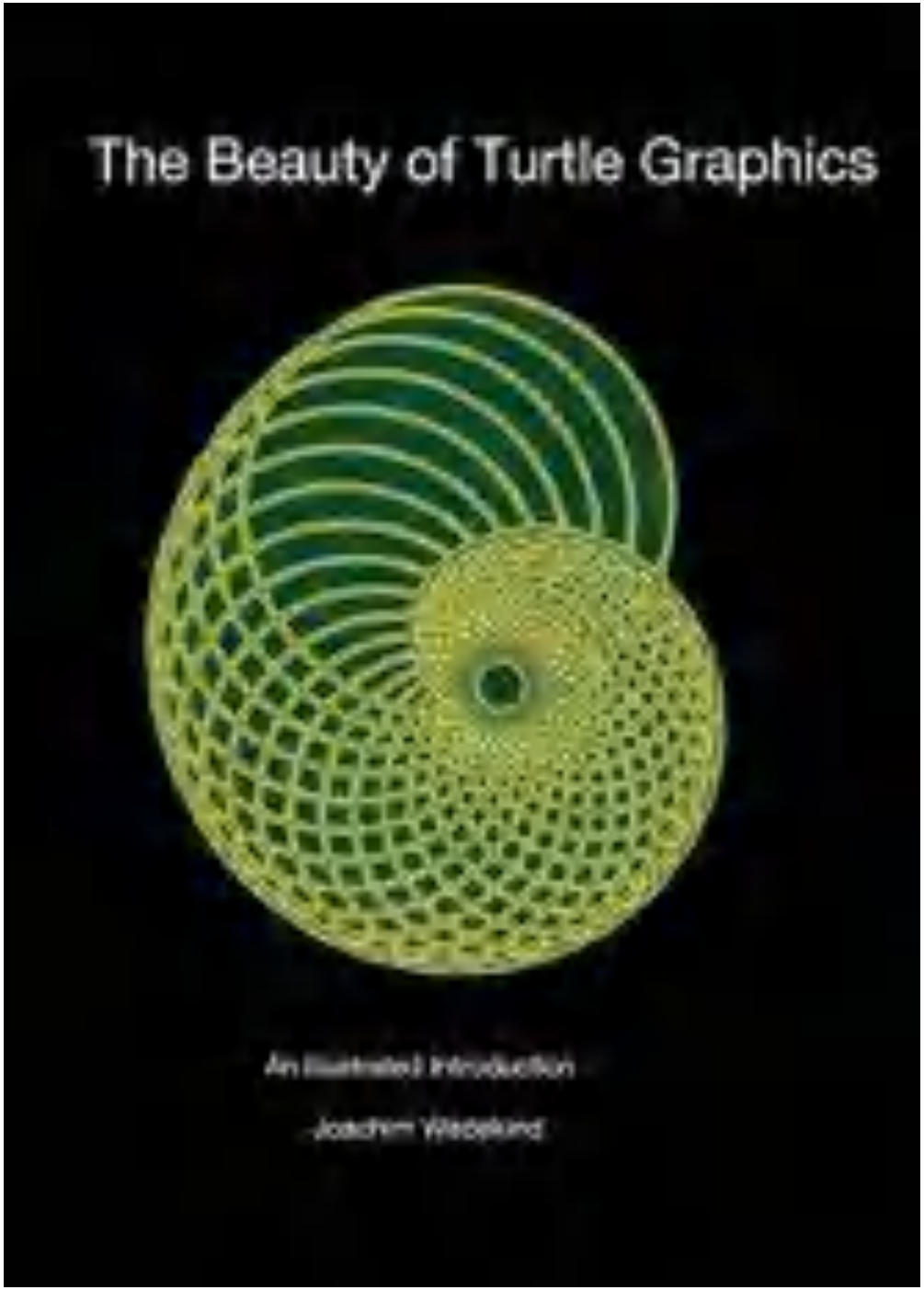

The Beauty of Turtle Graphics
An Illustrated Introduction
Joachim Wedekind

The Beauty of Turtle Graphics

An illustrated Introduction

Joachim Wedekind

Joachim Wedekind:
The Beauty of Turtle Graphics. An illustrated Introduction

Author: Dr. Joachim Wedekind
Eschenweg 26
72076 Tübingen

The present work has been carefully prepared. Nevertheless, the author accepts no liability for the correctness of information, notes and tips, nor for any printing errors.

Website:
http://programmieren.joachim-wedekind.de/logo-classics/

2020 Tübingen

Graphic design and typesetting: Joachim Wedekind

Imprint: http://joachim-wedekind.de/impressum/

Preface

Shortly after my professional entry into teaching technology and media didactics, I read for the first time in 1975 in an overview volume *„Computer und Unterricht“* (Eyferth et al., 1974) about the Logo project of Papert at MIT. Considered by the authors *"as one of the most remarkable attempts of meaningful computer use in class according to its curicular and didactic objectives*", they tried to prove this by means of typical program examples. These showed simple geometric line figures drawn with the "Turtle", a controllable robot.

These pictures were and are typical for the use of Logo in a school context. They can be found in early publications on Logo (the manuals for different Logo versions and many introductory books). I was immediately attracted by the images and the possibilities for their creation. For me, they have a very unique aesthetic with a high recognition value in the corresponding publications.

This book therefore focuses on typical images created with the turtle graphic. In some examples, the proximity to works of early computer art (which I have included as homages to the artists), but also to well-known optical illusions (which I call "Opticals") becomes very obvious.

Only basic programming concepts that are used to generate the respective images are briefly outlined. The book is therefore not an introduction to programming with Logo, but rather a picture book, which may inspire you to program these and comparable pictures yourself.

Tübingen, April 2020 Joachim Wedekind

By the way, one more thing: I apologize for the poor linguistic quality of this booklet. It is my own translation of the German original. In some sections I used the help of DeepL or Google Translate. Corrections and improvements are welcome.

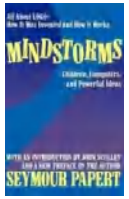
Advertising on my own behalf: There are two books and related websites by me about *Computer art* and *Opticals* that give programming the appropriate status:

J. Wedekind (2018). Codierte Kunst. http://digitalart.joachim-wedekind.de/about-the-book//

J. Wedekind (2019): Optische Täuschungen animieren für Dummies Junior. http://opticals.joachim-wedekind.de/das-buch/

Content

Introduction: Logo Classics



After years I rediscovered the book Mindstorms by Seymour Papert (Papert, 1980, 1982)[1]. In this book he unfolds his view of the computer as a mental tool and how children can access this tool with the help of Logo.

Turtle graphics was introduced as a central component of Logo in 1970. With a few basic commands it is possible to create simple, but also complex appealing graphics[2].

„TURTLE GEOMETRY is a different style of doing geometry, just as Euclid's axiomatic style and Descartes's analytic style are different from one another. Euclid's is a logical style. Descartes's is an algebraic style. Turtle geometry is a computational style of geometry.“ (Papert, 1982, S. 55)

The creation of these images can be easily understood using code examples[3]. Those who enjoy abstract-geometric graphics will find a variety of suggestions on how to create appealing images using simple programming technology.

[1] It is still worth reading, especially today, when it is considered when and in what form computer science lessons will be introduced in schools. The original English version of the book can be downloaded free of charge.

[2] A turtle-like robot was developed for the graphic output: the *Yellow Turtle* (Feurzeig, 2010). This vehicle could move and rotate, and raise and lower a pen. Lines can be drawn on a paper underneath. The robot was later replaced by a small symbol with a direction indicator on the screen.

[3] The examples are implemented with the visual programming environment Snap!, quasi a "granddaughter" of Logo. The graphics are interactive; their characteristics can be changed with sliders.

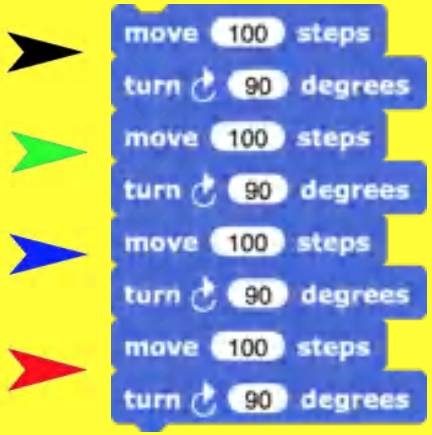
For me, the approach that underlies the turtle graphics is still best described by Seymour Papert himself in *Mindstorms*. Its basic principles are presented in a compact form and illustrated in Chapter 3: *Turtle Geometry: A Mathematics Made for Learning*. They will be illustrated here with interactive examples.

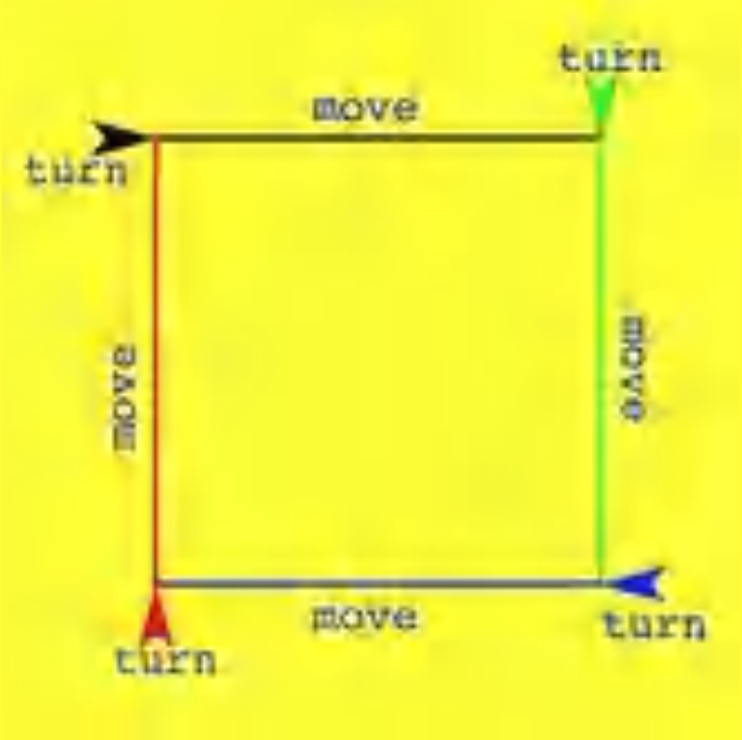
Note: In order to understand the examples, it is recommended to look at the complete code of the examples. The links to these can be found on the website **Logo Classics**. A click on the **program name** leads directly to the programs in the Snap! programming environment.

When **Snap!** is restarted, it opens with a predefined **initial state**: The **stage** has a certain **width** and **height**, is **empty** and has the color **white**. The **turtle** sits in the **middle** of the stage in the shape of an **arrow**, points to the **right** and has the color **black**. All these properties can be easily changed later.

The turtle can understand commands in the so-called *turtle language*. These control their **movement**. With the command **move**, it moves in a straight line in its direction of view, so it changes its position, but not its direction. With the command **turn** it changes its direction of view, but not its position.

In the example opposite the movements of the turtle are described by such commands and thus a square is drawn:

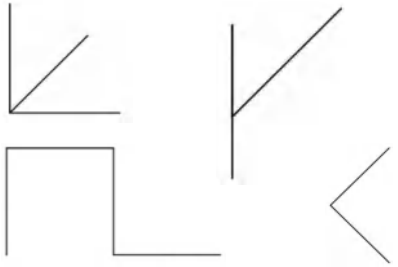




Line Graphics

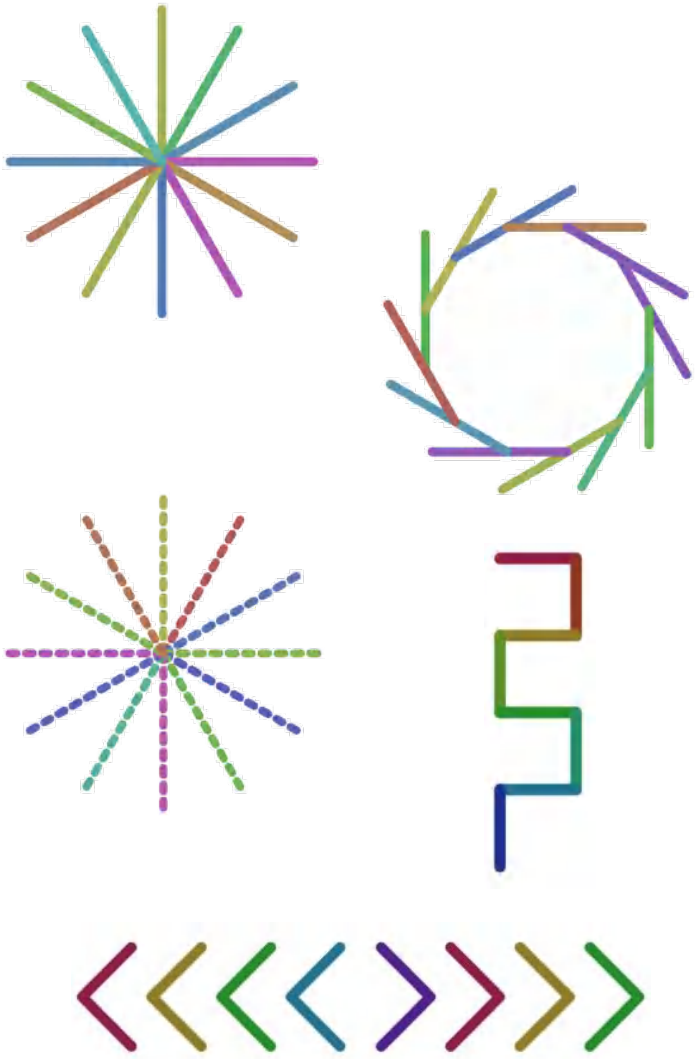
Before the turtle can draw, its pen has to be lowered with **pen down**. Only then does it leave a trace. The pen can be raised again with **pen up**.

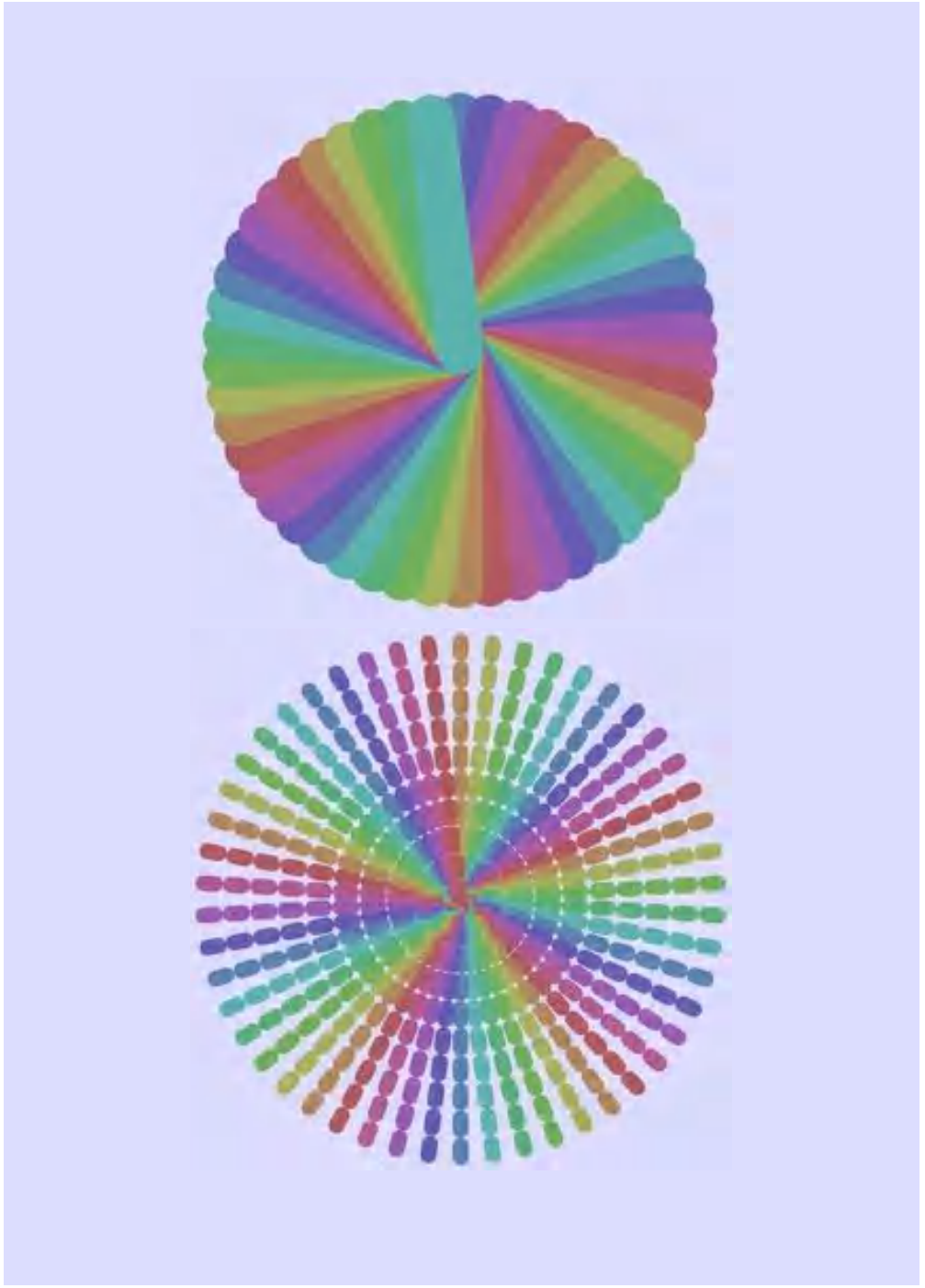
Many different basic patterns are already possible with just **move** and **turn** and **pen down** and **pen up**.



Iterations: The computer is particularly good at performing frequently recurring, dull but necessary actions patiently and quickly. For this purpose, Snap! includes the **repeat** block, whose enclosed commands are executed as often as specified.

If the basic patterns are combined and multiplied in the **repeat** loops, many attractive line graphics can already be created.





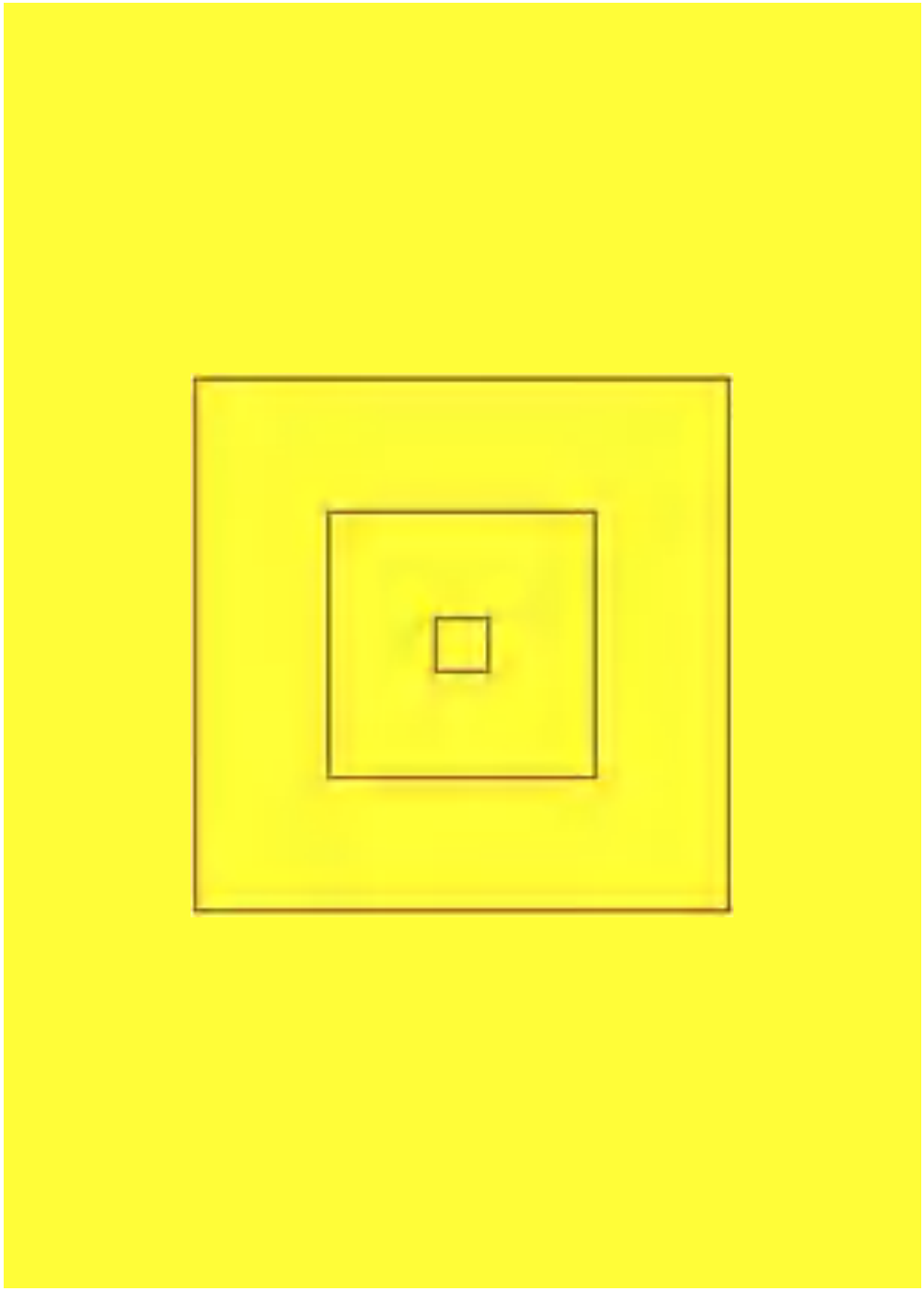
Squares (I)

In many introductions to programming, the first example of a simple sequence of commands is the sentence *Hello World* on the screen[4]. For me, drawing a square is in a way the *Hello World* in the languages of the Logo family! The square can be used to illustrate several important concepts:

Repetitions: In the **repeat** block, its commands will be executed as many times as specified, i.e. four times for the square.

Expandability and reusability: The turtle language can be expanded through your own **procedures**. These command sequences can then be used again and again. In the procedure **square**, the commands for drawing the four sides of the same length and rotating the turtle by 90 degrees are combined.

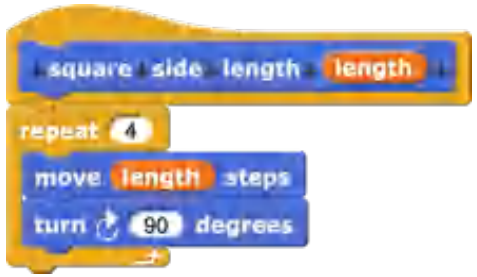
[4] see the list of *Hello World* programs in higher programming languages at Wikipedia



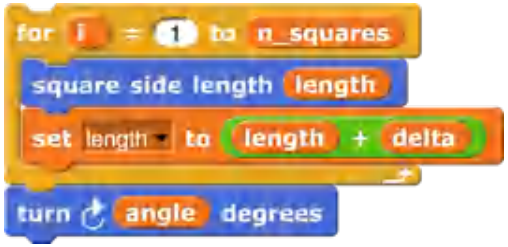
Squares (II)

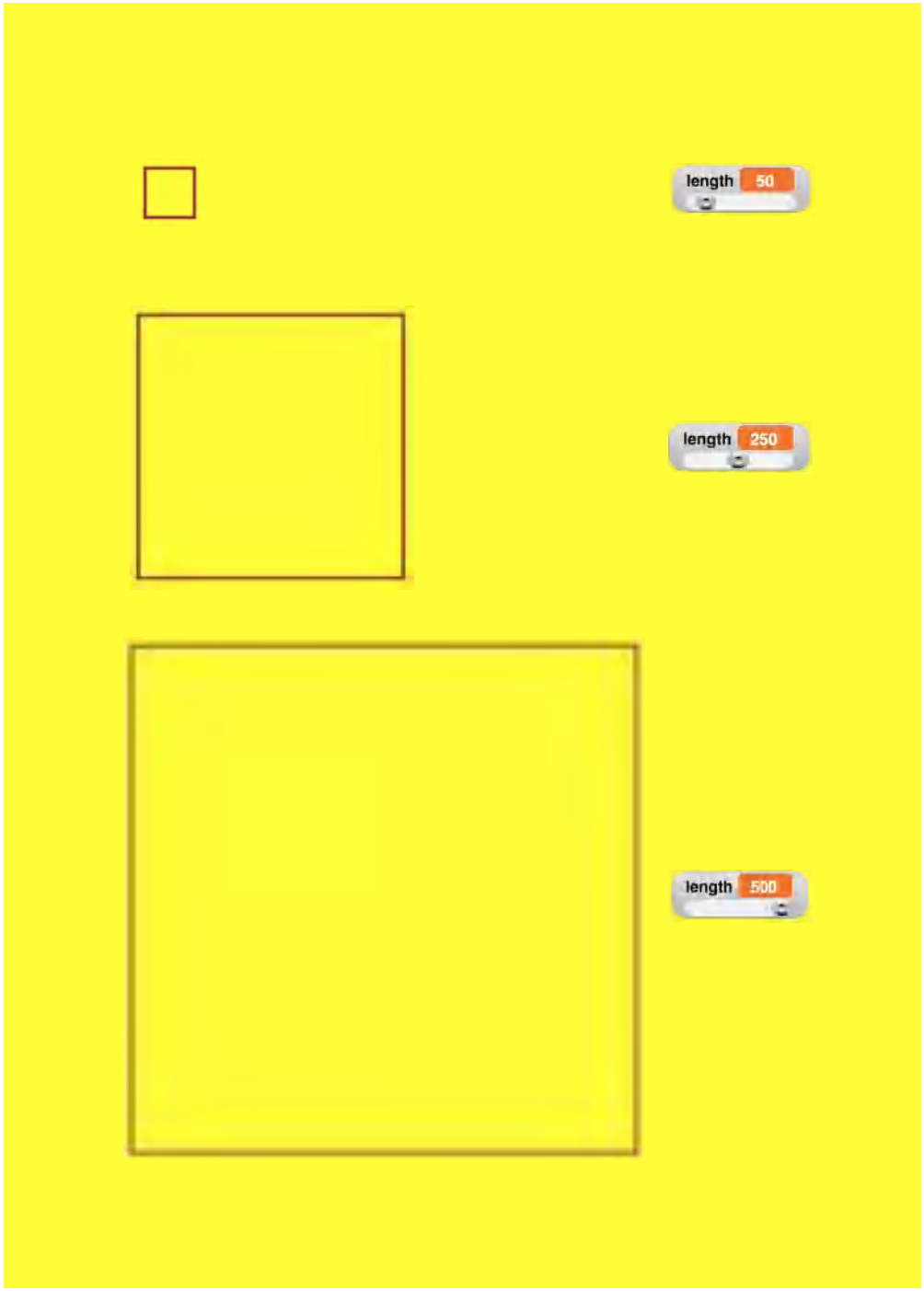
Variables: All kinds of data (numbers, texts, images, etc.) can be stored in variables. The computer will find the data safely, can use them again and again in the program and also change them.

The length of the sides is passed to the **square** procedure as **input parameter length**. The squares can now be drawn in any desired size.

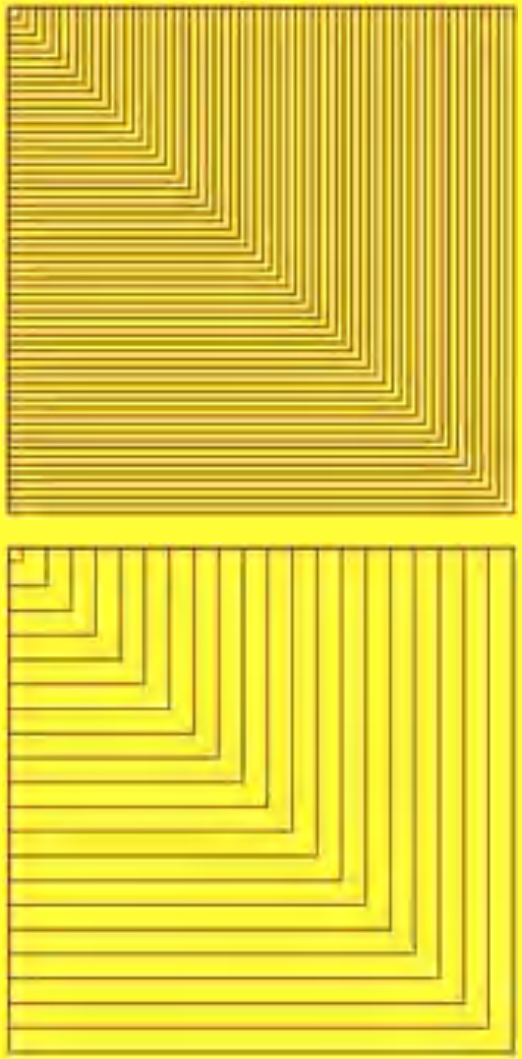


Simplification through repetitions: The reuse of procedures in repetition loops also allows more complex figures; so below for **nested squares** and **diamonds**, controlled via **n_squares, delta** and **angle**.

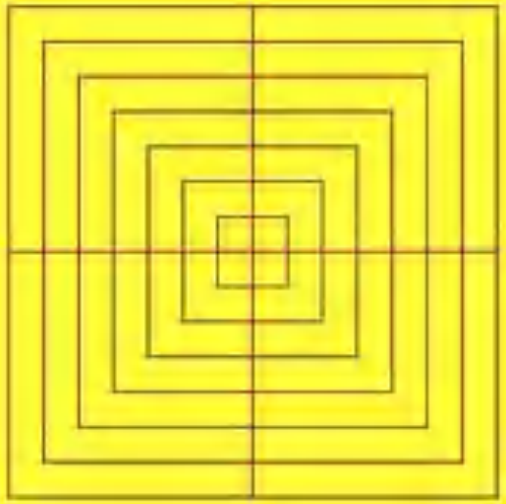




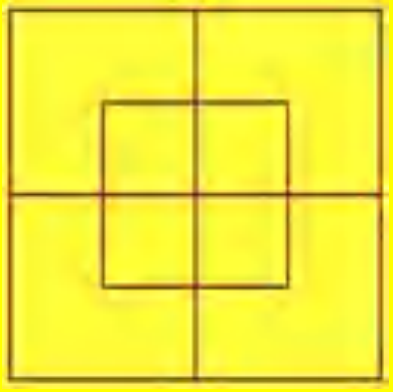
nested squares

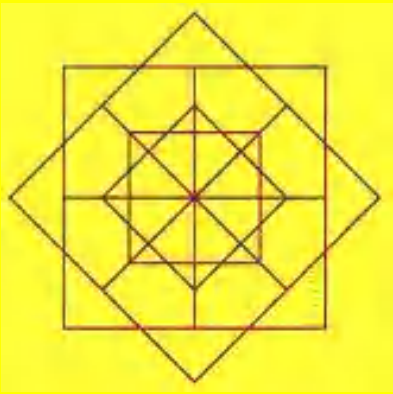


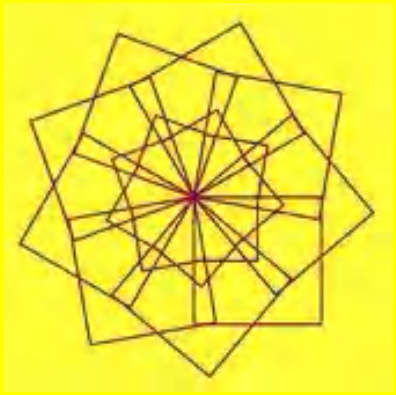
diamonds

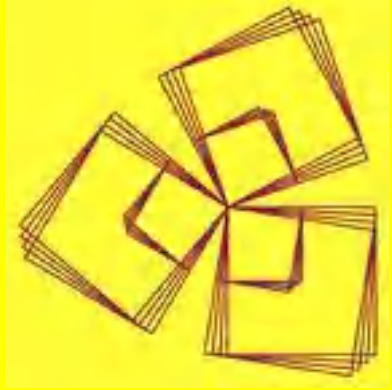


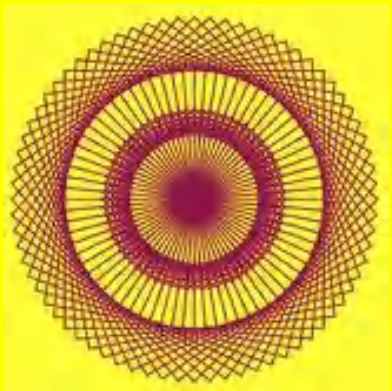
squares



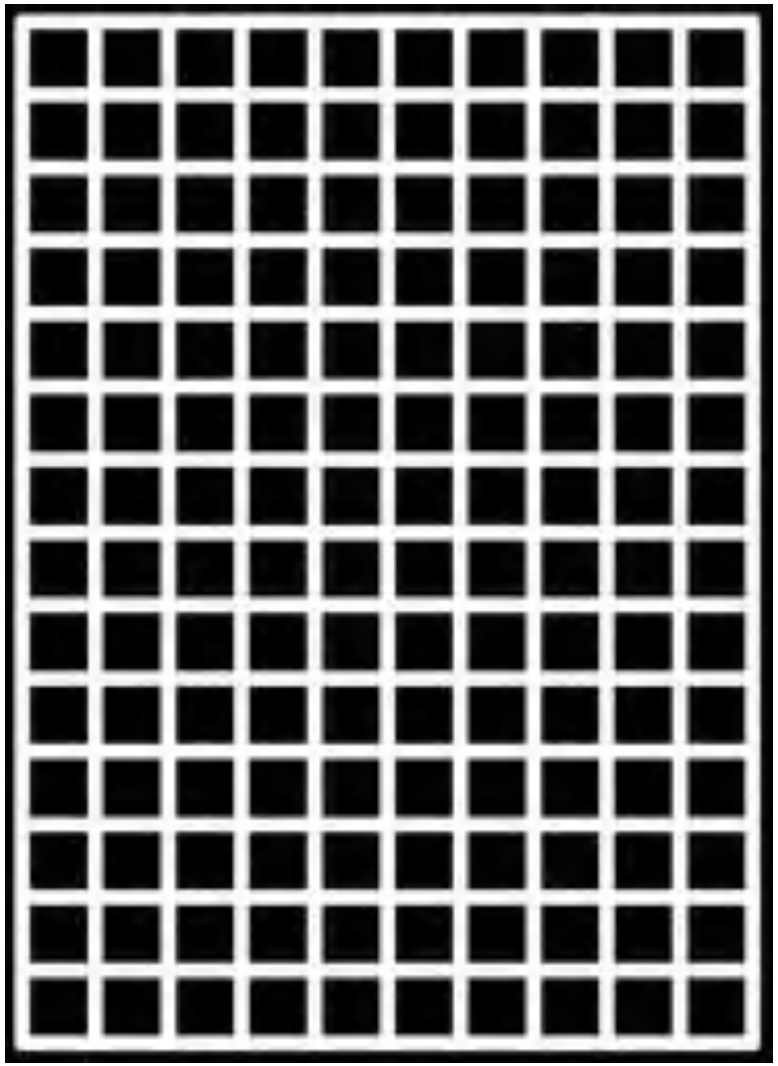




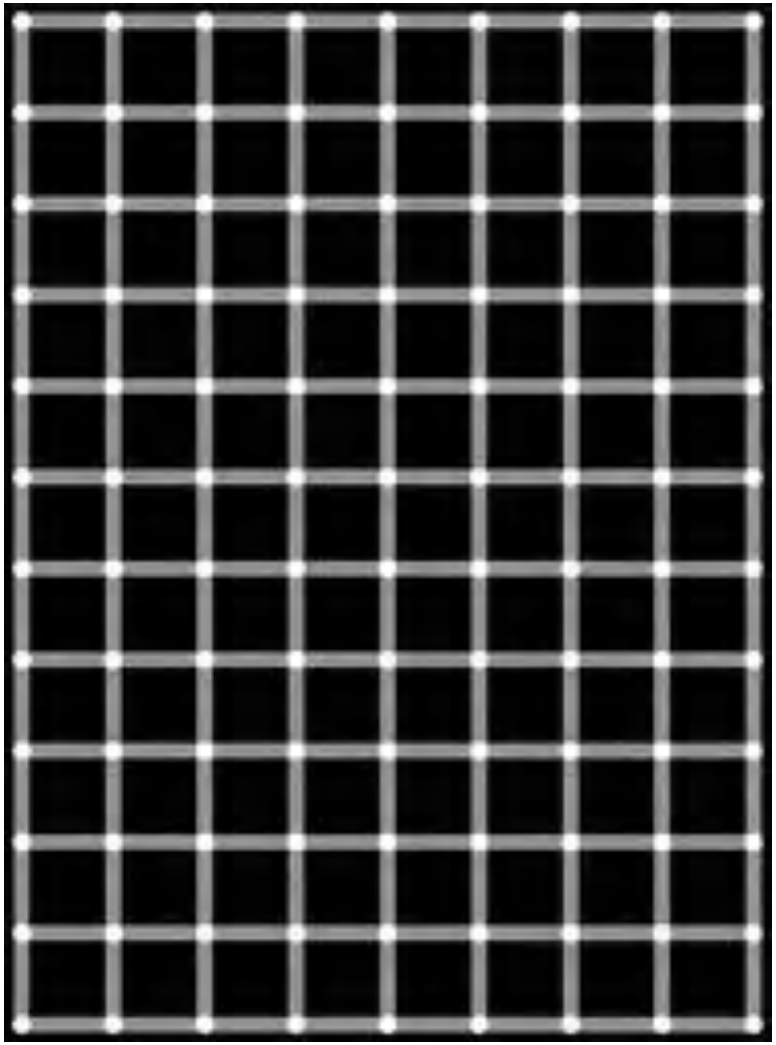




Opticals: Hermann-Hering Grid (grey dots)



Opticals: Scintillating Grid

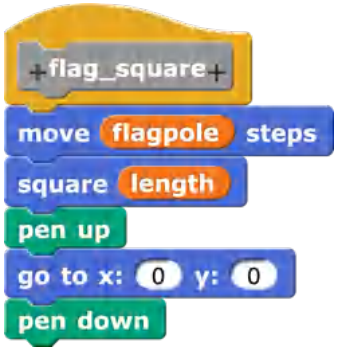


Flags, spiders, swirls

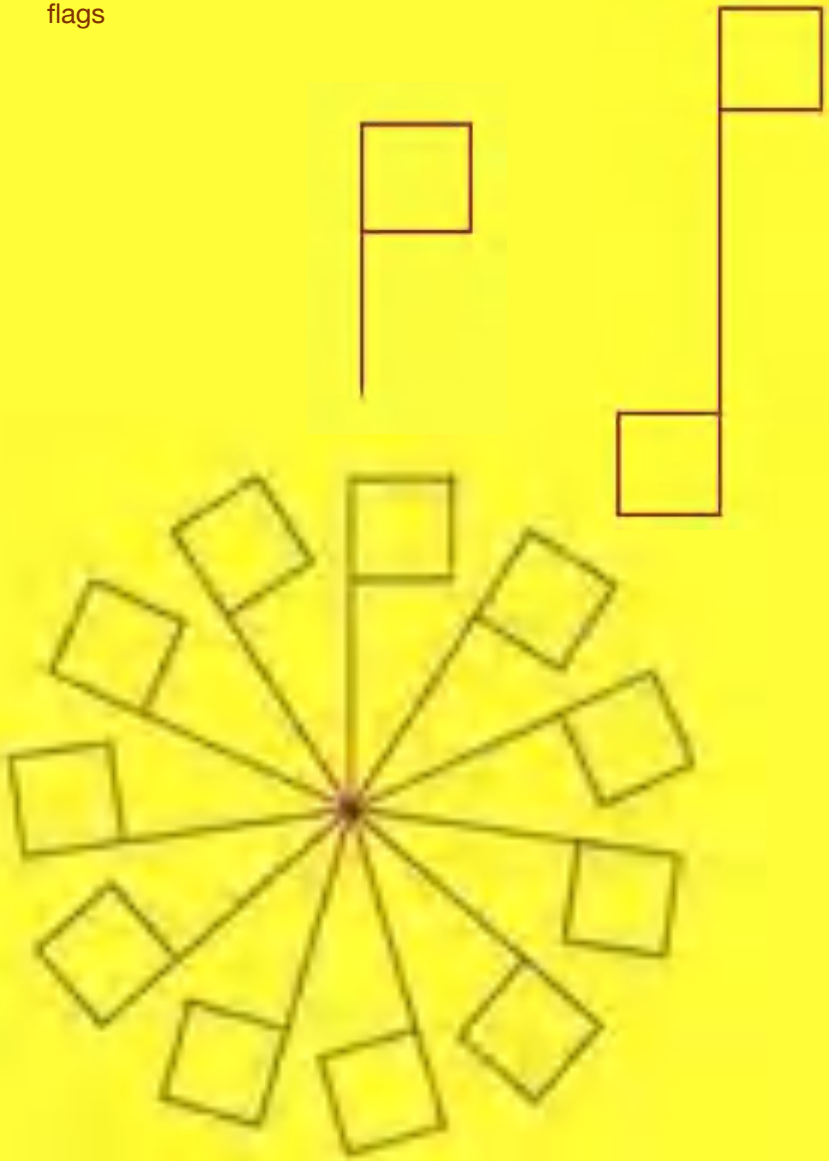
Reusability and location neutrality: The advantage of procedures is their reusability in new contexts. Of course, this also applies to the **square** procedure. Local neutrality is important, i.e. the return of the turtle to its starting point with its original direction of vision.

A procedure **flag_square** draws the flagpole of length **flagpole** and then the square with the side length **length**. Finally, the turtle is always moved to the middle of the stage.

Thus, simple structures can easily be combined into more complex structures. This applies to spiders and swirls (on the following double page).



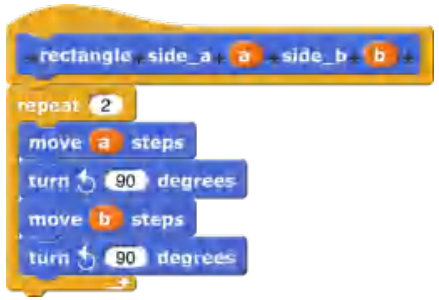
flags



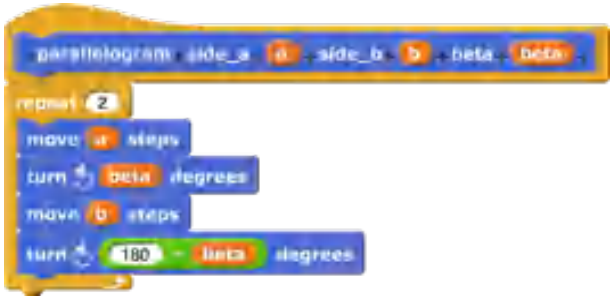
spider with offspring

Rectangles, parallelograms

Modifications and extensions: Procedures can be taken as a starting point for new uses. For example, the procedure **square** can easily be changed to a procedure **rectangle**, to which the width and height of the rectangle are passed as values **a** and **b**, respectively:

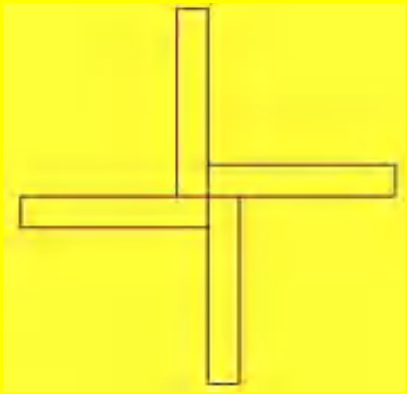


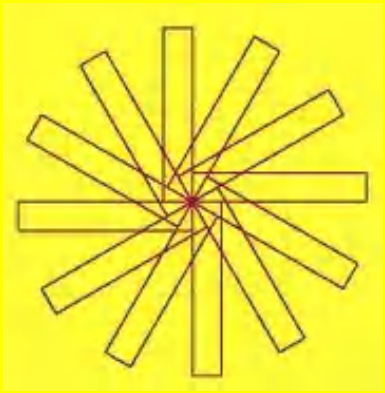
The procedure **rectangle** can in turn be extended to a procedure **parallelogram** if the angle **beta** is passed in addition to the side lengths (the angle **alpha** results in the parallelogram as **180 - beta**).

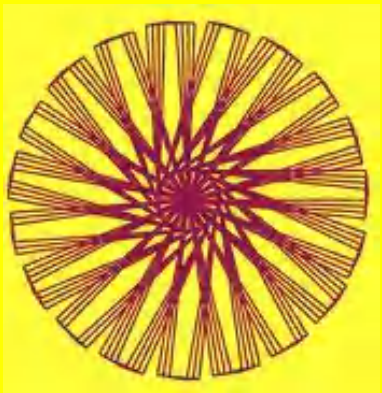


The **parallelogram** procedure now contains the initial procedures as special cases, namely the **rectangle** procedure with **beta = alpha = 90** and the **square** procedure with additional **a = b**!

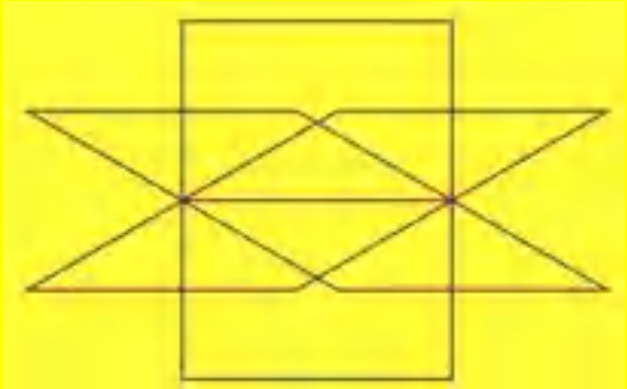
rectangles

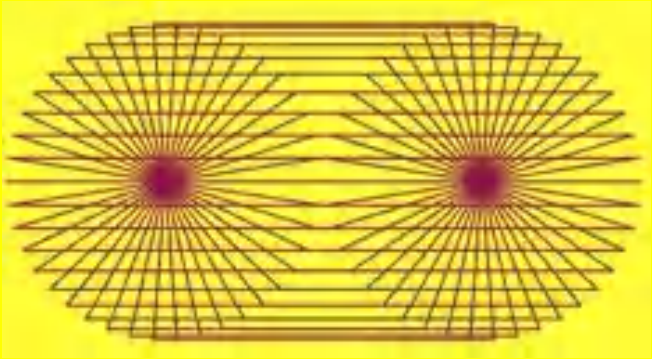


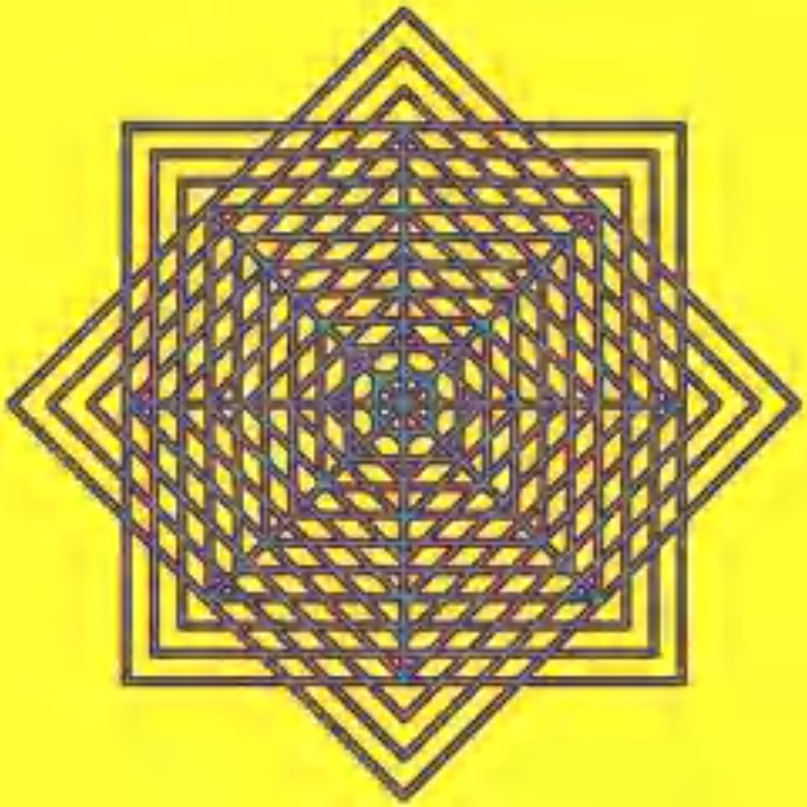


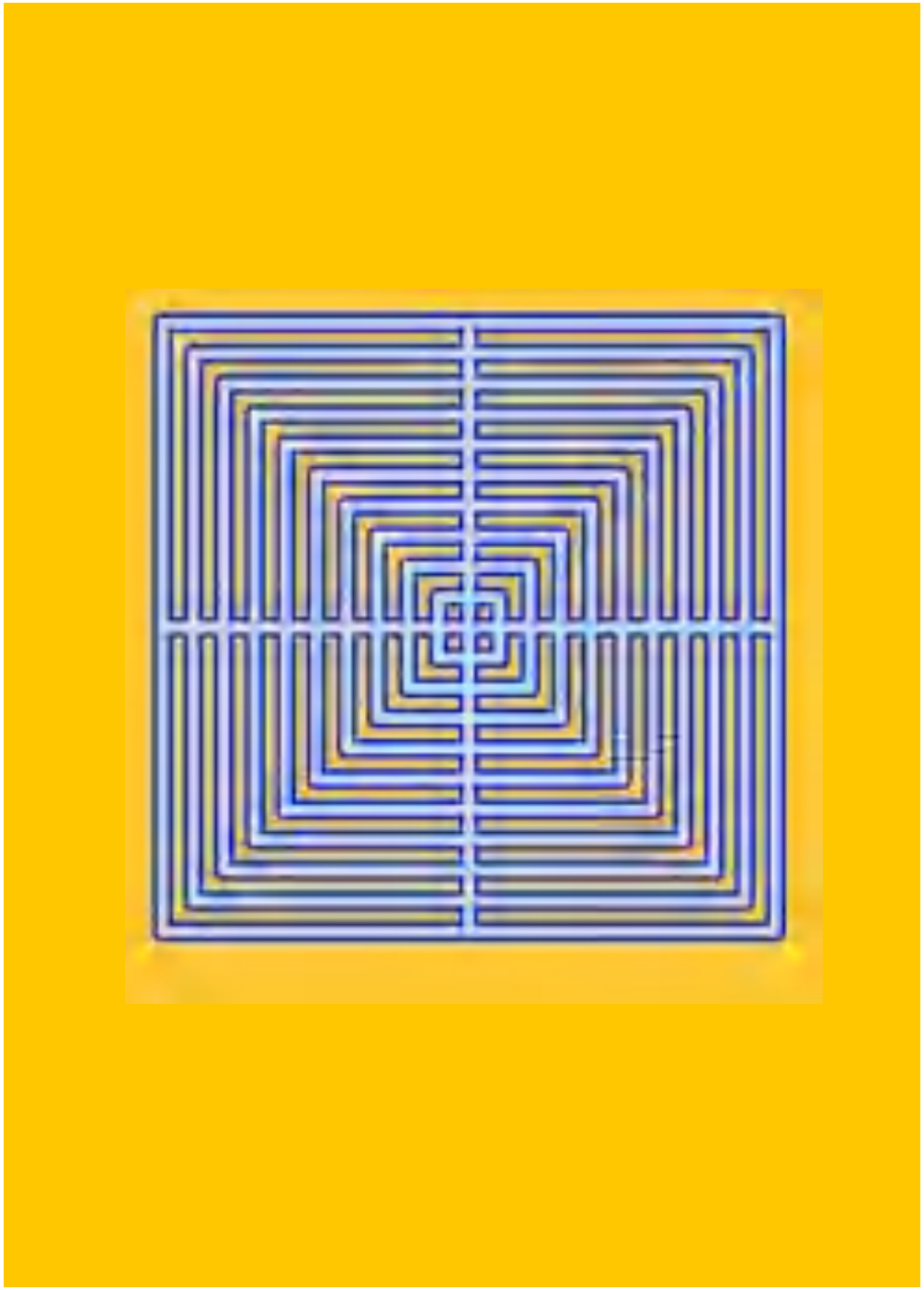


parallelograms

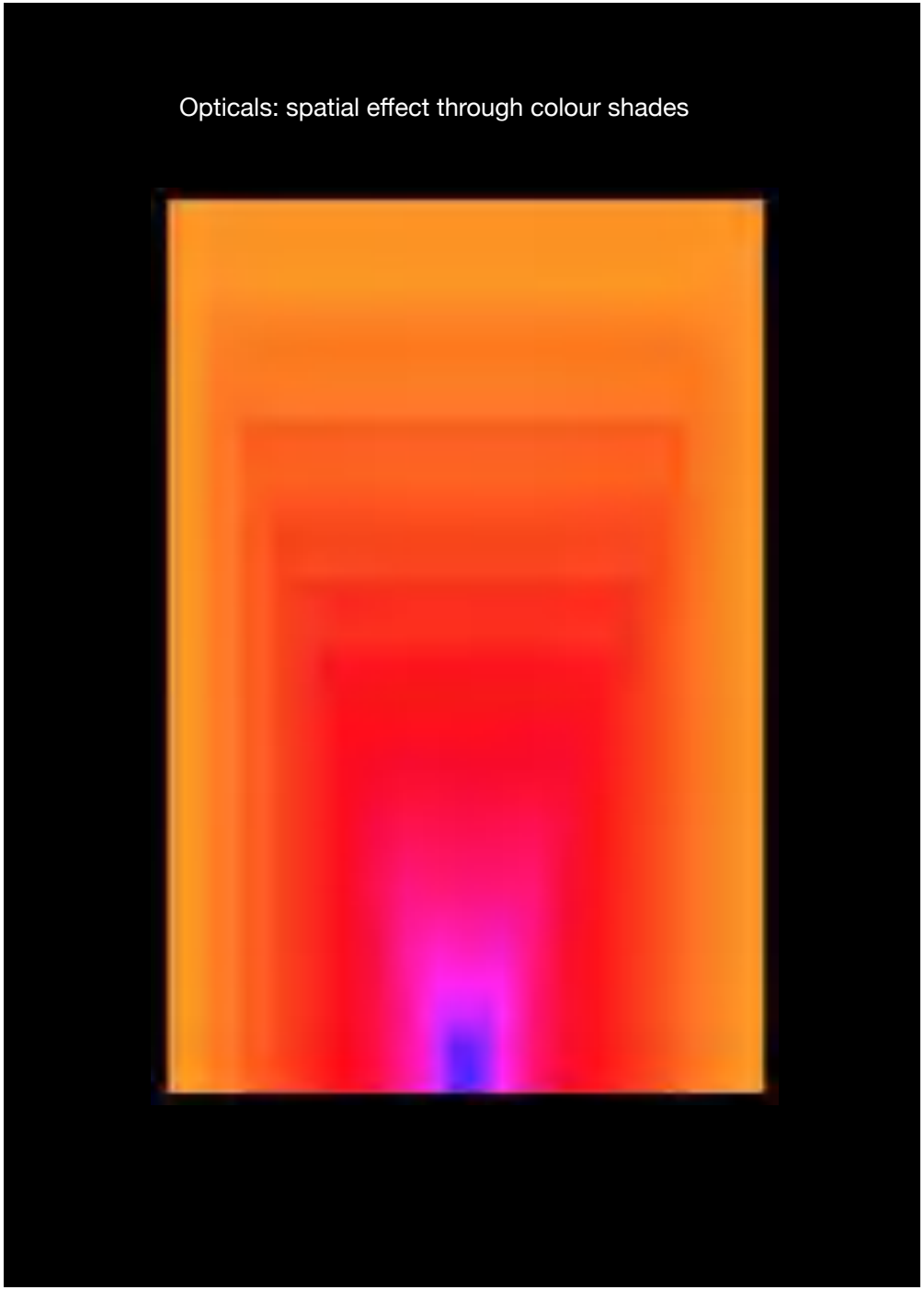


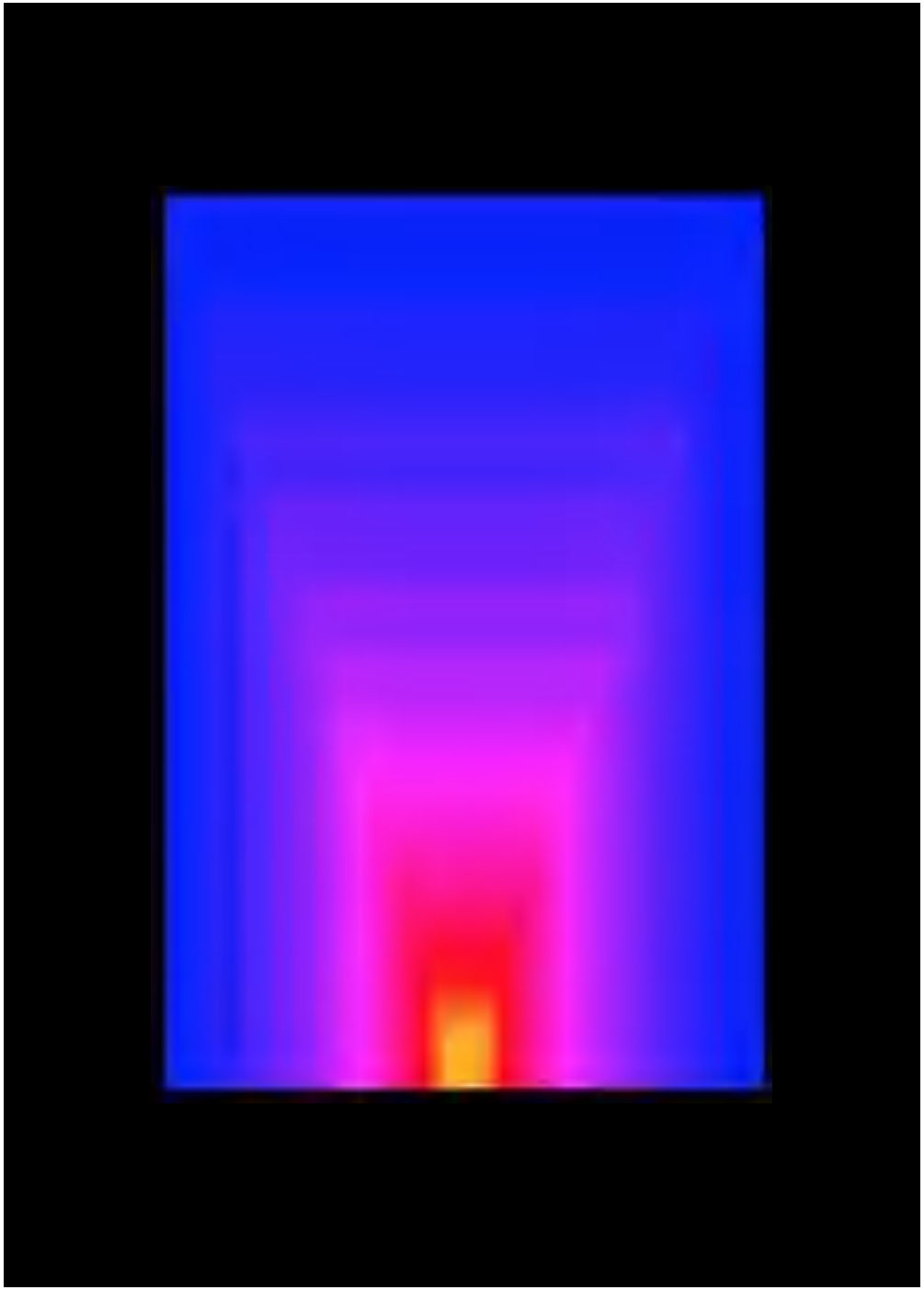






Opticals: spatial effect through colour shades

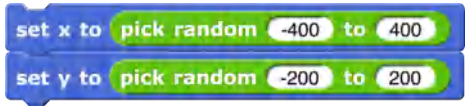




pick random ...

