The tangled allure of Recursion

Gian Marco Todesco
Gianmarco.todesco@gmail.com
Digital Video s.r.l.

One of the oldest algorithms we know, which is still in use, allows computing the greatest common divisor (GCD) of two numbers; it is attributed to Euclid, who describes it in his "*Elements*" (c. 300 BC). It can be expressed in this way:

- if the two numbers are equal, then their GCD is the shared value;
- if the numbers are different, then their GCD is equal to the GCD of the smallest number and their difference.

A noticeable feature of this method is its apparent circularity: to calculate the GCD of two numbers, you must calculate the GCD of two other numbers.
The procedure works (and is very effective) because the numbers shrink at each step while remaining positive, so sooner or later, they will become equal, and the procedure will eventually terminate.
The algorithm (at least in its definition) uses *recursion*, an extraordinary and effective conceptual tool used in mathematics, computer science and logic. It is a difficult concept to master, and it continues to fascinate and torment generations of students. The following few pages are meant to be a sort of exploratory walk into the recursion world, with particular attention to its applications in the field of graphics.

Recursion is a form of self-reference that is central in human thought when reflecting on its mental mechanisms. It is not surprising that recursion catches the imagination and is often used playfully. For example, a search of “recursion” on Google returns the initial suggestion "*Did you mean: 'recursion'*": winking at the most attentive readers.
Other playful uses of recursion are recursive acronyms. They are acronyms whose first word is the acronym itself. The most notable example is Richard Stallman's *GNU* project. *GNU* is a free software operating system, and the acronym means “GNU is Not Unix” (Unix is a different operating system).

A book that torments and delights the reader with recursion is *Gödel, Escher, Bach: an Eternal Golden Braid* (1979), by Douglas Hofstadter. The book contains the first example I have met of a recursive acronym and one of the best. Achilles, a character in the book, asks a Genie for a special wish: he wants one hundred desires instead of only three. This kind of wish is a “meta-wish”, and the Genie cannot grant it alone. It intends to help Achilles, so it

forwards the wish to its *meta-Genie* that in turns has to ask its *meta-meta-Genie*, and so on. Each Genie refers to the whole infinite sequence of all the other Genies as *GOD*. The name is a recursive acronym meaning “GOD Over Djinn”. Expanding the first word of the sentence, one gets “GOD Over Djinn Over Djinn” and so on, generating a new Djinn at each step. A very effective name for the infinite sequence of Djins!

Another extraordinary creation in the book is the concept of “quine program” or simply “quine”. The name is a homage to the philosopher and logician Willard Van Orman Quine (1908-2000). For Hofstadter, a “quine” is a program that outputs its source code when run. The program must not have access to its code, but it has to “compute” it by itself.
Creating such a program is a fun challenge that requires a fair amount of programming skill. Such a program could be considered a metaphor for living creatures that can reproduce themselves together with the code (DNA) that describes them.

Tower of Hanoi

Besides the playful aspects, recursion proves to be a potent and effective tool. Recursive algorithms are often more compact and concise than the non-recursive equivalent, and sometimes the recursive approach can solve problems that seemed too hard with other methods. The GCD and even the factorial (i.e. the product of the first consecutive integers: a typical example in the introductive courses about recursion), while defined with recursion, are usually computed differently, in a more effective way. For many other problems, the recursive solution is almost not avoidable.
A classic example of this kind of problems is the “*Tower of Hanoi*” puzzle. The puzzle, also known as “*Tower of Brahma*” or “*Lucas’ Tower*”, was invented in the 19th century by the French mathematician Edouard Lucas. It features several disks of different radiuses stacked on one of three posts. The game’s goal is to move the stack to another post by following two rules: one can move only one disk at a time and not put a larger disk on top of a smaller one. The game does not seem too complex. Indeed, the solution with 3-4 discs can be found quite easily by trial and error. If we consider a higher number of disks, then the number of possible disk arrangements grows dramatically. Even the number of moves needed by a perfect player to solve the puzzle becomes quickly very large. Four disks require 15 moves (see fig.1), while 20 disks require more than one million.

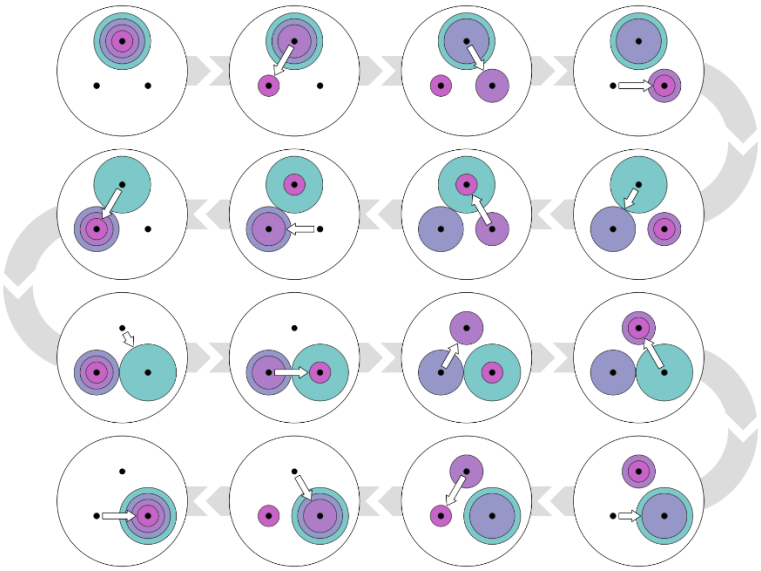

Figure 1 The shortest solution (15 steps) for the Tower of Hanoi puzzle with 4 disks

When trying to solve the puzzle, there are always three legal moves at each step. One option “goes back”, undoing the previous move, and can be discarded. The other two options require a non-trivial choice. Random choices make us wander in a labyrinth of different states from which it is difficult to escape. The correct solution requires many moves, but suboptimal solutions can require many, many more. Fig.2 shows an alternative solution for four disks. The alternative has no useless cycle: it visits each configuration of disks only once, but it requires 53 moves.

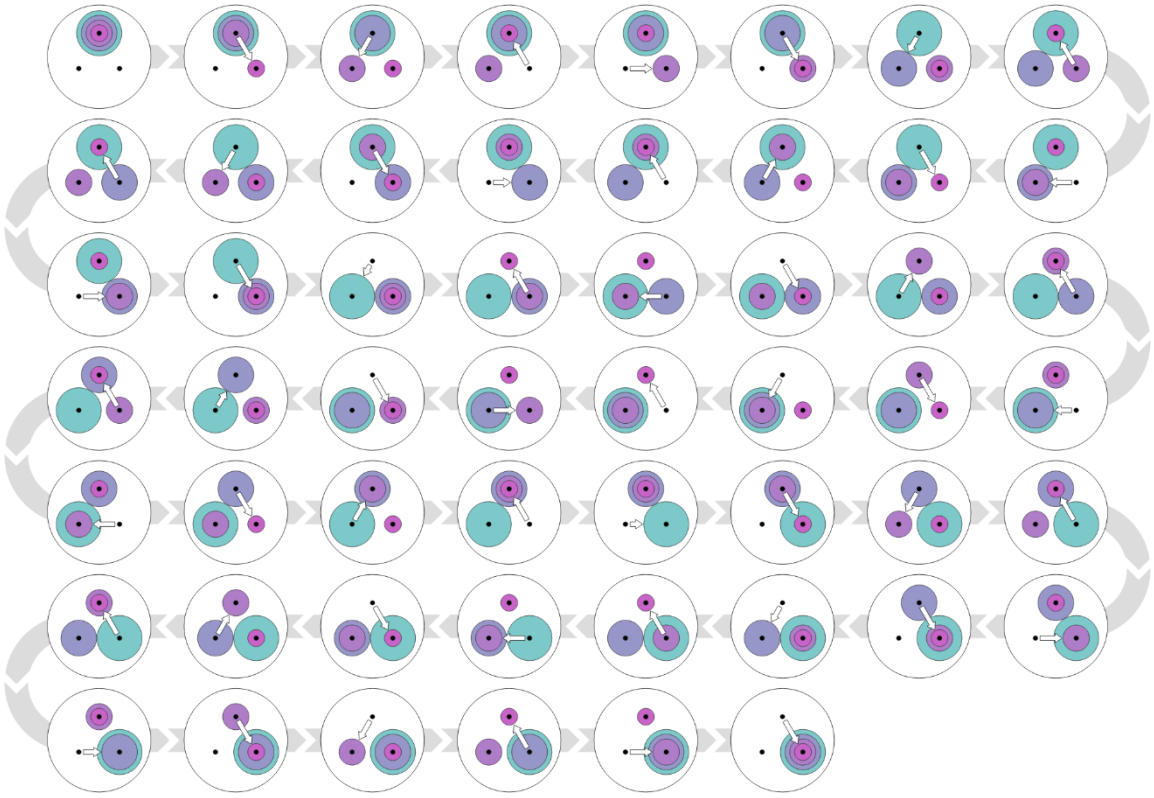

Figure 2 A non-optimal solution (53 steps). The solution, although inefficient, contains no repetitions.

Writing a program that finds the shortest solution is not a trivial task, and the recursive approach is very convenient. To tackle the problem recursively, we must define a straightforward case when the recursion stops (this is crucial to avoid a never-ending program). For the Hanoi Tower, the easy case is when we have just one disk. In this case, the solution is obvious, and it requires just one move.
To solve the general case (N disks, with N larger than one), we must pretend that we know how to solve the simpler configuration with one less disk. With this knowledge (that we assume to have), we move N-1 disks from the first post to the third one: the largest disk on the bottom can not interfere because the rules allow moving any other disk on top of the largest one as we wish. At this point, we are halfway: we transfer the largest disk from the first post to the second one, which is empty. Finally, we use the N-1 disks solution again and move the stack from the third post to the second, on top of the largest disk. This action completes the challenge.
During the actual solution, the N-1 subproblems are solved using the N-2 solution and so on, until we must solve the problem with N=1 that we solve directly, without the recursion.

