Environment and Planning B: Planning and Design 2014, volume 41, pages 110 – 137

doi:10.1068/b39107

A paradigm for interpreting tractable shape grammars

Kui Yue
Microsoft Inc., Redmond, WA 98052, USA; e-mail: kuiyue@microsoft.com
Ramesh Krishnamurti¶
School of Architecture, Carnegie Mellon University, Pittsburgh, PA 15213-3890, USA;
e-mail: ramesh@cmu.edu
Received 19 June 2012; in revised form 14 December 2012

Abstract. Shape grammars are, in general, intractable. Even amongst tractable shape grammars, their characteristics vary significantly. This paper describes a paradigm for practical general shape grammar interpreters, which aim to address computational difficulties posed by parameterization. The paradigm is expressed in terms of frameworks each comprising an underlying data structure, manipulation algorithms, and a metalanguage. The approach is illustrated through three exemplar frameworks.

Keywords: shape grammars, tractability, framework, computer implementation

1 Introduction

Shape grammars provide a rule-based, visual, and algorithmic approach to spatial designs (Stiny, 2006). Remarkably, since their inception (Stiny and Gips, 1971), there have been few computer programs that assist with using shape grammars in design. In a previous paper (Yue and Krishnamurti, 2013), we showed that shape grammars may not halt and can have exponential language space. Moreover, even practical shape grammars—halting grammars with polynomial language space—can be intractable. This implies that algorithms to interpret shape grammars fall into two categories: those that handle special shapes; and other, more general, algorithms with worst-case exponential time complexity, which are practical only for shapes of small size. The implication is that the best we can achieve, in practice, is to design and implement a shape grammar interpreter capable of handling a subset of grammars. In this paper we describe an approach to implementing a class of tractable shape grammars.

Even amongst tractable shape grammars, their characteristics vary significantly. One possible explanation for this is that the shape grammar formalism covers a wide spectrum of designs stemming from different disciplines. This richness in variety is shown indirectly by a number of well-known categories of shape grammars such as subshape-driven versus marker-driven, nonparametric versus parametric, or rectilinear versus curvilinear. The variety can also be observed in the details of basic operations of t, $-$, $+$, $\leqslant$, and R; their diversity forms different algebras [for example, U_0, U_1, U_2, and U_3 (Stiny, 1991)], on top of which further categories of shape grammars can be defined. For example, Knight (1999) examines variety by criteria of restrictions on rule format and ordering. Six types of shape grammars are distinguished: basic, nondeterministic, sequential, additive, deterministic, and unrestricted grammars. Such varieties are tangibly noticeable even when we focus on a specific subset of tractable grammars: for example, those based on two-dimensional rectilinear (with limited curved) shapes. The following is a comparison of three such examples, each based on the square or rectangle as the dominant vocabulary shape.

¶ Corresponding author.

The Baltimore Rowhouse grammar (Yue and Krishnamurti, 2008) captures a specific building style. Other examples of this type include the Queen Anne grammar (Flemming, 1987) and Frank Lloyd Wright's Prairie House grammar (Koning and Eizenberg, 1981). These are all parametric shape grammars, in which rule application does not depend on emergent shapes. Markers drive shape rule application, and configurations are rectangular or can be approximated as such. Moreover, parameterization is often limited to the height or width of a room, or to the ratio of a room partition. The central manipulation unit is a room (or space). Shape rules typically add a room, partition a room, or refine a room by adding windows, doors, etc. Figure 1 illustrates typical generic schema.

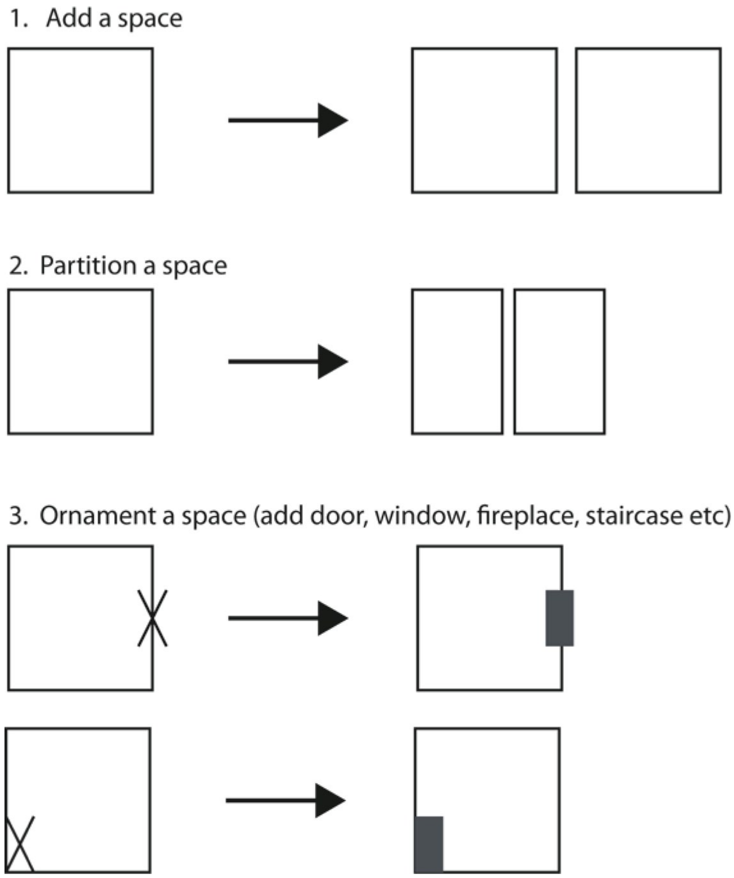


Figure 1. Typical schema found in parametric grammars describing floor plan layouts.

Figure 2 illustrates the rules and a sample derivation of a polyomino (Golomb, 1994) grammar based on a structure grammar (Carlson et al, 1991), an augmented variation of a formalism known as a set grammar (Stiny, 1982). The grey square in rule 3 depicts an exclusionary condition.

The Kindergarten grammar (Stiny, 1980) is another example of this type of grammar. These grammars treat designs as symbolic objects; designs are enforced to be elements of sets from which they are formed. Thus, integrity of the compositional units in designs is preserved, as these parts cannot be recombined and decomposed in different ways. This is in contrast to those grammars in which shape elements are decomposed and recombined freely so that new shapes can emerge: for example, the shape grammar shown in figure 3. Here we are essentially manipulating symbols in a two-dimensional space, thus making these grammars amenable to computer implementation. The resulting shapes are simply replacements of internal symbols that occur at the final stage, for the purpose of visualization.

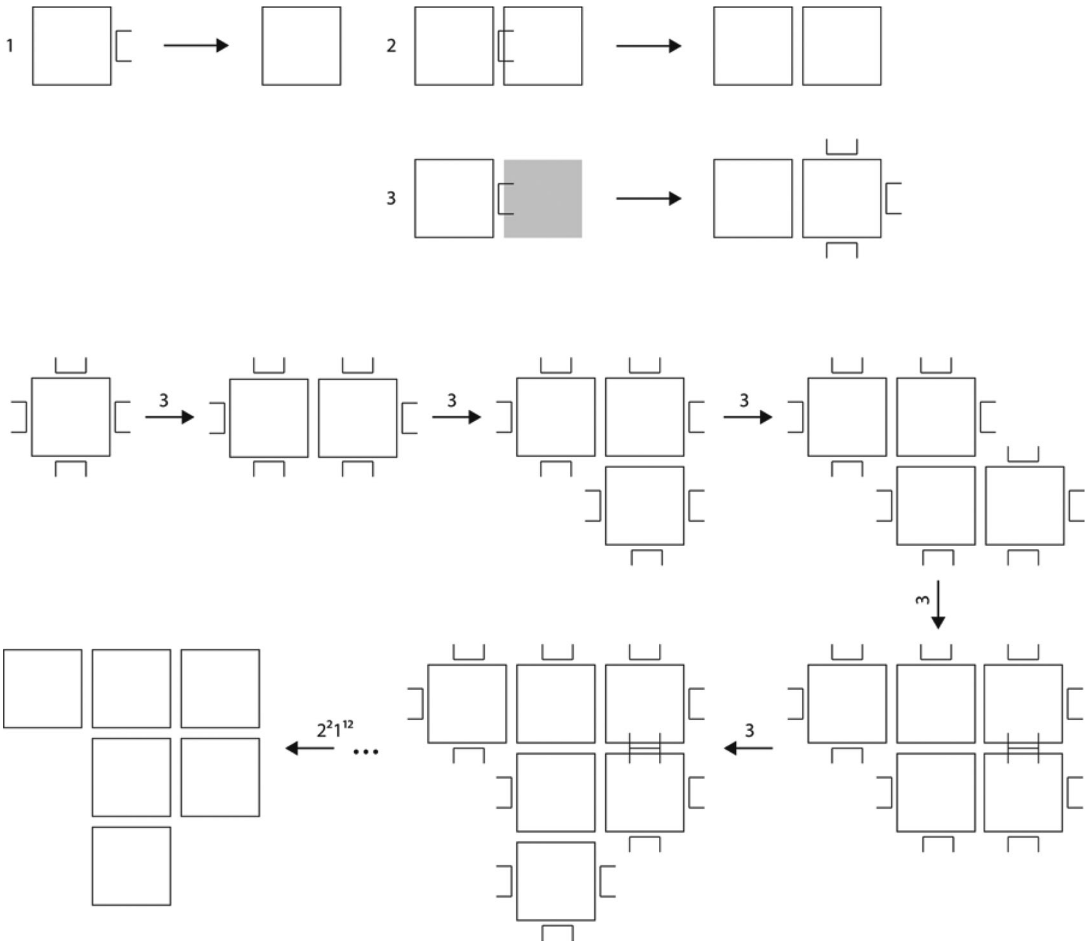


Figure 2. Rules of a polyomino grammar and sample derivations.

Figure 3 shows the rules and sample derivations of a classic Stiny (2011) shape grammar comprised of three shape rules. The first rule replicates a square forming squares overlapping a central square; the second rotates a square, and the third moves a square. In rule application sides of squares are cut into meaningful pieces of various lengths, that are related in a definite sequence, and squares move wherever one wants—left or right and up or down—yielding a variety of shapes that exceed expectations that might be associated with the square. This is an example of the kind of shape grammars where shape elements are free to be decomposed and recombined so that shape emergence is an important feature during shape rule application. In the Knight and Stiny (2001) nomenclature, computation here is nonclassical.

Table 1 shows a comparison of certain characteristics of the three grammar examples discussed above, of their importance to computer implementation. The variety in this table shows that, even for tractable shape grammars, it is still difficult to come up with the design of a single uniform interpreter. Remarkably, however, of the three grammars, the classic Stiny grammar is the most straightforward to implement (Krishnamurti, 1982); shapes rules, although subject to similarity transformations, are nonparametric—that is, shapes are fixed in their geometry. The polyomino grammar can be realized, symbolically, as shapes associated with symbols, and thus implemented in a straightforward fashion using attributed strings. Parametric shape grammars as exemplified by the Rowhouse grammar require further consideration.

