Universiteit Leiden

Master Computer Science

Generating Art using Genetic Programming

Name:	Benjamin Steffens
Student ID:	1551396
Date:	24/10/2023
Specialisation:	Advanced Data Analytics
1st supervisor:	Niki van Stein
2nd supervisor:	Anna V. Kononova
3rd supervisor:	Diederick Vermetten

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Due to the inherent subjective nature of art appreciation, designing a program to generate visually appealing or otherwise inspiring art may seem like an impossible task. However, the existence of evoMUSART [2], the International Conference on Artificial Intelligence in Music, Sound, Art and Design, suggests people will still try. Inspired by the various papers presented at evoMUSART, this thesis documents an attempt to use Genetic Programming, a sub-field of evolutionary algorithms, to generate and evolve art. Suggested indicators of potential aesthetic appeal implemented include rotational symmetry, positional alignment of individual parts, regulations on colour variety, and more. As part of the genetic programming nature of the implementation, generated art will evolve and, ideally, improve. Input-parameters will be experimented with to best tune the implementation to aesthetic appeal.

Table of Contents

Chapter 1

Introduction

As shown by the rising popularity of evoMUSART [2], the International Conference on Artificial Intelligence in Music, Sound, Art and Design, there is an increasing interest in using the fields of artificial intelligence and evolutionary computation for the development of generative visual art. Applications made in these fields are numerous. The Painting Fool [6] makes use of a hybrid hill-climbing and evolutionary approach to render scenes. Crossover and mutation are applied to a population of scenes for several generations. The most fit individual will have each of its attributes randomised a set number of times. If the new value leads to an improvement, it would be kept, otherwise discarded. Art Done Quick [7] is an app that allows its users to mutate or crossover generated art as a form of casual creation. By ensuring offspring are neither too similar nor too different from their parents users can feel in control. Art Done Quick generating titles for the artworks provides additional pleasure when the user sees the artwork in a new light. Deep Convolutional Neural Networks have been used in the Camera Obscurer app [42] to retrieve images visually similar to photos taken and uploaded by users. Retrieved images would be either abstract or realistic, where the former may serve as inspiration for design tasks.

In this thesis, genetic programming (GP) will be used to generate art; Specifically, compositions of ornaments. GP is a field of evolutionary computation in which a fitness or objective function is used to measure the performance of a program to solve a problem. Given this problem, a population of candidate solutions is generated and iterated on to solve it. The following research question (1) is posed: "Can genetic programming be used to compose ornaments on a canvas?" In tandem, another question (2) is posed: "Can genetic programming be used to evolve quantifiable aesthetic measures?"

Poli & Koza [21] list several properties commonly shared in areas where genetic programming has proven to be especially productive. GP was chosen due to how well the problem matches some of these properties, such as "An approximate solution is acceptable..." and "Small improvements in performance are routinely measured..."

The compositions are created by arranging ornaments/symbols on a canvas. ORNAMIKA is a repository/archive and company which "exist[s] to preserve and popularize ornamental heritage of the world and increase cultural empathy in modern society through design." [1]. These symbols, or ornaments, have been extracted from existing patterns by ORNAMIKA [1], see Figure 1. As part of their mission, ORNAMIKA also creates new graphics using these ornaments. The program presented in this thesis, SymbolPlacer, or SymPla, aims to mimic this, by use of GP.

More background information on generating art and genetic programming will be provided in Section 2. Section 3 will go into more detail regarding SymPla and its fitness function and elaborate on why the different aspects of the fitness function were chosen. Section 4 will elaborate on different experiments run on SymPla. The results will be interpreted in Section 5 and discussed in Section 6.

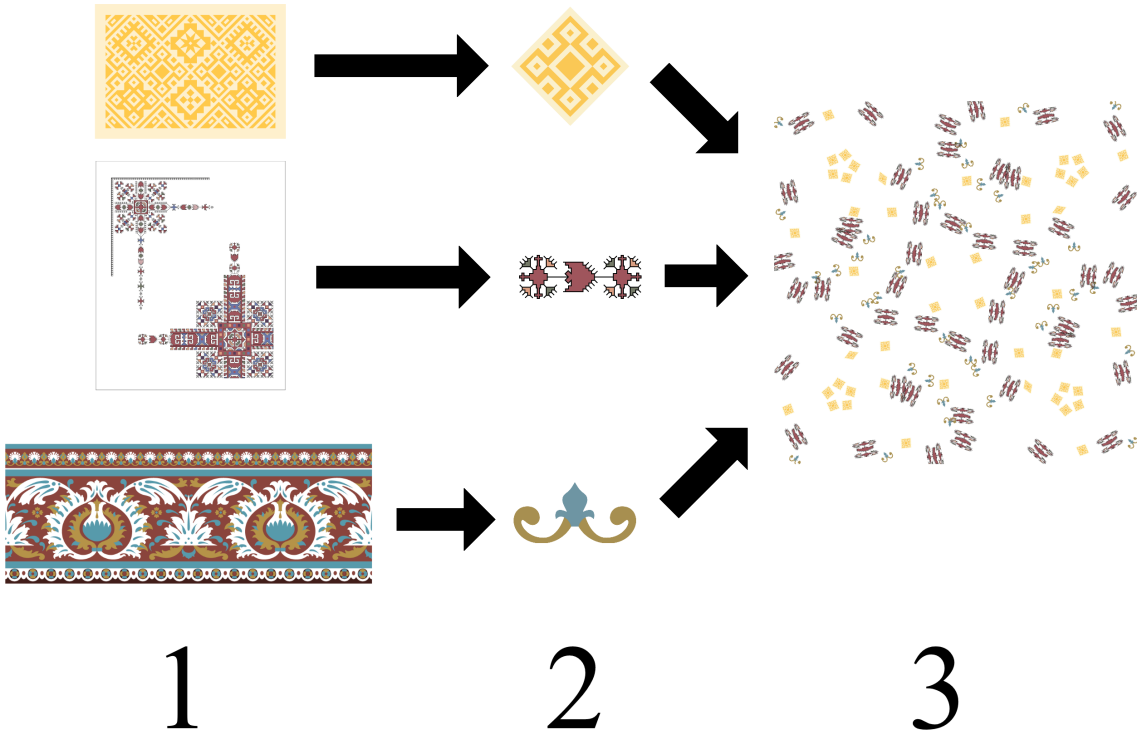


Fig. 1: Existing patterns (1) have had symbols extracted (2). These symbols get rotated and scaled before being placed on a canvas (3) by SymPla.

Chapter 2

Background

2.1 Generating Art

Many tools exist for the generation of art [38, 7, 22, 9]. Examples of work done in this field include:

- A program which combines patterns to generate Zentangle images [22]. Users can generate and interactively evolve images, or collaborate with other users by further evolving their images [40]. These images will be put through Wave Function Collapse [16] which will generate patterns similar to the input image. These patterns will be combined into the final Zentangle image output.
- An interface to let an AI draw art based on a person's electroencephalographic (EEG) waves [36]. A Generative Adversarial Network [15] is trained on EEG waves and paintings, both of which have been labelled with emotions. After training, a person's EEG signals will be processed by an encoder model before being input to another model generating the paintings. The result would be an AI which can make use of the human emotion it would otherwise oft lack.
- An AI which, when given input paintings, generates a chimera image to inspire artists [23]. By decomposing an artwork into objects of different classes, and reconstructing it using a visual universe, a collection of snippet images, these chimera images are created. By specifically using neural networks which are not trained on paintings and/or not state of the art in recognising objects in the input artwork, the ability to reinterpret works beyond the artists' intention would be mimicked.
- A genetic algorithm-based program used to evolve graphic designs [24]. Generated images are compared to a target image to calculate fitness. Despite this, goal is to help create designs that are disruptive and don't follow whatever is considered trendy design. One particular use-case mentioned is that of helping design posters.
- The Painting Fool [6], Art Done Quick [7] and Camera Obscurer [42], mentioned in Chapter 1.

Often, the goal in generating art is to create something that can generate works that provide aesthetic pleasure. The extent to which visual art is pleasing will be used to define aesthetic pleasure. Optimising for aesthetic pleasure can be difficult due to its subjective nature. Still, attempts have been made to measure what generally does and does not lead to aesthetic pleasure. Common causes for invoking aesthetic pleasure are interpretation, even where no meaning was intended [23, 42, 24], complexity and symmetry [47, 18], size (where bigger is better) [41], repeated exposure to an object [52, 51], prototypicality (how representative an object is to its category) [29], whether an object is curved or sharp [3] and the peak shift effect [35] to name a few.

Another aspect to generating aesthetically pleasing artworks is the process of improving the aesthetics. Ultimately, a universal aesthetic measure remains elusive [27], however various systems have been used to attempt to quantify and improve aesthetics. [38, 7, 22, 9] for example, provide a number of applications of bio-inspired systems in art.

2.2 Genetic Programming

As mentioned in Chapter 1, GP is a field of evolutionary computation which aims to solve a problem by generating and iterating over a population of candidate solutions. These candidate solutions come with a genotype and a resulting phenotype interpretation. The phenotype will be the end result, whereas the genotype will be a more abstract form to which genetic operations can be applied to. Genetic operations are used to iterate on the candidate solutions, with two of the most common genetic operations being crossover and mutation. Crossover allows two candidate solutions to (partially) exchange genetic info akin to sexual recombination [21]. Mutation allows a candidate solution to replace part of its genetic info with newly generated info, effectively abandoning part of its identity to adopt something new.

During the GP loop, generations of populations are iterated on. During each iteration, genetic operations would be applied to the offspring of the previous generation's candidate solutions, the parent population. Selection then determines which candidate solutions of the offspring population and the parent population would form the next generation's parent population. Typically, the fitness function plays a large role here, as the goal is to maximise the candidate solutions' fitness values. This process repeats for generations until some termination criterion is met. This criterion could be any of the following: A certain amount of candidate solutions has reached a specified fitness value, a limited budget of fitness evaluations has run out, a limited ceiling of generations has been reached [8], to name a few.

As an abstract form, the genotype of a candidate solution can be represented by various means. A short, by no-means comprehensible list of different representations of genotype (and sometimes phenotype) follows:

- Monolithic GP [8] represents the genotype as a tree. Nodes contain either functions or terminals. function nodes have at least one child, whereas terminal nodes are leaves. Crossover involves two trees swapping out subtrees. Mutation is performed by excising a tree's subtree, and generating a new subtree to take its place.
- Multigene GP [8], or MGGP, represents the genotype as a forest of smaller trees. Each tree's fitness value is weighed before being accumulated in a symbolic regression model. The trees' weights, the bias term, and the symbolic regression algorithm, providing the model, are all coefficients of this representation [39].
- Gene Expression Programming [8, 13], or GEP, attempts to get the best of both Genetic Algorithms' fixed-length strings and (monolithic) GP's tree representation. The genotype is represented as a fixed-length string separated into a head and a tail. The phenotype is a tree of which neither shape nor size is fixed. Even further inspiration is drawn from biology by allowing parts of the genotype to not encode into the phenotype. Whilst only applying genetic operators to the head of the string, this allows genetic operators to replace terminals with functions (expanding the phenotype tree) and vice versa. As a result, genetic operators do not need to be as restricted in their implementation as they would need to in GP or Genetic Algorithms [13].
- Linear GP [8, 31] represents programs as functions and applies genetic operators to these functions. By representing the candidate solutions as functions instead of trees, there is less of a need to build a genetic programming system in an interpreting language, which could otherwise be a major source of overhead [31].
- Dimensionally aware GP [20] aims to take away some of the abstract nature of GP. By adding units of measurement, or dimensionality, an effort is made to obtain results that are not only syntactically correct, but also semantically valuable. Variables and constants are accompanied by a unit of measurement exponent and operations such as addition, multiplication, etc. are redefined to handle these exponents in the function set.
- Cartesian GP [28] makes use of the graph as phenotype, whilst using an integer string for the genotype. Benefits over monolithic GP include a graph being more general than a tree, multiple forms of redundancy leading to an increased possibility of neutrality (multiple genotypes mapping to the same or similar phenotypes) and an improved ability to learn Boolean functions.
- Machine-coded GP or AIM-GP (Automatic Induction of Machine Code with Genetic Programming) [32] takes advantage of the lack of a need for an interpreter when using machine code as genotype. This saves time when executing (partial) solutions. Furthermore, the linear nature of machine code runs very analogous to genes in DNA, lending further credibility to AIM-GP.

Chapter 3

Methodology

In this thesis, SymPla [44] is proposed. SymPla is a program which uses GP to generate candidate solutions, rectangular-shaped canvases containing zero or more symbols arranged in such a fashion that they could, ideally, be aesthetically pleasing. Symbols are as explained in Section 1 and Figure 1. The canvas is the phenotype. SymPla is written in Python [49] and makes use of the pillow fork [5] of the Python Imaging Library [26], among other libraries. SymPla can be found online at `https://github.com/s1551396/GP` [44].

To keep things simple, monolithic GP, introduced in Section 2.2, was chosen as the model for SymPla. Monolithic GP will be referred to simply as GP from now on. In order to use GP, a genotype definition of a solution is also necessary. A candidate solution's genotype is a tree consisting of nodes and branches. Section 3.1 details how trees (genotype) translate to canvases (phenotype), and the make-up of a tree's nodes.

Whereas candidate solutions are initialised by randomly picking values and symbols from a pool of possibilities for each parameter, candidate solutions also need to be iterated on. SymPla does this in its main loop: Multiple trees are generated to form an initial parent population. The offspring population is formed by repeatedly selecting two candidate solutions from the parent population, using tournament selection [34]. These potentially go through mutation and/or crossover, before being added to the offspring population. Once the offspring population reaches a set capacity, it is added to the parent population. The merged population is then cut to the same size as the original parent population, leaving out the least fit candidate solutions, $(\mu+\lambda)$ selection. This population will serve as the parent population of the next iteration. This loop is repeated a given number of times. Once this process finishes, the most fit candidate solution from the parent population will be presented as solution. Pseudocode and a flowchart of this process are given in Figure 2 and Figure 3, respectively.

As mentioned before, Section 3.1 details how trees translate to canvases, and the make-up of a tree's nodes. Section 3.2 details how a tree's fitness is determined. Section 3.3 and Section 3.4 will cover the implementation of crossover and mutation, respectively.

3.1 Genotype

As mentioned before, a candidate solution's phenotype, the canvas, is a rectangular-shaped image file. The genotype is a tree complete with nodes and branches. A node's depth refers to how many edges are between it and the tree's root node. We will refer to nodes as units. A unit is a class that consists of a number of components. For a unit, u, its origin, scale and angle components are relative to its ancestors' and referred to as o_u, s_u and θ_u, respectively. In order to translate these nodes to the phenotype, there is also a need for absolute origin, scale and angle, O_u, S_u and Θ_u. u's parent is indicated as $u-1$. A list of all components of a unit follows.

- `children`. Either a symbol or a list of children-units, determined by arity.
- `arity`. This indicates how many children a unit has. Arity is zero if the unit is a leaf.
- `symbolIndex`. Index of the symbol stored in `children`. If unit is not a leaf, it doesn't have a symbol and `symbolIndex` is zero. `symbolIndex` is used to find the symbol in the `symbolsCache` if this option is enabled.
- `origin` in pixels. A tuple representing the unit's location relative to its parent's location. If the unit has a symbol, origin refers to where the centre of the symbol is located. A unit's absolute origin is defined in Definition 1.

  Definition 1 (Unit's absolute origin). *For unit, u, with relative origin $o_u = (o_u^x, o_u^y)$ and absolute origin, $O_u = (O_u^x, O_u^y)$, u's absolute origin, (O_u^x, O_u^y), is calculated as:*

  $O_u^x = \sum_{u=1}^{d} \cos(\theta_u) * o_u^x - \sin(\theta_u) * o_u^y + O_{u-1}^x.$
  $O_u^y = \sum_{u=1}^{d} \sin(\theta_u) * o_u^x + \cos(\theta_u) * o_u^y + O_{u-1}^y.$

  This calculation is an adaptation of Fisher et al.'s [14] transformation.

```
1: randomly generate parent population
2: initialize loopcounter to zero
3: while loopcounter is less than generations limit do

4:     initialize offspring population as empty set
5:     while size of offspring population is less than maximum offspring population size do

6:         tournament select parent population and store winner in offspring1
7:         tournament select parent population and store winner in offspring2

8:         initialize value to random value between zero and one
9:         if value is less than crossoverrate then
10:            crossover offspring1 and offspring2
11:        end if

12:        initialize value to random value between zero and one
13:        if value is less than mutationrate then
14:            mutate offspring1
15:        end if

16:        initialize value to random value between zero and one
17:        if value is less than mutationrate then
18:            mutate offspring2
19:        end if

20:        add offspring1 to offspring population
21:        if size of offspring population is less than maximum offspring population size then
22:            add offspring2 to offspring population
23:        end if
24:    end while

25:    select best of parent population combined with offspring population and store the result in parent population
26:    increment loopcounter
27: end while

28: pick and present the most fit unit in the parent population
```

Fig. 2: Pseudocode of SymPla's main process for iterating on candidate solutions.

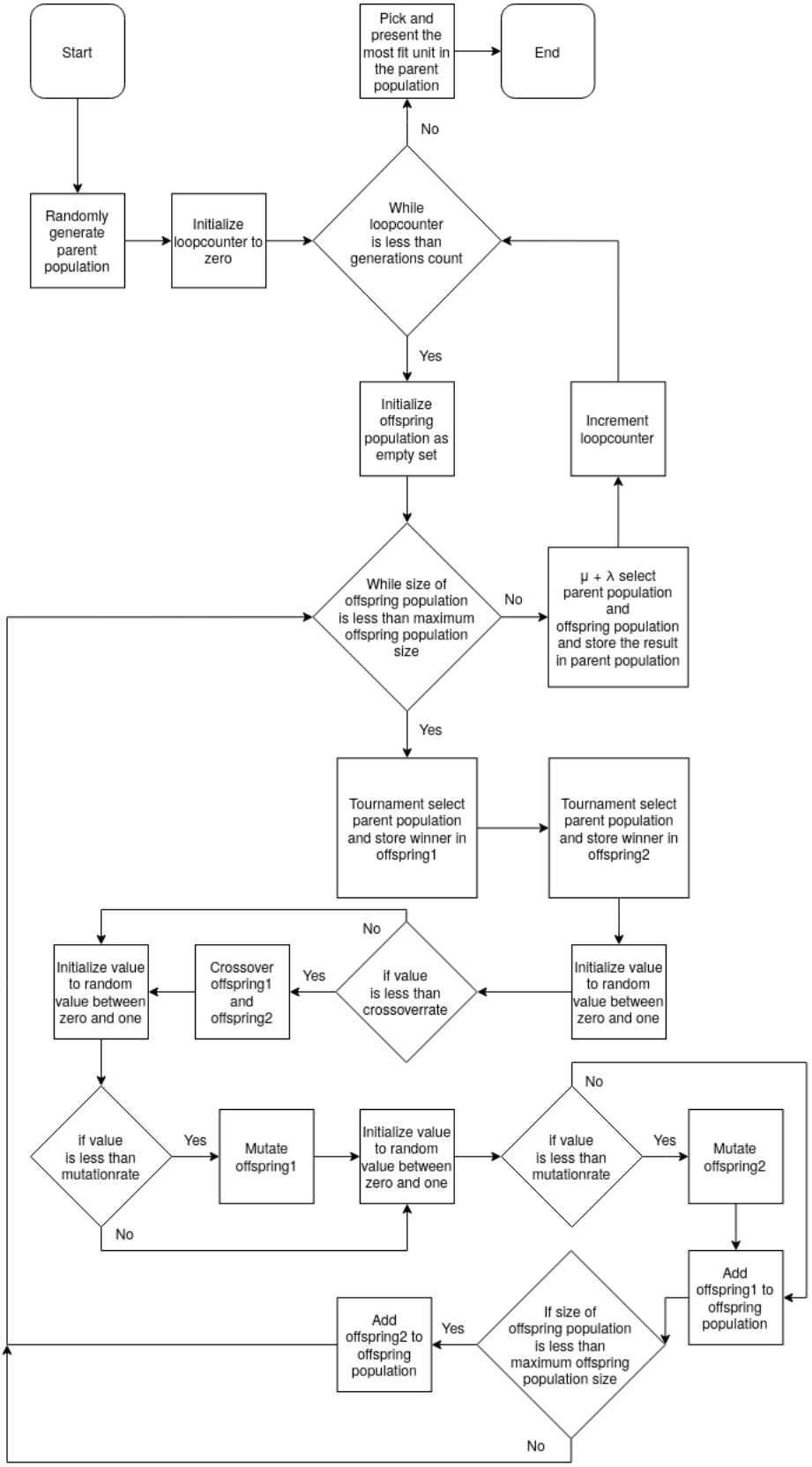


Fig. 3: Flowchart of SymPla's main process for iterating on candidate solutions.

– `angle` in degrees. The unit's rotation around its origin. Angle is relative to its parent's origin and angle. A unit's absolute angle is defined in Definition 2.

Definition 2 (Unit's absolute angle). *For unit, u, with relative angle θ_u, u's absolute angle, Θ_u, is calculated as:*

$$\Theta_u = \begin{cases} \theta_u, & \text{if } u \text{ is root.} \\ \Theta_{u-1} + \theta_u, & \text{otherwise.} \end{cases}$$

– `scale` in pixels. A tuple representing the unit's width and height relative to its parent's width and height. A unit's absolute scale is defined in Definition 3.

Definition 3 (Unit's absolute scale). *For unit, u, with relative scale $s_u = (s_u^x, s_u^y)$ and absolute scale, $S_u = (S_u^x, S_u^y)$, u's absolute scale, (S_u^x, S_u^y), is calculated as:*

$$S_u^x = \begin{cases} s_u^x, & \text{if } u \text{ is root.} \\ S_{u-1}^x + s_u^x, & \text{otherwise.} \end{cases} \qquad S_u^y = \begin{cases} s_u^y, & \text{if } u \text{ is root.} \\ S_{u-1}^y + s_u^y, & \text{otherwise.} \end{cases}$$

– `repetition`. An instance of the custom `Repetition` class. This class is used to draw a unit multiple times. For unit u with repetition r, being drawn for the xth time, r consists of the following components:
  - Count, c_r. The number of times u is repeated.
  - Origin, o_r. The relative origin of each repeatedly drawn unit to their previous iteration. $O_{u_x} = O_u + o_r * x$. Where O_u is as defined in Definition 1 and 2-tuple and scalar multiplication is as defined in Definition 4.

    Definition 4 (2-tuple and scalar multiplication). *Given 2-tuple $S = (x_1, y_1)$ and scalar a, $S * a = (x_1 * a, y_1 * a)$.*

  - Angle, θ_r. The relative angle of each repeatedly drawn unit to their previous iteration. $\Theta_{u_x} = \Theta_u + \theta_r * x$.
  - Scale, s_r. The relative scale of each repeatedly drawn unit to their previous iteration. $S_{u_x} = S_u + s_r * x$. Where 2-tuple and scalar multiplication is as defined in Definition 4.

– `fitness`. Holds the unit's fitness, if the unit is a root node, otherwise set to zero. Used to avoid recalculating fitness.
– `fitnessAspects`. List of individual fitness aspects' scores, before applying weights. Fitness aspects are detailed in Section 3.2.
– `parent`. The unit's parent. Set to NULL/None if the unit is a root node.
– `childNumber`. As units can have multiple children, this variable indicates the how-manieth child of its parent the unit is.
– `drawnLeaf`. This is used to count how many leaves were drawn for the DRAWABLELEAF fitness aspect.

Figure 5 displays an example unit tree. Figure 4 displays the drawn representation of this tree.

3.2 Fitness function

The fitness function, f, takes a Unit class instance, u, as input and produces a fitness score as output. The fitness score is an accumulation of punishments and rewards based on various factors and traits of u, which we will call fitness aspects, f_1 through f_6. Each of these aspects were initially chosen based on an intuitive understanding of aesthetics, ideally rewarding things that improve aesthetic pleasure, and punishing things that reduce aesthetic pleasure. Further motivation for each aspect can be found in their respective sections (Sections 3.2.1 to 3.2.7).

Each of the aspects of the fitness function, f_1, through f_6, is tested to see if expected results are being reproduced. The methodology for testing each fitness aspect will be discussed in Sections 3.2.1 to 3.2.7. For these tests a canvas size of 200x200 pixels is used (sheetWidth = sheetHeight = 200). arityMax, the maximum arity a node can have, is set to 3 and treeDepth, the maximum depth a tree can have, is set to 4. Each aspect, f_x, can hold a value between 0 and 100. This value is multiplied by a weight, w_x. Table 1 details which weights were attributed to which aspects. Punishments use negative weights, whereas rewards use positive weights. The final scores of each aspect are then added up to form the fitness score.

