An Evaluation of Shape/Split Grammars for Architecture

Technical Report CS-2009-23

Jingyuan Huang, Alex Pytel, Cherry Zhang, Stephen Mann,
Elodie Fourquet, Marshall Hahn, Kate Kinnear, Michael Lam, and William Cowan

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, CANADA N2L 3G1

Abstract

Shape grammars have been used as a method for modeling buildings, both in architecture and in computer graphics. In this report, we survey some of the shape grammar papers, present some projects using shape grammars, and give our evaluation of shape grammars in computer graphics. In particular, we feel shape grammars are useful for generating a large variety of buildings, such as might be needed in computer games or animations, and that they could also be useful in design for exploring a large number of similar variations. However, we feel they are less useful for the design of new buildings or modeling of existing buildings.

1 Introduction

The idea of regular, parameterizable structure in architecture has been around for centuries [Alb86, PolBC]. These ideas became more explicit with the use of shape grammars for describing buildings [Sti80a]. More recently, shape grammars have been used in computer graphics for various aspects of architecture [WWSR03, MWH$^+$06, MZWG07, LWW08]. In Fall 2008, a seminar course in the David R Cheriton School of Computer Science at the University of Waterloo studied shape grammars and their uses in the computer graphics style of architecture. In this report, we summarize our findings. In particular, we were interested in the following questions:

- How useful were shape grammars for describing buildings?
- How hard were shape grammars to implement for architectural purposes?
- How hard was it to describe buildings using shape grammars?

With regards to the first of these questions, we determined four potential ways in which shape grammars might be useful:

1. As a means of abstraction for understanding the structure of a building.
2. As a way to parameterize several types of buildings that could then be used to semi-randomly generate cityscapes for computer games or animation.
3. As a way to describe a particular building or set of buildings.
4. As a way to design a new building.

We can consider the last two questions in comparison to using a standard CAD package for the same purpose.

The first of these potential uses for shape grammars (as a means of abstraction) is really a question for architects; i.e., as computer scientists, we are not qualified to answer this question, and instead note that this is a possible way in which shape grammars could be useful in architecture.

With regards to the other potential uses and the motivating questions, a summary of our findings are:

- Shape grammars are harder to implement than context-free grammars, mainly because you need to consider conditions as to whether or not a production can be applied, and you need to do some geometric computations to perform the split. However, a basic shape grammar still is straightforward to implement.
- Describing a building using a shape grammar directly is difficult. The difficulty lies in deciding where to represent structure, and in the tedious details of getting the geometry of the shape correct. Some of this difficulty can be handled with a visual editor [LWW08].
- Shape grammars do seem like a reasonable way to create parameterized models of several buildings that could then be used to generate a cityscape. The main problems here would be in generating a large enough database so as to have a diverse looking city, and to handle layout in a reasonable fashion. The latter issue has been addressed by several authors, although we only read one of these papers [dSM06]. Since we only read this one paper on the topic, we do not make any observations about these city layout algorithms in this technical report.
- The method does not seem like a reasonable method to describe particular buildings, either existing or ones being designed. In particular, a parametric solid modeler seems like a more reasonable way to design such buildings, and the "parametric" part of the solid modeler could better handle the parameters available in shape grammars. While a visual editor would help the shape grammars here, it seems likely that it would be reimplementing much of a parametric solid modeler.

The remainder of this report supports our findings. This report is based on reading the relevant literature, discussing the papers in class, and on our own implementations of shape grammars and modeling of several buildings using them. Section 3 gives a synopsis of the papers we read for the course. Then Sections 4 through 6 describe the projects implemented for the course. The discussion of an individual project is often written in first person singular; this use of the first person singular reflects the work and opinion of the individual who worked on a particular project. The name of who implemented each project appears in the section header for the project.

Much of what is written in this report is our opinion, and others may have different opinions (indeed, even within our group there was some dissension).

2 Background

In the works of Alberti and Vitruvius [Alb86, PolBC] we can see the beginnings of shape grammars. Both authors describe parameters and variations on architecture (with an emphasis on temples) that suggest a set of parameterized rules. These basic parameterizable rules continue to appear in architecture and eventually led to the shape grammars of Stiny [Sti80a] and others.

Later, the ideas of shape grammars were brought into computer graphics [WWSR03, MWH$^+$06]. The initial emphasis was on creating grammars that could generate a set of buildings for computer animation and games, although later work extended the ideas to visual editing in an attempt to allow the design of new buildings [LWW08] and to model buildings found in archaeological sites [MVW$^+$06].

Since shape grammars are built on top of context free grammars (CFG), we briefly review CFGs here. A *context-free grammar* consists of a finite set of symbols and a set of *productions* P mapping the symbols to strings of symbols. Often the symbols are separated into *variables* and *terminals*, where the latter are primitive symbols that are not mapped by the productions. In some variations, though, the variables may also act as terminals. Usually a special symbol is designated the *start symbol.*

As an example of a simple context-free grammar, consider $\{\{B, I\}, \{G, T, F\}, P, B\}$ where P consists of

$$\begin{array}{rcl} B & \rightarrow & GIT \\ I & \rightarrow & IF \\ I & \rightarrow & F \end{array}$$

and where $\{B, I\}$ is the set of variables, $\{G, T, F\}$ is the set of terminals, and B is the start symbol. Starting with B, we could apply the first production to map B to GIT, apply the second production to map GIT to $GIFT$, and then apply the third production to map $GIFT$ to $GFFT$.

A variety of questions remain. In the above example, there are two choices for mapping I; which one should we choose? And for using context-free grammars for architecture, how does the string $GFFT$

represent a building? For architecture, both of these questions are answered by noting that we use more than a context-free grammar. In particular, each symbol has some geometry associated with it, and each production has restrictions on when the production can be applied (based on the geometry of the symbol to be mapped) and each production has information about how the geometry on the symbol on the left-side of a production is mapped into geometry on the right-side of the production. These details boost the shape grammars used for architecture beyond context-free. Regardless, much of the ideas of a context-free grammar still apply, although perhaps the use of "shape expressions" (i.e., regular expressions extended to have geometric information) would be sufficient, since shape grammars seem to rarely use the power of context free grammars.

The symbols of a shape grammar usually refer to a particular part of the building. For example, in the above grammar, T refers to the top floor and G refers to the ground floor. For readability, typically the symbols are descriptive strings, rather than single characters.

After having added geometry, a rendering would look like a set of boxes. To make these boxes look like buildings usually requires heavy use of texture mapping. In many papers on shape grammars for buildings, the authors try to find a balance between using additional productions and symbols to represent more complex geometry, and using texture mapping to reduce the complexity of the geometry.

Many of the above ideas are illustrated in Section 4, where the above grammar is used to model some simple buildings.

In addition to the context free grammar symbols discussed above, some of the grammars in this paper use constructions that are more common to regular expressions. In particular, a '+' symbol indicates that the symbol preceding it appears one or more times; a '*' indicates that the preceding symbol appears 0 or more times; a '?' symbol means the preceding symbol occurs 0 or 1 time; and a | is an 'or', so either of the two symbols on either side of the | may be used.

3 Papers Discussed

We give here a synopsis of the shape grammar related papers we studied in class. We also considered meshing for architecture papers, but we felt that was outside the scope of this technical report and do not discuss these papers here. And while there are other procedural modeling techniques [Hav05], we decided to focus on those based on shape grammars.

3.1 Early papers—1980

We briefly reviewed a few of the 1980 Stiny papers, including "Production systems and grammars: a uniform characterization" [Sti80b], which covered context-free languages and similar concepts, and "Introduction to shape and shape grammars" [Sti80a], which introduced shape grammars. The latter paper is more relevant to the later shape grammar papers we studied, although the ideas in this Stiny paper are significantly different from what was eventually used the computer graphics.

3.2 Instant Architecture—2003

This paper [WWSR03] presents a system that uses shape grammars to create 3D models of buildings in an automated way. The authors develop a particular kind of shape grammar, called a *split grammar*, whose vocabulary consists of shapes that are edges of closed convex 3D objects. The grammar takes its name from the split operation, which is a type of production rule that decomposes a shape into a set of smaller shapes. A split grammar derivation can also apply a more general substitution operation, but the original shape must explicitly obey the constraint that its volume contains that of the replacement shapes.

The authors note that real world buildings are characterized by a coherence of shape elements (usually in the horizontal or vertical direction) and propose to control the derivation of split grammars to achieve a similar effect. This control is imposed in the form of shape attributes whose values determine the selection of derivation rules. What assigns these attributes to shapes is a context-free control grammar. The attribute matching used to interface split and control grammars also provides the mechanism for the automation of the entire system and the support for a large database of derivation rules.

The resulting system is capable of producing a large class of realistic buildings and is flexible in combining architectural elements to create a lot of variation. However, it has a weakness when placing facade elements on buildings composed of a large number of intersecting primitive volumes. First, it is difficult to write parameterized rules that will properly space and fit facade elements on differently-sized and shaped facade surfaces. Second, the facade surfaces must be produced by the split grammar, but it is restricted from producing general polygons. These problems are solved in the "Procedural Modeling of Buildings" paper with a special mass-modeling system.

3.3 Procedural Modeling of Buildings—2006

This paper [MWH$^+$06] is a follow-up paper to the Instant Architecture paper [WWSR03] that includes more details from the earlier paper: the mass modeling system and several types of modeling rules are defined. This paper also introduces some extensions that improve how facade elements are placed on more complex mass models.

One such extension is the occlusion query, which tests for intersections between shapes. This feature can be used to avoid placing a window on a shape that is partially occluded. Another possible usage is to test whether the street is visible from a shape to determine whether it should be replaced by a door.

Snap lines, which are lines that span across a facade while coinciding with the boundaries of more dominant features, are another important extension. This feature can be used to align facade elements across the volumetric shapes constituting a complex mass model. However, the user must specify manually which shapes should be considered dominant. As an example, snap lines could be generated from a door. The divisions produced by the split rules used to generate surrounding tiles can then be told to "snap" to these lines.

While these extensions may seem simple, they help produce more general shape grammars that can be applied to a variety of different mass models without any changes. The authors were able to produce some large, detailed, somewhat repetitive cities that are certainly good enough for use with video games. One drawback of their approach is that city generation does not occur in real-time.

3.4 Study on Computer-Aided Design Support of Traditional Architectural Theories—2004

Choo's dissertation [Cho04] emphasizes the importance of incorporating design principles into architectural CAD tools, especially those that contain shape grammars. These principles are important for supporting the creative and conceptual aspects of architectural design. In effect, architects primarily evaluate designs in terms of visual interest and support for functionality, and design principles offer assistance with these key elements.

The aesthetic demands of architecture consist of maintaining unity and avoiding chaos, while achieving harmony and variety. While this goal is unlikely to be within the reach of algorithms, there are many applicable heuristics. Traditional rules of thumb include use of axes, grids and symmetry, as well as proportion and rhythm. More modern designs achieve unity through similarity and can have partially ordered complexity. For example, Figure 1 shows three examples of symmetries in buildings with similar structures, with the bottom row showing axes of symmetries for the buildings. The design on the right is more appropriate as it provides unity and symmetry by having only a single center, in contrast with the middle design.

Figure 2 shows the impact of window and door sizes on the visual appearance of a building. The small windows of the design on the left give the building a "prison" look, where security is a greater concern than aesthetics; the over-large windows in the design in the middle has the appearance of a factory building, where light in the factory is a larger concern than aesthetics. The design on the right has balanced openings providing an aesthetic visual rhythm and a practical balance of sun and shelter. These recommendations are made in the context of vernacular architecture [CPSK08].

Having identified and summarized key elements of an ordering system, Choo then proposes a system that acts as a 'computer design assistant', where design rules are combined with shape grammars to help users achieve order and harmony in their designs. Indeed despite some of the design principles being obtainable with shape grammars with rules derived from existing buildings, the grammar alone cannot guarantee aesthetic results.

In effect, there is a trade-off between the complexity and versatility of a grammar and the predictability of the results it generates. Explicit consideration of design principles by shape grammar should yield more aesthetically pleasing results. To do so it is necessary to impose restrictions that limit the diversity

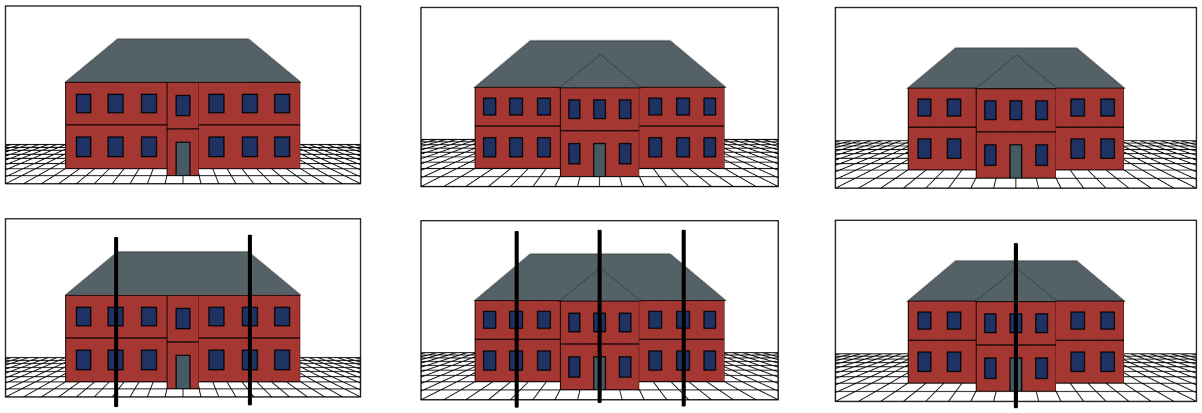


Figure 1: Examples of symmetry.

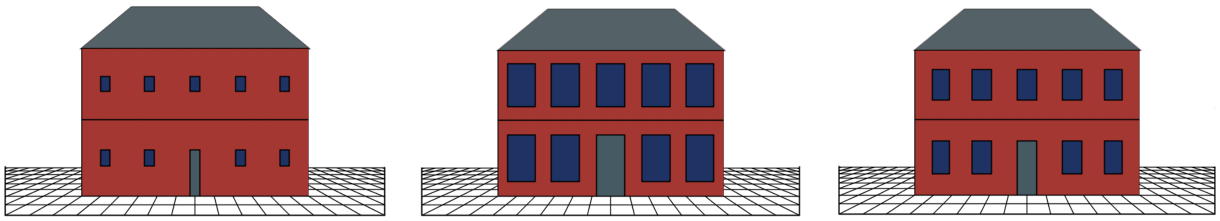


Figure 2: Proportion of openings.

of the models produced, which tends to increase the number of rules. One paper [MWH$^+$06] augments the complex grammar of Instant Architecture [WWSR03] to remedy designs that violate fundamental design principles, such as alignment and symmetry. Grammars used in computer graphics seem to be challenged in their spatial rules, torn between creating too much randomness in auto generated models and maintaining a condensed set of rules that respect major design principles.

