“Computer Implementation of Shape Grammars”

James Gips
Boston College

invited paper

Workshop on Shape Computation

MIT, 1999

http://www.shapegrammar.org/implement.pdf
Computer Implementation of Shape Grammars

James Gips
Computer Science Department
Boston College
Chestnut Hill, Mass. 02467

gips@bc.edu
http://www.cs.bc.edu/~gips

Just as there are symbolic thinkers, people who think mainly in terms of letters and numbers and other symbols, and visual thinkers, people who think primarily in terms of shapes and colors and spatial relationships, there can be symbolic computations and visual computations.

Clearly, symbolic computations are much more extensively developed; digital computers are based on symbols.

Shape grammars are intended to form a basis for purely visual computation. The primitives in shape grammars are shapes, rather than symbols. The relationships and operations are all spatial (e.g. similarity, rotation) rather than symbolic.

One can show the formal equivalence of traditional symbolic computations and shape grammars. The proof is in two steps: show how shape grammars can be implemented on a computer and show how a universal symbolic computing mechanism, traditionally a Turing machine, can be implemented as a shape grammar.

Shape grammars have been used in visual areas: in painting and sculpture, in architecture, in design education, in engineering design, in product design. People who develop or use shape grammars have had two choices: either simulate the shape grammar by hand or use / develop a program on a digital computer.

One can imagine a non-symbolic physical implementation of the generation of shapes using shape grammars. I am picturing a two dimensional light source with the light passing through a filter that has a cutout of the original shape in the shape grammar. This visual image of the shape then passes through a filter corresponding to the left side of a shape rule. If a match is found the corresponding right side is substituted visually. The generation of the shape proceeds with all representations being visual. The shapes generated are displayed directly as the output. This is fanciful but not inconceivable.

Thirty years ago computers were used almost exclusively for symbolic tasks (payrolls, airline reservations). People who worked with computers tended to be intensely symbolic thinkers. With the advent of computer graphics and then the
graphical user interface and then the web computers became friendlier for visual thinkers. But underlying the visual images and interfaces are symbols. A visual web page is represented by html text or a symbol based Javascript or Java program. Writing programs is still a symbol-intensive activity.

The tension in computer implementation of shape grammars is the tension between the visual nature of shape grammars and the people who want to use them and the inherently symbolic nature of the underlying computer representations and processing.

**What is the task performed by the program?**

There are several possible tasks for programs that implement shape grammars.

The most common task, and perhaps the first that comes to mind, is to aid in the generation of shapes from shape grammars. A program for shape grammar generation commonly is called a shape grammar **interpreter**. Here one enters a shape grammar into the computer and the program either generates shapes in the language or the user guides the program, for example in selecting the rule to be applied and where in the current shape to apply it. The program can have a particular shape grammar built-in, so it only generates plans for African pygmy thatched huts, or it can allow the user to enter in a shape grammar of a certain (restricted) type.

A second type of program is a **parsing** program. A parsing program is given a shape grammar and a shape. The program determines if the shape is in the language generated by the grammar and, if so, gives the sequence of rules that produces the shape. This is an analysis problem rather than a design problem. Here we might be given a plan and a shape grammar for determining pygmy thatched huts and the program would tell us whether or not the plan is indeed a syntactically correct plan of a thatched hut.

A third type of program is an **inference** program. The grammatical inference problem is given a set of shapes construct a shape grammar that generates the shapes (plus other shapes in the same "style"). So, we would give a grammatical inference program a corpus of known plans of pygmy thatched huts and the program would automatically generate a shape grammar for pygmy huts. Maybe it would even write the grammar up for publication.

If we consider symbol grammars instead of shape grammars, a generation program would produce grammatical sentences. This would be akin to our producing grammatical English sentences. A parsing program would be given proposed sentences to determine if they are grammatical. This would be akin to our analyzing or understanding sentences produced by others. An inference program would be given a set of grammatical sentences and be asked to create a reasonable
grammar for the language. This would be akin to our producing a grammar for a new (foreign) language given sentences in the language.

A fourth type of program would be a Computer Aided Design program for shape grammars. The program would help the user design shape grammars. It would be more than a shape grammar interpreter. It would assist the user in creating a shape grammar by providing sophisticated tools for the user. This follows Terry Knight's (1998) statement that "the process of developing an original grammar is analogous to the traditional design process." If computers can help designers of widgets, they can help designers of shape grammars. The flip of this type of program would be a shape grammar plug-in for a traditional computer aided design program that would use shape grammars to help the practicing designer.

