LEARNING FROM USERS AND THEIR INTERACTION WITH A DUAL-INTERFACE SHAPE-GRAMMAR IMPLEMENTATION

RUDI STOUFFS[1] and ANDREW LI[2]
[1]*National University of Singapore*
[1]*stouffs@nus.edu.sg*
[2]*Kyoto Institute of Technology*
[2]*i@andrew.li*

Abstract. We present a shape grammar implementation with two new characteristics. One is that it is visual and directly manipulable: users draw the shapes and rules in a modeling application. The other characteristic is advanced technical capabilities, such as non-visual attributes and higher-order elements like surfaces. It consists of three components running in Rhinoceros3d. We also report on workshops that introduced the implementation.

Keywords. Shape grammars; interaction; implementation.

1. Introduction

The shape grammar formalism was developed some 40 years ago (Stiny and Gips 1972, Stiny 1980), and the first implementation followed quickly (Krishnamurti 1982). Most subsequent implementations have attended to only the most basic technical capabilities of shape grammars: labeled points and straight lines in 2D space. Notable exceptions include the one by Chau et al. (2004), which implemented these elements in 3D space, and Grasl and Economou (2018), which implemented parametric grammars.

Numerous technical capabilities remained to be implemented: surfaces in 3D space, attributes, and descriptions, to name a few. And in addition to the technical issues, there is also the question of user interface, which has received attention only from Tapia (1999) and Grasl and Economou (2018).

We have been working on an implementation that addresses both advanced technical capabilities and the user interface. We expect that we will be able to develop it into a tool that is both powerful and congenial for visually oriented users like designers.

It consists of three components running in the Rhinoceros3d environment: a back end and two front-end interfaces. The back end handles shape calculations and subshape recognition, and works with both interfaces. One front end offers direct manipulation of points, lines, shapes, and rules; it aims to capture the visual immediacy of shape grammars. The other interface offers advanced technical capabilities, including surfaces and curved lines in 3D space, attributes, and descriptions; it aims to provide more of the technical power of shape grammars.

All are written in Python; the 'technical' interface is implemented as a Grasshopper plug-in. These components have been reported previously (Dy and Stouffs 2018, Li 2018).

In this paper, we briefly present the three components as well as a series of workshops we ran to disseminate the results, informally test the implementation and learn from users interacting with either interface. We then discuss our findings and suggest a way forward.

2. The implementation

The shape grammar implementation we present here consists of three components running in the Rhino modelling environment: a back end and two front-end interfaces. We briefly describe the components and their capabilities, starting with the back end. we refer elsewhere for more extensive explanations (Dy and Stouffs 2018, Li 2018).

2.1. THE BACK END

The back end consists of a code library and API developed in the Python programming language, termed the SortalGI shape grammar interpreter. The SortalGI library has been developed with flexibility in mind, not imposing any specific representation, but providing a series of representational building blocks, supporting both geometrical and non-geometrical data types, and allowing these data types to be combined into larger representations, using both object-attribute and disjunctive compositional relationships (Stouffs 2018a). For example, supporting, among others, points, line segments, plane segments and labels, points and labels can be combined into labeled points under an object-attribute relationships, while labeled points, line segments and labeled plane segments can be combined under a disjunctive relationship, allowing any of the components to be present independently of one another. Currently, the SortalGI library includes representational building blocks for points, line and plane segments, circular and elliptical arcs, quadratic Bezier curves, labels, weights, colors, enumerative values, and (parametric) descriptions, in both 2D and 3D.

In addition, each representational building block and, by virtue of the (formal) compositional relationships, each composite representation includes a mechanism to support emergence, that is, the ability to recognize a specific shape within a larger combination of unstructured geometrical elements. In fact, the SortalGI library distinguishes two alternative matching mechanisms for spatial elements. The first one supports visual emergence under transformations of translation, rotation, reflection and uniform scaling. These similarity transformations allow a square or any other figure to be recognized irrespective of location, orientation and size. Scaling is uniform and, thus, a rectangle will match any rectangle with the same proportion between length and width. Shape grammars relying on shape recognition based on similarity transformations are often termed non-parametric shape grammars.

The second matching mechanism supports associative emergence under topological constraints and associations of perpendicularity and parallelism.