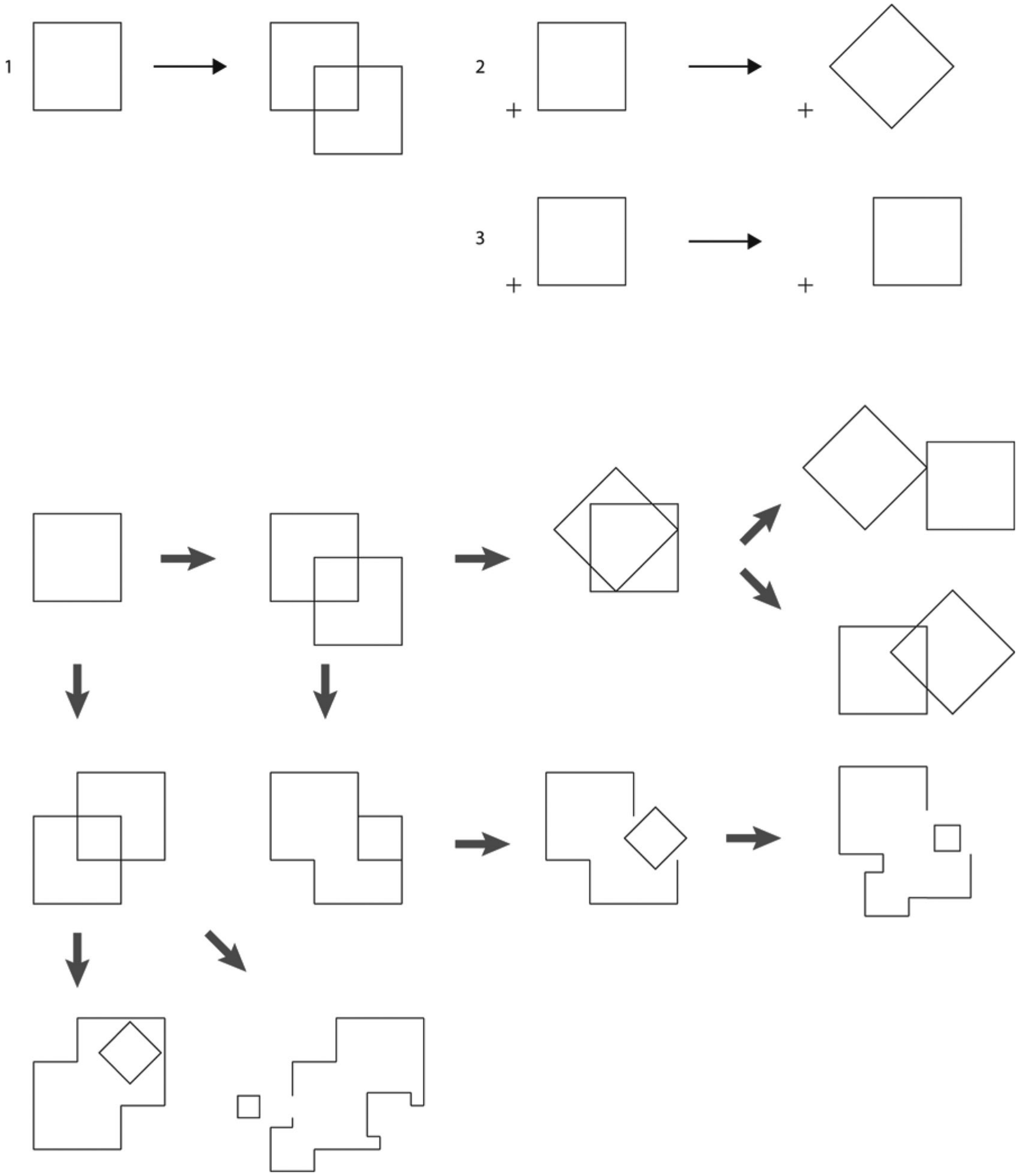


Figure 3. Rules and derivations of a grammar with emergence. Redrawn by the authors. Adapted from Stiny (2011).

Table 1. Comparison of characteristics important for computer implementation.

Grammar	Driver	Emergence	Manipulation unit	Parametric	Context
Rowhouse	marker	no	room	yes	sensitive
Polyomino	marker	no	symbol	no	sensitive
Stiny classic	subshape	yes	shape element	no	free

2 An approach to practical grammar interpreters

More often than not it is relatively straightforward to implement an interpreter for a special class of shape grammars: for example, grammars that capture building styles. As we cannot handle intractable shape grammars, why not focus on dealing with as many tractable shape grammars as possible, employing a concept, in spirit, comparable with or similar to approximation algorithms (Cormen et al, 2004). Following this idea, we propose an approach for practical, 'general' shape grammar interpreters, as shown in figure 4. The approach comprises a set of subinterpreters, one for each class of tractable shape grammars.

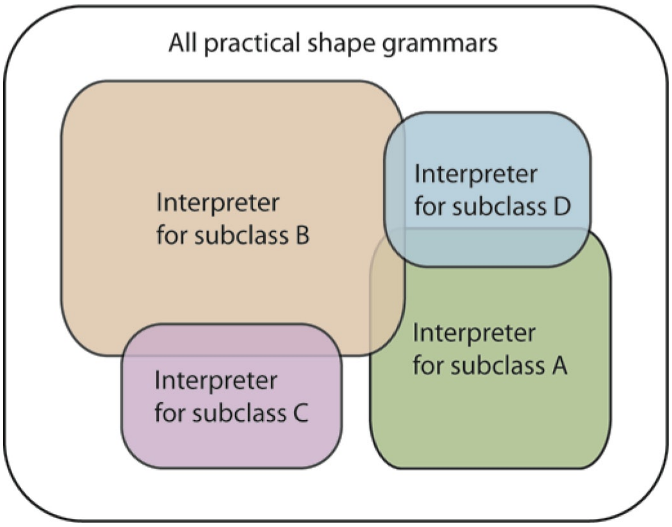


Figure 4. [In colour online.] An approach for a practical 'general' shape grammar interpreter.

In this way, collectively, most parametric shape grammars can be covered.

This approach is a perfect subject for applying techniques of object-oriented design: in particular, modularity, polymorphism, and inheritance (Grady et al, 2007). The top-level formalism of shape grammars can be implemented as abstract classes and methods, which are materialized in the subinterpreter for each class. The shared functionalities—for example, interfaces—can be implemented as part of the top-level infrastructure such that developers for the subclass interpreters are free from unnecessary redundant work.

The approach can be promoted to follow the successful model of the Eclipse project (http://www.eclipse.org), which aims at an open development platform comprising extensible frameworks, tools, and runtimes for building, deploying, and managing software across the lifecycle. By building a similar platform backed up by the above approach, researchers geographically dispersed around the world can collaboratively work on the same platform; each freely developing their idea as an *add-in*, thus contributing to their effort. Designers can freely download and exploit up-to-date grammar systems, testing new design ideas, suggesting new features, and reporting bugs. Such a platform fundamentally changes the past discrete structure of the research of implementing a shape grammar interpreter (Chau et al, 2004); duplicated work is significantly reduced, and the scope of the users greatly expanded.

It should be noted that the proposed approach depends on a classification of shape grammars into subclasses. Moreover, the classification is considered to be 'better' when the number of subclasses is smaller, and when, simultaneously, the scope covered, collectively, is larger. Here, the following research question immediately emerges: *what is the most optimal way of classifying shape grammars?*

3 Classification of shape grammars

There are many different ways of classifying shape grammars, each from a distinct perspective. For instance, shape grammars could be classified based on relatively obvious shape properties such as, two-dimensional versus three-dimensional shapes, rectilinear versus curvilinear shapes, and so on. Classification could be based on definition: for example, structure grammars (Carlson et al, 1991) or set grammars as defined by Stiny (1982). Classification from the field of formal linguistics could be introduced: for example, finite versus infinite grammars based on the size of the underlying language; that is, based on the size of the design space. Classification could also be based on properties of shape rules and/or their rule application: for example, nonparametric versus parametric shape grammars based on how shape rules are specified, marker-driven versus subshape-driven shape grammars based on how shape rule application is controlled, or context-free versus context-sensitive shape grammars based on neighbourhood dependency when shape rules are applied. Knight's (1999) six types fall into this latter category. However, none of the categories is truly appropriate. This is because each category is so broad as to comprise grammars with even greater variety. In other words, the criteria upon which shape grammars have been classified have not been based on elements that are fundamental to grammar interpretation or implementation.

Elements fundamental to any computer program are algorithms and data structures; this is evident from the title of Wirth's (1978) classic textbook *Algorithms + Data Structure = Programs.* This is equally true for shape grammar implementations. An implementation is, in essence, a computer program that manipulates the internal representation of a design—data structure—by a set of operations. The basic shape grammar operations of t, $-$, $+$, $\leqslant$, and R operate on a data structure, and details vary from one data structure to another. The exact procedure of searching for matching candidates depends on the data structure; so too does exact match verification. The underlying data structure, in turn, determines how to carry out these operations, and how efficient they are. Moreover, data structure fixes in advance the power of the shape rules built on top of them. Stiny (1994, page s53) remarks, "the antecedent definition of meaning parts and units limits the subsequent possibilities for inquiry … . Descriptions fix things in computations, and nothing is ever more than its description anticipates explicitly."

The argument can also be seen from a cognitive standpoint. The design of a data structure is simply a particular view of the underlying subject, which is present-at-hand. According to Winograd and Flores (1986, page 97), "Whenever we treat a situation as present-at-hand, analyzing it in terms of objects and their properties, we thereby create a blindness." Things covered by any current data structure correspond to those that are seen; the blind parts are left to other data structures.

In line with this argument, the underlying data structure used to support algorithms for the implementation fundamentally characterizes the corresponding class of shape grammars. Assuming that there is always a power difference between any two data structures adopted, and if no other data structure subsumes any of the adopted data structures, then we have reached an optimal classification.

4 Augmented practical 'general' approach

The 'general' approach comprises a set of subinterpreters, one for each class of shape grammars. Moreover, each class is backed up by a data structure, which reflects the internal characteristics of the corresponding subset of shape grammars.

Apart from the internal characteristics of shape grammars, there are other factors that influence computational tractability: for example, how shape grammars are designed and described. Traditionally, a shape grammar is designed to simply and succinctly describe an underlying building style, with little consideration given to how the grammar can be implemented. For example, as is often found in the literature, such descriptions of the form "If the back or sides are wide enough, rule 2 can be used …" are inherently countercomputable. As a result, in order to translate this into programming code, shape rules have to be quantitatively specified; furthermore, there has to be enough precision in the specification to disallow generation of ill-dimensioned configurations.

Closer examination also shows that there may be more than one way to describe a particular shape rule; it is possible that a certain way is easier to compute, and another might be computationally intractable. As a result, it is desirable to design an application programming interface-like framework to support the design of shape grammars; then, shape grammars that follow the framework are guaranteed to be computationally tractable. Such a framework is built on top of an underlying data structure and basic manipulation algorithms. Moreover, for the ease of code translation, a metalanguage built on top of the basic manipulation algorithms should also be developed. As grammars in different classes typically have differing underlying structures, the appropriate underlying data structure for the framework will be different. Consequently, the overall framework comprises a series of subframeworks, one for each class of shape grammars, as shown in figure 5.

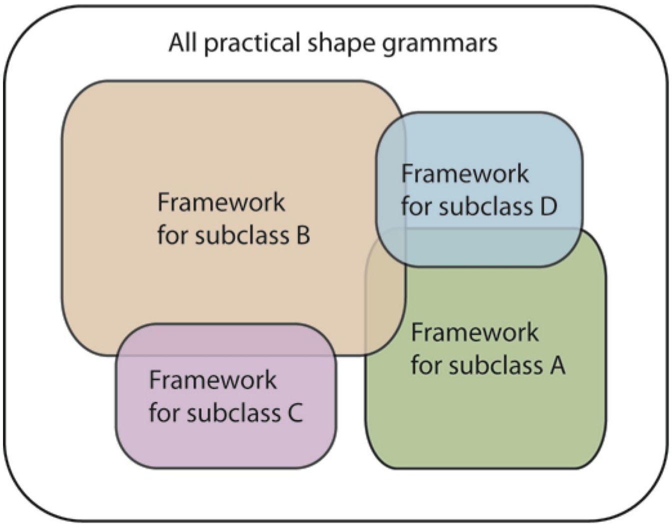


Figure 5. [In colour online.] A subframework for each subclass of tractable grammars.