**Brief Survey of Shape Grammar Implementations**

Hau Hing Chau kindly provided a list of shape grammar implementations. A modified version of his list follows:

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference</th>
<th>Tool(s)</th>
<th>Subshape</th>
<th>2D/3D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 simple interpreter</td>
<td>Gips 1975</td>
<td>SAIL [a]</td>
<td>No</td>
<td>2D</td>
</tr>
<tr>
<td>2 Shepard-Metzler analysis</td>
<td>Gips 1974</td>
<td>SAIL</td>
<td>No</td>
<td>2D/3D</td>
</tr>
<tr>
<td>3 shape grammar interpreter</td>
<td>Krishnamurti 1982</td>
<td></td>
<td>Yes</td>
<td>2D</td>
</tr>
<tr>
<td>4 shape generation system</td>
<td>Krishnamurti, Giraud 1986</td>
<td>PROLOG [b]</td>
<td>Yes</td>
<td>2D</td>
</tr>
<tr>
<td>5 Queen Anne houses</td>
<td>Flemming 1987</td>
<td>PROLOG</td>
<td>No</td>
<td>2D</td>
</tr>
<tr>
<td>6 shape grammar system</td>
<td>Chase 1989</td>
<td>PROLOG/Mac</td>
<td>Yes</td>
<td>2D</td>
</tr>
<tr>
<td>7 Genesis (CMU)</td>
<td>Heisserman 1991</td>
<td>C/CLP</td>
<td>No</td>
<td>3D</td>
</tr>
<tr>
<td>8 GRAIL</td>
<td>Krishnamurti, Giraud 1986</td>
<td>PROLOG/Mac</td>
<td>Yes</td>
<td>2D</td>
</tr>
<tr>
<td>9 grammatica</td>
<td>Carlson 1993</td>
<td></td>
<td>No</td>
<td></td>
</tr>
<tr>
<td>10 Stouffs 1994</td>
<td></td>
<td></td>
<td>Yes</td>
<td>2D/3D</td>
</tr>
<tr>
<td>11 Genesis (Boeing)</td>
<td>Heisserman 1994</td>
<td>C++/CLP</td>
<td>No</td>
<td>2D/3D</td>
</tr>
<tr>
<td>12 GEdit</td>
<td>* Tapia 1996</td>
<td>LISP [c]/Mac</td>
<td>Yes</td>
<td>2D</td>
</tr>
<tr>
<td>13 shape grammar editor</td>
<td>Shelden 1996</td>
<td>AutoCAD/AutoCAD</td>
<td>Yes</td>
<td>2D</td>
</tr>
<tr>
<td>14 implementation of basic grammar</td>
<td>Duarte Simondetti 1997</td>
<td>AutoCAD/AutoCAD</td>
<td>No</td>
<td>3D</td>
</tr>
<tr>
<td>15 shape grammar interpreter</td>
<td>Piazzalunga, Fitzhorn 1998</td>
<td>ACIS/LISP</td>
<td>No</td>
<td>3D</td>
</tr>
<tr>
<td>16 SG-Clips</td>
<td>Chien, et al. 1998</td>
<td>CLIPS</td>
<td></td>
<td>2D/3D</td>
</tr>
<tr>
<td>17 3D architecture form synthesizer</td>
<td>Wang 1998</td>
<td>Java/Open Inventor</td>
<td>No</td>
<td>3D</td>
</tr>
<tr>
<td>18 coffee maker grammar</td>
<td>** Agarwal and Cagan 1998</td>
<td>Java</td>
<td></td>
<td>2D/3D</td>
</tr>
</tbody>
</table>

* available on the web at [http://www.shapegrammar.org](http://www.shapegrammar.org) under Projects

** available on the web at [http://www-cad.me.cmu.edu/cdl/](http://www-cad.me.cmu.edu/cdl/) under Research
In the initial work on shape grammars (Stiny and Gips 1972) all generation was
done by hand. Dozens of paintings on canvas were produced with a paintbrush
from a pencil drawing.

The first program (Gips 1975) listed in the table above enabled the user to enter a
simple two rule shape grammar and then generate shapes produced by the (parallel)
application of the rules of the grammar. Only one polygonal vocabulary element
was allowed. The program ignored the issue of detecting subshapes. The second
program (Gips 1974) used a grammar to solve a well-known three-dimensional
perceptual task.

George Stiny (1975) developed the basic landscape for shape operations and
relations, including subshape, and their algorithms.

