
<tr>
<td>Figure 6</td>
<td>A generated bridge with highlighted AABBs</td>
<td>20</td>
</tr>
<tr>
<td>Figure 7</td>
<td>An example of a bridge shortening a support</td>
<td>20</td>
</tr>
<tr>
<td>Figure 8</td>
<td>An example of a bridge lengthening a support into a ditch</td>
<td>21</td>
</tr>
<tr>
<td>Figure 9</td>
<td>An in-game screenshot of a generated bridge</td>
<td>22</td>
</tr>
<tr>
<td>Figure 10</td>
<td>An in-game screenshot of a bridge generated with split pillars</td>
<td>22</td>
</tr>
<tr>
<td>Figure 11</td>
<td>A closeup of some of the pillars</td>
<td>23</td>
</tr>
<tr>
<td>Figure 12</td>
<td>One end of a tunnel generated by Appendix B</td>
<td>23</td>
</tr>
<tr>
<td>Figure 13</td>
<td>An in-game screenshot of the other end of figure 12</td>
<td>24</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Comparison of a non-observing and observing grammar</td>
<td>25</td>
</tr>
</tbody>
</table>
Introduction

As the video game industry has pushed for both larger and more realistic worlds, procedural content generation has come to the forefront as a useful tool for developers to use in creating said worlds. Procedural content generation, or PCG, is any algorithmic assistance to game designers during world creation. Within this field, there are multiple methodologies for generation, each of which can be measured on a variety of axes. Two important axes are how much control the designer has and how expressive the system is, which are especially important if multiple PCG systems are in use. For example, a rule-based system offers a higher degree of control than a purely random system or a machine learning model, but that extra control can hinder some third axis (say believability) when the two systems interact.

Rule-based generation allows for more control over certain features of the world, which is especially useful for “artificial” structures. However, most rule-based systems are usually ran in a preexisting environment. This leads to unrealistic results if the rules do not consider the surrounding environment. For example, placing supports for a bridge should not only take into consideration the distance from other supports, but also should consider the height of the terrain under the bridge, as to not be “floating” unrealistically.
Background

Generating data that follows some overall structure with minimal human involvement, in particular three-dimensional geometries, has been a topic of research for some time. The first work in this field was the formalization of shape grammars in two-dimensional contexts by Stiny (1980). A shape grammar takes the concept of grammars for formal automata and applies it to two-dimensional and three-dimensional spaces, allowing shapes to be constructed in a similar way to string construction in formal automata.

Procedural Content Generation

A decently comprehensive overview of the current state of procedural content generation is the website/book *Procedural Content Generation in Games* by Shaker et al. (2015). Within it, a few methods attempt to address similar problems related to context sensitivity, but each has limitations that prevent it from solving the problem of context sensitive rule-based generation. For example, chapter 3 contrasts binary space partitioning and agent-based algorithms. The former divides a 3D space into at most 8 octants, resulting in an octree where every node has up to 8 children. This lends itself well to creating maze-like interiors, but leads to difficulties when generating context-sensitive buildings. A parent node would have to evaluate specific children first before allowing others to evaluate in order to “observe” the existing world. Additionally, children may have to know what their siblings did, as a bridge’s supports may cross multiple octants. Agent-based generation fairs no better, as 3D cellular automata seem to be infrequently explored and 2D examples look more like natural formations than built ones.
Chapter 5 discusses grammar-based systems and their applications towards plants, graphs, and levels. Plant growth can be modeled with L-systems, a simple type of context-free grammar that allows for parallel rewriting. While such a system may be useful for fractal structures, like plants, most human constructions are not fractal and thus need a different, more general system. Graph grammars operate by pattern matching subgraphs and replacing or transforming the matched area. The concept of pattern matching is useful if generalized for other sorts of inputs, which will be the case for a 3D environment.

