Shape Grammars for Architectural Reconstruction

Macintyre Sunde

January 2022

1 Abstract

Many ancient cities have been reduced to foundations, meaning that many details about how they looked and how people lived there have been lost. Archaeologists can use digital reconstructions to communicate their theories, but these models can often take a long time to fabricate by hand. Shape grammars are a tool that provides both a rigorous method of forming visual hypotheses and an algorithmic way of building models in order to compare them to one another. They do this by generalizing similar complex forms-like the buildings of a city-to a handful of rules. These rules can define the look and feel of an ancient city and can generate many stylistically similar variations of the same city with little extra work. Shape grammars can start to bring to life long ruined cities, and help paint a more accurate picture of how their inhabitants might have lived.

2 Introduction

In computer graphics, content creation is one of the most changing tasks. There are many tools that give artists precise control over each vertex, but modeling by hand is slow and time consuming. In order to create large and detailed worlds, artists need tools that allow them to model repetitive and varied structures quickly and allow them to revise their work easily.

Procedural modeling allows geometry to be created semi-automatically based on a set of rules called a grammar. Noam Chomsky published the first formal definition of a grammar to test the hypothesis that natural language can be described based on a finite set of grammar rules. This hypothesis has since been proven false, but grammars have found a place in many Computer Graphics papers since. Przemyslaw Prusinkiewicz and Aristid Lindenmayer In "The Algorithmic Beauty of Plants" [5] used grammars produce to visually meaningful images and 3D models. In their approach, they use L-Systems (Lindenmayer Systems), a slightly different definition of a grammar, to simulate the growth of plants. The book also introduces Turtle Graphics, a method of generating images from strings where each character is given a rule for manipulating a virtual

pencil. When combined with the rules from an L-System, Turtle Graphics can create complex shapes closely resembling a wide variety of plants. [5]

L-Systems are the foundation for modern Shape Grammars. Another method of transforming a string, Shape Grammars use rules to place and manipulate shapes in 3D space. Pascal Müller (et. al.) use Shape Grammars to model buildings. Similar to the way plants can be broken into branches, buildings can be decomposed into a hierarchy of detail. Cities can be divided into floor plans, which can be replaced by volumes, sectioned into floors, and so on[4]. Shape Grammars are a formalization of L-Systems in that they provide the structure to generalize grammars to model a wider array of complex shapes.

This ability to define rules for the style and layout of buildings is what makes shape grammars perfect for archaeology. The ruins of an ancient city can provide enough input to a Shape Grammar to generate many different possibilities for what the city may have looked like. Jonathan Maïm (et. al.) use Shape Grammars to reconstruct the city of Pompeii using a variety of building types and a rough layout for the city foundations. The city was then used as a stage for a crowd simulation to further bring the city to life. More recently the SPACES project (Spatialized Performance And Ceremonial Event Simulations) aims to digitally reconstruct the city of Pachacamac, Peru in order to understand how large events may have looked for the pre-Colombian Inca.

3 Background

3.1 Grammars

All of the concepts in this paper are predicated on the idea of a *grammar*. Grammars are sets of rules that map one object to a set of other objects. These can be letters and strings as described by Lindenmayer and Chomsky, or shapes and volumes. A formal definition for the use of a grammar in linguistics was posed by Chomsky in the 1950's [1].

Definition 3.1.1. A *finite state grammar* G is a system with a set of finitely many states $S = \{s_0, ..., s_q\}$, a set $A = \{a_{ijk} \mid 0 \leq i, j \leq q \; 1 \leq k \leq N_{ij} \; \forall \; i, j\}$ of transition symbols, and a set of pairs $C = \{(s_i, s_j) \mid s_i, s_j \in S\}$. Each pair of states is said to be *connected*, and produces a_{ijk} when transitioning from one state to the next.

Let $s_{\alpha_0}, ..., s_{\alpha_m}$ be a sequence where $(s_{\alpha_i}, s_{\alpha_i+1}) \in C$ for each $i < m$. Using A, this sequence will generate the sentence $a_{\alpha_0\alpha_1k_1}, ..., a_{\alpha_{m-1}\alpha_mk_m}$ where $k_j \in N_{\alpha_i\alpha_{i+1}}$. The language containing only these sentences is called the language *generated* by G.