Anagrams

The Hanoi Towers Problem seems (and probably is) meant for recursion, but the recursion is very useful even in more common problems. For example, we can use the recursion effectively to generate all the anagrams of a word. To make things easy, we assume that the

word contains all different letters. Moreover, we don't check the generated anagrams in the dictionary, i.e. we accept meaningless anagrams.
In the novel "*Il Pendolo di Foucault*" by Umberto Eco, Jacopo Belbo, one of the main characters, owns a personal computer that he loves very much and named Abulafia. He found on the computer manual a small program (written in the *BASIC* programming language) that can generate all the permutations of four letters. He realises that he can use the program to generate all the God name permutations (the God name he means is *IHVH*; Apparently, Belbo does not note or does not care that the pair of H's will generate many duplicates).
The novel contains the factual listing of the program (15 lines). Inserting a computer program source code in a fictional book is an intriguing idea. There is a nice contrast between Belbo with his computer program and Diotallevi, Belbo's friend who studies the Kabbalah. He cannot understand the logic of the program and finds it "kabbalistic".
From a programming point of view, the code is not too lousy. It even uses a nice trick to place the last letter, but it has some serious flaws. The worst one is that it can work only with four-letter words. It can generate all the anagrams of "math", but to generate all the anagrams of "image", we should completely rewrite the program.

Writing a program that generates all the permutations of any number of letters is not so easy if we want to start from scratch without using any predefined function (today, many programming languages have predefined library functions to generate the permutations). If we're going to try, then recursion is a perfect approach.
The seed of recursion (the "straightforward case") is when we consider a word with a single letter. In this case, there is only one anagram: the letter itself.
Let us consider a word with more than one letter. We can take out the first letter and generate all the anagrams of the rest (the rest has one less character, and we pretend to know how to solve this easier problem). Then, we insert the first letter of the original word in each possible position into each generated anagram.
This procedure will generate all the anagrams of a word, regardless of the number of letters.
Fig. 3 shows a program (written in the Python programming language) that produces all the anagrams of "image" recursively. (One of the anagrams is, with some punctuation added: "*A gem: I!*". It is a decent anagram that is referring enthusiastically and recursively to itself!).

```
def anagrams(word):
        m = len(word)
        if m == 1:
                yield word
        else:
                for s in anagrams(word[1:]):
                        for i in range(m+1):
                                yield s[:i] + word[:1] + s[i:]

for s in anagrams('image'): print(s)
```

Figure 3 A Python program that generates all the anagrams of the word 'image'

Visual recursion

The world of images, from paintings to computer graphics, is very suitable for the recursive approach.

The basic idea is to create an image that contains one or more replicas of itself, possibly with some changes. In Western art, this technique is called *Mise en abyme*. The technique is also known as the *Droste effect*. This last name comes from a Dutch brand of cocoa (see fig. 4). The image contains a smaller replica that includes an even smaller one and so on. That implies an infinite sequence.

Figure 4 The original 1904 Droste cacao tin, designed by Jan Misset (1861–1931)

The Dutch artist Maurits Cornelis Escher used this technique with a fascinating twist. In its *Print Gallery* (1956), he creates a stunning vortex that merges the larger image seamlessly with its smaller replica.

Bart de Smit of Leiden University led a group that analysed the image thoroughly from a mathematical point of view. Today the effect is readily available in many image editing software. The image in fig. 5 has been created with *GIMP* (GNU Image Manipulation Program) with the *MathMap* plugin.

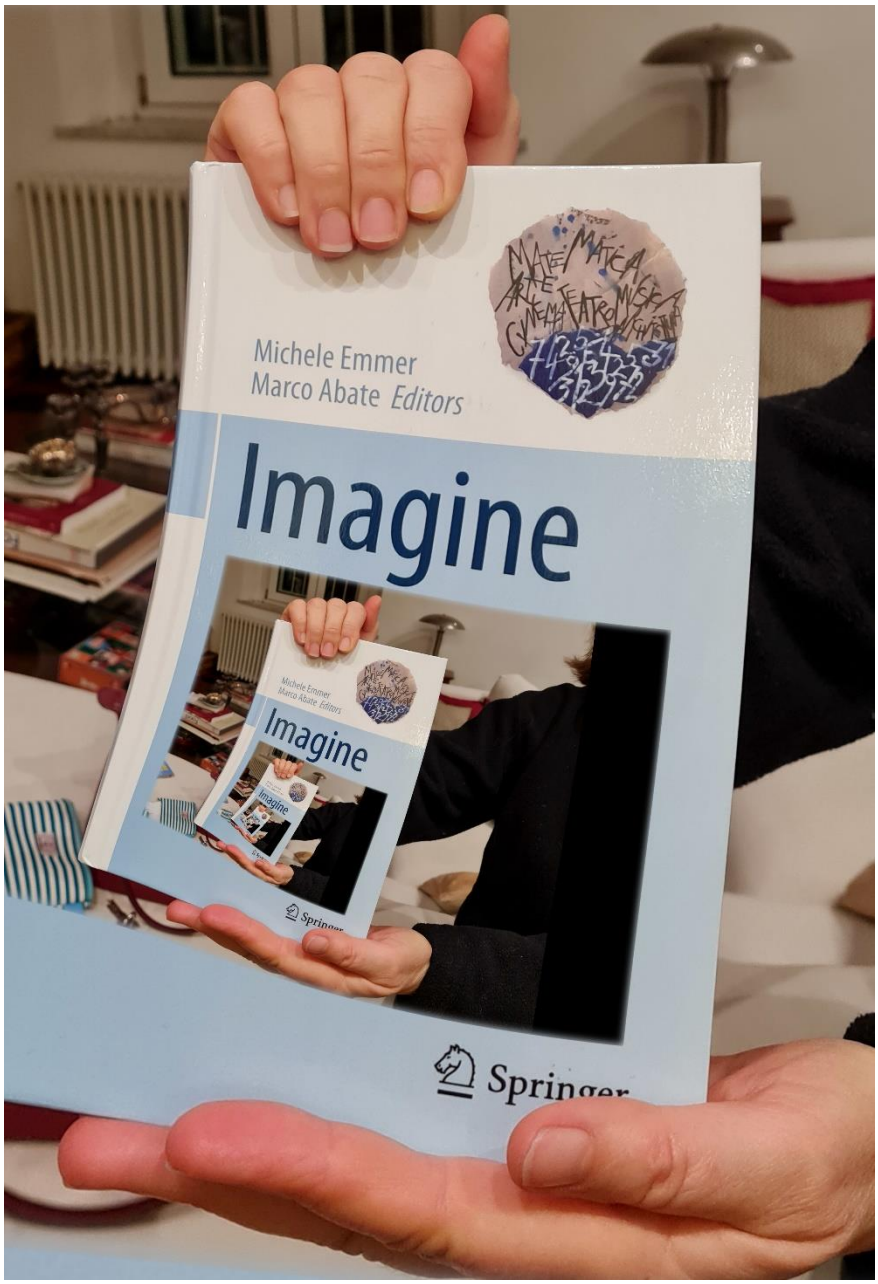

Figure 5 The "Droste Effect" in action.

Recursive drawings can be fascinating even when made of simple graphics primitives: e.g. simple straight segments. The dragon curve (*Heighway dragon*) is a remarkable geometric object with many interesting properties. It can be defined with the following recursive procedure: start with a simple segment connecting two points. Then replace the segment with two shorter segments of equal length, meeting at a right angle. The old segment must be the hypotenuse of an isosceles right triangle, while the two new segments must be the

catheti. At each recursive step, apply this procedure to each segment. The new segments must lie alternatively on the left and the right of the old segments.
Fig. 6 shows the first steps. Fig 7 is the result after 14 iterations.

