Shape Grammars for Architectural Reconstruction

Macintyre Sunde, Advisor: Aline Normoyle

January 2022

Contents

1 Abstract

Many ancient cities have been reduced to foundations, losing their architectural style and hints as to their usage. Today, Archaeologists use digital reconstructions to communicate their theories, but these models can often take a long time to fabricate, so the theories need to be well formed before they enter the stage of visualization. *Shape grammars* are a tool that provides a tool for formalizing hypotheses visually, in a way that can be rapidly tested and compared. They do this by encoding similar complex forms- like the windows of buildings-to a set of text rules. When properly defined, these rules can define the look of an ancient city and can generate many stylistically similar variations of theory 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 challenging tasks. There are many tools that give artists precise control over each vertex, but modeling by hand is slow and time consuming. Archaeological reconstructions require large and detailed city models which are often composed of many similar models. Archaeologists need a tool to help them quickly visualize how their theories for city structure actually look.

Procedural modeling allows geometry to be created semi-automatically based on a set of rules called a grammar. Przemyslaw Prusinkiewicz and Aristid Lindenmayer in *The Algorithmic Beauty of Plants* [5] used grammars to produce visually meaningful images and 3D models. In their approach, they use L-Systems or Lindenmayer Systems 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]

Similarly to L-systems, 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 [4]. Cities can be divided into floor

plans, which can be replaced by houses, sectioned into floors, and so on. Shape Grammars Generalize L-systems to handle 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 [3]. 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, located near Lima in modern day Peru, in order to understand how large ceremonial events may have looked for the pre-Columbian Inca [2].

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 developed L-Systems to adapt grammars to the modeling of plants [5]. One of the key differences between L-systems and Chomsky Grammars 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.

3.2 L-Systems

L-Systems are a set of rules for the production of strings of letters. Given a 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.

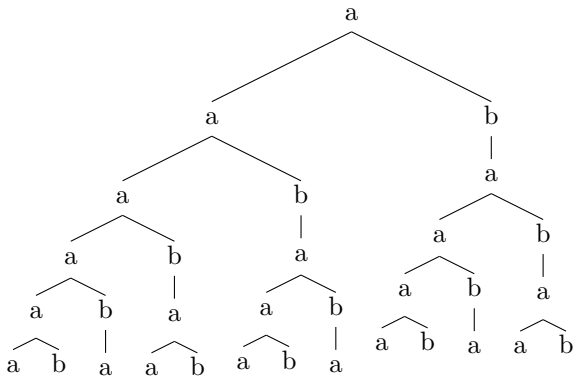
Definition 3.2.1. Let V be an alphabet and W the set of all words of the alphabet. An *L-System* is an ordered triplet $G = \langle V, \omega, P \rangle$. $\omega \in W$ is called the axiom. $P \subset V \times W$ is the set of productions $\{(a, x) \mid a \in V, x \in W\}$, written as $a \to x$. a and 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 w (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 $\langle (x, y), \sigma \rangle$ where x and y are the position of the turtle on a canvas and α is the angle of the turtle.

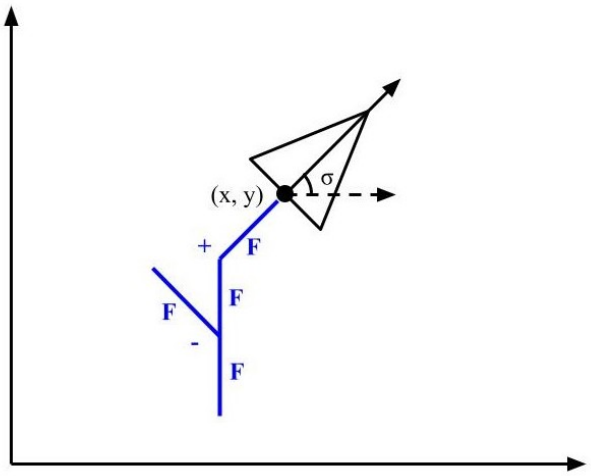


Figure 1: Turtle position (x, y) and rotation σ (in black) with an example drawing (in blue).