3.5 Procedural 3D Reconstruction of Puuc Buildings in Xkipche—2006

Shape grammars and other methods have been used to model Roman architecture[1], ancient Chinese architecture [LHZB05], and Mayan buildings [MVW$^+$06]. The last of these papers is a case study of using shape grammars to model architectural sites. In this paper, Muller et al. [MVW$^+$06] applied a procedural reconstruction method based on shape grammar to Mayan archaeological sites in the Puuc area of Mexico. Puuc architecture contains distinctive features and follows regular patterns. By studying the building types and architectural parts, Muller et al. were able to summarize shape grammar rules for a small Mayan site in Xkipche and complete its reconstruction.

Muller et al. claimed that such a procedural reconstruction method is more efficient than traditional CAD modeling methods. However, from the times mentioned in their paper, it is time consuming to create a complete set of rules for all the details presented in an archaeological site, and those were the times spent by a computer scientist familiar with shape grammars and not the archaeologists. Someone familiar with a parametric solid modeling package such as SolidWorks should be able to produce similar models in similar or less time. While learning SolidWorks is likely more than an archaeologist wants to do for modeling these buildings, learning shape grammars is also more than an archaeologist would be willing to do.

It is also important to note that shape grammars are not the optimal reconstruction method for a completely accurate reconstruction. The advantage of shape grammars lie in that they abstract high-level understanding of the architecture and their ability in creating buildings for new sites even when some architectural information is missing.

3.6 Image-based Procedural Modeling of Facades—2007

Texture mapping is an application that is heavily utilized by shape grammars to model buildings. Unfortunately, it is difficult to obtain high quality texture maps of building facades for many reasons, including partial occlusion by other objects; shadows and other difficult lighting conditions; and distortions produced by the camera angle.

This paper [MZWG07] discusses taking various building facade images, identifying tiles within the images (i.e., distinct, repeatable sections), and combining the information from multiple sections to extract shape grammar rules. With a working shape grammar, production of a plausible 3D model of the building is then possible.

The technique relies heavily on the accurate ability of image processing techniques (mutual information) to extract the correct splitting rules from a facade. While in an optimal situation this technique can produce a template shape grammar that can then be tweaked to obtain a good representation of the original building, the method is also limited to facades with relatively regular structural elements. The tile extraction process can also be sensitive to various architectural occlusion, strong reflections, and shadows.

While the authors feel that

> the important aspect of our application is that we assist the user with the most challenging part of facade modeling: we can derive the exact dimensions, ratios and spacing of architectural elements automatically,

we find that this technique has some limited advantages in automatically determining ratios and relative spacing over traditional measurement techniques assuming that the correct splitting rules are generated. Since the accuracy of the resulting shape grammar is reliant on the ability to properly extract tile sections, this method becomes much less useful as image complexity increases, or image discernibility decreases. However, the technique still has merit in reconstructing simple building facades (i.e., facades that have regular, distinct tiles), as well as upgrading certain texture qualities such as lighting (for

[1]Rome Reborn, http://www.romereborn.virginia.edu

example, shadows or bump mapping), reflections, and resolution. These attributes are especially useful if our goal is to transfer a simple facade into a full 3D model, or to use the produced shape grammar to induce other changes such as adding or altering structural elements.

3.7 Interactive Visual Editing of Grammars for Procedural Architecture—2008

The authors present a visual editing paradigm for an existing shape grammar [LWW08]. The system allows an entire grammar rule-base to be created without text editing; rules can be created and edited visually. The system is interactive and updates the model immediately to reflect changes to the rule base.

This system also provides local control over aspects of the building. Objects can be grouped in semantically meaningful ways to allow the user to make changes easily and logically. These changes are also persistent; modifications are stored independently of the grammar, eliminating the often substantial increase in grammar rules that is normally caused by local modifications. This also means that local changes can be reapplied when the building is regenerated, or even saved and applied to an entirely new building.

Because the visual editing paradigm relies heavily on the results of changes being immediately visible, this paper places more emphasis on the performance of the building generation process than previous papers in this area.

3.8 Bill Cowan—Shape Grammars in Architectural Practice

If almost a decade old, Terry Knight's MIT web pages[2] are probably still the best introduction to the use of shape grammars in architecture. He discusses their use in the analysis, design and pedagogy of architecture, concluding with the discouraging comment,

> "Shape grammars are more than twenty-five years old, but their potential in education and practice is still far from being realized."

Of his many references, the majority use shape grammars not to design architecture, but to analyse existing designs. A typical comment, describing their potential for teaching architectural design is 'They [shape grammars] reveal the thoughtfulness, the "individual genius", behind designs that students might otherwise take as unfathomable.'

In contrast, computer graphics applications of shape grammars are primarily generative: a grammar constructs algorithmically the large amount of architectural detail in typical built environments, detail that must exist, but that rarely enters the focal vision of an observer. As an illustration of the contracts between architectural analysis and computer graphics generation, we viewed and discussed Sydney Pollack's documentary film on Frank Gehry [Pol06]. This discussion also created a bridge between the first part of CS 888, on shape grammars, and the second part, on computer-supported curve and surface design, where the role of the computer in architectural practice is constructive rather than analytical.

Many modern architects, Antoni Gaudi and Paolo Soleri being two early examples, have desired to use more organic forms, which are defined by highly-curved surfaces. Until recently, however, using organic forms required the solution of two hard problems: determining the mechanical strength of arbitrarily shaped building components, and manufacturing one-off building components, like curved beams, at a reasonable price. In the last few decades, the computer has solved both these problems. Finite elements analysis plus exhaustive simulation of static forces solved the problems of mechanical strength; computer-controlled manufacturing solved the problem of manufacturing irregularly-shaped components economically. These changes place the computer at the centre of architectural work flows.

There are many exponents of organic forms in architecture: Norman Foster and Frank Gehry are probably the most prominent. In preparation of studying how computer graphics supports the design of curved surfaces, we looked at the design practice of Frank Gehry, which combines traditional design methodology at the conceptual stage with extensively computerized processes for the design and manufacturing of individual components. The process is one observed often in computer graphics[3]; the artist,

[2] `http://www.mit.edu/~tknight/IJDC/`

[3] Examples include computer animation and game design.

designer or architect creates a maquette; the maquette is digitized by hand; the digitized model is cleaned up and refined, after which it is the basis for further digital processing.

The large, smoothly-curving surfaces of organic architecture are well-suited to this treatment. The well-understood algorithms and data-structures of smooth surface clean up the imperfections of the digitized maquette providing a well-defined geometry, for which it is relatively easy to analyse the forces it exerts on the structure that supports it. Large smoothly-curved surfaces require subsidiary detail to provide vertical rhythms that draw the gaze into the scene. Most often this detail is produced by resolving the surface into a set of facets. (Facets are easy to manufacture and assemble, and in the form of meshes are easy for a CAD system to support.) The regular pattern of the facets immediately suggests possible description by simple shape grammars. Doing so requires shape grammars to evolve beyond the planar surfaces and rectangular volumes on which they are currently supported. The right combination of organic form with controlled regularity, deployed on a two-dimensional manifold of low curvatures, is likely to be found in conformal mappings.

4 Stephen Mann—A Simple Implementation

To test how hard split grammars are to implement, I decided to implement a simple split grammar. I used this to model several simple box buildings, and then extended it to model a more complex building, the Davis Centre at the University of Waterloo.

4.1 Modeling Simple Buildings

Some aspects of buildings are straightforward to model with context free grammars. For example, to get a simple box building with floors, the following grammar suffices:

$$\begin{aligned} \text{Non-terminals} &= \{B, I\} \\ \text{Terminals} &= \{G, T, F\} \\ \text{Start symbol} &= B \end{aligned}$$

$$\begin{aligned} B &\to GIT \\ I &\to IF \\ I &\to F \end{aligned}$$

where G represents the ground floor, T the top floor, I are intermediate floors, and F is a particular intermediate floor (the ground and top floors need to be distinguished from the intermediate floors since the ground floor has doors, etc., that do not appear in intermediate floors and the top of a building is even more different than the intermediate floors).

But two questions immediately arise: How to control the number of floors generated by the grammar? And where does the geometry come in? Both questions are hinted at in [WWSR03], and made clear explicitly in [MWH$^+$06, MVW$^+$06].

The answer to both questions is that geometric attributes are associated with each symbol. In its simplest form, each symbol represents an axis aligned box (which can be specified by the coordinates of the corner with minimum value in each coordinate and by its width, depth, and height). A production then takes the geometry of the symbol on the left of the production and distributes its geometry to the symbols on the right of the production. The height is controlled by an additional constraint on the productions: a production can be applied only if a symbol meets certain geometric constraints. For example, letting each floor F be of height 1, the $I \to IF$ production can only be applied if the height of I is at least 2. Otherwise, the $I \to F$ production must be applied.

Thus, our productions are better represented as

$$\begin{aligned} {}^{h\geq 3}\mathbf{B}^{wdh}_{xyz} &\to \mathbf{G}^{w,d,1}_{x,y,z}\ \mathbf{I}^{w,d,h-2}_{x,y,z+1}\ \mathbf{T}^{w,d,1}_{x,y,z+h-1}, \\ {}^{h\geq 2}\mathbf{I}^{wdh}_{xyz} &\to \mathbf{F}^{w,d,1}_{x,y,z}\ \mathbf{I}^{w,d,h-1}_{x,y,z+1} \\ \mathbf{I}^{wdh}_{xyz} &\to \mathbf{F}^{w,d,1}_{x,y,z} \end{aligned}$$

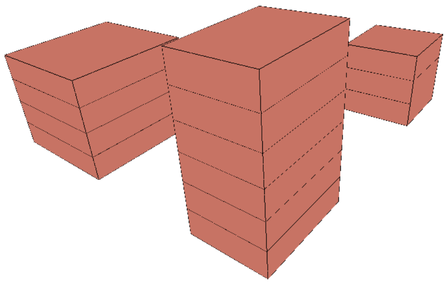


Figure 3: Simple buildings generated by a simple grammar.

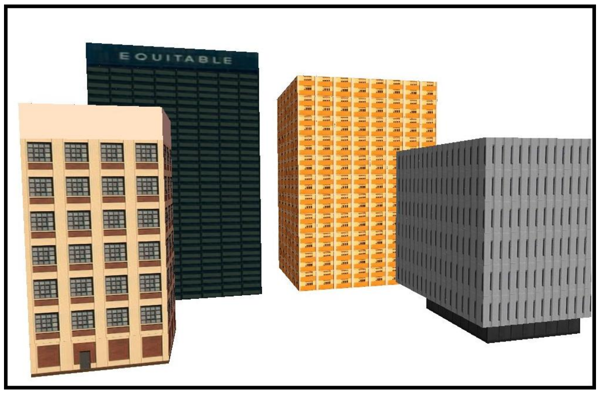


Figure 4: Four box buildings.

where the bold characters are the symbols of the production; the super- and sub-scripts of the symbol on the left represent its xyz-corner and its width, depth, and height. The condition on the left of the production is a requirement on the width, depth, and height of the symbol that must be met for the production to be applied. The super- and sub-scripts on the symbols to the right of the production tell how to distribute the geometry to each of the symbols on the right.

Using the above rules, we can generate buildings such as those shown in Figure 3 starting from $B^{5,5,3}_{0,0,0}$ $B^{3,3,4}_{10,10,0}$ and $B^{3,4,6}_{10,0,0}$. While the buildings are obvious too simple for most purposes, the appeal of the shape grammar is obvious: with little effort we can generate a broad variety of building-like shapes.

There are two ways to generate more realistic buildings: generate more complex geometry and texture mapping. A combination of the two is the ideal approach, and the question just becomes how much of each to do. In the interests of simplicity, I decided to lean heavily on texture maps, and generate just enough geometry to make texture mapping simple.

To this end, I took each of my intermediate floor (F) and ground floor symbols (G) and split them into cubes. I distinguished corner cells from non-corner cells, and generated one door symbol for each side of the ground floor. I then used four different sets of texture maps to generate the four buildings seen in Figure 4. The grammar used to generate these buildings appears in Appendix A.1. In all cases, the corner texture used was the same as the wall texture. And some buildings used the interior wall texture for all segments of the ground floor.

The textures were taken from four actual buildings. Some variations on the grammar were used to get variations on the shape. For example, Figure 5 shows the Dana Porter Library and shape grammar building generated from its textures. A few notes about this building and its textures are in order. First,

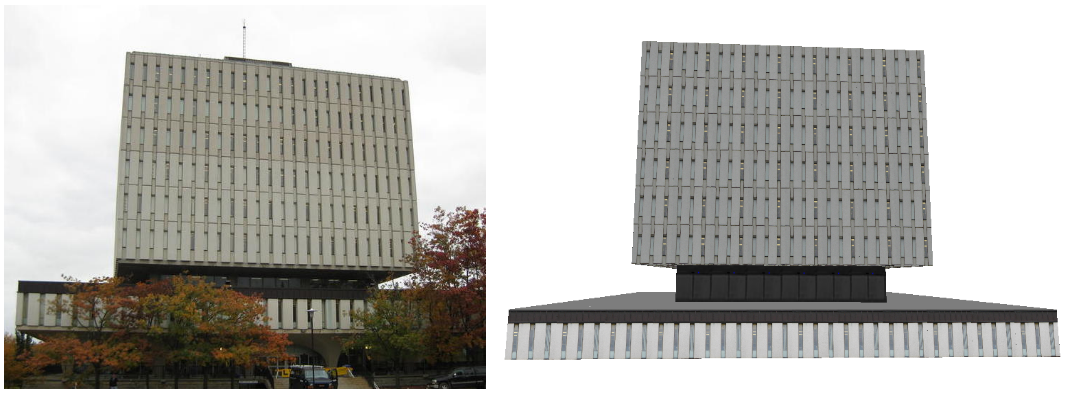


Figure 5: The Dana Porter library

the initial production rule that generates the ground floor geometry was modified from what appears in Appendix A.1 to

$$ {}^{w\geq 3\ d\geq 3\ h\geq 3}\mathbf{B}^{wdh}_{xyz} \rightarrow \mathbf{G}^{w-2,d-2,1}_{x+1,y+1,z}\ \mathbf{I}^{w,d,h-2}_{x,y,z+1}\ \mathbf{T}^{w,d,1}_{x,y,z+h-1}, $$

to create a ground floor that is smaller than the higher floors. And second, note that the grammar generated building is smaller than the actual library. This is one advantage of using this type of grammar for buildings: we can generate buildings of different sizes than the original. And finally, note that the actual Dana Porter Library has a non-flat roof and an additional lower floor that were not modeled with this shape grammar. The complete grammar for this model of the Dana Porter is in Appendix A.2.