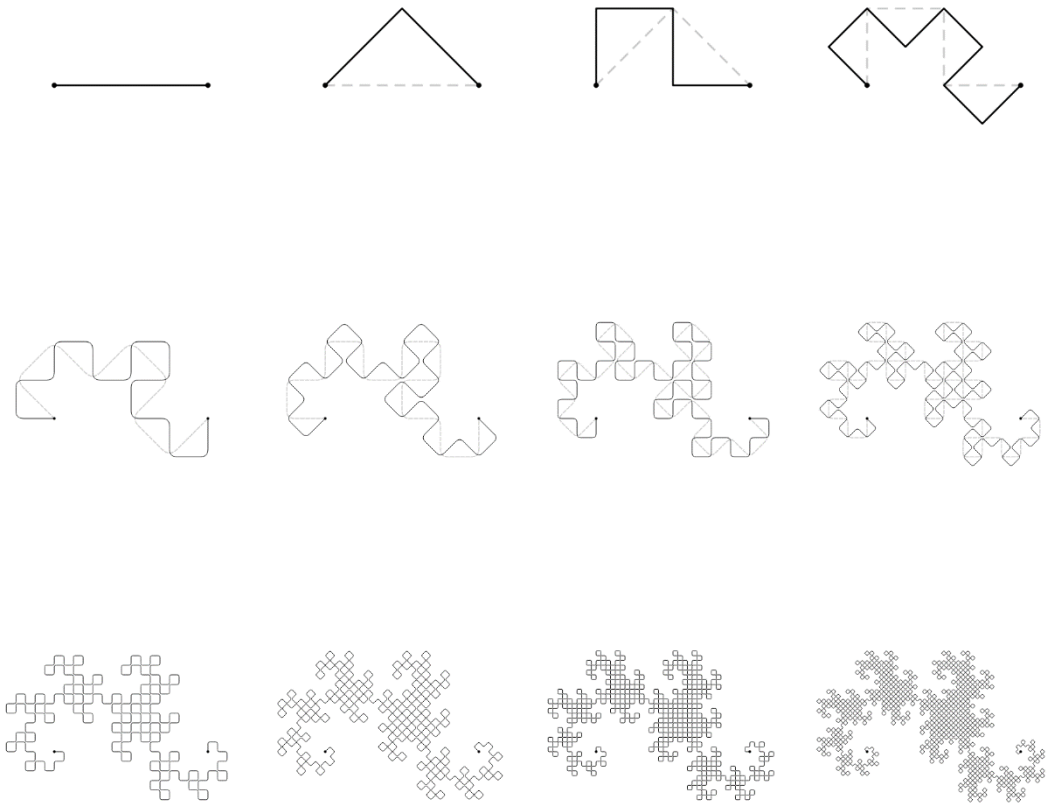

Figure 6 First steps of the generation of the Heighway Dragon Curve

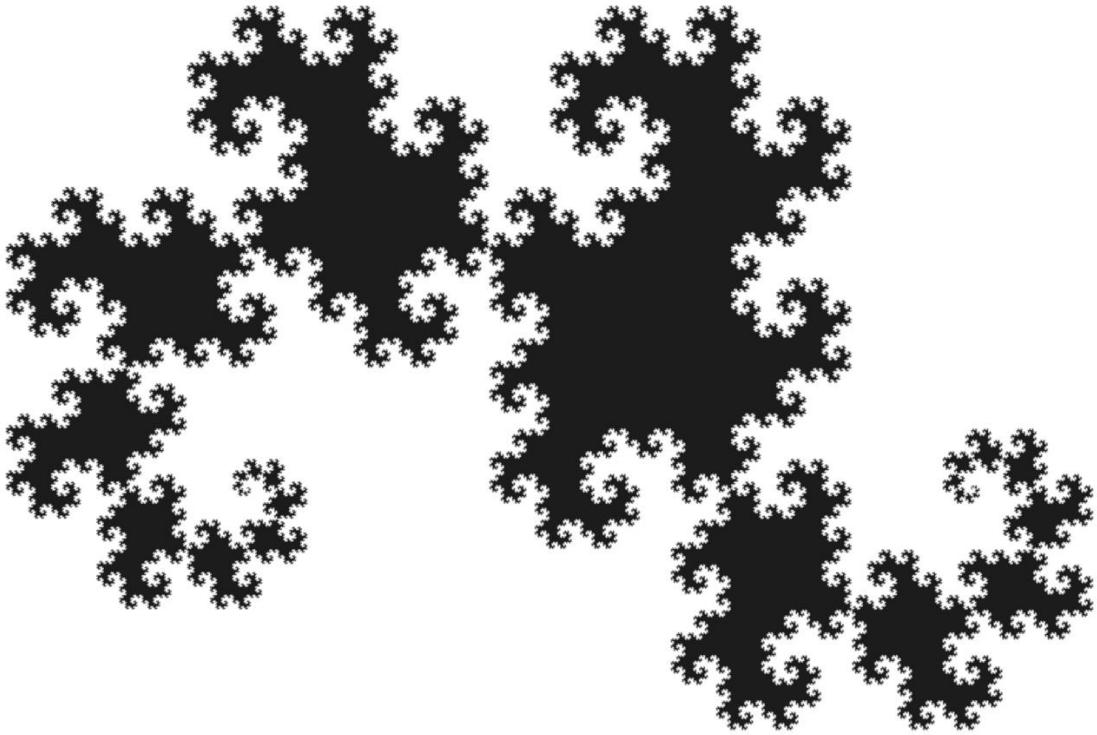

Figure 7 Level-20 Dragon Curve

The dragon curve is the limit approached as the above steps are followed indefinitely.
The curve contains infinite replicas of itself, rotated 45° respect to each other, and with a scale factor of$\sqrt{2}$.
It is possible to create a model of the dragon curve by folding a strip of paper. We cut a long and narrow strip of thin cardboard, e.g. a strip one cm wide and 20 cm long. We fold the strip in half, bringing the right edge to the left edge. Then we repeat the folding four times, always folding in the same direction. We then unfold the strip carefully so that each fold is at a right angle. To build a larger model is better to fold different strips and tape them together.

It is easy to draw the curve with coding. Fig. 8 shows an example that runs in *Google Chrome*, the Internet Browser. Run the browser and then open the console: activate the Chrome Menu in the upper-right-hand corner of the browser window and select *More Tools* > *Developer Tools*. You can also use the shortcut Option + ⌘ + J (on macOS), or Shift + CTRL + J (on Windows/Linux). Copy very carefully the text in Fig. 8 into the console.
Press *return* and the browser will show the 13th iteration of the dragon curve.
Write *dragon(7)* and then press *return* to generate the 7th iteration and so on.

```
function dragon(L) {
    let draw = (L,a,b,c,d) => {
        let e=(a+c-d+b)/2, f=(b+d+c-a)/2; return L ?
            draw(L-1,a,b,e,f)+draw(L-1,c,d,e,f) :
            "<line stroke='teal' x1='"+a+"' y1='"+b+"' x2='"+c+"' y2='"+d+"'/>";
    }
    document.body.innerHTML = "<svg width='800' height='500'>"+draw(L,200,180,650,180)+"</svg>";
} dragon(13)
```

Figure 8 A JavaScript code to draw the Dragon Curve in a Internet Browser.

At the limit, the curve fills a whole region in the plane: it touches every point included in its boundary. A bizarre behaviour for a one-dimensional entity such as a curve!
It is possible to put two identical curves next to each other: they will share part of their boundary with a perfect match. The dragon curve can even tessellate the whole plane (see fig. 9).

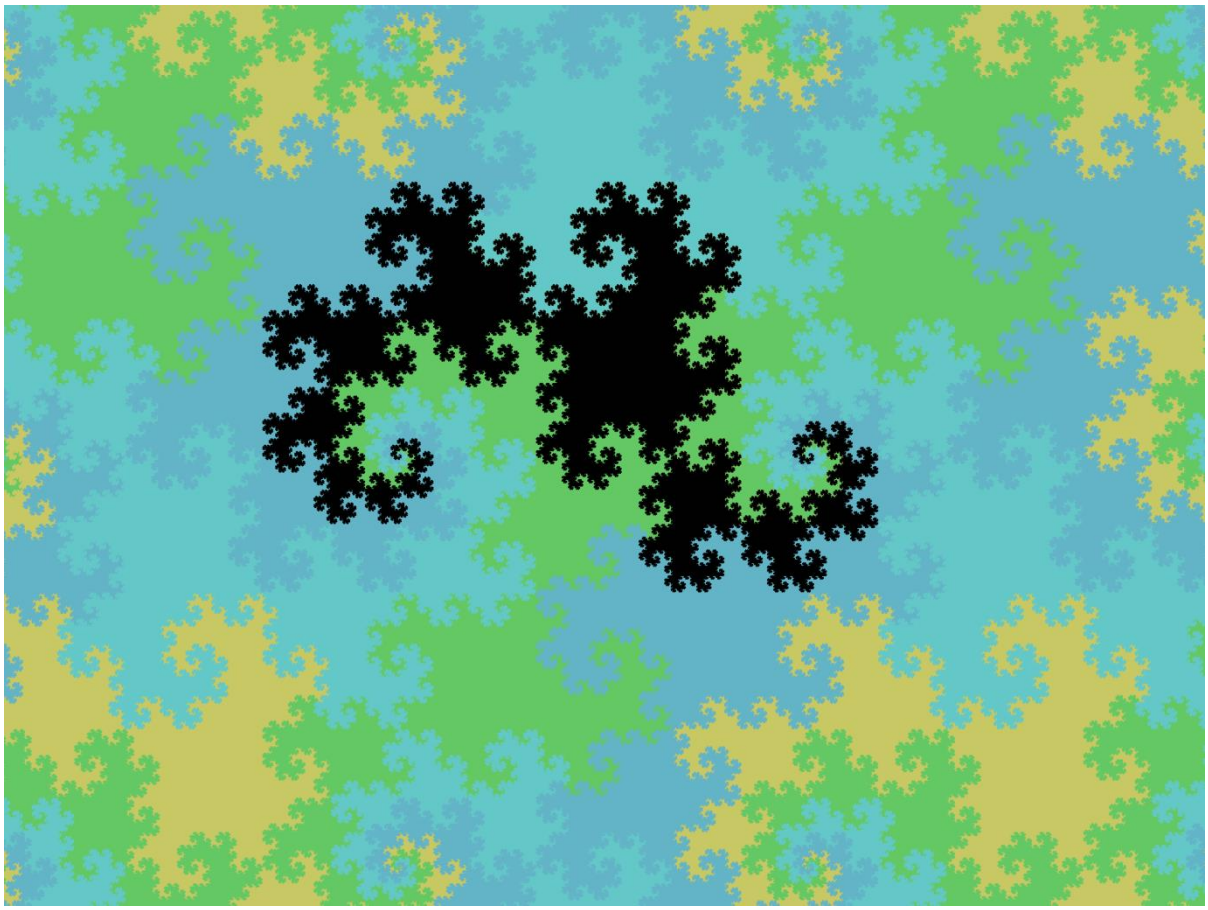

Figure 9 Many copies of the Dragon Curve can tessellate the whole plane

The first curve that passes through every point of a 2-dimensional region was discovered in 1890 by Italian mathematician Giuseppe Peano. The Peano curve, as the dragon curve and most of the many other space-filling curves, is defined with a recursive process: the actual curve is the limit of an infinite sequence of increasingly detailed curves.

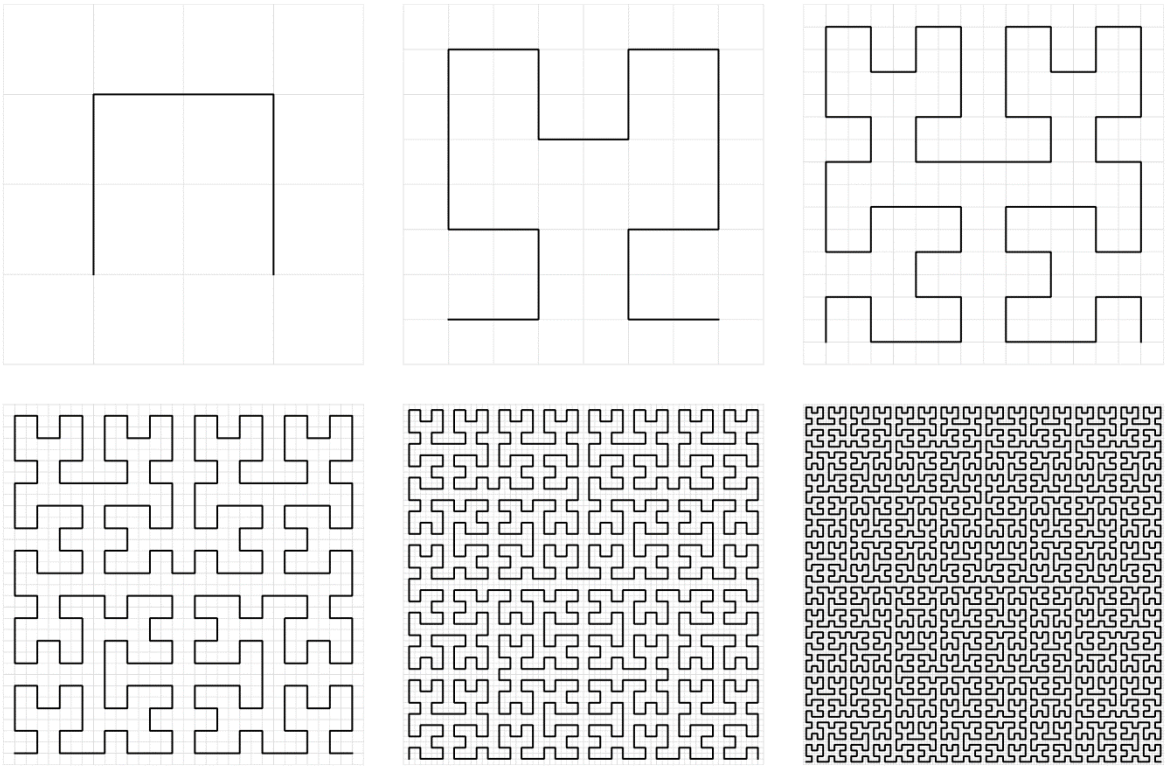

Figure 10 The first steps of the Hilbert Curve (NOTE: TO BE CHANGED WITH THE PEANO CURVE!!)