In 1968, the biologist Aristid Lindenmayer invented L-Systems with the idea to analyze plants rather than language [5]. One of the key differences between these works is that states in a Chomsky Grammar are applied sequentially, while rules in L-Systems are applied in parallel. This difference coincides with Lindenmayer's interest in the growth patterns of plants, because each part of a

plant grows simultaneously. Conversely, parts of a sentence are built sequentially and depend on previous words and phrases.

3.2 L-Systems

L-Systems are a set of rules for the production of strings of letters. Given starting word, known as an *axiom* (ω) and a set of productions from letters to words, an L-System generates a new word by replacing the letters of the axiom using the rules of the production. The simplest L-Systems are deterministic and context free, known as DOL-systems.

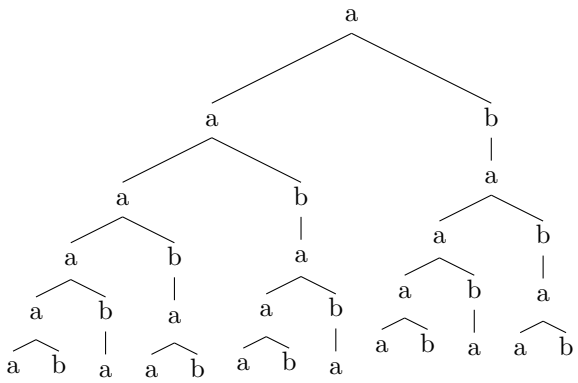
Definition 3.2.1. Let V be an alphabet and W the set of all words of the alphabet. An *L-System* is an ordered triplet $G = \langle V, \omega, P \rangle$. $\omega \in W$ is called the axiom. $P \subset V \times W$ is the set of productions $\{(a, x) \mid a \in V, x \in W\}$, written as $a \to x$. a and x are called the *predecessor* and *successor* of the production. L-Systems are said to be *deterministic* (or DOL-Systems) if and only if for each $a \in V$ there is exactly one $x \in W$ such that $(a, x) \in P$. [5]

Definition 3.2.2. Let $w = a_1, a_2, ..., a_m$ be a word in W. Then $v = x_0, x_1, ..., x_n$ is *directly derived* from v (written $w \Rightarrow v$) if and only if for all $a_i \in w$ there exists a rule $(a_i, x_i) \in P$. v has a derivation of length n if and only if there exist a sequence of words $\mu_0, \mu_1, ..., \mu_n$ such that $\mu_0 \Rightarrow \mu_1 \Rightarrow ... \Rightarrow \mu_n$, where $\mu_0 = w$ and $\mu_n = v$.

Every derivation of one word from another results in a tree that branches from each predecessor to the set of characters in its successor. I will call such a tree the *production tree* of $w \Rightarrow v$. Note that such a tree will have a height n, and a given level i in the tree can be read off as the word μ_i.

Example 3.2.1. Consider the following set of production rules. Even simple L-Systems are capable of generating relatively complex strings, making them a good starting point for a procedural generation algorithm.

$\omega : a \quad n = 5$
$p_0 : a \to ab$
$p_1 : b \to a$



3.2.1 Turtle Graphics

In order to start generating complex models, we need to translate the complex words of L-Systems into instructions for drawing shapes. This is the idea behind turtle graphics. In addition to production rules, each letter in a word has a rule for how to control a pencil (or, a turtle holding a pencil). A turtle T can be defined as $\langle (x, y), \alpha \rangle$ where x and y are the position of the turtle on a canvas and α is the angle of the turtle.

