UNIVERSITÄT
BERN

A Shape Grammar Interpreter Using Local Coordinates For Subshape Detection

Bachelor Thesis

Lars Wüthrich

Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

February 2018

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Manuel Leuenberger

Software Composition Group
Institut für Informatik
University of Bern, Switzerland

Abstract

Shape Grammars allow us to create complex and intricate recursive images by applying geometric transformations. Instead of rewriting strings as in text based grammars, the production rules directly operate on the geometry of a shape. However, there can be several possibilities where a given rule could be applied to a shape. The existing implementations and theoretical approaches assume that the decision where to apply a rule has to be made manually by the designer. In this work, various heuristics are considered to let the shape grammar interpreter run by itself with minimal user input at the beginning. In addition, a new method of subshape detection is presented based on point comparison in local coordinates, which is needed for the shape grammar interpreter to find subshapes to apply the rules to. This novel detection algorithm and several of the heuristics are implemented in a proof of concept interpreter and editor. They are used to create various images that would have been tedious to do in this detail by manual interaction with the interpreter.

Contents

List of Figures

List of Algorithms

Listings

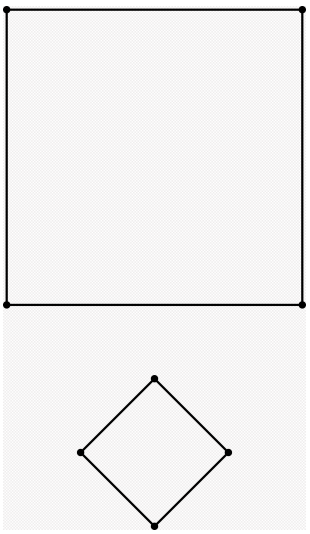
1 Introduction

Shape Grammars allow us to create complex and intricate recursive images by using geometric transformations. An implementation of a shape grammar interpreter can take specifications for rules and applies the rules to the starting shape, which can be rendered to produce images. Instead of rewriting strings as in text based grammars, the production rules of the form $\alpha \rightarrow \beta$ work directly on a shape, where α and β are also shapes themselves. Figure 1.1 shows such a rule where the transformation adds new points and lines to an existing square. This rewriting of geometry can be used to create more complex shapes by applying it repeatedly.

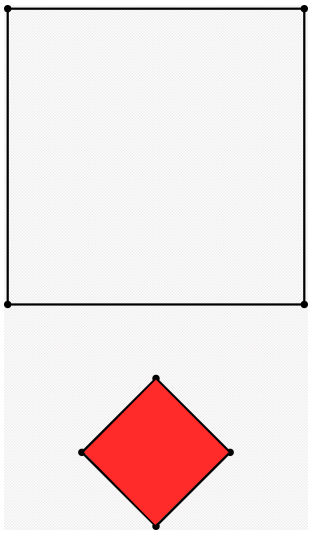
(a) α shape, the left hand side of the rule.

(b) β shape, the right hand side of the rule.

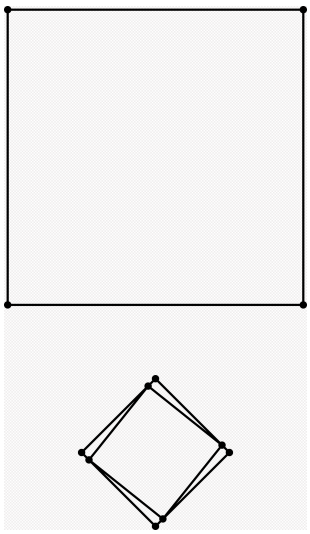
Figure 1.1: A rule with a left hand side α and a right hand side β.



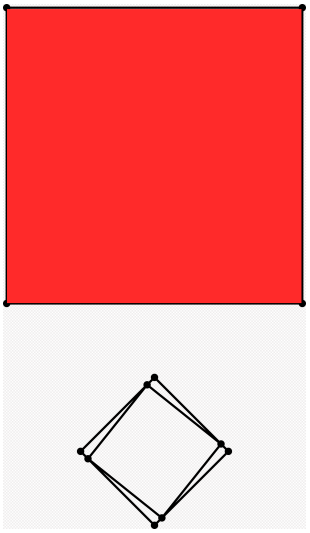
(a) We start with a shape called γ, the starting shape. It consists of two squares, the smaller one rotated.



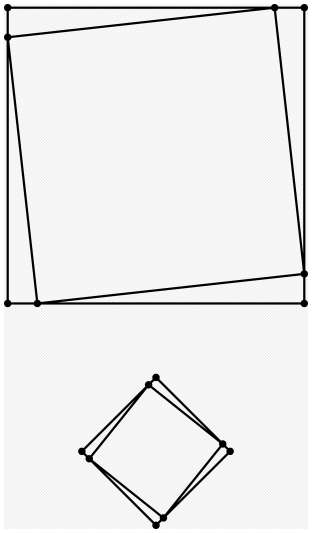
(b) The subshape match is indicated in red. The rule will be applied at this position.



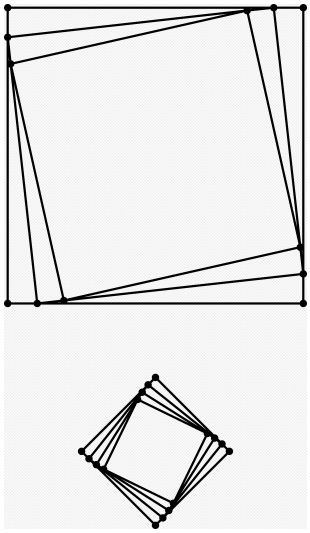
(c) Apply the rule once.



(d) Find a new subshape match. This time inside the big square.



(e) Apply the rule a second time.



(f) γ after applying the rule a few times more.

Figure 1.2: A starting shape that is transformed several times by the rule from Figure 1.1.

A shape grammar example is given in Figure 1.2, which shows how the rule from Figure 1.1 is applied. First, we try to find the left hand side shape called α of the rule in Figure 1.1 somewhere inside the starting shape called γ from Figure 1.2(a). We can see that there are two squares in γ and we arbitrarily choose the smaller square first indicated in Figure 1.2(b). The red area marks the subshape match we found by transforming α with a rotation, scaling, and translation. We call this transformation τ. The small square is therefore the shape $\tau(\alpha)$, which means the points in α are transformed to end up as the points from the marked subshape match. Then we apply the rule once to create the image in Figure 1.2(c). This is done by removing the red shape $\tau(\alpha)$ and adding the shape $\tau(\beta)$ from the rule in Figure 1.1. It is important to note that we have to apply the same transformation τ to β if we add it to the target shape γ. Otherwise β does not end up at the same position, with the correct scaling, and rotation as the red subshape $\tau(\alpha)$. After the shape is transformed by the rule, we find a new subshape and apply the rule again. This time we select the bigger square in Figure 1.2(d). Applying the rule now yields the image in Figure 1.2(e) and several more applications produces the image in Figure 1.2(f). The process of applying a rule to a shape γ can be written as $[\gamma - \tau(\alpha)] + \tau(\beta)$ according to Stiny [8]. We remove the left hand side and add the right hand side both with respect to a transformation τ.
In existing shape grammar implementations, the generation of designs is assumed to be an interactive process between the interpreter and the user. When a rule and a target shape are supplied, the interpreter presents all subshapes to which the rule can be applied. However, the selection of the subshape, to which to apply the rule, has to be done manually.
In this work, it is assumed that the user supplies parameters which are encoded in the grammar and lets the configured interpreter run by itself to create images without further user input. In order to do this, any unwanted transformations on a shape have to be filtered out. In addition, various scoring strategies are presented in Section 4.2, which help the interpreter decide by itself where to apply a rule if several possibilities exist. For example, in Figure 1.2(a) we have two squares the interpreter can choose from.
A self-running shape grammar interpreter can therefore be divided into the following distinct stages:

1. Rule and starting shape definition
   Define the rules and the starting shape either directly in code, using the domain specific language or by drawing them in the editor.

2. Declaration of filtering and selecting parameters
   From all possible subshapes some are not needed and can be filtered out, which is configured by these parameters. After the filtering, one match from the remaining ones has to be selected, which is configured by the selection parameters.

3. Subshape Detection (Chapter 3)

The interpreter has to find a transformation τ from a given left hand side α and a target shape γ.

4. Match Filtering (Section 4.1)
   If there are more than one subshape matches, unwanted ones have to be filtered out.

5. Scoring and selection of found matches (Section 4.2)
   From the remaining subshapes the best one is selected by scoring them with certain heuristics.

6. Rule application
   Apply the rule to γ with the given subshape.

The subshape detection problem in stage 3 is treated with a novel solution explained in Section 3.3. With the aim to test the applicability of the subshape detection algorithm 1, a shape grammar interpreter and a proof of concept editor is built in Pharo (Smalltalk). A small domain specific language helps us to quickly define shapes used in the editor and interpreter. Several of the filtering mechanisms presented in Section 4.1 and the match selection strategies in Section 4.2 are implemented. The code resides in a repository [12] and Chapter 8, describes the inner workings of the interpreter and editor.

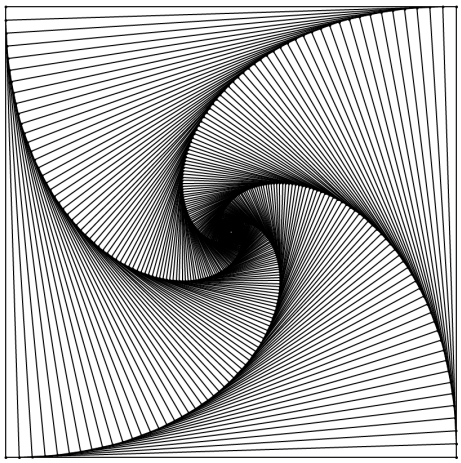


Figure 1.3: Recursive Square 170 rule steps

2
Related Work

Shape grammars were first defined by Stiny et al. [9] and further formalized by Stiny [8]. At that time any images created by these grammars were hand drawn and no computer implementation of such a shape grammar interpreter had yet existed. Although the work on shape grammars had a more theoretical nature then, many important concepts and notations were defined. The biggest technical difficulty in implementing a shape grammar interpreter poses the subshape detection problem. The left hand side α of the rule has to be found as a subshape inside a target shape γ in order to apply the rule. Krishnamurti [4] introduces a sophisticated algorithm to solve the subshape detection problem, which allows us to build such a shape grammar interpreter. It is based on finding a transformation τ from three points in the subshape to three others in the target shape. This algorithm finds the linear transformation from three points in α to any three points in γ by creating equations and solving them for the coefficients of the transformation. Therefore the method from Krishnamurti [4] can be used effectively to calculate this transformation τ.
A new algorithm is proposed here to solve the same problem but with a different method. It is based on point comparison in local coordinate systems. Both the potential subshape α and the target shape γ are transformed by matrix multiplication into a local coordinate system. If they match in the local system, α is a subshape of γ. The transformation matrix is recovered by going from the original system in α to the local system and then back to the system in γ. Because we have to transform both shapes to a local coordinate system instead of transforming α once to the coordinate system in γ it is slightly less efficient than the standard algorithm but has an intuitive idea behind it and is simple to implement.

Although shape grammars did not have many practical applications in the last years, they are getting more attractive for certain computer graphic applications, such as by Zhang et al. [13], to guide a grammar with vector fields over an object in order to create surface patterns. Santoni and Pellacini [7] use a shape grammar inspired concept called group grammars in order to subdivide and decorate already present geometry with tangle patterns.
Shape grammars were primarily used to design objects that could be manufactured, for example Agarwal and Cagan [1] or Brown et al. [2] show the means to create designs of real world objects. Apart from that, shape grammars could also be used in architecture to design buildings. In these problem domains a designer is needed who chooses where to apply a rule. Shape grammars were more seen as a tool for the designer and it was not necessary to create a self-running shape interpreter. Therefore, this aspect was apparently not well developed since the definition of shape grammars and this work explores certain characteristics needed in order to reduce user input. Let us assume we want to create several hundred images procedurally that should look similar but also show variations between them. Each of these images will have intricate patterns and require maybe a few hundred rule transformations from a given starting shape. If we want to do this by hand, it will be tedious and require a lot of time because every rule transformation needs several manual interactions by the designer. It would be much more convenient to specify parameters at the beginning, then let the interpreter select appropriate locations to apply a rule and let it run for a few hundred iterations. In order to create variations between the images, parameters are changed slightly and the interpreter is run again. This process requires much less user interaction and produces many different images that can also be hand selected afterwards. Match filtering and selection strategies are added to the interpreter in order to accomplish this. They help the interpreter to choose a suitable position to apply the rule from a number of possibilities.
Apart from the generation of the geometry itself, coloring is also an interesting aspect to improve the visuals of images. Stiny [8] presents a usable method for coloring a restricted set of grammars, as well as Knight [3] who defines a new type of grammar to create already coloured shapes. The editor implemented in this work uses a different method to color shapes. A function is defined per rule that transforms the right hand sides color relative to the color of the left hand side of the rule. This allows the interpreter to transform shapes with arbitrary colors instead of fixed hard-coded ones.
Finally the standard subshape detection algorithm from Krishnamurti [4] and the alternative Algorithm 1 proposed could also find utilization in problem areas different than shape grammars where only subshape matching is required.

3
Subshape Detection

In order to apply a rule to a target shape γ we first have to find the rule's left hand side α inside the target shape γ. The problem of finding a shape inside another shape regarding some transformation is called subshape detection, which means finding a transformation τ that leads to $\tau(\alpha)$ being a part of the target shape.
At a first glance, either distances or angles of connected points could be compared between the potential subshape and the target shape. However, such a simple comparison is not possible due to the fact that any linear transformation or finite composition like translation, rotation, and scaling may be applied to the subshape so that it is part of another shape. This makes it impossible to compare point distances because any non-uniform scaling or shearing would remove any distance similarities. Similarly, any angles may be changed by rotation, non-uniform scaling and/or reflections. This section addresses this problem by using a new algorithm based on point comparison in local coordinates. Instead of comparing points and lines directly we first transform both shapes into a coordinate system where we can easily detect if they are similar enough to count as subshapes.

3.1 Homogeneous Coordinates

In order to work with translation, scaling and rotation in a uniform way, points in a 2D coordinate system are represented by homogeneous coordinates (refer for example to Mortenson [6, p. 69–71]) in the form:

$$p = \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \text{ and } v = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$$

where p is a point (its z component is 1) and v is a vector (its z component is 0).

Transformation matrices are extended from 2×2 to 3×3.

$$M = \begin{pmatrix} m_{11} & m_{12} & x \\ m_{21} & m_{22} & y \\ 0 & 0 & 1 \end{pmatrix}$$

where x, y encode translation and m_{11}, m_{12}, m_{21}, m_{22} encode scaling and rotation.

An advantage of homogeneous coordinates is that we can represent both 2D vectors and points uniformly with a three coordinate vector by setting the z part either to zero (a vector or point at infinity) or to one (a point). In effect, homogeneous coordinates allow us to apply rotation, translation, and scaling uniformly to points and vectors by use of matrix multiplication.
We can define rotation, scaling, and translation as follows:

Rotation by θ

$$\begin{pmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} cos(\theta) \cdot x - sin(\theta) \cdot y \\ sin(\theta) \cdot x + cos(\theta) \cdot y \\ 1 \end{pmatrix}$$

Scaling with the vector $\begin{pmatrix} a \\ b \end{pmatrix}$

$$\begin{pmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} a \cdot x \\ b \cdot y \\ 1 \end{pmatrix}$$

Translation of a point by the vector $\begin{pmatrix} a \\ b \end{pmatrix}$

$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + a \\ y + b \\ 1 \end{pmatrix}$$

Translation of a vector does nothing as expected

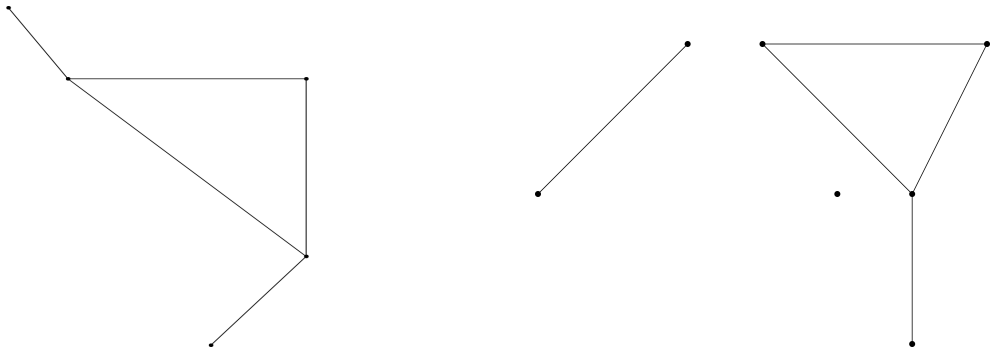
$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ 0 \end{pmatrix}$$

3.2 Grammar, Rule and Shape Representation

Our definitions try to adhere to the standard terms for example defined by Stiny [8].

Shape

A shape $S = \{P, L\}$ consists of a set of points $P = \{p_1, p_2, \ldots, p_n\}$, where each 2D point $p_i \in P$ is represented with homogeneous coordinates.
Additionally, $L = \{l_1, l_2, \ldots, l_m\}$ is a set of lines where $l_i = \{p_s, p_t\}$ references a starting and an endpoint in P.
Each point and line can also have a label. Stiny [8] calls it *labelled shape* due to the shape having *labelled points*. Furthermore, more labels and supplemental data can be attached to any point or line, for example point or line colour or properties like thickness and point radius. Also, points can exist on their own and do not need to be endpoints of a line. Not all lines have to be interconnected. Figure 3.1(a) shows an interconnected shape and Figure 3.1(b) shows a shape that has an unconnected line and a single point. This definition of a shape differs in regard to lines from the one used by Stiny [8] or by Krishnamurti [4] who use maximal line representation. Maximal lines group together all lines which are collinear, which does not need to be the case for line segments chosen in the implementation of the interpreter.



(a) (b)

Figure 3.1: Example of valid shapes

Starting Shape γ

γ is the shape we start with before any transformations. Rules are then applied to this shape in sequence to transform it. After several rule iterations on γ a final image is produced. A starting shape is similar to the initial symbol in formal grammars.

Subshape

A shape s is a subshape of another shape s' if all points and lines of s are contained in the shape s'. We try to find the left hand side α as a subshape in γ in order to apply the rule with regard to this subshape.

