A Unifying and Productive N-dimensional Fractal Algorithm

Manuel Diaz Regueiro

Retired Math Teacher, Spain; mdregueiro2@gmail.com

Abstract

The Menger-Diaz 3D fractal algorithm, published in Hyperseen [2], now presents its potential as an N-dimensional fractal algorithm. This algorithm serves as a unifying system, generating well-known fractals in both 2D and 3D. By modifying the existence matrix, it produces diverse fractal forms.
The study extends to n-dimensional generalization, with a particular focus on 2D and tesseracts in the fourth dimension. All of this explained with programming examples, analyzing the parameters, providing code examples to generate fractals in Rhino Python, highlighting the flexibility and simplicity of the algorithm.

.

Introduction

The algorithm, which we introduced for the first time at SCHULPT 2023 and later in Hyperseeing 2023, serves as a unifying system. It harmonizes various fractals into a single generative framework. By manipulating the existence matrix, this algorithm can generate diverse fractals,including the triangle, carpet, Sierpinski tetrahedron, Menger sponge, Vicsek fractal, and their variants.
Now, let’s explore its applications in two dimensions: It adapts seamlessly to different polygons. Furthermore, our study has expanded to n-dimensional generalization, with a particular focus on tesseracts in the fourth dimension. This advancement could unlock new possibilities within fractal geometry. However, achieving n-dimensional visualization requires novel representation systems capable of depicting dimensions beyond three.

The Menger Sponge

The Menger Sponge represents a three-dimensional extension of the one-dimensional Cantor set and the two-dimensional Sierpinski carpet. It was originally described by Karl Menger in 1926 [1] during his exploration of topological dimensions.
The construction process for a Menger sponge unfolds as follows:
1. Begin with a cube.
2. Divide each face of the cube into nine smaller squares, akin to a Rubik’s Cube. This subdivision results in 27 miniature cubes.
3. Remove the central smaller cube from each face, as well as the central smaller cube within the larger cube. This leaves us with 20 smaller cubes, constituting a level-1 Menger sponge (resembling a void cube).
4. Repeat steps two and three for each of the remaining smaller cubes, continuing this iterative process indefinitely.
The Menger Sponge showcases remarkable self-similarity and complexity, making it a captivating object of study in mathematics and fractal geometry.

Recursion

Recursion is a process that invokes itself either directly or indirectly. The corresponding function is referred to as a recursive function. In this iterative approach, a rule or operation is repeatedly applied to a given set of elements. However, to prevent infinite recursion, a stopping criterion is essential. Typically, only the level of recursion changes during each iteration.
The complexity of recursion varies based on the rules we apply. As we introduce increasingly intricate rules, the process becomes more intricate as well.
These complex rules, while flexible and easily modifiable, must be stored in the program. When rules alter the number of segments, squares, or cubes in each iteration, we need to track which elements exist at each step of the algorithm. In some cases, it suffices to store the rules for generating new elements in a matrix (referred to as the “existence matrix”). By retaining existing elements in this matrix, we can proceed to the next iteration. We can even consider this process as a form of “Menger iteration,” where the generated elements require a specific memory position.

Menger's Algorithm

Menger's algorithm stands as one of the earliest elegant examples that adheres to specific rules. Imagine taking a cube and dividing it, much like a Rubik's Cube, resulting in 27 smaller equal cubes. Now, eliminate the 6 cubes located at the center of each face and the one at the center of the original cube.
The Menger cube, also known as the Menger sponge, played a pivotal role in extending and popularizing the concept of fractals due to its beautiful and straightforward description. However, its practical application as an algorithm can be somewhat intricate. This fascinating structure has spread worldwide—I've encountered it in places like the Science Museum in Santa Cruz de Tenerife and the Verbum in Vigo. Menger's achievement remains universal and mesmerizing. While a single sentence can describe recursion, applying it effectively to a 3D program is no small feat. In certain versions, the new cubes are determined using a base-3 numbering system for the numbers 1 to 27.