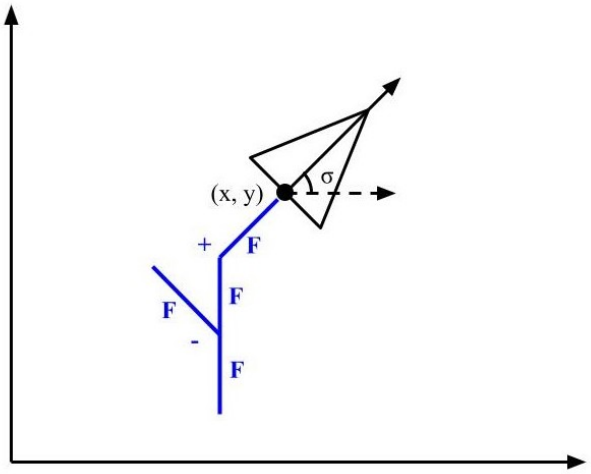


Figure 1: Turtle position and rotation (in black) with an example drawing (in blue).

The following are rules given by Prusinkiewicz and Lindenmayer to draw using L-Systems.

$F :=$ Move forward and draw a line. $x_{new} = x + cos(\alpha)$, $y_{new} = y + sin(\alpha)$
$f :=$ move forward without drawing a line
$+ :=$ turn to the right. $\alpha + \sigma$, where σ is a constant angle.
$- :=$ turn to the left. $\alpha - \sigma$.

Turtle Grammars use the idea of edge rewriting: replacing F with a string of characters in V. The other rules in the grammar are trivial, mapping the other characters to themselves. Characters which have only trivial production rules are called *terminal symbols* and those that don't are called *non-terminal symbols*.

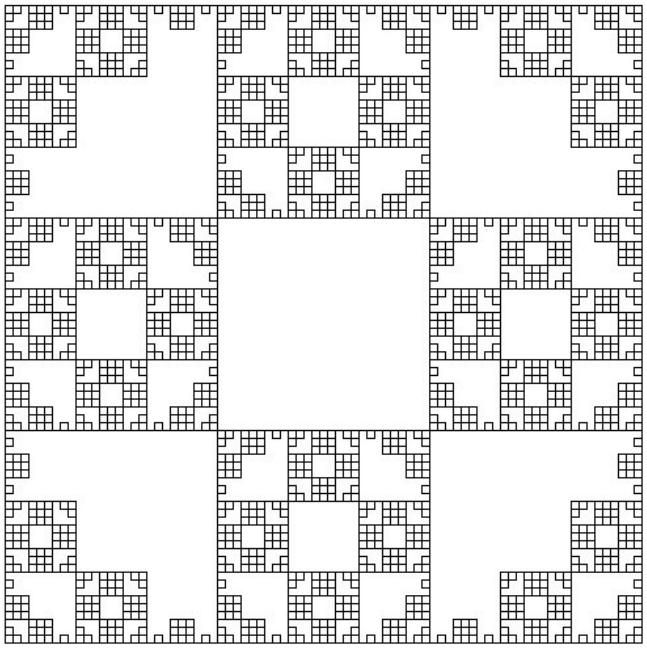


Figure 2: Example of a fractal shape generated with L-Systems.
ω : F-F-F-F $\sigma = 90°$ $n = 4$
p_0 : F $\rightarrow$ FF-F-F-F-FF

3.2.2 Branching Structures

With the addition of a *transform stack* it is possible to draw branching structures. A transform stack is simply an array of previous turtle states $\langle (x, y), \sigma \rangle$ of our turtle. A state can be saved by pushing it to the stack, and loaded by popping it off the stack allowing the turtle to jump between different points in a drawing. forming one branch before jumping to the base to continue with another (Figure 4). The previous grammar can be updated to include the following rules:

[:= push to the stack
] := pop from the stack

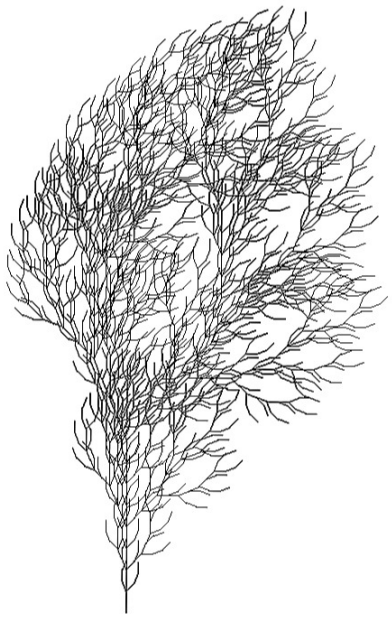


Figure 3: Example of a tree branching structure
ω : F $\sigma = 22.5^\circ$ $n = 4$
p_0 : F $\rightarrow$ FF-[-F+F+F]+[+F-F-F]

