View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace@MIT

Representing Shapes as Graphs: A Feasible Approach for the Computer Implementation of Parametric Visual Calculating

by

Thomas Alois Wortmann

Diploma in Architecture, University of Kassel, Germany, 2008

Submitted to the Department of Architecture

in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Architecture Studies

at the

Massachusetts Institute of Technology

June 2013

© 2013 Thomas Alois Wortmann. All Rights Reserved

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Signature of Author:____

Thomas Wortmann
Department of Architecture
May 23, 2013

Certified by:____

George Stiny
Professor of Design and Computation
Thesis Advisor

Accepted by:____

Takehiko Nagakura
Associate Professor of Design and Computation
Chair of the Department Committee on Graduate Students

Representing Shapes as Graphs: A Feasible Approach for the Computer Implementation of Parametric Visual Calculating

by

Thomas Alois Wortmann

Thesis Committee

George Stiny, PhD
Professor of Design and Computation, MIT
Thesis Advisor

Takehiko Nagakura, MArch, PhD
Associate Professor of Design and Computation, MIT
Thesis Reader

Rudi Stouffs, Ir-Arch, MsC, PhD
Associate Professor of Design Informatics, TU Delft
Thesis Reader

Representing Shapes as Graphs: A Feasible Approach for the Computer Implementation of Parametric Visual Calculating

by

Thomas Alois Wortmann

Submitted to the Department of Architecture on May 23, 2013 in Partial Fulfillment of the Requirements for the Degree of Master of Science in Architecture Studies at the Massachusetts Institute of Technology

Abstract

Computational design tools in architecture currently fall into two broad categories: Tools for representation and tools for generative design, including scripting. However, both categories address only relatively methodical aspects of designing, and do little to support the design freedom and serendipitous creativity that, for example, is afforded by iterative sketching. Calculating with visual rules provides an explicit notation for such artistic processes of seeing and drawing. Shape grammars have validated this approach by formalizing many existing designs and styles as visual rule-sets. In this way, visual rules store and transfer design knowledge. Visual calculating in a more general sense supports creativity by allowing a designer to apply any rule she wants, and to capriciously see and re-see the design. In contrast to other explicit design methodologies, visual calculating defines a decomposition into parts only after the design is calculated, thus allowing formalization without impeding design freedom.

Located at the intersection between design and computation, the computer implementation of visual calculating presents an opportunity for more designerly computational design tools. Since parametric visual calculating affords the largest set of design possibilities, the computer implementation of parametric visual calculating will allow flexible, rule-based design tools that intelligently combine design freedom with computational processing power. In order to compute with shapes, a symbolic representation for shapes is necessary. This thesis examines several symbolic representations for shapes, including graphs. Especially close attention is given to graph-based representations, since graphs are well suited to represent parametric shapes. Based on this analysis, this thesis proposes a new graph for parametric shapes that is clearer, more compact and closer the original formulation of visual calculating than existing approaches, while also strongly supporting design freedom. The thesis provides algorithms and heuristics to construct this "inverted" graph, for connected and unconnected shapes.

Thesis Advisor: George Stiny

Title: Professor of Design and Computation

Representing Shapes as Graphs

by

Thomas Alois Wortmann

Acknowledgments

I wish to thank my thesis advisor George Stiny, for providing the inspiration for this thesis, as well as for his continual support and guidance.

I wish to thank my readers Takehiko Nagakura and Rudi Stouffs, for their diligence and valuable advice.

Finally, I wish to thank my parents Alois and Margot, who have wholeheartedly supported me in this endeavor, like in all others. This thesis is dedicated to them.

Table of Contents

1 Introduction **10**

1.1 Design: Spontaneous Act or Explicit Process? 10

1.2 Visual Calculating Bridges Design Freedom and Explicit Methodologies 12

1.3 Computers in Architectural Design 14

1.4 Overview of the Thesis 15

2 Visual and Symbolic Calculating **16**

2.1 Designing with Shape Rules 16

2.2 Calculating with Maximal Elements: The Embedding and Part Relations 18

2.3 Transformations 21

2.4 The Problem of Decomposition 25

2.5 Symbolically Representing Visual Calculating 26

2.6 Precedents for Symbolic Representation and Computer Implementation 28

3 Shapes as Graphs **34**

3.1 Graphs and Graph Isomorphism 34

3.2 Differences between Shapes and Graphs 35

3.3 Representing Shapes as Graphs 37

4 The Shape Graph **42**

4.1 Constructing the Shape Graph 42

4.2 Types of Intersections 47

4.3 Heuristics for Representing Non-maximally Connected Shapes 48

4.4 Constructing Shape Graphs for Non-Maximally Connected Shapes 53

5 Conclusion **56**

5.1 Summary of the Thesis 56

5.2 Towards Design Tools with Flexible Constraints for Serendipitous Discovery 58

5.3 Further Research 59

6 Bibliography **62**

1 Introduction

This thesis is about how one can design with computers. The first section in this chapter introduces two contrasting aspects of designing: On the one hand, there is the creativity of design freedom, which can be fickle and seemingly arbitrary. On the other hand, there is the rigor of formal design methods. These two aspects can be reconciled by visual calculating, which is introduced in the second section. The third section briefly discusses how the contrast between design freedom and formal methodologies is reflected in architectural design practice, where computers are almost exclusively employed for explicitly defined tasks. Lastly, an overview of the remainder of the thesis is given.

1.1 Design: Spontaneous Act or Explicit Process?

This section discusses two contrasting notions of designing. One notion focusses on formal and methodological aspects, while the other underscores the intuitive and artistic nature of designing. Importantly, this dialectic also informs the manner in which computer are integrated in the design process. The notion of design freedom and serendipitous creativity is summed up by a quote from exhibition curator Ammann:

[An artist] works on something whose end product he can discern only very vaguely. . . . He has some idea, but he is constantly confronted with failure. For it is possible that what is emerging does not fit with his idea. One can also say that doing constantly changes the idea, because doing is more important than the idea. (After *(Gänshirt, 2007, p. 78)*)

Though, in this quote, Ammann refers to artists, his remark is easily generalized to all designers. Design is understood as a vague, poetic, and seemingly arbitrary activity, which prioritizes doing over reflecting and does not lend itself to easy systematization. Consequently, one might conclude that it is difficult, if not impossible, to preserve design knowledge or to support designing with computerized methods.

A directly opposite position is taken by Simon (1996), who, in *The Sciences of the Artificial*, claims that, in principle, all designing is open to explicit formalization. According to Simon, the mind of a designer, i.e., the “human information-processing system”, is “basically serial in its operation” with “limited memory structures, whose content can be changed radically” (ibid., 81). In other words, Simon assumes that the functionality of the human mind is comparable to a digital computer, and that “the theory of design” therefore is nothing more than the “general theory of [combinatorial] search” (ibid, 83). Consequently, not only can designing be formalized: It can be automated completely. However, there has been little progress in validation of this claim since the first publication of Simon’s book in 1969

(Dreyfus, 1992). Nevertheless, Simon has inspired many methodological approaches, especially in the field of "design science", i.e., "design research" (Bayazit, 2004).

Explicit design methods store and transfer design knowledge and thus make it available for computer implementation. However, a recurring problem is that design methods rely on a prior, less formal understanding of the design task. This understanding can concern the nature of the solution or other unstated assumptions (such as the supposition that users know what they want). In other words, the design task is reduced into a form that may fit the design method, but not necessarily the task. For example, in *Notes on the Synthesis of Form*, Alexander (1964) assumes that a design problem can be solved by listing all explicit and implicit requirements and their interactions. Based on this information, one can then decompose the design task into a set of smaller problems. However, an important aspect of designing has to take place before Alexander's method can be applied: Namely the definition of the requirements and their interactions, which necessitates value judgments by the designer[1]. As Alexander puts it, "two variables interact if and only if the designer can find some reason . . . which makes sense to him" (ibid., 109). In *What Computers Still Can't Do*, a similar point is made by Dreyfus (1992) in reference to a computer program presented by Newell, Shaw, and Simon as an example of creative thinking. Here, "the classification of the operators into essential and inessential . . . is introduced by the programmers before actual programming begins" (ibid., 116). In *Tools for Ideas: an Introduction to Architectural Design*, Gänshirt (2007) makes a more general but related point, namely that there can be no rationality without emotional judgment. He quotes the neuroscientist Antonio Damásio as saying that "people who have lost their ability to respond to things emotionally, who have lost their feelings . . . also lose the ability to plan ahead and act with an eye to the future" (ibid., 77).

Comparing the vagueness of intuitive design freedom with the limited rationality of explicit methodologies, one can conclude that some aspects of designing are open to formalization, while others rely on artistic intuition, and that both approaches complement each other. One the one hand, it is commonly accepted that the key to creative design exploration is design freedom – thinking out of the box, the shedding of pre-conceptions, etc. On the other hand, explicit methods rely, by their very nature, on clear problem definitions and a bounded range of solutions. This dialectic has also been described as problem setting versus problem solving, or as divergent versus convergent thinking (e.g., (Dym, Agogino, Eris, Fry, & Leifer, 2005)). Similarly, Gänshirt contrasts an artistic notion of designing with a methodical one:

---

[1] Alexander lists 141 needs in his design for an Indian village, none of which concern visual attractiveness (Alexander, 1964, pp. 137-142).

The simultaneity of different levels of action in one and the same act of designing makes it difficult to analyze further. The fact that designing is often mystified as something brilliant, intuitive and purely emotional is therefore not without reason. By comparison, the view of designing as a process (...) orders its procedures chronologically and thus reflect the way in which our activity is tied to time. In the process it contradicts the simultaneity of the overlapping and interdependent aspects of a design problem. Both approaches sum up essential aspects of design, but contradict each other and each remains unsatisfactory in its own right. (Gänshirt, 2007, p. 78)

To overcome this contradiction, Gänshirt proposes a design cycle of understanding the problem, expressing an idea, evaluating the proposed idea, etc. (ibid.). In other words, for Gänshirt, a cycle of seeing and doing bridges the contradiction between designing as a spontaneous artistic act and designing as an explicit process. Schön (1983) has a similar conception of designing: he sees it as a process of "Reflection-in-action" governed by "backtalk" and "reframing". For this thesis, design freedom and ambiguity are just as necessary for innovative design as rationality and explicit methods, since both need each other to reach their full potential. A vague initial idea can be concretized by a methodical approach, while the application of a design method needs creative inspiration to produce original results. Gänshirt's design cycle and Schön's reflection-in-action are descriptive models that integrate designing as a spontaneous act and designing as an explicit process. However, these models do not explain how these seemingly opposite approaches can be reconciled in detail. A more specific, rule-based formulation is provided by visual calculating, which is introduced in the next section.

1.2 Visual Calculating Bridges Design Freedom and Explicit Methodologies

Visual calculating, i.e., calculating with shapes, bridges the gap between design freedom and explicit methodologies. (Visual calculating is discussed more extensively in chapter 2.) Stiny has provided the current reference for visual calculating, *SHAPE: Talking about Seeing and Doing* (2006), and, similarly to Gänshirt, understands design as a process of seeing and doing. However, for Stiny, this view is rooted in the understanding that all design is rule-based, which is endorsed in this thesis. In this way, designing is understood as a process of successively applying visual rules, during which every result can give rise to new rules or new interpretations of the design. (See Figure 1.1 for an example of a visual rule and example designs created with that rule.) Calculating with visual rules thus goes beyond Gänshirt's design cycle by offering a formal notation for exploratory visual design. Visual calculating allows the storage and transfer of design knowledge as rules, but on the other hand guarantees maximum design freedom by allowing for, and even encouraging, ambiguity and the deviation from pre-conceived ideas.

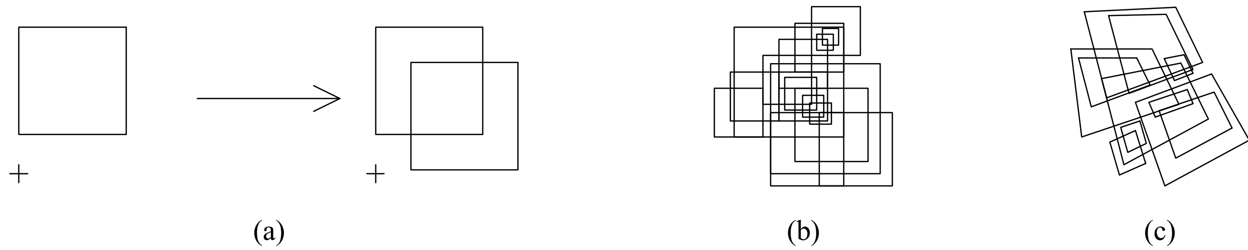


Figure 1.1 *(a) is an example of a shape rule, while (b) and (c) show example designs that were created with it.*

Stiny puts it like this:

Calculating with shapes is an open-ended process – like art and design. You're always free to try another rule. (Stiny, 2011)

Not only is the designer free to apply any rule, but, unlike the design methods discussed in the previous section, visual calculating allows her to see and re-see the design in any way she wants:

Shapes are indeterminate – ambiguous – before I calculate, and have constituent distinctions – parts – only as a result of using rules in an ongoing process. (Stiny, 2006, p. 50)

In other words, the design process can be formalized, but only after it has taken place. Visual calculating thus sees the apparent arbitrariness and fickleness of designing as essential for creative innovation, rather than as obstacles to more scientific methods of designing. Compared to other design methods, this is a key difference that promises to reconcile design freedom and explicit methodologies.

Like symbolic computation, which can be achieved by hand with pen-and-paper, mechanically with gears, or digitally with zeros and ones, visual calculating is medium-independent. However, it stands to reason that a rule-based exploratory design process can benefit from the flexibility and computational power of digital computers. From that perspective, the computer does not automate the design process, but stores and applies design rules as a convenience for the designer. Although, in the last thirty years, a serious and divers body of work regarding the computer implementation of visual calculating has been produced, the resulting computerized design tools were often limited in terms of their functionality and have not found widespread use. However, as discussed in section 2.6, there recently have been a number of interesting approaches to extend the functionality of such design tools. In line with this research, this thesis proposes a flexible computational representation for shapes as a foundation for the computer implementation of visual calculating and as a bridge between intuitive and formal aspects of designing. A short overview of the role of computers in architectural design is provided in the following section.

1.3 Computers in Architectural Design

To understand how the computer implementation of visual calculating differs from other computerized design tools, a short overview of architectural design with the computer is useful. In the 1980ies and 90ies, the representation of architecture shifted from an analogue into a digital mode of drawing, which simplified many of the challenges of drafting with ink pens on tracing paper. In a second step, BIM introduced computer models as a three-dimensional and information-rich alternative to two-dimensional architectural drawings. However, the generation of designs with hand-drawn sketches still is a widespread way of designing. After a design idea has been evolved in sketches, designs are explored in detail via more formalized drawings and models. From this perspective, the actual process of designing has not been changed all that much by the introduction of computerized tools.

However, there also has been a parallel interest in automating the design process. In *The Logic of Architecture*, Mitchell (1990, pp. 179-180) envisions an approach based on a cycle of automated design generation and testing. During the last decade, generative design tools have indeed spread among architectural designers (Burry, 2011). Often, techniques like scripting and parametric design are used to generate design variations, from which a final design is selected. Similar to the generate-and-test cycle proposed by Mitchell, these generative design techniques are sometimes combined with automated performance criteria to optimize certain aspects of a design. However, the spread of generative design techniques from academia into architectural education and practice has also highlighted important limitations:

To start with, the application of generative design techniques requires a significant initial time investment. Setting up a parametric model or script is much more time-consuming than making a few sketches. This can make it more difficult for a designer to change her mind when the initial approach turns out to be problematic. This may sound paradoxical, since the benefit of setting up a generative system supposedly lies in the larger number of design variations that can be examined. However, only design variations within the initially defined constraints of the generative system can be explored. Traditional design techniques like sketching usually examine a smaller set of possibilities, but at the same time allow a much wider range of different ideas. The pre-defined constraints of the generative system lead to another limitation: In order to apply a generative design technique effectively, a designer needs to have a clear idea of what she wants. We thus end up with a dilemma that parallels the contrast between design freedom and explicit methodologies discussed in the first section: It is desirable that computerized design tools support design exploration, and not only representation and optimization. However, creativity and design freedom should be not be impeded by pre-definitions and a bounded range of solutions. A

computer implementation of visual calculating promises to overcome this dilemma by allowing a designer to change her mind easily and, at the same time, to explore design ideas in a rigorous and efficient manner. The last section gives an overview of the remainder of the thesis.

1.4 Overview of the Thesis

This thesis offers a step towards more powerful and integrated computer implementations of visual calculating. Specifically, a flexible symbolic representation for shapes as a pre-condition for visual computing is proposed. (One needs to represent visual shapes symbolically in order to digitally compute with them.) The representation is based on graphs and includes parametric and non-parametric shapes. Parametric shapes are a generalization of non-parametric shapes and allow the definition of classes of shapes such as “all quadrilaterals” or “all polygons”. Calculating with parametric shapes thus increases design possibilities. Graphs afford this kind of design freedom, but can be more tightly constrained when necessary. As discussed in chapter 3, a similar approach has been proposed by Grasl and Economou (2011), however, the graph in this thesis is more compact, closer to the original definition of shapes, and can represent non-maximally connected shapes (i.e., shapes that are composed of non-intersecting elements).