Adopting a graph-based representation, this matching mechanism allows any arbitrary *n*-sided polygon to be recognized as such. In addition, if the polygon to be matched contains any parallel or perpendicular edges, these will also be considered as constraints. As such, a rectangle will match any rectangle irrespective of its proportion between length and width. There are few shape grammars, if any, in the literature that rely on associative emergence. Most are identified to require an explicit parametric matching mechanism, although there exist no general shape grammar implementation supporting any explicit parametric matching mechanism. Instead, other so-called parametric shape grammar implementations similarly rely on some form of associative emergence using a graph-based representation to support parametric-like shape grammars (Wortmann 2013, Strobbe et al. 2015, Grasl and Economou 2018).

The SortalGI API has been specifically developed to support the integration of the SortalGI library within the Rhino modelling environment. In particular, it not only acts as a programming interface providing access to the underlying functionality, but also supports the conversion of geometric data from Rhino into the SortalGI interpreter and back. Note that although the entire library is available within Rhino, the API does limit the extent of geometric and non-geometric element types that are supported, due to the need to graphically visualize the data within Rhino. The conversion procedure accepts Rhino GUIDs referencing Rhino objects and translates these into an appropriate SortalGI shape representation. Upon applying any rule to the shape, the new shape resulting from the rule application will be converted back into Rhino GUIDs to be visualized within Rhino. Note that the SortalGI library maintains a rule register and every rule is automatically added to the register. As such, rule names must be unique and rules can be retrieved from the register by name.

2.2. THE VISUAL INTERFACE

The visual interface forms part of the Rhino environment and adds to the many functionalities already available there. The user draws the components of the grammar simply as Rhino objects. In order to make these components parsable for communication to the back end, the interface needs to use some of Rhino's structural capabilities, such as layers. At the same time, we want to keep those very capabilities available to users to the extent possible. In anticipation of this tension between structure and freedom, we followed a few working guidelines.

Firstly, shapes are composed of Rhino objects, currently line curves and text dots. That is, the objects themselves are the shape; there is not some symbolic representation between the shape and the user. They are in the foreground, available for manipulation by the user, and persist between work sessions as the record of the grammar.

Secondly, the Rhino work space is divided between the system embodied by the visual interface in communication with the back end, and the user. The system needs a place to do its work, and so do users. One way we try to accommodate both system and users is by having the interface display rules and all calculated shapes in the positive-*y* half of the three-dimensional virtual work space. The other half is left for users to use as they will. The other way is by assigning rules and calculated shapes to their own, automatically named layers. This way, the system can identify rules and calculated shapes, and users can also create and name layers for their own use.

Thirdly, commands are implemented as Python scripts; users invoke a command by running a script. This is an interim measure; in future versions commands will be available through more straightforward means, like menu items.

These working guidelines are intended to make users' experience easier and more productive. Their first step is to initialize the Rhino document by running the *initialize* script, among others, creating a new layer, *Shape 0*, for the initial shape. Shapes on layers named *Shape n* are scrolled so they stay above the *x*-axis, with the newest always closest to the *x*-axis. If the document already contains a grammar, the front end reads the rules into the rule register, and users can resume using the grammar immediately. Otherwise, they should create at least an initial shape and a rule. To create an initial shape, users simply draw it with line curves and text dots in the upper right quadrant of the *xy*-plane on the layer *Shape 0*.

To create a rule, users draw in the lower two quadrants on any user-named layer. Then they run the *create rule* script, which prompts them to select: 1) the elements of the left shape; 2) a reference point for the left shape; 3) the elements of the right shape (if any); and 4) a reference point for the right shape (if not empty). The new rule becomes part of the rule register. The interface then draws the rule in the upper left quadrant on a new layer *Rule 1*. Around each of the left and right shapes is a three-dimensional frame which indicates the shape's coordinate system. The elements of the two shapes and both frames are locked in a single group for easy selection for rule application. If users draw a second rule, both rules will be scrolled away from the *x*-axis.

Having at least one shape and one rule, users can run the *apply rule* script, which prompts them to select: 1) the shape; 2) the subshape; and 3) the rule. By selecting a subshape, users can, for example, apply a rule to a single cell in a matrix (figure 1). When there are multiple potential rule applications, the system applies them all in parallel and draws the results within a row in the upper right quadrant, each shape on its own layer, scrolling them - along with all earlier generations of calculated shapes - away from the *x*-axis.

2.3. THE 'TECHNICAL' INTERFACE

Notwithstanding the many advantages of a visual interface, an alternative 'technical' interface that provides access to a larger set of SortalGI functionalities is beneficial. Hereto, we opted for a Rhino/Grasshopper plug-in (Dy and Stouffs 2018) providing a range of Grasshopper (GH) components that support the creation and manipulation of shapes and descriptions, and the creation and application of rules, including rule predicates and directives. Descriptions are semi-structured textual descriptions that can be parameterized within a description rule (Stouffs 2018b). Descriptions can also relate to spatial elements by querying their properties or specifying conditions on the values of such properties. Predicates serve to express additional conditions on the application of an associative shape rule that cannot simply be explicated within the left-hand side shape, while directives are value specifications that are required when applying an associative shape rule, where this value specification cannot be derived from or expressed within the right-hand-side shape (Stouffs 2019).