3.2.3 Stochastic L-Systems

Stochastic L-systems allow for random choice between rules in a production by using a probability distribution over the set. Using randomness produces more variation than can be found in DOL-systems and can more accurately simulate plant growth (Figure).

Definition 3.2.3. A stochastic L-system is an ordered tuple $G_\pi = \langle V, \omega, P, \pi \rangle$. π is the function $\pi : P \rightarrow (0, 1]$ that maps each production rule to its probability of occurrence. The sum of $\pi((a, x))$ for all productions containing a is always 1.

Definition 3.2.4. The derivation $w \Rightarrow v$ for $w, v \in W$ is called the *stochastic derivation* if the probability of applying a production p with predecessor a is $\pi(p)$. In other words, multiple rules containing the predecessor a can be chosen between in a single derivation step.

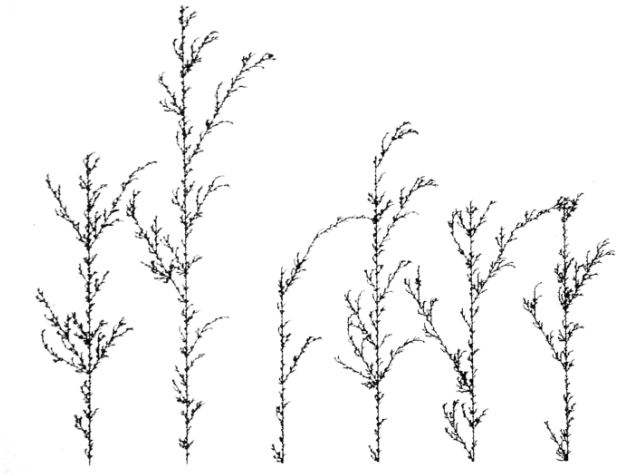


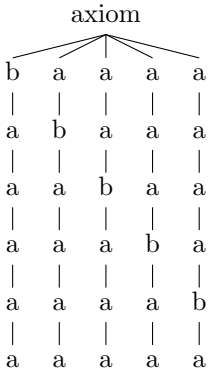
Figure 4: Example of a stochastic structure [5]
ω : F $\quad \sigma = \frac{\pi}{6} \quad n = 5$
p_0 : F (.33) $\rightarrow$ F[+F]F[-F]F
p_0 : F (.33) $\rightarrow$ F[+F]F
p_0 : F (.33) $\rightarrow$ F[-F]F

3.2.4 Context Sensitive L-Systems

To more finely control the shapes that L-Systems can produce, production rules can be given a specific context under which to be applied. For example, to simulate plants that only flower at specific points, we can use the previous and next symbols in the word to determine the proper placement. The *left* and *right context* are said to be the letters appearing before and after the predecessor of a production rule in a given word. In general, production rules with a context take precedence over rules without a context so long as the context is met.

Example 3.2.2. This L-system propagates b to the right.

ω = baaaa $\quad n = 5$
$p_0 : b < a \rightarrow b$
$p_1 : b \rightarrow a$



In turtle graphics that we only care about the context of line-drawing rules, and not about turtle-rotating ones. This can usually be handled by simply ignoring them when searching for the desired context. However, applying this principle naively to bracketed L-systems doesn't work. Simply ignoring brackets only searches for context on a single branch when the necessary context might lie on a different one, or on many different branches.

This problem is solved in two parts. First, when searching for the right context, skip over any pair of brackets and ignore any open bracket. Second, When searching for left context, consider each branch (or pair of brackets including anything that appears after the first pair of brackets).

