Artificial Intelligence for Engineering Design, Analysis and Manufacturing

cambridge.org/aie

A whole-grammar implementation of shape grammars for designers

Andrew I-kang Li

Kyoto Institute of Technology, Matsugasaki, Sakyo-ku, Kyoto 606-8585, Japan

Abstract

I present an implementation of shape grammars that is aimed at supporting designers. It has two parts: a grammar editor and a stand-alone interpreter. The editor is the modeling application Rhinoceros3d using Python scripts. The interpreter is general, is three-dimensional, and supports subshape detection. A grammar is a Rhinoceros3d model; thus users can manipulate all its parts directly and immediately. That is, they can modify any shape without selecting or invoking an editor, and they can lay out the parts of the grammar in any way they find meaningful. Using this approach, which I call a whole-grammar approach, users are shielded from most subdomain tasks, like typing text files or specifying transformations. Informal observations suggest that users of this implementation can work effectively.

Research Paper

Cite this article: Li AI-kang (2018). A whole-grammar implementation of shape grammars for designers. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* **32**, 200–207. https://doi.org/10.1017/S0890060417000336

Received: 25 October 2016
Revised: 3 May 2017
Accepted: 3 May 2017

Key words:
Computational design; Design computing; Designers; Shape grammar implementations; Shape grammar

Author for correspondence:
Andrew I-kang Li, E-mail: andrewli@kit.ac.jp

Introduction

The theory of shape grammars was first proposed over 40 years ago (Stiny & Gips, 1972), and implementations soon followed, from Gips (1975) to Grasl and Economou (2013) and Pauwels et al. (2015).[1]

These implementations have demonstrated that several attributes of shape grammars are technically feasible. The most important of these is *subshape detection* which, as Krishnamurti (2015, p. 940) points out, is the *sine qua non* of any "worthwhile" shape grammar system.

Another important attribute is *generality*. Whether they are creating a new language of designs or seeking to characterize an existing language, designers need to be able to create and use *their own* grammars, not just those of a type hard-coded in the particular implementation at hand.

Tapia's (1999) implementation, known as GEdit and long obsolete, was notable because it supported not only emergence and generality but also visual interaction. In particular, it allowed users to specify shapes and rules by drawing them rather than by typing text. This is a natural approach for designers and has since become common.

This approach can be extended, by allowing users to manipulate, not only shapes and rules, but also the grammar as a whole. In the present implementation, a grammar is a three-dimensional (3D) model in the modeling application Rhinoceros3d, commonly known as Rhino. Rule application is handled by a stand-alone interpreter. Information is shared between the two applications in the form of text files: users export a grammar file from Rhino to the interpreter, and they reimport a derivation file from the interpreter into Rhino.

Using Rhino as a grammar editor benefits users in two ways. First, they can manipulate any element (i.e., line or labeled point) in any shape both directly and immediately. In most other implementations, users must enter an edit mode or shape editor to manipulate the elements of a shape; this includes the interpreter presented here when used without Rhino.

Second, they can lay out the parts of the grammar – and new shapes[2] – in groups that they find meaningful, and can rearrange them at any time. That is, they can employ secondary notation, a capability that has not previously been available.

One way of understanding these benefits is to consider Knight and Stiny's (2015, original emphasis) formulation:

> making is *doing* and *sensing* with *stuff* to make *things*.

If users can easily manipulate the whole grammar, including all its parts and new shapes, they can also easily do and sense with grammars. I call this a whole-grammar implementation.

© Cambridge University Press 2018

CAMBRIDGE UNIVERSITY PRESS

[1] See Chase (2010) and Li et al. (2009*a*) for discussions.
[2] I use *shape* for a grammatical emphasis and *design* in a broader context, but the two terms are essentially interchangeable.

https://doi.org/10.1017/S0890060417000336 Published online by Cambridge University Press

About the implementation

The whole-grammar implementation has evolved from the implementation by Chau et al. (2004), and it is there that the account begins.

Chau's implementation

Chau's implementation is written in Perl. It handles lines and labeled points in 3D space and, like Tapia's GEdit, supports emergence and generality. Users can create, change, save, and reuse their own grammar files. They see the current shape and the current rule as drawings (Fig. 1). However, unlike users of GEdit, they cannot draw shapes; they type text.

The names of initial shapes and of rules are displayed in scrollable lists. Users select an initial shape and rule, which are graphically displayed on the current shape and rule canvases.

Applying the rule to the current shape involves several steps. Users first propose a transformation t under which the rule is to be applied. They do this by selecting one triple of points in the left shape A and another in the current shape C, and querying whether the two triples, in fact, specify a transformation. If there is a transformation, users query the system whether the rule can be applied under that transformation [i.e., whether $t(A) \leqslant C$] and, if so, direct the system to apply the rule and update the current shape. Users repeat this process as necessary, selecting rules as desired.

In terms of shape grammar, Chau's implementation works reliably, but it is not so easy for designers to use. Chau kindly shared his source code, and I assembled a team. My team and I essentially wrapped Chau's implementation in what we hoped would be an easier interface for users. This wrapped version (Li et al., 2009*a*) is the stand-alone interpreter of the whole-grammar implementation.

The stand-alone interpreter

In a screenshot (Fig. 2) of the stand-alone interpreter, known as Grammar Environment, the most obvious change from Chau's implementation is that initial shapes, rules, and next shapes are now graphically displayed. In this way, users interact with drawings, not with text.