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

$F :=$ Move forward and draw a line. $x_{new} = x + cos(\sigma),\ y_{new} = y + sin(\sigma)$
$f :=$ 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* all other symbols are called *non-terminal symbols*.

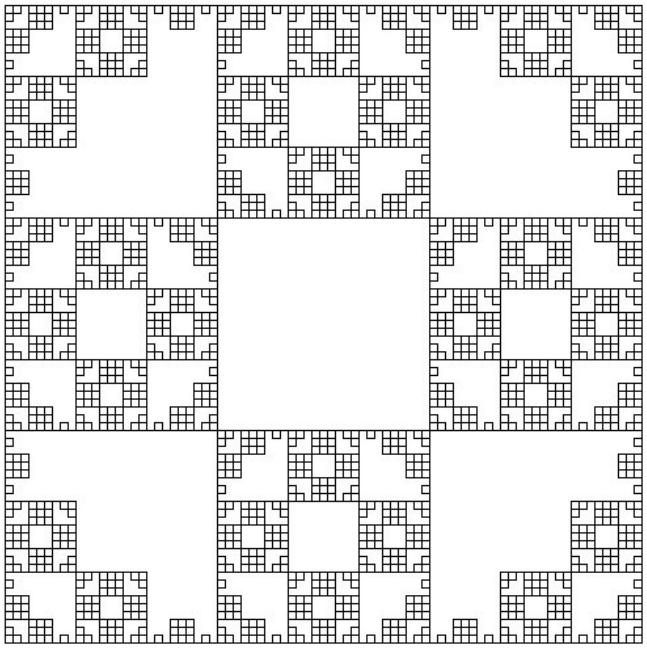


Figure 2: Example of a fractal shape generated with L-Systems.
ω : F-F-F-F $\sigma = 90°$ $n = 4$
p_0 : F $\rightarrow$ FF-F-F-F-FF

3.2.2 Transform Stack

With the addition of a *transform stack* it is possible to draw branching structures. A stack stores the turtle state prior to a branch so that the state can be restored after the branch is drawn (Figure 4). The previous grammar can be updated to include the following rules:

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

Grammars including these rules are often called bracketed L-systems.

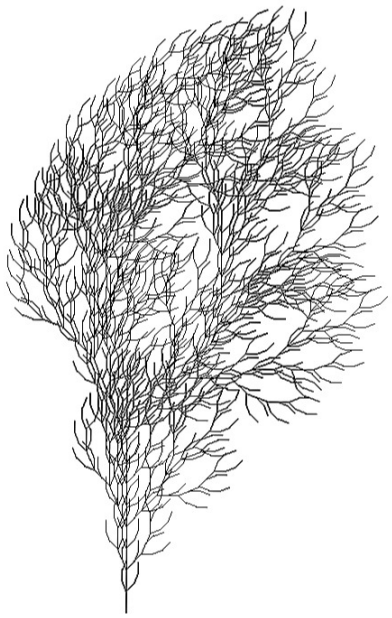


Figure 3: Example of a tree branching structure
ω : F $\sigma = 22.5°$ $n = 4$
p_0 : F $\rightarrow$ FF-[-F+F+F]+[+F-F-F]

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 variation that can be found in DOL-systems. This can add to the functionality of L-systems by enabling them to generate a range of drawings of a specific style instead of one explicit drawing (Figure 4).

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 in a single derivation step.