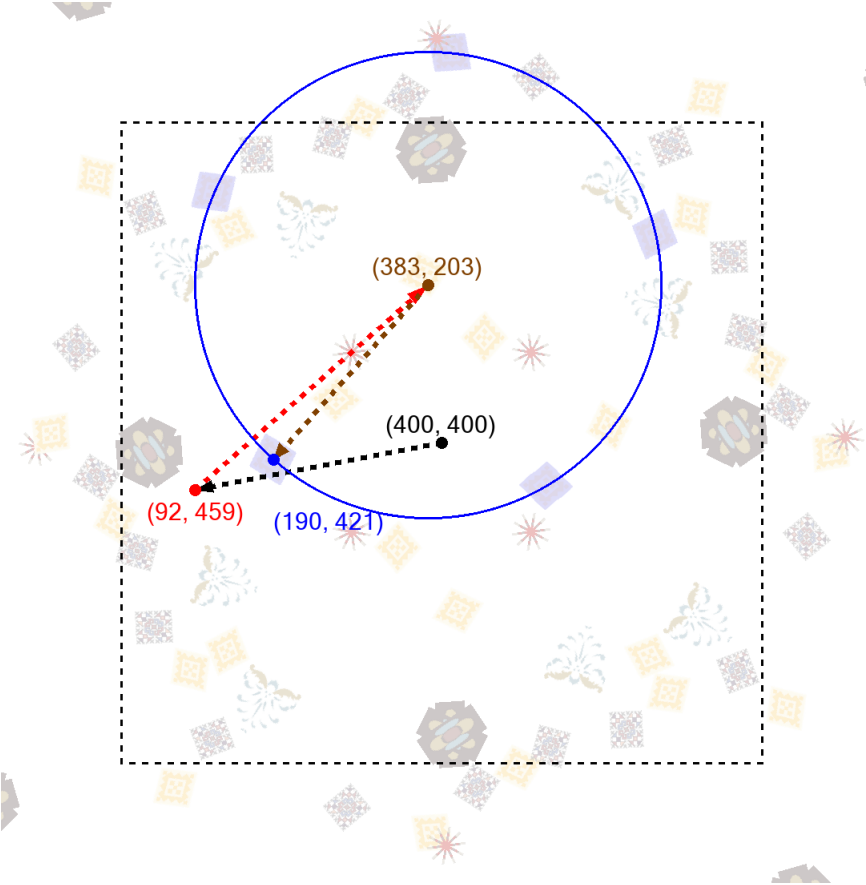


Fig. 4: A generated canvas with some additional highlighting for explanation. The area outside the dashed rectangle isn't drawn on the final canvas. The relative and absolute origin of the root, r is at (400, 400). r's first child, s, has an absolute origin at (92, 459). s' first child, t, has an absolute origin at (383, 203). t's first child, leaf u, has an absolute origin at (190, 421). Marked in blue is u's symbol. Also marked in blue are the four times this symbol is repeated due to s' repetition count being four. Note how the five symbols form an ordered rotation around t's origin.

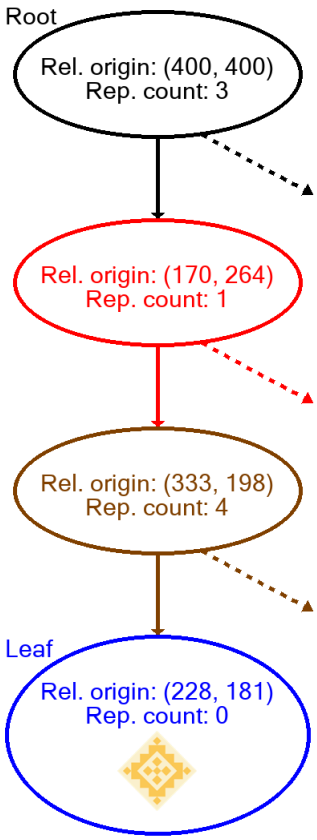


Fig. 5: A subtree of the canvas in Figure 4. Shown, per node, are the relative origin, repetition count and first child or symbol if the node is a leaf.

Fitness aspects					
f_1	f_2	f_3	f_4	f_5	f_6
-8	-1	$+5$	$+3$	-2	$+2$

Table 1: Default weight values, w_x, used by each fitness aspect, f_x.

3.2.1 Rotational symmetry

Given a unit, u, with repetition, r, rotational symmetry can be detected by using count, c_r, and angle, θ_r. Figure 6 provides an example of rotational symmetry. This aspect was chosen to take advantage of aesthetic pleasure provided by symmetry [47, 19]. A decision was made to scrap this fitness aspect, and instead incorporate it into the unit-generation process. This means that most generated units have varying orders of rotational symmetry.

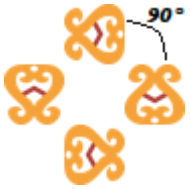


Fig. 6: An example of rotational symmetry. By taking the centre as an origin, the figure can fit in itself four times in a whole rotation. This means the order of rotational symmetry is four. [19]

3.2.2 Overlapping symbols

f_1 punishes overlapping symbols. Inspiration is loosely drawn from works such as [45, 4], where overlap is fundamental in generating images. Before drawing a symbol, we count the number of nontransparent pixels in the symbol, p_s. Each symbol's p_s is accumulated into an expected pixel count, p_e. Any pixel with an alpha channel value greater than zero is considered nontransparent. The formula for finding the value of the fitness aspect of overlap is $f_1 = 100 - 100 * (p/\max(p_e, 1))$ where p is the total number of nontransparent pixels in the canvas.

Using the symbol in Figure 7, Symbol1, we craft the tree in Figure 9, Tree1. Using Gimp's [46] histogram feature, see Figure 8, we can see that in Symbol1 5166 pixels are nontransparent. The expected nontransparent pixelcount for Tree1 where neither instances of Symbol1 overlap would be $5166 * 2 = 10332$. Using Gimp's [46] histogram feature on Tree1, see Figure 10, this expectation is met; f_1 is expectantly zero in this case.

Fig. 7: Symbol1, Klimov_022_reco_1. Resized to 99x100. This image is in RGBA mode. Each pixel uses 8 bits for each of four channels: red, green, blue, alpha.

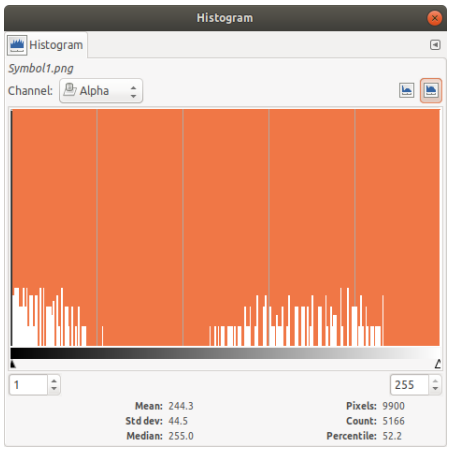


Fig. 8: Gimp [46] histogram of Symbol1 in Figure 7. Bottom right states 5166 pixels have an alpha value in interval [1, 255].

Rotating Symbol1 45 degrees using pillow [5] gives us a nontransparent pixelcount of 5164. Tree2, see Figure 11, draws this rotated symbol on top of a regular Symbol1. Due to partial overlap, the nontransparent pixelcount of Tree2 is not $5164 + 5166 = 10330$, the addition of Symbol1's nontransparent pixelcount with its 45-degree rotated counterpart's. Gimp's [46] histogram feature, see Figure 12, tells us Tree2 has 6024 nontransparent pixels. Our implementation did expect 10330 pixels, and did count 6024, making $f_1 \approx 86.2$.

Fig. 9: Tree1. A tree drawing two of Figure 7's symbol. The symbols do not overlap.

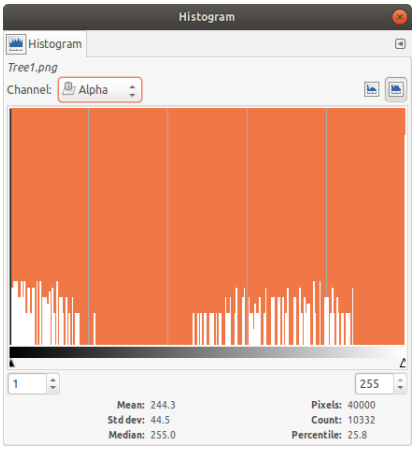


Fig. 10: Gimp [46] histogram of Figure 9. Bottom right states 10332 pixels have an alpha value in interval [1, 255].

To account for size of the canvas, the formula is changed to $f_1 = \min(\max(0, p_e - p) * 100/(l/8), 100)$, where $l = \texttt{sheetWidth} * \texttt{sheetHeight}$. This formula results in a fitness aspect value of $f_1 \approx 86.1$ for Tree2.

Fig. 11: Tree2. A tree drawing two of Figure 7's symbol. The symbols overlap partially.

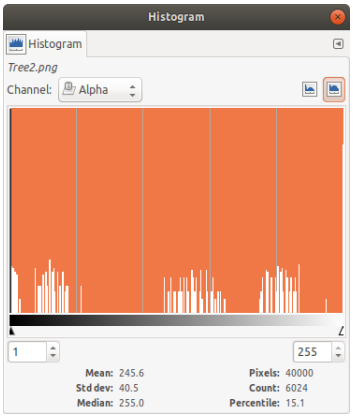


Fig. 12: Gimp [46] histogram of Figure 11. Bottom right states 6024 pixels have an alpha value in interval [1, 255].

3.2.3 Punish too few/too many pixels

f_2 punishes a ratio of empty pixels above a certain threshold, $\max_p$, or below a different, lower, threshold, $\min_p < \max_p$. This aspect was chosen to take advantage of aesthetic pleasure potentially provided by complexity [47].

Using Tree2, see Figures 11 and 12, we have a nontransparent pixel count of $p = 6024$. Using l as the width of the canvas multiplied by its height, the canvas being 200 by 200, then minimum pixel threshold is defined as $\min_p = l/8 = 5{,}000$. The maximum pixel threshold is defined as $\max_p = l/2 = 20{,}000$. Finally,

$$f_2 = \begin{cases} (p - \max_p)/\max((l - \max_p)/100, 1), & \text{if } p > \max_p \\ (\min_p - p)/\max((\min_p/100), 1), & \text{if } p < \min_p \\ 0, & \text{otherwise.} \end{cases}$$

Tree2, see Figures 11 and 12, has a pixel count greater than the minimum threshold, and lesser than the maximum threshold: $f_2 = 0$.

Tree3, see Figures 13 and 14, has a nontransparent pixel count of 31074. This value is greater than the maximum pixel count threshold: $f_2 \approx 55.4$.

Tree4, see Figures 15 and 16, has a nontransparent pixel count of 1346. This value is lesser than the minimum pixel count threshold: $f_2 \approx 73.1$.

Fig. 13: Tree3. A tree drawing nine of Figure 7's symbol. The symbols consume more than half of the canvas.

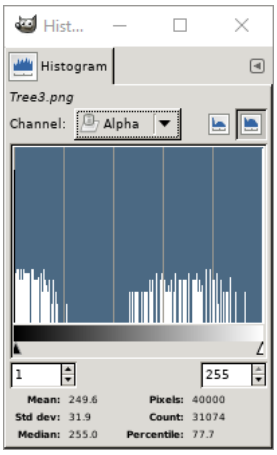


Fig. 14: Gimp [46] histogram of Figure 13. Bottom right states 31074 pixels have an alpha value in interval [1, 255].

Fig. 15: Tree4. A tree drawing one of Figure 7's symbol at half the scale. The symbol consumes less than an eighth of the canvas.

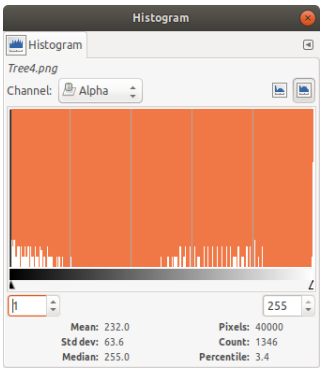


Fig. 16: Gimp [46] histogram of Figure 15. Bottom right states 1346 pixels have an alpha value in interval [1, 255].

3.2.4 Reward leaves drawn on the canvas

f_3 rewards symbols that are drawn on the canvas. This is done by increasing the reward based on how many draw-able leaves the tree has, correcting for the amount of leaves the tree has. This aspect was chosen to minimise the appearance of empty canvases.

A unit's leaves each contain a symbol which can be drawn multiple times on the canvas. l is the amount of leaves a tree has. d_l is the amount of a tree's leaves which are drawn at least once. The maximum amount of leaves a tree can have, $\max_l$, is the maximum arity any node can have to the power of the maximum depth the tree can have. For our parameters this would be $4^3 = 64$.

$f_3 = d_l/l * 100$. One problem that was run into is that trees with fewer leaves, need fewer of their leaves to be drawn on the canvas for a higher fitness score as well. A root node which is also a leaf will have the maximum fitness aspect score of $f_3 = 100$ if its one symbol can be drawn. Hence f_3 was changed to $f_3 = (d_l * 100)/\max(\max_l, 1)$. This indirectly rewards complexity as trees with more leaves are more likely to have a higher count of draw-able leaves.

Tree4, see Figure 15, consists of one leaf which is drawn; $f_3 \approx 1.6$.

Tree3, see Figure 13, despite drawing nine symbols, also consists of one drawn leaf. This leaf repeats twice, and its parent also repeats twice, hence nine symbols appear on the canvas; $f_3 \approx 1.6$.

Tree2, see Figure 11, consists of two drawn leaves: $f_3 \approx 3.1$.

Tree5, no figure, tries to draw a symbol out of the canvas' bounds. This leaves the canvas empty; $f_3 = 0$.

Tree6, see Figure 17, consists of 64 drawn leaves. This would be the maximum amount of possible draw-able leaves a tree could have under our current parameters; $f_3 = 100$.

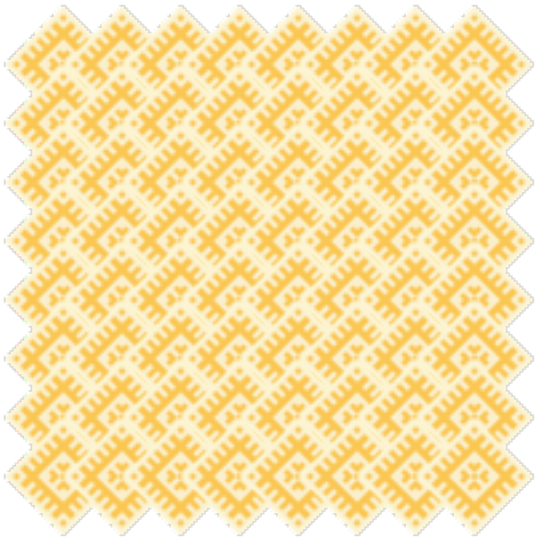


Fig. 17: Tree6. A tree drawing 64 of Figure 7's symbol. Each symbol drawn by a different leaf.

3.2.5 Reward symbols that touch one another

f_4 rewards symbols that touch one another. Per symbol, S_i, the location and size is compared to each other symbol, S_j, to determine whether there's overlap. As symbols are not necessarily rectangular, every pixel, p, in the overlapping area is iterated on to see if p is nontransparent in S_i and if any of the eight adjacent pixels - two horizontal, two vertical, four diagonal - is also nontransparent in S_j or vice versa. If p is nontransparent in both S_i and S_j, no reward for this symbol-pair is given as the symbols are not touching, but overlapping. This aspect was chosen out of personal curiosity.

To determine whether two symbols, S_i and S_j, touch, each symbol's location and size is tracked. By using `Image.getbbox` [5] transparent areas are cropped out. Comparing S_i's location and dimensions with S_j's reveals a surface, A, where overlap or touch on the canvas is possible. If so, SymPla generates two lists, L_i and L_j, of all pixels' alpha values in A. One for each symbol. If some alpha value $\alpha_i \in L_i$ such that $\alpha_i > 0$ and the corresponding $\alpha_j \in L_j$ such that $\alpha_j > 0$, SymPla will recognise the S_i and S_j symbol-pair as one featuring overlap. This pair will therefore be considered as not touching.

Otherwise, if some alpha value $\alpha_i \in L_i$ such that $\alpha_i > 0$ and one or more of the eight alpha values, α_j', directly above, below, left, right or diagonal to the corresponding $\alpha_j \in L_i$ such that $\alpha_j' > 0$, SymPla will recognise the S_i and S_j symbol-pair as touching.

$$f_4 = \begin{cases} 100 - \frac{\texttt{maxTouch} - C}{\texttt{maxTouch}} * 100, & \text{if } C < \text{maxTouch.} \\ 100, & \text{otherwise.} \end{cases}$$

Where `maxTouch` is the number of drawn symbols on the canvas multiplied by a parameter with a value in [0, 1]. This parameter is set to $\frac{1}{2}$. C is the amount of symbol pairs that touch.

For Tree1, Figure 9, $f_4 = 0$. The rightmost symbol starts at (100, 50), after the leftmost symbol ends at (99, 150), therefore these symbols are considered to not be touching.

Tree2, Figure 11, $f_4 = 0$. The only two symbols drawn overlap, therefore they are considered to not be touching.

Tree3, Figure 13, $f_4 = 100$. There are eight symbol-pairs that touch. Numbering the symbols from left-to-right, top-to-bottom, starting at S_1 at the top-left, the pairs are: (S_1, S_5), (S_2, S_4), (S_2, S_6), (S_3, S_5), (S_4, S_8), (S_5, S_7), (S_5, S_9), (S_6, S_8). `maxTouch` $= 9 * \frac{1}{2} = 4.5$. $C = 8$. As $C \geq$ `maxTouch`, $f_4 = 100$.

3.2.6 Punish a variety of colours being present

f_5 punishes canvases for having a number of distinct colours greater than some threshold. Most symbols use a few distinct colours. However, these colours are distinct only in human eyes, and are stored as many slightly different hues of the same colour on the system side, see Figure 18. scikit-learn's [33] DBSCAN [12] implementation was used to cluster colours, in an attempt to recapture the colours as perceived by human eyes. This aspect was inspired by the relevance of colour in various articles [25, 30, 17].

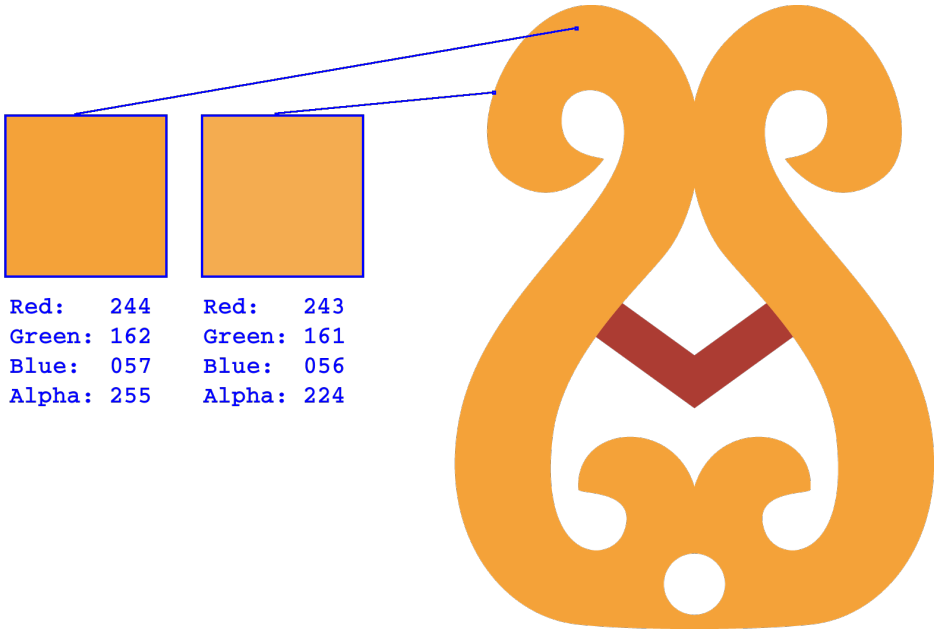


Fig. 18: Two pixels of the symbol on the right are highlighted and enlarged on the left. The RGBA values for each pixel are noted directly below the enlarged pixels. As the edge of the symbol is 'soft', pixels on this edge blend into the background, achieved by a gradually decreasing alpha-value. Other pixels have no transparency at all, indicated by alpha being 255.

Unlike the other fitness aspects, f_5 is tested on generated canvases of size 800x800. DBSCAN [12] uses two notable parameters, eps, the maximum distance determining whether two points are neighbours, and `minPts`, the minimum amount of neighbours a point needs to have to be considered a core point. Two core points, c_a and c_b, can form two separate clusters, unless they are density-reachable. This means there is a path $p_1 \rightarrow ... \rightarrow p_n$ where $p_1 = c_a$, $p_n = c_b$ and the distance between p_i and $p_{i+1} \leq$ eps, where $i \in [1, n]$.

Initially, a problem occurred in canvases containing many different colours. For these canvases, every core point is density-reachable by every other core point. The colours would end up blending together into one cluster, see Figure 19. To minimise the prevalence of this issue, eps was lowered.

Another issue with f_5 is the time taken to calculate it. At first, the distance between colours was determined by taking the sum of the absolute differences in the red, green and blue channels. As this would prove to be very time-consuming, Pillow's [5] `Image.histogram()` feature was used instead. This function returns "a list of pixel counts, one for each pixel value in the source image. ... If the image has more than one band, the histograms for all bands are concatenated." [5] This means a pixel can be grouped into multiple clusters, one per each channel. See also Figures 20 and 21.

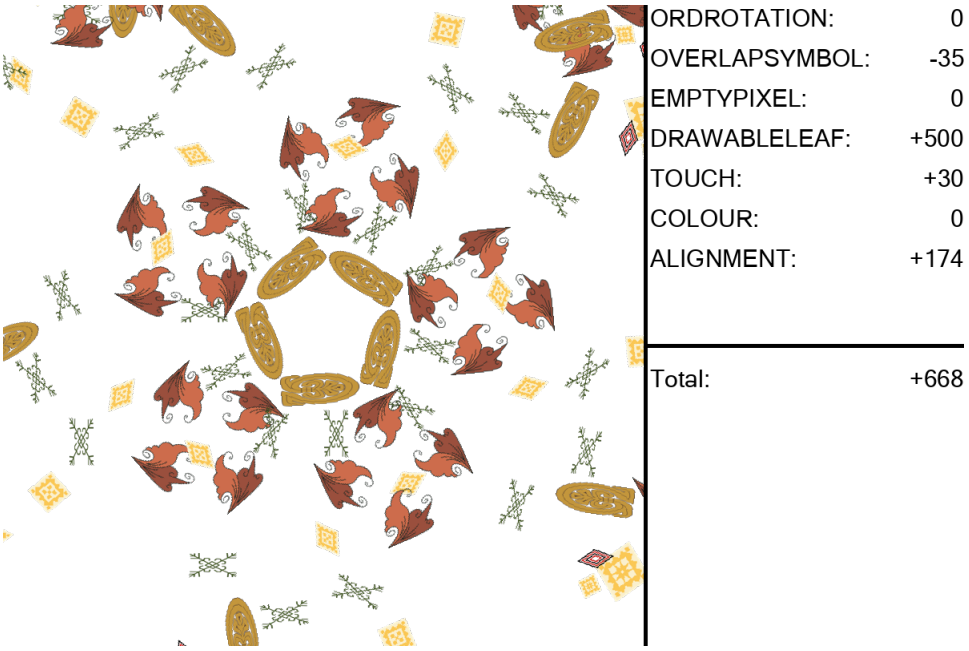


Fig. 19: A SymPla result. Canvas on the left, with fitness values shown on the right. Note how `COLOUR` is zero, meaning no punishment is given for f_5.

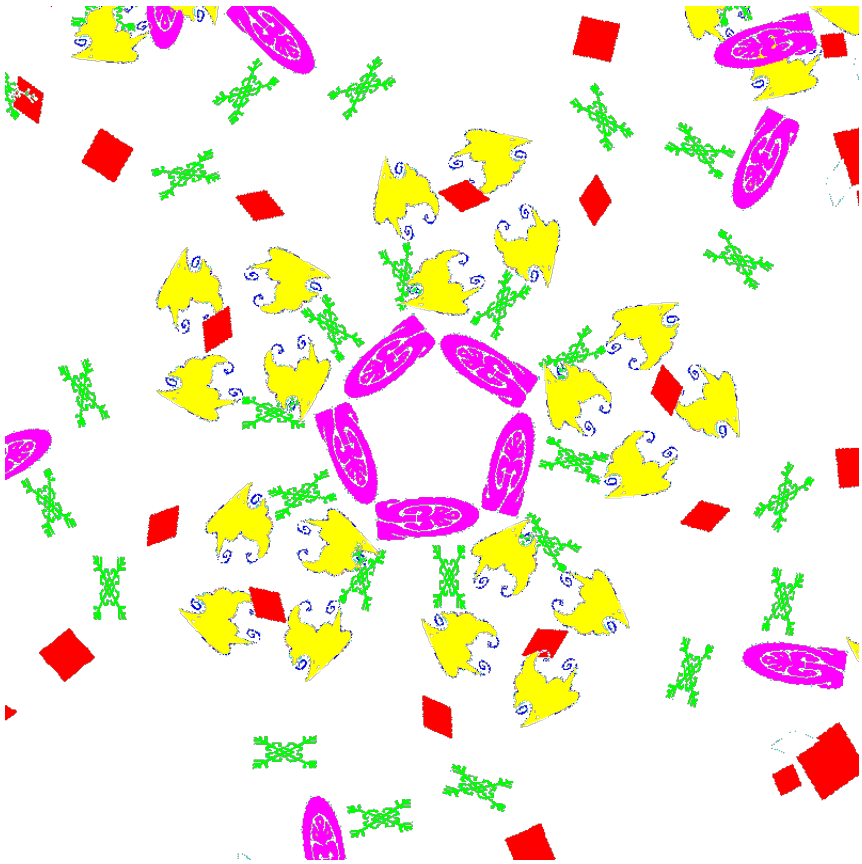


Fig. 20: Canvas from Figure 19 with cluster-highlighting. The colour of each pixel indicates which cluster it belongs to. The sum of the absolute differences in the red, green and blue channels is used to calculate each pixel's value, before running them through DBSCAN.