Rule

A rule $R = \{\alpha, \beta\} = \alpha \rightarrow \beta$ is defined by two shapes, α and β. Both have some common coordinate system, which is important to distinguish if, for example, β is scaled relative to α.
If the left hand side is found as some subshape $\tau(\alpha)$ in a target shape γ, $\tau(\alpha)$ will be replaced with $\tau(\beta)$. Some rule examples are given in Section 3.2.1. In the work from Stiny [8] a rule application is written as:
$\gamma' = [\gamma - \tau(\alpha)] + \tau(\beta)$, where $+, -$ represent the shape union and subtraction, meaning adding points and lines or respectively removing them. Apart from the subshape detection, our interpreter follows the same definition with respect to the transformation from γ to γ'. The left hand side α is removed as a subshape from γ whereas the right hand side β is added with respect to where α was found in γ.

Transformation τ

An euclidean transformation τ encoded in matrix form is used to transform points from α to γ. Transformations also transform points from α and γ into a local coordinate system and back which is heavily used in Algorithm 1.

Grammar

A grammar can be formalized according to Stiny [8] by a set of shapes, symbols, rules, and a starting shape. A grammar $G = \{r_1, r_2, \ldots, r_k\}$ modeled in the interpreter simply has a specified number of rules which can be chosen to transform the target shape γ. Such a transformation can be subdivided into several steps:

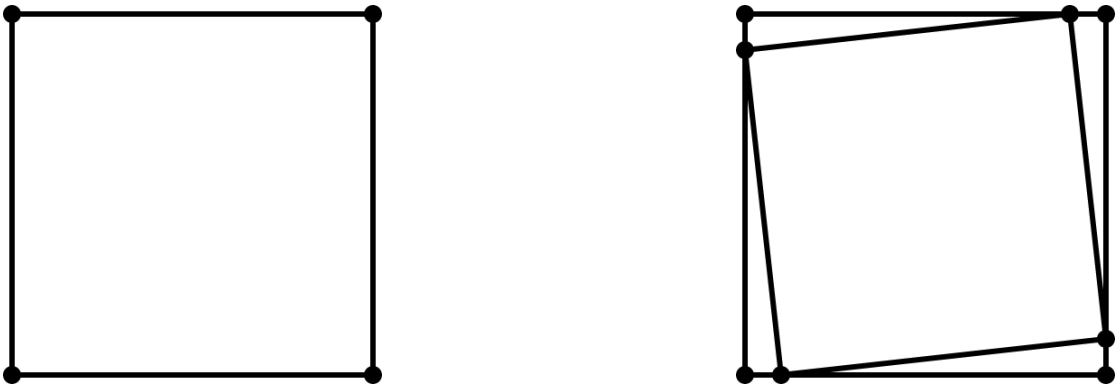
1. Select a rule $r := \alpha \rightarrow \beta$
2. Select a target γ
3. Find a transformation τ in order that $\tau(\alpha)$ is a subshape inside γ
   If no such transformation is found stop the interpreter
4. Apply the rule r on γ
   Remove $\tau(\alpha)$ from γ

Add $\tau(\beta)$ to γ
Go back to 3

5. Color the geometry and display it

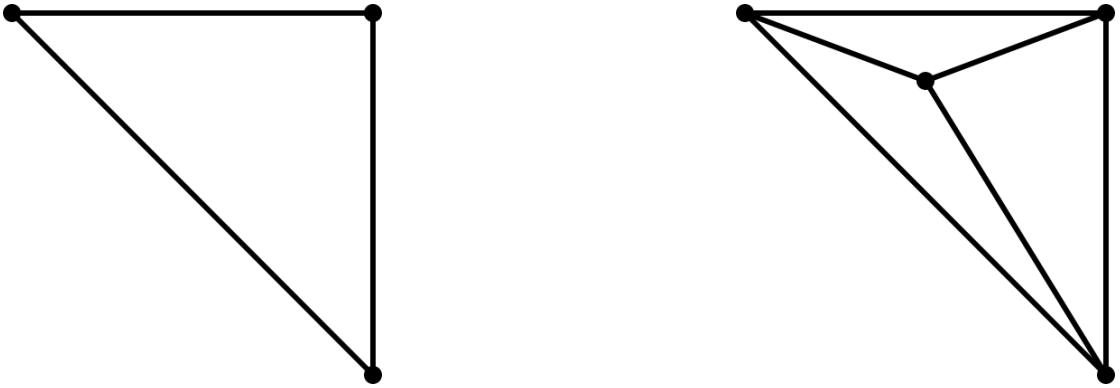
3.2.1 Example grammars

This section shows a few sample grammars each having only one rule. Note that these shapes do not depict graphs but points and lines in two dimensions. The coordinate system is omitted because the rule is already evident from the relationship between α and β. However, each rule would have its own coordinate system to which α and β are relative. For a self-running interpreter two types of rules are interesting to consider. A rule can create recursive patterns like Figure 3.2. Every time the rule is applied a smaller square is slightly rotated and inlaid into the existing ones. Another example is Figure 3.3 where a triangle is split into three smaller ones for each rule step. Another interesting rule category is the generation of tiling patterns with symmetric figures. Both of these types of rules can usually be applied indefinitely to a shape. Figure 3.4 shows a rule that produces a hexagonal tiling.



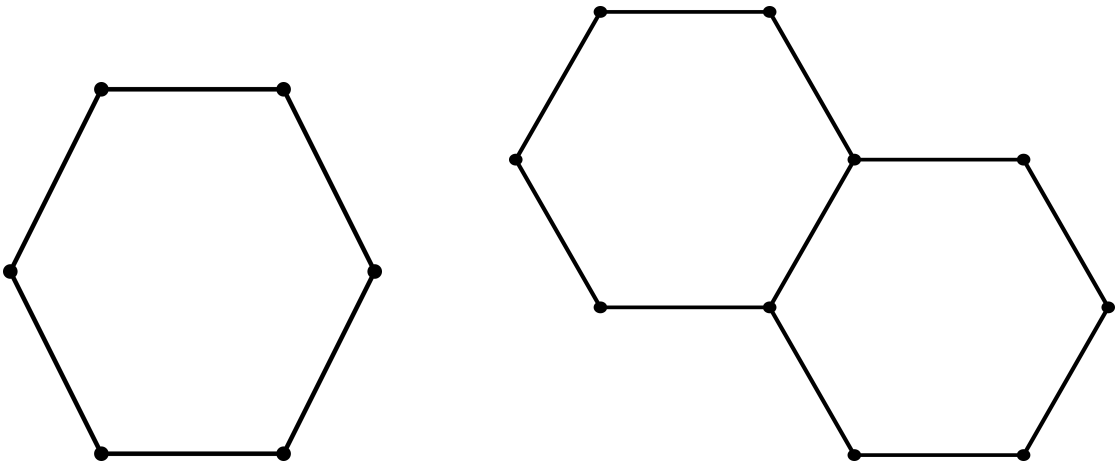
(a) α (b) β

Figure 3.2: Recursive Square Grammar



(a) α shape (b) β shape

Figure 3.3: Triangle Inlay Grammar



(a) α shape (b) β shape

Figure 3.4: Hexagon Tiling Grammar

3.3 Subshape Detection By Point Comparison In Local Coordinates

3.3.1 Coordinate Transformation

Coordinate transformations in 2D are the basis of the proposed subshape detection method. Transformations will work on 2D points represented with homogeneous coordinates.
For any three points p_0, p_1, p_2 we can create a new coordinate system by choosing $o = p_0$ as the new origin and defining the two vectors $v_1 = p_1 - p_0$ and $v_2 = p_2 - p_0$ as the axes for a new coordinate system.
A matrix used to transform all points into this new local coordinate system would look like this:

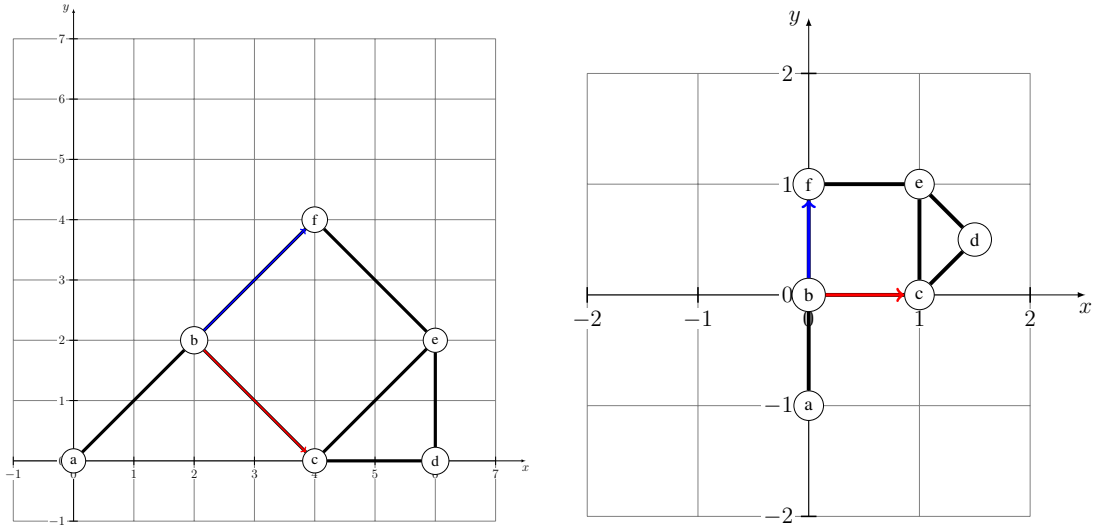
$$M_{world \to local} = M^{-1}_{local \to world} = \begin{pmatrix} v_1 & v_2 & o \\ & & \end{pmatrix}^{-1} = \begin{pmatrix} x_{v_1} & x_{v_2} & x_o \\ y_{v_1} & y_{v_2} & y_o \\ 0 & 0 & 1 \end{pmatrix}^{-1} \tag{3.1}$$

M^{-1} denotes the inverse of a matrix M. $M_{world \to local}$ is the matrix that transforms from the original coordinate system (the *world* system) into local coordinates defined by the new origin and axes. $M_{local \to world}$ does the inverse, going from the local coordinate system back to the original one. However, to invert a matrix M, the three vectors v_1, v_2, o have to be linearly independent.
To create a suitable transformation matrix, the three points need to form a valid coordinate system. If all three points are on the same line, v_1 and v_2 will be linearly dependent and the matrix will not be invertible. In Krishnamurti [4] those three points are called distinguishable. Even though the algorithm presented there is not using coordinate systems, the same restriction applies that the three points cannot be on the same line but need to form a triangle with a non-zero area. In our current implementation of the interpreter, the first three points are chosen as a coordinate system by default. If they do not form a coordinate system all subsequent points are considered. In case no such system can be found, an exception is thrown.

3.3.2 Transformation Matrix Example

This is a short example that shows how to transform the shape in Figure 3.5(a) into the shape in Figure 3.5(b). We choose the red arrow from point b to c as the new x-axis and the blue arrow from point b to f as the new y-axis.

The point b itself will be the new coordinate system's origin. To create a matrix that



(a) γ in original coordinate system

(b) γ in new local coordinate system

Figure 3.5: Transforming γ into a local coordinate system. The red and blue arrows denote the new x- and y-axis respectively.

transforms the shape γ in Figure 3.5(a) we will use Equation 3.1:

$$M_{world \to local} = M^{-1}_{local \to world} = \begin{pmatrix} v_1 & v_2 & o \\ & & \end{pmatrix}^{-1} = \begin{pmatrix} x_{v_1} & x_{v_2} & x_o \\ y_{v_1} & y_{v_2} & y_o \\ 0 & 0 & 1 \end{pmatrix}^{-1}$$

The new origin o is the point $b(2, 2)$.

$$o = \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix}$$

Note that the third entry of the vector is a 1, which means in homogeneous coordinates it is a point.
The new x axis v_1 is the vector from b to c.

$$v_1 = c - b = \begin{pmatrix} 4 \\ 0 \\ 1 \end{pmatrix} - \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ -2 \\ 0 \end{pmatrix}$$

In this case the z coordinate is 0 because v_1 is a vector and not a point.
In the same fashion we can calculate the new y-axis v_2:

$$v_2 = f - b = \begin{pmatrix} 4 \\ 4 \\ 1 \end{pmatrix} - \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} 2 \\ 2 \\ 0 \end{pmatrix}$$

Insert this into Equation 3.1:

$$M_{world \to local} = M^{-1}_{local \to world} = \begin{pmatrix} v_1 & v_2 & o \\ & & \end{pmatrix}^{-1} = \begin{pmatrix} 2 & 2 & 2 \\ -2 & 2 & 2 \\ 0 & 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} \frac{1}{4} & -\frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & -1 \\ 0 & 0 & 1 \end{pmatrix}$$

If we use this matrix $M_{world \to local}$ to transform every point in Figure 3.5(a) we should end up with the corresponding point in Figure 3.5(b) in local coordinates of the new coordinate system.
In order to show this, a few sample points are transformed.
The point b should end up as the new origin $(0, 0)$:

$$b' = M_{world \to local} \cdot b = \begin{pmatrix} \frac{1}{4} & -\frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & -1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} - \frac{1}{2} \\ \frac{1}{2} + \frac{1}{2} - 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

The points c and f can each be met by one step from the origin with one of the axes. Therefore they should have coordinates $(1, 0)$ and $(0, 1)$ respectively:

$$c' = M_{world \to local} \cdot c = \begin{pmatrix} \frac{1}{4} & -\frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & -1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{4} \cdot 4 \\ \frac{1}{4} \cdot 4 - 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

$$f' = M_{world \to local} \cdot f = \begin{pmatrix} \frac{1}{4} & -\frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & -1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 4 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{4}{4} - \frac{4}{4} \\ \frac{4}{4} + \frac{4}{4} - 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

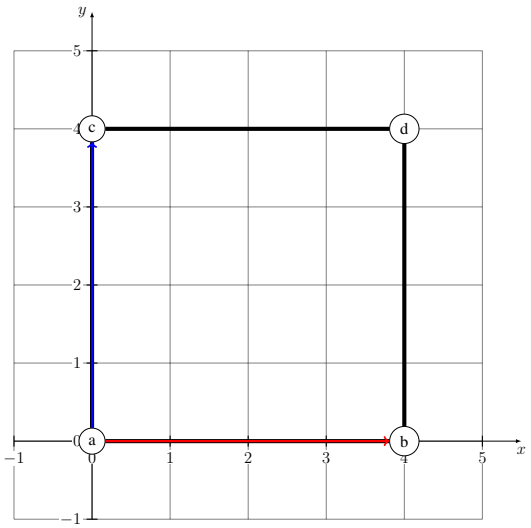
and as a last example point d:

$$d' = M_{world \to local} \cdot d = \begin{pmatrix} \frac{1}{4} & -\frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & -1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 6 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{1}{4} \cdot 6 \\ \frac{1}{4} \cdot 6 - 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1.5 \\ 0.5 \\ 1 \end{pmatrix}$$

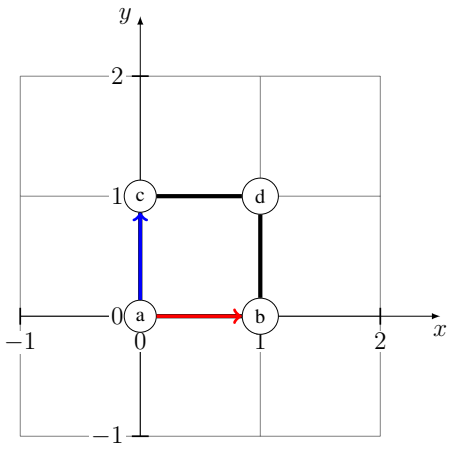
With the matrix $Mworld \to local$ we can transform every point into the local coordinates of the new system.

3.3.3 General Idea Behind The Algorithm

In order to find α such as in Figure 3.6(a) as a subshape inside the target shape γ for instance Figure 3.5(a), we first transform every point in α into a local coordinate system depicted in Figure 3.6(b). This is done by choosing any three points in α which form a coordinate system and creating a matrix out of thesese as outlined in Section 3.3.1.



(a) α shape in original coordinate system, yet untransformed



(b) α shape in local coordinates, scaled by $1/4$ in x and y direction.

Figure 3.6: Transforming α into a local coordinate system. The red and blue arrows denote the new x- and y-axis respectively.

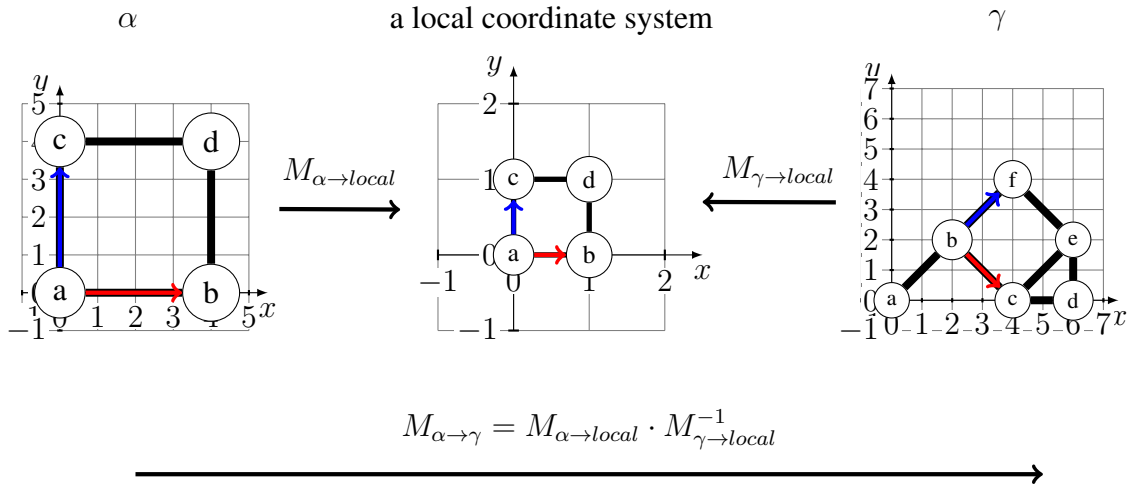


Figure 3.7: Recover the transformation from α to γ.

Similarly, γ is also transformed into a local coordinate system. If the points from α in the local system match up with points from γ and the same lines connect the same matched points in both, we have found a subshape match. The transformation matrix $M_{\alpha\to\gamma}$ which transforms points from α directly onto the matched points in γ can be reconstructed by going from $\alpha \to local\ \ system \to \gamma$ shown in Figure 3.7. Each of these coordinate changes are already explicitly present as transformation matrices.