4.2 Modeling a More Complex Building

The previous section gives some support for the idea that shape grammars could be used to generate simple downtowns without too much trouble. Clearly, grammars for additional types of buildings would be needed, grammars that would be more complex to model simple non-box features, as well as non-box geometry such as non-flat roofs.

Such buildings would all be of fairly simple geometry. As a second test of shape grammars, I decided to model a building with more complex geometry. Here, the goal is to make a reasonably accurate model of the building while keeping the shape grammar relatively simple. As a test building, I used the Davis Centre at the University of Waterloo (Figure 6, top).

As an initial production, I used the rule illustrated in Figure 7, which breaks the building up into seven pieces. The M symbol represents the main portion of the building, while the other six symbols represent exterior wings of the building. I later extended the first production to include symbols representing the exterior pipes on the building and the doors. The complete grammar appears in Appendix A.3. The shape grammar version of this building appears in Figure 6, middle.

A variety of notes are in order:

- The texture mapping on this building was poorly done. This is a result of having used a large texture for big blocks of the building. Had the shape grammar been extended to subdivide these blocks into smaller pieces, smaller textures could have been used to better effect.
- The symbol for the main part of the building (M) was evolved into a symbol for each floor, with the top floor getting a special symbol. The lower floors were then subdivided into smaller sections, and simple texture maps were used for each texture. The top level of the building was divided into an interior block and two exterior strips. The exterior strips were subdivided into smaller pieces, each of which was associated with a spline surface for its geometry to get the curved effect at the edge of the roof[4]. Initially, the center portion was left as a block and no attempt was made to

[4]Singly curved panels are rare in buildings as of 2008; while there are some curved panels in the Davis Centre, they are not in the portion I modeled using a spline surface, which instead was built as a set of flat pieces of glass.

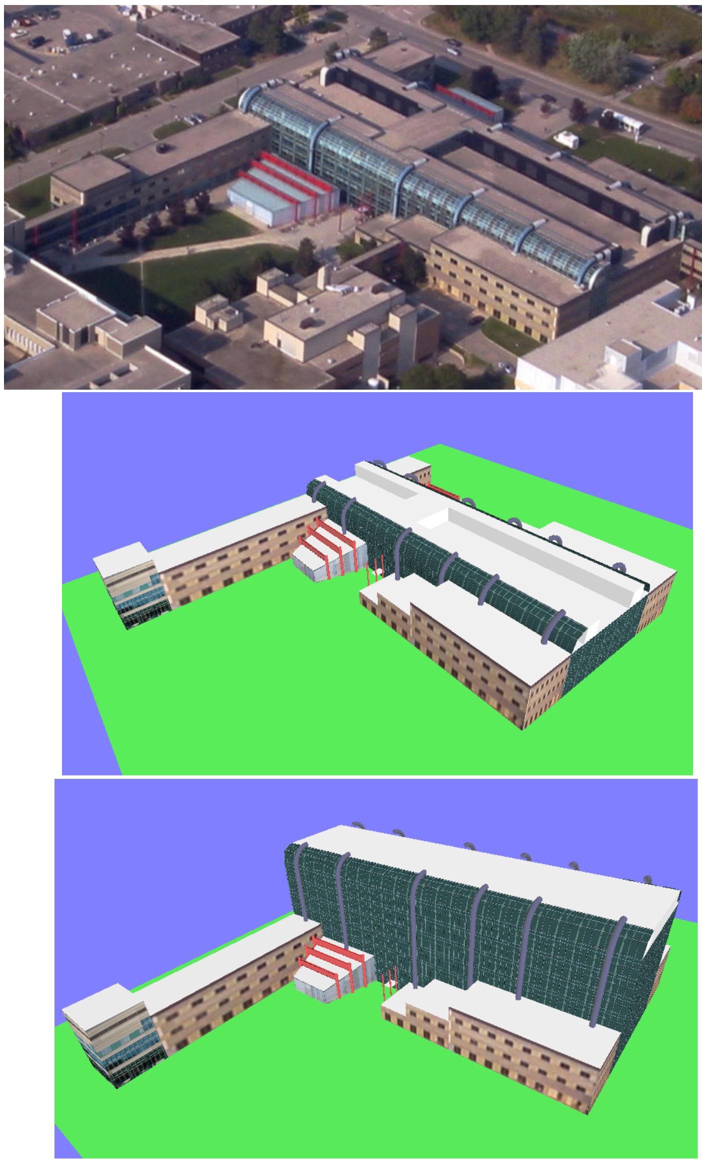


Figure 6: Top: The Davis Centre (photo courtesy of Jeff Orchard). Middle: Shape grammar rendering of Davis Centre. Bottom: A modified Davis Centre.

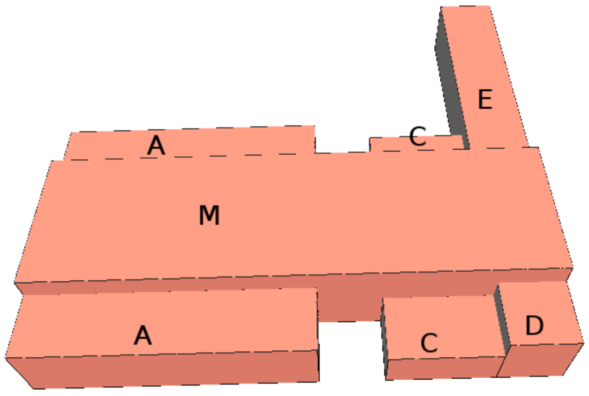


Figure 7: Davis Centre: $B \rightarrow MAACCDE$

model the interior of the roof. Later, some simple extensions were made to the grammar to model the roof; compare the roofs in Figure 6 bottom (no roof model) to Figure 6 middle (roof model).

- The pipes were hand positioned and each was modeled as a straight cylinder with a curved cylinder top piece (where the output was a quadrilateral approximation to the cylinder).
- The lectures halls (C) were modeled simply by creating the sloped box as the output, and later the three red, overhead cross beam structures were added. However, no attempt was made at modeling any parts of C using shape grammars.

Overall, the shape grammar created for this building was inadequate in the sense that only minor variations of the building could be created by merely adjusting parameters (see Figure 6, bottom, for an example of the Davis Centre with more floors; this image was made before some of the details of the top floor were added to the model). To a large part, this was because I found it difficult to model using shape grammars: it is far easier to texture map a big block than to break the block up into smaller pieces and texture map the smaller pieces. Further, I would have made a better model in less time had I used a parametric solid modeler, which would have also given me the capability to make more features parameterizable than I had in the model I created with a shape grammar.

4.3 Observations

- My grammars did not use the context-free aspects of a CFG—my grammars were really just regular expressions. An example of where it would be a more context-free grammar is to change the ground floor productions for the simple grammar of Appendix A.1 from

$$\begin{aligned} S &\rightarrow DU \\ U &\rightarrow sU \\ U &\rightarrow s \end{aligned}$$

  to

$$\begin{aligned} S &\rightarrow sSs \\ S &\rightarrow s \end{aligned}$$

  This would have placed the door in the middle of the wall.

- My intent was to make a shape grammar that was as close as possible to a context-free grammar. However, "repeat" constructions used in [WWSR03, MWH+06] papers are superior, since for modeling, you want to split a floor symbol into all the windows with a single production rather than through a sequence of productions as used by my grammar. In particular, splitting into all sections in one production makes distributing the geometry uniformly much easier.

- My grammars and the Instant Architecture grammars are context sensitive[5] since effectively you can get $A^nB^nC^n$. The context sensitive power comes from the geometry rules that are attached to the symbols.
- Initially, I only associated textures with terminal symbols. However, this results in having to replicate symbols for grammatically equal floor that have different textures. I therefore modified my code so that non-terminals symbols could have textures, and that these textures would be passed to the symbols in a production.
- Since I was trying to emulate CFG productions, I used single letter symbols in my grammars. With even moderately complex buildings, you start to run out of vaguely meaningful symbols. Clearly, strings are better than single letter symbols here, although possibly one could use fixed length strings of two or three characters.

4.4 Discussion

Based on a reading of many shape grammar for architecture papers, and on my own implementation and tests, I feel that shape grammars can be useful for generating cityscapes for use in computer animations and computer games. Their use for modeling particular buildings (whether for description or for design) is less clear. In my own experience in modeling one building (the Davis Centre), I found I would have preferred to use a parametric solid modeler instead of shape grammars.

Some of my concerns with using shape grammars for modeling specific buildings could likely be addressed by a good visual editor. However, at that point, the modeler would have reproduced many of the features of a parametric solid modeler and the shape grammar benefits would be less apparent. And one can imagine going the other way: adding support for architecture to an existing CAD package. Such extensions already exist: StructureWorks is an example of an architectural specific add-on to SolidWorks that appears to handle everything that shape grammars do.

One final comment: although I worked in the Davis Centre for 15 years, and I am well aware of its wings and lectures halls, building the shape grammar for the Davis Centre highlighted this structure to me more strongly. In this sense, there was an abstraction benefit to using the shape grammars, although it is unclear whether this understanding of the Davis Centre's structure was a result of shape grammars or just the process of modeling the building.

5 Jingyuan Huang—Inca Reconstruction Using Shape Grammar

For my project, I tried developing a shape grammar to model Incan buildings. Incan buildings have relatively simple and regular structures and it should be possible to describe such buildings using several shape grammar rules. The purpose of this project was to demonstrate the feasibility of using shape grammar as a tool to recreate simple Incan sites.

The system I implemented allows the user to load an image of an Incan site. The user then creates a *site plan* by locating buildings, walls, and stairs in the image using the mouse. The system assigns a default number of doors and windows to the walls, although the user can override these defaults. Once the building locations have been sketched, the system then uses the shape grammar described below to generate a 3D model of the buildings. Additionally, there is an "automatic" mode that will generate a random Incan village.

The following shape grammar rules are used to generate an Incan house.

[5]The usual definition of context sensitive languages does not allow for a reduction in the length of the string; I am using a more general sense of context sensitive that does allow for reduction of the length of the string, which makes the grammars Turing equivalent.

Non-terminals	=	{ *house, facades, front-facade, back-facade, side-facade* }
Terminals	=	{ ROOF, PILLAR, SHARP-WALL, WALL, WINDOW, DOOR }
Start symbol	=	*house*
house	→	*facades* ROOF
facades	→	*front-facade* | *back-facade* | *side-facade*
front-facade	→	PILLAR
front-facade	→	*regular-facade*
back-facade	→	*regular-facade*
side-facade	→	*regular-facade*
side-facade	→	*regular-facade* SHARP-WALL
regular-facade	→	WALL+
regular-facade	→	WALL (WINDOW | DOOR)* WALL
regular-facade	→	WALL WINDOW* DOOR WINDOW* WALL
regular-facade	→	WALL (WINDOW DOOR)+ WINDOW WALL

A set of parameters are associated with the house to determine its appearance. The parameters can be individually modified from GUI for each house. When the parameters specified by the user exceed the size of the house, the sizes of windows and doors are adjusted to maintain the desired number of windows and doors.

Only simple Incan walls are supported by the system. Incan walls with niches are not included. There is no shape grammar rule for the Incan walls, since their locations are quite ad hoc. Users must specify where to create those walls and they are not included in random generation.

When a site plan is provided, there is no rule for generating stairs and terraces. When a site is generated randomly, there are additional rules that govern the site's generation:

Non-terminal	=	{ *plan, left-terraces, stairs, right-terraces, house* }
Terminals	=	{ TERRACE, STAIR }
Start symbol	=	*plan*
plan	→	*left-terraces stairs right-terraces* (*stairs*?)
left-terraces	→	TERRACE TERRACE+
right-terraces	→	*terrace-with-house terrace-with-house terrace-with-house*+
terrace-with-house	→	TERRACE *house house*
terrace-with-house	→	TERRACE *house house house*
terrace-with-house	→	TERRACE *house house house house*
stairs	→	STAIR+

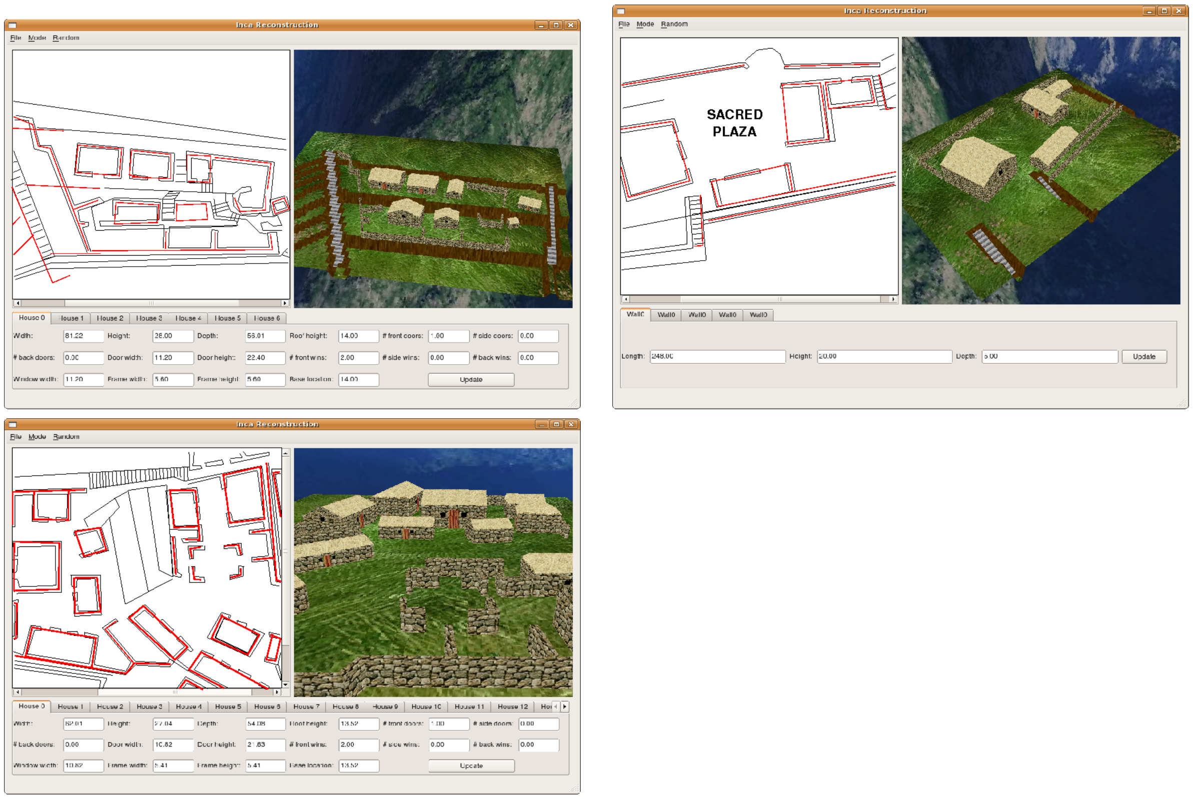
The choices of rules depend on probability. The location and the orientation of the stairs, the partitions of the terraces, the number of houses, and the orientation of the houses all depend on probability as well. Those rules are definitely not sufficient if we want to generate a realistic Incan site randomly, but they suffice as a demonstration of the idea. The rules above are all hard-coded, which imposes a limitation on the possible tasks that the system can handle.