The second chapter introduces visual calculating and gives an overview of the key problems and approaches to symbolically computing with shapes. The third chapter discusses graphs and compares five different graph-based representations for shapes, including the *inverted* graph proposed by this thesis. In the fourth chapter, algorithms for the inverted graph are provided. The last chapter offers a summary, sketches an opportunity for a creativity-enriching, rule-based exploratory design tool, and points out avenues for further research.

2 Visual and Symbolic Calculating

This chapter reviews visual calculating as a rule-based formalization of visual design processes. First a general overview is given. The second section explains the embedding and part relations, two fundamental concepts of visual calculating, and the third transformations, i.e., the different ways in which shape rules are applied. In the fourth section, the important problem of decomposition is addressed, demonstrating how calculating with shapes contrasts with the symbolic calculations of computers. The fifth section explains key problems in bridging visual calculating and symbolic computation, while the last section discusses precedents of how visual calculating is implemented on computers.

2.1 Designing with Shape Rules

The idea of designing with visual rules is commonly associated with shape grammars. The first publication on shape grammars was "Shape Grammars and the Generative Specification of Painting and Sculpture" (Stiny & Gips, 1972). There, a shape grammar is described as a set of shape rules that replace the shape that is drawn on the left-hand side of a rule with the shape on the right-hand side:

$$S_L \rightarrow S_R$$

For an example, see (a) in Figure 2.1. A special shape is the *empty* shape, which, if on the left-hand side, allows the creation of new shapes from scratch:

$$\varnothing \rightarrow S_R$$

For an example, see (b) in Figure 2.1.

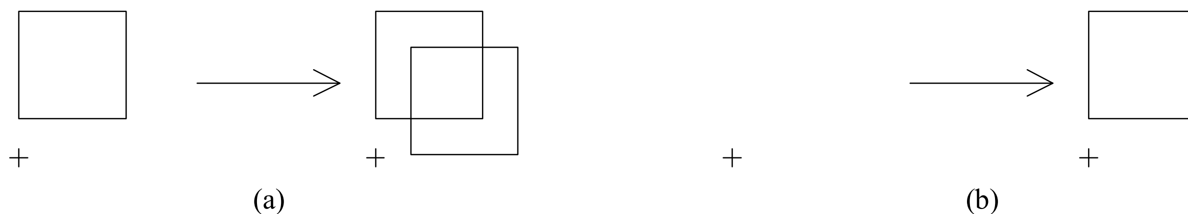


Figure 2.1 *An example shape rule is given with (a), and an example rule with the empty shape on the left-hand side with (b).*

By applying the same rule multiple times, simple shape grammars create astonishingly rich visual designs. This richness from simple rules is especially true of recursive shape grammars, like the Ice-Ray

grammar (Stiny, 1977). Here, recursion means that a shape rule applies to its own result, which is the case for the derivation from shape (a) to shape (d) in Figure 2.2.

Figure 2.2 *Rule (a) from Figure 2.1 is applied to shape (a) to derive shape (b). Rule (a) is also applied to shape (b) to derive shape (c) and to shape (c) to derive shape (d).*

Shape grammars thus succinctly encapsulate design processes as sets of rules reproducing designs of a certain kind or style (Knight, 1994). Much of the early research on shape grammars focused on defining grammars for existing designs. A well-known example is the Palladian grammar (Stiny & Mitchell, The palladian grammar, 1978). This grammar not only reproduced the plans of villas that were designed by Palladio, but also generated new plans in the Palladian style, allowing the enumeration of "all possible" Palladian villa plans for a given size (Stiny & Mitchell, Counting palladian plans, 1978). Arguably, this type of analysis not only leads to a deeper understanding of the examined designs in terms of their methodology, but also serves to validate two central assumptions behind the shape grammar formalism. The first assumptions is that much, if not all, of designing can be understood as rule-based; the second, that shape grammars therefore provide an adequate formalization of the design process.

However, in accepting these two hypotheses, one has to conclude that, at its most creative, visual calculating is not only a method for reproducing designs by applying formalized design knowledge gained from the analysis of already known designs. Rather, one can understand visual calculating as a formalization of the process by which original designs are created. This view is expressed by Stiny (2006), and is adopted in this thesis. Visual calculating understands design as an iterative process of seeing and doing, where the designer applies a rule, looks at the result, and applies another rule based on what he sees. Instead of a shape grammar specifying the sequence of rules, as a designer you can "use any rule(s) you want, whenever you want to" (Stiny, 2011). In other words, visual calculating does not relive the designer of her responsibility to invent new rules and make (potentially spontaneous) decisions. However, this apparent arbitrariness does not mean that one should not record new rules or repeat existing ones. On the contrary, from the perspective of visual calculating, reapplying the same rules is exactly what designers do when they create visually coherent designs. Repetition and copying thus go hand-in-hand with creative invention. Visual calculating provides a notation for this type of design process by expressing visual ideas in the more definite form of shape rules. In this way, design intentions are

clarified and made available for reference and pedagogy. (For an example of design education with shape rules, see (Özkar, 2011).) Visual calculating thus overcomes the apparent opposition between explicit design methodologies and inspired design freedom. The next section discusses three fundamental aspects of visual calculating: maximal elements, the embedding relation, and the part relation.

2.2 Calculating with Maximal Elements: The Embedding and Part Relations

In visual calculating, a shape is defined as a finite set of *maximal elements*[2]. Common examples of elements are points, line segments, and planes, though elements can also include curves and other types of geometric entities. For an element to be *maximal*, it cannot be contained in, overlapping with, or adjacent to other maximal elements of the same type. Two non-maximal elements of the same type that do contain, overlap, or adjoin each other are joined into one maximal element via the *reduction rules* (Stiny, 2006, p. 187). Requiring elements to be maximal ensures that a shape is composed of the smallest set of the largest possible elements. Rectilinear elements are defined in algebras U_{ij} for unlabeled shapes, in algebras V_{ij} for labeled shapes, and in algebras W_{ij} for weighted shapes. The dimensionality of the elements is denoted by i and the dimensionality of the arrangement of the elements by j. Maximal elements with dimension i greater than zero potentially contain an indefinite number of smaller elements, while zero-dimensional elements (i.e., points) only contain one element, namely themselves. For example, a shape composed of elements from the algebra U_{12} consists of unlabeled line segments in the plane. Intersections and boundaries are not parts of maximal elements, since they are defined in different algebras. Examples of boundaries are the start and end points of a line segment, which, like the intersection point of two line segments, are zero-dimensional elements in the algebra U_{02}. This thesis primarily focuses on shapes in the algebra U_{12} and on their intersections and boundaries. However, many of the presented concepts can be generalized to other algebras, which is why, in this thesis, line segments are often referred to as elements.

Calculating with shapes is governed by the *embedding relation*, which defines how the elements of shapes interact. Similar to set theory, if one adds two identical elements together, the result is just one element, as in (a) in Figure 2.3. If, like in this case, the maximal elements of two shapes overlap completely, we say that the two shapes *embed* into each other. The reduction rules are a consequence of the embedding relation, since the addition of overlapping or adjoining elements results in maximal elements, as in (a) and (c) in Figure 2.3. In the case of (c), the partial square embeds into the square, but not vice versa. In other words, the partial square is a *subshape* of the square. If, on the other hand, one

[2] If not noted otherwise, the definitions in this section follow Stiny (2006). For a chronological survey of the development of shape and shape grammar definitions see Yue (2009, pp. 25-43).

subtracts two identical elements from each other, as in (b) in Figure 2.3, the result is the empty shape. Hence, when visually calculating with shape S,

$$S + S = S$$

and

$$S - S = \varnothing.$$

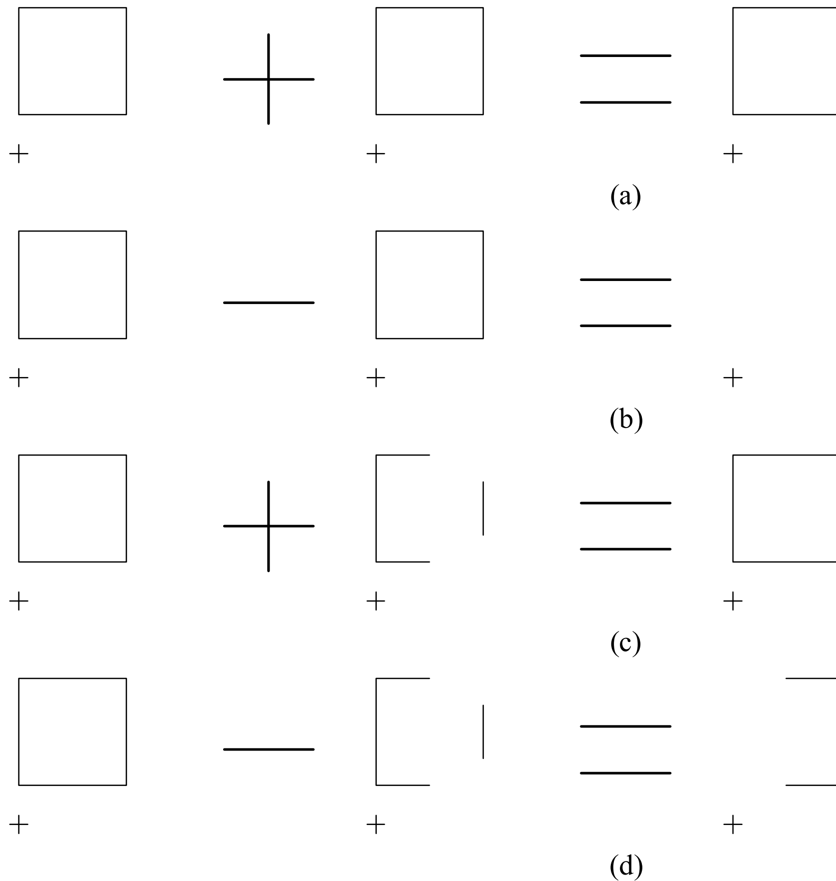


Figure 2.3 *(a) shows the addition of two squares; (b), the subtraction. In (c), parts of a square are added to a square and, in (d), parts of a square a subtracted from a square.*

Complementing the embedding relation, the *part relation* is another critical property of visual calculating: Since maximal elements of $i > 0$ (i.e., elements that are not points) potentially contain an indefinite number of smaller elements, any combination of parts of maximal elements, i.e., any subshape, can be seen as a new shape. In other words, a shape can be decomposed in indefinitely many ways. (This is where visual calculating deviates from set theory, since a set has defined decompositions based on its members.) For example, in (d) in Figure 2.3, a square is decomposed by subtracting parts of a square from

it. More examples are given in Figure 2.4, where the two overlapping squares of shape (a) create another square, with the sides of the third, smaller, square formed by parts of maximal elements, as demonstrated by decomposition (b). Subshapes like the third square are also called *emergent* shapes and offer interesting opportunities for rule application. (For example, rule (a) from Figure 2.1 can be cursively applied to the third square created by the two overlapping squares, which accounts for the derivation in Figure 2.2.) As is visible in Figure 2.4, many other, in fact, indefinite, decompositions are possible for shape (a). (Note how (e), (f), and (g) divide (a) in an arbitrary fashion, without regard to intersection points.) The embedding and part relations thus allow a designer to see shapes in ambiguous and even mutually exclusive ways and in this way encourage the serendipitous developments that take designs in new directions.

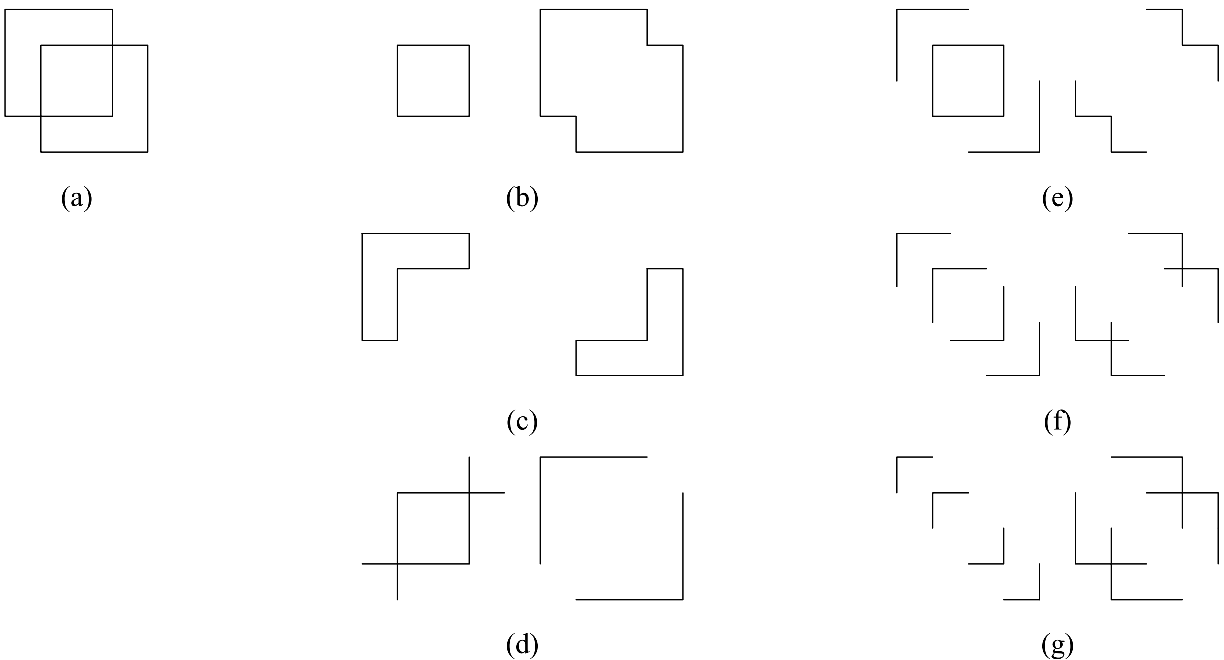


Figure 2.4 *In each drawing (b) – (g), shape (a) is decomposed in a different fashion. Note that a decomposition does not have to be based on intersection points, as is the case for (e), (f), and (g).*

Both the embedding and part relations are crucial for rule application. One can apply a rule only on shapes where the left-hand shape of the rule embeds into, in other words, when one can find a shape or subshape that is the same as the left-hand shape of the rule. (Shapes or subshapes that are the "same" are called *isomorphic*.) The embedding relation ensures that the elements of a shape are maximal, which, via the part relation, allows a designer to select any combination of a shape's parts for rule application. When, in this way, a matching shape or subshape S_A is found, a rule is applied to a S_A by subtracting the rule's

left-hand shape S_L from the original shape, and then adding the rule's right-hand shape S_R, resulting in shape S_B (also see Figure 2.5.):

$$S_L \rightarrow S_R$$

$$S_A - t(S_L) + t(S_R) = S_B.$$

Note that rule application necessarily involves a transformation t that maps S_L onto S_A , since rules are defined separately from the designs that they create. Like the embedding and part relations, this transformation is an important aspect of seeing shapes in different ways, and is discussed in more detail in the next section.

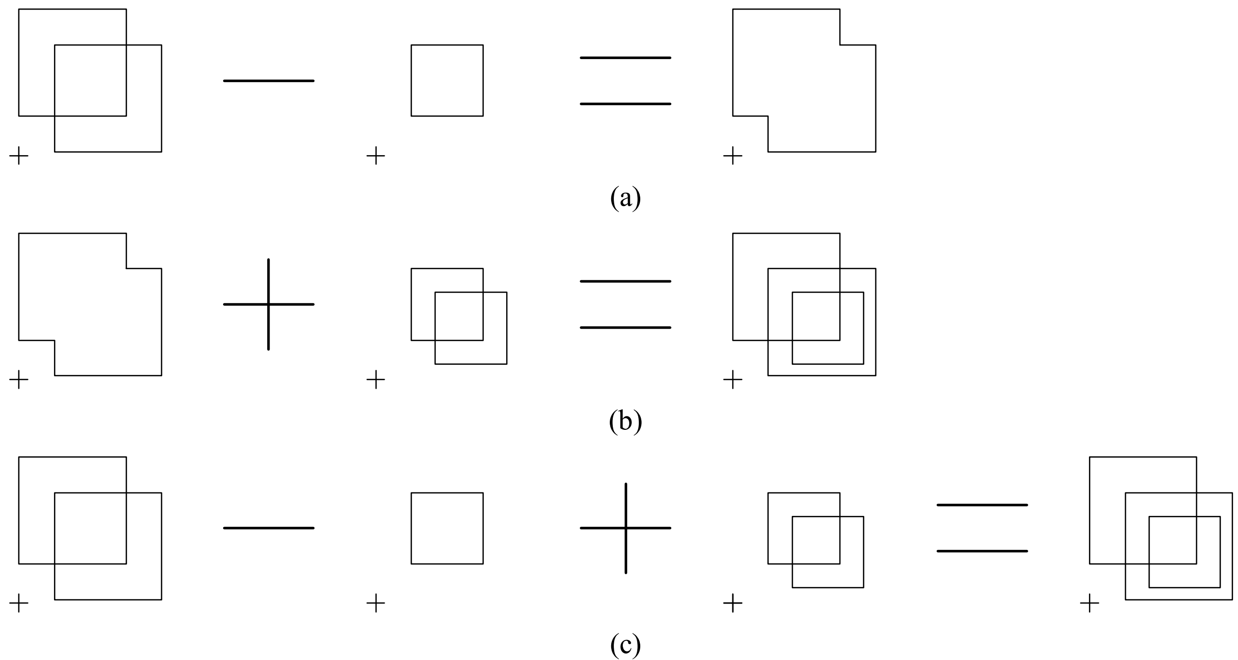


Figure 2.5 *This figure shows the application of rule (a) from Figure 2.1 through difference and union. In (a), the subshape to which the rule is applied is subtracted from the shape. In (b), the right-hand side of the rule is added to the remainder. (c) shows how (a) and (b) combine to apply the rule.*

2.3 Transformations

Transformations govern which shapes are valid "inputs" for a rule. As discussed in the previous section, one can apply a rule to a shape when one sees that the rule's left-hand side embeds into the shape. Formally, a rule is applicable to (sub-)shapes that are *isomorphic* to the left-hand shape of the rule, and two shapes are isomorphic when there is a one-to-one mapping (i.e., a bijection) t between their maximal

elements that preserves the appropriate relationships (such as proportions, angles, etc.) between them. Then, two shapes S_A and S_B embed into each other when

$$\in t \mid t(S_A) = S_B.$$

More often, S_A is isomorphic to a part of S_B. Then, S_A embeds into S_B as a *subshape*:

$$\in t \mid t(S_A) \subseteq S_B.$$

Importantly, there are different transformations t that, according to the intentions of the designer, can govern a visual calculation. For example, all of the shapes in Figure 2.1 can, under different transformations, be seen as isomorphic to a square. A set of shapes that is isomorphic for a given transformation t is also called an *equivalence class*.