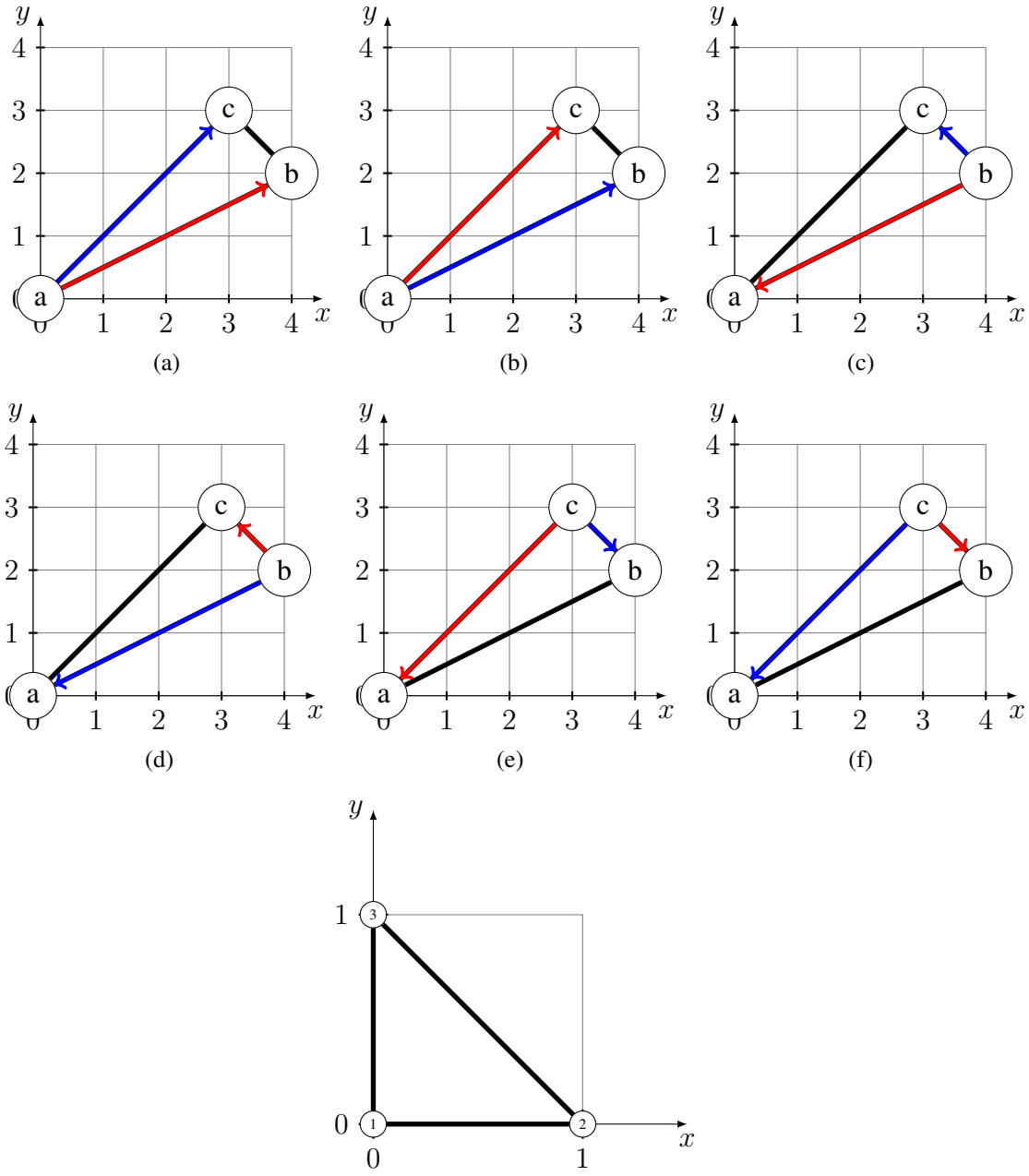
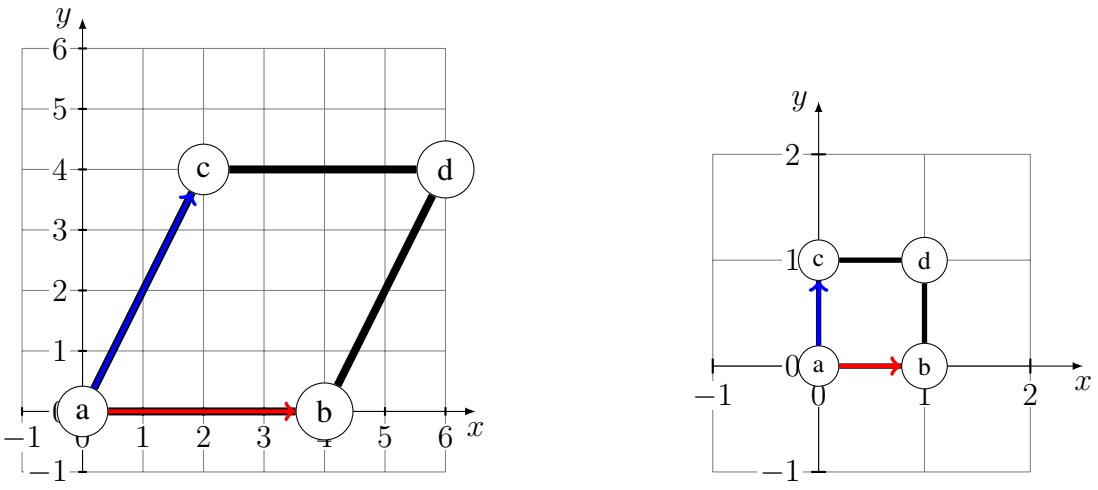


Figure 3.8: Six different coordinate system choices lead to the same triangle in local coordinates ignoring their labels. The red and blue arrows denote the new x and y axes respectively.

Note that several choices of three points in Figure 3.8 lead to the same triangle apart from ordering of its points. Every one of these triangles in local coordinates would be matched with the triangle in Figure 3.3 for example. This leads to the interesting question what should be considered a subshape and what not. A parallelogram for



(a) An untransformed parallelogram

(b) The same parallelogram now a square in local coordinates

Figure 3.9: Any parallelogram and square are subshapes of each other

instance would also be a subshape of a square and vice versa as in Figure 3.9. This might be counter intuitive to what might be considered as a subshape and could be configured, see Section 4.1.2.

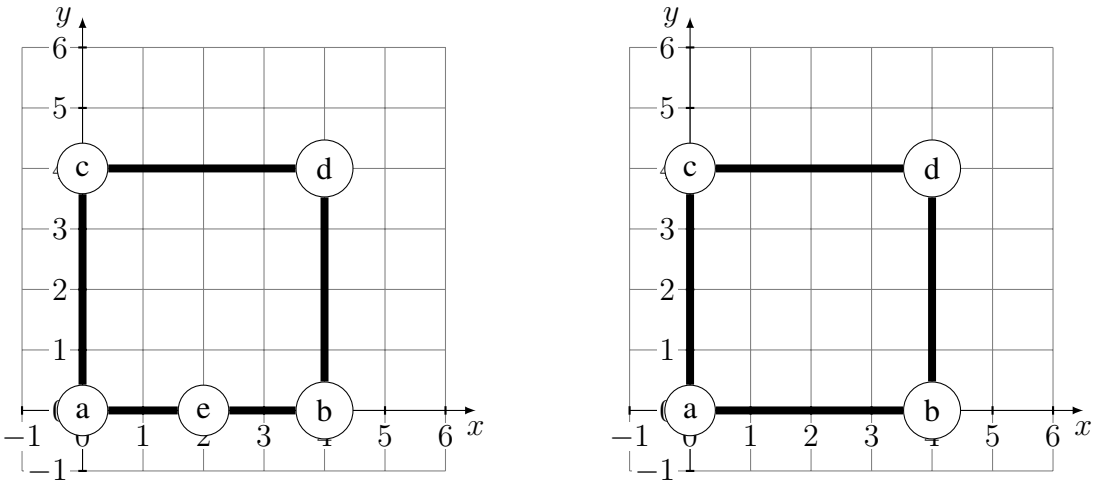
3.3.4 Maximal Lines

In the beginning of writing the interpreter it was not clear whether maximal lines described by Stiny [8] and by Krishnamurti [4] were by all means necessary. Only when progressing further with the editor an example was found that requires maximal lines instead of line segments. Lines are grouped into one common maximal line, if they share endpoints and are collinear or if the collinear lines overlap. In this case, they form visually a bigger line with endpoints spanning over the whole collection of these overlapping lines. Figure 3.10 is a good example for the problem. Even though the squares in Figure 3.10(a) and Figure 3.10(b) look inherently the same, the second square would not match inside the first if we compare their line segments.

If we compare line segments directly, we see that point a is not connected to point b in Figure 3.10(a) because a and b are connected to the point e. Therefore the line segments in Figure 3.10(a) do not match with the line segments in the other Figure 3.10(b).

In this case, neither the left shape is a subshape of the right nor the right shape a subshape of the left one.

If we compare using maximal lines the following happens. The segments $a - e$ and $e - b$ are grouped together to a maximal line $a - b$ because points of both segments are collinear and they share one endpoint e. The maximal line in the right Figure 3.10(b) is also the line $a - b$, which is contained in the maximal line $a - b$ of the left shape. Therefore, the right shape is a subshape of the left and vice versa. The subshape detection



(a) A square with an additional point

(b) A simple square

Figure 3.10: Example in which maximal lines would be needed. (b) would not match inside (a) even though both have the same general outline

algorithm explained in the following section can still find valid subshapes even though maximal lines are not implemented in the interpreter. The point comparison in local coordinates does not suffer from this and generates all valid transformations τ if all points match. However, to find all possible subshapes the $doLinesMatch$ call in the algorithm needs to address maximal lines in the actual implementation.

3.3.5 Algorithms

Algorithm 1: Subshape Detection $findSubshapes(\alpha, \gamma)$

Input:
$\alpha :=$ a shape consisting of points ($\alpha.points$) and lines ($\alpha.lines$)
$\gamma :=$ the target shape which is checked for subshapes
Output:
$matches := \{m_1, m_2, m_3, \ldots, m_n\}$ a list of all found matches
$m_i = \{M_{\alpha\to\gamma}, pointMap, lineMap\}$
where
$M_{\alpha\to\gamma}$: is the transformation matrix from the coordinates in α to γ
$pointMap$: maps matched points in α to their corresponding points in γ
$lineMap$: maps matched lines in α to lines in γ

```
1  matches := {}
2  p1, p2, p3 := findCoordinatePointsIn(α.points)
3  M_local→α := createMatrix(p1, p2, p3)
4  M_α→local := (M_local→α)^-1
5  // α' contains points in local coordinates
6  α' := M_α→local.transform(α)
7  for p1, p2, p3 ∈ γ, p_i ≠ p_j, i ≠ j do
8      M_local→γ := createMatrix(p1, p2, p3)
9      if M_local→γ.determinant() = 0 then
10         continue
11     M_γ→local := (M_local→γ)^-1
12     // γ' contains points in local coordinates
13     γ' := M_γ→local.transform(γ)
14     pointsMatch := doPointsMatch(α'.points, γ'.points, pointMap)
15     if pointsMatch then
16         linesMatch := doLinesMatch(α.lines, γ.lines, pointMap)
17         if linesMatch then
18             lineMap := storeLineMatches(α, γ)
19             match := Match.new
20             M_alpha→γ := M_α→local · M_local→γ
21             match.transformation := M_alpha→γ
22             match.pointMap := pointMap
23             match.lineMap := lineMap
24             matches.add(match)
25 return matches
```

Algorithm 2: Find Coordinate Points $findCoordinatePointsIn(P)$

Input:
$P := \{p_1, p_2, \ldots, p_n\}$ a set of points with n elements

Output:
$\{p_1, p_2, p_3\}$ a triple of points which form a coordinate system

```
1 p_1 := P.at(1)
2 p_2 := P.at(2)
3 for i := 3 to P.size do
4 |   p_i := P.at(i)
5 |   M := createMatrix(p_1, p_2, p_i)
6 |   if M.determinant() ≠ 0 then
7 |   |   return {p_1, p_2, p_i}
8 return Error: No Coordinate System Found
```

Algorithm 3: Create a transformation matrix $createMatrix(p_1, p_2, p_3)$

Input:
p_1, p_2, p_3 three points defining a valid coordinate system

Output:
$M_{local \to world}$ a matrix transforming points from the local coordinate system specified by p_1, p_2, p_3 back to the original "world" coordinate system
where p_1 is the origin
$v_1 := p_2 - p_1$ is the first axis
$v_2 := p_3 - p_1$ is the second axis

```
1 origin := p_1
2 v_1 := p_2 − p_1
3 v_2 := p_3 − p_1
4 M_{local→world} := Matrix.new
5 M_{local→world}.setColumn(1, v_1)
6 M_{local→world}.setColumn(2, v_2)
7 M_{local→world}.setColumn(3, origin)
8 return M_{local→world}
```

3.3.6 Computational Complexity

In this section we show that the subshape detection algorithm runs in polynomial time. The algorithm is briefly compared with the standard method from Krishnamurti [4]. Algorithm 1 denotes the complete subshape detection to find all subshape matches from a shape α in another shape γ. Two helper functions are used, one to find a suitable coordinate system in a set of points (Algorithm 2) and another one to create a transformation matrix (Algorithm 3). In order to define the computational complexity following variables are used:

- n number of points in a shape
- l number of lines in a shape
- n_α number of points in shape α
- l_α number of lines in α
- n_γ number of points in shape γ
- l_γ number of lines in γ

Any point comparisons are done within an equality range, because coordinates cannot be compared directly due to floating point imprecisions. Points which have a smaller distance from each other than a small experimentally determined range are considered *equal*.
The $findCoordinatePointsIn$ call from Algorithm 2 to find a coordinate system in a shape is in $\mathcal{O}(n)$ if n is the number of points in the given shape. The loop runs once through all points and builds a coordinate transformation matrix described in Algorithm 3.3.1. The $createMatrix$ call in the Algorithm 3 needs constant time k. If this matrix has a determinant which is non-zero, a coordinate system can be built from those three points and the transformation matrix is returned.
Line 2 and 6 from Algorithm 1 are in $\mathcal{O}(n_\alpha)$ because the transform call transforms every point in α with the matrix $M_{\alpha \to local}$. Therefore, it loops through every point in α.
The loop running from lines 7 to 23 is called $n_\gamma \cdot (n_\gamma - 1) \cdot (n_\gamma - 2)$ times because all three distinct points in γ are considered. The $doPointsMatch$ call in line 13 goes over all points in α and checks whether the coordinates match with all points in γ, therefore having to run $n_\alpha \cdot n_\gamma$ times. If all the points match, all the lines of the shape α which was found in γ are considered, whether the rules left hand side α has the same lines as the subshape found in γ, the dictionary $pointMap$ maps points in α to subshape points in γ. The $doLinesMatch$ call therefore goes over all lines in α and for every line goes over all lines in γ, to see if the starting and end point of each line matches, therefore having to run $l_\alpha \cdot l_\gamma$ times.

The whole subshape detection algorithm runs in
$\mathcal{O}(n_\alpha + n_\gamma \cdot (n_\gamma - 1) \cdot (n_{\gamma-2}) \cdot (n_\gamma + n_\alpha \cdot n_\gamma + l_\alpha \cdot l_\gamma))$.
Due to the nature of shape grammars, rules usually stay fixed, therefore the number of points in α do not change and the runtime can be considered for a fixed α:
$\mathcal{O}_\alpha(n_\gamma \cdot (n_\gamma - 1) \cdot (n_{\gamma-2}) \cdot (n_\gamma + n_\gamma + l_\gamma)) = \mathcal{O}_\alpha(n_\gamma^4 + n_\gamma^3 \cdot l_\gamma)$

If we only look at the point comparison part, which is enough to find a transformation τ so that the points match, this will be in $\mathcal{O}_\alpha(n_\gamma^4)$.
This method of subshape detection is slightly less efficient than the algorithm presented by Krishnamurti [4]. The standard algorithm for subshape detection only transforms points from α to γ. The method presented here has to transform both points in α, as well as from γ into a local coordinate system. Because the number of points in γ increase this additional transformation from γ to a local coordinate system is costly and increases the degree of the polynomial by one. Even though this is the case, it is still a polynomial and the algorithm is used effectively in the implemented interpreter.

4
Match Filtering And Selection

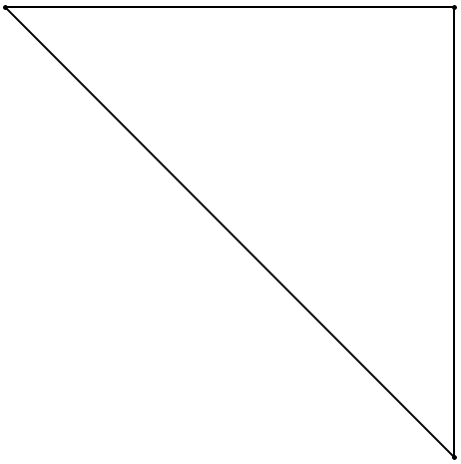
4.1 Filtering

The implementation goal of the interpreter is to reduce active input while the image generation is ongoing. The interpreter has to find subshape matches of α inside the target shape and choose according to some strategy. This section provides a few simple conditions by which unwanted subshape matches can be removed. The filter 4.1.1 and 4.1.2 have been implemented in the current interpreter, and the other ones are described briefly.

