INFORMATION AS ORDER HIDDEN WITHIN CHANCE

From fractals and cellular automata to biology[1]

by ALBERTO STRUMIA[2]
Advanced School of Interdisciplinary Research (www.adsir.it)
Interdisciplinary Encyclopedia of Religion and Science (www.inters.org)
I.N.D.A.M. F. Severi (www.altamatematica.it)

[1]**DOI**: 10.17421/2037-2329-2019-AS-2
[2]www.albertostrumia.it.

Contents

Introduction

"There is no science on *singulars* (Latin, *scientia non est de singularibus*"[1]), because human science is a *knowledge through universals.* In fact the human intellect acquires its knowledge *abstracting* from matter the *universal form* organizing each *singular* "matter" body. So our mind, being "immaterial", does not know *singulars,* while our senses do, being "material" as part of our "matter" body.

Surprisingly the latter principle, which was well known by Aristotle and mediaeval authors like Thomas Aquinas and his followers, seems to be attained in some way, at least in some of its aspects and through a different way, by our contemporary logicians, mathematicians and experts of information theory.

Knowing *universally,* in terms of our *informational logic,* appears to mean the capability to find a *law* or an *algorithm* the *string* code of which is *shorter than the list of all individual entities* when they are singularly collected in a set. So modern science seems to have rediscovered, in some sense, the ancient Aristotelian-Thomistic principle according to which not all the entities may be described (*logic, cognition, science*) or built (*ontology, metaphysics, physics*) by an *algorithm* (string shorter than the list of individuals). In fact there are entities, the string describing which cannot be other than the list of each single entity (*incompressible string).* Or, in terms of propositions, not any *proposition* (*string*) is *decidable* (by means of a *theorem*) within an axiomatic system (*undecidability*), since it cannot be reduced to the string of the axioms, according to the well known Gödel's undecidability theorem.[2]

Only a *divine mind* can know all singular details of an entity. While our *human mind* knows through universals, so it cannot find an *algorithm* describing all entities (*whole theory* or *theory of everything*) and all aspects of each of them. Only divine mind which knows/creates each single entity, both according to a universal form and to each individualizing matter, is able to catch all singular details.

Aquinas offered a logico-metaphysical explanation of such a difference between human and divine science.

> The reason for this will be clear if we consider the difference between the relation to the thing had by its likeness in our intellect and that had by its likeness in the divine intellect. For the likeness in our intellect is received from a thing in so far as the thing acts upon our intellect by previously acting upon our senses. Now, matter, because of the feebleness of its existence (for it is being only potentially), cannot be a principle of action; hence, a thing which acts upon our soul acts only through its form; consequently, the likeness of a thing which is impressed upon

> our sense and purified by several stages until it reaches the intellect is a likeness only of the form.[3]
>
> On the other hand, the likeness of things in the divine intellect is one which causes things; for, whether a thing has a vigorous or a feeble share in the act of being, it has this from God alone; and because each thing participates in an act of existence given by God, the likeness of each is found in Him. Consequently, the immaterial likeness in God is a likeness, not only of the form, but also of the matter. Now, in order that a thing be known, its likeness must be in the knower, though it need not be in him in the same manner as it is in reality. Hence, our intellect does not know singulars, because the knowledge of these depends upon matter, and the likeness of matter is not in our intellect. It is not because a likeness of the singular is in our intellect in an immaterial way. The divine intellect, however, can know singulars, since it possesses a likeness of matter, although in an immaterial way.[1]

The sequence of such singular elements of a *whole* appears to us as completely *random,* since we cannot – because of principle reasons and not only because of technical difficulties – deduce by a rule (*algorithm*) any of the next element starting from the knowledge of the previous ones. But the datum of the incompressibility of a string, which we perceive as *randomness,* does not mean *non-sense* of that string, but simply that it is *self-explained* being reason to itself and therefore it needs no further explanation, being a *fundamental law,* even though rather complex. As Gregory Chaitin has observed:

> for example, a regular string of $1s$ and $0s$ describing some data such as $0101010101\cdots$ which continues for 1000 digits can be encapsulated in a shorter instruction "repeat 01 500 times". A completely random string of digits cannot be reduced to a shorter program at all. It is said to be algorithmically incompressible ([4] pg 141).

[1] "Cuius ratio manifeste apparet, si consideretur diversa habitudo quam habent ad rem similitudo rei quae est in intellectu nostro, et similitudo rei quae est in intellectu divino. Illa enim quae est in intellectu nostro, est accepta a re secundum quod res agit in intellectum nostrum, agendo per prius in sensum; materia autem, propter debilitatem sui esse, quia est in potentia ens tantum, non potest esse principium agendi; et ideo res quae agit in animam nostram, agit solum per formam. Unde similitudo rei quae imprimitur in sensum nostrum, et per quosdam gradus depurata, usque ad intellectum pertingit, est tantum similitudo formae. The reason for this will be clear if we consider the difference between the relation to the thing had by its likeness in our intellect and that had by its likeness in the divine intellect. For the likeness in our intellect is received from a thing in so far as the thing acts upon our intellect by previously acting upon our senses. Now, matter, because of the feebleness of its existence (for it is being only potentially), cannot be a principle of action; hence, a thing which acts upon our soul acts only through its form; consequently, the likeness of a thing which is impressed upon our sense and purified by several stages until it reaches the intellect is a likeness only of the form. Sed similitudo rerum quae est in intellectu divino, est factiva rei; res autem, sive forte sive debile esse participet, hoc non habet nisi a Deo; et secundum hoc similitudo omnis rei in Deo existit quod res illa a Deo esse participat: unde similitudo immaterialis quae est in Deo, non solum est similitudo formae, sed materiae. Et quia ad hoc quod aliquid cognoscatur, requiritur quod similitudo eius sit in cognoscente, non autem quod sit per modum quo est in re: inde est quod intellectus noster non cognoscit singularia, quorum cognitio ex materia dependet quia non est in eo similitudo materiae; non autem propter hoc quod similitudo sit in eo immaterialiter: sed intellectus divinus, qui habet similitudinem materiae, quamvis immaterialiter, potest singularia cognoscere", Thomas Aquinas, *De veritate,* q. 2, a. 5co. Latin text in [1].

That notwithstanding, in some relevant and not so rare circumstance, the *whole* may reveal an *order* and an *organized structure,* capable to perform special activities (*operations*) as it happens, *e.g.,* in biological living systems, or even in some physical and chemical *complex systems.* At present we do not know any *compressed string* (*law* or *algorithm*) capable to generate the actual sequence of the genetic code of a living being and we are compelled to list its individual elements one after the other as if they were provided randomly by nature. Something similar happens in the context of arithmetic when we deal with *prime numbers*, the sequence of which appears randomly distributed into the ordered set of natural numbers.

An intensive discussion is animating the scientific world about the logical consistency of the idea itself of a *theory of everything.* A relevant example of different opinions about the matter is offered, *e.g,* by the contemporary debate between Stephen Wolfram and Gregory Chaitin. Wolfram is convinced that

> in the end it will turn out that every detail of our universe does indeed follow rules that can be represented by a very simple program – and that everything we see will ultimately emerge just from running this program ([5] pg 545).

Wolfram's conviction seems to arise by his deep experience with *cellular automata,* which may evolve into very *complex structures,* even being governed by very *simple algorithmic rules.*

> In the existing sciences whenever a phenomenon is encountered that seems complex it is taken almost for granted that the phenomenon must be the result of some underlying mechanism that is itself complex. But my discovery that simple programs can produce great complexity makes it clear that this is not in fact correct. And indeed in the later parts of this book I will show that even remarkably simple programs seem to capture the essential mechanisms responsible for all sorts of important phenomena that in the past have always seemed far too complex to allow any simple explanation.
>
> It is not uncommon in the history of science that new ways of thinking are what finally allow longstanding issues to be addressed. But I have been amazed at just how many issues central to the foundations of the existing sciences I have been able to address by using the idea of thinking in terms of simple programs ([5] pg 4).

While on the contrary Chaitin considers *random strings* (*incompressibile strings*) as admissible in nature as *indecidable propositions* exist in an axiomatic system.

> Wolfram has a very different view of complexity from mine. [...] Wolfram's view is that simple laws, simple combinatorial structures can produce very complicated unpredictable behavior. π is a good example. If you didn't know where they come from its digits would look completely random. In fact, Wolfram says, maybe the universe contain non randomness, maybe everything is actually deterministic, maybe its only pseudorandomness. And how could you tell the difference? The

> illusion of free will is because the future is too hard to predict but its not really unpredictable ([6] pg 113).

In the present report we try to show how some *complex* – even if relatively simple or simplified – *ordered structures* may arise:

– either by already *ordered initial conditions*

– or by *random initial conditions,* provided that a suitable information (*law/algorithm*) is assigned, governing the evolution of the system.

When such law (*shorter string*) is not known the full list (*uncompressed*[2] *string*) of the elements involved in the structure of the system must be assigned in order to simulate the generation process of the system either assigning *sequentially ordered* initial conditions or *random* initial conditions.

We emphasize that the probability to reach a final ordered structure simply starting from random points, if some information coded in a suitable *string* (it does not matter if compressed or incompressible) is not provided is practically zero in presence of a great number of elements as in our universe. So we could say that order (*information*) may be hidden within *chance* but not completely suppressed.

Ordered structures appear as *attractors* towards which the generation process tends even when the initial conditions are chosen randomly within a *basin of attraction.* An intriguing investigation might be based on the conjecture that universe could be modeled as a huge attractor involving minor nested attractors (the galaxies) nested in turn with inner attractors along a chain reaching living individuals and so on, down to cells, molecules, atoms and elementary particles.

At present we do not know if living systems and some physical complex systems are fully determined by *algorithms* (*compressed strings*) or, at least, some of their properties are governed by full lists of instructions (*incompressible strings*).

But we may guess that the light differentiations among the bodies of individuals of the same species can be generated, when genetic code is identical (as it happens for true twins), thanks to the random bifurcations own to the non-linearity of the *laws* governing body generation.

In our report we show and discuss some examples of *ordered structures* one is able to generate by simple computing programs, either coded into *compressed strings,* or into *uncompressed data files.*[3]

The exposition will be developed according the following order.

– Chapter 1 will introduce didactically some of the nowadays topics about the evolution of the notion of information which was born in communication engineering and now

[2] May be that in future one will be able to compress such a string thanks to some algorithm or, on the contrary, will be able to show its incompressibility.

[3] At present we do not know if those data files are to be considered incompressible or if they might be compressed in future thanks to a non-trivial algorithm, *i.e.,* a global mathematical law and not by local shortcuts similar to those employed to reduce graphic files dimension.

has become relevant in physics and mainly in biology. On an interdisciplinary viewpoint some meaningful philosophical reflections by some authors will be quoted, adding also our own reflections, in order to compare contemporary notion of *information* and *Aristotelian form.*

– The other chapters of the report will be devoted all to show how several examples of *ordered structures* may be generated following either *ordered processes* or *random processes,* provided that we know enough *information* (*law*) to characterize their own properties. Each example will be presented both by images and by the related programming code written in *Python 3* or in *POV-Ray 3.7* ray tracing language.

  In particular we consider each structure:

  = as a *whole,* when it just appears complete as a final result of some code execution (*algorithmic information*);
  = as *generated sequentially,* point by point, according to an *ordered* assignment of the initial conditions of a computing process driven by the same *information*, so that it results that *order arises from order*;
  = as *generated randomly,* as an *attractor* defined by the same *information* as before, but starting from initial conditions chosen *completely by chance* within a suitable basin of attraction;
  = as *generated by a cellular automaton,* the replication rule of which is provided by the same *information* as in previous examples, while the choice of the contiguous place where each daughter cell is be generated is random.

– Chapters from 2 to 7 will be devoted to examples arising from *fractal geometry* (in particular Mandelbrot sets, Julia sets, and Newton's method sets). In detail:

  = in chapter 2 we examine $2D$ fractal structures;
  = in chapter 3 we deal with different techniques to accomplish the passage from $2D$ to $3D$ shapes either as wholes or as generated point by point;
  = in chapter 4 we show how to obtain *fractal landscapes,* which are the simplest way of $3D$ fractal rendering, since the third dimension (height) is given simply by a suitable function of the *escape rate* characterizing the process generation;
  = in chapter 5 we attack the main difficulty in representing a genuine fractal shape in three dimensions. The solution is provided by assigning lower and upper limit conditions in order to identify a fractal shell to be built and represented point by point. The chapter will deal with cylindrical symmetry shapes obtained through a complete rotation of a $2D$ fractal around a symmetry axis. The rendering quality of images will be greatly increased employing a ray tracing language as *POV-Ray 3.7,* rather than *Python 3*;
  = in chapter 6 we apply the methodology proposed in the previous chapter to more general (*i.e.,* even non-cylindrically symmetric) $3D$ shapes, generated recurring to *quaternions* and *hypercomplex numbers.* The graphical results are fascinating;

= in chapter 7 we present our main original result which is related to the generation of $3D$ shapes by *cellular automata* with random choice of the daughter cell near location. The chapter applies the methodology to $3D$ fractals.

– The last two chapters of the report are devoted to applications to biology. In particular:

= in chapter 8 we propose a sort of rough model of a human heart external shape generation, cell by cell, following some of the methodologies applied to fractals in the previous chapters;

= in chapter 9 a more realistic heart external shape model of cell by cell heart generation by cellular automata is finally presented, based on a public data list (non compressed string) one can find on the web.

– Some conclusive remarks end the report.

Chapter 1

Today's information towards Aristotelian form

New perspectives on information in mathematics, physics and biology

In this first chapter we intend to introduce some elementary notions about the increasingly meaningful role of *information* in the context of the *biological sciences,* starting from the early decades of the 21st century.[1] A role which seems to involve both the question on the evolution of species and the matter of the emergence of life. In a wider sense information is playing a significant role in *order emergence* (self organization) of the *structure* and the *dynamics* of physical and biochemical complex systems.

Nowadays we can see biologists, non-linear systems physicists, computer scientists and philosophers collaborate in a same research group in order to investigate new simulation models and theories about emergent life, organ formation in a body and mutations of species. Most of these topics involve relevant philosophical problems related to the possibile and unavoidable quest for an ontological interpretation of such theories beside to suggest heuristic paths orienting the research.

People are now especially interested in proposing some definition of *information* which is more fundamental and relevant than the traditional one arisen in the field of noise free communication engineering. Significant steps have been carried out thanks to the analogy recognized between *information* and *negative thermodynamic entropy,* when non-equilibrium thermodynamics of open systems exchanging matter energy and information with the environment was developed by several authors. In the latter context the emergence of *ordered structures* within the physical open thermodynamic systems, governed by a sort of theleonomic dynamics, has oriented the researches to test how such thermodynamical systems could provide models for biological organisms and life emergence. Meanwhile the non-linear mechanics of dynamic systems discovered the existence of the *attractors, i.e.* solutions towards

[1] I dealt with this subject for the first time in the last chapter of my book [7] to which the present chapter is largely inspired.

which all the trajectories, the initial conditions of which belong to a suitable *basin of attraction*, tend for time increasing values. Such attractors may be *stable* or *unstable* depending of the parameters characterizing each of them and may switch from stability to instability in correspondence to the parameters value switching. A comparison between a similar behavior and and the change from life to death of a living system was considered as straightforward. Moreover some properties of a non-linear system appeared as *global* (holistic) and not reducible to a sort of summation of more elementary *local* (reductionistic) properties.

So the idea that some *information* characterizing the *structure* and the *dynamics* of the *whole*, which is not deducible starting from the properties of its single *parts* as if they were independent of the whole, suggested quite naturally to compare our contemporary notion of *information* with the ancient but always fascinating notion of Aristotelian *form*.

Those ideas have been applied also to the *species* of living beings and not only the *individual* and the question

- if a sort of *information* may somehow orient the evolution of species, involving *attractors* and *repellers,* even if the initial conditions are determined by chance
- or if only chance and natural selection are enough to explain evolution.

At present two schools of thinking are in competion:[8]

- The former school defends a neo-Darwinian position according to which the only *random genetic mutations* are enough to explain an evolution improving the qualities of species by spontaneous emergence of *new information*.
- The latter, on the contrary, suggests that chance may not be enough to explain a gain (evolution) in the level in species, since an adequate *cause* is required in order to activate the emergence of *new information* from the *potentiality of matter*[2] as, in a greatly different historical and cultural context Aristotele proposed.

Therefore an increasing interest in Aristotelian doctrine of *form* appears today no more so peregrine as it was only until some decades ago.

Surprisingly *experimental investigations* and mainly *computer simulations* provide relevant results supporting the ideas of the second stream of thinking.

- In fact simulations show that the great majority of random mutations are not of advantage for the species since they do not improve the ability to survive of the mutant individuals and only very few do. Moreover a sort of increasing *genetic entropy* accompanies mutations which destroys information rather than increasing it. A situation resembling the behavior of thermodynamic entropy the increasing of which, according to the second principle, decreases the power of heat in order to be transformed into mechanical work. Random genetic mutations cause more *disorder* (loss of information) than *order* (organization).

[2]See[8] *General Introduction*, pgs XIII-XIX.

– Moreover the mutations result not to be genetically permanent, since they disappear in the descendants after few generations. In practice it has been shown that a *threshold* (minimum number of mutant individuals) exists under which the effect of mutations (either damaging or improving) extinguish after few generations.[9]

– Computer simulations, at least until now,[3] has provided results which seem not to be favorable to a *merely random mechanism* of a process improving the species.

Then the researchers have been induced to examine in more depth the notion of *information* as a new *immaterial factor* playing an essential role, even if not yet well understood, either in governing the evolution of species and the birth of life and the emergence of an ordered structure in complex systems. In this framework at least two main problems arise.

– How to *define* information and how to try to provide a *model* of information behavior?

– Which is the *cause* of emergence of information in material systems (*i.e.,* systems carrying mass and energy)?[4]

The *reductionistic* and materialistic approach attempting to explain *information* as a mass-energy phenomenon, identifying it with its material carrier, has been just universally recognized as inadequate to describe experience. As a matter of fact we experience every day how information can be transferred from some material support to any other one, without alteration of its informational content. Since the early times of telecommunications and cybernetics it appeared as an evidence what Norbert Wiener (1894-1964), one of the fathers of information theory, said:

> Information is information, not matter or energy. No materialism which does not admit this can survive at the present day ([12] pg 132).

1.1 Heuristic operative definitions of information

We can easily recognize an increasing progression along the history of the attempts to achieve a proper definition of *information*. Starting from the early purely *descriptive* definitions, based on a physical and statistical approach as it was suggested by a comparison with thermodynamics and statistical mechanics, further steps were made in order to formulate more abstract and *causally explicative* definitions. In literature we may find references at least to the following kinds of theories of *information* and related definitions:

– *the classical theory of information*;

– *the theory of specified complex information*;

[3] For instance we may mention computer programs like *Tierra, Mendel* and *Avida* simulating random mutations involved in species evolution.[10] [11]

[4] According to an Aristotelian way of speaking we could say: *which is the adequate cause of the eduction of the form from matter potency?*

– *the algorithmic theory of information*;

– the *universal theory of information*;[13]

– la *the pragmatic theory of information* which is concerned to the cost of the machineries and networks required to process information.[14]

Here we will examine some of the previous definitions which seem to be more relevant even for their philosophical implications.

We remark that none of those definitions appear to be exhaustive. So their approach to the notion of *information* is to be considered as heuristic, operative, in progress, in oder to attempt to reach a more deeply *essential*[5] definition. Together to all those efforts it proves to be useful and somehow clarifying to take into account also the notion of Aristotelian *form*.

1.1.1 The classical theory of information

The classical theory of information, originally own to Claude Shannon (1916-2001) is based on the statistical mechanics with the engineering purpose of softening as possibile undesired signals (*noise*) emphasizing, on the contrary, the carrier of the relevant information to be transmitted by a sender to a receiver. For this aim it proves to be enough and very efficient to restrict the object of investigation to the *syntactic* statistical analysis of the material *symbols* required to data transmission, *e.g* along cables or aether, their storage into physical memories and their processing.

The basic original idea of Shannon was that of relating the notion of *information* to the probability of some event to happen or not, the behavior of which seemed to him very similar to that of the *negative thermodynamic entropy*. So he conjectured a definition of *information* as:

$$I = -\log_b P, \tag{1.1}$$

I being interpreted as a measure of information, P the probability of the event occurrence and b the basis of the numeric code employed.

If a very likely event happens we gain a very law information. On the contrary if it does not occur (or its contrary happens) we are more informed and almost compelled to a deeper investigation about that phenomenon. The formula is assumed to be the same as that which characterizes the thermodynamic entropy, except for the minus sign ([15] §IV).

1.1.2 The theory of complex specified information

The *theory of complex specified information* proposed by William Dembsky (b. 1960) adds to the classical information theory a sort of *finality criterion* orienting chance to reach some

[5] When we say *essential* we mean a definition catching what information is properly in itself and not only when it is related to aspects coexisting with it, like the string by which it is coded, the different kinds of memories on which it is stored, the costs required for it to be processed, and so on.

result at the end of a process. The main problem a similar approach is the lack of a mathematical or symbolic formalization of that theleonomic factor which remains an extrinsic philosophical conjecture. Therefore such a theory is often evaluated as non-scientific as the entire approach of the so called *intelligent design* ([13] pg 17).

Finality may enter legitimately within a scientific theory if it results to be a part or a consequence (*e.g,* as a mathematical solution) of the laws (equations) governing a complex system (physical, biological or other, [16] §VI, 1). In any case, the existence of an element external to a theory needs to be demonstrated as a logical consequence of the internal axioms of the theory itself, which is required to avoid internal contradictions.

1.1.3 The algorithmic theory of information

At present it seems to me that the approach of the *algorithmic theory of information,* adequately enriched by a *semantic* interpretation and content, is the most promising one, for the development of a mature scientific theory of information contributing to physics of complex systems and biology, and even to philosophy.

The *theory of algorithmic information,* proposed and enriched by Ray Solomonoff (1926-2009), Andrej Nikolaevi Kolmogorov (1903-1987) an Gregory Chaitin (b. 1947), is concerned with *complexity* – as it is suitably defined within the theory itself – of the *symbols* involved in data and object structures.

First of all a definition of *algorithm* is required.

> An *algorithm* is a sequence of operations capable of bringing about the solution to a problem in a finite number of steps ([15] §V).

Such definition is enough wide to host different kinds of algorithms involving different levels of information, progressively approaching to the Aristotelian notion of *form*. We will examine, by means of some examples, the methodological and epistemological relevance of the corresponding different levels and some implications for biology, foundation theory and even philosophy.

1.2 Some examples of algorithms

We limit ourselves to three simple well known examples of algorithm emphasizing the different *level of information* involved in each one.

- The *first level* consists in a simple *sequence of operations* to be executed in order to solve some problem. In this case the kind of information involved is merely *operational* and does not involve any sort of definition of some entity. On an Aristotelian-Thomistic point on view it looks like the description of an *accidental mutation* of some entity built as a cluster (aggregate) of substances which is not endowed of a unique *substantial form*.
- The *second level,* as we will see, is ontologically more relevant, since it actually *defines an entity* determining its *structure*. Philosophically we can say that the *information*

involved in the algorithm properly defines the *essence* of an entity, just as an Aristotelian *form*.

– The *third level* also *defines* an entity characterizing the *dynamics* which generates its *structure*, rather than defining immediately the *structure* as a *whole*. According to the Aristotelian-Thomistic terminology we say that the *information* involved in the algorithm specifies the *nature* of the generating information.

Let us now examine those examples.

1.2.1 Algorithm to exchange the liquid contained in two different glasses

Let us consider two glasses, say A and B, filled respectively of water and wine. We want to transfer the water from A into B and *vice versa*.

$$A, B \to B, A$$

The problem is easily solved with the aid of a third empty glass C. Then the required algorithm is the following:

1) pour the water contained in A into C: $A \to C$,
2) pour the wine contained in B into A: $B \to A$,
3) pour the water now contained in C into B: $C \to B$

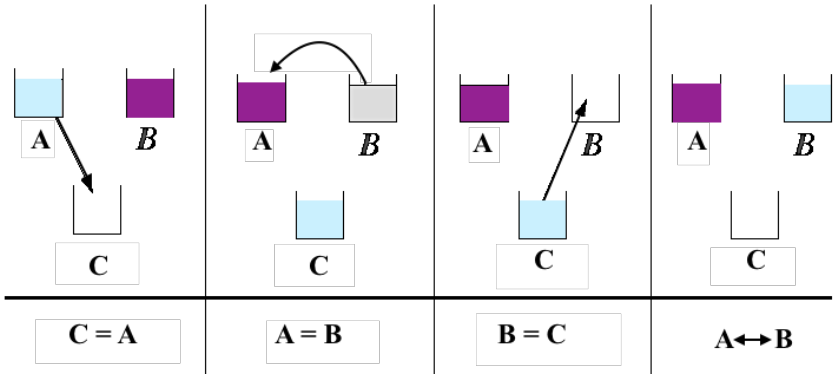


Fig.1 - Exchange the liquid contained in two glasses

At the end of the procedure the desired exchange will result. The water which was into A will have been transferred into B and the wine originally in B will be now in A.

The algorithm, simply, describes an *operative* procedure which provides a mutation (*becoming*), while it does neither define nor give consistency (*being*) to an entity.

Let us now examine a second kind of algorithm which, on the contrary, is actually able to *define* the structure (*essence*) of a new entity.

1.2.2 Algorithm to generate a *fractal*

Roughly speaking we can characterize a *fractal* as an infinitely rippled curve or surface the level of complexity of which is preserved at any magnification scale.[6]

What is remarkable is the circumstance that the mathematical computation generating a fractal, beside providing an operational procedure, properly *defines* and in the same time *actuates* constructively its entity.