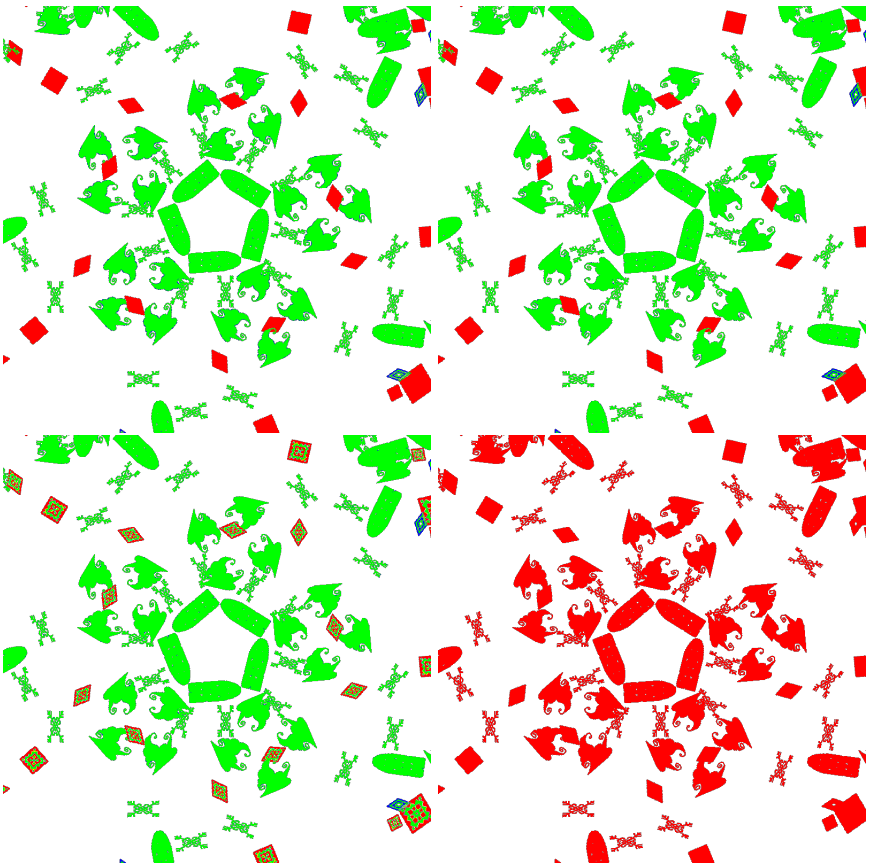


Fig. 21: Canvas from Figure 19 with cluster-highlighting per band. The colour of each pixel indicates which cluster it belongs to. Each canvas represents a different band (red (top-left), green (top-right), blue (bottom-left), alpha (bottom-right)). This is how a pixel could be part of up to four different clusters.

Finally, a decision was made to instead apply DBSCAN [12] to a canvas of size 100 by 100 times the amount of different symbols. This canvas would contain one instance of each symbol. Each symbol would be resized to w by h, where either w or h would be 100 pixels and the other dimension would hold a value such that the original width-height ratio would be preserved. This would allow for there to be no overlapping of symbols. A program, `DBSCANSymbols` was written to create this canvas.

`DBSCANSymbols` would then apply DBSCAN [12] to this canvas and keep track, per symbol, which clusters were represented in that symbol. By comparing the size of each symbol's cluster-set to human-manually counted colours of that symbol, `DBSCANSymbols` would be able to calculate an error per symbol, which would accumulate to a total error for all symbols. This process would repeat for different `eps` and `minPts` values, with the aim of finding for which parameters the total error would be lowest. The lowest error found For 82 symbols was 38. `eps = 13`, `minPts = 400`.

Now, instead of having to use DBSCAN [12] in the fitness function, each symbol was assigned a set of colour clusters by `DBSCANSymbol.py`. This means that these values are pre-calculated, speeding up the fitness function.

3.2.7 Reward aligning symbols

f_6 rewards symbols that align. Each symbol's polar coordinates, distance and angle, are calculated by taking the centre of the canvas as origin. The difference in distance to the origin and angle of each symbol-pair are then added to determine f_6. The Cartesian distance between each symbol pair is used to weigh the effect the pair has on f_6. This aspect was inspired by wallpaper [19] patterns and patterns artificially made by certain animals [37, 48].

$f_6 = 100 - (f_6^{'} / \max(\frac{(n-1)*n}{2}, 1)) * 100,$
where n is the amount of drawn symbols and
$f_6^{'} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (|\frac{R_{S_i} - R_{S_j}}{\texttt{max symbol distance}/2}| + |\frac{\Theta_{S_i}^{'} - \Theta_{S_j}^{'}}{4\pi}|) * (1 - \frac{\sqrt{(O_{S_i}^{x} - O_{S_j}^{x})^2 + (O_{S_i}^{y} - O_{S_j}^{y})^2}}{\texttt{max symbol distance}}),$
where R_{S_a} is the polar distance of symbol a, $\Theta_{S_a}^{'}$ is the polar angle of symbol a, $O_{S_a}^{b}$ is the Cartesian absolute origin of symbol a, `max symbol distance` is $\sqrt{\texttt{sheetWidth}^2 + \texttt{sheetHeight}^2}$.

For Tree4, Figure 15, $f_6 = 0$ as there are no symbol pairs. Tree1, Figure 9, contains two symbols, hence one symbol pair; $f_6 \approx 83.9$. Tree2, Figure 11, like Tree1, contains two symbols, however, both are at the origin, meaning their polar coordinates are (0, 0). This means they align perfectly, hence $f_6 = 100$. Tree3, Figure 13, with nine symbols, contains 36 pairs; $f_6 \approx 85.3$.

3.3 Crossover

The offspring population is formed by repeatedly selecting two children from the parent population using tournament selection [34]. Tournament selection was chosen as it's "the most commonly employed method for selecting individuals in GP" [34]. If two children crossover, a node in each tree will be chosen at random. This node, its children, its children's children, and so forth, will form a subtree. The two trees' subtrees are then swapped out, resulting in two new trees.

As crossover would very often result in trees with lower fitness, most trees would not appear in the next generation. To reduce stagnation, the crossover-rate was set to 0.75, meaning there is a three-in-four chance two children will crossover.

Crossover could lead to ever expanding trees, and therefore bloat. To counter this, a maximum tree depth parameter was used. Should a tree, due to crossover, exceed the maximum depth, nodes exceeding this depth will be pruned. Any nodes that are turned into leaves as a result of this, will be assigned a randomly selected symbol.

There is an example crossover between two trees in Figure 22.

3.4 Mutation

If a unit mutates, one of its nodes will be selected at random. This node, its children, its children's children, and so forth, will form a subtree.

As with crossover, mutation also rarely leads to an improvement of fitness. To reduce stagnation, the mutation rate is set to 0.75, meaning there is a three-in-four chance a child will mutate.

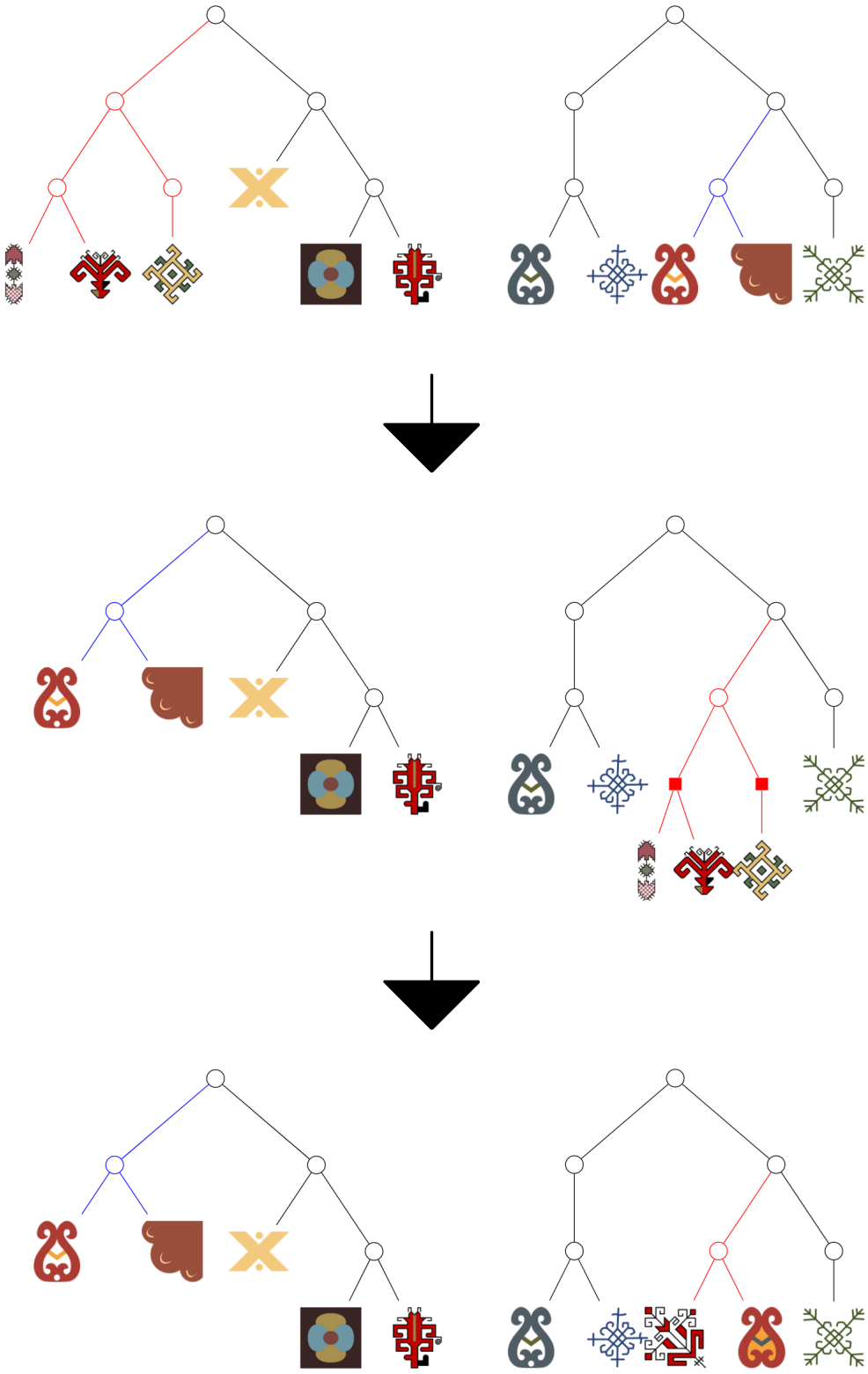


Fig. 22: Crossover between the left and right trees. First, two subtrees are selected, highlighted in red for the left tree, blue for the right tree. Next, the subtrees are swapped out. The left tree discards its subtree and takes on the right tree's, whereas the right tree takes on the left tree's subtree in the spot where it discarded its subtree. Finally, to ensure the maximum tree depth is preserved, the right tree has two nodes converted to leaves. These nodes are marked as filled-out rectangles in the previous step.

Node height is used to determine if a node is eligible for this selection. A node's height is determined by the depth of its deepest leaf relative to its own depth. A leaf has a height of zero. A node whose children are leaves has a height of one, etc. A node can be selected if its height is greater or equal to zero and lesser or equal to half of the maximum tree depth.

The selected node's subtree will be discarded; A new subtree is generated to take its place, whilst ensuring the generated subtree will not make the original tree exceed the maximum tree depth. There is an example mutation in Figure 23.

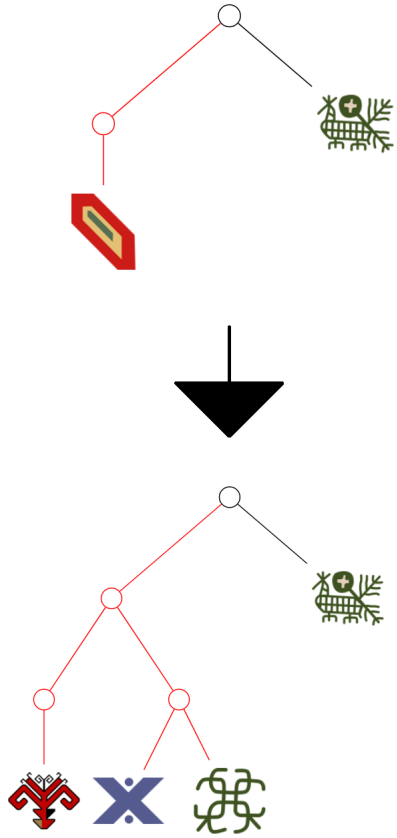


Fig. 23: The top tree will undergo mutation. First a subtree is selected, highlighted in red. Then, the subtree is discarded and replaced by a newly generated subtree.

Chapter 4

Experiments

Two experiments were performed. Both experiments attempt to reveal if fitness is affected by different parameter settings, and how these settings affect fitness evolution. Experiment 2 also tests whether a higher fitness score reflects a more aesthetic canvas.

4.1 Experiment 1

SymPla was run under six different parameter settings, p_1 through p_6, using the same seed for the random number generator to ensure results are not affected by seed difference. For each of these six runs, SymPla would run 25 times, producing samples P_i. The average fitness aspects' scores, f_1 through f_6, and overall fitness scores, f, for these samples can be found in Table 2 and are plotted in Figures 24 and 25. The evolution of the fitness value is plotted in Figure 26. Five parameters were experimented with:

1. gen, the number of generations a run would go through before returning the most fit individual. Default value is 200.
2. parentPop, the number of units making up the parent population. Default value is 100.
3. childPop, the number of units making up the child population. Default value is 50.
4. treeDepth, the maximum depth a tree could have. Default value is 3.
5. maxArity, the maximum arity any node in any tree could have. Default value is 2.

	Parameter values					Results averaged over the runs							
	gen	parentPop	childPop	treeDepth	maxArity	f_1	f_2	f_3	f_4	f_5	f_6	Total	Time
p_1	200	100	50	3	2	−27	−6	+395	+102	−7	+175	+631	4059
p_2	100	100	50	3	2	−28	−13	+337	+109	−3	+176	+576	2546
p_3	200	50	50	3	2	−17	−4	+397	+121	−7	+174	+663	3952
p_4	200	100	25	3	2	−43	−9	+330	+106	−7	+173	+549	2309
p_5	200	100	50	4	2	−39	−14	+191	+195	−6	+174	+501	21617
p_6	200	100	50	3	3	−36	−17	+111	+198	−2	+177	+430	7531

Table 2: SymPla was run 25 times for the parameter settings p_i for $i \in (1, 2, ..., 6)$ on the left, producing samples P_i for $i \in (1, 2, ..., 6)$. The average fitness aspect scores and overall fitness for these samples are in the right columns. Fitness aspects are as detailed in Section 3: f_1 is symbol overlap. f_2 is pixel count. f_3 is amount of drawn leaves. f_4 is symbol touch. f_5 is colour variety. f_6 is symbol alignment. The corresponding weights can be found in Table 1. A fitness aspect's score can take a value anywhere between zero and a hundred, multiplied by its corresponding weight. The final column, Time, is the average time taken per run of the corresponding sample. Time is expressed in seconds.

Overall, save for p_2 and p_4, all settings use about the same amount of fitness evaluations or fitness budget, with slight variations due to mutation and crossover not being guaranteed to trigger. Figure 26, made in IOHanalyzer [10], depicts how fitness improves over a run's generations. As can be seen in Figure 26, p_2 ends up with a smaller budget due to p_2 being the same as default, p_1, but using a smaller generation size. p_4 also uses up a smaller budget as it's the same as p_1, but has a smaller sized child population. p_3 uses a smaller parent population, but this only impacts the fitness budget for one generation. p_5 and p_6 take over the same parameters affecting the fitness budget from p_1, instead making the cost of each fitness evaluation more expensive by increasing the complexity of the genotype.

As another means of visualising the evolution of the canvases, Figure 27 features a collage of a run of parameter setting $p1$. This collage shows how the canvas with the highest evaluated fitness changes over generations. Appendix C also contains collages for one run of each of the other fitness parameter settings, p_2 through p_6.

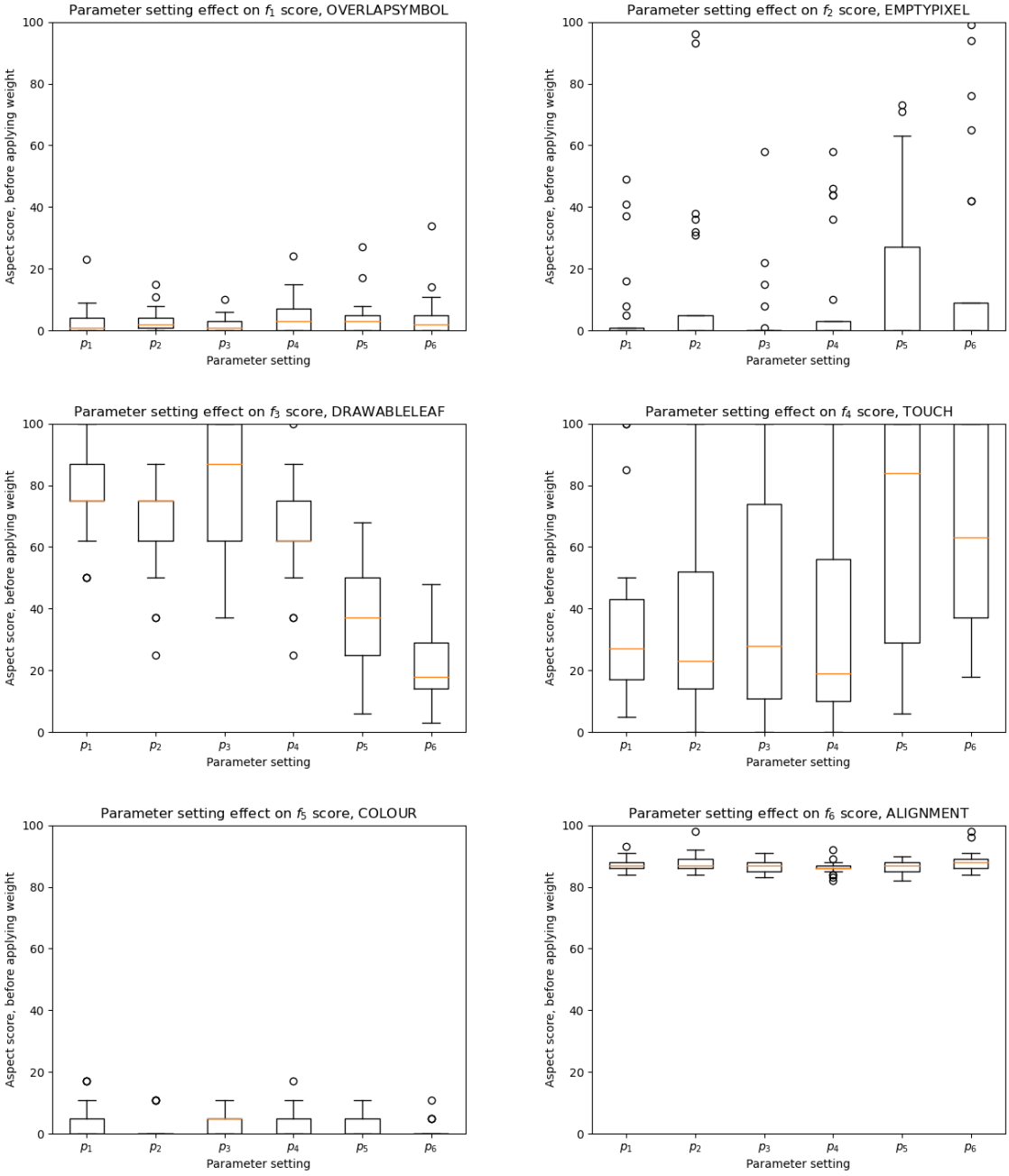


Fig. 24: Box-plots for each fitness aspect. Each box represents the fitness aspect's score, as indicated on the vertical axis, for the most fit individual of 25 runs given the specific parameter setting, p_1 through p_6, as indicated on the horizontal axis.

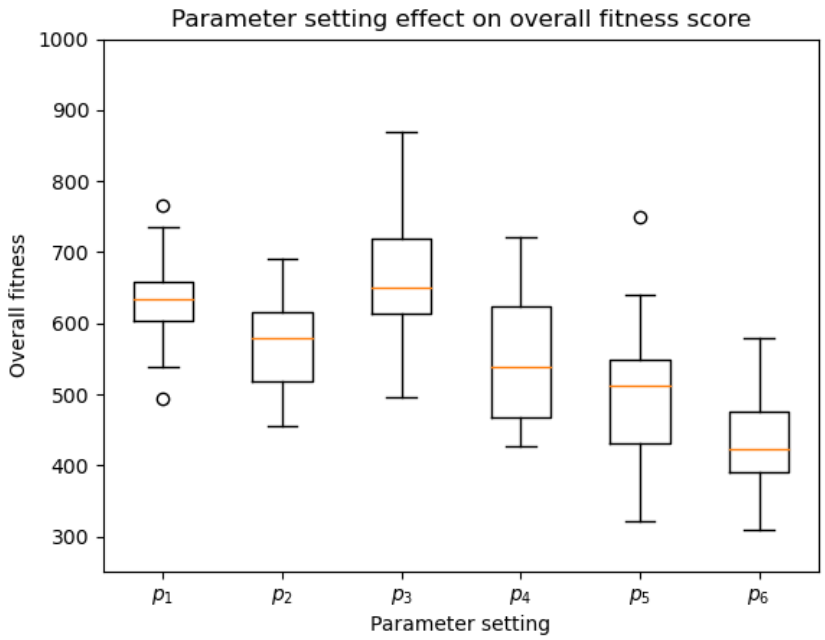


Fig. 25: Box-plot for overall fitness. Each run produces a most fit individual. As SymPla was run 25 times for each parameter setting, p_1 through p_6, as indicated on the horizontal axis, 25 most fit individuals were created for each parameter setting. Each box represents the fitness scores, as indicated on the vertical axis, for these 25 most fit individuals. Using the weights as specified in Table 1, fitness can take a value in $[-1100, +1000]$.