Example 3.2.3. Consider the following grammar:

$\omega = JFF[FFJF]F_{match}[J[J[JF]F]JJJ.$
$p_0 : JFF < F > JJF \rightarrow FJ$

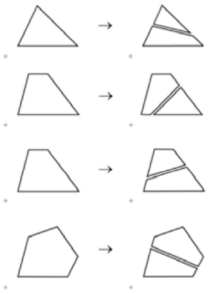
Matching the rule p_0 to the character F_{match}, we ignore $[FFJF]$ in the left context, and match $[J[J[JF]F$ while ignoring $[JF]$ in the right context.

4 Shape Grammars

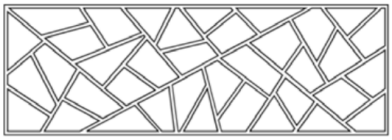
Shape Grammars are an extension of L-Systems and Chomsky Grammars, but are extended for modeling in any dimension. An early example of Shape Grammar comes from George Stiny's, *Ice-Ray: A Note on the Generation of Chinese Lattice Designs* where he uses old lattice designs to inspire a Shape Grammar (Figure 5). The production rules of this grammar replace a polygon of 3, 4, or 5 sides with two smaller polygons, taking into account their relative areas[6]. More recently, work by Pascal Müller (et. al.) expands on the work of Stiney, Lindenmayer, Prusinkiewicz, and Chomsky to design a system called CGA (Computer Generated Architecture) Shape, designed to model buildings.

4.1 Shape

In order to define Shape Grammars, it is important to understand shape in the right context. Shapes are the combination of a name or ID and geometry. The



(a) Production rules.



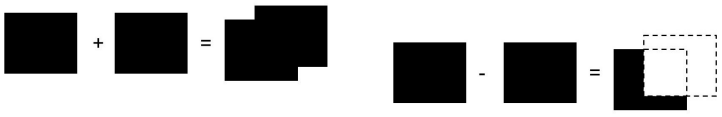
(b) Lattice design inspired by Chengtu, Szechwan, 1800 AD.

Figure 5: George Stiny's ice-ray grammar [6]

ID serves to identify similar shapes in production rules. As with L-Systems these symbols can either be terminal or non-terminal. The geometry of a shape is a collection of points, lines, and polygons that can form volumes in $\mathbb{R}^3$ [7].

CGA Shape uses tags, which are strings that represent groups of shapes with specific properties. It also includes specific geometric attributes suc as position P, scale S, and frame $F = \langle i, j, k \rangle$ where i, j, k are orthonormal. The *scope* of a shape is defined in terms of these three attributes [4]. CGA Shape sums all resulting shapes to create mass-models for buildings.

Similar to numeric arithmetic, we can do operations to shapes (Figure 6). Importantly, the shape operations defined for use in shape grammars, can create and destroy vertices, edges, and faces leave only the external surface (or perimeter) of the resulting shape [7].



(a) Sum: $A + B$
Sum is analogous to the union ($\cup$) of shapes A and B where the resulting shape contains all points in A and B.

(b) Difference: $A - B$
The difference of A and B contains all points in A that are not in B.

(c) Product: $A\dot{B} = A - (A - B)$
Product is analogous to the intersection ($\cap$) of A and B where the result is all points both in A and in B.

(d) Symmetric difference:
$A \oplus B = (A - B) + (B - A)$
Symmetric difference is the opposite of product and can alternatively be represented as the sum of two subtraction operations.

Figure 6: Shape arithmetic

4.2 Production Process

The production of a Shape Grammar works by replacing symbols in an axiom to build a set of symbols that corresponds to a set of shapes. Instead of simply deleting shapes as Stiny does, CGA Shape deactivates shapes after they are replaced, in order to query them in future production rules. The production process is as follows: [4]

1. Select an active shape in the set B.
2. Choose a production rule with $(B, x) \in P$ to compute the successor B'.
3. Mark the shape B as inactive, add B' to the model, and repeat from step (1).

Like Chomsky Grammars, CGA Shape is derived sequentially to provide more control over the result [4]. This avoids the possibility of accidentally placing two building components in the same location in space. Moreover,

sequential production allows for dependency from one feature to another at or above its height in the production tree.