Among all fractals we choose here, as an example, a typical *Julia set* (the *dragon*). The algorithm is the following.

- We consider a complex number;[7] $z_0 = x_0 + i\,y_0$ the real part (x_0) and the imaginary part (y_0) of which run inside a suitable interval: $[-l, l\,]$;

- we choose another complex number $c = a + i\,b$ which is maintained constant along the whole procedure, as an identifier of the *Julia set* itself. In the example of fig. 2 we have set $c = 0.27334 + i\,0.00642$;

- we define a sequence of complex numbers $z_n = x_n + i\,y_n,\ n = 0, 1, 2, \cdots$, the initial term of which is just z_0 and each next number is obtained adding c to the previous one squared. We have so the recurrence rule:

$$z_{n+1} = z_n^2 + c, \tag{1.2}$$

- we take the sum of a significantly high number of subsequent terms of the sequence;[8]

- At the end we evaluate the absolute value h of the sum obtained:[9].

$$h = \left| \sum_{k=0}^{n} z_k \right| > R. \tag{1.3}$$

  If h is greater than a suitable value R, before established, we paint on a computer display a pixel of co-ordinates (x_0, y_0) with a precise color (or respectively a gray level) of a suitable color map (or grayscale).

Manifestly the algorithm beside providing an operating procedure *defines* essentially the *structure* of a new entity, namely a *Julia set,* while constructing it.

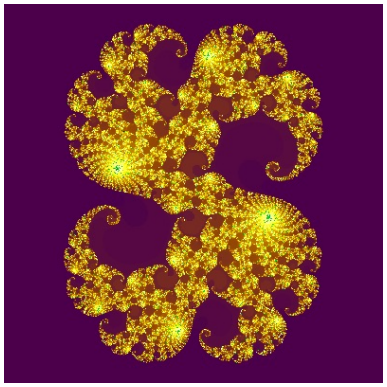
[6] Fractals are more precisely classified considering their *fractal dimension,* a measure of the fraction of plane or space they fill when they are considered as wholes. One may see, *e.g.,* my *Fractal Gallery* (www.albertostrumia.it/?q=content/galleria-di-frattali-fractal-gallery) beside several papers and books with astonishing pictures of fractals.

[7] We remember that a complex number has the general form $z = x + i\,y$ where x, y are two real numbers and i is the imaginary unit, *i.e.,* a number the square of which is, by definition, -1.

[8] In principle the infinite series of all the terms of the sequence should be taken. In practice, on a computer a finite number of terms can be added. The greater is the number, the better will result the details in the picture.

[9] We remember that the absolute value or *modulo* of a complex number $z = x+iy$ is given by $|z| = \sqrt{x^2 + y^2}$.

$$
\begin{array}{ccccccc}
 & & & & & & z_0 \\
 & & & & & & \downarrow \\
 & & & & z_1 & = & z_0^2 + c \\
 & & & & \downarrow & & \\
 & & z_2 & = & z_1^2 & + & c \\
 & & \downarrow & & & & \\
z_3 & = & z_2^2 & + & c & & \\
\cdots & & \cdots & & \cdots & & \cdots
\end{array}
$$



$c = 0.27334 + i\,0.00642$

Fig.2 - Generation of a Julia set

1.2.3 Algorithm to determine a fractal *basin of attraction* of a chaotic magnetic pendulum

Our third example is provided by physics rather than mathematics. It consists also in a fractal set the *structure* of which results as an effect of the chaotic *dynamics* governing a magnetic pendulum driven by three magnets located in the vertices of an equilateral triangle.

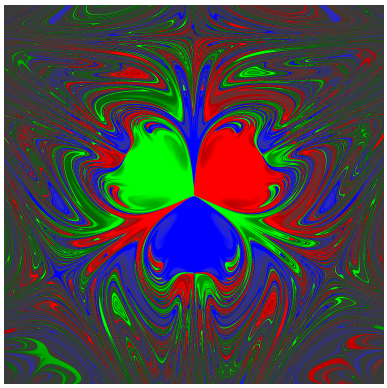


Fig.3 - Fractal basin of attraction of a chaotic magnetic pendulum

The motion of the pendulum appears to be *random* at all when it is observed at some time interval and with no regularity or order. Each trajectory seems to end onto one of the magnets without any choice criterion. That notwithstanding the *dynamics* is driven by a precise *information* arising from the laws of physics, since the arrival magnet depends exactly on the starting point from which the pendulum is initially released.

The pendulum dynamics being *complex* – determined by *non-linear* laws – it results to be strongly *sensitive to the initial conditions*. The starting point being even slightly displaced,

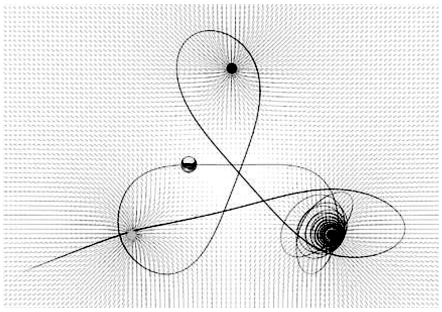


Fig.4 - A chaotic trajectory of a magnetic pendulum

the arrival magnet may change. So the *basin of attraction* (set of the initial conditions) related to the dynamics of the pendulum, exhibits a quite precise fractal *structure*.

We point out that, in the present example the *information* which determines the fractal *structure* of the *basin of attraction* is determined through the *dynamics* of motion.

Graphically the fractal basin is painted assigning distinct colors (or gray levels) dependent on the arrival magnet of the pendulum.

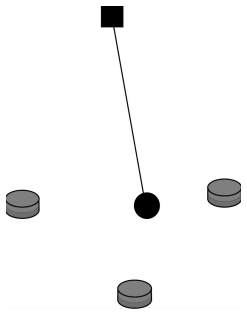


Fig.5 - Sketch of a magnetic pendulum

1.2.4 Remark

We want to emphasize, now, that people investigating *algorithmic information* are generally interested in defining the *quantity of information* involved into a computer program algorithm, which is viewed simply as a *code string*. Therefore a string program which solves some problem is considered as more rich of information as shorter is its code string. A matter involving a pragmatic instance of efficiency, minimizing time machine and then costs of program running.

But it is known that not any problem is *computable,* since a string including an infinte number of characters, in many cases, cannot be *compressed* into a shorter one. Moreover also strings including a finite number of characters often cannot be compressed into a shorter one.

In the language of set theory a similar circumstance arises because only a class of sets may be defined by a *law* (*shorter string*) according to which their elements are generated,

thanks to the *replacement axiom.* All the remaining sets can be defined only listing their elements one by one (*incompressible string*).

Within the frame of Gödel's theorem we can see the same problem as a matter of *decidable propositions* which correspond to a computable Gödel's number and *undecidable propositions* which are related to non-computable Gödel's numbers. This is what one means when says that not all numbers are *computable,* since it does not exist a formula (*shorter string*) enabling us to evaluate all their digits avoiding to list them one by one.

As a consequence, attaining physical dynamical systems, and especially biological and cognitive ones, we know that not all their activities are computable. So the irreducible *qualitative* and properly *ontological* aspects of their behavior has acquired a great relevance even on a scientific point of view beside their philosophical importance.

Many of those non computable aspects concern *information* and related *algorithms.* A *semantic* approach seems now to be required beside the purely *syntactic* one developed in the classical information theory. Because the algorithm, as here is intended, is no longer simply identified with the *string* on which it is coded – sum (*whole*) of the characters (*parts*) of which it is composed – rather being a *definition,* actualizing the *dynamics* of some resulting new *entity.*

Rather such a definition is a *logical law* defining an entity and a sort of *ontological form/information* actuating its *structure* (*essence*) and its *dynamics* (*nature*).

We emphasize that such a notion of algorithm, together with the previous *philosophical* interpretation seems to reveal a first but non-trivial rigorously scientific attempt to approach the *definition/essence* of the entity the *structure/organization* and the *dynamics/nature* of which are generated by the algorithm itself. We remember, in fact, that according to the Aristotelian-Thomisitic ontology the *nature* is just the *essence as principle of acting.*[10]

So an *algorithmic information* involves more of philosophical content than some mere *quantitative measure* of information. Scientific investigation on information has become aware of this semantic exceeding contribution and is just attempting to grasp it with more and more suitable definitions. Information is recognized to be more than the length compression of a string of code.[11] In the frame of the mathematical physics of non-linear *dynamical systems,* for instance, a relevant approach to *form/information* has been developed following a methodology which is known as *qualitative analysis* of motion. Similar models are applied even in a biological context, in order to model the evolution of species or the emergence of self-organization during the transition from non-living matter to living organisms.

All these research exhibit some non-trivial philosophical relevance since they investigate, as a matter of fact, the *essence/nature* of some entities by means of constructive definitions. Most likely more refined mathematical instruments will be required in order to formalize

[10] "Acting depends on nature, which is the principle of acting (*actio dependet a natura, quae est principium actionis*)", Tomas Aquinas, *In I Sent.,* Lib. 3, d. 18, q. 1, a. 1co; "the word *nature*, so considered, appears to mean the *essence* of something, in order to its proper action (*nomen autem naturae hoc modo sumptae videtur significare essentiam rei, secundum quod habet ordinem vel ordinationem ad propriam operationem rei*)", *De Ente et Essentia,* chap 1 (See[1]).

[11] Among the first mathematicians who approached in a rigorous way the problem of characterizing the *information,* according to a careful comparison with the Aristotelian *form,* we have to mention René Thom (1923-2002), of whom we cite his famous book[17].

adequately *information* as a sort of algorithm and mathematics itself will widen as a true *theory of entities* (*formal ontology*). So information could involve both computable aspects and non-computable ones.

1.3 Main characters of information

It is now relevant to show which are the main characters of *information* which have been caught by the different theories, as they have been developed in science. In particular we will be able to recognize a progressive approaching between the scientific definitions of *information* and some aspects of the Aristotelian-Thomistic notion of *form.*[12]

In particular we are able to identify some of its proper elements (*formally* defining characters) and some other elements required to its carriers, *i.e.,* material supports ([13] pgs 13-17).

i) *Code and syntax*: at a first level of *Shannon's communication theory*[18] we find, first of all, the presence of a code, a symbolic *alphabet* allowing to tie (*i.e.,* to write) information onto some *material support* which is needed for it to be carried. Moreover, since any alphabet requires to be governed by suitable rules, a *syntax* is to be added so that the alphabet becomes useful to code information.

So we will have:

– a set of conventional *symbols* called the *alphabet*;

– a set of conventional *rules* which must be enough to state what is allowed in organizing the symbols, which we call the *syntax.*

That not withstanding information, in itself, is independent of the matter medium across which it is traveling, which only allows to it to be carried. Any carrier can be exchanged with another carrier of just the same information. And the carrier, as it is, cannot produce any information by itself, neither as *efficient* cause, nor as *formal* cause, nor as *final* cause.

ii) *meaning*: meaning is the essential attribute of information as it is coded into a language in order to its *communication* (*i.e.,* transfer and interpretation).

– The *words,* either they are written or spoken, may be used used to symbolically represent entities of any kind: events and/or concepts, *i.e.,* everything.

– Moreover (this is the relevance of symbols) the signified entities need not to be physically present together with the words, since they take their place, representing them and communicating something about them just as if they were actually present.

[12] In Aristotelian-Thomistic view by *form* one means an immaterial principle acting in such a way that an entity is what it is and nothing else. On a *logical* viewpoint it identifies the so called *metaphysical definition* of an entity; on a *metaphysical* viewpoint it identifies the *structure* and the *dynamics* of an entity so determining its *nature.*

– Experimentally it has always been observed, until now, that chemical and physical (*i.e.,* purely material) processes, as such, are unable to perform any symbolic substitution. We mean, here, material processes which are not driven by some external control system informing the behavior of the process itself.

iii) *Expected action*: information appears as something which is sent by a *sender* in order that a *receiver* executes a precise operation to achieve some goal.

– The *receiver* starts operating soon after reading and decoding the message. In some situations the sequence of the operations may be even very long and difficult to be executed.

– The *receiver* may be required to *decide* if the operation is to be executed or not, completely or only partially. If the decision is "yes", the operation will be executed as required by the *sender*. In particular two kinds of receivers are to be distinguished, *i.e.*:

= an *intelligent* and *free* receiver, who is able to understand the *meaning* of the message;

= or a *machine* which is unable of understanding and freely choosing.

The former, being intelligent, can answer the sender's request according to a freely choice among several different strategies. The latter, being an *automatism,* is totally driven by the control program. In both cases machines may be necessary to perform the required operations.

iv) *Intended purpose*: before the message is sent the sender needs some internal mental process motivating him to formulate and send the message as such.

– This process is generally highly complex and involves some need, motivation and will that something is received and executed by somebody/something else.

– in particular, when the operation is so hard that it could not be executed by any receiver, the sender must carefully evaluate if the chosen receiver is adequate to perform the duty.

– If the whole process is successfully performed the sender's intent will be achieved satisfactorily.

– So the sender's intent appears to be essentially at the origin of the message.

– The receiver's success in executing the sender's intent is the result of the entire operation of communication of information.

The previous four attributes seem to be required to characterize unambiguously the notion of *information*. Therefore a possible formal definition of *universal information* (UI) must include them all. Here is such a kind of definition: ([13] pg 16)

> A symbolically encoded, abstractly represented message conveying the expected action and the intended purpose.

In order that a similar definition may become scientifically employable we need to formalize it, in turn, into a suitable *symbolic language,* so that we are able to use it in *computations* (as for *computable* matters), or in the frame of a *qualitative analysis* (as for *non-computable* matters).

It is interesting to follow Stuart Kauffmann's (b. 1939) remarks on a progressive deeper approach to the notion and the theory of *information.*

> I begin with Shannon's famous information theory. Shannon chooses, on purpose, to ignore any semantics, and concentrate on purely syntactic symbol strings, or messages over some pre-chosen symbol alphabet [...].
>
> It is clear that Shannon's invention requires that the ensemble of all possible messages [...] be stable head of time. Without this statement, the entropy of the information source cannot be defined. Now let's turn to evolution. We saw above that we cannot pre-state the adjacent possibilities of the evolution of the biosphere by Darwinian preadaptations. Thus, we cannot construct anything like Shannon's probability measure over the future evolution of the biosphere [...]
>
> The same concerns arise for Kolmogorov, who again requires a defined alphabet and symbol strings of some length distribution in that alphabet. Again, Kolmogorov uses only a syntactic approach. Life is deeply semantic with no pre-stated alphabet, no source, no definable entropy of a source, but unpre-statable causal consequences which alone or together may find a use in an evolving Kantian whole of a cell or organism.
>
> In summary, standard information theory, both purely syntactic and requiring a pre-stated sample space, is largely useless with respect to evolution. On the other hand, there is a persistent becoming of ever novel structures and processes that constitute specific novel and integrated functionalities in the [...] wholes that co-create the evolving biosphere. [...]
>
> We need a new theory of embodied functional information in a cell, ecosystem or the biosphere.[19]

1.4 Emergence and evolution of biological information

The relevant interest in the role of *information* in biology raises at least three main questions in the context of scientific research.

– The first question is related to the *emergence*, or the *origin* of *biological information.*

  According to an Aristotelian terminology we should talk of *eduction* of the *form* from the potency of matter. So the problem for the search of an adequate *efficient cause* in order to obtain such an eduction arises each time a *substantial mutation* transforming some entity into another one happens in a stable way.

In the contemporary scientific context this matter is often viewed as the problem of *information production* or *information increment* within some system (physical, biological, etc.). There is a tendency to guess that information may be produced or increased *spontaneously,* without an adequate causation, thanks to self-organization capability of the system itself, arising by a sequence of random events.

– A second question, which is strictly tied to the previous one, is related to the *evolution* of information, *i.e.,* its mutation in time. In particular its spontaneous increment within some system, especially a living system.

– The last question attains the problem of *coding* and *copying* biological information. Clearly biological information is no longer considered as residing only in the *DNA* code. Rather it appears as layered at several levels, even on the same biochemical, electrochemical or, generally, physical medium.

The assumption that life complexity is only a spontaneous result of *non-linearity* of *chaotic systems,* has been shown to be incompatible with the numerical mathematical simulation models implemented on a computer, starting from their governing equations ([20] and related bibliography).

> The explosion in the amount of biological information [...] requires explanation ([22] pg 204).

The useful *non-ambiguous beneficial mutations* (*i.e.,* non-damaging at any level) arising from natural selection, result to be extremely rare. Chance seems not to be enough to generate improvements without an adequate cause.[21]

On the contrary a process of *loss of information* (*genetic entropy*) is revealed, because of *deleterious mutations* which result to be the most likely mutations. So a sort of defensive barrier, conservative of complexity stability appears.[11]

As to biological information coding scientists has observed that the *genetic units* consist in very precise instructions, coded in such a rich language that "any gene exhibits a level of complexity resembling that of a book" ([22] pg 203). More languages (*genetic codes*) are present in the same genoma, with multiple levels (even three-dimensional), coding biological information, forming a network with several layers.

Computer simulation models did not succeed in attempting to explain neither the emergence nor the increment of information, not withstanding both computer programs and the human genoma exhibit very resembling repetitive code schemes.[23]

Information is responsible of *organization* and *order* emergence within the structure of a system, so that information increasing implies order increasing. What numeric simulation – based on statistical mechanics and non-equilibrium thermodynamics – show, on the contrary, is that order is not spontaneously generated within the system, even if this latter is *open* (being able of exchanging matter and energy with the external environment). Information appears in a system, only in presence of a *causal agent* external to the system acting on it.

> If an increase in order is extremely improbable when a system is closed, it is still extremely improbable when the system is open, unless something is entering which makes it not extremely improbable ([24] pg 174).

The process of *self-organization* is activated thanks to the action of such an *efficient/formal cause,* which resembles to what, according to the Aristotelian-Thomistic theory is called *eduction* of a *substantial form* from the potency of matter.

Starting from our recent knowledge on the physics of non-linear systems and the thermodynamics of non-equilibrium governing open dissipative systems, attempts are made in order to model the process of *information emergence* form matter (emergence of an organized structure in matter) by means of stable *attractors.*

The dynamics of those attractors, not withstanding it appears chaotic and dominated by chance, is able to construct ordered structures. In fact the phase trajectories, solutions to dynamics, even starting at random from different initial conditions belonging to a *basin of attraction* (which may be even fractal), tend to fill precise regions of the phase space. So a *whole* arises from a confluence of *parts,* which are only apparently separated, being on the contrary non separable from the whole they are building, thanks to an information governing the structure and the dynamics of the process.

Kauffmann's intuition that a new kind of notion of information, which is not merely statistical and syntactical, but involves also the semantic aspects seems to drive research towards the right direction.

In particular the idea that some *asymptotically stable attractor* may be a good information carrier:

- on one side ensures the presence of some information leading to *structured order* emerging within a system;
- on the other side allows that chance play a wide role in the *dynamics* of the system, since the choice of initial conditions of the evolutive trajectories, within the *basin of attraction,* is left to chance without preventing that they all reach asymptotically the attractor itself.

So there does not exist any law in the arbitrary choice of the initial condition of the trajectories with the basin of attraction – the behavior of which may result even unpredictable if the attractor is *chaotic* – but some law exists within the dynamics of the system, involving some finality in its attractor solution. Such a finality (*intended purpose*) is typically a character of *information.*

In principle several *analogous levels* of organization and finality may be obtained *nesting* several attractors into a hierarchy, so that some level of attractors is attracted in turn by a level of higher degree, until some *first universal attractor* is reached, which by definition cannot be attracted further, in order to prevent the occurrence of a logical paradox like that of the *universal set.*

i) A lower *level of organization* could be, *e.g.,* provided by a set of stable attractors representing the *molecules,* the dynamics of which is governed by

ii) an immediately higher level of attractors organizing *e.g., cells,* the dynamics of which is ruled

iii) by an higher level of attractors representing the *organs* of a living system;

iv) a fourth level of attractors shapes the structure and the functionalities of *individual living beings* of different species;

v) a fifth level of attractors will organize the *species* of living beings, and so on.

In principle one could guess, according to such a model of *nested attractors,* the existence of a chain starting at the level of the *elementary particles* and reaching the level of the *universe as a whole.*

The chain is broken when some attractor flips from *stability* to *instability,* because of the occurrence of some accidental *cause* modifying the values of the parameters involved in the law of its level of dynamics. Then it happens that the *second principle of thermodynamics,* locally overcomes with the result of increasing *disorder*: the ordered organization of the system is partially damaged or fully destroyed.

The whole scheme of chained attractors reminds a sort of *fractal structure,* even if it is not necessarily self similar in all its properties. We will be concerned with fractals in some of the next chapters, at least in relation to the aspects involved in our investigation.

At present research is open on these topics and a widened mathematics appears to be required resembling, at some levels, a sort of new version of *ontology,* suitably formalized.

Chapter 2

Two-dimensional structures from algorithms

The roles of order and chance: the example of 2d-fractals

In the first chapter we presented some introductory considerations on *information* in comparison with the Aristotelian-Thomistic notion of *form*. A special attention has been devoted to the notion of *algorithmic information* viewed as a suitable candidate to approach the logical/ontological notions of *definition/essence* regarding the *structure* of an entity and respectively its *nature* on the side of its *dynamics* according to which it can operate and especially the dynamics generating the entity itself.[1]

Two simple didactical examples of fractal generation were enough to show how a suitable assigned (mathematical or physical) *law* can hide the capability of *defining/constructing* an entire complex entity, which exhibits a precise ordered structure like a *Julia set* and a fractal *basin of attraction* of a *magnetic pendulum*.

In the present chapter we will examine in more deepness how *order* and *chance* may enter into the *definition/structure* and the *dynamics/nature* of some kinds of *organized entities*.

2.1 Ordered entity structures generated by ordered processes

2.1.1 Analytic geometry

As a first trivial class of ordered entity structures built by a very simple mathematical law we can consider the entire environment of Cartesian *analytic geometry*.

[1] We remember that, roughly speaking, by Aristoelian-Thomistic notion of *form* we mean a non-material principle organizing the structure of an entity, while by *nature* we mean the same principle as it is able to determine the *dynamics* of the same entity.

In fact *algebraic equations* involving two or respectively three variables (co-ordinates) as:

$$f(x,y)=0, \qquad g(x,y,z)=0, \tag{2.1}$$

or *algebraic inequalities* as:

$$f(x,y)\leq 0, \qquad g(x,y,z)\leq 0, \tag{2.2}$$

are enough to determine univocally a set of points in a Cartesian plane or, respectively, $3D$ space,[2] f, g being functions of the co-ordinates x, y or x, y, z of each point belonging to the set.[3] Equations identify *paths* on Cartesian plane or respectively surfaces[4] in space, while inequalities determine a region of points aside the paths (internal or external if the path and the surface are closed), of the same dimension as the plane or respectively the space.

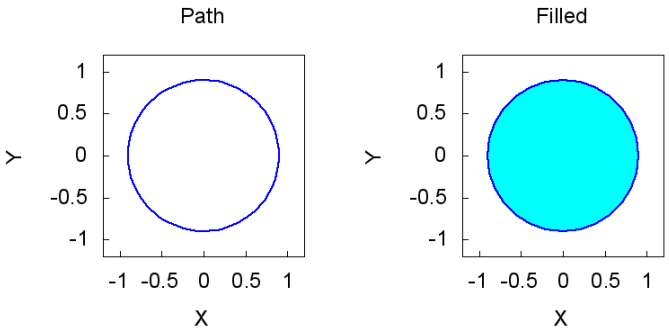


Fig.1 - a) *Path set* defined by the Cartesian equation $x^2+y^2=R^2, R=0.9$;
b) *Filled set* defined by the inequality $x^2+y^2\leq R^2, R=0.9$.

Such structures are essentially simple, *i.e.,* non-complex, since they do not exhibit self-similarity properties, being built by a non-iterative procedure. In this sense we have qualified them as a trivial class of sets. In effect they represent at most a sort of first level of idealized approximation to real bodies.

The interest of even apparently simple structures arises, as it has been pointed out by René Thom, if we consider their boundaries as *singularities* emerging within a continuous body represented by the whole plane, or respectively the whole space[5]. In fact each boundary manifold of Cartesian equation $f(x_i)=0, i=1,2,\cdots,n-1$ (where n is the space dimensionality), represents, for physical bodies, a front across which some physical quantity, as *e.g.,*

[2]More generally also in hyperspace of any dimension.

[3]Of course inequalities like $f(x,y)\geq 0, g(x,y,z)\geq 0$ can be reduced to the form (2.2) multiplying each member by -1.

[4]More generally manifolds in hyperspace of any dimension.

[5]Or hyperspace if we consider higher dimensional abstract spaces.

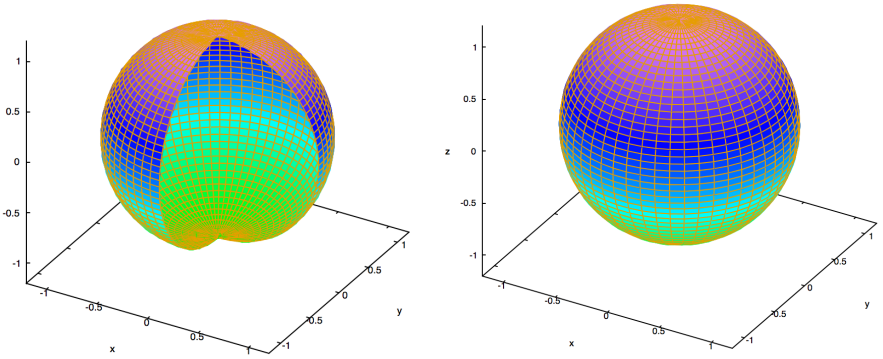


Fig.2 - a) Section of a *surface set* defined by the Cartesian equation $x^2 + y^2 + z^2 = R^2, R = 1.0$;
b) *Filled set (body)* defined by the inequality $x^2 + y^2 + z^2 \leq R^2, R = 1.0$.

mass density, becomes discontinuous so that some body may be retailed and distinguished by the other ones, determining its geometrical *form*.[6].