Less obvious is the simplification of rule application. We consolidate the steps mentioned above into a single user command, known as *show distinct* (next shapes). Users specify a rule and invoke this command to obtain all next shapes, if any, resulting from applying the rule to the current shape.[3]

The interpreter does this by dealing with the point triples as follows. First, it considers only endpoints and intersections, to prevent infinite matches. It finds a point triple in A, and tests it against all point triples in C. It keeps a list of all transformations t confirmed by a match of triples. Then it calculates all next shapes C' and displays the results as drawings in a scrollable list. Users select one, and the current shape is updated accordingly.

This is a straightforward automation of Chau's procedure. It is not very efficient and can surely be optimized, but it performs the essential function of shielding users from non-grammatical work.

[3]That is, given $A \rightarrow B$ and C, all $C' = [C - t(A)] + t(B)$ for all t such that $t(A) \leq C$.

The shape editors

For specifying shapes, we provide two tools. One is an internal shape editor (Li et al., 2009*b*). It is primitive but, again, it allows users to work with drawings.[4]

The other tool is an Autocad applet. With this applet, users specify an initial shape or a rule by drawing one or two shapes, and the applet creates a text file. They import individual files into Grammar Environment, where they assemble a grammar which can be saved as a single text file and reused. Users can use any capability of Autocad to create 3D shapes composed of lines and labeled points.

However, this Autocad applet has become unusable, because of repeated changes in file format. In making a replacement, I[5] not only use a different modeling application – Rhino – but also take the whole-grammar approach. Users do not export individual initial shapes and rules; they export the whole grammar. This is the grammar editor of the present implementation.

The grammar editor

In developing the grammar editor, I seek a balance between two opposing aims. On the one hand, I want to minimize the structure of the Rhino document in order to maximize the "WYSIWYG-iness" and manipulability of the grammar. On the other hand, the document needs structure to be parsable.

My approach is to put initial shapes and rules (which for brevity I will call *components*) on their own layers. This imposes some effort on users, but it preserves their freedom to control the layout. Users choose the layer names (and hence the component names), which can be displayed near the components.

In addition, the coordinate system of each shape in a rule must be specified, a step that users frequently overlook. Cubic frames specify the local coordinate systems. They also demarcate the volume within which users should draw the shapes and provide the means for the system to distinguish left and right rule shapes, which are on the same layer.[6] Frames are block instances – instances of a master object – so users can hide them when appropriate. Left and right shapes in a rule are identified by their relative x-positions, so users can rearrange them freely (as long as they have different x-positions).[7]

In this way, the system can distinguish initial shapes from rules and, within a rule, left and right shapes. Initial shapes and rules have names. This is enough structure to make the grammar parsable.

As for user actions, those involving the editing of components are native Rhino actions; there is nothing to add. What does need to be implemented are actions involving the *structure* of the grammar. These actions create new layers and frames: *new grammar*, *new initial shape*, and *new rule*. There are also scripts for input and output: *export grammar*, *import derivation*, and *import final shape*.[8]

These actions are implemented as Python scripts. For ease of installation by users, they are not wrapped in icons or dedicated

[4]It also turns out to be convenient and sufficient for novice users.

[5]The team is no more.

[6]The size of the frames corresponds to the size of the canvases in the interpreter: an object that fits into a frame fits into a canvas.

[7]I am assuming that users organize grammars in two dimensions, particularly for publication. However, the space is 3D, and users may indeed take advantage of this.

[8]By *final shape* I mean the last shape in a derivation. A *new shape* is any newly created shape.

https://doi.org/10.1017/S0890060417000336 Published online by Cambridge University Press