In the previous pictures, everything was clear and therefore predictable, determined by the program commands. **Randomness** can provide for moments of surprise in the pictures, because all characteristics of the turtle can easily be changed randomly. The basis is the **random function** **pick random 1 to 10**, which provides evenly distributed random numbers between 1 and 10.

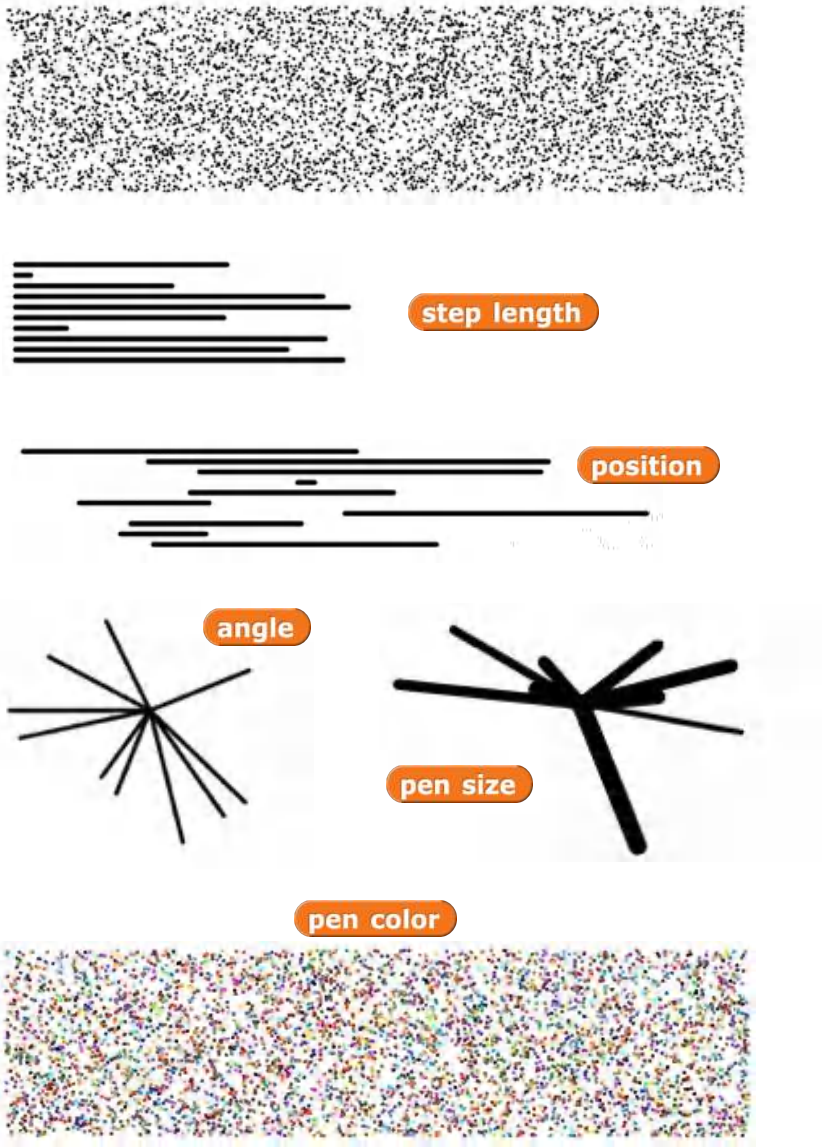
The range for the random numbers can be freely selected. With such limits, then it is spoken of **guided randomness**. For example,



is used to determine the random positions of the turtle. The picture on the top right shows accordingly the rather even distribution of points.

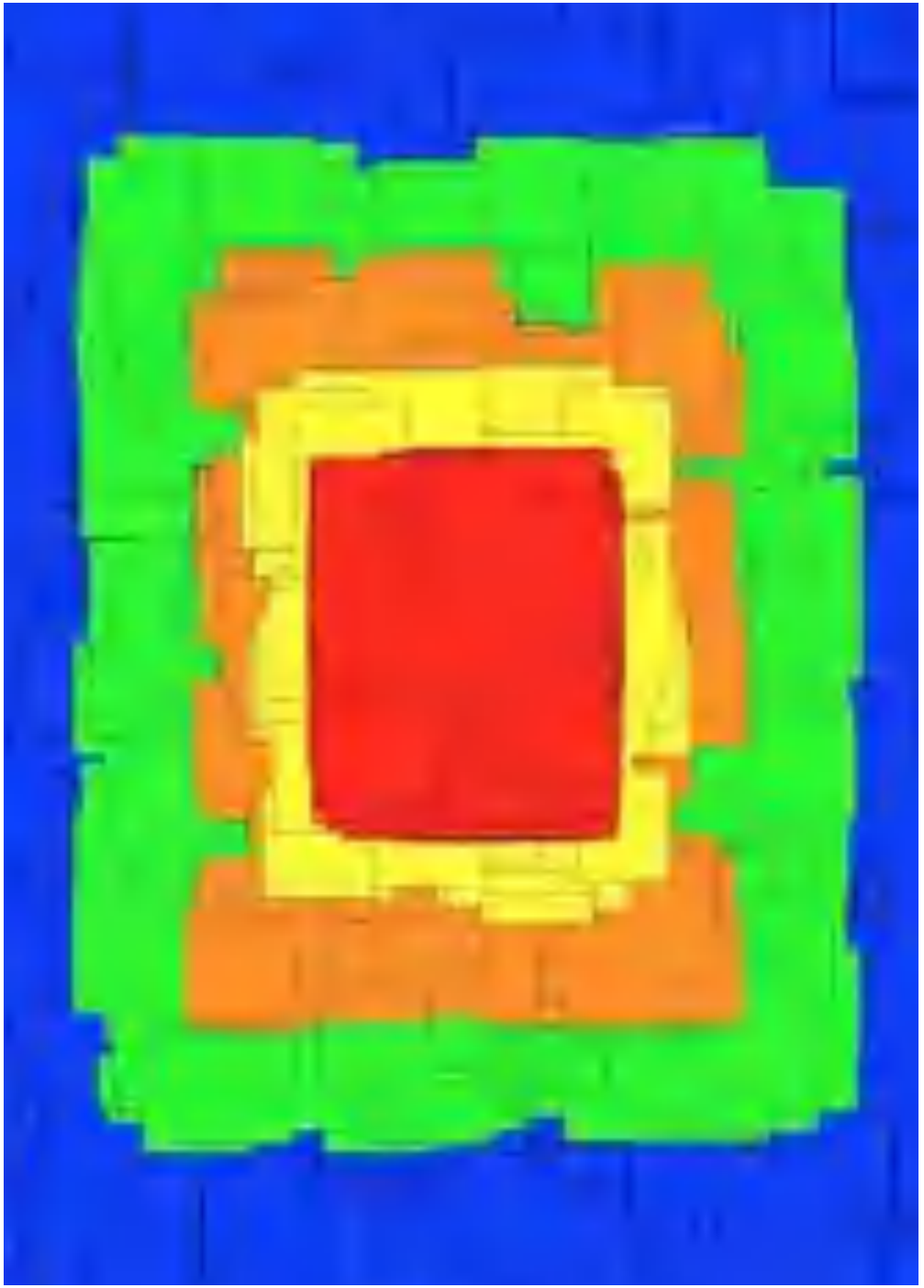
Chance can change everything! That affects, among other things

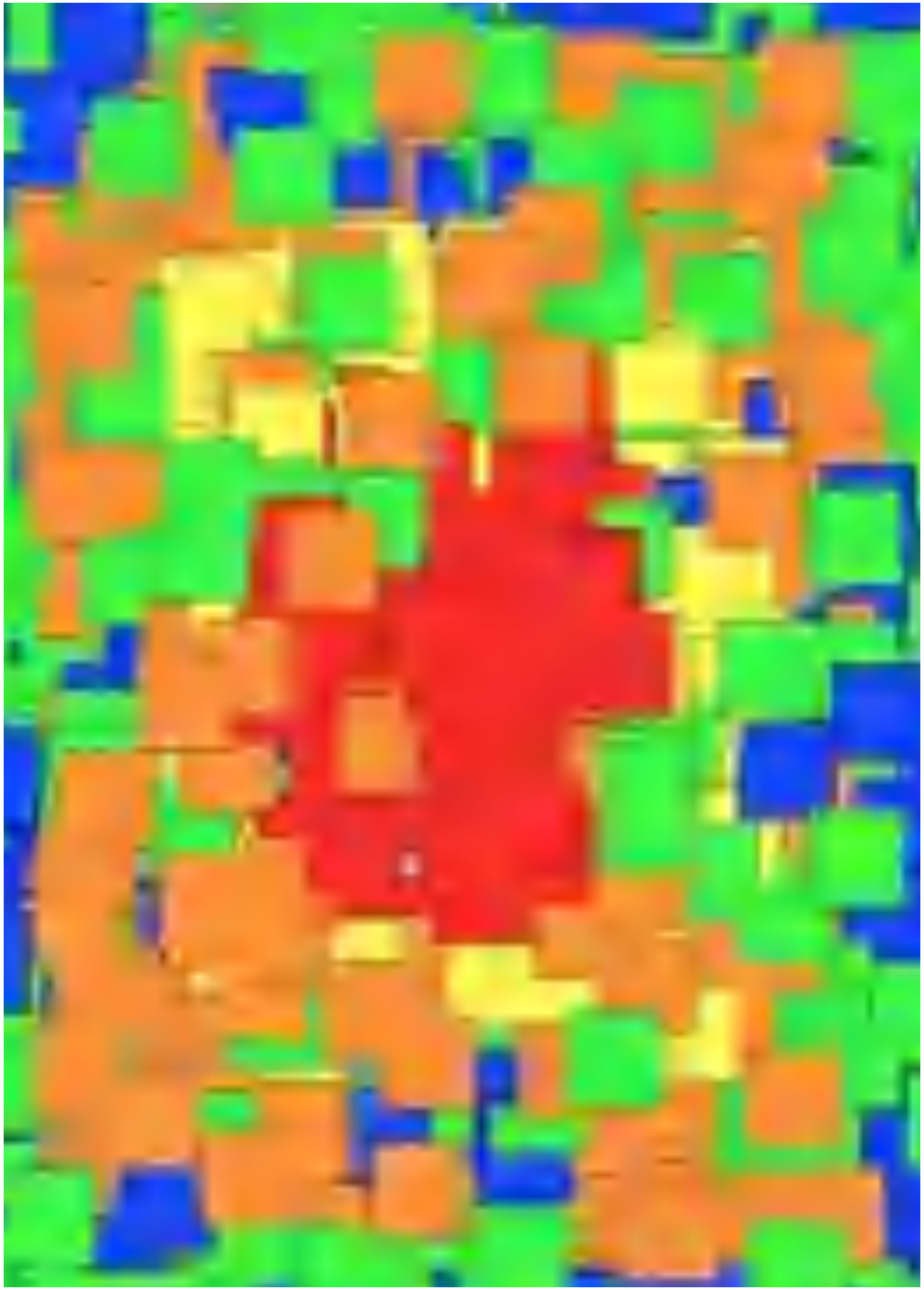
- the step length with **move**,
- the position with **go to**,
- the angle with **turn**,
- the pen thickness with **set pen size to**,
- or the color with **set pen color to**.



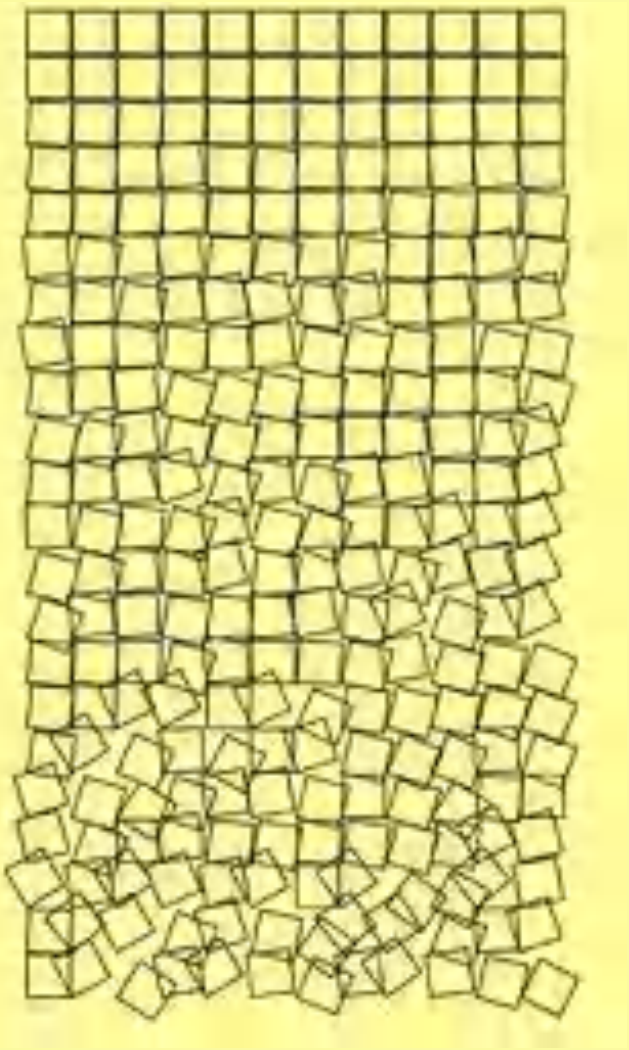
Homage à Nees: Bunch of lines

Homage à Nees: Corner lines

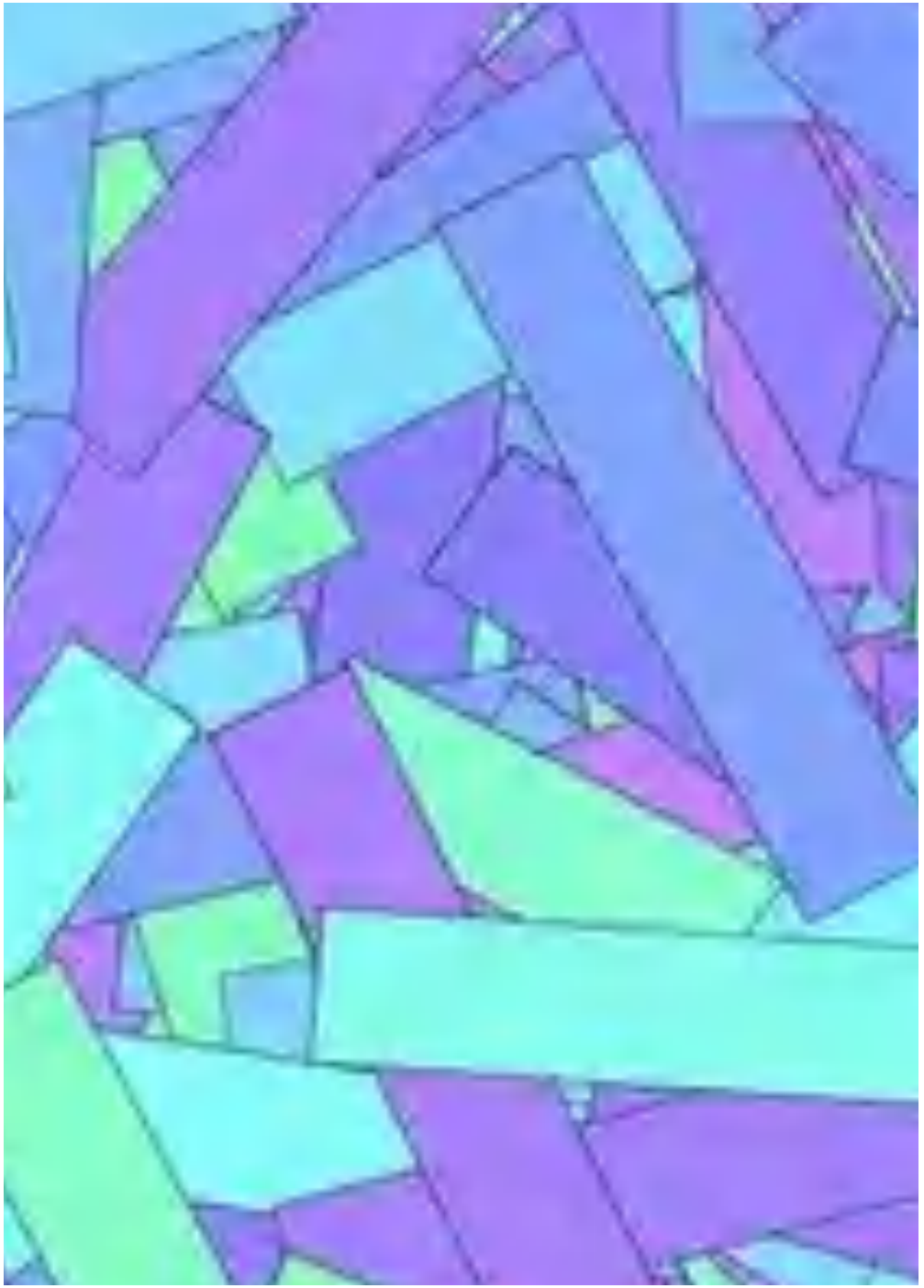




Homage à Franke: Squares

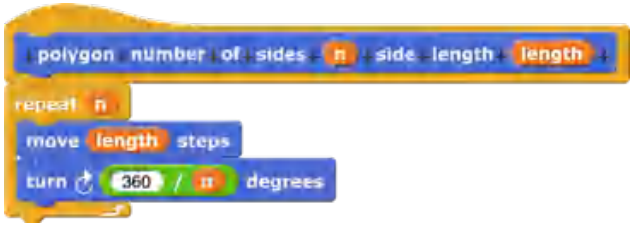


Homage à Nees: Gravel



Polygons

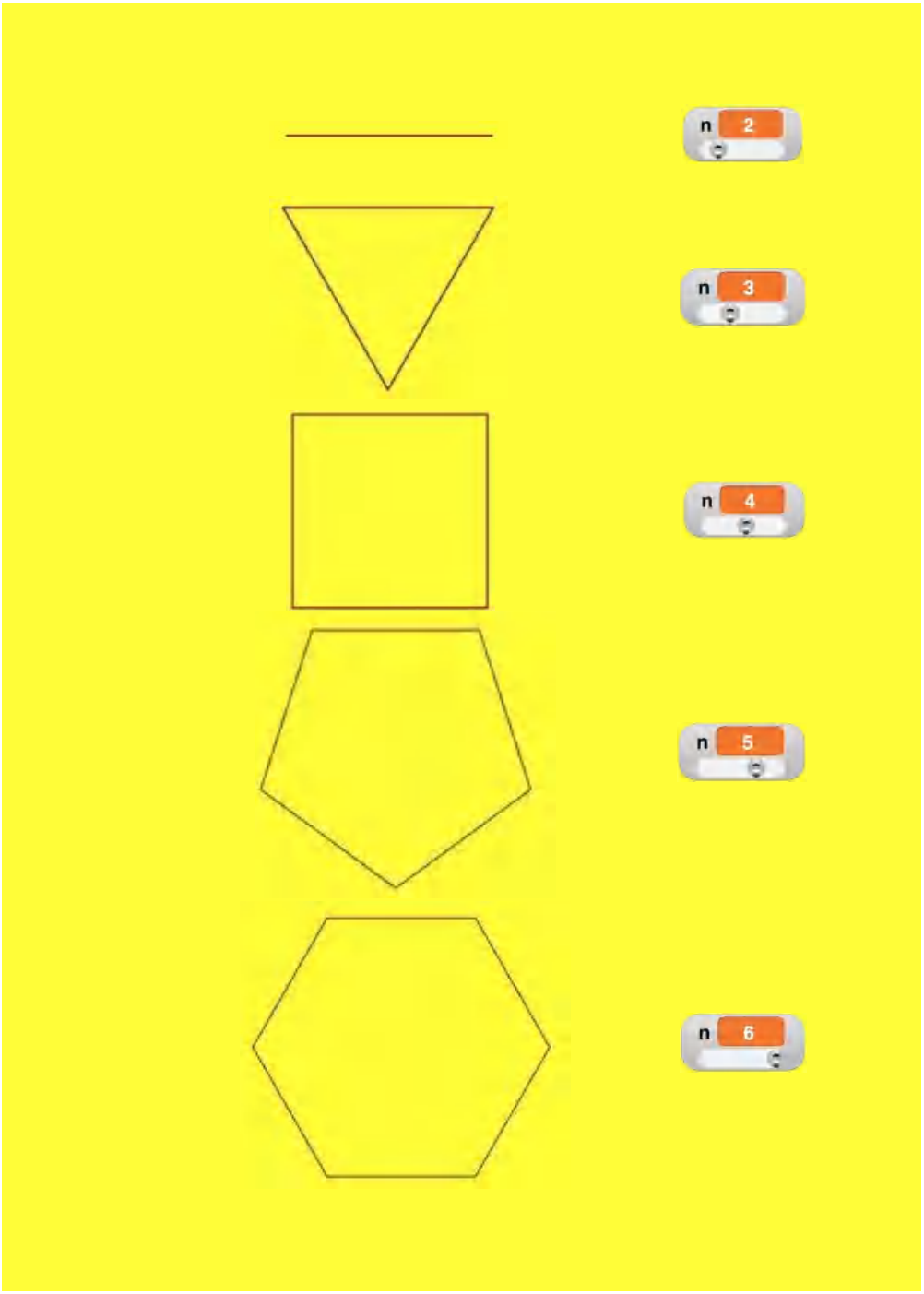
The **generalization** of programs or procedures is the **step from the familiar to the unknown**. A small change in the procedure **square** makes this clear. Just by introducing a parameter **n**, which determines the number of corners of a polygon, instead of squares we can draw any regular polygon in a new procedure **polygon** with sides of equal length and equal angles.



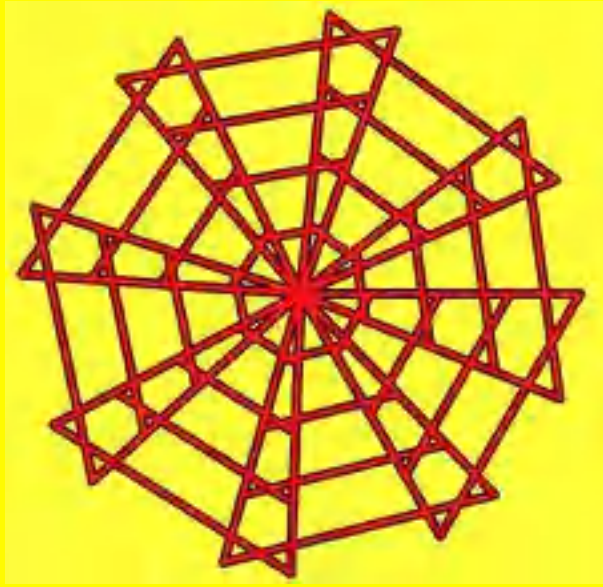
Compared to the procedure **square**, only the angle of rotation changes in addition to the input values. With **360/n** it is ensured that the angle sum in the polygon always results in 360 degrees.

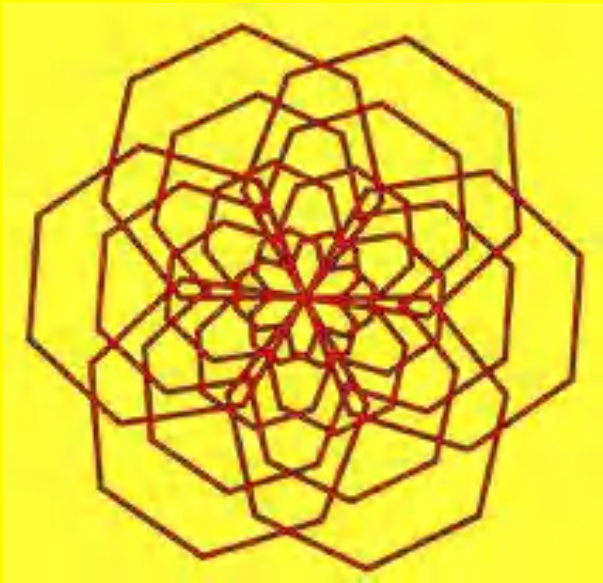
Interesting: In all regular polygons the turtle turns 360 degrees. Papert (Papert, 1982, p. 76) calls this **The Total Turtle Trip Theorem**:

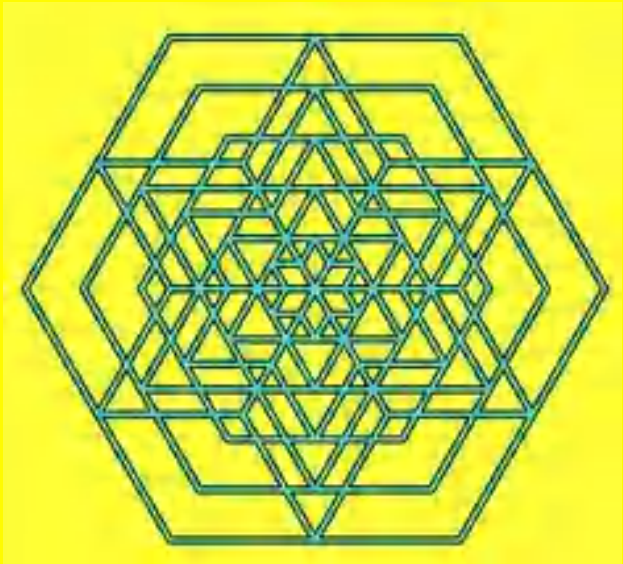
„If a Turtle takes a trip around the boundary of any area and ends up in the state in which it started, then the sum of all turns will be 360 degrees.“

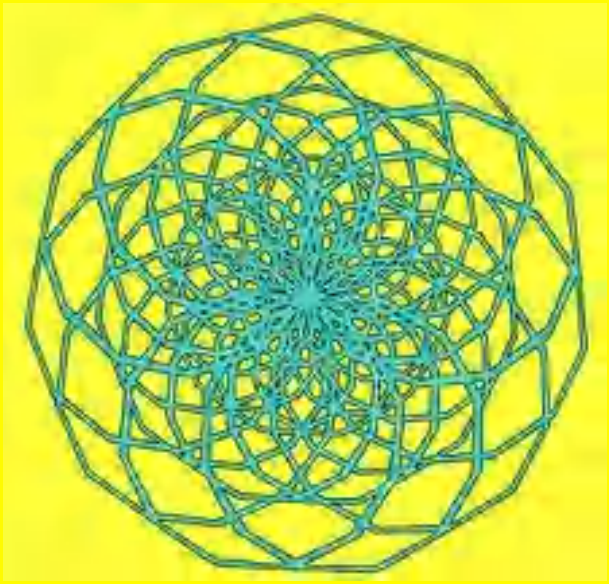


nested polygons

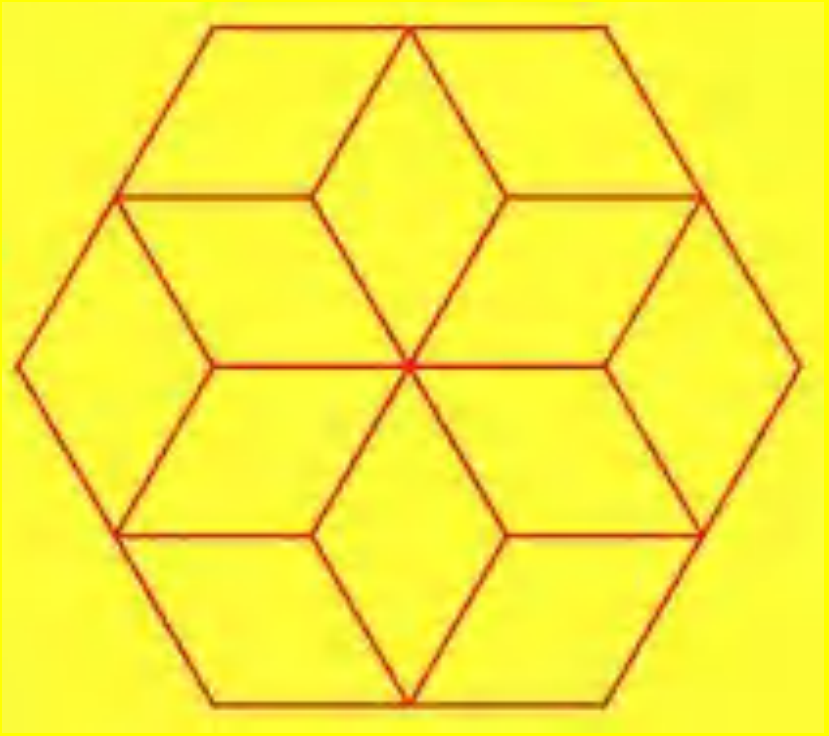


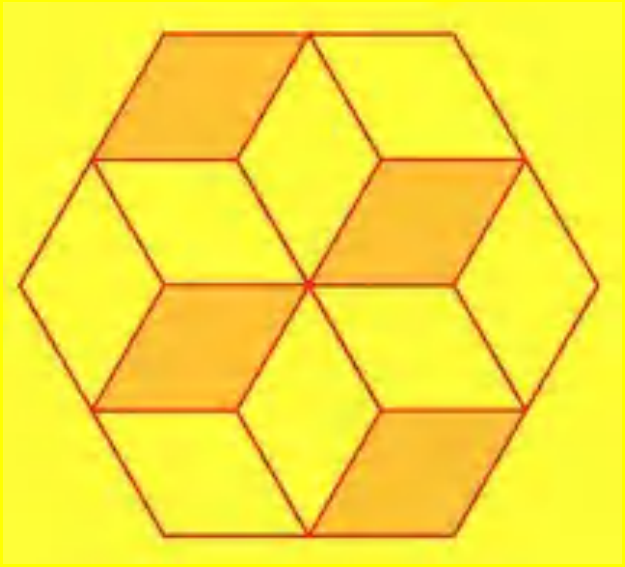


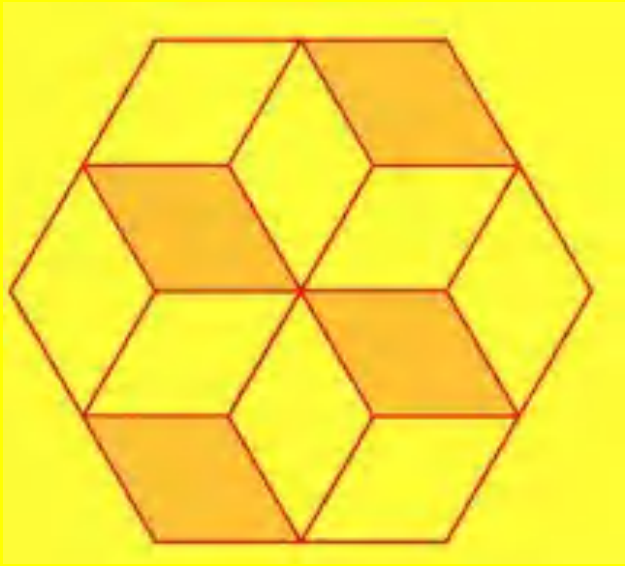


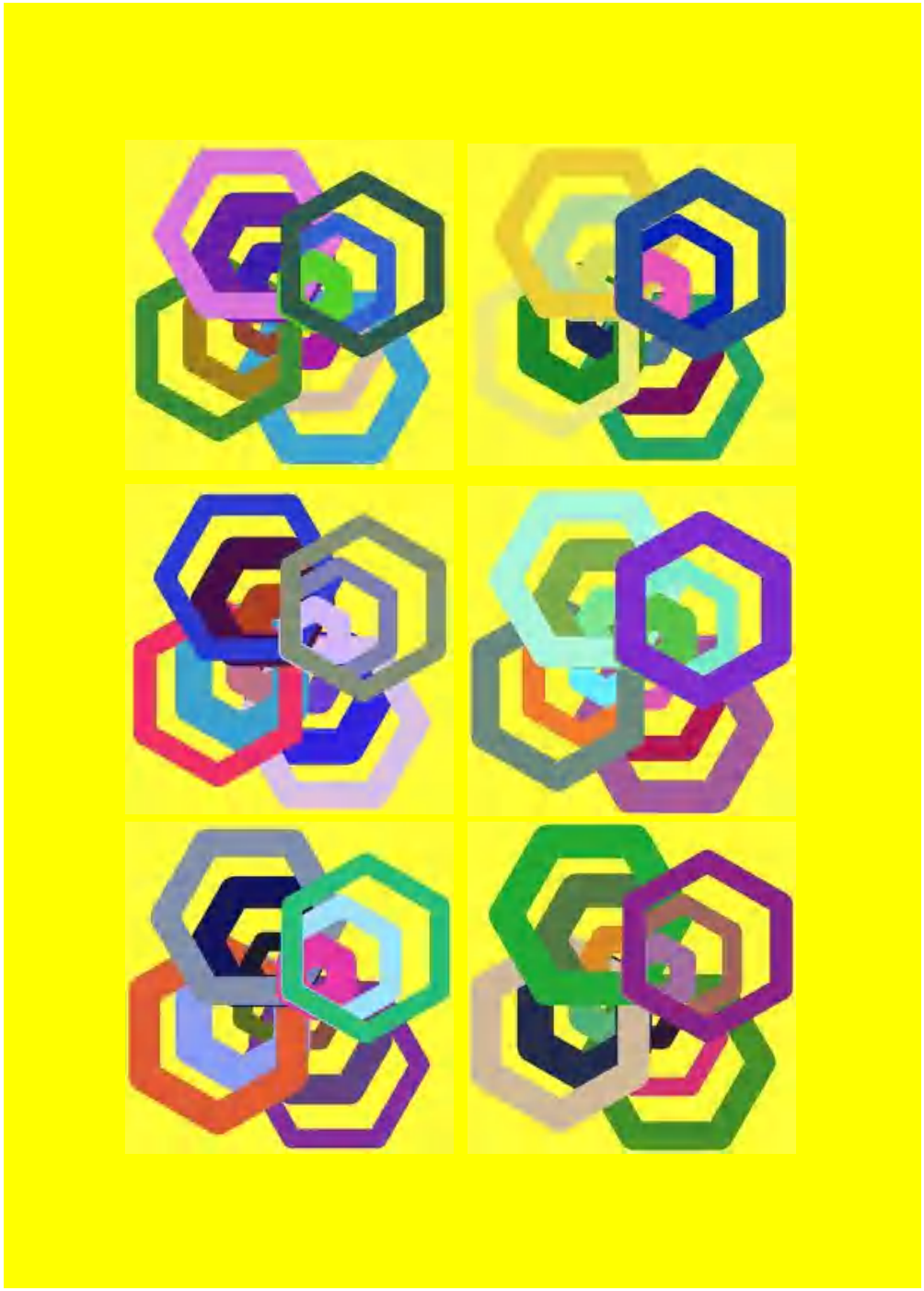


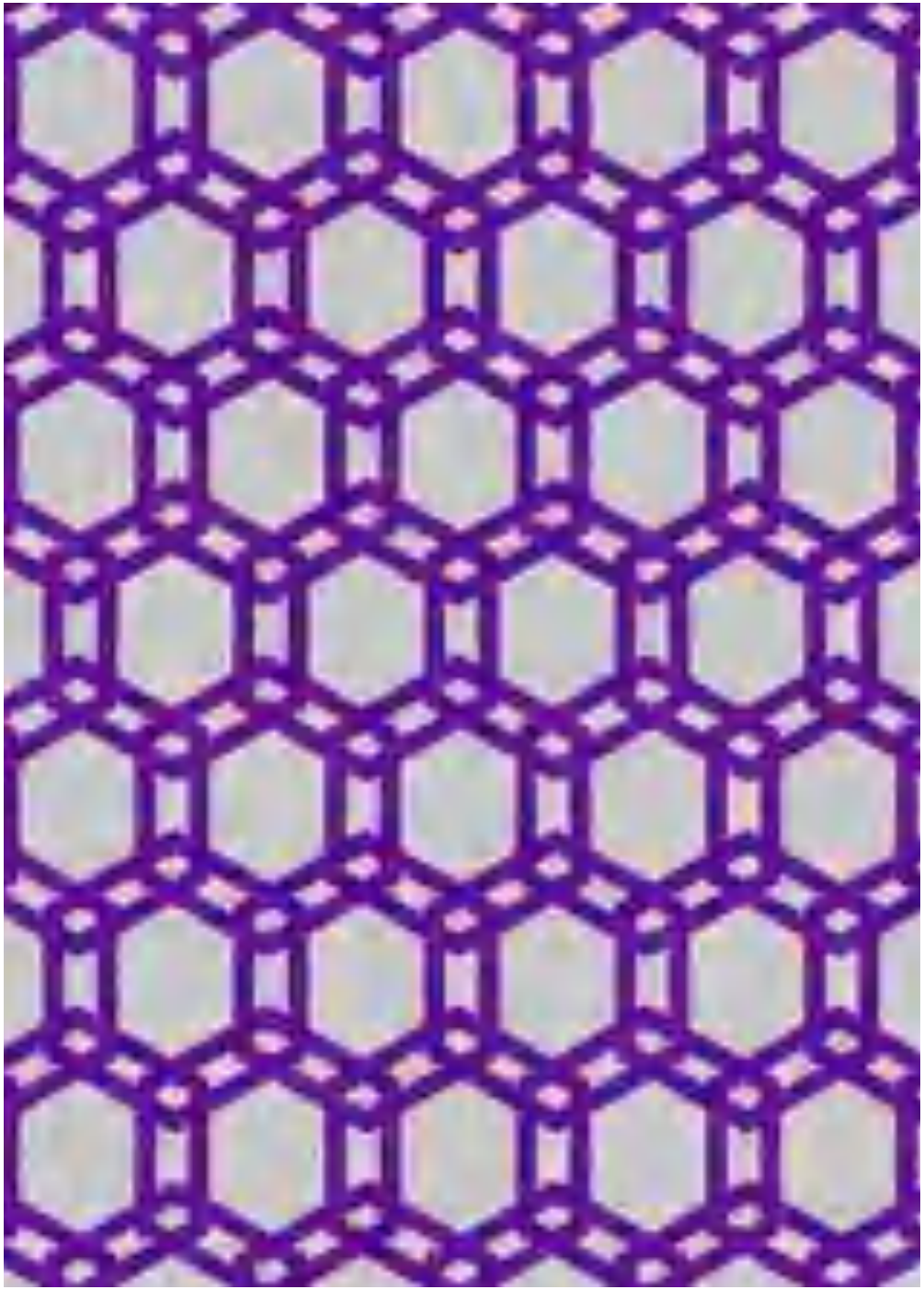
Opticals: Ambiguous Figure Necker Cube
(6 hexagons, angle 60)

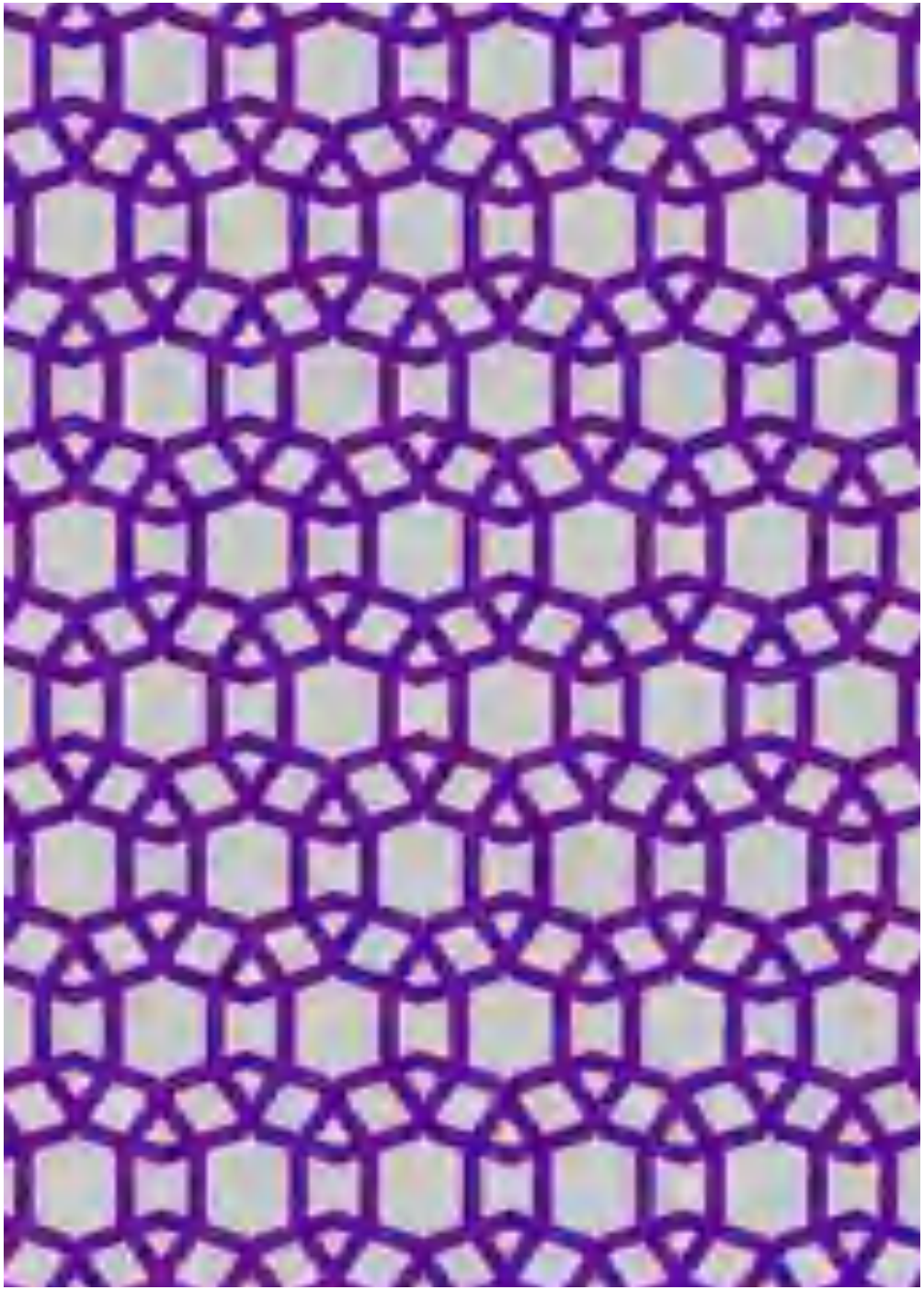


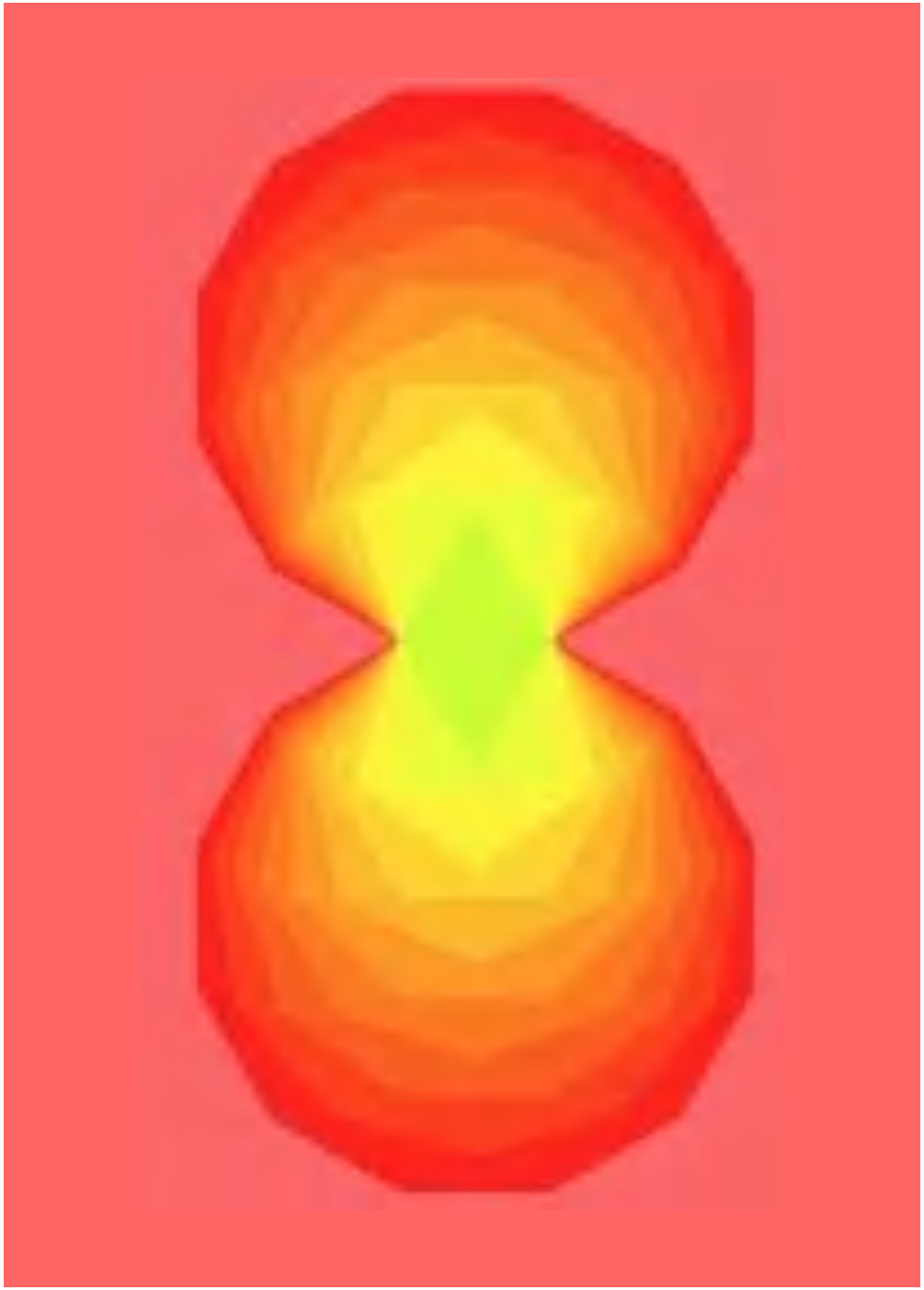


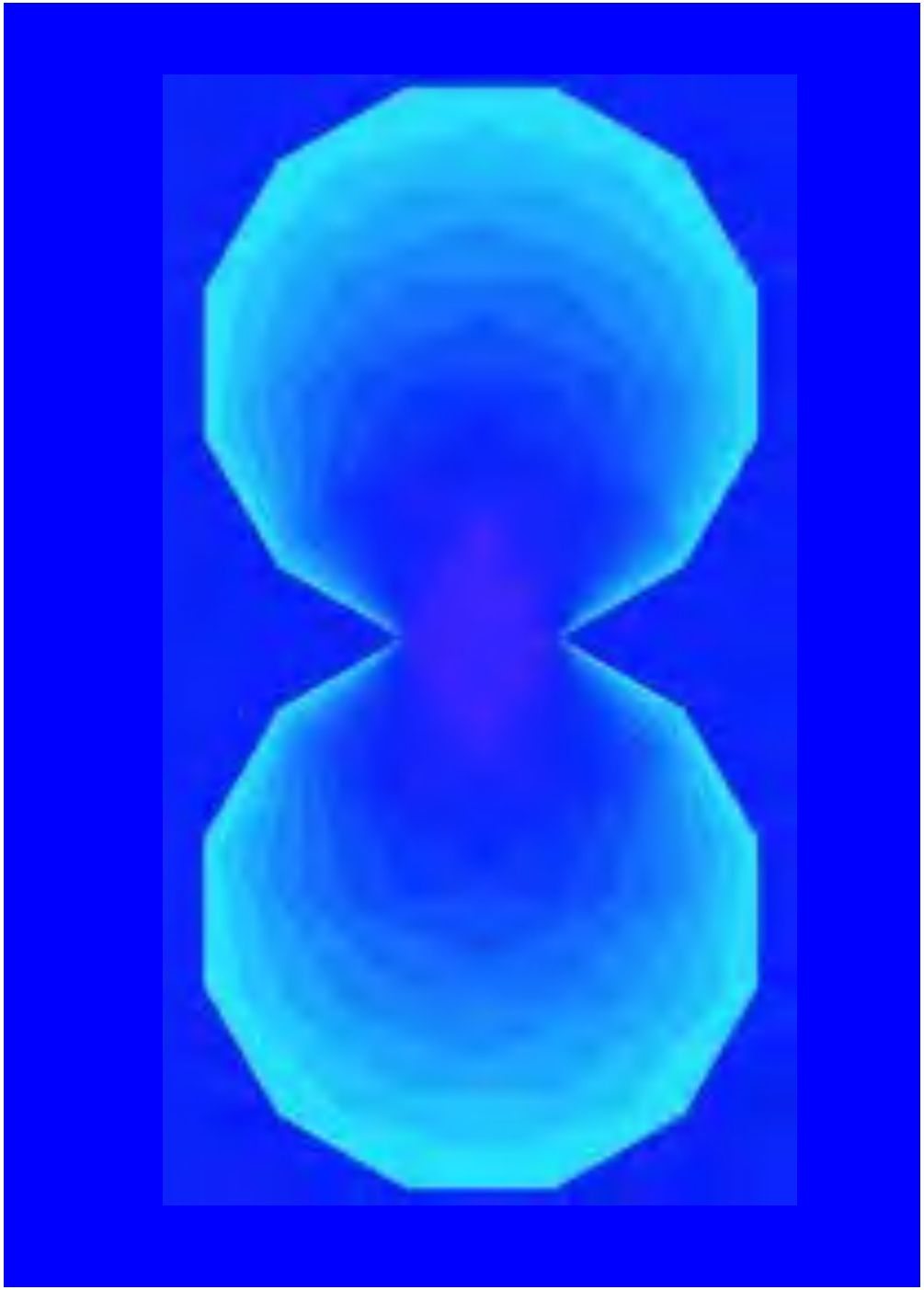








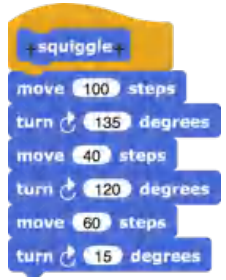




Squiggle, squaggle & Co.

For the complete **Total Turtle Trip Theorem** there is an important generalization. Even with open polygons, the turtle ends in the initial state when the sum of all right and left rotations is again 360 degrees or a multiple thereof.

Brian Harvey introduced an example with the funny name **squiggle** (Harvey, 1982, p. 186 ff.). The always same sequence of pairwise **move** and **turn** movements can be easily and flexibly implemented with a procedure **designs** and by summarizing the movement data in a **list**.

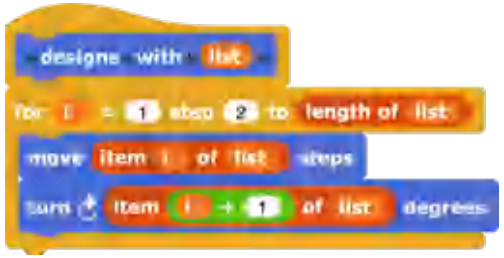


Lists are fields in which numbers and/or texts, lists and even program code can be saved and called up again.

The movement data for **squiggle** and two other variants called **squaggle** and **squoggle** then result in the following lists:

With **designs**, similar variants can now be easily drawn.

But it is only through repetition and by changing the values for the movements that the actually interesting patterns are created.

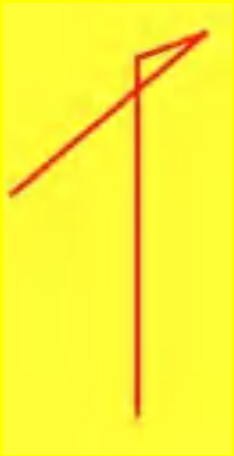


In addition there is a similar example by Hal Abelson (1983, p. 49) with the simple name **design**.