2.1.2 Two-dimensional fractals in a plane

A more relevant class of structures is provided by *fractal structures,* which can be obtained, generally, by an "adequate" number[7] of iterations of some mathematical law. A typical property of fractals is *self-similarity* (exact or at least statistical). In fact they exhibit repeated geometrical shapes at any scale they may be examined. So it results impossible to decompose them into simpler elementary parts characterized by a lower level of complexity, *i.e.,* to reduce their *fractal dimension*[8] analyzing them at any deeper (in principle even infinitesimal) scale level. A physical limit is imposed only by the computational power of our machines.

Here we are not interested in examining in detail fractals and their properties, but we are interested in emphasizing that generally many of them are generated by procedures that apply a mathematical algorithm (*law*) following an *ordered* sequence of operations, so that an *ordered* (*self-similar*) fractal structure of the resulting entity arises. In this sense we can say that "*order* generates *order*". It is not surprising to obtain *order* from *order*; more surprise

[6]Here the word *form* means mainly *shape* and the related *information* law from which that shape is obtained. The problem has been examined in mathematical detail, with an effort to establish comparison to the Aristotelian notion of *space* and *form* by R.Thom, in [17].

[7]Where "adequate" means, in principle "infinite", and in practice "sufficiently great" in order that self-similarity appears according to the desired detail level.

[8]We remember that fractal dimension is a measure of the fraction of straight line filled by a set of points, or the fraction of plane filled by a curve (increased by one), or the fraction of space filled by a shape, increased by two, and so on. Generally the formula $D = \log N / \log K$ is employed to evaluate *a-priori* or to estimate *a-posteriori* (with the *box counting* method) the fractal dimension of a fractal path. D represents the fractal dimension, N the number of segments with which, at recursion step $n+1$, one replaces the segment obtained at the step n, being divided into K parts.

will arise when we will observe "*order* arising from *chance*" thanks to a *law* hidden into the apparent disorder.

Examples

Typically, the programs generating *Mandelbrot set, Julia sets* and *Newton's method fractals* perform a *sequentially ordered* scanning of a square region of the Cartesian plane applying to the co-ordinates of each point a *recursion law* in order to determine if it belongs to the fractal set[9] or not, and plot to the computer display a pixel of a color corresponding to the numerical result obtained.[10]

In particular *Mandelbrot set, Julia sets* nad *Newton's method sets* are representations on the *complex plane* of the domain of convergence of a complex series, while other kinds of fractals may arise by sequential operations on real numbers.

The degree of complexity of those and similar fractals may depend:

i) on the combined effect of the level of non-linearity of the function (*law*) involved in the recursion procedure;

ii) on the number of iterations of the procedure itself;

iii) and on the number of the control parameters included into the *law*.

An intensively studied *recursion law* has the general form:

$$z_{n+1} = f(z_n) + c, \tag{2.3}$$

where $z = x + iy, c = a + ib$ are complex numbers and $f(z)$ is an assigned complex function. The search for the convergence domain of the series:

$$\sum_{n=1}^{+\infty} z_n \equiv z_1 + z_2 + \cdots + z_n + \cdots, \tag{2.4}$$

leads to fractal sets.[11] Gaston Julia (1893-1978) and Benoit Mandelbrot (1924-2010) studied in detail the simplest non-trivial case when:

$$f(z) = z^2. \tag{2.5}$$

In particular when it is assumed that $z_0 = 0$ and c swaps the entire complex plane, the *Mandelbrot set* is generated, while, on the contrary, when c is fixed, during calculations, at some chosen value and z_0 swaps the complex plane, the *Julia sets* are obtained.

[9] The set being defined as the convergence domain of a suitable series.

[10] When colors are chosen according to suitable color maps the beauty of the picture may result of great effect.

[11] With the exception of the trivial function $f(z) = z$.

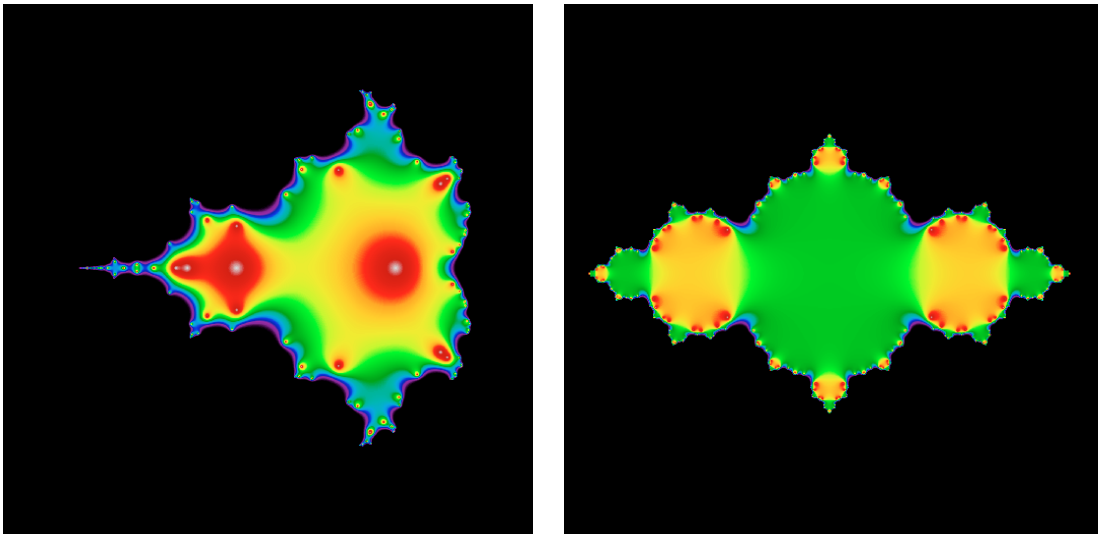


Fig.3 - a) Mandelbrot set; b) Julia set ($c = -0.7454294$)

The *Matplolib* module in *Python 3* provides a very efficient set of instructions to build $2D$ fractals as *wholes*.

Python 3 codes to generate pictures in fig. 3

```
##########################################
# 2D Mandelbrot set with complex arrays
# (matplotlib module)
##########################################

import numpy as np           # import numpy module
import matplotlib.pyplot as plt      # import matplotlib module

n = 8                # set number of cycles
Cx = -.8              # set initial x parameter shift
Cy = 0.0              # set initial y parameter shift
L = 1.7              # set square area side
M = 2024              # set side number of pixels

x = np.linspace(Cx-L,Cx+L,M)      # x variable array
y = np.linspace(Cy-L,Cy+L,M)      # y variable array
X,Y = np.meshgrid(x,y,sparse=True)    # square area grid
Z = np.zeros(M)          # complex starting points area
C = X + 1j*Y            # complex plane area

for k in range(1,n+1):          # recursion cycle
  Z1 = Z**2 + C
  Z = Z1
W = np.e**(-abs(Z))          # smoothed sum moduls

plt.imshow(W,interpolation='nearest', cmap=plt.cm.nipy_spectral)
plt.axis("off")

plt.show()              # plot image
```

```
##################################################
# 2D Julia set with complex arrays (c=-0.7454294)
# (matplotlib module)
##################################################

import numpy as np            # import numpy module
import matplotlib.pyplot as plt      # import matplotlib module

n = 9                # set number of cycles
Cx =  -0.7454294          # set c parameter real part value
Cy = 0              # set c parameter imaginary part value
C = Cx + 1j*Cy
L = 1.7              # set square area side
M = 2024              # set side number of pixels

x = np.linspace(-L,L,M)        # x variable array
y = np.linspace(-L,L,M)        # y variable array
X,Y = np.meshgrid(x,y,sparse=True)    # square area grid
Z = X + 1j*Y            # complex plane area

for k in range(1,n+1):          # recursion cycle
  Z1 = Z**2 + C
  Z = Z1
W = np.e**(-abs(Z))          # smoothed sum moduls

plt.imshow(W,interpolation='nearest', cmap=plt.cm.nipy_spectral)
plt.axis("off")

plt.show()              # plot image
```

We may observe that while the *Mandelbrot set* shape is simply stretched and scaled if z_0 is fixed at a value different from zero (see fig. 4), the *Julia sets* assume very different shapes depending on the choice of the parameter c (see figs 5-6).

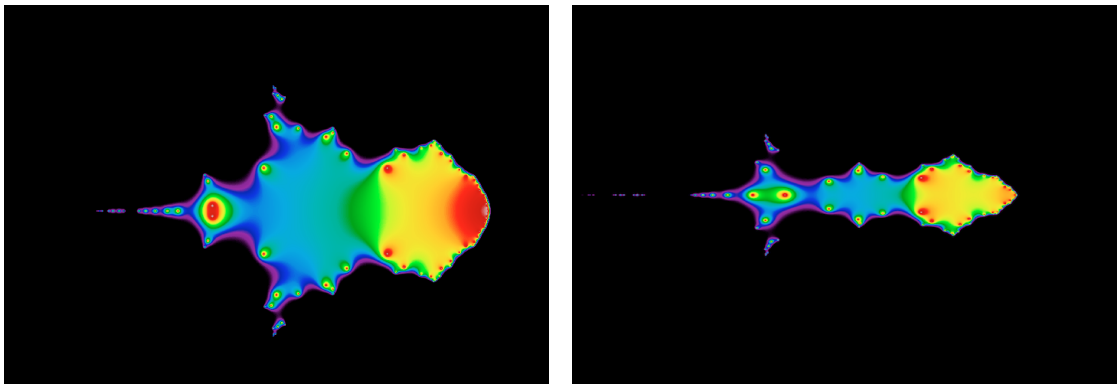


Fig.4 - a) Mandelbrot set ($z_0 = 1.0$); b) Mandelbrot set ($z_0 = 1.3$)

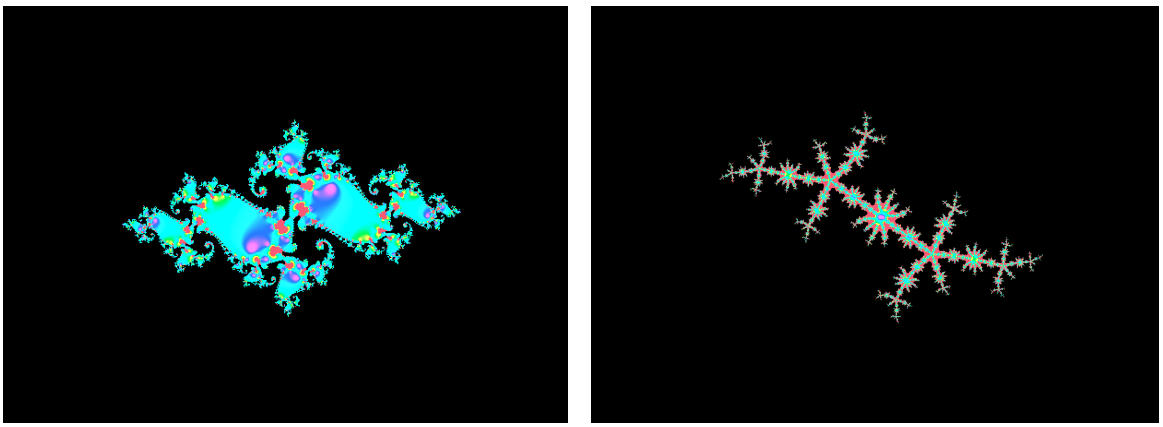


Fig.5 - a) Julia set for $c = -0.7454294 + i0.113089$; b) Julia set for $c = -0.561321 - i0.641$

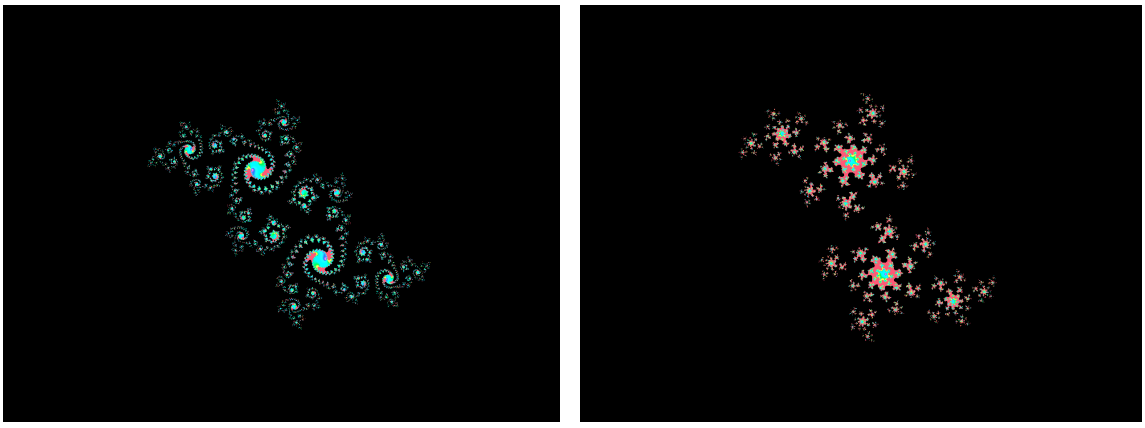


Fig.6 - a) Julia set for $c = -0.2009 - 0.67037$ b) Julia set for $c = 0.11031 - i0.67037$

Generalized Mandelbrot sets have been obtained starting from a different choice of the function $f(z)$, as it is shown in figs 7-10.

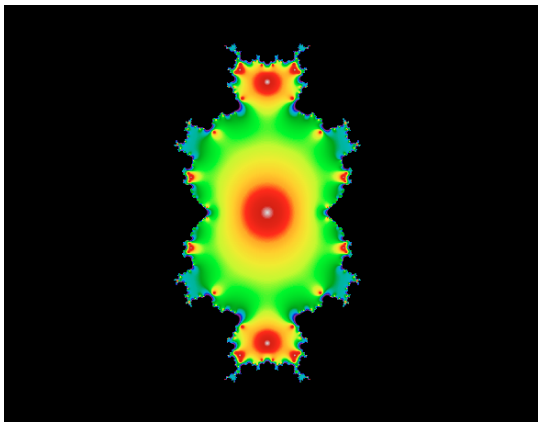


Fig.7 - $f(z) = z^3 + c$

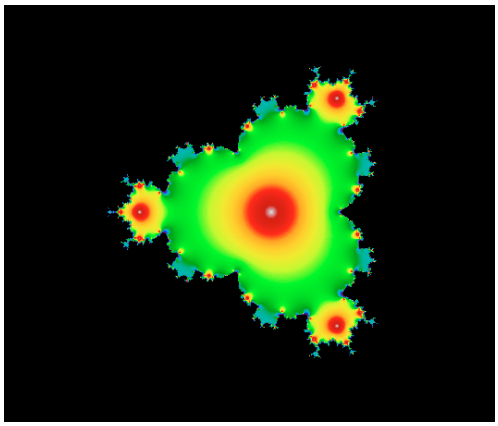


Fig.8 - $f(z) = z^4 + c$

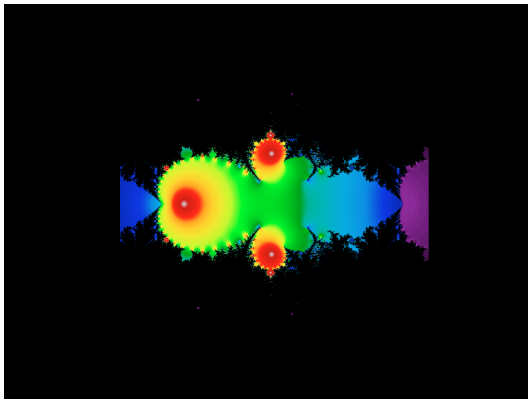


Fig.9 - $f(z) = \cos^2 z + c$

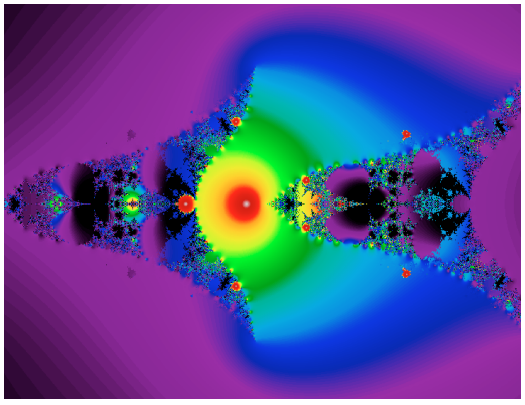


Fig.10 - $f(z) = \tan^2 z + c$

Similarly generalized Julia sets can be obtained starting from a different choice of the function $f(z)$ (see figs 11-12).

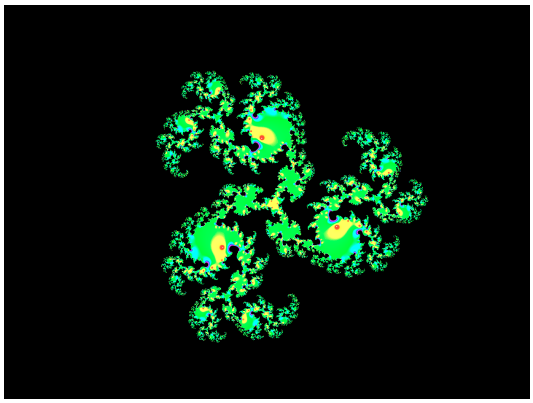


Fig.11 - $f(z) = z^3 + c$

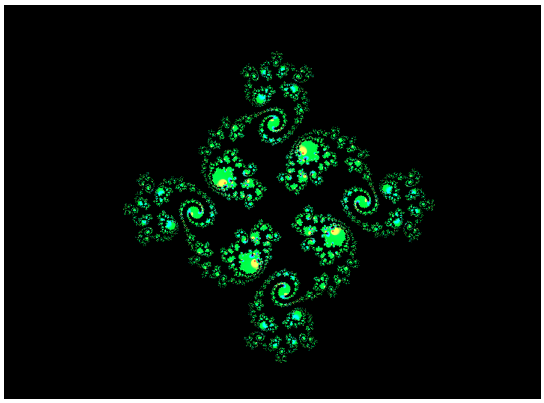


Fig.12 - $f(z) = z^4 + c$

We conclude this section with two examples based on *Newton's method*.

Newton's method is used to check approximated zero solutions to polynomials of any degree, being based on the recursion law:[12]

$$z_{k+1} = z_k - \frac{f(z_k)}{f'(z_k)}. \tag{2.6}$$

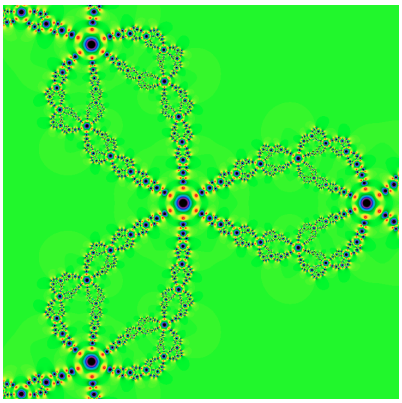


Fig.13 - $f(z) = z^3 + 1$

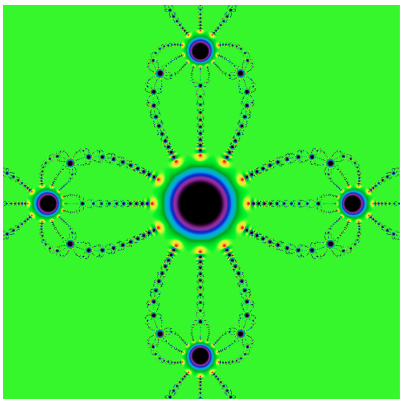


Fig.14 - $f(z) = z^4 + 1$

[12]This law can be obtained by the first order Taylor expansion of $f(z)$ in the neighborhood of z_0, given by: $f(z) = f(z_0) + f'(z_0)(z - z_0)$. Requiring that $f(z_{k+1}) = 0$ and setting $z_0 = z_k$ it results, solving by z_k that $z_{k+1} = z_k - \frac{f(z_k)}{f'(z_k)}$, provided that it is assumed that $f'(z_k \neq 0$.

Python 3 codes to generate Figs 13 and 14

```
################################################
# Newton's method set (Z**3+1=0 or Z**4+1=0)
# with complex arrays (matplotlib module)
################################################

import numpy as np           # import numpy module
import matplotlib.pyplot as plt      # import matplotlib module

n = 8  # alternative n = 12        # set number of cycles
Cx = 0.0              # set initial x parameter shift
Cy = 0.0              # set initial y parameter shift
L = 1.0              # set square area side
M = 2024              # set side number of pixels

x = np.linspace(-L-Cx,L-Cx,M)      # x variable array
y = np.linspace(-L-Cy,L-Cy,M)      # y variable array
X,Y = np.meshgrid(x,y,sparse=True)    # square area grid

Z = X + 1j*Y            # complex plane area

for k in range(1,n+1):          # recursion cycle
  Z1 = Z - (Z**3 + 1)/(3*Z**2)
  # alternative Z1 = Z - (Z**4 + 1)/(4*Z**3)
  Z = Z1

W = np.e**(-.5*abs(Z))        # smoothed sum moduls

plt.imshow(W,interpolation='nearest', cmap=plt.cm.nipy_spectral)
plt.axis("off")
plt.show()              # plot image
```

In the examples we have just presented the *recursion law* (*information*) – according to a philosophical perspective – plays a role which appears to be similar to that of a *form* or *essence* respect to the resulting *entity* (the fractal object), since it defines exactly and univocally its structure. While the individual paper sheet on which the image is printed or the individual screen on which it is displayed plays the role of *matter* determining each *singular* actualization of the *form*. We emphasize that, in the previous examples, the action of the *form* is revealed only as final result of its operation. So the fractal object is considered as a *whole,* while the process of the emergence of its *ordered* structure by the action *form* is not revealed.

In order to reveal how an *algorithm* operates in generating the *whole,* starting from unrelated *parts,* we need a different programming strategy which allows to show the fractal emergence *point by point* and not only as a final *whole*.

In $2D$ we have can achieve easily our goal thanks, *e.g.,* to *Graphics* module in *Python 3*.

Showing the ordered sequential process generating $2D$ fractals *point by point*

Therefore, beside showing the pictures of fractals *as wholes,* it is relevant[13] for us to show also the possibile dynamics capable to generate each image, examining the evolutionary process of image generation at each stage, so revealing the role of *form/information* as *operating nature.*

An elementary process is provided by scanning a region of the complex (or xy) plane *sequentially* (raw after raw, column after column), so that it appears clearly as *order generates order.*

Steps of the generation process of *Mandelbrot set, Julia set* ($c = 0.7454294$) and *Newton's method set* ($f(z) = z^6 + 1$) are shown in figs 15-17. Here are the related programming codes.

Python 3 codes to generate Figs 15, 16 and 17

```
################################################################
# Sequential ordered steps of 2D Mandelbrot set generation
# (graphics module)
################################################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*   # import graphics module

Radius = 10   # set escape rate threshold
x0 = .5   # set initial x co-ordinate shift
y0 = 0.0   # set initial y co-ordinate shift
Side = 1.2   # set square area side
M = 300    # set side number of elementary squares
N = 1   # set color map scale factor
Num = 256*N   # set number of cycles
sT=5   #  set step jump

win = GraphWin("Mandelbrot set", int(5*M/3),int(5*M/3))  # set window title
win.setBackground("white")     # set background color

def rectCol(p,q,w):   # define elementary cell
    Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
    Point(int(p+sT/2),int(q+sT/2)))
    Rect.draw(win).setFill(color_rgb(int(w),int(128-w/2),
    int(128+w/2)))

for p in range(1,M,sT):   # column scanning cycle
  Incy = y0 - Side + 2*Side/M*p   # define column scanning function
  for q in range(1,M,sT):   # raw scanning cycle
    Incx = x0 - Side + 2*Side/M*q   # define raw scanning function
    x = 0.0   # set starting x co-ordinate
    y = 0.0   # set starting y co-ordinate
    w = 0   # set starting escape modulus value
    for n in range(1,Num):   # recursion cycle
      xx = x*x - y*y - Incx
      yy = 2*x*y - Incy
      x = xx
      y = yy
```

[13] In particular for future applications to biology, as it will be shown in chapters 8 and 9.