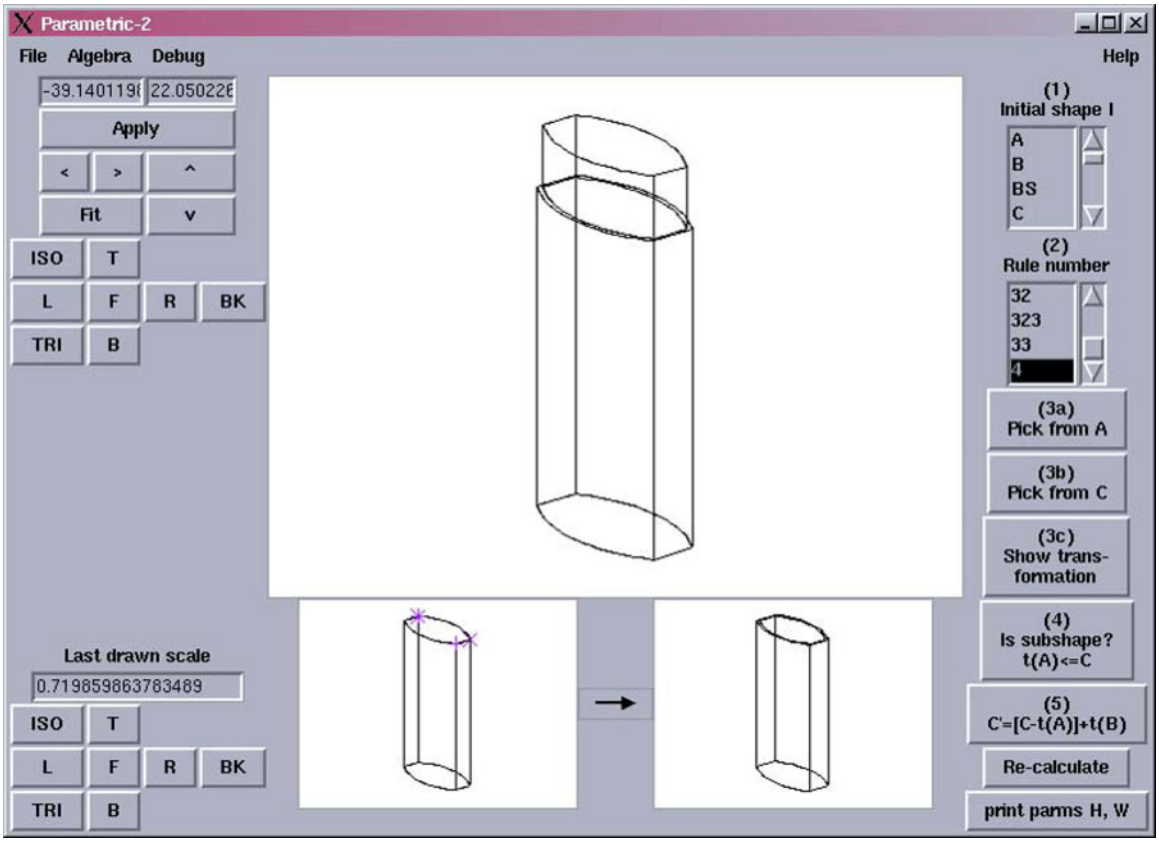


Fig 1. Chau's implementation. The current shape and the rule to be applied are graphically displayed in the center. On the right, from top to bottom: the list of initial shape names, the list of rule names, and the buttons for each step of rule application (enter two-point triples to specify a transformation *t*, query whether the rule is applicable under *t*, and, if so, apply the rule under *t*). (With the permission of Hau Hing CHAU.)

menu items, and users run them with the *run PythonScript* command or from the Python script editor.

Using the implementation

To demonstrate how users work with the implementation, I describe a typical scenario: developing a simple grammar and using the resulting shapes.

Editing the grammar

The users open a new Rhino document and set it up as a grammar document by running the *new grammar* script. The script creates two layers: one with one frame for an initial shape and another with two frames for the two shapes of a rule. Now they can draw 3D shapes with lines and annotation text dots as labeled points.[9]

To add components, they run the *new initial shape* and the *new rule* scripts, which create new layers and frames. To delete or revise components, they use native Rhino commands. They can arrange and rearrange components to help their thinking.

Let us suppose that they have drawn a two-rule grammar for generating Sierpiński gaskets (Fig. 3). They export the grammar by running the *export grammar* script, which generates a text file.

Testing the grammar

To test the grammar, the users switch to the interpreter and imports the grammar file with the *import grammar* command. The interpreter displays all the initial shapes and rules on scrollable canvases (Fig. 2). They select the initial shape,[10] and the interpreter displays it as the current shape, ready to be transformed into a rule application.

To see the possible next shapes – given the rules and the current shape – the users click *show distinct* (*all rules*). The interpreter displays all the possible next shapes in a scrollable window (Fig. 2). They select one to replace the current shape and continues transforming the current shape until they are satisfied. The interpreter retains a record of the derivation.

If they want to undo or redo a rule application, they can move backward and forward through the derivation, as in a browser. They may also decide that they will not get a satisfactory result with the grammar as it stands. In this case, they simply switch back to Rhino, revise the grammar, and run the revised version in the interpreter.

Let us assume that the users are satisfied with the final shape. They export their results by invoking the *save derivation* command, which creates a text file. They import the file into Rhino using the *import derivation* script, and the script draws the

[9]In published grammars, labels are often shapes that are not subject to transformation, that is, symbols. Text may be less appealing to look at, but it is functionally equivalent.

[10]In this implementation, a grammar may have more than one initial shape, so the user must select one as the first current shape.

https://doi.org/10.1017/S0890060417000336 Published online by Cambridge University Press