Some results are shown in Figures 8 and 9. Figure 8 gives several reconstructed models based on site plans in Wright's book [WZ04]. To generate these models, the user only needs to select the lines that represent the different structures in the plan. A simple site reconstruction can be done within a minute or two, with some additional time to adjust the details of the houses. Note that these reconstructions are "as the site might have once appeared", since at the actual site, the walls are in ruins and the buildings no longer have roofs.

Figure 9 shows two sites randomly generated using the site grammar rules. These generated sites have one to two sets of stairs and several terraces. The houses on the terraces are roughly aligned but with some noise introduced by the random process. There is no user input required for the random generation.

5.1 Discussion

Shape grammars in computation are a specific class of production systems that generate geometric shapes. They are able to create a new design based on the architectural rules summarized by designers


Inca Reconstruction
File Mode Random
SACRED PLAZA
House 0 House 1 House 2 House 3 House 4 House 5 House 6
Wall0 Wall0 Wall0 Wall0 Wall0
House 0 House 1 House 2 House 3 House 4 House 5 House 6 House 7 House 8 House 9 House 10 House 11 House 12 Hou
Update

Width: 81.22 Height: 28.00 Depth: 56.01 Roof height: 14.00 # front doors: 1.00 # side doors: 0.00
# back doors: 0.00 Door width: 11.20 Door height: 22.40 # front wins: 2.00 # side wins: 0.00 # back wins: 0.00
Window width: 11.20 Frame width: 5.60 Frame height: 5.60 Base location: 14.00

Length: 248.00 Height: 20.00 Depth: 5.00

Width: 62.01 Height: 27.04 Depth: 54.08 Roof height: 13.52 # front doors: 1.00 # side doors: 0.00
# back doors: 0.00 Door width: 10.82 Door height: 21.63 # front wins: 2.00 # side wins: 0.00 # back wins: 0.00
Window width: 10.82 Frame width: 5.41 Frame height: 5.41 Base location: 13.52

Figure 8: User generated sites based on plans. In the window on the left in each image, black lines are from the site plan, while the red lines are drawn by the user.

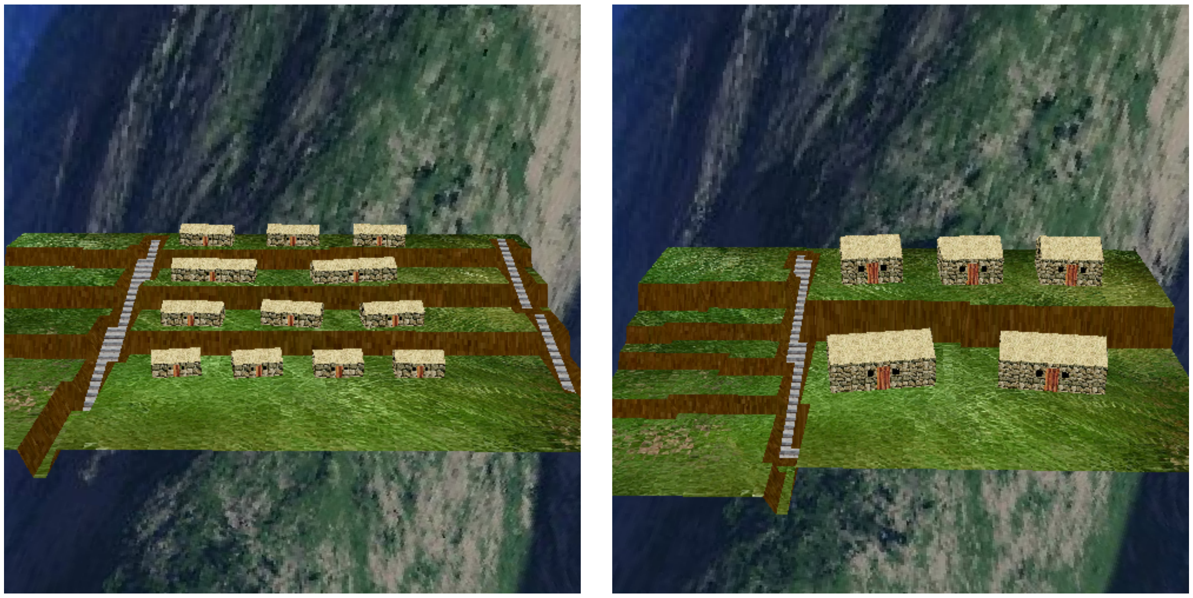


Figure 9: Examples of randomly generated plans.

and architects. In Xkipche's case [MVW$^+$06], the rules were summarized by archaeologists. Like the Puuc buildings in Xkipche, Inca buildings have consistent designs that make them suitable for shape grammar modeling.

Compared to Puuc buildings, Inca houses have less decorations and fewer components in the facades. The variation in the rules mainly comes from the different number of doors and windows. It is also possible to have different types of front and side facades and the roof form can change as well, which slightly increases the complexity of the rules. Inca architecture has two other main elements besides its houses: Incan walls and terraces. They play an important role in city construction therefore it is necessary to include them in the shape grammar rule base. However, since I did not have enough information on how the Incans placed the wall inside their villages or cities, I choose to let the user model the walls manually instead of devising a formal representation of the wall placement.

I believe that modeling using shape grammars is probably more time consuming compared to traditional modeling methods such as CAD. It is difficult to design rules for complicated shapes, and the results may or may not be satisfactory. For example, in the Xkipche paper, a large amount of additional rules were created for the decorations with repetitive patterns. When the pattern is less repetitive, shape grammar is not quite capable of modeling those decorations. The Inca buildings in this project have simple shapes, and it is easy to extract shape grammar rules. However, if we consider the time spent on building the system to interpret the rules, manual modeling can still outperform shape grammars.

One big advantage of shape grammars is the possibility of generating new designs. The random generation mode in this project tries to demonstrate this idea. If we encode our knowledge of city planning into shape grammar rules, then it is easy for someone without that domain knowledge to generate a sensible design using the system. This may not have any practical use in archaeology, but can be useful in modern urban planning.

6 Cherry Zhang— Shape Grammars for Room Layout

There are many existing applications for room planning[6]. These applications are based on "drag-and-drop" concepts, in which users have full control of the planning process. I studied some examples of split grammars in [WWSR03] and [MWH$^+$06]. In these papers, buildings are first considered as a whole, then they are subdivided into smaller components such as floors and facades, until an indivisible component, such as a window, is reached. I noticed that similar ideas can be used for room layouts.

The purpose of this project is to show that simple room layouts can be generated automatically using shape grammars. The concept of using shape grammars to generate room layouts can be applied to real-world applications such as interior design, games, dorm and hotel room design, and layout planning.

There are several reasons for using shape grammars for interior design:

Pattern: Room layouts have certain patterns in furniture arrangements. For example, a double-bed dorm always has two beds, two desks, two chairs, two closets and possibly other furniture. The beds are usually placed at the corners of the room. In a dining room, most people prefer to have the same number of chairs at each side of the table. These patterns allow us to write grammars or production rules to generate room layouts automatically.

Variety: If we incorporate randomness in choosing production rules, then the algorithm will help us produce various room layouts.

Speed: Drag-and-drop applications are more suitable for planning a small number of rooms because they require users to manually place each furniture. If we want to generate a large number of different room layouts, then using shape grammars is a better choice. Shape grammar based applications require minimal effort from users and rely on fast computer algorithms.

In this project, I derived my own shape grammars for room layouts based on the following concepts:

Split grammar: We start with a large object/shape. Then we split it into smaller objects/shapes according to some production rules. Splitting ends when we reach a terminal.

Attribute grammar: We use parameters, either specified by users or by the system itself, to control the selection of production rules.

[6]http://mydeco.com/rooms/planner/, http://www.homedesignersoftware.com/interior/

L-Systems: An L-System is recursive in nature because the same production rules can be applied again and again.

6.1 Problems with existing shape grammars

In this course, we studied several papers on generating buildings with shape grammars. We observed two major problems with existing shape grammars:

Complex Input: Most applications for generating buildings with shape grammars require the user to input complex shape grammars. Therefore, the users have to be technical enough to write shape grammars that can be parsed and compiled by the application.

Evaluation: Shape grammars can generate a large variety of buildings. Some results are unfeasible in real life. Therefore we need some evaluation schemes or algorithms to filter the results. We must also allow users to be able to participate in the filtering process.

In my application, I addressed the first issue by implementing simple grammars and parsers, targeted specifically for room layouts, all of which was placed behind a GUI. While this has the disadvantage of a lack of flexibility in the grammar, it means that the user does not need to learn shape grammars. Likewise, while my software filters out some of the infeasible layouts, the GUI provides a method for the user to select feasible layouts and reject infeasible ones.

6.2 Shape grammars for room layout

The fundamental part of the project is a new shape grammar that can be used to generate different room layouts. I used a set of nonterminal symbols {*bedroom, sleep_area, study_area*} to represent functional areas or spatial divisions of a room, and a set of terminals {*BED,CLOSET,CHAIR,DESK*} to represent furniture. I defined three rules for the new shape grammar: split, place_at_edge, and place_at_corner. Split divides the functional area that appears on the left hand side of the rule to two functional areas or pieces of furniture. place_at_edge and place_at_corner place furniture at the edge or corner of a functional area.

The new shape grammar used in this project resembles context-free grammars because it defines a set of nonterminals, terminals and rules. It resembles split grammars because split is the basic rule of this grammar. Compared to the existing shape grammars presented in other papers, the new shape grammar is simple. It contains smaller sets of symbols and rules that are easy to use.

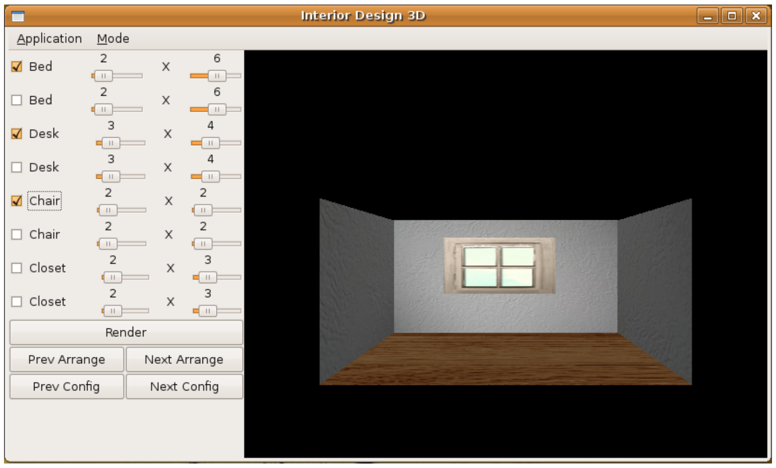
A GUI is provided to allow the user to select the furniture to place in the room (Figure 10). The software then generates a set of possible configurations of the selected furniture in the room, where the configurations meet various constraints. To view the these configurations, the users presses the “Next Arrange” and “Prev Arrange” buttons.

6.3 Examples

In general, the start symbol of the grammar is the room itself. The non-terminals are functional areas or spatial divisions of the room. The terminal symbols are the furniture. The productions start with a room, and subdivide it into smaller components and place the furniture within the room. Figure 11 illustrates the subdivision process used in this project.

6.3.1 Example 1: single-bed dorm layouts

The following shape grammars are used to generate single-bed dorm layouts:



☑	Bed	2	x	6
☐	Bed	2	x	6
☑	Desk	3	x	4
☐	Desk	3	x	4
☑	Chair	2	x	2
☐	Chair	2	x	2
☐	Closet	2	x	3
☐	Closet	2	x	3

Figure 10: The interface of the dorm layout application.

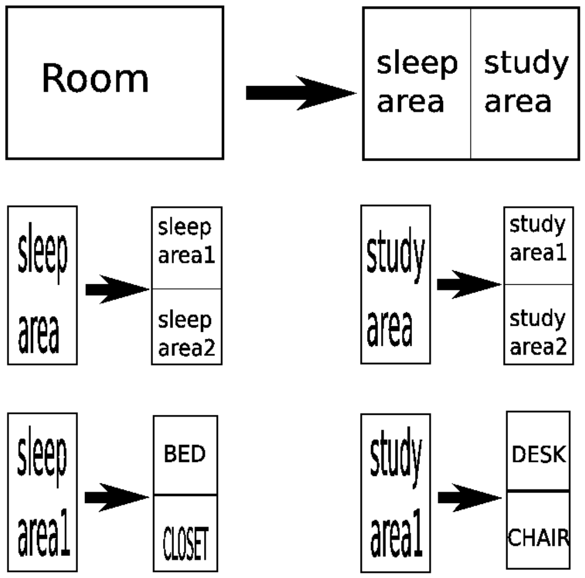


Figure 11: An example of the subdivision process using “split”.

Non-terminals = { *sleep-area, study-area* }
Terminals = { BED, DESK, CHAIR, CLOSET }
Start symbol = *bedroom*

bedroom → split {‘X’, 0.5, 0.5} *sleep-area* | *study-area*
bedroom → split {‘Y’, 0.5, 0.5} *sleep-area* | *study-area*
sleep-area → split {‘X’, 0.7, 0.3} BED | CLOSET
sleep-area → split {‘Y’, 0.7, 0.3} BED | CLOSET
study-area → split {‘X’, 0.2, 0.8} DESK | CHAIR
study-area → split {‘Y’, 0.2, 0.8} DESK | CHAIR

place-at-edge{*study-area*, DESK, CHAIR}
place-at-corner{*sleep-area*, BED}
place-at-corner{*sleep-area*, CLOSET}

Note that the last three constructs are external to the shape grammar. After parsing, each terminal symbol (i.e., BED, DESK, etc.) will end up in a non-terminal symbol (i.e., sleep_area, study_area, etc.). Graphically, non-terminal symbols are represented by areas that are bigger than terminal symbols. Therefore, there are multiple ways to place a terminal symbol within a non-terminal symbol. For example, if we want to place a BED in a sleep_area, we can put it at any one of the four corners, or along any edge of the area, or in the middle of the area. In real-life, people follow simple heuristics such as not putting their beds in the middle of the room or a particular area. As a result, after parsing the split grammar, each terminal symbol is placed in a non-terminal symbol according to simple heuristics people follow in real-life.