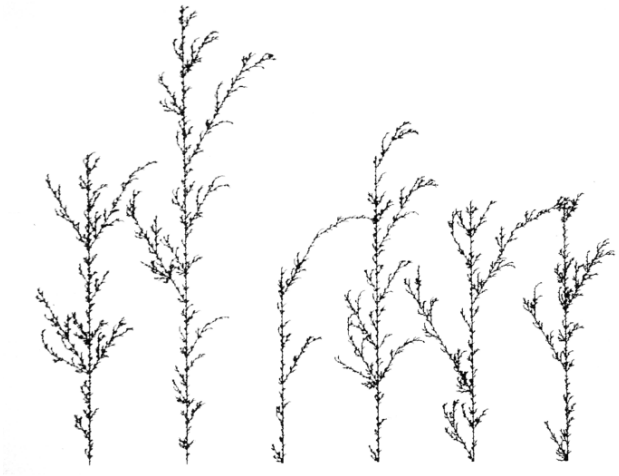


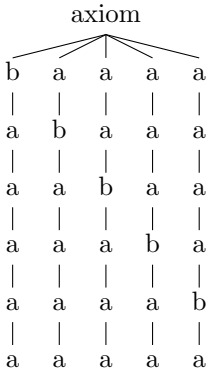
Figure 4: Example of a stochastic structure [5]
ω : F $\quad \sigma = \frac{\pi}{6} \quad n = 5$
p_0 : F (.33) $\rightarrow$ F[+F]F[-F]F
p_0 : F (.33) $\rightarrow$ F[+F]F
p_0 : F (.33) $\rightarrow$ F[-F]F

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 context sensitive L-systems, we usually only care about the context of rules that draw lines. This can usually be handled by ignoring other rules 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 the left context, consider each branch (or pair of brackets including anything that appears after the first pair of brackets).

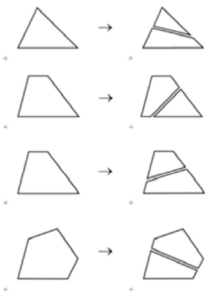
Example 3.2.3. Consider the following grammar:

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

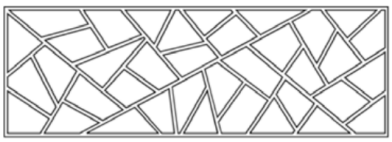
Matching the rule p_0 to the character F_{match}, we ignore $[J]$ in the left context, and $[JF]$ in the right context. match $[J[J[JF]F$ while ignoring $[JF]$ in the right context.

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[8]. More recently, work by Pascal Müller (et. al.) expands on the work of Stiney, Lindenmayer, Prusinkiewicz, and Chomsky to design a system called CGA Shape (Computer Generated Architecture), designed to model buildings [4].



(a) Production rules.



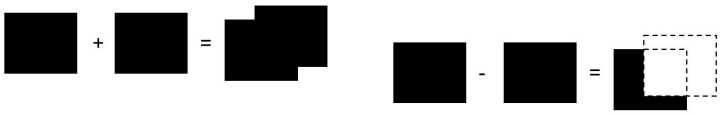
(b) Lattice design inspired by Chengtu, Szechwan, 1800 AD.

Figure 5: George Stiny's ice-ray grammar [8]

4.1 Shape

In order to define shape grammars, it is important to understand shape in the right context. Shapes are the combination of an ID, geometry, a set of tags, and other geometric attributes. The 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 $\mathbb{R}^3$ [9]. A tag is a string that represents groups of shapes with specific properties. Shapes can also include specific geometric attributes such 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].

Shapes can be combined using operations from set theory (Figure 6). The shape operations defined for use in shape grammars can create and destroy vertices, edges, and faces leaving only the surface or perimeter of the resulting shape [9].



(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. CGA Shape keeps track of the rules that have been called and the shapes that have been placed in order to query them in future 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 rules are applied 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.

4.3 Production Rules

The 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 $(p_1 + p_2 + ... + p_n = 1)$

(rule #): (ID): (condition) $\rightarrow$ { (list of functions) } (p_1)
$\rightarrow$ { (list of functions) } (p_2)
...
$\rightarrow$ { (list of functions) } (p_n)

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, R_x, R_y, R_z rotates the scope around a given axis and open and close brackets 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).