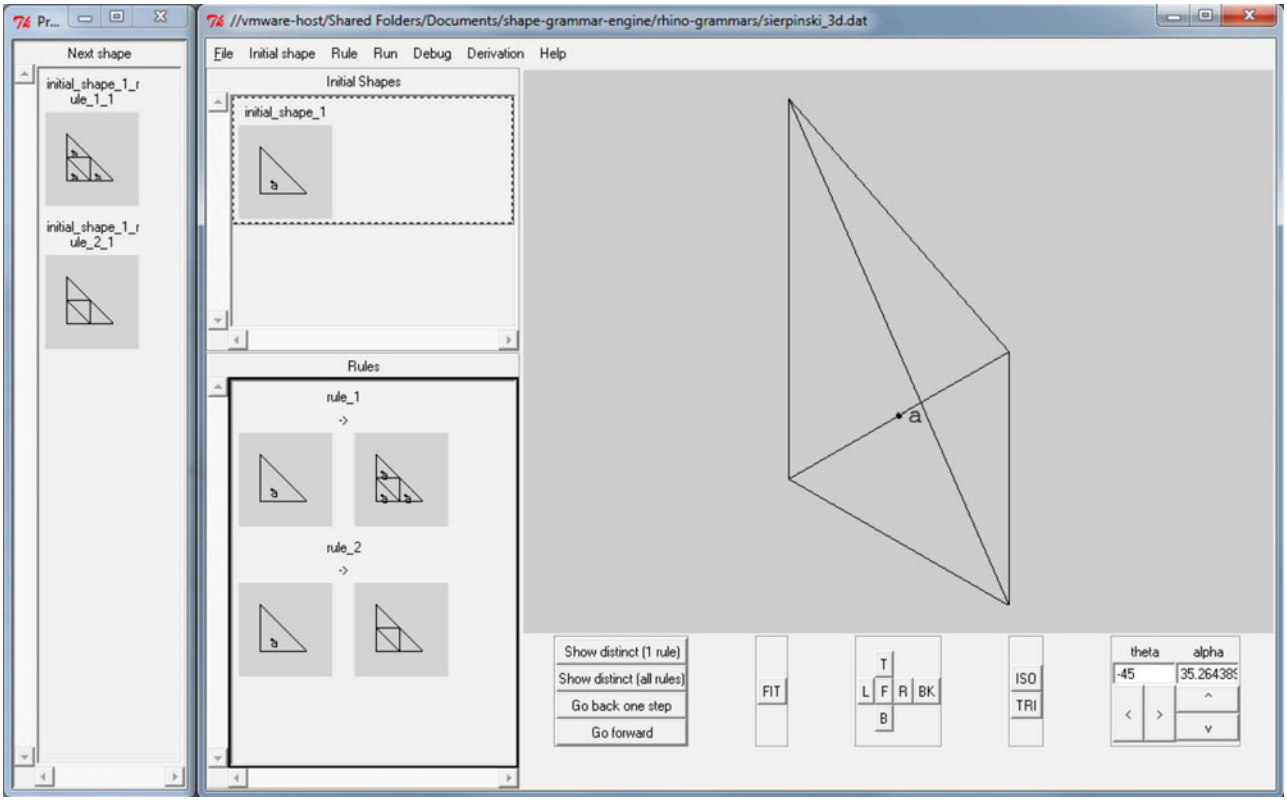


Fig. 2. The Sierpiński grammar in the interpreter. The large window on the right displays the initial shape (in plan view on the canvas on the upper left); the current shape, which at this point is the same as the initial shape (in an isometric view on the large canvas on the right); and the two rules in plan view (on the canvas below the initial shape). The narrow window on the left displays the two possible next shapes obtainable from the current shape.

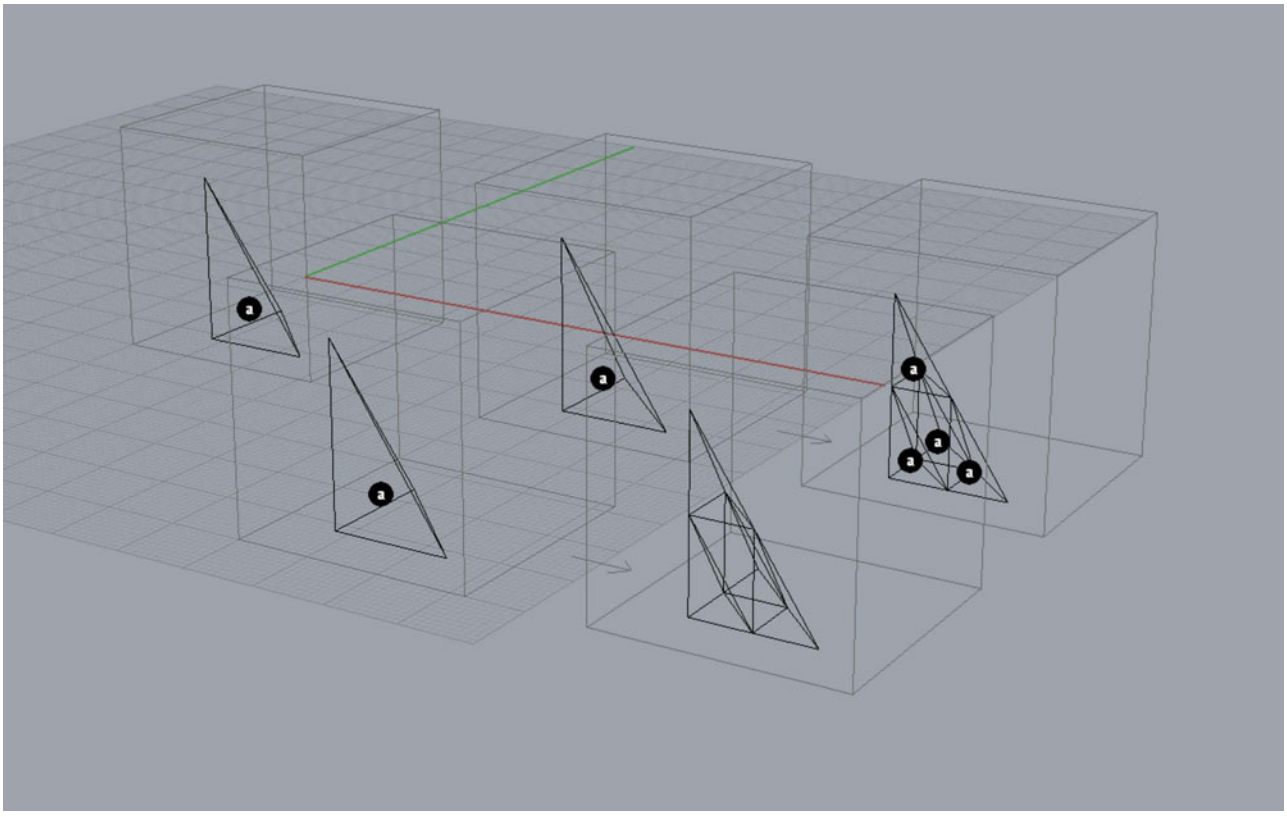


Fig. 3. A grammar in Rhino. It consists of one initial shape (back left) and two rules (right, with arrows), and generates Sierpiński gaskets.