In fig. 10, there are the first steps of another example, created by David Hilbert in 1891. This last curve is well suited for creative use: the image sequence shows that the more refined the grid, the darker the image. We can modulate the darkness locally, just changing the depth of the recursion depending on the position. Fig. 11 shows the result of this experiment.

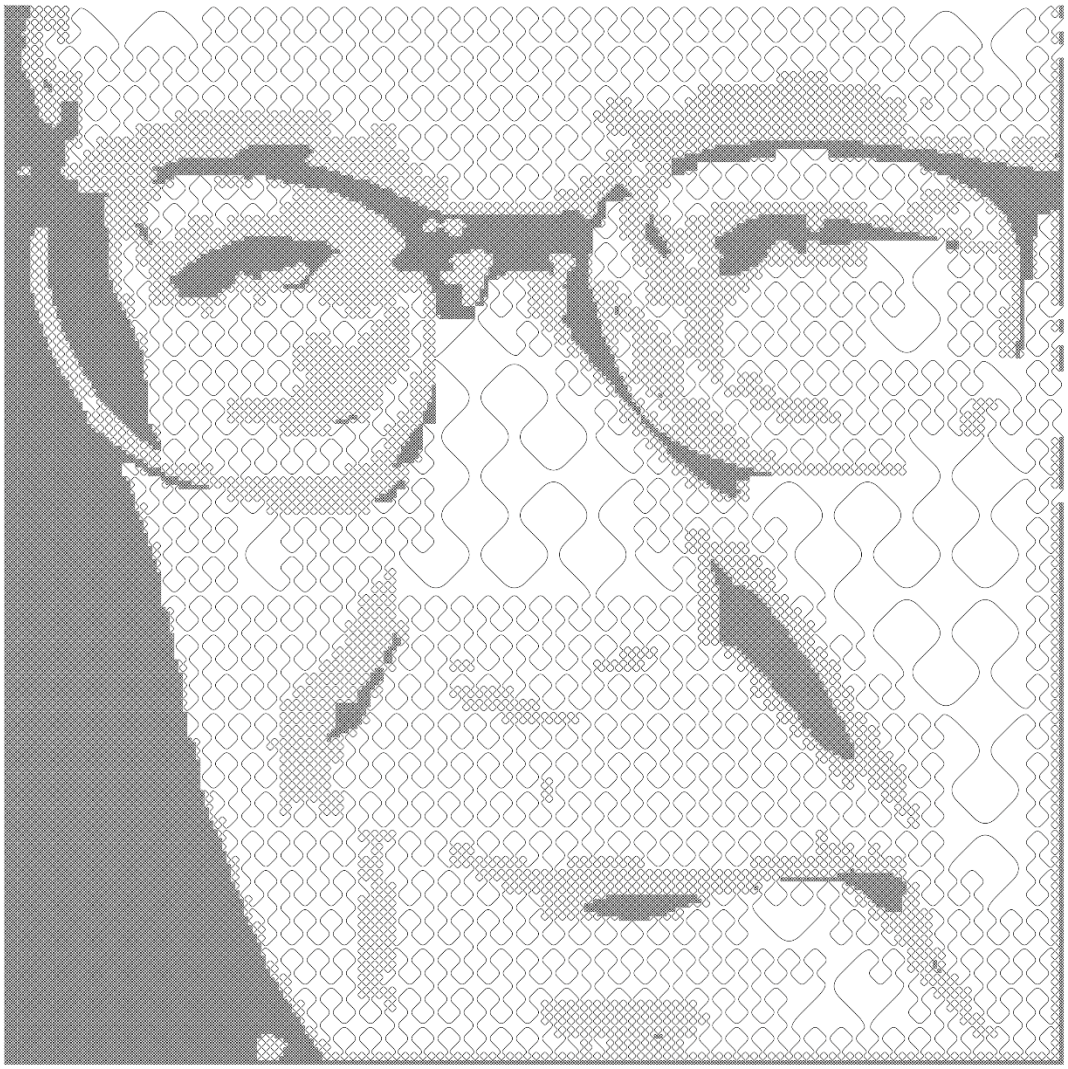

Figure 11 Peano curve, with a recursion level that changes for each point. The drawing is made of a single stroke.

Sierpiński triangle

The Sierpiński triangle is a well-known geometric pattern closely related to recursion.
It comes out in different contexts and, as we will see, shows many deep interconnections among various fields. As in the previous examples, we can generate it with a simple recursive procedure. We start with a triangle (any triangle works well, but we usually select an equilateral triangle). We replace the triangle with three smaller congruent copies, touching at the vertices and arranged in the outline of the original triangle. We repeat the process for each triangle infinitely.
Fig. 12 shows the process. The bottom triangle has eight steps of recursion and contains 6561 tiny triangles.

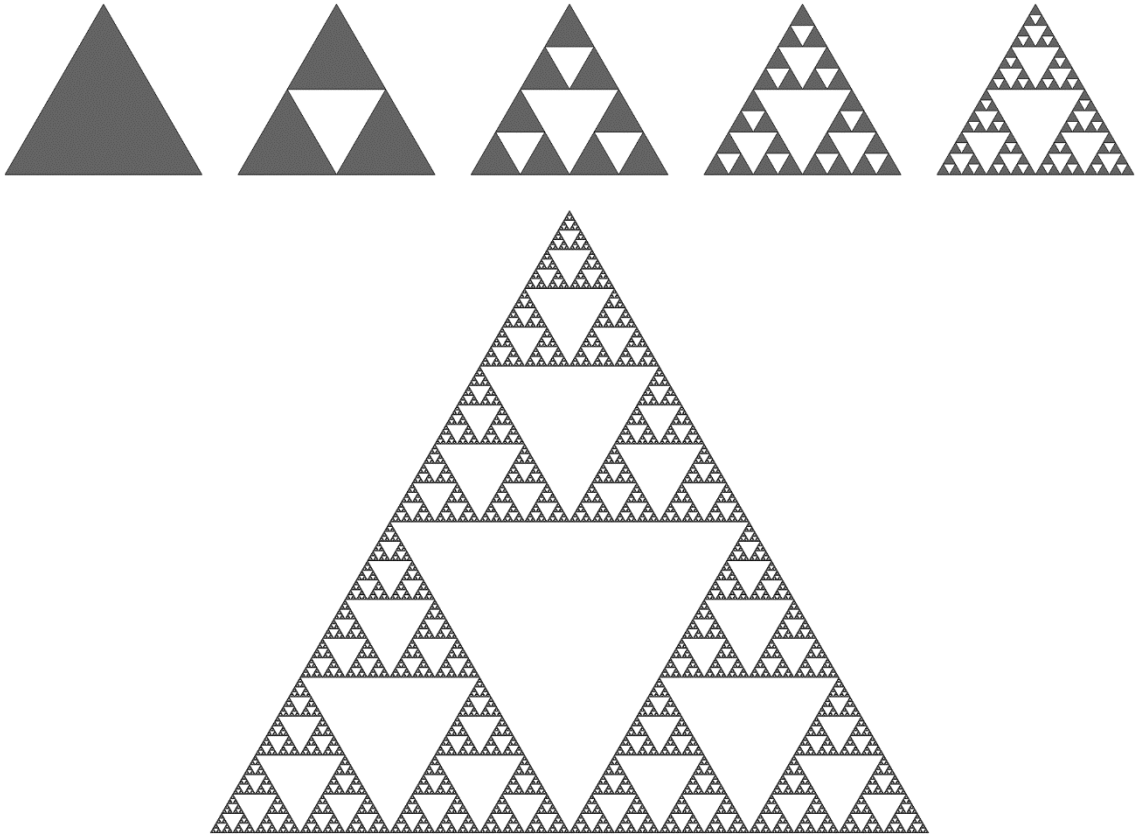

Figure 12 The first steps of the Sierpiński triangle

The pattern is beautiful, and it has been used in art long before its mathematical formalisation. For example, there is a level-4 triangle on the floor of the basilica of San Clemente in Rome, dating late 11th century.

One interesting geometrical feature of the Sierpiński triangle (in common with very many other similar shapes) is its behaviour when we change the size. If we double the size of the first triangle, we obtain three copies of the original pattern, so the area must be three times larger. This outcome is unusual. We intuitively expect that the effect of a scale transformation depends on the dimension of the shape: if we draw a circle twice the size, then the length of the circumference should double as well ($2^1 = 2$), but the area should be four times larger ($2^2 = 4$). According to this reasoning, we should conclude that the dimension of the Sierpiński triangle is between one and two: the shape is more than a unidimensional curve but less than a plane figure. We can say that it has a “fractal dimension” close to 1.585. ($2^{1.585}$ is close to 3).

To emphasise that the Sierpiński triangle is halfway between one-dimensional and bi-dimensional, we will show how to describe it as a curve. The procedure is similar to the one we have already used in the previous paragraphs (the curve is not a plane filler in this case). Fig. 13 shows the first steps. At each stage, each segment is substituted by three smaller pieces forming 120° angles among them. After eleven steps, the result is indistinguishable from that of the first procedure (the one with triangles).