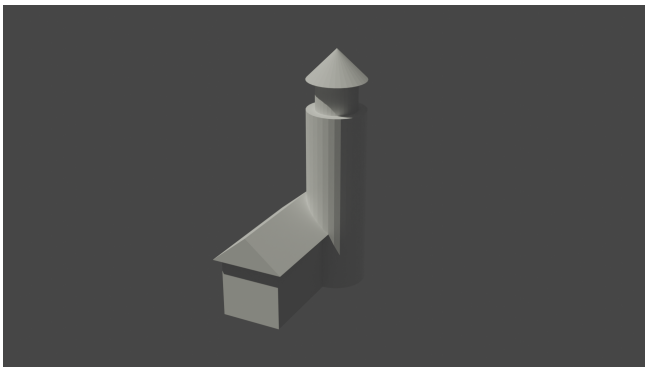


Figure 7: 1: A $\rightarrow$ $[T(0,0,1)\ S(1,2,1)\ I(cube)\ T(0,0,1)\ S(1.2,2.2,1)\ I(\text{"}roof\text{"})]$
$T(1,2,3)\ S(1,1,3)\ I(cylinder)\ T(0,0,3.5)\ S(0.7,0.7,0.5)\ I(cylinder)$
$T(0,0,1)\ S(1,1,0.5)\ I(cone)$

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 into 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 models out of simple 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 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 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 aligns more detailed shape rules to dominant lines in the 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.

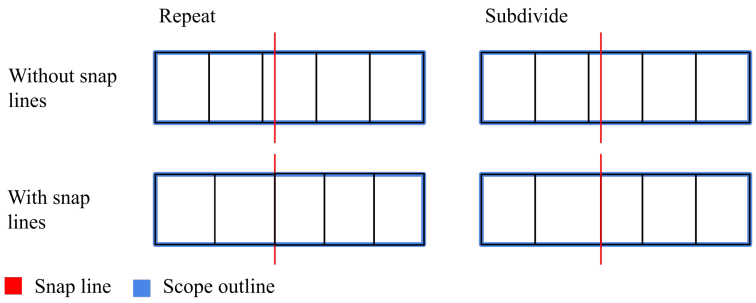


Figure 8: Left: snap lines change every split of a Repeat. Right: snap lines only change the nearest two splits of a subdivision.

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. Hypotheses can be formalized in the shape grammar syntax and visualized by generating shapes with those rules. However, building a grammar is not always 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 Pompeii (Figure 9. In the work, the shape grammar follows these 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

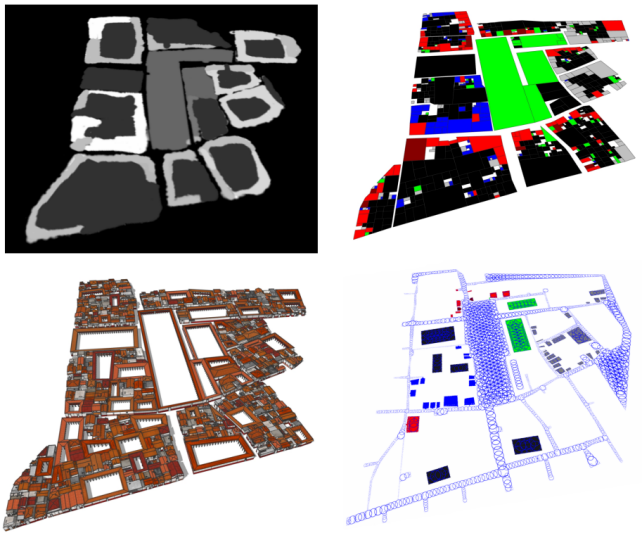


Figure 9: The top two images show a rough sketch and final design of the axiom for the grammar that is color coded to match the uses of different buildings. The bottom two images show the final result and navigation mesh that is generated by the grammar.