Ramesh Krishnamurti (1980, 1981, 1982) did pioneering work on data structures and
algorithms for solving the general subshape and rule application problem for two
developed computational models for three-dimensional grammars. Ugo
Piazzalunga and Patrick Fitzhorn (1998) describe a three-dimensional shape
grammar interpreter and issues involved in the complexity of three-dimensional
implementations.

Ulrich Flemming (1987) developed a Prolog program that implemented his
grammar for Queen Anne houses.

Currently two programs are available on the web. Mark Tapia’s (1999) GEdit is a
general two-dimensional shape grammar interpreter. Manish Agarwal and Jon
Cagan’s (1998) coffee maker grammar has been implemented in a Java program.

Mark Tapia (1999) describes well the relationship between shape grammars and
computer implementations:

> Shape grammars naturally lend themselves to computer
> implementations: the computer handles the bookkeeping tasks (the
> representation and computation of shapes, rules, and grammars, and
> the presentation of correct design alternatives) and the designer
> specifies, explores, develops design languages, and selects alternatives.
Current Issues

The interface

The issue of the user interface is one of the most critical in the design of computer implementations for shape grammars. People who use shape grammars or want to learn to use them tend to be visual in their thinking. People who implement software must be good symbolic thinkers; they may also be good visual thinkers. People who think well both visually and symbolically seem to be quite rare.

Quoting Terry Knight (1998):

In most all of the foregoing work, computational problems related to encoding rules and their execution in a computer program took precedence over user interfaces. Thus, most of these systems were not suitable, or ready, for general use by non-programmers, novice users of shape grammars, or design practitioners. Interface issues were addressed in a comprehensive way for the first time in Tapia (1996, 1999). In his work, Tapia distinguishes between "internal" computational aspects of a shape grammar interpreter and "external" interface aspects. Focusing on both aspects, Tapia defines improved computational machinery for a general two-dimensional shape grammar interpreter, along with a simple, intuitive, visual interface.

A student learning to use shape grammars by using an interpreter program is learning both about shape grammars and how to use the program. The two are not the same. The program should be transparent, even helpful. Certainly it should be easier to use the program to try out shape grammars than it is to try them by hand.

Students often learn to use shape grammars with 2 rule or 5 rule grammars. The most complex grammars, like the grammar for Frank Lloyd Wright's prairie houses (Koning and Eizenberg 1981) or for coffee makers (Agarwal and Cagan 1998), are on the order of 100 rules. The optimal interface for a 5 rule grammar probably is different than the optimal interface for a 100 rule grammar. For starters, 5 rules all fit on the screen at the same time while 100 rules don't. The more complex grammars usually work in phases and can be decomposed into sequences of rules for adding the living zone or the filter unit. 100 rules seems to be a complexity barrier that has held for almost 20 years. Perhaps to create grammars of 1,000 rules or 10,000 rules we will need computer tools.

Developing a good interface for a 3D grammar interpreter of course is even more challenging.


**Parametric grammars**

In response to my request for suggestions about computer implementations, Ulrich Flemming wrote:

> I'm still hoping someone comes up with a robust implementation of a PARAMETERIZED SHAPE GRAMMAR interpreter. ... It needs a solid formal foundation ... and a very good GUI that allows for the graphical definition of parameterized shape rules (a tricky, but intriguing proposition). I would use it to set up a laboratory for the experimentation with and investigation of architectural forms.

Chien et al. (1998), students of Flemming’s, discuss their frustrations in attempting to design and implement a program for parameterized shape grammars. They point out difficulties in determining parametric rule application. Is the subshape algorithm for parametric shape grammars a solved (solvable) problem?

**Surprises and the subshape problem**

George Stiny has written eloquently and persuasively on the crucial importance of surprises and emergent behavior for shape grammars (for example, Stiny 1994). And yet, one wonders for the application of shape grammars to solve specific design problems about the importance of surprises and emergent behavior. At issue is whether it is important to solve the general subshape problem in every interpreter, whether a full search needs to be accomplished before any rule can be applied. It seems to me that perhaps 50% of the effort that has gone into shape grammar implementation has been related to this issue. In the previous section the difficulty of the subshape problem for parametric shape grammars was discussed. Even for non-parametric grammars, solving the subshape problem for interpreters of large grammars, of grammars with curves, of three-dimensional grammars, even of grammars with non-90 degree rotations is expensive not just computationally, but in terms of the most critical commodity -- the software designer’s and software implementer’s time. Is this a good investment? Is implementing a shape grammar as a set grammar (Stiny 1982) as has sometimes been done (Knight 1998) or by having the user decide where to apply which rule a better solution in some cases?