Chapter 8 examines answer set programming, a particular approach to logic programming for map generation. The core idea is to list constraints which the world must satisfy and generate a set of possible answers, from which one is picked. While the mechanism of constraints is related to context sensitivity, two major differences make the implementation shown in the book unusable for our problem. Firstly, the constraints in answer set problem operate over a blank world, whereas context-sensitive grammars operate within an already existing world. The constraint system would have to allow for reading the preexisting state of tiles, complicating the grammar. Secondly, the constraint system generates a whole set of possible answers with only one being chosen. This is rather inefficient, as we only need one of those solutions in the end.

Chapter 9 lays out how the representation of game content leads to tradeoffs in generating it. For our particular problem, the issue of representation is solved with a “positive” representation, that is one where the default state is empty. Later, the chapter covers compositional pattern-producing networks, a type of artificial neural network that can be evolved to create multiple “unique” designs for everything from Mario levels to flowers. While this kind of evolution creates a very diverse set of results, like evolution, it is unguided,
working only towards a short term goal with no discernible plan. This is in stark contrast to civil architecture, which always has mostly long term goals and defined plans.

In summary, much of the preexisting work has useful aspects, but no single concept or system addresses all of the goals of this thesis. The most suitable, preexisting whole system is shape grammars, if they were to be combined with some sort of constraint system from answer set/logic programming. This would allow for a human designer to both design a structure and place restrictions on the locations that could host said structure.

**Shape Grammars**

While shape grammars could generate an entire world, that is a rather large problem to be tackled all by one system. The grammar would be large and produce a highly regular world, which is not always desirable. Often, the user of the grammar wants the result to fit within a particular space in an already existing world, making this the more common case. Since grammars are not required to use all the space provided for their output, the user is thus specifying a maximum size.

Later work by [Wonka et al. (2003)](#) applied this concept of a maximum size to generating architecture via split grammars. A split grammar takes some three-dimensional space, and either splits it into some shapes which are contained within it, or converts it into another shape that is contained within it. Note that in both types of rule, the result will always be inside the original split, meaning that the initial space defines the maximum extent that the generated structure can occupy. Additionally, the original paper used a control grammar and parameter matching to encode higher level goals. These two features allowed the use of a database of rules to minimize human intervention in the generation. However, this is a
rather complex and potentially overly flexible system. This complexity could be compressed into one language, allowing for a simpler yet equally expressive system.

Compressing the complexity would mean the use of a substantially different style of language for describing the grammar. Hohmann et al. (2010) chose to focus more of the description of a building into an imperative scripting language, as this would allow for more complex calculations alongside the actual grammar. They note that “GML as a stack-based language is just “by coincidence” well suited to express shape grammars that typically also use a stack for evaluation”, which suggests that any programming language with a concept of a stack (be it a call stack, an evaluation stack, or a stack data structure) is generally usable to implement such a system. This is an important result, as purely stack-based languages are usually domain specific and differ significantly from more general purpose ones.

**Generative Design in Minecraft Content (GDMC)**

Comparing Procedural Content Generation algorithms is a difficult task, as there are numerous constraints that must be agreed upon between teams before development begins. Additional development time must also be allotted to create the common testing environment before the different implementations can even start. A convenient solution to this problem is to use an already existing world system as a base, preferably one that has already proven successful as a sandbox. One such example is the rather popular video game Minecraft, which is a rather archetypical sandbox in that there are no clear goals or play style. Salge et al. (2018) considered this game “a suitable testbed for [a] competition” in which a panel of human judges evaluated submissions using various criteria. Additionally, the existing modding community around the game had not only a preexisting interest in such problems,
but also has maintained tools for editing both save data and executed code. One such tool that formed the bedrock of the aforementioned competition is MCEdit, a world and save data editor typically used by humans for large and complex edits. It has a plugin system based using the Python language to allow users to implement complex tasks as a concise script.