5 Implementation

This semester I implemented a Shape Grammar scripting language (SGL) in Unity engine. The language functions as a user interface, to design shape grammars in much the same way as grammars are designed in CGA Shape [4]. SGL is interpreted to C# and uses Unity's API to manipulate geometry. The parser for the language is implemented using an LALR parser generator written in C# [6]. A parser generator was necessary for this project because although SGA is not turing complete, it still has a complex grammar.

SGL has some important differences from CGA Shape. CGA Shape uses a priority scheme to determine the order in which grammar rules are run. Each rule is assigned a priority, and added to a corresponding queue. The rules are run by choosing the rule at the position 1 in the lowest priority queue first. SGL chooses whether each rule is added to the front or back of a single queue at each

production rule. Each production rule has its own scope and the relationships between scopes are kept track of using a tree.

5.1 Language Features

The structure of SGL file is as follows. At the top of the file is a header containing local variables and global, grammar variables relating to aspects of the generation. Production rules are defined by a name, followed by one or more lists of grammar rules and production rules. SGL implements a set of rules similar to those of turtle graphics, plus Subdivide from CGA Shape and 'at depth', which executes a rule after a certain depth in the generation process.

1. Translate: `T(x, y, z)`
2. Rotate: `R(x, y, z)`
3. Scale: `S(x, y, z)`
4. Set scale: `SS(x, y, z)`
5. Subdivide scope: `Subdiv(num_divisions, axis, prod_rule_list)`
6. Push: `Push()`
7. Pop: `Pop()`
8. Place Shape: `PlaceShape("name_of_shape")`
9. At Depth: `AtDepth(depth, rule)`

An example grammar rule looks like this:

```
rule1: { T(1, 2, 3) R(45, 45, 0) PlaceShape("Cube") }
```

SGL also handles a random choice between many rules.

```
rule2:
(1){ T(1, 2, 3) R(45, 45, 0) PlaceShape("Cube") }
(1){ T(1, -2, -3) R(45, 0, 0) PlaceShape("Cylinder") }
```

5.2 SGL Architecture

SGL is implemented using Unity and C#. It has three main components: the parser, the driver and the editor. In this section I will discuss how each was implemented.

5.2.1 Parser

Although Unity streamlines the process of rendering and geometric transformation, it was difficult to find an appropriate parser generator. Unity expects that third party compilers be written in C++ and used as plugins, but SGL is an interpreter and needs direct access to the Unity API at parse-time. There is no parser generator native to Unity, so SGL uses a modified third party .NET package. [6]. Because the package was not designed for unity, the dependency for JSON serialization would not work and I had to edit the package to use one that is supported within the Unity framework.

The parser works by defining a set of rules for parsing SGL, and instantiating one of several rules classes based on what combination of characters it sees. The inline grammar and production rules are added to an operation queue. The production rule definitions are added to a dictionary to be referenced at build-time.

5.2.2 Driver

The driver calls various functions from the parser and serves as an intermediary between Unity gameObjects and the parser. All of the grammar rules for manipulating geometry are defined as lambda functions and attached to rule classes in the op-queue. Unity is designed for easy manipulation of transforms, but that means that there is not an easy way to save the state of a transform and revert back to it. To solve this, SGL uses a scope class with a position, quaternion rotation, and scale to hold the scope for each production rule. When placing an object, the placement rule references the scope of the rule that called it, turning the scope into the transform of a Unity GameObject.

5.2.3 Editor

SGL uses the Unity Editor library to run the parser and driver in the Unity Editor window. This is essentially a GUI overlay of the driver class with buttons to parse a text grammar file and generate a mesh and the list of primitives that the grammar file can access.

6 Results

Using SGL we can define a diverse range of non-trivial geometrical models in text. In this section, we show its use for generating trees, fractals and buildings.