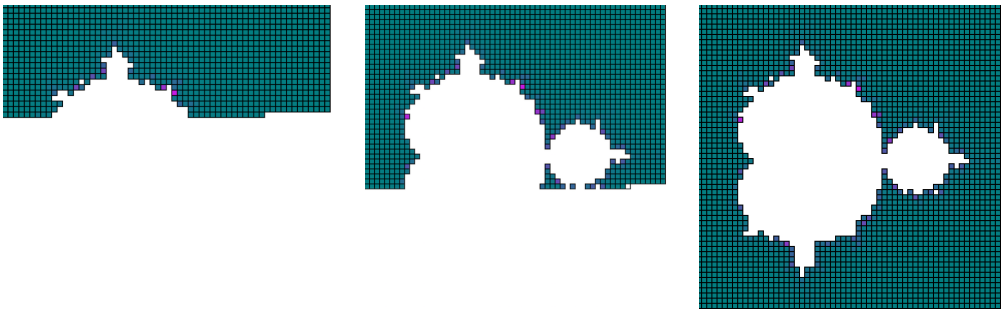


Fig.15 - Rough scheme of ordered sequential generation of Mandelbrot set

VIEW ANIMATION (requires internet connection)

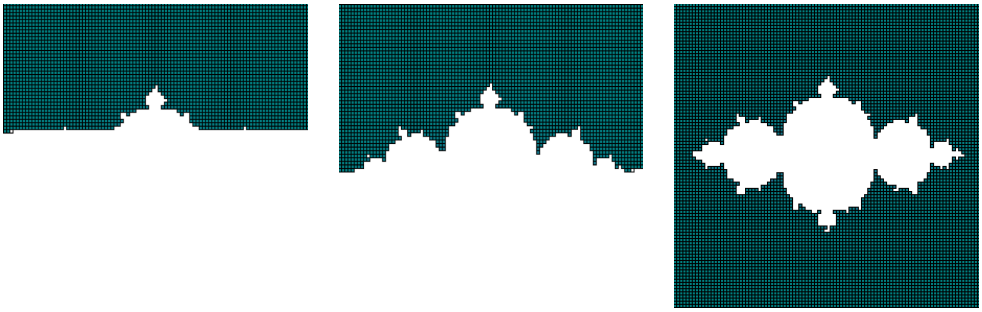


Fig.16 - Rough scheme of ordered sequential generation of a Julia set ($c = 0.7454294$)

VIEW ANIMATION (requires internet connection)

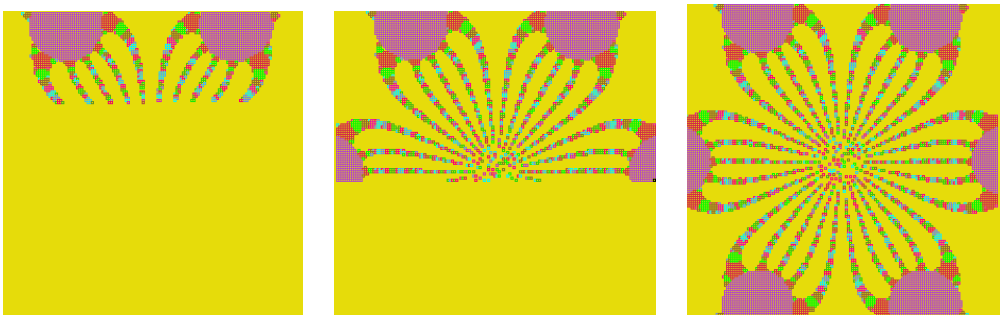


Fig.17 - Rough scheme of ordered sequential generation of a Newton's method set ($f(z) = z^6 + 1$)

VIEW ANIMATION (requires internet connection)

```
      if x*x + y*y > Radius:   # escape rate condition
        w = n/N   # escape modulus normalization
        rectCol(int(M/3+q),int(M/3+p),int(w))   # plot elementary cell
        break  # interrupt cycle

win.getMouse()   # wait for mouse click
win.close()   # close window



############################################################
# Sequential ordered steps of a 2D Julia set generation
# (c=0.7454294)
# (graphics module)
############################################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*   # import graphics module

Radius = 10   # set escape rate threshold
Cx = 0.7454294   # set c  parameter real part value
Cy = 0.0   # set c  parameter imaginary part value
Side = 1.7   # set square area side
M = 300    # set side number of elementary squares
N = 1   # set color map scale factor
Num = 256*N   # set number of cycles
sT=3  #  set step jump

win = GraphWin("Julia set", 5*M/3,5*M/3)  # set window title
win.setBackground("white")     # set background color

def rectCol(p,q,w):   # define elementary cell
    Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
    Point(int(p+sT/2),int(q+sT/2)))
    Rect.draw(win).setFill(color_rgb(int(w),int(128-w/2),
    int(128+w/2)))

for p in range(1,M,sT):   # column scanning cycle
  Incy = - Side + 2*Side/M*p   # define column scanning function
  for q in range(1,M,sT):   # raw scanning cycle
    Incx = - Side + 2*Side/M*q   # define raw scanning function
    x = Incx   # set starting increment of x co-ordinate
    y = Incy   # set starting increment of y co-ordinate
    w = 0   # set starting escape modulus value
    for n in range(1,Num):  # recursion cycle
      xx = x*x - y*y - Cx
      yy = 2*x*y - Cy
      x = xx
      y = yy
      if x*x + y*y > Radius:   # escape rate condition
        w = n/N   # escape modulus normalization
        rectCol(int(M/3+q),int(M/3+p),int(w))   # plot elementary cell
        break  # interrupt cycle

win.getMouse()   # wait for mouse click
win.close()   # close window
```

```
############################################################
# Sequential ordered steps of a 2D Newton's method set
# generation - Polynomial f(z) = z**6+1
# (graphics module)
############################################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*   # import graphics module
import numpy as np   # import numpy module

Radius = .5   # set escape rate threshold
Cx = 0.0   # set initial x parameter shift
Cy = 0.0   # set initial y parameter shift
Side = .8   # set square area side
M = 300    # set side number of elementary squares
N = 1   # set color map scale factor
Num = 256*N  # set number of cycles
sT=2   #  set step jump

win = GraphWin("Newton's method set", 5*M/3,5*M/3)  # set window title
win.setBackground(color_rgb(230,220,10))    # set background color

def rectCol(p,q,w):   # define elementary cell
  Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
  Point(int(p+sT/2),int(q+sT/2)))
  Rect.draw(win).setOutline(color_rgb(np.int(255*np.sin(w)**2),
  np.int(255*np.cos(w)**2),np.int(255*np.cos(w/2)**2)))

# Alternative values Cx 0.1747, 0.1747    Cy -.072,-1.072
# Side 0.0015, 0.00015 Num 1024

for p in range(1,M,sT):   # column scanning cycle
  Incy = - Side + 2*Side/M*p   # define column scanning function
  for q in range(1,M,sT):   # raw scanning cycle
    Incx = - Side + 2*Side/M*q   # define raw scanning function
    x = Incx   # set starting increment of x co-ordinate
    y = Incy   # set starting increment of y co-ordinate
    w = 0   # set starting escape modulus value
    for n in range(1,Num):   # recursion cycle
      xx = 5*x/6.0 - x*(x*x*x*x - 10*x*x*y*y +5*y*y*y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/6.0
      yy = 5*y/6.0 + y*(5*x*x*x*x - 10*x*x*y*y + y*y*y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/6.0
      x = xx
      y = yy
      if (x-Cx)*(x-Cx) + (y-Cy)*(y-Cy) < Radius:  # escape rate condition
        w = n/N   # escape modulus normalization
        rectCol(int(M/3+q),int(M/3+p),int(w))   # plot elementary cell
        break  # interrupt cycle

win.getMouse()   # wait for mouse click
win.close()   # close window
```

2.2 Ordered entity structures generated by random processes

Now it is relevant, especially regarding biological applications, to observe that the *ordered sequential process,* we have just tested in the previous §2.1 does not provide the only possibile

dynamics capable to generate *ordered* structures. In alternative to assign the *initial conditions* – starting from which one applies the mathematical *algorithm* (*information*) generating the corresponding point of the plot (pixel or cell on the screen) – according to a *sequential order,* we may always choose them *at random.*

With that choice each point or cell will appear on the computer display here and there, *randomly.* But at the end of the process, the same *ordered structure* of the structure will result. In other words, *chance* seems to *generate order,* but only thanks to the *information* hidden within the mathematical law encoded in the algorithm. The ordinating principle is *information* and not *chance* as such.

2.2.1 Showing the random process generating $2D$ fractals

Mandelbrot, Julia and Newton's method sets

A typical example is offered by $2D$ *fractals.*

We present, for a comparison with the sequential process, pictures and *Python 3* related codes generating *Mandelbrot, Julia* and *Newton's method* fractals arising starting from random initial conditions. Of course the smaller are the elementary squares building the plot the more refined image will result.[14]

We show in figs 18, 19 and 20 the same *Mandelbrot, Julia* and *Newton's method sets* considered before, now generated by random assignment of initial conditions. One may recognize how order, initially lacking appears slowly step by step,

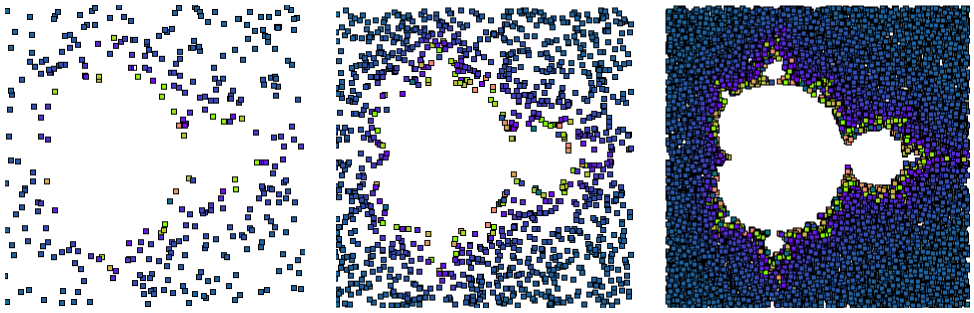


Fig.18 - Rough scheme of random generation of Mandelbrot set

VIEW ANIMATION (requires internet connection)

[14]Naturally a more refined image requires a longer computing time.

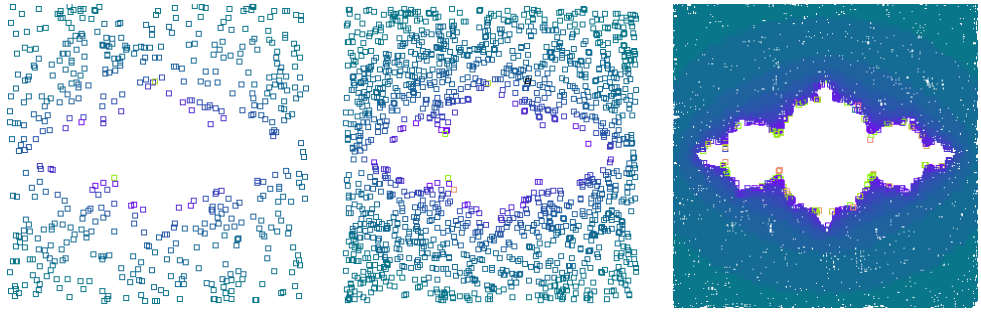


Fig.19 - Rough scheme of random generation of a Julia set ($c = 0.7454294$)

VIEW ANIMATION (requires internet connection)

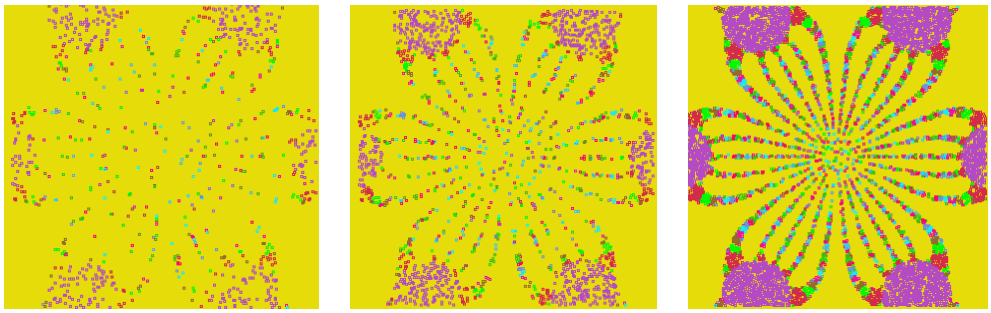


Fig.20 - Rough scheme of random generation of a Newton's method set ($f(z) = z^6 + 1$)

VIEW ANIMATION (requires internet connection)

Python 3 codes to generate Figs 18, 19 and 20

```
##########################################
# 2D Mandelbrot set random generation
# (graphics module)
##########################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*  # import graphics module
import random  # import random module

Radius = 10  # set escape rate threshold
Cx = .5  # set initial x co-ordinate shift
Cy = 0.0  # set initial y co-ordinate shift
Side = 1.3   # set square area side
M = 300    # set side number of elementary squares
N = 1   # set color map scale factor
```

```
Num = 256*N   # set number of cycles
sT=5   #  set step jump

win = GraphWin("Mandelbrot set", int(5*M/3),int(5*M/3))   # set window title

def rectCol(p,q,w):      # define elementary cell
  Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
  Point(int(p+sT/2),int(q+sT/2)))
  Rect.draw(win).setFill(color_rgb(int(10*w%255),
  int((128-10*w)%255),int((128+10*w)%255)))

# Alternative values Cx 0.1747, 0.1747 Cy -.072,-1.072
# Side 0.0015, 0.00015 Num 1024

i = 1  # set non-zero index value
while i > 0:   # set random co-ordinates choice cycles
  p = random.randrange(1,M)
  q = random.randrange(1,M)
  Incx = Cx - Side + 2*Side/M*q
  Incy = Cy - Side + 2*Side/M*p
  x = 0.0  # set starting x co-ordinate
  y = 0.0  # set starting y co-ordinate
  w = 0  # set starting escape modulus value
  for n in range(1,Num):
    xx = x*x - y*y - Incx
    yy = 2*x*y - Incy
    x = xx
    y = yy
    if x*x + y*y > Radius:   # escape rate condition
      w = n/N   # escape modulus normalization
      rectCol(int(M/3+q),int(M/3+p),int(w))  # plot elementary cell
      break    # interrupt cycle



###################################
# 2D Julia set random generation
# (graphics module)
###################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*  # import graphics module
import random  # import random module

Radius = 10  # set escape rate threshold
Cx = 0.7454294   # set c  parameter real part value
Cy = 0.0  # set c  parameter imaginary part value
Side = 1.7   # set square area side
M = 300    # set side number of elementary squares
N = 1   # set color map scale factor
Num = 256*N   # set number of cycles
sT=2   #  set step jump

win = GraphWin("Julia set", 5*M/3,5*M/3)  # set window title

def rectCol(p,q,w):   # define elementary cell
    Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
    Point(int(p+sT/2),int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(int(10*w%255),
```

```
    int((128-10*w)%255),int((128+10*w)%255)))

# Alternative values Cx 0.1747, 0.1747 Cy -.072,-1.072
# Side 0.0015, 0.00015 Num 1024

i = 1  # set non-zero index value
while i > 0:   # set random co-ordinates choice cycles
  p = random.randrange(1,M)
  q = random.randrange(1,M)
  Incx = - Side + 2*Side/M*q   # set x increment
  Incy = - Side + 2*Side/M*p   # set y increment
  x = Incx
  y = Incy
  w = 0   # set starting escape modulus value
  for n in range(1,Num):
    xx = x*x - y*y - Cx
    yy = 2*x*y - Cy
    x = xx
    y = yy
    if x*x + y*y > Radius:    # escape rate condition
      w = n/N     # escape modulus normalization
      rectCol(int(M/3+q),int(M/3+p),int(w))   # plot elementary cell
      break    # interrupt cycle



###############################################
# 2D Newton's method set random generation
# Polynomial f(z)=z**6+1
# (graphics module)
###############################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*  # import graphics module
import numpy as np   # import numpy module
import random    # import random module

Radius = .5   # set escape rate threshold
Cx = 0.0   # set initial x parameter shift
Cy = 0.0   # set initial y parameter shift
Side = .8    # set square area side
M = 300     # set side number of elementary squares
N = 1    # set color map scale factor
Num = 256*N   # set number of cycles
sT=2   #  set step jump

win = GraphWin("Newton's method set", 5*M/3,5*M/3)  # set window title
win.setBackground(color_rgb(230,220,10))      # set background color

def rectCol(p,q,w):   # define elementary cell
  Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
  Point(int(p+sT/2),int(q+sT/2)))
  Rect.draw(win).setOutline(color_rgb(np.int(255*np.sin(w)**2),
  np.int(255*np.cos(w)**2),np.int(255*np.cos(w/2)**2)))

# Alternative values Cx 0.1747, 0.1747 Cy -.072,-1.072
# Side 0.0015, 0.00015 Num 1024

i = 1  # set non-zero index value
```

```
while i > 0:    # set random co-ordinates choice cycles
  p = random.randrange(1,M)
  q = random.randrange(1,M)
  Incx = - Side + 2*Side/M*q    # set starting increment of x co-ordinate
  Incy = - Side + 2*Side/M*p    # set starting increment of y co-ordinate
  x = Incx
  y = Incy
  w = 0  # set starting escape modulus value
  for n in range(1,Num):
    xx = 5*x/6.0 - x*(x*x*x*x - 10*x*x*y*y +5*y*y*y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/6.0
    yy = 5*y/6.0 + y*(5*x*x*x*x - 10*x*x*y*y + y*y*y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/6.0
    x = xx
    y = yy
    if (x-Cx)*(x-Cx) + (y-Cy)*(y-Cy) < Radius:    # escape rate condition
      w = n/N   # escape modulus normalization
      rectCol(int(M/3+q),int(M/3+p),int(w))    # plot elementary cell
      break   # interrupt cycle
```

The iterating function system (IFS) generating natural ordered fractal structures

Another example of *ordered structures* generated by a *random dynamics* governed by a mathematical *algorithmic information* is offered by the fractals obtained applying the *Iterated Function System* (*IFS*). This method is employed, generally, to model shapes existing in nature, like coast or mountain profiles, leaves, ferns, trees, clouds and so on. The recursion law of the algorithm is characterized by *affine transformations* of type:

$$x_{p\,n+1} = a_p\,x_n + b_p\,y_n + c_p, \qquad y_{p\,n+1} = d_p\,x_n + e_p\,y_n + f_p, \tag{2.7}$$

where $a_p, b_p, c_p, d_p, e_p, f_p$ are constant coefficients the value of which is chosen in a suitable way in order to obtain the desired shape. This method introduces chance at the level of the *random probability* p according to which each coefficient value may occur. So, if *e.g.,* a randomly chosen p' is greater than p_1 and less than p_2, the coefficients will assume the values $a_{p_1}, b_{p_1}, c_{p_1}, d_{p_1}, e_{p_1}, f_{p_1}$. While if a different random value p'' of the probability occurs, say, between p_2 and p_3 the coefficients will be assigned to the different values $a_{p_2}, b_{p_2}, c_{p_2}, d_{p_2}, e_{p_2}, f_{p_2}$. Then the law (2.7) is changed according to some chance criterion. Typical examples are the *fractal fern* (fig. 21), the *fractal tree* (fig. 22), or the *Sierpinski triangle* (fig. 23).

Fig.21 - Fern generation steps (IFS method)

VIEW ANIMATION (requires internet connection)

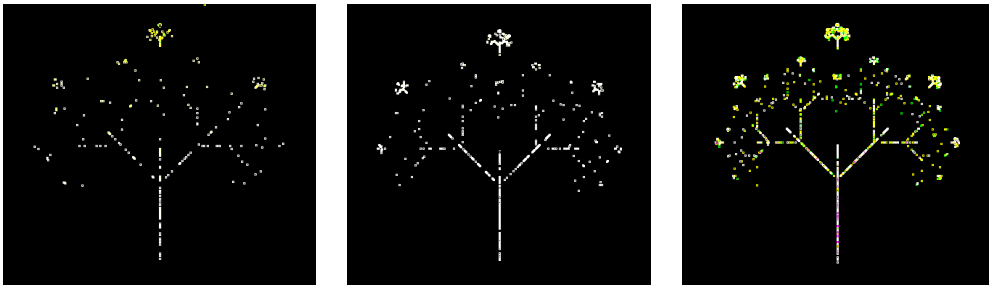


Fig.22 - Tree generation steps (IFS method)

VIEW ANIMATION (requires internet connection)

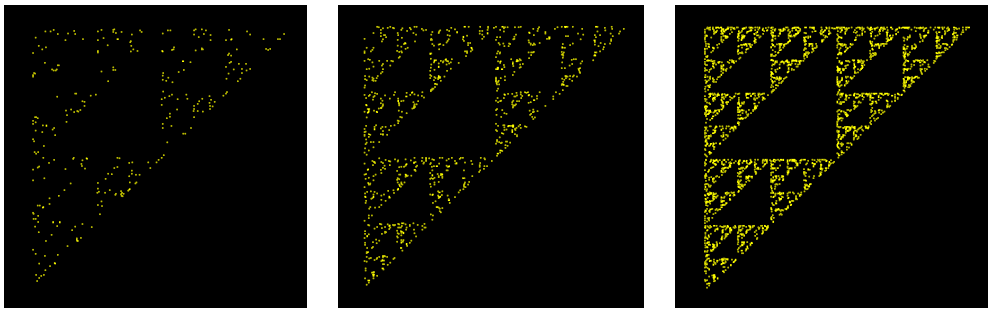


Fig.23 - Sierpinski triangle generation steps (IFS method)

VIEW ANIMATION (requires internet connection)

The related computer codes of programs to generate such images are given below.

Python 3 codes to generate figs 21, 22 and 23

```
################################
# IFS fern random generation
# (graphics module)
################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*  # import graphics module
import numpy as np    # import numpy module
import random    # import random module

Mxy=[[0.0,0.0,0.0,0.6,0.0,0.0,0.01],          # assign probability matrix
    [0.85,0.04,-0.04,0.85,0.0,1.6,0.85],
    [0.2,-0.26,0.23,0.22,0.0,1.6,0.07],
    [-0.15,0.28,0.26,0.24,0.0,0.44,0.07]]

a = [0,.85, .2, -.15]       # assign affine transformations coefficients
```

```
b = [0, .04, -.26, .28]
c = [0, -.004, .23, .26]
d = [.16, .85, .22, .24]
e = [0, 0, 0, 0]
f = [0, 1.6, 1.6, .44]

M = 300    # set side number of elementary squares
Num = 30000   # set number of cycles
sT = 1    #  set step jump

win = GraphWin("Fern", 2*M,2*M)   # set window title
win.setBackground('black')       # set background color

def rectCol(p,q,w):     # define elementary cell
    Rect = Rectangle(Point(np.int(p-sT/2),np.int(q-sT/2)),
    Point(np.int(p+sT/2),np.int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(w,255-np.int(.5*w),
    np.int(.5*w)))

x = 1    # set x initial value
y = 1    # set x initial value

for n in range(0,Num):    # set random probabilities choice cycles
  P = random.random()
  if P <= Mxy[0][6]:
    k = 0
  elif P <= Mxy[0][6] + Mxy[1][6]:
    k = 1
  elif P <= Mxy[0][6] + Mxy[1][6] + Mxy[2][6]:
    k = 2
  else:
    k = 3

  xx = a[k]*x+b[k]*y+e[k]     # affine transformation recursion cycle
  yy = c[k]*x+d[k]*y+f[k]
  x = xx
  y = yy

  rectCol(np.int(M+50*x),np.int(2*M-30-50*y),np.int(np.abs(20*y)))    # plot elementary cell

win.getMouse()    # wait for mouse click
win.close()    # close window


################################
# IFS tree random generation
# (graphics module)
################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*  # import graphics module
import numpy as np    # import numpy module
import random    # import random module

Mxy=[[0.195 , -0.488, 0.344 , 0.443 , 0.4431, 0.2452],          # assign probability matrix
  [0.462 , 0.414 , -0.252, 0.361 , 0.2511, 0.5692],
  [-0.058, -0.07 , 0.453 , -0.111, 0.5976, 0.0969],
  [-0.035, 0.07  , -0.469, -0.022, 0.4884, 0.5069],
  [-0.637, 0     , 0     , 0.501 , 0.8662, 0.2513]]
```

```
# Mxy=[[0.0,0.0,0.0,0.6,0.0,0.0,0.01],
#     [0.85,0.04,-0.04,0.85,0.0,1.6,0.85],
#     [0.2,-0.26,0.23,0.22,0.0,1.6,0.07],
#     [-0.15,0.28,0.26,0.24,0.0,0.44,0.07]]

a = [0,.42, .42, .1]     # assign affine transformations coefficients
b = [0, -.42, .42, 0]
c = [0, .42, -.42, 0]
d = [.5, .42, .42, .1]
e = [0, 0, 0, 0]
f = [0, .2, .2, .5]

M = 300    # set side number of elementary squares
Num = 3000  # set number of cycles
sT = 2    #  set step jump

win = GraphWin("Tree", 2*M,2*M)   # set window title
win.setBackground('black')       # set background color

def rectCol(p,q,w):       # define elementary cell
    Rect = Rectangle(Point(np.int(p-sT/2),np.int(q-sT/2)),
    Point(np.int(p+sT/2), np.int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(w,255-w,np.int(.5*w)))

x = 1    # set x initial value
y = 1    # set x initial value

for n in range(0,Num):     # set random probabilities choice cycles
  k = random.randrange(0,4)

  xx = a[k]*x+b[k]*y+e[k]          # affine transformation recursion cycle
  yy = c[k]*x+d[k]*y+f[k]
  x = xx
  y = yy

  # plot elementary cell
  rectCol(np.int(M+600*x),np.int(2*M-100-600*y), np.int(n*128/Num+np.abs(100*(.8-y))))

win.getMouse()    # wait for mouse click
win.close()   # close window



##############################################
# IFS Sierpinski triangle random generation
# (graphics module)
##############################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*   # import graphics module
import numpy as np     # import numpy module
import random    # import random module

a = [0.50,  0.50,  0.50]       # assign affine transformations coefficients
b = [0.00,  0.00,  0.00]
c = [0.00,  0.00,  0.00]
d = [0.50,  0.50,  0.50]
e = [0.00,  0.00,  0.50]
```

```
f = [0.00,  0.50,  0.50]

M = 300    # set side number of elementary squares
Num = 10000   # set number of cycles
sT = 1   #  set step jump

win = GraphWin("Sierpinski triangle", 2*M,2*M)   # set window title
win.setBackground('black')   # set background color

def rectCol(p,q,w):        # define elementary cell
    Rect = Rectangle(Point(np.int(p-sT/2),
    np.int(q-sT/2)),Point(np.int(p+sT/2),np.int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(w,w,0))

x = 1    # set x initial value
y = 1    # set x initial value

for n in range(0,Num):       # set random probabilities choice cycles
  k = random.randrange(0,3)

  xx = a[k]*x+b[k]*y+e[k]           # affine transformation recursion cycle
  yy = c[k]*x+d[k]*y+f[k]
  x = xx
  y = yy

  rectCol(np.int(np.int(.2*M)+500*x),np.int(2*M-50-500*y),255)      # plot elementary cell

win.getMouse()   # wait for mouse click
win.close()      # close window
```

2.2.2 Remark

The generation of an ordered structure starting from random initial conditions seems especially interesting in order to model biological entities (cells, organs, etc.). In fact a living ordered structure seems to appear, quite magically, by random process and arise by chance. Actually chance involves only the *initial conditions* and perhaps some of the subsequent *bifurcations* of the generation process dynamics, while some information hidden into, *e.g.,* the *DNA* and other supports, governs the entire generation dynamics. Such information may probably be hidden into some very complex string, or a nested structure of strings which partly is able to write its code step by step, as an unfolding strip.