5 Three exemplar subframeworks

In this section we examine three subframework exemplars as a way of illustrating the approach to practical, 'general' interpretation of shape grammars.

In selecting subframeworks to illustrate the approach, it was deemed advantageous to initially select a subframework for a subclass of shape grammars with the largest population. It turns out that shape grammars that capture building style happen to be good choice. Of all the shape grammar applications reviewed by Chau et al (2004), about half deal with architectural plans. Moreover, conventional buildings (namely, buildings with rectangular spaces or dominated by such spaces) are often the subject matter. Consequently, a subframework for shape grammars capturing corpora of conventional building types (namely, the rectangular subframework) is chosen.

Two-dimensional polygons are another kind of shape widely used in shape grammars: for example, Chinese ice-ray lattices (Stiny, 1977) and Hepplewhite-style chair back grammars (Knight, 1981). Thus, a subframework for two-dimensional polygonal shapes is also chosen. From the appearance, such a subframework can be viewed as an extension of the rectangular subframework. Yet, as is shown in section 7, the extension is not straightforward. In fact, both application context and basic manipulations are quite different.

The rectangular subframework relies on a graph-like data structure. This suggests that there might be a relationship between shape and graph grammars; the former has been investigated mainly in the field of design, in particular, architectural design; the latter has been widely studied in computer science. The comparison in section 7.1.3 shows that both differ significantly although there is noticeable commonality. Graph grammars are most useful when dealing with those shape grammars which are dimensionless and context free. Accordingly, we consider graph grammars as a subframework for implementing dimensionless, context-free shape grammars. For reasons of space, the rectangular subframework is explained in detail, while the other two subframeworks are discussed in brief with less detail.

6 Rectangular subframework

Conventional buildings are buildings with rectangular spaces or dominated by such. A rectangular space is specified by a set of walls in such a way that the space is considered rectangular by the human vision system. Amongst many variations, a space can be specified by four walls jointed to one another, four disjoint walls, three walls, or framed by four corners. See figure 6.

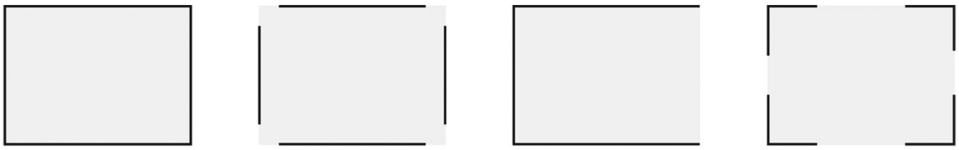


Figure 6. Examples of a rectangular space.

Spaces (rooms) are central to buildings—whence, to shape grammars that describe building styles. For shape grammars capturing corpora of conventional building types, shape rules are parametrically specified in such a way that parametric subshape recognition consists, typically, of searching a special room under certain constraints, and actually matching labels. Such grammars generally start with a rough layout; details, such as openings and staircase, are added at a subsequent stage. There are two main ways of generating a layout: space subdivision and space aggregation. Combination of the two is also possible.

6.1 A graph-like data structure

The interpreter needs a data structure to represent layouts with rectangular spaces: that is, a data structure that contains topological information of spaces as well as concrete geometry (in this paper, two-dimensional) data of a layout including walls, doors, windows, and staircases. It needs to support viewing a layout as a whole, viewing a layout from a particular room with its neighbourhood, or simply focusing on a particular room itself. Moreover, the data structure needs to support Euclidean transformations augmented by both uniform and anamorphic scaling.

A graph-like data structure has been designed to specify such rectangular spaces. There is a boundary node [coloured blue (light gray) and tagged by the label B] for each corner of the rectangular space, as well as a node for each endpoint of a wall. These nodes are connected by either a wall edge (solid line) or an empty edge (dotted line). A central node [coloured red (dark gray) and tagged by the label R] represents the room corresponding to the space, and connects to the four corners by diagonal edges (dashed lines). It is needed for manipulating boundary nodes of room units, such as dividing a wall through node insertions (coloured white and tagged by the label W), for creating an opening in a wall by changing the opening's edge type to *empty*, and so on. More information about a room is recorded in the room node: for example, a staircase within the space. Windows and doors

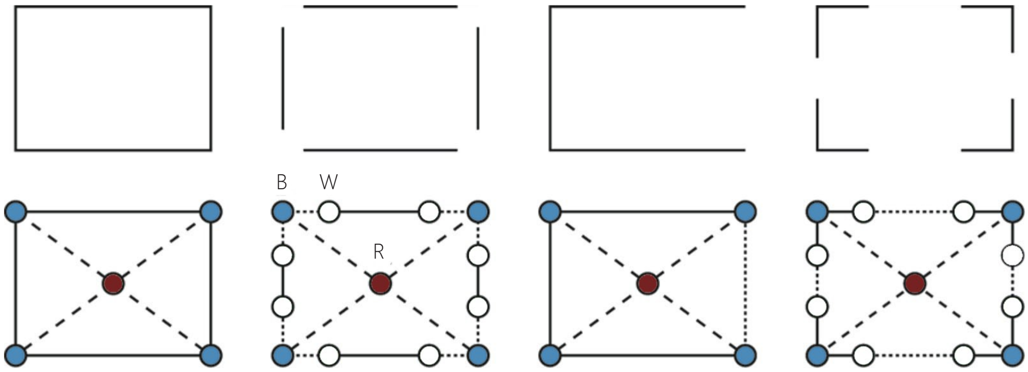


Figure 7. [In colour online.] Graph-like data structure representation of a space.

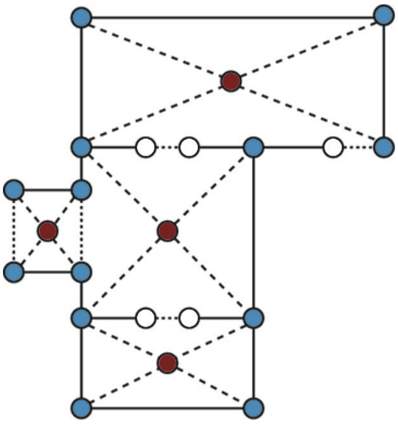


Figure 8. [In colour online.] A layout represented by a set of graph units.

are assigned as attributes of wall edges. Further, unlike traditional graph data structures, the angle at each corner is set to be a right angle. A node has at most eight neighbours. Figure 7 illustrates the graph-like data structure for the different variations of a space given in figure 6. A set of such graph units can be combined to represent complex layouts comprising rectangular spaces. See figure 8.

6.2 Transformations of the graph-like data structure

The target layout is assumed to comprise only rectangular spaces, and the allowable transformations are Euclidean with uniform and anamorphic scaling. As shape rule application is label-driven, translation is automatically handled. The graph-like data structure is capable of easily handling uniform and anamorphic scaling: by, firstly matching room names, then labels on corner nodes, and, lastly, by comparing possible room ratio or dimension requirements.

As a result, only rotations and reflections remain to be considered. As the spaces are rectangular, rotations are limited to multiples of 90° and reflections are either vertical or vertical. Moreover, a horizontal reflection can be viewed as a combination of a horizontal reflection and a rotation. Hence, any combination of reflections and rotations is equivalent to a combination of horizontal reflections and rotations. Consequently, the following transformations are all we actually need to consider:

- `R0`: default; no rotation, with possible translation and/or scale.
- `R90`, `R180`, `R270`: a rotation of 90°, 180°, and 270°, respectively, with possible translation and/or scale.

- RR0, RR90, RR180, RR270: (first a rotation of 0°, 90°, 180°, and 270°, respectively, followed by a vertical reflection) vertical reflection, horizontal reflection, or their combination, with possible translation and/or scale.

As shown in figure 9, transformations can be implemented on the data structure by index manipulation. Each of the eight possible neighbours of a node is assigned an index from 0 to 7; indices are then transformed simply by modulo arithmetic. For example, index +2 (modulo 8), counterclockwise rotates neighbour vertices through 90°. Other rotations and reflections are likewise achieved. By viewing the original neighbour relationship for each node with the transformed indices, we obtain the same transformation of the whole graph. By taking advantage of this fact, we need to manipulate only the interior layout instead of the left-hand side of every shape rule. Consequently, we only need to consider how to

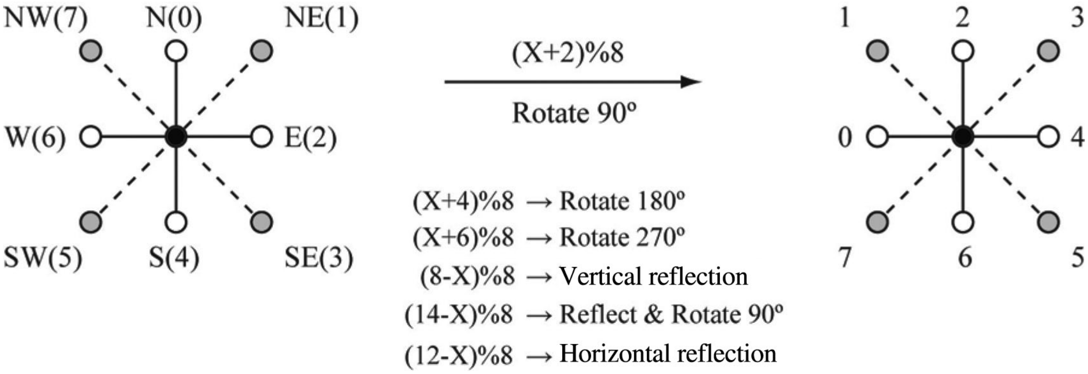


Figure 9. Transformations of the graph-like data structures.

apply shape rules with the default transformation, which is automatically applicable to the configuration under different possible transformations. This gives the same results, but is much simpler to achieve.

6.3 Common functions for the graph-like data structure

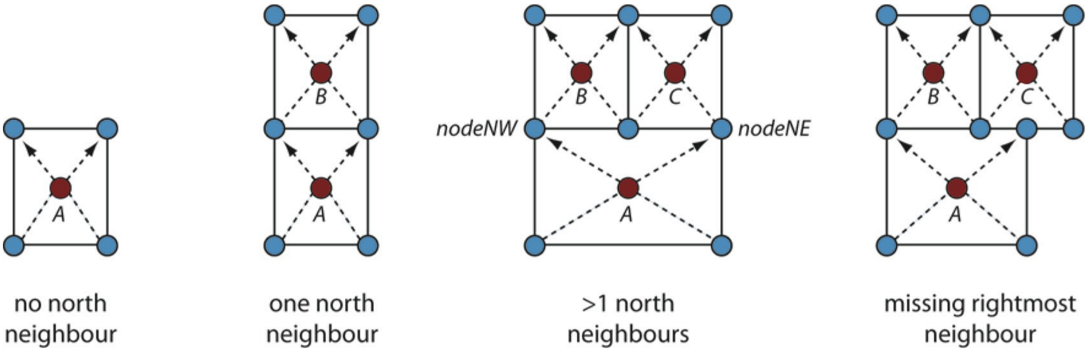
With the graph-like data structure, a layout is represented by an eight-way doubly linked list formed by nodes and edges. Shape rule application manipulates this structure, and a set of common functions shared by the shape rules can be identified.