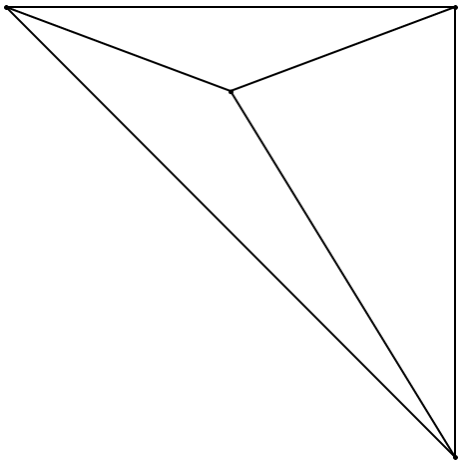
4.1.1 Filter Intersections

When generating new points some of them might already be present in the shape. In the extreme case, a rule transformation would change nothing in the base shape. If a rule application on a specific subshape has no effect e.g. all generated points are already there, we can discard it. Based on preferences, a match could also be discarded if there are only some points already present or they can be reused.
Also, by moving points or creating lines, they might intersect. If a match leads to an intersection of lines, it can also be discarded or allowed, again based on preferences of the user. Figure 4.1 shows such a case of self intersecting lines when using the triangle inlay rule from Figure 3.3 which splits triangles into three new triangles. If after splitting a triangle once we choose the big triangle again as a subshape a line intersection occurs. Note that a triangle has several subshapes given rotation, reflection, and scaling which

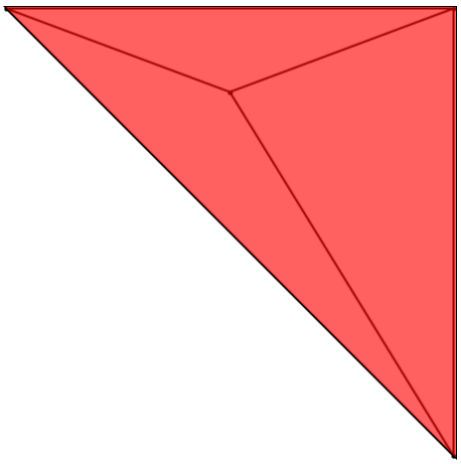
was mentioned in Section 3.3. The filtering based on intersection in the interpreter works by checking whether any new generated line intersects with all the present lines.



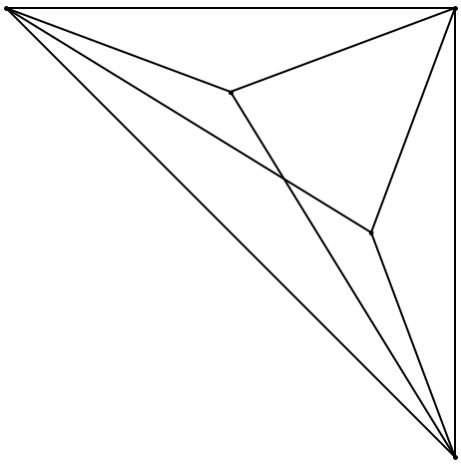
(a) Start (0 Rule applications)



(b) The triangle after applying the rule once.



(c) This time choose the original *wrong* triangle to split indicated in red.



(d) This leads to an intersection.

Figure 4.1: Applying the triangle inlay rule seen in Figure 3.3 a few times leads to self intersection of lines.

4.1.2 Filter Based On Transformation Properties

The Subshape Detection Algorithm 1 will match shapes which are rotated, translated, scaled and/or sheared. Although this is very powerful, not all of these operations are necessarily wanted. Maybe only uniformly scaling is allowed, so a square will only

match as a square and not as a rectangle or maybe the shape should not be rotated. The naive algorithm will collect all matches, regardless of scaling, translation, rotation and shearing. All those operations are encoded in a transformation matrix in homogeneous coordinates and it is possible to extract them by matrix decomposition. For example, if no rotation is allowed, extract the angle and check if it is bigger than zero and discard it if this condition evaluates to true. Or if uniform scaling is desired, extract scale x and y to check if those are equal. In any case, this makes it possible to mold the algorithm to conform to desired properties when matching shapes.

4.1.3 Filter Based On Bounding Box

When using a shape grammar it is often the case that images of a certain size (width, height) should be created. However, by simply running the algorithm numerous times, and thus changing the geometry of the final shape, it is not guaranteed that the geometry created will be restricted to the inside. This bounding box or arbitrary shape is called a *limiting shape* by Stiny et al. [9]. Restricting a shape grammar is also mentioned by Tapia [11], which should prevent the generation of very large or small shapes. Even though restricting a grammar is already mentioned in Tapia [11], it still requires additional user interaction to run the interpreter.
If a rule transformation applies new geometry evenly around a certain point, one could put the bounding box on the shape in a way that it compasses everything after the generation. So the rule fills a big enough area and then one can cut out an image from this area by using the given bounding box. Although this might work in certain cases, a rule could simply expand geometry in one direction, never filling a big enough area to cut an image out of it. Regarding this, it seems to be easier and more performant to restrict rule applications inside a bounding box because outside geometry will not be used. There are two ways leading to different textures:

1. Check every match if the points generated or translated from it (by β) are inside the bounding box
   By checking this, it is sure that no points will ever leave the bounding box. This is the go-to method if no lines should be cut in half by the bounding box. However, this leads to empty space between the borders of the image and the drawn geometry which might not be desired.

2. Check every match and only reject those that generate geometry outside of it.

4.1.4 Filter Based On Pixel Size

After transforming a shape several times the generated points could be very near to already present ones. This is especially true if a grammar operates recursively like the

rule in Figure 3.2. Any new points will move closer and closer together until they cannot be distinguished anymore, which is clearly present in Figure 1.3. Based on the pixel grid of an image, any rule application which produces points falling between the grid coordinates can be discarded.

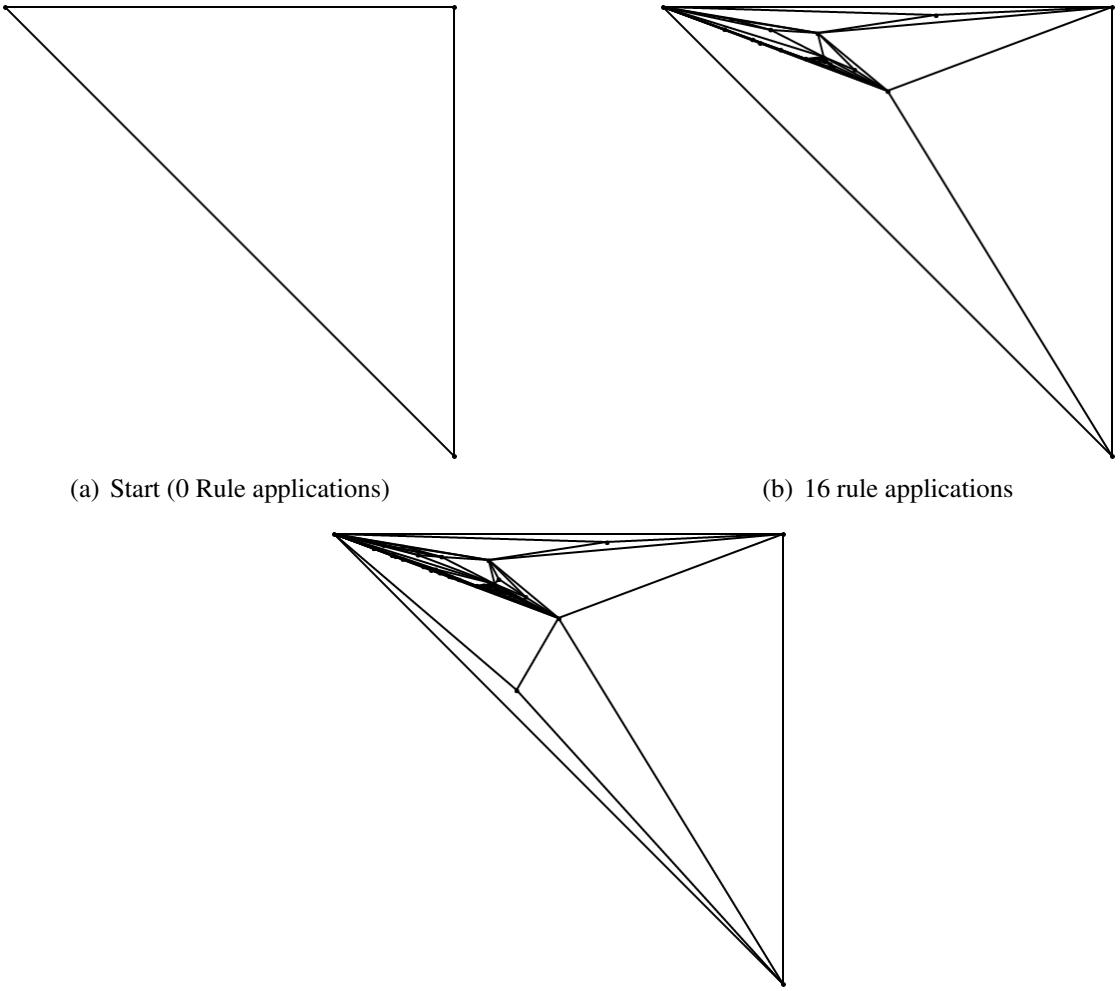
4.2 Match Selection Problem

Because a shape is made up of many points and lines, there usually are several subshapes inside. The subshape detection algorithm 1 outlined in the Section 3.3 will collect all these matches with relevant data. For example, it collects which points in α match to which other point in the target shape and also a transformation matrix used to get from points in α to γ. With all these different matches, one has to be chosen in order to apply the rule. Depending on the choice of subshape to use in each step of a rule application, different images can be produced. This section outlines several basic heuristics that can be applied given a rule and a shape to solve the match selection problem.
Selection Rules have also been mentioned by Stiny et al. [9], although very basic and restricted to simple grammars. The heuristics outlined here try to be broader and applicable for various rules.
Each heuristic will give each subshape match a score and then choose from all matches the one with the best score according to the specific heuristic. In related work such as by Krishnamurti [5] it was always assumed that a designer would choose which match to use for a rule step. However, it would be desirable to streamline the creation of textures by only setting input parameters at the beginning, and let the algorithm run and choose matches to use by itself. The methods presented in this section enable a more autonomous interpreter, which requires less user interaction. Possible rule selection strategies are presented below. The heuristics from Sections 4.2.1.1, 4.2.1.2, and 4.2.2 are currently supported, the others are briefly described.

4.2.1 Probability Based Heuristics

4.2.1.1 Random Choice

Given that we would like to apply the rule uniformly over the whole geometry instead of concentrating in one specific area, the obvious first choice might be a random selection. A short example shows why this is not recommended at least for some grammars.



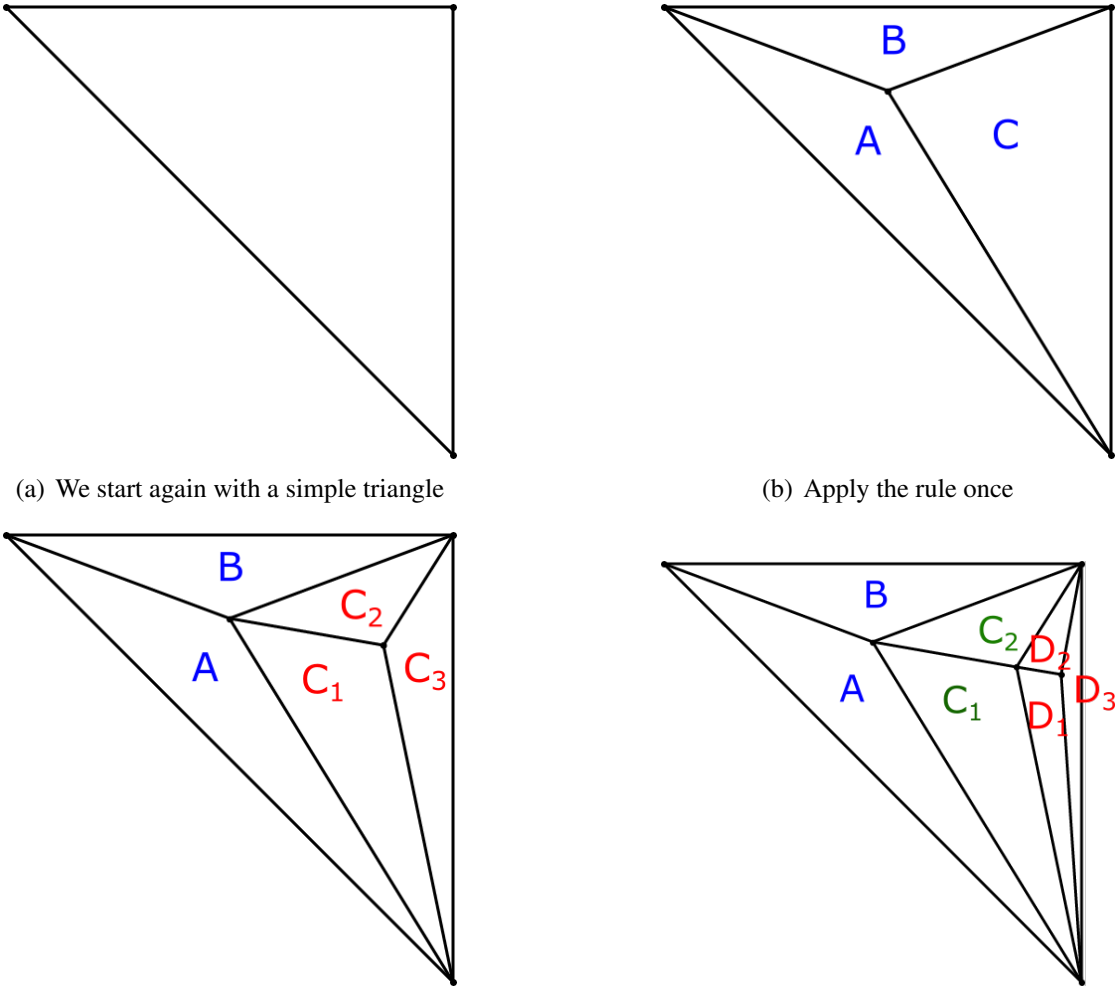
(a) Start (0 Rule applications)

(b) 16 rule applications

(c) 35 rule applications

Figure 4.2: Triangle Grammar applied randomly on a single triangle

To show why the rule only applies in a certain area we can consider Figure 4.3.



(a) We start again with a simple triangle

(b) Apply the rule once

(c) After two rule applications

(d) Apply the rule once more

Figure 4.3: Triangle inlay grammar applied with labelled triangle subshapes

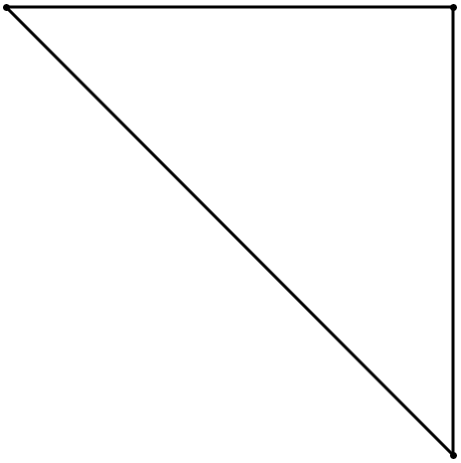
After applying the rule once we get three triangles A, B, and C as shown in Figure 4.3(b). A random choice will pick each of the three triangles with an equal probability of $P(A) = P(B) = P(C) = \frac{1}{3}$. Applying the rule once more leads to the triangles A, B, C_1, C_2, and C_3 as in Figure 4.3(d). The probability of picking triangles A, B randomly has now actually decreased to $P(A) = P(B) = \frac{1}{5}$ and choosing a smaller triangle will now happen with a probability of $P(C_1 \cup C_2 \cup C_3) = \frac{3}{5}$. This means the probability of picking an *older* triangle decreases while the probability of splitting a smaller triangle increases which leads to the situation in Figure 4.2 only expanding in one corner of the triangle. A random approach as in Figure 4.2 is clearly not desired for this type of rule.

4.2.1.2 Balanced Random

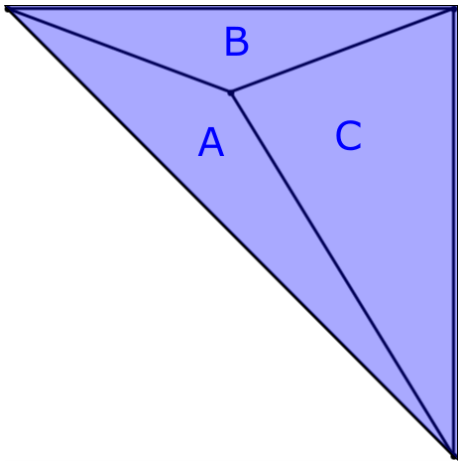
Given the problem of a Random Choice in 4.2.1.1 we can alter the selection heuristic a bit to factor out all the matches on a same shape (but in different configurations) to remove the skewing of probabilities against unsplitted triangles.

This is done by grouping together triangles which were created in the same step as shown in Figure 4.4. We first select a random group and then choose randomly from all triangles in the selected group. In the first two steps (Figures 4.4(a) and 4.4(b)) random selection happens as before. However, if we look at the triangles now in Figure 4.4(c) we see that there are two groups. Group 1 contains triangles A, B and group 2 triangles C_1, C_2, and C_3. In this case the probability of choosing group 1 is: $P(G_1) = \frac{1}{2}$
which leads to the probability of A, B:
$P(A) = P(B) = P(G_1) \cdot P(B|G_1) = \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$
In the next step in Figure 4.4(d):
$P(A) = P(G_1) \cdot P(A|G_1) = \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{6}$

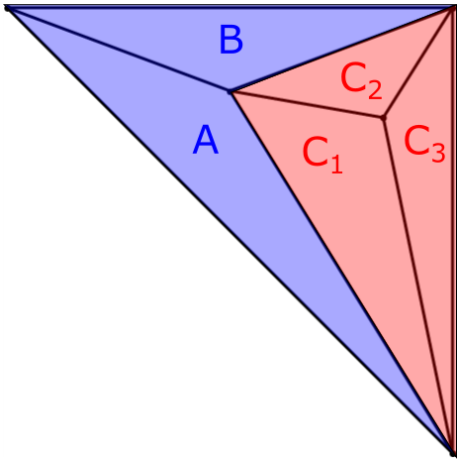
The probabilities of older triangles still decrease but less noticeable than the random selection heuristic and produces images such as in Figure 4.5. Even though this *balanced* random method seems to create a slightly more uniform image than 4.2.1.1 visible in Figure 4.5, it does not produce perfectly uniform splitted triangles. This strategy seems widely applicable, even more so if not perfectly uniform rule transformations are required.