2.3 Non-computable $2D$ structures

Only for some special structures[15] one is able to find a *mathematical formula* (*law*) which allows to define a sort of *essence* of some entity (*body* or *system*), which may be coded into a string shorter than the mere list of the co-ordinates of each point of the body or system itself. We have seen, in the previous sections, the examples of $2D$ fractals as significant structures.

[15]Some of those structures are well known and have been deeply studied.

Very many situations are known such that a compact formula cannot be found

– either because of some *technical difficulties*

– or for *principle reasons.*

In the former case one may always hope that in future a skillful and lucky researcher will be able to grasp such hidden law. In the latter case this lucky circumstance will be impossible, since the *Gödel's number* representing this formula is non-computable and the associated proof of the law results *undecidable* within the axiomatic system. According to computer science language one says that the *string* of the list of all the co-ordinates of the system points results to be incompressible and no regular order appears examining the sequence of the digits of the string, or the map of the points representing them geometrically. In some situation the compression of the string may be made *locally,* thanks to some technical trick,[16] but it does not exist a *global* unique formula (*shorter string*) compacting the whole structure of the system, defining it as a sort of *essence.*

2.3.1 Sequential process generating a map of prime numbers

An interesting example of non-compressible string seems to be offered (at least until now) by the sequence of the *prime numbers.*[17]

In fact, at least at present, we do not know any *law* to generate a number formed by the sequence of the first n prime numbers, shorter than the full list of those number themselves; like *e.g.,* the number built by the sequence of the first 5 numbers greater than 2. The first 5 prime numbers greater than 2 are $3, 5, 7, 11, 13$ and the number resulting is 3571113. In similar situations an ordered dynamics as the sequential scanning of a region of the Cartesian plane does not seem to produce any order. In the following figures we have plotted a portion of the Cartesian plane in such a way that

– red pixels are associated to points the absolute values of the co-ordinates of which are both prime numbers;

– green pixels are related to points of prime abscissa and non-prime ordinate;

– blue pixels are related to points of non-prime abscissa and prime ordinate;

– white pixels are related to points the co-ordinates of which are both non-prime numbers.

In particular, in fig. 24 the dynamics generated the plot is *sequentially ordered,* while in fig. 25 the dynamics generating the plot is *random.* In both cases the co-ordinates of each point are to be evaluated individually since there is no recursion formula allowing to generate the subsequent prime number starting from a known one. We point out that notwithstanding

[16] Generally the compression methods of image or text files are based on such local expedients which allow to shorten, *e.g.* a sequence of identical digits.

[17] We remember that a natural number n is said to be *prime* if it allows as exact divisors only the unit (1) and itself (n).

that the figure is plotted according to some symmetry criterion, since for each pair of prime numbers (x, y) we plot four symmetric points of co-ordinates $(x, y), (-x, y), (x, -y), (-x, -y)$, the visual perception of such symmetries is gradually lost being overridden by the randomness of the prime number sequence.

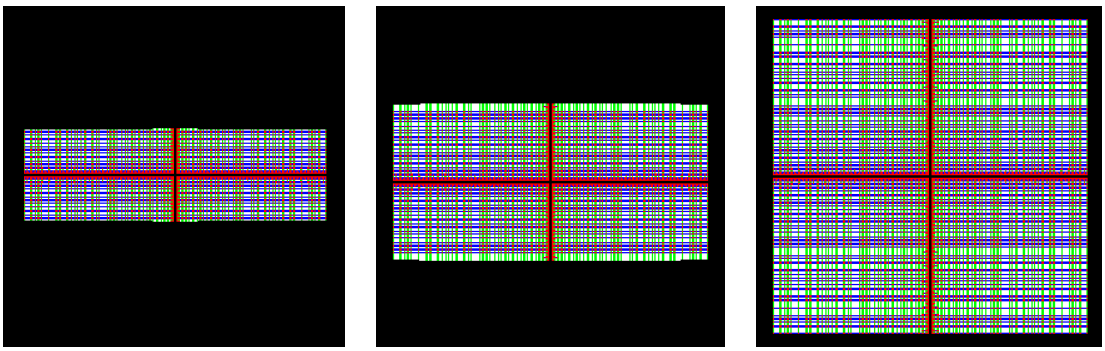


Fig.24 - Prime numbers sequential generation steps

VIEW ANIMATION (requires internet connection)

Python 3 code to generate images in fig. 24

```
############################
# Prime numbers > 2
# sequential generation
# (graphics module)
############################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*     # import graphics module
import numpy as np       # import numpy module

M = 253    # set side number of elementary square (max number to be checked)
sT=1    #  set step jump

P = np.zeros(M)    #  x co-ordinate array
Q = np.zeros(M)   #  y co-ordinate array
r = np.zeros([M,M])    # zero red color matrix
g = np.zeros([M,M])   # zero green color matrix
b = np.zeros([M,M])   # zero blue color matrix

win = GraphWin("Prime set (sequential)", int(10*M/3),int(10*M/3))    # set window title
win.setBackground("black")     # set background color

def rectCol(p,q,R,G,B):         # define elementary cell
    Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),Point(int(p+sT/2),
    int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(R,G,B))

for p in range(3,M,2):    # x sequential cycle
```

```
  for q in range(3,M,2):     # y sequential cycle
    j = 3     # set initial prime number
    Wp = 1    # set default x control index
    Wq = 1    # set default y control index

    while j < p and j < q:     # prime number checking cycle
      if p%j != 0:
        WWp = 1
        Wp = Wp*WWp  # ensures that no quotient is exact
      else:
        WWp = 0
        Wp = Wp*WWp  # ensures that no quotient is exact
      if q%j != 0:
        WWq = 1
        Wq = Wq*WWq  # ensures that no quotient is exact
      else:
        WWq = 0
        Wq = Wq*WWq  # ensures that no quotient is exact
      j = j + 2

    if (Wp == 1 and Wq == 1):     # red color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 255
      g[p,q] = 0
      b[p,q] = 0
    elif (Wp == 1 and Wq == 0):      # green color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 0
      g[p,q] = 255
      b[p,q] = 0
    elif (Wp == 0 and Wq == 1):      # blue color select  conditions
      P[p] = p
      Q[q]
      r[p,q] = 0
      g[p,q] = 0
      b[p,q] = 255
    elif (Wp == 0 and Wq == 0):     # white color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 255
      g[p,q] = 255
      b[p,q] = 255

for q in range(3,M,2):       # sequential  plot cycles
  for p in range (3,M,2):
    rectCol(int(5*M/3-P[p]),int(5*M/3+Q[q]),
    np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
    rectCol(int(5*M/3+P[p]),int(5*M/
    3+Q[q]),np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
    rectCol(int(5*M/3+P[p]),int(5*M/3-Q[q]),
    np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
    rectCol(int(5*M/3-P[p]),int(5*M/3-Q[q]),
    np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))

win.getMouse()     # wait for mouse click
win.close()      # close window
```

2.3.2 Random process generating a map of prime numbers

Here are the images and code related to ordered pairs of prime number generation according to a *random* choice of initial conditions. Neither *sequentially ordered* nor *random* dynamics seems to generate order.

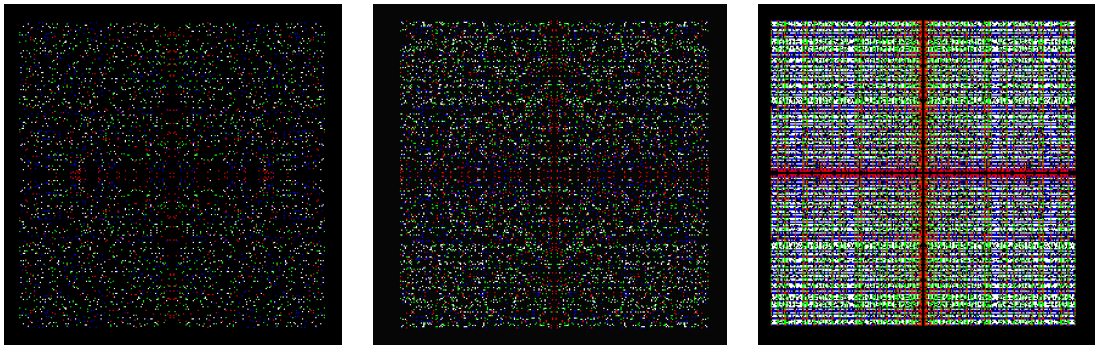


Fig.25 - Prime numbers random generation steps

VIEW ANIMATION (requires internet connection)

Python 3 code to generate images in fig. 25

```
########################
# Prime numbers > 2
# random generation
# (graphics module)
########################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import*      # import graphics module
import numpy as np       # import numpy module
import random as rd      # import random module

M = 253    # set side number of elementary square (max number to be checked)
sT=1    #  set step jump

P = np.zeros(M)     #  x co-ordinate array
Q = np.zeros(M)    #  y co-ordinate array

r = np.zeros([M,M])     # zero red color matrix
g = np.zeros([M,M])    # zero green color matrix
b = np.zeros([M,M])    # zero blue color matrix

win = GraphWin("Prime set (random)", int(10*M/3),int(10*M/3))    # set window title
win.setBackground("black")      # set background color

def rectCol(p,q,R,G,B):          # define elementary cell
```

```
    Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),Point(int(p+sT/2),
    int(q+sT/2)))
    Rect.draw(win).setOutline(color_rgb(R,G,B))

for p in range(3,M,2):      # x sequential cycle
  for q in range(3,M,2):      # y sequential cycle
    j = 3     # set initial prime number
    Wp = 1      # set default x control index
    Wq = 1    # set default y control index

    while j < p and j < q:       # prime number checking cycle
      if p%j != 0:
        WWp = 1
        Wp = Wp*WWp  # ensures that no quotient is exact
      else:
        WWp = 0
        Wp = Wp*WWp  # ensures that no quotient is exact
      if q%j != 0:
        WWq = 1
        Wq = Wq*WWq  # ensures that no quotient is exact
      else:
        WWq = 0
        Wq = Wq*WWq  # ensures that no quotient is exact
      j = j + 2

    if (Wp == 1 and Wq == 1):      # red color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 255
      g[p,q] = 0
      b[p,q] = 0
    elif (Wp == 1 and Wq == 0):    # green color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 0
      g[p,q] = 255
      b[p,q] = 0
    elif (Wp == 0 and Wq == 1):    # blue color select  conditions
      P[p] = p
      Q[q]
      r[p,q] = 0
      g[p,q] = 0
      b[p,q] = 255
    elif (Wp == 0 and Wq == 0):    # white color select  conditions
      P[p] = p
      Q[q] = q
      r[p,q] = 255
      g[p,q] = 255
      b[p,q] = 255


i = 1      # set non-zero control paramter value
while i > 0:    # random point selection cycles
  p = rd.randrange(1,M,2)
  q = rd.randrange(1,M,2)

  # plot cell
  rectCol(int(5*M/3-P[p]),int(5*M/3+Q[q]),
  np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
  rectCol(int(5*M/3+P[p]),int(5*M/
  3+Q[q]),np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
  rectCol(int(5*M/3+P[p]),int(5*M/3-Q[q]),
```

```
np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
rectCol(int(5*M/3-P[p]),int(5*M/3-Q[q]),
np.int(r[p,q]),np.int(g[p,q]),np.int(b[p,q]))
```

Chapter 3

Three-dimensional structures from algorithms

The roles of order and chance and the rendering problem

In order to attempt to model biological structures like, *e.g.,* organs, and their generating dynamics, we need, at present, to make a further step, *i.e.,* the passage from $2D$ to $3D$ systems. As we will see one of the main technical problems one encounters, just at the first level of modeling the geometrical shape of $3D$ structures, is related to the matter of rendering their boundary surface in a reasonably realistic and satisfactory way.

a) We will start examining some examples of complex $3D$ fractals according to different methods of surface rendering;[1]

b) Later we will propose a first rough model of the external surface of a human heart generated by parametric equations in $3D$ space.

Sequential and *random* procedures will be examined in both cases.

3.1 *Python 3* rendering of surfaces

Python 3 Mathplotlib provides a nice rendering of geometric shapes (boundary of $3D$ structures) when either the *Cartesian equation* or a set of *parametric equations* of the surface to be rendered is known.

In the next subsection §3.1.1 we start with a simple geometrical example of a surface defined by a Cartesian equation, while in §3.1.2 we offer an example of a surface defined by its parametric equations.

[1] *Python 3* and *POV-Ray* programming languages will be employed.

3.1.1 Cartesian equation surface rendering

Here is the example of a *Python 3 matplotlib* plot of the surface of Cartesian equation:

$$z = e^{-.0.5(x^2+y^2)} \cos \sqrt{x^2 + y^2}. \tag{3.1}$$

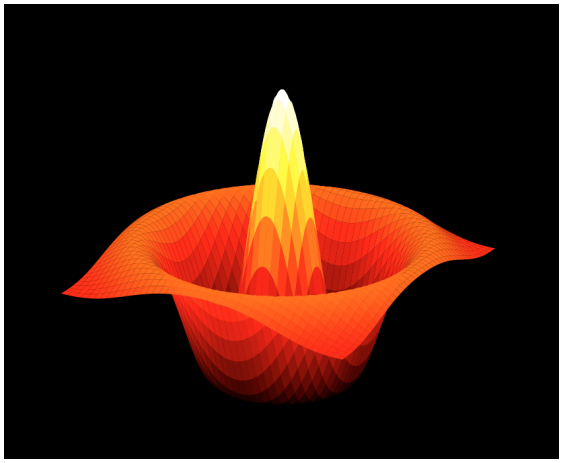


Fig.1 - Python 3 matplotlib rendering of a surface of Cartesian equation $z = e^{-.0.5(x^2+y^2)} \cos \sqrt{x^2 + y^2}$

And the related program code is given below.

Python 3 code to generate fig. 1

```
##################################
# Whole 3D surface generation
# by Cartesian equation
# (matplotlib module)
##################################

from mpl_toolkits.mplot3d import Axes3D    # import matplotlib module
import matplotlib.pyplot as plt    # import matplotlib pylot module
from matplotlib import cm     # import matplotlib colormap module
import numpy as np   # import numpy module

plt.style.use('dark_background')  # set black background color

R = 4.0   # set radius value
X = np.arange(-2*np.pi, 2*np.pi, 0.1)   # set x co-ordinate array values
Y = np.arange(-2*np.pi, 2*np.pi, 0.1)   # set y co-ordinate array values
X, Y = np.meshgrid(X, Y)   # set xy grid
Z = np.e**(-.05*(X**2+Y**2))*np.cos(np.sqrt(X**2 + Y**2))   # set z function of x,y

fig = plt.figure()    # define 3D plot and axes
ax = fig.gca(projection='3d')

ax.plot_surface(X, Y, Z, cmap='hot', linewidth=0)   # plot surface as a whole

ax.set_zlim(-.3, 1.0)     # set z axis limits
```

```
ax.axis("off")   # do not plot co-ordinate axes

plt.show()    # show surface a a whole
```

3.1.2 The need of parametric equations surface rendering

Even if rendering is quite satisfactory some problems arise, for our purposes, employing Cartesian equation rendering with *matplotlib*.

a) A first problem arises since surface *Cartesian equations* may plot only one-value functions and are not properly apt to render closed shapes like cells, organs and biological structures.[2] So one is led to prefer *parametric equations* representation to model such systems, so overcoming the problem.

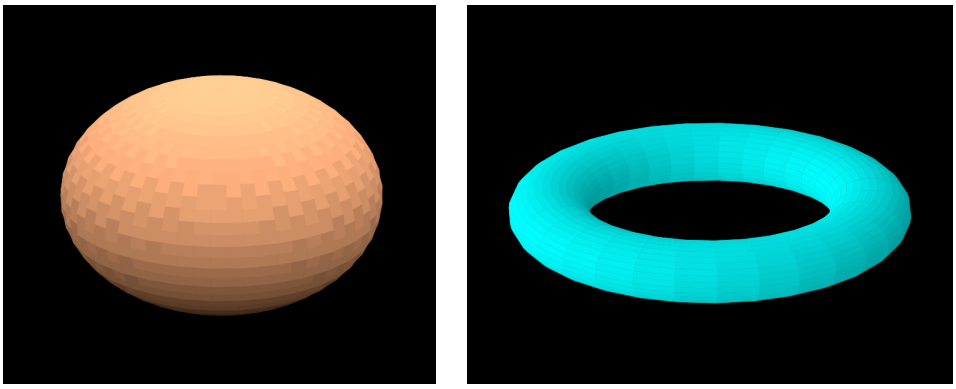


Fig.2 - Python 3 matplotlib rendering of a sphere and a torus by parametric equations

---

Python 3 code to generate fig. 2

```
####################################
# Whole sphere generation
# by paramteric equations
# (matplotlib module)
####################################

from mpl_toolkits.mplot3d import Axes3D   # import matplotlib module
import matplotlib.pyplot as plt   # import matplotlib module
import numpy as np    # import numpy module

angle = np.linspace(0, 2 * np.pi, 32)   #  define cylindrical co-ordinates array
```

[2]In principle one could join more surfaces generated by more one-value functions. But further difficulties may arise when tangent plane to surfaces become vertical (infinite derivatives). So this approach to surface rendering seems to be not raccomanded for our purposes.

```
theta, phi = np.meshgrid(angle, angle)   # set theta, phi grid

R = 1.0     # set radius value
X = R * np.cos(phi) * np.cos(theta)     # evaluate x co-ordinate array values
Y = R * np.cos(phi) * np.sin(theta)      # evaluate y co-ordinate array values
Z = R * np.sin(phi)     # evaluate z co-ordinate array values

plt.style.use('dark_background')     # set black background color

fig = plt.figure()     # define 3D plot and axes
ax = fig.gca(projection = '3d')

ax.set_xlim3d(-1, 1)        # set x axis limits
ax.set_ylim3d(-1, 1)        # set y axis limits
ax.set_zlim3d(-1, 1)        # set z axis limits

ax.plot_surface(X, Y, Z, cmap = "copper", rstride = 1, cstride = 1)   # plot surface as a whole
ax.axis('off')    # do not plot co-ordinate axes
plt.show()     # show surface a a whole



##############################
# Whole torus generation
# by paramteric equations
# (matplotlib module)
##############################

from mpl_toolkits.mplot3d import Axes3D     # import matplotlib module
import matplotlib.pyplot as plt      # import matplotlib module
import numpy as np      # import numpy module

angle = np.linspace(0, 2 * np.pi, 32)      #  define cylindrical co-ordinates array
theta, phi = np.meshgrid(angle, angle)     # set theta, phi grid
r, R = .25, 1.
X = (R + r * np.cos(phi)) * np.cos(theta)     # evaluate x co-ordinate array values
Y = (R + r * np.cos(phi)) * np.sin(theta)      # evaluate y co-ordinate array values
Z = r * np.sin(phi)

plt.style.use('dark_background')     # set black background color

fig = plt.figure()      # define 3D plot and axes
ax = fig.gca(projection = '3d')
ax.set_xlim3d(-1, 1)        # set x axis limits
ax.set_ylim3d(-1, 1)        # set y axis limits
ax.set_zlim3d(-1, 1)        # set z axis limits

ax.plot_surface(X, Y, Z, color = 'cyan', rstride = 1,cstride = 1)   # plot surface as a whole

ax.axis('off')   # do not plot co-ordinate axes
plt.show()  # show surface a a whole
```

---

b) A second problem arises because the module *matplotlib* plots a surface by *meshes*[3] as a *whole* and does not allow to evidence the process of the manifold construction step

[3] *Meshes* are small portions of surface (generally curved 4 or 3-lateral) which generally allow satisfactory colormap and lighiting effects.

by step. While, in order to our intent it is quite interesting to show the constructive process either when it is *sequentially ordered* or when it evolves *randomly*. This second problem may be overridden building a surface *point by point,* by parametric equations identifying the coordinates of each *point* of the shape to be modeled. By *point* here we necessarily mean a finite dimensional elementary cell suitably chosen, like a very small disk or sphere, or something else. Either *graphics module* (faster) or *matplotlib module* animation (slower but genuinly 3-*dimensional*) can be usefully employed. The relevant advantage of a *point by point* shape generation, in order to model biological systems, will be that each elementary sphere may be considered a rough but significant approximation of a living cell. So the full process could appear as a simulation of cell multiplication generating a complete organ of a living being. In particular a *random* cell replication process will be of special interest when each daughter cell is located contiguously respect to the mother cell. The latter circumstance characterizes the well known *cellular automata.*[4]

c) The latter method solves the previous problem but involves a new one providing, generally, a poor, when not totally unrealistic, rendering quality.

The following images show some results related both to *sequentially ordered* and *random* generation processes of a spherical shell obtained implementing either *graphics* or *matplotlib* modules of *Python 3*.

Sphere generated by Python 3 "graphycs" module

i) *Sequentially ordered* generation process

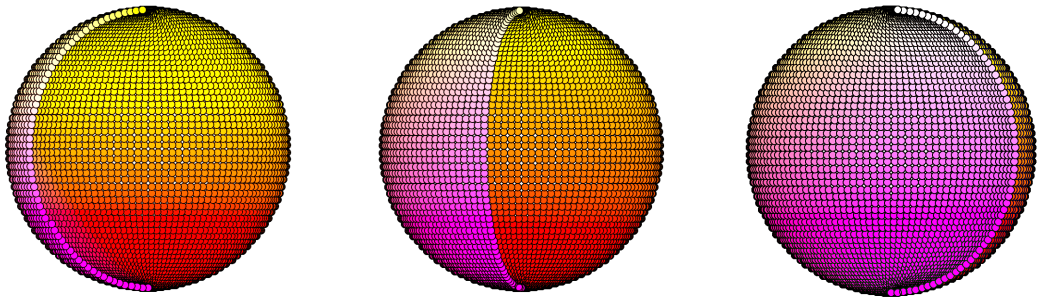


Fig.3 - Sphere generated sequentially by small disks (graphics module)

VIEW ANIMATION (requires internet connection)

[4]See, chapter **??**.