Some common functions are relatively easy to carry out: for example, splitting a room into two and merging two rooms into one, or finding a room with a given name. Others are more complicated; examples include finding the north neighbour(s) of a given room, and finding the shared wall of two given rooms. The following section describes the algorithm and pseudo-code for these examples.

6.3.1 *Finding the north neighbour(s) of a given room*

A room may have zero, one, or more north neighbours (figure 10), which can be represented by a list of room nodes. Intuitively, to find the north neighbour(s) of A, we start by finding A's northeast corner node, nodeNE, and northwest corner node, nodeNW. Then, we traverse through each corner node from nodeNE (inclusive) to nodeNW (exclusive) along the westerly direction to find its north neighbours. All north neighbours found are desired room nodes. For example, in figure 10(c), the north neighbours found are B, and C. However, as shown in figure 10(d), this intuitive algorithm will miss the rightmost neighbour room, that is, when two neighbour rooms only partially overlap so that nodeNE is on the south edge of that neighbour room, and is not the desired end node. Therefore, we need to modify the intuitive algorithm to have the correct start and end nodes to loop through.

It can be proven that nodeNW is always the correct end node as a north neighbour B has to overlap with room A, which means room B must has a southeast corner node, nodeSE, at



(a) (b) (c) (d)

Figure 10. [In colour online.] Different cases for the north neighbour(s) of a room.

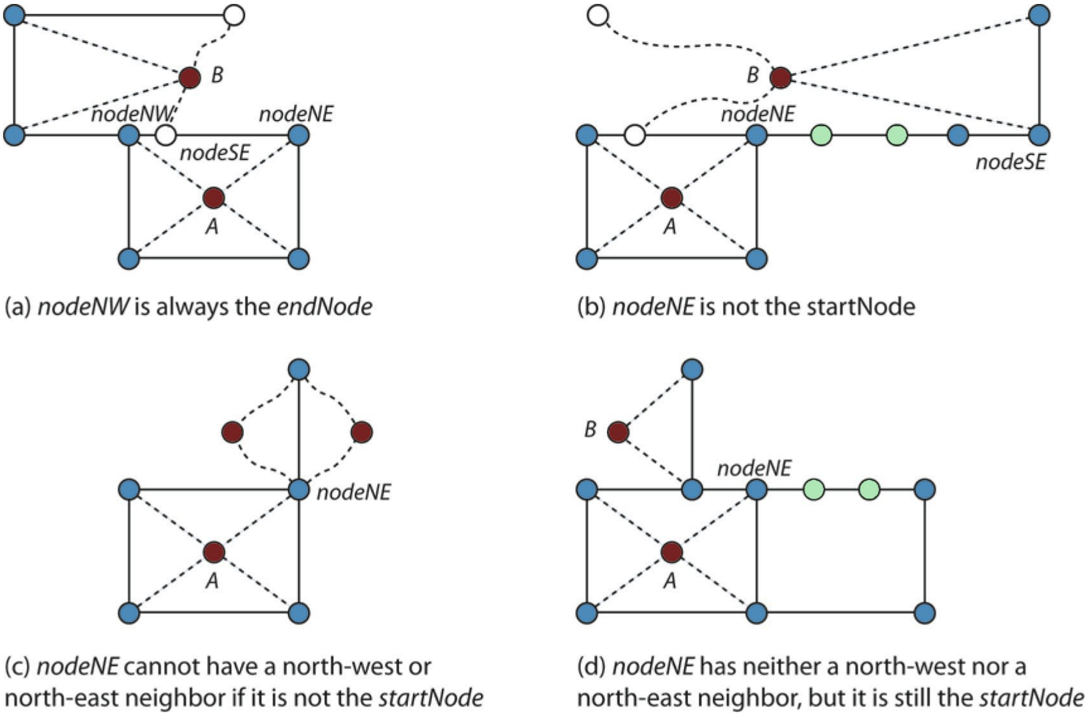


Figure 11. [In colour online.] The start and end nodes for finding neighbour room(s).

the right side of nodeNW [figure 11(a)], or is nodeNW [figure 11(c)]. Otherwise, B is not a north neighbour of A.

The algorithm for finding the north neighbours of a given room is shown in figure 12. Finding the north neighbour(s) of a given room is a special case of finding any neighbour(s) of a given room. It turns out all that finding neighbour functions in the other three directions can be implemented as finding the north neighbour(s) under a certain transformation. For example, the east neighbour(s) of a given room is the same as the north neighbour(s) of the given room under an R90 transformation.

6.3.2 *Finding the shared wall of two given rooms*

In the data structure, the shared wall of two given rooms is represented as a list of nodes connected by edges; the simplest form of a shared wall is given by two nodes connected by an edge. The pseudo-code for the algorithm is given in figure 13.

For any two given input room nodes, A and B, in general, the rooms may not be neighbouring rooms at all. If, however, A and B are real neighbours, B can be in any one of four

```
findNorthNeighbours (A, T)
    (All operations related to directions are under transformation T)
    endNode  ← north-west neighbour of A
    nodeNE ← north-east neighbour of A
    startNode ← nodeNE
    if nodeNE has neither a north-west nor north-east neighbor
    then {
        search for a right neighbour, node, of nodeNE, with a north-west neighbour
        if found
        then {
          startNode ← node
            go through each node between startNode (inclusive) and endNode (exclusive),
            and get all north-west neighbours, neighbours
          }
     }
    return neighbours

findEastNeighbours (A)  // Other neighbours are likewise defined
      return findNorthNeighbours(A, R90)
```

Figure 12. Algorithm for finding the north neighbour(s) of a room.

directions from A. Therefore, it is necessary for the algorithm to test all four sides of A; for each particular side, it is simple to test whether B is in the north neighbours under a given transformation T.

If B is determined as a neighbour of A at a given side, the exact start node, wStart, and end node, wEnd, need to be further determined. The edge from the northeast node, nodeNE, to the north-west node, nodeNW, of room A under transformation T is guaranteed to be the wall of room A, but not necessarily the wall of room B [figure 14(a)]. As a result, wStart may be actually a node to the right of nodeNE. This node is found by traversing from nodeNE to nodeNW testing whether B is its northwest neighbour or not. Similarly, wEnd may be actually a node to the left of nodeNW. This node is found by traversing from nodeNW to nodeNE testing whether B is its northeast neighbour or not.

6.4 Metalanguages

All common functions collectively form an API (application programming interface), which expresses the capability of its underlying data structure. Such an API facilitates the design of shape rules, in a way similar to how a programming language API, say, the Java API helps in building Java applications. Moreover, grammar designers can apply the API to ensure the computability of their designed grammars.

A metalanguage is a language with which to describe another language. We employ a metalanguage to express shape rules. Normally, shape rules are described pictorially, which is inherently ambiguous, and difficult to translate to a computer program. Equally, describing shape rules in a programming language is likewise cumbersome for the typical designer who is creating the shape grammar. A metalanguage serves as a middleman helping to express shape rules in a manner formally more rigorous than pictorial description, yet closer in form to natural language. Every metalanguage is defined relative to its subframework. A metalanguage facilitates manual translation to computer code. Ultimately, of course, it would be preferable to have the metalanguage be automatically translated into the target programming language, similar in spirit to MetaL (http://www.meta-language.net/faq.html).

The API supports descriptions of grammars via expressions in the metalanguage; that is, shape rules can be designed in a rigorous fashion so as to be easily translated into pieces of code, the ultimate format by which to interpret shape rules. Essentially, the metalanguage is a set of function calls, which are predefined in the API.

```
findSharedWall (A, B)
    transformations ←{R0, R90, R180, R270}
    foreach transformation in transformations {
        results ← findSharedNorthWall (A, B, transformation)
        if results is not null
        then return {results, transformation}
    }

    return null


findSharedNorthWall (A, B, transformation)
    neighbours ← findNorthNeighbours(A, transformation)
    if B not in neighbours
    then return null
    nodeNE ← north-east neighbour of A
    nodeNW ← north-west neighbour of A
    wStart ← null
    wEnd ← null
    for each node from nodeNE to nodeNW {
        if north-west neighbour of node is A
        then {wStart ← node; break; }
    }
    if wStart is null
    then {wStart ← nodeNE}                // [figure 14(b)]
    foreach node from nodeNW to nodeNE
        if north-east neighbour of node is B
        then {wEnd ← node; break; }
    }
    if wEnd is null
    then {wEnd ← nodeNW}                  // [figure 14(b)]
    return {wStart, wEnd}
```

Figure 13. Algorithm for finding the shared wall of two neighbouring rooms.

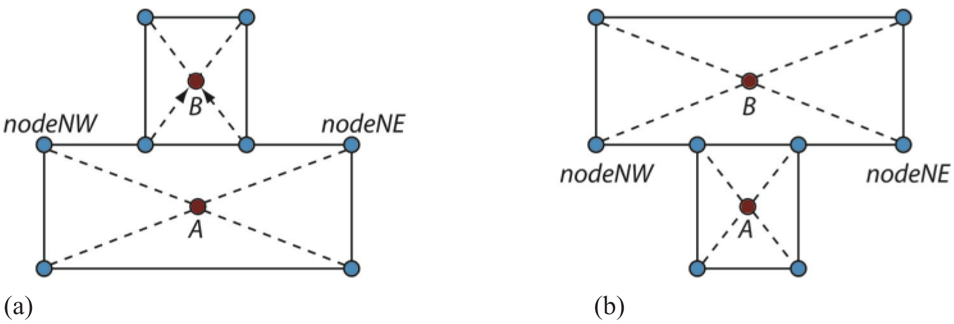
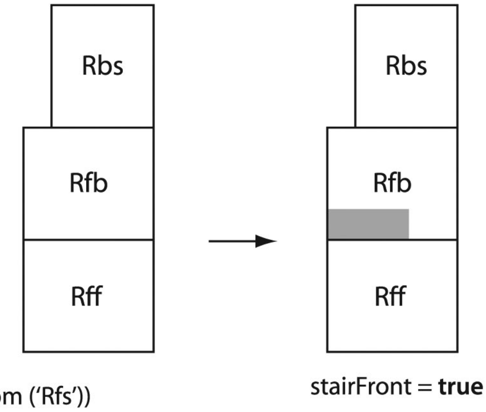


Figure 14. [In colour online.] Finding wStart and wEnd.

This rule adds a hall way centered about the front door to the front block

Precondition: '*Rff*' exists, and is three-bay (2 windows and 1 door).

Transformation: N/A




```
if ( roomExists ('Rff') && numberOfBays () == 3)
then {
    rooms=getRoomsBetween ('Rfb', 'Rff')
    w = hallwayWidth( getDoor ('frontDoor'), getRoom ('Rfs'))
    foreach (room in rooms) {
        room.horSplit (name=room.getName, width=*, name='tmp', width=w)
        hf.merge (getRoom ('tmp))
    }
    hf.name('Hf')
}
```

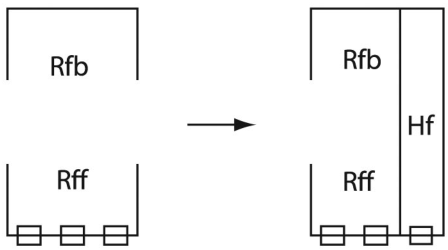
This rule adds a staircase to room 'Rfb.'

Explicit condition: No staircase.
'*Rfb*' and '*Rff*' exist and are neighbours.

Implicit condition: No 'SfS.'
Width of front block is ≤ 18'.

Overall condition: *stairFront* is false.
'*Rfb*' exists. No '*SfS*.'
Width of front block is ≤ 18'.