6.1 Trees

The tree in Figure 10 is generated using a simple 'Y' branching structure, and a rotation around the local y-axis of $\pm 45^{\circ}$. Random structures are created by pruning the tree at each level about 10% of the time.

```
tree: { edge() branch() }

branch:
(1){
    T(0, 2, 0)
    Push() R(20, 0, 0) edge() R(0, 45, 0) branch() Pop()
    R(-20, 0, 0) edge() R(0, -45, 0) branch()
}
(.1){ S(1, 1, 1) }

edge: {
    Push() T(0, 1, 0) SS(.2, 2, .2) PlaceShape("Cube") Pop() }
```

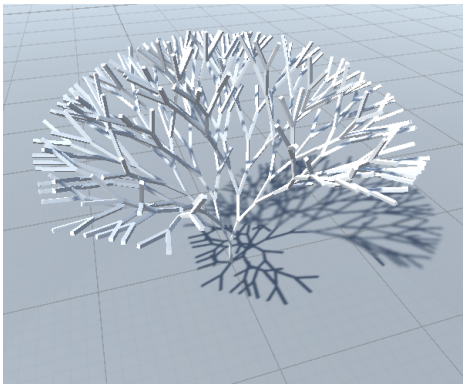


Figure 10: One variation generated by from the tree shape grammar.

6.2 Fractals

More complex fractals can be implemented using the subdivide grammar rule. Figure 11 is an example of the Menger sponge. Achieving this result using SGL requires the use of a 'null' rule which does not modify the geometry at all and serves as a placeholder for an operation in subdivide. `AtDepth` prevents the desired iteration of the sponge from being covered by a previous iteration.

```
start: { cx() }

cx:  { Subdiv(3, 0, sgprod{ cy(), cy1(), cy() }) }
cy:  { Subdiv(3, 1, sgprod{ cz1(), cz(), cz1() }) }
cy1: { Subdiv(3, 1, sgprod{ cz(), null(), cz() }) }
cz:  { Subdiv(3, 2, sgprod{ branch(), null(), branch() }) }
cz1: { Subdiv(3, 2, sgprod{ branch(), branch(), branch() })}
branch: { AtDepth(10, sgprod{ box() }) cx() }

box: { PlaceShape("Cube") }

null: { S(1, 1, 1) }
```

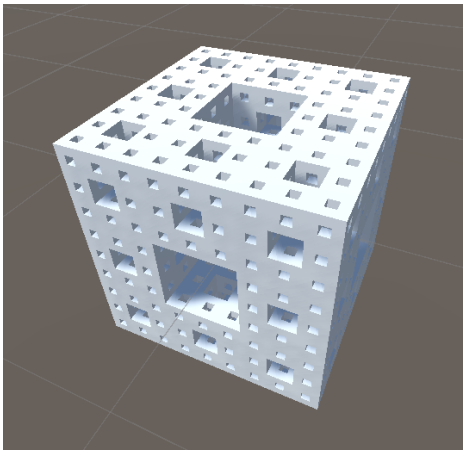


Figure 11: The Menger sponge fractal at three iterations. Generating geometry beyond this point gets very slow because of the exponential growth of the number of holes