6.3.2 Example 2: double-bed dorm layouts

The following shape grammars are used to generate double-bed dorm layouts:

Start symbol = *bedroom*
Non-terminals = { *sleep-area, sleep-area-1, sleep-area-2, study-area, study-area-1, study-area-2*}
Terminals = { BED-1, BED-2, DESK-1, DESK-2, CHAIR-1, CHAIR-2, CLOSET-1, CLOSET-2}

bedroom → split {‘X’, 0.5, 0.5} *sleep-area* | *study-area*
bedroom → split {‘Y’, 0.5, 0.5} *sleep-area* | *study-area*
sleep-area → split {‘X’, 0.5, 0.5} *sleep-area-1* | *sleep-area-2*
sleep-area-1 → split {‘X’, 0.7, 0.3} BED-1 | CLOSET-1
sleep-area-1 → split {‘Y’, 0.7, 0.3} BED-1 | CLOSET-1
sleep-area-2 → split {‘X’, 0.7, 0.3} BED-2 | CLOSET-2
sleep-area-2 → split {‘Y’, 0.7, 0.3} BED-2 | CLOSET-2
study-area → split {‘X’, 0.5, 0.5} *study-area-1* | *study-area-2*
study-area → split {‘Y’, 0.5, 0.5} *study-area-1* | *study-area-2*
study-area-1 → split {‘X’, 0.2, 0.8} DESK-1 | CHAIR-1
study-area-1 → split {‘Y’, 0.2, 0.8} DESK-1 | CHAIR-1
study-area-2 → split {‘X’, 0.2, 0.8} DESK-2 | CHAIR-2
study-area-2 → split {‘Y’, 0.2, 0.8} DESK-2 | CHAIR-2

place-at-corner{*sleep-area-1*, BED-1, CLOSET-1}
place-at-corner{*sleep-area-2*, BED-2, CLOSET-2}
place-at-edge{*study-area-1*, DESK-1, CHAIR-1}
place-at-edge{*study-area-2*, DESK-2, CHAIR-2}

6.4 Implementation Details

To solve the complex input problem with the existing shape grammars, I implemented a *grammar generating* function in my application. Instead of shape grammars, users only have to specify furniture types, number of each type of furniture and optionally the dimension of each piece of furniture. The application

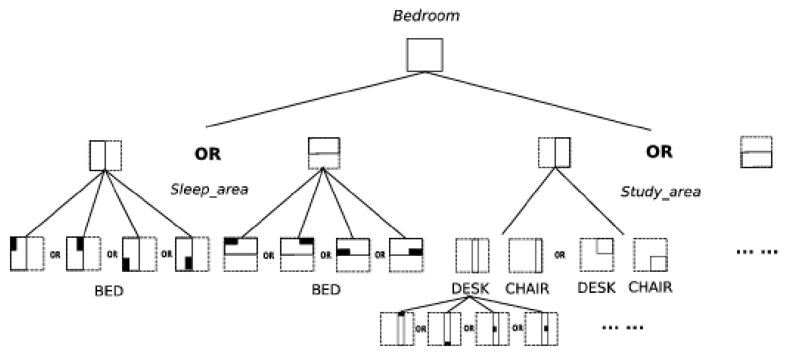


Figure 12: The configuration tree for spatial subdivision and furniture arrangements for a bedroom with one bed, one desk and one chair.

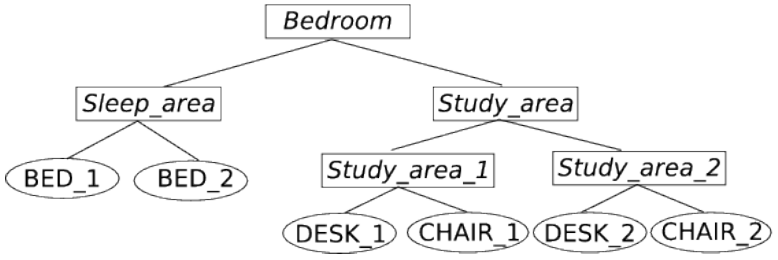


Figure 13: The parse tree for the shape grammars for generating a bedroom with two beds, two desks and two chairs.

generates a unique set of shape grammars automatically according to user inputs. For now the shape grammar generation is mainly hard-coded to cover all possible inputs.

To partially solve the evaluation issue, I used mixed-initiative design. For each set of shape grammars, the application generates a relatively large *configuration tree* (Figure 12), which stores a set of both feasible and infeasible layouts (an infeasible layout would be one that places the furniture in a manner that people never would, such as placing the desk in front of the bed or a chair in front of the door). The current application presents all the layouts to the users and lets them decide what they like.

The most important data structure in the application is a tree. There are three different types of nodes in the tree: *division nodes*, *level nodes*, and *leaf nodes*. Division nodes represent the functional spaces of a room (e.g., a sleep area or a study area); they are the non-terminals in the shape grammars. Leaf nodes represent furniture; they are the terminals in the shape grammars. Level nodes are parents of leaf nodes. They are mainly used to easily identify leaf nodes. Division nodes can be divided into other division nodes based on a split rule in the shape grammar. Leaf nodes (furniture) can only be arranged, not divided. All types of nodes store a set of possible configurations they have. A layout is a combination of different configurations of each leaf node. A graphical representation of the concept is shown in Figure 13.

The current application applies a limited number of rules in placing furniture. For example, beds and closets are always placed at the corner of the subspace they are in. Chairs are always placed in front of desks. These rules help reduce the number of infeasible layouts and keep the configuration tree relatively small.

Figure 14–18 show both some feasible and infeasible layouts of rooms constructed with the grammars in Example 1 and Example 2.

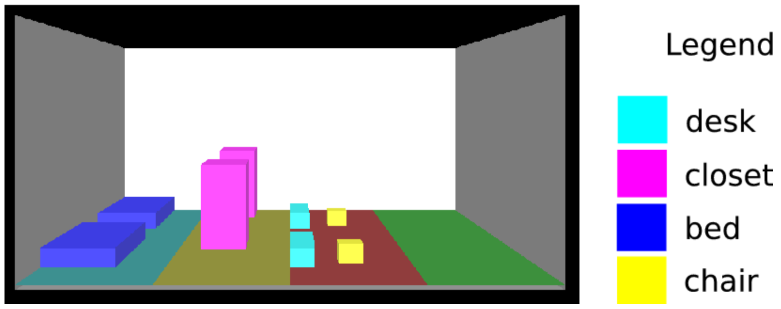


Figure 14: An example showing spatial subdivisions of functional areas.

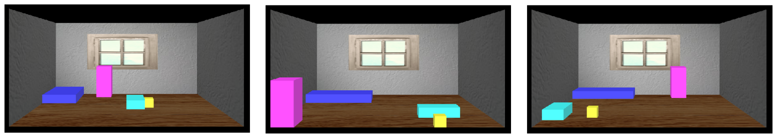


Figure 15: Some feasible layouts for a single-bed dorm. The shape grammar is shown in Example 1.

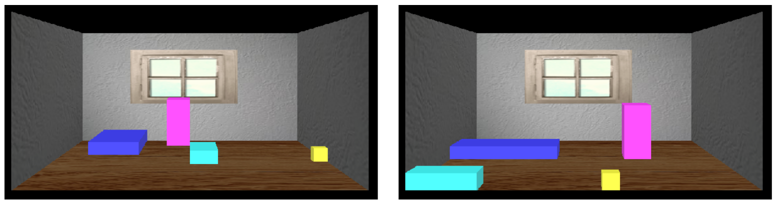


Figure 16: An infeasible layout for a single-bed dorm. The shape grammar is shown in Example 1.

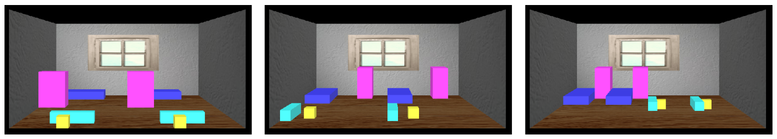


Figure 17: Some feasible layouts for a two-bed dorm. The shape grammar is shown in Example 2.

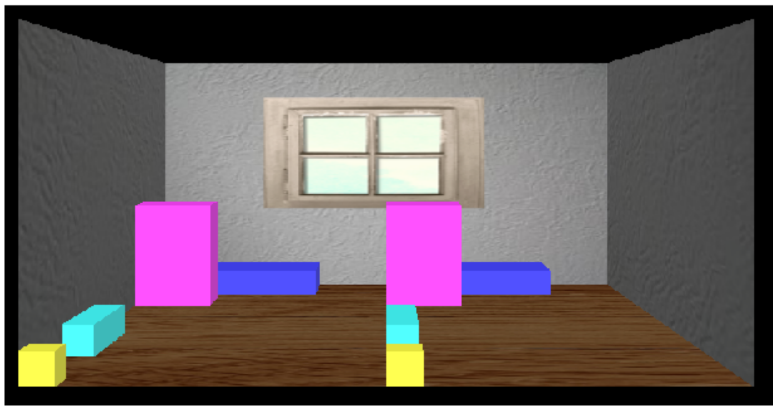


Figure 18: An infeasible layout for a two-bed dorm. The shape grammar is shown in Example 2.

6.5 Discussion

The most straightforward way to extend the application is to hard-code more rules for generating shape grammars based on user inputs. For now, the configuration tree is binary for division and level nodes. That is, I only allow split-to-two-parts operations for divisions. It should not be too hard to extend the binary tree to multi-children tree to support other split operations.

There are three ways for users to select their desired layout from all possible layouts. The first way, which my application uses, is to allow the user to browse through all possible layouts and explicitly select their desired one. A second way is *guided-search.* In guided-search, I present a few distinctive layouts and ask users to choose the one that is most similar to their desired layout. The application then searches through only the part of the configuration tree that contains the selected layout. The results of the search will be a set of layouts similar to the selected layout. In theory, guided-search should be faster and work better for large configuration trees. A third way for users to select their desired layout is to allow them to manipulate the arrangement of each functional space or furniture. This idea is similar to Lipp et al.'s [LWW08]. I think the third way defeats the purpose of having shape grammars. If an application relies too much on user interactions or gives too much freedom to the users, then it is more of a drag-and-drop application.

Some possible future work includes:

1. Better models for the furniture and rooms to make the application look graphically appealing.
2. Support more types of furniture and rooms. To add support for additional furniture, I can simply expand my parser to have more division and level nodes. The configuration tree will grow in depth. On the other hand, adding support for additional room types, such as dining rooms and living rooms, is difficult. For example, the way to divide the functional spaces of a living room or a dining room is different from that of a bedroom. Dining rooms usually have a table in the middle and chairs around the table. Split operations are not the best choice in dividing the dining room space. This means that to support additional room types, a new parser and division algorithm are needed.
3. Implement guided-search algorithms.
4. Extend configuration trees to support multiple children. This is useful if we want to group furniture together. For example, the sleep-area could consist of three children: bed, bedside table, and chair. Then instead of subdividing the sleep-area, the application can put all the three furniture into one area. The advantage of this approach is that the arrangement of furniture will appear to be less rigid due to the removal of split operations.

7 Alex Pytel—Shape Grammars and Recursive Construction

Shape grammars are a form of CFGs augmented with geometry rules and possibly other rules (such as a database of building facades). The need to control the derivation of a shape grammar motivates augmenting the grammars with control symbols and abstracting them to focus on their combinatoric properties. For instance, in their building modeling system Wonka et al. use the combination of split and control grammars only to derive the syntactic and spatial framework of the building; the geometric interpretation step is separate [WWSR03].

For my project, I implemented place grammars. A *place grammar* is a new formulation of a shape grammar that attempts to integrate three types of operations typically involved in modeling with shape grammars:

1. combinatoric composition of abstract features based on grammar derivation,
2. specification and placement (transformation) of concrete geometric features according to item 1,
3. control of grammar rule application with several types of constraints, such as those based on labels.

The placement of geometric features in item 2 takes place by transforming each new shape into the coordinate system of a parent shape. In this way, a place grammar operates on the 2D shell of a 3D object. The geometry created by each place grammar rule can, in principle, be an arbitrary mesh.

A place grammar consists of a set of *labels*, which correspond to the variables of a CFG, and *shape rules*, which correspond to the productions of a CFG. Syntactically, for my project I wrote the shape

```
(shape_rule
    (name af_split2v)
    (label afsplit)
    (%constants
        (s 0.20))

    (constraint height > 5)
    (constraint width > 5)
    (constraint aspect > 0.2)

    (output
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex 0 0 0)
            (vertex s 0 0)
            (vertex s 1 0)
            (vertex 0 1 0))
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex s 0 0)
            (vertex 1 0 0)
            (vertex 1 1 0)
            (vertex s 1 0))
    )

)
```

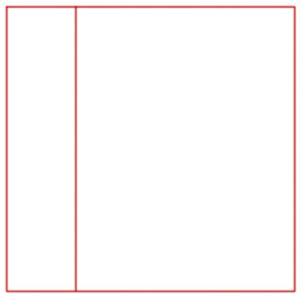


Figure 19: A place grammar shape rule. The output shape is shown on the right.

rules in a name-value pair structure, similar to Lisp. An example of a shape rule is given in Figure 19. This shape rule is part of a place grammar that produces an abstract pattern (Figure 20) on the side of a tower (Figure 21), which is also produced by a place grammar.

The example shows how place grammars integrate the three aspects of grammar-based modeling at the syntactic level. The shape rule explicitly includes an *output* section that specifies the geometry produced by the rule. In this case, the geometry consists of two rectangles side by side as shown on the right of the figure. They are specified using *vertex* commands with respect to a local coordinate system (LCS). When the shape rule is applied to a parent shape, the rule's output shapes will be transformed appropriately.

The shape rule will be applied to an existing shape that satisfies certain criteria. In this case, it must be labeled with the same *label* as the rule (*afsplit*) and its dimensions in the world coordinate system (WCS) have to meet each *constraint* clause. The specification of the two output rectangles shows how a parent shape's labels come about. Note that a shape's labels have a probabilistic weight associated with them, which can introduce a stochastic component into the process of rule application.

A shape rule has an optional *name* field that can be used to clarify what a rule does. For example, at the start of the entire grammar that the rule in Figure 19 is part of (see Appendix B), there is a comment indicating that 'af_split2v' is an "uneven vertical split".