Transformation: N/A




```
if (! stairExists() && ! roomExists('SfS') && roomExists('Rfb') && getFrontBlock().width ≤ 18')
then {
    room('Rfb').addStaircase(position='bottom&crossFrontDoor', width=6,  height=4, getFrontDoor())
}
```

Figure 15. Example rules in the rectangular subframework and their metalanguage.

Figure 15 shows two such examples. The metalanguage is in the form of an *if–then* statement; the *if*-part determines whether the rule is applicable or not; the *then*-part specifies how to do the rewriting.

7 Polygonal subframework

Geometrically, the polygonal subframework may appear, quite simply, to be an extension of the rectangular subframework. Closer examination actually tells a different story; both the typical application context and the basic manipulations of this polygonal subframework are distinct from the rectangular subframework.

While the rectangular subframework works for shape grammars describing building layouts, the polygonal subframework does not. The reason is that the majority of building spaces are rectangular rather than polygonal. Instead, shape grammars involving polygonal shapes are more common in describing other kinds of designs: for example, Chinese ice-ray lattices (Stiny, 1977), Hepplewhite-style chair backs (Knight, 1981), as well as abstract paintings (Knight, 1989), see, for example, the nonrepresentational paintings of Fritz Glarner. Such shape grammars are typically parametric and marker driven. The central manipulation is subdivision, which is the theme selected for the polygonal subframework. Besides subdivision, there are other auxiliary manipulations, such as filling colours, inscribing to the initial shape with a shape of triangle, pentagon, hexagon, and so on (Stiny, 1977). Such auxiliary manipulations cannot be generated by subdivision and are handled in a special way, by adding extra functions, or by other means, for example, treating the shape to be inscribed as part of the initial shape.

Subdivision is a procedure for dividing a polygon into two smaller ones by a 'cutting' line, which is a straight-line segment, a joint line of two segments, or a polyline of multiple line segments. As a result, transformations become unnecessary, since an equal effect can be achieved by changing the coordinates of the endpoints of the cutting line. For example, figure 16 shows a shape rule in which a triangle is subdivided into a smaller triangle and a quadrilateral. Shape rule (b) is a vertical reflection of the shape rule (a).

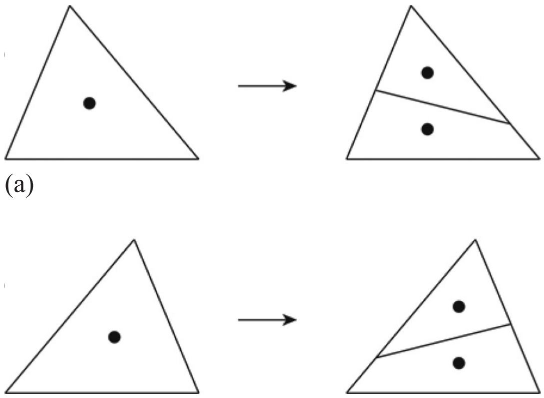


Figure 16. A subdivision shape rule (a) and its vertical reflection (b).

The determination of the position of a cutting line starts with inserting a point or multiple points in the interior of a polygon or on its boundary; then the cutting line is generated by connecting new inserted points to other existing points of the polygon, possibly involving line extensions and intersections, or by simply interconnecting the new inserted points. There are typically constraints over the candidate position of a new point. The constraints can be a fixed position like the centroid of a polygon, an interval on a line, or a particular region. This means that there are generally infinitely many ways to position a new point. Two ways frequently used to position the new point are *manual pickup* and *random selection*.

Noticeably, a 'subdivision' of the polygonal subframework is different from a 'splitting' of a rectangular subframework. The former is typically oblique while the latter is always horizontal or vertical. Moreover, a cutting line of the former often has infinitely many possibilities while the position of a splitting line of the latter is usually uniquely 'fixed'.

7.1 Common functions of polygonal subframework

Key common functions include the function of dividing a simple polygon into two by a cutting line, and those determining the positions of the new points.

7.1.1 *Dividing a simple polygon by a cutting line*

Here, we consider a more general function, of dividing a simple polygon G into multiple subpolygons by a cutting line C [figure 17(a)]. This problem can be solved by converting the problem to finding the intersection of two arbitrary (may not be simple) polygons, which has been well studied (Greiner and Hormann, 1998; O'Rourke, 1998; Stouffs, 1994; Stouffs

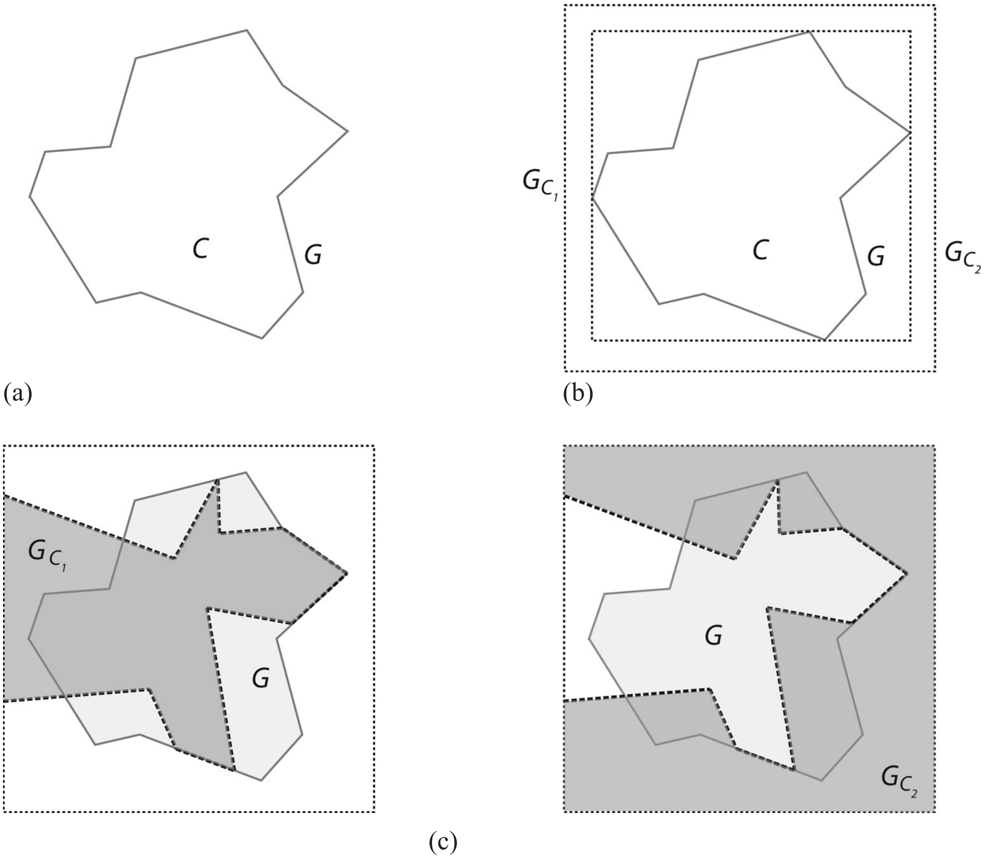


Figure 17. Dividing a simple polygon by intersection of two arbitrary polygons.

and Krishnamurti, 2006). This is done by first finding a rectangle, which is larger than the bounding box of polygon G, and then forming two simple polygons, G_{C1} and G_{C2}, by extending the starting and ending line segments of the cutting line [figure (17b)]; the desired results will be $(G \cap G_{C1}) \cup (G \cap G_{C2})$ [figure 17(c)].

A simpler but new algorithm is introduced here. This algorithm is inspired by Greiner and Hormann (1998). It takes advantage of the special properties of cutting lines in the polygonal subframework. A cutting line is always interior or on the boundary of the polygon G and has no self-intersection. Moreover, the start point P_s and end point P_e of the cutting line are on the boundary of polygon G [figure 18(a)]. In fact, any cutting line can be reshaped to satisfy these conditions. See figure 19.

Figure 18 illustrates the steps when applying the algorithm. The algorithm starts from the start point P_s of the reshaped cutting line, marching to the end point P_e, segment by segment. Each segment is tested for whether intersecting the polygon G or not by testing whether the other endpoint falls on the boundary of polygon G: for example, endpoint P_1 for segment S_1, P_2 for segment S_2 [figure 18(a)].

When an intersection is found, two new polygons are created by using the cutting line marched so far and continuously marching right and left, respectively, along the polygon G until going back to the start point. The segments marched are then removed from the cutting line so that a new cutting line is formed for the next step (the dashed line in figure 18). If the new cutting line is empty, then both new polygons are the desired results. Otherwise, by using the point-inside test on the two new polygons with the point P next to the start point of the new cutting line, the one which P does not fall inside (dark shaded polygons in figure 18) is the desired result, and the other (lightly shaded polygons in figure 18), together with the new

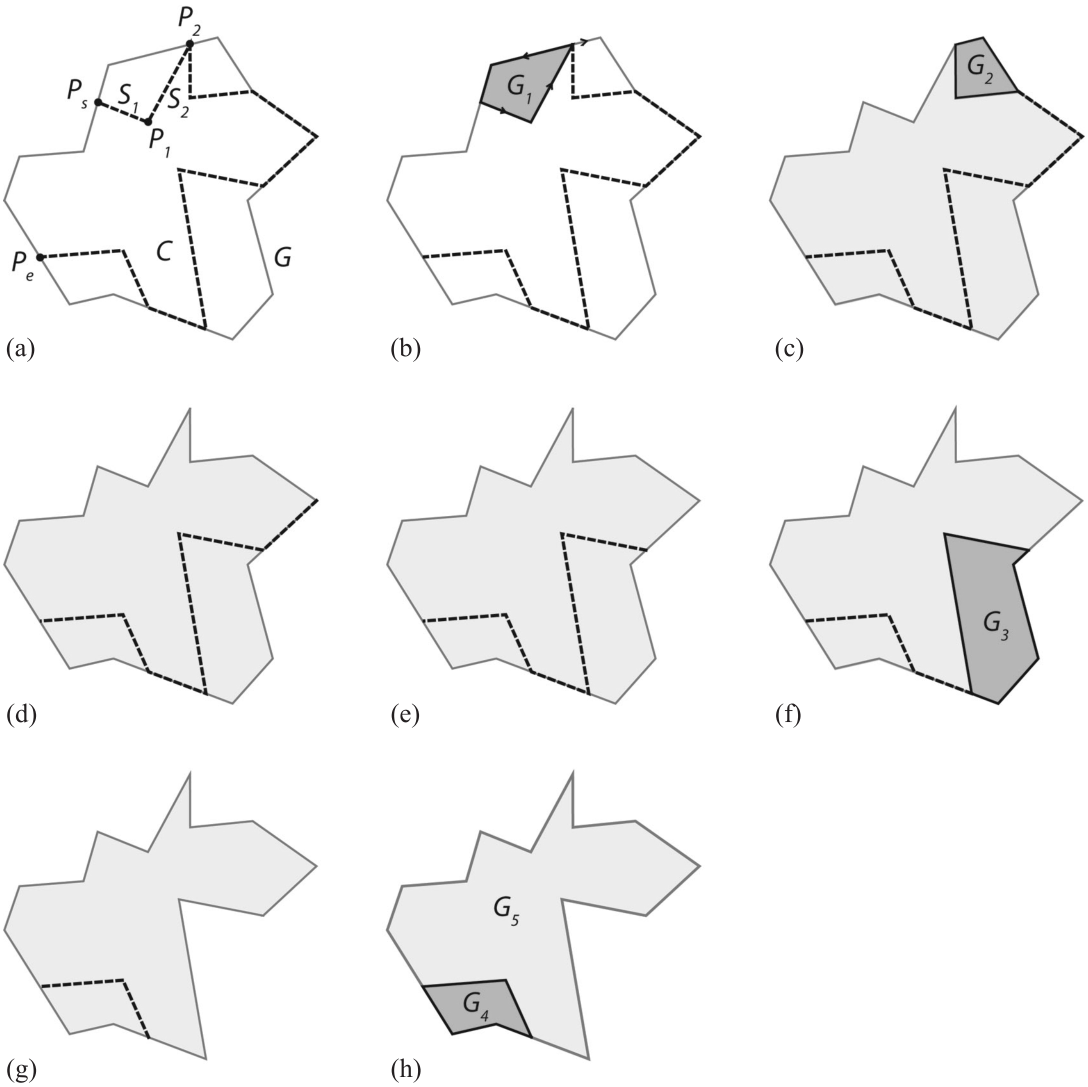


Figure 18. Applying a marching algorithm for polygon subdivision.