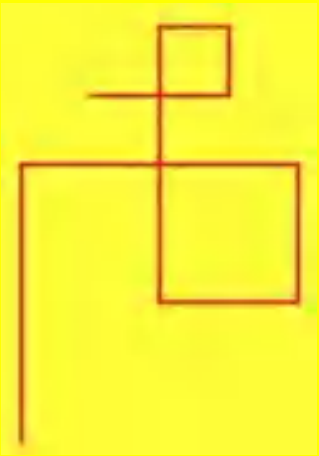
squiggle

squaggle

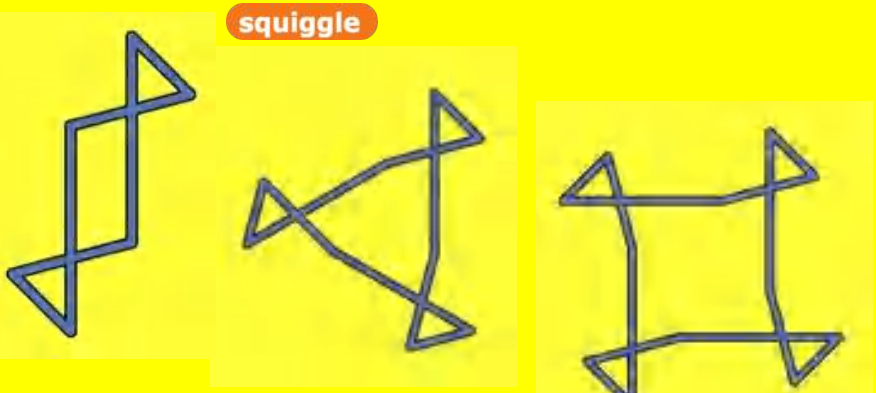
squoggle



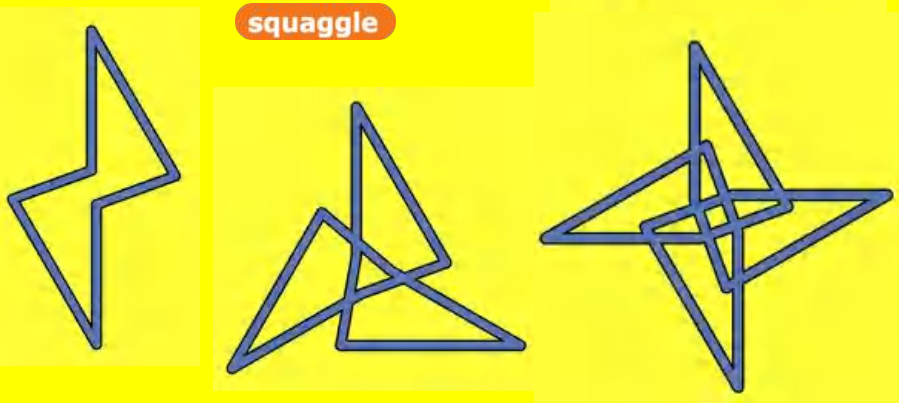
design



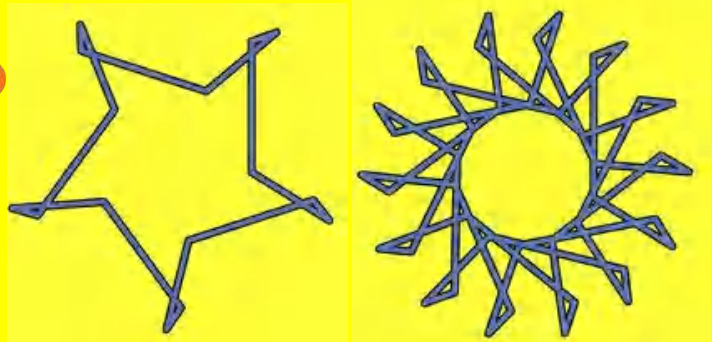
squiggle

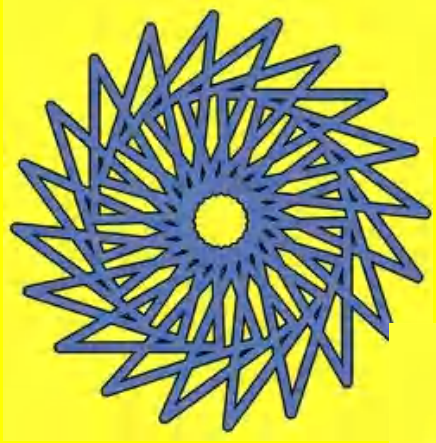


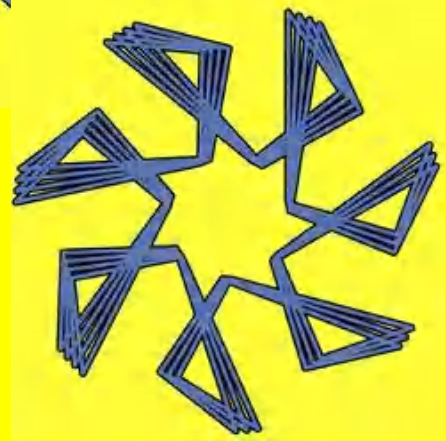
squaggle

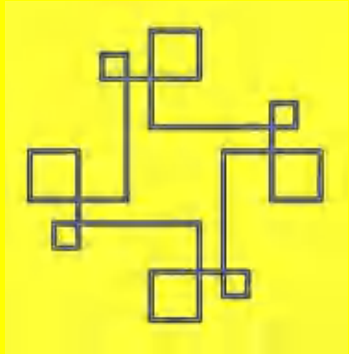


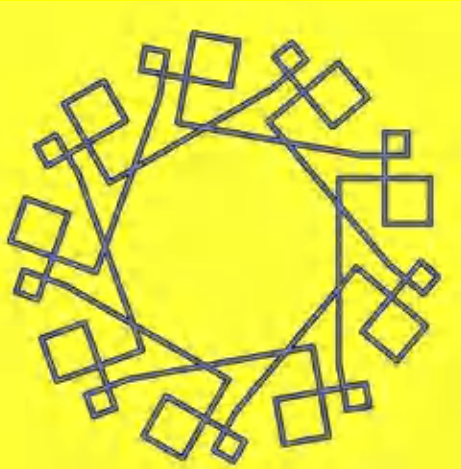
squoggle

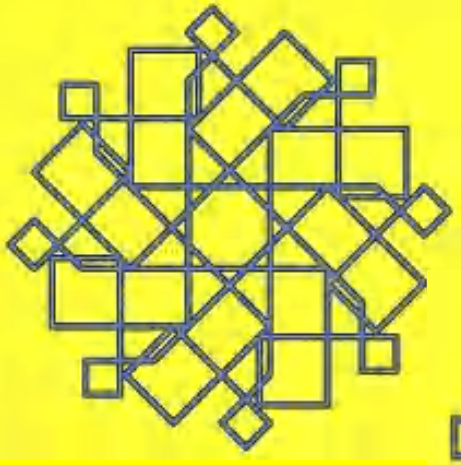


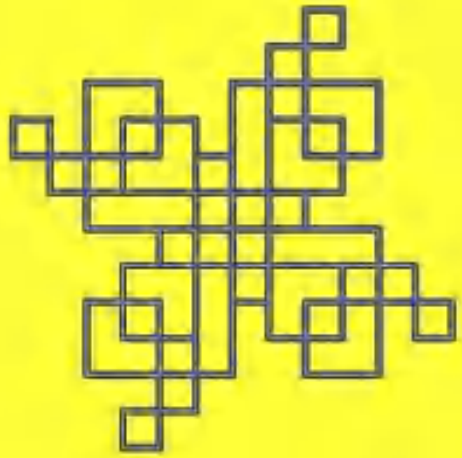


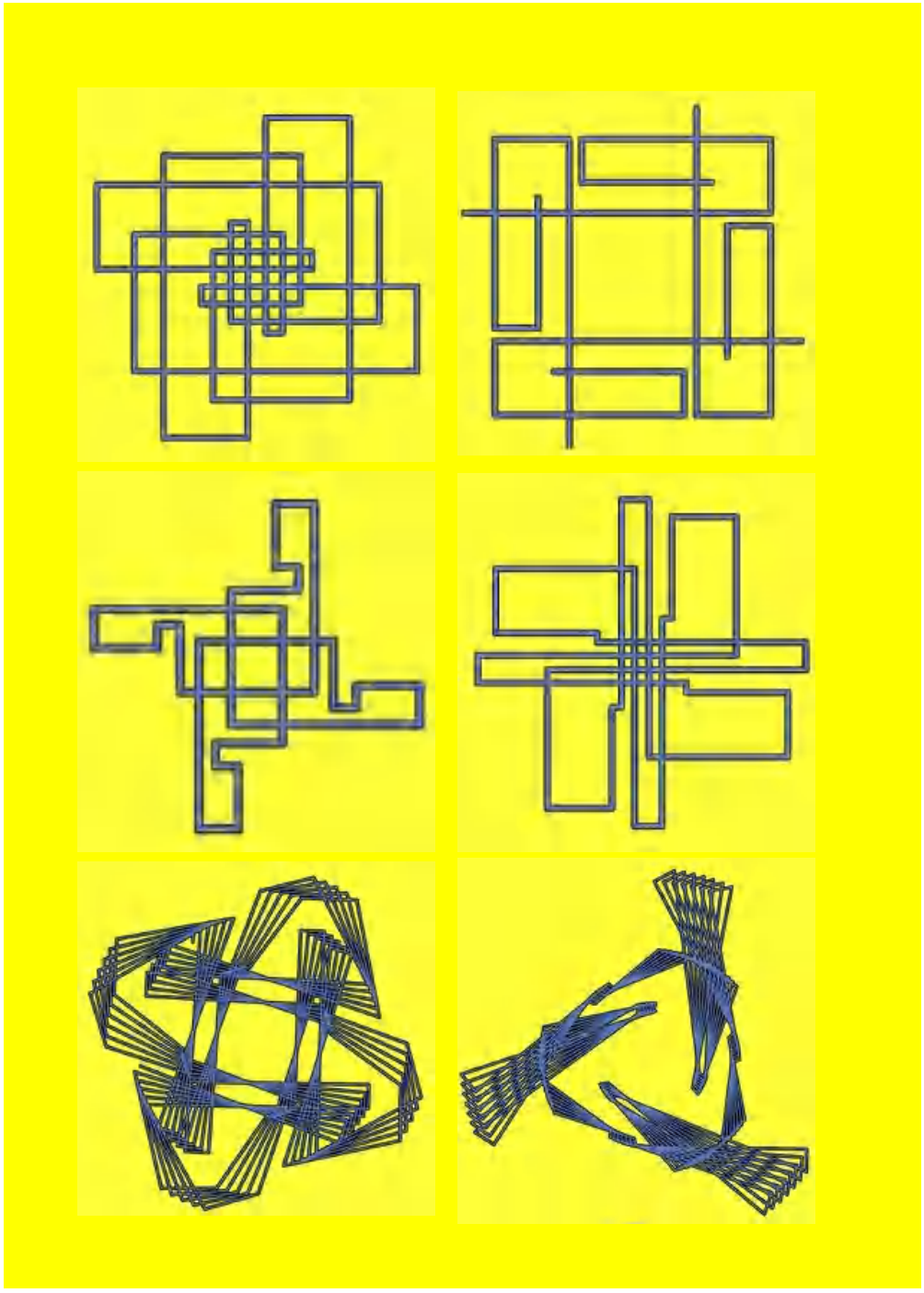


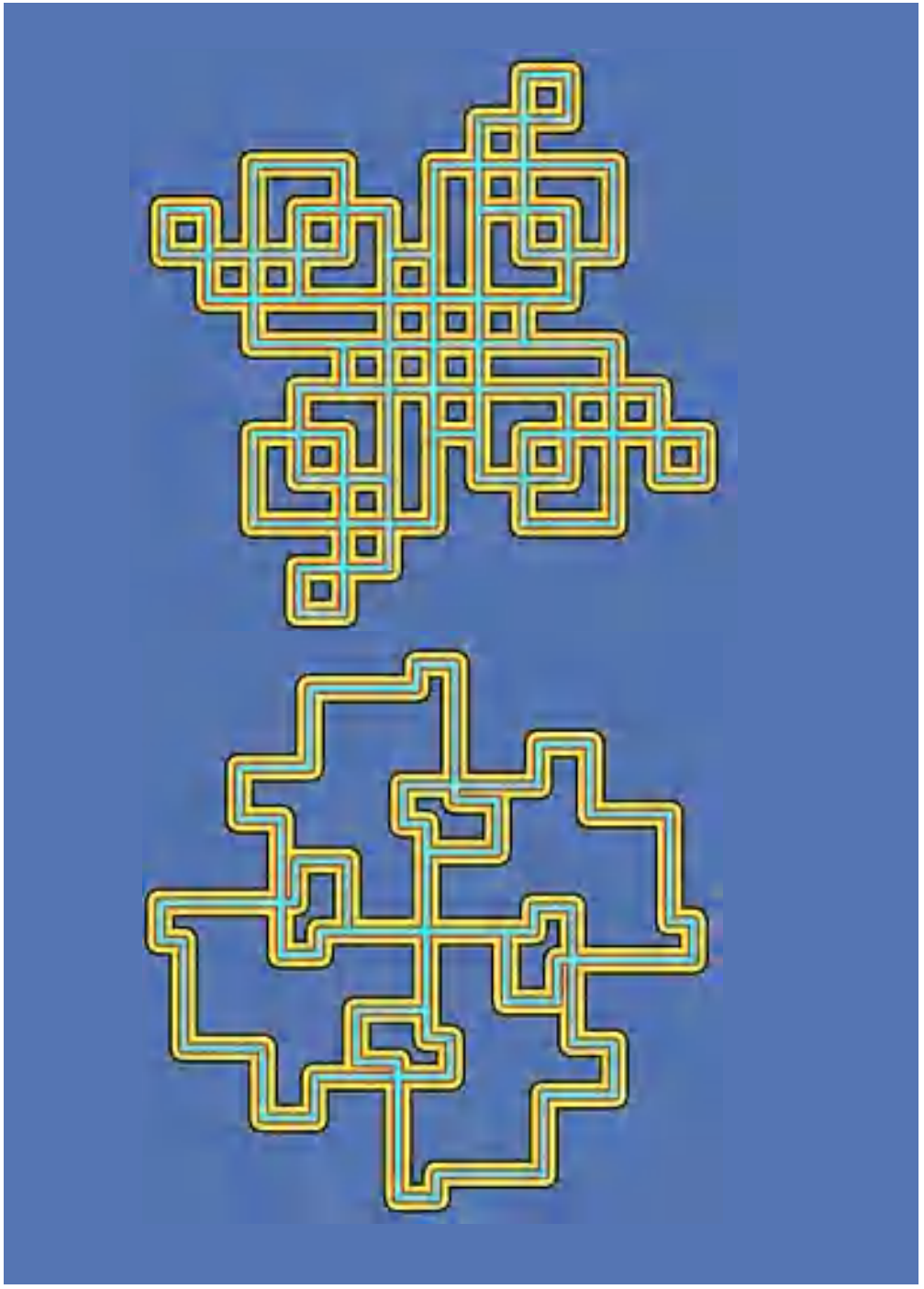


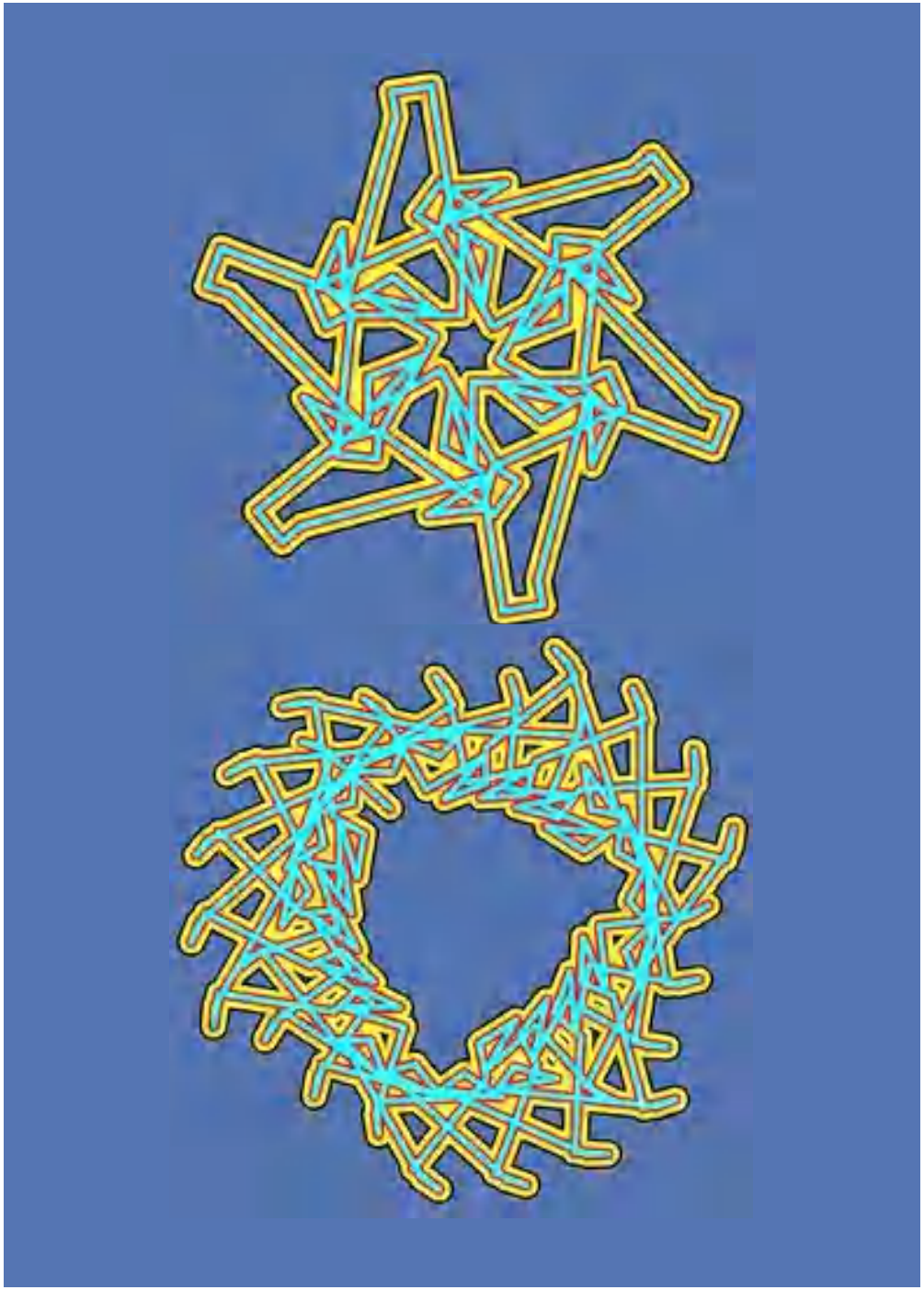












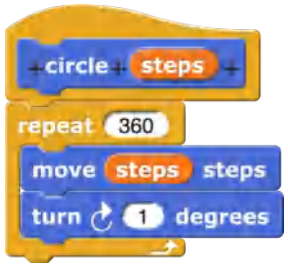
Circles (I)

When drawing figures, Papert liked to let the children play turtle themselves. He speaks of **bodysyntonic learning** or **ego-syntonic learning** to make clear the relationship between geometric operations and physical experience.

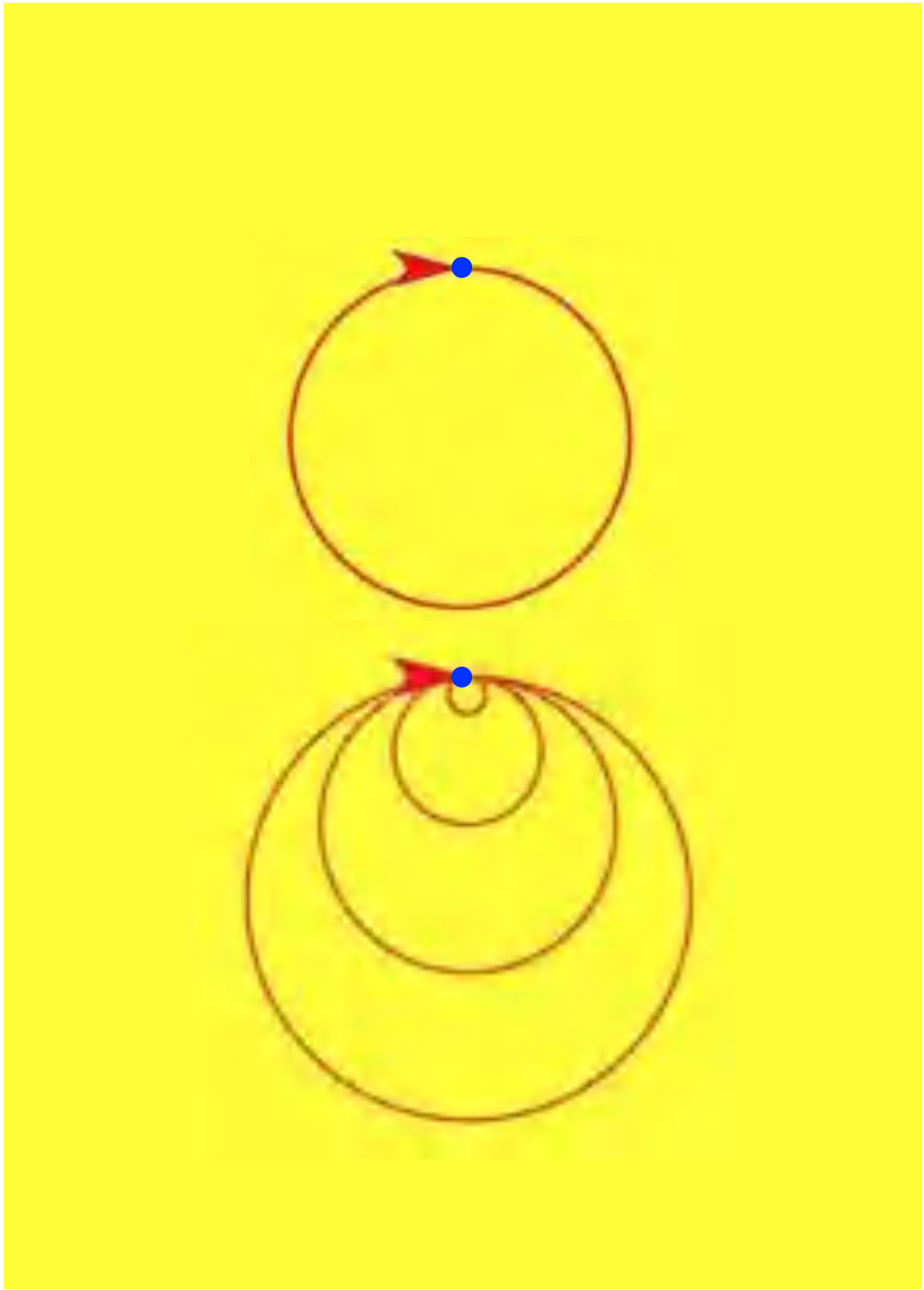
A prime example of this is the drawing of a circle: from the repeated sequence of body movements "*take a small step forward, turn a little*" a circle is actually formed.

From the verbal description it is only a small step to the program. A repeat loop with the **repaet** block shows the expected result.

If circles of different sizes are to be drawn, this can be achieved in very different ways. One possibility is a procedure **circle**, where the radius of the circle is determined by the number of **steps** of the turtle:

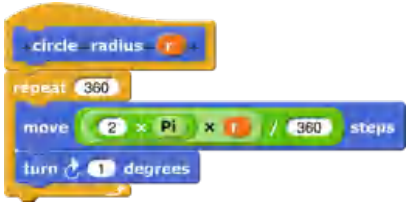


The procedure **circle** is thus very similar to the procedure **polygon**, whereby here the angle is kept constant at one degree.



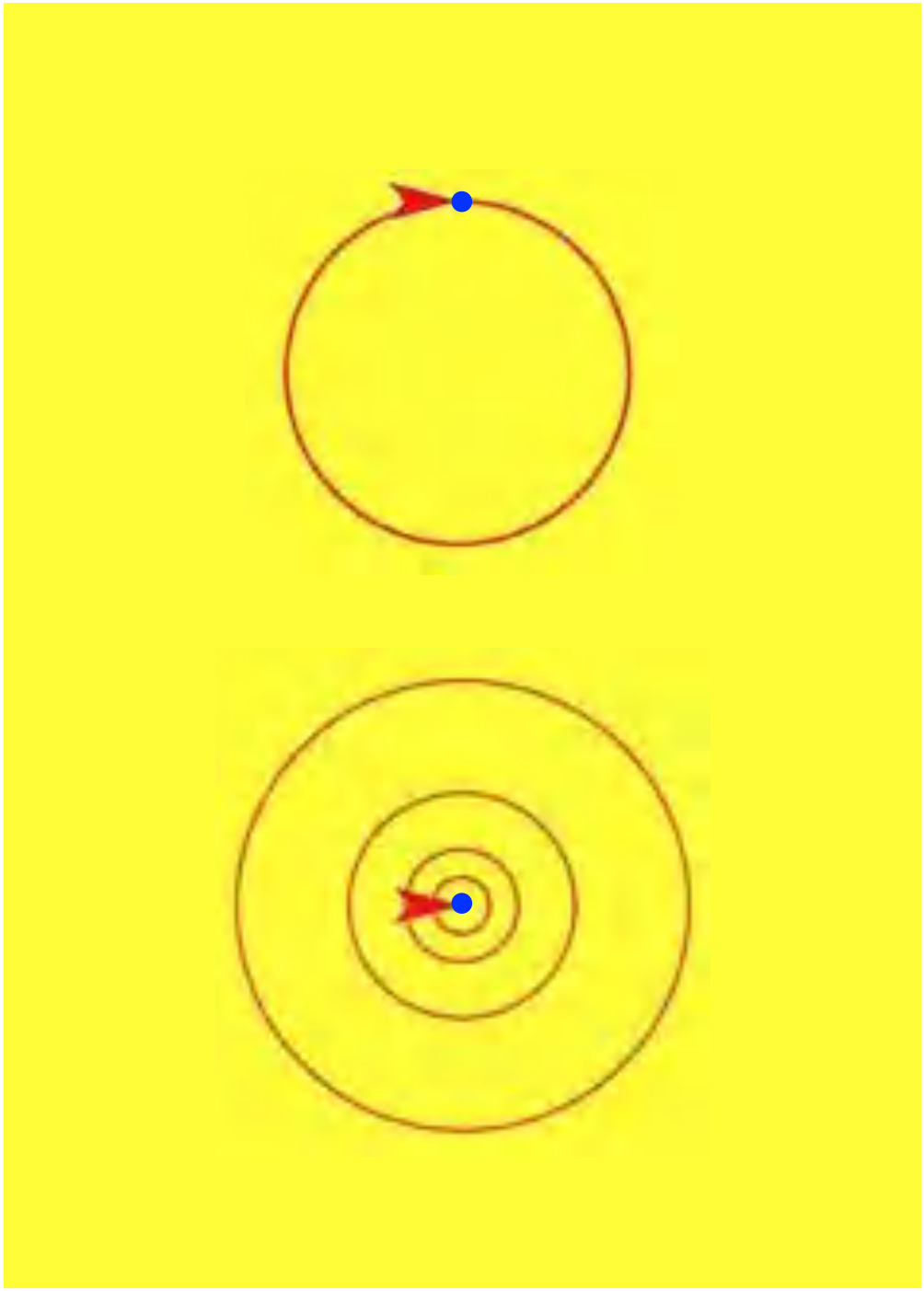
Circles (II)

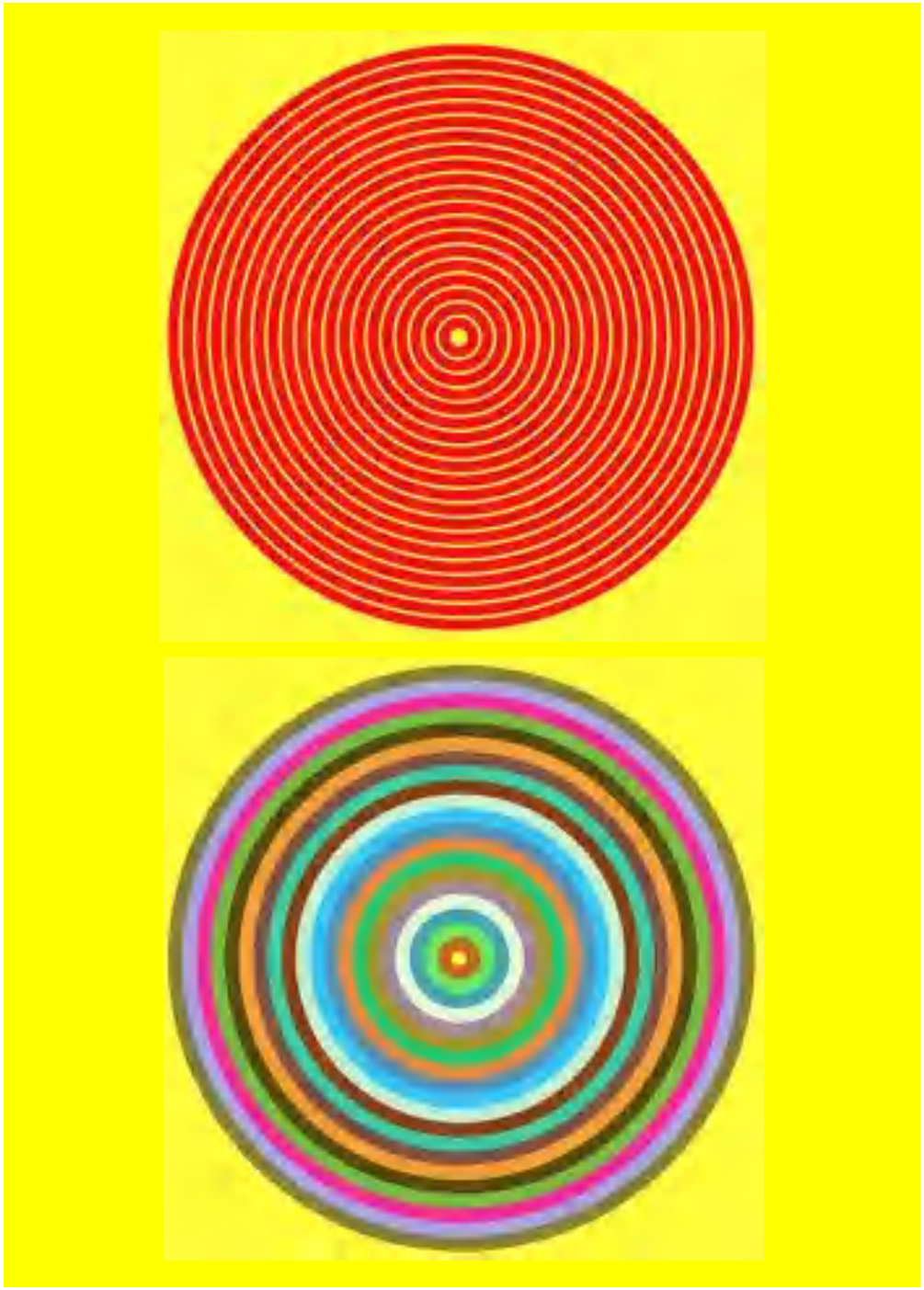
With a little mathematics, the circles can be determined more precisely. From the formula for the **circumference** and the **mathematical constant π** the required step sizes of the turtle can be calculated using the **radius**: **U = 2 π r.** This results in the procedure **circle radius**:

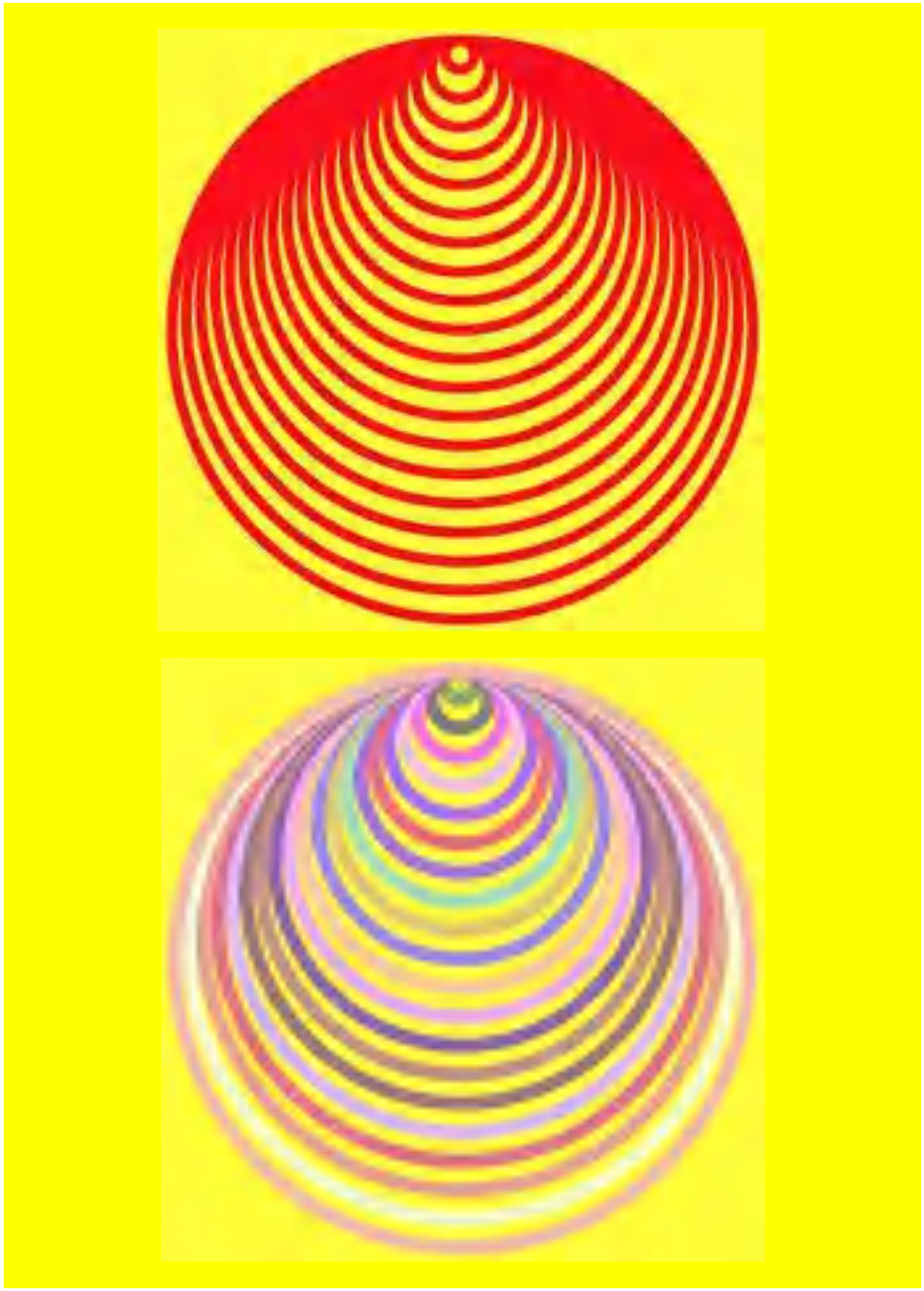


When drawing circles on paper, their position is determined with the ruler, and their radius with the compass around the selected center. A procedure **circle araund** that corresponds to this is somewhat more extensive:

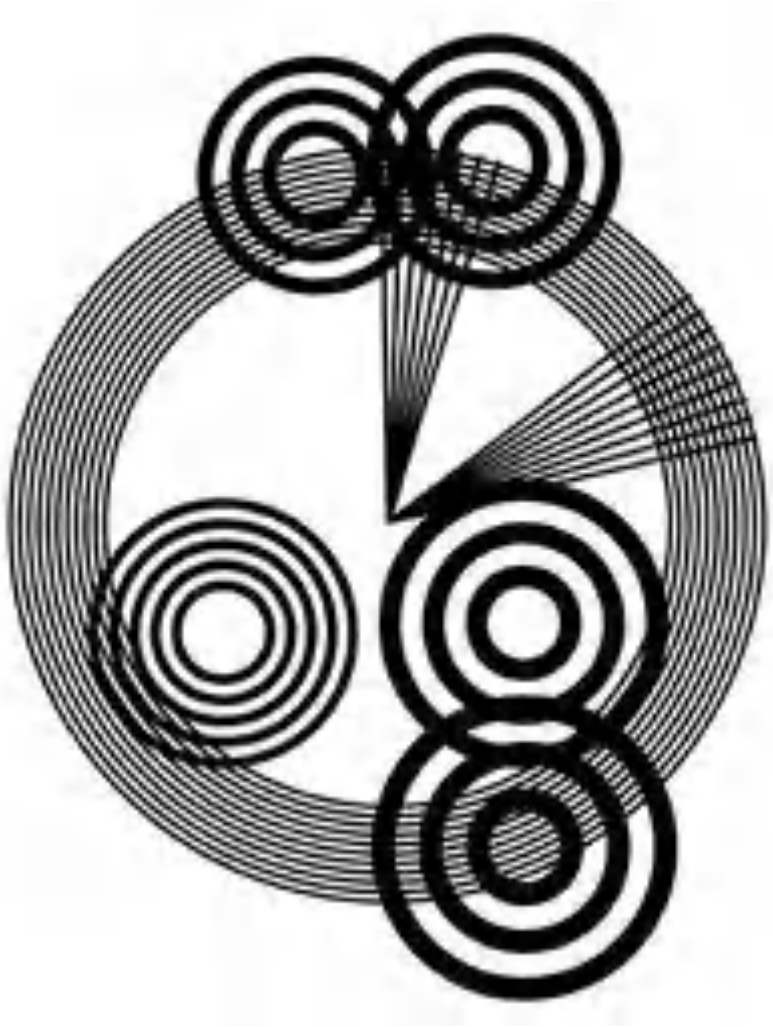
- First the turtle is sent to the selected point **[x,y]**.
- From this point it is sent with **radius** to a starting point on the circle and
- draws the circle from there.
- Finally, the turtle is sent back to the starting point.







lines and circles



Homage à Delaunay: Rhythm

Homage à Calder

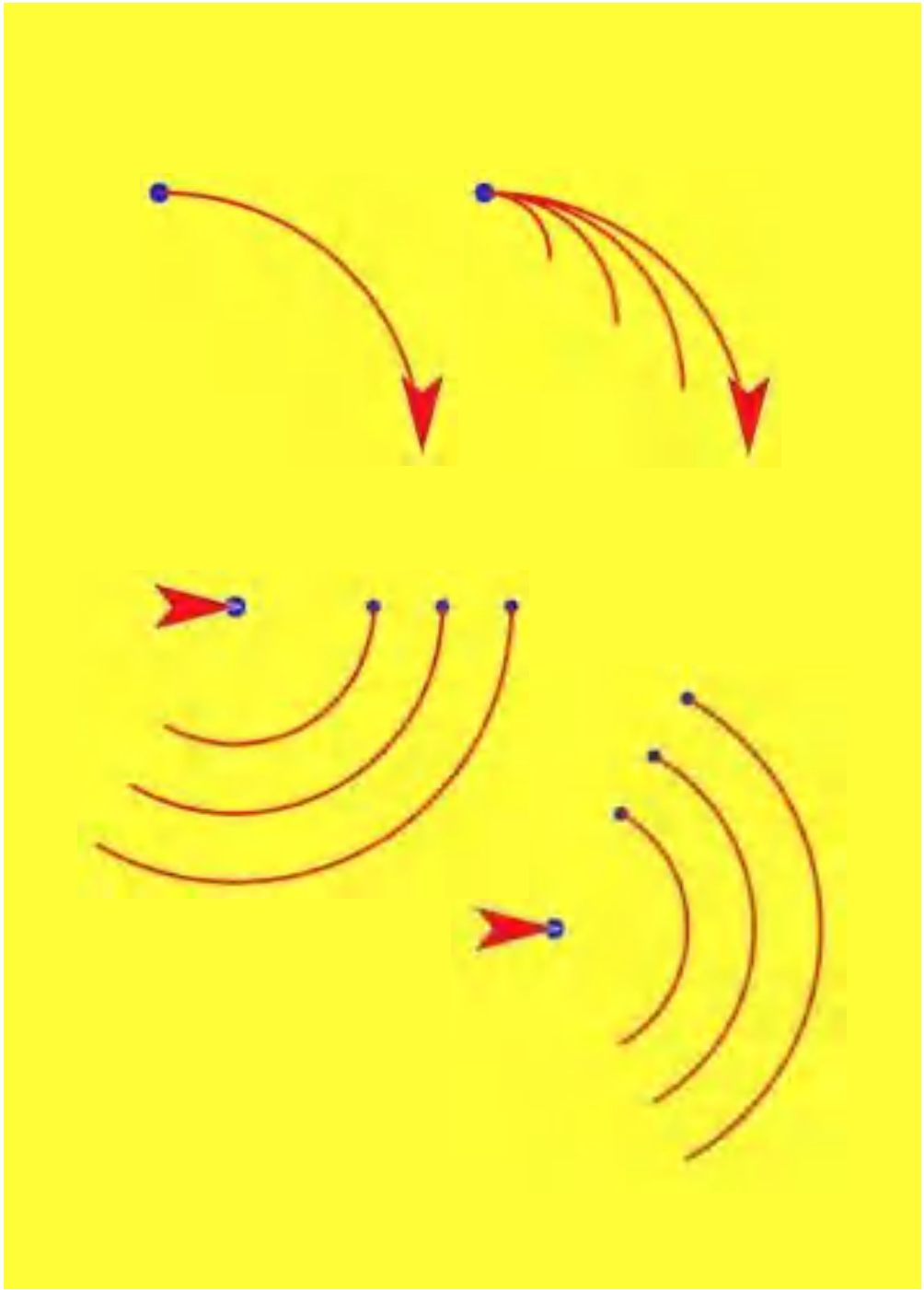
Arcs

Circles are the prime example of the relationship between geometric operations and physical experience, as described by Papert. They can also be used to show that **omission** can create new **diversity**. Thus circles can be turned into arcs by limiting steps and rotations.

As with circles, there are again several ways to create them, starting from the center point.

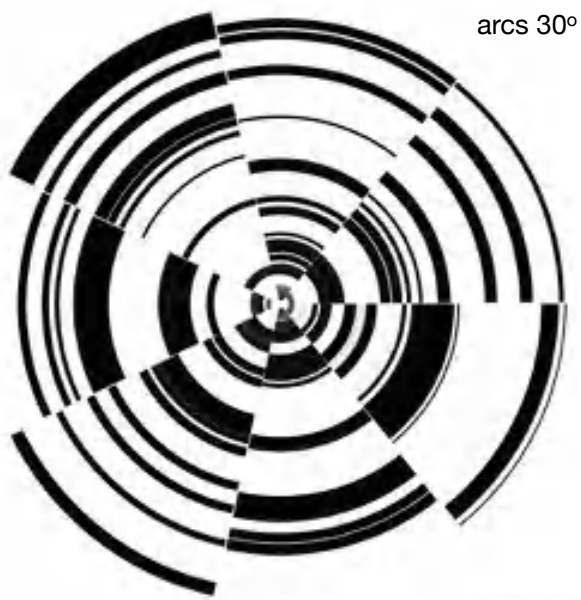
Drawing with a selected center point is also obvious for circular arcs. A distinction must be made between arcs that start **from a point** and arcs that go **around a point**. These procedures are now somewhat more extensive:

By combining them many new interesting figures are created.



arcs 90º

arcs 30º < random < 60º

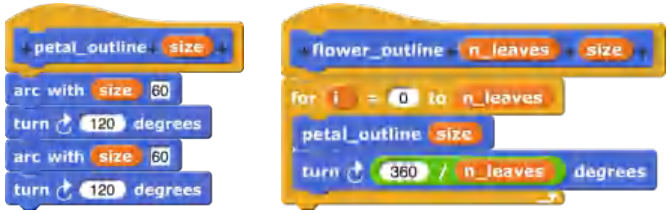


arcs 180º

shifted arcs 180º

Arcs - applications with variations

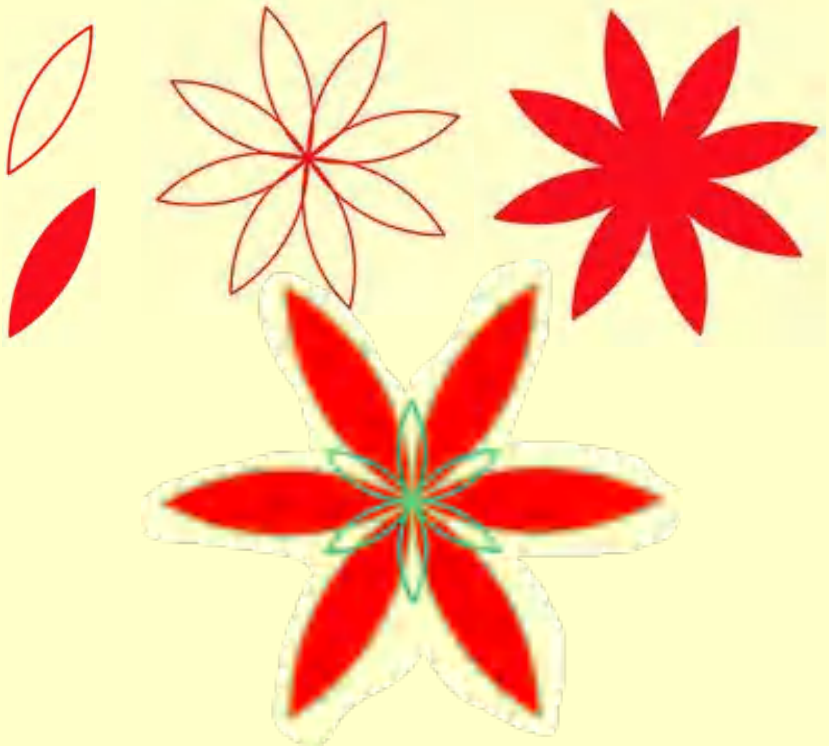
Arcs can be used as the basis for more complex figures. Leaves can be drawn very easily, which in turn can be combined into flowers.



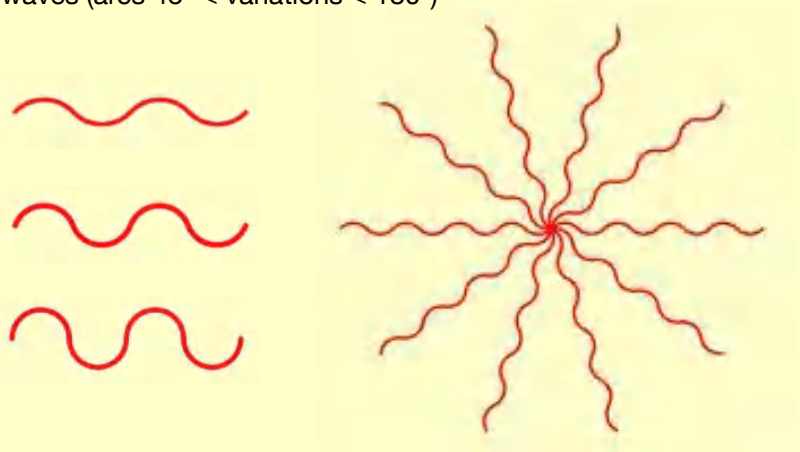
With the **fill** command, colored filled petals can also be created and used in appropriate combinations.

Further variations are possible, if one distinguishes between right- and left-turning arcs (which internally differ only by the commands **turn 1 degrees** or **turn -1 degrees**). Quite different waves can result.

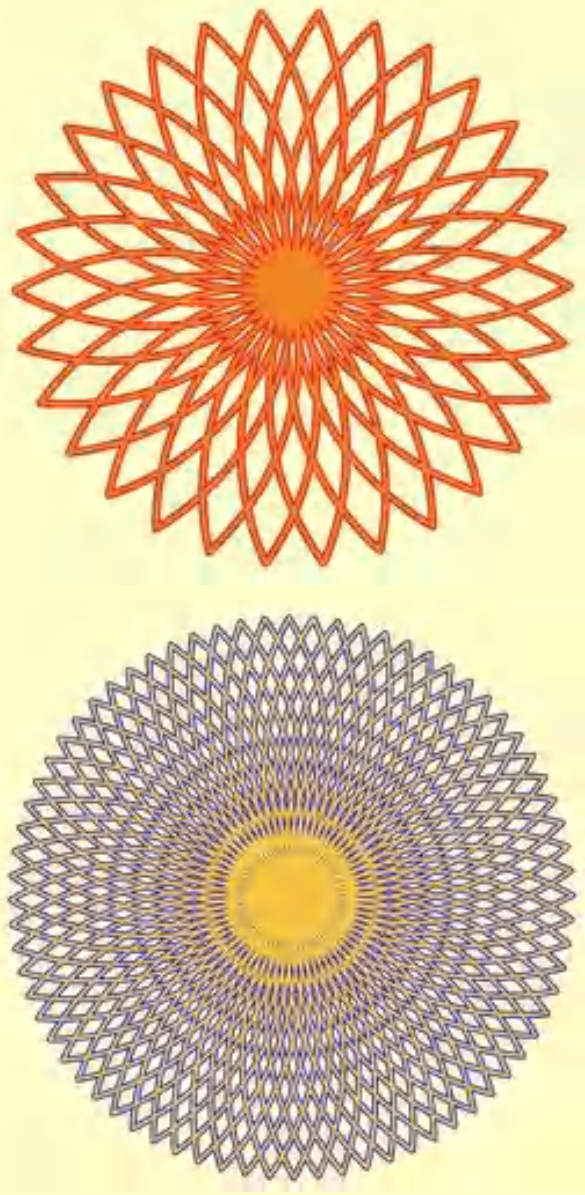
leaf & flower (arcs 120º)



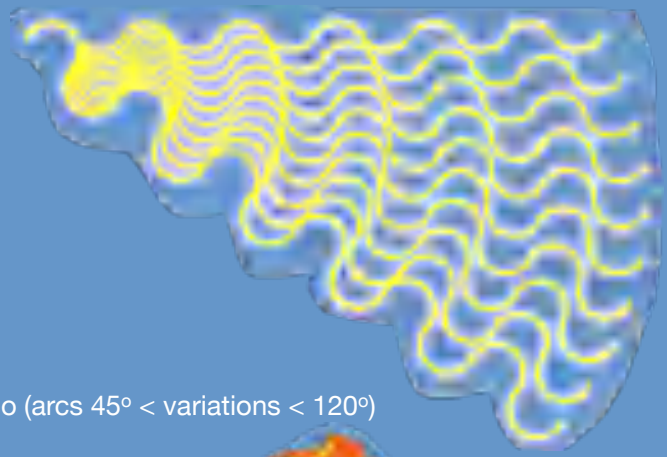
waves (arcs 45º < variations < 180º)



flowers (arcs 120º)

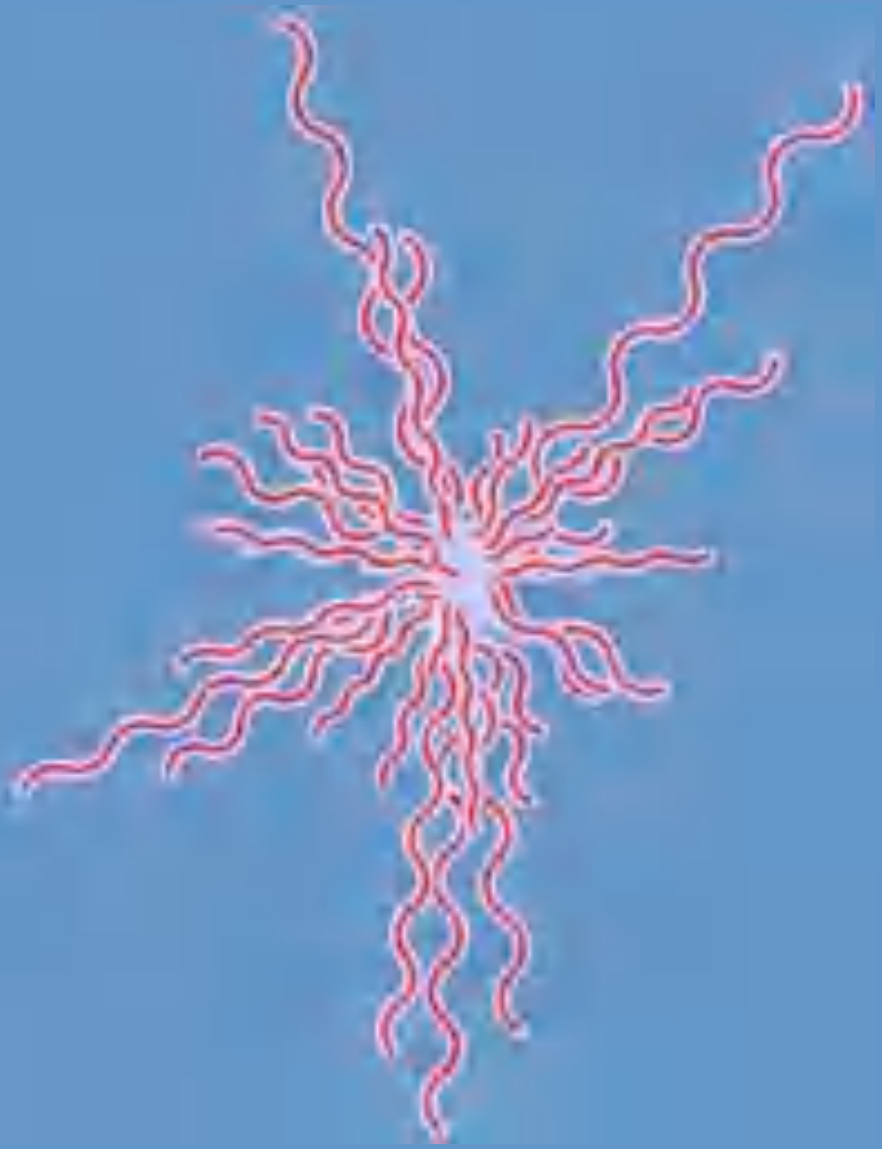


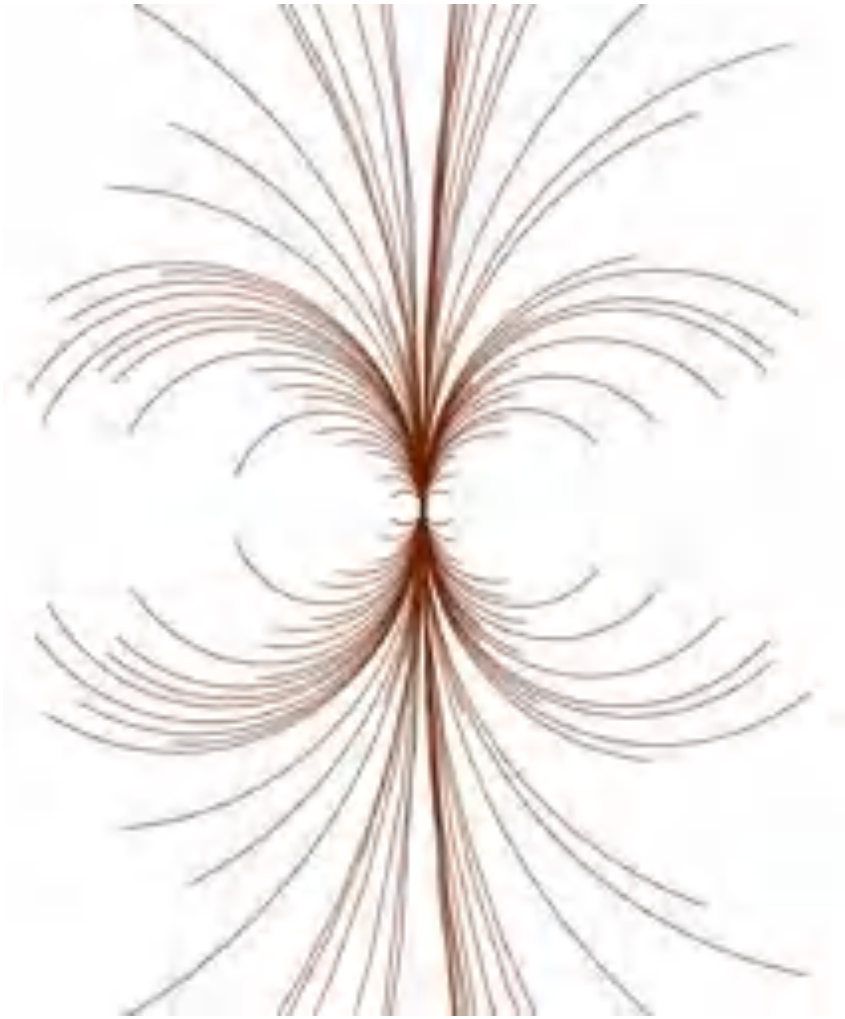
waves (arcs 120° < variations < 200°)



volcano (arcs 45° < variations < 120°)

amoeba (arcs 45º < variants < 120º)





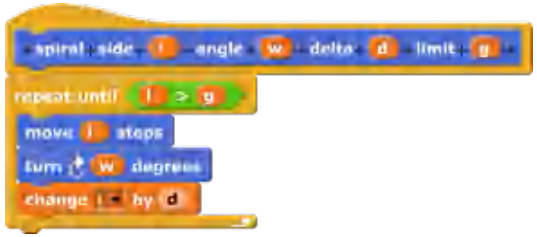
Homage à Nees: Arcs (arcs 90º - 360º)

Homage à Sýkora: Lines (arcs 30° - 120°)

Spirals

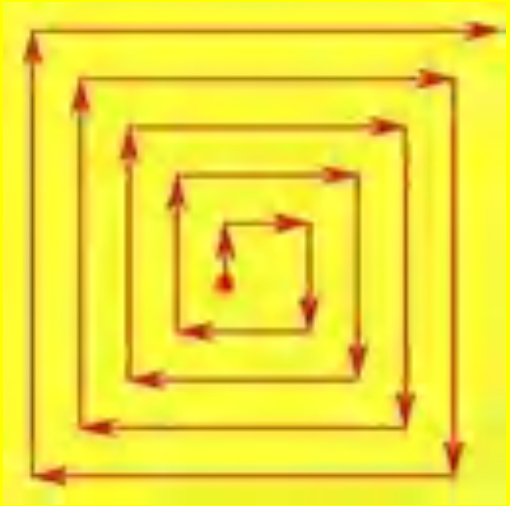
If in the regular polygons the **lengths** of the sides and the **angles** between the sides become variable, we get **spirals**. There are several possibilities for this, which result in interesting differences.