https://doi.org/10.1017/S0890060417000336 Published online by Cambridge University Press

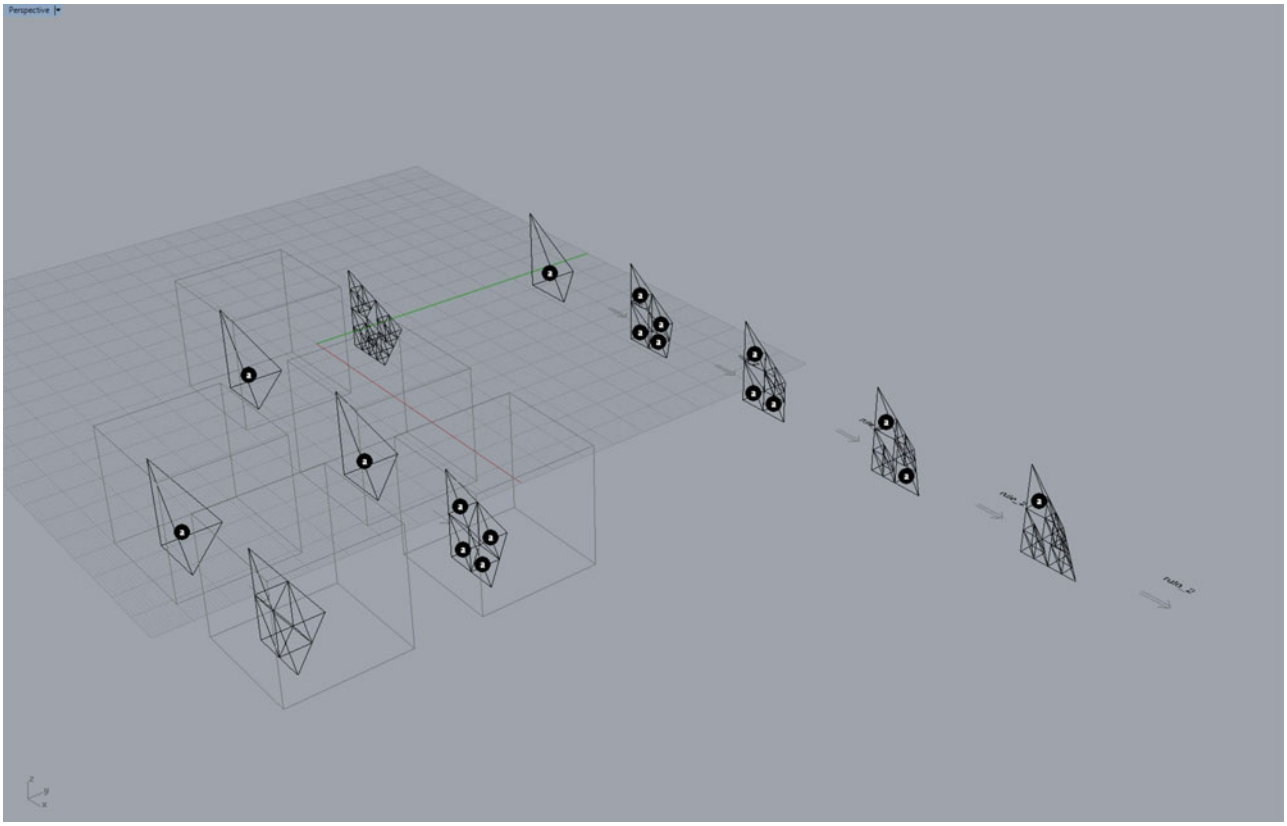


Fig. 4. The Sierpiński grammar and a derivation in Rhino. The grammar is on the lower left; the derivation is on the right and consists of the initial shape, the final shape, and four intermediate shapes.

derivation in the document (Fig. 4). The derivation includes the initial shape, the final shape, and all the intermediate shapes.

Using the resulting shapes

Now that the derivation is a part of the Rhino document, all its shapes are available to the user. They can, for example, convert them to surface models or solid models, use them for fabrication, or render them for publication. They can use them as input for further manipulation by Grasshopper or any other plug-in in Rhino.

Experiences with users

I have used the implementation in several types of classes. Most participants were undergraduate design students. Some had had experience with Rhino; virtually none had experience with shape grammar, except in one class.

I did not aim to conduct rigorous usability studies. Rather, I took advantage of the classes to observe participants informally as they learned to use the implementation and then did design work with it. I aimed to understand how the implementation was helping or hindering them in doing their work, how "transparent" it was.

The classes

The classes all began with the same activities. First, participants did a set of short shape grammar exercises by hand: specify and apply rules; given rules, find the outcome; given the outcome, find the rule. Next, they used the interpreter to run pre-written grammars. Then, in Rhino, they learned basic operations (drawing lines and text dots, handling layers and views) and how to run the Python scripts. After these activities, the program varied, according to the length and focus of the class.