Python 3 code to generate fig. 3

```
################################
# Sequential sphere generation
# by Paramteric equations
# (graphics module)
##############################7####

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys

sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import *   # import graphics module
import numpy as np     # import numpy module

x = 200    # set x initial value
y = 100    # set y initial value
z = 100    # set z initial value

bgColor = "white"    # set background color
title = "Sphere"    # set window title
winWidth = 400   # set window width
winHeight = 400   # set window height

def pBall(x,y,z,rayBall,colBall):     # def cell function
  X = x
  Y = y
  Circle(Point(X,Y),rayBall)
  Circle(Point(X,Y),rayBall).draw(win).setFill(colBall)

win = GraphWin(title, winWidth, winHeight)     # define window
win.setBackground(bgColor)    # set background color

n = 126   #  set number of angular steps

x0 = np.int(.5*winWidth)   # set sphere center x co-ordinate
y0 = np.int(.5*winHeight)  # set sphere center y co-ordinate
z0 = 0    # set sphere center z co-ordinate

R = 150   # set sphere radius  value
t = 0     # set initial t parameter value
u = 0   # set initial u parameter value

# def 3D rendering functions on 2D plane
def xx(t):
  return R*np.sin(np.pi-np.pi*t/n)


def yy(t):
  return R*np.cos(np.pi-np.pi*t/n)

def co(u):
  return np.cos(2*np.pi*u/n)

def si(u):
  return np.sin(2*np.pi*u/n)

def x(t,u):
  return xx(t)*co(u)
```

```
def z(t,u):
  return yy(t)*si(u)

# plot sphere point by point by spherical co-ordinates parametric equations
for u in range(0,n,1):
  for t in range(0,n,2):
    pBall(x0 + x(t,u),y0 - yy(t),0,4, color_rgb(255,2*t,2*u))

win.getMouse()    # wait for mouse click
win.close()    # close window
```

---

ii) *Random* generation process

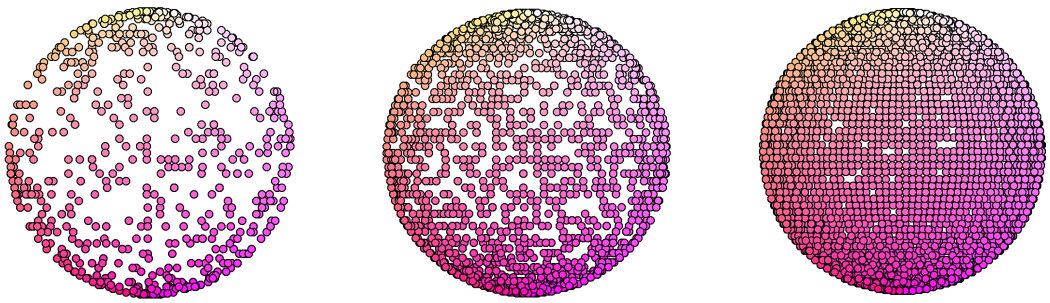


Fig.4 - Sphere generated randomly by small disks (graphics module)

VIEW ANIMATION (requires internet connection)

Python 3 code to generatefig. 4

```
####################################
# Random sphere generation
# by Paramteric equations
# (graphics module)
####################################

# specify the absolute path of mod graphics folder (depends on user's choice)
import sys

sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")

from graphics import *  # import graphics module
import numpy as np   # import numpy module
import random as rd   # import random module

x = 200    # set x initial value
y = 100    # set y initial value
z = 100    # set z initial value
```

```
bgColor = "white"    # set background color
title = "Sphere"    # set window title
winWidth = 400   # set window width
winHeight = 400   # set window height

def pBall(x,y,z,rayBall,colBall):      # def cell function
  X = x
  Y = y

  Circle(Point(X,Y),rayBall)
  Circle(Point(X,Y),rayBall).draw(win).setFill(colBall)

win = GraphWin(title, winWidth, winHeight)    # define window
win.setBackground(bgColor)   # set background color

n = 126  #  set number of angular steps

x0 = np.int(.5*winWidth)   # set sphere center x co-ordinate
y0 = np.int(.5*winHeight)  # set sphere center y co-ordinate
z0 = 0    # set sphere center z co-ordinate

R = 150   # set sphere radius  value
t = 0     # set initial t parameter value
u = 0   # set initial u parameter value

# def 3D rendering functions on 2D plane
def xx(t):
  return R*np.sin(np.pi-np.pi*t/n)


def yy(t):
  return R*np.cos(np.pi-np.pi*t/n)

def co(u):
  return np.cos(2*np.pi*u/n)

def si(u):
  return np.sin(2*np.pi*u/n)

def x(t,u):
  return xx(t)*co(u)

def z(t,u):
  return yy(t)*si(u)

i = 1   # set positive value for random cycle index
while i > 0:   # random cycle
  u = rd.randrange(np.int(n/2),n,1)   # assign random values to parameter u
  t = rd.randrange(1,n,2)  # assign random values to parameter t

  # plot sphere point by point by spherical co-ordinates parametric equations
  pBall(x0 + x(t,u),y0 - yy(t),0,4, color_rgb(255,2*t,2*u))

# notice: the infinite loop needs to be stopped by the user when at the desired image stage
```

Sphere generated by Python 3 “matplotlib” module

The *matplotlib* module performs a genuine $3D$ plot and a more precise structure than the *graphics* module plots avoiding overlapping of contiguous small spheres as it results more

evident especially in a random generating process. But it takes a longer computing time.

i) *Sequentially ordered* generation process[5]

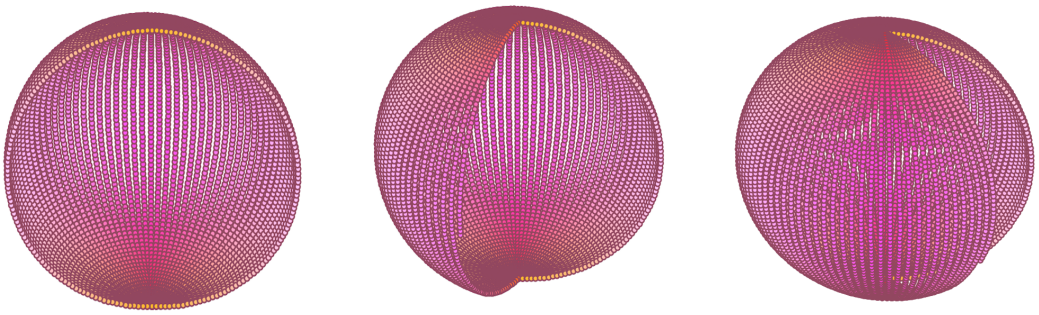


Fig.5 - Sphere generated sequentially by small disks (matplotlib module)

VIEW ANIMATION (requires internet connection)

Python 3 code to generatefig. 5

```
###################################
# Sequential sphere generation
# by Paramteric equations
# (matplotlib module)
###################################

from mpl_toolkits import mplot3d  # import matplotlib module
import matplotlib.pyplot as plt  # import matplotlib pyplot module
import numpy as np      # import numpy module
import random as rd     # import random module


r = 6.0   # set radius value
n = 100   # set number of cycles

theta = np.linspace(0.0, 2*np.pi, n)   # set theta co-ordinate array
phi = np.linspace(0.0, np.pi, n)   # set phi co-ordinate array
theta, phi = np.meshgrid(theta, phi)  # set theta,phi grid

X = - r*np.sin(-.35*np.pi+theta)*np.sin(phi)  # spherical to Cartesian transformations
Y = r*np.cos(-.35*np.pi+theta)*np.sin(phi)
Z = r*np.cos(phi)

ax = plt.axes(projection='3d', aspect=.85)    # set 3D axes system

ax.set_xlim3d([-4.5, 4.5])   # set x limits
ax.set_ylim3d([-5.0, 5.0])   # set y limits
ax.set_zlim3d([-5.0, 4.0])   # set z limits
```

[5]Animations requiring too long time will be limited in few minutes.

```
ax.set_axis_off()   # do not plot axes

for u in range(0,n,1):    # sequential parameter u cycle
  for t in range(0,n,1):    # sequential parameter t cycle
    ax.scatter(X[t,u], Y[t,u], Z[t,u], s = 12,   # plot poins
    c = [1,.7*np.abs(np.cos(2*np.pi*(1+u/n))),
    np.abs(np.sin(np.pi*(1+t/n)))], marker="o",
    edgecolor=[.5,.2,.3],zorder=2)
    plt.pause(0.001)   # animation pause interval

plt.show()   # show plots
```

---

ii) *Random* generation process

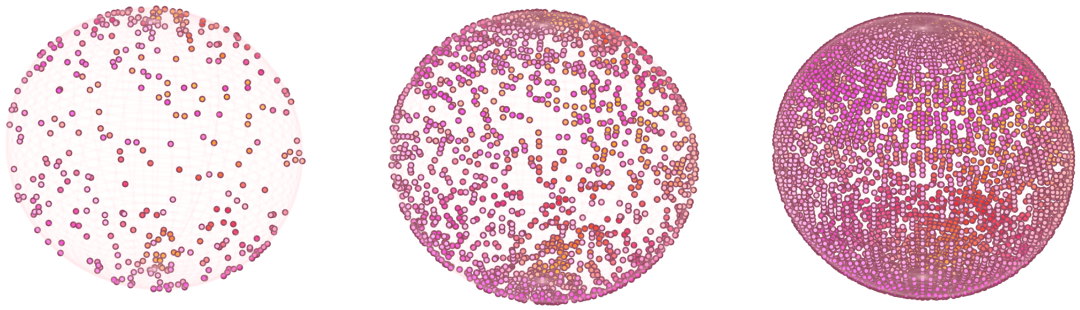


Fig.6 - Sphere generated randomly by small disks (matplotlib module)

VIEW ANIMATION (requires internet connection)

Python 3 code to generate fig. 6

```
###################################
# Random sphere generation
# by Paramteric equations
# (matplotlib module)
###################################

from mpl_toolkits import mplot3d  # import matplotlib module
import matplotlib.pyplot as plt   # import pylot module
import numpy as np    # import numpy module
import random as rd    # import random module

r = 6.0  # set radius value
n = 100  # set number of cycles

theta = np.linspace(0.0, 2*np.pi, n)   # set theta co-ordinate array
phi = np.linspace(0.0, np.pi, n)  # set phi co-ordinate array
theta, phi = np.meshgrid(theta, phi) # set theta,phi grid

X = r * np.sin(theta)*np.sin(phi)    # spherical to Cartesian transformations
Y = r * np.cos(theta)*np.sin(phi)
Z = r*np.cos(phi)
```

```
ax = plt.axes(projection='3d', aspect=.85)   # set 3D axes system
ax.plot_wireframe(X, Y, Z, edgecolor='pink',zorder=1,alpha=0.08)    # background sphere

ax.set_xlim3d([-4.5, 4.5])   # set x limits
ax.set_ylim3d([-5.0, 5.0])   # set y limits
ax.set_zlim3d([-5.0, 4.0])   # set z limits
ax.set_axis_off()   # do not plot axes

while r > 0:      # random cycle
  indX = np.random.choice(X.shape[0], 1, replace=False)
  indY = np.random.choice(Y.shape[1], 1, replace=False)

  ax.scatter(X[indX,indY], Y[indX,indY], Z[indX,indY],
  s = 12, c = [1,.7*np.abs(np.cos(2*np.pi*(1+indX/n))),
  np.abs(np.sin(np.pi*(1+indY/n)))], marker="o",
  edgecolor=[.5,.2,.3],zorder=2)
  plt.pause(0.001)  # animation pause time

plt.show()  # show plot
```

In order to accomplish a highly satisfactory rendering one usefully is led to replace *Python 3* with a graphically more performant *ray tracing* programming language like *POV-Ray.*

3.2 *POV-Ray* rendering of surfaces

Three dimensional image rendering is improved enormously recurring to a *ray tracing* program language like *POV-Ray,* which offers a more realistic resemblance to the objects, as one can just see even in simple shapes like, *e.g.,* a sphere or a torus (see fig. 7).

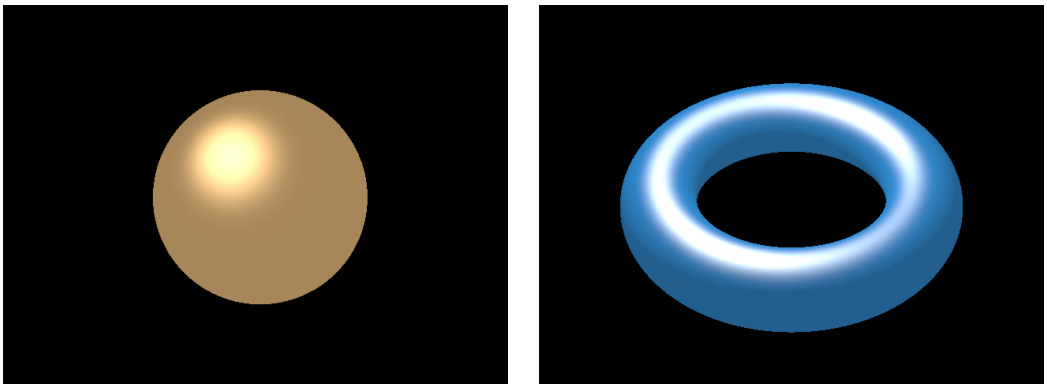


Fig.7 - POV-Ray rendering of a sphere and a torus as wholes

A ray tracing software proves very useful in order to a realistic rendering of $3D$ bodies as wholes. But it may be easily adapted even to show how some structures can be generated *point by point* in a similar way as *python3* can do. This way of implementing rendering algorithms proves especially useful, for our purposes, in relation to *biological systems,* generated by *cells*

and not only. The procedure requires to build a lot of images, like photograms recording each step fo the generation process, from which it is also possible to make animated movies. Of course very heavy calculations are needed and the computational time increases as the complexity of the structure to be generated grows.

3.2.1 Sphere generated sequentially by small balls

The method is easily illustrated by a first very simple example of an *ordered* structure like a sphere generated by a *sequentially ordered process*.

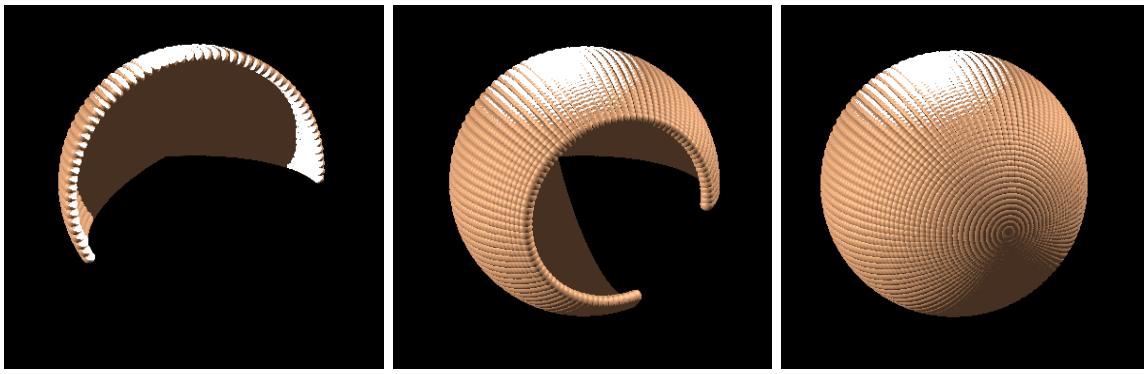


Fig.8 - Sphere generated sequentially by small balls (POV-Ray 3.7)

VIEW ANIMATION (requires internet connection)

POV-Ray 3.7 code to generate fig. 8

```
//================================================
// Sequential 3D surface generation (sphere)
// by POV-Ray 3.7
//================================================

#include "glass.inc"       // include pigment files
#include "colors.inc"
#include "metals.inc"

global_settings {assumed_gamma 1.0}    // set gamma value
background { color rgb <00,0.0,0.0> }   // set black background color

camera {                                    // set view point (camera) location
    location <-10, 10, -30>
    look_at  <0, 0,  0>
}

light_source {                             // set point light sources location
  < -50, 20, -10>
  rgb <1.000000, 1.000000, 1.000000> * 2.0   // set white light color and intensity
 }

light_source {
```

```
  < 20, 20, -10>
  rgb <1.000000, 1.000000, 1.000000> * 2.0   // set white light color and intensity
}

light_source {
    <-3, 10, -3>
    color White
}

#declare R = 9;               // set spere radius value
#declare n = 100;          // set number of cycles
#declare X = array[n+1][n+1];   // define x co-rdinate array
#declare Y = array[n+1][n+1];   // define y co-rdinate array
#declare Z = array[n+1][n+1];   // define z co-rdinate array

#for (p, 0, n)     // polar co-ordinate theta cycle
#declare Th = p*2*pi/n;

#for (q, 0, n)    // polar co-ordinate phi cycle
#declare Ph = q*pi/n;

#declare X[p][q] = R*cos(Th)*sin(Ph);    // polar to Cartesian co-ordinates
#declare Y[p][q] = R*sin(Th)*sin(Ph);
#declare Z[p][q] = R*cos(Ph);

#end  // end for q   // for cycles end
#end  // end for p

#declare i = 0;    // define animation cycles indices
#declare j = 0;

#if ( i <= n )   //  index conditionals
#if (j <= n )
#declare j = (n+5)*clock;   // animation clock
#declare i = (n+5)*clock;

#for(p,0,i)   // individual cell plotting sequential cycles
#for(q,0,j)

sphere {
       < X[p][q], Y[p][q], Z[p][q] >, .4   // spherical cell location and radius
     texture {                             // pigment and finishing choices
     pigment{ P_Copper3 }}
     finish {
      ambient .1
      specular .5
      metallic
}}

 #end // end for q    //  for cycles end
 #end // end for p
 #end  // end for j   // conditionals end
 #end  // end for i
```

3.2.2 Sphere generated randomly by small balls

The same spherical *ordered* structure can be generated also choosing *randomly* the co-ordinates of each elementary ball, instead of assigning them sequentially.

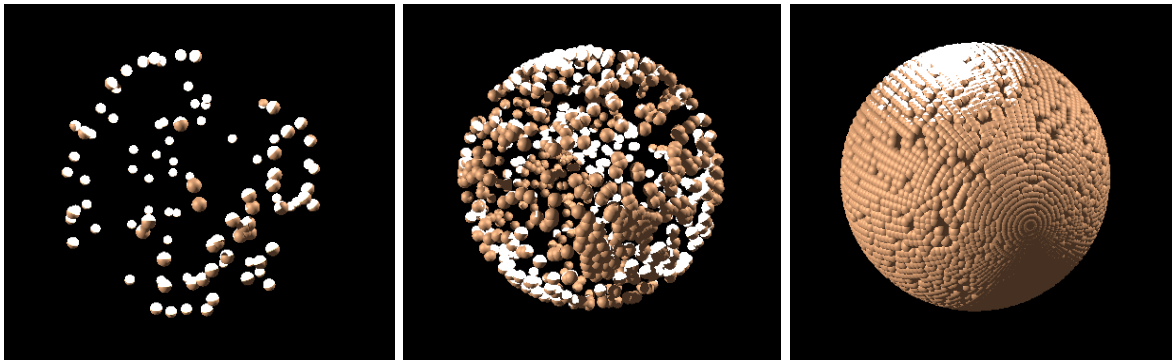


Fig.9 - Sphere generated randomly by small balls (POV-Ray 3.7)

VIEW ANIMATION (requires internet connection)

POV-Ray 3.7 code to generatefig. 9

```
//===============================================
// Random 3D surface generation (Sphere)
//  by POV-Ray 3.7
//===============================================

#include "glass.inc"       // include pigment files
#include "colors.inc"
#include "metals.inc"

global_settings {assumed_gamma 1.0}    // set gamma value
background { color rgb <00,0.0,0.0> }   // set black background color

camera {                                    // set view point (camera) location
    location <-10, 10, -30>
    look_at  <0, 0,  0>
}

light_source {                             // set point light sources location
  < -50, 20, -10>
  rgb <1.000000, 1.000000, 1.000000> * 2.0   // set white light color and intensity
 }

light_source {
  < 20, 20, -10>
  rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

light_source {
    <-3, 10, -3>
    color White
}

#declare R = 9;              // set spere radius value
#declare n = 100;          // set number of cycles
#declare X = array[n+1][n+1];   // define x co-rdinate array
#declare Y = array[n+1][n+1];   // define y co-rdinate array
```

```
#declare Z = array[n+1][n+1];   // define z co-rdinate array

#for (p, 0, n)     // polar co-ordinate theta cycle
#declare Th = p*2*pi/n;

#for (q, 0, n)    // polar co-ordinate phi cycle
#declare Ph = q*pi/n;

#declare X[p][q] = R*cos(Th)*sin(Ph);   // polar to Cartesian co-ordinates
#declare Y[p][q] = R*sin(Th)*sin(Ph);
#declare Z[p][q] = R*cos(Ph);

#end  // end for q   // for cycles end
#end  // end for p

#declare Rnd_1 = seed (1153);     // declare random indices
#declare Rnd_2 = seed (553) ;

#declare j = (n+5)*clock;     // animation clock
#declare i = (n+5)*clock;

#for(r,0,i)    //  individual cell plotting random cycles
#for(s,0,j)
#declare p = int(n*rand(Rnd_1));   // set random variables
#declare q = int(n*rand(Rnd_2));

sphere {
       < X[p][q], Y[p][q], Z[p][q] >, .4   // spherical cell location and radius
    texture {                                    // pigment and finishing choices
    pigment{ P_Copper3 }}
    finish {
    ambient .1
    specular .5
    metallic
}}

 #end //    for cycles end
 #end
```

3.3 Three-dimensional fractals

In the next three chapters we will apply the methodologies we have elaborated until now to render *complex* $3D$ ordered structures as *fractals* generated

– either by *sequentially ordered* processes

– or by *random* process

guided by a same *algorithmic law*.

As we have seen for the two-dimensional structures, *order*

– appears *manifestly* when a *sequentially ordered* process is employed

– while it seems to be *hidden by chance* when a *random* process is performed.

a) A first class of $3D$ *fractals* we will present is offered by the so called *fractal landscapes.*[6] Resulting images may look even very impressive and beautiful, but on a mathematical standpoint they do not increase the *level of complexity* involved in the related $2D$ fractals.

b) A second class of $3D$ *fractals* can be built performing a *rotation* of a $2D$ *fractal* around a symmetry axis, if any.[7] Also this procedure does not increase the *level of complexity* of the structure, notwithstanding the beauty of the resulting pictures.

c) A third class of $3D$ *complex fractal structures* is obtained recurring to *quaternions* or *hpercomplex numbers* which generalize complex numbers adding further imaginary units. In this case the *level of complexity* is *actually increased* respect to the related $2D$ sets of *Mandelbrot, Julia* and *Newton's method* kinds.[8]

[6] See, chapter 4.
[7] See, chapter 5.
[8] See, chapter 6.

Chapter 4

$3D$ fractal landscapes

Rendering structures as wholes or by sequential or random processes

4.1 *Python 3* rendering of fractal landscapes

Fractal landscapes represent the first attempt to add a third dimension to fractal structures. The method consists in adding a third Cartesian axis (z) to the abscissa (x) and ordinate (y) axes of a plane fractal structure, which is naturally provided by the *escape rate* (number of recursion cycles needed to reach some threshold value) or by the value achieved by the modulus of a recursive function after a suitable number of cycles characterizing each point of the related $2D$-fractal. *Python 3* offers several $3D$ plotting methods like *mesh, contour* and *scatter* plots which are generally applied to graphics of functions and data, and can be usefully used even to paint *fractal landscapes* as wholes. Here are some examples of each method.

4.1.1 Mesh plots

Mesh plots are generally useful to render *regular* surfaces as *wholes,* defined by *Cartesian* or *parametric* equations. Surprisingly *matplotlib* module proves efficient even in rendering fractal surfaces, which are highly non-regular because of strong jumps between the values assumed even in very near points in the domain. The *escape rate* or the *modulus of the recursive function* can be represented either as *elevation* respect to the ground xy plane or as *depth*, so that landscapes similar to *mountain* or *valley* and *lake, sea* scenarios can be realized.

a) Here are two scenes of *Mandelbrot landscapes,* the former as a *mountain* and the latter as a *lake*. A suitable choice of the color maps contributes essentially to the visual impact. Computational algorithm is based on complex arrays.[1]

[1] See chap. 2, §2.1.2. The method is very fast but does not allow an high number of recursion cycles because of overflow occurrences.