Existence matrix

The concept of an existence matrix is crucial when dealing with recursive algorithms like Menger's. Essentially, we need a set of rules to determine which cubes persist in each recursion and which ones do not. Consider a given level of recursion, denoted as "l". At this level, we have a total of 27^l cube positions, and our task is to determine whether each position will ultimately be occupied or remain empty.
To achieve this, we establish rules for the permanence or exclusion of cubes generated during each recursion. If our goal is to replicate the Menger fractal, there are straightforward ways to memorize these rules. However, if we aim to generalize Menger's algorithm by adjusting the number of segments into which each original cube edge is divided (represented by the variable *nsegm*), we need a more flexible approach.
Enter the existence matrix—a three-dimensional representation that encodes Boolean values (True or False, or 1 or 0) for each possible position. This matrix organizes the locations systematically, making it easy to reference the existence status of a cube at a specific place or its exclusion.
In summary, our task is to determine which positions are excluded or remain occupied during each iteration. Armed with this existence matrix, we can explore a multitude of new algorithms and shapes. The possibilities are vast, especially considering that the number of 3-dimensional matrices we can create is 2^nsegm^3.
This existence matrix applies not only to the classic Menger algorithm (where each cube is divided into sub cubes) but also to the Menger-Diaz algorithm. In the latter, each cube "multiplies" into many other cubes in space, following the same existence matrix rules.
While both methods are nearly equivalent, the Menger-Diaz approach requires rescaling at each iteration (or at the end). Implementing the second algorithm becomes more straightforward using tools like Rhino 3D, where we can apply copy instructions to objects in various locations determined by the existence matrix.
In essence, Menger represents a top-down design, while Menger-Diaz leverages existing object-copying instructions for a more flexible, bottom-up approach. By adjusting parameters like the distance factor (*d*), which ensures that polyhedra or polygons align at vertices, edges, or faces, we can create novel forms. Keep in mind that experimenting with these concepts—whether through Rhino or other 3D programs—offers deeper insights into their behavior.

Menger-Diaz Algorithm

The Menger-Diaz algorithm transforms Menger's design into a parametric model, offering multiple options—a strength of Rhino design and Rhino with Grasshopper. To achieve this, we define the new parameters *nsegm* (number of segments) and *d* (separation).
Menger-Diaz is a more productive algorithm because it expands the application of the Menger algorithm by varying these parameters. The separation distance between objects depends on the specific figures we're working with, often requiring experimental adjustments. For regular polygons, the radius parameter defines the size of the polygon. For instance, an octagon's size is determined by the radius of the circle in which it is inscribed. Depending on orientation, we may also need to calculate the apothem of the octagon. As a general assumption for squares, we take radius = 50 and d = 35.

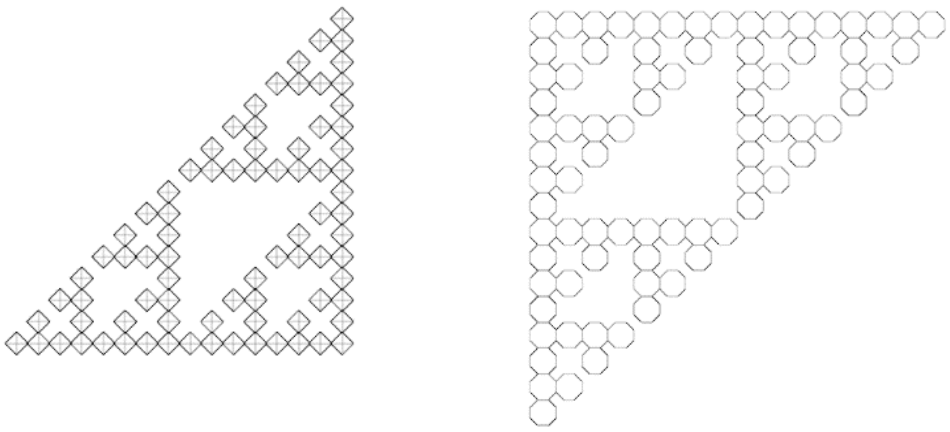
Units: We use the units available in the Rhino program (e.g., mm, cm, meters). Since we create pattern figures that are perfectly scalable while preserving the pattern, the visual result doesn’t depend on the chosen units.
In summary, Menger-Diaz involves parameterization of the following:
1. Initial Atom: This may differ from a cube.
2. Number of Segments: Each edge of the cube is divided into segments.
3. Distance in each Iteration: Determines how the figure evolves.
4. Iteration Rules: These can lead to numerically large results.
5. Criteria for Beauty or Usefulness: Defining what forms are aesthetically pleasing or useful.
6. Handling Dimensions: We can modify Menger-Diaz for dimensions less than three and address challenges when dimensions exceed 3. The algorithm encourages thinking in n dimensions.