1. *Two short workshops* on shape grammar. Because these workshops were so short (2 and 3 h), participants did the hand exercises and used the interpreter only; they did not use Rhino. They worked on screen and produced no physical output.
2. *A 1-day (7-h) workshop* on shape grammar. Most participants were experienced designers, and several knew how to use shape grammars. Once they had learned how to use the implementation, they experimented with it freely. They worked on screen and produced no physical output.
3. *A 1-week intensive course* in computer-aided design, covering Rhino, Grasshopper, and shape grammar. Participants used all these tools to design and render a pavilion. They produced no physical output.
4. *Two workshops on mass customization*. In the shorter workshop (2 days), groups of participants designed and laser-cut families of products like eyeglass frames and watch bands. In the longer workshop (5 days), the subject was housing. Groups of participants studied corpora of dwelling plans, derived grammars, and created new designs. They produced models with laser cutters and 3D printers.

https://doi.org/10.1017/S0890060417000336 Published online by Cambridge University Press

5. *A one-term course* on designing and making with shape grammars (15 sessions of 90 min). Participants implemented versions of grammars in the literature [e.g., Knight's (1980) Hepplewhite-style chair back grammar], created and laser-cut new designs (Fig. 5). The final assignment was to design and laser-cut a family of products. Participants created objects such as brooches, place mats, and flat-pack Christmas trees.

Observations

From these classes, I report a number of informal observations. Here I focus on the grammar editor; for observations on the interpreter, see Li et al. (2009*a*). Many observations were of common problems; I compiled these into an FAQ on the implementation's website[11] and used it in later classes.

Setting up

In most cases, setting up had to be done by the users themselves. This involves three main steps: installing the interpreter, installing the Python scripts, and setting the Python module search path in Rhino. These steps in turn sometimes involve details that were unfamiliar to users, so the process was time-consuming. This was a problem in the short workshops.

Using Rhino

Of Rhino's native capabilities, users need to know very little. Beyond basic interaction (e.g., manipulating views and layers), they need to know only how to draw lines and text dots. Only Rhino novices encountered difficulties; these mostly related to drawing in three dimensions. For example, they sometimes drew lines that appeared to be parallel to the *xy*-plane but were not.

Structuring grammars

To ensure parsability, users must structure their grammars according to a few guidelines.

- A shape must be composed only of individual straight lines. Users sometimes used objects such as polylines and rectangles.
- A shape must be composed of maximal lines. Users sometimes drew overlapping or abutting lines.
- Components must be located on their own layers. Users – mostly Rhino novices – sometimes neglected to do this.
- A frame specifies a local coordinate system for the shape it encloses. Users sometimes positioned shapes inconsistently.

Users made use of the ability to rearrange components, going through periods of relative messiness and then tidying up. This they did, for example, by grouping rules by left shape or by order of application. They usually grouped rules vertically, with the arrows aligned, and kept the relative positions of the two shapes and the arrow unchanged. This clearly suggests that arranging components is a way for users to think about the grammar. It is possible to lay out the components using three dimensions, but no users did this.

Switching between editing and testing

Users developed their grammars iteratively and easily mastered the steps involved in transferring files between Rhino and Grammar Environment. Nevertheless, the steps are unrelated to grammars and therefore are distractions that should be minimized or eliminated.

In addition, users preferred the *import final shape* script and rarely used the *import derivation* script. This suggests that they were more interested in accumulating and developing new shapes in a non-linear way. This was inconsistent with the model I had in mind, in which users would explore the design space by uncovering branches of the derivation tree. I consider this point more in the discussion below.

Using Rhino for post-production

In the term-long course, students fabricated objects on the laser cutter regularly. They imported new shapes from the interpreter to Rhino, and then exported them as Illustrator files. The shapes, which had been grammatically created, were not always exactly what had to be in the Illustrator files, so students often had to manipulate the shapes by hand or other non-grammatical means, such as a Python script.

In some cases, the shapes were diagrams that had to be developed further. Students converted straight lines to curved lines, offset single lines to create double lines, and trimmed lines. In other cases, they assigned colors to distinguish cutting lines and etching lines for the laser cutter. Post-processing is convenient because it is straightforward to move the shapes into Rhino.

One student laser-cut dozens of brooches. Another student, designing flat-pack Christmas trees, used a fractal approach that could generate many designs. But even a single design, after just a few iterations, became so complex that the interpreter became intolerably slow. He had no time to create another design.

Discussion

Next step: integration into Rhino

On the whole, users of the whole-grammar implementation worked effectively: they focused on design tasks and were minimally distracted by sub-domain tasks. One reason is that editing is transparent. Users can manipulate all parts of a grammar directly and immediately. Another reason is that post-processing is also transparent. Users can manipulate shapes non-grammatically, for example by replacing straight lines with curves.

Clearly, the next step is to take greater advantage of Rhino. One way is to abandon the external interpreter and build a new one inside Rhino. Testing can be direct and immediate, just as editing is already. New shapes would be drawn in Rhino in the first instance and would be immediately available for non-grammatical processing.

The result would not be a dedicated grammar tool; rather, it would be a new functionality in Rhino. All that is needed is standards for parsability and the ability to apply rules.

In making such a move, of course, I run the same risk that I met with the Autocad applet, namely that Rhino may change in ways that I cannot keep up with. I think this is inevitable; Gips (1999) has said as much. The only alternative to relying on an existing modeling application is to develop my own, a task for which I have neither time nor talent. I think that, on balance, the advantages outweigh the disadvantages.

Other questions

At this point, I would like to speculate a little about some of the observations above.