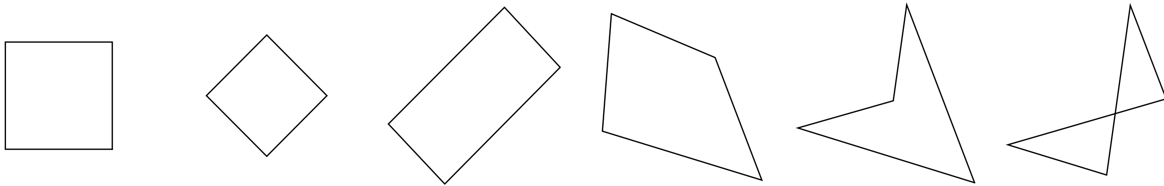


Figure 2.6 *Isomorphisms of a square based on different transformations t.*

Visual calculations can be characterized by their transformations. In this thesis, five sets of transformations are distinguished: *Translation*, *Isometry*, *Similarity*, *Linearity*, and *Everything else*. Five derivations based on these transformations and rule (a) from Figure 2.1 are given in Figure 2.7. Figure 2.8 gives an overview of the transformations included in the different categories.

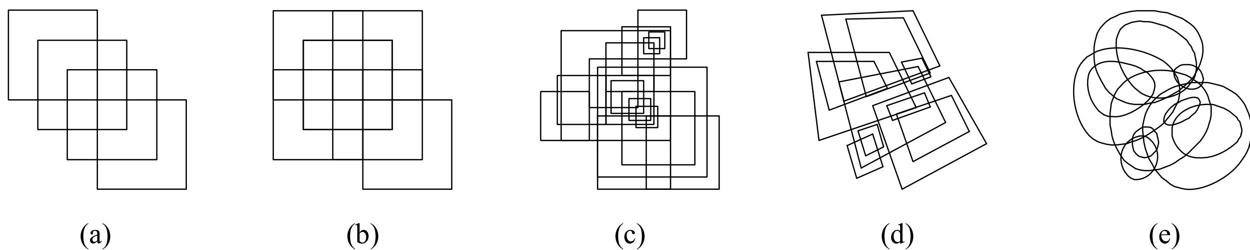


Figure 2.7 *Five derivations based on rule (a) from Figure 2.1. Derivation (a) is based on translation and (b) on isometry. Similarity governs (c), and linearity (d). Finally, (e) is an example of a derivation based on everything else, which in this case means curved shapes.*

TRANSLATION

In this category, the only allowed transformation is translation. In other words, a shape can only be moved, but not rotated or mirrored. Consequently, there is only one way in which rule (a) from Figure 2.1 can apply to the squares in derivation (a) in Figure 2.7, leading to the diagonal arrangement.

ISOMETRY

Isometry includes the rigid, or Euclidean, transformations. Here a shape can be translated, rotated, and mirrored. As implied by the term "rigid", these transformations preserve a shape's size, proportions, and angles, though not necessarily its orientation. In derivation (b) in Figure 2.7, rule (a) from Figure 2.1 is applied in different orientations based on the symmetry of the square. However, the rule cannot apply to the smaller and larger squares that emerge from the overlap of the original ones, since scaling is excluded.

SIMILARITY

Many shape grammars allow similarity transformations, which, next to the Euclidean transformations, include proportional scaling. Proportional scaling preserves a shape's proportions and angles, but not its size, as is visible in derivation (c) in Figure 2.7. In contrast to isometry, in this case rule (a) from Figure 2.1 can apply to different sized squares.

LINERARITY

Linear transformations include all transformations that can be expressed as a transformation matrix[3]. This includes translation, isometry and similarity, but also affine transformations, perspective projections, etc. In a linear transformation, the angles and proportions of a shape are not preserved, in contrast to more general characteristics, like maximal elements and convexity. This also is the case in derivation (d) in Figure 2.7. In the special case of an affine transformation, collinearity and ratios of distances are preserved as well.

EVERYTHING ELSE

This last category covers all cases, including ones where the transformation cannot be expressed as a transformation matrix. For example, shape isomorphism can be topologically defined based on relations of connectivity between intersection points, which is the case for derivation (e) in Figure 2.7, where rule (a) from Figure 2.1 applies to all closed shapes, including curved ones. One can also preserve other characteristics such as the convexity or number of sides of a shape. Another possibility is to base shape

[3] Transformations that can be expressed by a transformation matrix are convenient for computer implementation.

isomorphism on the area enclosed by a shape, or the fact that a shape is a polygon, etc. etc. There are thus no limits to the designer's imagination.

TRANSLATION	ISOMETRY	SIMILARITY	LINEARITY	EVERYTHING ELSE
Translation	Translation	Translation	Translation	Translation
	Rotation	Rotation	Rotation	Rotation
	Mirroring	Mirroring	Mirroring	Mirroring
		Proportional Scaling	Proportional Scaling	Proportional Scaling
			Linear Transformations	Linear Transformations
				Everything Else
		NON-PARAMETRIC	PARAMETRIC	

Figure 2.8 *In this table, the columns represent the five sets of transformations. Each row represents an individual transformation. Note how each set contains progressively more transformations. Also note the division between non-parametric and parametric (i.e., non-similar) sets of transformations.*

In the literature, the translation, isometry and similarity transformations are collectively referred to as *non-parametric* transformations, while non-similar transformations are also known as *parametric* transformations. Parametric shape grammars were first introduced as grammars with *open terms*, i.e., shapes without fixed proportions or angles (Stiny, 1980). Subsequently, visual calculating was defined for general transformations *t* (Stiny, 1990). While, as discussed in section 2.5, parametric visual calculating is hard to implement on a computer, the linear and some of the simpler non-linear cases can be addressed by representing shapes as graphs, which is the motivation for this thesis. Since shape isomorphism based on parametric transformations has fewer constraints, it represents a generalization of non-parametric shape isomorphism. However, both are specific examples of the transformation *t*, whose generality imbues visual calculating with power and flexibility by allowing a designer to apply rules according to her personal way of seeing. Transformations thus are another way in which visual calculating supports design freedom, next to the unlimited opportunities for decomposing shapes discussed in the previous section. The next section discusses the problem of decomposition, which arises when one symbolically represents

visual designs, for example, in computer programs. Further aspects of the computer implementation of visual calculating are discussed in the last two sections.

2.4 The Problem of Decomposition

In comparing visual calculating with conventional, symbolic calculating, the probably largest difference is constituted by maximal elements. While, as discussed in section 2.2, visual calculating represents shapes by the smallest set of the largest possible elements, symbolic calculating often breaks elements into their smallest constituents. Digital computers, for example, represent data as zeros and ones. However, it is impossible to define the parts of a shape in advance while allowing a designer all the ways of seeing that she might want to use. The part relation allows indefinite decompositions for shapes, all of which might be useful to a designer. In other words, shapes are inherently ambiguous in terms of their "atomic designs" (Stiny, 1987), which is why decompositions can only be applied retroactively (Stiny, 1994). Decompositions are one of the key challenges for the computer implementation of visual calculating. If a symbolic representation imposes an inappropriate decomposition onto a shape, a shape rule that would otherwise apply might not be applicable. Often, different design derivations decompose a shape in mutually exclusive ways. The problem of ambiguity in decomposing shapes is discussed at length in Stiny (2006, pp. 65-70), where a description of shapes according to their "lowest-level constituents", proposed by a computer scientist, is given as a dissuasive example. The problem of decomposition is of great significance for design, since, arguably, the ability to re-see a given situation in a new light, or, in this case, to decompose a shape in a different way is a key competence for any creative activity. Art students learn how to escape "traditional habits of daily perception" (Eisner, 2002, p. 68), while engineering students must at least "tolerate ambiguity … in viewing design" (Dym, Agogino, Eris, Fry, & Leifer, 2005).

Not coincidentally, the problem of decomposition is encountered in many conventional CAD programs and with computational design tools for architecture such as BIM, parametric[4] design, or scripting, where the parts of shapes are fixed in advance. (See Kolarevic (2003) for an overview on computational design tools in architecture.) Often, the manner in which these tools represent designs makes it hard for a designer to follow up on new ideas or to see designs from a new perspective. A simple example is that, in a CAD program, the third square created by two overlapping squares (see (a) in Figure 2.4) would mostly likely not be represented as a square. Developing symbolic representations for shapes without

[4] Parametric in the context of conventional digital design tools means objects with varying parameters, as opposed to the parametric transformations discussed in the previous section.

inappropriate decompositions, like the graph-based representation presented in this thesis, thus promises a new generation of more designerly computational design tools.

2.5 Symbolically Representing Visual Calculating

Visual calculating bridges design freedom and creativity on the one hand, and the need for explicit design methodologies on the other. Calculating with shapes gives designers the freedom to see and re-see, but affords enough rigor to formalize designs when necessary, for example, for pedagogy or design automation. Many conventional design methods lack the freedom to see things differently, for example, computational design tools such as parametric modeling and BIM or engineering design methods based on atomic decompositions. (For examples of engineering design methods, see "Functional Analysis" or "Quality Function Development" in (Otto & Wood, 2001).) Given that visual calculating is a designerly, yet rule-based design methodology, it seems a natural candidate for computer implementation. In other words, it should be possible and useful to represent design processes symbolically on a computer, as they can be explicitly formalized as visual calculations. Since computers are designed for automated rule application, a symbolic representation for visual calculating promises greater speed and convenience relative to hand calculation, while the virtual display of the computer screen allows geometries that are hard to control with pen-and-paper (such as curves or three-dimensional shapes, etc.). A computer implementation of visual calculating thus promises exploratory design tools that support regularities in design processes, as well as spontaneous and surprising changes. Indeed, as discussed below, there have been many computer implementations of visual calculating in the past thirty years, though most were limited in scope and did not find a broad audience. An ideal computer implementation, described by Gips (1999) as "the Big Enchilada", should be able to deal with a broad range of shapes, such as parametric shapes, curved shapes, shapes with complex labels, etc., and would therefore be "qualitatively more useful than today's programs" (ibid.). For the past decade, this vision has remained unrealized; however, one can understand the graph representation developed in this thesis as a step towards this goal.

A fundamental question for the computer implementation of visual calculating is symbolic representation. Since digital computers manipulate only symbols, the step from visual calculation to symbolic computation requires the symbolic representation of visual shapes. Symbolic representations allow access to the powerful machinery of digital computers, but at the risk of losing something essential in terms of design freedom, as already touched upon with the problem of decomposition in the previous section. The tension between visual calculating as a rule-based, though intuitive creative activity and symbolic computation with its desire for explicitly defined problems is expressed in this quote:

It is claimed that, in design, ambiguity serves a positive and deliberate function. In principle, shape grammars can be devised to take advantage of ambiguity in creating novel designs. However, ambiguity, in general, is inherently counter-computable (...). (Yue & Krishnamurti, 2008)

The key challenge for computer implementation is thus to preserve as much ambiguity as possible. Specifically, this means that shapes should be symbolically represented without imposing inadequate decompositions. For example, shapes should not be represented with a (necessarily finite) set of pre-defined types, since pre-defined types have fixed decompositions instead of allowing many. Avoiding inadequate decompositions means that a computer implementation should preserve the embedding and part relations, which leads to the problem of *subshape recognition*, i.e., the problem of discovering, in a given design, subshapes that are isomorphic to the left-hand side of a rule. Subshape recognition is a hard computational problem because of the embedding and part relations. It is difficult for computers to find *emergent* subshapes, like the square created by the two overlapping squares in (a) in Figure 2.4. Chase (1997) sees the embedding and part relations as critical challenges for the computer implementation of visual calculating. These challenges are directly related to the question of how shapes are represented, since, as we will see below, different computational techniques are available for different types of representations. Several non-parametric implementations have successfully addressed subshape recognition by employing geometric representations.

Another challenge is posed by the parametric transformations discussed in section 2.3. The implementation of parametric transformations needs flexible representations of shapes that preserve key properties while releasing others. Representations for parametric shapes severely complicate subshape recognition, since, for a parametric shape, one can no longer find subshapes based on the shape's geometry, such as angles or proportions. Extending this insight, Yue, Krishnamurti, and Gobler (2009) claim that "it is impossible to implement a parametric shape grammar interpreter" which can handle more than "a subset of grammars". As examples for subsets, they mention shape grammars without the embedding and part relations or shape grammars based exclusively on rectangles. Although this claim of impossibility is probably exaggerated, we will see below that there are not many representations for parametric shapes, and that most of these representations do not support the embedding and part relations. Paradoxically, some representations for parametric shapes support the embedding and part relations while categorizing shapes according to pre-defined hierarchies or types, which is another way of imposing potentially inadequate decompositions.

An important detail of the symbolic representation of shapes is numeric precision. In a computer, numbers are stored with finite numbers of digits, which can lead to problems of accuracy such that shapes

that should be recognized as isomorphic are not, and vice versa. Krishnamurti (1980) addresses this problem by constraining shapes and their transformations to rational numbers (i.e., numbers that can be represented as integer fractions). Tapia (1996, pp. 84-86) constrains shapes to a grid, which allows him to represent shapes with only integer numbers. In three dimensions, the problem of accuracy becomes more acute (Stouffs, 1994, pp. 185-187). However, while important for actual computer implementation, the problem of numeric precision is not a focus of this thesis. The next section will give an overview of existing symbolic representations and the corresponding computer implementations.

2.6 Precedents for Symbolic Representation and Computer Implementation

The below overview categorizes existing symbolic representations and the computer implementations based on those representations according to the three aspects of the (non-)employment of pre-defined types, embedding (including the part relation), and parametric transformations, with a focus on relatively recent contributions[5].

CATEGORY I *Representations with pre-defined types without embedding and parametric shapes*

Implementing a system for visual calculating without the embedding and part relations is computationally straightforward, since one can represent shapes as discrete objects, without having to test for emergent subshapes. Out of twenty-one implementations listed by Chau, Chen, McKay, and de Pennington (2004), twelve do not support the embedding and part relations. Of those twelve, about ten are non-parametric. A relatively recent example is Shaper 2D by McGill (2002), a purposefully restricted, two-dimensional implementation for educational purposes. In three dimensions, a similar approach has been taken by Hoisl and Shea (2009). From a computational point of view, the implementation of visual calculating without the embedding and part relations or parametric shapes is unproblematic. However, because of their limited functionality, implementations in this category also are relatively uninteresting as tools for exploratory design.

CATEGORY II *Representations with embedding without pre-defined types and parametric shapes*

Chau, et al. (2004) list eight implementations that support the embedding and part relations, though apparently none are parametric. Much of the work in this category is based on an algorithm for subshape recognition in two dimensions described by Krishnamurti (1981). Krishnamurti's algorithm relies on a reference triangle that is defined by three randomly chosen, non-collinear points of a shape. Alternative

[5] For more extensive accounts of existing implementations of visual calculating see (Gips, 1999), (Chau, Chen, McKay, & de Pennington, 2004), and (Yue, 2009, pp. 21-25).

algorithms and a corresponding two-dimensional implementation were developed by Chase (1989). GRAIL is a three-dimensional implementation presented by Krishnamurti (1992) that is based on plane segments. GEdit, developed by Tapia (1999), is considered one of the best implementations, mainly because of its interface (Chau, Chen, McKay, & de Pennington, 2004). In GEdit, subshape recognition is achieved by constraining the ends of line segments to a pre-defined, two-dimensional grid. In that way, a shape can be described with a reference rectangle defined by two diagonal grid points. This reference grid defines a "maximum resolution", which can be seen as imposing a definite decomposition and thus as a limitation of design freedom. Chau, et al. (2004) present an implementation for calculating with lines in three dimensions. This implementation has been extended with a graphical user interface by Li, Chau, Chen, and Wang (2009). Stouffs and Krishnamurti (2004) have provided a "unified foundation for arithmetic in any shape algebra", i.e., a geometric representation for shapes composed of points, line segments, or plane segments in one, two, or three dimensions. They also have extended this representation to the boundaries of three-dimensional shapes (Stouffs & Krishnamurti, 2006). SGI by Trescak, Rodrigues, and Esteva (2009) is a two-dimensional implementation notable for significantly improving Krishnamurti (1981). Jowers and Earl (2011) present a two-dimensional implementation for curved shapes based on the mathematics of quadratic Bézier curves. (The mathematical representation of curved shapes is discussed in (Jowers & Earl, 2010).)