The rule in Figure 19 corresponds to the context free grammar production

$$afsplit \quad \rightarrow \quad (afsplit \mid af\_z)\,(afsplit \mid af\_z).$$

By expanding to remove the |s we obtain four context free grammar productions:

$$\begin{aligned} afsplit &\rightarrow afsplit\ afsplit \\ afsplit &\rightarrow afsplit\ af\_z \\ afsplit &\rightarrow af\_z\ afsplit \\ afsplit &\rightarrow af\_z\ af\_z \end{aligned}$$

where the '*afsplit*' to the left of the arrows corresponds to the label of the shape rule (the third line of Figure 19), and the symbols on the right side of the arrow are the labels within the output field of the rule. The choice between the four rules is really two probabilistic choices within each *shape_spec*,

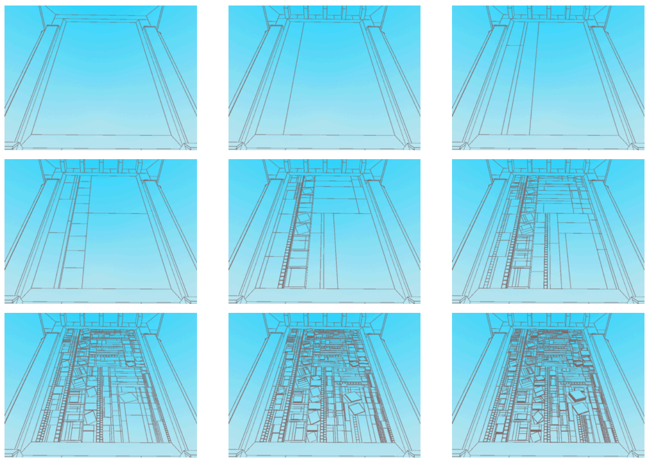


Figure 20: Developing details with a simple place grammar.

with each choice being between *afsplit* and *af_z*. For each choice, the probabilities are determined by the *weight* values of the labels.

The following is the full set of constructs and features that can be leveraged in the current place grammar implementation.

Labels: a set of objects, which can be attached to both shapes and rules; each label has a value (a string) and can have properties like priority, weight, and a uniqueness constraint.

Label Set Constraints: constraints that apply to all labels attached to a shape; for example, a constraint that stipulates path-based rule selection.

Shape Blueprint: a type of shape which is not tied to a coordinate system located on the shell of the derived object. These are the shapes that appear in the output section of rules.

Built Shape: a type of shape with a local coordinate system and WCS dimensions. The modelled object's shell is made up of shapes of this type.

Shape Rule: A rule consists of a single label, constraints on the input built shape (such as width, height, and aspect), special processing directives (for applying the rule multiple times in a horizontal or vertical direction), and a list of output shape blueprints.

Shape Subrule: a set of output shape blueprints manipulated as a group by a shape rule; for example, they can be reflected across a horizontal LCS axis.

Depth Hint: a special attribute placed on a shape blueprint. It provides a way for built shapes on different levels of the derivation to coordinate their depth (the amount they are extruded in the direction of the normal).

Center-and-Scale Hint: a special attribute placed on a shape blueprint. It effectively allows two built shapes on different levels of the derivation to be the same size in the x and y LCS directions. The required transformations of the LCS are performed by the shape rule when it is being applied.

The following are more detailed descriptions of how the components of the place grammar operate:

Rule Application: A rule is applied to a built shape when constraints on the built shape's dimensions are met and the rule and the built shape share a label. The shape rule uses the information in the built shape to transform and place each one of its output shape blueprints, which then become built shapes.

Rule Selection: Normal rule selection for a built shape proceeds by selecting one of its labels (respecting priority and weight), and then selecting a rule that also has that label. Alternatively, a cached rule selection can be used (path-based rule selection), and it is possible for a rule to be used precisely once on a derivation level.

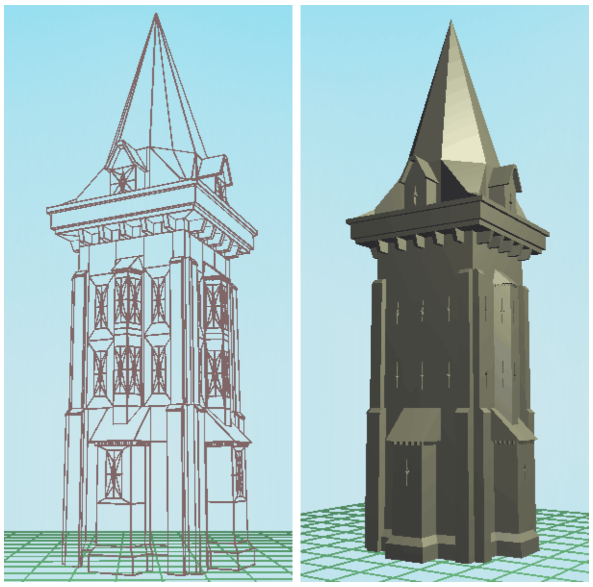


Figure 21: A building created with a place grammar.

Termination: A rule will not be selected for a built shape if it does not pass the constraints of any rule, it has a terminal label set constraint, its label set is empty, or there is no rule matching any label in the shape's label set. When no built shapes can be derived, the grammar derivation terminates globally.

Figure 21 shows an example building designed with a shape grammar. Some details on this grammar appear in Appendix B.

7.1 Discussion

Despite the apparent multitude of features, a place grammar is straightforward to use due to the unifying theme of placing a blueprint shape in respect to an LCS of a previously placed built shape. In the majority of cases, the inter-level attribute communication is not needed, which means that the grammar's specification consists mostly of the vertex coordinates of the shape blueprints. The label control system is particularly easy to use, since rules are only allowed a single label and any changes in the labels are a direct by-product of exchanging one shape for another during the derivation. In contrast, the system described by Wonka et al. [WWSR03] incorporated a stand-alone CFG control grammar with a sophisticated computationally expensive matching heuristic (not something one could trace by hand).

Place grammars can in principle be used to construct any 3D object (as a shell)—even the interior of a building. Indeed, the rule application of a place grammar only concerns itself with an LCS established by a previous built shape, which means that the derivation is not in any way limited by the current topology of the object and can easily change it.

Texturing could be considered a limitation of place grammars. While the components placed by each grammar rule could be trivially textured directly in the rule with explicit lists of texture coordinates (alongside the vertex coordinates in shape blueprints), that approach is tedious. A better way would be to texture parts of a whole level of derivation using coordinates generated based on the WCS position of the resulting built shapes. This is not straightforward.

There are some difficulties associated with the fact that all shapes are derived in parallel with little communication (only special attributes like hints or cached rule application, etc.) between the separate rule applications. While shapes occurring within the same rule can be modified as one conceptual unit to fix any problem with their relative arrangement, those produced by different rules generally cannot. This is an obstacle for operations such as removing T-intersections and beveling edges.

Place grammars lend themselves easily to a top-down component-based modeling approach, since the shape blueprints making up each rule are located in the same LCS (at (0, 0, 0)). This suggests an excellent way to interface a CAD package (with convenient geometry-shaping tools) with a place grammar: the CAD package could be used to create complicated geometry and the place grammar could transform and place it.

A place grammar for a building can also create its own floor plan (footprint) easily, a task that is usually treated separately. In a place grammar, the integration of this task is also attractive from a conceptual point of view: a place grammar has explicit access to the WCS coordinates of the currently derived object, which can allow it to easily perform occlusion queries (for example in the depth direction—away from the object).

8 Conclusions

Based on the papers we read and the projects we implemented, as a group we felt that shape grammars had some uses in architecture, although perhaps not as many as others might claim. In particular, shape grammars appear useful for generating a large variety of buildings, such as might be needed in computer games or animations. They could also be useful in design for exploring a large number of similar variations. We do feel that such shape grammars would be improved by explicitly embodying symmetry and other ideas such as those suggested by Choo [Cho04].

Overall we were less convinced of their usefulness in either design of new buildings or modeling of existing buildings. In each case, a parametric solid modeler seemed like a better choice. While some of the shape grammar papers commented that a parametric solid modeler is too complex for an archaeologist, an architect, etc., to learn for the modeling they do, we note that building shape grammars is no easier. For example, in the Puuc project, the shape grammars were designed by a computer scientist on-site at the archaeological excavation. Had a designer skilled with a parametric solid modeler been on-site, we are confident that models of similar or higher quality could have been built.

On a related topic, some of the papers on shape grammars discuss having a visual interface to the shape grammar to make it easier for someone who knows nothing about shape grammars to design with them. To us, it seemed like this was akin to extending the shape grammars to a solid modeler, which would be similar to reducing the functionality of a solid modeler to focus on the specific design task. Regardless, this did not seem to be an area of strength for shape grammars.

A Grammars for Simple Implementation

The following subsections give the grammars for some of the buildings discussed in Section 4.

A.1 Grammar for box buildings

Simplified grammar:

$$\begin{aligned}\text{Non-terminals} &= \{B, G, I, T, F, W, S, U\}\\ \text{Terminals} &= \{C, w, D\}\\ \text{Start symbol} &= B\end{aligned}$$

1. (B → GIT) (ground floor 2 stories)
2. B → GIT
3. I → FI
4. I → F
5. F → CWCWCWCW
6. W → wW (in x)
7. W → wW (in y)
8. W → w
9. G → cScScScS
10. S → DU (in x)
11. S → DU (in y)
12. U → sU (in x)
13. U → sU (in y)
14. U → s
15. S → s

Annotated productions for box buildings.

$$\begin{array}{rcl}
{}^{w\geq 3\ d\geq 3\ h\geq 7}\mathbf{B}^{wdh}_{xyz} & \rightarrow & \mathbf{G}^{w,d,2}_{x,y,z}\ \mathbf{I}^{w,d,h-3}_{x,y,z+2}\ \mathbf{T}^{w,d,1}_{x,y,z+h-1}, \\
{}^{w\geq 3\ d\geq 3\ h\geq 3}\mathbf{B}^{wdh}_{xyz} & \rightarrow & \mathbf{G}^{w,d,1}_{x,y,z}\ \mathbf{I}^{w,d,h-2}_{x,y,z+1}\ \mathbf{T}^{w,d,1}_{x,y,z+h-1}, \\
{}^{w\geq 3\ d\geq 3\ h\geq 2}\mathbf{I}^{wdh}_{xyz} & \rightarrow & \mathbf{F}^{w,d,1}_{x,y,z}\ \mathbf{I}^{w,d,h-1}_{x,y,z+1} \\
{}^{w\geq 3\ d\geq 3}\mathbf{I}^{wdh}_{xyz} & \rightarrow & \mathbf{F}^{w,d,1}_{x,y,z} \\
 & & \\
{}^{w\geq 2\ d\geq 2}\mathbf{F}^{wdh}_{xyz} & \rightarrow & \mathbf{C}^{1,1,h}_{x,y,z}\ \mathbf{W}^{w-2,1,1}_{x+1,y,z}\ \mathbf{C}^{1,1,1}_{x+w-1,y,z}\ \mathbf{W}^{1,d-2,1}_{x+w-1,y+1,z} \\
 & & \mathbf{C}^{1,1,1}_{x+w-1,y+d-1,z}\ \mathbf{W}^{w-2,1,1}_{x+1,y+d-1,z}\ \mathbf{C}^{1,1,1}_{x,y+d-1,z}\ \mathbf{W}^{1,d-2,1}_{x,y+1,z} \\
{}^{w\geq 2}\mathbf{W}^{wdh}_{xyz} & \rightarrow & \mathbf{w}^{1,1,1}_{x,y,z}\ \mathbf{W}^{w-1,1,1}_{x+1,y,z} \\
{}^{d\geq 2}\mathbf{W}^{wdh}_{xyz} & \rightarrow & \mathbf{w}^{1,1,1}_{x,y,z}\ \mathbf{W}^{1,d-1,1}_{x,y+1,z} \\
\mathbf{W}^{wdh}_{xyz} & \rightarrow & \mathbf{w}^{1,1,1}_{x,y,z} \\
{}^{w\geq 2\ d\geq 2}\mathbf{G}^{wdh}_{xyz} & \rightarrow & \mathbf{c}^{1,1,h}_{x,y,z}\ \mathbf{S}^{w-2,1,1}_{x+1,y,z}\ \mathbf{c}^{1,1,1}_{x+w-1,y,z}\ \mathbf{S}^{1,d-2,1}_{x+w-1,y+1,z} \\
 & & \mathbf{c}^{1,1,1}_{x+w-1,y+d-1,z}\ \mathbf{S}^{w-2,1,1}_{x+1,y+d-1,z}\ \mathbf{c}^{1,1,1}_{x,y+d-1,z}\ \mathbf{S}^{1,d-2,1}_{x,y+1,z} \\
{}^{w\geq 2}\mathbf{S}^{wdh}_{xyz} & \rightarrow & \mathbf{d}^{1,1,1}_{x,y,z}\ \mathbf{U}^{w-1,1,1}_{x+1,y,z} \\
{}^{w\geq 2}\mathbf{U}^{wdh}_{xyz} & \rightarrow & \mathbf{s}^{1,1,1}_{x,y,z}\ \mathbf{U}^{w-1,1,1}_{x+1,y,z} \\
{}^{d\geq 2}\mathbf{S}^{wdh}_{xyz} & \rightarrow & \mathbf{d}^{1,1,1}_{x,y,z}\ \mathbf{U}^{1,d-1,1}_{x,y+1,z} \\
{}^{d\geq 2}\mathbf{U}^{wdh}_{xyz} & \rightarrow & \mathbf{s}^{1,1,1}_{x,y,z}\ \mathbf{U}^{1,d-1,1}_{x,y+1,z} \\
\mathbf{S}^{wdh}_{xyz} & \rightarrow & \mathbf{s}^{1,1,1}_{x,y,z} \\
\mathbf{U}^{wdh}_{xyz} & \rightarrow & \mathbf{s}^{1,1,1}_{x,y,z}
\end{array}$$

w wall texture; **C** corner texture; **T** top floor texture **s** ground floor texture; **d** door texture; **c** ground floor corner;

A.2 Dana Porter Grammar

Since the floors of a box building are the same (unless there is a door or some other such feature), I wrote a simpler grammar for the Dana Porter Library. The annotation of the grammar is similar to the box building, except in the first production the lower floors have varying sizes and offsets. Also, in this grammar, the texture is associated with a symbol in the first production and inherited by the child symbols.

$$\begin{array}{rcl}
\text{Non-terminals} & = & \{B, F, I\} \\
\text{Terminals} & = & \{G, C, w\} \\
\text{Start symbol} & = & B
\end{array}$$

$$\begin{array}{rcl}
B & \rightarrow & F\ F\ G\ F\ I \\
I & \rightarrow & F\ I \\
I & \rightarrow & F \\
F & \rightarrow & C\ W\ C\ W\ C\ W\ C\ W \\
W & \rightarrow & w\ W \\
W & \rightarrow & w\ W \\
W & \rightarrow & w
\end{array}$$

Note that 'G' (the "ground" floor) is a single, large block, unlike 'F', which gets divided into separate walls (W).