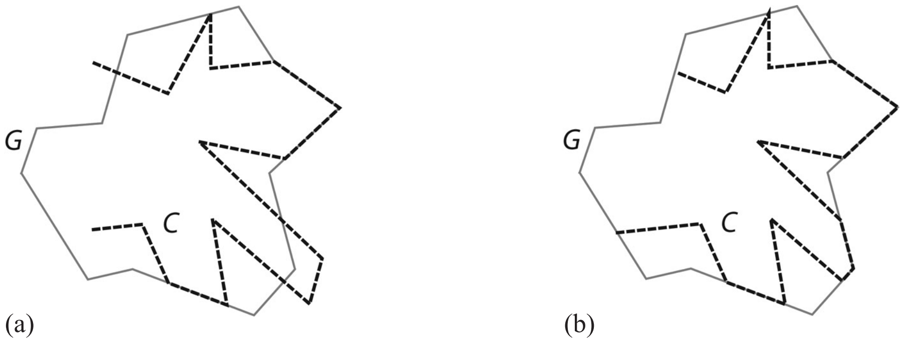


Figure 19. Reshaping the cutting line.

cutting will be used as the input for the next step. The above procedure is repeated and the entire algorithm stops when the cutting line becomes empty [figure 18(f)].

There are possible degenerate cases when some matched segments overlap with the some other segments of the polygon G [figures 18(c), (d), and (f). In such cases, the number of segments in one of the two new polygons must be two; this can be easily tested and ignored.

Another issue with such cases is that the segment coming from the cutting line is colinear and connected to the next segment coming from the polygon input, and these two should be merged into one [figure 18(f)]. The algorithm is given in figure 20.

Testing for intersection dominates the running time of the marching algorithm. The total number of intersection tests for marching along the cutting line is mn, where m and n is the number of segments in C and G, respectively. As a result, its complexity is $O(mn)$.

7.1.2 *Determining the positions of the new points*

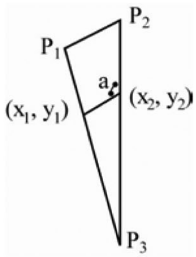
Determining the positions of the new points can be done in two ways, randomly or manually. Manual determination needs the support of an interface: for example, highlighting the candidate regions, and enforcing further constraints for the next new point after a new point has been picked. Random determination requires computing all candidates of intervals and regions, and randomly selecting a point.

```
dividePolygon (G, C, newPolygons)
    S = get first segment from C
    while (the other endpoint of S not falling on G){
        marchedSegments.add(S)
        S = get next segment from C
    }
    marchedSegments.add(S)
    I = the other endpoint of S
    C = C.remove(marchedSegments)
    divide the intersected segment of G into two if I is not its endpoints
    newPolygon1 = marching along marchedSegments and turning right at endP
        until reaching the start
    newPolygon2 = marching along marchedSegments and turning left at endP
        until reaching the start
    if (C is empty)
    then {
        newPolygons.add(newPolygon1)
        newPolygons.add(newPolygon2)
    return
    }
    P2 = the point next to the starting point in the new cutting line
    if (P2 falls inside newPolygon1)
    then {
        newPolygons.add(newPolygon2)
        dividePolygon(newPolygon1, C, newPolygons)
    }
    else {
        newPolygons.add(newPolygon1)
dividePolygon(newPolygon2, C, newPolygons)
```

Figure 20. A simple marching algorithm to divide a simple polygon by a cutting line.

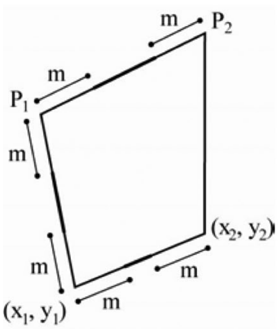
7.1.3 *Metalanguage for polygonal subframework*

Figure 21 illustrates the metalanguage for three examples taken from Knight (1980). The first randomly selects two points from candidate intervals. The metalanguage enforces the constraint on angle a, a $\geqslant 90^{\circ}$, by first, determining (x_2, y_2), second, creating a line through (x_2, y_2) and a perpendicular to line P_2P_3, third, computing the intersection (x, y) of



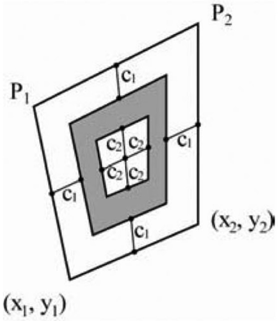
Point (x_1, y_1) is lies between 5/8 and 3/4 the distance from point P_3 to point P_1. Likewise point (x_2, y_2) is between 5/8 and 3/4 the distance from point P_3 to point P_2.
Angle a must be ⩾90°.

```
(x2,y2)=randomPick(interval(getPoint('P3'),getPoint('P2'),5/8,3/4))
line=perpendicular((x2,y2),getLine('P2','P3'))
(x,y)=intersection(line, getLine('P1','P3'))
(x1,y1)=randomPick(interval(getPoint('P3'),getPoint('P1'),5/8,3/4)&
interval((x,y), getPoint('P3')))
dividePolygon([(x1,y1),(x2,y2)], [P1,P3,P2], [])
```



Point (x_3, y_3) can be on one of the sides of the quadrilateral at an interval of m units away from the endpoints of these lines. m is a fixed constant.

```
(x3,y3)=manualPick(interval(getPoint('P1'),getPoint('P2'),m) |
interval(getPoint('P1'),(x1,y1),m) |
interval((x1,y1),(x2,y2),m))
```



Point (x_3, y_3) can be within the area defined by constants c_1 and c_2. This area is inside the quadrilateral with boundaries parallel to the boundaries of the quadrilateral.

```
(x3,y3)=manualPick(area(getPolygon('P1','P2','x2'+'y2','x1'+'y1'), c1, c2))
dividePolygon([(x1,y1), (x3,y3), P2], [P1,(x1,y1),(x2,y2),P2], [])
```

Figure 21. Metalanguage examples for picking up new points under constraints (adapted from Knight, 1980).

the new line with line P_1P_3 to obtain the interval [(x, y), P_3], and last, using the intersection of intervals [(x, y), P_3] and [P_3, P_1, 5/8, 3/4] as the interval for (x_1, y_1).

The second and third are examples of manually picking up points from candidate intervals and regions. All candidate intervals and regions are computed and combined as a candidate pool.

8 Graph subframework

The rectangular subframework may give the impression that shape grammars are just special cases of graph grammars (Brouno, 1990; Rozenberg, 1997), which have been widely studied in the computer science. The following discussion shows that both significantly differ from one another. However, graph grammars can be used as a subframework to solve certain dimensionless, context-free shape grammars.

8.1 Shape and graph grammars

Graphs provide a natural way of describing complex situations on an intuitive level. At a certain level, this characteristic caters to the advantage that visual languages (that is, shapes) possess. Graph grammars are rule-based modification of graphs through graph rule application. Graph grammars have been developed as an extension to graphs of formal string grammars (also known as generative grammar, or phrase structure grammars). Among string grammars, context-free grammars are the best understood; they have proven extremely useful in practical applications and powerful enough to generate a wide spectrum of interesting formal languages. Analogously, most research focuses on 'context-free' graph grammars, which typically means local modifications of graphs without 'global' constraints.

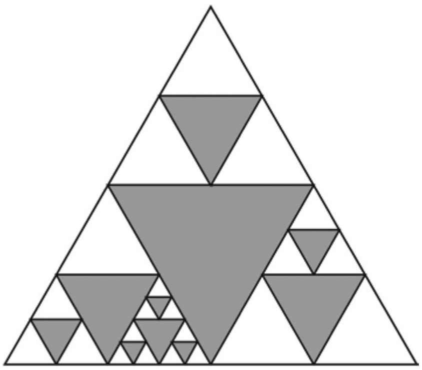


Figure 22. A Sierpinski gasket.

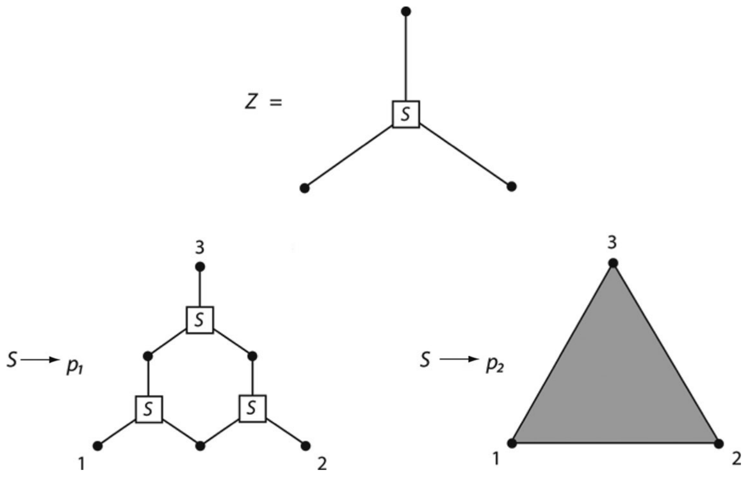


Figure 23. Collage grammar for the Sierpinski gasket. Adapted from Drewes and Kreowski (1999).

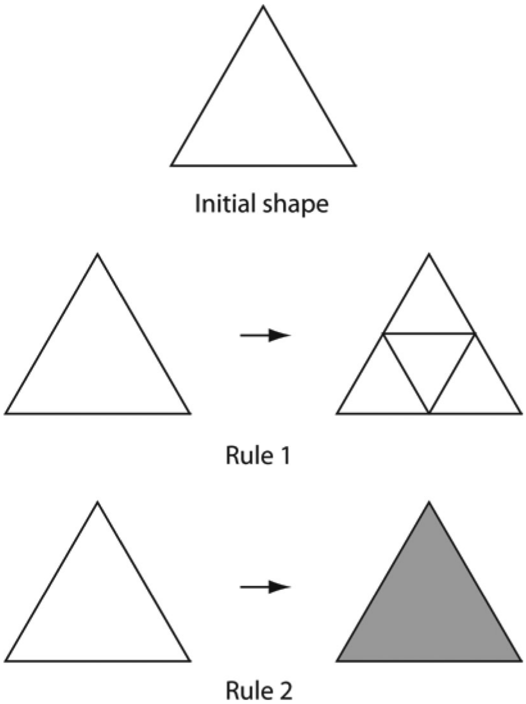


Figure 24. A shape grammar for the Sierpinski gasket.

Rule application on graphs is, typically, label driven. There are two basic choices for rewriting a graph: *node replacement* and *hyperedge replacement*.