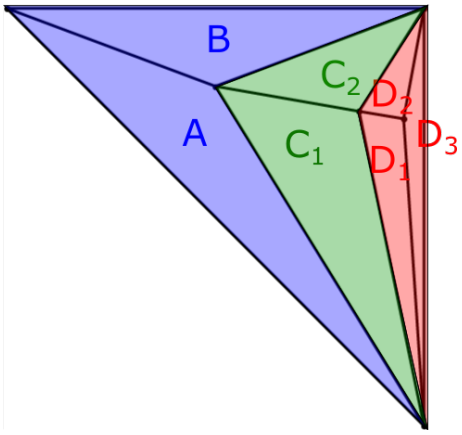
(a) We start again with a simple triangle



(b) Apply the rule once and group together triangles

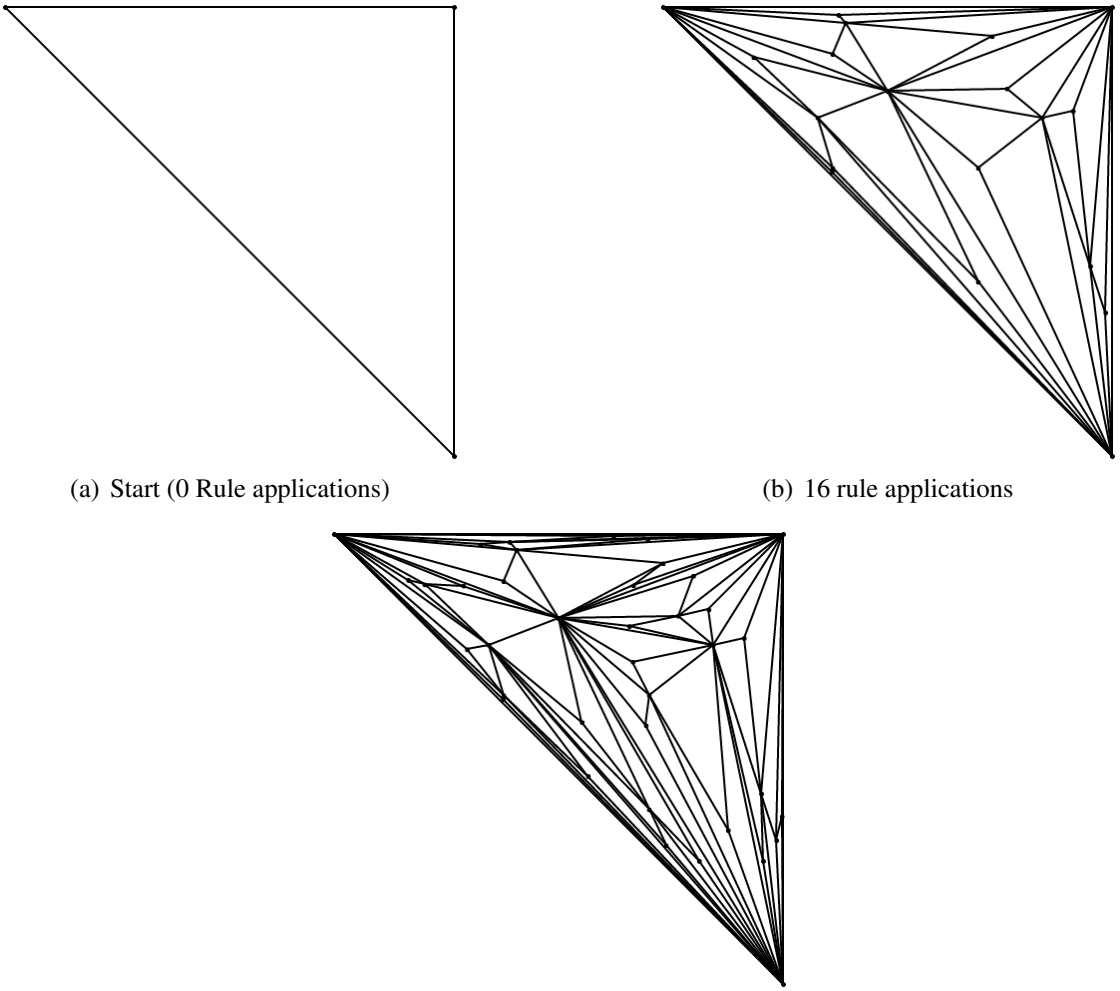


(c) After a new transformation group again triangles together which were created in the same step



(d) Apply the rule once more

Figure 4.4: Example how to group triangles with SGBalancedRandom heuristic



(a) Start (0 Rule applications)

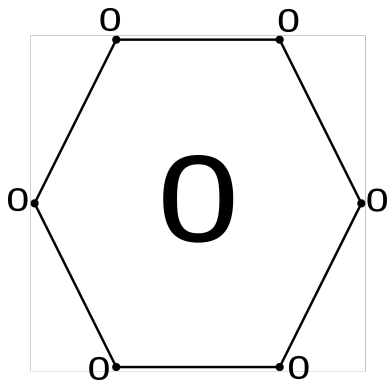
(b) 16 rule applications

(c) 35 rule applications

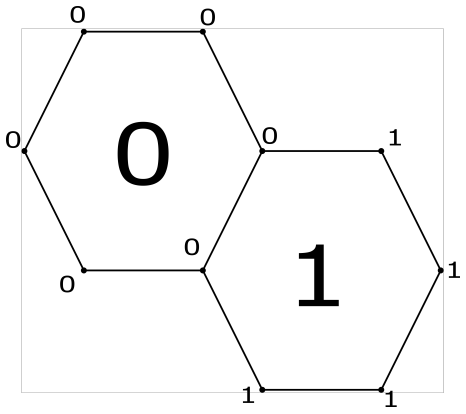
Figure 4.5: Triangle Grammar applied with balanced random heuristic on a single triangle

4.2.2 Age Based Heuristics

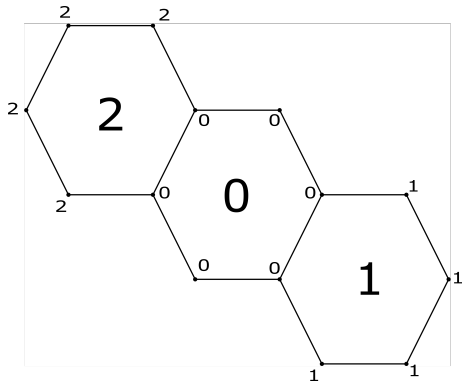
Every point in the shape γ has an attribute to store in which rule step it was created. With this information we can calculate the age of a point by comparing its creation time with the current number of steps already performed. Before any rule application, every point is assigned a creation time attribute of zero. Any new points created by the first rule step will have a creation time of one and this attribute increase in every rule step. With this setup we can score a subshape based on the age of its points.
For example, we can first assign shapes the maximum age of creation value of all its points as an attribute. Then we choose the shape that has the minimum of this assigned attribute. This guarantees that subshapes with older points will take precedence over newly created shapes or, on the contrary, scoring subshapes higher if they have more recently created points in them.
This heuristic is very promising for certain grammars like the grammar in Figure 3.4. For any tiling pattern generating rules, this heuristic is a good fit because it will prefer any subshapes near the center of the shape where points have a higher age than the points at the edge of the tiling. A small example how the selection by using the highest age of a shape works follows.
We start with an initial hexagon γ where every point has a rule creation time of zero in Figure 4.6. In the middle of every hexagon is the highest rule step attribute of its points, where a higher attribute means it was created in a later step. Therefore, if we want to pick the *oldest* hexagon, we choose one with the lowest inscribed number. We add in the first rule step another hexagon and give every new point a creation time of 1 in Figure 4.6(a). In the next step we choose the hexagon with the minimal creation time inscribed which is again the first square and leads to Figure 4.6(c). After filling up, we can only continue with a hexagon that has a minimal maximum age of 1 and therefore we have to expand evenly around the initial hexagon in Figure 4.6(d). Therefore it is possible to create complete tiling patterns as in Figure 4.7.



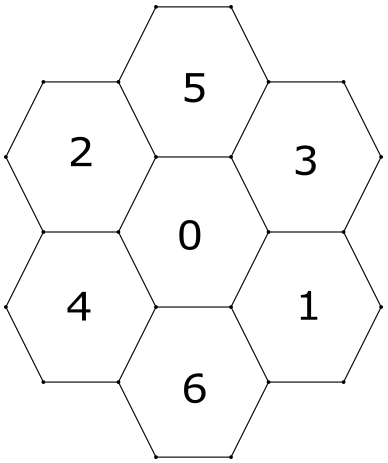
(a) Start (0 Rule applications)



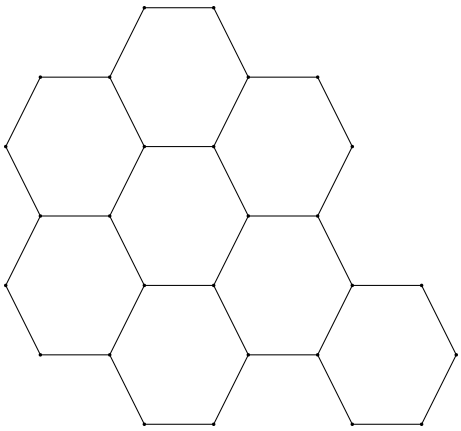
(b) The newer hexagon has a one inscribed



(c) After another rule step a second hexagon is new with an id of two

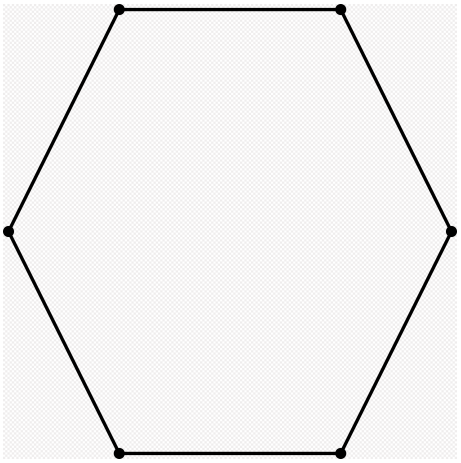


(d) In total 6 hexagons with increasing ids

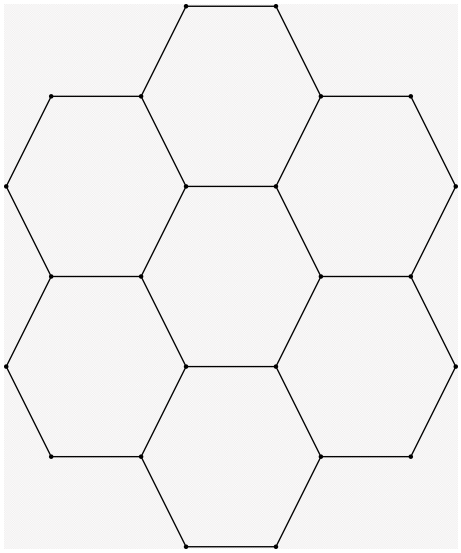


(e) Choose again 1 to expand the pattern outward

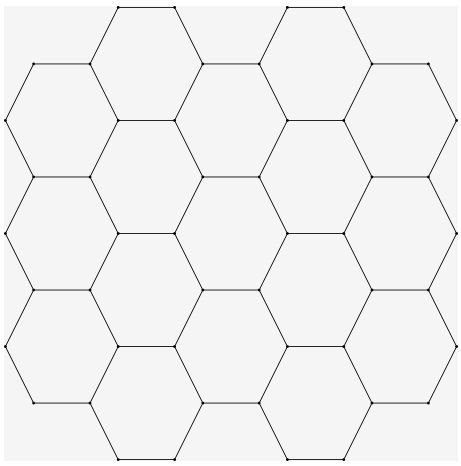
Figure 4.6: Example how to select hexagons based on when they were created



(a) Start (0 Rule applications)



(b) 6 rule applications



(c) 16 rule applications

Figure 4.7: Hexagon tiling grammar created with age selection, preferring subshapes with a higher age.

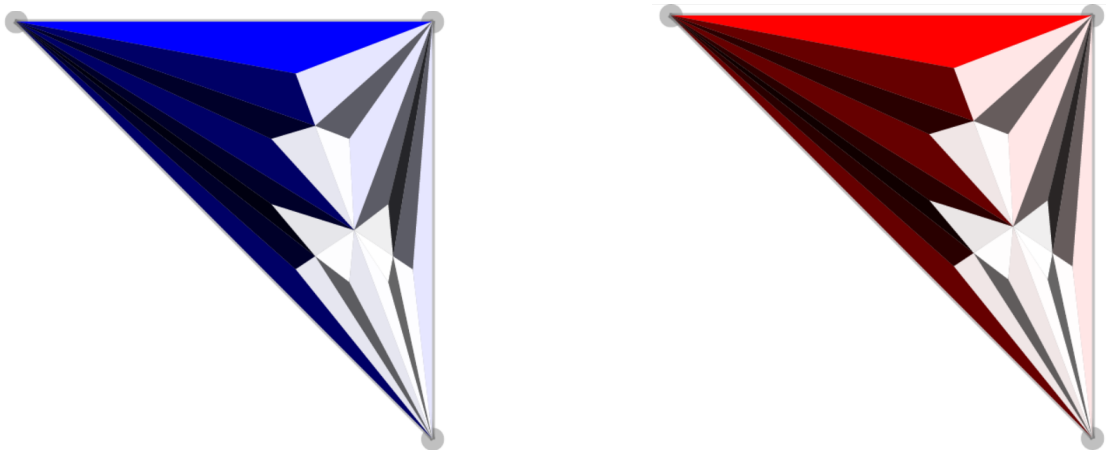
4.2.3 Shape Geometry Heuristics

The triangle inlay grammar in Figure 3.3 shows the distinct property that we would like to split the triangle having the biggest area, whereas the recursive square grammar from Figure 3.2 requires the interpreter to choose the smallest square. Therefore we can use the subshape size of a match for scoring, usually preferring a bigger size. Considering tiling grammars, every subshape has the same size and this method of scoring is not applicable. In case a subshape has no area due to being single lines or points, the convex hull can be used to attach a size to a subshape. Another simple strategy to score matches is to consider properties of a shape such as distances between points, line lengths, or vertex degree. Any of these properties could also be used for selecting subshape matches.

4.3 Coloring And Additional Data Tags

Because shape grammars only work on shape geometry, the produced images do not have colors and need to be colored manually after the generation steps. Stiny et al. [9] describe material specification having the form of painting rules. Each step of a rule adds a new shape which overlaps a previous one. Coloring rules were defined by the intersection or union of these shapes. Even though these coloring rules by set description can be used, they do not seem to be particularly useful for more than a handful of operations. Shapes cannot be described as a set if they are not closed. In addition set description gets quite unwieldy due to combinatorial explosion when describing combinations of intersections and unions between a number of shapes. Instead of defining material properties and colors depending on set description of areas, the primitives in a shape, its points and lines could be labelled directly for coloring. Stiny [8] mentions adding the additional label *weight* to points and lines describing line thickness or point radius. Stiny [8] also describe weights for solids, which can produce several tones when areas with different weights are drawn atop of each other.
In this implementation another approach is used:
Instead of defining more rules that operate on a finite set of chosen colors, every rule has a function that assigns colors to areas in β relative to the colors used in areas from α. This function creates the new color of the three new triangles relative to the color of the split triangle. Because this function can take any color as input instead of a fixed number of transformations, we can apply it to an arbitrarily colored starting shape. Figure 4.8 shows the result of such a color transformation given a starting triangle with such an arbitrary color, in this case either blue or red.

Although the current implementation of the interpreter does not store areas as faces or polygons as primitives, but instead only stores points and lines, a label defining the polygon area can still be added to the shape. When drawing, these polygon labels are



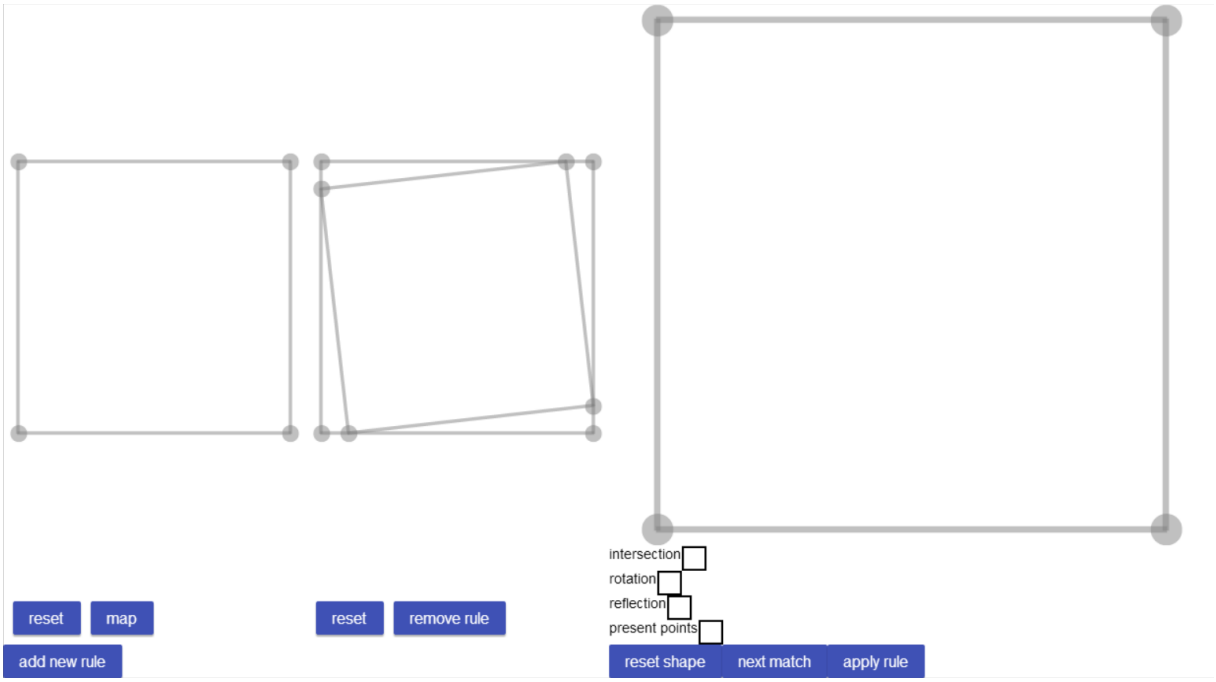
(a) Transformation of a blue colored triangle.

(b) Transformation of a red colored triangle

Figure 4.8: Shape grammar rules can work on any colored starting shape. The color transformations are defined relative to the input color.

extracted from the shape and drawn based on the properties described in the labels, e.g. its color.
A different approach also is described by Knight [3], defining a new type of grammar called a *Color Grammar* that can be used to create colored shapes. This might lead to even more color variations, either instead of or together with the function approach used in the implementation of the editor in this work.
Furthermore, there is newer work on so called *sortal grammars* described by Stouffs [10], where the same example as in Figure 3.2 was given, changing coloring between black and white. This might also be usable if one wants to add more coloring possibilities.
The shape itself can also contain additional data, either used for rendering or data used for other purposes with the shape. For instance, texture ids could be assigned to faces or other rendering properties can be distributed over a mesh by a subdividing shape grammars, which can then be used by a modern renderer to create a more realistic image instead of the more abstract ones currently made.