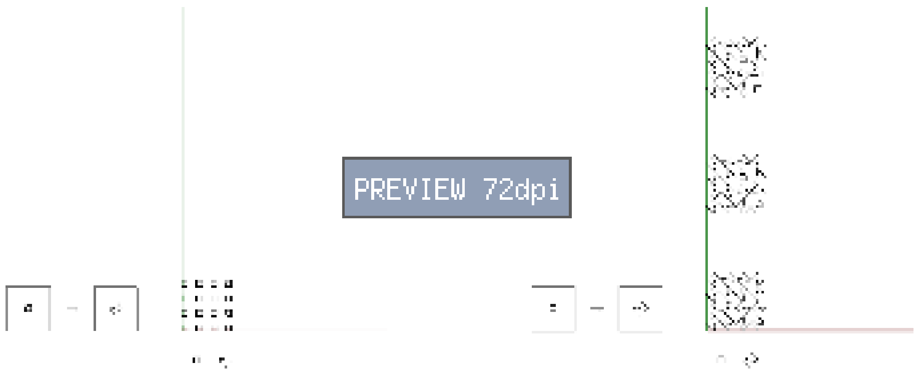
Figure 1. The workspace of the visual interface. Left: The rule (with frames around the two shapes) is in the upper left quadrant, while the initial shape (the 16 squares) is in the upper right quadrant. The two shapes in the lower right quadrant are the shapes drawn by the user to define the rule. Right: Given the 16 squares, the same rule is applied 16 times, with the last 3 generations shown here; the first 13 generations are scrolled away from the x-axis.

While, in principle, the visual interface can also support these additional features, the important question is how these functionalities can be integrated visually in a way that doesn't impede the designer. In the GH interface, on the other hand, it is simply a question of adding one or more GH components. As such, additional functionalities can easily be tested and explored and provided to designers who are willing to put up with a non-visual interface. Note that the GH interface requires no literal programming or scripting, although creating a GH model can be considered as a form of visual programming.

The plug-in supports both "non-parametric" and associative rules, shapes including points, line segments, plane segments, circles, ellipses, (circular) arcs and quadratic Bezier curves, both with and without description attributes, and stand-alone descriptions. Descriptions serve to represent both labels (within double quotes) and more general descriptions (see section 2.1).

The plug-in defines both a rule object and a shape object. A shape object contains up to three components, the SortalGI representation of the shape, the Rhino visualization of the same shape (minus stand-alone descriptions), and any stand-alone descriptions. Stand-alone descriptions are necessarily typed, the predefined description types are specified as input to the *Setup* component. Shapes can be constructed using either the *Shape* or the *dShape* component, the latter allowing for the specification of stand-alone descriptions. Both components accept an optional reference point, allowing the left-hand-side and right-hand-side shape of a rule to be specified at different locations while relating them via their reference points. Attribute descriptions can be added to any geometry using one of the *Text Point*, *Text Curve* and *Text Surface* components. In addition, there exist components to move, rotate, scale and mirror a shape object, as well as find the sum, product (intersection) or difference (complement) of two shapes.

A rule object is either a “non-parametric” or an associative SortalGI rule. The *Rule* component constructs a “non-parametric” rule from a rule name and a left-hand-side and right-hand-side shape object. Similarly, the *pRule* component constructs an associative rule and additionally accepts a list of predicates and directives (see section 2.1). In each case, only the right-hand-side shape is allowed to be an empty shape.

There exist four components to apply a rule object onto a shape object. The *Apply* component returns a single rule application that can either be randomly selected from all possible rule applications or using a rule application index. The *Apply All* component yields a list of shape objects corresponding to all possible rule applications. The *Apply All Together* component applies all possible rule applications in sequence, ignoring the fact that some elements that would be required for rule applications may not be present anymore after the first possible rule application is performed. Finally, the *Derive* component accepts a list of one or more rule objects and applies these in sequence, each on the result of the preceding rule application. Here too, the specific rule application can either be randomly selected from all possible rule applications or using a rule application index. All four components also accept an optional subshape object to limit potential matches. Rule application is always performed on the entire shape, not the subshape. In the case that no rule applications can be found, the original shape is returned, such that any consecutive rule application component may still execute.