CGA Shape gives each rule a priority based on how detailed the predecessor of the rule is. Shapes are selected based on the highest priority rule that contains an active shape. Thus, the successors are derived mostly breadth first, although-depending on the choice of axiom and production rules, it is not strictly breadth first.

4.3 Production Rules

There CGA Shape defines a variety of different functions for manipulating the scope and placing shapes. Like L-Systems, the production rules of CGA Shape can be stochastic and context sensitive.

Example 4.3.1.
Context sensitive production rule:

(rule #): (ID): (condition) $\rightarrow$ { (list of functions) }

Context sensitive and stochastic where p_i to the probability of each rule occurring $(p_1 + p_2 + ... + p_n = 1)$

$$\begin{aligned}\text{(rule \#): (ID): (condition)} &\rightarrow \{ \text{ (list of functions) } \} \ (p_1) \\ &\rightarrow \{ \text{ (list of functions) } \} \ (p_2) \\ &... \\ &\rightarrow \{ \text{ (list of functions) } \} \ (p_n)\end{aligned}$$

4.3.1 Scope Rules

Scope rules allow for the manipulation of shapes. $T(x, y, z)$ adds a translation vector to the scope position, $S(x, y, z)$ scales the scope, and R_x, R_y, R_z which rotate the frame around a given axis. $[,]$ are used to push and pop the current scope to and from the transform stack. $I(objectid)$ creates an instance of a 3D model using the current scope as the origin. The scope is passed down to any non-terminal symbols as the production process continues (Figure 7).

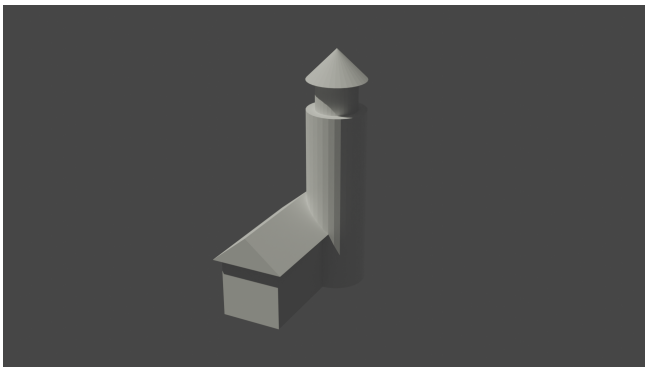


Figure 7: 1: A $\rightarrow$ $[T(0,0,1)\ S(1,2,1)\ I(cube)\ T(0,0,1)\ S(1.2,2.2,1)\ I(\text{"}roof\text{"})]$
$T(1,2,3)\ S(1,1,3)\ I(cylinder)\ T(0,0,3.5)\ S(0.7,0.7,0.5)\ I(cylinder)$
$T(0,0,1)\ S(1,1,0.5)\ I(cone)$

4.3.2 Split Rules

Split rules divide shapes along different axes. They take the form
Subdiv$(axis, d_1, d_2, ..., d_n)\{s_1 \mid s_2 \mid ... \mid s_n\}$ where d_i is the distance between each split and s_i is the production rule for each respective split. The width of the scope is not always known, making it impossible to provide numeric values for every split. In these cases any number of d_i can be relative, denoted r. The numeric value of r is

$$(scope.scale - \sum_{i=1}^{m} d_i)/n$$

where *scope.scale* is the scale along the desired axis, m is the number of numeric values and n is the number of relative values.

4.3.3 Repeat

Repeat rules tile elements as many times as there is room on the given axis. They take the form Repeat$(axis, d)\{s\}$ where d is the distance between elements and s is the next production rule.

4.3.4 Component Split

It is often the case that shapes need to be split in to their component parts (faces, edges, or vertices) to be operated on by production rules. Component split rules take the form Comp$(type)\{s\}$ where *type* represents the component (e.g. "edges", "faces", "vertices") and s is the shape that is created from the split.