Adaptation of the algorithm to 2D

When transitioning from a 3D algorithm to a 2D representation, what changes do we need to make? Let’s explore the necessary adjustments. We’ll consider the following parameters:
1. Dimension of Euclidean Space (*dim*): This parameter defines the space in which the fractal is represented.
2. Number of Segments (*nsegm*): Each side of the square is divided into segments.
3. Recursion Level (*level*): Determines the depth of the fractal.
4. Separation Distance (*d*): Typically, *d* equals size of a segment multiplied by 2. d=Cube edge/nsegm*2.
5. *Radius* and *d* : These parameters play a role in shaping the fractal.
6. Program Simplicity: 2D programs tend to be simpler than their 3D counterparts.
Advantages of 2D:
1. Ease of Review and Understanding: The reduced complexity in 2D makes these algorithms more accessible for review and comprehension.
2. High Number of Illustrations: To verify results, we often create numerous illustrations.
3. Challenges in 4D: In 4D, projecting results back to 3D becomes challenging, limiting our exploration to a single example.
I’ll proceed to create the programs in Rhino Python, starting with a Sierpinski triangle.

```
import rhinoscriptsyntax as rs
nsegm=2
a=[[1,0],[1,1]]
#We draw a regular polygon in Rhino with 4 sides, squares rotated
# the z coordinates in the plane z=0, are 0
d=25
for level in range(1,4):
    objs=rs.AllObjects()
     for i in range(nsegm):
        for j in range(nsegm):
            if a[i][j]==1:
                rs.CopyObject(objs,(i*d*nsegm**level,j*d*nsegm**level,0))
      rs.DeleteObjects(objs);
```

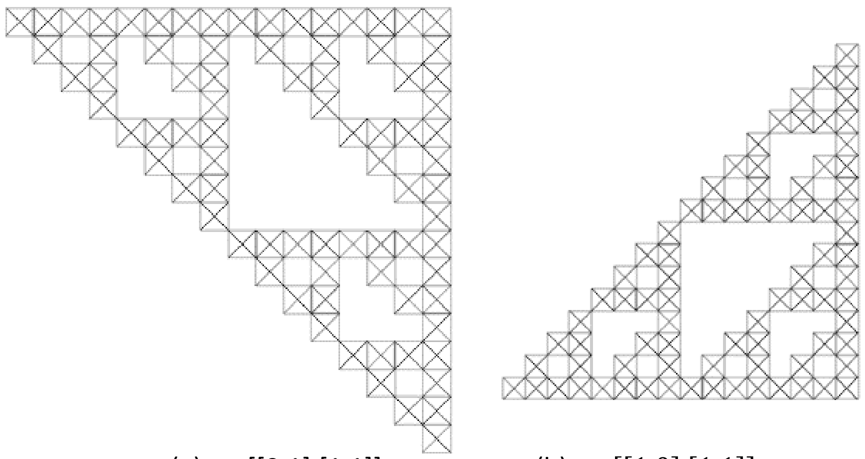
Figures 1, 2 are the result of changing a zero in the existence matrix. What will the figure be like if a=[[1,1],[0,1]] with octagons? In this case, *d* is related to the apothem of the octagon.