**Our Approach**

Many of these previous systems worked over arbitrary three-dimensional worlds, where boxes can have dimensions with rational or even real lengths. While this allows for highly precise three-dimensional modeling of real-world structures, it adds to the potentially unnecessary complexity to such a system. Additionally, the resulting evaluation systems are not easily ported between platforms, meaning that the rules and thus the grammars would need to be rewritten or at least adapted. Eger (2022) addresses both of these issues while implementing their box-split grammar system. Their work merges two previously separate lines of thought (box grammars and split grammars) into one cohesive system. They also use a platform-agnostic approach for the grammar rules, which are written as a collection of annotated Python functions instead of a more traditional grammar. This allows for easily constraining individual rules, whether by probability or by some arbitrary Python expression.

**Alternative Approaches**

One related, alternate approach to procedural content generation is cellular automata. Green et al. (2019) explores this as an implementation strategy, using it specifically for interior
layouts. A floor of a building was seeded with rooms that grew by one block each turn. After all rooms could not grow any further, walls and doors were placed according to the boundaries defined by the previous step. This system is great for producing more organic-looking results due to being based on a natural process. However, “as structures get larger, they often become disorienting due to the absence of any interior room landmarks”, which seems symptomatic of a lack of larger scale planning on the part of the system. Our approach does produce results that appear more formulaic and artificial, but that quality may be needed in human constructions (e.g. roads, bridges, castles, etc).

Another approach involves generating random levels from a single handcrafted example via a three-dimensional Generative Adversarial Network (GAN). [Awiszus et al.] (2021) use such an machine learning architecture, alongside a block2vec representation inspired by word2vec in Natural Language Processing, to generate arbitrarily sized random levels from just a single input. While this is a much more approachable system for the average Minecraft player, “as [the] proposed method aims to learn and generate structures directly in 3D voxel space” rather than require a different style of reasoning (shape grammars) to create structures, the lack of higher level structure definition means that the “semantic correctness of [generated] structures is not enforced, which can result in for example nonsensical houses.”
Formal Definition

The formal definition of a box-split grammar for this paper is an interesting application of formal languages onto a three-dimensional world. This three-dimensional world consists of axis-aligned bounding boxes (AABBs), that is hexahedra where all edges are either parallel or perpendicular to the X, Y, and Z axes of the overall world. These boxes must have nonzero lengths in all three dimensions, otherwise they would become two-dimensional planes. Additionally, for this paper, these boxes are “integer locked” in that their edges can only start and end at integer values of the three axes. In summary, an AABB is defined by two tuples. One is an origin vertex \( o = (o_x, o_y, o_z) \in (\mathbb{Z}, \mathbb{Z}, \mathbb{Z}) \) and the other is a size tuple \( s = (\delta_x, \delta_y, \delta_z) \in (\mathbb{N}, \mathbb{N}, \mathbb{N}) \). These two tuples are, for convenience, accessible via the \( \text{origin} \) and \( \text{size} \) functions, respectively. Both of these functions also have “dimension-sensitive” variants (namely \( \text{origin}_d \) and \( \text{size}_d \)) that return only the value for that dimension \( d \). A box is said to contain any point \( v = (v_x, v_y, v_z) \) if \( \forall d \in \{x, y, z\}, o_d \leq v_d < o_d + \delta_d \), notated tersely as \( v \in B \). If the box has \( s = (1, 1, 1) \), it is a unit box, which is the smallest possible box. This definition results in a system similar to [Eger (2022)](#), but adds the requirement of “integer locking”.

The input to the grammar is a tuple of the world before the grammar is applied and a single root AABB that the grammar will work within. The output is the world after the grammar has been applied, which may be the same as the input world. The world is defined by a set of materials \( M \) and a mapping between one or more integer locations \((x, y, z)\) and a single material \( m \in M \). An “air” material can be functionally used as a null material. The world can thus be treated as a function that takes a location and returns the material at that
The grammar can be defined as a tuple $(S, R, O, s_0)$, where $S$ is the set of nonterminal symbols, $R$ is the set of possibly-constrained rules each associated with one symbol in $S$, $O$ is the set of observation terminal symbols, and $s_0$ is the starting symbol applied to the root AABB in the input. A grammar rule may do a one of a four actions over an input box, two of which are “terminal” and the other two are “nonterminal” figure 2.