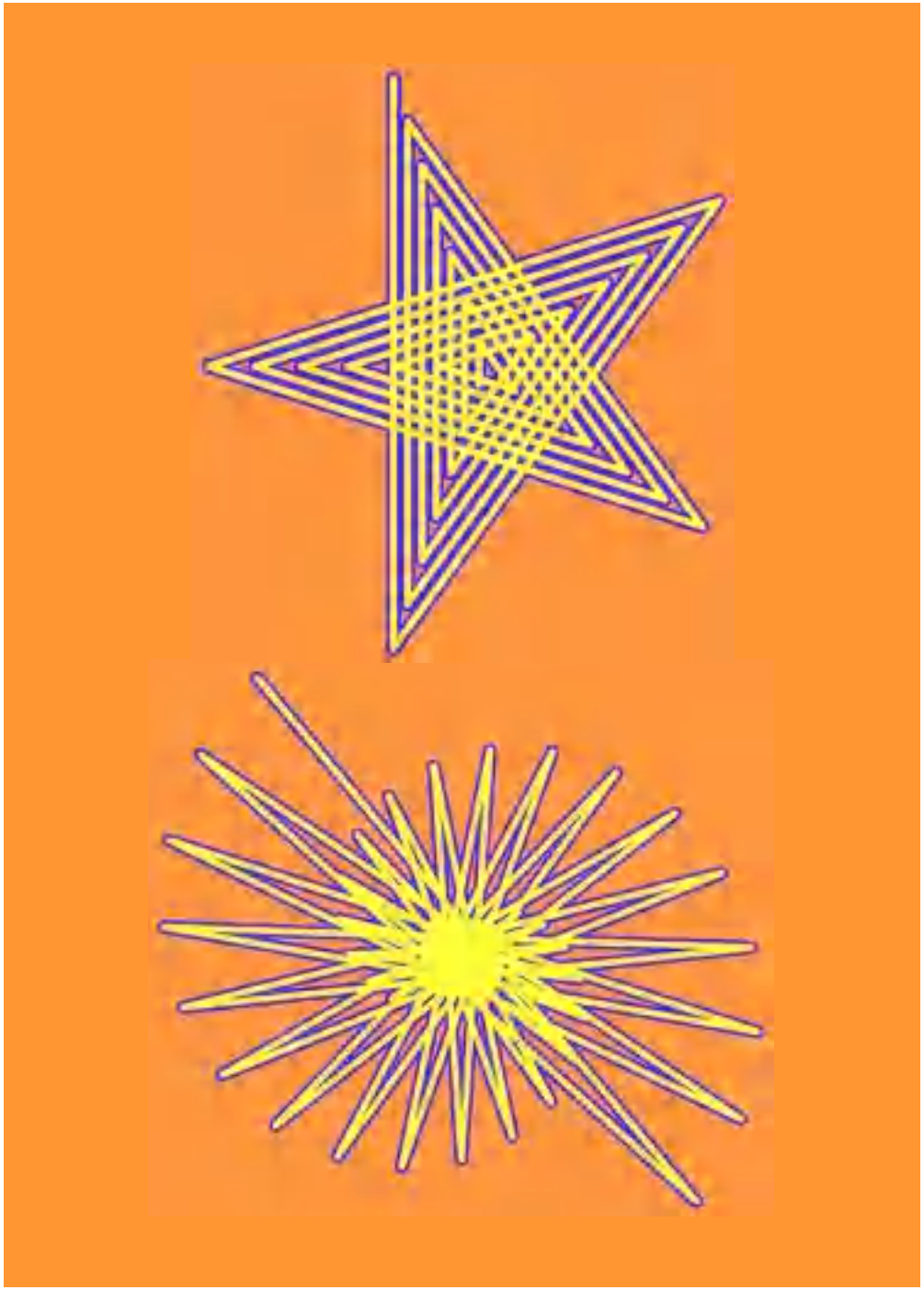
Similar to the procedure **polygon**, the turtle walks a distance and rotates by an angle after each step. However, in the new procedure **spiral**, the distance is increased by a factor **delta d** each time (upper right picture). So that this procedure does not continue endlessly, a **limit g** must be set for the distance. If the **condition l > g** returns the value **true**, this **limit** is exceeded and the procedure is aborted.

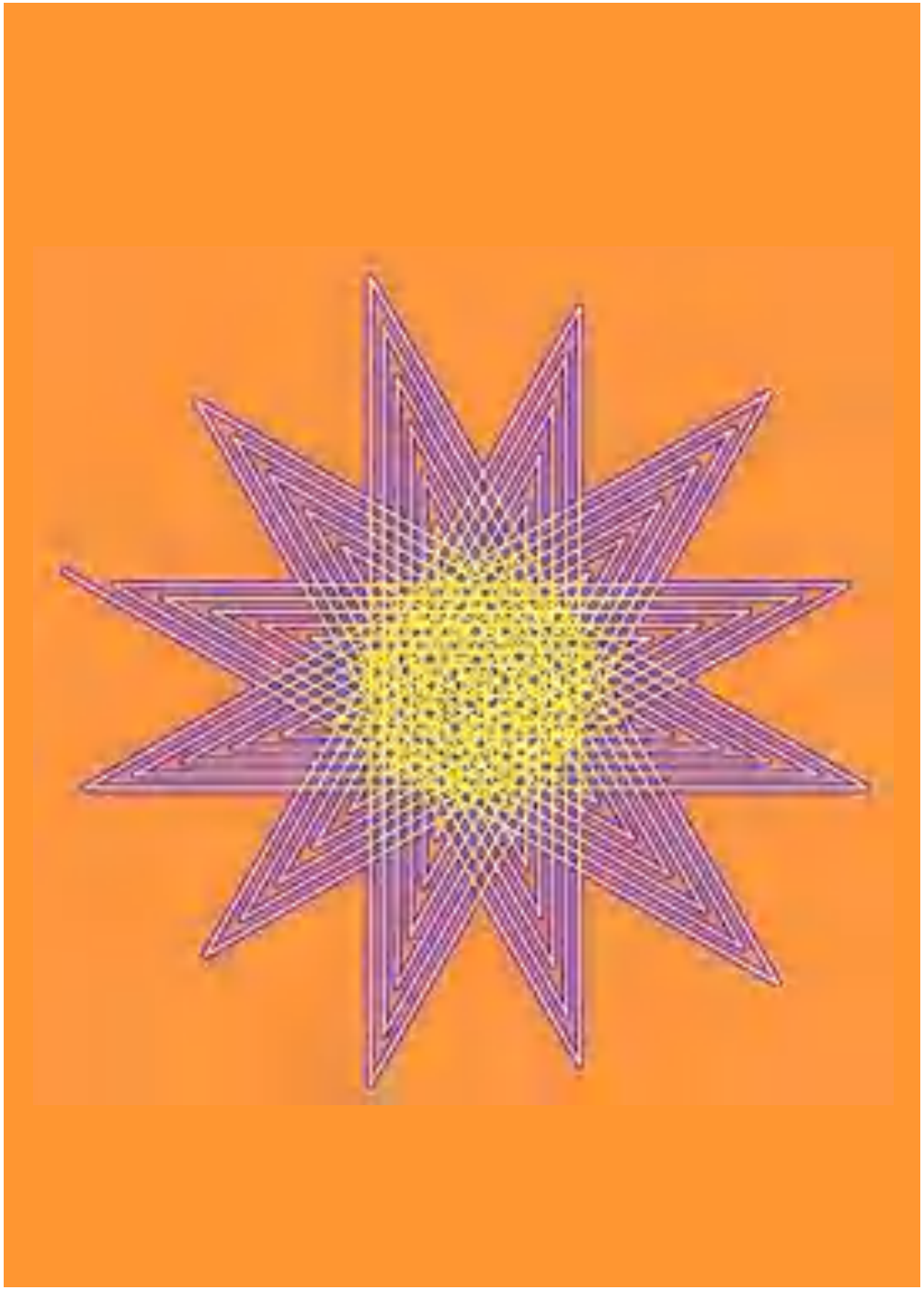


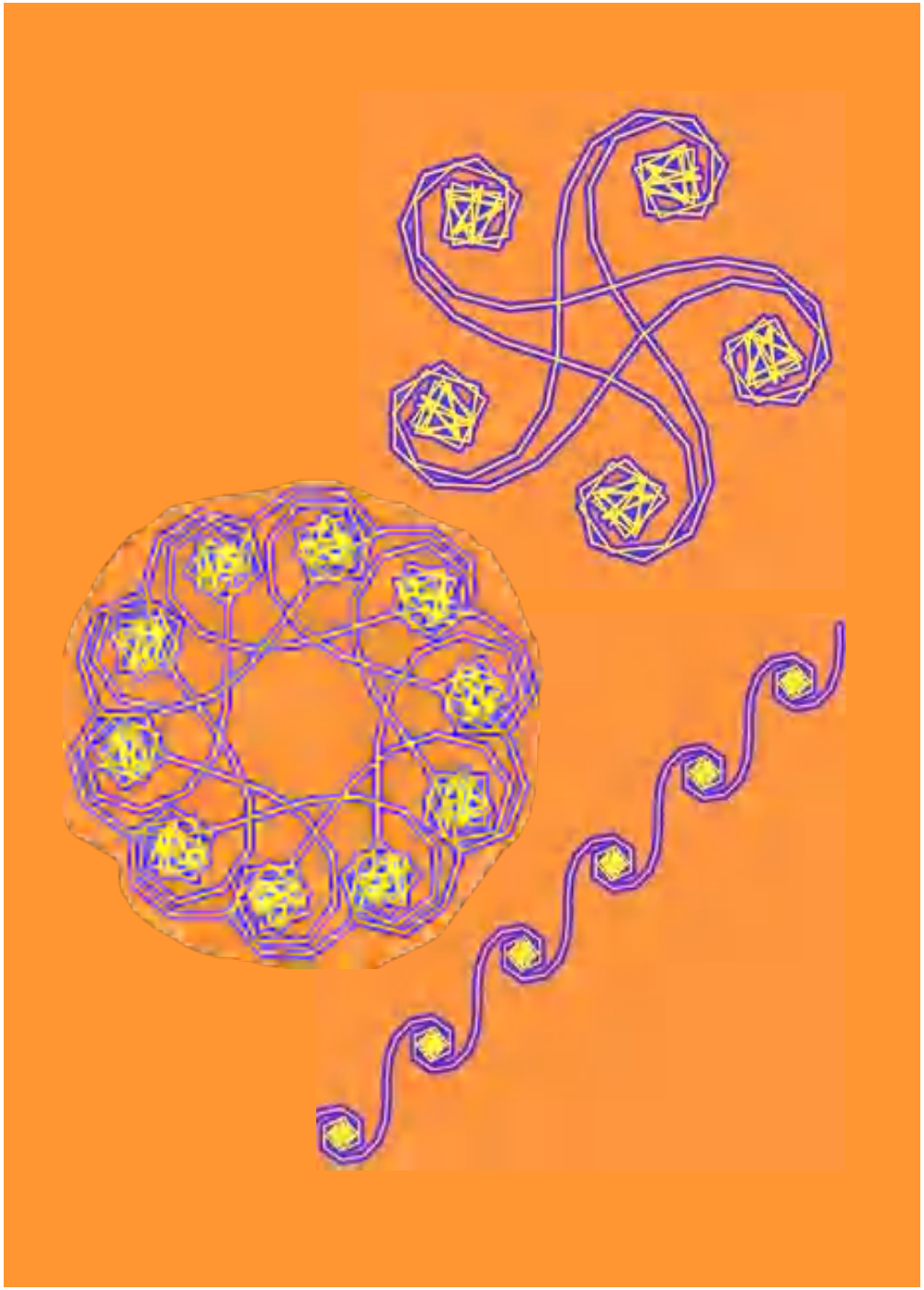
Instead of the distance, the angle of rotation can of course also be increased. To do this, change the command **change l by d** to **change w by d**.

The result is then a completely different one (picture below right). The curve is a clothoid (also known as Cornu-Spiral oder Euler-Spiral), where the curvature grows linearly with the path length.









Recursive spirals

Recursion as a form of **repetition** is often introduced with spirals. Recursive procedures can contain one or more **self-calls**. The principle is particularly powerful because (also modified) parameters can be passed in this call.

A **limit g** must be specified so that the calls do not continue indefinitely. If the corresponding condition returns the value **true**, this **limit** is exceeded and the calls therefor are ended. Depending on the location of the self call within the procedure, three types of recursion are distinguished:

initial recursion:

- termination condition
- **self-call** of the procedure
- instruction(s)

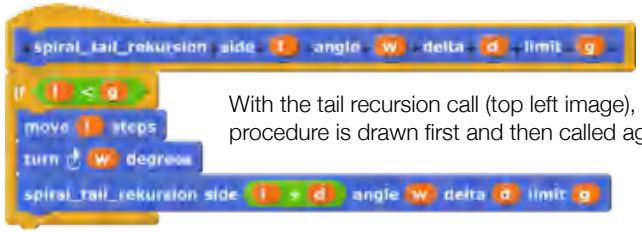
centric recursion

- termination condition
- instruction(s)
- **self-call** of the procedure
- instruction(s)

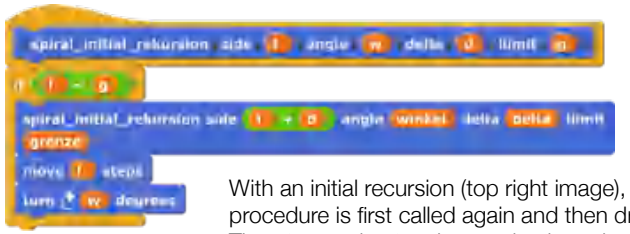
tail recursive:

- termination condition
- instruction(s)
- **self-call** of the procedure

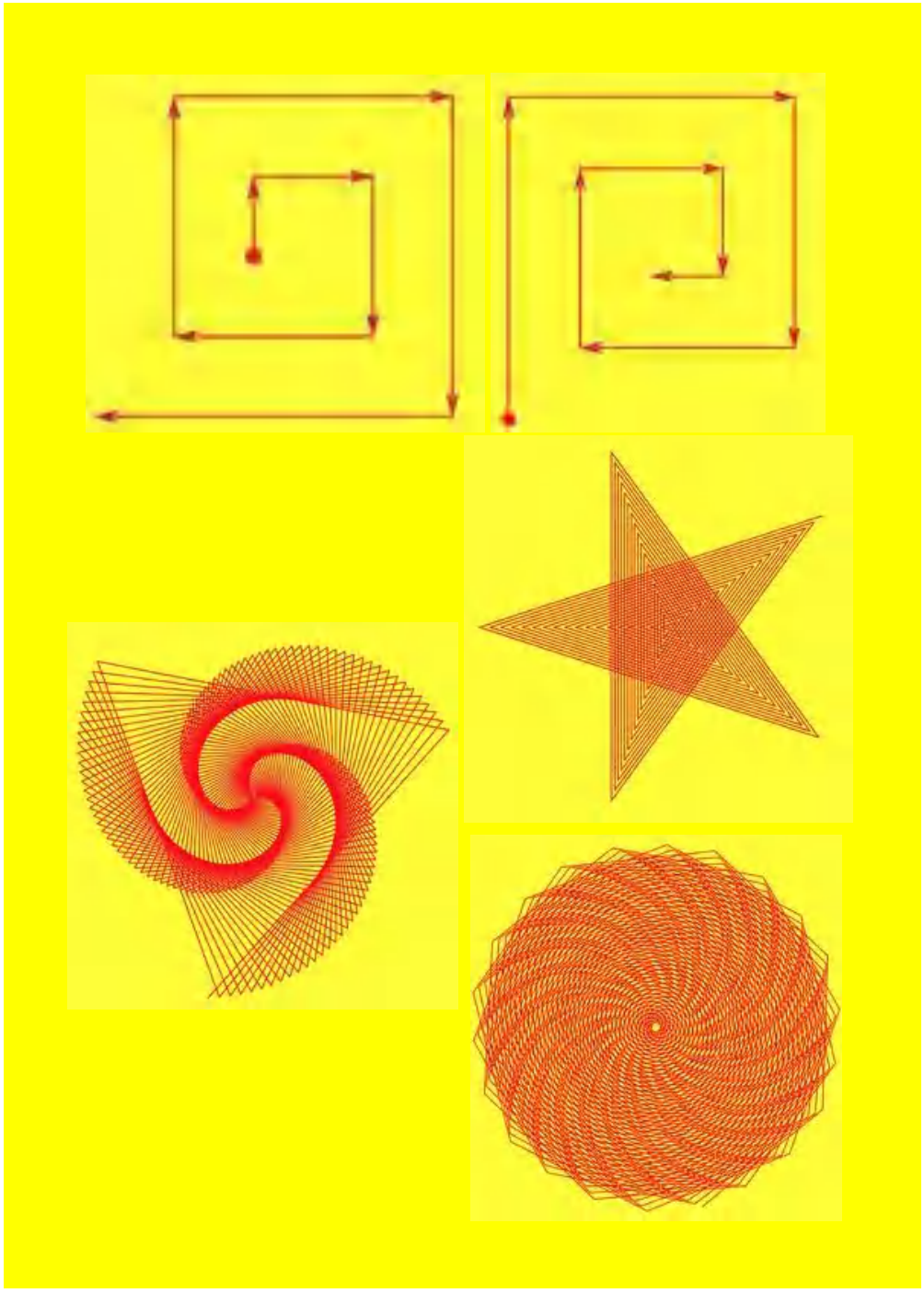
The process is easy to follow on the spirals.

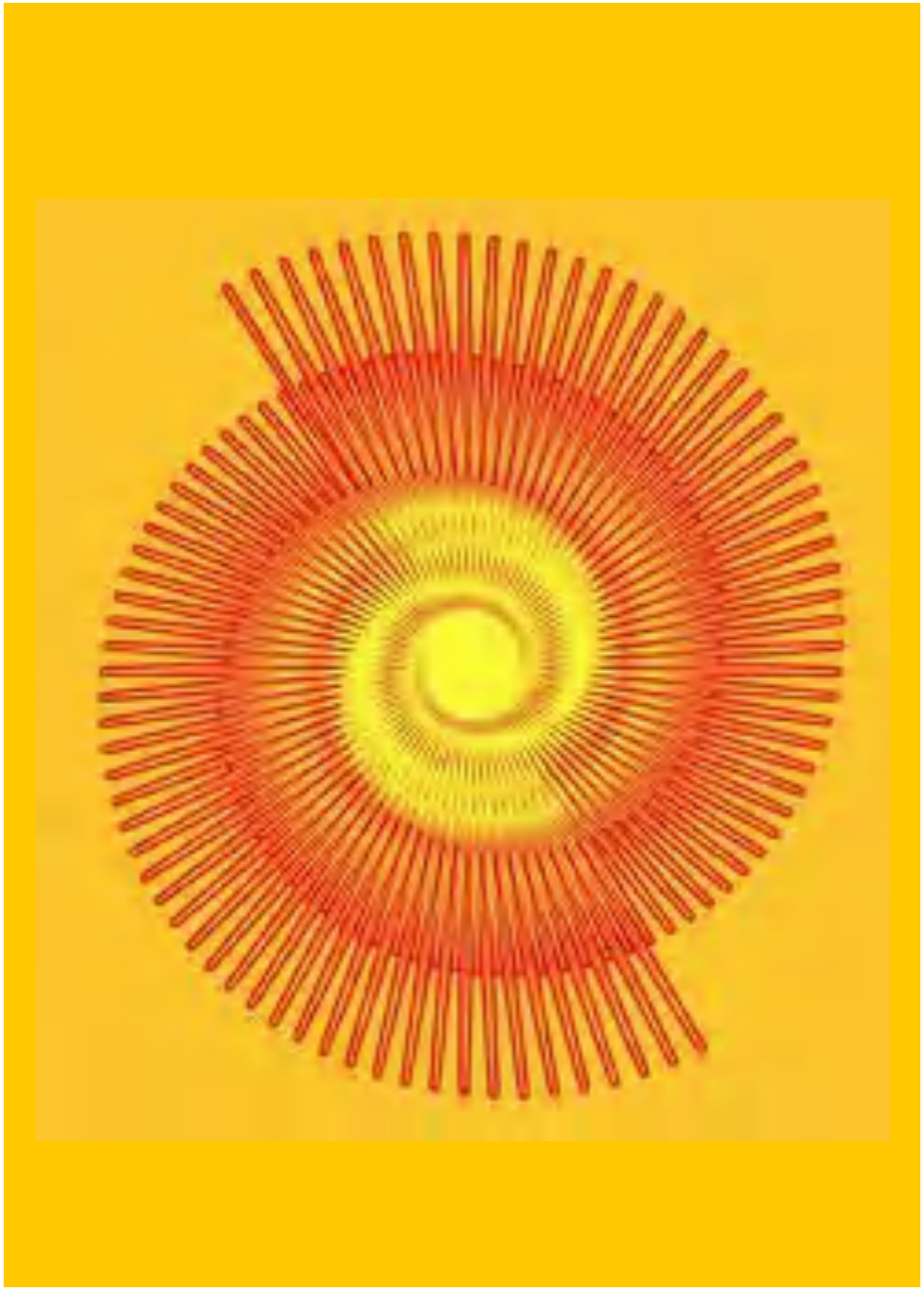


With the tail recursion call (top left image), the procedure is drawn first and then called again.



With an initial recursion (top right image), the procedure is first called again and then drawn. Therefore, only after the termination criterion has been met, the lines are drawn during the "way back" with the length accumulated up to that point.



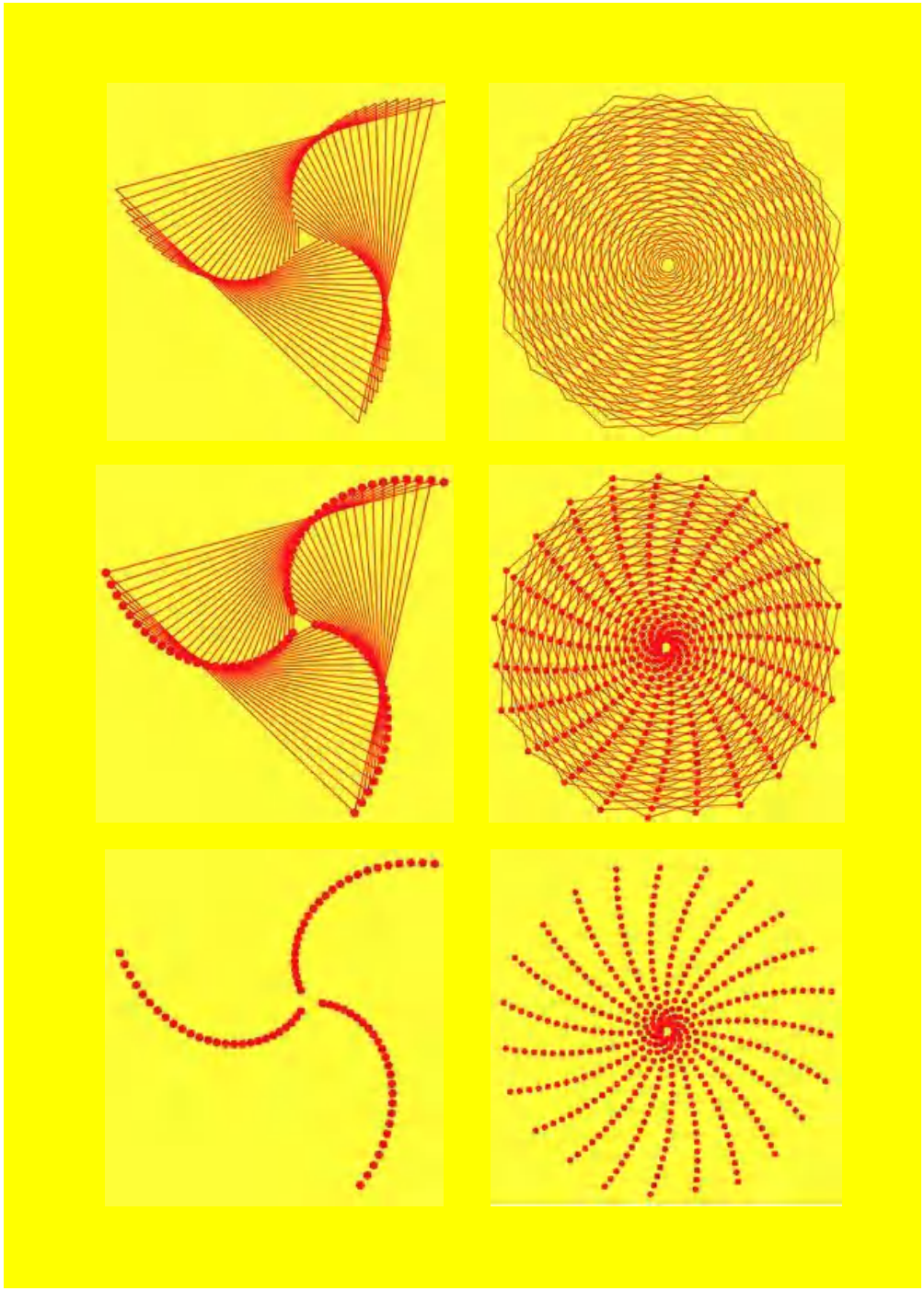


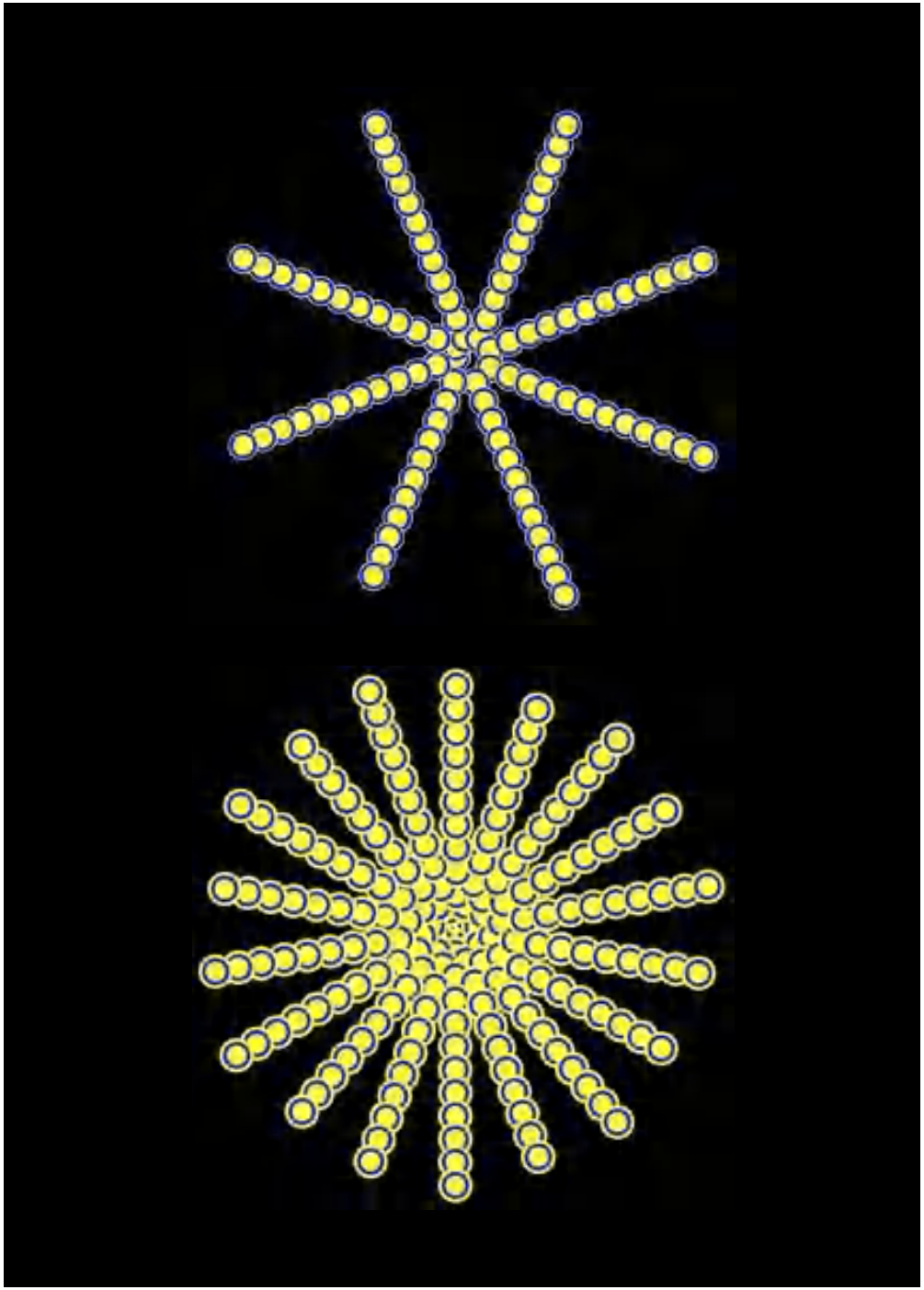
Dot spirals

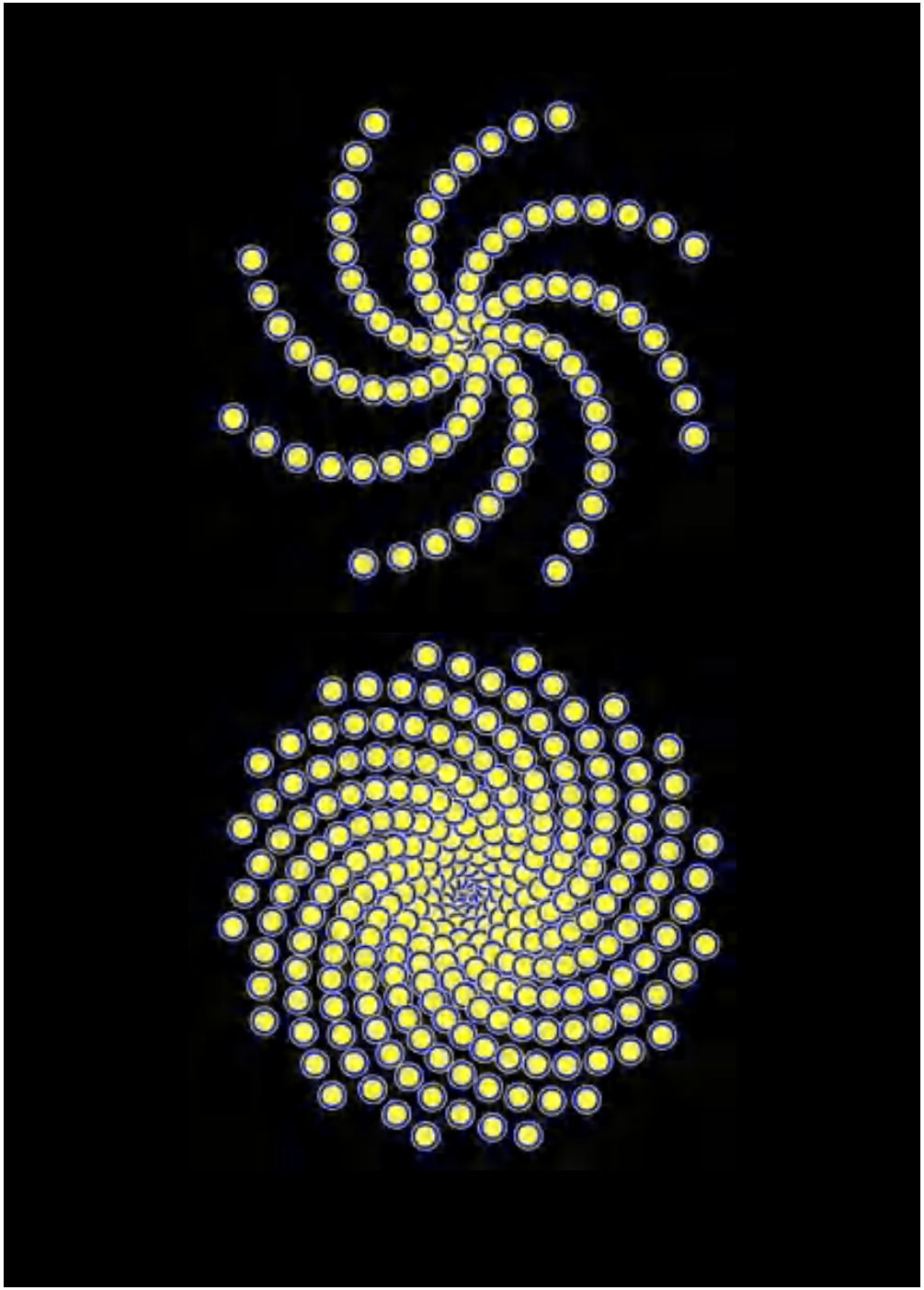
The basic structure of spirals often becomes much more visible if not only the lines between the spiral points are drawn, but their starting (and end) points.

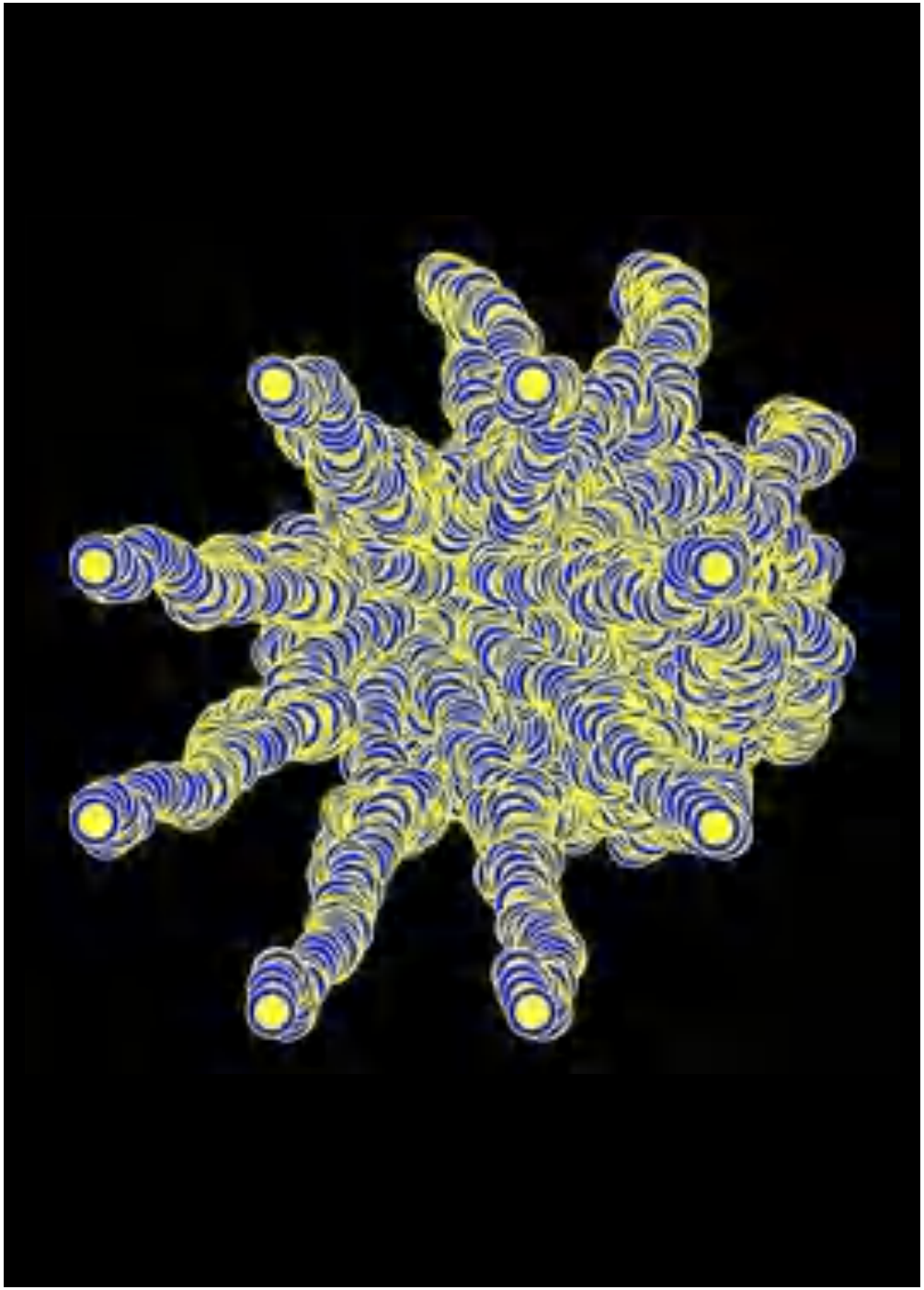
Dots of a certain pen thickness are drawn by not telling the turtle the number of steps for the **move steps** command and thereby "stepping on the spot" as it were. A yellow dot with a thickness of 60 then arises as follows:

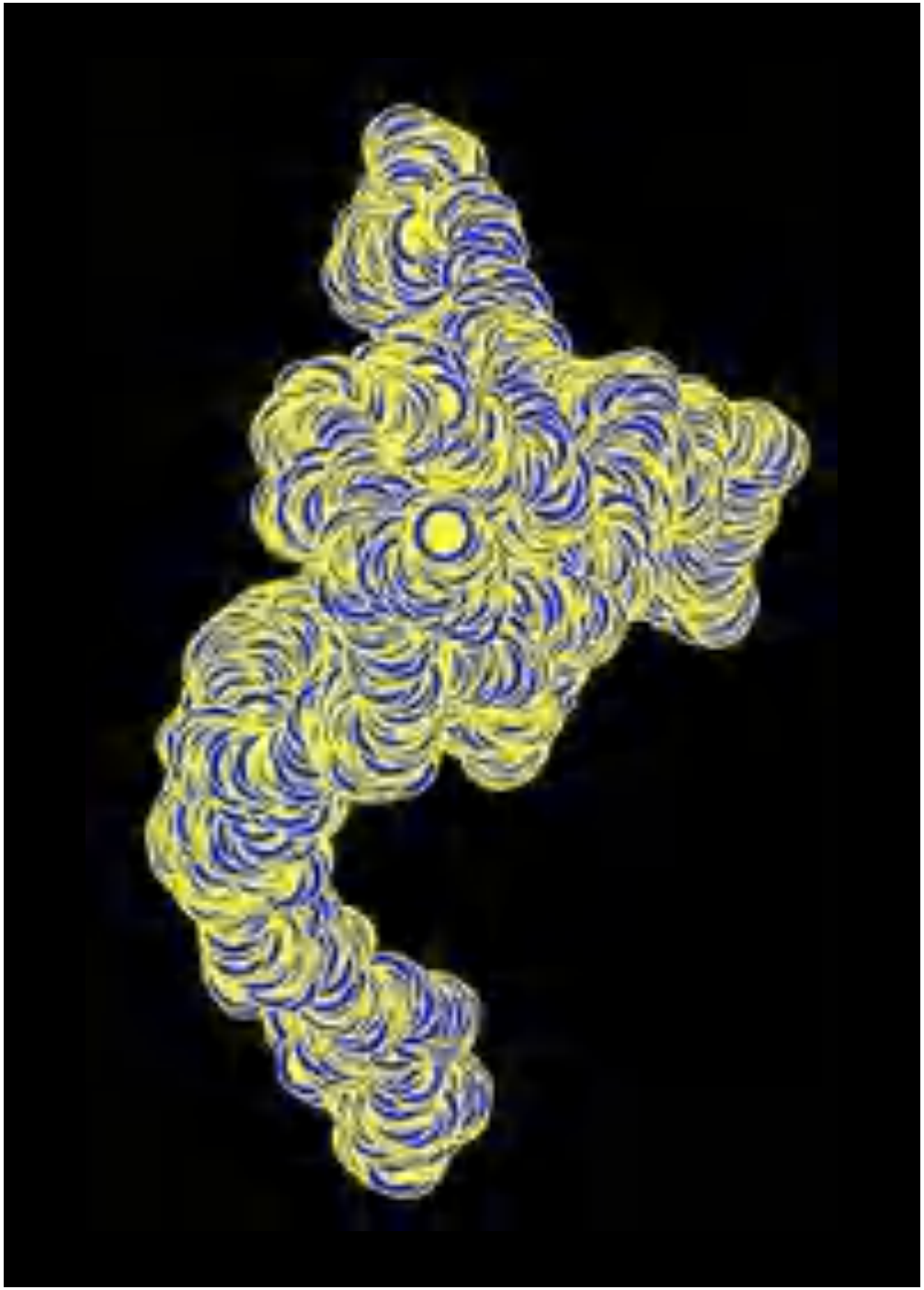
Small changes in the characteristics of the spirals, such as the **length**, the **angle** or the change **delta**, can cause large changes in the point structures. If this is combined with random changes, completely different structures are created again.











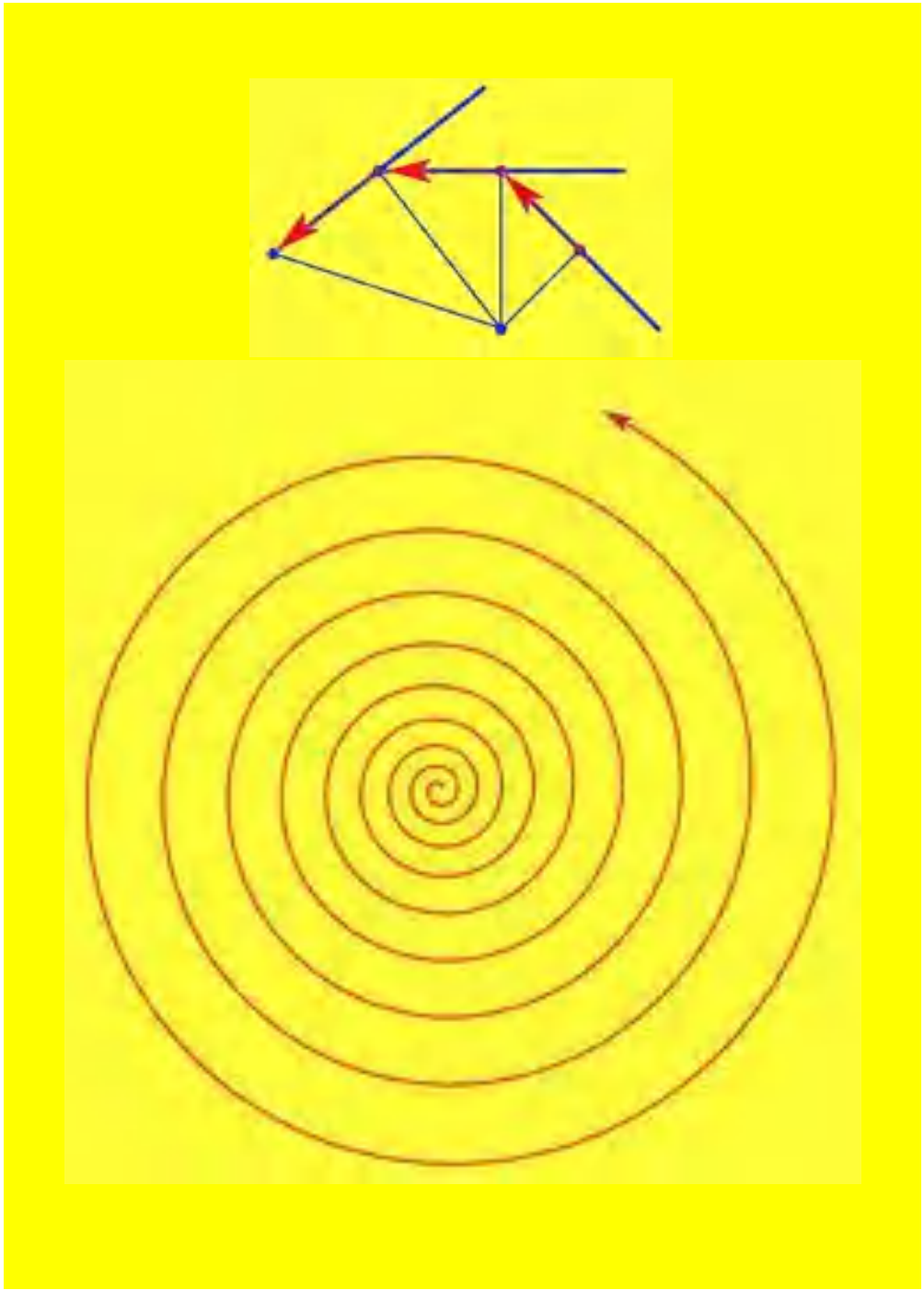
Arrow geometry

The turtle also has **sensors**. This allows different values to be measured and further processed:

- **distance to center/mouse-pointer/object** provides the distance (in pixels) from the current position of the turtle to the center of the image, to the mouse pointer or to another object to be specified.
- **direction to center/mouse-pointer/object** provides the direction (in degrees) from the current position of the turtle to the center of the image, to the mouse pointer or to another object to be specified.
- In the same way, further properties can be recognized and measured. For example, **hue, saturation ... at** delivers the color values below the mouse pointer or another object to be specified.

This can be used to create spirals in a completely different way; this is called **arrow geometry**.

Starting position is a point on a circle around a given center (**blue** in the picture on the top right). For this point, then the **tangent of the circle** can be drawn (perpendicular to the distance line to the centre). On this tangent the turtle is moved by **length**. At its new position, the tangent is drawn again and the turtle is moved by **length + delta**.



Recursive Trees

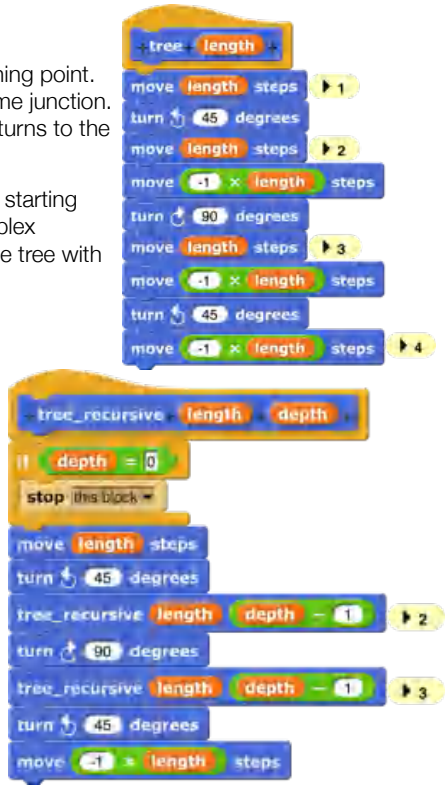
Drawing trees is a popular example of the use of **recursion** in turtle graphics. One reason for this is that when drawing a tree, there are always recurring sequences:

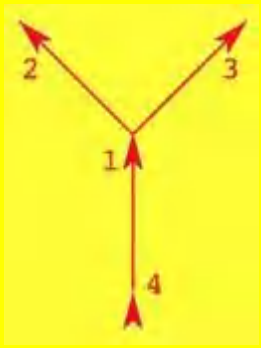
1. First the trunk is drawn.
2. A left branch is drawn at a branching point.
3. A right branch is drawn at the same junction.
4. Important: at the end the turtle returns to the starting point.

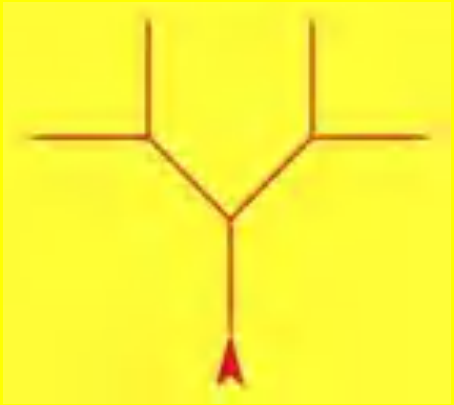
If the branches are each used as the starting point for further ramifications, a complex branching structure is created, i.e. the tree with branches.

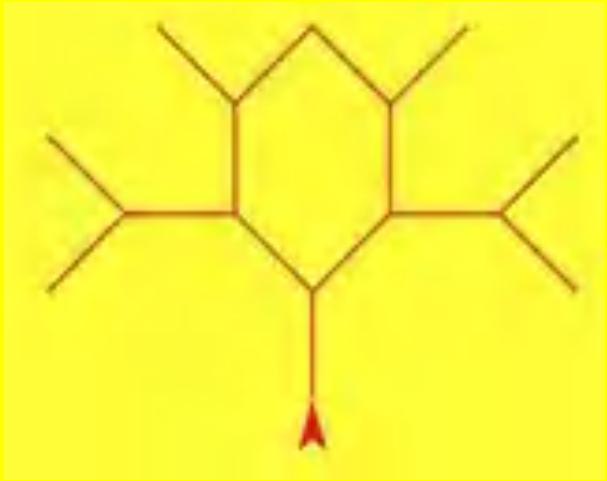
In the recursive version, therefore, **move length steps** when drawing the branches is replaced by the recursive call **tree_recursive length depth**. In the figure on the right the results for a depth of 1 to 4 are shown. If **depth = 0** returns the value **true**, drawing is aborted.

Also the **length** and the **angle** at which the branches branch off can be changed depending on the **depth**. Combined with line thickness and colour, the results then resemble very real trees of certain species. Other values result in abstract figures with area-filling patterns.