5
Editor and Interpreter



(a)

Figure 5.1: The editor with different areas to define rules and shapes. Rules can be applied to the γ shape on the right by pressing the *apply rule* button.

The main goal of the editor is to test the new subshape detection algorithm outlined in Section 3.3. It acts as the front end of the shape grammar interpreter and we can visually debug shapes with it. A rule is applied and the result of a rule transformation on the shape can be examined at every step. The rules and the starting shape can also be modified with the editor. This is more convenient than specifying shapes and rules by code. The editor has two main areas to draw the rule and the starting shape γ, which can be seen in Figure 5.2(a). Both areas allow us to add new points or remove existing ones. In addition, lines can also be removed or created by connecting two points. Figure 5.2(b) shows how we can set some of the filter parameters from Section 4.1 by clicking on the corresponding checkboxes. After defining the α and β shape of the rule on the left drawing area and γ on the right we can press the *apply rule* button in Figure 5.2(b) to transform the shape.
In order to prevent redefining the same shape every time, standard configurations can be loaded by code. We can open the editor with a previously set rule and starting shape γ, which can be defined with a small domain specific language (DSL). This DSL has been created in Pharo itself without parsing the source due to the expressive syntax of Smalltalk. A small excerpt how a shape is created can be seen in Listing 1.

```
shape := SGShapeBuilder new
   points: {#a -> (0@0). #b -> (1@0). #c -> (1@1)};
   lines: {#a -> #b. #b -> #c. #c -> #a};
   build.
```

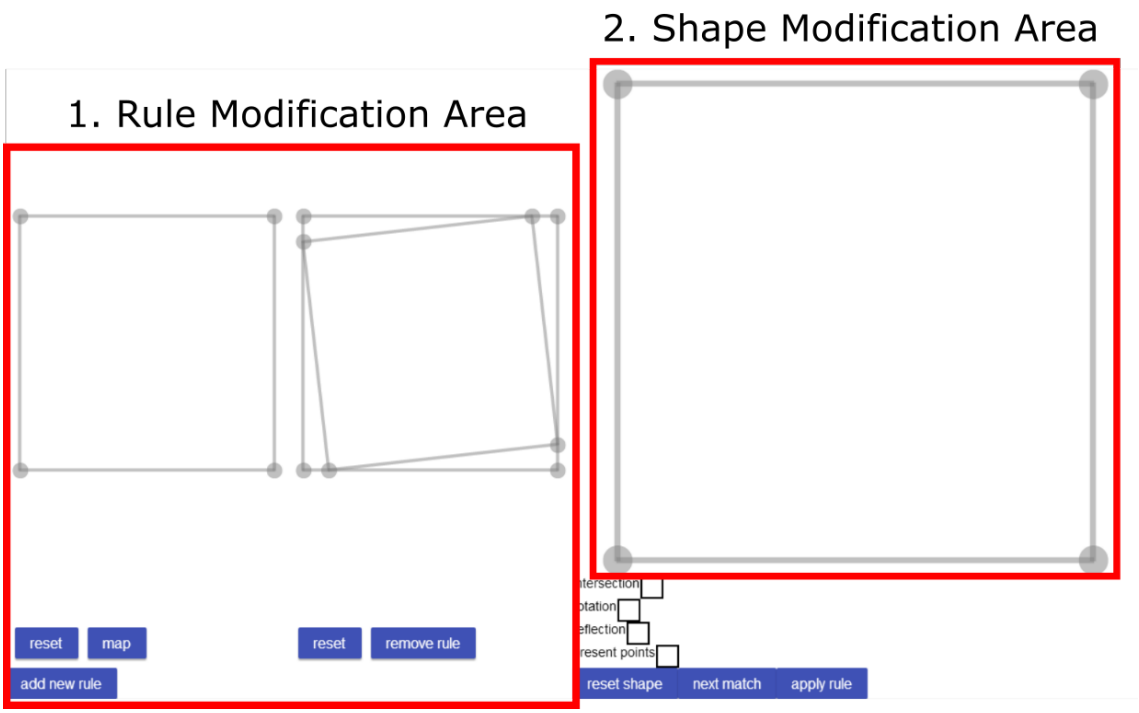
Listing 1: Create a triangle shape

Instead of using the editor, it is also possible to just run the interpreter with parameters provided in code without using a graphical interface. The Listing 2 shows a small example how we can run and create images.

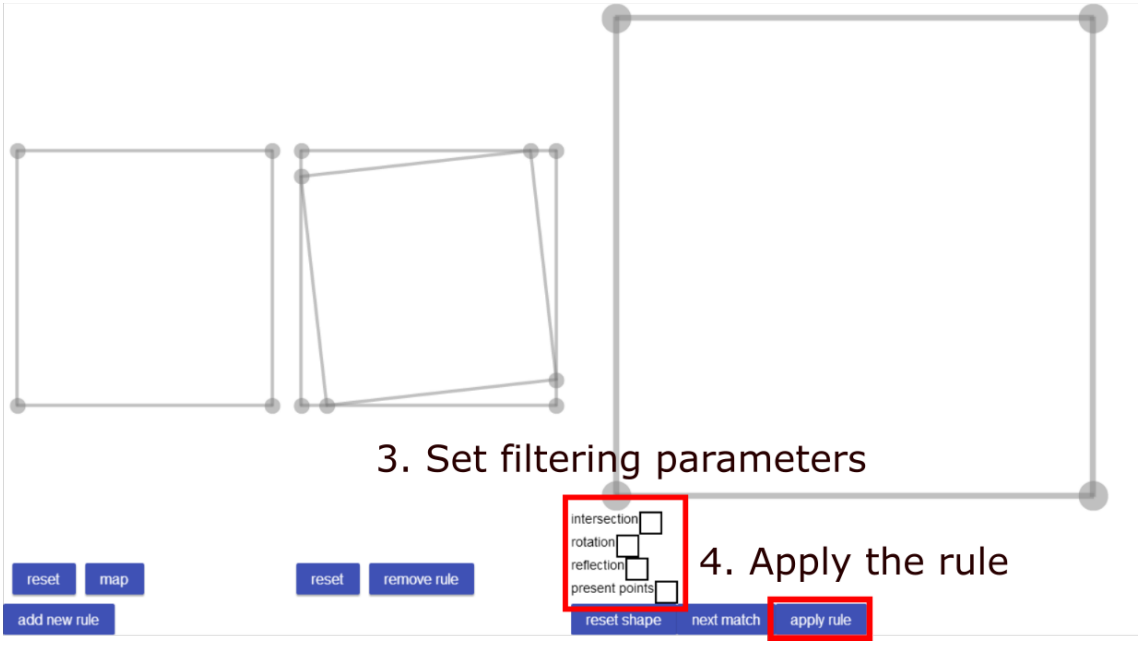
```
builder := SGImageBuilder new.
builder
   from: 1 to: 35 by: 5;
   background: Color white;
   config: (SampleConfigurations new triangleInlayConfig);
   filterIntersections;
   name: 'triangle_inlay_balanced';
   folder: self baseFolder;
   size: 500@500;
   selector: (SGBalancedSelector new);
   export.
```

Listing 2: Use the SGImageBuilder to run the interpreter and produce image output

We can specify how many times the rule is to be applied to a shape. In this case, we go from one application to 35 and output an image every five rule steps. The filtering parameters and match selection strategies from Section 4.2 can be set directly in code. This setup makes it possible to quickly test and prototype the interpreter and editor.



(a) Area to define a rule (1) and a starting shape (2)



(b) Set filtering parameters by clicking on a checkbox (3) and apply the rule (4)

Figure 5.2: Set the filtering parameters and apply the previously defined rule.

6 Evaluation

The novel subshape detection presented in Algorithm 1 Section 3.3 works for all grammars described in Section 3.2.1. However, only by examining these examples and without further proof it is not clear whether the algorithm is really equivalent to the one in Krishnamurti [4], though intuition indicates very strongly that it should be. In practice, the interpreter is usable to produce images such as in Figures 1.3, 4.5, and Figure 4.8. Therefore the method presented in Algorithm 1 is a usable alternative to the commonly used algorithm from Krishnamurti [4]. It is quite simple to implement with transformation matrices using homogeneous coordinates.
Even though we examined several selection strategies in 4.2, only some grammars such as in Figures 3.3, 3.2, and Figure 3.4 have been considered. More conclusions on running an interpreter independently without further user input could be drawn by analyzing grammars with different properties from the examined ones. The editor and the domain specific language used for defining shapes have greatly helped in finding defects concerning the interpreter. The selection strategies are used to create many rule transformations on a shape without user interaction. Images like in Figure 4.5(c), 4.2(c), and 1.3 with up to more than a hundred transformations would have been tedious to create by hand.

7 Conclusion

We build a shape grammar interpreter and proof-of-concept editor given Algorithm 1 proposed in Section 3.3. The new subshape detection method is based on comparison of two shapes α and γ, each having their own coordinate systems, in a common local system. The complete transformation from one shape to the other can be obtained by chaining two transformations together. First, by going from α coordinates to a local system, then by going from this local system into the coordinate system of γ. The interpreter is robust and can be used to create various images, either uncolored as in Figures 4.2, and 4.7 or colored as in Figure 4.8. We create these images without further user input, apart from supplying parameters at the beginning, by using match selection and filtering strategies. The interpreter uses match selection and filtering to choose by itself where to apply a rule and therefore does not require manual user input while running.
Further work could be done regarding how multiple rules in a grammar play together. In this case they could be applied either in sequence or the interpreter could assign probabilities to them for choosing randomly. In addition, there might be many more useful selection strategies than mentioned in the Section 4.2. This heavily depends on the intent of the user and also which rules are chosen to transform the starting shape. Only a sample of rules have been considered in Section 3.2.1. Other strategies might be more applicable when choosing a different set of rules.
By building an interpreter and proof of concept editor, we have shown that the algorithm 1 can be used as an alternative to the one described in Krishnamurti [4]. With help of the subshape detection algorithm and the strategies for the subshape selection it is possible to build a self-running shape grammar interpreter.

8 Anleitung zu wissenschaftlichen Arbeiten

This chapter describes the implementation details and usage of the shape grammar interpreter and editor of the thesis. The source code of the implementation is hosted on github [12]. The interpreter is programmed in Pharo (Smalltalk)[1] and the editor uses the low level UI framework Bloc[2] to define GUI elements and render them to the screen. Even though Bloc is still in development, it could already be used effectively for the purpose of creating a usable editor as the front-end for the shape grammar interpreter. The backend is the interpreter that can perform subshape detection and transform shapes with given shape grammar rules. A video showing the editor is available on the repository [12].

8.1 Interpreter

8.1.1 Class structure

Shape grammars have several key components that we can model directly in the class structure. First, we need to define two-dimensional geometric shapes.
Points and vectors are both defined by the `SGVector` class. Each `SGLine` has a reference to two `SGVector` endpoints. The `SGShape` class as shown in Figure 8.1 defines a shape by having a collection of points stored in `SGVectors` and lines as `SGLines`. We also

[1] `https://pharo.org/`
[2] `https://github.com/pharo-graphics/Bloc`

have the possibility to add and remove points or lines and to query whether a point or line already belongs to a shape. A rule has two shapes α and β. In addition, a rule possesses

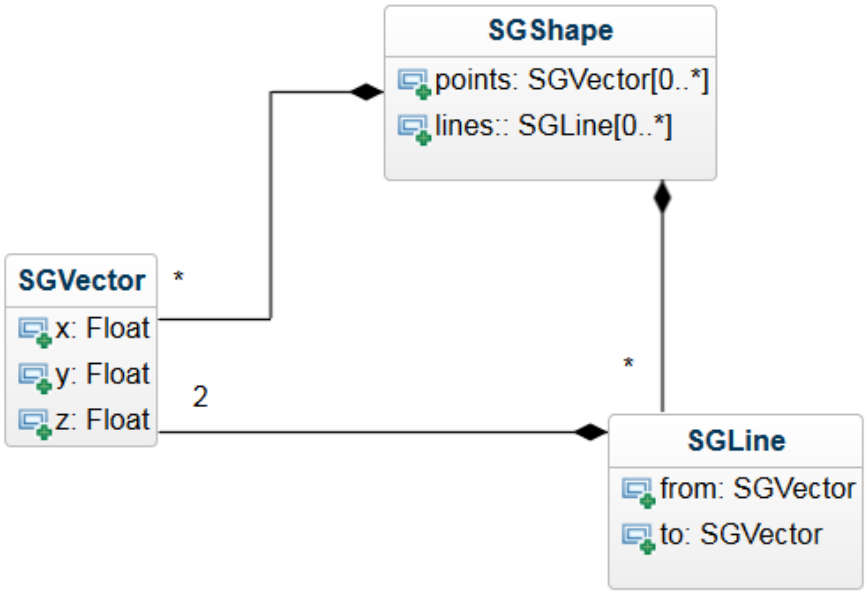


Figure 8.1: The `SGShape`, `SGVector`, and `SGLine` classes. They define a shape with points and lines.

a dictionary, mapping points from α to β. Such a rule is shown in Figure 8.2. Matrix transformations as applied by the `SGMatrix` class work on vectors like usual but it should be noted that they use homogeneous coordinates for 2D geometric transformations. After doing subshape detection of the shape α inside a target shape γ all subshapes with relevant data are stored in separate `SGShapeMatch` instances visible in Figure 8.3.

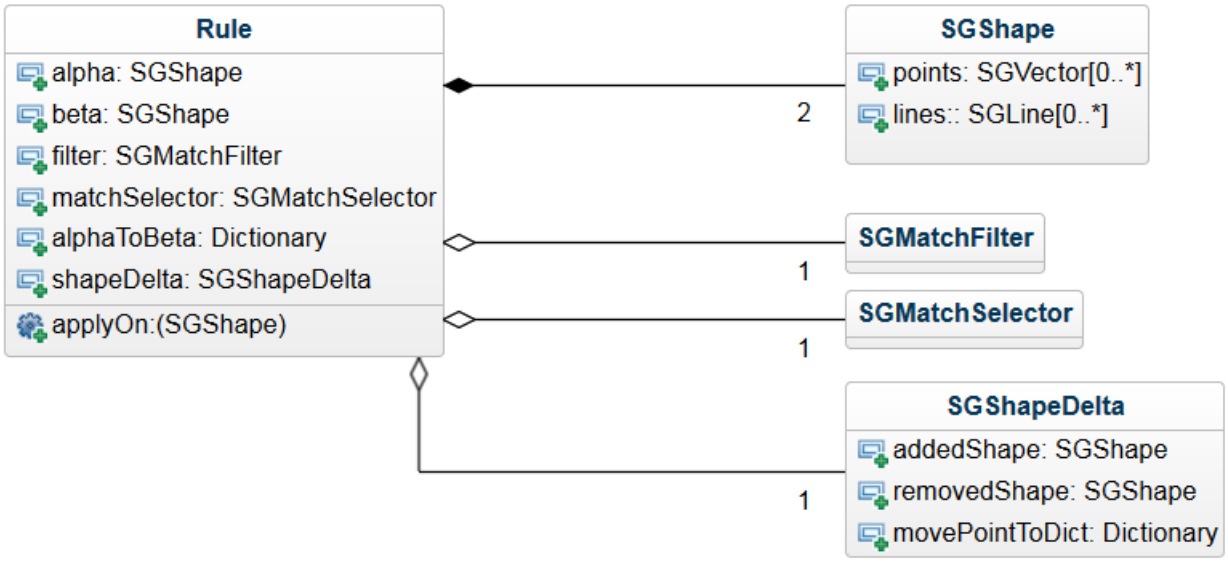


Figure 8.2: The `SGRule` class. It has an α and a β `SGShape`. The transformation from α to β is stored in the `SGShapeDelta` class.

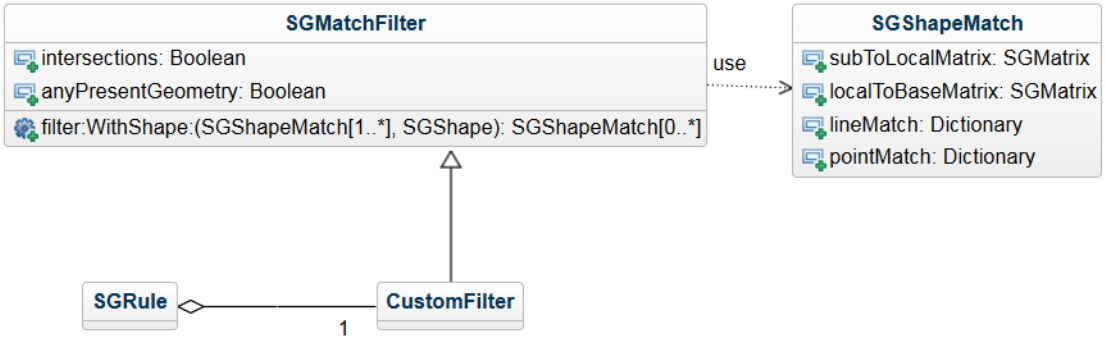


Figure 8.3: A `SGMatchFilter` filters a list of subshape matches. The subshape matches are modeled with the `SGShapeMatch` class that stores additional data about a subshape match.

For example, the transformation matrix going from α to a local coordinate system and then to the coordinate system in γ is stored in two separate matrices `subToLocalMatrix` and `localToBaseMatrix`. Two dictionaries `lineMap` and `alphaToBase` store the mappings from points and lines in α to points and lines in γ.

The `SGMatchFilter` class filters such a list of shape matches to remove unwanted ones. A rule has such a filter and we can create custom filters by extending from the `SGMatchFilter` as seen in Figure 8.3.

In addition to a filter, a rule also has a `SGMatchSelector` in order to select from possible subshape matches a suitable one to apply its shape transformation. Figure 8.4 show three implemented match selectors. A match selector receives a list of `SGShapeMatch` classes and returns a single selected one. More selectors can be added by extending the `SGMatchSelector` base class and then adding the custom selector to the rule.