Shape grammars are rule-based rewriting systems of shapes. In many ways, these can be viewed as an extension of formal string grammars to shapes. Their shared roots imply a close connection between graph and shape grammars. As an example, Drewes and Kreowski (1999) investigated the properties of collage grammars, a special case of graph grammars, and applied them to generate pictures: for example, a Sierpinski gasket (figures 22 and 23). Likewise, such pictures can be also succinctly described by shape grammars (Piazzalunga and

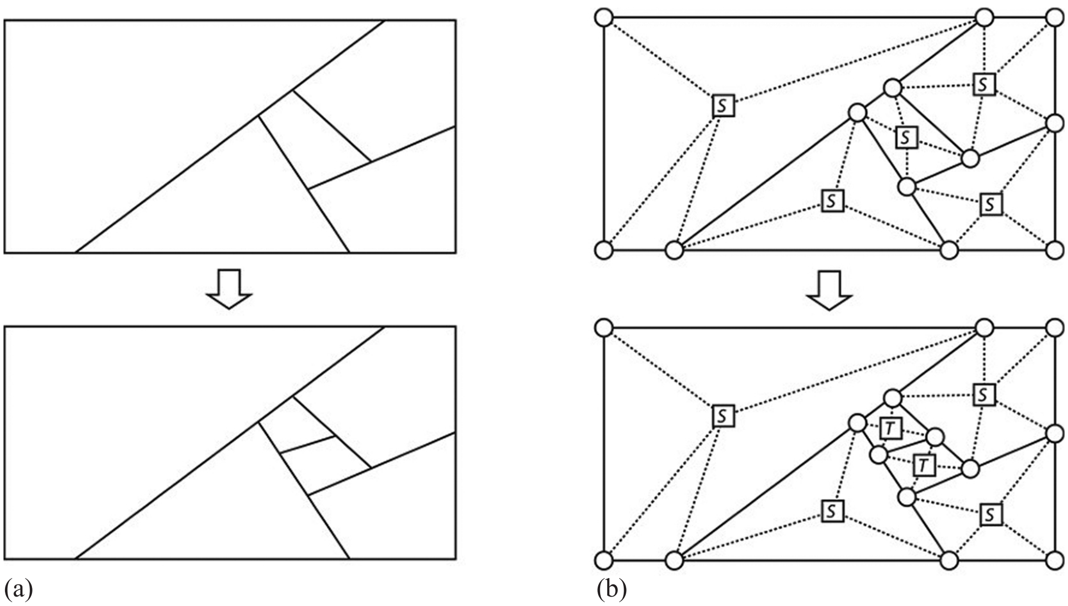


Figure 25. Implementing the ice-ray grammar as a graph grammar: (a) a step in the ice-ray derivation, (b) the corresponding hypergraph derivation.

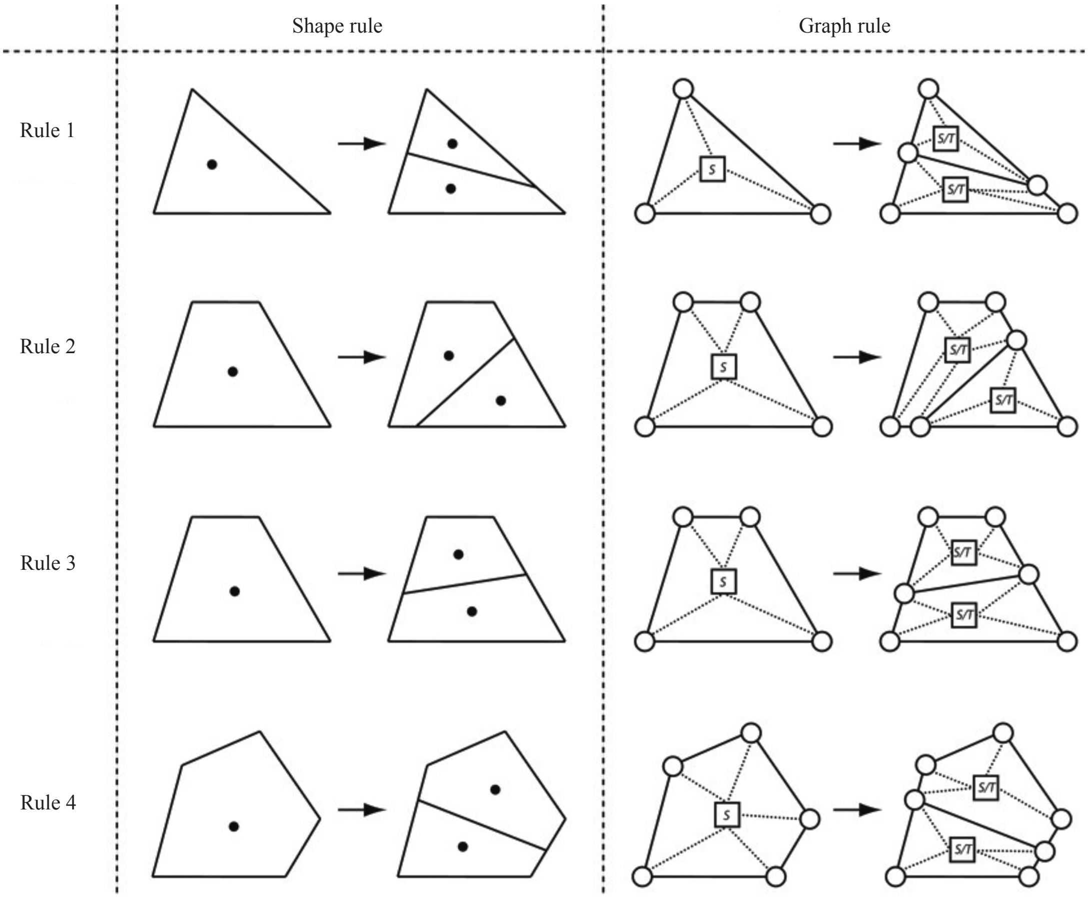
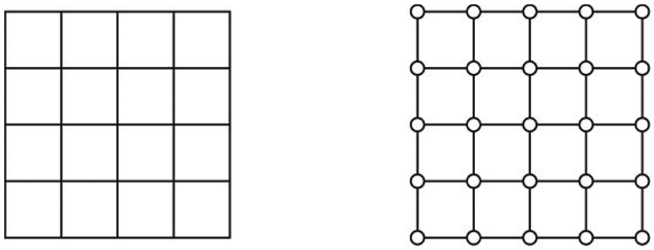


Figure 26. Shape and corresponding graph rules of the ice-ray grammar.

Fitzhorn, 1998; Stiny, 1977) (figure 24). This suggests that there is an intersection between graph and shape grammars.

Consequentially, shape grammars can take advantage of graph grammar research results, especially for 'context-free' shape grammars; that is, when shape rewriting happens locally. For example, as shown in figure 25, ice-ray grammars (Stiny, 1977), which essentially describes a process of polygon subdivision, can be implemented as a graph grammar. Each point corresponds to a vertex and each polygon is decorated with a hyperedge (the vertices drawn in squares together with dashed tentacles). Figure 26 shows shape rules versus corresponding graph rules of ice-ray grammars: the right-hand hyperedges are labeled either S as candidates for further rule application, or T for no further rule application; the choice is based on certain criteria, for example, the area of the underlying polygon. Rule 3 is applied in figure 25. Note that there is a necessary step to convert graphs to figures when using graph grammars to generate designs; depending on the details of the conversion, such graph grammars may show different appearances (figures 23 and 26).

On the other hand, shapes differ significantly from graphs and so do their grammars. Shape grammars do not deal solely with pure pictures; they are usually imbued with semantics, and represent designs in reality. In this respect, dimensions become typically important. Graph grammars, however, are inherently dimensionless. Moreover, semantics make most shape grammars context sensitive; this greatly limits whatever advantages are provided by those nice theorems of graph grammars (on the assumption that the grammars are context free).



(a) (b)

Figure 27. Subshape recognition in a grid figure: (a) a grid figure, (b) the corresponding graph.

Graph grammars are essentially label driven; this puts further restrictions in helping solve the fundamental problem of subshape recognition in shape grammars. As a classical example (figure 27), there are many, potential uncountable, square subshapes in a grid figure. Converting the grid figure to a graph does not change the basic characteristics of the problem.

8.2 Graph grammars as a subframework

Research on *collage grammars* (Drewes and Kreowski, 1999; Drewes et al, 1996) shows how graph grammars can be used as a subframework in the 'general' approach to shape grammar interpretation. Such graph grammars are essentially parametric and label driven. The underlying data structure is obviously a graph, typically undirected in the context of generating designs. The central step in using graph grammars to generate designs is the iterative application of a set of graph rules, which is known as graph transformation in the literature (Heckel, 2006). Moreover, the manipulation of the underlying graph is also achieved through graph transformation. Thus, the key common function is the application of a graph rule.

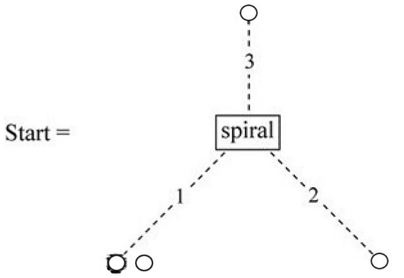
8.3 Graph rule application

Algorithms for graph rule application (that is, graph transformation) have been previously investigated (Rozenberg, 1997). In general, a graph rule *r* is defined by six tuples (*L*, *R*, *K*, *glue*, *emb*, *appl*): (i) *L* and *R* are left-hand side and right-hand side graphs, respectively; (ii) *K* is a subgraph of *L called* the interface graph; (iii) *glue* is an occurrence of *K* in *R*, relating the interface graph with the right hand side; (iv) *emb* is an embedding relation, relating nodes of *L* to nodes of *R*; and (v) *appl* is a set specifying the application conditions for the rule (Andries et al, 1999). It is possible that *K*, *glue*, *emb*, or *appl* is empty—certain combination of emptyness forms a rule with special properties, for example, rules without application conditions, or rules with an empty embedding relation corresponds to single-pushout rules.

The application of *r* to a graph *G* replaces an occurrence of the left-hand side *L* in *G* by the right hand-side *R*. This is done through three stages: (i) removing a part of the occurrence of *L* from *G*, (ii) gluing *R* and the remaining graph *D*, and (iii) connecting *R* with *D* via the insertion of new edges between the nodes of *R* and those of *D*. Note that the left-hand side matches all isomorphic graphs and this subsumes geometry transformations, which are usually important in the application of shape grammars. The details of the algorithm can be found in Andries et al (1999).

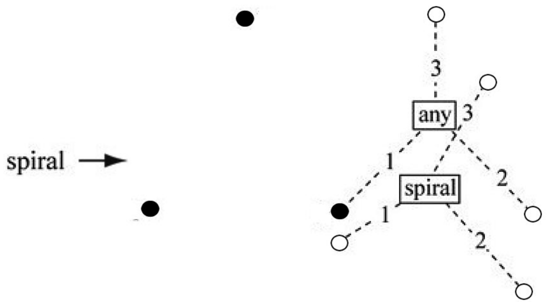
8.3.1 *Metalanguage for graph subframework*

The metalanguage for the graph subframework is mainly to call the graph rule application function by specifying the details of the graph rules, with auxiliary functions to convert the final graph to shapes. Figure 28 illustrates the style of a spiral collage grammar and its corresponding metalanguage description (Drewes et al, 1996).



Starting point is a hyperedge *spiral* with three tentacles.