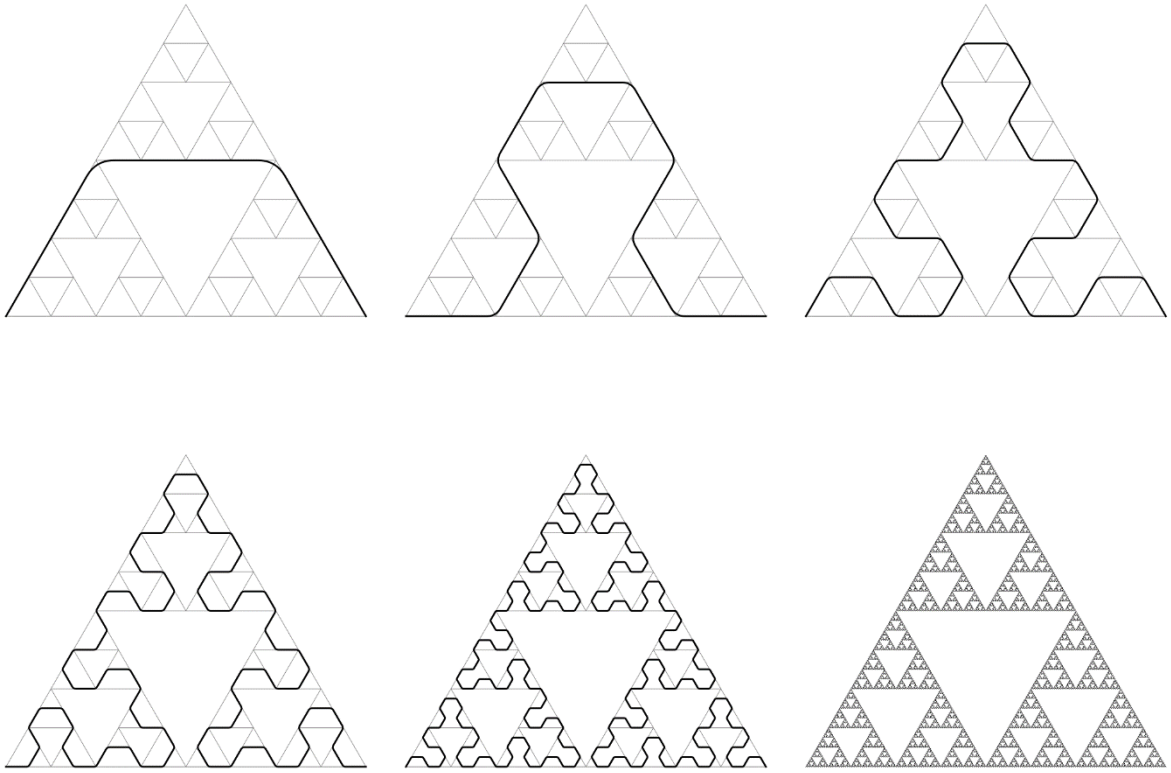

Figure 13 The Sierpiński triangle as a curve

We inadvertently touched the Sierpiński triangle in the previous paragraph when we played with the Hanoi Tower. The set of all the possible states of a Hanoi puzzle with N disks makes a graph with the arcs representing the allowed moves. The graphs are usually depicted with a diagram, with circles for the vertices and lines for the arcs. The positioning of the circles is not determined by the graph and follows aesthetic criteria. In this case, there is a very natural arrangement. In the game, the smaller disk can always move on each post, and therefore the whole graph must be made of tiny triangles connected by the vertices. The other connections (legal moves) assemble these triangles in a predetermined shape. Fig. 14 shows the graph of the puzzle with four disks: it is a Sierpiński triangle!

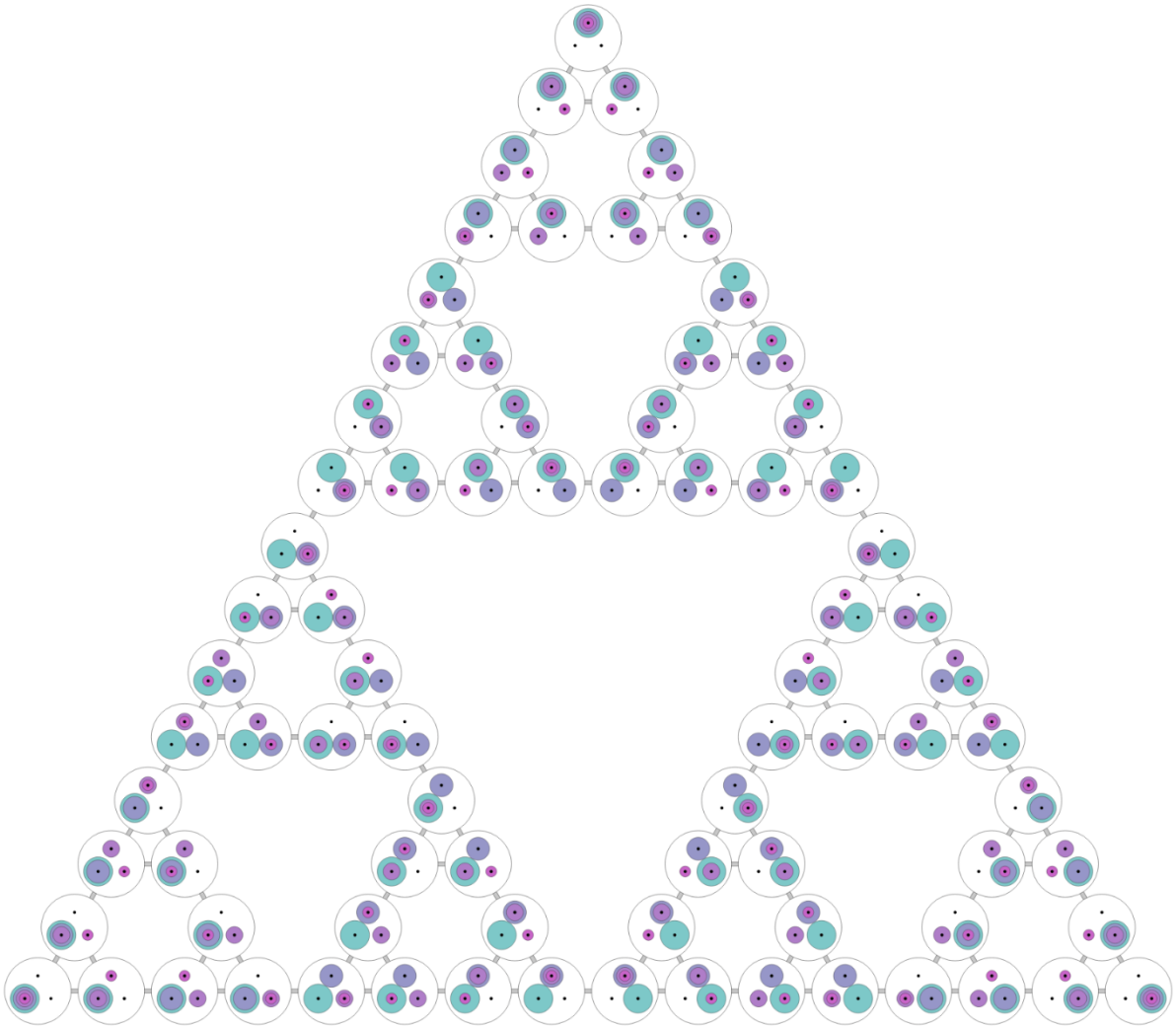

Figure 14 The complete graph of moves for the Tower of Hanoi puzzle with 4 disks

The link between the Sierpiński triangle and the Hanoi Tower is surprising, but there are many other unexpected connections. An example is Pascal's Triangle, which is made of rows of integer numbers. The first row contains only the number 1. Each number in the following rows is the sum of the two other numbers above it in the previous row (treating missing numbers as zero). Let us draw the Pascal's Triangle using different colours for odd and even numbers: odd numbers form the Sierpiński pattern again!

An even stranger connection comes from the *Chaos Game* invented by Michael Barnsley in 1988. The game is an algorithm that generates patterns with a random walk. The algorithm depends on some parameters and can create different shapes (in 2D, 3D and even in higher dimensions). Let us start with the simplest one. We start with a triangle (as in the previous example, we will consider an equilateral triangle, but any triangle will work as well). Then we take a random point somewhere in the plane. It could be the centre of the triangle, but the initial position is not very important.
At each step, we select one of the three triangle vertices at random and move the point halfway toward the chosen vertex. Because of the random choices at each step, it is impossible to predict where the point will be after a given number of iterations, but

apparently, some spots are more likely than others. If we draw many points, a pattern slowly emerges. Fig. 15 shows the first 10’000 points: they look like the Sierpiński triangle!