(a) a=[[1,1],[1,0]], d=25 (b) a=[[1,1],[0,1]], d=46.19

Figure 1: *Sierpinski fractal* (a) *With squares rotated (b) With octagons*

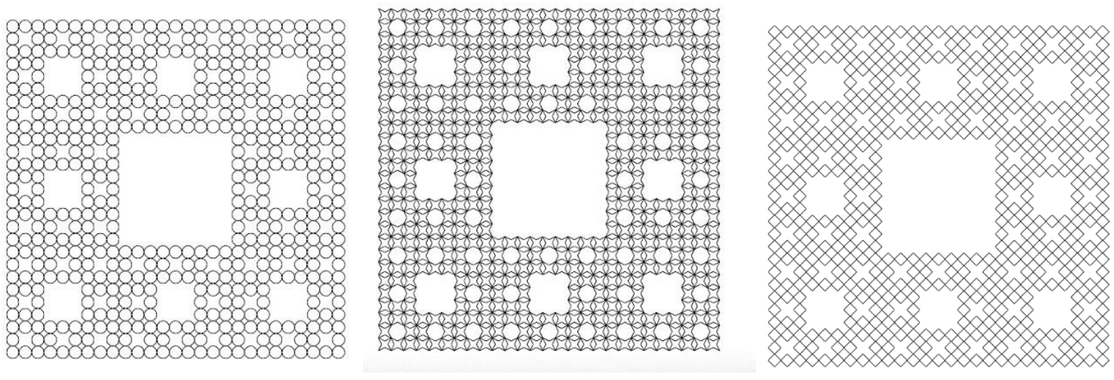
Variants of Sierpinski triangles with squares. Now radius=50, d=35



(a) a=[[0,1],[1,1]] (b) a=[[1,0],[1,1]]

Figure 2: *Sierpinski fractal with squares.*

2D Menger sponge or Sierpinski's carpet a=[[1,1,1],[1,0,1],[1,1,1]], nsegm=3



(a) (b) (c)

Figure 3: *Menger fractal* (a) *with octagons, d=33. (b)With four pointed stars (c) With square rotated d=32*

```
import rhinoscriptsyntax as rs
# octagon, 4 star, square, square rotated 90 n=3
d=32
for level in range(1,4):
    objs=rs.AllObjects()
    for i in range(nsegm):
       for j in range(nsegm):
         if a[i][j]==1:
           rs.CopyObject(objs,(i*d*nsegm**level,j*d*nsegm**level,0))
   rs.DeleteObjects(objs);
```

If we wanted to do with circles then we would use rs.AddCircle((0,0,0),radius) and, of course, d=radius.

Vicsek's fractal with octagons and with squares

a=[[0,1,0],[1,1,1],[0,1,0]] His 3d equivalent is Mosely [9]

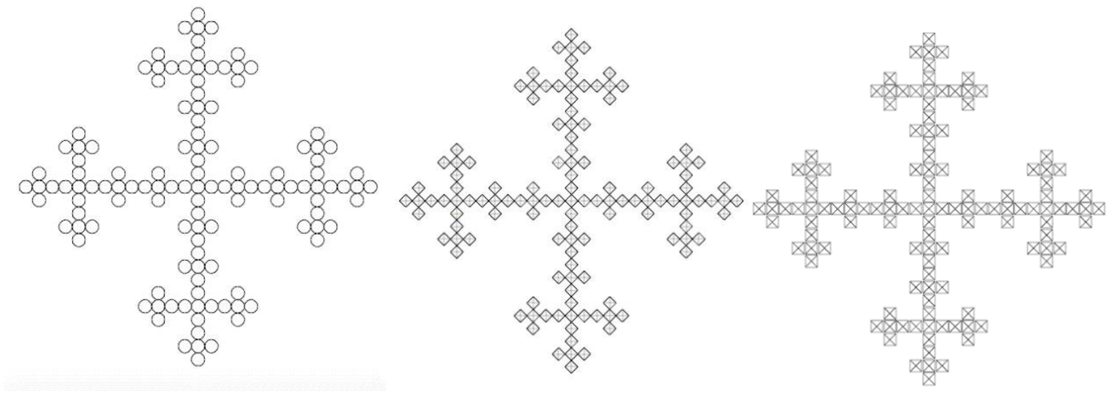
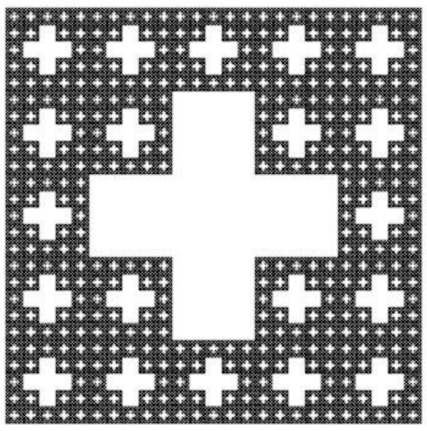
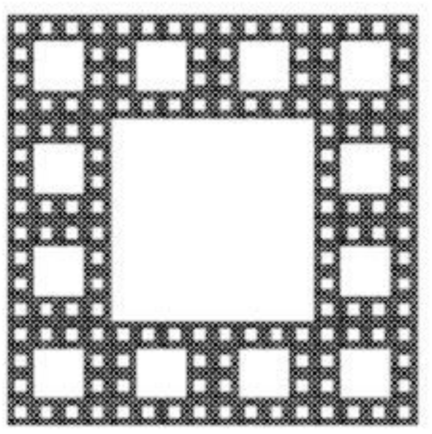