The annotated productions for the Dana Porter Library are

$$\begin{array}{rcl}
{}^{w\geq 8\ d\geq 8\ h\geq 3}\mathbf{B}^{wdh}_{xyz} & \rightarrow & \mathbf{F}^{w-2,d-2,1}_{x+1,y+1,z}\ \mathbf{F}^{w,d,1}_{x,y,z+1}\ \mathbf{G}^{w-2,d-2,1}_{x+1,y+1,z+1}\ \mathbf{F}^{w-6,d-6,1}_{x+3,y+3,z+2}\ \mathbf{I}^{w-4,d-4,h-2}_{x+2,y+2,z+3} \\
{}^{w\geq 3\ d\geq 3\ h\geq 2}\mathbf{I}^{wdh}_{xyz} & \rightarrow & \mathbf{F}^{w,d,1}_{x,y,z}\ \mathbf{I}^{w,d,h-1}_{x,y,z+1} \\
{}^{w\geq 3\ d\geq 3}\mathbf{I}^{wdh}_{xyz} & \rightarrow & \mathbf{F}^{w,d,1}_{x,y,z} \\
{}^{w\geq 2\ d\geq 2}\mathbf{F}^{wdh}_{xyz} & \rightarrow & \mathbf{C}^{1,1,h}_{x,y,z}\ \mathbf{W}^{w-2,1,1}_{x+1,y,z}\ \mathbf{C}^{1,1,1}_{x+w-1,y,z}\ \mathbf{W}^{1,d-2,1}_{x+w-1,y+1,z} \\
 & & \mathbf{C}^{1,1,1}_{x+w-1,y+d-1,z}\ \mathbf{W}^{w-2,1,1}_{x+1,y+d-1,z}\ \mathbf{C}^{1,1,1}_{x,y+d-1,z}\ \mathbf{W}^{1,d-2,1}_{x,y+1,z} \\
{}^{w\geq 2}\mathbf{W}^{wdh}_{xyz} & \rightarrow & \mathbf{w}^{1,1,1}_{x,y,z}\ \mathbf{W}^{w-1,1,1}_{x+1,y,z} \\
{}^{d\geq 2}\mathbf{W}^{wdh}_{xyz} & \rightarrow & \mathbf{w}^{1,1,1}_{x,y,z}\ \mathbf{W}^{1,d-1,1}_{x,y+1,z} \\
\mathbf{W}^{wdh}_{xyz} & \rightarrow & \mathbf{w}^{1,1,1}_{x,y,z}
\end{array}$$

There are two "floors" (**F** and **G**) at height height $z+1$ because each floor is split into four walls (**W**), and each wall is split into a linear array of boxes (**w**). If the difference in depth between adjacent floors is greater than 1 (the size of **w**), then a gap appears between floors. Thus, the two floor symbols at the same depth are there to fill this gap. This is an artifact of the way my production rules work (i.e., by splitting walls into boxes of size 1).

A.3 Grammar for Davis Centre

The grammar for the Davis Centre is more complex than that of the simple box buildings and that of the Dana Porter Library. This complexity arises primarily because there are more individual components (wings, pipes, entrances, roof details) than in the other buildings. However, many of the subcomponents are terminal symbols generated in the first production. The geometry of these terminal symbols is generated by C code, or (in the case of wings) just treated as a large box to texture map. A production system that refined the geometry of these terminal symbols would be much larger.

$$\begin{array}{rcl}
{}^{w\geq 3\ d\geq 3\ h\geq 3}\mathbf{B}^{wdh}_{xyz} & \rightarrow & \mathbf{M}^{24,7,4}_{x,y+4,z}\ \mathbf{A}^{13,3,3}_{x,y+1,z}\ \mathbf{A}^{13,3,3}_{x,y+11,z}\ \mathbf{C}^{5,4,2}_{x+16,y,z}\ \mathbf{C}^{5,4,2}_{x+16,y+11,z}\ \mathbf{D}^{3,4,3}_{x+21,y,z}\ \mathbf{E}^{3,13,3}_{x+21,y+11,z} \\
 & & \mathbf{p}^{1,1,3}_{x+1,y+3,z}\ \mathbf{p}^{1,1,3}_{x+5,y+3,z}\ \mathbf{p}^{1,1,3}_{x+8,y+3,z}\ \mathbf{p}^{1,1,3}_{x+12,y+3,z}\ \mathbf{p}^{1,1,3}_{x+18,y+3,z}\ \mathbf{p}^{1,1,3}_{x+22,y+3,z} \\
 & & \mathbf{q}^{1,1,3}_{x+1,y+11,z}\ \mathbf{q}^{1,1,3}_{x+5,y+11,z}\ \mathbf{q}^{1,1,3}_{x+8,y+11,z}\ \mathbf{q}^{1,1,3}_{x+12,y+11,z}\ \mathbf{q}^{1,1,3}_{x+18,y+11,z}\ \mathbf{q}^{1,1,3}_{x+22,y+11,z} \\
 & & \mathbf{d}^{1,2,1}_{x+14,y+2,z}\ \mathbf{d}^{1,2,1}_{x+14,y+11,z} \\
\mathbf{A}^{wdh}_{xyz} & \rightarrow & \mathbf{a}^{8,3,3}_{x,y,z}\ \mathbf{b}^{3,3,2}_{x+8,y,z}\ \mathbf{c}^{2,3,1}_{x+11,y,z} \\
\mathbf{E}^{wdh}_{xyz} & \rightarrow & \mathbf{e}^{3,13,3}_{x,y,z}\ \mathbf{f}^{3,3,4}_{x,y+13,z} \\
{}^{w\geq 3\ d\geq 3\ h\geq 3}\mathbf{M}^{wdh}_{xyz} & \rightarrow & \mathbf{I}^{w,d,h-1}_{x,y,z}\ \mathbf{T}^{w,d,1}_{x,y,z+h-1}, \\
{}^{w\geq 3\ d\geq 3\ h\geq 2}\mathbf{I}^{wdh}_{xyz} & \rightarrow & \mathbf{F}^{w,d,1}_{x,y,z}\ \mathbf{I}^{w,d,h-1}_{x,y,z+1}
\end{array}$$

$$
\begin{array}{rcl}
{}^{w\geq 3\ d\geq 3}\mathbf{I}^{wdh}_{xyz} & \rightarrow & \mathbf{F}^{w,d,1}_{x,y,z} \\
{}^{w\geq 2\ d\geq 2}\mathbf{F}^{wdh}_{xyz} & \rightarrow & \mathbf{H}^{1,1,h}_{x,y,z}\ \mathbf{W}^{w-2,1,1}_{x+1,y,z}\ \mathbf{H}^{1,1,1}_{x+w-1,y,z}\ \mathbf{W}^{1,d-2,1}_{x+w-1,y+1,z} \\
 & & \mathbf{H}^{1,1,1}_{x+w-1,y+d-1,z}\ \mathbf{W}^{w-2,1,1}_{x+1,y+d-1,z}\ \mathbf{H}^{1,1,1}_{x,y+d-1,z}\ \mathbf{W}^{1,d-2,1}_{x,y+1,z} \\
{}^{w\geq 2}\mathbf{W}^{wdh}_{xyz} & \rightarrow & \mathbf{w}^{1,1,1}_{x,y,z}\ \mathbf{W}^{w-1,1,1}_{x+1,y,z} \\
{}^{d\geq 2}\mathbf{W}^{wdh}_{xyz} & \rightarrow & \mathbf{w}^{1,1,1}_{x,y,z}\ \mathbf{W}^{1,d-1,1}_{x,y+1,z} \\
\mathbf{W}^{wdh}_{xyz} & \rightarrow & \mathbf{w}^{1,1,1}_{x,y,z} \\
\mathbf{T}^{wdh}_{xyz} & \rightarrow & \mathbf{s}^{1,1,1}_{x,y,z}\ \mathbf{t}^{w-2,1,1}_{x+1,y,z}\ \mathbf{u}^{1,1,1}_{x+w-1,y,z}\ \mathbf{v}^{1,1,1}_{x+w-1,y+d-1,z}\ \mathbf{x}^{w-2,1,1}_{x+1,y+d-1,z}\ \mathbf{y}^{1,1,1}_{x,y+d-1,z} \\
 & & \mathbf{m}^{w,1,1}_{x,y+1,z}\ \mathbf{m}^{w,1,1}_{x,y+d-2,z}\ \mathbf{n}^{w,d-4,1}_{x,y+2,z}\ \mathbf{m}^{3,d-4,1}_{x+13,y+2,z} \\
{}^{w\geq 2}\mathbf{t}^{wdh}_{xyz} & \rightarrow & \mathbf{t}^{1,1,1}_{x,y,z}\ \mathbf{t}^{w-1,1,1}_{x+1,y,z} \\
{}^{w\geq 2}\mathbf{x}^{wdh}_{xyz} & \rightarrow & \mathbf{x}^{1,1,1}_{x,y,z}\ \mathbf{x}^{w-1,1,1}_{x+1,y,z}
\end{array}
$$

Symbol	Function
B	The entire DC
M	The main body of the DC
A	The three level wings of the DC
C	The lecture halls of the DC
D	The small wing of the DC
E	The long wing that leads to math and has a food court
p q	The large pipes on the outside of the DC
D	The entrances to the DC
a b c	The three sections of **A**
e f	The two sections of **E**
T	The top floor of the main building
I	The interior floors of the main building
F	A single interior floors of the main building
H	A corner of an interior floor
W	A side of an interior floor
w	A single non-corner block of an interior floor
s u v y	A corner of the top floor
t x	A long side of the top floor
m n	Interior blocks of the top floor

B Place Grammar Example

In general, a shape grammar (as discussed in Section 7) may combine a small number of elements to produce a large variety of simple buildings or a few highly detailed ones. However, this property of compact representation is only combinatoric in nature: a grammar may be small in respect to the number of elements and associated rules of combining them, but the elements themselves can have long descriptions. For this reason, place grammars tend to be long, as they include concrete vertex by vertex encodings of the elements' geometry. Table 1 gives some statistics concerning the tower building shown in Figure 21 (the whole grammar is too long to include here).

On the other hand the grammar for creating the geometric pattern shown in Figure 20 operates on simple geometry and, therefore, is not completely overwhelmed with vertex statements. Figures 23–27 show an annotated specification of the grammar.

Figure 22 illustrates this place grammar. There are four classes of rules in this grammar: splitting rules, extrusion rules (of type 1 or 2) and repeating rules. In this figure, the contents of a box show the possible set of rules that may be applicable if the geometric constraints are satisfied. Each edge is labeled with the rule that maps it to another box. The derivation continues until none of the output shapes satisfy dimension constraints (some terminal shapes are also produced, but they are not shown).

As mentioned in the previous paragraph, the rules are part of a set of rules that are conceptually grouped as splitting rules, based on the type of operation they perform. This conceptual grouping arises

Total lines (without multi-line comments)	996	100%
Vertex statements	454	45.6%
Lone ')' and empty lines	152	15.3%
Rules	12	
Meta commands (e.g. constants)	62	

Table 1: A count of entities in the tower place grammar. The top of the table contains single line entities and their contribution to the total; the bottom contains a count of interesting multiline constructs.

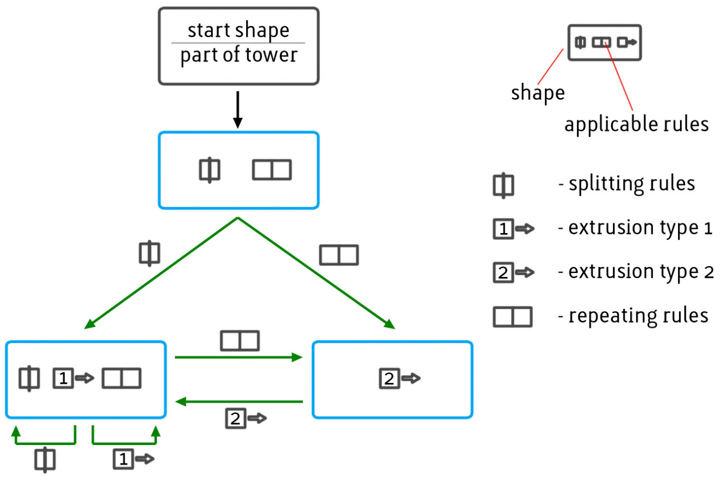


Figure 22: A conceptual representation of possible derivation paths of the place grammar from Appendix B.

from both the rules' geometry and their defining labels. In other words, rule application, production of concrete geometry, and derivation control are all tightly related. For this reason, Figure 22 is not just a scheme of rule application, but also an illustration of the intended role of the rules in constructing a set of objects with specific properties (abstract patterns in this case).

References

[Alb86] Leon Battista Alberti. *The Ten Books of Architecture: The 1755 Leoni Edition.* Dover, 1986.

[Cho04] Seung Yeon Choo. *Study on Computer-Aided Design Support of Traditional Architectural Theories.* PhD thesis, Technological University of Munich, January 2004.

[CPSK08] Marianne Cusato, Ben Pentreath, Richard Sammons, and Leon Krier. *Get Your House Right: Architectural Elements to Use and Avoid.* Sterling, 2008.

[dSM06] Luiz Gonzaga da Silveira and Soraia Raupp Musse. Real-time generation of populated virtual cities. In *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 155–164, New York, NY, USA, 2006. ACM.

[Hav05] S. Havemann. *Generative mesh modeling.* PhD thesis, Technische Universität Braunschweig, Germany, 2005.

[LHZB05] H Liu, W Hua, D Zhou, and H Bao. Building chinese ancient architectures in seconds. In V.S Sunderam, G.D. van Albada, P.M.A. Sloot, and J.J. Dongarra, editors, *ICCS 2005*, volume 3515, pages 248–255. Springer, 2005.

[LWW08] Markus Lipp, Peter Wonka, and Michael Wimmer. Interactive visual editing of grammars for procedural architecture. *ACM Transacations on Graphics*, 21(3), 2008.

[MVW$^+$06] Pascal Müller, Tijl Vereenooghe, Peter Wonka, Iken Paap, and Luc Van Gool. Procedural 3d reconstruction of Puuc buildings in Xkipcé. In *Eurographics Symposium on Virtual Reality, Archaeology and Cultural Heritage (VAST)*, pages 139–146, 2006.