[11] http://andrew.li/interpreter/create-grammar.html

https://doi.org/10.1017/S0890060417000336 Published online by Cambridge University Press

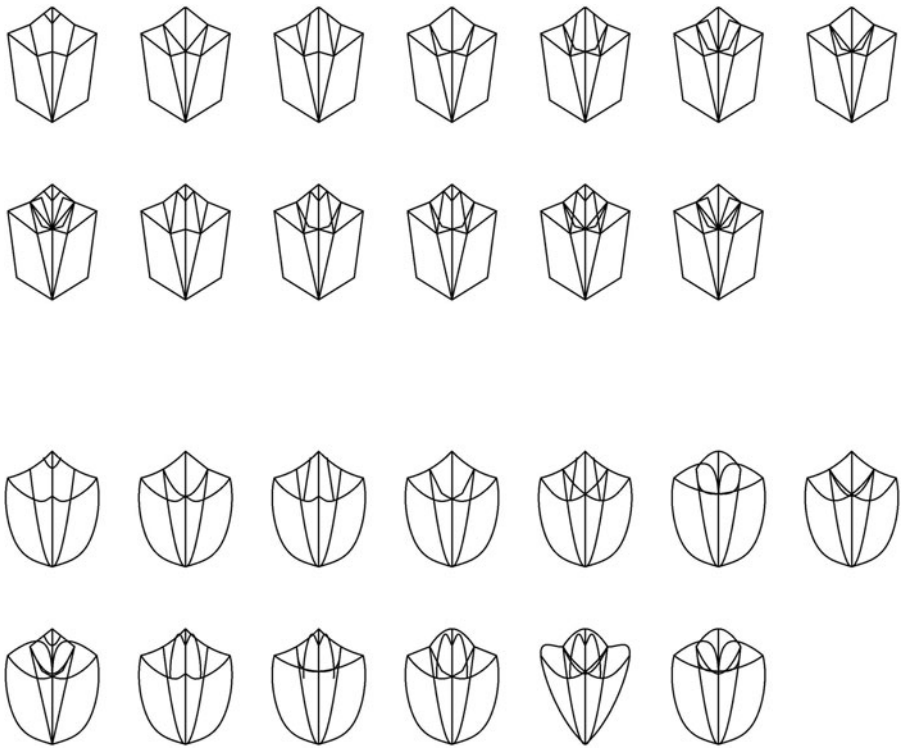


Fig. 5. Hepplewhite-style chair back diagrams after Knight (1980) by a student, NAKABE Kazutaka. Above are the diagrams with straight lines as generated by the grammar; below are the diagrams with the straight lines replaced by curved lines. (With the permission of NAKABE Kazutaka.)

One of those observations is that users did not draw very precisely. For example, they drew shapes that they expected to match but did not, because the shapes differed slightly. (The problem was usually that they had not used a snap or other appropriate drafting tool.) Or, they had not paid attention to the local coordinate system. This is not so surprising since most users were new to both shape grammar and Rhino.

To the extent that this is the reason for imprecision, the implementation serves to "train" users by enforcing shape grammar conventions, just as an interpreter of a programming language enforces that language's conventions. Eventually, users assimilate the conventions.

Another observation is that users imported more final shapes than derivations. Again, a possible explanation is that they had not yet acquired grammatical models of thought because they were novices.

But there is another explanation for these observations: amateur or designerly pragmatism. Woodbury (2010, p. 9) has written about the pragmatism of amateur programmers, and I think it applies equally well to amateur grammar users.

> Amateurs satisfice – they leave abstraction, generality and reuse mostly for "real programmers". … Amateurs program because they have a task to complete for which programming is a good tool. The task is foremost, the tool need only be adequate to it. Amateurs write most programs used in our world. Yet almost all programming tools are designed for the professional and are overly complex for the tasks amateurs attempt.

Woodbury's observation is consistent with my own and suggests several points for consideration while preparing to build an interpreter inside Rhino.

Formality versus dynamism

Once the internal interpreter is implemented, the Rhino workspace will be the scene of all the grammatical action, and new shapes will appear alongside existing ones. How will users organize them all? It is too early to know, but just raising the question suggests other ways to work with grammars.

For instance, given that designers seem to work in a nonlinear way, it is easy to imagine that they might feel confined by a static grammar. They might prefer to associate any three shapes (current, before, and after) to generate a fourth. That is, they would choose the current shape from among the shapes in the workspace, and create a rule on the fly to transform it. Parsing might be simplified, and the need for structure reduced. Heisserman *et al.* (2004) use such an approach.

Fuzziness versus imprecision

If we say that designers work fuzzily rather than imprecisely, then how and to what extent should an implementation accommodate this fuzziness?

Take, for instance, non-maximal shapes, those with overlapping or abutting lines. If a user draws a non-maximal shape, it seems reasonable for the system to maximize it. This is consistent with shape grammar theory, where restructuring is dynamic and instantaneous.

What about matching? How close does a match have to be? I suspect that designers often find rules too strict and prefer what they call *moves*. Technically, moves can be characterized as schemas, but they lack the precise assignments associated with schemas. Perfectly general schemas are difficult to implement, but fuzzy rules may be easier and almost as useful.

In any case, this suggests that the theoretical definition of schemas may not be a complete guide to action. In this respect, Jowers *et al.*'s (2010) visual implementation is an interesting example of fuzzy rule application.