Figure 4: *Vicsek fractal (a) with octagons (b) with rotated squares (c) and squares*

Now we do nsegm>3
Let's look at something more complicated, the cross:

Cross in 2D

a=[[1,1,1,1,1], [1,1,0,1,1], [1,0,0,0,1], [1,1,0,1,1], [1,1,1,1,1]]
nsegm=5, d=14, level=3

(a) (b)

Figure 5: *Sierpinski's carpet Cross (a) and Great Hole4 (b)*

Great hole 4(b), nsegm=4, a=[[1,1,1,1],[1,0,0,1],[1,0,0,1],[1,1,1,1]]

The algorithm 3D

In PythonScript of Rhino for a Sierpinski tetrahedron, but is the same for any list of nested vectors changing the variable *a* and the size of cube. For each level you must repeat the algorithm changing the variable *level.*

```
import rhinoscriptsyntax as rs

objs = rs.AllObjects()

level=1 # the level

d=30#distance. its depends on polyhedron and their size

a= [[[1,0], [0,1]], [[0,1], [1,0]]]

for i in range(2):

  for j in range(2):

    for k in range(2):

       if a[i][j][k]==1:

         rs.CopyObject(objs,(i*d*2**level, j*d*2**level, 5k*d*2**level))

rs.DeleteObjects(objs);
```

You can see numerous examples of Menger-Diaz 3D in [2] and [3], including all 3x3 symmetric face cubes.

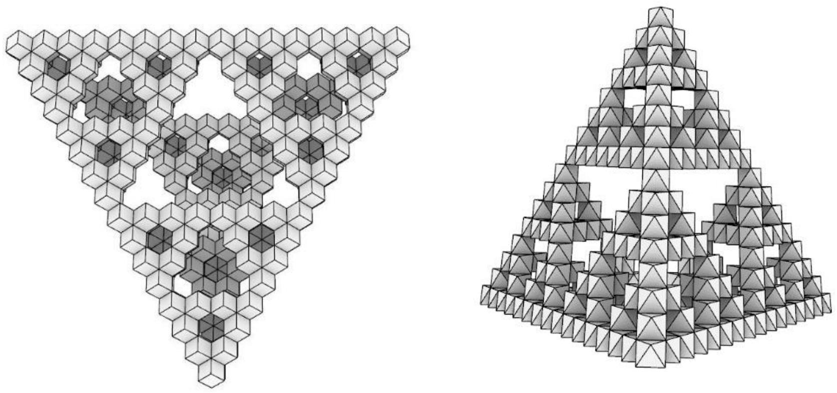


Figure 6: *Sierpinski tetrahedron with (a) Rhombic dodecahedron (b) Cuboctahedron*