6.3 Architecture

The purpose of SGL is to design buildings for the SPACES project. Using a set of pre-made meshes from the Unity asset store, I implemented a grammar to generate a randomized set of medieval houses (Figure 12). The grammar separates the house into sections and aligns those sections in a random order, starting with the front door. The grammar itself, is a long file positioning all of the meshes to form the sections, but the most essential part is as follows:

```
start: { p1() }

p1:
(1){ end_lg_d() T(0, 0, -2) long_hall_lg() T(0, 0, -2) p2() }
(1){ end_lg_d() T(0, 0, -2)
    T(2, 0, -.5) R(0, -90, 0) corner() T(0, 0, -2) p3() }
(1){ end_lg_d() T(0, 0, -2)
    T(1, 0, -.5) S(-1, 1, 1) T(1, 0, 0) R(0, -90, 0) corner()
    T(0, 0, -2) p3() }
(1){ end_sm_d() T(0, 0, -2) p3() }
(1){ end_sm_d() T(0, 0, -2)
    R(0, 180, 0) corner() R(0, -90, 0) T(0, 0, -2.5) p2() }
(1){ end_sm_d() T(0, 0, -2)
    S(-1, 1, 1) R(0, 180, 0) corner()
    R(0, -90, 0) T(0, 0, -2.5) p2() }

p2:
(1){ S(1, 1, -1) end_lg() }
(1){ long_hall_lg() T(0, 0, -2)
    T(2, 0, -.5) R(0, -90, 0) corner() T(0, 0, -2) p3() }
(1){ long_hall_lg() T(0, 0, -2)
    T(1, 0, -.5) S(-1, 1, 1) T(1, 0, 0) R(0, -90, 0) corner()
    T(0, 0, -2) p3() }

p3:
(1){ long_hall_sm() T(0, 0, -2) S(1, 1, -1) end_sm() }
(1){ S(1, 1, -1) end_sm() }
(1){ long_hall_sm() R(0, 180, 0) T(0, 0, 2)
    corner() R(0, -90, 0) T(0, 0, -2.5) p4() }
(1){ long_hall_sm() R(0, 180, 0) T(0, 0, 2) S(-1, 1, 1)
    corner() R(0, -90, 0) T(0, 0, -2.5) p4() }

p4:
(1){ long_hall_lg() T(0, 0, -2) S(1, 1, -1) end_lg() }
(1){ S(1, 1, -1) end_lg() }
```

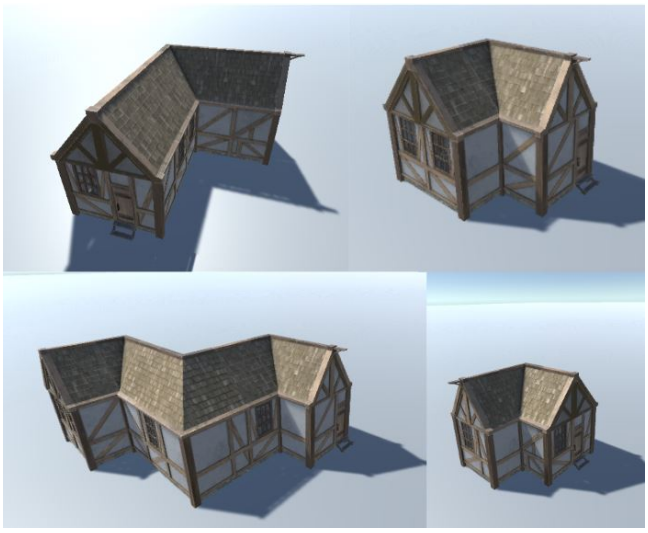


Figure 12: One of the variations produced by the medieval house grammar.

7 Future work

7.1 Genetic Search Algorithms

One extension to this project is to find different ways of producing a mesh from multiply defined production rules. Currently SGL uses the non-normalized probabilities to choose between two definitions for the same rule, but this can easily lead to undesired results if the grammar is not specifically defined. Especially for complex grammars, it is hard to foresee all of the possible meshes that could be produced. Thus, it is advantageous to find a smarter way of searching through the space of all choices for producing a mesh. One of these methods uses genetic algorithms. A genetic algorithm uses a cost function to evaluate a production based on a set of rules for how it should look, and changes its decisions for the next productions based on how the previous ones performed[**?**]. This is something that I will work on in the coming weeks as I wrap up this project.

7.2 Usability

The largest aspect of this project that needs work is the ease of use. Something that I have noticed when writing my grammars is that coming up with values for different translations and rotations is not intuitive. I know of a couple papers that have talked about making the user experience more adaptive by adding the ability to edit models after they have been 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] rolando95, "Rolando95/csharpparsergenerator," *GitHub*.

> https://github.com/rolando95/CSharpParserGenerator

[7] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

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.

[8] 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.

[9] ——, "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

[10] 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.