**Terminal Actions**

The first possible terminal action is to do nothing. More formally, for the input box $B$, this action maps every material $m$ in the output world $W_o$ to the same $m$ in the input world $W_i$ for all $(x, y, z) \in B$. This may be a superficially useless action, but there are interesting use cases. Let us consider generating supports for a “realistic” bridge over uneven terrain. Each support will need to vary its height based on the surface directly under it, without
regard for what the heights of other supports are. A do-nothing or no-op rule is useful as a constrained base case for when the box is filled with terrain, with the recursive case simply making a section of the support and recurring.

The other terminal action is to fill the box with some material \( m \). This action is rather simple, as when a more precise position is needed, the nonterminal actions can be used to fill only part of the box. For “emptying” a box, the world would either need to support a “null” material, or could provide a material that is considered the empty material. More precisely, for a box \( B \) to be filled with material \( m \), each vertex \( v \in B \) must have its mapping in the world updated to be \( m \).

**Nonterminals Actions**

The first nonterminal action is splitting the box into some child boxes and attaching symbols to each. A split can only work over one dimension at a time, simplifying notation and implementation. Defining these child boxes can be done using a composition of three
different types of splits: fixed figure 3(a), repeating figure 3(b), and relative figure 3(c).

A fixed split will take in a nonempty list of length $n$ containing positive whole numbers $S$ denoting the lengths of the child boxes, an input box $B$, and a dimension $d$. Its results are a list of boxes $R$ of length $n$, with $\forall i \in \{1 \ldots n\} \ R_i$ is the box with its size in $d$ being $S_i$ and its origin in $d$ being $\text{origin}_d(B) + \sum_{j=1}^{i-1} S_j$. If $\sum S \neq \text{size}_d(B)$, then this split expected a box with a different size than was provided and will fail. figure 3(a) uses the Fibonacci sequence as its input.

A repeating split will take in a single $s$ denoting the length of each child box, an input box $B$, and a dimension $d$. Its results are a list of boxes $R$ of length $\text{size}_d(B)/s$, with $\forall i \in \{1 \ldots n\} \ R_i$ is the box with its size in $d$ being $s$ and its origin in $d$ being $\text{origin}_d(B) + s(i - 1)$. If the box cannot be split without remainder (formally $\text{size}_d(B) \not\equiv 0 \pmod{s}$), then this split will fail. figure 3(b) uses $s = 10$ to split at a high level, subdividing each resulting box into one of size 9 and one of size 1.

The final split variant, relative, allows positive or negative integers, excluding 0, in its length $n$ list $S$, while still taking an input box $B$ and a dimension $d$. Its results are a list of boxes $R$ of length $n$. The calculation of $R_i$ is trickier here, as the size of each box (and thus the offsets) are computed via a conditional. If $S_i > 0$, then $\text{size}_d(R_i) = S_i$. Since $S_i$ can never be 0, the only other case is $S_i < 0$, in which the box is given a minimum
length of 1. Once all boxes are processed, any remaining size is distributed to boxes in
the second case ($S_i < 0$), with priority towards larger absolute values. For an example
of the second case in action, consider the list $S = [-1, -4]$ and $size_d(B) = 10$. In this
case, the resulting lengths will be [2, 8], as the priority will act as the ratio between the
sizes of the two expanding boxes. More formally, box $R_i$ should be expanded by 1 iff
$S_i < 0$ and $\forall j \in \{1 \ldots n\}, j \neq i, S_j < 0, size_d(R_i)/|S_i| < size_d(R_j)/|S_j|$, namely that
the box to be expanded has a smaller ratio relative to its “scale factor” $|S_i|$ than all other
boxes to their respective scale factors. This means that since no box can have a non-
positive size, for any relative split, there is a minimum required size, computed by the
expression $n + \sum_{s \in S, s > 0}(s - 1)$. However if, at the end of this process, any box has a
size of 0 or the minimum required size is larger than the size of the input box, formally
$n + \sum_{s \in S, s > 0}(s - 1) > size_d(B)$, this split will fail. Figure 3(c) splits a box into 5 equal
compartments with 1 wide separators.