4.4 Complex Modeling

Using a combination of shape rules it is relatively straightforward to build complex mass-models out of primitives. However, adding facades and other small details to these models is non-trivial because there is no general rule for how to handle any 2D shape. Before moving to smaller details a mass model is split into its component faces. Any resulting triangles and rectangles can be further detailed, while shapes that are too complex are kept as they are. To keep the facades consistent with the rest of the model, CGA Shape uses a combination of occlusion testing and snap lines to keep any details from overlapping each other, or larger parts of the model and to place details along important contours of the mass-model.

4.4.1 Occlusion

Occlusion tests determine whether the current scope overlaps partially, or fully with another scope. These take the form Shape.occ(tag) == o where o can be "none", "part", or "full". The tag represents a specific subset of shapes that share a property. For example, a particularly useful tag is "noparent" which applies to shapes that are not an ancestor to the current shape in the production tree. Such shapes will usually occlude the current shape. Tags can also be arbitrarily assigned to different shapes to create more specific filters.

Example 4.4.1. The following rules will only place a door if no other scope intersects with the current one. The "noparent" tag is used to search for any scope that is not an ancestor of the current scope in the production tree.

1: A: Scope.occ("noparent") == "none" $\rightarrow$ *door*
2: A: Scope.occ("noparent") == "part" $\rightarrow$ *wall*

4.4.2 Snapping

Snapping functions to align more detailed shape rules to dominant lines in the mas-model. These *snap lines* are generated by the rule Snap($axis$), which creates a snap line at the current scope along a given axis. Snap lines are used with the repeat and split rules, by using Repeat($axis$ + "S", d){s} where "S" indicates the use of snap lines. When subdividing across a snap line, the subdivision closest to the line moves to the snap line without affecting the other divisions. Repeating across a snap line scales all repetitions before and after the division closest to the snap line so as to have even spacing everywhere (Figure 8.

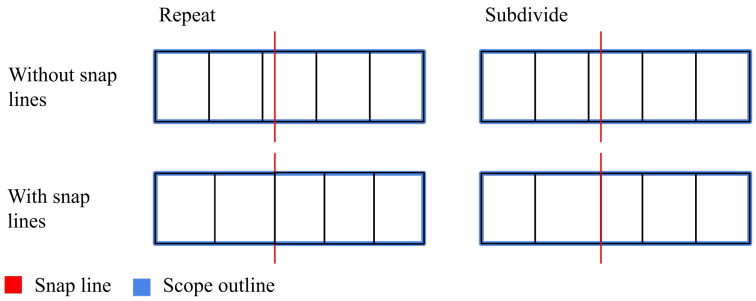


Figure 8: Left: snap lines change every split of a Repeat. Right: snap lines only change the nearest two splits of a subdivision.

4.5 Archaeological Applications of Shape Grammars

Shape grammars provide an excellent way of both forming and testing hypotheses about what ancient cities looked like. However, building up the components of a grammar is not trivial. The grammar rules must be crafted to closely match the architectural style of the time, and an axiom for the grammar must be generated from the footprint of the city. The 2007 paper "Populating Ancient Pompeii with Crowds of Virtual Romans"[3] uses Geographic Information Systems (GIS) data to approximate the footprint of Pompaii (Figure 9. The shape grammar that the paper uses three general steps:

1. Extrude a building footprint in the axiom and tag each face with its relation to the road (eg. front, back, left, right).
2. Place a roof volume, an optional second floor, or leave the roof flat.
3. for each non-occluded side, either place a door if it is front-facing, or place a window