The approaches in this category that were examined so far rely on the specific geometry of the shape for subshape recognition, for example, a reference triangle or rectangle defined by the shape's points, or the mathematical properties of curves. In other words, shapes are represented geometrically. However, since these approaches rely on the specific geometry of shapes, such as their angles and proportions, they are not feasible for parametric shapes. There are, however, two notable alternative approaches: Pixels and graphs. Keles, Özkar, and Tari (2010) represent shapes as graphs, which, as will be explained below, is a promising approach for parametric implementation. (In this case, the implementation is non-parametric, with subshape recognition based on Krishnamurti (1981). The next chapter discusses graphs, as well as the specifics of the representations by Keles et al. (2010).) In a more recent paper, Keles, Özkar, and Tari (2012) outline an implementation employing image recognition and a pixel-based representation. On the one hand, pixel-based approaches enable the use of arbitrarily curved shapes and the integration of hand-drawn sketches, as also demonstrated by Jowers, Hogg, McKay, Chau, and de Pennington (2010). On the other hand, pixel-based approaches are very intensive in terms of computation, since one needs to compare individual pixels instead of maximal elements. Moreover, similarly to GEdit's grid-based approach, pixel-based approaches are limited by a maximum resolution, in this case by the number of pixels. Pixel-based approaches do not lend themselves to parametric implementation, since, as discussed

in section 2.3, isomorphic parametric shapes can be visually different from each other. This category is largely resolved in terms of fundamental computational problems, however, as demonstrated by some of the precedents, there are still interesting areas to explore, for example, different geometries or better user interfaces. However, both of these topics are beyond the scope of this thesis.

CATEGORY III *Representations with pre-defined types and parametric shapes without embedding*

Compared to categories I and II, there are significantly fewer implementations in this category. The two implementations listed by Chau, et al. (2004) are application specific, i.e., they handle only a set of parametric shapes whose representations have been fixed in advance. Consequently, they generate specific types of objects. For example, the implementation of the parametric, three-dimensional Queen Anne grammar by Flemming (1987) produces only houses in the Queen Anne style. Similarly, Agarwal and Cagan (1998) have implemented their "language of coffee makers" to generate three-dimensional coffee makers from parametrically defined components. These types of implementations are close to contemporary digital design tools like scripting or parametric CAD programs, which also employ pre-defined parametric objects. A more general approach is taken by Yue (2009, pp. 115-121), who presents a representation for parametric polygonal shapes based on graph grammars, though without the embedding relation. In terms of computational problems, this category seems resolved, since, like in category I, shapes are represented as discrete objects without the need to test for emergent subshapes. This category offers little in terms of exploratory design tools because of the limitations to design freedom inherent in this approach. In the last two decades, commercial parametric design programs with capabilities similar to the precedents in this category (e.g., CATIA, Revit, etc.) have become widely used.

CATEGORY IV *Representations with pre-defined types, embedding and parametric shapes*

Few precedents implement visual calculating with the embedding and part relations and parametric shapes. Chau, et al. (2004) list no implementations with this capability. The earliest attempt is by Nagakura (1995) and relies on pre-defined parametric shapes in a manner similar to the implementations in category III. Here, the designer needs to explicitly program definitions for shapes before he can design with them. (For example, see ibid., 272.) Support for the part relation is therefore limited to subshapes that have been explicitly defined in advance. McCormack and Cagan (2002) present a two-dimensional parametric shape representation based on a hierarchy of constraints, which they also apply to curved shapes (McCormack & Cagan, 2006). This hierarchy, for example, always preserves ninety degree angles, while allowing other angles are to vary. According to the authors, "the levels of the hierarchy are defined so that the most constrained lines of a shape are those lines that the designer intended exactly" (ibid., 915). However, this approach is based on the doubtful assumption that the designer's intentions are

predictable and constant. As argued in section 2.5, representations of shapes based on pre-defined types, like the ones in this category, are severely limiting design freedom. Pre-defined types are often defended with claims about what a designer will or will not do. For example, Nagakura assumes that “although it is possible to subdivide a line segment at any point regardless of intersection (phantom subdivision) . . . this is unlikely because of the psychology of human image recognition” (Nagakura, 1995, p. 72). However, this type of argument only underscores that the unpredictability of designers is a defining characteristic of the creative process. It is for this reason that, instead of balancing “utmost flexibility” with “the cost of losing useful explicit structures in the drawing” (Nagakura, 1995, p. 87), computer implementations of visual calculating should preserve as much design freedom as possible. (An important, but secondary consideration is the development of an easy-to-use user interface to manage the complexity that can arise from this freedom.) In terms of developing better tools for exploratory design, this category is problematic, with the precedents inhabiting an unstable middle ground between the pre-defined objects in category III, and the flexible representations of categories II and V.

CATEGORY V *Representations without pre-defined types with embedding and parametric shapes*

This thesis presents a representation for parametric shapes with the embedding and part relations and without the pre-definitions of the representations in category IV. The representation is graph-based, since, as discussed above, other approaches based on pixels or geometry are limited to non-parametric shapes. Graphs on the other hand represent topological relationships of connectivity that can be constrained according to various properties such as angles, proportions, convexity, parallelism, etc. In this manner, a variety of parametric and non-parametric transformations can be represented. Critically, graph representations of this type can be constructed “on the fly”. (Algorithms that construct a graph representation in polynomial time are described in chapter 4.) In fact, a new graph can be constructed with every change of the design, i.e., there is no need to maintain a specific symbolic representation as the design changes. Subshape recognition is achieved by searching for subgraphs in graphs. This type of search is also known as *subgraph matching* and is a topic of research in the field of graph-grammars[6] (see, for example, (Batz, 2006)).

---

[6] Subgraph matching is known to be an NP-complete problem (Batz, 2006). The parametric implementation of visual calculating therefore is an NP-complete problem insofar such an implementation relies on graphs for subshape recognition. In a sense, this reasoning supports the claim by Yue et al. (2009) that “parametric subshape recognition, in general, is NP-hard”, which, as mentioned in the previous section, leads them to claim that the implementation of visual calculating is possible only for special cases. However, their proof relies on relating subshape recognition to the clique problem, which is known to be NP-hard only for non-planar graphs. (See (Nishizeki & Chiba, 2008) for a polynomial algorithm for finding cliques in planar graphs.) Since, by definition, shapes are always planar, this proof only is correct for non-planar graph representations of shapes. (See section 3.2 for a discussion of planar and non-planar graphs, and how they relate to shapes.) The algorithmic complexity of

Another, very similar precedent in this category was developed by Grasl and Economou (2011), though only for shapes composed of connected maximal elements. (The representation developed in this thesis also extends to non-maximally connected shapes.) Grasl and Economou (ibid.) employ a graph algorithms library to find candidate parametric subshapes by searching for isomorphic subgraphs. In a second step, these candidates are filtered according to various geometric constraints.

As discussed above, graph-based representations for shapes have various advantages that make them promising candidates for the development of exploratory design tools based on parametric visual calculating. They are flexible enough to accommodate different geometries, labels etc., however, this is a topic for further research. After a general introduction to graphs and how they can symbolically represent shapes, the next chapter compares the approach of this thesis with the approaches of Keles et al. (2010) and Grasl and Economou (2011).

---

parametric subshape recognition therefore depends on the choice of symbolic representation. Also, note that, although subgraph matching for non-planar graphs is NP-complete, good results are possible in practice. For example, Batz (2006) presents an algorithm that can solve "several problem instances which may occur in practice . . . in reasonable time".

3 Shapes as Graphs

This chapter recapitulates formal definitions for graphs, examines the similarities and differences between shapes and graphs, and compares five basic possibilities for representing shapes as graphs. One can choose between a maximal graph, where maximal elements, e.g., line segments, are graph edges and boundaries are nodes, a direct graph, where maximal elements are edges and the elements' intersections are nodes, an over-complete graph, where, compared to the direct graph, an element is represented with additional edges, an inverted graph, where elements are nodes and intersections are edges, and an elaborate graph that represents both elements and intersections as nodes. From these possibilities, the inverted graph is chosen for its suitability for computer implementation, as well as its relative compactness, clarity, and proximity to the original definition of shapes.

3.1 Graphs and Graph Isomorphism

In graph theory, a graph is defined as an ordered pair *(V, E)*, where *V* is a set of vertices and *E* is a set of edges disjoint from *V*, together with an *incidence function* $\psi(e) = \{u, v\}$ that associates each edge with a pair of vertices (Bondy & Murty, 2008). Similar to maximal elements in shapes, edges and/or vertices can be labeled, for example, with weights or colors. A graph with ordered vertex pairs is called a *directed graph*. In a *multigraph*, multiple edges can join the same pair of vertices. Graphs can easily be visualized by drawing them on the plane. For *planar graphs,* this mapping is possible without intersecting edges. Various graph types are illustrated by Figure 3.1.

Two graphs $G_A = (V_A, E_A)$ and $G_B = (V_B, E_B)$ are *isomorphic* when there are mappings φ_V and φ_E such that for every vertex v_a in V_A, there is a corresponding vertex v_b in V_B and vice versa (if applicable with matching labels), and for every edge e_a in E_A there is a corresponding edge e_b in E_B and vice versa (if applicable with matching direction and labels):

$G_A = G_B$
when
$\in \varphi(\varphi_V : V_A \to V_B,\ \varphi_E : E_A \to E_B)$
such that
$\psi_A(e_a) = u_a v_a \Leftrightarrow \psi_B(\varphi_E(e_a)) = \varphi_V(u_a)\varphi_V(v_a)$
and
$\psi_B(e_b) = u_b v_b \Leftrightarrow \psi_A(\varphi_E(e_b)) = \varphi_V(u_b)\varphi_V(v_b))$.

G_A is a *subgraph* of G_B when there are mappings φ_V and φ_E such that for every vertex v_a in V_A, there is a corresponding vertex v_b in $V_{B,}$ and for every edge e_a in E_A there is a corresponding edge e_b in E_B:

$G_A \subseteq G_B$
when
$\in \varphi(\varphi_V : V_A \to V_B,\ \varphi_E : E_A \to E_B))$
such that
$\psi_A(e_a) = u_a v_a \Leftrightarrow \psi_B(\varphi_E(e_a)) = \varphi_V(u_a)\varphi_V(v_a)$.

For example, in Figure 3.1, graphs (a) and (b) are isomorphic, and both are subgraphs of graph (f). Similar to intersection points and line segments for parametric shapes in U_{12}, the mapping of nodes and edges preserves topological relationships of connectivity. Additional affinities and differences between shapes and graphs are discussed in the next section.

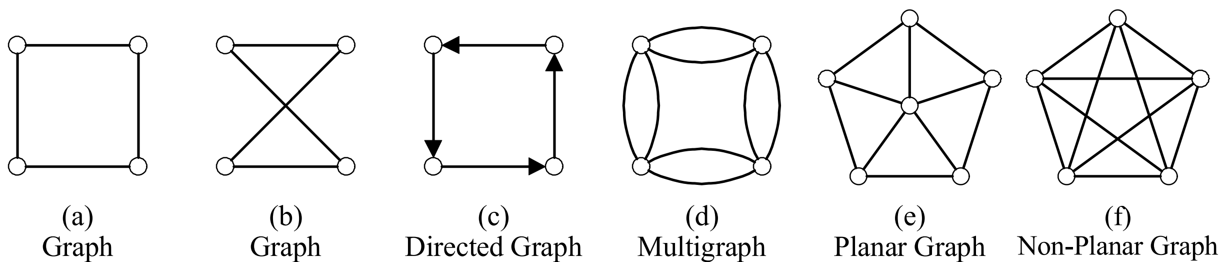


Figure 3.1 *Types of Graphs.*

3.2 Differences between Shapes and Graphs

Shapes and graphs can be visually similar in that both can consist of connected line segments. There also is a formal parallelism between subshapes and subgraphs, and both shapes and graphs can carry labels. A graph representation for shapes might therefore seem both natural and unproblematic. However, there also are a number of important differences:

Most importantly, graphs lack the embedding and part relations, which are defining characteristics of visual calculating. As discussed in section 2.2, maximal elements are more-dimensional and contain indefinite decompositions of themselves, while graph edges are zero-dimensional *symbols*. One might easily express a graph as a labeled (and thus symbolic) shape, but a graph by definition is a symbolic representation. In practice, this means that when a shape is directly interpreted as a graph, intersections become nodes and intersected maximal elements are split into multiple graph edges (for an example, see Figure 3.2). This splitting of maximal elements imposes a definite and potentially inappropriate decomposition. Specifically, as demonstrated in Figure 3.2, one cannot easily recognize parts composed of more than one collinear segment. (The problem of decomposition is discussed at length in section 2.4.)

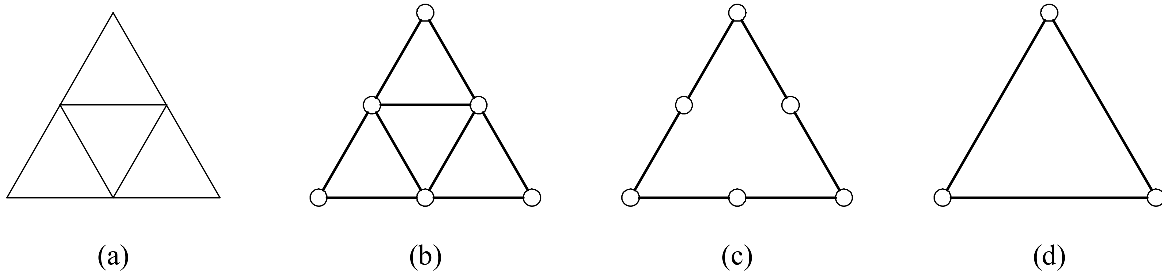


Figure 3.2 *Shape (a) is directly interpreted as a graph, resulting in graph (b). The large triangle in shape (a) is similar, and thus isomorphic, to the small triangles in (a). However, the graph of the large triangle, (c), is different from the graph of a small triangle, (d), which is an undesirable distinction.*

The subtle, yet fundamental distinction between shapes as visual entities and graphs as graphical symbols also lies at the basis of the second difference. While shapes are interpreted by the transformation *t*, graphs symbolically represent topological relationships of connectivity. (In other words, graphs always are interpreted by the same "transformation.") As long as connectivity relations are preserved, the representation of nodes and graph edges can be manipulated freely. The same graph can thus have very different visual representations, as demonstrated by the two isomorphic graphs (a) and (b) in Figure 3.1. For a shape, the transformation *t* that will allow it to be recognized as isomorphic with another shape will often be narrower, though it can at times also be wider, as discussed in section 2.3. For example, one might want to distinguish between convex, concave, and self-intersecting shapes or accept all polygons as isomorphic. This difference is acknowledged in the graph-grammar-based representation by Grasl and Economou (2011), where a number of geometric conditions filter isomorphic shapes from a larger set of isomorphic graphs.

The third and final difference is related to the previous point about graphs as symbolic representations of connectivity. While intersection points exist implicitly in shapes as a consequence of intersecting maximal elements, in a graph connections between edges are explicitly symbolized by nodes. (Note that nodes are included in the formal definition of a graph, while intersections are not included in the formal definition of a shape.) This explicit representation allows topologically equivalent mappings and is crucial for the distinction between non-planar and planar graphs. A planar graph can be drawn in the plane such that all edge intersections are nodes, while, in non-planar graphs, some edge intersections are not recognized. (For example, compare graphs (e) and (f) in Figure 3.1. Also note that (b) in Figure 3.1 is, in fact, planar, since it can be drawn into the plane without intersecting edges. It that case, it would look like (a).) In shapes, on the other hand, all intersections of maximal elements are equally valid, since a shape is a visual configuration instead of a symbolic representation. In other words, there can be no such thing as a

non-planar shape. Informed by the three differences of embedding, transformations, and intersections, the concluding section of this chapter discusses three different possibilities for representing shapes as graphs.

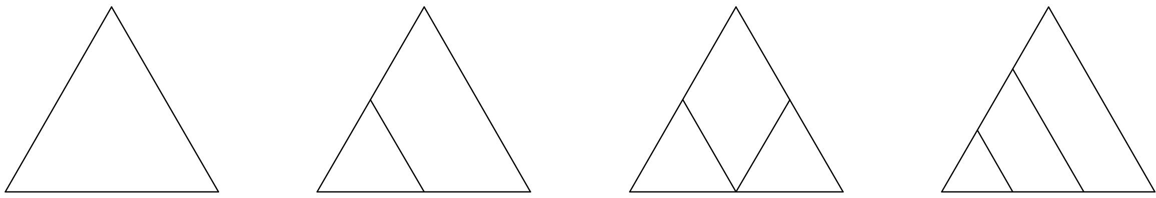
3.3 Representing Shapes as Graphs

As flexible data structures with numerous possibilities, graphs afford many choices in representing shapes, without one necessarily being the most "correct". However, one can propose a number of criteria that will aid in the following discussion. At the very least, the representation should capture the defining properties of a shape. These properties include the embedding and part relations, as well as maximal elements and their intersections. This is the case for only two of the five possibilities discussed in this section. Secondly, the representation should be as compact as possible. In other words, it should include relatively small numbers of nodes and edges to optimize computational processing. Lastly, the representation should be analytically clear with a transparent mapping between shape and graph. This aesthetic consideration is less important for actual computer implementations, but relevant for discussion and pedagogy.