Each box has an internal orientation, defined by how the X, Y, and Z axes relate to the axes
of the parent. This allows rules to reorient their children for more appropriate placement.
Consider the example of generating a bridge inside any arbitrary AABB. Bridges tend to
have one horizontal dimension longer than the other, but if the grammar were written to
require say the X dimension to be longer, then any box $B$ where $size_x(B) \leq size_z(B)$ could
not be used. To solve this problem, the initial grammar rule could reorient its sole child to
have its local X be the larger and Z to be the smaller. A reorientation is thus just a one-to-one
mapping from each dimension $d$ to a destination $d_r$. Since the mapping is one-to-one, two
dimensions cannot map to the same $d_r$ and every $d_r$ must have a corresponding $d$. 

12
Rule Constraints

When a symbol \( s \) is attached to a box \( B \), all rules associated with \( s \) are collected into a set of candidates \( C \). Each constrained rule in \( C \) is checked to see that its constraint is upheld in the input world \( W_i \), with any that fail being removed from \( C \). Then, a random rule \( r \) is selected from those that remain in the set and is applied to \( B \). The randomness can be either unweighted or weighted per rule. This application \( r(B) \) may succeed or fail. In case of failure, \( r \) is removed from the set \( C \) and a different \( r \) is selected for application. If \( C \) is empty at any point, then the symbol attachment fails and causes the parent application to fail as well.

A constraint is a function that takes as input the lengths of the box and the materials of every vertex in the box. The constraint returns whether it is held for the inputs. This may seem wasteful, but sending all of the inputs allows for composite constraints modeling boolean conjunction and disjunction. Note that because two sibling boxes (ones which are the same number of actions from the root box) can never intersect with each other, each rule only needs to check its constraint once per box.

An example simple constraint \( c_1 \) could be that the box is filled with a particular material \( m \). Formally, \( \forall (x, y, z) \in B, W_i(x, y, z) = m \). Another simple constraint \( c_2 \) could be that the box is at least 5 units long in the X dimension (\( size_x(B) \geq 5 \)). A logical conjunction \( c_3 = c_1 \land c_2 \) could thus be used to test for both conditions being satisfied on the same box and a logical disjunction \( c_4 = c_1 \lor c_2 \) could thus be used to test for either condition being satisfied on the same box. This notation allows one constraint to encode multiple subconditions, allowing composition.
Problem Description

There are two aspects to the problem that this paper addresses. First is the selection of an appropriate environment for testing a real implementation that follows the above formal definition. The selection was already constrained by the requirements of the aforementioned formal definition, namely an AABB world mapping exactly one material to each unit box. Additionally, it would be preferable to reuse an already existing one, instead of making a bespoke environment for this paper. This would show the flexibility of such an observing grammar.

The second aspect is an example grammar to show the resilience of the implementation. The primary test is that the constraint system evaluates observation constraints properly. An important secondary test is whether the systems handles failure correctly, both at the level of individual rules that have the potential to recover and at the whole system level, where recovery is not possible. A constraint here is that the grammar be small for the purposes of debugging, as the more moving parts there are, the harder it is to track them.
Methodology

Addressing the aspects in order for no particular reason, the choice of implementation system is rather easy. Minecraft fits all of the system criteria nicely. It is famous for its block-focused design and utility as a “blank canvas”. One of the popular community-made tools for it is MCEdit, a world editing software with Python scripting support. Previous work with MCEdit includes the GDMC \cite{Salge2018} and a non-observing grammar system \cite{Eger2022}. The latter was used as the starting point for development, as it is open source and provides a point-of-reference as to previous work in the field.