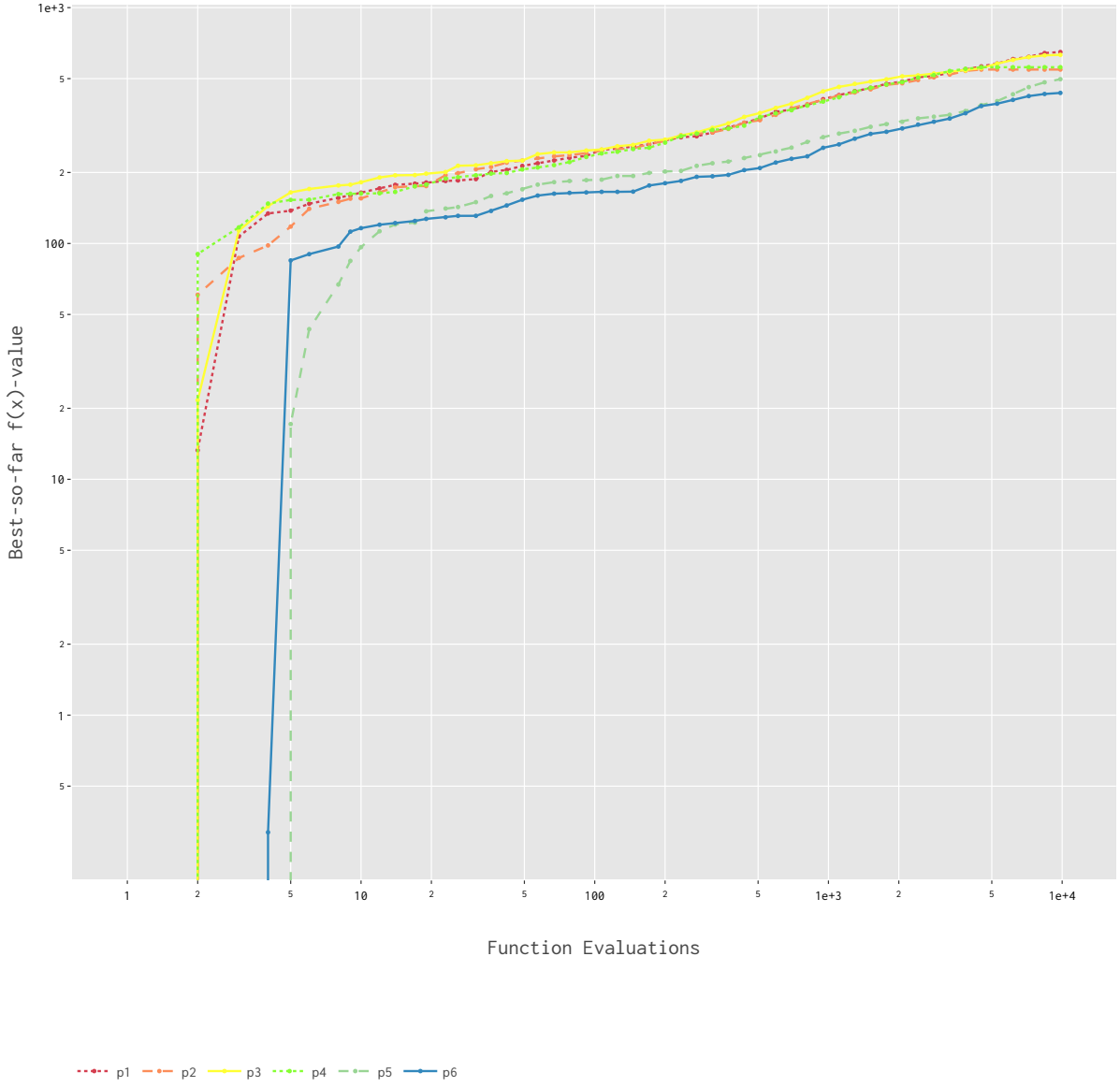
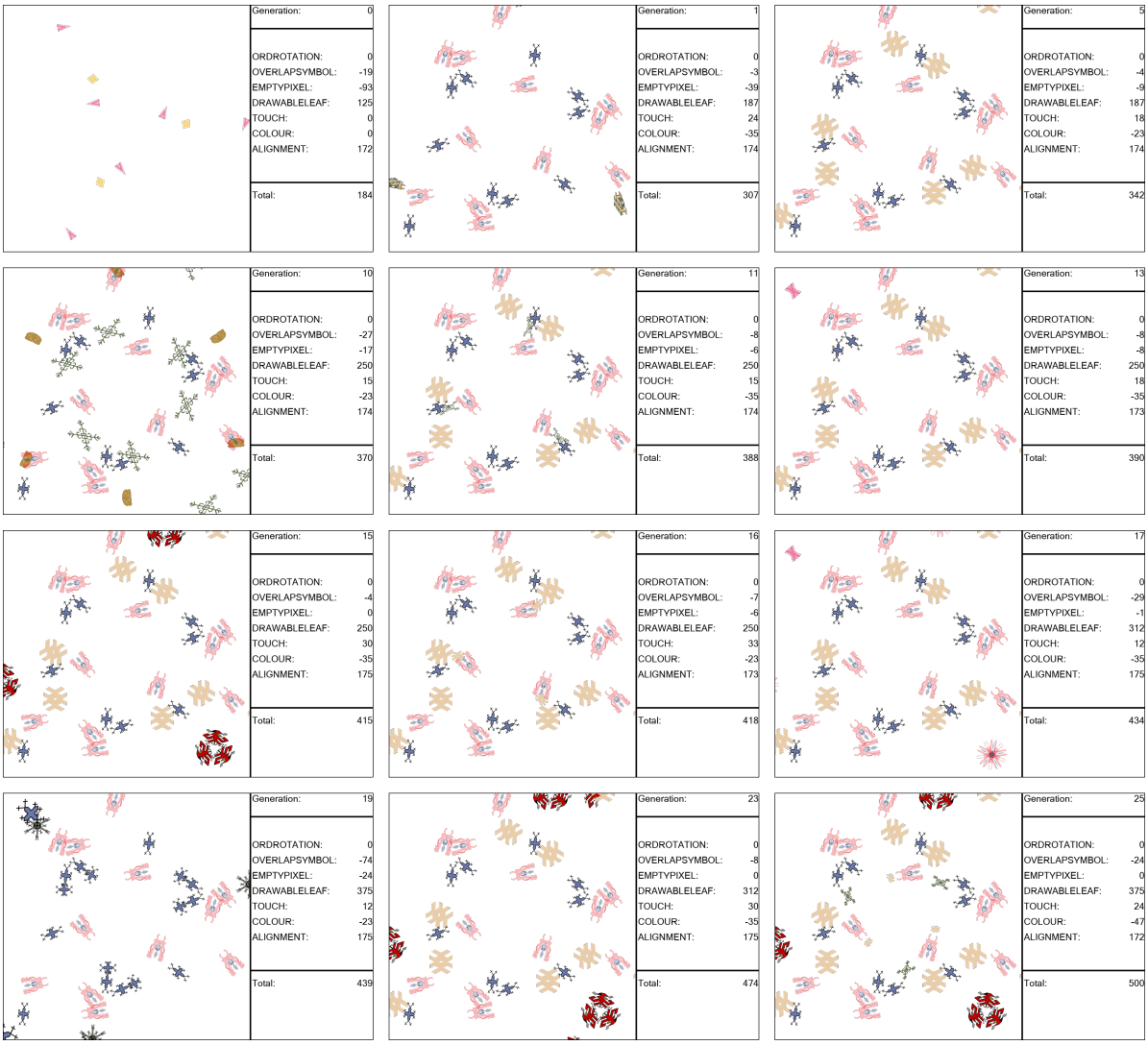
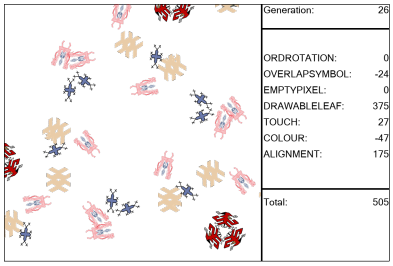


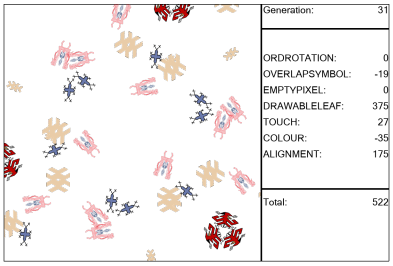
Fig. 26: Best-so-far fitness plotted against number of fitness function evaluations for each parameter setting, p_1 through p_6. After generation of the initial population, new fitness function evaluations take place when mutation and/or crossover happens, during generation of the next generation's population. Note how best-so-far fitness evaluation for p_2 and p_4 stagnate after ~ 5000 fitness evaluations, as these two parameter settings only get about half the evaluation budget as the others.

Fig. 27: Collage of the most fit canvases of each generation, generated by SymPla under parameter setting p_1. Generations for which the most fit canvas is identical to the previous generation's, are not displayed.

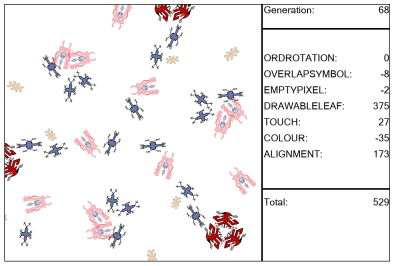




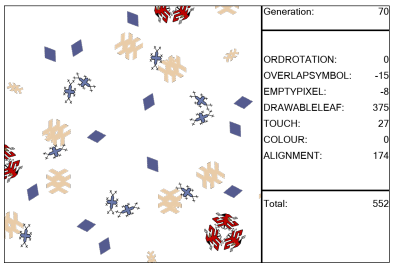
Generation:	26
ORDROTATION:	0
OVERLAPSYMBOL:	-24
EMPTYPIXEL:	0
DRAWABLELEAF:	375
TOUCH:	27
COLOUR:	-47
ALIGNMENT:	175
Total:	505



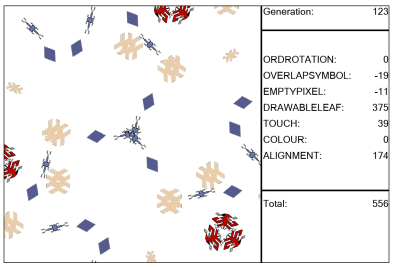
Generation:	31
ORDROTATION:	0
OVERLAPSYMBOL:	-19
EMPTYPIXEL:	0
DRAWABLELEAF:	375
TOUCH:	27
COLOUR:	-35
ALIGNMENT:	175
Total:	522



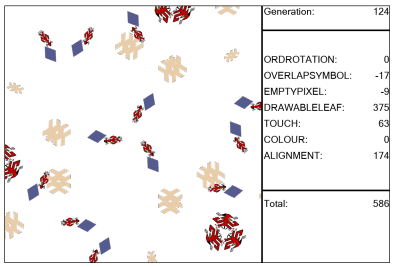
Generation:	68
ORDROTATION:	0
OVERLAPSYMBOL:	-8
EMPTYPIXEL:	-2
DRAWABLELEAF:	375
TOUCH:	27
COLOUR:	-35
ALIGNMENT:	173
Total:	529



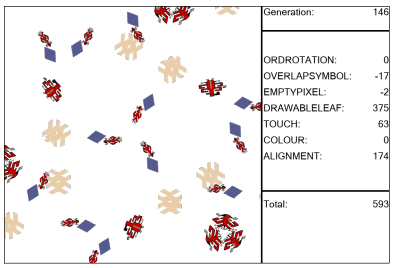
Generation:	70
ORDROTATION:	0
OVERLAPSYMBOL:	-15
EMPTYPIXEL:	-8
DRAWABLELEAF:	375
TOUCH:	27
COLOUR:	0
ALIGNMENT:	174
Total:	552



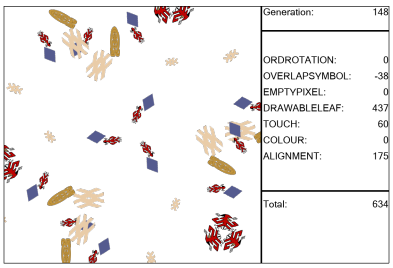
Generation:	123
ORDROTATION:	0
OVERLAPSYMBOL:	-19
EMPTYPIXEL:	-11
DRAWABLELEAF:	375
TOUCH:	39
COLOUR:	0
ALIGNMENT:	174
Total:	556



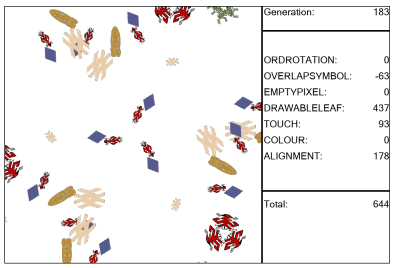
Generation:	124
ORDROTATION:	0
OVERLAPSYMBOL:	-17
EMPTYPIXEL:	-9
DRAWABLELEAF:	375
TOUCH:	63
COLOUR:	0
ALIGNMENT:	174
Total:	586



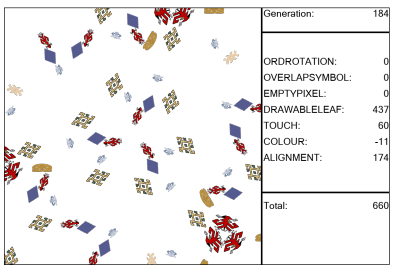
Generation:	146
ORDROTATION:	0
OVERLAPSYMBOL:	-17
EMPTYPIXEL:	-2
DRAWABLELEAF:	375
TOUCH:	63
COLOUR:	0
ALIGNMENT:	174
Total:	593



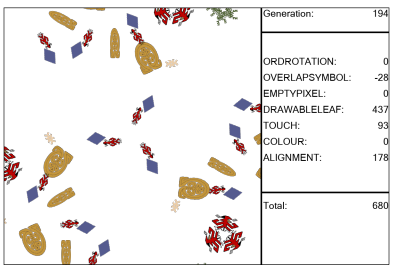
Generation:	148
ORDROTATION:	0
OVERLAPSYMBOL:	-38
EMPTYPIXEL:	0
DRAWABLELEAF:	437
TOUCH:	60
COLOUR:	0
ALIGNMENT:	175
Total:	634



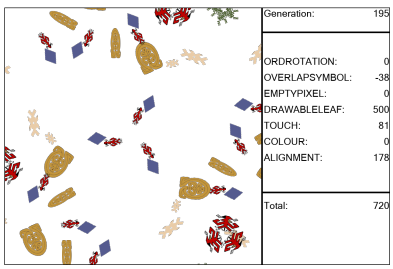
Generation:	183
ORDROTATION:	0
OVERLAPSYMBOL:	-63
EMPTYPIXEL:	0
DRAWABLELEAF:	437
TOUCH:	93
COLOUR:	0
ALIGNMENT:	178
Total:	644



Generation:	184
ORDROTATION:	0
OVERLAPSYMBOL:	0
EMPTYPIXEL:	0
DRAWABLELEAF:	437
TOUCH:	60
COLOUR:	-11
ALIGNMENT:	174
Total:	660



Generation:	194
ORDROTATION:	0
OVERLAPSYMBOL:	-28
EMPTYPIXEL:	0
DRAWABLELEAF:	437
TOUCH:	93
COLOUR:	0
ALIGNMENT:	178
Total:	680



Generation:	195
ORDROTATION:	0
OVERLAPSYMBOL:	-38
EMPTYPIXEL:	0
DRAWABLELEAF:	500
TOUCH:	81
COLOUR:	0
ALIGNMENT:	178
Total:	720

Next, to determine whether the different parameter settings affect the final fitness values, Kolmogorov-Smirnov tests [43] (specifically `scipy.stats.kstest(alternative='two-sided', mode='exact')` from SciPy v1.7.1 [50]) were run. One for each sample pair. The results can be found in Table 3. The null hypothesis is: "Two samples are drawn from the same distribution." This hypothesis does get rejected for some sample pairs, implying different parameter settings do affect fitness.

Sample pair	KS-test statistic	p-value	Reject or do not reject null hypothesis.
P_1, P_2	0.44	0.0148	Do not reject.
P_1, P_3	0.28	0.2850	Do not reject.
P_1, P_4	0.52	0.0019	Reject.
P_1, P_5	0.68	$8.4942e-06$	Reject.
P_1, P_6	0.88	$3.1010e-10$	Reject.
P_2, P_3	0.48	0.0056	Do not reject.
P_2, P_4	0.28	0.2850	Do not reject.
P_2, P_5	0.44	0.0148	Do not reject.
P_2, P_6	0.76	$2.5141e-07$	Reject.
P_3, P_4	0.52	0.0019	Reject.
P_3, P_5	0.64	$3.9640e-05$	Reject.
P_3, P_6	0.84	$3.6437e-09$	Reject.
P_4, P_5	0.32	0.1558	Do not reject.
P_4, P_6	0.6	0.0002	Reject.
P_5, P_6	0.4	0.0356	Do not reject.

Table 3: Results of running KS-tests [43] on the overall fitness values of each sample. One test for each (P_i, P_j) pair where $i, j \in (1, 2, ..., 6)$ and $i \neq j$. The null hypothesis is that the two samples are drawn from the same distribution. The alternative hypothesis is that the two samples are drawn from different distributions. The α-value is set at $\alpha = 0.05$. Applying the Bonferroni [11] correction, we reject the null hypothesis if $p \leq \frac{\alpha}{15} \approx 0.0033$.

As some of the parameters that were modified (gen, parentPop, childPop) would result in a greater or smaller evaluation budget, the decision was made to run another experiment where these parameters would remain constant. This experiment is detailed in Section 4.2.

4.2 Experiment 2

SymPla was run under three different parameter settings, p_1, p_2, p_3, all using the same seed for the random number generator to ensure results are not affected by seed difference. For each of these three settings SymPla would run 25 times, producing samples P_1, P_2, P_3. The resulting average fitness aspects' scores and overall fitness scores for these samples can be found in Table 4 and are plotted in Figures 28 and 29. for each parameter setting, the number of generations is 200, with a parent population size of 100 and a child population size of 50. Two parameters were experimented with:

1. treeDepth, the maximum depth a tree could have. Default value is 3.
2. maxArity, the maximum arity any node in any tree could have. Default value is 2.

	Parameter values		Results averaged over the runs							
	treeDepth	maxArity	f_1	f_2	f_3	f_4	f_5	f_6	Total	Time
p_1	3	2	−27	−6	+395	+102	−7	+175	+631	4073
p_2	4	2	−39	−14	+191	+195	−6	+174	+501	17010
p_3	3	3	−36	−17	+111	+198	−2	+177	+430	6614

Table 4: SymPla was run 25 times for the parameter settings p_i for $i \in (1, 2, 3)$ on the left, producing samples P_i for $i \in (1, 2, 3)$. The average fitness aspect scores and overall fitness for these samples are in the right columns. Fitness aspects are as detailed in Section 3: f_1 is symbol overlap. f_2 is pixel count. f_3 is amount of drawn leaves. f_4 is symbol touch. f_5 is colour variety. f_6 is symbol alignment. The corresponding weights can be found in Table 1. A fitness aspect's score can take a value anywhere between zero and a hundred, multiplied by its corresponding weight. The final column, Time, is the average time taken per run of the corresponding sample. Time is expressed in seconds.

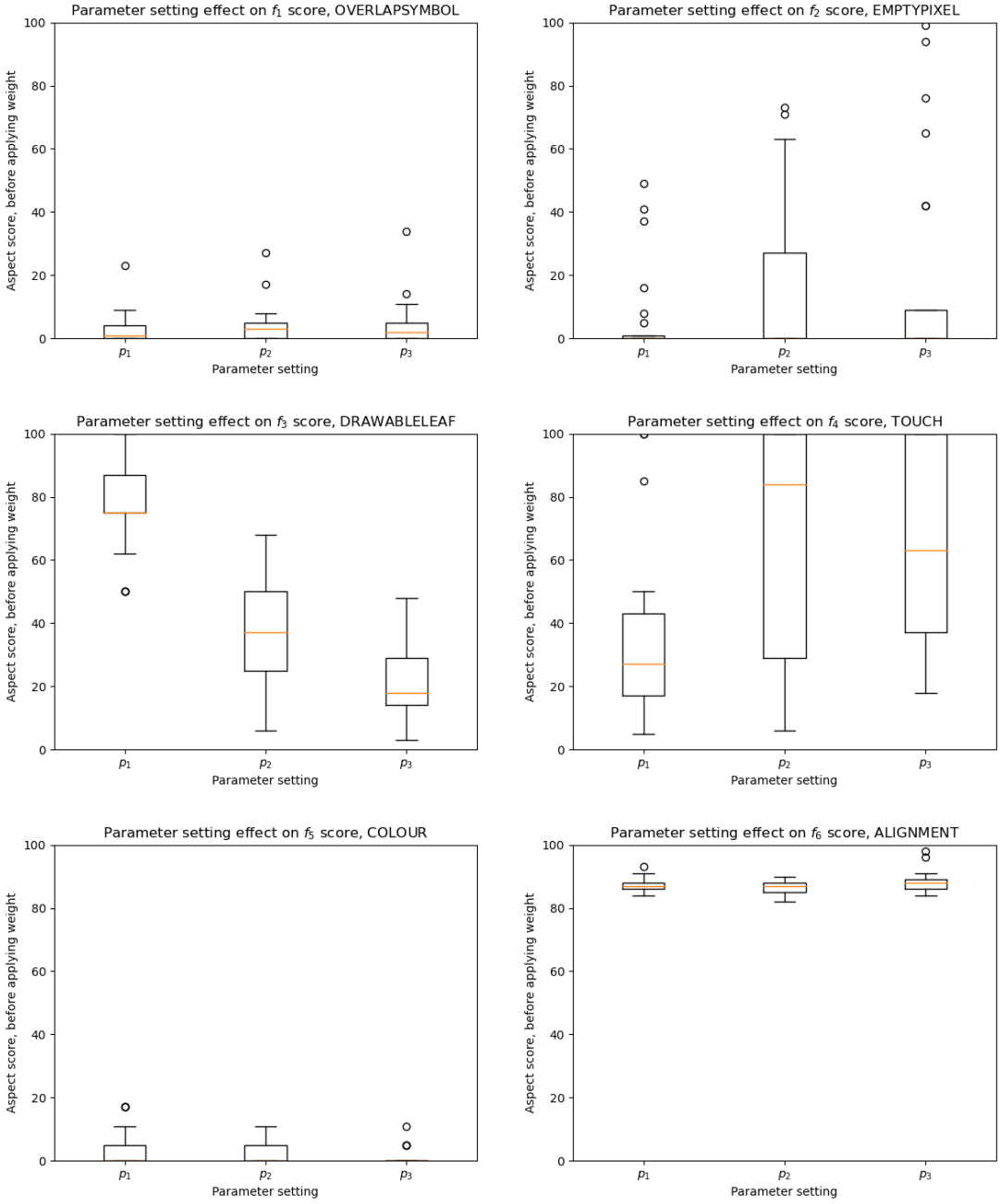


Fig. 28: Box-plots for each fitness aspect. Each box represents the fitness aspect's score, as indicated on the vertical axis, for the most fit individual of 25 runs given the specific parameter setting, p_1 through p_3, as indicated on the horizontal axis.

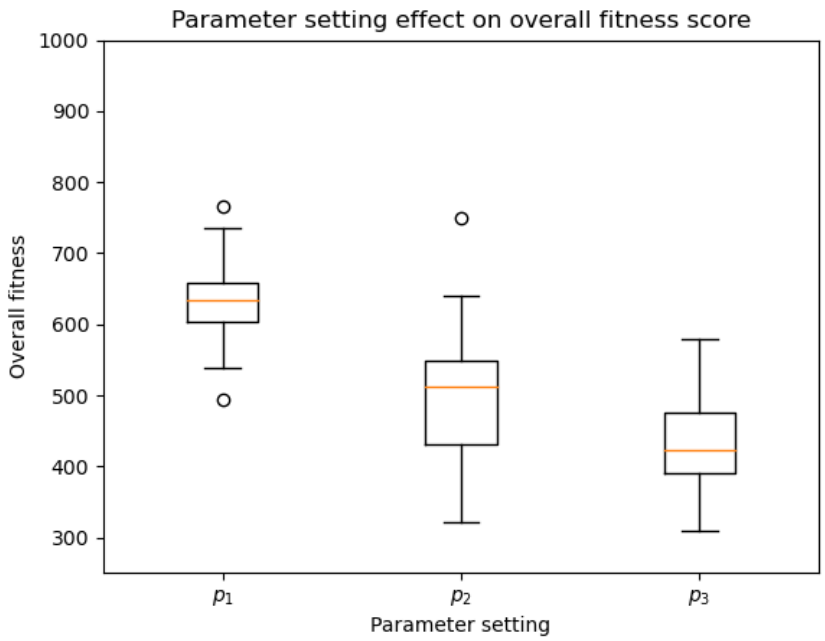


Fig. 29: Box-plot for overall fitness. Each box represents the fitness score, as indicated on the vertical axis, for the most fit individual of 25 runs given the specific parameter setting, p_1 through p_3, as indicated on the horizontal axis. Using the weights as specified in Table 1, fitness can take a value in $[-1100, +1000]$.

Next, to determine whether the different parameter settings affect the final fitness values, Kolmogorov-Smirnov tests [43] (specifically `scipy.stats.kstest(alternative='two-sided', mode='exact')` from SciPy v1.7.1 [50]) were run. One for each sample pair. The results can be found in Table 3.

Sample pair	KS-test statistic	p-value	Reject or do not reject null hypothesis.
P_1, P_2	0.68	$8.4942e-06$	Reject.
P_1, P_3	0.88	$3.1010e-10$	Reject.
P_2, P_3	0.4	0.0356	Do not reject.

Table 5: Results of running KS-tests [43] on the overall fitness values of each sample. One test for each (P_i, P_j) pair where $i, j \in (1, 2, 3)$ and $i \neq j$. The null hypothesis is that the two samples are drawn from the same distribution. The alternative hypothesis is that the two samples are drawn from different distributions. The α-value is set at $\alpha = 0.05$. Applying the Bonferroni [11] correction, we reject the null hypothesis if $p \leq \frac{\alpha}{3} \approx 0.017$.

In Table 5 the null hypothesis is rejected when comparing P_1 to P_2 and P_1 to P_3, yet when comparing P_2 to P_3 the null hypothesis is not rejected. This implies that different parameter settings do affect fitness. The decision was made to run SymPla another 100 times under parameter setting p_1. The resulting sample, P_1', had average aspect scores of $f_1 = -33$, $f_2 = -8$, $f_3 = +377$, $f_4 = +126$, $f_5 = -7$, $f_6 = +175$, an average total fitness of $+629$ and an average time per run of 3533 seconds.

By personally looking at P_1's generated canvases, a distinction was made between "pleasing" and "non-pleasing" canvases. 21 were deemed as "pleasing", 79 were deemed to be "non-pleasing". Figures 31 and 32 in Appendix A and Appendix B display these canvases. To determine if the difference between pleasing and non-pleasing canvases is captured by any or multiple of the fitness aspects, a boxplot was made, see Figure 30. more KS-tests were run as well, see Table 6.