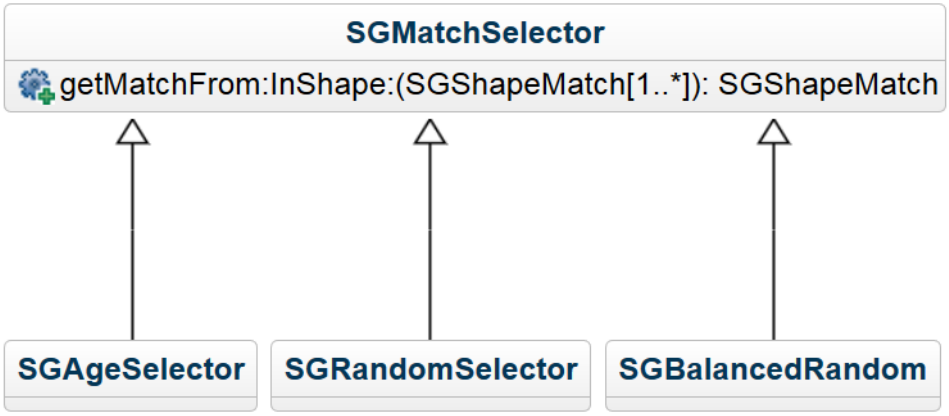


Figure 8.4: A few match selectors that extend the `SGMatchSelector` class. Custom selectors can be implemented by extending this base class.

8.1.2 Shape Transformation

There are five different operations performed on the shape in the implemented interpreter:

1. Remove Points
2. Remove Lines
3. Add Points
4. Add Lines
5. Translate existing points

A change from one shape to another by these five operations is encapsulated in a `SGShapeDelta` class.
A `SGShapeDelta` as seen in Figure 8.2 contains the *Add Points* and *Add Lines* operation in a `SGShape` instance variable called `addedShape`. The *Remove Points* and *Remove Lines* operations are stored in another `SGShape` variable called `removedShape`. In addition, the *Translate existing points* is stored in a dictionary that maps a point to its new position. If we want to apply a transformation on a target shape γ we can just add the `addedShape`'s geometry to γ and remove the `removedShape` from it. After this we just move the points in γ as defined by the `movePointTo` dictionary.
Also, it is very easy to define the reverse operation from a given `SGShapeDelta`. Simply exchange the added and removed shapes position and we set the `movePointToDict` mapping to the old point position instead of the new one stored in the original shape delta.

The `SGShapeDelta` class is also used in the rule representation. A rule has two shapes α and β and we change the geometry between these two shapes again with the same operations. Therefore a shape delta can be constructed to define the relative change between α and β again in a `SGShapeDelta` class.

In the end the interpreter has two components. A starting shape on which the rule is applied and a grammar containing several rules. This is shown in Figure 8.5.

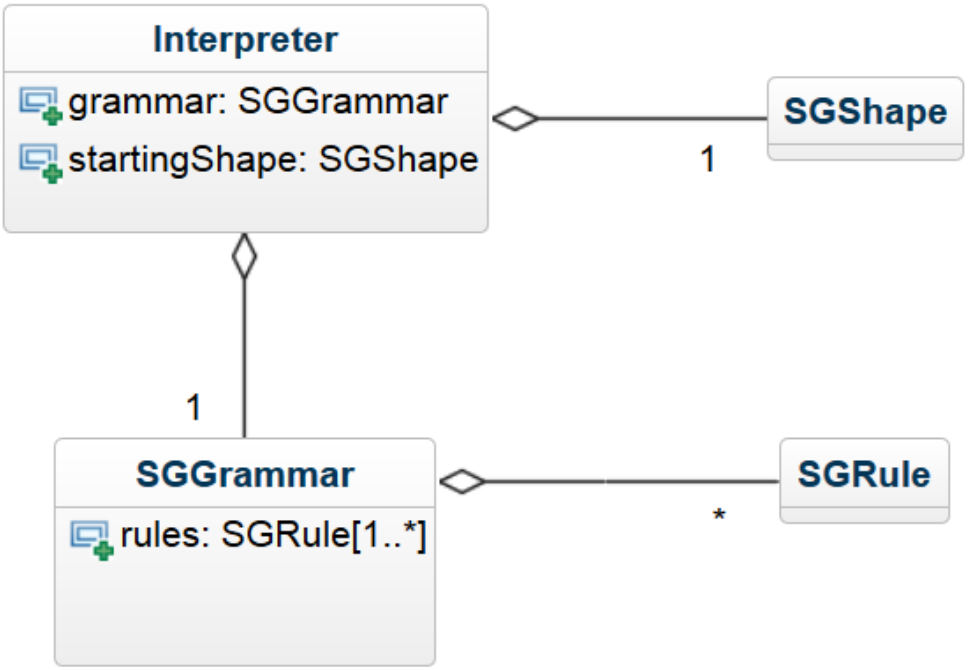


Figure 8.5: The interpreter

8.2 Domain Specific Language

In order to define shapes and other parameters for the interpreter a compact domain specific language (DSL) has been developed in Pharo (Smalltalk). This DSL is used to create shapes, rules, and grammars and is easier to use than defining them in code. In addition, the `SGImageBuilder` class can be used to create a series of images given a starting shape and a grammar.

8.2.1 Shape Creation

```
shape := SGShapeBuilder new
points: {#a -> (0@0). #b -> (1@0). #c -> (1@1)};
lines: {#a -> #b. #b -> #c. #c -> #a};
build.
```

Listing 3: Create a shape with manually specified labels.

```
shape := SGShapeBuilder new
points: {};
lines: {(0@0) -> (1@0). (1@0) -> (1@1). (1@1) -> (0@0)};
build.
```

Listing 4: Create a shape by specifying points inside the line declarations.

```
shape := SGShapeBuilder new
points: {(0@0). (1@0). (1@1)};
lines: {1 -> 2. 2 -> 3. 3 -> 1};
build.
```

Listing 5: Create a shape where ids are assigned automatically.

```
shape := SGShapeBuilder new
points: {#a -> (0@0). #b -> (1@0)};
lines: {#a -> (1@0). #b -> (1@1). 1 -> #a};
build.
```

Listing 6: Create a shape with mixed id declarations.

```
shapeBuilder := SGShapeBuilder new.
shapeBuilder point: (#a -> (0@0)).
shapeBuilder point: (#b -> (1@0)).
shapeBuilder point: (#c -> (1@1)).
shapeBuilder line: (#a -> #b).
shapeBuilder line: (#b -> #c).
shapeBuilder line: (#c -> #a).
build.
```

Listing 7: Create a shape by specifying points and lines each by a separate message sent to the shape builder.

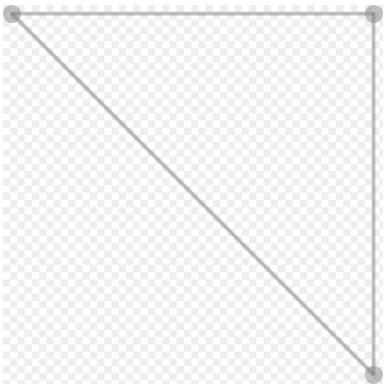


Figure 8.6: The triangle that is created by the `SGShapeBuilder` above.

A shape has points and lines. Points are passed as an array $\{\}$ where each entry is separated from the other by a full stop. Every entry has the form $\langle id \rangle \rightarrow (x@y)$ where the $\langle id \rangle$ can be any object to identify a point. For example numbers can be used as an id, any objects in general or as in the example in Listing 3 Smalltalk symbols. The position will be specified in $(x@y)$ where x and y are the coordinates. The id can also be omitted and the points will be automatically assigned a number which is incremented for every point starting at index 1 as in Listing 5. We can also specify points directly in the line declaration part as shown in Listing 4.
Lines are also passed as an array $\{\}$ and have the form:
$\langle id_1 \rangle \rightarrow \langle id_2 \rangle$
$\langle id_1 \rangle \rightarrow (x@y)$
$(x@y) \rightarrow \langle id_2 \rangle$
$(x_1@y_1) \rightarrow (x_2@y_2)$
where id_1 specifies the id of the starting point and id_2 the id of the ending point of a line. If a point $(x@y)$ is given as the start or endpoint instead of an id, the list of points is browsed to see whether a point with the given coordinates already exists. If not, the point is added with the next available id. Therefore it is possible to simply specify lines and

the corresponding points are added automatically as in Listing 6. Listing 7 shows how to declare points and lines separately instead of passing all at the same time in arrays.

8.2.2 Rule Creation

A rule consists of the shapes α and β. In addition, a mapping has to be specified from points in α to points in β. Usually, points in α map to points with the same coordinates in β. If the coordinates from α to β differ it means the point is moved when the rule is applied. A rule can be defined by using the DSL as seen in Listing 8.

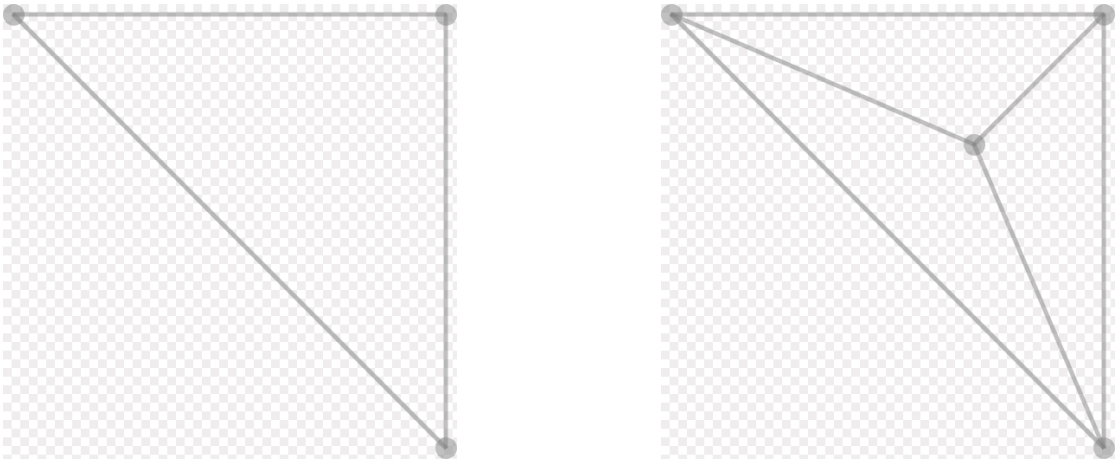
```
ruleBuilder := SGRuleBuilder new.
ruleBuilder alpha
points: {(#a -> (0 @ 0)). (#b -> (1 @ 0)). (#c -> (1 @ 1))}.
ruleBuilder alpha
lines: {(#a -> #b). (#b -> #c). (#c -> #a)}.
ruleBuilder beta
points: {(#a -> (0 @ 0)). (#b -> (1 @ 0)).
(#c -> (1 @ 1)). (#d -> (0.7 @ 0.3))}.
ruleBuilder beta
lines: {(#a -> #b). (#b -> #c). (#c -> #a).
(#a -> #d). (#b -> #d). (#c -> #d)}.
ruleBuilder map: {(#a -> #a). (#b -> #b). (#c -> #c)}.
rule := ruleBuilder build.
```

Listing 8: Create a rule

```
ruleBuilder := SGRuleBuilder new.
ruleBuilder alpha
points: {(#a -> (0 @ 0)). (#b -> (1 @ 0)). (#c -> (1 @ 1))}.
"... add some lines for alpha"
ruleBuilder beta
points: {(#a -> (0 @ 0)). (#b -> (1 @ 0)).
(#c -> (1 @ 1)). (#d -> (0.7 @ 0.3))}.
"... add some lines for beta"
ruleBuilder automap.
rule := ruleBuilder build.
```

Listing 9: Create a rule where points with the same coordinates are automatically mapped.

The α and β shapes can be created according to section 8.2.1. The `SGRuleBuilder#map` message requires an array $\{\}$ of ids in the form:
$id_\alpha \rightarrow id_\beta$
where id_α is a point id in the α shape and id_β is point id in the β shape.
A point mapping from alpha to beta can also be written as:
$(x_\alpha @ y_\alpha) \rightarrow (x_\beta @ y_\beta)$



(a) α shape of rule

(b) β shape of rule

Figure 8.7: The rule defined by the listings above.

If a point is specified instead of an id, it will be inferred based on the coordinates if possible. Instead of specifying all mappings by hand, the message `SGRuleBuilder#automap` can be used, which means that points from α to β with the same x and y coordinates are associated as in Listing 9. Figure 8.7 shows the visual representation of the rules defined in Listing 8 and 9.

8.2.3 Grammars and Configurations

```
grammarBuilder := SGGrammarBuilder new.
grammBuilder name: 'A simple Grammar'.
r1 := grammarBuilder newRule.
"... define rules as before"
"add a new rule by accessing a new SGRuleBuilder"
r2 := grammarBuilder newRule.
"..."
grammar := grammarBuilder build.
```

Listing 10: Creating a `SGGrammar`

A grammar in the implementation is just a set of rules. Listing 10 shows how a grammar is created with the `SGGrammarBuilder` class. Each rule can be specified as outlined in section 8.2.2 and the grammar can be named.
In addition to a grammar, the interpreter also needs a starting shape. Both, the grammar and the starting shape, are encapsulated in an `SGConfiguration`, which can be created as shown in Figure 11.

```
configBuilder := SGConfigurationBuilder new.
grammarBuilder := configBuilder grammar.
"... specify grammar"
shapeBuilder := configBuilder shape.
"... specify the starting shape"
configuration := configBuilder build.
```

Listing 11: Creating a `SGConfiguration`

8.2.4 Parameter Specification

It is possible to modify the selection process of the interpreter. In a first step the interpreter filters out unwanted subshape matches where a rule could be applied. This is configured with the `SGMatchFilter`, which is shown in Listing 12.

```
ruleBuilder := SGRuleBuilder new.
ruleBuilder filter: (SGMatchFilter new
intersections: true;
anyPresentGeometry: true;
yourself).
... "define rest of the rule"
```

Listing 12: Filtering options

Currently following filter options are available:

1. Filter *intersection*: Filters a rule application if it introduces a line that intersects with an already present line.
2. Filter *anyPresentGeometry*: If a rule would create a point or line that already exists at this position then the specific rule application is also filtered out.

By default no filters are applied.

In a second step the interpreter has to choose a subshape match where the rule should be applied. This is configurable by choosing a `SGMatchSelector` from several selection strategies:

```
ruleBuilder := SGRuleBuilder new.
ruleBuilder matchSelector: (SGBalancedSelector new).
```

Listing 13: Use of the `SGBalancedSelector`

```
ruleBuilder := SGRuleBuilder new.
ruleBuilder matchSelector: (SGRandomSelector new).
```

Listing 14: Use of the `SGRandomSelector`

```
ruleBuilder := SGRuleBuilder new.
ruleBuilder matchSelector: (SGDegreeSelector new).
```

Listing 15: Use of the `SGDegreeSelector`

Any selector which extends the general `SGMatchSelector` interface can be added to a rule. Listings 13, 14, and Listing 15 show three implemented selectors. This way it is also possible to implement custom selectors and set them to a rule without modifying the codebase itself.

8.2.5 Image Generation

In order to export `SGShapes` one can use two methods as shown in Listing 16 and 17.

```
shape := SGShapeBuilder new.
"... define your shape"
"store it"
shapeElement := SGShapeElement shape: shape.
shapeElement asSpartaForm writePNGFileNamed: 'filename.png'
```

Listing 16: Use of the `SGShapeElement`, which extends `BlElement`.

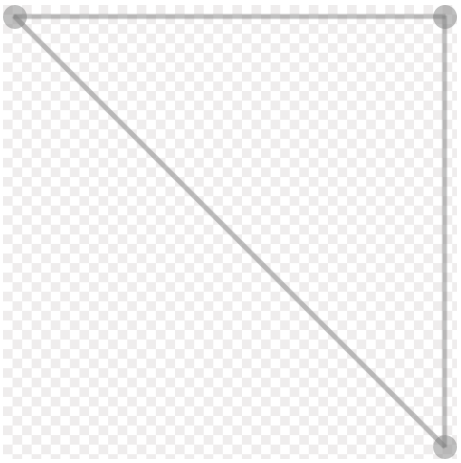
```
builder := SGImageBuilder new.
builder from: 1 to: 35 by: 5;
config: (SampleConfigurations new triangleInlayConfig );
filterIntersections;
name: 'triangle_inlay_balanced';
folder: self baseFolder;
size: 500@500;
selector: (SGBalancedSelector new);
export.
```

Listing 17: Use of the `SGImageBuilder`

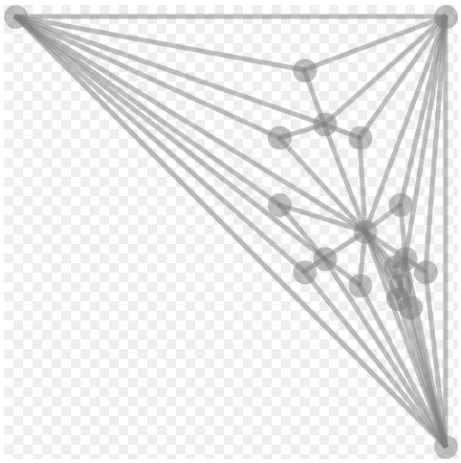
Listing 16 shows how to store a shape manually with a method that is implemented in the Bloc library. Another approach is to use the `SGImageBuilder` class as in Listing 17, which has several options to configure the image creation:

- from: $\langle x \rangle$ to: $\langle y \rangle$ by: $\langle z \rangle$
  The rule is applied $\langle y \rangle$ times to the shape. Every $\langle z \rangle$ step an image is taken in the range from $\langle x \rangle$ to $\langle y \rangle$. This allows to create several images each created after every rule step. In addition the first and the last image are always stored by default.
- config: - Defines a starting shape and a grammar as explained in Section 8.2.3.
- name: - Describes the name of the image.
- folder: - The folder in which the images reside after the generation.
- size: - Size of the images created.
- selector: - Convenience function to specify a match selector. This can also be achieved by creating a selector and storing it in a rule as in Section 8.2.4.
- filterIntersections - Another convenience function to specify some filters as in Section 8.2.4.
- export - Finally start running the grammar and store the generated images.

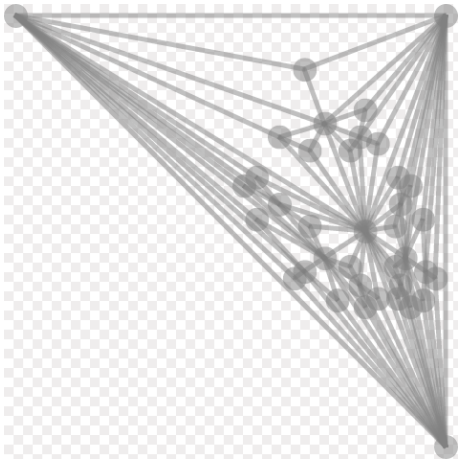
Figure 8.8 shows three images created by the `SGImageBuilder` class.