Given the criteria of completeness, compactness, and transparency, five options deserve attention[7]. The earliest approach was introduced by Stiny (1975) and can be called the *maximal graph*. Here, each maximal element is represented as a graph edge, and its two element boundaries as nodes. Intersections are not represented. (See graphs (1b) to (4b) in Figure 3.3.) Maximal graphs clearly show the maximal elements of a shape and preserve the embedding and part relations. However, because they do not include intersections, maximal graphs tend to consist of several unconnected parts, as is visible for graphs (2b), (3b), and (4b) in Figure 3.3. This un-connectedness makes the maximal graph unsuitable for computer implementation. It also is incomplete since it does not include intersections. For n elements, the maximal graph requires $\Theta(n)$ edges and $\Theta(n)$ nodes.

A second possibility is the *direct graph*, which represents maximal elements as edges and intersections as nodes. For shapes in U_{12}, direct graphs have the appeal of being visually very similar to the original shape, as can be seen for graphs (1c) to (4c) in Figure 3.3. Following this approach, Yue (2009, pp. 115-121) presents a graph grammar-based representation, which, however, does not support embedding. As discussed in the previous section, direct graphs tend to split maximal elements, which prevents the indefinite decompositions afforded by the embedding and part relations. (This is the case for (2c), (3c), and (4c).) Consequently, the direct graph is incomplete. For n elements with a maximum of n^2 intersections, the direct graph requires $O(n^2)$ edges and $O(n^2)$ nodes.

[7] For additional options, including the employment of hyperedges, see Grasl and Economou (2011).



(1a) Shape 1 | (2a) Shape 2 | (3a) Shape 3 | (4a) Shape 4

(1b) 3 Nodes 3 Edges | (2b) 5 Nodes 4 Edges | (3b) 6 Nodes 5 Edges | (4b) 7 Nodes 5 Edges

(1c) 3 Nodes 3 Edges | (2c) 5 Nodes 6 Edges | (3c) 6 Nodes 8 Edges | (4c) 7 Nodes 9 Edges

(1d) 3 Nodes 3 Edges | (2d) 5 Nodes 8 Edges | (3d) 6 Nodes 11 Edges | (4d) 7 Nodes 15 Edges

Figure 3.3 *The four shapes in the top row are represented with maximal graphs in the second row, with direct graphs in the third row, and with over-complete graphs in the bottom row.*

The third approach is presented by Keles et al. (2010). Compared with the direct graph, this *over-complete graph* has additional edges, such that all pairs of intersections on a maximal element are connected by an edge. (See graphs (1d) to (4d) in Figure 3.3.) In this way, multiple decompositions are preserved, though at the expense of a larger number of edges and with the resulting need to keep track of

which set of edges represents a maximal element. In Figure 3.3, all edges of a maximal element are identically labeled; however, it is unclear where global information about a maximal element, such as its length or orientation, would be stored in such a case. Specifically, $n(n+1)/2$ graph edges are needed for each element with $n-1$ intersections, assuming that the intersections are in different locations. For n elements with a maximum of n^2 intersections, the over-complete graph requires $O(n^2)$ nodes to represent the intersections and $O(n^3)$ edges to represent the n maximal elements. If all elements intersect each other in the same location, only $\Omega(1)$ nodes and $\Omega(n)$ edges are needed. However, despite this cubical increase in complexity and corresponding loss in compactness and transparency, the problem of decomposition remains fundamentally unresolved. Strictly speaking, a complete representation of a maximal element with an over-complete graph would require an indefinite number of edges, since the part relation allows indefinite decompositions.

As a fourth option, one can reverse the previous model and represent elements as nodes and edges as intersections (see (1e) to (4e) in Figure 3.4). This model is adopted in this thesis. At first glance, the *inverted graph* may seem counterintuitive, but it elegantly accounts for the embedding and part relations by representing each maximal element as undivided in one node. One can see this model as an evolution of the maximal graph. Maximal elements, represented by Stiny (1975) as one edge with two nodes, are condensed into one node. An intersection between two elements can now be represented as an edge, which completes the representation and avoids the unconnected parts of the maximal graph. One might reject this model for not accounting for intersections with more than two lines intersecting in the same point (named *coincident intersections*), since single edges represent the connections between only two lines (named *distinct intersections*). However, coincident intersections can be handled with relative ease by labeling the edges representing them. An intersection of three elements will then be represented by three identically labeled edges, one for each pair of elements. This is visible in graph (3e) in Figure 3.4, where the three edges representing a coincident intersection are labeled with "4". As explained in the next chapter, there are different types of intersections between two elements, which, in this case, can be elegantly represented by a labeled edge, regardless whether an intersection is coincident. For this approach, a shape composed of n elements with a maximum of n^2 intersections will require exactly $\Theta(n)$ nodes to represent the elements and at most $O(n^2)$ edges to represent the intersections, regardless of whether they are coincident.

The last possibility is presented by Grasl and Economou (2011) and might be called the *elaborate graph*. Here one represents *both* elements and intersections as nodes, as in graphs (1f) to (4f) in Figure 3.4. This model accounts for the embedding and part relations and coincident intersections, but only with a loss of compactness and transparency. This graph also makes it more complicated to represent the properties of

intersections, such as angles or types, since different data types are needed for intersections with different numbers of elements. Again assuming that a shape consists of n elements with a maximum of n^2 intersections, the elaborate graph requires $O(n^2)$ nodes for the elements and their intersections, and $O(n^2)$ edges to connect them. If all elements intersect in the same location, only $\Omega(n)$ nodes and $\Omega(n)$ edges are required.

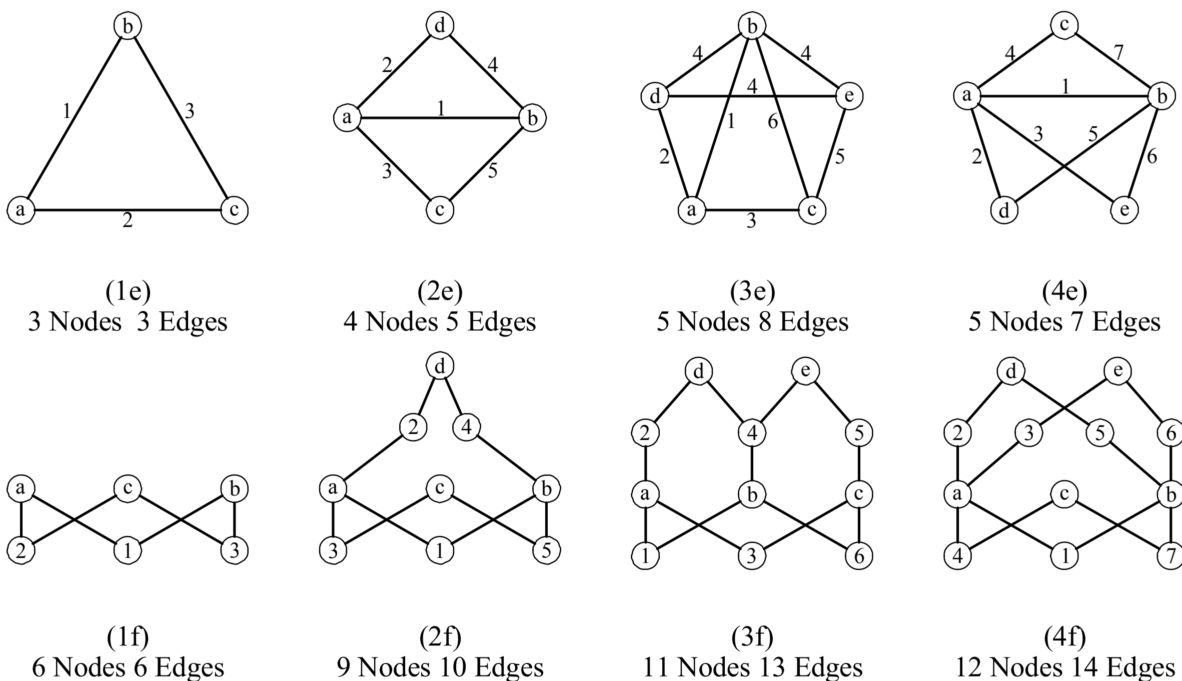


Figure 3.4 *The four shapes from Figure 3.5 are represented with inverse graphs in the first row and with elaborate graphs in the bottom row.*

In selecting a graph for computer implementation, one can immediately rule out the maximal and direct graphs, since the first is unconnected and does not represent intersections and the second splits maximal elements in potentially inappropriate decompositions. The over-complete graph does not support indefinite decompositions in a straightforward manner, but will be compared with the two other approaches for the sake of the discussion. Examining the compactness of the over-complete, inverted, and elaborate graphs, we find that for both the over-complete and the elaborate graphs, the size of the graph depends not only on the number of elements, but also on the number of distinct intersections. When many intersections are coincident, the elaborate graph is the most compact. However, one can expect most shapes to have mostly distinct intersections and only a small number of coincident ones. In that more general case, the inverted graph is the most parsimonious, while also preserving the embedding and part

relations[8]. Compared to the inverted graph, the over-complete graph is unclear since a maximal element can correspond to any number of distinct edges, as demonstrated with graph (4d) in Figure 3.3. While clearly representing the embedding and part relations and multiple intersections, the elaborate graph, on the other hand, is potentially confusing in its employment of nodes for both elements and intersections. Usually, graph edges represent a connection between two entities, as is the case for the over-complete and the inverted graphs. In the elaborate graph, a connection between two elements is represented by two edges and one node, which can be a source of unclarity. This also points to another stylistic difference between the over-complete, inverted, and elaborate graphs, namely how intersections are handled. The inverted graph understands intersections as connections between two maximal elements, while the over-complete and elaborate graphs sees intersection as objects of their own. In that sense, the inverted graph is closest to how visual calculating formally defines shapes: There, as mentioned in section 3.2, intersections are seen as implicit consequences of how maximal elements are arranged, rather than as entities of their own. The next chapter outlines the inverted graph in detail.

[8] When implemented on a computer, the inverted and elaborate graphs might well turn out to be of similar size and complexity, since the edges of the elaborate graph do not carry information, but the edges of the inverted graph do.

4 The Shape Graph

This chapter first describes a method for constructing a shape graph for shapes in the algebra U_{12} (planar line segments), based on the previously outlined inverted approach. The shape graph is an *oriented multigraph*. To generate this representation from a non-maximal shape, one organizes the shape's line segments into infinite elements (in U_{12}, infinite lines), calculates the intersections of those elements, and then constructs the shape graph by mapping infinite elements to nodes and their intersections to edges. The second section of this chapter examines the different intersection types of infinite elements in U_{12}. In the third section, these types are developed into heuristics for representing non-maximally connected shapes. An algorithm that employs these heuristics to construct shape graphs for both maximally and non-maximally connected shapes is presented in the last section.

4.1 Constructing the Shape Graph

We assume that, initially, a shape is represented as a set of non-maximal elements, such as the line segments a designer might draw in a conventional CAD program. From these elements, we generate a set of *infinite elements*, with each containing at least one maximal element. In U_{12}, infinite elements are (infinite) lines that carry at least one (finite) maximal line segment; in U_{23} they are planes carrying at least one maximal planar polygon. (A similar approach for representing line segments is employed by both Krishnamurti (1980) and Chase (1989).) As explained below, each infinite element also carries a sequence of *half-intersections* that characterize an intersection point from the perspective of the infinite element. The shape graph is created by mapping infinite elements to nodes and half-intersections to edges. The proposed procedure can also be extended to other algebras, for example, by representing planes as nodes and their intersections as edges.

ALGORITHM A *(Infinite Element Creation)*

A1 Construct the set of infinite elements

We receive a non-maximal shape in U_{12} as a set of n non-maximal line segments:

$S = \{s_1, s_2, ..., s_n\}$.

A non-maximal element s_i is defined by its boundaries, that is, its start point and end point. Let the start point p_{si} be the left-most point of the element, and the end point p_{ei} be the right-most. Vertical elements extend from bottom to top:

$s_i = (p_{si}, p_{ei})$.

From S, we first construct a pair of ordered sequences (i.e., ordered sets). The first sequence contains m lines, sorted according to their slopes and containing the n line segments:

$$L = \{L_1, L_2, ..., L_m\}.$$

The second sequence contains m sequences of half-intersections, corresponding to the lines in L:

$$I = \{I_1, I_2, ..., I_m\}.$$

A line L_i is defined as a sequence of boundaries containing the start points and end points of all segments in S collinear with L_i. Each line contains at least one start point and one end point. The boundaries are sorted from left to right and from bottom to top, with coincident boundaries sorted such that start points are first and endpoints are second, thus defining the line's orientation and geometry[9]:

$$L_i = \{p_{s1}, p_{e1}, p_{s2}, p_{e2}, ..., p_{sn}, p_{en}\}.$$

If a line segment (p_{s1},p_{e1}) is contained in another segment (p_{s2}, p_{e2}), the sequence would be ordered as $\{p_{s1}, p_{s2}, p_{e2}, p_{e1}\}$. If the two segments overlap, the sequence is $\{p_{s1}, p_{s2}, p_{e1}, p_{e2}\}$. To construct the infinite elements, we sort the n segments according to their slopes and partition them into m sets of collinear segments, in a total running time of $O(n^2)$. The $2n$ boundaries can be sorted in $O(n^2)$ time as well.

A2 Apply the reduction rules

We apply the reduction rules by iterating over the boundaries in each sequence L_i while maintaining a counter i (starting with $i = 0$ for each sequence). If the current boundary b is a start point, we increase i by one. Subsequently, if i is larger than one, we delete b:

$$b = p_s \rightarrow i = i + 1$$
$$(b = p_s) \wedge (i > 1) \rightarrow b = \varnothing.$$

Else, if the current boundary is an end point, we decrease i by one. Subsequently, if i is larger than zero, we delete b:

$$p = p_e \rightarrow i = i - 1$$
$$(p = p_e) \wedge (i > 0) \rightarrow b = \varnothing.$$

This step can be achieved in $O(n)$ time, since we iterate only once over each sequence L_i. (See Figure 4.1 for a visual description of this step.)

---

[9] For actual computer implementation, it is convenient to represent lines in parametric form and associated coordinates with a single parameter.

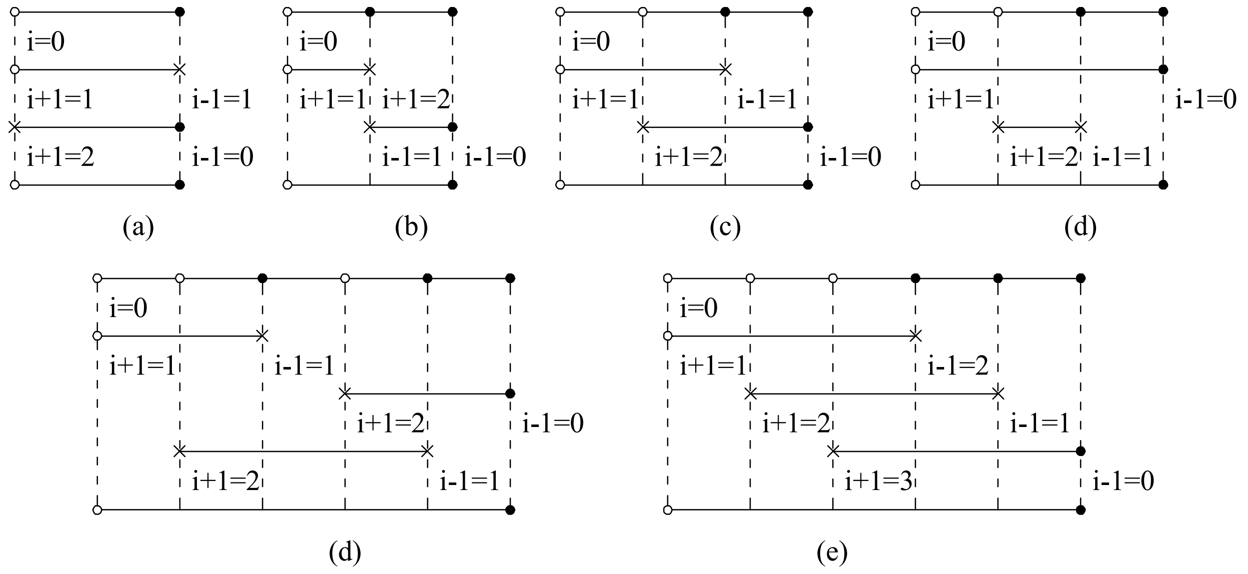


Figure 4.1 *For each diagram, the top row shows a non-maximal element composed of several segments. For each segment, a white dot indicates the start point; a black dot, the end point. Below the top row, each segment is shown in an individual row. At the lower right of each segment boundary, the value and change of the counter is displayed, with the algorithm passing over the boundaries from left to right and from top to bottom. For start points, the counter increases with one; for endpoints, it decreases with one. When, at a start point, the counter is larger than one, the start point is deleted. When, at an end point, the counter is larger than zero, the end point is deleted. A deleted boundary is drawn with an* x. *In the bottom row of each drawing, the maximal element obtained by the algorithm is displayed. (a) shows two coincident segments; (b), two adjoining segments; (c), two overlapping segments; and (d), a segment that is contained in another segment. (d) and (e) are examples of three overlapping segments.*

A similar approach is taken by Krishnamurti (1980). Chase employs a less efficient, recursive, technique (1989). After we have iterated over the boundaries in L_i in the described manner, L_i is maximal, i.e., it contains neither adjacent start points nor adjacent end points. An adjacent start point and end point define a *maximal* element; an adjacent end point and start point, an *empty* element:

$$\{..., p_{s3}, p_{e4}, ...\} \rightarrow \{p_{s3}, p_{e4}\} = maximal$$
$$\{..., p_{e4}, p_{s5}, ...\} \rightarrow \{p_{e4}, p_{s5}\} = empty.$$

A3 Calculate half-intersections

For each line L_i, we construct the sequence I_i by calculating all half-intersections generated by that line and by sorting the half-intersections according their position relative to the line's orientation:

$$I_i = \{i_1, i_2, ..., i_n\}.$$

Let a *half-intersection* be defined as a four-tuple, containing references to the intersection point and the intersected line, as well as to the two elements on the intersected line that are located to the left and right of the intersecting line:

$$i_k = (p_k, L_j, l_j, r_j)$$
with
$$l_j = (p_1, p_k),\ r_j = (p_k, p_2),$$
and
$$p_k, p_1, p_2 \in L_j.$$

If the intersected line holds an element that extends across the intersection point, the two elements are identical:

$$l_j = r_j = (p_1, p_2) \text{ with}$$
$$p_1, p_2 \in L_j.$$

If two lines do not intersect, they are parallel. We denote this non-intersection as a half-intersection, with the corresponding empty element spanning from the last boundary on each line to infinity[10]:

$$l_j = r_j = (p_n, p_\infty) \text{ with}$$
$$p_n \in L_j.$$

For this step, we calculate a total of $2m^2$ half-intersections, which can be sorted in $O(n^4)$ time. $O(n^4)$ also is the total running time for algorithm **A**. Having prepared the infinite elements and their intersections, we are ready to construct the shape graph.