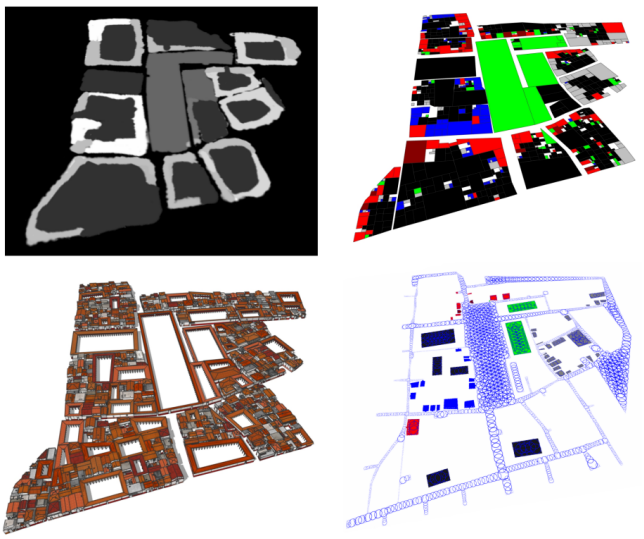


Figure 9: The top two images show a rough sketch and final design of the axiom for the grammar that is color coded to match the uses of different buildings. The bottom two images show the final result and navigation mesh that is generated by the grammar.

5 Future Work

My aim for next semester is to improve upon the algorithms that I have research and apply them to the SPACES project. SPACES combines shape grammars and crowd simulations with hopes to parameterize them in order to identify realistic theories for how large events celebrations in these cities might have looked. The project is focusing on the city of Pachacamac, a town that was used for religious pilgrimage, making it a good place to study celebrations. It is a unique city to design a shape grammar for because there are a number of buildings that only appear a single time. A Shape Grammar needs to be interwoven with pre-made geometry seamlessly. To make this integration happen I will do more research into modifying the a production tree after it is generated.

References

[1] N. Chomsky, "Three models for the description of language," *IRE Trans. on Information Theory*, vol. 2, no. 3, pp. 113–124, 1956.

This paper is outlines the first formal definition for grammars as a description for natural languages.

[2] K. Chow, A. Normoyle, J. Nicewinter, C. L. Erickson, and N. I. Badler, "Crowd and procession hypothesis testing for large-scale archaeological

sites," *MARCH Workshop, IEEE International Conference on Artificial Intelligence And Virtual Reality (IEEE AIVR)*, 2019.

The SPACES project aims to create a system for visualizing large events in ancient cities.

[3] J. Maïm, S. Haegler, B. Yersin, P. Mueller, D. Thalmann, and L. V. Gool, "Populating ancient pompeii with crowds of virtual romans," *The Eurographics Association*, 2007.

This paper serves as an excellent example of how shape grammars can be used to aid in the archaeological reconstruction of ancient cities.

[4] P. Müller, P. Wonka, S. Haegler, A. Ulmer, and L. V. Gool, "Procedural modeling of buildings," *ACM SIGGRAPH 2006 Papers*, p. 614–623, 2006.

This paper describes the CGA Shape, a shape grammar designed for buildings and cities. Müller(et. al.)'s work forms the main body of my paper, and provides all the necessary structure to start reconstructing archaeological cities.

[5] P. Prusinkiewicz and A. Lindenmayer, *The Algorithmic Beauty of Plants*. New York: Springer-Verlag, 1990.

This book is a guide to L-Systems and Turtle Graphics, which are foundational to the development of Shape Grammars. I am using chapters 1 and 2 as the most of the book is dedicated to reproducing plant-like structures rather than buildings.

[6] G. Stiny, "Ice-ray: A note on the generation of chinese lattice designs," *Environment and Planning B: Planning and Design B4*, pp. 89–98, 1977.

This paper details Stiney's famous Ice Ray grammar, and it's inspiration from Chinese Latice designs. The Ice Ray grammar was helpful for me to understand how shape grammars can work, and I use it in this paper as an example.

[7] ——, "Intro to shape grammars," *MIT OpenCourseWare*, 2018.

Stiny provides an accesible intro to shape grammars, and why they are powerful in this course. The course provides free access to Lecture notes and slides, and Stiney's book Shape: Talking about Seeing and Doing. The course can be found at: ocw.mit.edu/courses/architecture/4-540-introduction-to-shape-grammars-i-fall-2018

[8] M. Özkar and S. Kotsopoulos, "Introduction to shape grammars," *SIGGRAPH*, 2008.

This slide deck is a quick and visual introduction to shape grammars. It outlines the work of George Stiny in defining what a shape grammar is and explains the inherent ambiguities of shape, and how to do shape arithmetic. The slides have many examples of the applications of shape grammars in architecture, that was designed using them.