(a) Start (0 Rule applications)



(b) 16 rule applications



(c) 35 rule applications

Figure 8.8: A few images that were created by the `SGImagebuilder`.

8.2.6 Image Creation Infrastructure In Pharo

A lot of images were created either for use in the thesis directly or to test out certain properties while developing the different filters and selectors. In order to streamline the process of image creation and to update the images in case the editor or interpreter changed, handy features of the Pharo environment were used. All image examples were implemented as a distinct message in the `SGImageExamples` class. Every message of this kind uses the `SGImageBuilder` to create and save an image in a specified base folder, in this case the image folder of the thesis. An example of such a message is shown in Listing 18.

```
triangleInlayBalanced
<script: 'SGImageExamples new triangleInlayBalanced'>
| builder |
builder := SGImageBuilder new.
builder from: 1 to: 35 by: 5;
config: (SampleConfigurations new triangleInlayConfig );
filterIntersections;
pointColour: Color black;
pointSize: 5;
lineColour: Color black;
lineWidth: 2;
background: Color white;
name: 'triangle_inlay_balanced';
folder: self baseFolder;
size: 500@500;
selector: (SGBalancedSelector new);
export
```

Listing 18: A message to create a series of images with the pragma ⟨script:⟩

A message to create an example image is annotated with the pragma ⟨script: 'SGImage-Examples new messageName'⟩ which allows you to run the message directly from the system browser as shown in Figure 8.9. Listing 18 shows the code of such a message that can be run from the system browser.

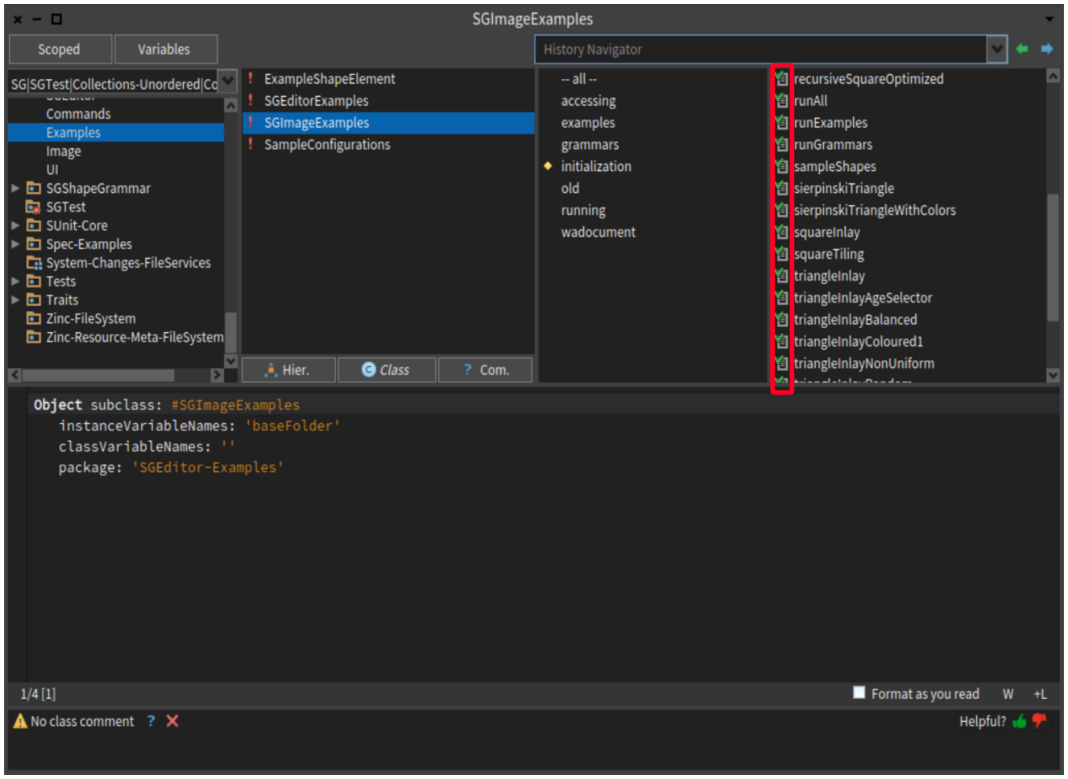


Figure 8.9: Run any message to create an image directly from the system browser.

In addition Pharo has some other nice properties that can help debug generated shapes and rules. By using the pragma ⟨gtExample⟩ we can also inspect a shape from the system browser, which can be seen in Figure 8.10. Figure 8.11 shows the shape if we inspect it.

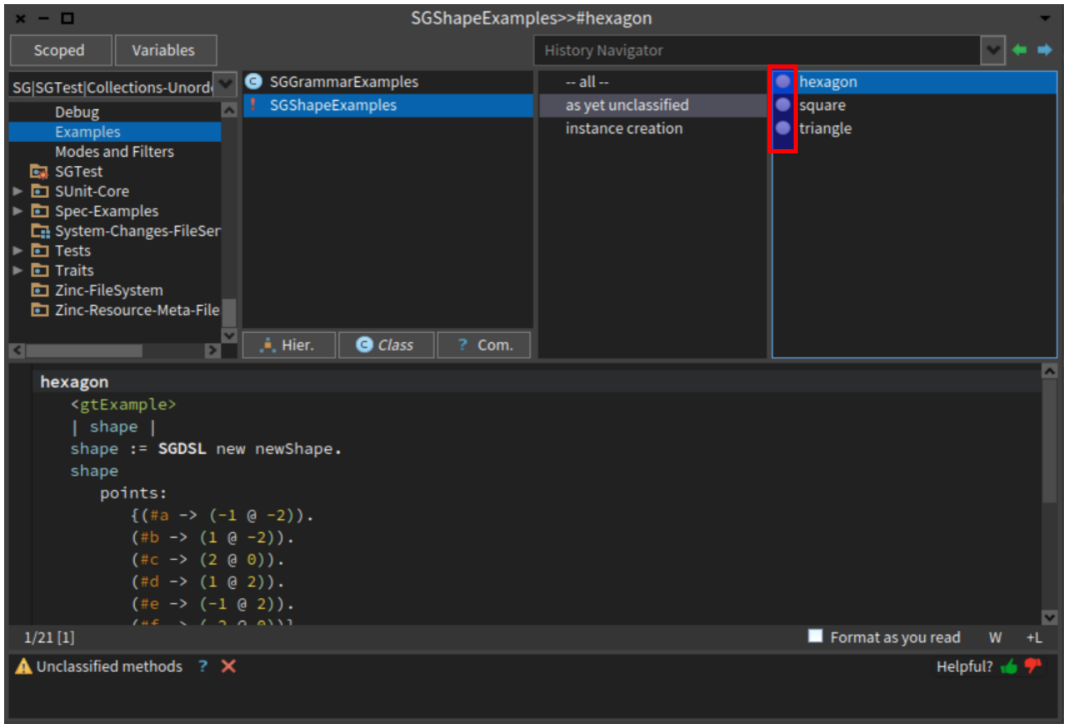


Figure 8.10: Debug the shapes from the system browser.

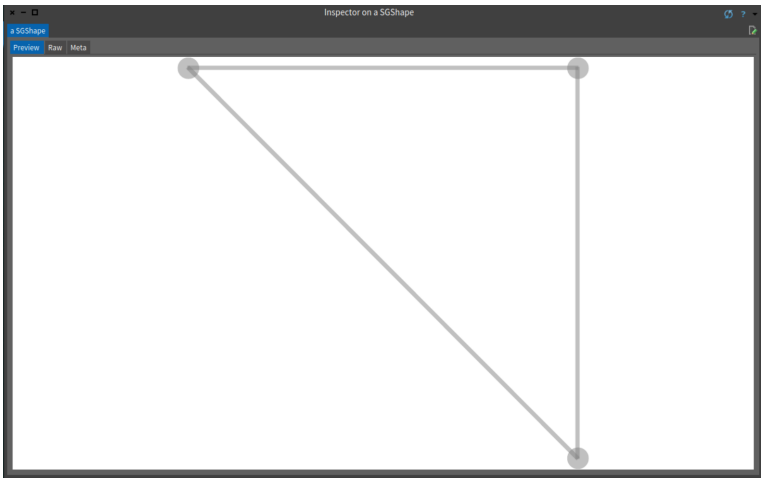


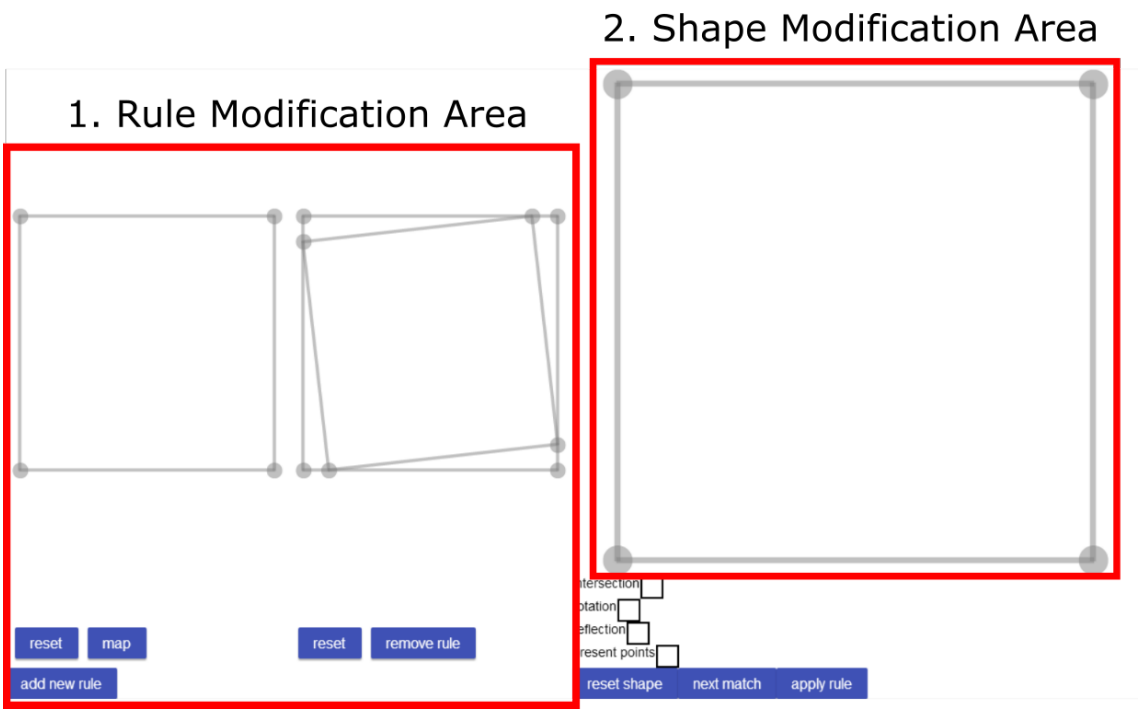
Figure 8.11: View of the selected shape.

8.3 Editor

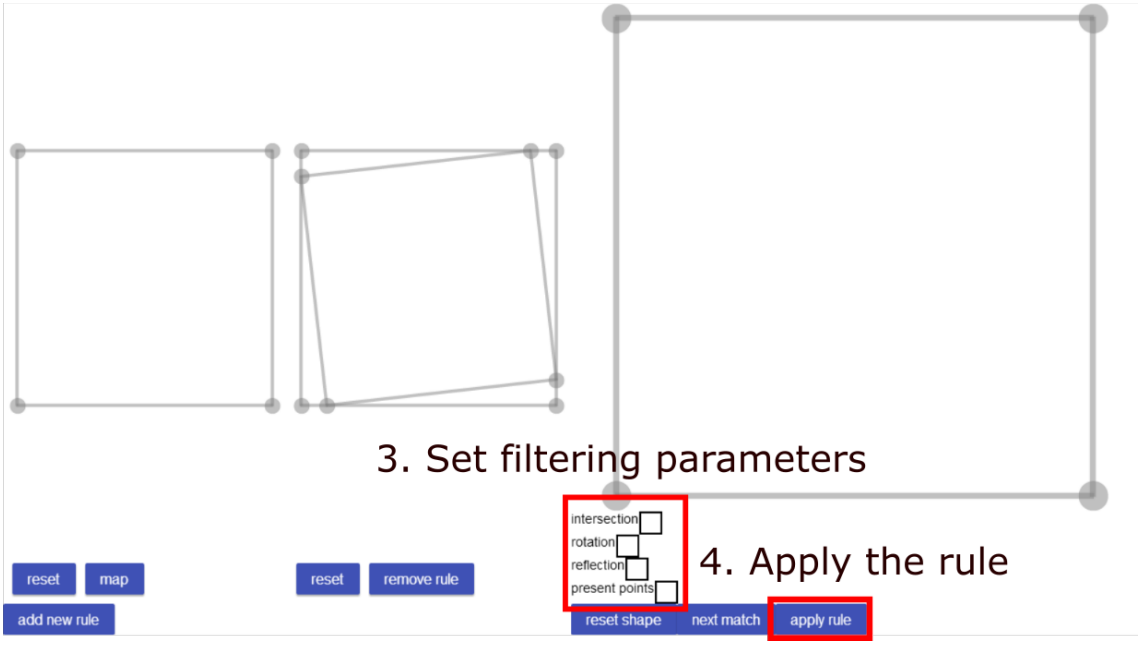
8.3.1 Basic usage

The editor has two main areas. First, we can edit the points and lines of a rule by drawing in the rule area visible in Figure 8.12(a). A rule has two shapes α and β of which both can be manipulated separately. We can add a point by clicking somewhere on the shape. A line can be added by selecting two different points and pressing ctrl. To remove a line, simply press the D key when both endpoints are selected. A point can be deleted by selecting it first and then pressing the D key. The target shape γ in Figure 8.12(a) on the right can be modified the same way.

After defining α, β, and γ we can select filtering parameters as shown in Figure 8.12(b) by clicking on the checkboxes.
Then we can apply the rule by clicking on the *apply rule* button in Figure 8.12(b). The γ shape on the right will be transformed and the changes are directly visible.



(a) Area to define a rule (1) and a starting shape (2)



(b) Set filtering parameters by clicking on a checkbox (3) and apply the rule (4)

Figure 8.12: Set the filtering parameters and apply the previously defined rule.

8.3.2 Implementation in Bloc

Regarding the implementation, most domain objects in shape grammars such as shapes, rules, and grammars themselves have their own widgets. These specific Bloc elements such as the `SGShapeElement`, `SGRuleElement`, and `SGGrammarElement` can each display their associated domain object. So we can define a shape, store it into a `SGShapeElement`, and display it with the use of Bloc. In the same manner we can define rules and grammars and directly inspect them from the Pharo workspace. This allows to quickly define these domain models and visualize them with the widgets in Bloc. Any domain modeling mistakes are directly visible and do not require more tedious debugging steps by running code to display them manually. A video showing the editor is available on the repository [12].

8.3.3 Reflection

Programming in Pharo has been a very pleasant experience. Two main features really stand out. First, the ability to inspect custom made domain objects visually. Shapes defined in code or with the domain specific language can directly be visualized from the system browser. This means faster and better feedback with respect to recently written code. It makes the development also more interactive. Second, the reflective nature of Smalltalk. It is for example possible to list all methods in a certain protocol and run them in order. All methods that create images are added to a certain protocol. If we want to regenerate all images we can run every message in this protocol. Also we can define messages that return shapes in a certain protocol and dynamically use all these messages to display all shapes to choose from in the editor. These and other features were very convenient for creating and exporting images. Also the expressive syntax of Pharo (Smalltalk) was used to create a domain specific language without the necessity to parse the source code. All in all, we feel that the usage of Pharo was a good choice and can improve productivity.

Bibliography

[1] M Agarwal and J Cagan. A blend of different tastes: The language of coffeemakers. *Environment and Planning B: Planning and Design*, 25(2):205–226, 1998. doi: 10.1068/b250205.

[2] K. N. Brown, C. A. McMahon, and J. H. Sims Williams. A formal language for the design of manufacturable objects. In *Proceedings of the IFIP TC5/WG5.2 Workshop on Formal Design Methods for CAD*, pages 135–155, Jun 1994.

[3] Terry Knight. Color grammars: Designing with lines and colors. *Environment and Planning B: Planning and Design*, 16:417–449, Jan 1989.

[4] R Krishnamurti. The construction of shapes. *Environment and Planning B*, 8(1): 5–40, 1981.

[5] Ramesh Krishnamurti. SGI: An interpreter for shape grammars. *Technical Report, Centre for Configurational Studies, Design Discipline, The Open University*, Jan 1982.

[6] Michael E. Mortenson. *Mathematics for Computer Graphics Applications: An Introduction to the Mathematics and Geometry of CAD/Cam, Geometric Modeling, Scientific Visualization*. Industrial Press, Inc., New York, NY, USA, 2nd edition, 1999. ISBN 083113111X.

[7] Christian Santoni and Fabio Pellacini. gTangle: A grammar for the procedural generation of tangle patterns. *ACM Trans. Graph.*, 35(6):182:1–182:11, Nov 2016. ISSN 0730-0301. doi: 10.1145/2980179.2982417.

[8] G Stiny. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3):343–351, 1980. doi: 10.1068/b070343.

[9] George Stiny, James Gips, George Stiny, and James Gips. Shape grammars and the generative specification of painting and sculpture. In *Segmentation of Buildings for 3DGeneralisation. In: Proceedings of the Workshop on generalisation and multiple representation , Leicester*, 1971.

[10] Rudi Stouffs. On shape grammars, color grammars and sortal grammars: a sortal grammar interpreter for varying shape grammar formalisms. In *Digital Physicality - Proceedings of the 30th eCAADe Conference - Volume 1 (eds. H. Achten, et al.), pp. 479-487, Faculty of Architecture, Czech Technical University, Prague, Czech Republic, 2012*, Nov 2012.

[11] M Tapia. A visual implementation of a shape grammar system. *Environment and Planning B: Planning and Design*, 26(1):59–73, 1999.

[12] Lars Wüthrich. Shape Grammar interpreter and editor implemented in Pharo. `https://github.com/Laeri/shapegrammar_pharo`, 2018.

[13] E. Zhang, P. Wonka, Y. Kobayashi, F. Bao, and Y. Li. Geometry synthesis on surfaces using field-guided shape grammars. *IEEE Transactions on Visualization & Computer Graphics*, 17:231–243, Feb 2010. ISSN 1077-2626. doi: 10.1109/TVCG.2010.36.