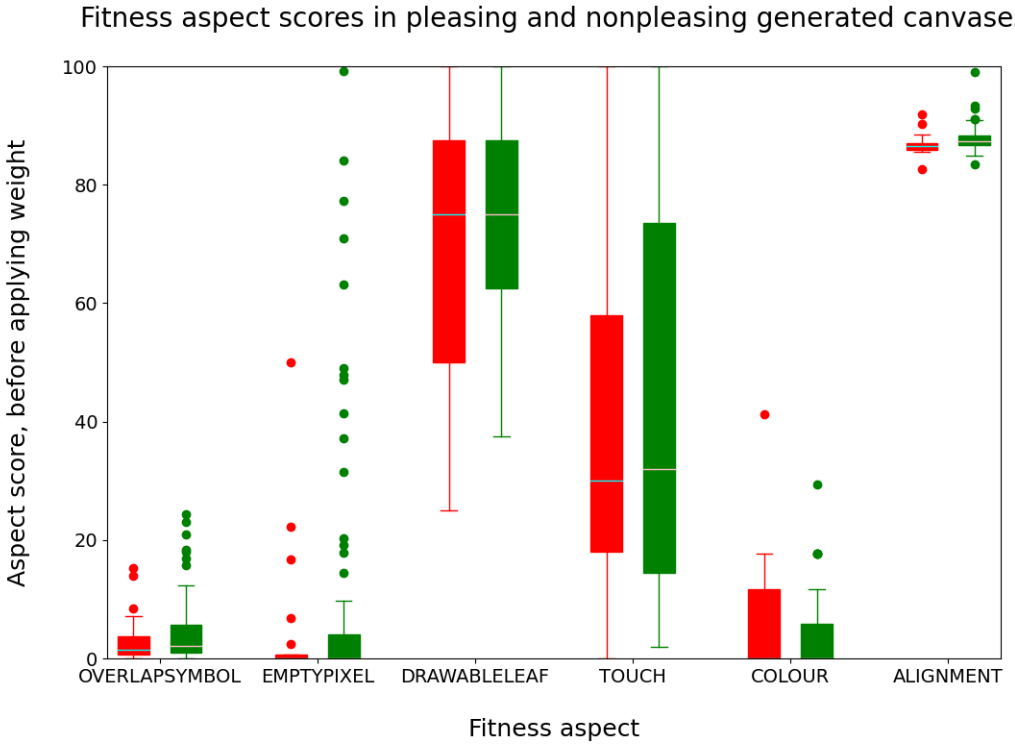


Fig. 30: Boxplot comparing values of fitness aspect scores in pleasing and non-pleasing canvases. Red represents pleasing images. Green represents non-pleasing images. The horizontal line in each candle represents the median value.

Fitness aspect	KS-test statistic	p-value	Reject or do not reject null hypothesis.
f_1, OVERLAPSYMBOL	0.1820	0.5708	Do not reject.
f_2, EMPTYPIXEL	0.0916	0.9957	Do not reject.
f_3, DRAWABLELEAF	0.1549	0.7535	Do not reject.
f_4, TOUCH	0.1447	0.8199	Do not reject.
f_5, COLOUR	0.1844	0.5546	Do not reject.
f_6, ALIGNMENT	0.3822	0.0110	Do not reject.

Table 6: Results of running KS-tests [43] on the aspect scores for pleasing and non-pleasing canvases, as depicted in Figure 30. One test for each f_i fitness aspect, where $i \in (1, ..., 6)$. For each fitness aspect, the null hypothesis is that the pleasing and nonpleasing canvases draw their scores from the same distribution. For each fitness aspect, the alternative hypothesis is that the pleasing and nonpleasing canvases draw their scores from different distributions. The α-value is set at $\alpha = 0.05$. Applying the Bonferroni [11] correction, we reject the null hypothesis if $p \leq \frac{\alpha}{6} \approx 0.008$.

As the null hypothesis is not rejected for any fitness aspect, f_i where $i \in (1, .., 6)$, in Table 6, the implication is raised that none of the fitness aspects' scores, affect whether a canvas is pleasing or nonpleasing. The similarity of the candles in the boxplot of Figure 30 supports this conclusion.

Chapter 5

Conclusion

In this thesis, genetic programming and multiple aesthetic measures were implemented in SymPla, which, by transforming and placing symbols on an initially clear canvas, sets out to answer the research question (1): "Can genetic programming be used to compose and evolve ornaments on a canvas?" as well as (2): "Can genetic programming be used to evolve quantifiable aesthetic measures?"

The different fitness aspects used as aesthetic measures to determine the aesthetic pleasure a generated canvas would provide, do not all get consistently fully optimised under default parameters, as shown in Figure 30. Note that OVERLAPSYMBOL, EMPTYPIXEL and COLOUR are rewarded for minimising their aspect score, unlike DRAWABLELEAF, TOUCH and ALIGNMENT which are rewarded for maximising their aspect score.

- As shown by the multitude of outliers, especially simple to get wrong seems to be EMPTYPIXEL, which punishes the canvas for either having too little or too much content. Granted, the median value is still zero. COLOUR also has a median value of zero, with fewer outliers but more scores in the relatively small interquartile range.
- DRAWABLELEAF, which in theory would be simple to optimise by ensuring a tree's every node has its arity set to max, still commonly does not reach score 100. This might also be due to leaves in the tree which do not end up getting drawn on the canvas.
- Despite having a very high average fitness score of > 80, ALIGNMENT generally varies very little. There is a very small interquartile range with few outliers. SymPla seems to have difficulty optimising this arguably nebulous aesthetic measure. Like OVERLAPSYMBOL and COLOUR, the different parameter settings don't seem to have much of an effect on ALIGNMENT either, as can be seen in Figure 28.
- TOUCH seems to highly vary in both pleasing and nonpleasing images according to Figure 30. Unlike DRAWABLELEAF's high variation, TOUCH' median is quite a bit lower.

To see how well the implemented aesthetic measures, or fitness aspects, capture aesthetic beauty, the author used their own subjective appreciation of the generated canvases to classify pleasing and non-pleasing canvases, as depicted in Appendix A and Appendix B. Ultimately, as shown in Table 6, no significant difference was found in any of the fitness aspects between pleasing and non-pleasing images, implying that the fitness aspects do not reflect at least the author's interpretation of aesthetic beauty.

Whereas Appendix A and Appendix B aim to glean value from the aesthetic measures used, Figure 27 in Section 4.1 and figs. 33 to 37 in Appendix C set out to answer question (1) and question (2) by showcasing how canvases develop over the generations.

Chapter 6

Discussion & Future Work

Whether by use of genetic programming, or evolutionary computation in general, many (successful) attempts have been made by others to create applications or programs which can generate images to inspire people [42, 7, 6]. Something this thesis did not pursue is evaluate whether the canvases generated by SymPla can inspire people. Camera Obscurer [42] is a work that also aims at inspiring people, and a crowdsource experiment was set up to test its efficacy. Perhaps a setup similar to the one of this experiment could be used to evaluate SymPla in future work.

The posed research questions (1) and (2) are not explicitly answered in Section 5. Instead, Appendix C and Figure 27 are referenced and hailed as answer. Appendix C does go into how canvases develop over time, but offers little deeper interpretation.

Appendix A and Appendix B originally set out to answer a previous research question, "Can genetic programming be used to inspire artists or designers?" This question was dropped as Appendix A and Appendix B would not provide an adequate answer.

Something that could have provided more insight is testing values lower than the default values for parameter settings in Section 4. Of the parameter settings mentioned in Section 4, p_2 uses a higher treeDepth than default, yet there was no parameter setting using a value for treeDepth lower than the default. The same goes for maxArity and p_3, where p_3 sets maxArity greater than default, but no similar parameter setting, using a value lower than the default was run.

Another potentially missed opportunity lies in the lack of incorporating different crossover and mutation rates in the parameter settings presented in Section 4.

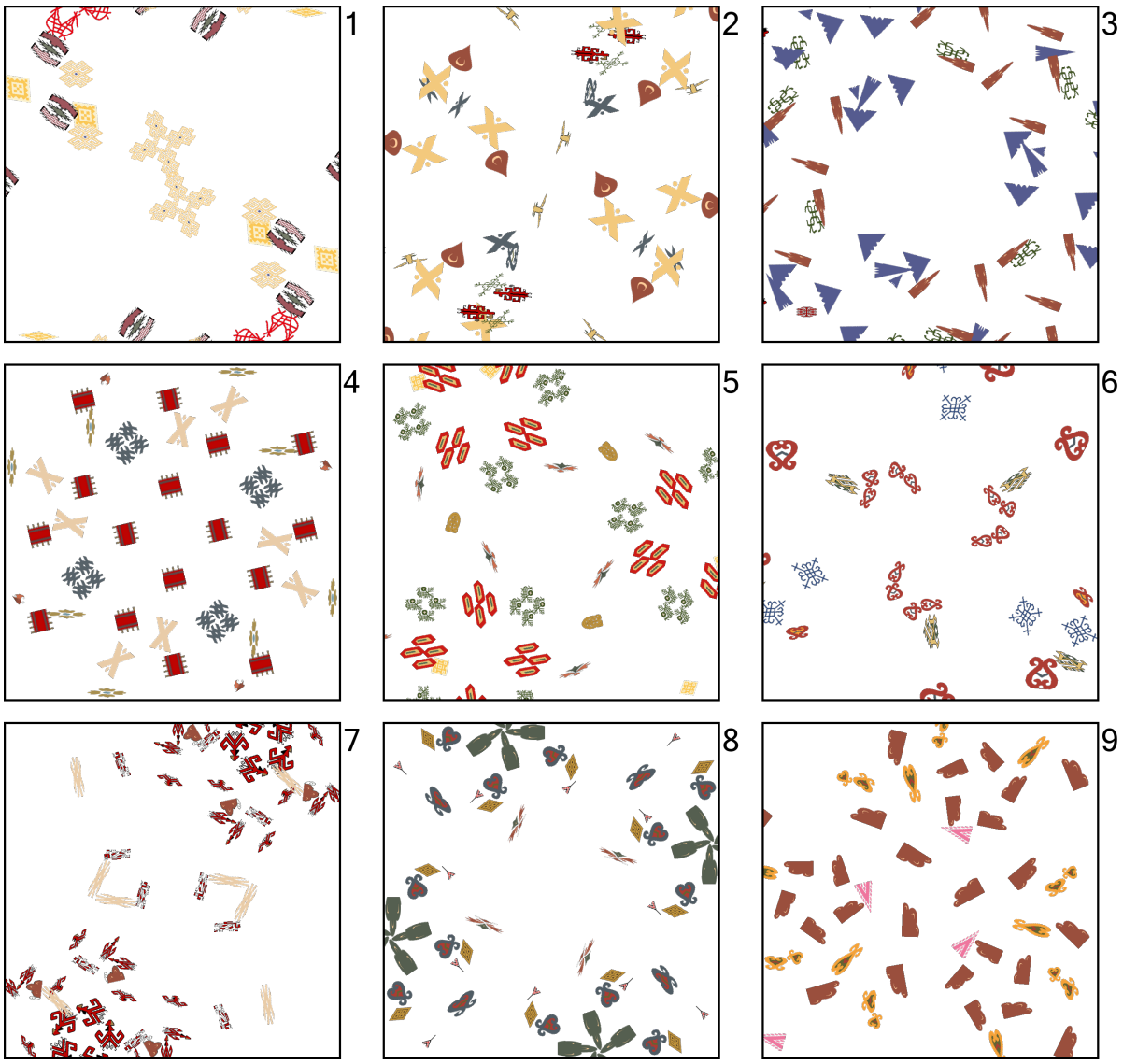
Some of the data in Section 4 can be misleading. Of the fitness aspects, f_3, DRAWABLELEAF, is biased towards the default parameter settings, p_1. Both an increased treeDepth or maxArity increases the amount of leaves a tree can have, which also means a greater chance for at least one of those leaves to not be drawn on the canvas. This is reflected by the results in Figure 28, where p_1 has the highest median value for DRAWABLELEAF. At the opposite end of the spectrum is f_4, TOUCH, which is biased towards p_2 and p_3 instead, as more complex trees means more symbols drawn on the canvas, which increases the likelihood of symbols touching one another.

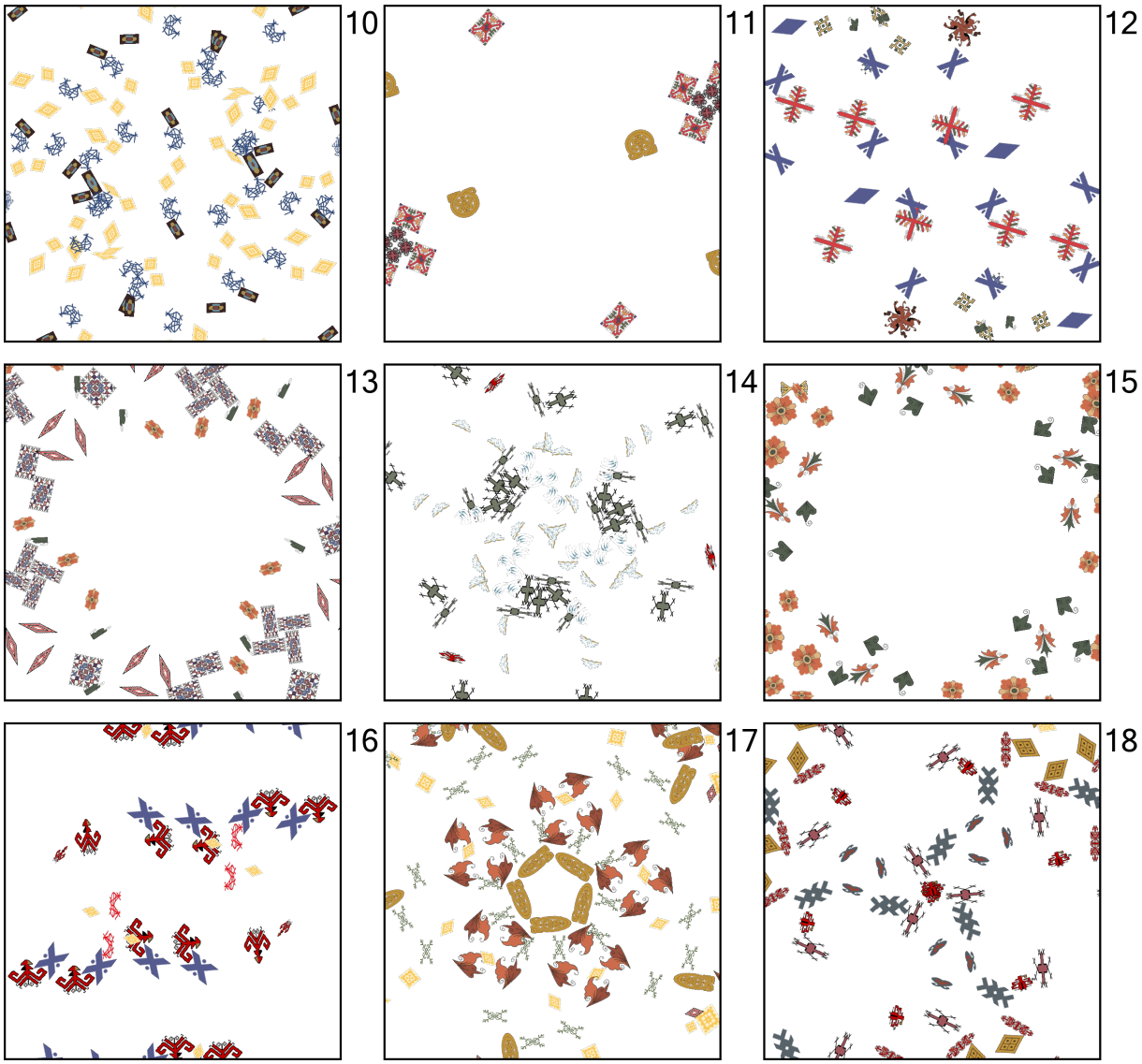
Regarding the fitness aspect for touching symbols, as mentioned in Section 3.2.5, Tree1, Figure 9, $f_4 = 0$. The rightmost symbol starts at (100, 50), after the leftmost symbol ends at (99, 150), therefore these symbols are considered to not be touching. Instead of TOUCH requiring an overlap of exactly one pixel, it could be argued that if a pixel of one symbol is next to a pixel of another, these two symbols should be considered as touching. If adapting this new definition of TOUCH, symbol pair (S_1, S_5) in Tree3, Figure 13, as mentioned in Section 3.2.5, would not be considered as touching as there is a one-pixel overlap of pixels that are mostly blending into the transparent background. In that case it might be better if overlap is determined if alpha $>$ some value greater than 0 instead of alpha > 0.

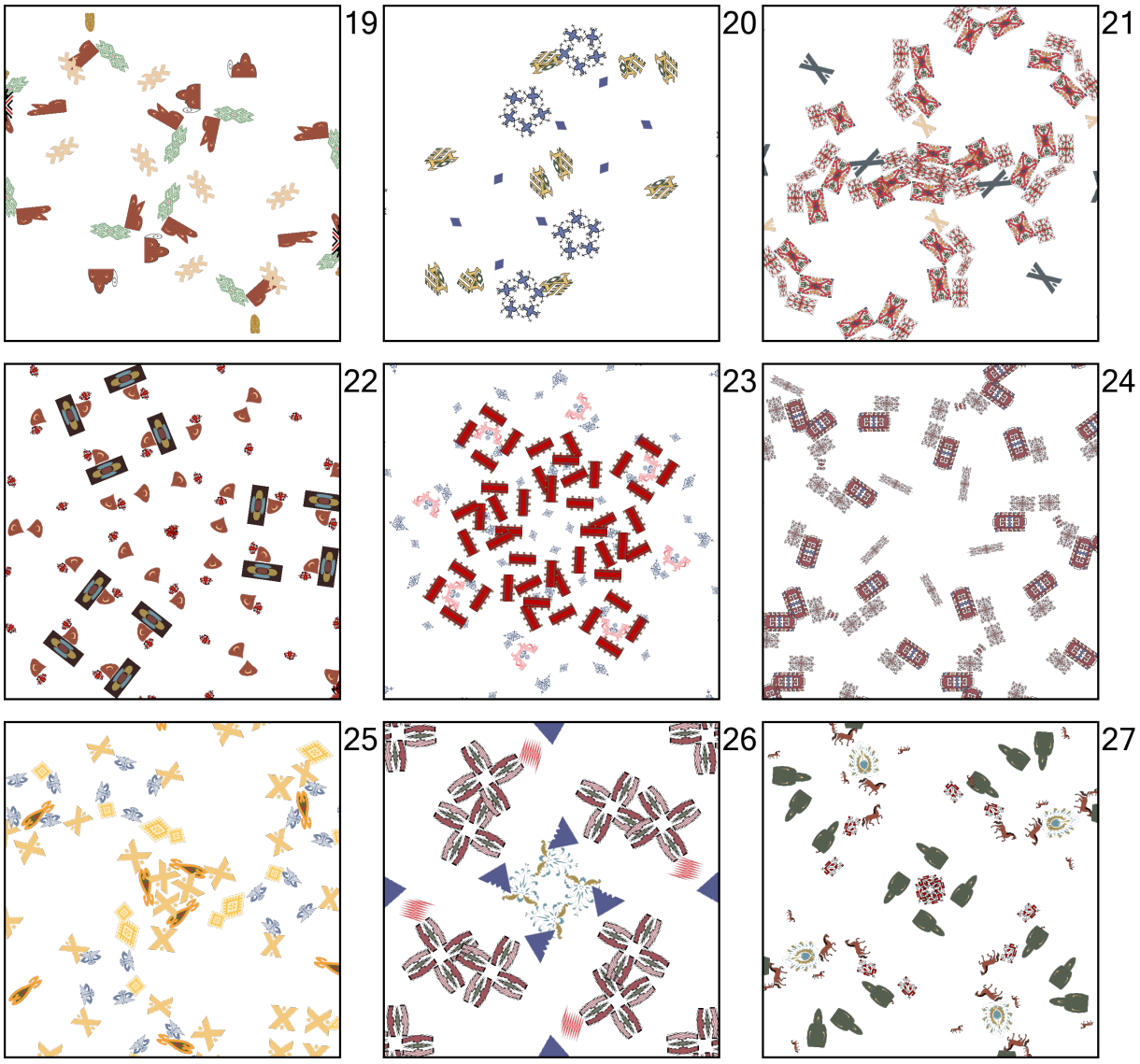
Appendix A: Non-pleasing Images

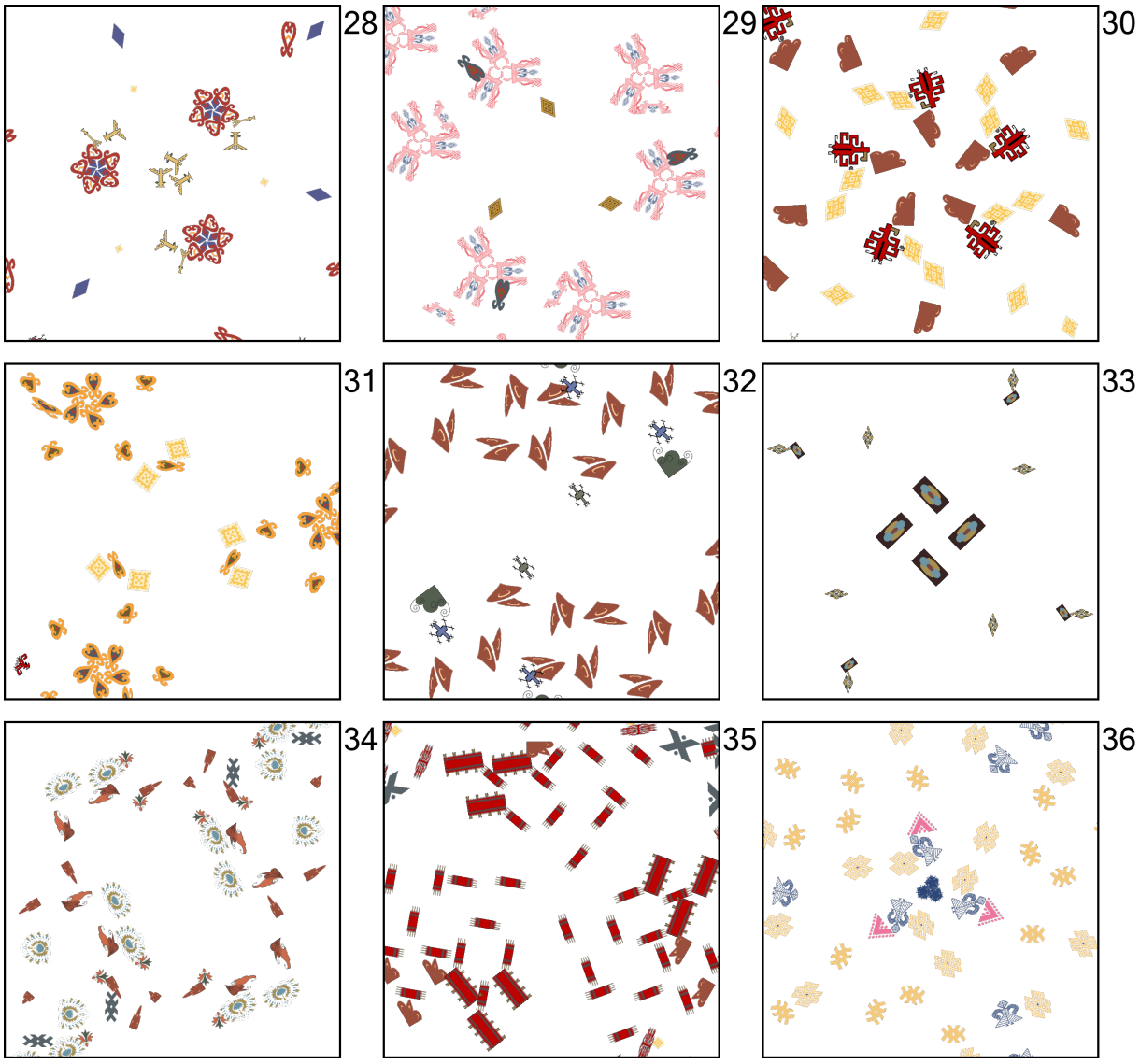
SymPla was run under p_1 one-hundred times. Each run's most-fit individual was collected. Of these 100 canvases, 21 were "pleasing" canvases and 79 were "non-pleasing". "Pleasing" and "non-pleasing" are defined as by the author's own taste. Figure 31 displays the 79 "non-pleasing" canvases. Certain canvases look "non-pleasing" due to large empty areas on the canvas, such as image 1, 7, 11 and 46 in Figure 31. Other images are very busy, such as image 62 and 72.