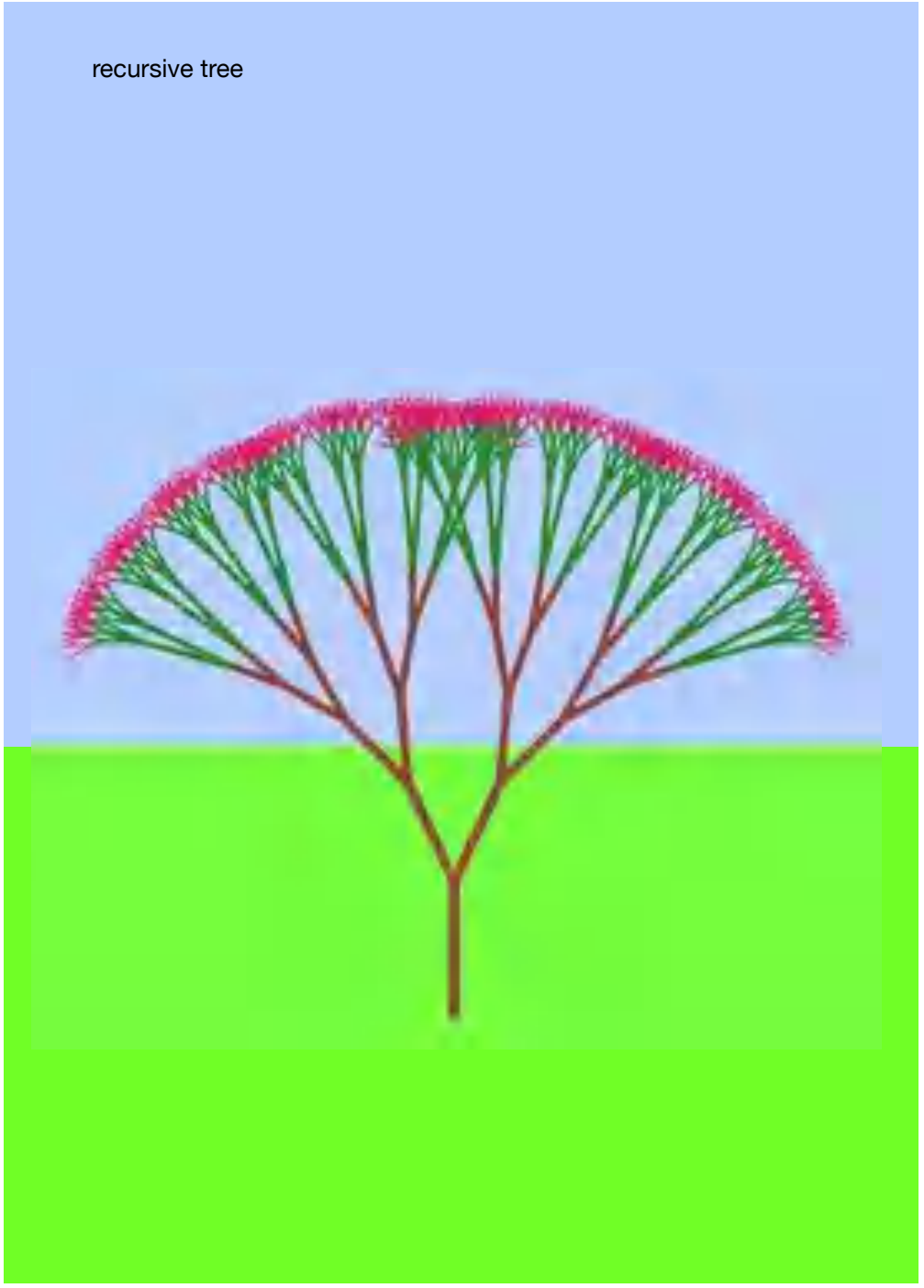




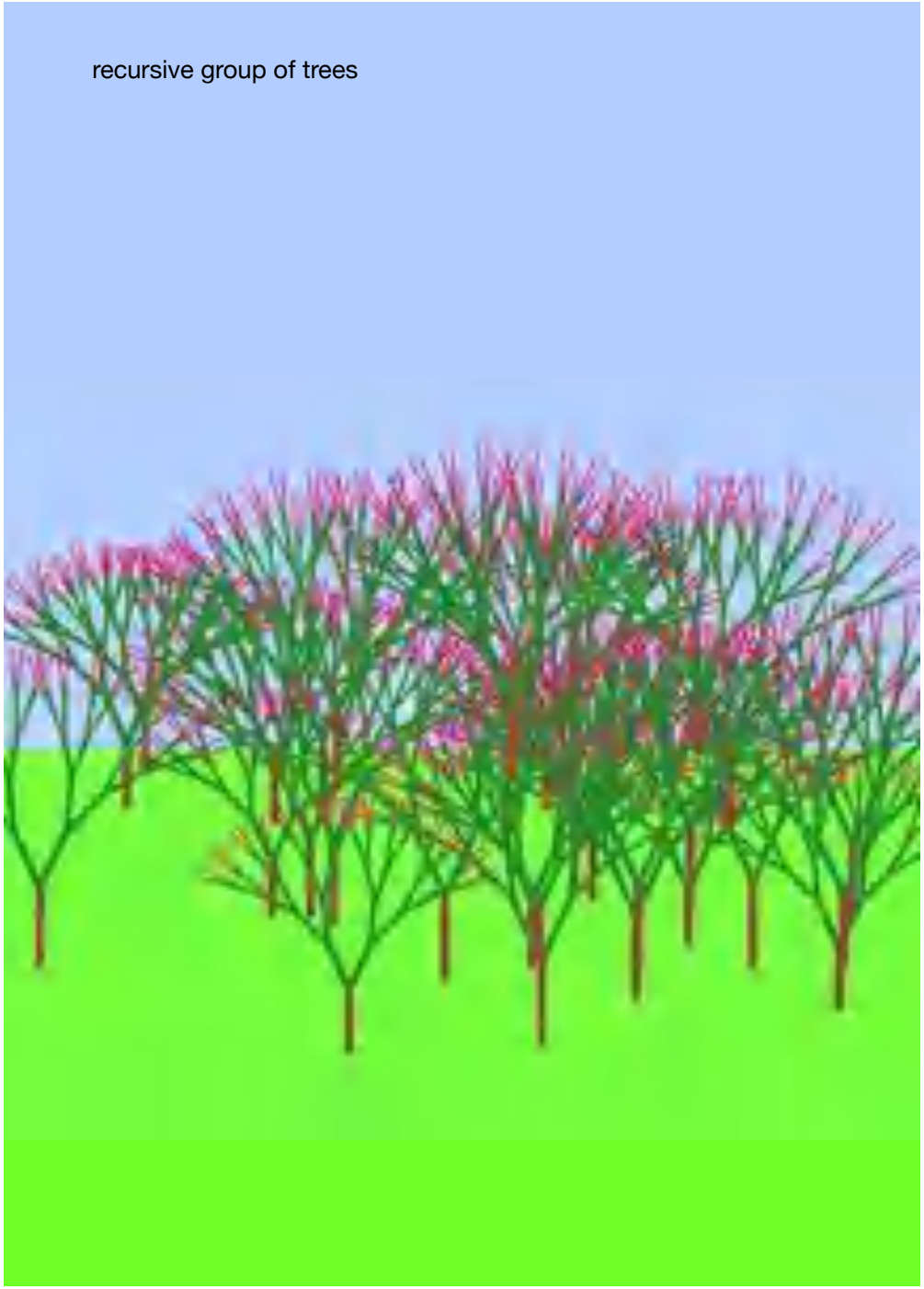




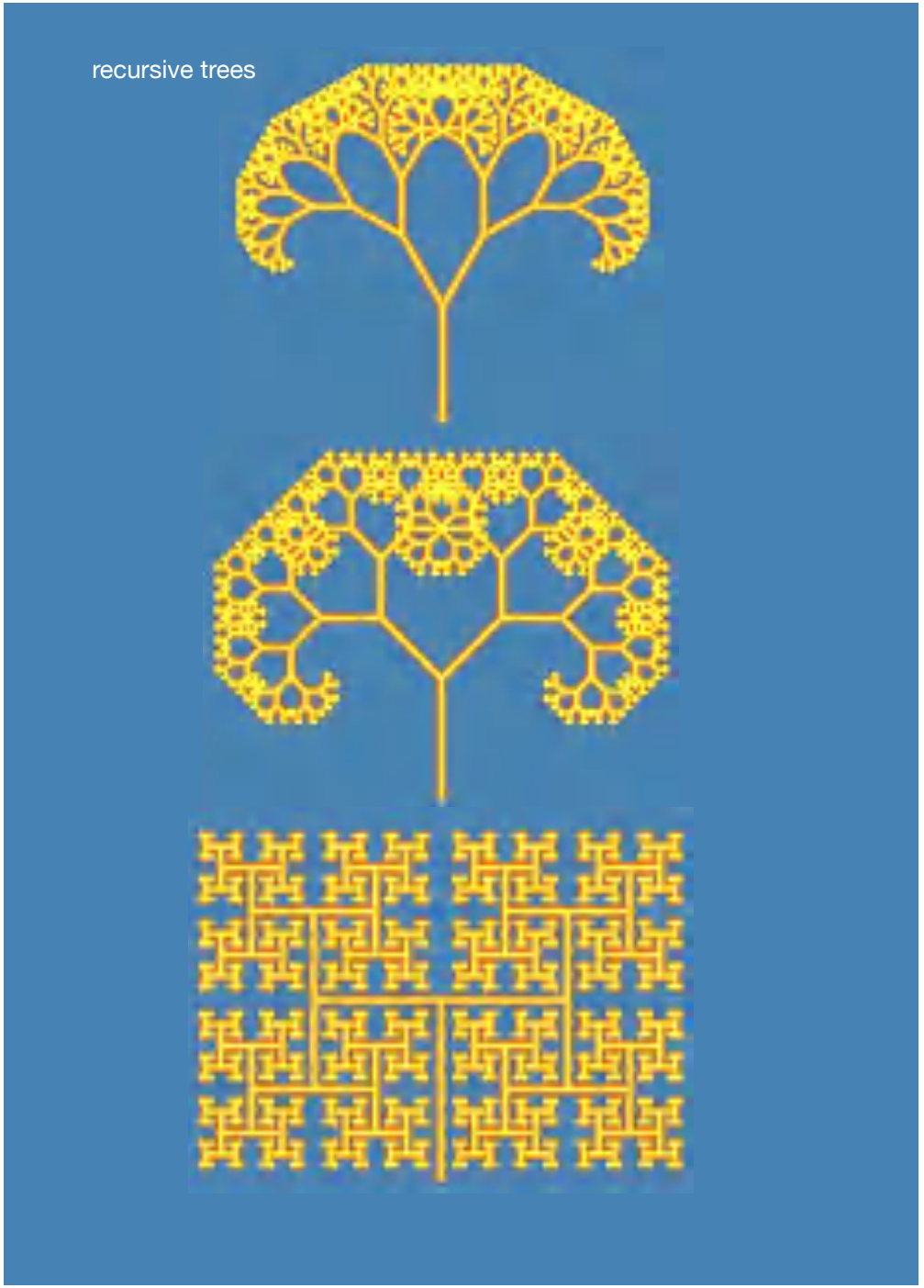
recursive tree

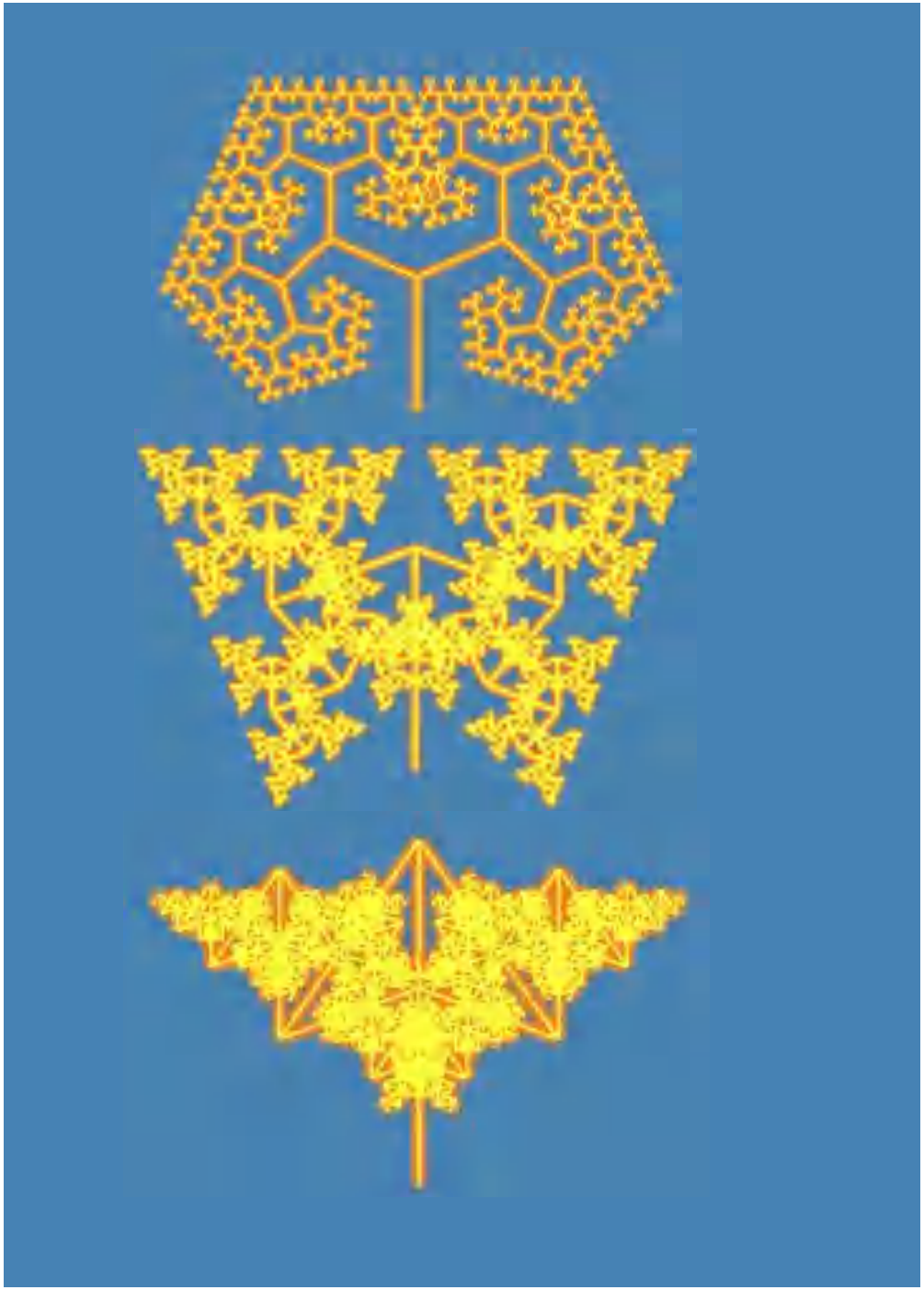


recursive group of trees



recursive trees



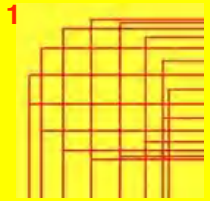


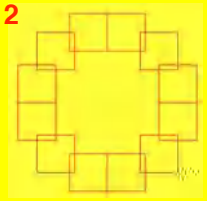
Combination of known things

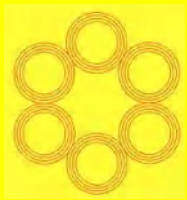
With the programs and procedures shown so far, points, lines, polygons, circles, arcs, spirals, and recursive trees could be drawn. Thus, a powerful **figure construction kit** is already available. In a further step these elements can be combined with each other[5]. This results in new structures again:

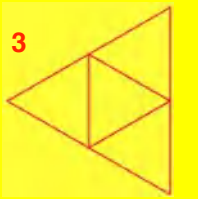
1. lines on circles
2. polygons on circles
3. polygon combinations
4. circle combinations
5. recursive triangles, squares, polygons
6. recursive polygon combinations
7. recursive circles

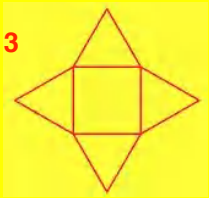
[5] So since known elements are used, I will not print snippets of code in this section.