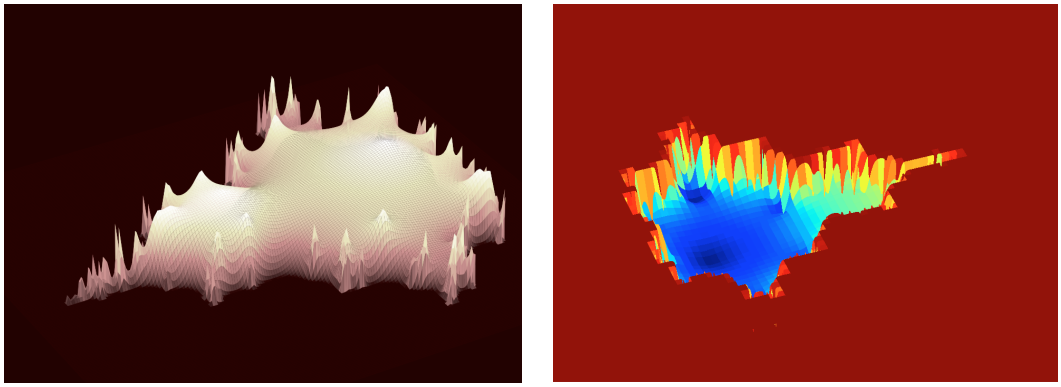


Fig.1 - Python 3 matplotlib landscapes mesh rendering of a Mandelbrot set as a whole

---

Python 3 codes to generate fig 1 pictures

```
############################################################
# Mandelbrot mountain landscape generation as a whole
# (matplotlib Mesh plot)
############################################################

import matplotlib.pyplot as plt        # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module

fig = plt.figure()   # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=-120,elev=45)  # set view orientation
ax.dist = 5.0  # set viewpoint distance
ax.set_facecolor([.1,0.0,0.0])  # set ground color

n = 8      # set number of cycles
dx = -0.7     # set initial x parameter shift
dy = 0.0      # set initial y parameter shift
L = 1.5     # set square area side
M = 200   # set side number of pixels

def f(Z):    # def scale damping of the elevation function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)      # x variable array
y = np.linspace(-L+dy,L+dy,M)       # y variable array
X,Y = np.meshgrid(x,y)     # square area grid
Z = np.zeros(M)       # complex plane starting points area
W = np.zeros((M,M))   # zero matrix of elevation values
C = X + 1j*Y       # complex plane area

for k in range(1,n+1):     # recursion cycle
   ZZ = Z**2 + C
   Z = ZZ
   W =  f(Z)     # smoothed sum moduls
```

```
ax.set_xlim(dx-L,dx+L)     # set x axis limits
ax.set_zlim(dy-L,dy+L)     # set y axis limits
ax.set_zlim(-L,2*L)     # set z axis limits
ax.axis("off")    # do not plot axes
ax.plot_surface(X, Y, W, rstride=1, cstride=1, cmap="pink")   # plot surface as a whole
plt.show()   # show plot



##########################################################
# Mandelbrot valley landscape generation as a whole
# (matplotlib Mesh plot)
##########################################################

import matplotlib.pyplot as plt    # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm   # import color maps module
import numpy as np   # import numpy module

fig = plt.figure()    # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=70,elev=70)  # set view orientation
ax.dist = 6  # set viewpoint distance
ax.set_facecolor([.5,0.0,0.0])  # set ground color


n = 9    # set number of cycles
dx = -0.6     # set initial x parameter shift
dy = 0.0      # set initial y parameter shift
L = 1.4     # set square area side
M = 400   # set side number of pixels

x = np.linspace(dx-L,dx+L,M)   # x variable array
y = np.linspace(dy-L,dy+L,M)   # y variable array
X,Y = np.meshgrid(x,y,sparse=True)    # square area grid
Z = np.zeros(M)    # complex plane starting points area
C = X + 1j*Y     # complex plane area

for k in range(1,n+1):      # recursion cycle
   ZZ = Z*Z + C
   Z = ZZ
   W = np.e**(-.6*np.abs(Z))    # smoothed sum moduls

ax.set_xlim(dx-L,dx+L)    # set x axis limits
ax.set_zlim(dy-L,dy+L)    # set y axis limits
ax.set_zlim(-L,L)   # set z axis limits
ax.axis("off")   # do not plot axes

# plot surface as a whole
surf = ax.plot_surface(X, Y, -W, cmap=cm.jet,linewidth=0, antialiased=False)
plt.show()    show plot
```

---

b) Here are two scenes of *Julia landscapes,* the former as a *mountain* and the latter as a *lake* and the related *python 3* code. Of course also in this picture a suitable choice of

the color maps contributes essentially to the visual effect. Computational algorithm is based too on complex arrays.[2]

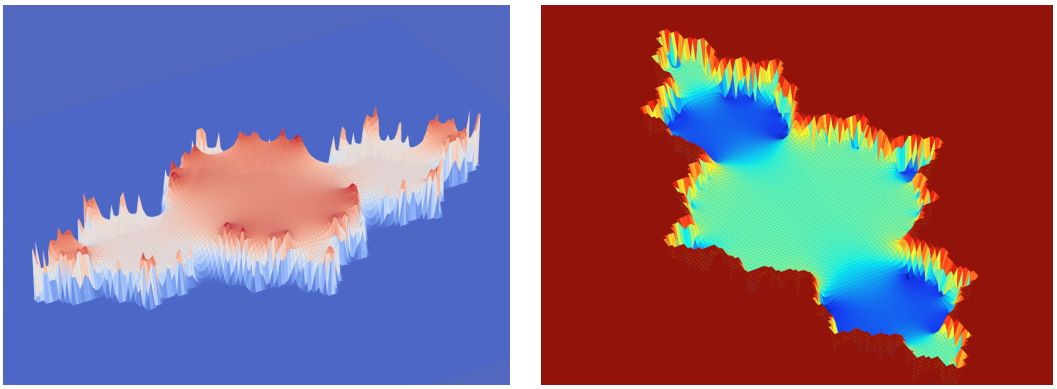


Fig.2 - Python 3 landscapes mesh rendering of a Julia set as a whole

---

Python 3 codes to generate fig 2 pictures

```
#########################################################
# Julia mountain landscape generation as a whole
# (matplotlib Mesh plot)
#########################################################

import matplotlib.pyplot as plt      # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np    # import numpy module

fig = plt.figure()    # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=150,elev=60)  # set view orientation
ax.dist = 3.5  # set viewpoint distance
ax.set_facecolor([0.5,0.0,0.0])  # set ground color

n = 9     # set number of cycles
dx = 0.0    # set initial x parameter shift
dy = 0.0    # set initial y parameter shift
L = 2.0    # set square area side
M = 200  # set side number of pixels

def f(Z):     # def scale damping of the elevation function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)      # x variable array
y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(x,y)    # square area grid
cX = -0.7454294     # set parameter C real part value
```

[2]See chap. 2, §2.1.2.

```
cY = 0     # set parameter C imaginary part value
C = cX + 1j*cY    # complex C matrix
W = np.zeros((M,M))   # zero matrix of elevation values
Z = X + 1j*Y    # complex plane area

for k in range(1,n+1):     # recursion cycle
   ZZ = Z**2 + C
   Z = ZZ
   W =  f(Z)

ax.set_xlim(dx-L,dx+L)    # set x axis limits
ax.set_zlim(dy-L,dy+L)    # set y axis limits
ax.set_zlim(-2*L,2*L)     # set z axis limits
ax.axis("off")   # do not plot axes
ax.plot_surface(X, Y, -W, rstride=1, cstride=1, cmap="jet")  # plot surface as a whole
plt.show()      # show plot



################################################
# Julia valley landscape generation as a whole
# (matplotlib Mesh plot)
################################################

import matplotlib.pyplot as plt        # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module

fig = plt.figure()   # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=150,elev=60)  # set view orientation
ax.dist = 3.5  # set viewpoint distance
ax.set_facecolor([0.5,0.0,0.0])  # set ground color

n = 9     # set number of cycles
dx = 0.0     # set initial x parameter shift
dy = 0.0     # set initial y parameter shift
L = 2.0     # set square area side
M = 200   # set side number of pixels

def f(Z):    # def scale damping of the depth function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)       # x variable array
y = np.linspace(-L+dy,L+dy,M)       # y variable array
X,Y = np.meshgrid(x,y)      # square area grid
cX = -0.7454294     # set parameter C real part value
cY = 0     # set parameter C imaginary part value
C = cX + 1j*cY      # complex C matrix
W = np.zeros((M,M))     # zero matrix of elevation values
Z = X + 1j*Y    # complex plane area

for k in range(1,n+1):     # recursion cycle
   ZZ = Z**2 + C
   Z = ZZ
   W =  f(Z)

ax.set_xlim(dx-L,dx+L)     # set x axis limits
ax.set_zlim(dy-L,dy+L)     # set y axis limits
ax.set_zlim(-2*L,2*L)       # set z axis limits
```

```
ax.axis("off")   # do not plot axes
ax.plot_surface(X, Y, -W, rstride=1, cstride=1, cmap="jet")  # plot surface as a whole
plt.show()   # show plot
```

---

c) And here are two scenes of *Newton's method set landscapes,* the former as a *mountain* and the latter as a *lake* and the related *python 3* code.[3]

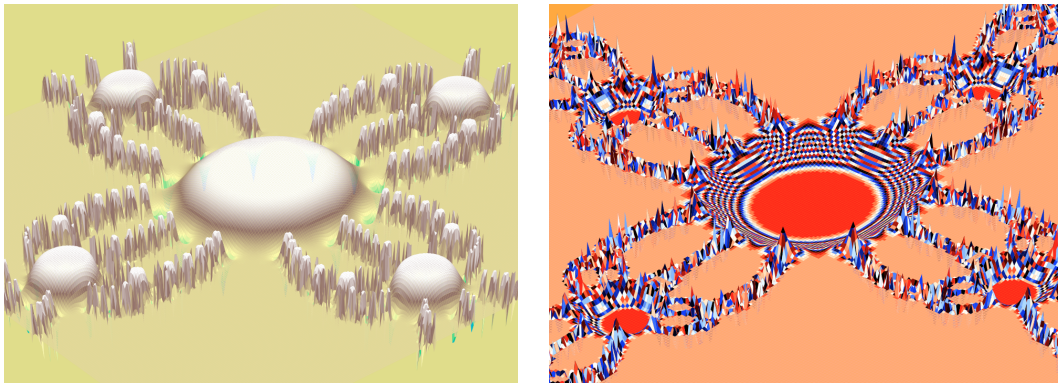


Fig.3 - Python 3 matplotlib landscapes mesh rendering of a Newton set as a whole

---

Python 3 codes to generate fig 3 pictures

```
######################################################################
# Newton's method mountain landscape generation as a whole
# (matplotlib Mesh plot)
######################################################################

import matplotlib.pyplot as plt      # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np    # import numpy module

fig = plt.figure()    # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=-130,elev=45)  # set view orientation
ax.dist = 4.3  # set viewpoint distance
ax.set_facecolor([.85,.85,.45])  # set ground color

n = 8     # set number of cycles
dx = 0.0    # set initial x parameter shift
dy = 0.0    # set initial y parameter shift
L = 1.0   # set square area side
M = 300  # set side number of pixels
```

[3] See chap. 2, §2.1.2.

```
def f(Z):     # def scale damping of the elevation function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)      # x variable array
y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(x,y)    # square area grid
Z = X + 1j*Y   # complex plane area

for k in range(1,n+1):     # recursion cycle
   ZZ = Z - (Z**4 + 1)/(4*Z**3)
   Z = ZZ
   W =  f(Z)

ax.set_xlim(dx-L,dx+L)    # set x axis limits
ax.set_zlim(dy-L,dy+L)    # set y axis limits
ax.set_zlim(-2.5*L,2*L)    # set z axis limits
ax.axis("off")   # do not plot axes
ax.plot_surface(X, Y, -W, rstride=1, cstride=1, cmap="terrain")   # plot surface as a whole
plt.show()     # show plot



############################################################
# Newton's method valley landscape generation as a whole
# (matplotlib Mesh plot)
############################################################

import matplotlib.pyplot as plt      # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np    # import numpy module

fig = plt.figure()    # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=-140,elev=45)  # set view orientation
ax.dist = 3.0  # set viewpoint distance
ax.set_facecolor([1.0,.6,.15])  # set ground color

n = 8     # set number of cycles
dx = 0.0    # set initial x parameter shift
dy = 0.0    # set initial y parameter shift
L = 1.3   # set square area side
M = 300  # set side number of pixels

def f(Z):     # def scale damping of the depth function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)      # x variable array
y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(x,y)    # square area grid
Z = X + 1j*Y   # complex plane area

for k in range(1,n+1):     # recursion cycle
   ZZ = Z - (Z**4 + 1)/(4*Z**3)
   Z = ZZ
   W =  f(Z)

ax.set_xlim(dx-L,dx+L)    # set x axis limits
ax.set_zlim(dy-L,dy+L)    # set y axis limits
ax.set_zlim(-3.5*L,4*L)    # set z axis limits
```

```
ax.axis("off")   # do not plot axes
ax.plot_surface(X, Y, W, rstride=1, cstride=1, cmap='flag')   # plot surface as a whole
plt.show()     # show plot
```

4.1.2 Contour plots

A spectacular alternative to *mesh* plots to generate *fractal landscapes* is offered by *contour* plots. The advantage of this methodology is that of showing the level curves along which the *modulus of the recursive function* assumes a constant value. In the following figures we show the same structure sets (*Mandelbrot, Julia* and *Newton's method*) related contour plot landscapes in order to offer to the reader the ability to compare the results.

a) *Mandelbrot contour plot landscapes*

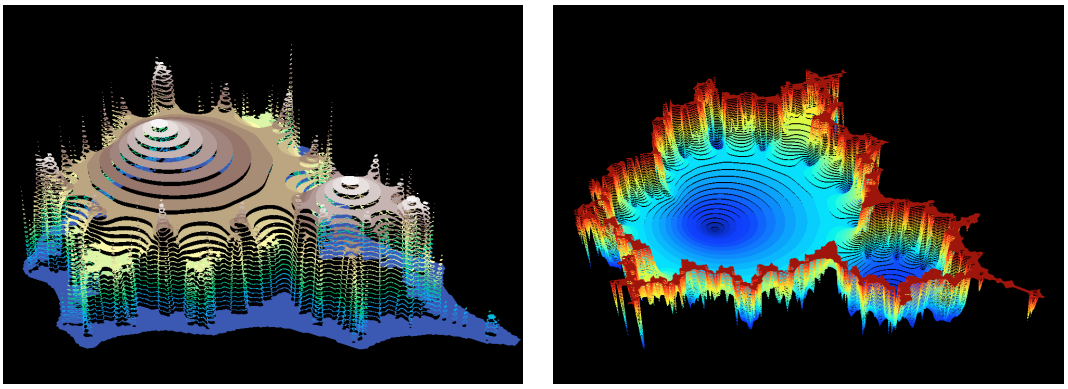


Fig.4 - Python 3 matplotlib landscapes contour rendering of a Mandelbrot set as a whole

---

Python 3 codes to generate fig 4 pictures

```
############################################################
# Mandelbrot mountain landscape generation as a whole
# (matplotlib Contour plot)
############################################################

import matplotlib.pyplot as plt        # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module

fig = plt.figure()   # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=120,elev=45)  # set view orientation
```

```
ax.dist = 5  # set viewpoint distance
ax.set_facecolor([0.0,0.0,0.0])  # set background color

n = 16      # set number of cycles
dx = -0.6     # set initial x parameter shift
dy = 0.0     # set initial y parameter shift
L = 1.3    # set square area side
M = 200  # set side number of pixels

def f(Z):      # def scale damping of the elevation function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)      # x variable array
y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(x,y)     # square area grid
Z = np.zeros(M)      # complex plane starting points area
W = np.zeros((M,M))   # zero matrix of elevation values
C = X + 1j*Y      # complex plane area

for k in range(1,n+1):     # recursion cycle
   ZZ = Z**2 + C
   Z = ZZ
   W =  f(Z)

ax.set_xlim(dx-L,dx+L)     # set x axis limits
ax.set_zlim(dy-L,dy+L)     # set y axis limits
ax.set_zlim(-.5*L,1.5*L)    # set z axis limits
ax.axis("off")    # do not plot axes
ax.contourf3D(X, Y, W, 2*n, cmap="terrain")   # make contour plot
plt.show()   # show plot



##########################################################
# Mandelbrot valley landscape generation as a whole
# (matplotlib Contour plot)
##########################################################

import matplotlib.pyplot as plt        # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module

fig = plt.figure()   # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=120,elev=60)  # set view orientation
ax.dist = 4.5  # set viewpoint distance
ax.set_facecolor([0.0,0.0,0.0])  # set background color

n = 20      # set number of cycles
dx = -0.6     # set initial x parameter shift
dy = 0.0     # set initial y parameter shift
L = 1.3    # set square area side
M = 200  # set side number of pixels

def f(Z):     # def scale damping of the elevation function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)      # x variable array
y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(x,y)     # square area grid
```

```
Z = np.zeros(M)      # complex plane starting points area
W = np.zeros((M,M))   # zero matrix of elevation values
C = X + 1j*Y      # complex plane area

for k in range(1,n+1):     # recursion cycle
   ZZ = Z**2 + C
   Z = ZZ
   W =  f(Z)

ax.set_xlim(dx-L,dx+L)     # set x axis limits
ax.set_zlim(dy-L,dy+L)     # set y axis limits
ax.set_zlim(-L,1.5*L)         # set x axis limits
ax.axis("off")   # do not plot axes
ax.contourf3D(X, Y, -W, 2*n, cmap="jet")   # make contour plot
plt.show()   # show plot
```

---

b) *Julia contour plot landscapes*

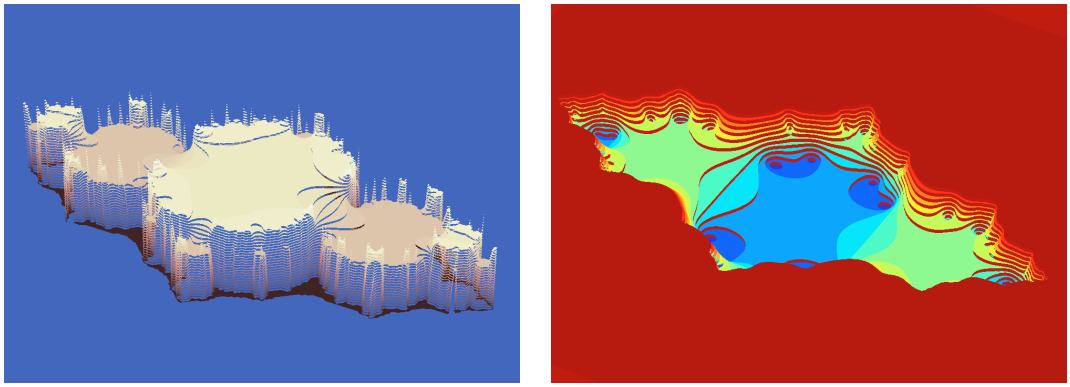


Fig.5 - Python 3 matplotlib landscapes contour rendering of a Julia set as a whole

---

Python 3 codes to generate fig 5 pictures

```
#########################################################
# Julia mountain landscape generation as a whole
# (matplotlib Contour plot)
#########################################################

import matplotlib.pyplot as plt        # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module

fig = plt.figure()   # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=120,elev=45)  # set view orientation
```

```
ax.dist = 4.5  # set viewpoint distance
ax.set_facecolor([0.0,0.3,0.6])  # set background color

n = 16     # set number of cycles
dx = -0.1    # set initial x parameter shift
dy = 0.0     # set initial y parameter shift
L = 1.7   # set square area side
M = 200  # set side number of pixels

def f(Z):     # def scale damping of the elevation function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)      # x variable array
y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(x,y)     # square area grid
cX = -0.7454294   # C parameter real part value
cY = 0   # C parameter imaginary part value
C = cX + 1j*cY      # complex C matrix
W = np.zeros((M,M))     # zero matrix of elevation values
Z = X + 1j*Y    # complex plane area

for k in range(1,n+1):    # recursion cycle
   ZZ = Z**2 + C
   Z = ZZ
   W =  f(Z)

ax.set_xlim(dx-L,dx+L)    # set x axis limits
ax.set_zlim(dy-L,dy+L)    # set y axis limits
ax.set_zlim(-.8*L,1.5*L)   # set z axis limits
ax.axis("off")   # do not plot axes
ax.contourf3D(X, Y, W, 2*n, cmap="pink")   # make contour plot
plt.show()  # show plot



#####################################################
# Julia valley landscape generation as a whole
# (matplotlib Contour plot)
#####################################################

import matplotlib.pyplot as plt        # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module

fig = plt.figure()  # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=120,elev=60)  # set view orientation
ax.dist = 3.7  # set viewpoint distance
ax.set_facecolor([0.7,0.0,0.0])  # set background color

n = 6    # set number of cycles
dx = -0.1    # set initial x parameter shift
dy = 0.0    # set initial y parameter shift
L = 2.0   # set square area side
M = 200  # set side number of pixels

def f(Z):     # def scale damping of the elevation function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)     # x variable array
```

```
y = np.linspace(-L+dy,L+dy,M)     # y variable array
X,Y = np.meshgrid(x,y)     # square area grid
cX = -0.7454294   # C parameter real part value
cY = 0  # C parameter imaginary part value
C = cX + 1j*cY     # complex C matrix
W = np.zeros((M,M))     # zero matrix of depth values
Z = X + 1j*Y   # complex plane area

for k in range(1,n+1):    # recursion cycle
   ZZ = Z**2 + C
   Z = ZZ
   W =  f(Z)

ax.set_xlim(dx-L,dx+L)    # set x axis limits
ax.set_zlim(dy-L,dy+L)    # set y axis limits
ax.set_zlim(-1.5*L,1.5*L)    # set z axis limits
ax.axis("off")   # do not plot axes
ax.contourf3D(X, Y, -W, 2*n, cmap="jet")   # make contour plot
plt.show() # show plot
```

---

c) *Newton's method contour plot landscapes*

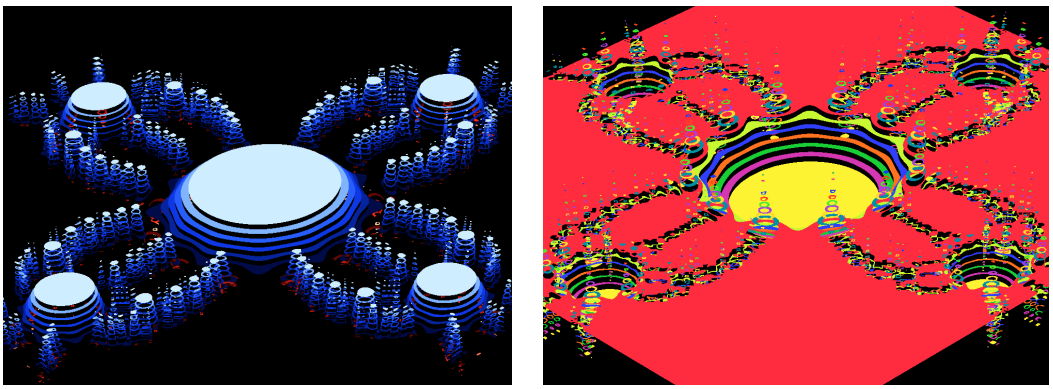


Fig.6 - Python 3 matplotlib landscapes contour rendering of a Newton's method set as a whole

---

Python 3 codes to generate fig 6 pictures

```
################################################################
# Newton's method mountain landscape generation as a whole
# (matplotlib Contour plot)
################################################################

import matplotlib.pyplot as plt        # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module
```

```
fig = plt.figure()  # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=-130,elev=45)  # set view orientation
ax.dist = 4.3  # set viewpoint distance
ax.set_facecolor([.85,.85,.45])  # set background color

n = 8     # set number of cycles
dx = 0.0    # set initial x parameter shift
dy = 0.0    # set initial y parameter shift
L = 1.0  # set square area side
M = 300  # set side number of pixels

def f(Z):     # def scale damping of the depth function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)      # x variable array
y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(x,y)     # square area grid
Z = X + 1j*Y   # complex plane area

for k in range(1,n+1):    # recursion cycle
   ZZ = Z - (Z**4 + 1)/(4*Z**3)
   Z = ZZ
   W =  f(Z)

ax.set_xlim(dx-L,dx+L)    # set x axis limits
ax.set_zlim(dy-L,dy+L)    # set y axis limits
ax.set_zlim(-2.5*L,2*L)    # set z axis limits
ax.axis("off")   # do not plot axes
ax.plot_surface(X, Y, -W, rstride=1, cstride=1, cmap="terrain")   # make contour plot
plt.show()    # show plot



############################################################
# Newton's method valley landscape generation as a whole
# (matplotlib Contour plot)
############################################################

import matplotlib.pyplot as plt        # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module

fig = plt.figure()  # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=-140,elev=45)  # set view orientation
ax.dist = 3.0  # set viewpoint distance
ax.set_facecolor([1.0,.6,.15])  # set background color

n = 8    # set number of cycles
dx = 0.0    # set initial x parameter shift
dy = 0.0    # set initial y parameter shift
L = 1.3  # set square area side
M = 300  # set side number of pixels

def f(Z):     # def scale damping of the elevation function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)      # x variable array
```

```
y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(x,y)     # square area grid
Z = X + 1j*Y   # complex plane area

for k in range(1,n+1):    # recursion cycle
   ZZ = Z - (Z**4 + 1)/(4*Z**3)
   Z = ZZ
   W =  f(Z)

ax.set_xlim(dx-L,dx+L)    # set x axis limits
ax.set_zlim(dy-L,dy+L)    # set y axis limits
ax.set_zlim(-3.5*L,4*L)    # set z axis limits
ax.axis("off")   # do not plot axes
ax.plot_surface(X, Y, W, rstride=1, cstride=1, cmap='flag')   # make contour plot
plt.show()    # show plot
```

4.1.3 Scatter plots

Sequential rendering

The *point by point* process of construction of $3D$ ordered structures requires a longer time machine calculation than the $2D$ structures, unless we drastically reduce the image resolution. As we have already seen about the *sphere*[4] it may performed:

- either by *Python 3* employing *graphics* modulus or *matplotlib* modulus
- or, with better rendering results, by *POV-Ray 3.7*

following a *sequentially ordered* strategy or a *random* strategy, which apparently hides the *law* governing an *algorithm*.

We will see, now, how to apply those methods to build *fractal landscapes*. We limit ourselves to one example for *Mandelbrot* set and one example for a *Julia* set, leaving to the interested reader the pleasure (or trouble...) of implementing further more applications.[5]

a) *Mandelbrot scatter sequential landscape*

Python 3 code to generate fig 7

```
##############################################
# Sequential Mandelbrot mountain landscape
# generation (matplotlib scatter plot)
##############################################

import matplotlib.pyplot as plt        # import matplotlib modules
```

[4] See, §3.2.1 and §3.2.2.

[5] The algorithmic procedure here implemented involves a compression of the shape during the generation process. While to speed up animation we have reduced resolution and taken of account of the symmetry of the set.