Additionally, a *Matches* component returns not the rule applications, but a list of the matched shapes that give rise to these rule applications. This allows, among others, for a divide-and-conquer approach, collecting the list of matched shapes, then performing a series of rules onto each matched shape separately, in order to increase efficiency. All application components, including the Matches component, also return a translation vector or list thereof for visualizing the resulting shape(s) aside from the original shape (in the x-direction) and separated (in the y-direction).

Figure 2. Working with the ‘technical’ interface. Top: The GH interface illustrating the basic template of rule specification: collecting the Rhino geometries, defining the left-hand-side and right-hand-side shape objects and defining the rule object, upon which the rule can be applied. Bottom: The Rhino workspace including the rule shapes and initial shape drawn by the user as well as the sequence of rule applications resulting from the ‘SGI Derive’ component.

3. Informal testing

In order to understand how designers would work with the interpreter, we ran three workshops with the two interfaces in different combinations: first the ‘technical’ interface, then the visual interface; first visual, then ‘technical’; and, finally, ‘technical’ only. Most participants were design students with little or no previous exposure to shape grammars.

3.1. CAAD FUTURES WORKSHOP

In June 2019, we ran a four-day workshop before the CAAD Futures conference at KAIST, Daejeon, South Korea. The first two days were devoted to the ‘technical’ interface, the third to the visual interface, and the fourth to scripting for direct communication with the back end. One difficulty was that most participants had little or no experience with GH and therefore had to learn both the ‘technical’ interface and GH at the same time. Another difficulty was that all examples presented were constructed entirely in GH, resulting in models that were difficult for novices to understand.

Under these circumstances, the visual interface seemed particularly easy for users to grasp. Not only could they manipulate objects directly (e.g., by drawing them), but they had fewer capabilities (in terms of actions, matching, and data types) to deal with.

3.2. YUNTECH WORKSHOP

Two months later, we ran an eight-day workshop at National Yunlin University of Science and Technology (aka Yuntech), Taiwan. In this workshop, we presented the visual interface before the ‘technical’ interface. We also took advantage of the ample access to laser cutters and 3D printers.

Using the visual interface, and inspired by figure 1, the first exercises focused on using just one or two rules to apply to each cell of a square grid and to materialize the outcome as laser-cut coasters. By relying on just one or two rules to create interesting results, participants came to understand matching under similarity transformations, especially with respect to symmetries, and were able to iterate the process by simply updating or replacing a single rule. In fact, the physical materialization of the outcome greatly benefited the iterative process because of the different representations (drawing and physical object).

Halfway through, we switched to the ‘technical’ interface, but maintained the habit of drawing the rule’s shapes in Rhino; then we used a GH template to collect the Rhino information and construct the left shape, the right shape, and the rule (e.g., figure 2).

We tried two approaches. The first was a 3D extension of the coasters, using a 3D grid and one or more 3D rules. The other was a recursive or fractal approach, in which one or more rules were applied iteratively through self-similarity. Both approaches are feasible with both interfaces, so the emphasis fell on the interfaces and how best to use them. Participants sometimes used additional functionality, including actions such as derivations, associative matching, and data types such as labels.

3.3. ECAADE + SIGRADI WORKSHOP

Subsequently, the first author ran a two-day workshop before the eCAADe + SIGraDi conference at the University of Porto, Portugal. In this workshop, participants used only the 'technical' interface. Learning from the previous workshops, he continued the approach of drawing shapes in Rhino and then from those shapes constructing rules in GH. Compared to the other workshops, there were fewer participants, and most of them had had previous experience with GH.

On the first day, participants used the similarity-based matching mechanism; on the second, associative matching. The rule sets were small, but participants were able to apply rules both in parallel and sequentially, and thus explore forms extensively (figure 3).

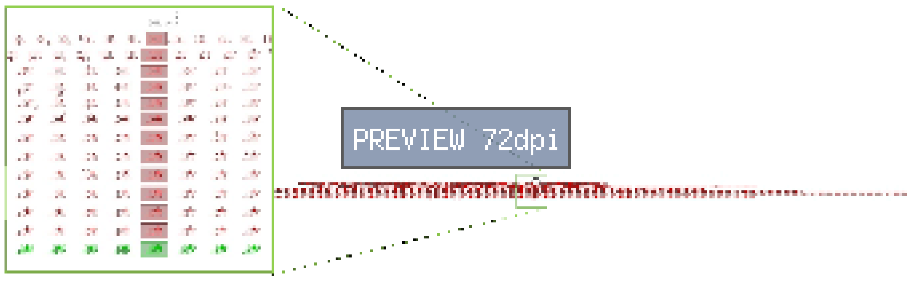

Figure 3. Exploration of forms using the 'technical' interface in a combination with other GH functionality (image courtesy of Šimon Prokop).