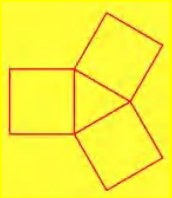


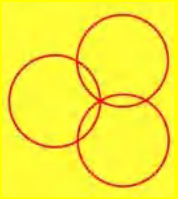


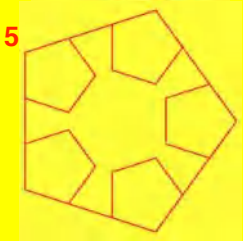


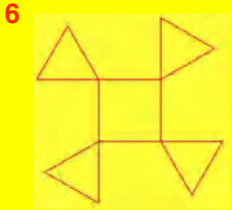


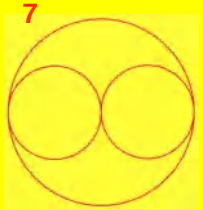


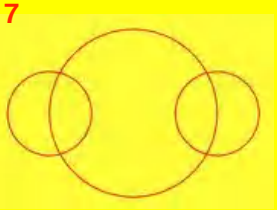




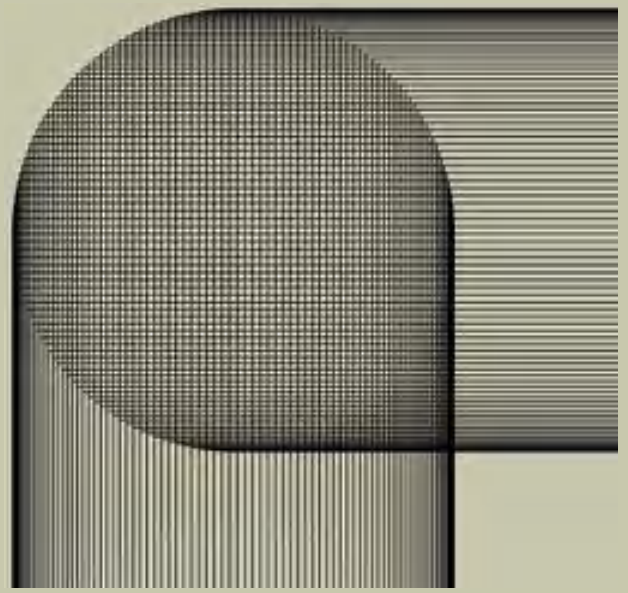


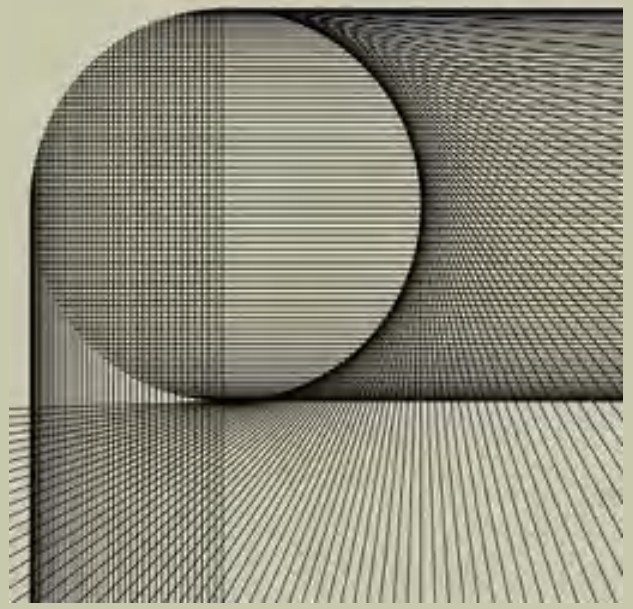


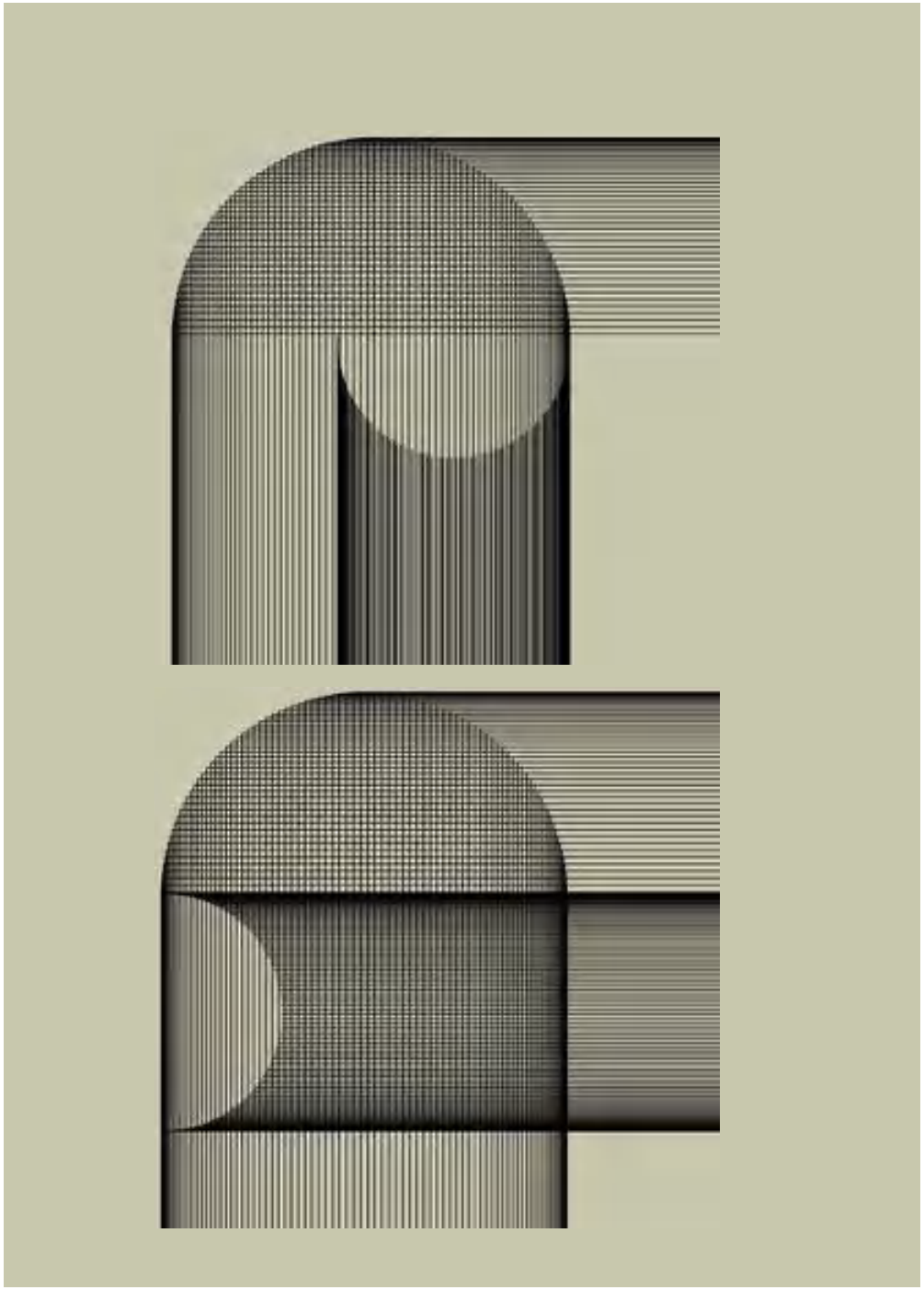


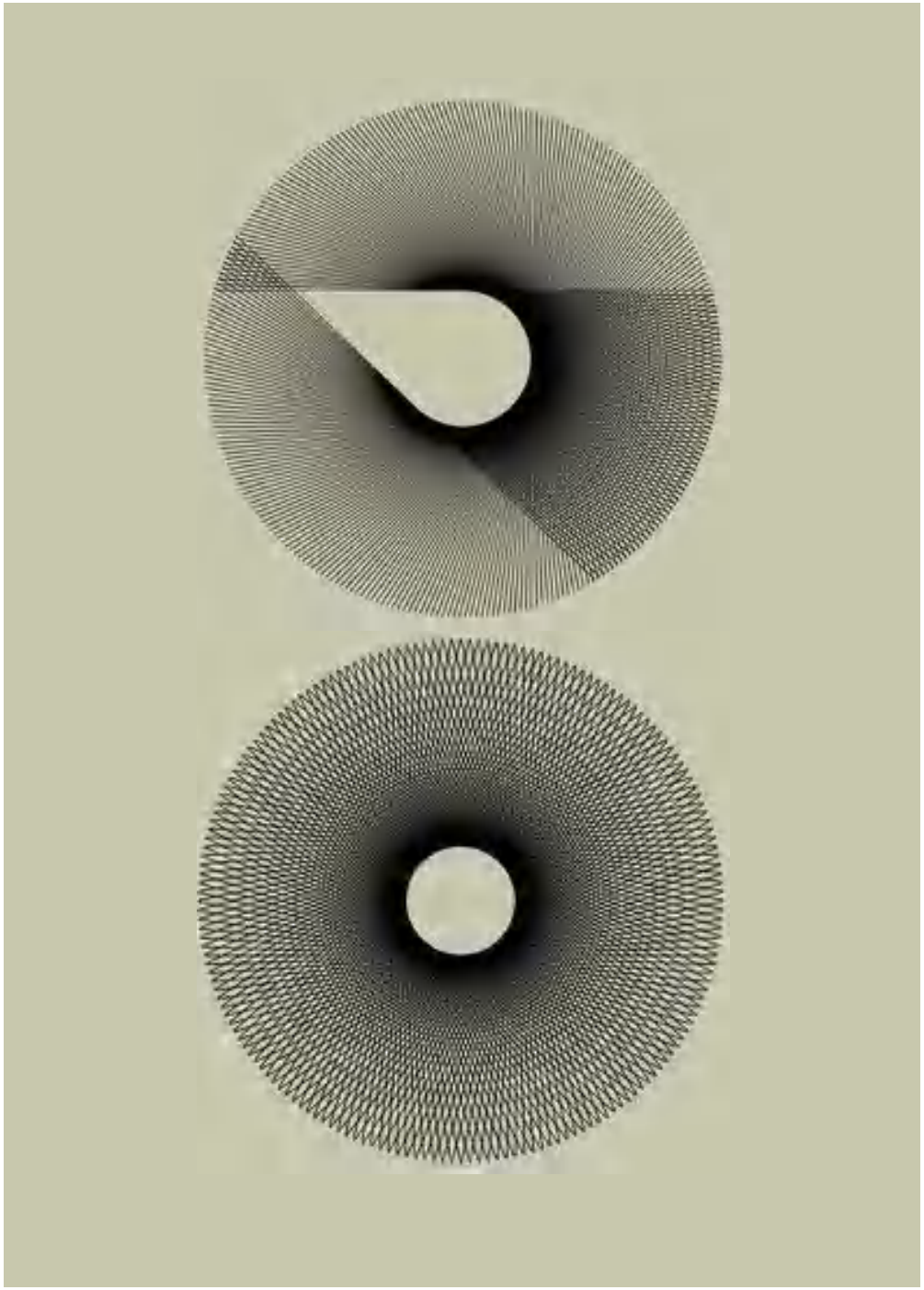


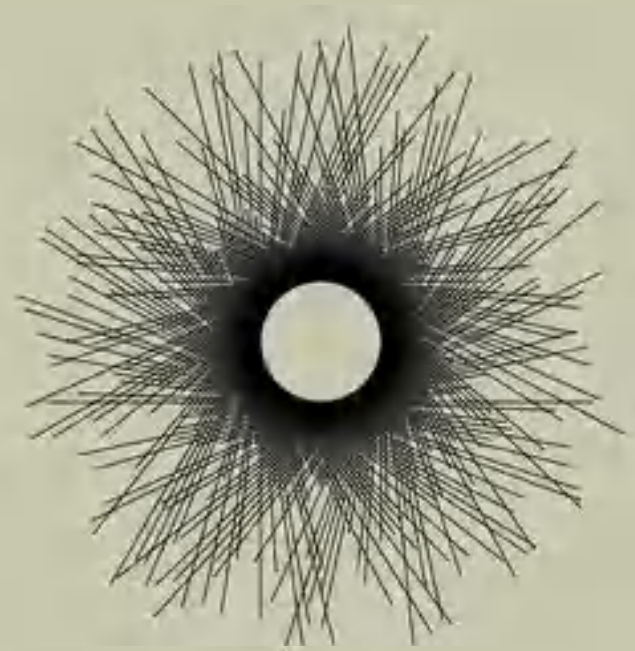
lines on circles

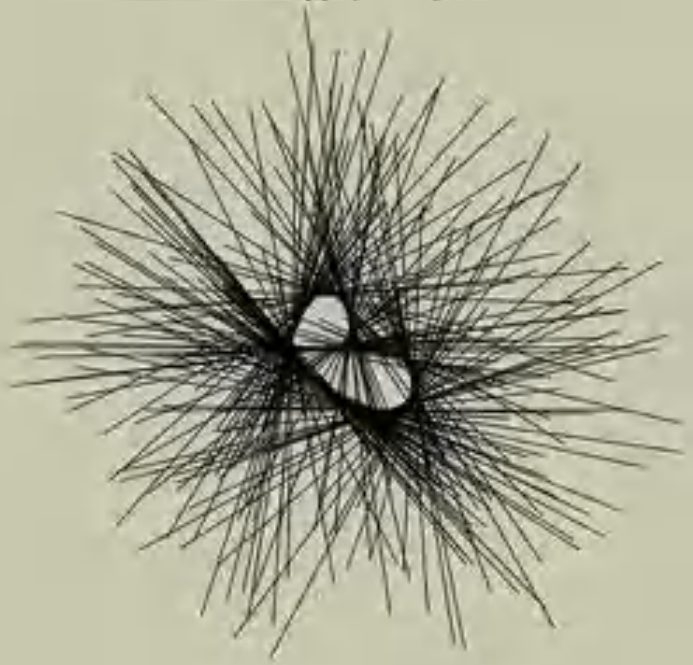




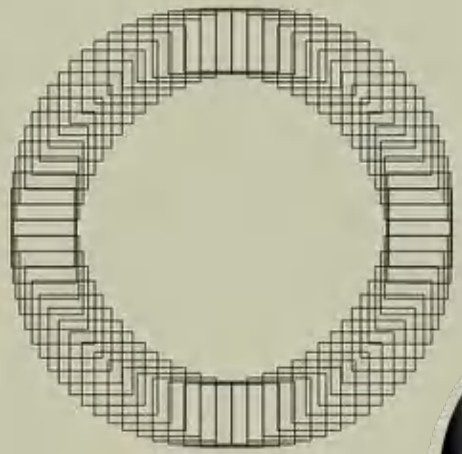


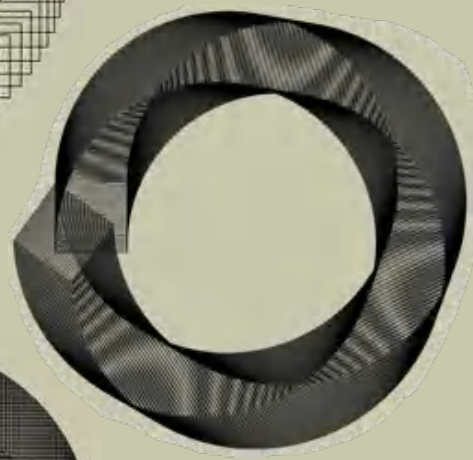


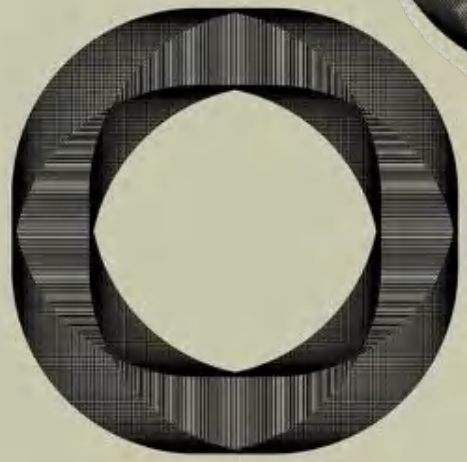


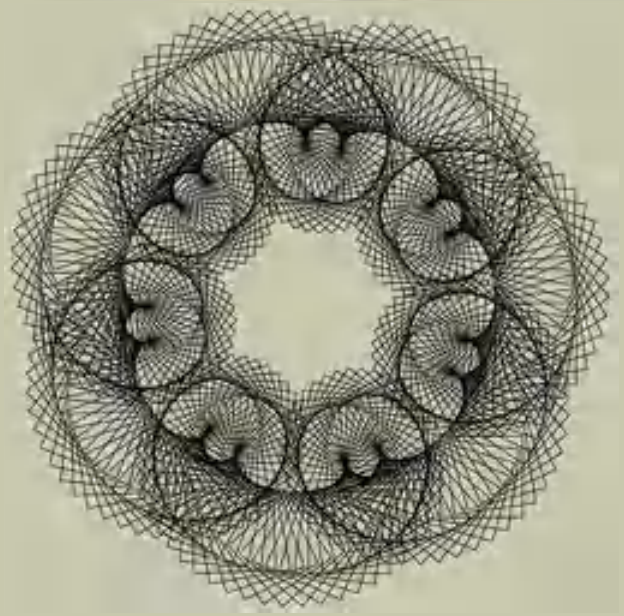


squares on a circle

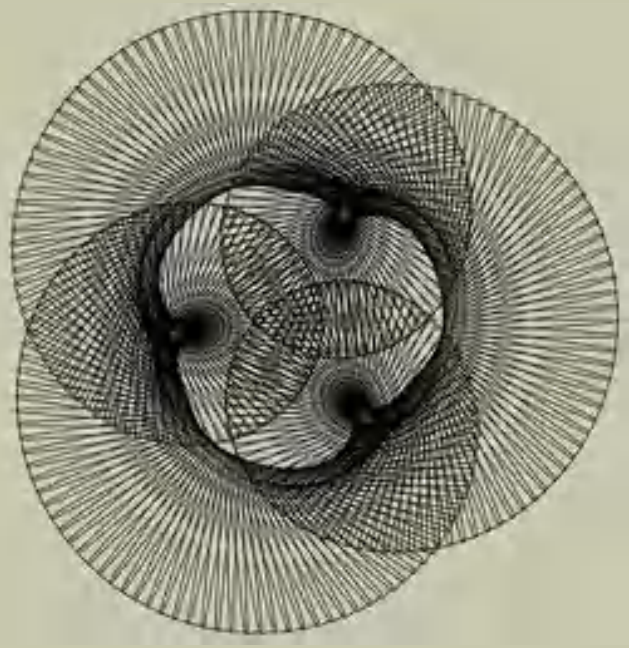




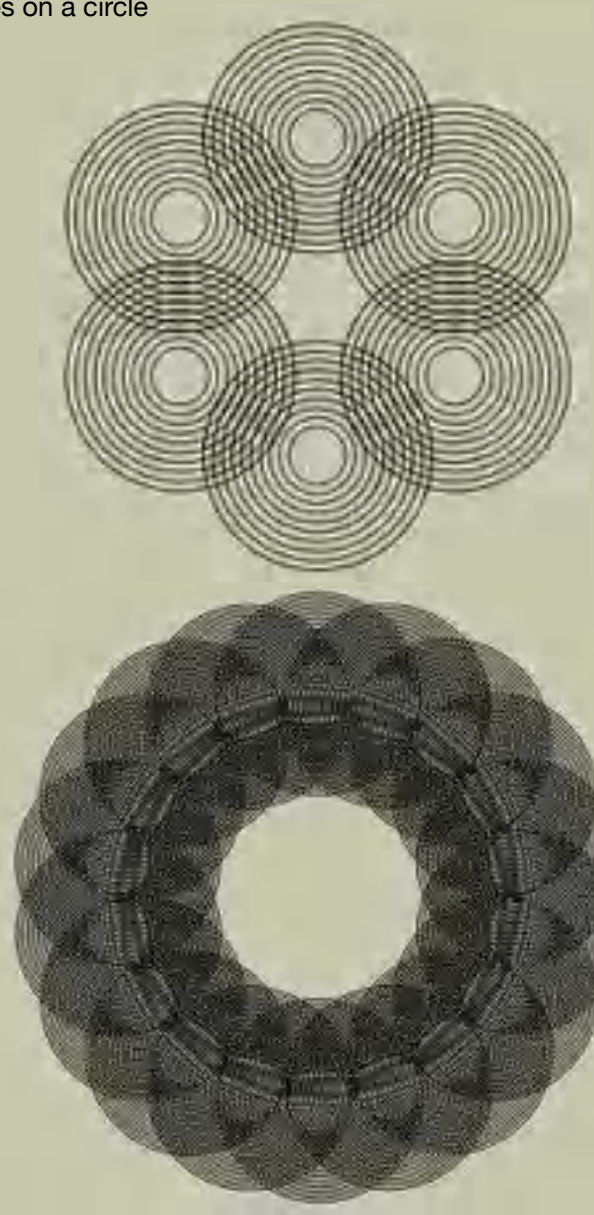




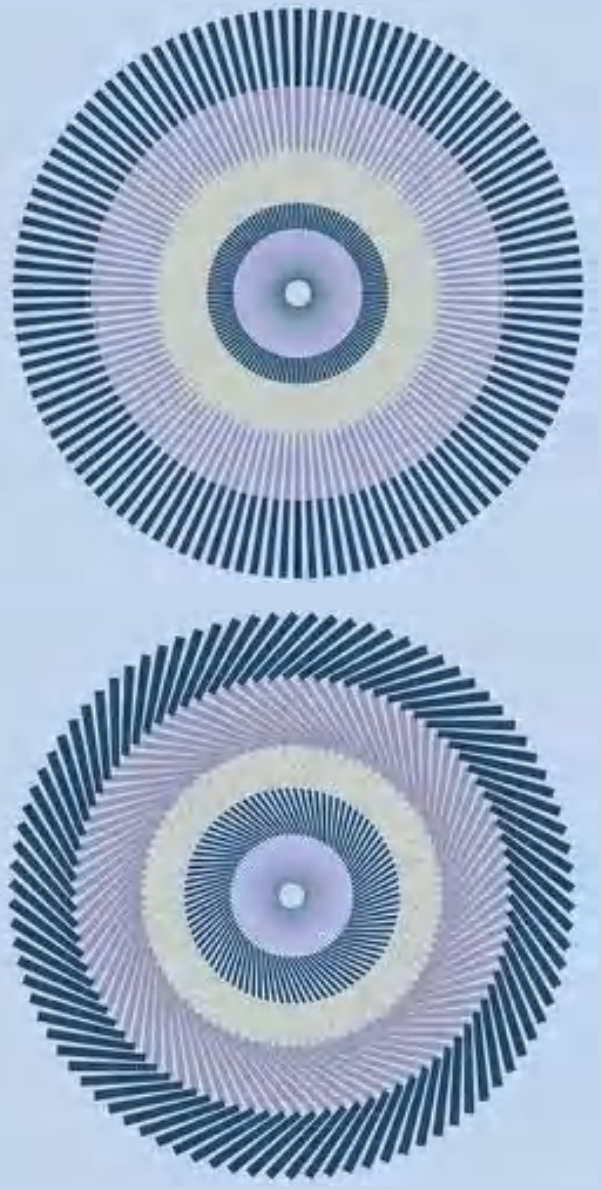
rectangles on a circle

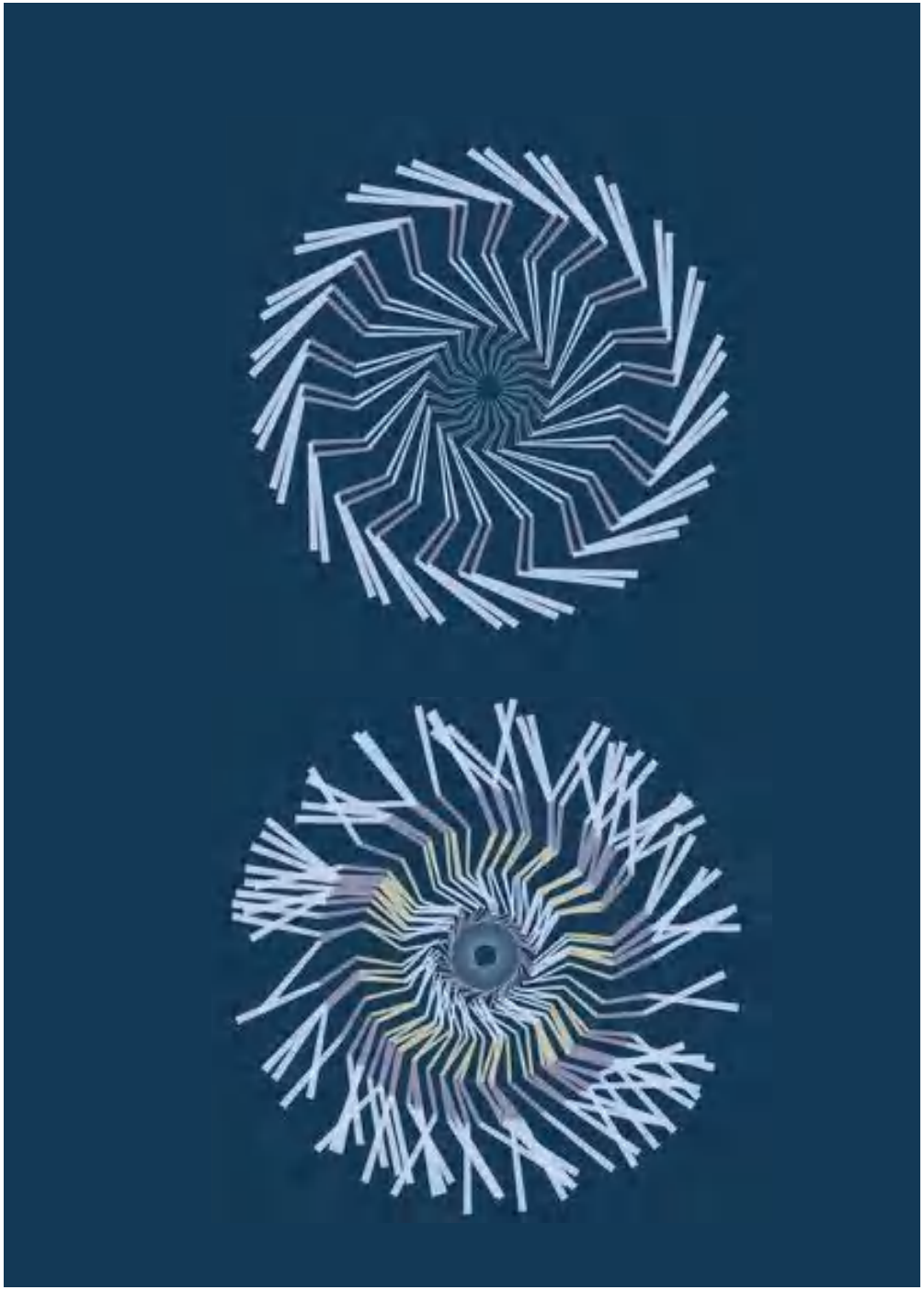


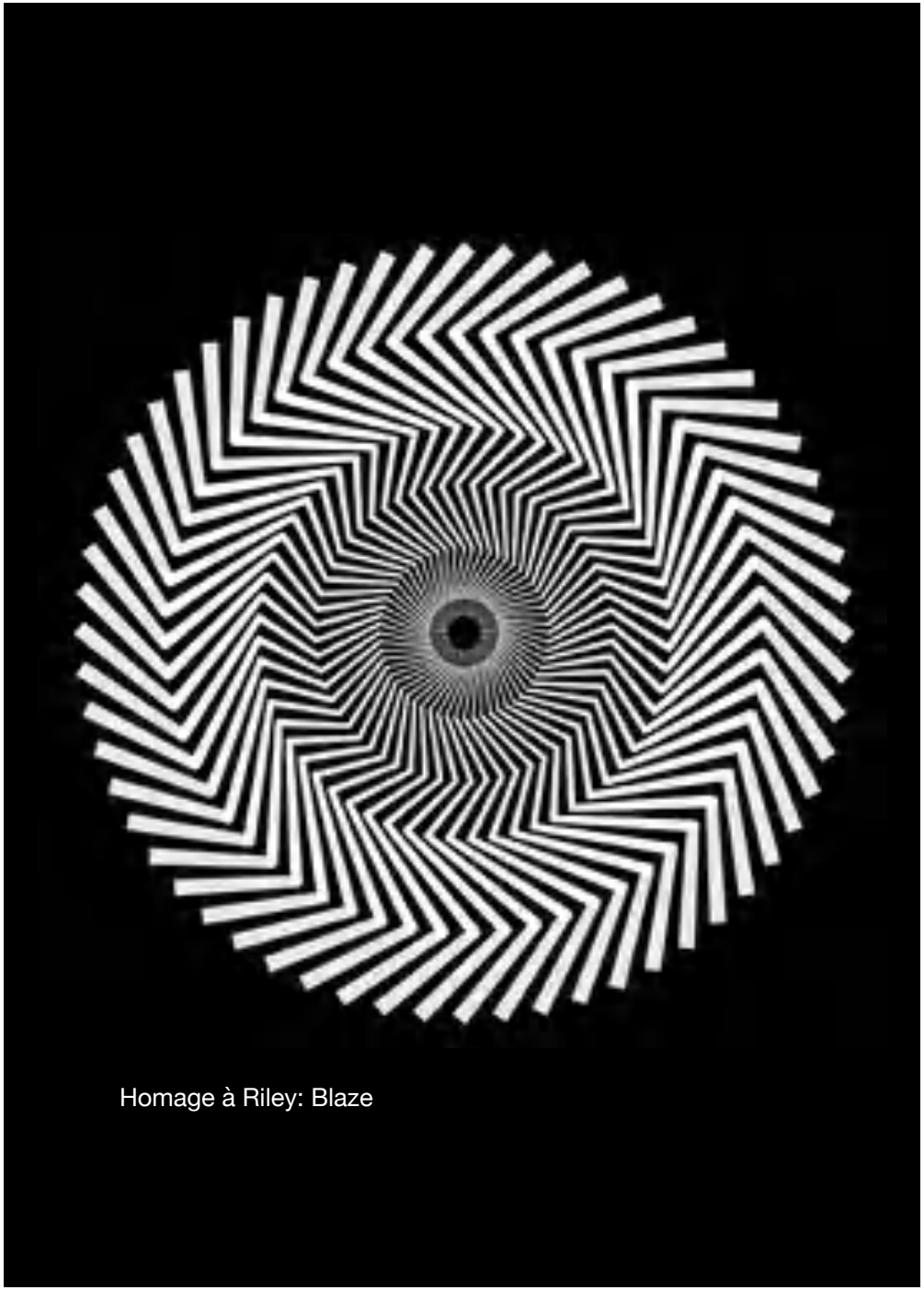
circles on a circle



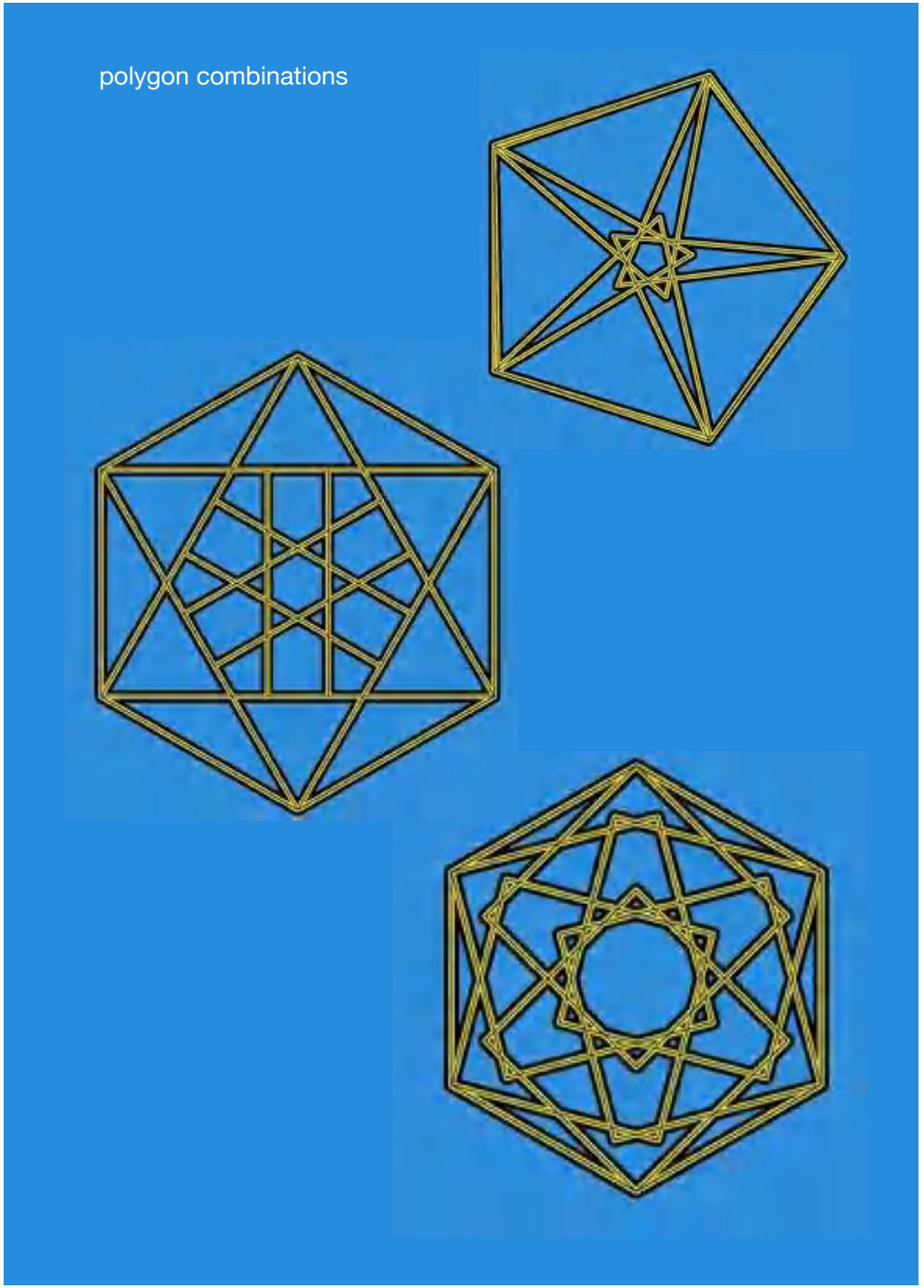
lines on circles

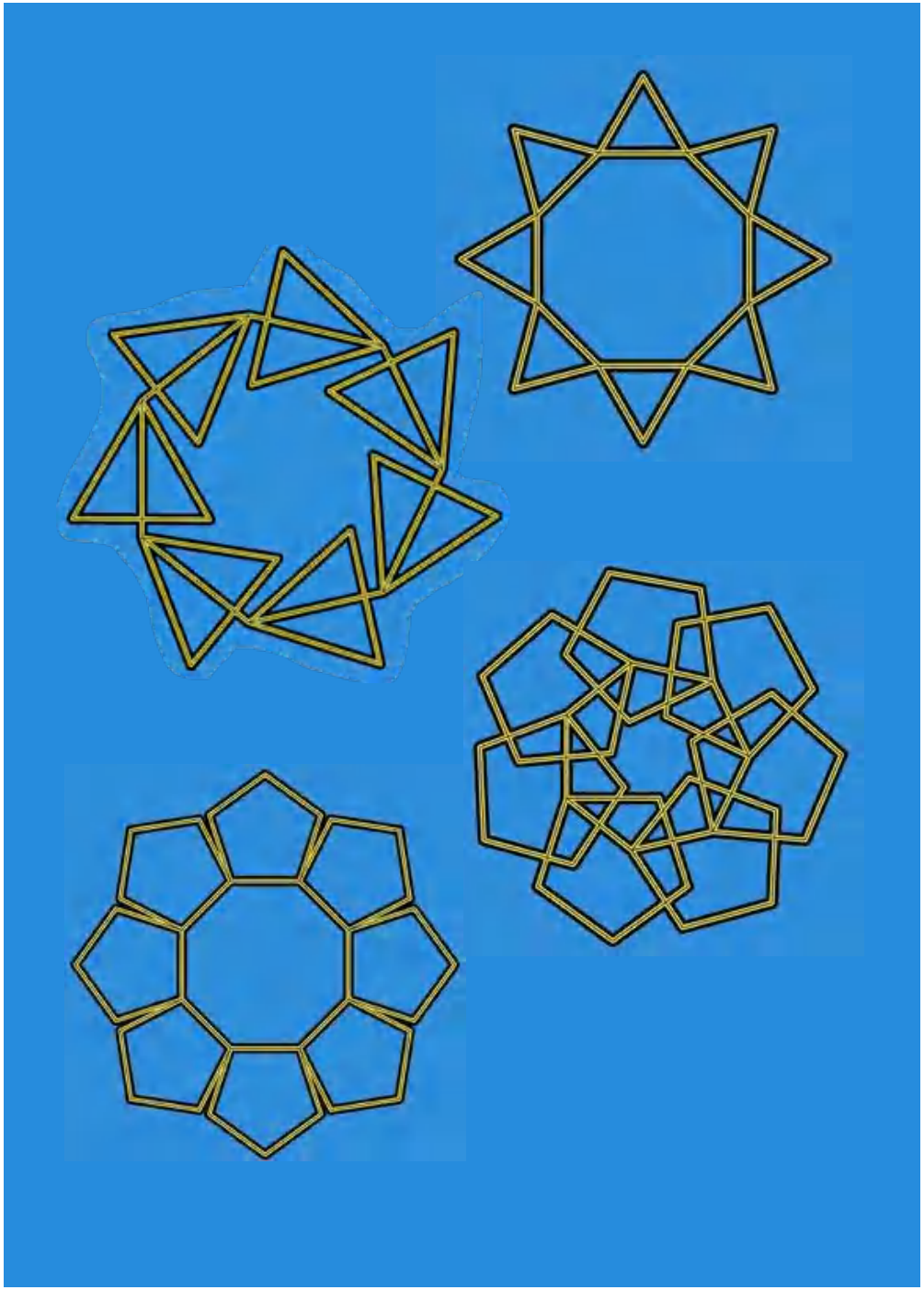




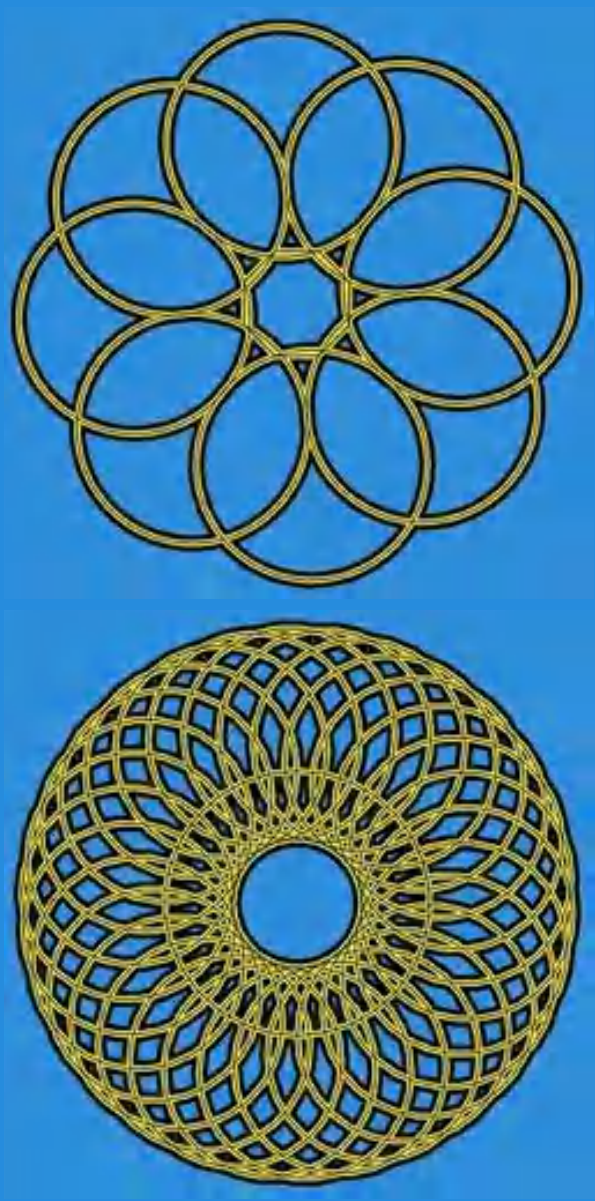


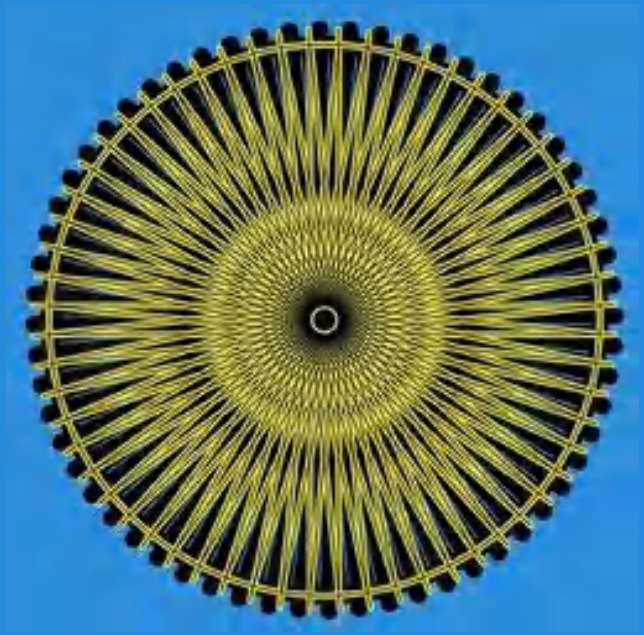
Homage à Riley: Blaze


polygon combinations

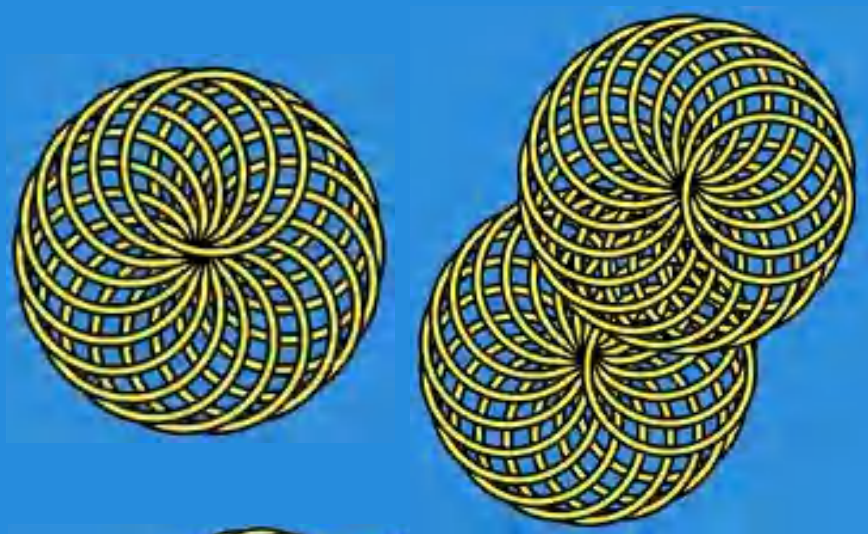


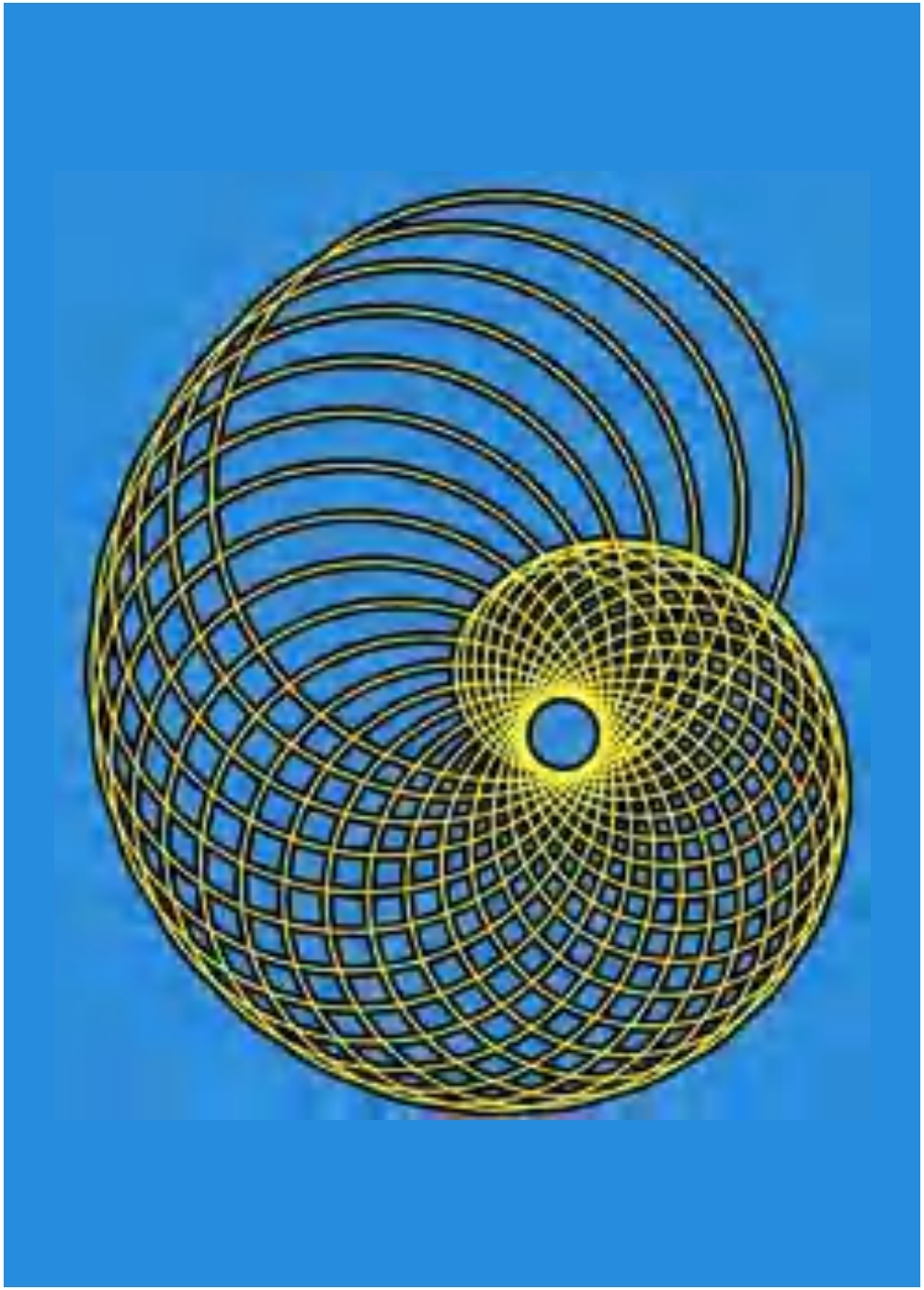
polygon combinations



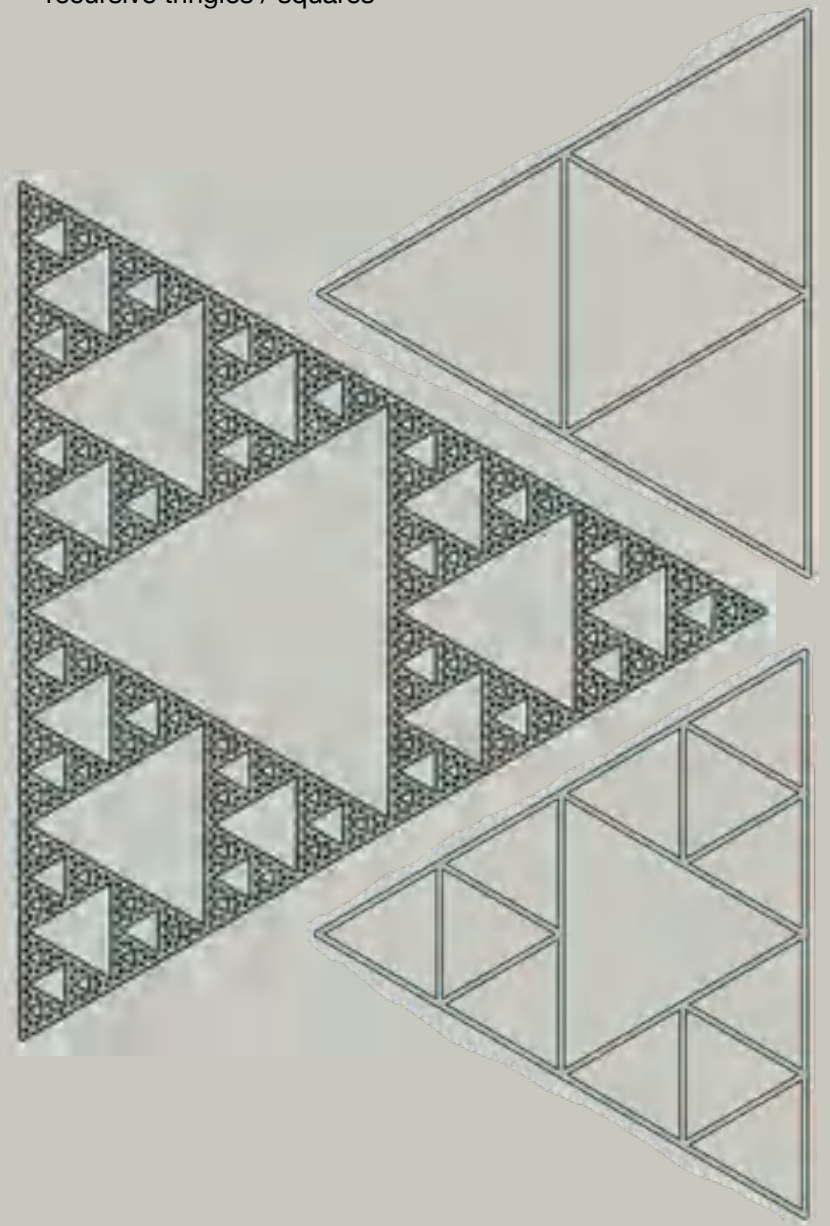


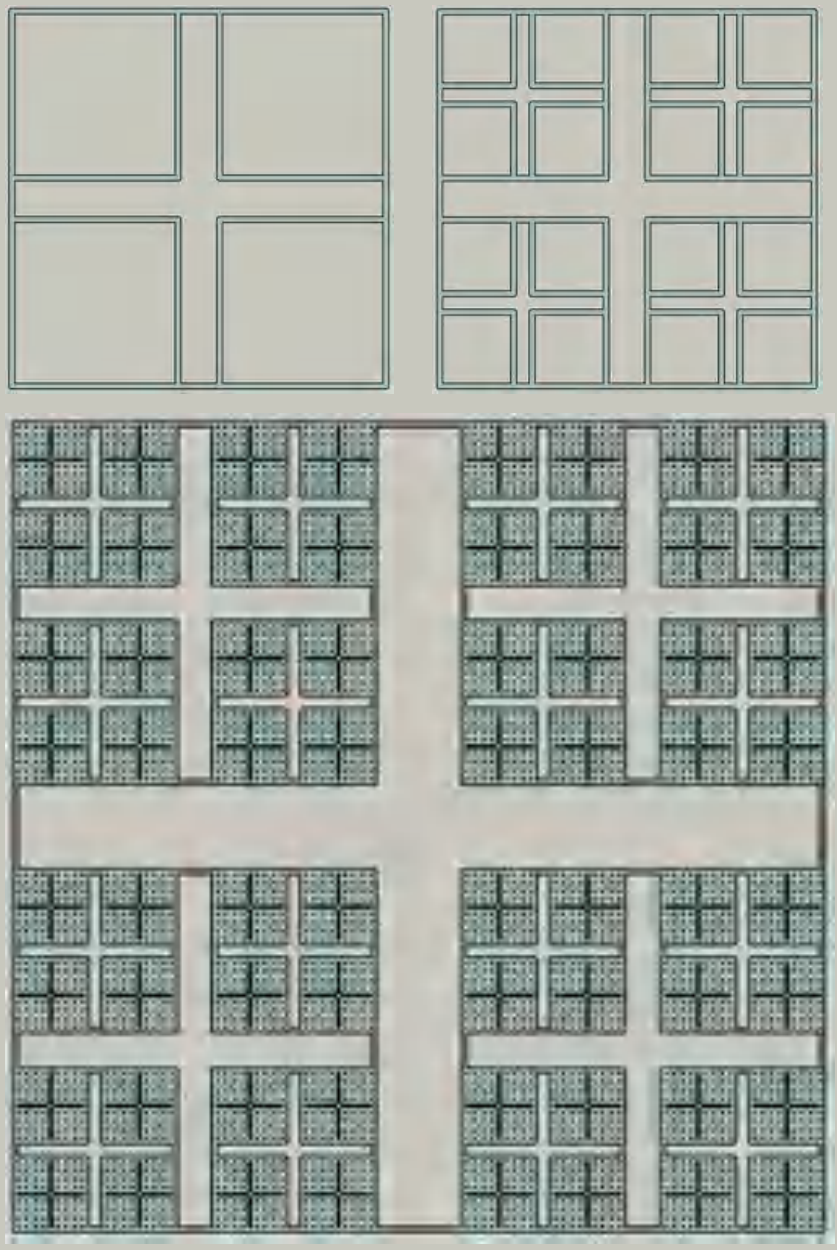
circle combinations



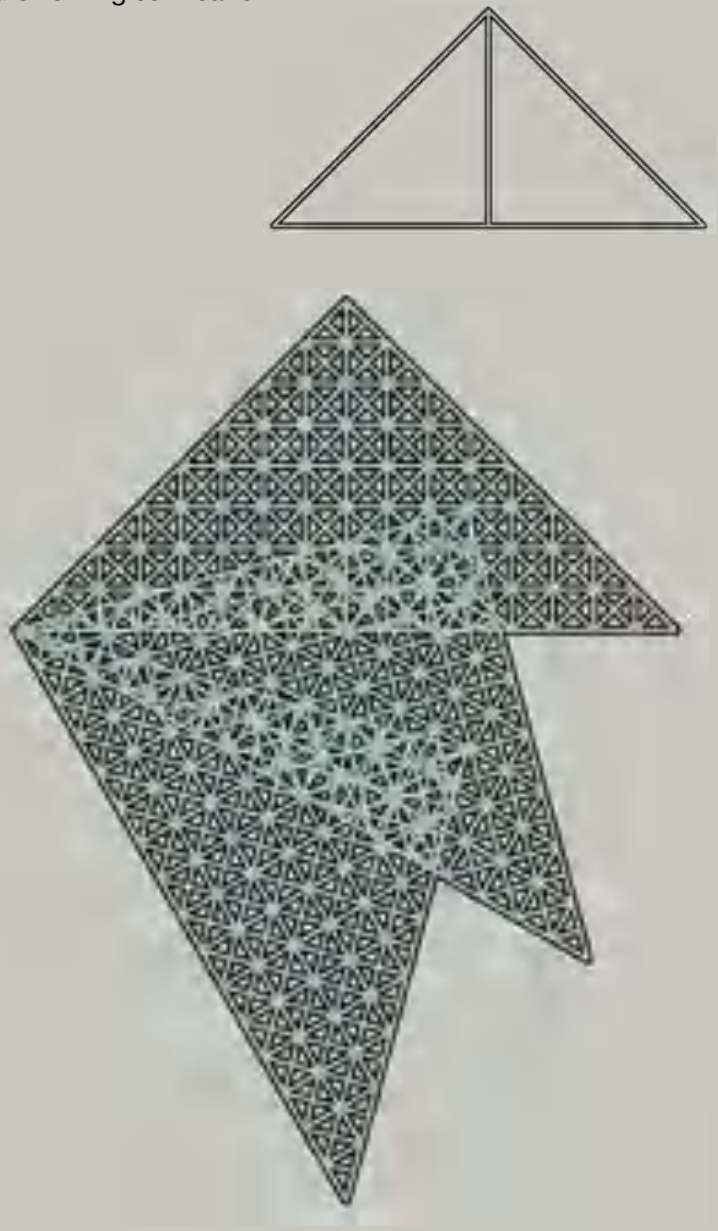


recursive tringles / squares

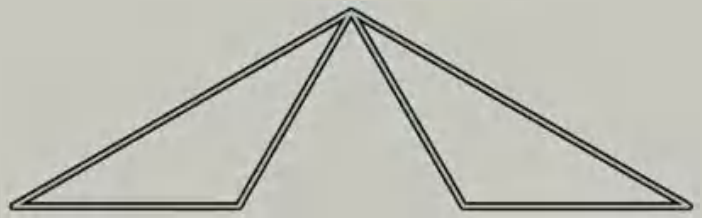


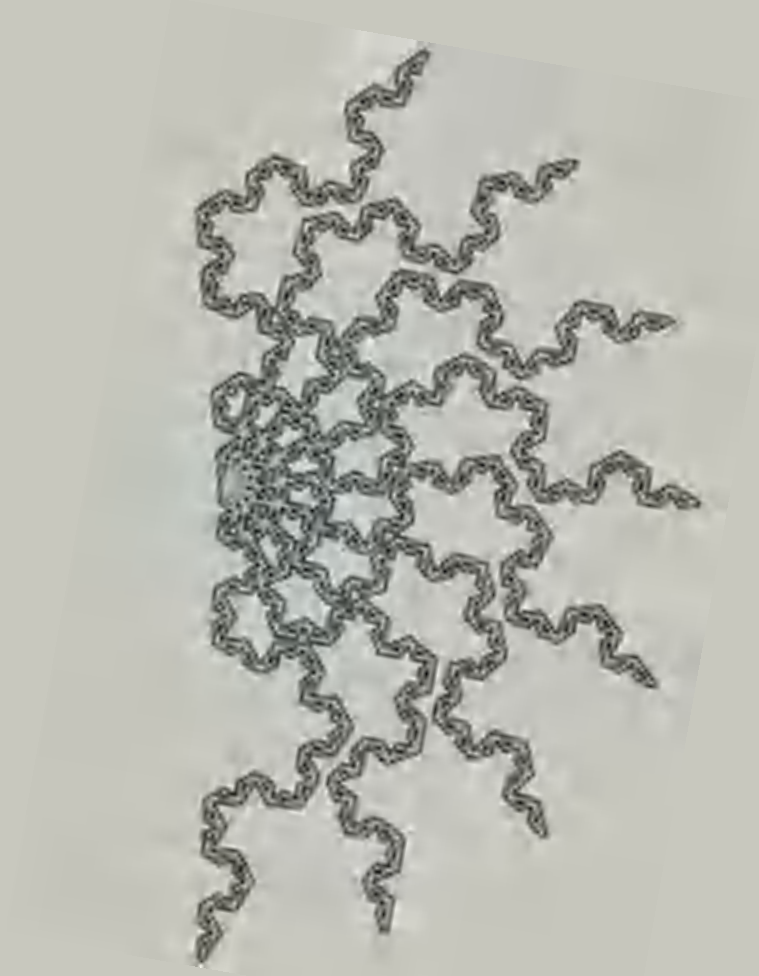


recursive tringles: Peano

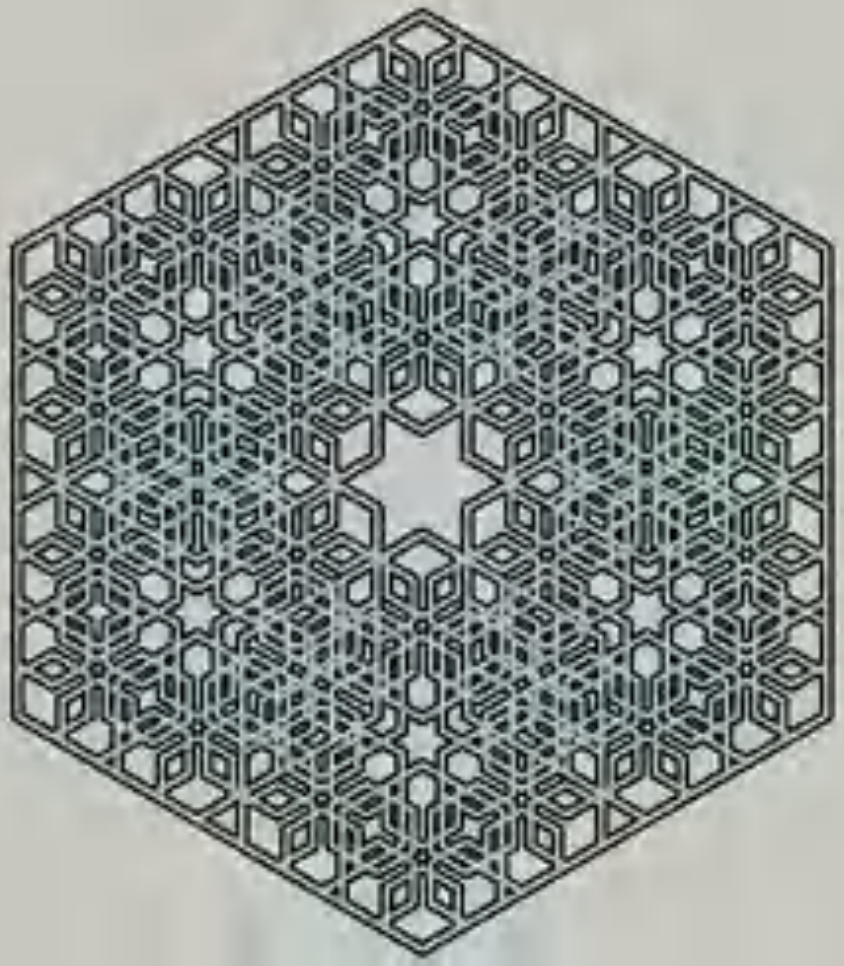


recursive tringles: Koch

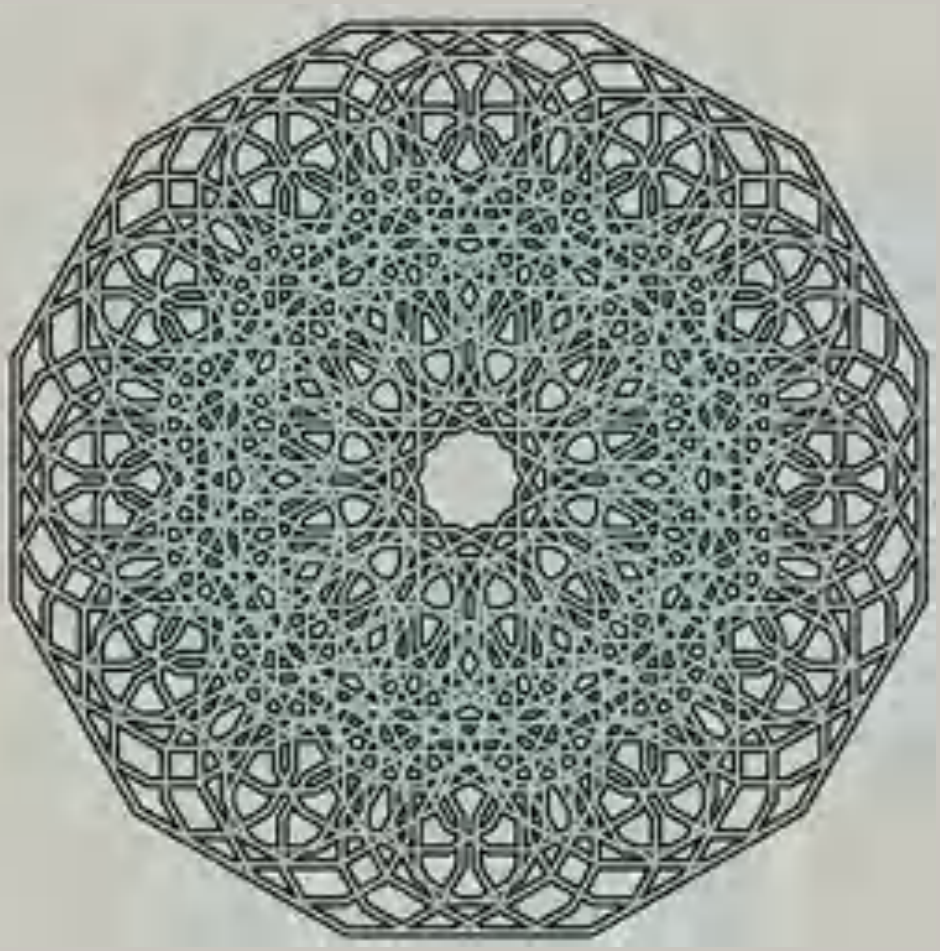




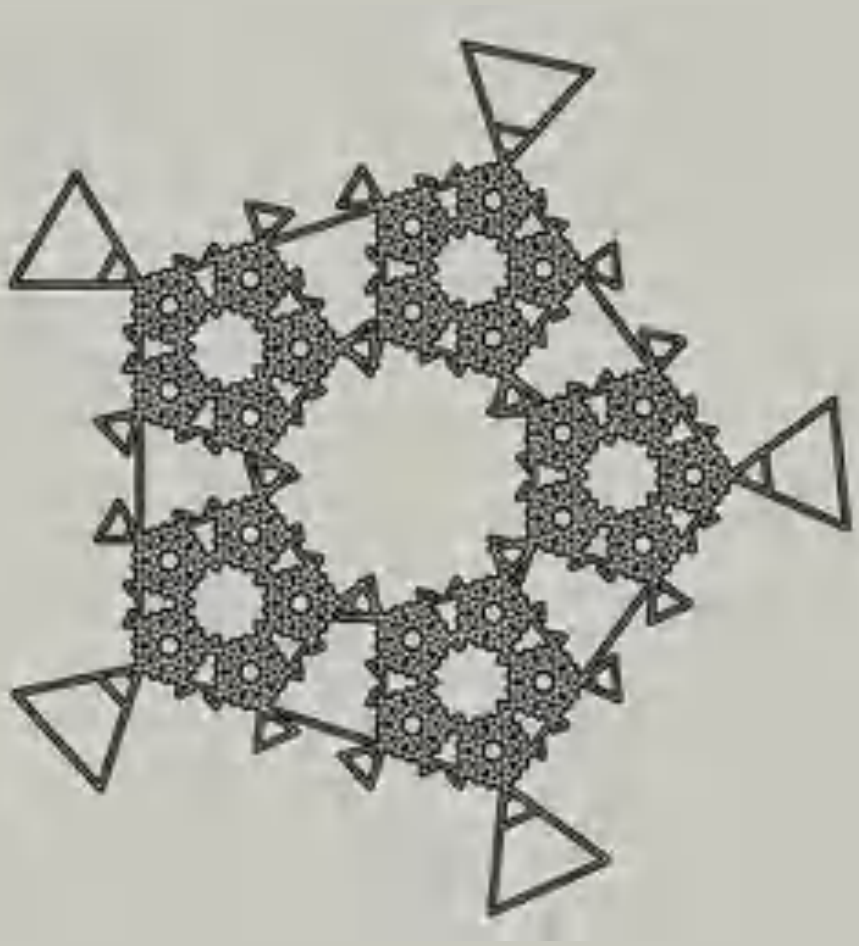
recursive hexagon

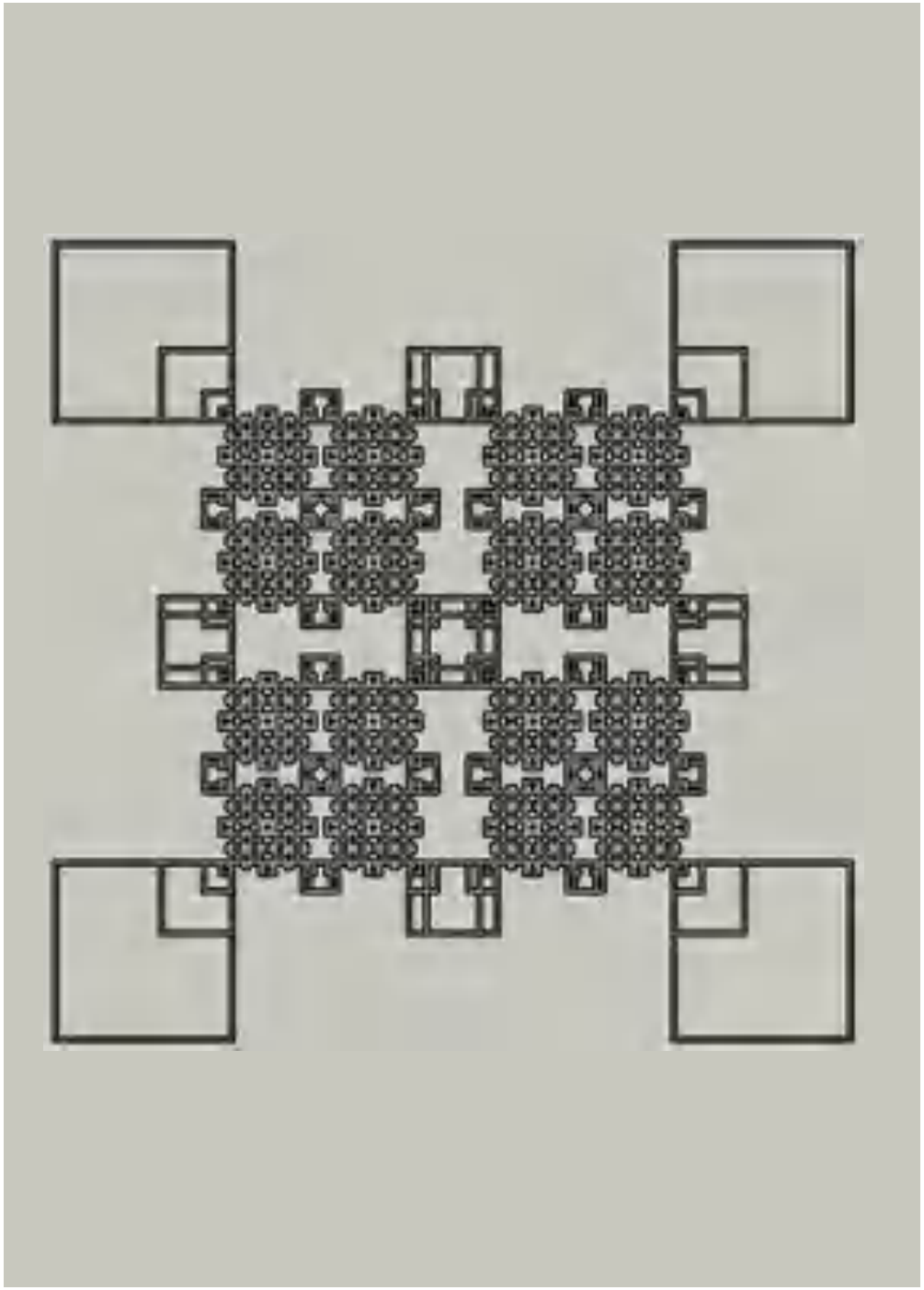


recursive dodecagon

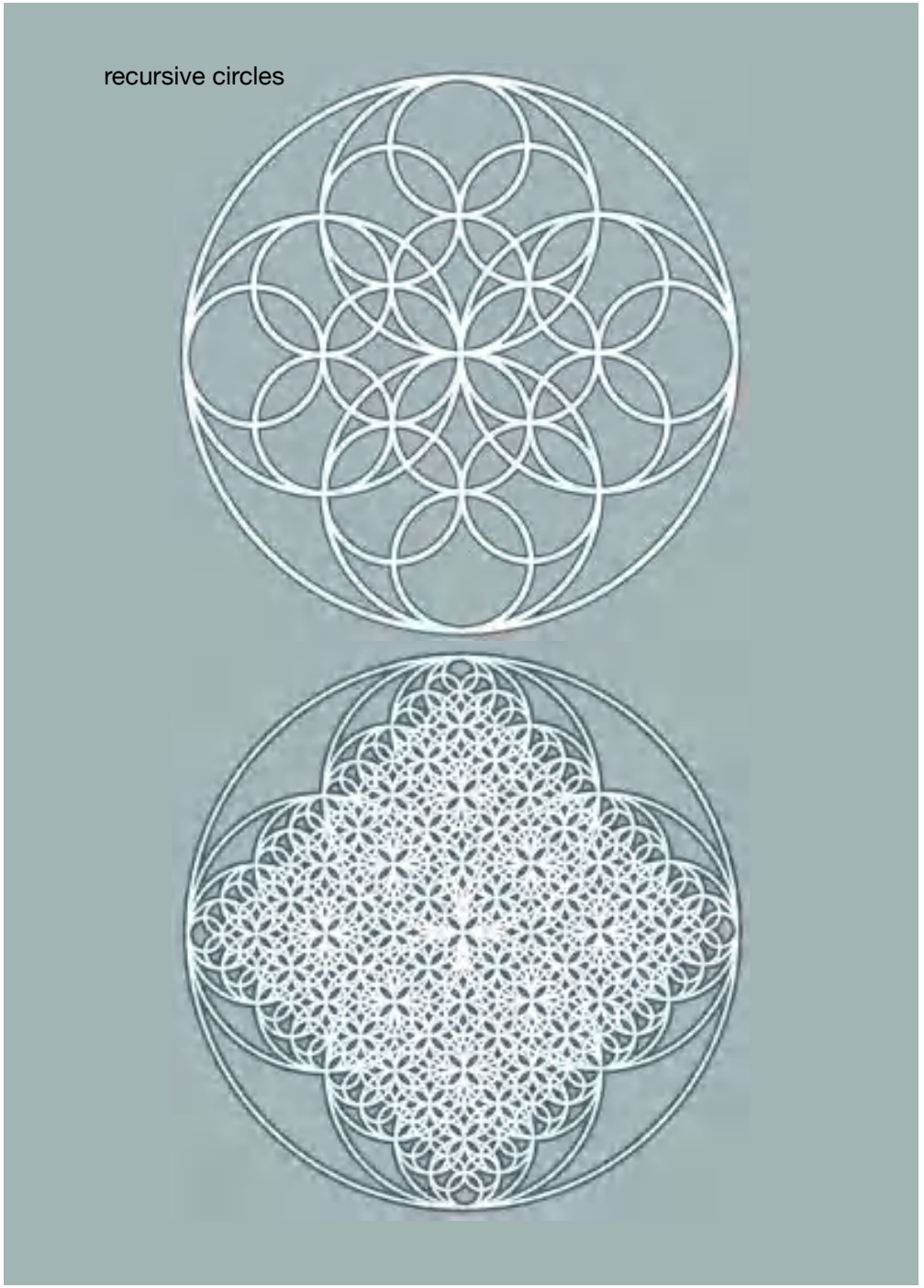


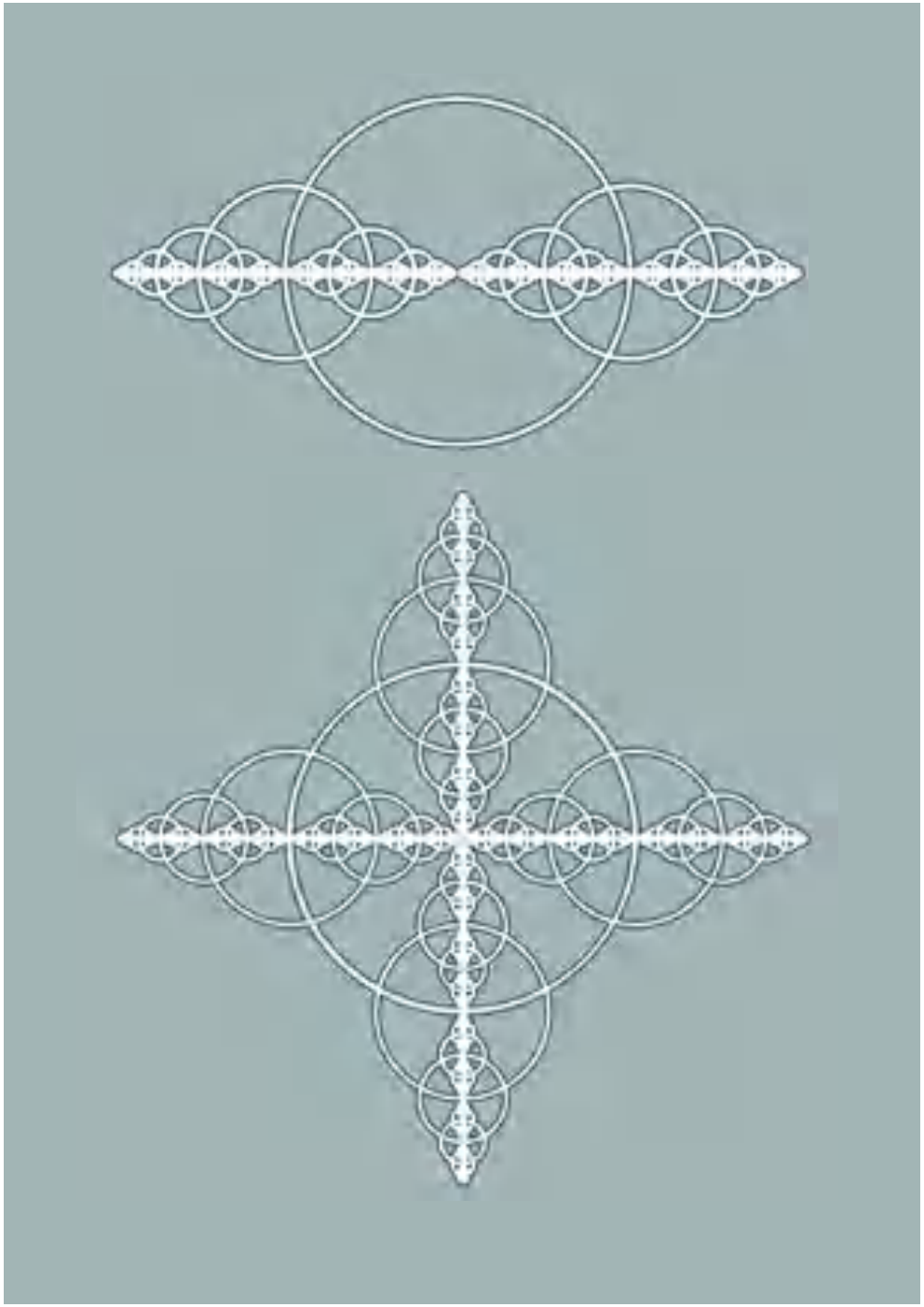
recursive polygon combinations





recursive circles





Supersigns

The principle of the following examples is the **rotation** and **mirroring** of lines, arcs or polygons. This way new "supersigns" can be created. By this we mean perceived entireties composed of the elementary signs.

Then, by **repeating** and **combining** the new supersigns in a grid, interesting endless patterns can be created. Such periodic repetitions are often found in decorative patterns.

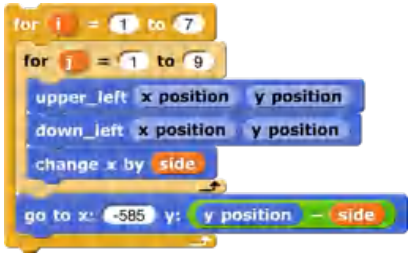
For the following applications, four similar graphic elements are created: **down_left**, **upper_left**, **down_right**, **and upper_right**:

1. three corners
2. (diagonal) lines
3. triangles
4. Escher triangles [6]
5. bars
6. arcs

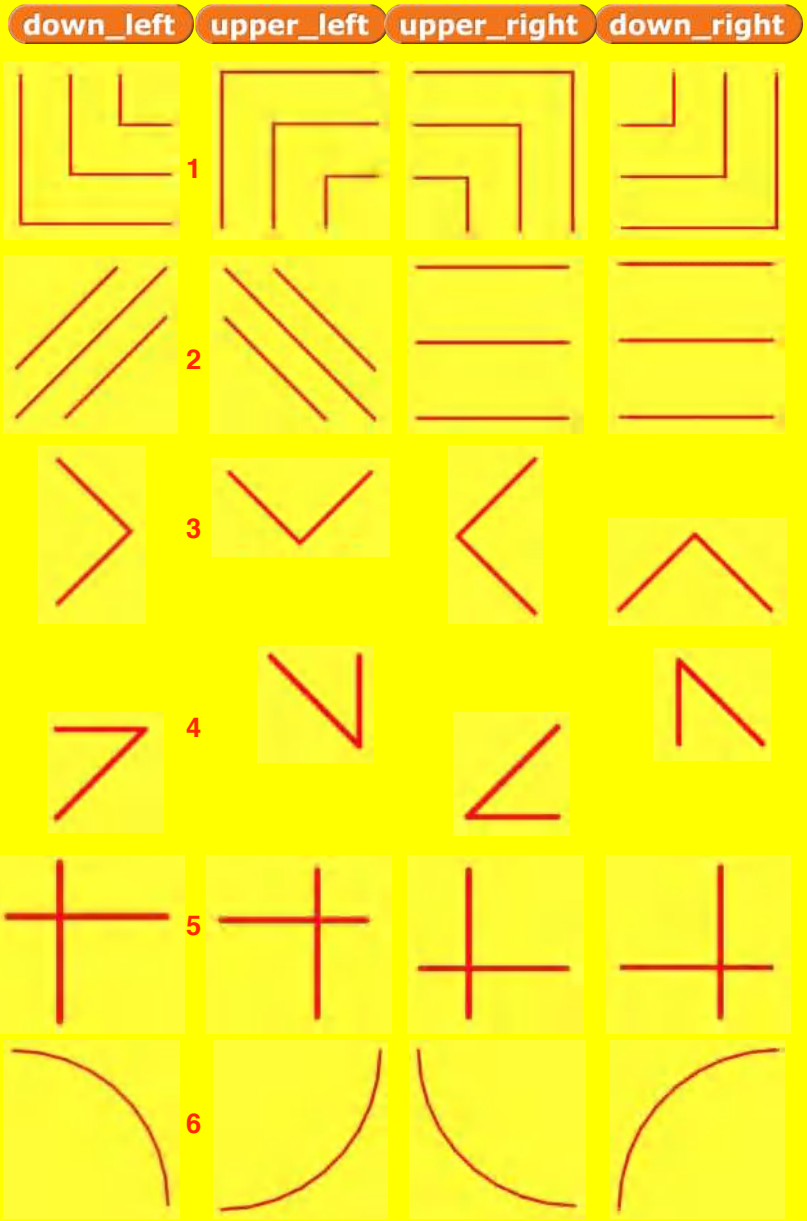
The procedures for the graphic elements are internally always structured in the same way. The example **down_left** shows that simple commands for moving the turtle are sufficient.

Make sure that the turtle is back at the starting point after drawing.

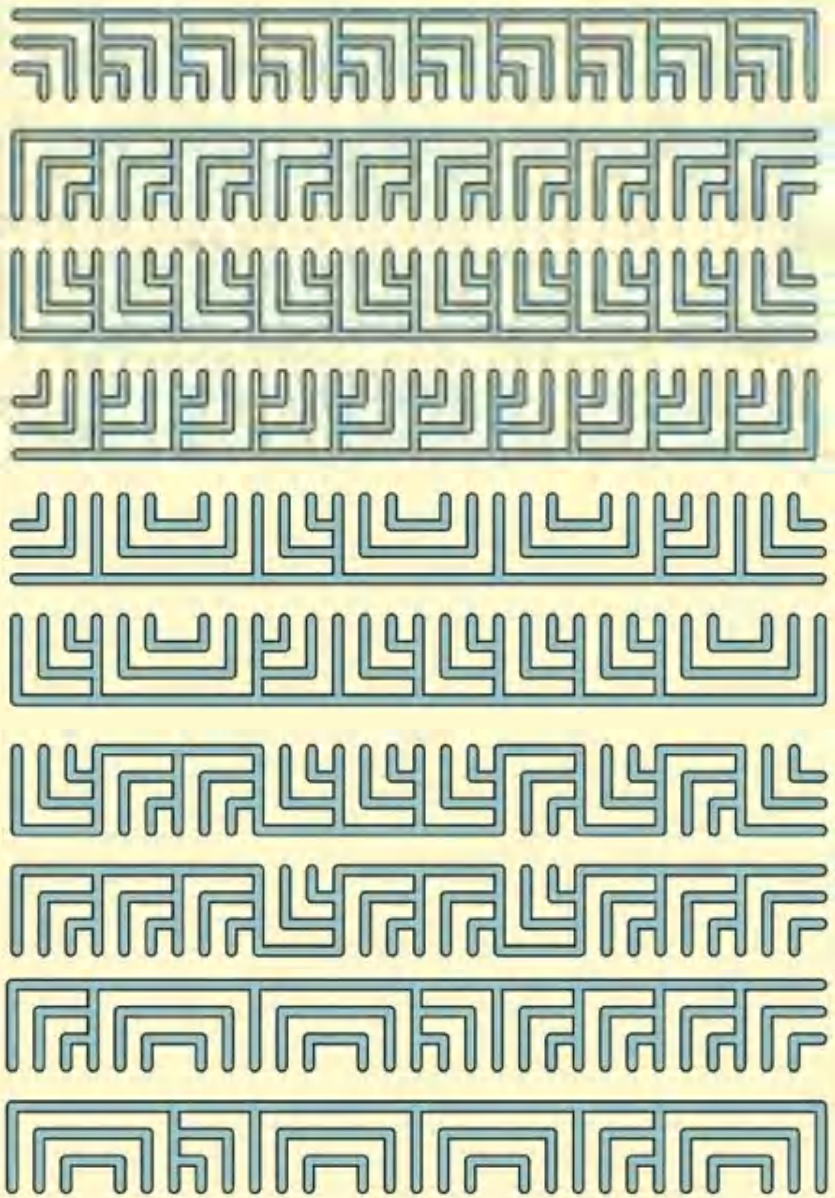
The elements can then be repeated systematically or randomly within loops.

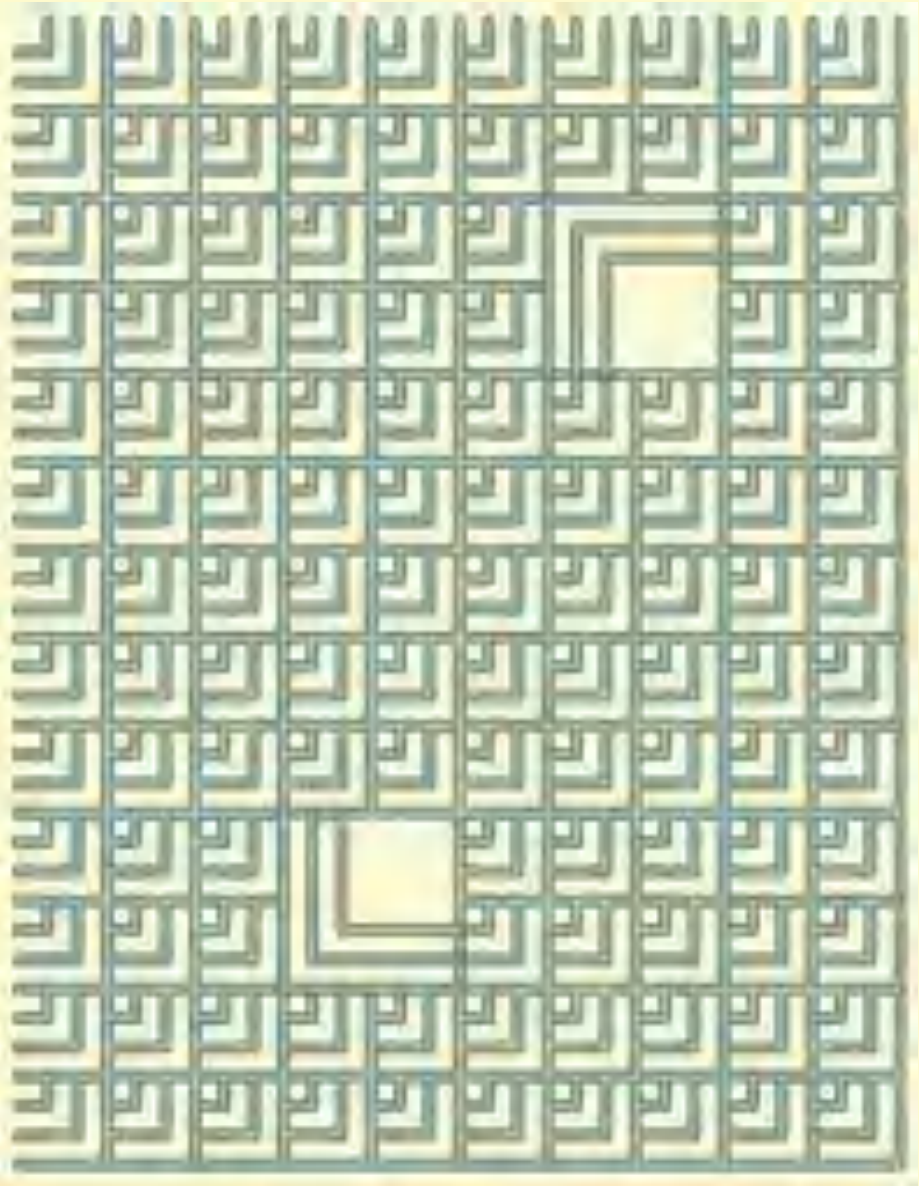


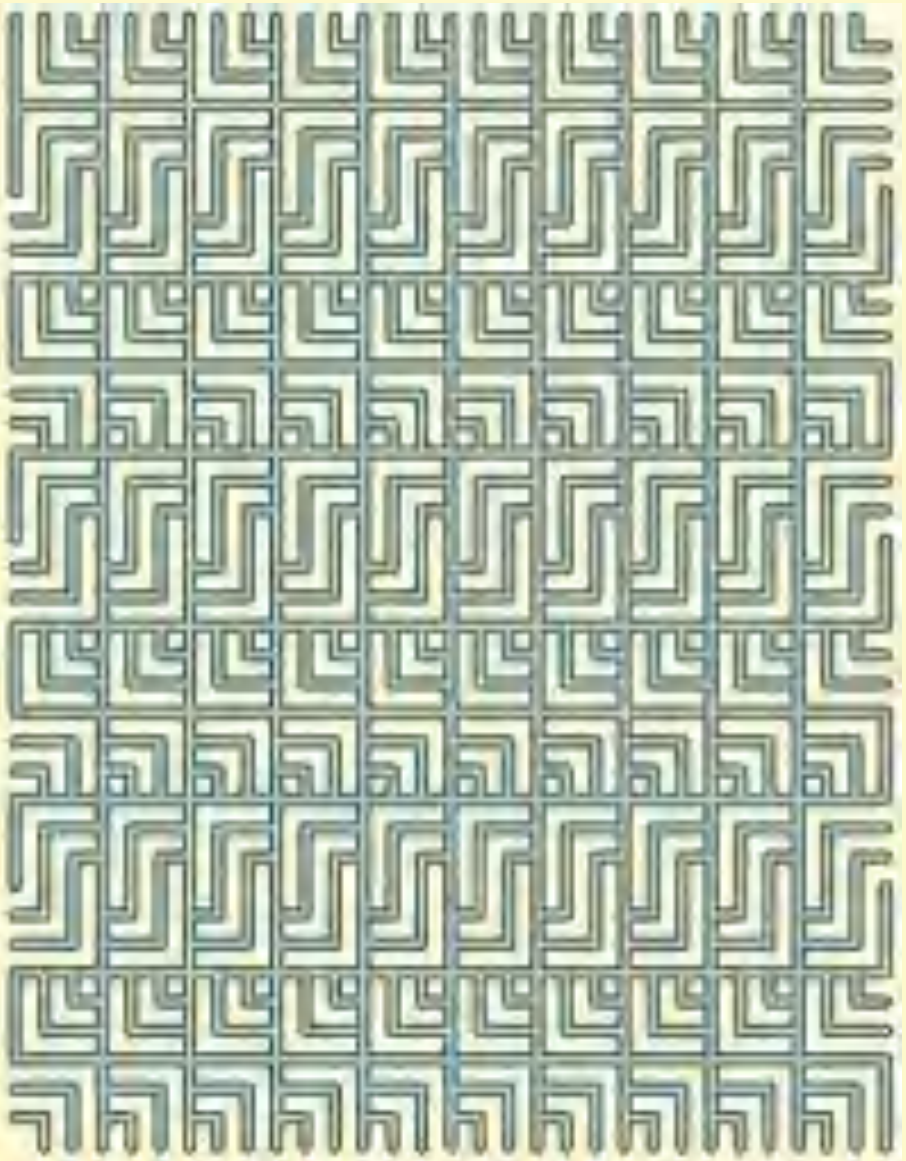
[6] The example can be found in one of his early workbooks, in which he experimented with such patterns (Schattschneider, 1990, p. 45).