For the second aspect, a simple bridge was selected as the basis for the input grammar. An important characteristic of the grammar is that the bridge will add piers (the supports under the span) only as far as needed, which is the ground level. Since the ground level may vary within a given input AABB, this would test the constraint system in detecting whether the next box was already full of solid blocks. Since the given input may not have ground all along its bottom and the grammar cannot reach outside of the input, this will test the failure modes of the system. Finally, the bridge’s design can be kept simple (e.g. no curves, piers are straight, span is flat).

The grammar is specified as a set of Python functions, annotated with @rule. In its simplest form, the @rule annotation marks a function foo as an unconstrained rule, meaning that executing the foo() statement in another rule will always applies foo. Since these are normal Python functions, parameters can be used to pass data and possibly change their behavior. Execution starts at the perform function, which tries to expand the bridge rule. The split function is used to implement all three abstract split operations, depending
on its parameters. The \texttt{with} or \texttt{while} where the \texttt{split} is used encapsulates the rules applied to the subblocks.

\textbf{Implementation}

There are two core components to the implementation. The first is the map from rule names to the constrained functions that are their options. A function is added to this map with the \texttt{@rule} annotation, which means that the map is initialized on module load. The second component is a stack of yet-to-be evaluated boxes (the box stack), with the top being the currently evaluated one. If a rule makes subblocks (e.g. a call to \texttt{split}), those subblocks are pushed onto the stack. When a rule is successfully evaluated, it is popped off the stack. If it fails, all of its yet-to-be evaluated siblings are popped and the parent is evaluated again without the failing rule. When the function is invoked inside another \texttt{@rule} function, the top block is used as its context for constraint evaluation. If a constraint fails, that rule is excluded from the set of possible options and another is selected. When no options fit the constraints, the rule fails.

The constraint parameter of an \texttt{@rule} annotation uses an arbitrary expression to constrain whether that particular rule is applicable. Important for this paper is the Observation constraints. The three provided are \texttt{FULL}, \texttt{NONEMPTY}, and \texttt{EMPTY}. Each checks whether the box is full of, contains at least one of, or has none of a specified block type, respectively. The only two block types currently distinguished are \texttt{SOLID} and \texttt{NONSOLID}. Minecraft has significantly more blocks than this, but limiting the grammar to a simple binary distinction allows porting to other games.

In Figure 4, the \texttt{@rule} annotations map the function name \texttt{column_vertical_fill} to
@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 4: An example rule for constructing bridge supports

Figure 5: Diagram of the happy and sad paths of column_vertical_fill
the two constrained variants shown. When the `column_vertical_fill` rule is invoked inside another `@rule`, the top box is used to check the constraint. In this case, if the box is not already full of solid blocks, the ELSE constraint for the second variant will be satisfied. In the second variant, the `split` adds two blocks to the top of the stack and recurses. Eventually, either the first box with a relative size of \(-1\) will have a size of 0, a failing case, or it will be full of solid blocks, triggering the first variant. Since the first variant calls `skip`, the box is simply popped from the stack. The `fill` call is then evaluated, which fills the now topmost box with some blocks. The call stack of `column_vertical_fills` is then popped in sync with the box stack, with each calling `fill` on the corresponding box.

Figure 5 shows, in a more visual manner, the two possible paths for each evaluation of the rule.
Results

The grammar for the aforementioned example bridge is found in Appendix A. It consists of 6 rules implemented in 93 lines of Python, while the overall framework totals near one thousand lines\cite{1}. These thousand lines are mostly in SplitGrammar.py, with the remainder split over GrammarBox.py and MCSplitGrammar.py. SplitGrammar.py implements the core logic of the grammar system, while MCSplitGrammar.py is the wrapper for Minecraft in particular. GrammarBox.py contains the implementation of the BoundingBox class, which represents AABBs.

Generated Examples

The grammar in Appendix A was used to generate figures 6 to 9 and 14(b). The remaining figures, namely 10 and 11 were generated by a modified version of Appendix A. Instead of keeping the Z dimension of the pillars intact, they were split into two 1-by-1 columns after the first layer, creating an arch-like effect. The modified grammar allowed each column to find its own height, regardless of the height of its partner on the other side of the arch (see figure 11). This only required the addition of two rules in between the columns and column_vertical_fill rules (Appendix A.1).