Start
    Edge ‘spiral’ tri
    End Collage



Add rotated *spiral* hyperedge and an *any* hyperedge.

```
Rule ‘spiral’
        Pins tri
        Edge ‘any’ (100,0) (200,0) (150,100)
        Edge ‘spiral’ ROT2D (100, 10, 0, 0, rad(rotSpiral))
                      ROT2D (198, 10, 0, 0, rad(rotSpiral))
                      ROT2D (149,110, 0, 0, rad(rotSpiral))
        End Collage
```

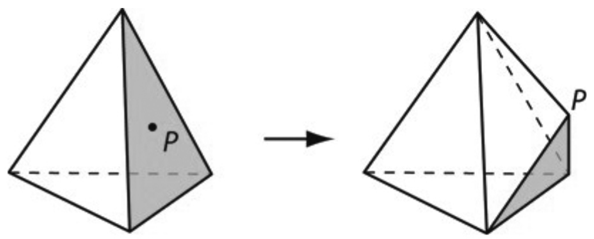
Figure 28. Metalanguage description for rules of a spiral collage grammar. Adapted from Drewes et al (1996).

9 The third dimension

The three subframeworks illustrated thus far are all two-dimensional. There is nothing intrinsic in the approach to prevent a subframework from being three-dimensional. The boundary solid grammar (Heisserman, 1994) is a case in point.

The representation of solid objects is composed of two parts: topology and geometry. The topology is represented as a graph composed of nodes and arcs—the nodes are topological elements, and the arcs represent the adjacencies between such elements. The geometry contains vertex coordinates for polyhedral solids. The topology together with the geometry forms a boundary representation. The basic operations are the Euler operators (Mäntylä, 1988) for modifying the topology, vertex coordinate assignment for modifying the geometry, label addition and removal, and state change to indicate the current status; these are all specified as predicates in the declarative programming language, CLP(R). Figure 29 shows one such example of a basic operation, namely, the `point_face` operator, which pulls out a surface onto a point, correctly modifying the number of edges and vertices of the face.

Supported by a set of basic operations, boundary solid grammars specify a subclass of shape grammars, which, in the context of this paper, specifies a subframework. Accordingly, the boundary representation is the underlying data structure, the algorithms are those for the



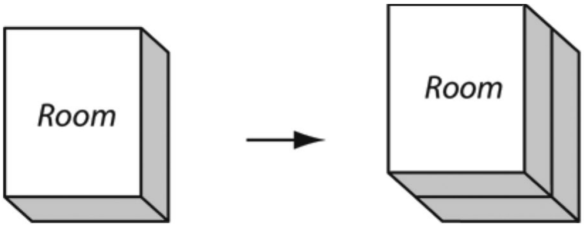

```
point_face(Face, Height):-
    face_eh(Face, Eh),
    ccw_eh(Eh, LastEh),
    edgeh_v(Eh, V),
    face_normal(Face, Normal),
    face_center(Face, Center),
    mev(V, LastEh, VTop, EhBt),
    other_eh(EhBt, EhTb),
    scalar(Height, Normal, Direct),
    vecplus(Center, Direct, CTop),
    set_vertext(VTop, CTop),
    point_face_1(LastEh, Eh, VTop, EhTb).

point_face_1(EndEh, EndEh, _, _).
point_face_1(EndEh, Eh, VTop, EhTb):-
    cw_eh(Eh, NextEh),
    edgeh_v(NextEh, V),
    v_coord(V, C),
    mefl(V, Eh, VTop, EhTb, NewEhBt, _, _),
    other_eh(NewEhBt, NewEhTb),
    point_face_1(EngEh, NextEh, VTop, NewEhTb).
```

Figure 29. Point_face operator (adapted from Heisserman and Woodbury, 1993).

basic operations, and the metalanguage are expressions in CLP(R), and the subclass of shape grammars contains those describable by the basic operations. Figure 30 shows an example of a shape rule for adding a second floor above a room.

10 Paradigm

The *Oxford English Dictionary* gives the basic meaning of the term *paradigm* as "an exemplar, a pattern followed, an epitome or a model." It is a term that is frequently employed within the design profession to indicate an archetype or outstanding example. Design paradigms comprise functional precedents for design solutions (Wake, 2000). In this paper we have developed an approach for implementing practical parametric shape grammars by, essentially, subdividing grammars into subclasses of tractable shape grammars. We have illustrated the




```
description(example_shape_rule, 'Add the second floor to a room.').

lhs(example_shape_rule, [Top, Height, Room], [Room]):-
    state(second),
    label(Room, name, S),
    member(S, [room, parlor, kitchen, dining, hall, pantry]),
    not(label(Room, mark, stacked)),
    top(Top, Room),
    room_height(Height).

rhs(example_shape_rule, [Top, Height, Room]):-
    stack_solid(Top, Height, NewRoom),
    make_label(Room, below, NewRoom),
    make_label(NewRoom, floor, second),
    room_colour(RColour),
    set_solid_colour(NewRoom, RColour).
```

Figure 30. Rule for adding a second floor above a room (adapted from Heisserman and Woodbury, 1993).

approach through several subframeworks each specified by an underlying data structure, basic manipulation algorithms, and a description metalanguage. Each subframework specifies a way of implementing a subclass of shape grammars. In terms of language space, the language covered by a subframework is identical to the language of a subclass of shape grammars. However, each subframework takes advantage of special characteristics of the corresponding subclass of shape grammars so that implementation is manageable. That is, although the language spaces are equal, the implementation may not truly implement the shape grammar formalism as described in its formal definition. On the other hand, in practice, many shape grammars are 'special' in that they are, indeed, tractable. Whence, it is feasible to consider a paradigm—comprising elements of similarities both practical and general—for implementing and hence, interpreting, tractable shape grammars. The approach presented in this paper constitutes such a paradigm.

Acknowledgement. This research was supported in part by a grant from US Army Corps of Engineers, Engineer Research and Development Center, Champaign, IL. Any opinions, findings, conclusions, or recommendations presented in this paper are those of the authors and do not necessarily reflect the views of CERL.

References

Andries M, Engels G, Habel A, Hoffmann B, Kreowski H-J, Kuske S, Plump D, Schurr A, Taentzer G, 1999, "Graph transformation for specification and programming" *Science of Computer Programming* **34** 1–54

Brouno C, 1990, "Graph rewriting: an algebraic and logic approach", in *Handbook of Theoretical Computer Science (Volume B): Formal Models and Semantics* (MIT Press, Cambridge, MA) pp 193–242

Carlson C, McKelvey R, Woodbury R, 1991, "An introduction to structure and structure grammars" *Environment and Planning B: Planning and Design* **18** 417–426

Chau H H, Chen X, McKay A, Pennington A, 2004, "Evaluation of a 3D shape grammar implementation", in *Design Computing and Cognition '04* Ed. J S Gero (Kluwer Academic, Dordrecht) pp 357–376

Cormen T H, Leiserson C E, Rivest R L, Stein C, 2004 *Introduction to Algorithms* 2nd edition (MIT Press, Cambridge, MA)

Drewes F, Kreowski H J, 1999, "Picture generation by collage grammars", in *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages, and Tools* (World Scientific Publishing, Singapore) pp 397–457

Drewes F, Kreowski H-J, Schwabe N, 1996, "COLLAGE-ONE: a system for evaluation and visualization of collage grammars" *Machine Graphics and Vision* **5** 393–402

Flemming U, 1987, "More than the sum of parts: the grammar of Queen Anne houses" *Environment and Planning B: Planning and Design* **14** 323–350

Golomb S W, 1994 *Polyominoes* (Princeton University Press, Princeton, NJ)

Grady B, Robert M, Michael E, Bobbi Y, Jim C, Kelli H, 2007 *Object-oriented Analysis and Design with Applications* 3rd edition (Addison-Wesley Professional, Indianapolis, IN)

Greiner G, Hormann K, 1998, "Efficient clipping of arbitrary polygons" *ACM Transactions on Graphics* **17** 71–83

Heckel R, 2006, "Graph transformation in a nutshell" *Electronic Notes in Theoretical Computer Science* **148** 187–198

Heisserman J, 1994, "Generative geometric design" *IEEE Computer Graphics and Applications* **14** 37–45

Heisserman J, Woodbury R, 1993, "Generating languages of solid models", in *Proceedings of the Second ACM symposium on Solid Modeling and Applications* (ACM, Montreal, Quebec) pp 103–112

Knight T W, 1980, "The generation of Hepplewhite-style chair back designs" *Environment and Planning B* **7** 227–238

Knight T W, 1981, "The forty-one steps: the languages of Japanese tea-room designs" *Environment and Planning B: Planning and Design* **8** 97–114

Knight T W, l989, "Transformations of the De Stijl art: the paintings of Georges Vantangerloo and Fritz Glarner" *Environment and Planning B: Planning and Design* **16** 51–98

Knight T W, 1999, "Shape grammars: six types" *Environment and Planning B: Planning and Design* **26** 15–31

Knight T W, Stiny G, 2001, "Classical and non-classical computation" *Information Technology* **5** 355–372

Koning H, Eizenberg J, 1981, "The language of the prairie: Frank Lloyd Wright's prairie houses" *Environment and Planning B: Planning and Design* **8** 295–323

Krishnamurti R, 1982, "SGI: an interpreter for shape grammars", technical report, Centre for Configurational Studies, The Open University

Mäntylä M, 1988 *An Introduction to Solid Modeling* (Computer Science Press, College Park, MD)

O'Rourke J, 1998 *Computational Geometry in C* (Cambridge University Press, Cambridge)

Piazzalunga U, Fitzhorn P, 1998, "Note on a three-dimensional shape grammar interpreter" *Environment and Planning B: Planning and Design* **25** 11–30

Rozenburg G, 1997 *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1: Foundations* (World Scientific Publishing, Singapore)

Stiny G, 1977, "Ice-ray: a note on Chinese lattice designs" *Environment and Planning B* **4** 89–98

Stiny G, 1980, "Kindergarten grammars: designing with Froebel's building gifts" *Environment and Planning B* **7** 409–462
Stiny G, 1982, "Spatial relations and grammars" *Environment and Planning B: Planning and Design* **9** 113–114
Stiny G, 1991, "The algebras of design" *Research in Engineering Design* **2** 171–181
Stiny G, 1994, "Shape rules: closure, continuity, and emergence" *Environment and Planning B: Planning and Design* **21** s49–s78
Stiny G, 2006 *Shape: Talking about Seeing and Doing* (MIT Press, Cambridge, MA)
Stiny G, 2011, "What rule(s) should I use?" *Nexus Network Journal* **13**(1) 15–47
Stiny G, Gips J, 1971, "Shape grammars and the generative specification of painting and sculpture", in *Information Processing 71* Ed. C V Freiman (North-Holland, Amsterdam) pp 1460–1465
Stouffs R, 1994 *The Algebra of Shapes* PhD dissertation, Department of Architecture, Carnegie Mellon University, Pittsburgh, PA
Stouffs R, Krishnamurti R, 2006, "Algorithms for the classification and construction of the boundary of shapes" *Journal of Design Research* **5** 54–95
Wake W K, 2000 *Design Paradigms: A Sourcebook for Creative Visualization* (John Wiley, Chichester, Sussex)
Winograd T, Flores F, 1986 *Understanding Computers and Cognition: A New Foundation for Design* (Addison Wesley, Boston, MA)
Wirth N, 1978 *Algorithms + Data Structures = Programs* (Prentice-Hall, Upper Saddle River, NJ)
Yue K, Krishnamurti R, 2008, "A technique for implementing a computational-friendly shape grammar interpreter", in *Design Computing and Cognition: Proceedings of the Third International Conference on Design Computing and Cognition* Eds J S Gero, A K Goel (Springer, Atlanta, GA) pp 61–80
Yue K, Krishnamurti R, 2013, "Tractable shape grammars" *Environment and Planning B: Planning and Design* **40** 576–594