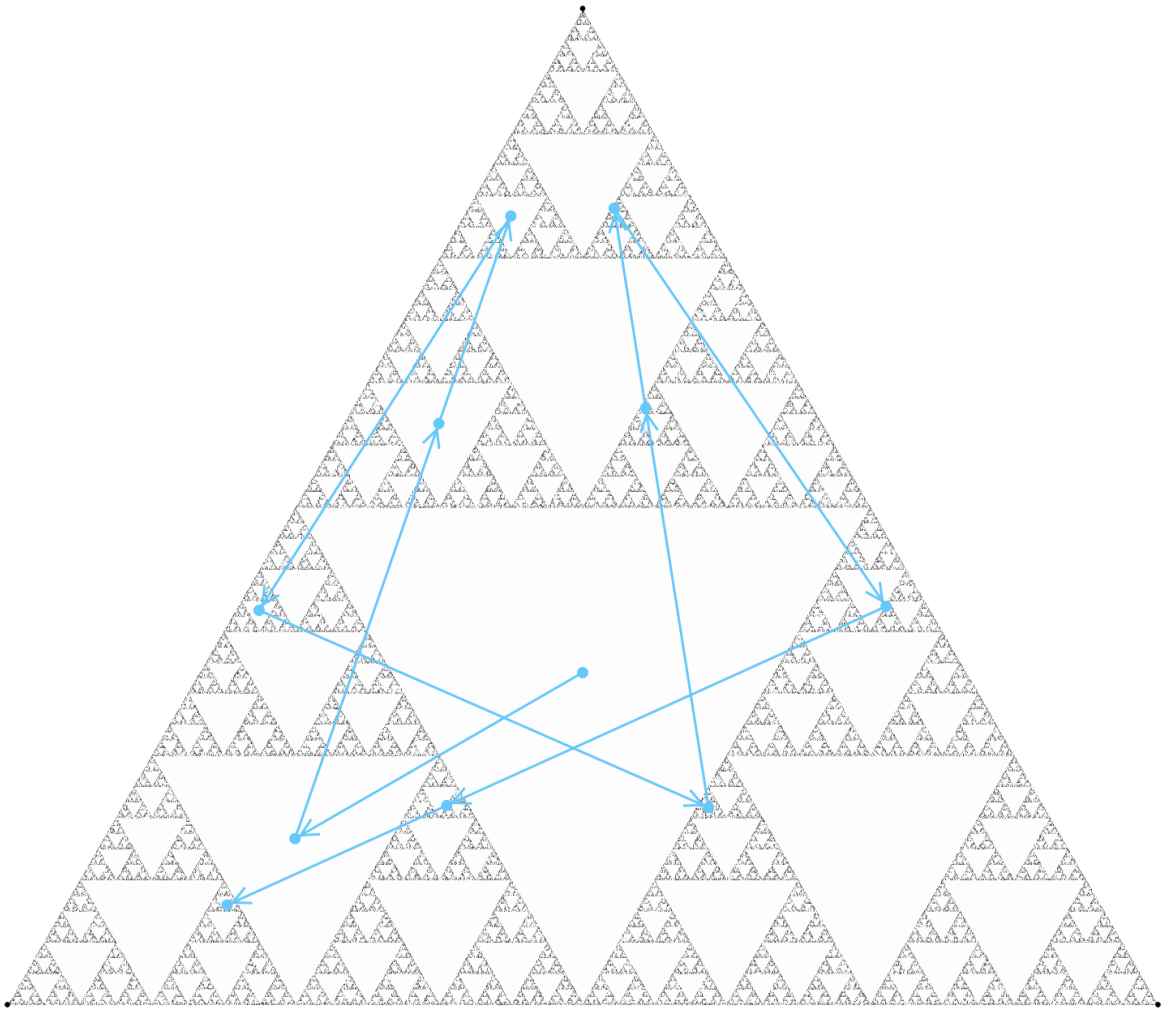

Figure 15 The Chaos Game. 100'000 points. The first 10 are highlighted

A connection between our fractal set and the Chaos Game is easy to spot. Each game’s move is a scale transformation with the centre at the chosen vertex and a scale factor equal to ½. It transforms the whole Sierpiński triangle into one of its three smaller replicas. Therefore if a point belongs to the fractal set, then it will remain on the set forever. Because of the shrink, points outside of the set become closer and closer to the set at each iteration.

Playing the Chaos Game with different parameters is also enjoyable. We can change the number of vertices (and their placement), the scale factor (we used ½), and add constraints on the random choice of the vertex (e.g. avoid choosing the same vertex twice in a row). Different parameters will create different patterns (albeit not always fractal).
A remarkable but straightforward variation shown in Fig. 16 uses a regular pentagon instead of a triangle (and no other changes).

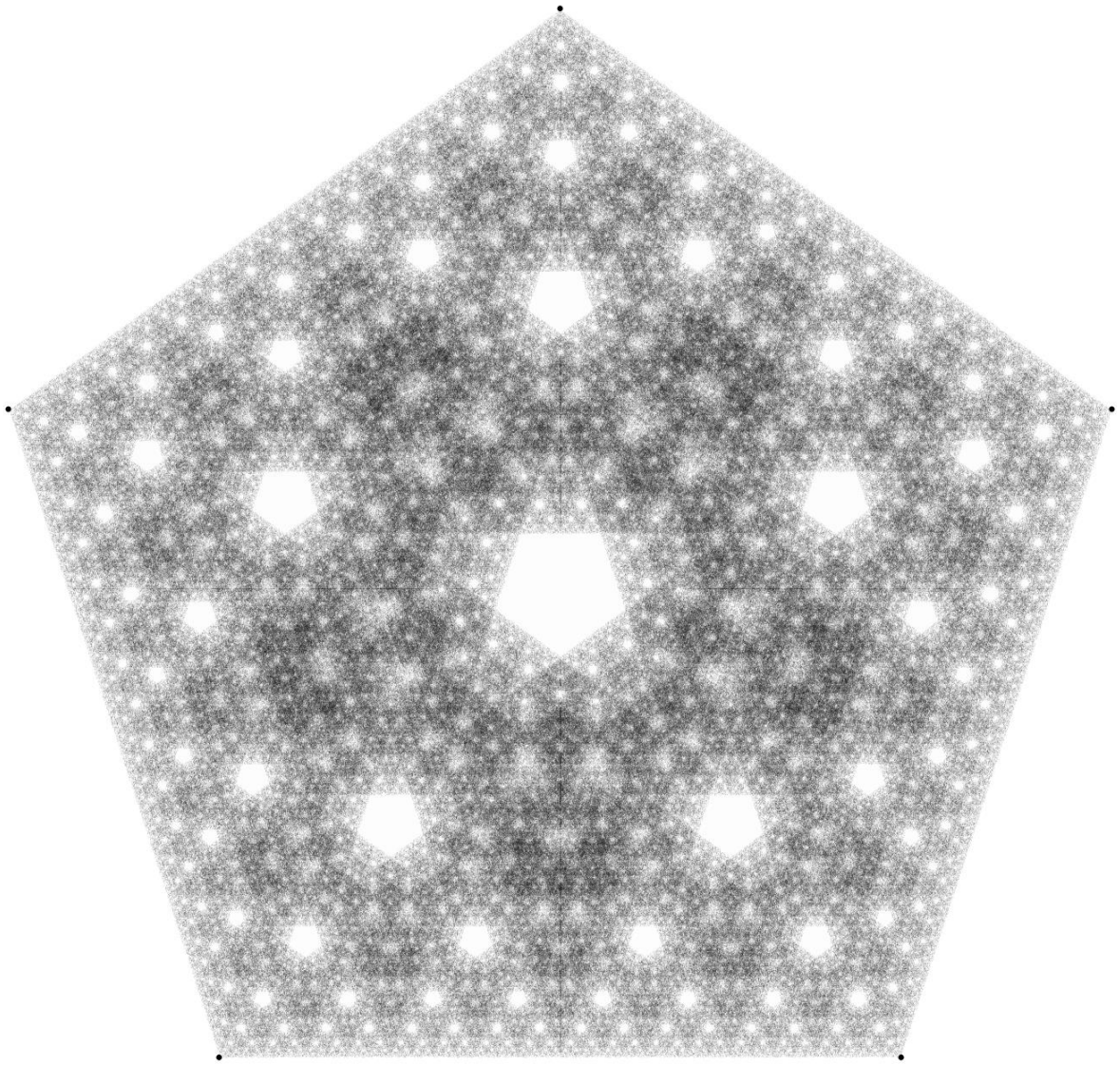

Figure 16 The Chaos Game with 5 vertices. 5'000'000 points

Natural shapes

Michael Barnsley extended the Chaos Game using a slightly more complex set of rules. The original Chaos Game uses three scale transformations selected at random at each step (the scale factor is 0.5 and the scale centres are the three vertices of the triangle). Barnsley decided to use a more general type of transformation: the affine transformation. An affine transformation can include a rotation, a translation, a non-uniform scaling and any combination. It preserves lines and parallelism, but not necessarily angles and distances. One of Barnsley's best known (and beautiful) models, the *Barnsley fern*, uses four carefully crafted affine transformations selected at random with different weights at each step. The idea is very similar to the original Chaos Game. We start with a point (in a random position); at each stage, we select one of the four affine transformations and use it to move the point to a new position. The selection is made at random but with given probabilities for the four different outcomes.

We draw a dot at each point position. Fig. 17 shows the result with three millions points. In the following picture, you can also see the effects of the four transformations on a given triangle.

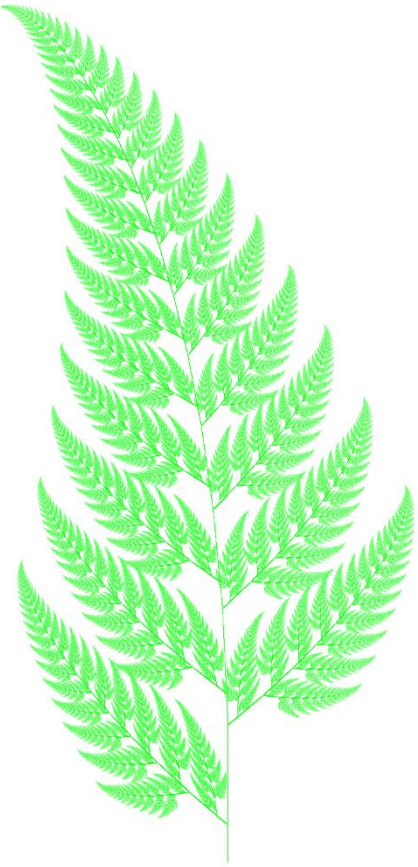

Figure 17 The Barnsley Fern

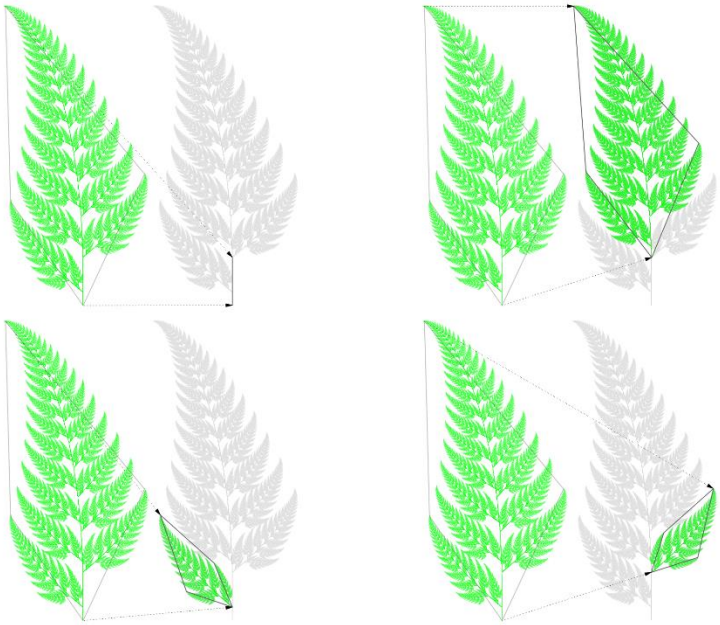

Figure 18 The four transformations used to create the Barnsley fern

Barnsley selected the four transformations' parameters and weights carefully so that the generated pattern resembles a leaf of an actual plant: the Black Spleenwort. It is interesting that a simple algorithm can obtain such a faithful representation of a natural object. With Barnsley words: "[...] provide models for certain plants, leaves, and ferns, by virtue of the self-similarity which often occurs in branching structures in nature".

Fractals and self-similar, recursive patterns are indeed excellent models for natural shapes. Even with a deterministic algorithm (one without the random choices of the Chaos Game), we can create forms with some sort of organic features (see fig. 19).