To demonstrate that other structures can be generated with this system, Appendix B contains the grammar for figures 12 and 13 which is a tunnel that only adds walls where there are already solid blocks, as evidenced by the ends following the preexisting contours. Additionally, the roof is only added if at least one of the blocks in that row was solid, shown

\cite{https://github.com/Nicholas-Baron/SplitGrammars}
by the difference in length between the overhang and road.

![Figure 6: A generated bridge with highlighted AABBs](image)

Figure 6: A generated bridge with highlighted AABBs

![Figure 7: An example of a bridge shortening a support](image)

Figure 7: An example of a bridge shortening a support

Figure 6 is an example where the ground is flat, and as such could have been generated by a grammar without observational rules. Using the same grammar, Figures 7 and 8 are examples over uneven ground, requiring the observational rules. In Figure 7, the right support is shorter as the bridge crosses a small rise in the ground, while Figure 8 shows a bridge crossing a ditch, with one of the supports ending in it. Figure 9 is taken from within the official Minecraft game and shows a generated bridge crossing a “naturally” generated desert valley. For a more striking demonstration, consider figures 10 and 11, which show a bridge with arch-like supports. In figure 11, the left and right pillars are of different heights, due to the right side being over a cave entrance.
Figure 8: An example of a bridge lengthening a support into a ditch

A direct comparison with a simpler grammar is needed to fully demonstrate the advantages of this system. The rules in figure 4 could be simplified to a single `fill()`, both reducing the total number of rules and obviating the need for an observation system. However, figure 14(a) shows the result of such a grammar, with a support that runs into the ground beyond the required length. By contrast, using the rules from figure 4 (which are part of Appendix A), the support stops at the ground level, wherever that may be (figure 14(b)).
Figure 9: An in-game screenshot of a generated bridge

Figure 10: An in-game screenshot of a bridge generated with split pillars
Figure 11: A closeup of some of the pillars

Figure 12: One end of a tunnel generated by Appendix B
Figure 13: An in-game screenshot of the other end of figure 12.
Figure 14: Comparison of a non-observing and observing grammar
Conclusion

Shape grammars, while useful in generating “artificial” structures, have been traditionally limited by the inability to conform the result to its surroundings. To address this problem, this paper explored the concept and implementation of observational rules, which are grammar rules that observe the contents of a specified region of the world.

Adding observational rules that match particular features of the environment allows grammars to appropriately adapt to their surroundings while still generating the intended structure. A similar concept in constraint systems is already in use, but they focus on the dimensional characteristics of the box (i.e. its width, height, and depth) and do not handle application failures nicely. This new type of rule can be easily integrated into existing constraint systems in a portable and intuitive manner, with failures propagating up to a rule with possible alternatives and selecting the next alternative.

The example in this paper is that of a bridge where the supports stop at the level of the ground. This is done by having two alternatives, one that observes the ground and the other that recurses to build the column. These rules can be added in a platform-agnostic way, which allows for porting this concept to other similar AABB-based systems. An abstract semantic model of such a system is additionally detailed to allow for comparisons in a formal context.

Future Work

This paper use a bridge as the example grammar in testing the observational rules. However, other structures exist that can benefit from this system. Consider a fort that wants to be
positioned outside of a valley, or a pier that is required to extend some length into water and have moorings only near the water. Both of these examples would benefit from observational rules.

The AABB restriction could be loosened up to allow for other 3D world schemes to take advantage of this style of grammar. Currently, the restriction simplifies the grammar by requiring splits to be defined by a simple plane. Dealing with any arbitrary 3D shape will necessitate more complex split definitions and a more complex shape representation. However, not every game world is represented entirely with nice AABBs.