Differences between Menger and our Algorithm

The traditional algorithm divides a cube into smaller cubes respecting the shape and eliminating certain subdivisions. Our algorithm does not divide the cube but places it at predetermined distances that depend on n^l , where *l* is the level and *n* is the size of the units of the cube. Enlarges rather than divides the figure (although we are supposed to eventually scale or shrink to the size of the first figure at level 0)
To understand the distinction between our proposed algorithm and Menger's, we need to consider the digital world and how easily elements can be copied within it.
In our digital day-to-day, we continuously copy and reposition elements. Menger, in his pre-digital or analog world, envisioned a top-to-bottom approach—dividing a figure by extracting its predefined parts recursively. For this purpose, he used an easily divisible shape: the cube. In our proposal, we take a different path. We start from the bottom (with the first brick) and construct the figure using Menger recursion.
Our approach involves building rather than separating. At each iteration, the resulting figure becomes the "brick" for the next step. This flexibility allows us to create an algorithm with only a few lines of code, applicable in various situations. Here's the basic process:

1. **Lay the First Brick:**
   - Start with an initial shape (the "brick or atom").
   - Place this brick in a loop.
2. **Position Copies:**
   - Calculate the distance.
   - Position copies of the initial shape at predefined points within the loop.
3. **Repeat the Loop:**
   - Convert the resulting figure back to the initial shape from step 2.
   - Repeat the process.

In Menger's Algorithm the number 3, the cube, and 7 empty spaces are fundamental. Some daring ones that we mention in the bibliography keep the number 3, the cubes, and change the position of the empty spaces. We do not have all the possible bibliography, which, in reality, is more extensive, so we cite the best known, even the recommended ones, but that we were not previously aware of.

In our algorithm, we focus on the cube as the primary shape (for better understanding). We also create images of shapes with *nsegm*=3, not to emphasize them as our main contribution, but to facilitate comparison with familiar concepts. Specifically, we are dealing with cubes composed of nsegm^3 smaller cubes, where *nsegm* ranges from 2 to 3 (similar to Menger's approach). Refer to the article [2] for further details.
This exploration leads to an infinity of possible forms. For instance, when nsegm=6, there are 2^6^3 potential fractals—more than the number of atoms in the Sun. Representing all these fractals on paper would be impossible (and not all of them would be aesthetically pleasing).
And what if we don't work within a cube? Consider other shapes like rectangular prisms, stepped pyramids, or any three-dimensional network of points. The challenge lies in determining placement positions for each copied element in every iteration. There is plenty of exciting work ahead for those who want to explore it further!

Into the Fourth Dimension and beyond

There is a Bridges 2023 article on the Menger cube in the fourth dimension and this is the origin of this article. In that article they present mathematical ideas about the sections and their visual result, but not an algorithm to build it.
The construction of n-dimensional Menger cubes is very simple with our algorithm:
-First of all, the existence matrices are larger (we already saw how the existence matrices in 2D are smaller than in 3D).
-Secondly, the algorithm only changes with the number of *for* lines (one for each dimension), so if in 2D we have 2 *for* lines, in 3D we have 3 *for* lines, in 4 dimensions we will have 4 *for*
lines, etc.

```
for x in range(nsegm):
    for y in range(nsegm):
        for z in range(nsegm):
            for w in range(nsegm):
```

-Thirdly, the coordinates in 4 dimensions are 4 and that must be reflected in the instruction rs.CopyObject(objs,(x*d*nsegm**level,y*d*nsegm**level,z*d*nsegm**level, w*d*nsegm**level)), instruction that does not exist in Rhino, obviously, and it is necessary to do the calculations of the chosen 3D projection of the 4d fractal.
The problem is that Rhino does not display 4 dimensions, but it does save in Obj files, which do save points in 4 dimensions. So progress on this topic should go in the direction of having 4D figure viewers in 3D for certain figure orientations (which exist in the form of Python programs or Unity programs), and in the coding of objects of 4,5, 6,..., n dimensions as files that store their n-dimensional characteristics. And the existence of viewing programs for these n-dimensional shapes. However, we can create 4D visualizations of tesseracts if we think of it in a Dalí Cross-style unfolding of the hypercube in 3D representation.