[MWH$^+$06] Pascal Mueller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Transacations on Graphics*, 25(3):614–623, 2006.

[MZWG07] Pascal Mueller, Gang Zeng, Peter Wonka, and Luc Van Gool. Image-based procedural modeling of facades. *ACM Transactions on Graphics*, 26(3):1–9, 2007.

[Pol06] Sydney Pollack. Sketches of frank gehry. Sony Pictures Classics DVD, 2006.

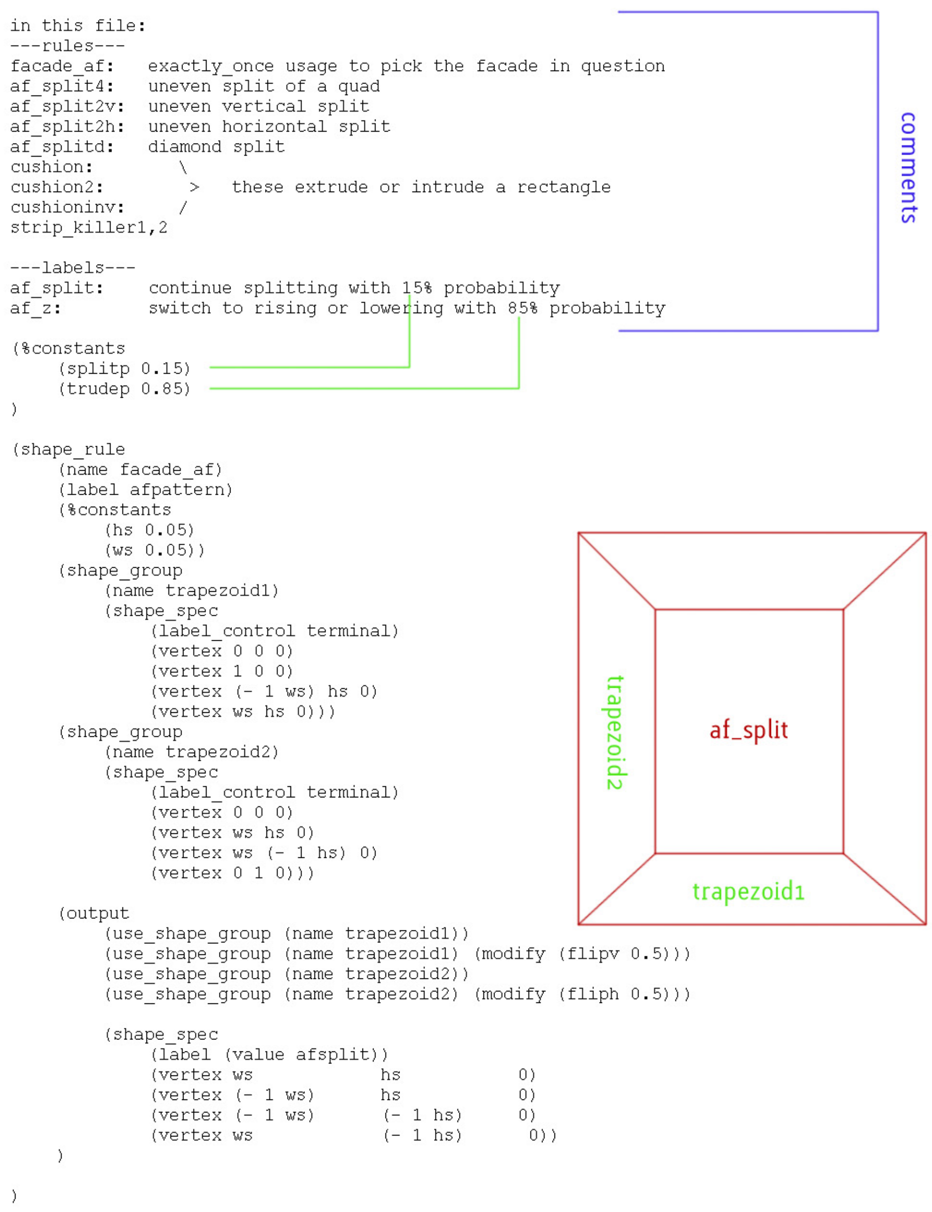
[PolBC] Vitruvius Pollio. *The Ten Books on Architecture.* 15 BC.

[Sti80a] G Stiny. Introduction to shape and shape gramars. In *Environment and Planning B 7*, pages 343–361. 1980.

[Sti80b] G Stiny. Production systems and grammars: a uniform characterization. In *Environment and Planning B 7*, pages 313–314. 1980.

[WWSR03] Peter Wonka, Michael Wimmer, Francois Sillion, and William Ribarsky. Instant architecture. *ACM Transacations on Graphics*, 21(3):669–677, 2003.

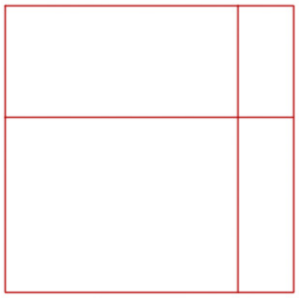
[WZ04] Ruth M. Wright and Alfredo Valencia Zegarra. *The Machu Picchu Guidebook: A Self-Guided Tour.* Johnson Books, 2004.



```
in this file:
---rules---
facade_af:   exactly_once usage to pick the facade in question
af_split4:   uneven split of a quad
af_split2v:  uneven vertical split
af_split2h:  uneven horizontal split
af_splitd:   diamond split
cushion:        \
cushion2:        >   these extrude or intrude a rectangle
cushioninv:     /
strip_killer1,2

---labels---
af_split:    continue splitting with 15% probability
af_z:        switch to rising or lowering with 85% probability

(%constants
    (splitp 0.15)
    (trudep 0.85)
)

(shape_rule
    (name facade_af)
    (label afpattern)
    (%constants
        (hs 0.05)
        (ws 0.05))
    (shape_group
        (name trapezoid1)
        (shape_spec
            (label_control terminal)
            (vertex 0 0 0)
            (vertex 1 0 0)
            (vertex (- 1 ws) hs 0)
            (vertex ws hs 0)))
    (shape_group
        (name trapezoid2)
        (shape_spec
            (label_control terminal)
            (vertex 0 0 0)
            (vertex ws hs 0)
            (vertex ws (- 1 hs) 0)
            (vertex 0 1 0)))

    (output
        (use_shape_group (name trapezoid1))
        (use_shape_group (name trapezoid1) (modify (flipv 0.5)))
        (use_shape_group (name trapezoid2))
        (use_shape_group (name trapezoid2) (modify (fliph 0.5)))

        (shape_spec
            (label (value afsplit))
            (vertex ws          hs          0)
            (vertex (- 1 ws)    hs          0)
            (vertex (- 1 ws)    (- 1 hs)    0)
            (vertex ws          (- 1 hs)     0))
    )

)
```

Figure 23: Place grammar page 1.

```
(shape_rule
    (name af_split4)
    (label afsplit)
    (%constants
        (ws 0.80)
        (hs 0.65))

    (constraint height > 5)    input can not
    (constraint width > 5)      be too small

    (output
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex 0 0 0)
            (vertex ws 0 0)
            (vertex ws hs 0)
            (vertex 0 hs 0))
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex ws 0 0)
            (vertex 1 0 0)
            (vertex 1 hs 0)
            (vertex ws hs 0))
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex ws hs 0)
            (vertex 1 hs 0)
            (vertex 1 1 0)
            (vertex ws 1 0))
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex 0 hs 0)
            (vertex ws hs 0)
            (vertex ws 1 0)
            (vertex 0 1 0))
    )

)
(shape_rule
    (name af_split2v)
    (label afsplit)
    (%constants
        (s 0.20))

    (constraint height > 5)
    (constraint width > 5)
    (constraint aspect > 0.2)—— not too tall

    (output
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex 0 0 0)
            (vertex s 0 0)
            (vertex s 1 0)
            (vertex 0 1 0))
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex s 0 0)
            (vertex 1 0 0)
            (vertex 1 1 0)
            (vertex s 1 0))
    )

)
```



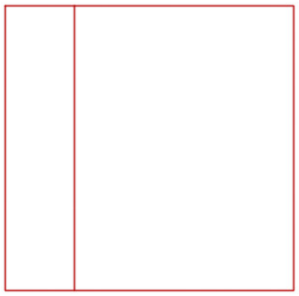


Figure 24: Place grammar page 2.

```
(shape_rule
    (name af_split2h)
    (label afsplit)
    (%constants
        (s 0.50))

    (constraint height > 5)
    (constraint width > 5)
    (constraint aspect < 5)

    (output
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex 0 0 0)
            (vertex 1 0 0)
            (vertex 1 s 0)
            (vertex 0 s 0))
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex 0 s 0)
            (vertex 1 s 0)
            (vertex 1 1 0)
            (vertex 0 1 0))
    )

)

(shape_rule
    (name af_splitd)
    (label afsplit)
    (%constants
        (s 0.30)
        (b 0.70))

    (constraint height < 40)
    (constraint width < 40)
    (constraint height > 10)
    (constraint width > 10)
    (constraint aspect < 1.5)
    (constraint aspect > 0.7)

    (output
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex s 0 0)
            (vertex 1 s 0)
            (vertex b 1 0)
            (vertex 0 b 0))
        (shape_spec
            (label_control terminal)
            (vertex 0 0 0)
            (vertex s 0 0)
            (vertex 0 b 0))
        (shape_spec
            (label_control terminal)
            (vertex s 0 0)
            (vertex 1 0 0)
            (vertex 1 s 0))
        (shape_spec
            (label_control terminal)
            (vertex 1 s 0)
            (vertex 1 1 0)
            (vertex b 1 0))
        (shape_spec
            (label_control terminal)
            (vertex b 1 0)
            (vertex 0 1 0)
            (vertex 0 b 0))
    )

)
```

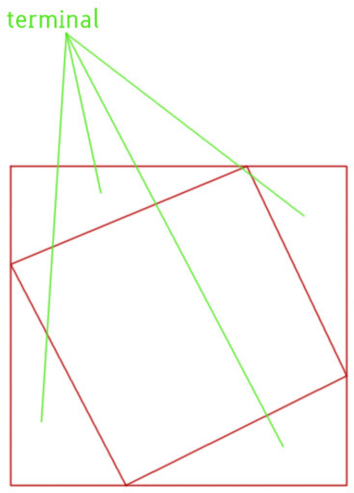


Figure 25: Place grammar page 3.

```
(shape_rule
     (name strip_killer1)
     (label afsplit)
     (repeat right -5)

     (constraint aspect > 10)

     (output
          (shape_spec
               (label (value af_z2))
               (vertex 0 0 0)
               (vertex 1 0 0)
               (vertex 1 1 0)
               (vertex 0 1 0))
     )

)
```

these two rules break up strips that are too tall or too wide, resulting in rows of similar shapes

```
(shape_rule
     (name strip_killer2)
     (label afsplit)
     (repeat up -5)

     (constraint aspect < 0.1)

     (output
          (shape_spec
               (label (value af_z2))
               (vertex 0 0 0)
               (vertex 1 0 0)
               (vertex 1 1 0)
               (vertex 0 1 0))
     )

)
```

```
(shape_rule
     (name cushion)
     (label af_z)
     (%constants
          (hh 0.05)
     (constraint height < 15)
     (constraint width < 15)
     (constraint height > 5)
     (constraint width > 5)

     (output
          (shape_spec
               (label (value afsplit) (weight splitp))
               (label (value af_z) (weight trudep))
               (vertex 0 0 hh)
               (vertex 1 0 hh)
               (vertex 1 1 hh)
               (vertex 0 1 hh))
          (shape_spec
               (label_control terminal)
               (vertex 0 0 0)
               (vertex 0 0 hh)
               (vertex 0 1 hh)
               (vertex 0 1 0))
          (shape_spec
               (label_control terminal)
               (vertex 1 0 hh)
               (vertex 1 0 0)
               (vertex 1 1 0)
               (vertex 1 1 hh))
          (shape_spec
               (label_control terminal)
               (vertex 0 1 hh)
               (vertex 1 1 hh)
               (vertex 1 1 0)
               (vertex 0 1 0))
          (shape_spec
               (label_control terminal)
               (vertex 0 0 0)
               (vertex 1 0 0)
               (vertex 1 0 hh)
               (vertex 0 0 hh))
     )
)
```

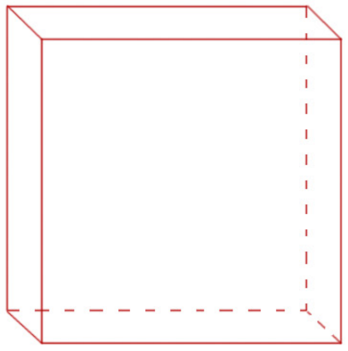


Figure 26: Place grammar page 4.

```
(shape_rule
    (name cushion_inv)
    (label af_z)
    (%constants
        (hh -0.03)
    )
    (constraint height < 15)
    (constraint width < 15)
    (constraint height > 5)
    (constraint width > 5)

    (output
```

(continues the same as rule 'cushion'...)

```
(shape_rule
    (name cushion2)
    (label af_z2)
    (%constants
        (hh 0.05)
        (s  0.05)
    )

    (output
        (shape_spec
            (label (value afsplit) (weight splitp))
            (label (value af_z) (weight trudep))
            (vertex s s hh)
            (vertex (- 1 s) s hh)
            (vertex (- 1 s) (- 1 s) hh)
            (vertex s (- 1 s) hh))
        (shape_spec
            (label_control terminal)
            (vertex 0 0 0)
            (vertex s s hh)
            (vertex s (- 1 s) hh)
            (vertex 0 1 0))
        (shape_spec
            (label_control terminal)
            (vertex (- 1 s) s hh)
            (vertex 1 0 0)
            (vertex 1 1 0)
            (vertex (- 1 s) (- 1 s) hh))
        (shape_spec
            (label_control terminal)
            (vertex s (- 1 s) hh)
            (vertex (- 1 s) (- 1 s) hh)
            (vertex 1 1 0)
            (vertex 0 1 0))
        (shape_spec
            (label_control terminal)
            (vertex 0 0 0)
            (vertex 1 0 0)
            (vertex (- 1 s) s hh)
            (vertex s s hh))
    )
)
```

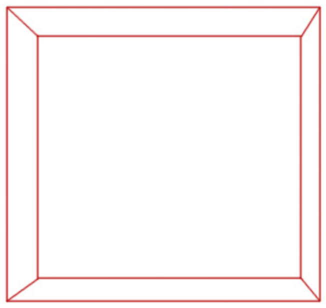

(tapered box)

Figure 27: Place grammar page 5.