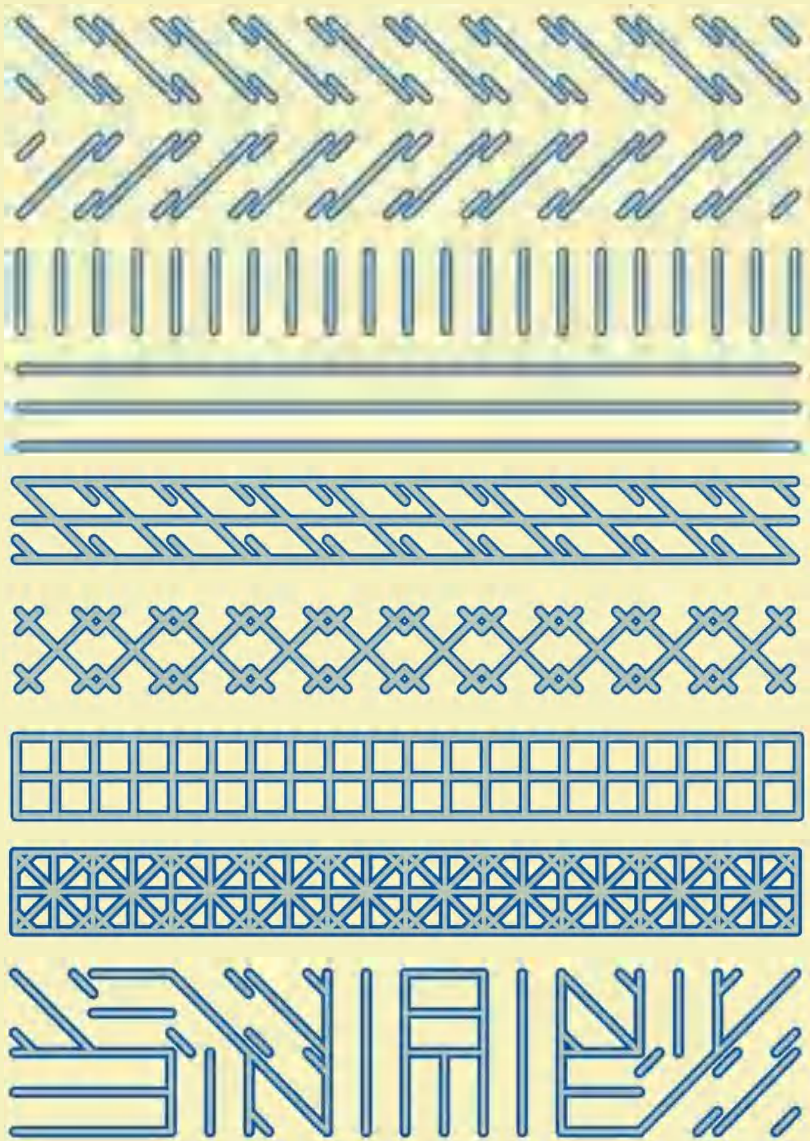
three corners

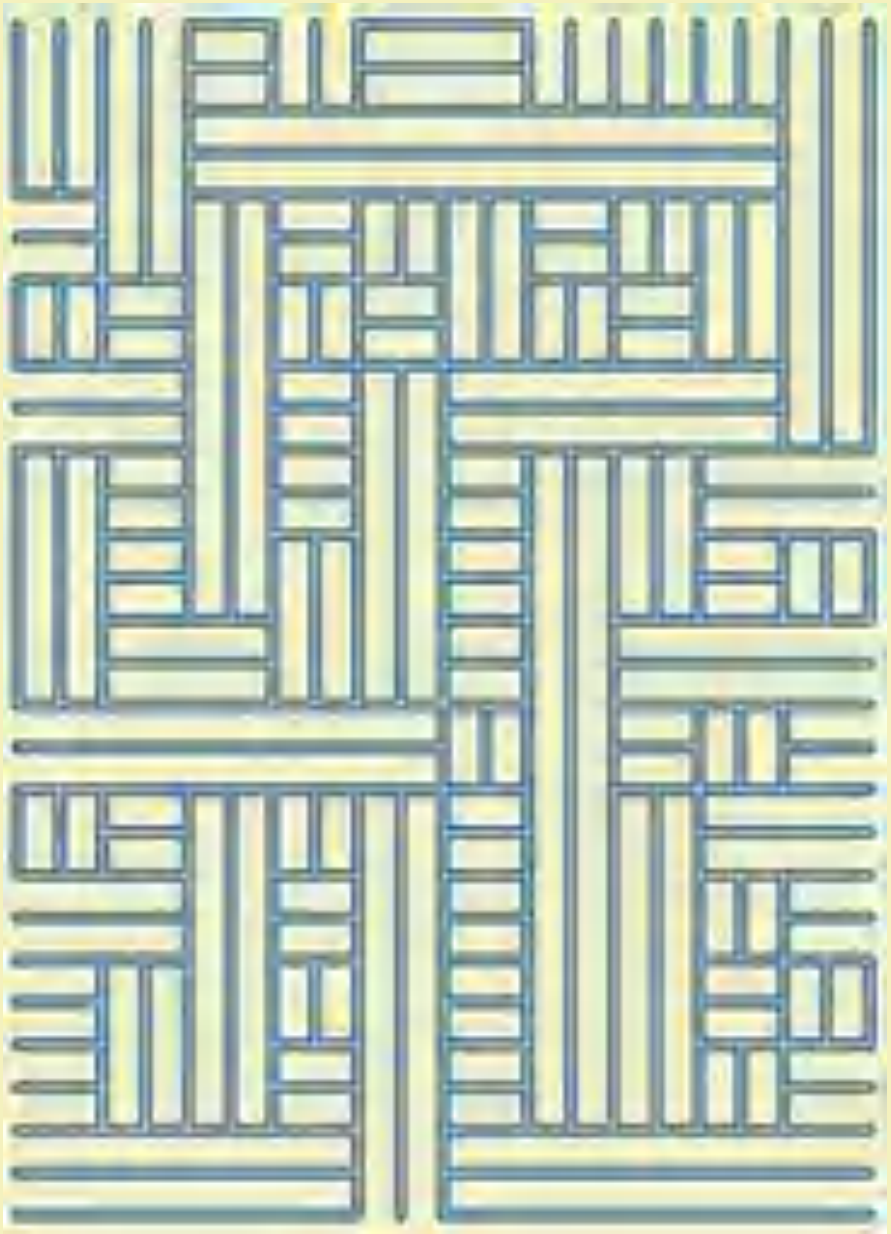


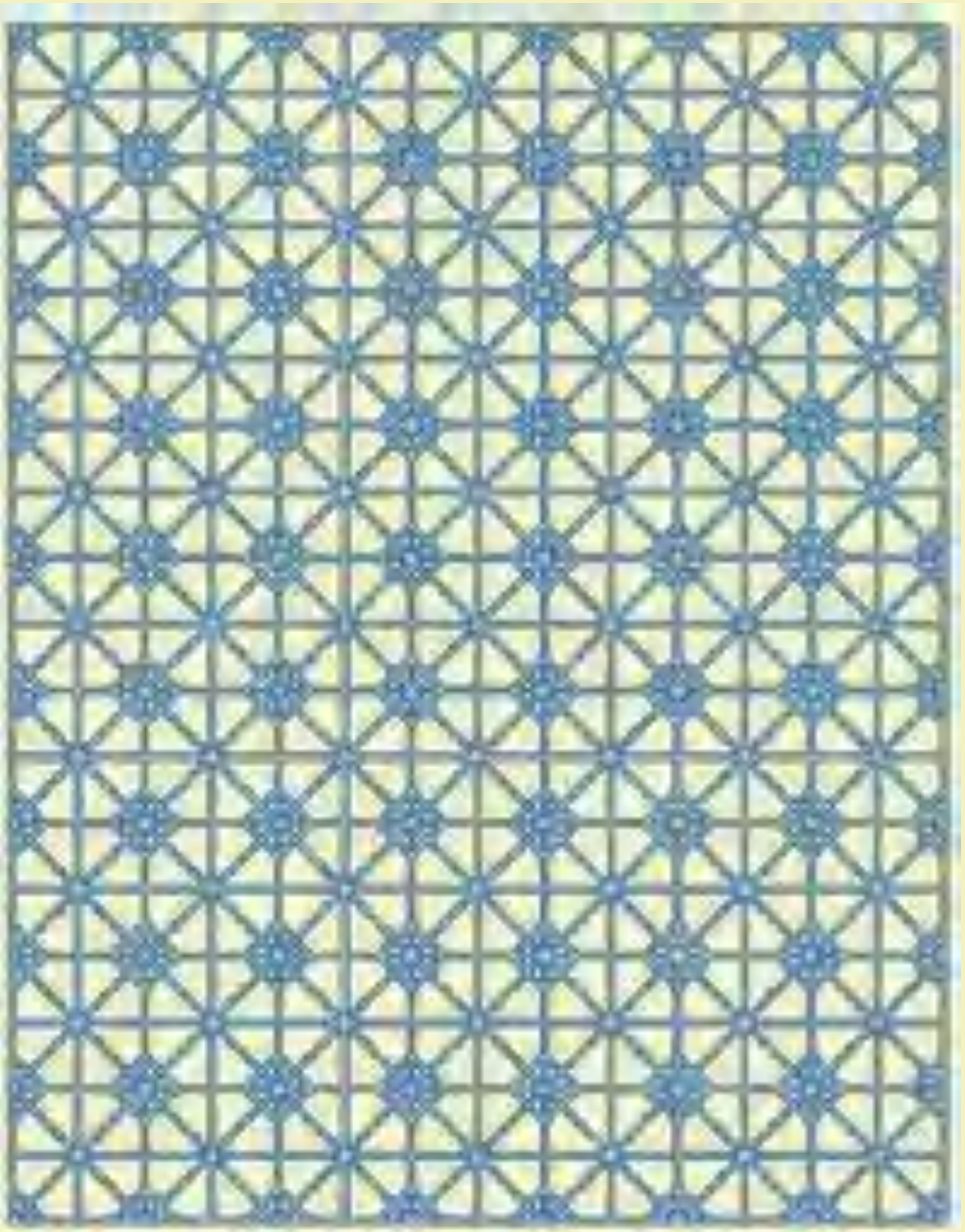




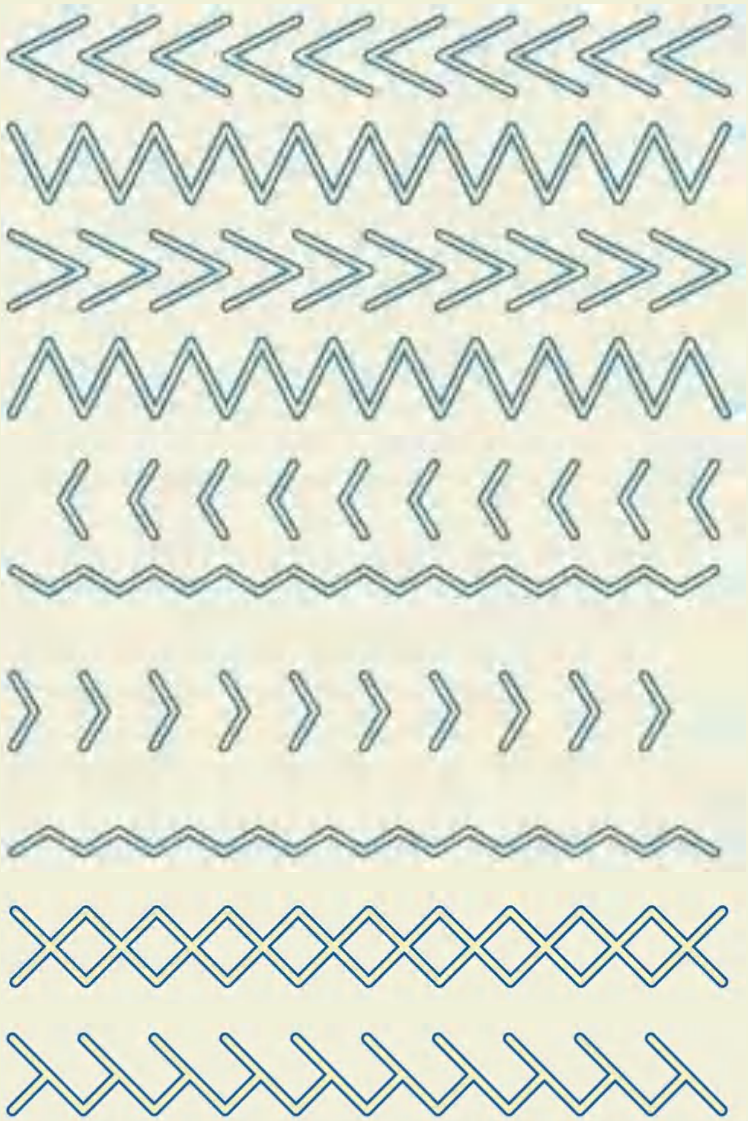
diagonal lines

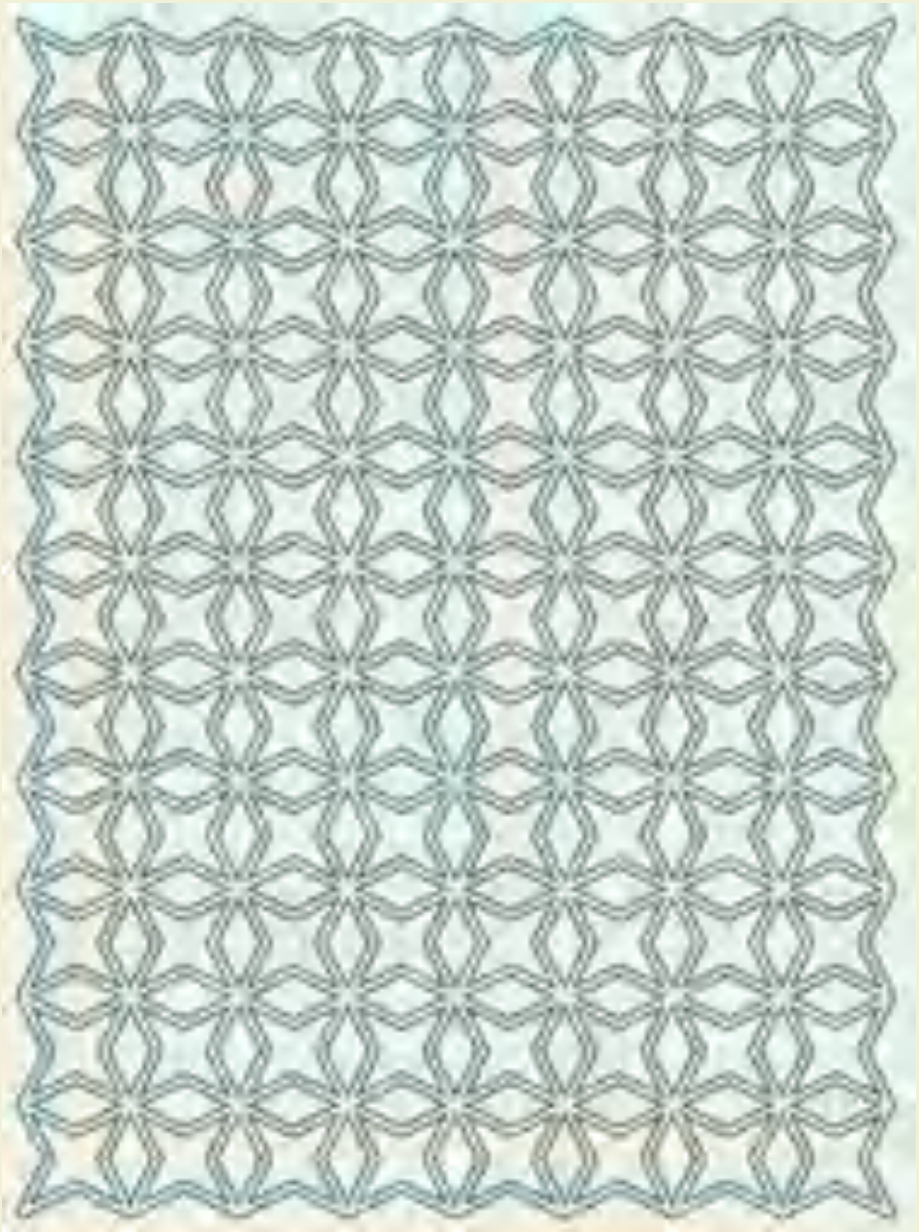


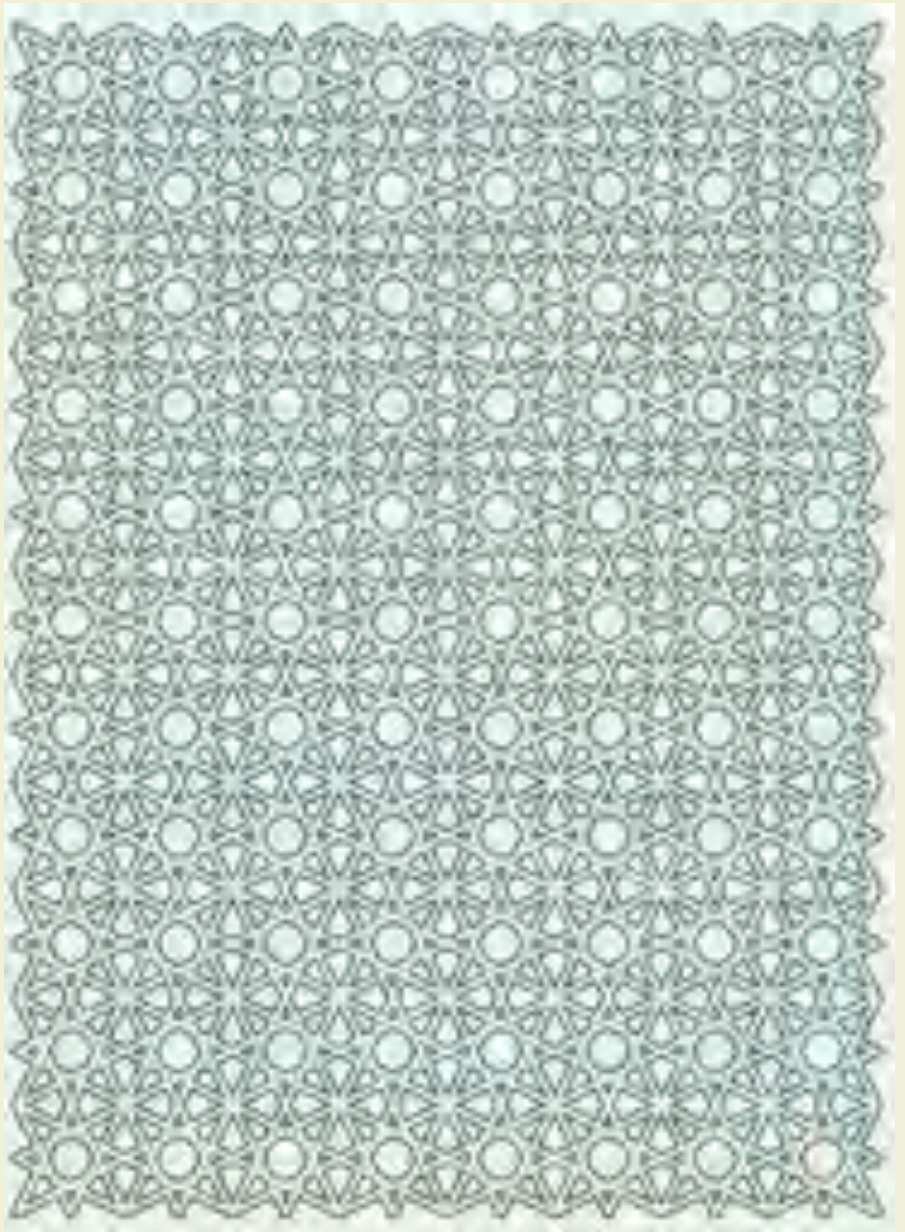


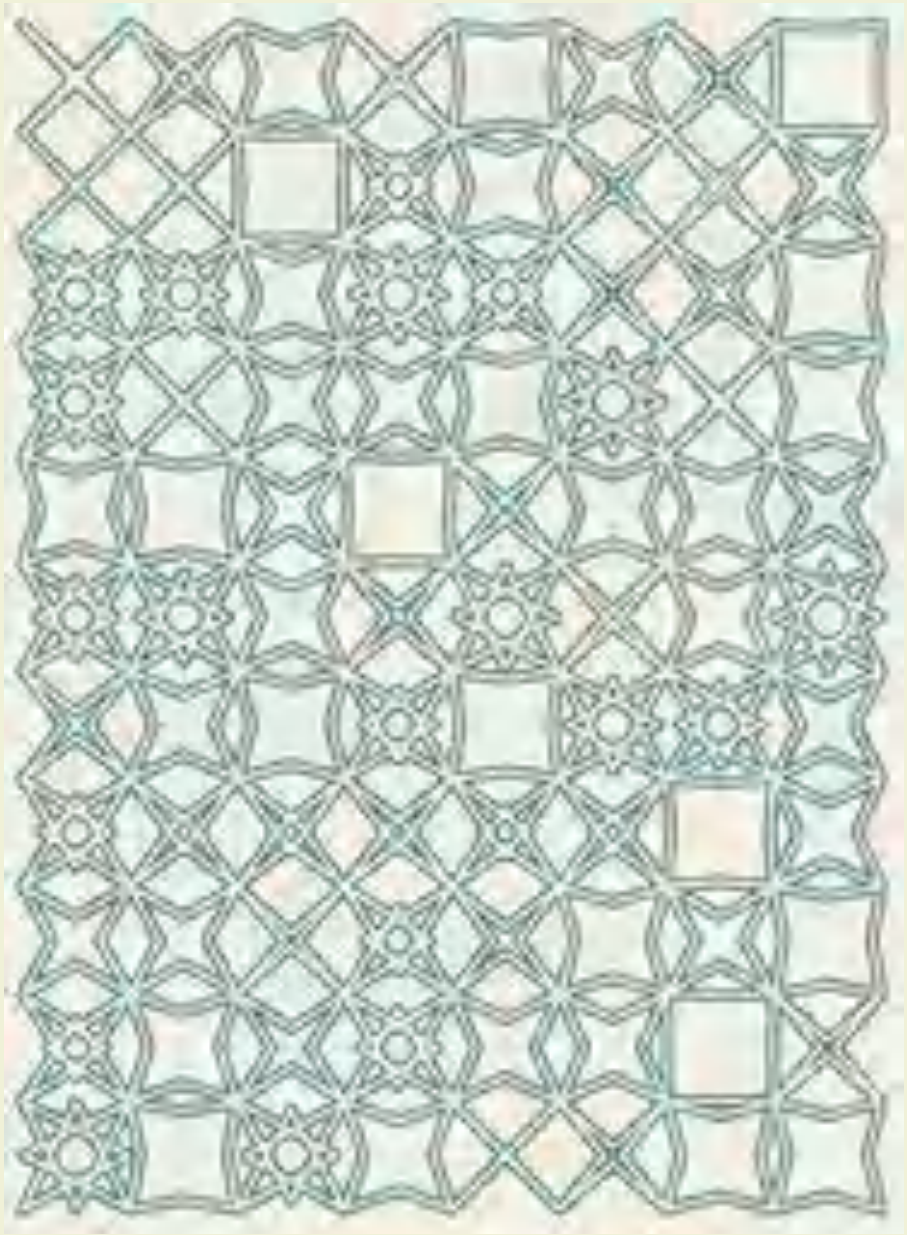


triangles

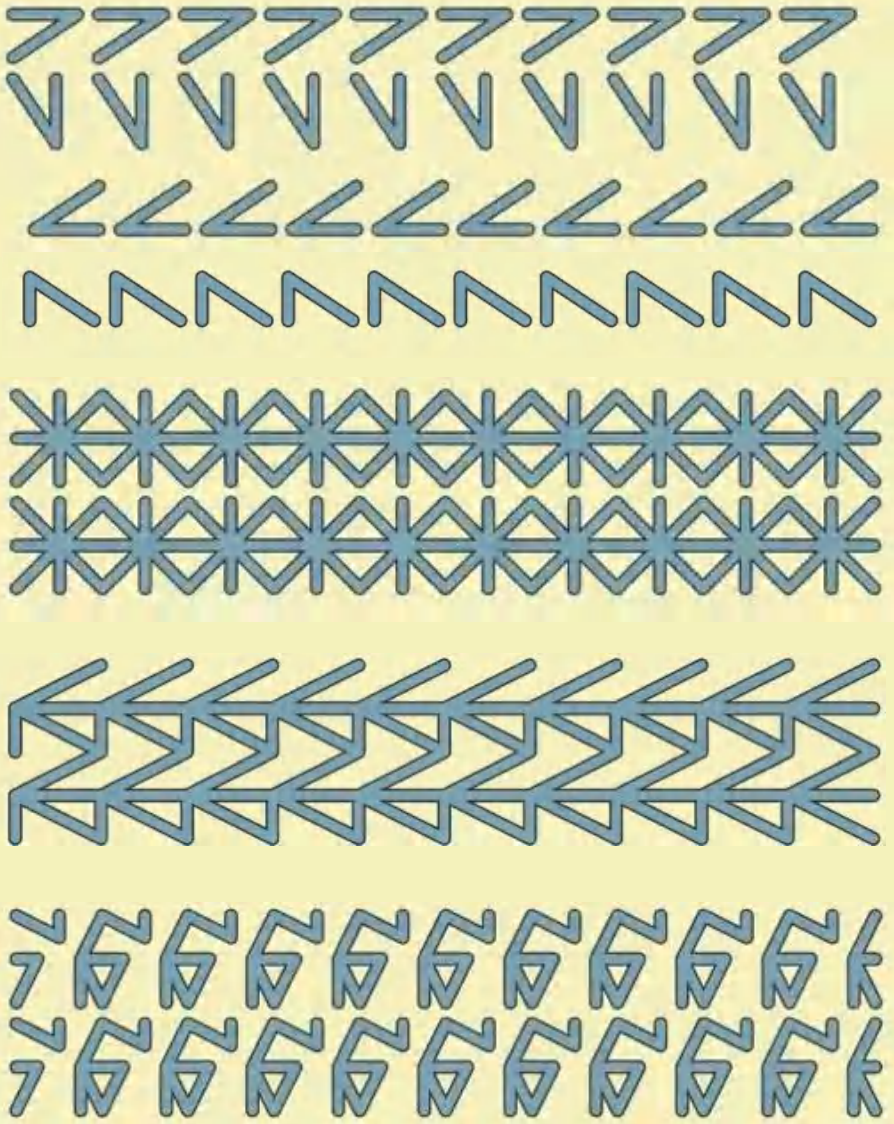


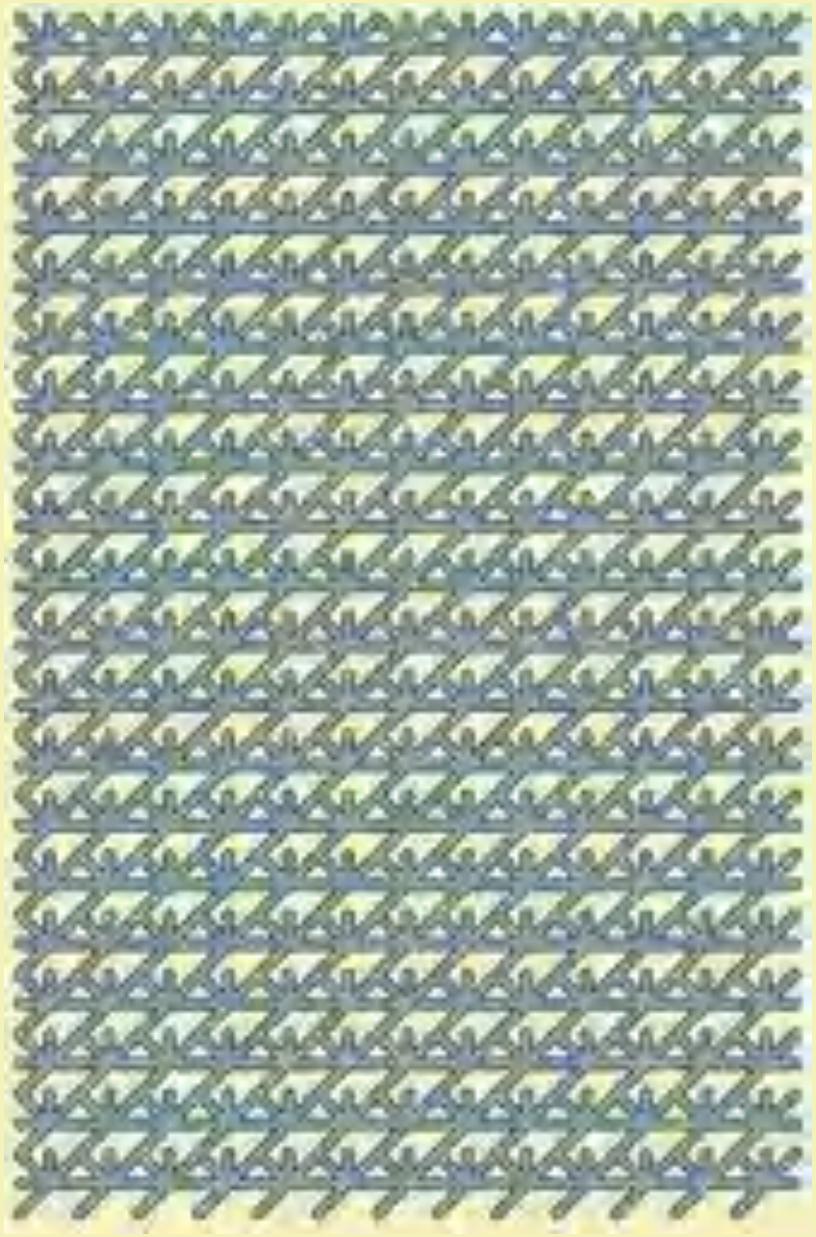


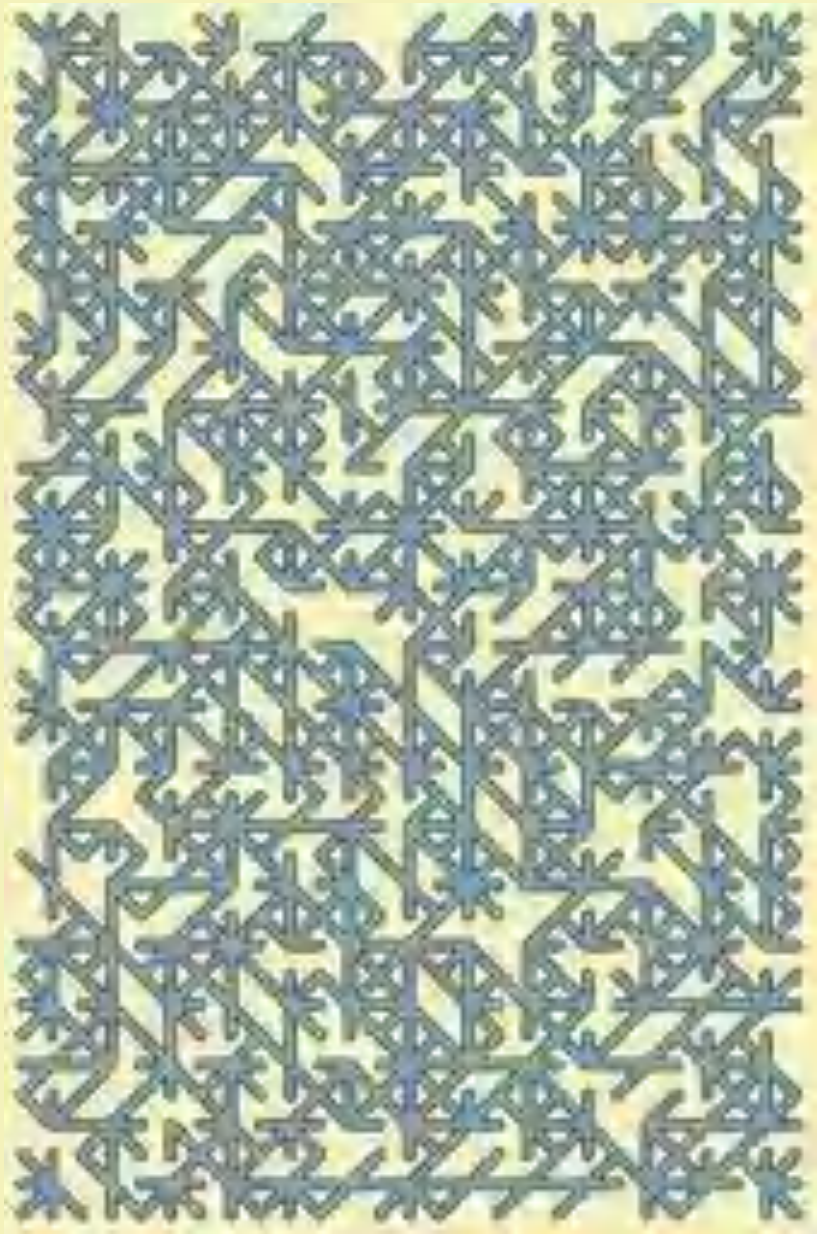


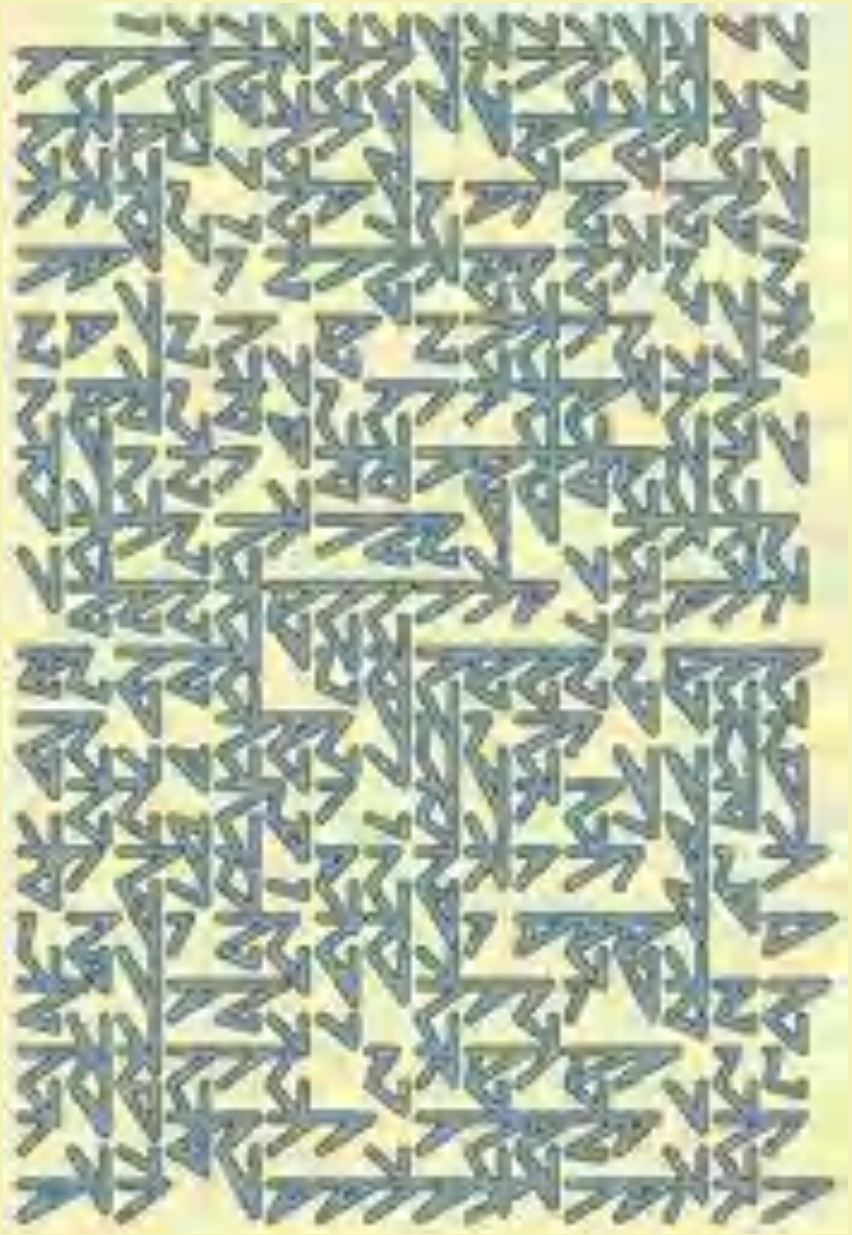


Escher-triangles

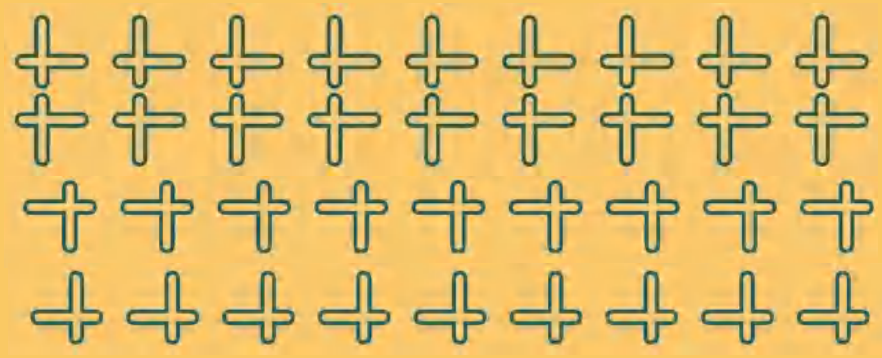


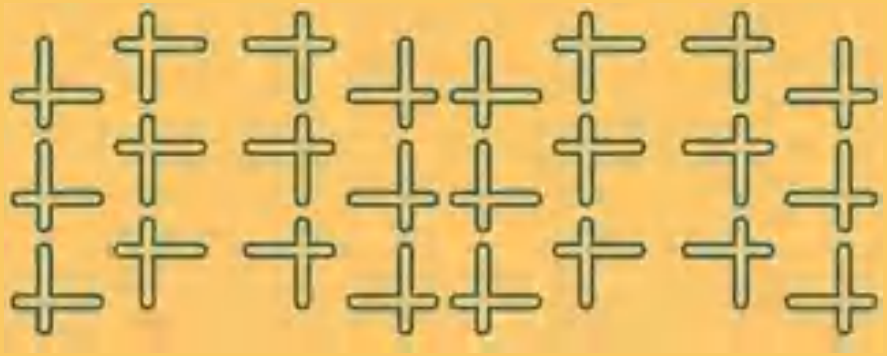


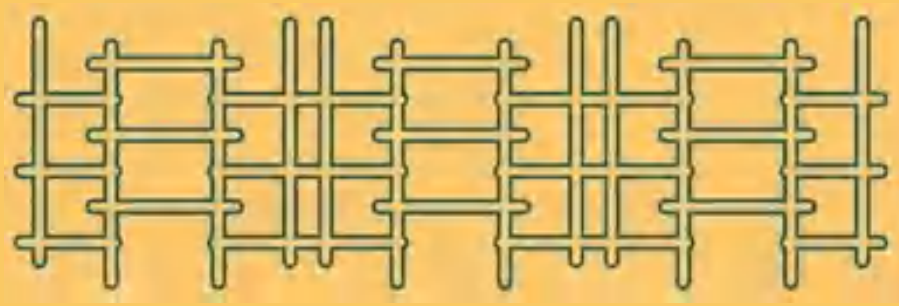


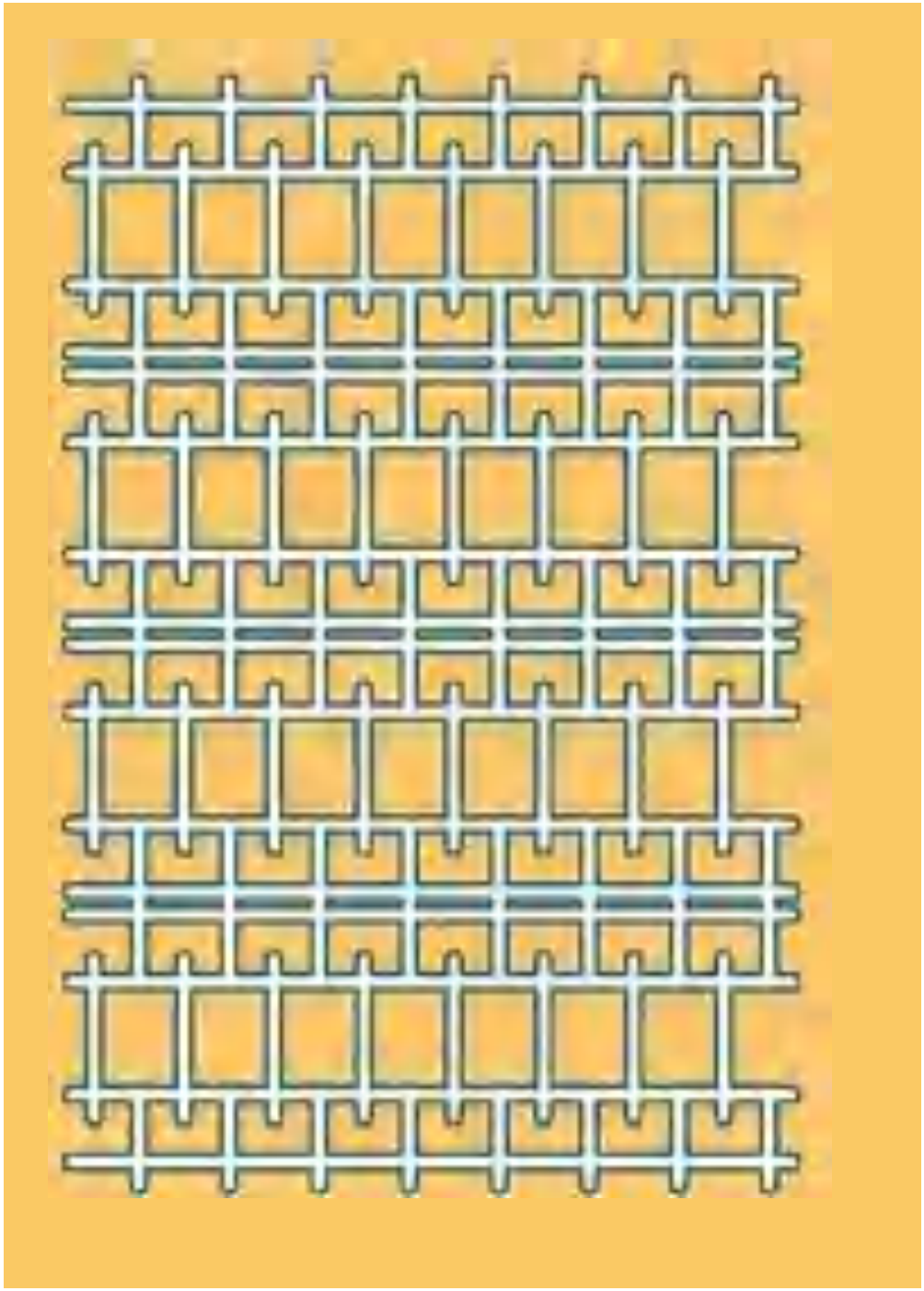


bars

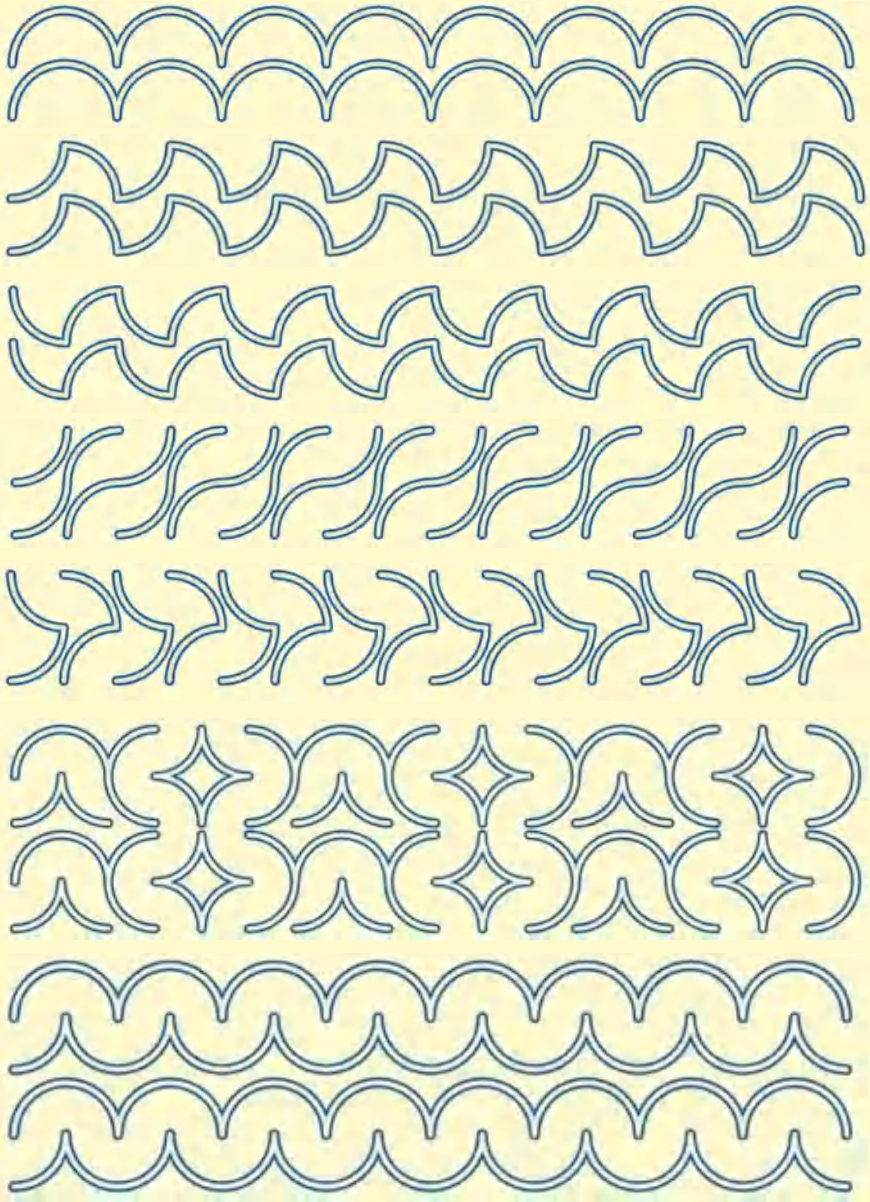


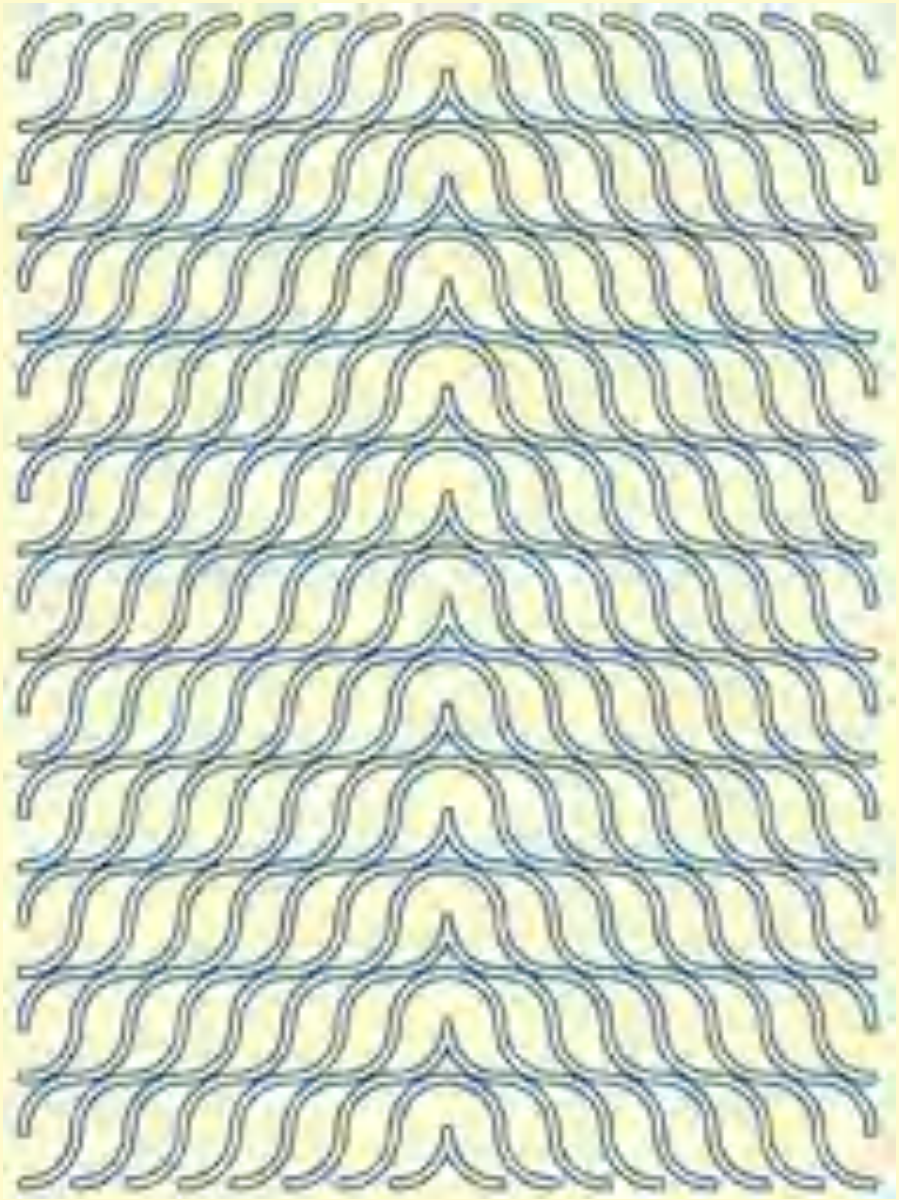


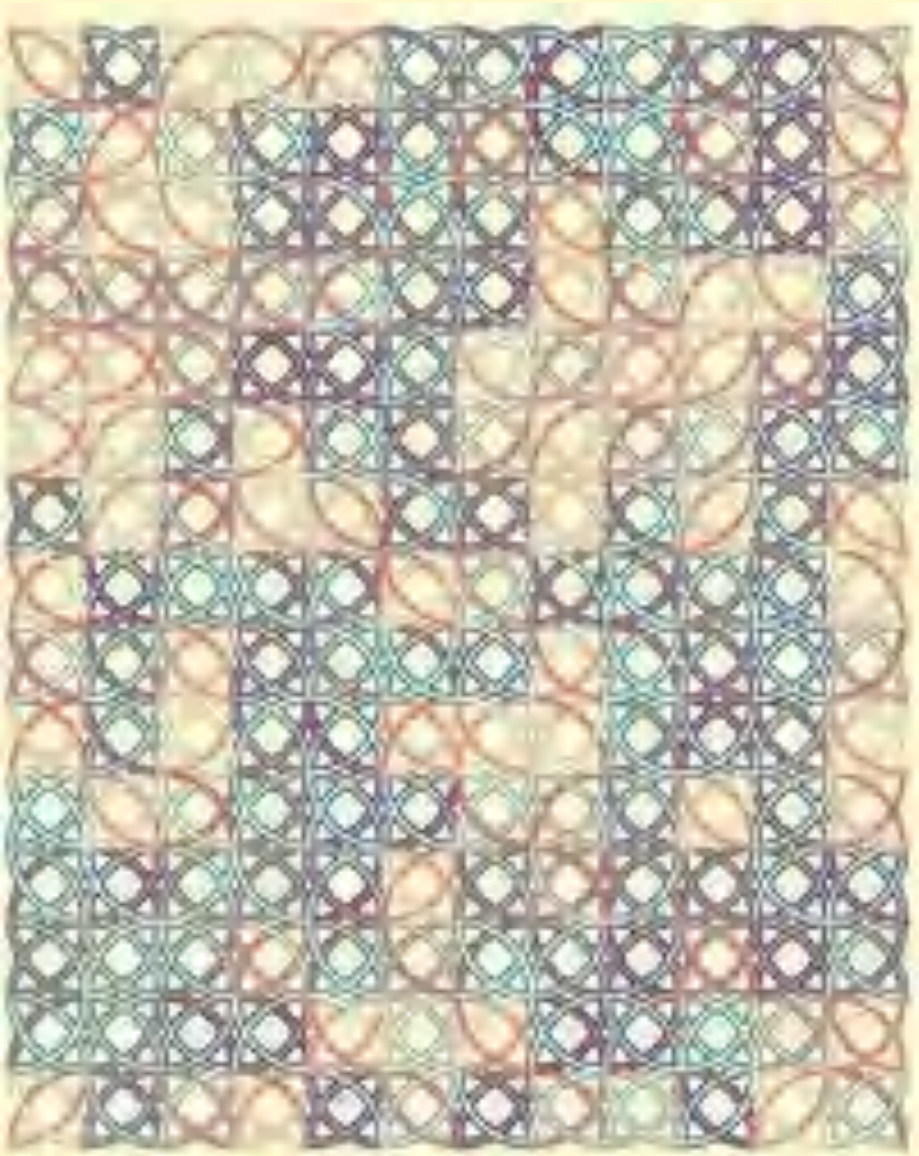


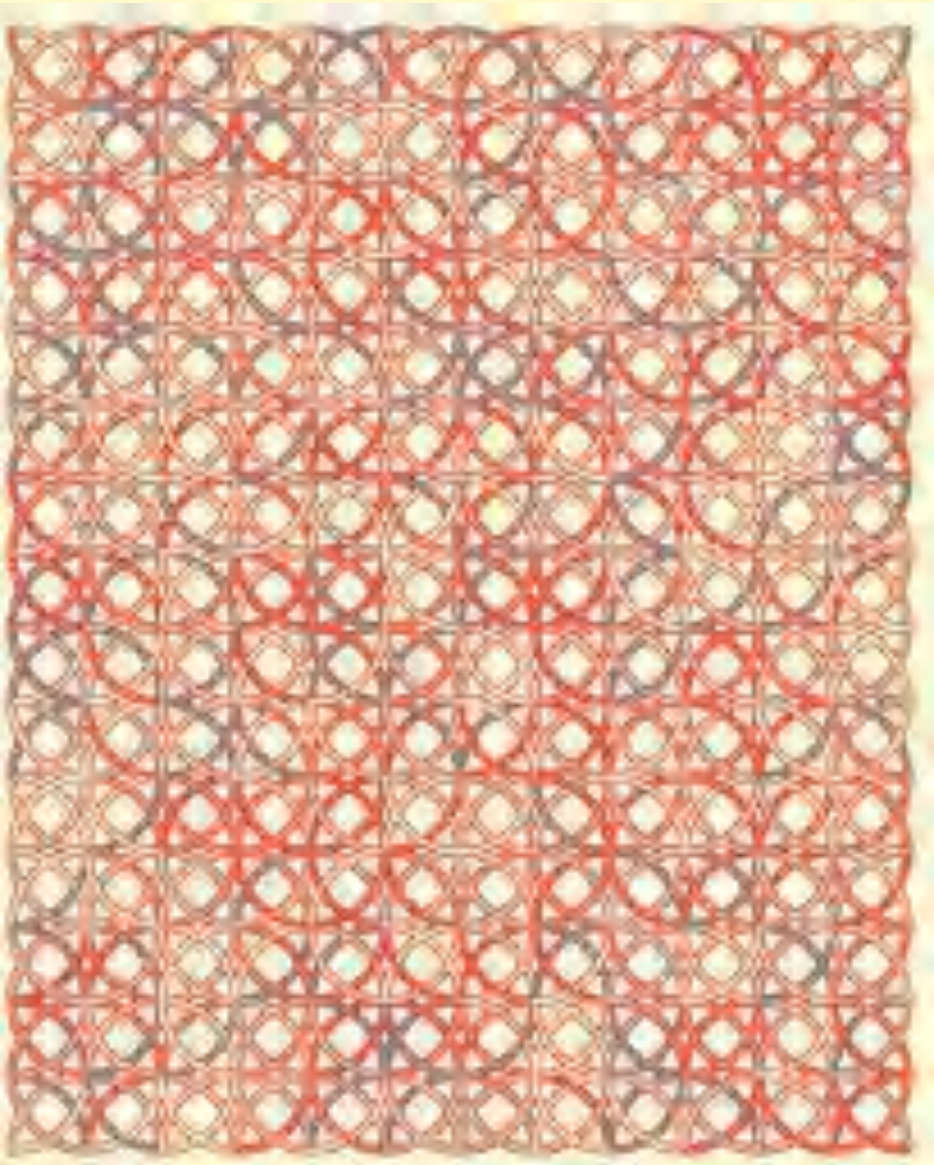


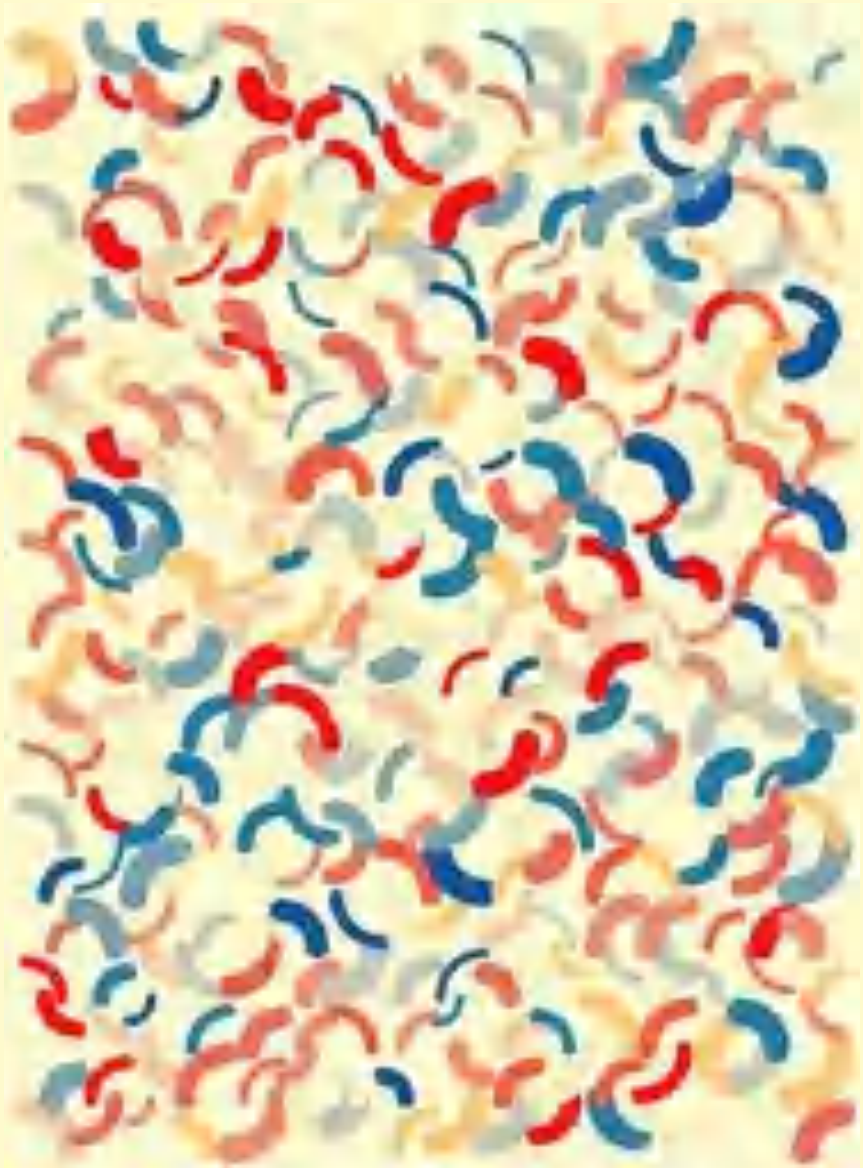
arcs











Moiré-patterns

By **superimposing** the graphic elements already known (lines, circles, points), surprising optical effects can be achieved (this is called the Moiré effect). Their creation is very simple[7].

The procedure is the same throughout (the blue dot in the graphics on the right always marks the center of the image):

- A first basic pattern is created with an element from the figure construction kit. This pattern forms the background (in the picture on the right, lines, circles, points and again lines from top to bottom).
- Depending on the desired effect, one or more additional patterns can be created and superimposed on the first pattern.
- The second pattern is usually slightly offset and/or rotated compared to the first one (in the picture on the right in the middle, lines, circles, points and again circles, slightly shifted from top to bottom).

The results can be found in the image on the right in the third column on the far right.