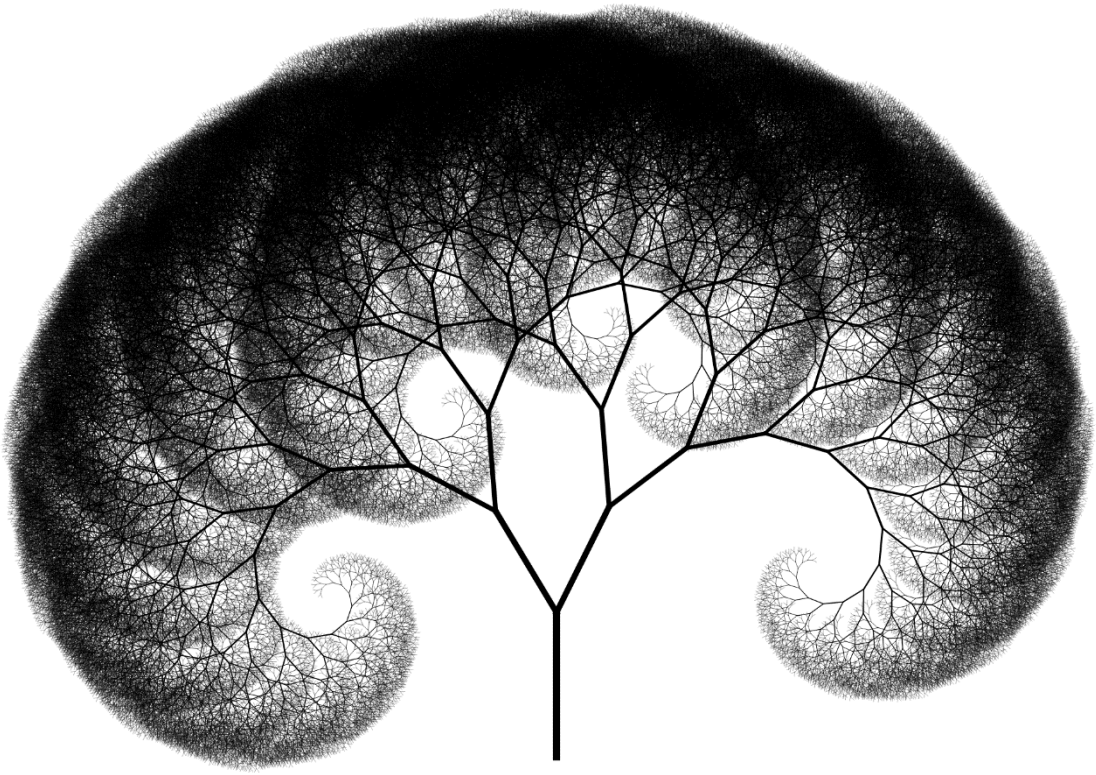

Figure 19 A deterministic recursive tree (no random parameters); 18 levels of recurson

3D

Most of the self-similar shapes we have presented have a natural extension in 3D geometry. For instance, you can build a Sierpiński tetrahedron using the same logic of the Sierpiński triangle. The 3D models maintain the charm of their flat counterpart. Building self-similar 3D models with a computer program, glue and paper, or a 3D printer is a rewarding activity. The Menger Sponge is a particularly tempting model. It starts with a simple cube. We divide the cube into 27 smaller cubes arranged as in the Rubik's cube. We then remove the inner cube in the centre and the six cubes in the middle of the original cube faces. The twenty remaining cubes make a level-1 Menger sponge. We repeat the process on the remaining cubes to create the next level sponge. Fig. 20 shows a level-5 Menger sponge.

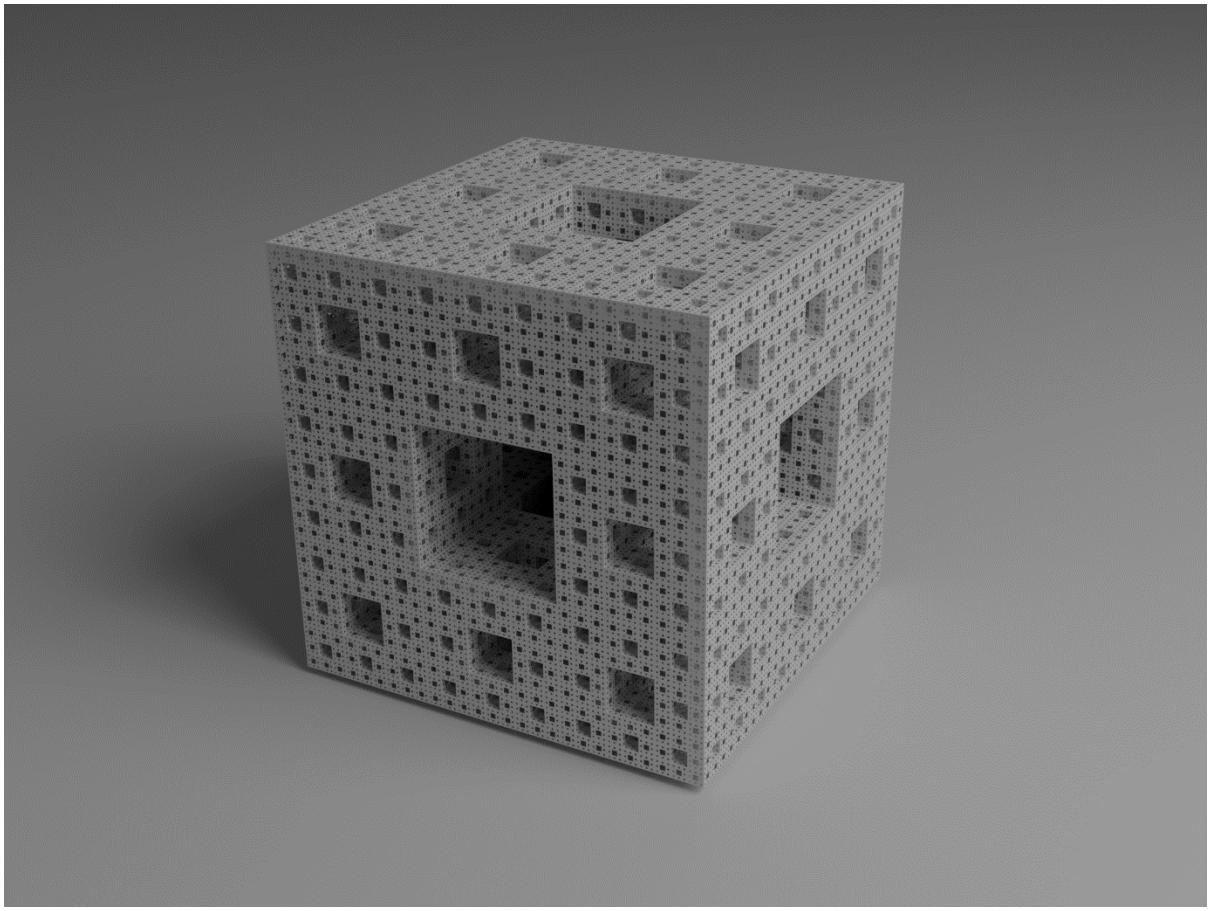

Figure 20 A level-5 Menger Sponge

The Menger Sponge is a three-dimensional generalisation of the Cantor set: a fascinating uni-dimensional pattern presented by Georg Cantor in 1883 but discovered in 1874 by Henry John Stephen Smith. The pattern starts with a segment. At each step, we delete the middle third of every remaining segment. The process must continue ad infinitum. It leaves a set of points aligned along the initial segment with a lot of interesting properties.
We can extend the process to any dimension. In two dimensions, the pattern is called the Sierpiński carpet. We start with a square, divide it into nine smaller congruent squares arranged in a 3x3 grid and remove the central square. Then repeat for each remaining square ad infinitum.

The straightforward geometry of the cube makes it easy to build the Menger Sponge (at least for the first few recursive levels). You can even use Origami to construct small cubes and assemble the sponge.
The problem with 3D self-similar shapes is the tremendous growth of the number of elements depending on the recursion level. For the Meger Sponge, each successive level of recursion requires twenty times more elemental cubes. A simple level-2 Sponge is made of 400 cubes. Creating a paper model of a level-3 (or the astounding level-4) Menger Sponge is a formidable task better suited for teamwork. In 2014, the *MegaMenger* project, conceived by Matt Parker and Laura Taalman, coordinated many worldwide groups to create a level-4 Menger Sponge. The groups eventually built twenty level-3 Menger Sponges, made out of over a million folded business cards. In principle, the collaboration could have assembled those level-3 models into a monster level-4 Menger Sponge.

Even with computer graphics, creating shapes with such a large number of elements can be problematic. The problems begin gradually. The 8’000 cubes that make a level-3 Sponge are manageable: it is possible (and easy) to create an interactive model that runs on a regular Internet browser in real-time. Level-4 requires 160’000 cubes, thus almost two million triangles: the animation becomes slow and jerky. The next level is out of reach with this technique.
We, therefore, must use a different approach to get to higher levels. The crucial point is noticing that, while the number of faces to draw is exceptionally large, the number of distinct planes to which these faces belong is much smaller. The idea is to draw all the faces belonging to the same plane at once.
Let us consider one of the distinct planes. We create an image representing the section of the Menger Sponge relative to that plane. We leave the parts of the image not covered by Sponge faces as transparent. Then we draw a single square face, large as the whole cube, using that image as a texture.
Finally, we repeat this operation for each distinct plane, and we obtain a lovely model using relatively few faces.
Unfortunately, the section images are not identical. Some (at least the first and the last along each cube axis) are just the Sierpinski carpet. The level-1 sponge (4 slices along each axis) requires only that image. Level-2 requires two different slices, and the number doubles for each subsequent level. Fig.21 shows the 16 sections required for a level-4 Sponge; The next level requires 32 different sections. The number is not so small, but that sponge has 244 slices along each axis and 3.2 million elemental cubes! (see fig.20 again).