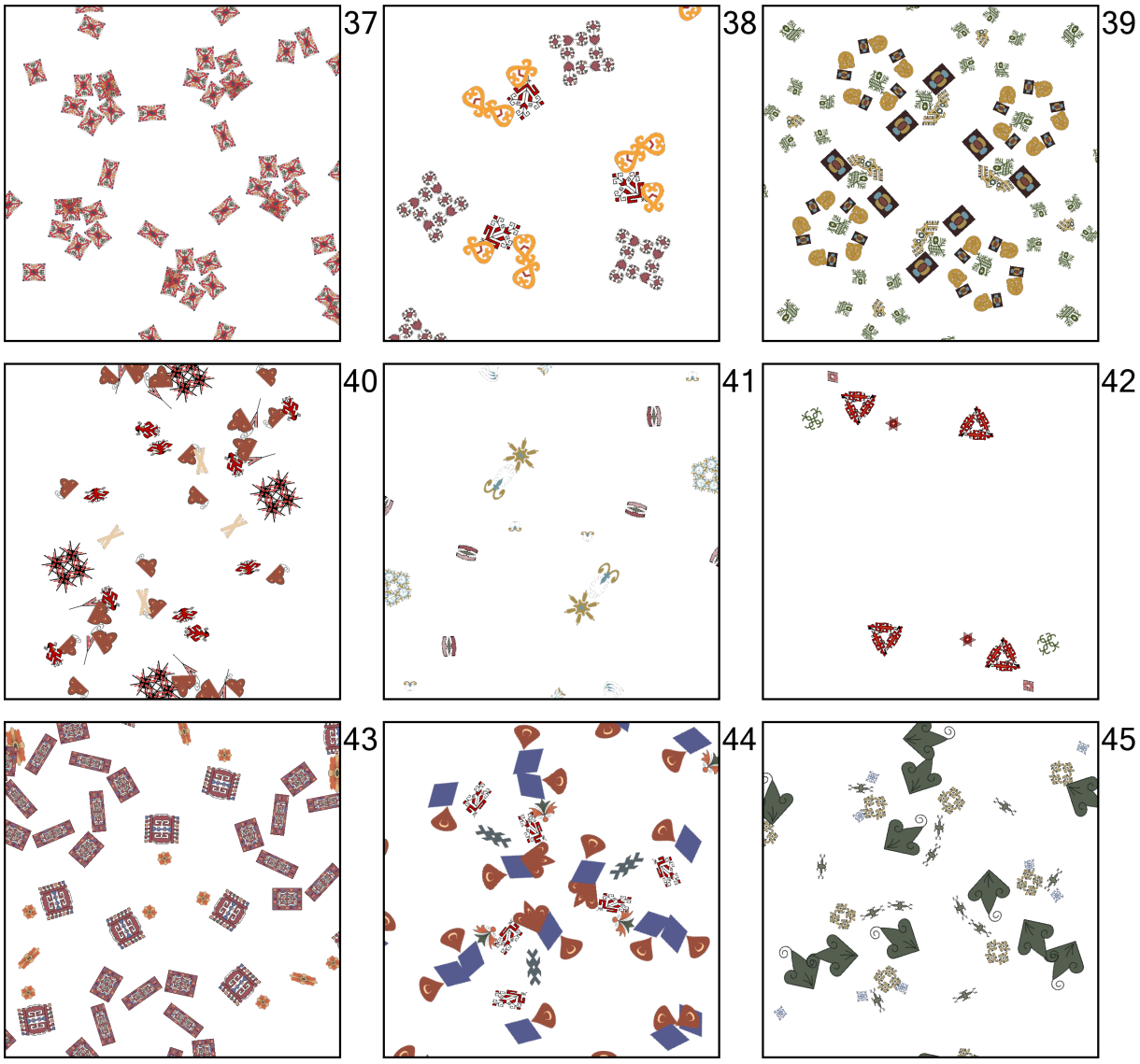
Fig. 31: The 79 "non-pleasing" canvases are displayed here. Sorted by ascending order of generation.

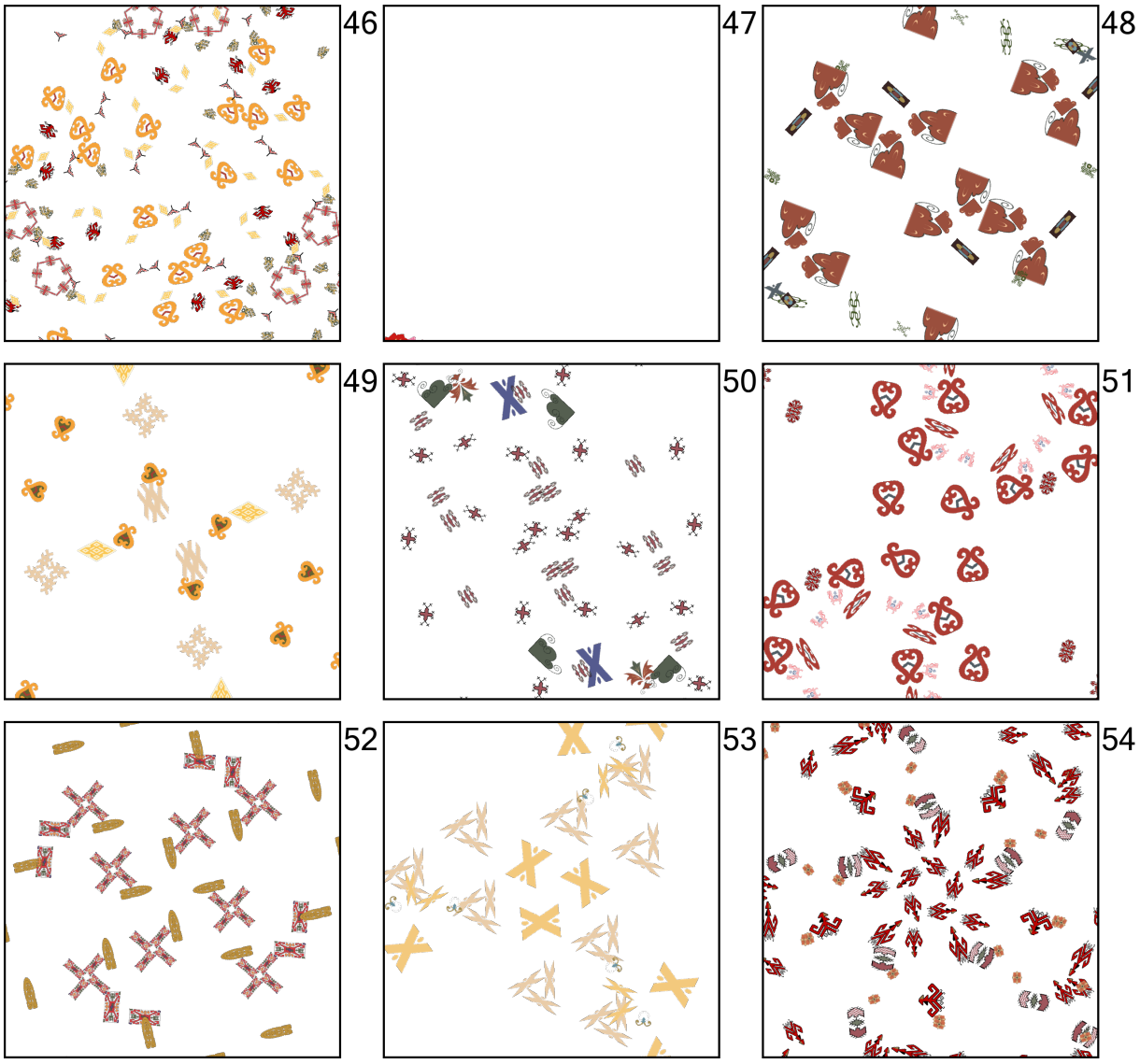


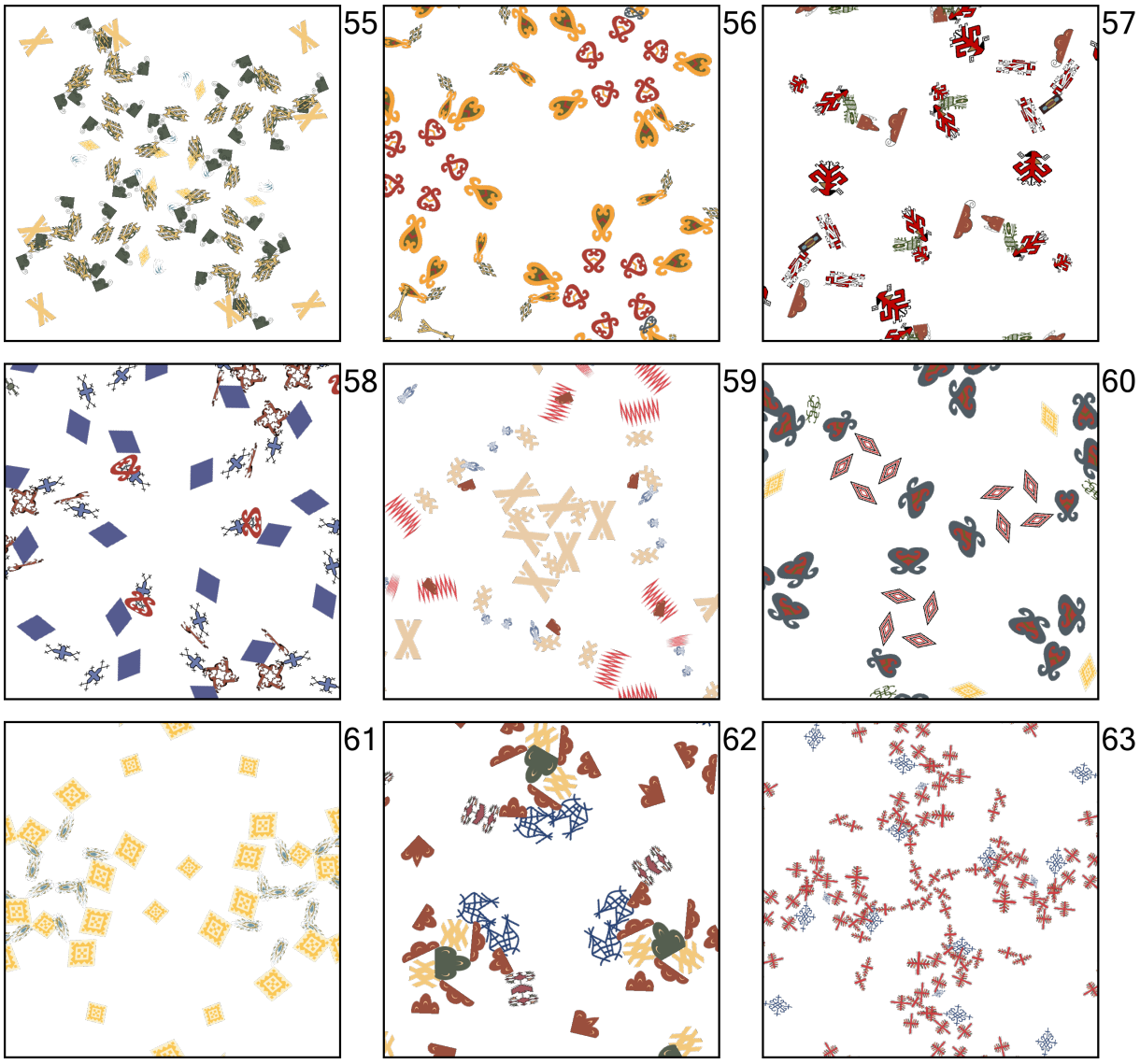

10
11
12
13
14
15
16
17
18

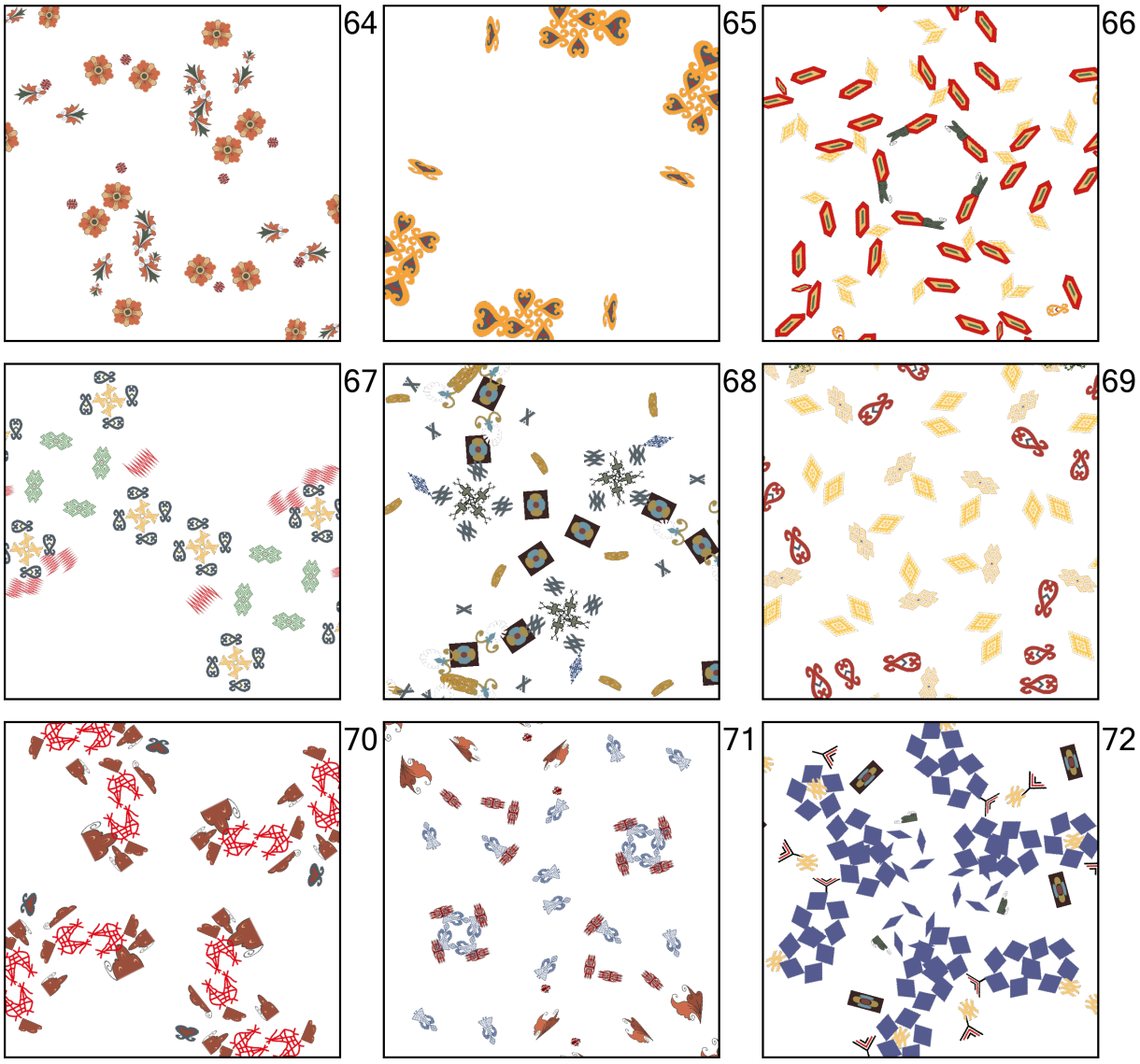

19
20
21
22
23
24
25
26
27

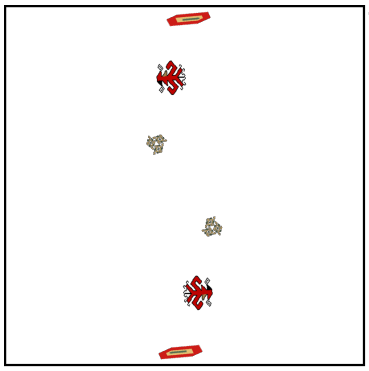

28
29
30
31
32
33
34
35
36

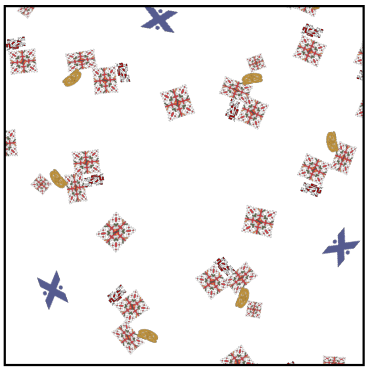

37
38
39
40
41
42
43
44
45

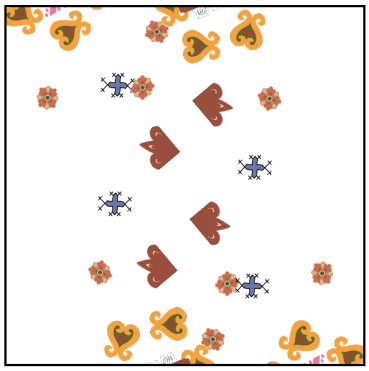

46
47
48
49
50
51
52
53
54

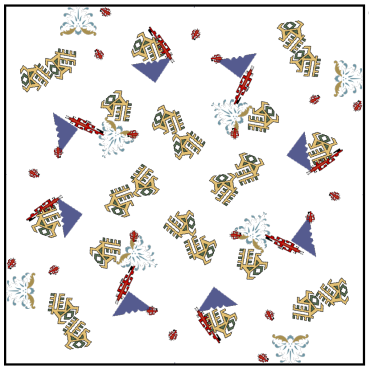

55
56
57
58
59
60
61
62
63

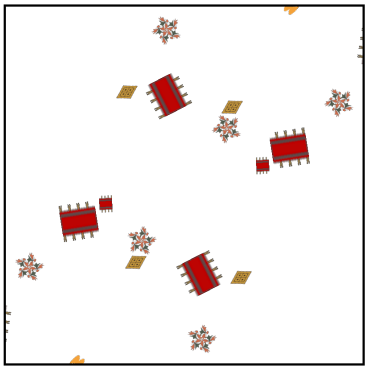

64
65
66
67
68
69
70
71
72

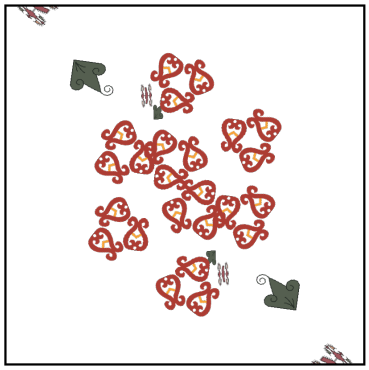

73

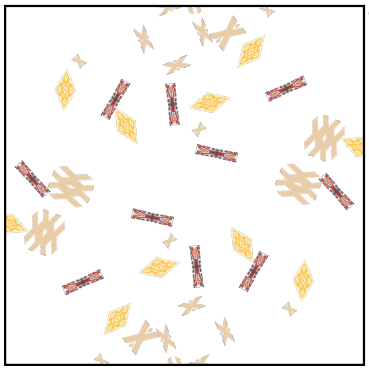

74


75


76


77

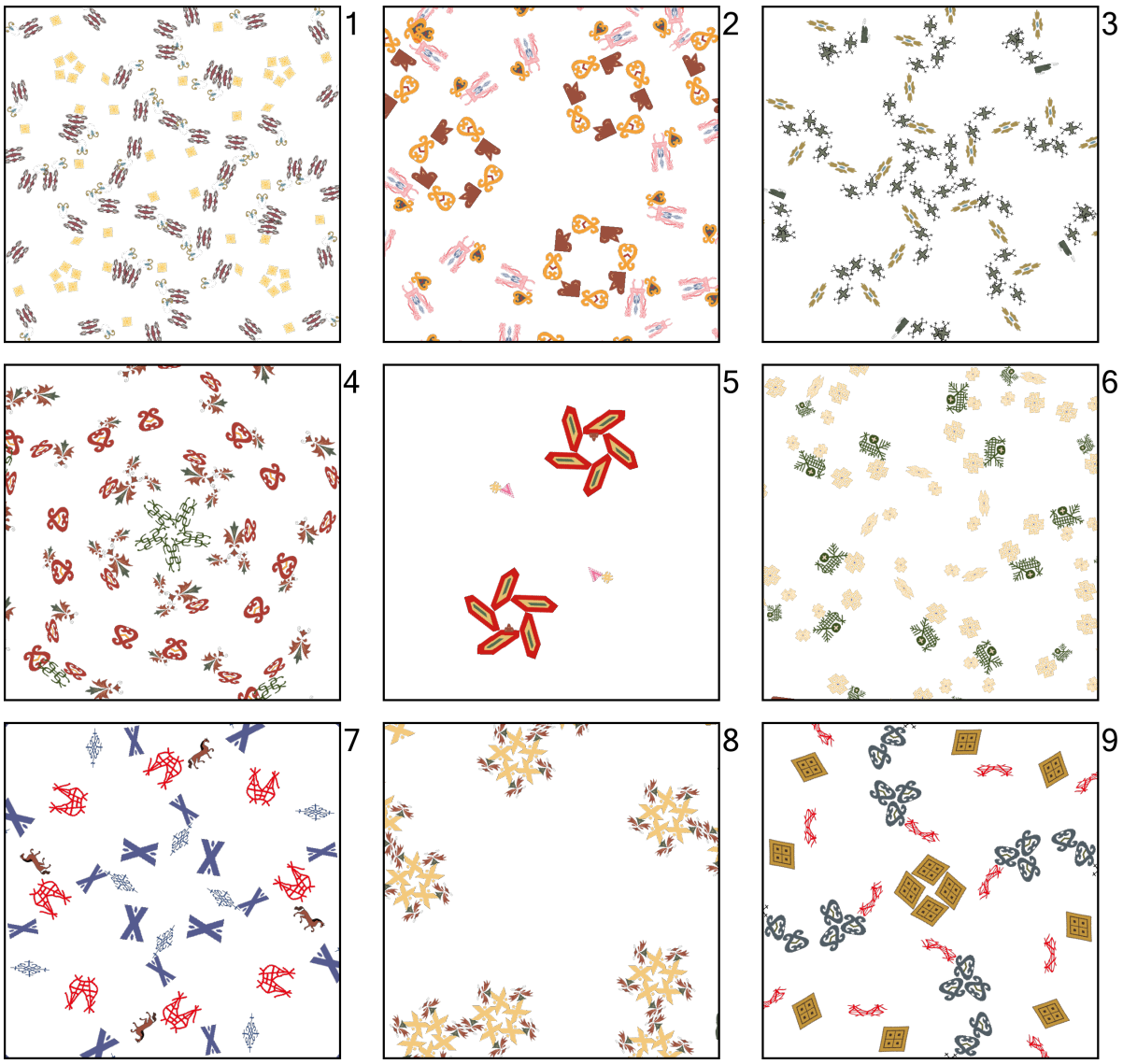

78

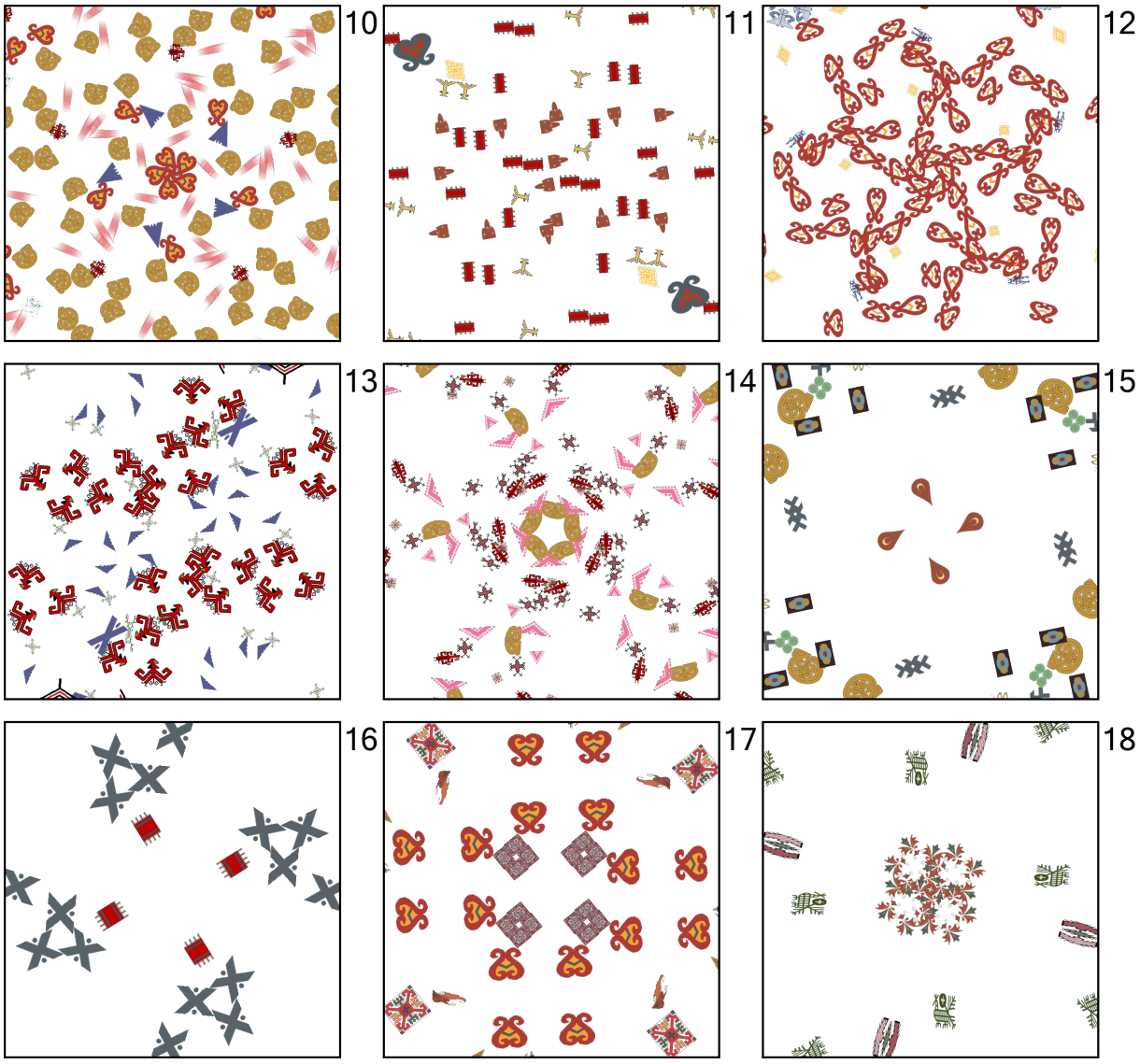

79

Appendix B: Pleasing Images

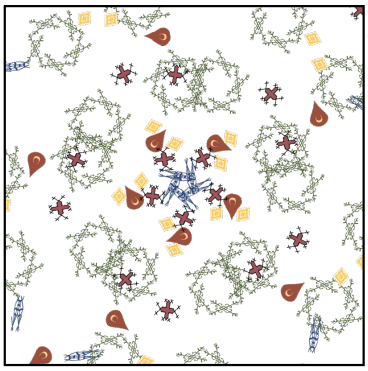
SymPla was run under p_1 one-hundred times. Each run's most-fit individual was collected. Of these 100 canvases, 21 were "pleasing" canvases and 79 were "non-pleasing". "Pleasing" and "non-pleasing" are defined as by the author's own taste. Figure 32 displays the 21 "pleasing" canvases. Certain canvases look "pleasing" due to 'full' shapes forming, such as the three circlish shapes in canvas 2, the two 'buzz-saws' in canvas 5 or the 'shurikenesque' shapes in canvas 8. Other canvases seem structured, such as canvas 10, 17 and 19. Finally, some canvases seem to represent whirlpools, such as canvas 4, 14 and 21.