Integrating these observational grammars into a larger system as “subgrammars” would allow for increased modularity and builds on the observational functions already established in this thesis. For example, a bridge would generate iff it had a road to connect to at both of its endpoints.
References


from mcplatform import *

from pymclevel import alphaMaterials, MCSchematic, MCLevel, BoundingBox

import json

import time  # for timing

import utilityFunctions as uf

from MCSplitGrammar import (Constraint, Dimension, Direction, clearrules, expand_grammar, fill, reorient, rule, split, start_symbol, void,
from SplitGrammar import skip, empty, atom, Observation, BlockType

COLUMN_GAP_OPTION = "Gap between Columns"

COLUMN_GAP_DEFAULT = 2
COLUMN_GAP_MIN = 1
COLUMN_GAP_MAX = 64
COLUMN_GAP_SETTINGS = (COLUMN_GAP_DEFAULT, COLUMN_GAP_MIN, COLUMN_GAP_MAX)

inputs = (
    ("Random Bridge Generator", "label"),
    ("Creator: Nicholas Baron", "label"),
    (COLUMN_GAP_OPTION, COLUMN_GAP_SETTINGS),
)

@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()

@rule

def columns(options):
    COLUMN_GAP = options[COLUMN_GAP_OPTION]
    assert COLUMN_GAP >= COLUMN_GAP_MIN
    items = split(Dimension.X, [COLUMN_GAP, 1], repeat=True)
    while items:
        skip()
        column_vertical_fill()

@rule

def underspan(options):
    COLUMN_GAP = options[COLUMN_GAP_OPTION]
assert COLUMN_GAP >= COLUMN_GAP_MIN

with split(Dimension.X, [-1, COLUMN_GAP]):
    columns(options)
    skip()

@rule
def bridge_layout(options):
    with split(Dimension.Y, [-1, 1]):
        underspan(options)
        fill()

@rule
def bridge(options):
    with reorient(x=Dimension.LARGEST, y=Dimension.Y):
        bridge_layout(options)

def perform(level, box, options):
    expand_grammar(bridge, level, box, rule_params=(options,),
                   json_file="bridge.json")
Appendix A.1

@rule
def split_legs():
    with split(Dimension.Z, [1, -1, 1]):
        column_vertical_fill()
        skip()
        column_vertical_fill()

@rule
def arched_column():
    with split(Dimension.Y, [-1, 1]):
        split_legs()
        fill()

@rule
def columns(options):
    COLUMN_GAP = options[COLUMN_GAP_OPTION]
    assert COLUMN_GAP >= COLUMN_GAP_MIN
    items = split(Dimension.X, [COLUMN_GAP, 1], repeat=True)
    while items:
79   skip()
80   arched_column()
import json

import time  # for timing

import utilityFunctions as uf

from mcplatform import *

from pymclevel import BoundingBox, MCLevel, MCSchematic, alphaMaterials

from MCSplitGrammar import (Constraint, Dimension, Direction, clearrules, expand_grammar, fill, reorient, rule, split, start_symbol, void)

from SplitGrammar import BlockType, Observation, atom, empty, skip

@rule(constraint=Observation.NONEMPTY(BlockType.SOLID))

def replace_solid():
    fill()
@rule(constraint=Constraint.ELSE)

def replace_solid():
    skip()

@rule
def replace_x_row():
    items = split(Dimension.X, [1], repeat=True)
    while items:
        replace_solid()

@rule
def roof():
    replace_x_row()

@rule
def wall():
    items = split(Dimension.Y, [1], repeat=True)
    while items:
        replace_x_row()
@rule
def middle():
    with split(Dimension.Z, [1, -1, 1]):
        wall()
        void()
        wall()


@rule
def floor():
    fill()


@rule(constraint=Dimension.Z >= 4)
def tunnel_layout():
    with split(Dimension.Y, [1, -1, 1]):
        floor()
        middle()
        roof()

@rule(constraint=Dimension.Y > 4)
def tunnel():
    with reorient(x=Dimension.LARGEST, y=Dimension.Y):
        tunnel_layout()

def perform(level, box, options):
    expand_grammar(tunnel, level, box, json_file="bridge.json")