Synthesis versus analysis

Up to this point, my observations have been about designers: people who create designs, or synthesists. But there is another

https://doi.org/10.1017/S0890060417000336 Published online by Cambridge University Press

category of grammar users that I have not observed: those who look at existing designs and try to understand them by describing them grammatically. They are known as analysts, and they frequently study style; Stiny and Mitchell (1978) have given a clear framework for this type of problem.

I suspect that analysts are more likely to be interested in formal structures like derivations and the design space since the task itself is formal. They might want to use the implementation differently from synthesists. For example, they might like to work with the derivation tree specified by a grammar.

Users may also do both synthesis and analysis. For example, they may analyze a sample of designs, formulate a grammar, modify the grammar, and create new designs. In this case, they would probably want to make use of all the capabilities that can be provided.

All of these are intriguing questions and should be investigated in the future.

Conclusion

The whole-grammar approach already helps users with developing grammars, creating new shapes, and further processing those shapes.[12] The next step in this direction is to build an interpreter inside Rhino; work has already begun. Users will use this interpreter, not as a single tool in a grammatical universe, but as another tool in the Rhino toolbox. This change of emphasis suggests new ways to support grammar users, ways that are less formal than what is usually seen in the literature and perhaps more congenial to designers. Like so many things grammatical, the results could be surprising.

Acknowledgments. I would like to thank Hau Hing CHAU (University of Leeds, UK) for sharing his code in the first instance; the Hong Kong Research Grants Committee, for supporting the development of Grammar Environment; CHANG Teng-wen (National Yunlin University of Science and Technology, Taiwan), HUANG Weixin (Tsinghua University, China), LEE Ji-hyun (KAIST, Korea), and Kyoto Design Lab (Kyoto Institute of Technology, Japan) for supporting the workshops and classes; and the anonymous reviewers, not only for their comments on the paper, but also for their suggestions for future work.

References

Chase S (2010) Shape grammar implementations: the last 35 years. DCC Workshop, Stuttgart, 11 July 2010. Available at http://www.slideshare.net/schase56/dcc-2010-grammars-workshop-chaserevisedcompresseshape-grammar-implementations-the-last-35-years (Accessed 25 April 2017).

Chau HH, Chen XJ, McKay A and de Pennington A (2004) Evaluation of a 3D shape grammar implementation. In Gero JS (ed.). *Design Computing and Cognition '04*. Dordrecht: Kluwer, pp. 357–376.

Gips J (1975) *Shape Grammars and Their Uses: Artificial Perception, Shape Generation and Computer Aesthetics*. Basel: Birkhäuser.

Gips J (1999) *Computer Implementations of Shape Grammars*. Workshop on Shape Computation. Cambridge, Mass: MIT.

Grasl T and Economou A (2013) From topologies to shapes: parametric shape grammars implemented by graphs. *Environment and Planning B: Planning and Design* **40**(5), 905–922.

Heisserman J, Mattikalli R and Callahan S (2004) A grammatical approach to design generation and its application to aircraft systems. In Akın Ö, Krishnamurti R and Lam KP (eds). *Generative CAD Systems*. Pittsburgh: Carnegie Mellon University, pp. 403–418.

Jowers I, Hogg DC, McKay A, Chau HH and de Pennington A (2010) Shape detection with vision: implementing shape grammars in conceptual design. *Research in Engineering Design* **21**(4), 235–247.

Knight TW (1980) The generation of Hepplewhite-style chair-back designs. *Environment and Planning B: Planning & Design* **7**, 227–238.

Knight T and Stiny G (2015) Making grammars: from computing with shapes to computing with things. *Design Studies* **41**, 8–28.

Krishnamurti R (2015) Mulling over shapes, rules and numbers. *Nexus Network Journal* **17**(3), 925–945.

Li AI, Chau HH, Chen L and Wang Y (2009*a*) A prototype system for developing two- and three-dimensional shape grammars. In *Proceedings of the 14th International Conference on Computer-Aided Architectural Design Research in Asia, CAADRIA 2009*, pp. 717–726.

Li AI, Chen L, Wang Y and Chau HH (2009*b*) Editing shapes in a prototype two- and three-dimensional shape grammar environment. In *Computation: The New Realm of Architectural Design (27th eCAADe Conference Proceedings)*, Istanbul, pp. 243–250.

Pauwels P, Strobbe T, Eloy S and DeMeyer R (2015). Shape grammars for architectural design: the need for reframing. In Celani G, Sperling DM, Moara J and Franco S (eds). *Computer-Aided Architectural Design Futures: The Next City – New Technologies and the Future of the Built Environment: 16th International Conference, CAAD Futures 2015, São Paulo, Brazil, July 8–10, 2015. Selected Papers*. Berlin: Springer, pp. 507–526.

Stiny G and Gips J (1972) Shape grammars and the generative specification of painting and sculpture. In Frieman CV (ed.) *Information Processing '71*. Amsterdam: North-Holland, pp. 1460–1465.

Stiny G and Mitchell WJ (1978) The Palladian grammar. *Environment and Planning B: Planning & Design* **5**, 5–18.

Tapia MA (1999) A visual implementation of a shape grammar system. *Environment & Planning B: Planning & Design* **26**, 59–73.

Woodbury R (2010) *Elements of Parametric Design*. London: Routledge.

Andrew I-kang Li is an Associate Professor of Design and Architecture at Kyoto Institute of Technology, Kyoto, Japan.

[12]The interpreter, scripts, and documentation are available at http://andrew.li/interpreter/

https://doi.org/10.1017/S0890060417000336 Published online by Cambridge University Press