Fig. 32: The 21 "pleasing" canvases are displayed here. Sorted by ascending order of generation.

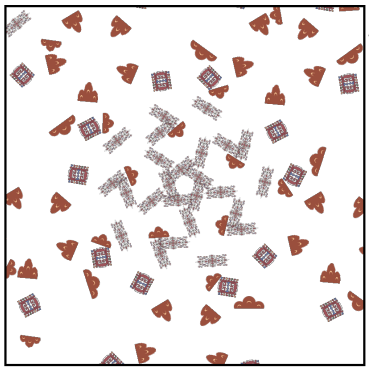



10
11
12
13
14
15
16
17
18

19



20



21

Appendix C: Evolving Images

As part of visualising the evolution of canvases by SymPla, collages of a run of each parameter setting p_1 through p_6, of Experiment 1, Section 4.1, were made. Figure 27 in Section 4.1 and Figures 33 to 37 display these collages.

Fig. 33: Collage of the most fit canvases of each generation, generated by SymPla under parameter setting p_2, detailed in Section 4.1. Generations for which the most fit canvas is identical to the previous generation's, are not displayed. Note how generation 93's most fit canvas has a lower overall fitness than the previous most fit canvas, shown at generation 47. This is due to restarting: When the most fit canvas remains the same for too many generations the population is cleared and generated anew.

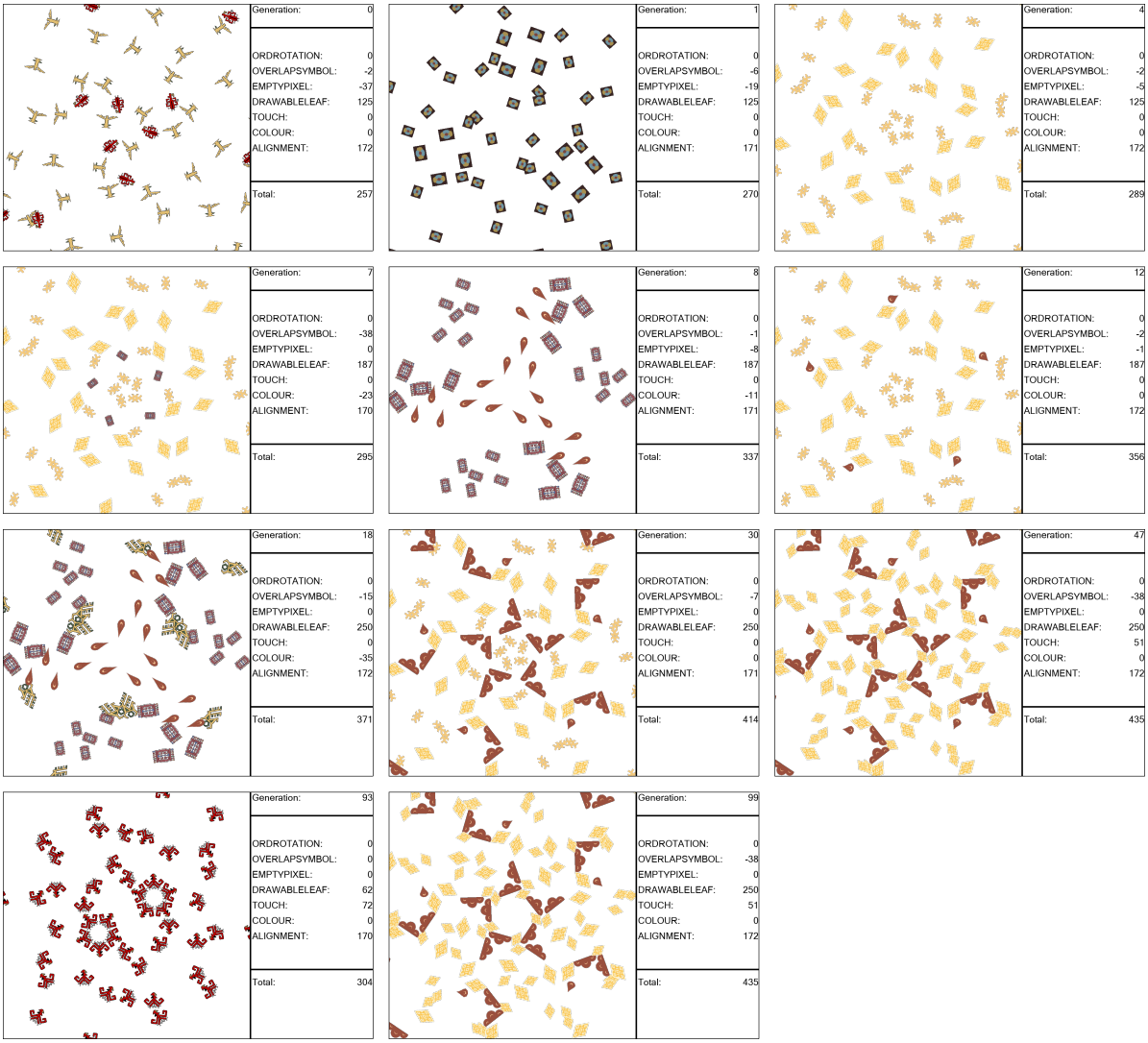
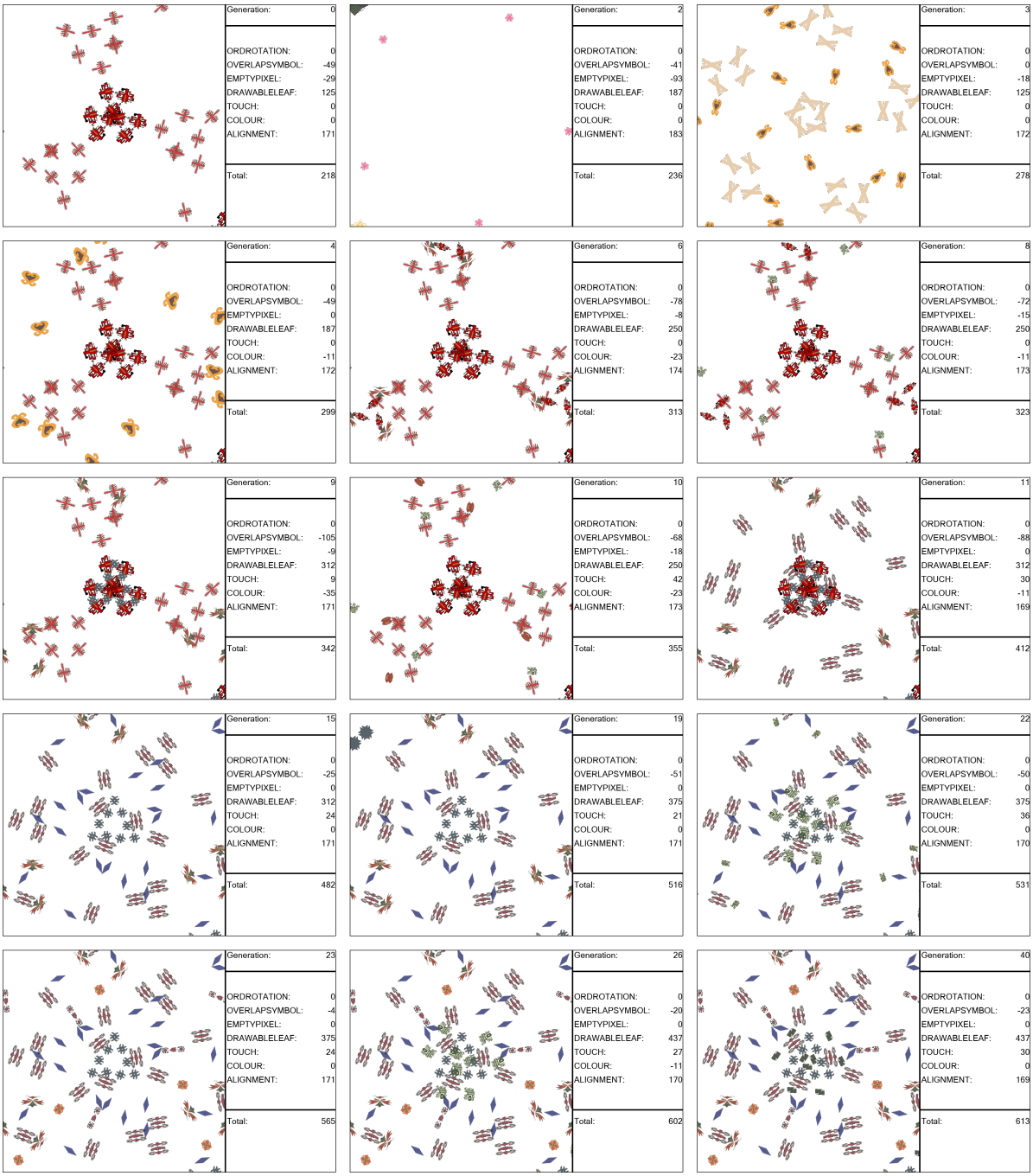


Fig. 34: Collage of the most fit canvases of each generation, generated by SymPla under parameter setting p_3, detailed in Section 4.1. Generations for which the most fit canvas is identical to the previous generation's, are not displayed.



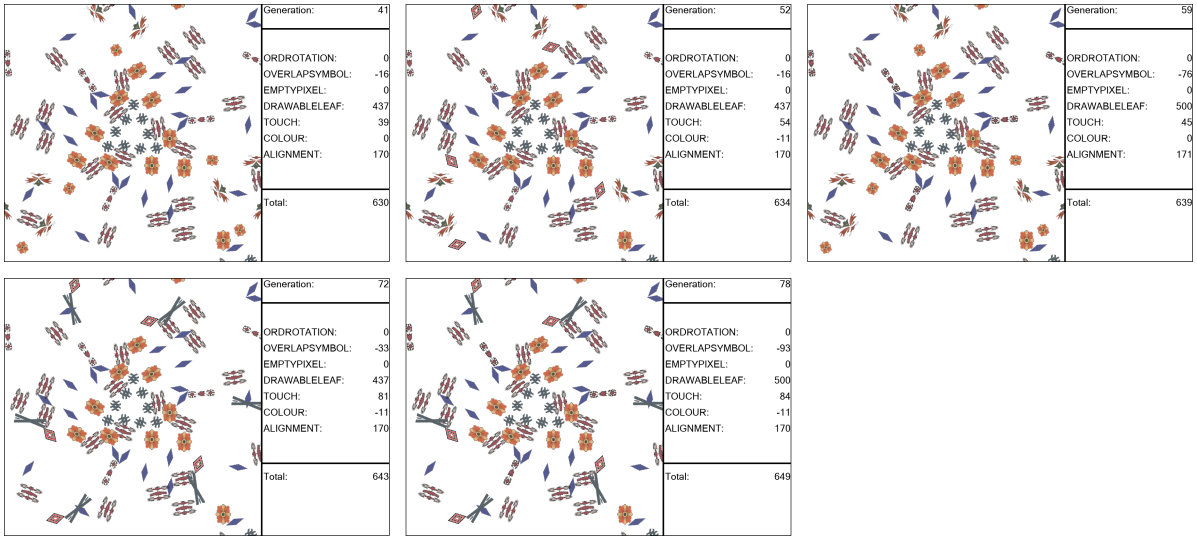
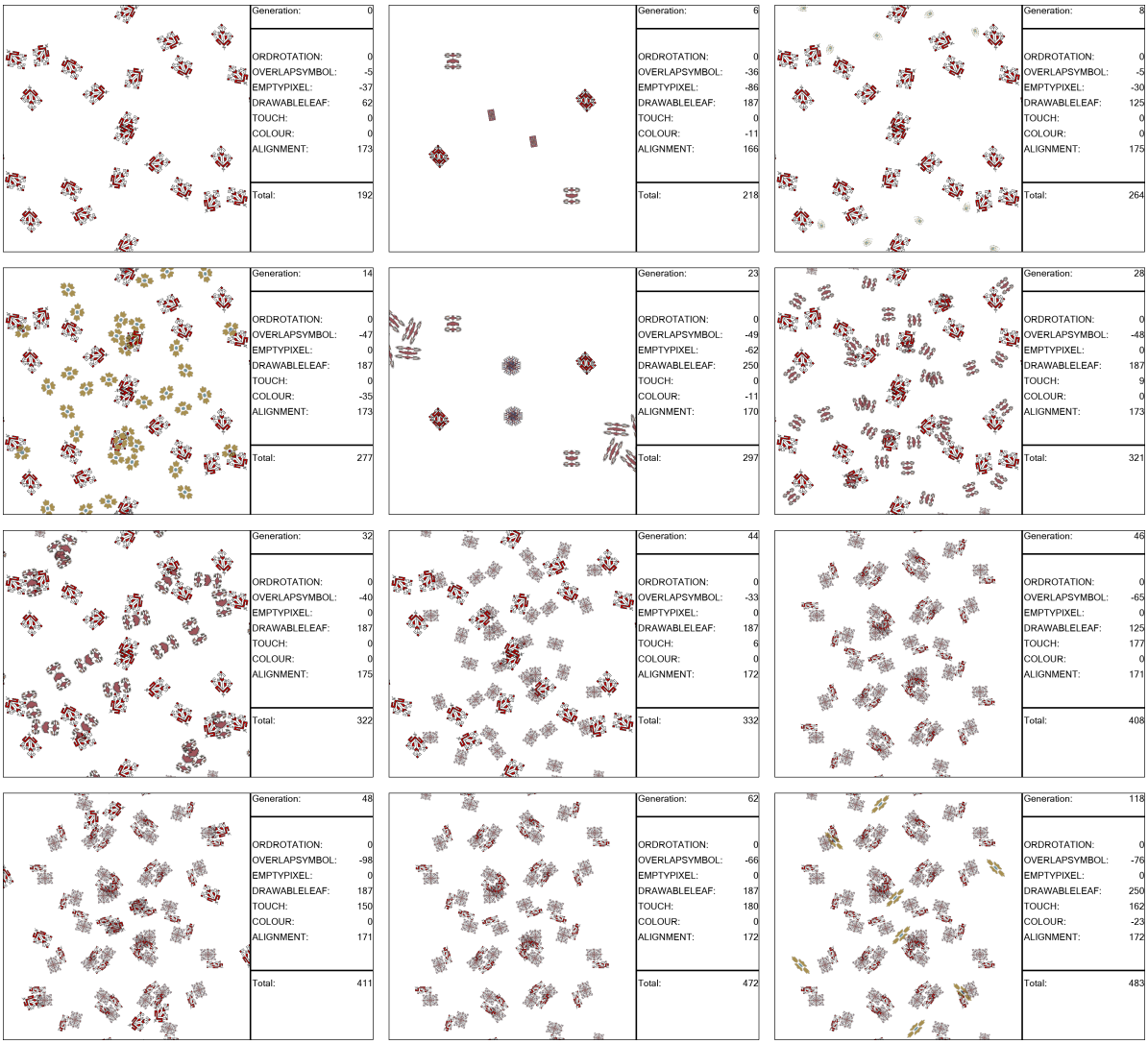


Fig. 35: Collage of the most fit canvases of each generation, generated by SymPla under parameter setting p_4, detailed in Section 4.1. Generations for which the most fit canvas is identical to the previous generation's, are not displayed.



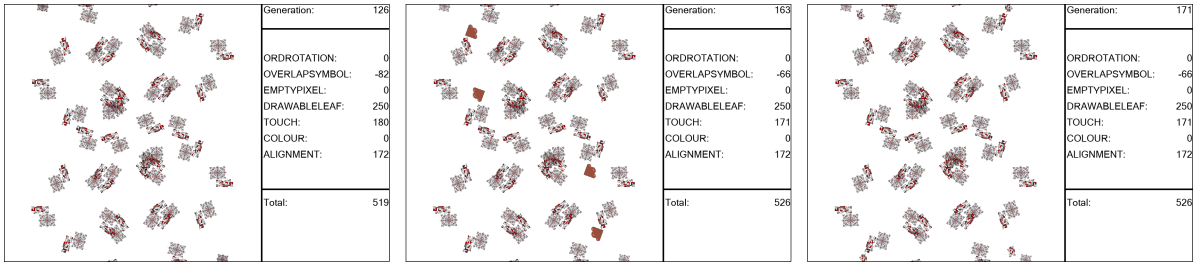
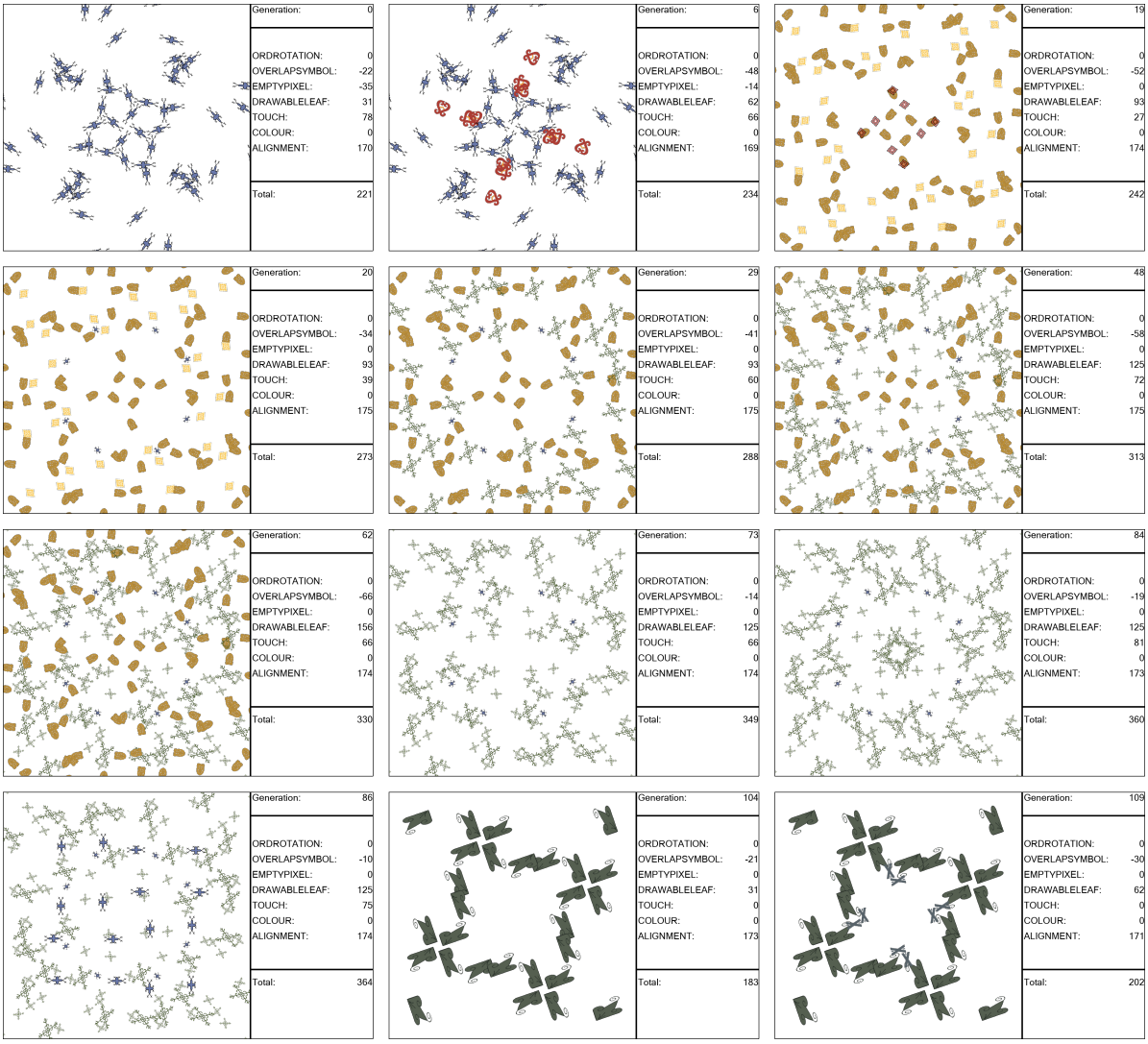
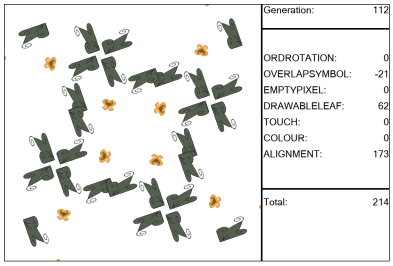
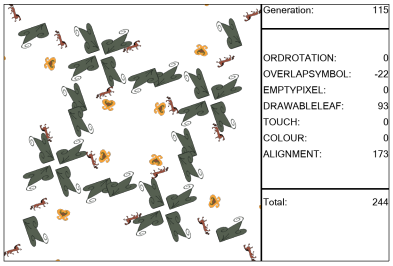


Fig. 36: Collage of the most fit canvases of each generation, generated by SymPla under parameter setting p_5, detailed in Section 4.1. Generations for which the most fit canvas is identical to the previous generation's, are not displayed. Note how after the restart in generation 104, the most fit canvas gradually improves its fitness score to 378 by generation 194, exceeding the fitness score of the most fit canvas of generation 86, which is 364, the best fitness score of this run prior to the restart.