ALGORITHM B *(Shape Graph Construction)*

B1 Construct the shape graph

Given a shape S represented as a sequence of infinite elements L and a sequence of sequences of half-intersections I, we interpret the infinite elements as a set of nodes, and the sequences of half-intersections as sets of labeled half-edges (see Figure 4.2 for an illustration):

$$M : S \rightarrow G(L, I).$$

[10] Note that parallel lines “intersect” only once, namely in the direction towards the “end” of the lines. When searching for a shape, a candidate line therefore has to be examined twice: Once with its direction unchanged, and once with its direction reversed.

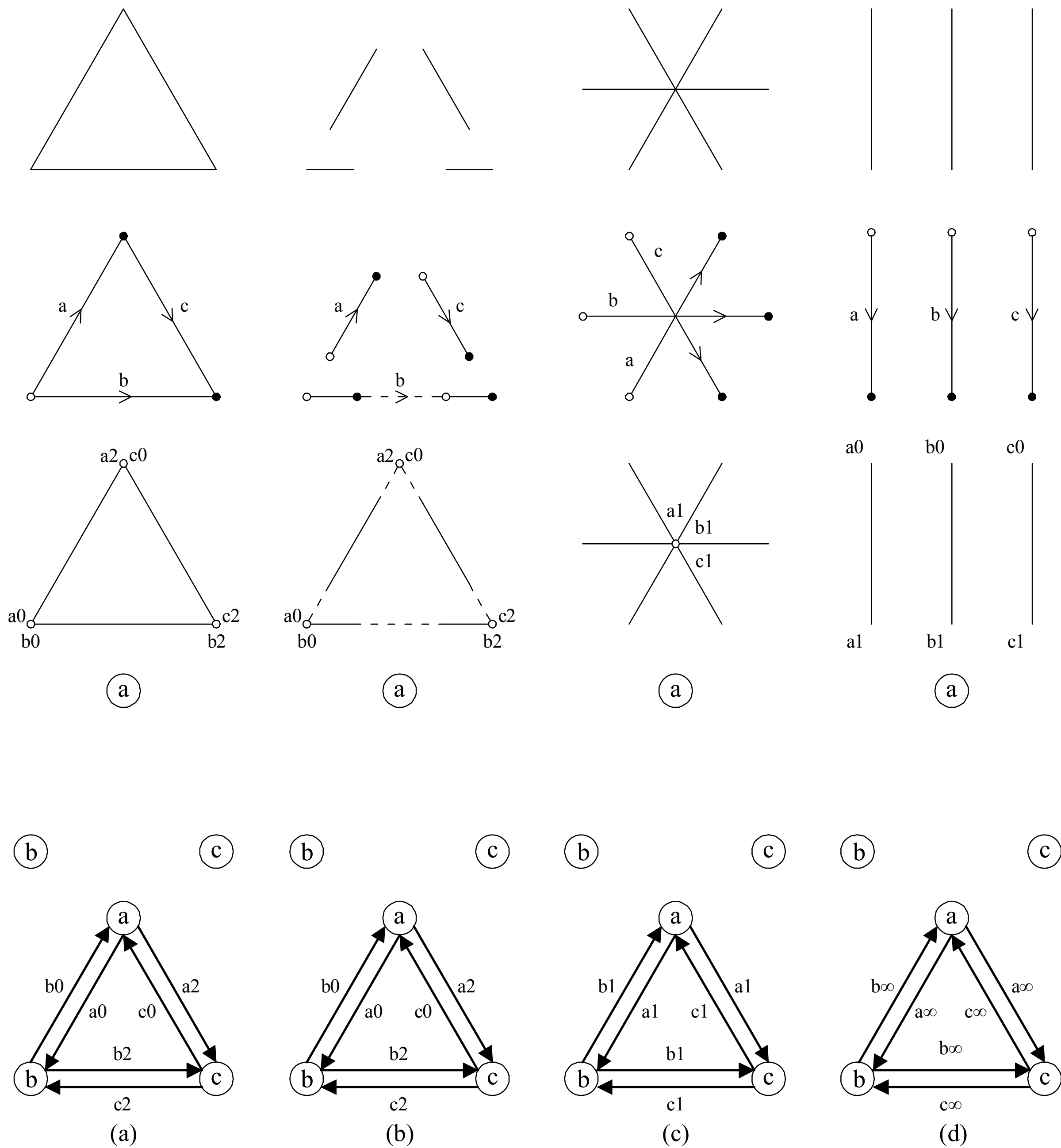


Figure 4.2 *The four shapes in the first row are shown with oriented infinite elements in the second row. In the fourth row, these infinite elements are interpreted as nodes. The (half-)intersections of the infinite elements are shown in the third row, leading to the completed shape graphs in the fifth row. (*a0*) indicates a half-intersection located on element* a *at position* 0*, etc.) Each half-edge represents one line intersecting another line, which is why two corresponding half-edges represent one intersection point, i.e., two lines intersecting each other. Note how labeled half-edges represent a coincident intersection of three lines in column (c) and parallel non-intersections in column (d). Also note that, although the shape in column (b) is non-maximally connected, the corresponding shape graph is topologically identical to (a). (Non-maximally connected shapes like (b) and (d) are treated more extensively in sections 4.3 and 4.4.) The difference between shapes (a) and (b) is characterized by the intersection type associated with the edges (see section 4.2), and the maximal elements of the infinite elements associated with the individual nodes.*

In this way, a half-intersection i_k is represented as an oriented half-edge from the node of the intersecting line to the node of the intersected line, labeled according to its position p_k on the intersecting line. We refer to these connections as *half-edges*, since every connection between two nodes actually consists of a complementary pair of (half) edges, and to the (half) edge pair as an edge[11]. An intersection between two lines is thus represented as a pair of labeled half-edges oriented in opposite directions, since, from the perspective of the first line, the first line is intersecting the second line, and vice versa. The two lines are represented by the two nodes joined by the two half-edges:

$$i_k \in I_i = (p_k, L_j, l_j, r_j)$$
$$i_l \in I_j = (p_l, L_i, l_i, r_i)$$
$$\varphi(i_k) = (L_i, L_j)$$
$$\varphi(i_l) = (L_j, L_i).$$

Since two nodes are connected by two oriented edges, the shape graph is an *oriented multigraph*. The running time for algorithm **B** depends on the number of $m \leq n$ nodes and $2m^2 \leq 2n^2$ half-edges and therefore is $O(n^2)$. Hence, the total running time for algorithms **A** (Infinite Element Creation) and **B** (Shape Graph Construction) is $O(n^4)$. In this section, we have assumed that all intersections between infinite elements are included in the shape graph. An algorithm that considers different intersection types is presented in section 4.4.

4.2 Types of Intersections

Before we discuss how shape graphs can represent non-maximally connected shapes, this section examines different types of intersections. Together, the left and the right elements of a half-intersection identify four distinct types of intersections, as shown in Figure 4.3.

These types generate 16 combinations, since both half-intersections have one of the four types associated with them. In the shape graph, the same is true for the two half-edges that together represent an intersection. Intersections are also termed *registration marks* in the literature (Stiny, 2006). Taking this concept further, registration marks can be grouped into three types: *full* intersections, where both lines contain a maximal element at the intersection point; *partial* intersections, where one of the two lines contains a maximal element at the intersection point; and *empty* intersections, where both lines hold an empty element at the intersection point. An additional type is the *infinite* non-intersection of two parallel

[11] For computer implementation, it is convenient to include a pointer to the complementary half-edge with each half-edge, similar to doubly connected edge lists.

lines, which is also an empty intersection. Infinite intersections are used to represent special cases like the three parallel lines of shape (d) in Figure 4.2.

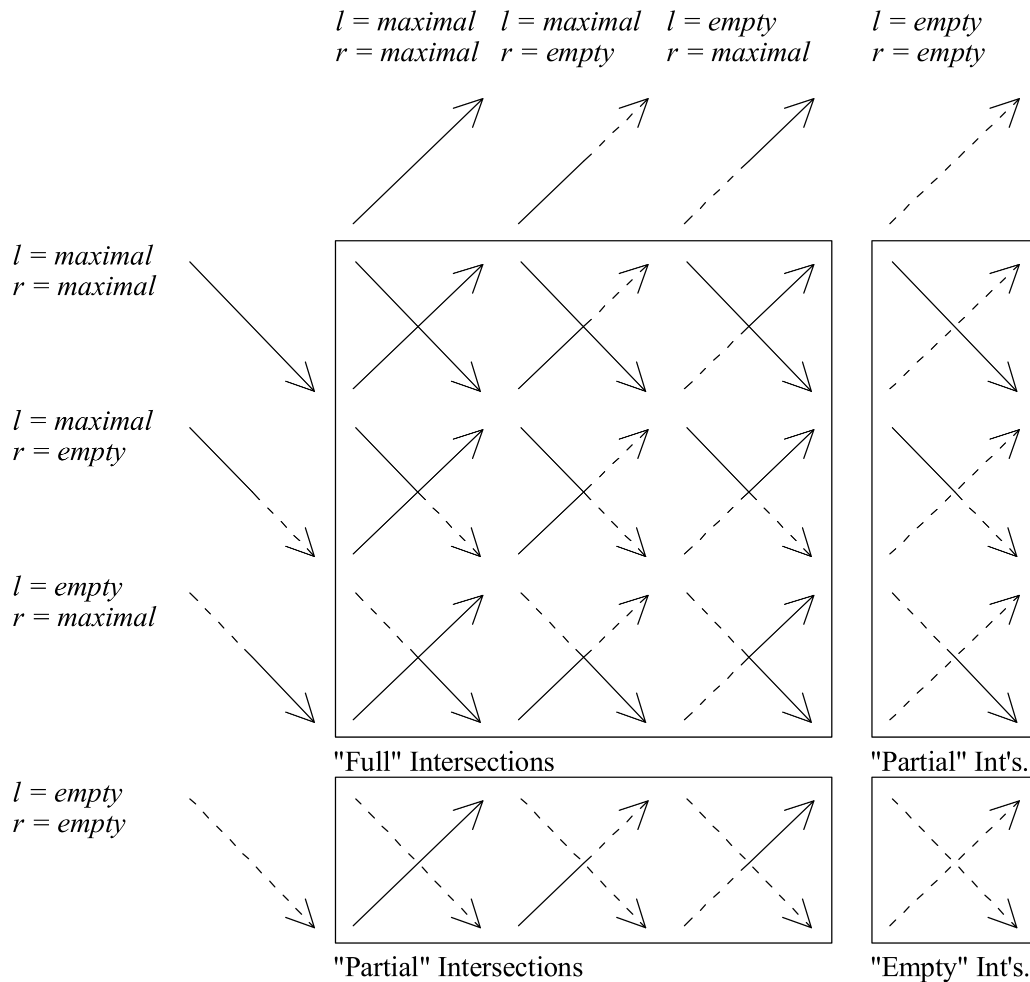


Figure 4.3 *The four types of half-intersections generate 16 intersection types, which are grouped into full, partial, and empty intersections.*

Let a line segment bounded by two consecutive intersection points be termed a *bounded segment*. Let bounded segments that do not contain (part of) a maximal element be called *empty*, and segments bounded on only one side be called *open*. (Open segments are also known as half-infinite lines or rays.) The next section explores how different types of intersections and bounded and open segments aid with representing non-maximally connected shapes.

4.3 Heuristics for Representing Non-maximally Connected Shapes

In graph theory, a *path* is defined as a sequence of edges connecting a sequence of nodes. (That is, the first edge connects the first and the second node; the second edge, the second and third node; and so on.)

Then a graph is *connected* if there is a path between every pair of nodes. Otherwise, the graph is *unconnected*. Similarly, for shapes, let a *path* be a sequence of infinite elements, connected by a sequence of intersections. (In other words, the first intersection occurs between the first and the second element; the second intersection, between the second and third element; and so on.) In U_{12}, where maximal elements are line segments and infinite elements are lines, let a path consisting of only full intersections, with maximal elements between consecutive intersections, be a *maximal* path. Then a shape is *maximally connected* if there is a maximal path between every pair of infinite elements, and *non-maximally connected* if there is a non-maximal path (i.e., a path including partial and/or empty intersections, or consecutive intersections with empty elements between them) between at least one pair of infinite elements. For example, shapes (c), (f), and (g) in Figure 4.4 are non-maximally connected. As is explained below, non-maximally connected shapes can result in unconnected graphs, depending on which intersection points are included in the representation.

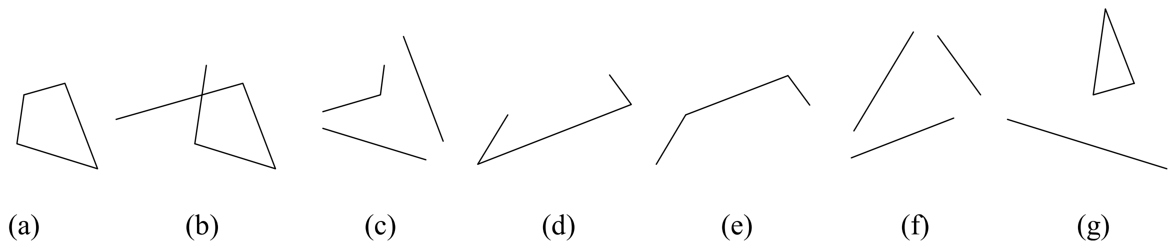


Figure 4.4 *Seven shapes. Shapes (c), (f) and (g) are non-maximally connected.*

Representing non-maximally connected shapes with graphs is less straightforward than one might think. Previous graph-based representations have included only maximally connected shapes, while Krishnamurti (1981) has demonstrated a method for representing non-maximally connected shapes when t is non-parametric. His method, however, does not work for graphs, especially when t is not a similarity transformation (i.e., t is parametric). In constructing graphs for shapes that are non-maximally connected and with less constrained transformations t, the key question is which intersection points should be included. At a minimum, it is necessary to include enough intersections so that the shape is connected. Otherwise, the resulting graph is unconnected and therefore does not represent the shape as a whole. Selecting the exactly right set of intersection points, however, is a problem of ambiguity, since it depends on the perceptions of the individual designer. As we will see below, one can include different sets of points for each shape in Figure 4.4, with each set defining a different shape graph. This ambiguity has important implications for visual calculating. If too many points are included, the shape graph is over-defined and might not allow embedding into the shapes expected by a designer. If, on the other hand, the point set is too small, a shape rule applies in much wider range of cases than anticipated. (Note that this

only applies to shapes that one is searching for, as opposed to shapes that one is searching in. In a shape where one is searching in, all intersection should be included.)

Gestalt psychology offers guidelines as to which representations might be most appropriate, for example, the "law of continuation and good closure" (Wertheimer, 2012, pp. 149-160), which states that human perception tends to complete ambiguous figures into more definite shapes. However, this kind of guideline is by no means definitive or applicable in all case. While, in Figure 4.4, shape (c) should probably be represented as an incomplete chevron and shape (f) as an incomplete triangle, shape (b) eschews easy categorization. It can be perceived as a quadrilateral with extensions, a triangle that is both incomplete and subdivided, or an incomplete chevron. Shape (g) reads like two separate shapes, but could be an incomplete chevron. In a computer implementation, an algorithmic representation of maximally non-connected shapes therefore cannot aspire to an "optimal" solution. Instead, the implementation should provide an algorithm that is applicable for a broad range of cases, while leaving flexibility for designers to come up with customized representations. Below, the author of this thesis has applied his personal judgment to determine which representations are most appropriate. However, and more importantly, the proposed framework represents both maximally and non-maximally connected shapes and is general enough to be adapted to individual perspectives (e.g., through additional or refined heuristics). The framework consists of four simple heuristics, employing the intersection types discussed in the previous section, and an algorithm that combines three of the four heuristics in a hierarchical fashion. The heuristics are discussed below, while the algorithm is presented in the next section.