4. Discussion

Our observations of users come from workshops, not rigorous tests, but we believe that these observations suggest some lessons.

4.1. SHAPE GRAMMARS AND THE INTERPRETER IN THE LARGER DESIGN PROCESS

One lesson is that fabrication seems to complete a design iteration for users. It makes the virtual real, and gives designers another representation to evaluate. Furthermore, in such an iteration, the interpreter is used in combination with other tools in the Rhino environment.

Our aim is to create, not a pure shape grammar environment, but a tool for designers. And as Woodbury (2019) observed, designers are pragmatic and concerned more with results than with technical niceties. We aim to support designers, pragmatism and all, and we do so by embedding the interpreter in an environment that is rich in design tools. We help them generate designs rapidly and do not insist on, say, organizing those design according to a derivation tree. The derivation tree, of course, is a construct interesting to theoretical types like the present authors, but so far no user has ever asked about it.

4.2. THE RELATION BETWEEN THE TWO INTERFACES

The second lesson is similar to the first, as it also involves a tension between the two interfaces, specifically the complementary trade-offs they offer between immediacy and intuitive obviousness on the one hand and technical power on the other.

Of participants using the ‘technical’ interface, there were a (very) few who were familiar with both GH and shape grammars. Those participants were able to operate somewhat non-visually and, as with Rhino, see the SortalGI plug-in as adding to the many functionalities already available in GH (figure 3). They were able to consider higher-order issues like grammatical algorithms (Stouffs and Hou 2019). But they were a minority; most participants were not ‘power users’.

This suggests that there is more than one type of user, and that we should consider how to fashion the two interfaces to accommodate as many types as possible. For example, some novices may be happy to remain novices as long as they can generate designs easily. But others may want to become power users. How can the visual interface help both types?

Should we aim to make the visual interface more powerful? Can we do so without compromising its immediacy? Or should we maintain both interfaces side by side? This may seem the more obvious approach, but we need more experience to be sure.

5. Conclusion

The two lessons above suggest that we still have much to learn about how designers use a shape grammar interpreter in their work. We will continue this process of development and informal testing in hopes of enabling designers to work easily and effectively with grammars. Having mainly solved the technical part, the emphasis is now on the human/designer.

References

Chau, H.H., Chen, X.J., McKay, A. and de Pennington, A. 2004, Evaluation of a 3D shape grammar implementation, *in* J.S. Gero (ed.), *Design computing and cognition ’04*, Kluwer, Dordrecht, 357–376.

Dy, B. and Stouffs, R.: 2018, Combining geometries and descriptions: a shape grammar plug-in for Grasshopper, *Proceedings of eCAADe 2018, Vol. 2*, Lodz, 499–508.

Grasl, T. and Economou, A.: 2018, From shapes to topologies and back: an introduction to a general parametric shape grammar interpreter, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **32**(2), 208–224.

Krishnamurti, R.: 1982, *SGI: a shape grammar interpreter*, n.p.

Li, A.: 2018, A whole-grammar implementation of shape grammars for designers, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **32**(2), 200–207.

Stiny, G.: 1980, Introduction to shape and shape grammars, *Environment and Planning B: Planning and Design*, **7**, 343-351.

Stiny, G. and Gips, J.: 1972, Shape grammars and the generative specification of painting and sculpture, *Information processing*, **71**, 1460–1465.

Stouffs, R.: 2018a, Implementation issues of parallel shape grammars, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **32**, 162–176.

Stouffs, R.: 2018b, Description grammars: precedents revisited, *Environment and Planning B: Urban Analytics and City Science*, **45**(1), 124–144.

Stouffs, R. and Hou, D.: 2018, Comnposite shape rules, *Proceedings of DCC*, Lecco, Italy, 439–457.

Strobbe, T., Pauwels, P., Verstraeten, R., De Meyer, R. and Van Campenhout, J.: 2015, Toward a visual approach in the exploration of shape grammars, *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, **29**(4), 503–521.

Tapia, M.: 1999, A visual implementation of a shape grammar system, *Environment and Planning B: Planning and Design*, **26**, 59–73.

Woodbury, R.: 2019, *untitled*, personal communication.

Wortmann, T.: 2013, *Representing shapes as graphs*, Master's Thesis, MIT.