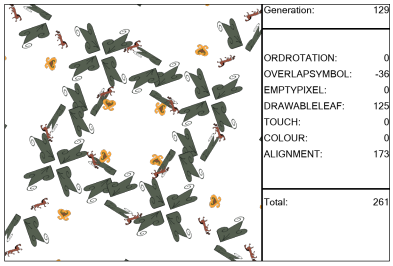




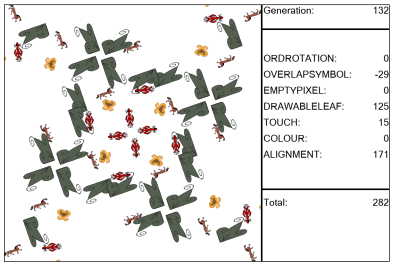
Generation:	112
ORDROTATION:	0
OVERLAPSYMBOL:	-21
EMPTYPIXEL:	0
DRAWABLELEAF:	62
TOUCH:	0
COLOUR:	0
ALIGNMENT:	173
Total:	214



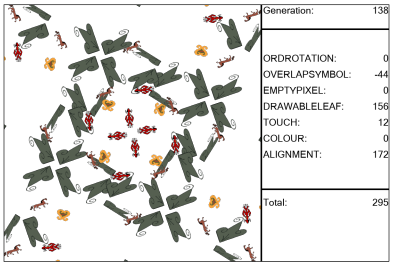
Generation:	115
ORDROTATION:	0
OVERLAPSYMBOL:	-22
EMPTYPIXEL:	0
DRAWABLELEAF:	93
TOUCH:	0
COLOUR:	0
ALIGNMENT:	173
Total:	244



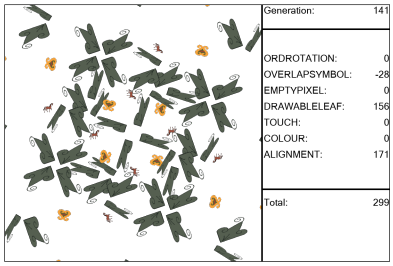
Generation:	129
ORDROTATION:	0
OVERLAPSYMBOL:	-36
EMPTYPIXEL:	0
DRAWABLELEAF:	125
TOUCH:	0
COLOUR:	0
ALIGNMENT:	173
Total:	261



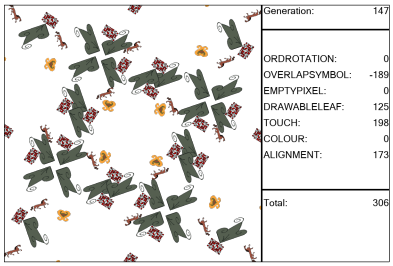
Generation:	132
ORDROTATION:	0
OVERLAPSYMBOL:	-29
EMPTYPIXEL:	0
DRAWABLELEAF:	125
TOUCH:	15
COLOUR:	0
ALIGNMENT:	171
Total:	282



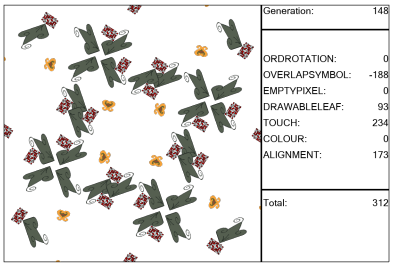
Generation:	138
ORDROTATION:	0
OVERLAPSYMBOL:	-44
EMPTYPIXEL:	0
DRAWABLELEAF:	156
TOUCH:	12
COLOUR:	0
ALIGNMENT:	172
Total:	295



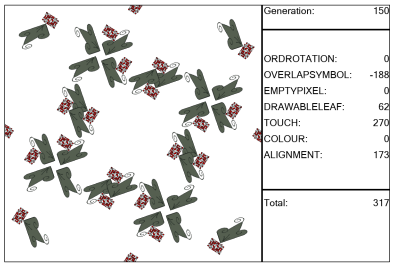
Generation:	141
ORDROTATION:	0
OVERLAPSYMBOL:	-28
EMPTYPIXEL:	0
DRAWABLELEAF:	156
TOUCH:	0
COLOUR:	0
ALIGNMENT:	171
Total:	299



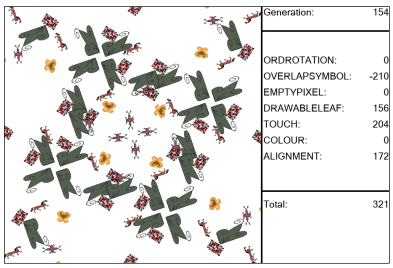
Generation:	147
ORDROTATION:	0
OVERLAPSYMBOL:	-189
EMPTYPIXEL:	0
DRAWABLELEAF:	125
TOUCH:	198
COLOUR:	0
ALIGNMENT:	173
Total:	306



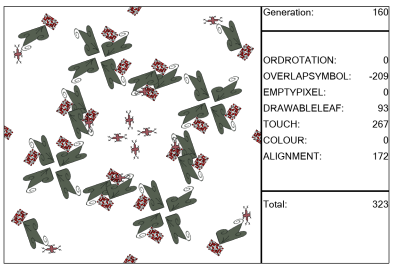
Generation:	148
ORDROTATION:	0
OVERLAPSYMBOL:	-188
EMPTYPIXEL:	0
DRAWABLELEAF:	93
TOUCH:	234
COLOUR:	0
ALIGNMENT:	173
Total:	312



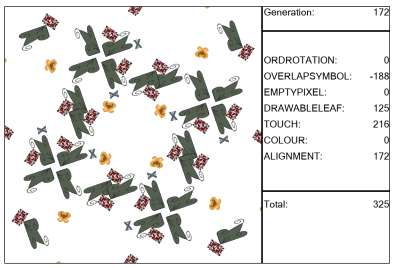
Generation:	150
ORDROTATION:	0
OVERLAPSYMBOL:	-188
EMPTYPIXEL:	0
DRAWABLELEAF:	62
TOUCH:	270
COLOUR:	0
ALIGNMENT:	173
Total:	317



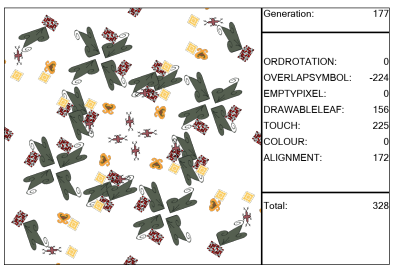
Generation:	154
ORDROTATION:	0
OVERLAPSYMBOL:	-210
EMPTYPIXEL:	0
DRAWABLELEAF:	156
TOUCH:	204
COLOUR:	0
ALIGNMENT:	172
Total:	321



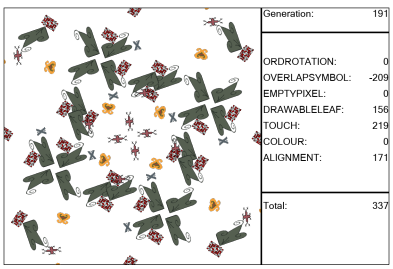
Generation:	160
ORDROTATION:	0
OVERLAPSYMBOL:	-209
EMPTYPIXEL:	0
DRAWABLELEAF:	93
TOUCH:	267
COLOUR:	0
ALIGNMENT:	172
Total:	323



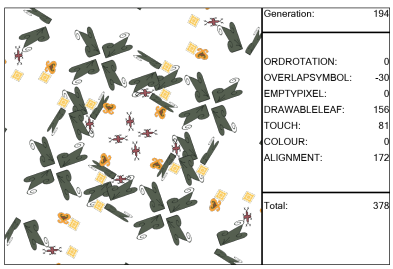
Generation:	172
ORDROTATION:	0
OVERLAPSYMBOL:	-188
EMPTYPIXEL:	0
DRAWABLELEAF:	125
TOUCH:	216
COLOUR:	0
ALIGNMENT:	172
Total:	325



Generation:	177
ORDROTATION:	0
OVERLAPSYMBOL:	-224
EMPTYPIXEL:	0
DRAWABLELEAF:	156
TOUCH:	225
COLOUR:	0
ALIGNMENT:	172
Total:	328

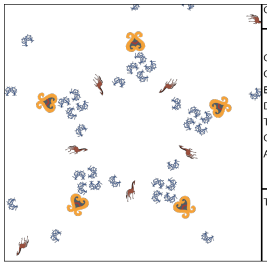


Generation:	191
ORDROTATION:	0
OVERLAPSYMBOL:	-209
EMPTYPIXEL:	0
DRAWABLELEAF:	156
TOUCH:	219
COLOUR:	0
ALIGNMENT:	171
Total:	337

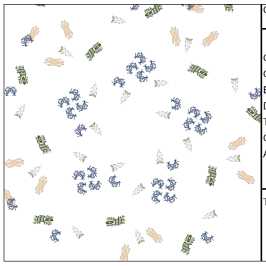


Generation:	194
ORDROTATION:	0
OVERLAPSYMBOL:	-30
EMPTYPIXEL:	0
DRAWABLELEAF:	156
TOUCH:	81
COLOUR:	0
ALIGNMENT:	172
Total:	378

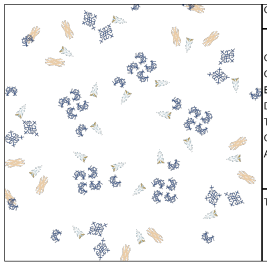
Fig. 37: Collage of the most fit canvases of each generation, generated by SymPla under parameter setting p_6, detailed in Section 4.1. Generations for which the most fit canvas is identical to the previous generation's, are not displayed. Note the restarts at generations 24 and 75.



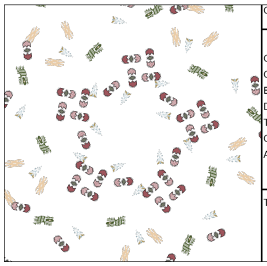
Generation:	80
ORDROTATION:	0
OVERLAPSYMBOL:	0
EMPTYPIXEL:	-47
DRAWABLELEAF:	55
TOUCH:	0
COLOUR:	0
ALIGNMENT:	172
Total:	180



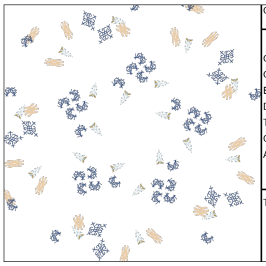
Generation:	82
ORDROTATION:	0
OVERLAPSYMBOL:	-7
EMPTYPIXEL:	-24
DRAWABLELEAF:	74
TOUCH:	0
COLOUR:	-23
ALIGNMENT:	172
Total:	190



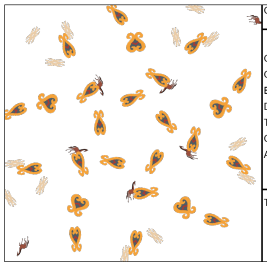
Generation:	86
ORDROTATION:	0
OVERLAPSYMBOL:	-7
EMPTYPIXEL:	-28
DRAWABLELEAF:	74
TOUCH:	0
COLOUR:	0
ALIGNMENT:	172
Total:	210



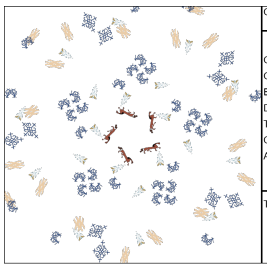
Generation:	87
ORDROTATION:	0
OVERLAPSYMBOL:	-2
EMPTYPIXEL:	-1
DRAWABLELEAF:	74
TOUCH:	6
COLOUR:	-23
ALIGNMENT:	173
Total:	225



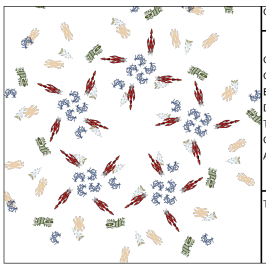
Generation:	88
ORDROTATION:	0
OVERLAPSYMBOL:	-9
EMPTYPIXEL:	-21
DRAWABLELEAF:	92
TOUCH:	0
COLOUR:	0
ALIGNMENT:	173
Total:	234



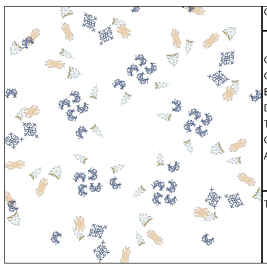
Generation:	93
ORDROTATION:	0
OVERLAPSYMBOL:	-7
EMPTYPIXEL:	-2
DRAWABLELEAF:	74
TOUCH:	0
COLOUR:	0
ALIGNMENT:	173
Total:	237



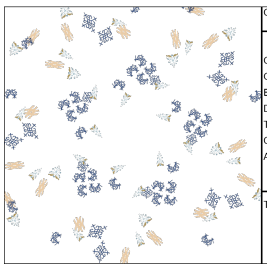
Generation:	94
ORDROTATION:	0
OVERLAPSYMBOL:	-9
EMPTYPIXEL:	-16
DRAWABLELEAF:	111
TOUCH:	0
COLOUR:	-11
ALIGNMENT:	172
Total:	244



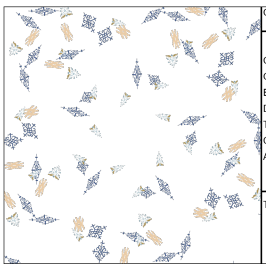
Generation:	95
ORDROTATION:	0
OVERLAPSYMBOL:	-15
EMPTYPIXEL:	0
DRAWABLELEAF:	111
TOUCH:	0
COLOUR:	-23
ALIGNMENT:	172
Total:	245



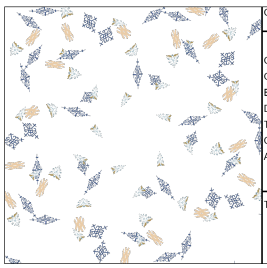
Generation:	96
ORDROTATION:	0
OVERLAPSYMBOL:	-9
EMPTYPIXEL:	-11
DRAWABLELEAF:	111
TOUCH:	6
COLOUR:	0
ALIGNMENT:	173
Total:	269



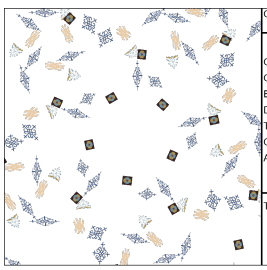
Generation:	106
ORDROTATION:	0
OVERLAPSYMBOL:	-16
EMPTYPIXEL:	-7
DRAWABLELEAF:	129
TOUCH:	6
COLOUR:	0
ALIGNMENT:	173
Total:	284



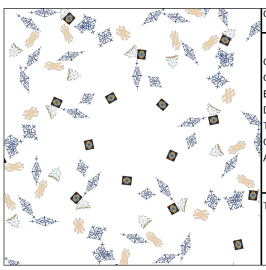
Generation:	112
ORDROTATION:	0
OVERLAPSYMBOL:	-22
EMPTYPIXEL:	-4
DRAWABLELEAF:	129
TOUCH:	12
COLOUR:	0
ALIGNMENT:	175
Total:	290



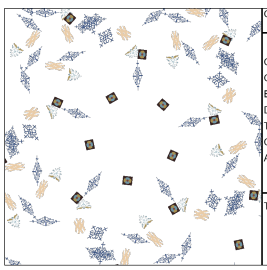
Generation:	116
ORDROTATION:	0
OVERLAPSYMBOL:	-22
EMPTYPIXEL:	0
DRAWABLELEAF:	148
TOUCH:	12
COLOUR:	0
ALIGNMENT:	175
Total:	313



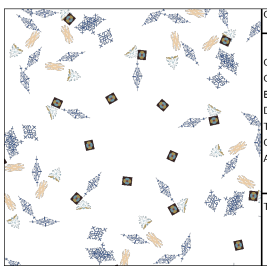
Generation:	130
ORDROTATION:	0
OVERLAPSYMBOL:	-6
EMPTYPIXEL:	0
DRAWABLELEAF:	148
TOUCH:	42
COLOUR:	-11
ALIGNMENT:	176
Total:	348



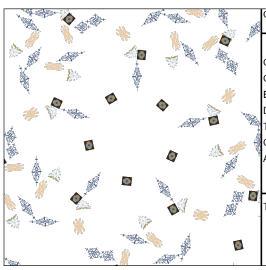
Generation:	142
ORDROTATION:	0
OVERLAPSYMBOL:	-15
EMPTYPIXEL:	0
DRAWABLELEAF:	166
TOUCH:	42
COLOUR:	-11
ALIGNMENT:	176
Total:	358



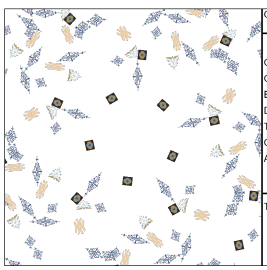
Generation:	147
ORDROTATION:	0
OVERLAPSYMBOL:	-24
EMPTYPIXEL:	0
DRAWABLELEAF:	166
TOUCH:	51
COLOUR:	-11
ALIGNMENT:	177
Total:	358



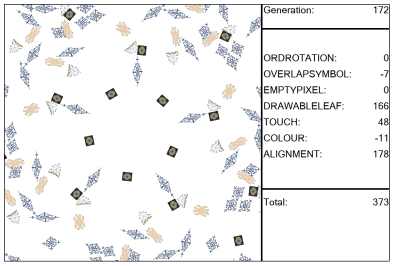
Generation:	158
ORDROTATION:	0
OVERLAPSYMBOL:	-24
EMPTYPIXEL:	-2
DRAWABLELEAF:	166
TOUCH:	54
COLOUR:	-11
ALIGNMENT:	177
Total:	359



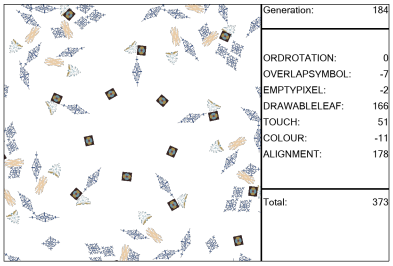
Generation:	162
ORDROTATION:	0
OVERLAPSYMBOL:	-6
EMPTYPIXEL:	-6
DRAWABLELEAF:	148
TOUCH:	60
COLOUR:	-11
ALIGNMENT:	177
Total:	360



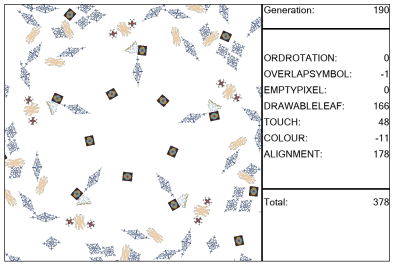
Generation:	165
ORDROTATION:	0
OVERLAPSYMBOL:	-13
EMPTYPIXEL:	0
DRAWABLELEAF:	166
TOUCH:	51
COLOUR:	-11
ALIGNMENT:	177
Total:	370



Generation:	172
ORDROTATION:	0
OVERLAPSYMBOL:	-7
EMPTYPIXEL:	0
DRAWABLELEAF:	166
TOUCH:	48
COLOUR:	-11
ALIGNMENT:	178
Total:	373



Generation:	184
ORDROTATION:	0
OVERLAPSYMBOL:	-7
EMPTYPIXEL:	-2
DRAWABLELEAF:	166
TOUCH:	51
COLOUR:	-11
ALIGNMENT:	178
Total:	373



Generation:	190
ORDROTATION:	0
OVERLAPSYMBOL:	-1
EMPTYPIXEL:	0
DRAWABLELEAF:	166
TOUCH:	48
COLOUR:	-11
ALIGNMENT:	178
Total:	378

References

1. ORNAMIKA. https://ornamika.com. Accessed: 7/7/2022.
2. Artificial Intelligence in Music, Sound, Art and Design: 11th International Conference, EvoMUSART 2022, Held as Part of EvoStar 2022, Madrid, Spain, April 20–22, 2022, Proceedings (Berlin, Heidelberg, 2022), Springer-Verlag.
3. Bar, M., and Neta, M. Humans prefer curved visual objects. Psychological Science 17, 8 (2006), 645–648. PMID: 16913943.
4. Berg, J., Berggren, N. G. A., Borgeteien, S. A., Jahren, C. R. A., Sajid, A., and Nichele, S. Evolved art with transparent, overlapping, and geometric shapes. CoRR abs/1904.06110 (2019).
5. Clark, A. Pillow (pil fork) documentation, 2015.
6. Colton, S. Automatic invention of fitness functions with application to scene generation. vol. 4974, pp. 381–391.
7. Colton, S., McCormack, J., Berns, S., Petrovskaya, E., and Cook, M. Adapting and enhancing evolutionary art for casual creation. In Artificial Intelligence in Music, Sound, Art and Design (Cham, 2020), J. Romero, A. Ekárt, T. Martins, and J. Correia, Eds., Springer International Publishing, pp. 17–34.
8. Danandeh Mehr, A., Nourani, V., Kahya, E., Hrnjica, B., Sattar, A. M., and Yaseen, Z. M. Genetic programming in water resources engineering: A state-of-the-art review. Journal of Hydrology 566 (2018), 643–667.
9. den Heijer, E., and Eiben, A. Comparing aesthetic measures for evolutionary art. pp. 311–320.
10. Doerr, C., Wang, H., Ye, F., van Rijn, S., and Bäck, T. IOHprofiler: A Benchmarking and Profiling Tool for Iterative Optimization Heuristics. arXiv e-prints:1810.05281 (Oct. 2018).
11. Dunn, O. J. Multiple comparisons among means. American Statistical Association (1961), 52–64.
12. Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. A density-based algorithm for discovering clusters in large spatial databases with noise. In Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (1996), KDD'96, AAAI Press, p. 226–231.
13. Ferreira, C. Gene expression programming: a new adaptive algorithm for solving problems.
14. Fisher, R., Perkins, S., Walker, A., and Wolfart, E. Geometric operations - rotate, 2003.
15. Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In Advances in Neural Information Processing Systems (2014), Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Weinberger, Eds., vol. 27, Curran Associates, Inc.
16. Gumin, M. Wave function collapse algorithm, 2016.
17. Hasler, D., and Suesstrunk, S. Measuring colourfulness in natural images. Proceedings of SPIE - The International Society for Optical Engineering 5007 (06 2003), 87–95.
18. Javaheri Javid, M. A. Aesthetic evaluation of experimental stimuli using spatial complexity and kolmogorov complexity. In Artificial Intelligence in Music, Sound, Art and Design (Cham, 2022), T. Martins, N. Rodríguez-Fernández, and S. M. Rebelo, Eds., Springer International Publishing, pp. 117–130.
19. Joyce, D. Wallpaper groups, 1997.
20. Keijzer, M., and Babovic, V. Dimensionally aware genetic programming. Gecco-99: Proceedings of the Genetic and Evolutionary Computation Conference (01 1999).
21. Koza, J. R., and Poli, R. Genetic Programming. Springer US, Boston, MA, 2005, pp. 127–164.
22. Krolikowski, A., Friday, S., Quintanilla, A., and Schrum, J. Quantum zentanglement: Combining picbreeder and wave function collapse to create zentangles®. In Artificial Intelligence in Music, Sound, Art and Design (Cham, 2020), J. Romero, A. Ekárt, T. Martins, and J. Correia, Eds., Springer International Publishing, pp. 49–65.
23. Lamiroy, B., and Potier, E. Lamuse: Leveraging Artificial Intelligence for Sparking Inspiration. 04 2022, pp. 148–161.
24. Lopes, D., Correia, J., and Machado, P. Evodesigner: Towards aiding creativity in graphic design. In Artificial Intelligence in Music, Sound, Art and Design (Cham, 2022), T. Martins, N. Rodríguez-Fernández, and S. M. Rebelo, Eds., Springer International Publishing, pp. 162–178.
25. Lourenço, N., Assunção, F., Maçãs, C., and Machado, P. Evofashion: Customising fashion through evolution. pp. 176–189.
26. Lundh, F. PIL, Python Imaging Library, 1995.
27. McCormack, J., and Lomas, A. Understanding aesthetic evaluation using deep learning. In Artificial Intelligence in Music, Sound, Art and Design (Cham, 2020), J. Romero, A. Ekárt, T. Martins, and J. Correia, Eds., Springer International Publishing, pp. 118–133.
28. Miller, J. F., and Thomson, P. Cartesian genetic programming. In Genetic Programming (Berlin, Heidelberg, 2000), R. Poli, W. Banzhaf, W. B. Langdon, J. Miller, P. Nordin, and T. C. Fogarty, Eds., Springer Berlin Heidelberg, pp. 121–132.
29. Nedungadi, P., and Hutchinson, J. W. The prototypicality of brands: Relationships with brand awareness, preference and usage. ACR North American Advances (1985).
30. Neumann, A., Alexander, B., and Neumann, F. Evolutionary image transition and painting using random walks, 2020.
31. Nordin, P. A compiling genetic programming system that directly manipulates the machine code. Advances in genetic programming 1 (1994), 311.

32. Nordin, P., Banzhaf, W., and Francone, F. D. Efficient Evolution of Machine Code for CISC Architectures Using Instruction Blocks and Homologous Crossover. MIT Press, Cambridge, MA, USA, 1999, p. 275–299.
33. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12 (2011), 2825–2830.
34. Poli, R., Langdon, W. B., and McPhee, N. F. A field guide to genetic programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).
35. Ramachandran, V., and Hirstein, W. The science of art: A neurological theory of aesthetic experience. Journal of Consciousness Studies 6 (01 1999), 15–51.
36. Riccio, P., Galati, F., Zuluaga, M. A., De Martin, J. C., and Nichele, S. Translating emotions from eeg to visual arts. In Artificial Intelligence in Music, Sound, Art and Design (Cham, 2022), T. Martins, N. Rodríguez-Fernández, and S. M. Rebelo, Eds., Springer International Publishing, pp. 243–258.
37. Richter, H. Visual art inspired by the collective feeding behavior of sand-bubbler crabs. In Computational Intelligence in Music, Sound, Art and Design (Cham, 2018), A. Liapis, J. J. Romero Cardalda, and A. Ekárt, Eds., Springer International Publishing, pp. 1–17.
38. Romero, J., and Machado, P., Eds. Evolutionary Visual Art and Design. Natural Computing Series. Springer Berlin Heidelberg, 2008.
39. Searson, D. P. Gptips 2: an open-source software platform for symbolic data mining, 2014.
40. Secretan, J., Beato, N., D'Ambrosio, D., Rodriguez, A., Campbell, A., Folsom-Kovarik, J., and Stanley, K. Picbreeder: A case study in collaborative evolutionary exploration of design space. Evolutionary computation 19 (10 2010), 373–403.
41. Silvera, D. H., Josephs, R. A., and Giesler, R. B. Bigger is better: the influence of physical size on aesthetic preference judgments. Journal of Behavioral Decision Making 15, 3 (2002), 189–202.
42. Singh, D., Rajcic, N., Colton, S., and McCormack, J. Camera obscurer: Generative art for design inspiration. In Computational Intelligence in Music, Sound, Art and Design (Cham, 2019), A. Ekárt, A. Liapis, and M. L. Castro Pena, Eds., Springer International Publishing, pp. 51–68.
43. Smirnov, N. V. Estimate of deviation between empirical distribution functions in two independent samples. Bulletin Moscow University 2, 2 (1939), 3–16.
44. Steffens, B. Symbolplacer, 2023.
45. Tarimo, W. T., Xing, C., Dong, L. V., and Li, C. Evolving images using transparent overlapping polygons.
46. The GIMP Development Team. Gimp.
47. Tinio, P. P., and Leder, H. Just how stable are stable aesthetic features? symmetry, complexity, and the jaws of massive familiarization. Acta Psychologica 130, 3 (2009), 241–250.
48. Urbano, P. The light show: Flashing fireflies gathering and flying over digital images. In Computational Intelligence in Music, Sound, Art and Design (Cham, 2018), A. Liapis, J. J. Romero Cardalda, and A. Ekárt, Eds., Springer International Publishing, pp. 283–298.
49. Van Rossum, G., and Drake, F. L. Python 3 Reference Manual. CreateSpace, Scotts Valley, CA, 2009.
50. Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods 17 (2020), 261–272.
51. Zajonc, R. Mere exposure: A gateway to the subliminal. Current Directions in Psychological Science 10, 6 (2001), 224–228.
52. Zajonc, R. B. Attitudinal effects of mere exposure. Journal of Personality and Social Psychology 9 (1968), 1–27.