HEURISTIC 1 *Only accept full intersections*

This simplest of heuristics can result in unconnected graphs for non-maximally connected shapes such as shapes (c), (f) and (g) in Figure 4.5. Since parallel non-intersections are empty intersections, they are not included. This heuristic is applied by Grasl and Economou (2011).

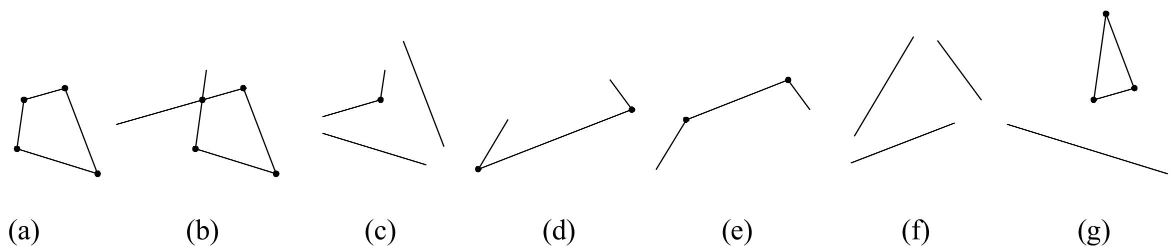


Figure 4.5 *The dots indicate intersections included by Heuristic 1 (Only accept full intersections) for the shapes from Figure 4.4.*

HEURISTIC 2 *Include all intersections*

This heuristic is the default for the shape graph construction algorithm presented in the first section of this chapter. Though representing non-maximally connected shapes in a straightforward manner, this heuristic also introduces potentially unwanted distinctions and a loss of symmetry. With the exception of triangle (f), the shapes in Figure 4.6 appear over-defined, i.e., they include more intersection points than necessary. However, this heuristic is useful when constructing shape graphs that will be scanned for subshapes, since, in that case, including all intersection points preserves a maximum of possibilities.

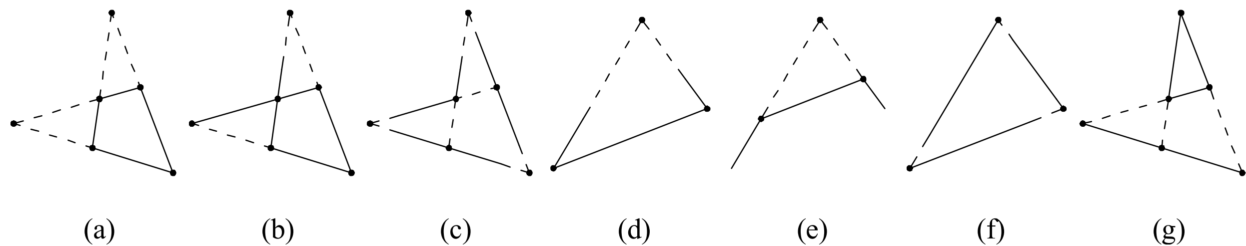


Figure 4.6 *Here, the six shapes from Figure 4.4 are drawn with line segments extended to their intersection points. The dots indicate intersections included under Heuristic 2 (Include all intersections).*

For example, if we do not constrain angles and proportions, quadrilateral (a) in Figure 4.5 possesses eight-fold symmetry. It can then be rotated in four different ways and mirrored in two, and still be embedded into itself. However, once we expand the definition of the shape to include the two empty intersections formed by the extended sides of the quadrilateral, as for (a) in Figure 4.6, we can rotate the shape just in two ways, so that it now possesses only four-fold symmetry. Perhaps surprisingly, the same would be true for a square with parallel sides (if parallel non-intersections are not specifically excluded).

HEURISTIC 3 *Only include intersections with non-empty bounded or open segments*

This heuristic focuses on bounded segments, defined by two consecutive intersection points, and on open segments, defined by the last or first intersection point of an infinite element, instead of on individual intersections. We only include an intersection point when it is a boundary of at least two non-collinear, non-empty bounded or open segments. As shown in Figure 4.7, this heuristic addresses the problems of potential loss of symmetry (see shape (a)) and non-maximally connected shapes (see shapes (c) and (f)). However, shape (g) would still result in an unconnected graph. We also can represent a shape composed of only parallel line segments, such as shape (d) in Figure 4.2, and related special cases.

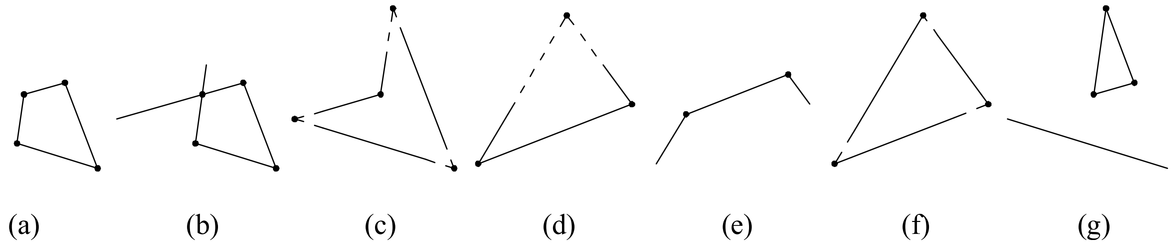


Figure 4.7 *The six shapes from Figure 4.4, with intersections included under Heuristic 3 (Only include intersections with non-empty bounded segments).*

However, this heuristic also introduces distinctions between shapes that might strike a designer as artificial. For example, in Figure 4.7, shape (d) is represented as a triangle, but shape (e) is not. In this case, (e) embeds into (d), but not (d) into (e), although one can easily perceive both shapes as potentially isomorphic U-shapes. The same would be true if the sides of (d) and (e) were parallel (again assuming that parallel non-intersections are not specifically excluded).

HEURISTIC 4 *Only include non-empty intersections*

With this last heuristic, we include full intersections as well as partial ones, without regard to bounded segments. As is visible in Figure 4.8, this heuristic is problematic. For example, shape (c) is not represented as a chevron, and the graph of shape (f) would be unconnected. However, this heuristic recognizes the subdivided triangle in (b), which, in special cases, might be desirable. It also results in a connected graph for (g). Since parallel non-intersections are empty intersections, they are not included.

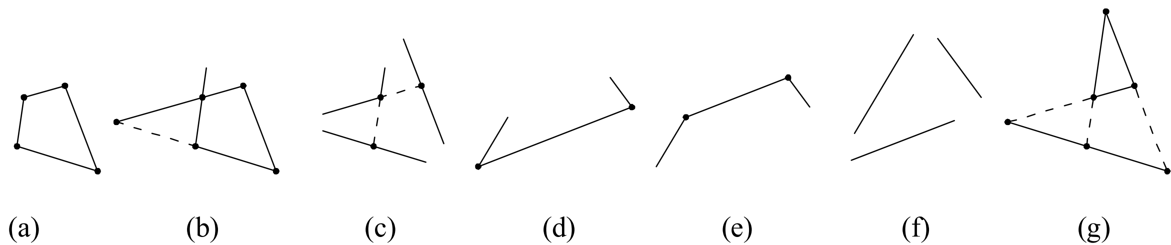


Figure 4.8 *The six shapes from Figure 4.4, with intersections included under Heuristic 4 (Only include non-empty intersections).*

Each of the previously discussed heuristics has, in some cases, proved unsatisfactory. However, one can achieve convincing results for a large number of cases by combining heuristics in a hierarchal fashion, which is the idea behind the algorithm in the following section.

4.4 Constructing Shape Graphs for Non-Maximally Connected Shapes

The algorithm described below employs, in a hierarchical series of steps, three of the four heuristics developed in the previous section to construct shape graphs for both maximally and non-maximally connected shapes. (As we have seen, Heuristic 4 (Only include non-empty intersections) is problematic and should be reserved for special cases.) Note that this algorithm should be applied only to shapes that one is searching for as subshapes. Shapes that will be scanned for subshapes should always be constructed with all intersections points to preserve a maximum of possibilities.

ALGORITHM C *(Shape Graph Construction with Heuristics)*

C1 Partitioning the graph edges

We are given a set of nodes L representing infinite elements and a set of edges I representing their intersections (including parallel non-intersections), with each edge composed of two half-edges representing half-intersections. We partition I into three non-intersecting subsets. The first subset contains all edges representing full intersections. Of the remaining edges, we include those representing a boundary of at least two non-collinear, non-empty (bounded or open) segments in the second set. The remaining edges are included in the third set.

$I = (A, B, C).$

We test if an edge represents a full intersection by examining the intersection type defined by its two half-edges. This can be done in linear time. To check if an intersection point is bounding at least two non-collinear, non-empty segments, we test for both half-intersections if there is a least one (part of) a maximal element between the previous and the next half-intersection on the intersecting line. In other words, we test for both half-intersections if at least one of the two bounded (or open) segments to the left and right of the half-intersection is non-empty. Since both the maximal elements and the bounded segments of a line are represented as ordered sets, we can perform this test with two binary searches in $O(\log n)$ time. Hence, the $O(n^2)$ edges can be partitioned into the three subsets in $O(n^2 \log n)$ time.

C2 Construct a shape graph with full intersections

We construct a shape graph from the set of nodes representing infinite elements L and the set of edges representing full intersections A (see Figure 4.9):

$G_A = (L, A).$

This step corresponds to heuristic one from the previous section. If the shape graph is connected, the representation is completed. Otherwise, we continue to the next step. Constructing the shape graph takes $O(n^2)$ time (see algorithm **B** in Section 4.1). To test if a shape graph is connected, we traverse the graph, with a random node as the starting point. (In other words, we explore the extent of the graph by traversing nodes and edges, for example, by using depth-first search or breadth-first search). If, at the end of the traversal, the number of traversed nodes is equal to the number of nodes in L, the shape graph is connected. The running time for the traversal linearly depends on the number of edges and therefore is $O(n^2)$.

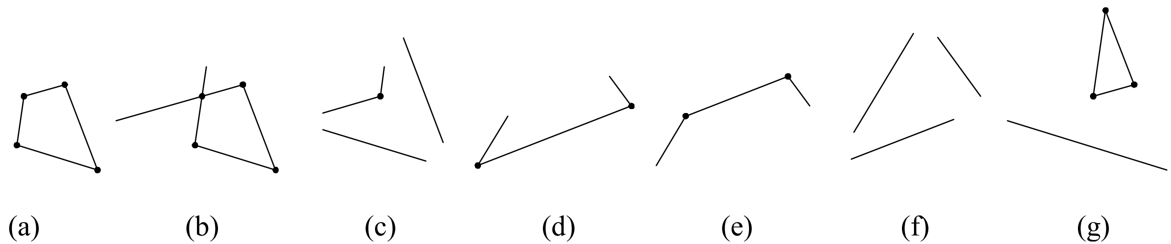


Figure 4.9 *The six shapes from Figure 4.4, after step C2. The black dots indicate intersections added during this step.*

C3 Construct a shape graph with intersections with non-empty segments

We include the set of intersections with non-empty segments B in G_A to construct a new shape graph (see Figure 4.10):

$G_B = (L, A \cup B).$

This step corresponds to heuristic three from the previous section. If the shape graph is connected, the representation is completed, otherwise we continue to the next step.

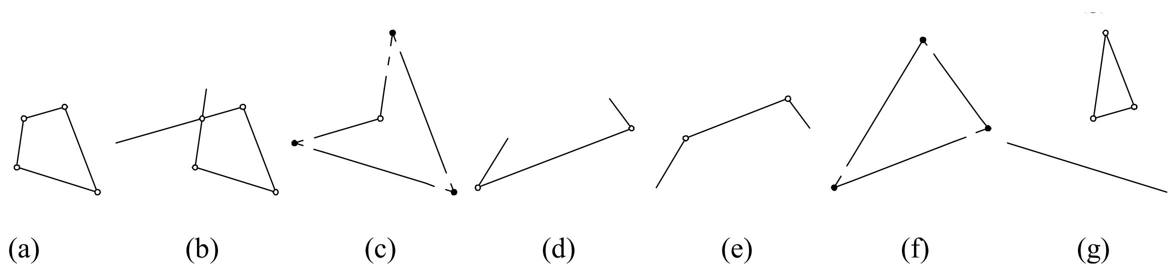


Figure 4.10 *The six shapes from Figure 4.9, after step C3. The black dots indicate intersections added during this step, the white dots intersections that were added previously. Note that shape (d) did not receive a third point, since it was already completed in step C2.*

C4 Construct a shape graph with all intersections

We include the set *C,* consisting of the remaining partial and empty intersections, with G_C to construct a new shape graph (see Figure 4.11):

$$G_D = (L, A \cup B \cup C) = (L, I).$$

This step corresponds to heuristic two from the previous section. Since this graph includes all intersections (including parallel non-intersections), it is guaranteed to be connected.

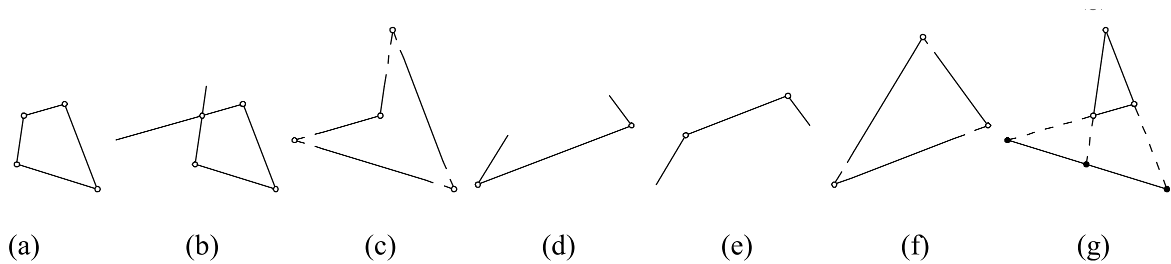


Figure 4.11 *The six shapes from Figure 4.10, after step C4. Note that intersections were only added for (g), since all other shapes were connected previously.*

Since a shape graph is constructed (or expanded) in steps **C2**, **C3**, and **C4,** their running time is identical, namely $O(n^2)$. Therefore, the total running time for algorithm **C** is $O(n^2 \log n)$. To construct a shape graph for both maximally and non-maximally connected shapes, we combine algorithms **A** and **C**, resulting in a total running time of $O(n^4)$.

As demonstrated by Figure 4.11, the above algorithm appropriately represents a variety of shapes. Though even better heuristics certainly are a worthwhile topic for further research, ultimately the designer should decide which approach is viable for a given design. In their implementation, Keles et al. (2010) rely on choices by the designer to create adequate representations. The purpose of algorithm **C** is to provide a good baseline, which, if necessary, can be refined by input from the designer (e.g., by explicitly selecting the points to be included in the symbolic representation of a shape). The next chapter summarizes this thesis, sketches the potential for a graph-based design tool based on visual parametric calculating, and points out avenues for further research.

5 Conclusion

In this concluding chapter, the first section will summarize the thesis. Subsequently, a creativity-enriching, rule-based design tool based on graphs is outlined as an objective to which this thesis contributes. Based on this objective, the last section outlines an agenda for further research.

5.1 Summary of the Thesis

The thesis is motivated by two conflicting notions of design: A formal one that underlies explicit design methodologies, and an artistic one that views design freedom as primary. This dialectic can be reconciled with *visual calculating*, i.e., calculating with shapes, which provides a formal, rule-based notation for exploratory visual design, without the limitations that usually accompany explicit design methods. (For example, to apply explicit methods effectively, designers need to know in advance what they want to achieve.) Crucially, although sets of visual rules can be codified as shape grammars (Stiny & Gips, 1972), while designing, designers can apply any rules they want in any way they want (Stiny, 2011). Visual calculating supports this freedom by allowing a definite decomposition of a design only after the design process has taken place. This property results from the e*mbedding relation*, which guarantees that a shape is always composed of *maximal elements* (Stiny, 2006). In other words, a shape is represented by the smallest set of the largest possible elements. Complementing the embedding relation, the *part relation* ensures that any part of a maximal element can be picked out for rule application. In other words, a shape can be decomposed in indefinitely many ways. Maximal elements and indefinite decompositions thus guarantee design freedom. Maximal elements are an important contrast between visual calculating and other design methods, which, as a first step, tend to decompose a design problem. (This prior decomposition can make it hard for designers to change their minds.) Calculating with maximal elements also contrasts with digital computation, which works with the largest set of the smallest possible elements: zeros and ones.