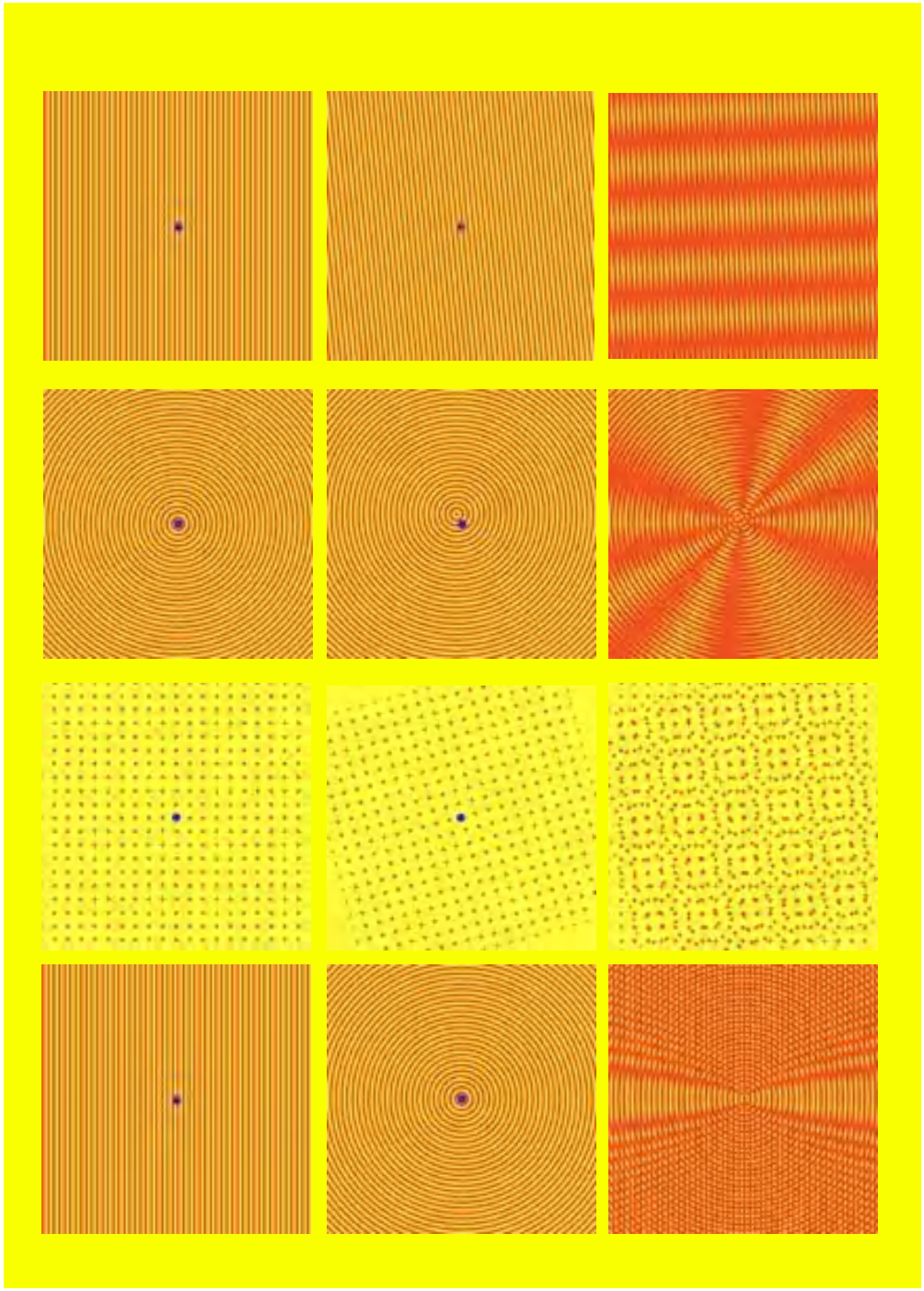
Moiré patterns are important in various technical areas. But they have also been used in painting since Op-Art.

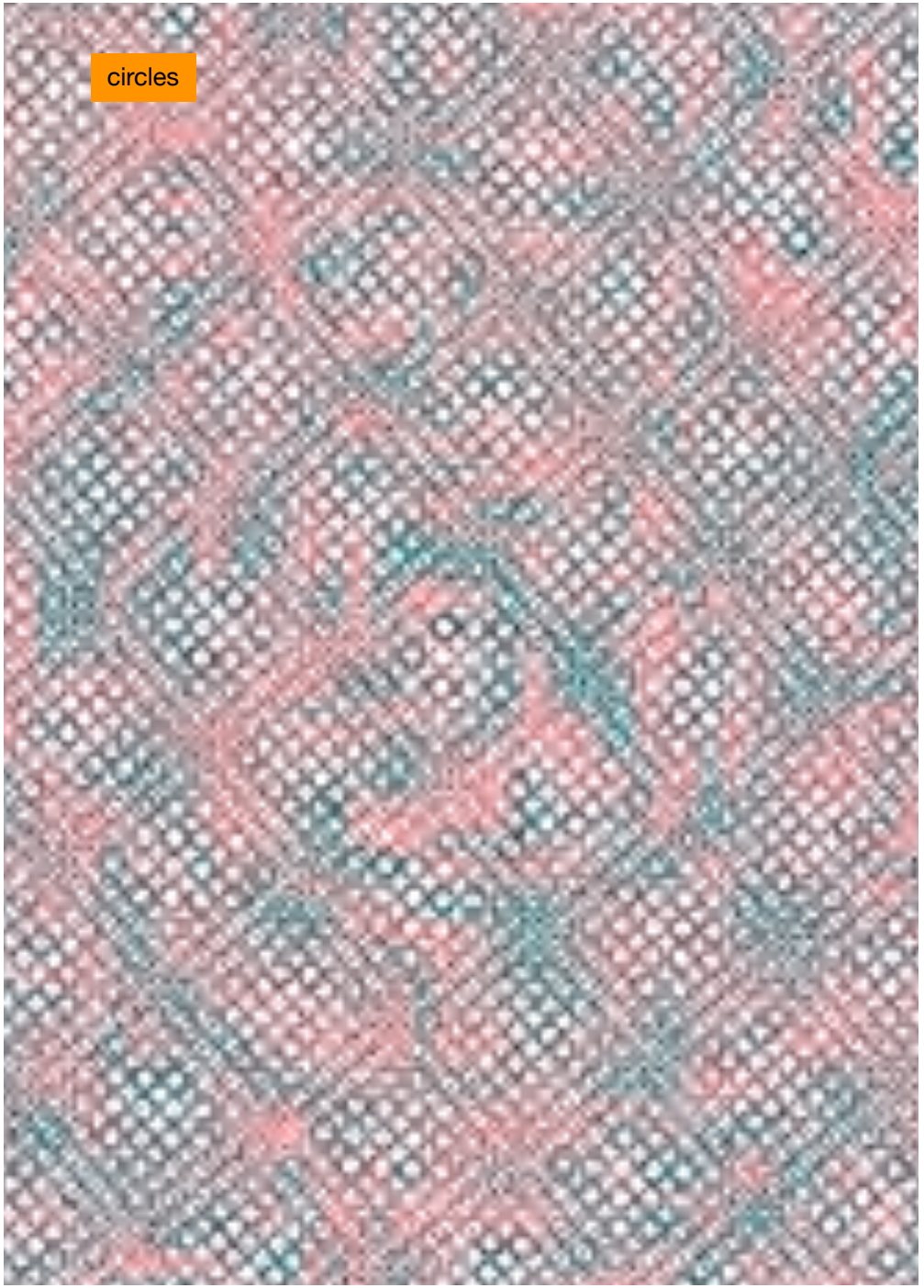
The combination of different shapes in different colours and sizes often produces surprising results[8].

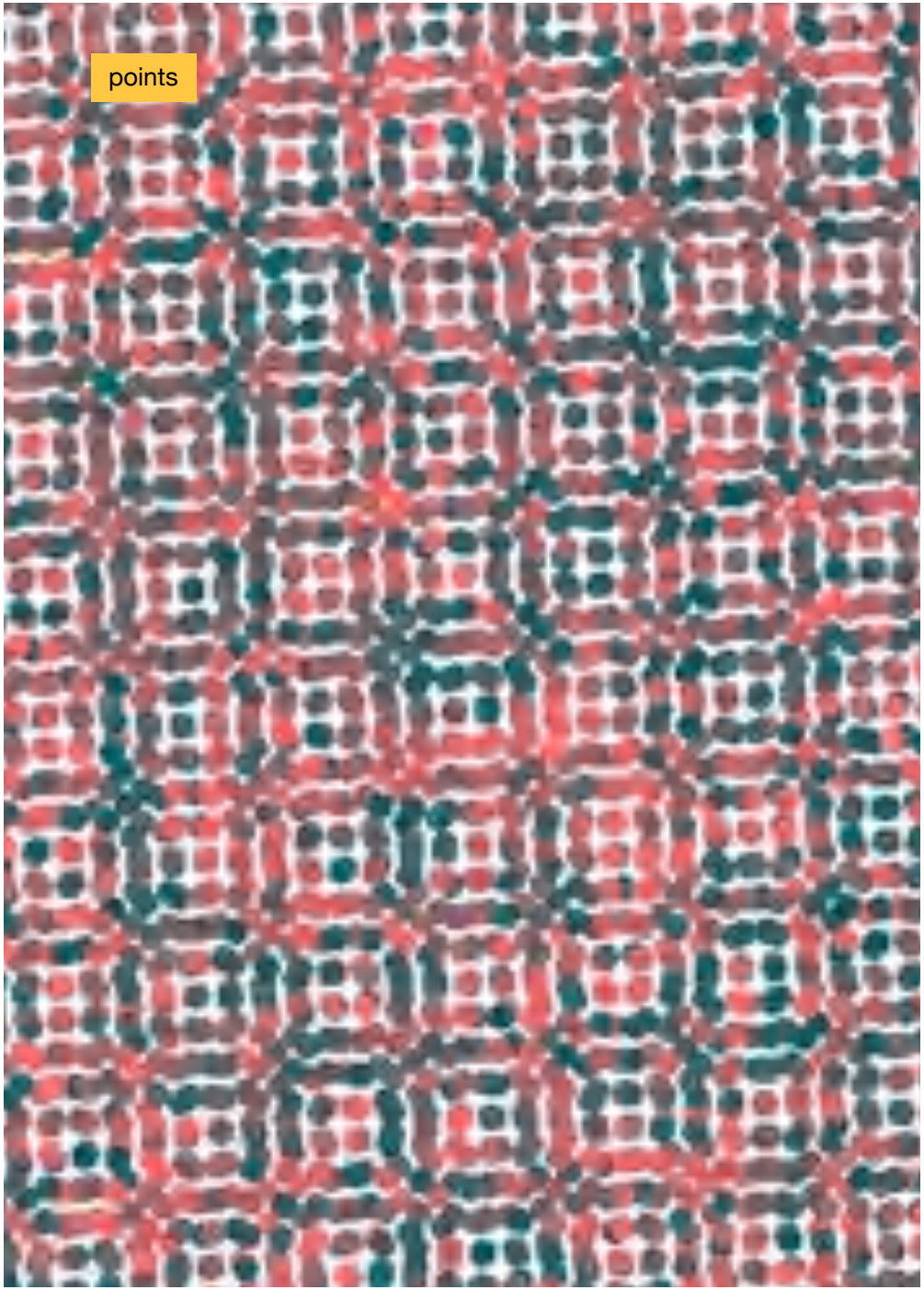
---

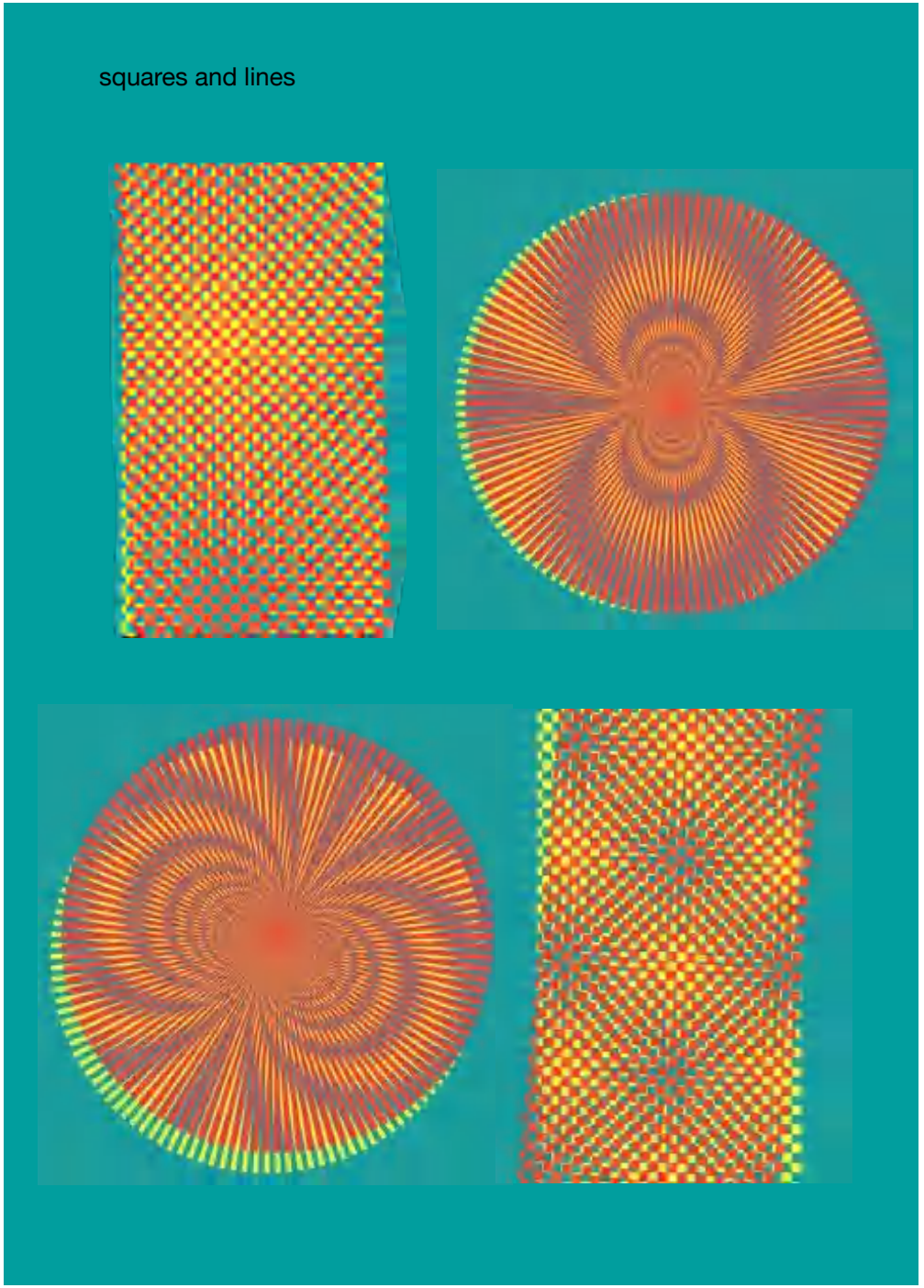
[7] Static images are already suitable for presentation; however, they are particularly appealing to viewers if they run as program-controlled animation or, ideally, can be influenced interactively themselves (by manually shifting the patterns).

[8] With the moiré index, Carsten Nicolai has presented an entire book in which he carries out precisely such systematic transformations of basic elements in grids. Pretty much anything can be recoded according to the principle presented here.

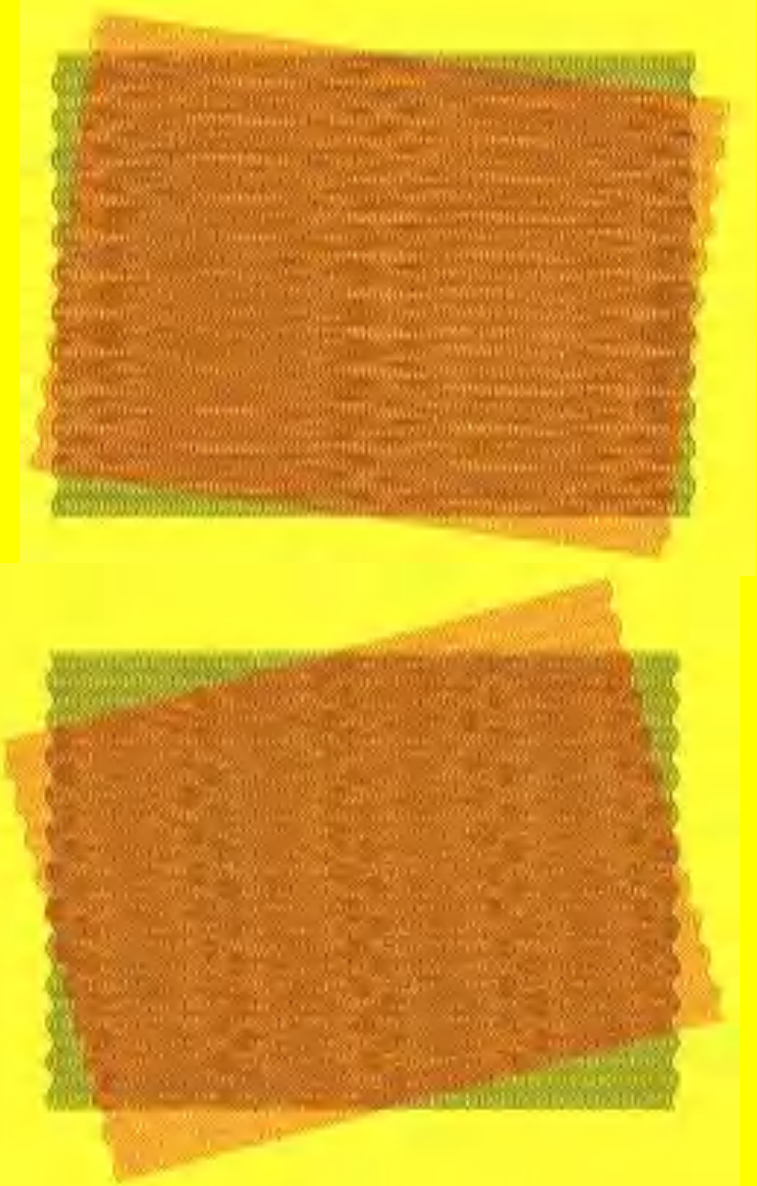


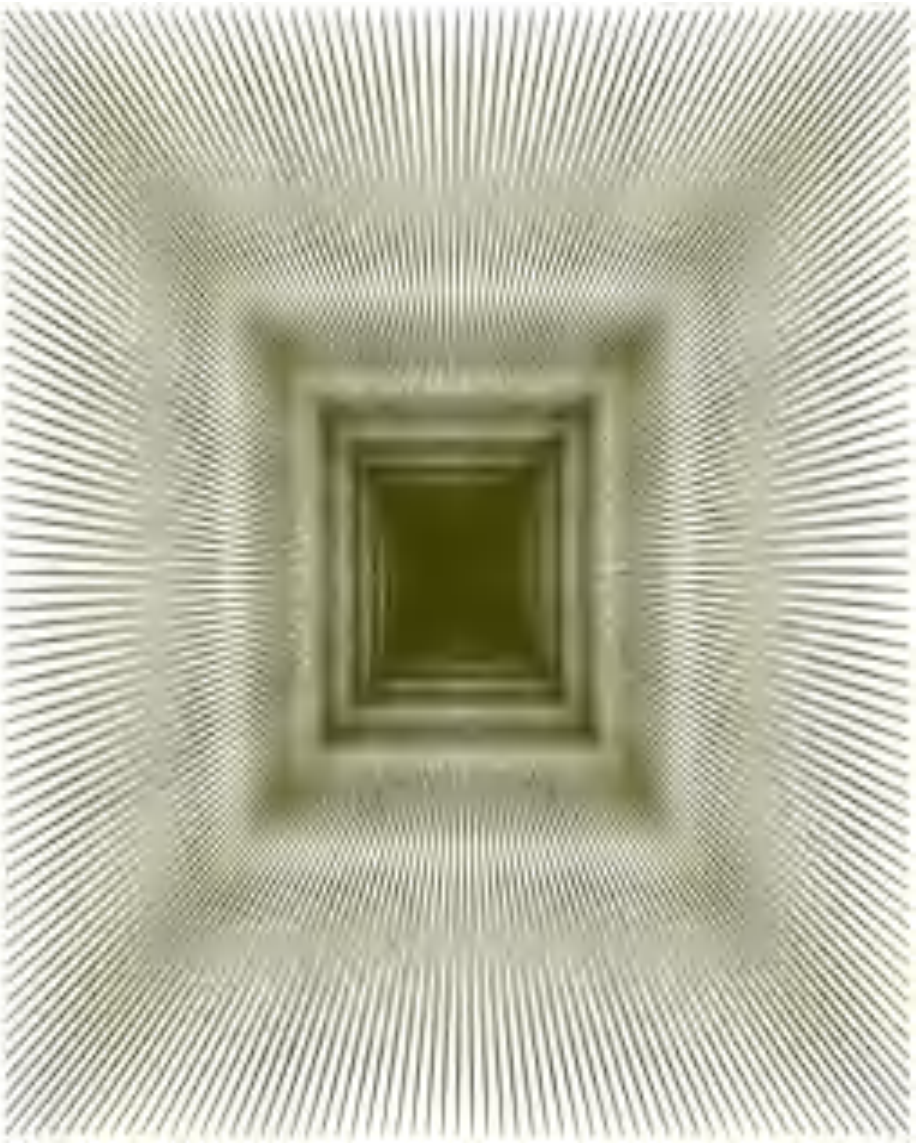

circles


points

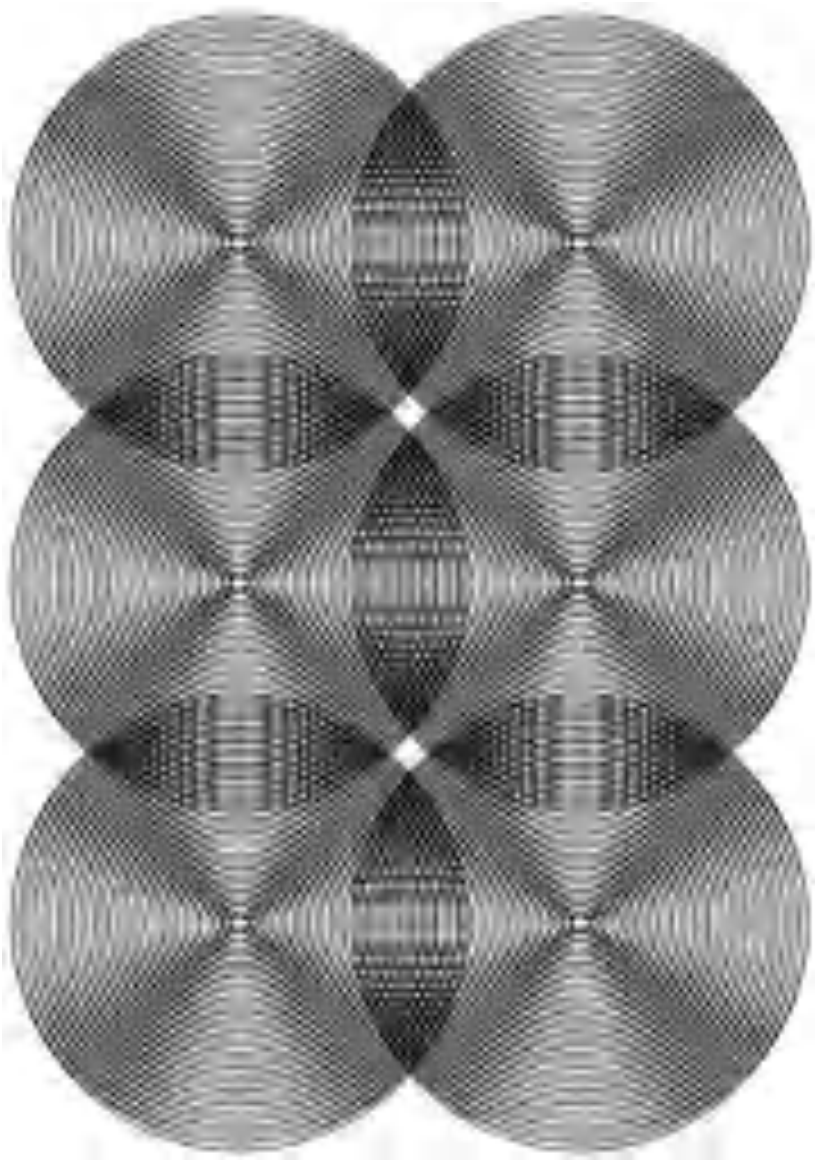

squares and lines

arcs





Homage à Biasi: Lot Nr. 639

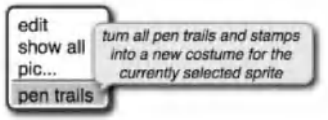


Homage à Oxenaar: 45 Cent Stamp

Stamping instead of drawing

There are useful functions in the programming environment Snap! that can greatly simplify the drawing of complex graphics. In all previous projects, for example, the graphic elements were created by moving the turtle with the pencil down. However, it is also possible to make the turtle - in the form of its **costume** - into a graphic element itself.

For each graphic created, a context menu can be opened by **right-clicking** on the **stage**, in which such a costume is created from the graphic by selecting the option **pen trails**.

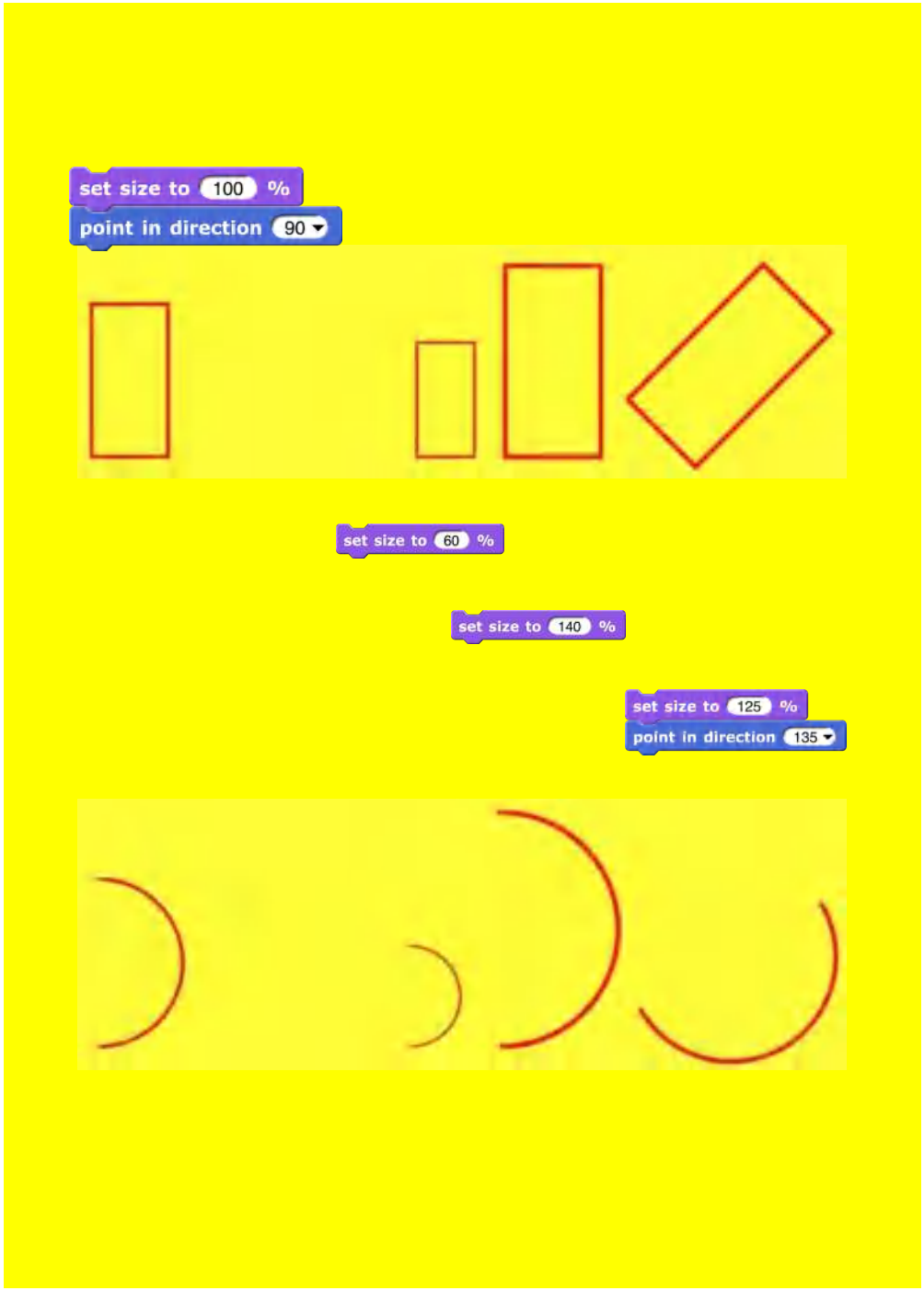


A costume list can be found in the program area under the tab **Costumes**.

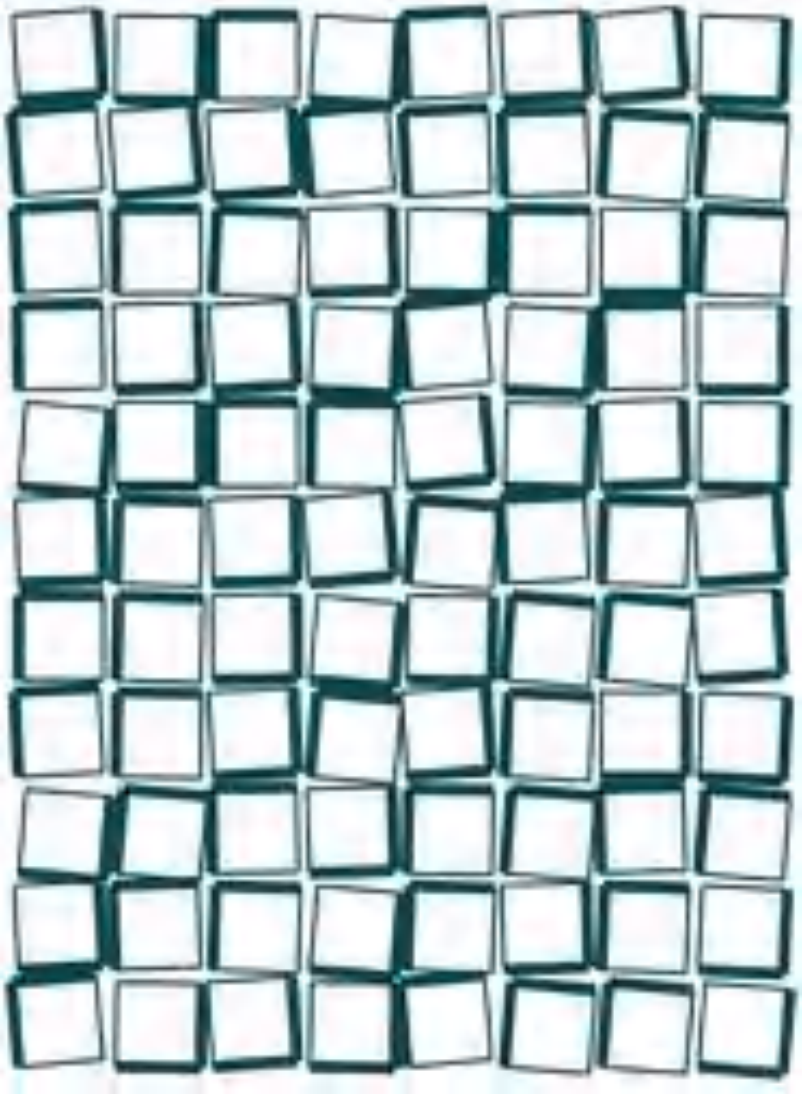
The **Looks** category provides commands for customizing the costumes.The command **switch to costume** replaces the current turtle costume with the selected costume. **next costume** replaces the current costume with the next costume from the **Costumes** list. By **set size to x %**, the size of the costume can be changed in percent, with **change size by x** by x pixels.

The important thing now is that with **stamp** the selected costume can be "stamped" at the current position of the turtle. It remains there, even if the turtle is moved further.

This makes it possible to multiply complex graphics more easily and significantly faster than with the corresponding procedures in **repeat**-loops. The following pictures can show this as an example.

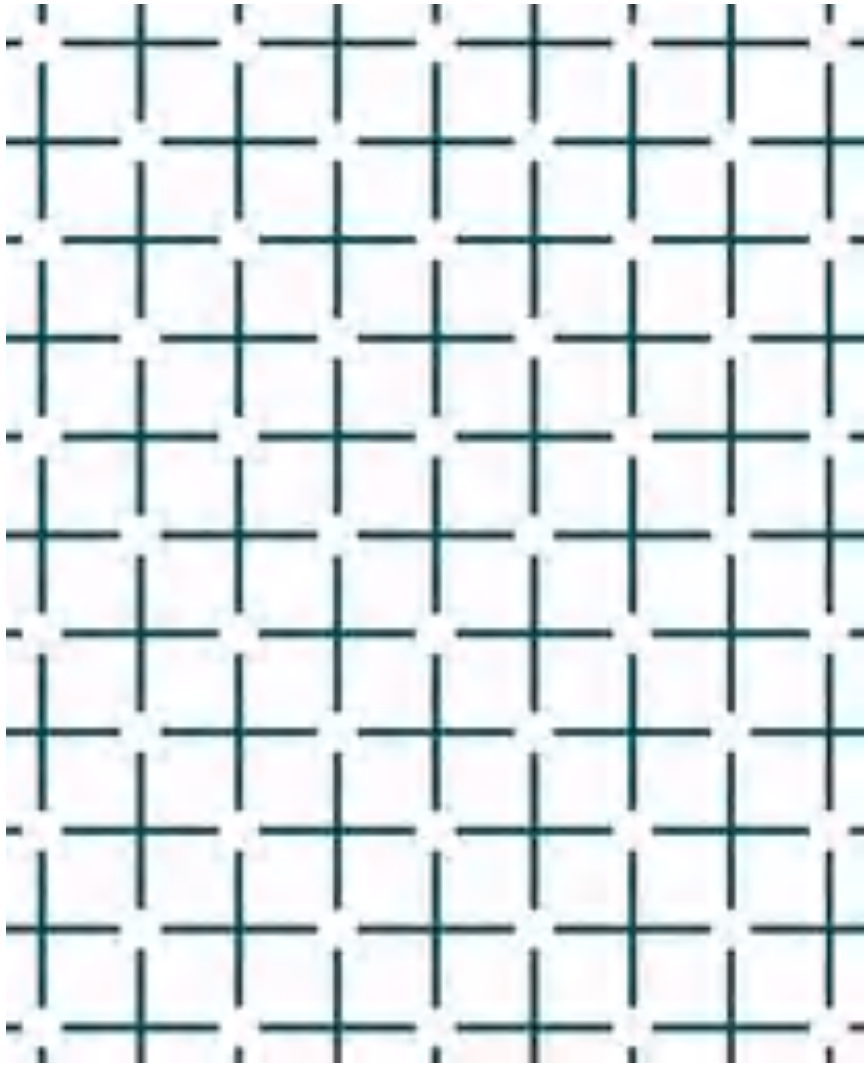


squares



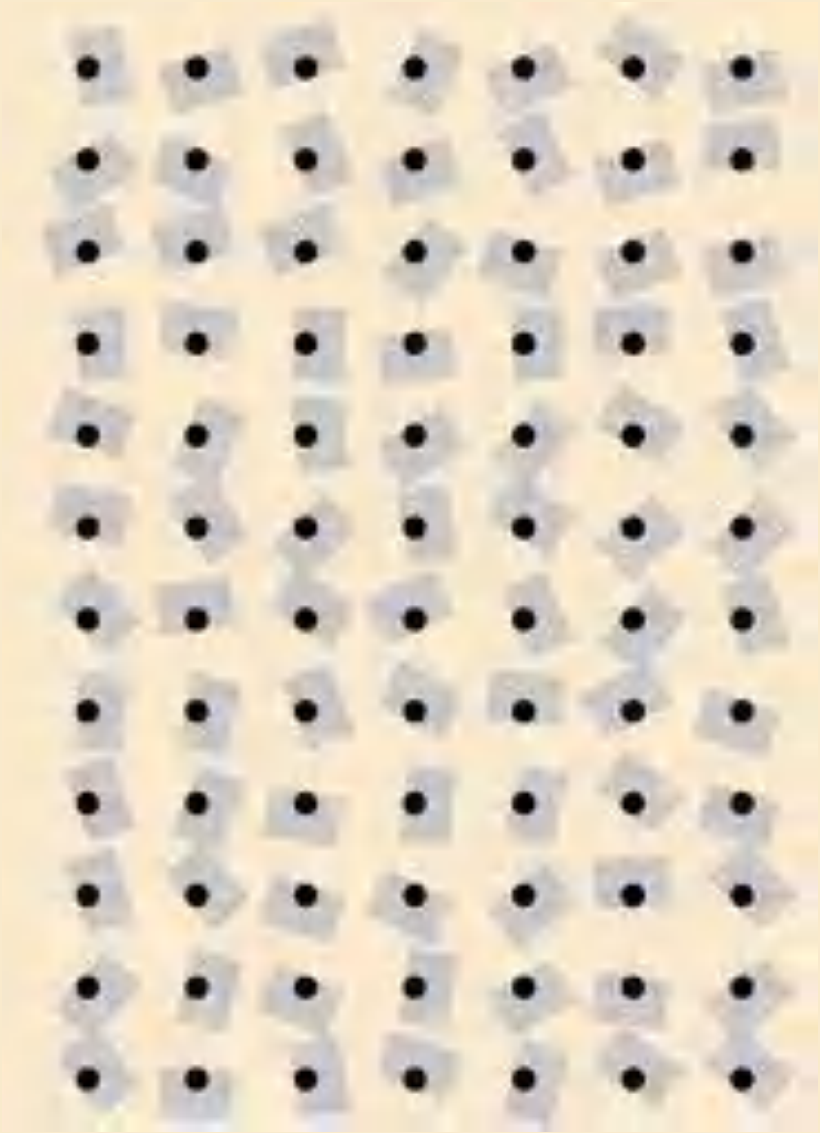
Homage à Mohr: P-105

lines



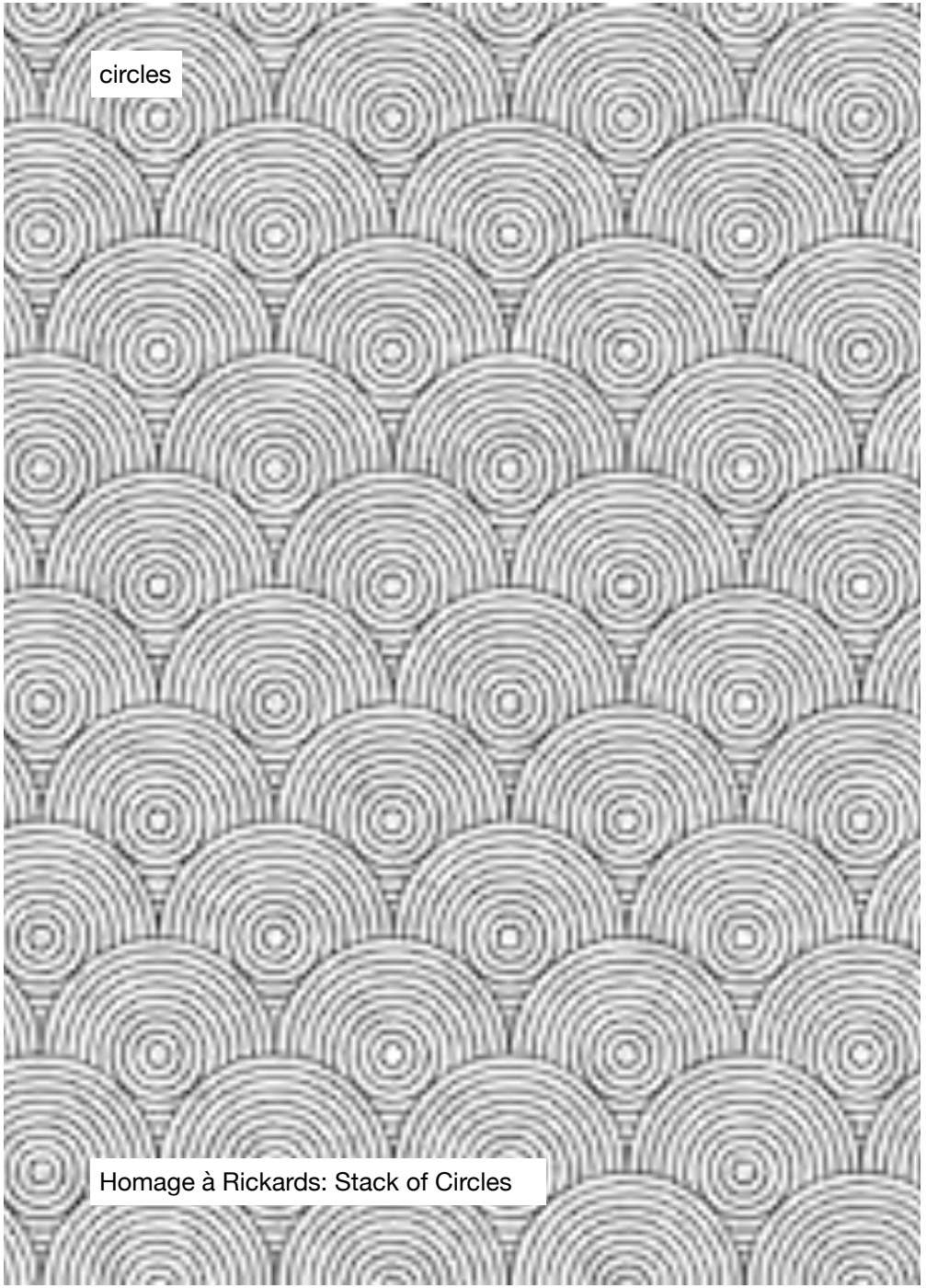
Opticals: Ehrenstein-illusion

lines and points



Homage à Le Parc: Rotations

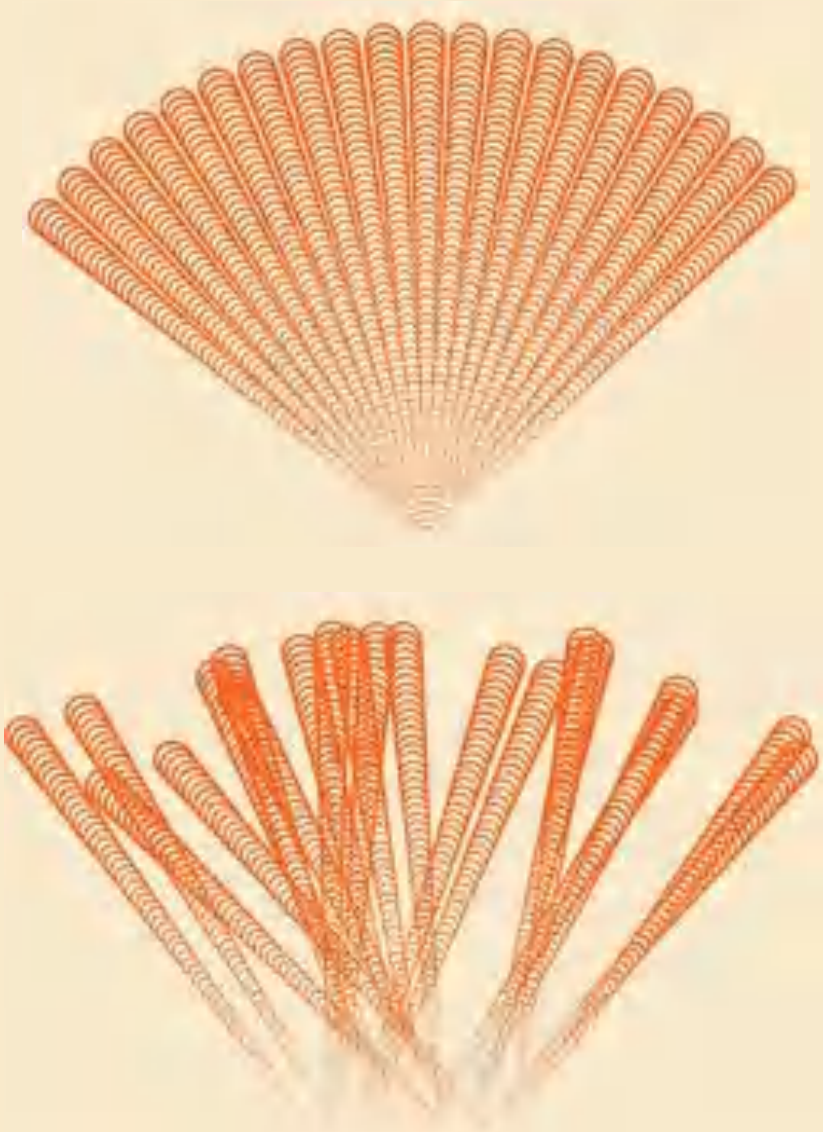
Homage à Le Parc: Dots

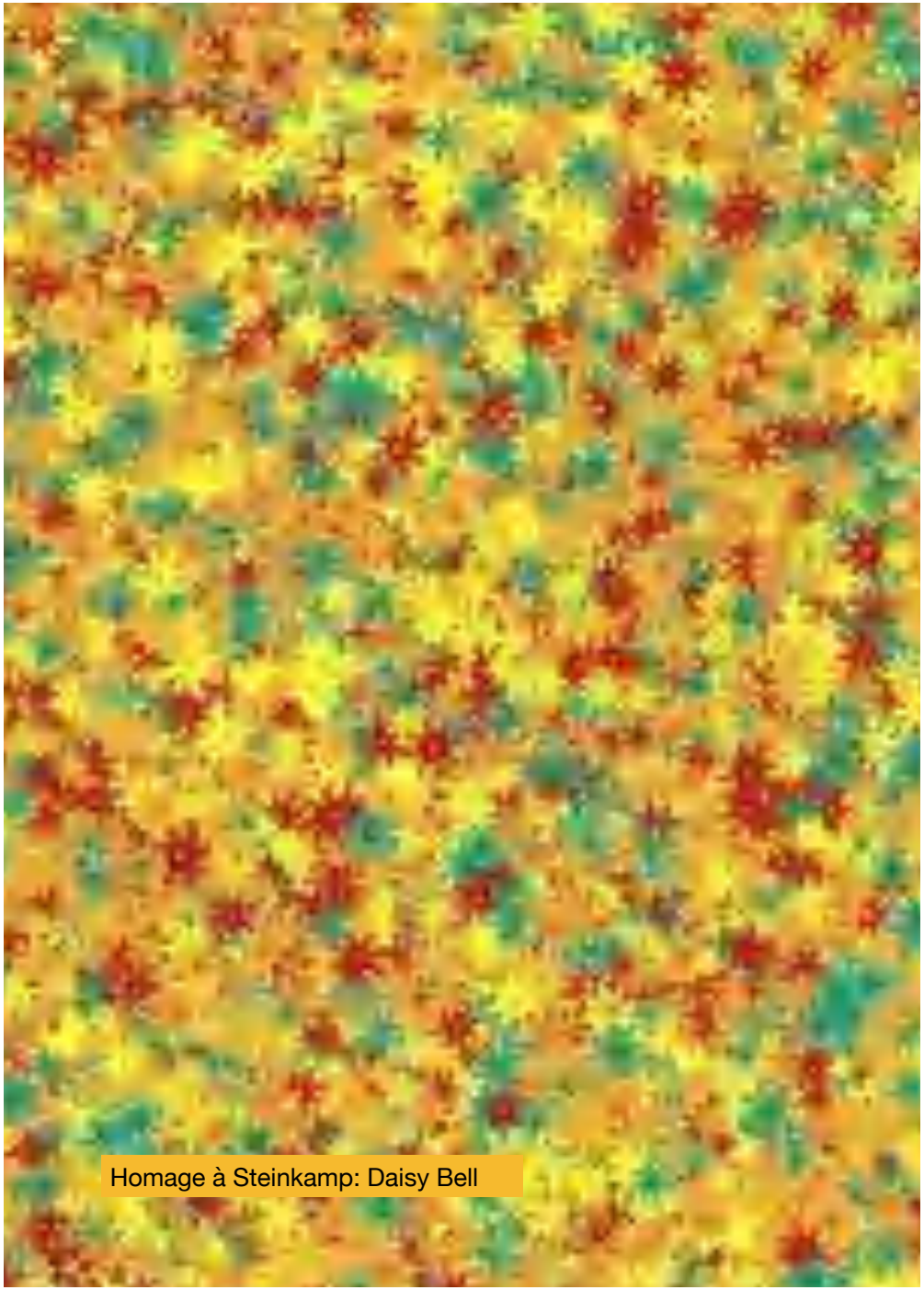


Homage à Rickards: Stack of Circles

rectangles

arcs





Homage à Steinkamp: Daisy Bell

Outlook: Animation and Interaction

You can find the complete code of the examples on the website Logo Classics. There, a click on the **program name** leads directly to the programs in the Snap! programming environment.

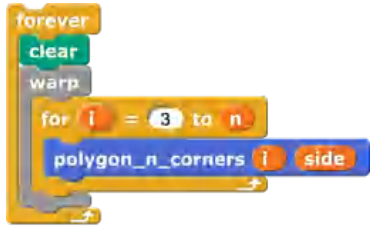
Some of the programs differ somewhat from the sequence shown in the text. This is because I have prepared the programs - as far as reasonable - for animation. Furthermore these programs are interactive, i.e. they can be controlled very easily by the user. The procedure is always the same and easy to understand:

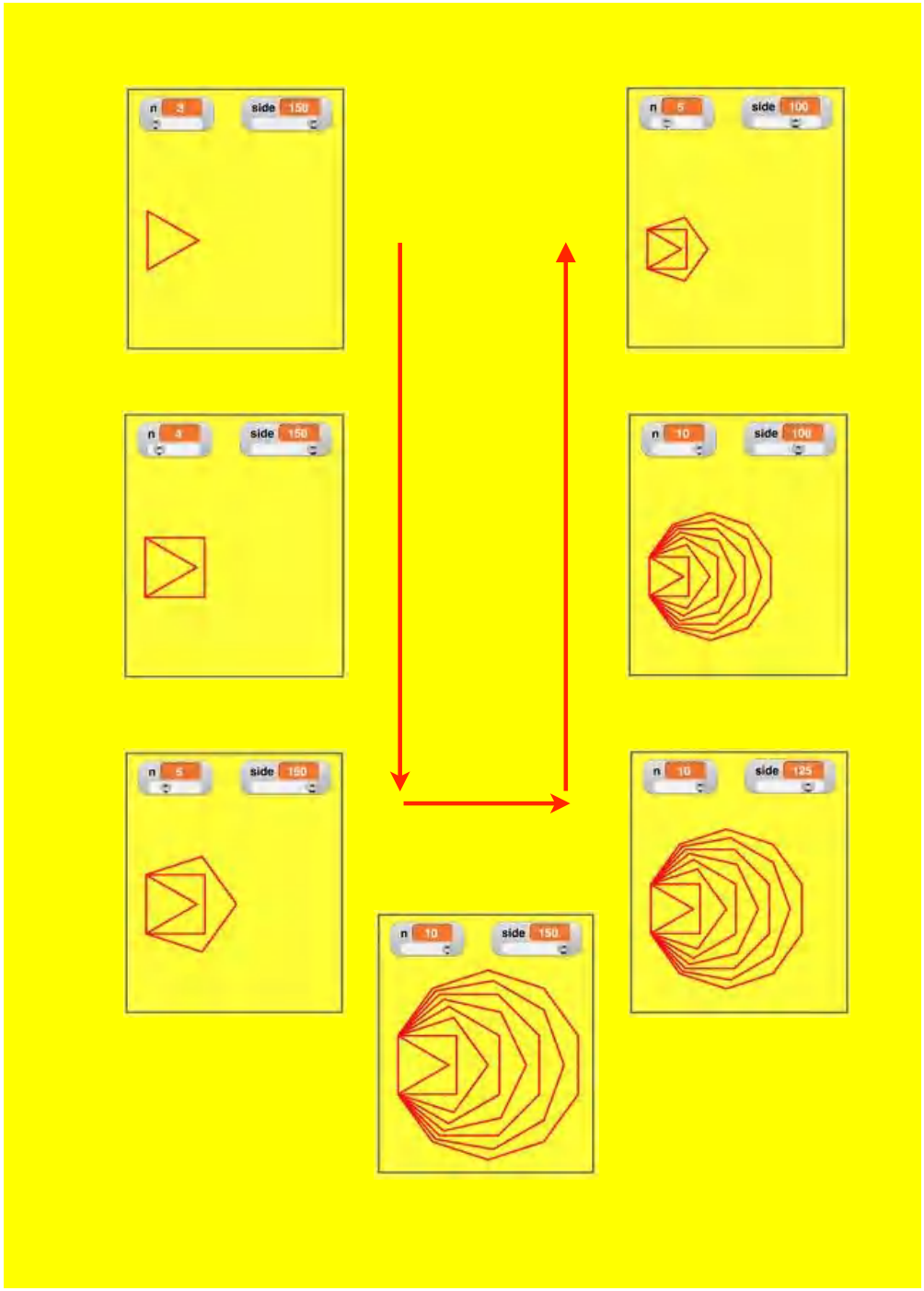
The animation principle is very simple and corresponds to the traditional stop-motion technique. Each image is drawn by the computer. The current image is continuously replaced by a new image with updated properties (such as color, length, angle, etc.). So it is the principle of **paint - wipe - paint ...** If this happens fast enough, we get a flicker-free, animated image.

The animation takes place in an endless loop (**forever**). First the stage is erased (**clear**), followed by the **commands required to draw the picture**. The **warp** command is necessary because it ensures that the graphic output only takes place when the image internally is completely drawn. This makes the image output more fluid!

Sliders are provided for the characteristic values so that the viewers can control the images interactively. If values are changed, the image is updated with these values during the next repeat loop. To avoid undesired effects, sensible **minimum** and **maximum values** should be preset for the parameters.

In the concrete example on the right-hand side, the number of corners and their side length are set dynamically for the polygons (which of course can only be displayed statically here).




n 3
side 150
n 5
side 100
n 4
side 150
n 10
side 100
n 5
side 150
n 10
side 125
n 10
side 150

Literature

Abelson, H. (1982). Logo for the Apple][. New York: McGraw-Hill.

Eyferth, K., Fischer, K., Kling, U., Korte, W., Laubsch, J., Löthe, H., Schmidt, R., Schulte, H. & Werkhofer, K. (1974). Computer im Unterricht. Formen, Erfolge und Grenzen einer Lerntechnologie in der Schule. Stuttgart: Klett.

Feurzeig, W. (2010). Toward a Culture of Creativity: A Personal Perspective on Logo's Early Years and Ongoing Potential. International Journal of Computers for Mathematical Learning, Vol. 15, 3, pp. 257-265.

Harvey, B. (1997). Computer Science Logo Style. V. 1: Symbolic Computing. Cambridge: MIT Press. (Download des Buches: https://people.eecs.berkeley.edu/~bh/v1-toc2.html)

Nicolai, C. (2010). moiré index. Berlin: Die Gestalten Verlag.

Papert, S. (1980). Mindstorms: children, computers, and powerful ideas. New York: Basic Books. (Download: http://worrydream.com/refs/Papert%20-%20Mindstorms%201st%20ed.pdf)

Schattschneider, D. (1990). Visions of Symmetry. New York: Freeman and Company,

Photo credits:

Book cover Mindstorms (p. 1): Photo J. Wedekind

Floor turtle (p. 1): http://cyberneticzoo.com/cyberneticanimals/1969-the-logo-turtle-seymour-papert-marvin-minsky-et-al-american/

All other pictures are screenshots from the respective programs or graphics generated with them.

Turtle geometry is a different style of doing geometry, just as Euclid's axiomatic style and Descartes's analytic style are different from each other. Euclid's is a logical style, Descartes's is an algebraic style. Turtle geometry is a computational style of geometry. [Papert, 1980]

This book traverses via typical hoops covered with turtle graphics. They can often be found in early publications on logo programs and introductory courses but they develop their own aesthetics.

Using code examples in the visual programming environment Processing, the creation of these pictures can be easily understood. Those who enjoy abstract generative graphics will find a variety of suggestions on how to create appealing images using simple programming techniques.