**Curves, curved surfaces, curved objects**

Most work on shape grammar implementations has involved straight lines or planar surfaces. Yet almost all objects in life, even artificial objects, have curves and curved surfaces. Allowing curves of a certain family (for example, second degree polynomials) introduces considerable complexity. Allowing arbitrary curves doesn’t seem feasible. What are the best ways to deal with curves and curved surfaces and objects?
Underlying representations

Many different underlying representations for shapes have been used in computer implementations. The user only cares to the extent the selection of representation affects the behavior of the program. Clearly the dimensionality and vocabulary of shapes allowed in the grammar affects the representation decision. Is the subshape problem to be solved? Some representations work in theory but in practice preclude finding subshapes, for example in rotated shapes because of the irrationality of the resulting coordinates. ("In theory, theory and practice are the same, but in practice they are not.")

For the software designer or programmer the underlying representation(s) is one of the key decisions that is made. Is there some particular representation that should become the standard or some standard class library that should be used or developed? This might allow for people to more easily build on others' work. What have we learned?

Extensions to shape grammars

What are useful extensions to shape grammars? What are vital extensions to shape grammars? How can they be implemented in a shape grammar program? Important examples of extensions include colors (Knight 1989), weights (Stiny 1992), and sortal grammars (Stouffs and Krishnamurti 1997).

Difference between proof-of-concept software and production software

There is a world of difference between developing software that works in the lab and software that works on a day-to-day basis at customers' sites. Getting the software to the point that the demo works is all of the glamour but 10% of the work. Software needs to be field tested. Portions need to be redesigned and re-implemented. Online documentation needs to be provided for the users. Bugs need to be found and fixed. Meanwhile the platform -- language, operating system, hardware -- keeps evolving. Software that worked on release 2.3 of the language or operating system or browser might not work on release 3, despite everyone's claims of upwards compatibility. A newly introduced CPU or video card might throw off a library used in the program. Programs require constant maintenance to remain usable.

Institutionally, software created by students is notoriously difficult to keep running over longer periods of time. The students receive their degrees and are off. If the program was created by one student, perhaps the student is interested in maintaining it afterwards. If a group of students is involved then the software was probably the brainchild of the faculty sponsor. It is difficult to get new students to
learn and maintain software written by previous students. Grants expire. People get new ideas and want to start over again from the beginning. Maintaining and incrementally improving software is not very glamorous and doesn't lead to many dissertations. The result is programs with short lives.

Almost all of the software development to date has been university based. Perhaps the research labs of large companies might be interested in becoming more involved. Another approach is to spin the software off into startup companies. Is there a market yet for shape grammar based software? I am waiting for the first shape grammar billionaire. (We took shapegrammar.org for the shape grammar research site. The first shape grammar entrepreneur can take shapegrammar.com) In today's climate we might have patented the whole idea of shape grammars.

Stable, usable software requires a stable infrastructure to support it. Practicing designers and engineers want stable software with updates and technical support and a strong likelihood that this will continue for five or ten years. We're not there yet in the development cycle but it's a serious issue to think about.

The Big Enchilada or one piece at a time (or both)

One can imagine The Big Enchilada, the shape grammar interpreter program with everything -- a slick interface, parametric application, 2D and 3D, curves, extensions, all kinds of tools and goodies for the user. The program would be qualitatively more useful than today's programs. It might well take an order of magnitude more effort than today's programs. It seems time to launch a major computer implementation project. I have in mind a project with a post-doc and three or four graduate students working over several years with a budget on the order of a million dollars.

In addition, we need to encourage the one and two person efforts that advance the field as in the past. We always need the individual, often a graduate student, with a burning idea about how to do something the right way and the drive to implement it in a program.

Acknowledgments

Not having been active in the field for the past 20 years (!) I had a lot of catching up to do. My thanks to Jon Cagan, Scott Chase, Hau Hing Chau, Ulrich Flemming, Terry Knight, Ramesh Krishnamurti, George Stiny, Mark Tapia, and Bob Tilove for their very helpful emails and conversations. To the extent I didn't catch up or get matters right or recognize important work or even get ahead of the curve, I apologize. I suspect I will be set right at the conference.
References


Stuny G, 1975, Pictorial and Formal Aspects of Shape and Shape Grammars (Basel: Birkhauser)