The dialectic between explicit methodologies and design freedom can also be observed in the field of computerized tools for architectural design. Generative design techniques like scripting or parametric design suffer from the same limitations as other explicit methods, which is why a computer implementation of visual calculating promises an opportunity for more designerly design tools. Research about the computer implementation of visual calculating has been ongoing for the past thirty years, but implementations have remained limited in scope and popularity (Gips, 1999). Relatively recently, computer implementations with three-dimensional (Stouffs & Krishnamurti, 2006) and curved (Jowers & Earl, 2011) geometries have been developed. However, the implementation of *parametric* visual calculating has received comparatively little attention. (Parametric visual calculating allows the

arguments of a rule to vary geometrically. For example, a parametric rule can apply to "all quadrilaterals", as opposed to "all squares". In other words, rule application is governed by different transformations, some of which are parametric.) In fact, considerable effort has been expended to show that parametric visual calculating is difficult, if not impossible, to implement on a computer ((Yue, Krishnamurti, & Gobler, 2009) and (Yue, 2009, pp. 78-90)).

A key question for the implementation of parametric visual calculating is how shapes should be symbolically represented. (Shapes are visual entities, but computers work only with symbols.) Though there are several approaches to choose from, most of them do not work with parametric shapes, including geometry-based approaches (Krishnamurti, 1981) and image recognition (Keles, Özkar, & Tari, 2012). Other approaches apply restrictions, such as pre-defined objects (Nagakura, 1995) or hierarchies (McCormack & Cagan, 2002) that undermine the design freedom that makes visual calculating uniquely promising for computerized design tools. The remaining option, which is explored in detail in this thesis, is to represent shapes as graphs. (A graph is a data-structure composed of nodes and edges, with the edges representing connections between the nodes.) Graphs are suitable for representing parametric shapes because they are based on topological connectivity, and not on fixed geometric features. Rather than loosening a geometric representation to make it parametric, graphs start out as flexible data-structures but can be constrained tighter when necessary. Graphs have been employed in several computer implementations (for example, (Yue & Krishnamurti, 2008) and (Keles, Özkar, & Tari, 2010)), but only one (Grasl & Economou, 2011) employs graphs for parametric visual calculating while preserving the design freedom that is manifested by the embedding and part relations. The thesis compares several graph-based representations for rectilinear, two-dimensional shapes and proposes an *inverted graph* that is clearer, more compact, and closer to the original formulation of visual calculating than the *elaborate graph* by Grasl and Economou (ibid.). In the inverted graph, nodes represent *infinite elements* (the collinear extension of maximal elements) and edges their intersections.

Another key problem for the computer implementation of visual calculating is *subshape recognition*: In order to apply a visual rule to a shape, one needs to find, in the shape, instances of the argument of the rule, which is itself a shape. This problem is especially difficult for parametric visual calculating, since, when searching for instances of parametric shapes, a computer often has to examine many possibilities. When employing a graph-based representation, subshapes can be found via graph search. Graph search is a well-known algorithmic technique, but can be difficult to compute[12]. However, graph search has many

[12] General graph search is NP-complete, which means that, in theory, the time for some computations exceeds the age of the universe. However, many NP-complete problems are tractable in practice (Dasgupta, Papadimitriou, & Vazirani, 2008, pp. 269-293).

applications and continues to be an area of active research (Jungnickel, 2012). Advances in graph search indicate that, in many practical cases, the theoretical difficulty of this technique translates into workable solutions (Batz, 2006).

The thesis describes algorithms that construct the inverted graph for two-dimensional, rectilinear shapes, with special attention given to *non-maximally* connected shapes (i.e., shapes with non-intersecting maximal elements). When interpreted naively as graphs, non-maximally connected shapes result in unconnected graphs. Since unconnected graphs are unsuitable for computation, non-maximally connected shapes need to be interpreted as connected graphs. This need for interpretation potentially introduces unwanted definitions and distinctions, which is a recurring challenge for the symbolic implementation of visual calculating. Non-maximally connected shapes, though addressed early for non-parametric visual calculating (Krishnamurti, 1981), have not received prior attention in the context of parametric visual calculating with graphs. In interpreting non-maximally connected shapes as connected graphs, the key question is which intersection points (also known as *registration marks*), should be included: If one includes too few intersection points, the resulting graph is unconnected; if one includes to many, the graph is over-defined, i.e., more constrained than a designer would expect. To address this question, the thesis proposes four heuristics based on a typology of intersection points, and an algorithm that combines three of those heuristics in a hierarchal fashion. The algorithm achieves good results for a variety of shapes; however, it ultimately should be up to the individual designer to decide how to represent a given shape appropriately. In a design tool, designers should be free to choose heuristics, or to pick an appropriate set of intersection points by hand, in case the proposed algorithm results in an inappropriate symbolic representation. The next section outlines how the flexible application of constraining heuristics can lead to creativity- enriching, rule-based design tools.

5.2 Towards Design Tools with Flexible Constraints for Serendipitous Discovery

The freedom to choose and customize constraints is a big advantage of the proposed, graph-based representation. Since graphs are highly flexible data structures, they can be (un)constrained almost at will. The only constraints inherent in graphs are topological relationships of connectivity, but even those can be released to some degree. One can, for example, apply a rule to “any closed polygon”, by searching for closed cycles in the shape graph. (In graphs, a cycle is a path that returns to the starting node.) By requiring the graph search to preserve different types of geometric information (such as length, angles, proportions, convexity, and/or parallelism), rule application with different transformations can be achieved: By constraining angles, distances and the embedding of maximal elements, shapes can be restricted to *similarity transformations*, while, for *affine transformations*, one would release angles, but

preserve proportions of point distances along lines. As an example of a more general transformation, one could preserve only embedding and convexity, etc. etc. Shape graphs can also be constrained by non-geometric information, such as embedding or labels, which creates even more possibilities. For example, shapes can be defined to be of certain colors or line weights.

One can thus imagine a computerized tool that not only lets designers experiment with shapes, but also with the heuristics and constraints that govern rule application and the creation of symbolic representations. From this perspective, the prime virtue of constraints is not that they "narrow a vast design space" (Tapia, 1996, p. 93), but that they become part of design experimentation. In that way, a computer implementation would not only be a convenient tool that stores and automatically applies visual rules: By offering designers the choice from a range of constraints that are hard to replicate by hand-calculation, the flexible, nimble application of visual rules can lead to surprising results and, in that manner, stimulate creativity in ways that go beyond non-parametric visual calculating. (To be more useful to designers, such a creativity enhancing, rule-based design tool would have to include geometries other than two-dimensional line segments.) I hope that this thesis is a step towards such a rule-based design tool for serendipitous discovery based on all kinds of heuristics, constraints, and geometries. In any case, the thesis articulates a clear agenda for further research in that direction.

5.3 Further Research

As already mentioned in the previous section, one interesting avenue for further research is the extension of the inverted graph to other geometries and dimensions. For example, it is of interest to connect the inverted graph to previous research about the computer implementation of visual calculating with three-dimensional (Stouffs, 1994) or curved (Jowers & Earl, 2011) elements. The inverted graph, with its logic of nodes as infinite elements and their intersections as edges, extends naturally to other geometries; however, the details of such an extension certainly deserve closer attention.

Another interesting topic are additional and possibly more sophisticated heuristics for unconnected shapes, since the heuristics outlined in this thesis do not make any claims to completeness or optimality. A related issue is the application of geometric and other constraints to restrict the graph search for subshapes. These constraints afford various (parametric) transformations that certainly merit exploration in more detail. Most likely, the examples from the previous section form only a small part of the intriguing transformations that are afforded by the inverted graph. In this context, one would also have to explore how to apply rules with non-linear transformations (i.e., transformations that cannot be expressed as a transformation matrix).

An especially interesting challenge is the development of a customized graph search algorithm. As mentioned in the first section, graph search algorithms are a topic of research in computer science. However, a graph search algorithm that is tailor-made for (parametric) visual calculating is potentially much more efficient than general-purpose approaches like the algorithm employed by Grasl and Economou (2011)[13]. This algorithm finds subgraphs based on topological relationships, which, in a second step, have to be constrained to specific geometries. Especially in the case of non-maximally connected shapes, this approach, though relatively easy to implement, is much less efficient than a comparable algorithm that would integrate the desired geometric constraints into the graph search.

The last avenue of research has received almost no attention in this thesis: The question of what kind of user interface would be appropriate for a design tool of the type sketched in the previous section. As we have seen, there are many possibilities for different visual calculations based on various geometries, heuristics, and constraints. Making these complex opportunities available in an intuitive and simple manner is an important design task in its own right. The agenda outlined in this section holds great promise, so let's get to it!

[13] Grasl and Economou employ a library (GrGen.NET), which uses *search plans* to optimize graph search. Before the (NP-complete) graph search is begun, the subgraph is analyzed by another, polynomial, algorithm that finds an efficient searching strategy. Although the search plan does not guarantee tractability, it has been shown to have good results in many cases (Batz, 2006).

6 Bibliography

Agarwal, M., & Cagan, J. (1998). A blend of different tastes: The language of coffeemakers. *Environment and Planning B: Planning and Design, Volume 25*, 205-226.

Alexander, C. (1964). *Notes on the synthesis of form.* Cambridge, MA: Harvard University Press.

Batz, G. V. (2006). *An optimization technique for subgraph matching strategies.* Karlsruhe, Germany: Institut für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe.

Bayazit, N. (2004). Investigating design: A review of forty years of design research. *Design Issues, Volume 20*, 16-29.

Bondy, J. A., & Murty, U. S. (2008). *Graph theory.* New York, NY: Springer.

Burry, M. (2011). *Scripting cultures: architectural design and programming.* Chichester, UK: Wiley.

Chase, S. C. (1989). Shapes and shape grammars: From mathematical model to computer implementation. *Environment and Planning B: Planning and Design, Volume 16*, 215-242.

Chase, S. C. (1997). *Emergence, creativity and computational tractability.* Preprints of workshop, Interactive Systems for Supporting the Emergence of Concepts and Ideas, CHI '97: Atlanta, GA.

Chau, H. H., Chen, X., McKay, A., & de Pennington, A. (2004). Evaluation of a 3D shape grammar implementation. In J. S. Gero (Ed.), *Design and Cognition '04* (pp. 357-376). Dordrecht, Netherlands: Kluwer Academic Publishers.

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2008). *Algorithms.* New York, NY: McGraw-Hill.

Dreyfus, H. L. (1992). *What computers still can't do: A critique of artificial reason.* Cambridge, MA: MIT Press.

Dym, C. L., Agogino, A. M., Eris, O., Fry, D. D., & Leifer, L. J. (2005). Engineering design thinking, teaching, and learning. *Journal of Engineering Education, Volume 94*, 103-120.

Eisner, E. (2002). *The arts and the creation of the mind.* New Haven, CT: Yale University Press.

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

Gänshirt, C. (2007). *Tools for ideas: An introduction to architectural design.* Basel, Switzerland: Birkhäuser.

Gips, J. (1999). Computer implementation of shape grammars. *Invited paper, Workshop on Shape Computation*. Cambridge, MA.

Grasl, T., & Economou, A. (2011). GRAPE: Using graph grammars to implement shape grammars. *2011 Proceedings of the Symposium on Simulation for Architecture and Urban Design*, (pp. 21-28).

Herbert, S. (1996). *The sciences of the artificial* (3rd ed.). Cambridge, MA: MIT Press.

Hoisl, F., & Shea, K. (2009). Exploring the integration of spatial grammars and open-source CAD systems. *Proceedings of ICED 09, the 17th International Conference on Engineering Design, Vol. 6, Design Methods and Tools (pt. 2)* (pp. 427-438). Palo Alto, CA: ICED.

Jowers, I., & Earl, C. (2010). The construction of curved shapes. *Environment and Planning B: Planning and Design, Volume 37*, 42-58.

Jowers, I., & Earl, C. (2011). Implementation of curved shape grammars. *Environment and Planning B: Planning and Design, Volume 38*, 616-635.

Jowers, I., Hogg, D. C., McKay, A., Chau, H. H., & de Pennington, A. (2010). Shape detection with vision: Implementing shape grammars in conceptual design. *Research in Engineering Design, Volume 21*, 235-247.

Jungnickel, D. (2012). *Graphs, networks and algorithms.* New York, NY: Springer.

Keles, H. Y., Özkar, M., & Tari, S. (2010). Embedding shapes without predefined parts. *Environment and Planning B: Planning and Design, Volume 37*, 664–681.

Keles, H. Y., Özkar, M., & Tari, S. (2012). Weighted shapes for embedding perceived wholes. *Environment and Planning B: Planning and Design, Volume 39*, 360-375.

Knight, T. W. (1994). *Transformations in design.* Cambridge, MA: Cambridge University Press.

Kolarevic, B. (2003). *Architecture in the digital age: Design and manufacturing.* New York, NY: Spon Press.

Krishnamurti, R. (1980). The arithmetic of shapes. *Environment and Planning B: Planning and Design, Volume 7*, 463-484.

Krishnamurti, R. (1981). The construction of shapes. *Environment and Planning B: Planning and Design, Volume 8*, 5-40.

Krishnamurti, R. (1992). The arithmetic of maximal planes. *Environment and Planning B: Planning and Design, Volume 19*, 431-464.

Krishnamurti, R., & Stouffs, R. (2004). *The boundary of a shape and its classification.* Pittsburgh, PA: CMU School of Architecture.

Li, A. I., Chau, H. H., Chen, L., & Wang, Y. (2009). A prototype system for developing two- and three-dimensional shape grammars. *CAADRIA 2009: Proceedings of the 14th International Conference on Computer-Aided Architecture Design Research in Asia* (pp. 717–726). Douliou, Taiwan: Department of Digital Media Design, National Yunlin University of Science and Technology.

McCormack, J. P., & Cagan, J. (2002). Supporting designers' hierarchies through parametric shape recognition. *Environment and Planning B: Planning and Design, Volume 29*, 913-931.

McCormack, J. P., & Cagan, J. (2006). Curve-based shape matching: Supporting designers' hierarchies through parametric shape recognition of arbitrary geometry. *Environment and Planning B: Planning and Design, Volume 33*, 523-540.

McGill, M. C. (2002). Shaper2D: Visual software for learning shape grammars. *Connecting the Real and the Virtual - design e-ducation, 20th eCAADe Conference Proceedings* (pp. 148-151). Warsaw, Poland: eCAADe.

Mitchell, W. J. (1990). *The logic of architecture.* Cambridge, MA: MIT Press.

Nagakura, T. (1995). *Form-processing: A system for architectural design.* Cambridge: Harvard University, MA.

Nishizeki, T., & Chiba, N. (2008). *Planar graphs: Theory and algorithms.* Mineola, NY: Dover.

Otto, K. N., & Wood, K. L. (2001). *Product design: Techniques in reverse engineering and new product development.* Upper Saddle River, NJ: Prentice Hall.

Özkar, M. (2011). Visual schemas: pragmatics of design learning in foundations studios. *Nexus Network Journal 13*, 113-130.

Schön, D. A. (1983). *The reflective practitioner: how professionals think in action.* New York, NY: Basic Books.

Stiny, G. (1975). *Pictorial and formal aspects of shape and shape grammars.* Basel, Switzerland: Birkhäuser.

Stiny, G. (1977). Ice-ray: A note on the generation of chinese lattice designs. *Environment and Planning B: Planning and Design, Volume 4*, 89-98.

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

Stiny, G. (1987). Letter to the editor. *Environment and Planning B: Planning and Design, Volume 14*, 225-227.

Stiny, G. (1990). What is a design? *Environment and Planning B: Planning and Design, Volume 17*, 97-103.

Stiny, G. (1994). Shape rules: Closure, continuity, and emergence. *Environment and Planning B: Planning and Design, Volume 21*, S49-S78.

Stiny, G. (2006). *Shape: Talking about seeing and doing.* Cambridge, MA: MIT Press.

Stiny, G. (2011). What rule(s) should I use? *Nexus Network Journal, Volume 13*, 15-47.

Stiny, G., & Gips, J. (1972). Shape grammars and the generative specification of painting and sculpture. *Information Processing 71*, 1460-1465.

Stiny, G., & Mitchell, W. J. (1978). Counting palladian plans. *Environment and Planning B: Planning and Design, Volume 5*, 189-198.

Stiny, G., & Mitchell, W. J. (1978). The palladian grammar. *Environment and Planning B: Planning and Design, Volume 5*, 5-18.

Stouffs, R. (1994). *The algebra of shapes.* Pittsburgh, PA: Carnegie Mellon University, School of Architecture.

Stouffs, R., & Krishnamurti, R. (2006). Algorithms for classifying and constructing the boundary of a shape. *The Journal of Design Research, Volume 5*, 54-95.

Tapia, M. (1996). *From shape To style.* Toronto, Canada: University of Toronto, Graduate Department of Computer Science.

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

Trescak, T., Rodriguez, I., & Esteva, M. (2009). General shape grammar interpreter for intelligent designs generations. *2009 Sixth International Conference on Computer Graphics, Imaging and Visualization* (pp. 235-240). Tianjin, China: IEEE.

Wertheimer, M. (2012). *On perceived motion and figural organization.* (L. Spillman, Ed.) Cambridge, MA: MIT Press.

Yue, K. (2009). *Computation-friendly shape grammars: With application to determining the interior layout of buildings from image data.* Pittsburgh: Carnegie Mellon University, School of Architecture.

Yue, K., & Krishnamurti, R. (2008). A technique for implementing a computation-friendly shape grammar interpreter. *Design Computing and Cognition '08* (pp. 61-80). New York, NY: Springer.

Yue, K., Krishnamurti, R., & Gobler, F. (2009). *Computation-friendly shape grammars.* Pittsburgh, PA: Carnegie Mellon University, School of Architecture.