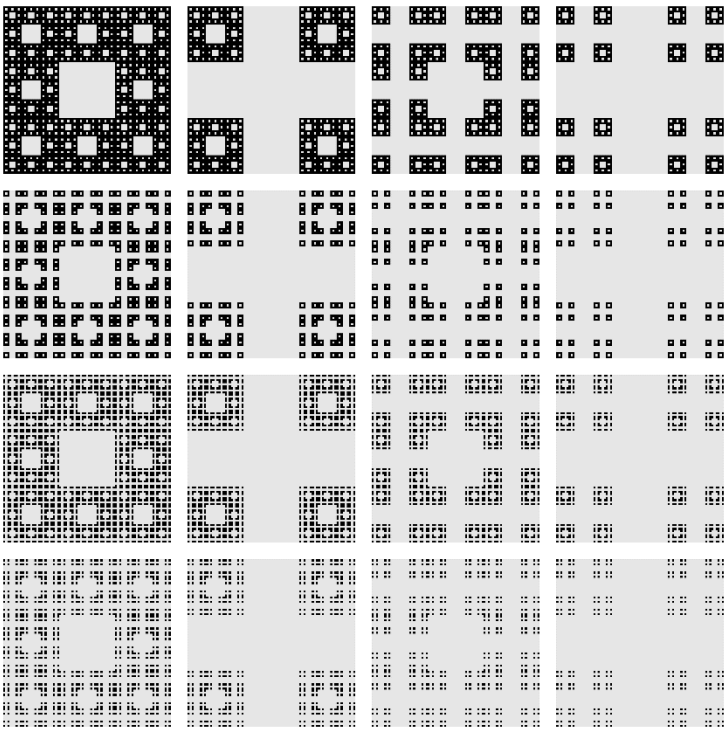

Figure 21 The 16 different sections for a level-4 Menger Sponge

With higher levels, the number of different images becomes problematic. Moreover, the required resolution of each image must be more significant because of the ever more minute details.
To climb to even higher levels, we need a mixed approach: we create a level-5 (or level-6) sponge using the section strategy, and then we assemble 400 (or 8000) copies of this model.
Writing a computer program that visualises a level-7 Menger Sponge spinning in real-time on the computer screen is a rewarding achievement.

Another challenge is trying to get not-orthogonal sections. This task requires a different and more technical strategy (programming the GPU); the results are lovely images. Fig. 22 shows a cross-section passing for the sponge centre forming 45° angles with the cube axes.

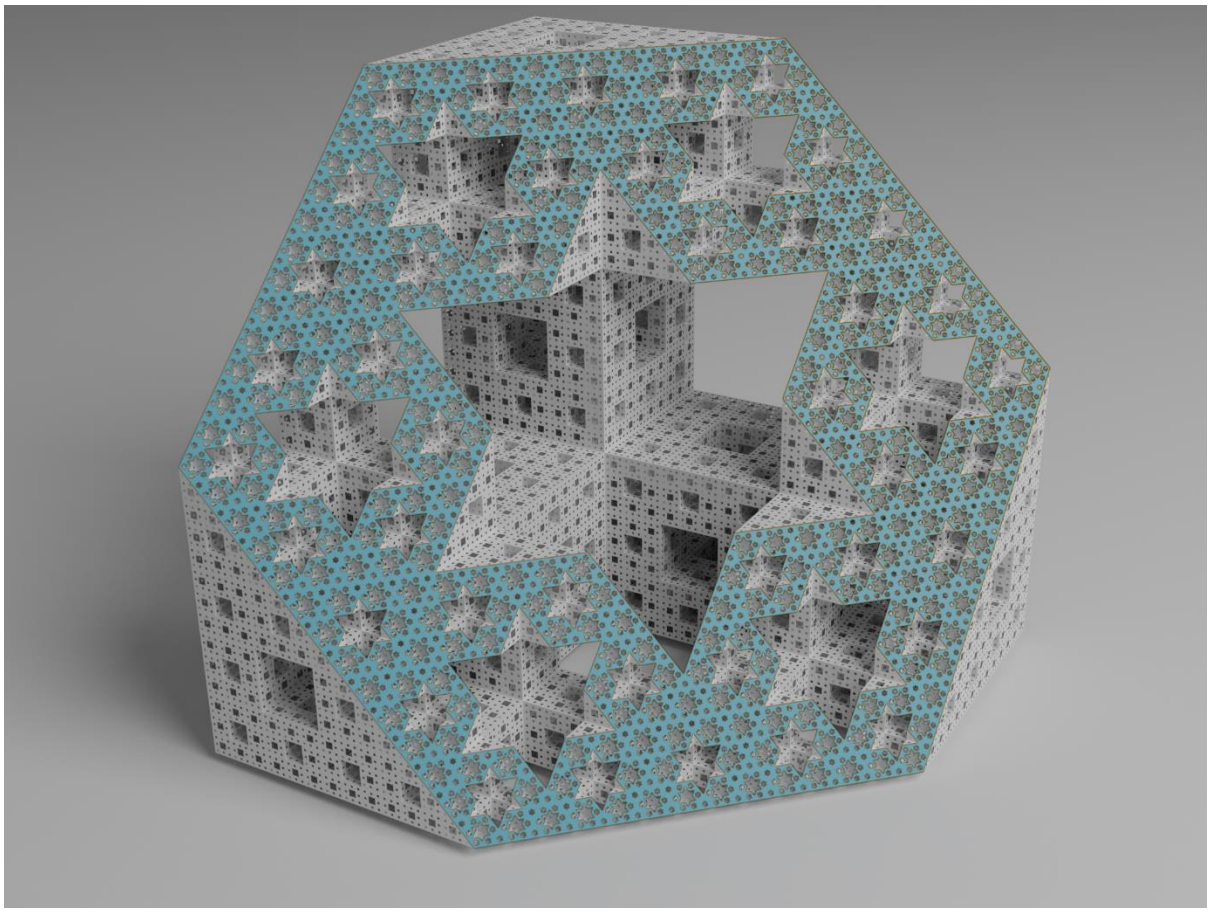

Figure 22 A level-5 Menger Sponge cut with a non orthogonal plane

Alexander's horned sphere

The last model that we meet in our walk in the world of recursion is another weird shape.
It is a paradoxical figure discovered in 1924 by J. W. Alexander to disprove a very reasonable and intuitive assumption.
Let us consider a solid sphere. A loop outside the sphere can be shrunk into a point without touching it. The torus has different properties. A loop linked to the torus can not be shrunk into a point without passing through the torus.
That is a very well-known topological property of the ball and the torus (more precisely, it is a property of the space outside these shapes).

It is clear that the geometrical properties of the solids don't matter: the cube behaves like the ball, and a cup of coffee (with a handle) acts like the torus.
The very reasonable assumption is that the space outside any shape topologically equivalent to the ball is like the space outside the ball: any loop can shrink into a point without touching the surface.

In two dimensions, this is a fact: it is the Jordan - Schonflies theorem. In 1921 Alexander declared it was able to extend this result to the three-dimensional case. Before publishing a paper, he found an error in his demonstration. Eventually, he discovered a counterexample, showing the claim is false indeed! The weird discovered shape is called the Alexander Horned Sphere. It is topologically equivalent to a ball, but the outside is not equivalent to the ball outside: it contains loops that can not shrink into points without touching the shape.

To build the Horned Sphere, we must follow a recursive procedure.
We start with a torus with a small missing section. It is topologically equivalent to a sphere; geometrically, it reminds the letter C.
We take two smaller copies and glue them to the opposite end of the C. We arrange the position to chain the two smaller toruses together.
Then we repeat the process on the smaller C's and so on, ad infinitum.
At each stage, we get a better approximation of the final shape.
Alexander's horned sphere is the union of all the infinite stages.

It is possible to demonstrate that the resulting weird solid is still equivalent to a ball, like all the successive approximations. On the other hand, the space outside is so deeply tangled and knotted that it is pretty different from the space outside the ball. A loop around the first C is hopelessly linked and can not shrink to a point without crossing the surface.

The scheme invented by Alexander and described in the previous paragraphs features many "corners" and creases. Still, they can be rounded to obtain a smooth surface without affecting the topological properties.
I've created a small animation that visualises the growth of the horns up to a certain recursion level. Fig. 23 shows some frames of the video.
The model is lovely even beyond its mathematical meaning.
It seems a couple of hands that want to hug. In the beginning, there are only two, but after a while, there are many, many more.
This image acquired new meanings in these extraordinary and complicated pandemic years.

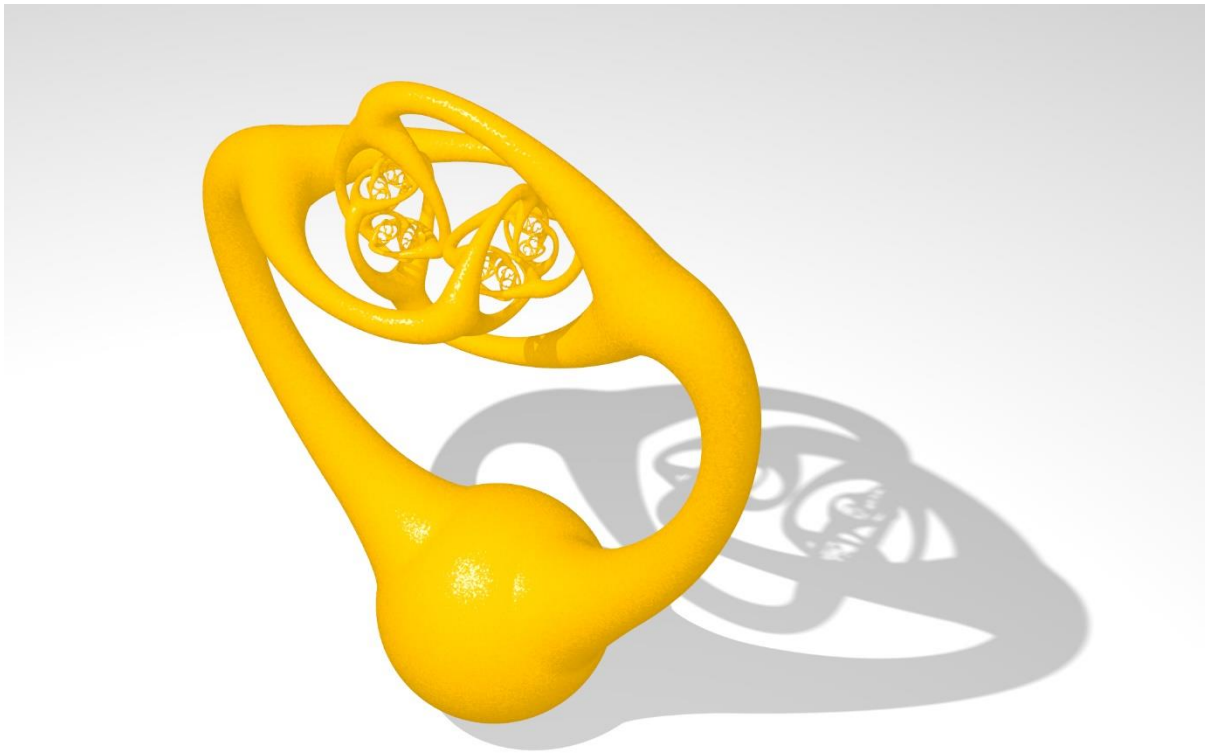

Figure 23

Bibliography

M. F. Barnsley, Fractals Everywhere, Academic Press, Boston, MA, 1988.
Ecc.