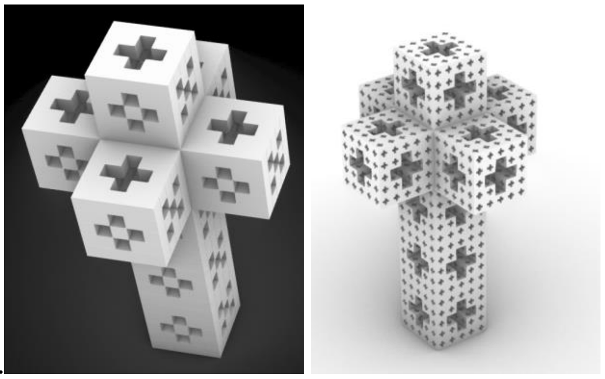


Figure 7: *Menger-Diaz Cross tesseracts in Dalí Cross style net.*

Conclusions and new advances

Numerous algorithms exist for generating Sierpinski triangles, Menger cubes, and their variants- possibly hundreds. However, none is as unifying as the one presented here. Our algorithm, simple and concise, can be applied to various geometric objects (polygons or polyhedra) and adapted to different positions (such as horizontal and vertical squares) with minimal effort. The key lies in adjusting numerical parameters, such as *d*.

Wolfram's more general algorithm [4] (found on page 188) addresses square conversion, but it shares some conceptual similarities with the existence matrix definition. Despite this, it lacks implementation for the *nsegm* parameter, and its mystical quality can obscure understanding.

Unlike Wolfram's approach, our algorithm seamlessly integrates into 2D or 3D design programs like Rhino without requiring a complete reinvention of the software. However, the mystique surrounding the fourth dimension poses challenges for n-dimensional geometric visualization.

In our algorithm, each dimension corresponds to a straightforward "for" loop in the program. Yet, the problem of displaying geometric objects in dimensions greater than 3 remains unsolved. Similarly, saving n-dimensional geometric objects in standardized formats (yet to be defined) represents another avenue for future exploration.

As we venture into higher dimensions, we embrace both the beauty and complexity that await us. The path forward involves not only mathematical insights but also practical solutions for visualizing and preserving these intricate forms.

References

[1] K. Menger, *Dimensionstheorie*. 1928.

[2] M. Diaz and L. Diaz . An Algorithm for Extending Menger-Type Fractal Structures. Hyperseeing 2023

[3] M. Diaz and L. Diaz . Variations on Menger-Díaz Fractals

https://easychair.org/publications/preprint/QFNq69

[4] Wolfram, Stephen. *A New Kind of Science*. Wolfram Media, 2002.

[5] C. A. Reiter, “Sierpinski fractals and GCDs,” *Comput Graph*, vol. 18, no. 6, pp. 885–891, Nov.1994, doi: 10.1016/0097-8493(94)90015-9.

[6] Benoît B. Mandelbrot, *Fractals: Form, Chance, and Dimensions*. San Francisco, 1977.

[7] Y. D. Sergeyev, “Evaluating the exact infinitesimal values of area of Sierpinski’s carpet and volume of Menger’s sponge,” *Chaos Solitons Fractals*, vol. 42, no. 5, pp. 3042–3046, Dec.2009, doi: 10.1016/j.chaos.2009.04.013.

[8] Eric Baird, “The Jerusalem cube,” *Magazine Tangente*, p. 45, 2013.

[9] L. Wade, “Folding Fractal Art from 49,000 Business Cards,” *Wired*, 2017. Accessed: Dec. 28, 2022. [Online]. Available: https://www.wired.com/2012/09/folded-fractal-art-cards/

[10] George W. Hart. Procedural Generation of Sculptural Forms.
https://archive.bridgesmathart.org/2008/bridges2008-209.html

[11] https://demonstrations.wolfram.com/SubstitutionSystemsIn3D/

[12] https://blogs.mathworks.com/cleve/2022/01/15/the-menger-sponge-complement-and-associate/

[13] SCHULPT 2023 Show and Tell July