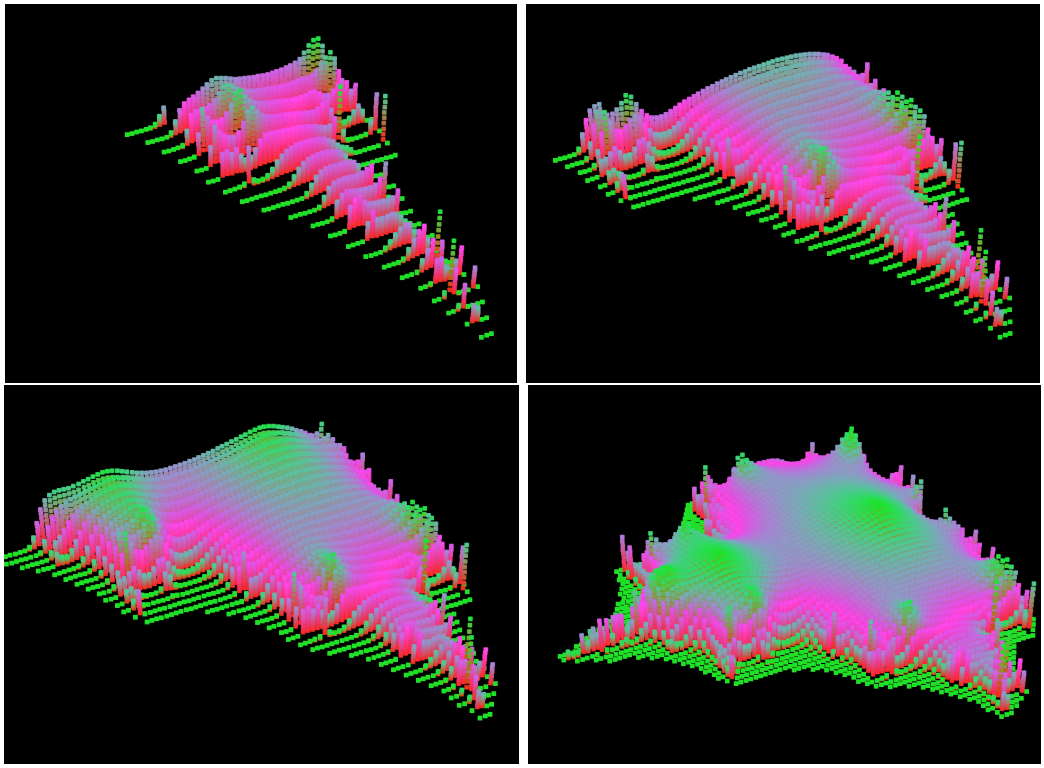


Fig.7 - Mandelbrot mountain landscape generated sequentially by small disks (matplotlib scatter plot)

VIEW ANIMATION (requires internet connection)

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module

fig = plt.figure()  # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=-120,elev=45)  # set view orientation
ax.dist = 5.0  # set viewpoint distance
ax.set_facecolor([0.0,0.0,0.0])  # set background color

n = 8   # set number of cycles
dx = -0.7  # set initial x parameter shift
dy = 0.0   # set initial y parameter shift
L = 1.5   # set square area side
M = 200  # set side number of pixels

def f(Z):     # def scale damping of the elevation function
    return np.e**(-np.abs(Z))

x = np.linspace(-L+dx,L+dx,M)      # x variable array
y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(x,y)     # square area grid
Z = np.zeros(M)     # complex plane starting points area
```

```
W = np.zeros((M,M))   # zero matrix of elevation values
C = X + 1j*Y     # complex plane area

for k in range(1,n+1):     # recursion cycle
   ZZ = Z**2 + C
   Z = ZZ
   W =  f(Z)

ax.set_xlim(dx-L,dx+L)     # set x axis limits
ax.set_zlim(dy-L,dy+L)     # set y axis limits
ax.set_zlim(-L,2*L)           # set z axis limits
ax.axis("off")   # do not plot axes

for p in range(1,M,2):     # make scatter sequential plot
   for q in range(1,M,2):
      if W[p,q] > 0:
         for i in range(1,10,1):
            ax.scatter(X[p,q], Y[p,q], W[p,q]*i/10, s=2, c=
            [1.0-np.abs(np.cos(np.pi*W[p,q]))*i/10,
            np.abs(np.cos(np.pi*W[p,q]))*i/10,
            np.abs(np.sin(np.pi*W[p,q]))*i/10], cmap='cm.hsv', marker="s",zorder=2)

      plt.pause(0.01)     # animation pause interval

plt.show()   # show plot
```

---

b) *Julia ($c = -0.7454294$) scatter sequential landscape*

Python 3 code to generate fig 8

```
############################################################
# Sequential Julia (c = -0.7454294) mountain landscape
# generation (matplotlib scatter plot)
############################################################

import matplotlib.pyplot as plt        # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module

fig = plt.figure()  # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=200,elev=60)  # set view orientation
ax.dist = 3.5  # set viewpoint distance
ax.set_facecolor([0.0,0.1,0.2])  # set background color

n = 11   # set number of cycles
R = 0.1   # set radius value

dx = 0.0  # set initial x parameter shift
dy = 0.0  # set initial y parameter shift
L = 1.4   # set square area side
M = 400  # set side number of pixels

X = np.linspace(-L+dx,L+dx,M)      # x variable array
```

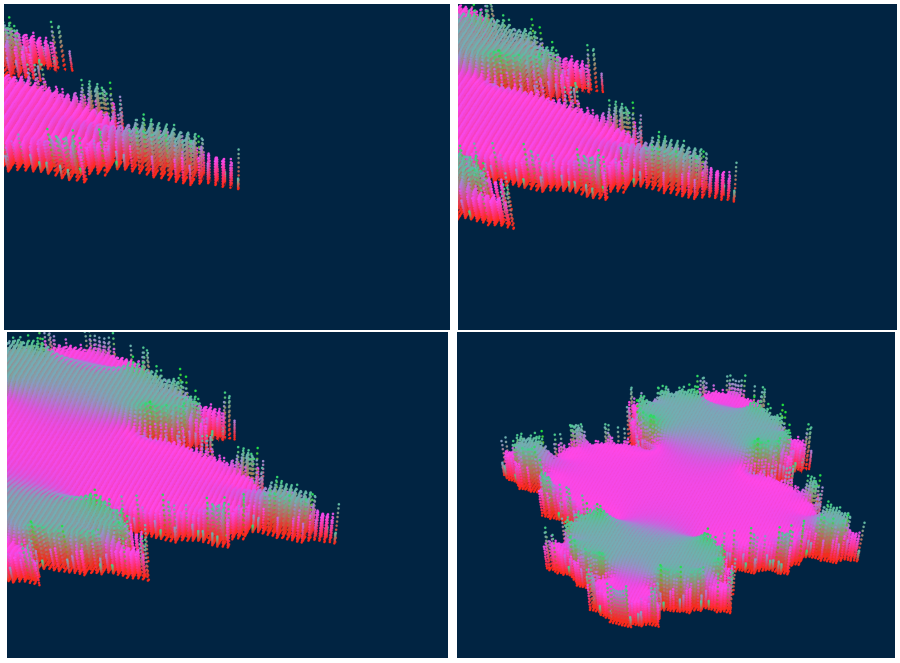


Fig.8 - Julia ($c = -0.7454294$) mountain landscape generated sequentially by small disks (matplotlib scatter plot)

VIEW ANIMATION (requires internet connection)

```
Y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(X,Y)     # square area grid
Z = X + 1j*Y    # complex plane area

cX = -0.7454294  # set c parameter real part value
cY = 0.0   # set c parameter imaginary part value

C = cX + 1j*cY       # complex C matrix
W = np.zeros((M,M))      # zero matrix of depth values

for k in range(0,n):    # recursion cycle
   ZZ = Z**2 + C
   Z = ZZ
W =  np.e**(-np.abs(Z))

ax.set_xlim(dx-2*L,dx+2*L)   # set x axis limits
ax.set_zlim(dy-2*L,dy+2*L)   # set y axis limits
ax.set_zlim(-1.3*L,2*L)         # set z axis limits
ax.axis("off")   # do not plot axes

for q in range(0,M,2):      # make scatter sequential plot
   for p in range(0,M,2):
      if W[p,q] > R:
         for i in range(0,10,1):
            ax.scatter(X[p,q], Y[p,q], W[p,q]*i/10, s = 1, c =
            [1.0-np.abs(np.cos(np.pi*W[p,q]))*i/10,
            np.abs(np.cos(np.pi*W[p,q]))*i/10,
```

```
            np.abs(np.sin(np.pi*W[p,q]))*i/10], marker="o",zorder=2)
      else:
         ax.scatter(L+X[p,q], Y[p,q], 0.0, s = 1, c =
         [0.0,0.1,0.2], marker="o",zorder=2)

    plt.pause(0.001)     # animation pause interval

plt.show()    # show plot
```

Random rendering

a) *Mandelbrot scatter random landscape*

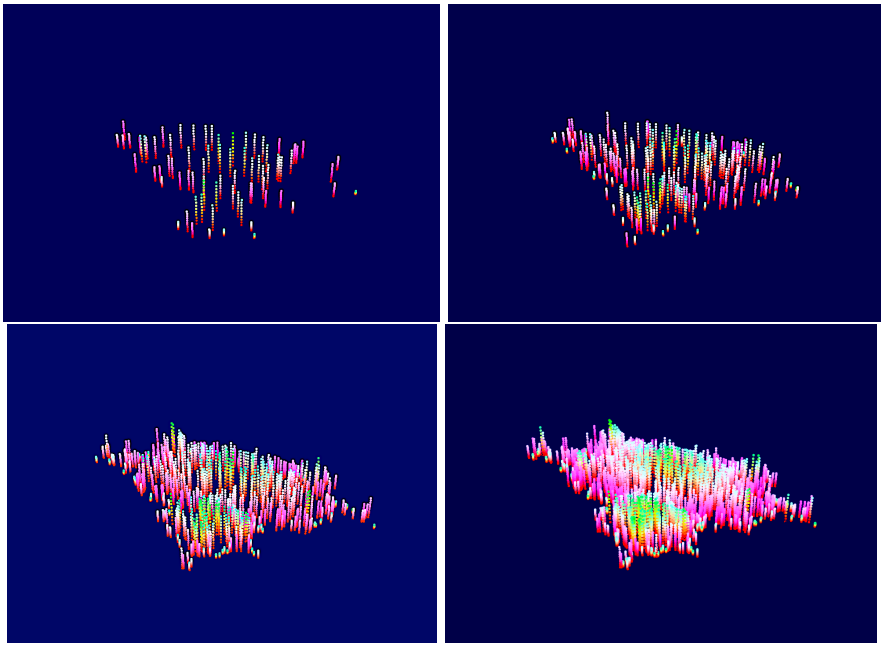


Fig.9 - Mandelbrot mountain landscape generated randomly by small disks (matplotlib scatter plot)

VIEW ANIMATION (requires internet connection)

Python 3 code to generate fig 9

```
##########################################
# Random Mandelbrot mountain landscape
# generation (matplotlib scatter plot)
##########################################

import matplotlib.pyplot as plt        # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
```

```
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module
import random as rd  # import random module

fig = plt.figure()  # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=200,elev=60)  # set view orientation
ax.dist = 5.0  # set viewpoint distance
ax.set_facecolor([0.0,0.1,0.2])  # set background color

n = 8   # set number of cycles
dx = -0.3   # set initial x parameter shift
dy = 0.0    # set initial y parameter shift
R = 0.1  # set radius value
L = 1.5   # set square area side
M = 400  # set side number of pixels

X = np.linspace(-L+dx,L+dx,M)      # x variable array
Y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(X,Y)    # square area grid
Z = np.zeros(M)    # complex plane starting points area

cX = np.linspace(-L+dx,L+dx,M)       # c parameter real part array
cY = np.linspace(-L+dy,L+dy,M)       # c parameter imaginary part array

cX,cY = np.meshgrid(cX,cY)  #  c parameter grid

C = cX + 1j*cY      # complex C matrix
W = np.zeros((M,M))   # zero matrix of elevation values

for k in range(0,n):     # recursion cycle
   ZZ = Z**2 + C
   Z = ZZ
W =  np.e**(-np.abs(Z))


ax.set_xlim(dx-2*L,dx+2*L)     # set x axis limits
ax.set_zlim(dy-2*L,dy+2*L)     # set y axis limits
ax.set_zlim(-1.3*L,2*L)           # set z axis limits
ax.axis("off")   # do not plot axes


while R > 0:       # make scatter random plot
   p = rd.randrange(0,M,1)
   q = rd.randrange(0,M,1)
   if W[p,q] > R:
      for i in range(0,10,1):
         ax.scatter(X[p,q], Y[p,q], W[p,q]*i/10, s = 1, c =
         [1.0-np.abs(np.cos(np.pi*W[p,q]))*i/10,
         np.abs(np.cos(np.pi*W[p,q]))*i/10,
         np.abs(np.sin(np.pi*W[p,q]))*i/10], marker="o",zorder=2)
   else:
      ax.scatter(X[p,q], Y[p,q], 0.0, s = 1, c = [0.0,0.1,0.2], marker="o",zorder=2)

   plt.pause(0.001)  # animation pause interval

plt.show()   # show plot
```

b) *Julia* ($c = -0.7454294$) *scatter random landscape*

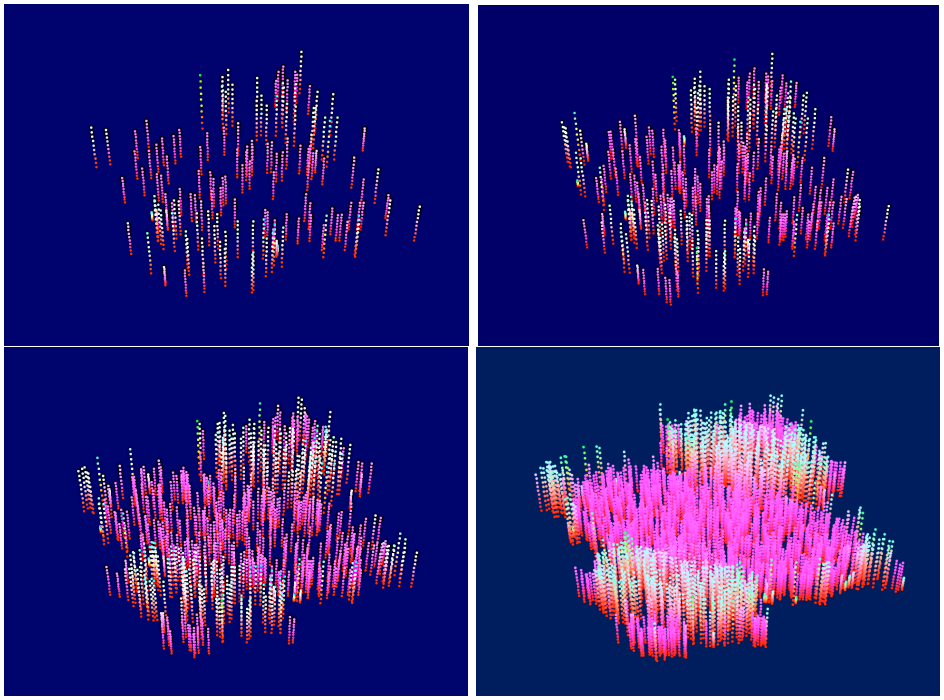


Fig.10 - Julia mountain landscape generated randomly by small disks (matplotlib scatter plot)

VIEW ANIMATION (requires internet connection)

Python 3 code to generate fig 10

```
##########################################
# Random Julia mountain landscape
# generation (matplotlib scatter plot)
##########################################

import matplotlib.pyplot as plt        # import matplotlib modules
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.figure as fg
from matplotlib import cm    # import color maps module
import numpy as np   # import numpy module
import random as rd  # import random module

fig = plt.figure() # set 3D figure environment
ax =  fig.add_subplot(111, projection='3d')
ax.view_init(azim=200,elev=60)  # set view orientation
ax.dist = 3.5  # set viewpoint distance
ax.set_facecolor([0.0,0.1,0.2])  # set background color

n = 11   # set number of cycles
dx = 0.0   # set initial x parameter shift
dy = 0.0   # set initial y parameter shift
```

```
R = 0.1  # set radius value
L = 1.4  # set square area side
M = 400  # set side number of pixels

X = np.linspace(-L+dx,L+dx,M)      # x variable array
Y = np.linspace(-L+dy,L+dy,M)      # y variable array
X,Y = np.meshgrid(X,Y)   # square area grid
Z = X + 1j*Y     # complex plane area

cX = -0.7454294   # set parameter c real part value
cY = 0.0   # set parameter c imaginary part value

C = cX + 1j*cY      # complex C matrix
W = np.zeros((M,M))   # zero matrix of elevation values

for k in range(0,n):    # recursion cycle
   ZZ = Z**2 + C
   Z = ZZ
W =  np.e**(-np.abs(Z))

ax.set_xlim(dx-2*L,dx+2*L)     # set x axis limits
ax.set_zlim(dy-2*L,dy+2*L)     # set y axis limits
ax.set_zlim(-1.3*L,2*L)            # set z axis limits
ax.axis("off")  # do not plot axes

while R > 0:      # make scatter random plot
   p = rd.randrange(0,M,1)
   q = rd.randrange(0,M,1)
   if W[p,q] > R:
      for i in range(0,10,1):
         ax.scatter(X[p,q], Y[p,q], W[p,q]*i/10, s = 1, c =
         [1.0-np.abs(np.cos(np.pi*W[p,q]))*i/10,
         np.abs(np.cos(np.pi*W[p,q]))*i/10,
         np.abs(np.sin(np.pi*W[p,q]))*i/10]
         , marker="o",zorder=2)
   else:
      ax.scatter(L+X[p,q], Y[p,q], 0.0, s = 1, c =  [0.0,0.1,0.2], marker="o",zorder=2)

   plt.pause(0.001)   # animation pause interval

plt.show()   # show plot
```

4.2 POV-Ray rendering of fractal landscapes

4.2.1 The POV-Ray 3.7 functions “Mandel” and “Julia”

The $3D$ rendering software *POV-Ray 3.7* offers some special functions explicitly devoted to Mandelbrot and Julia sets generated by powers of order n (up to $n = 33$).

Moreover the same “Mandel” and “Julia” functions allow, by rotation in $3D$ to obtain “flat” landscapes of the same sets in a very short computation time.

But No elevation is provided by those tools.

These functions provide refined pictures of those sets as “flat” wholes like, *e.g.,* the ones in figs 11 and 12.

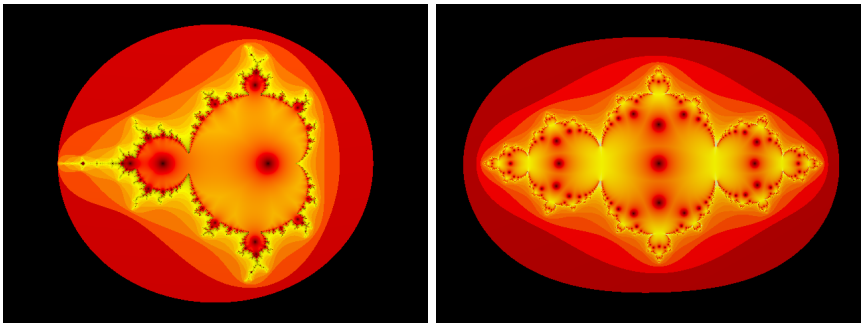


Fig.11 - Mandelbrot and Julia $2D$ sets generated as wholes by POV-Ray 3.7
(functions "Mandel" and "Julia")

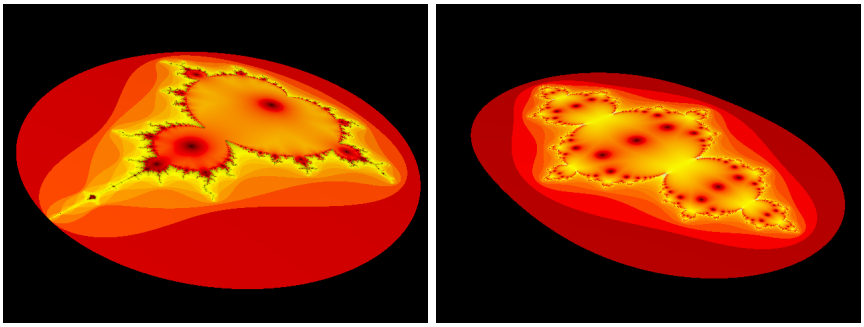


Fig.12 - Mandelbrot and Julia "flat" landscapes sets generated as wholes by POV-Ray 3.7
(functions "Mandel" and "Julia")

POV-Ray 3.7 codes to generate fig 11 and 12

```
//=========================================
// Mandelbrot set 2D and 3D landscape
// generation (POV-Ray 3.7 Mandel function)
//=========================================

 background { color rgb <0.0,0.0,0.0> }    // set black background color

 camera { // set view point (camera) location
   perspective
   location  < 0, 0, -3.5 >
   // location  < 1.0, 2.0, -2.85>    // alternative camera location for landscape
   right x * 1
   up y * 3/4
   angle 60
   look_at < 0.0, 0.0, 0.0>
}

light_source { // set point light sources location
```

```
   < 0.0, 10, -10>
   rgb <1.000000, 1.000000, 1.000000> * 2.0     // set white light color and intensity
   }

box {<-2, -2, 0>, <2, 2, 0.1>        // set image box area
  pigment {
    mandel 30
    exponent 2
    interior 1, .5
    exterior 1,.8
    color_map {
      [0 rgb 0]
      [0.05 rgb x]
      [0.3 rgb x+y]
      [1 rgb 0]
      [1 rgb 0]
      }
    }
//  rotate <90,-70,-10>  //  alternative rotation  for landscape
//  translate < 0.5,0.7,0> //  alternative translation for lansdscape
translate < 0.5,0.0,0> // front 2D image
}



//=========================================
// Julia set 2D and 3D landscape
// generation (POV-Ray 3.7 Julia function)
//=========================================

 background { color rgb <0.0,0.0,0.0> }       // set black background color

 camera { // set view point (camera) location
   perspective
   location < 1.3, 0, -3>
   // location< 1.0, 2.0, -3>     //alternative camera location for landscape
   right x * 1
   up y * 3/4
   angle 60
   look_at < 0.0, 0.0, 0.0>
}

light_source { // set point light sources location
   < 0.0, 10, -10>
   rgb <1.000000, 1.000000, 1.000000> * 2.0    // set white light color and intensity
}

box {<-2, -2, 0>, <2, 2, 0.1>       // set image box area
  pigment {
    julia < -0.7454294, 0.0 >, 30
    interior 1, 1
    color_map {
      [0 rgb 0]
      [0.07 rgb x]
      [0.5 rgb x+y]
      [1 rgb .6]
      [1 rgb 0]
      }
    }
   rotate < 1.8, -28, 0 >
 // rotate <90,30,0> // alternative rotation for landscape
   translate < .1, 0, 0 >}
```

4.2.2 POV-Ray 3.7 “height field” function rendering

A very performant function implemented in *POV-Ray 3.7* to render landscapes, and in particular fractal scenes as wholes, is provided by the “height field” function. This latter reads preformed $2D$ images and assigns a different elevation to each gray level or color level encountered in the original $2D$ picture. Special rendering effects may be suitably added in the program list in order to obtain beautiful pictures.

Here are some examples of *Mandelbrot* landscapes.

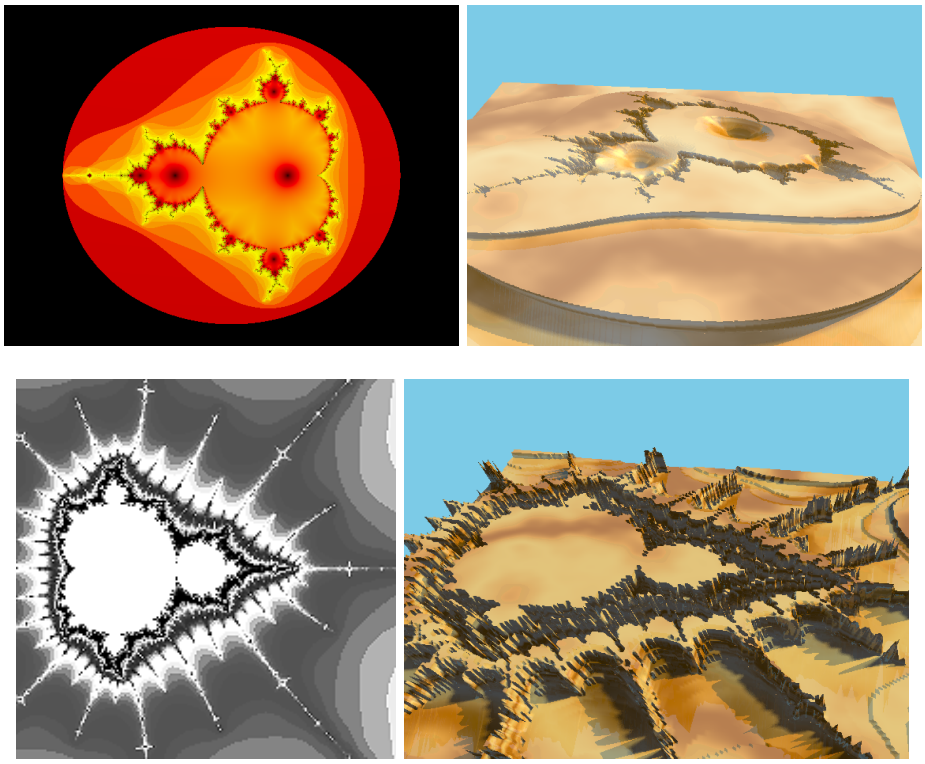


Fig.13 - Mandelbrot landscapes generated as a whole by POV-Ray 3.7 “height field” function starting from the $2D$ color and gray level images on the left

---

POV-Ray 3.7 code to generate fig 13, 14 and 15

```
//=================================================
// Mandelbrot (Julia, Newton's method) landscape
// generation (POV-Ray 3.7 height field function)
//=================================================

#include "colors.inc"   // Standard Color definitions
#include "textures.inc"   // Standard Texture definitions

camera{ // set view point (camera) location
location <2.6, 15.0, -26>
```

```
look_at 0.0
angle 35
}

background { color red 0.2 green 0.6 blue 0.8}    // Set a color of the background (sky)

light_source   // create a regular point light source
{
  0*x // light's position (translated below)
  color Orange // light's color
  translate <1000, 1000, -100>
}
light_source  // Area light source (creates soft shadows)
{
  0*x // light's position (translated below)
  color Bronze  // light's color
  area_light
  <8, 0, 0> <0, 0, 8> // lights spread out across this distance (x * z)
  4, 4               // total number of lights in grid (4x*4z = 16 lights)
  adaptive 0          // 0,1,2,3...
  jitter              // adds random softening of light
  translate <40, 80, -40>   // <x y z> position of light
}

height_field { // heigh field rendering
png "PATH/Mandel2D.png"  // alternative  "PATH/Julia2D.png"   "PATH/Mandel2D.png"
smooth
pigment {agate
  agate_turb 1.0
  }

finish { ambient 0.2 specular 0.5 reflection 0.2 }
translate <-.52, 1.5, -.7>
rotate < -5, 0, 0 >
scale <17, 1.75, 17>
}
```

The figs 14 and 15 are examples of *Julia* and *Newton's method* landscapes obtained by the "height field" function.

For our purposes intended to evidence the constructive process of the structures step by step, the representation of the $3D$ shapes is not enough and a *point by point* method is required, similar to what we have seen implementing *Python 3* programs. We will see in the next two sections how to build fractal landscapes *point by point* following both a *sequentiallt* ordered process and a *random* process. We remember that the intent of all the present exposition of our book, is that of showing that a random procedure – which is genuinely governed by chance – may be able to reach and ordered structure system as an attractor, if it is governed by a suitable law (algorithm).

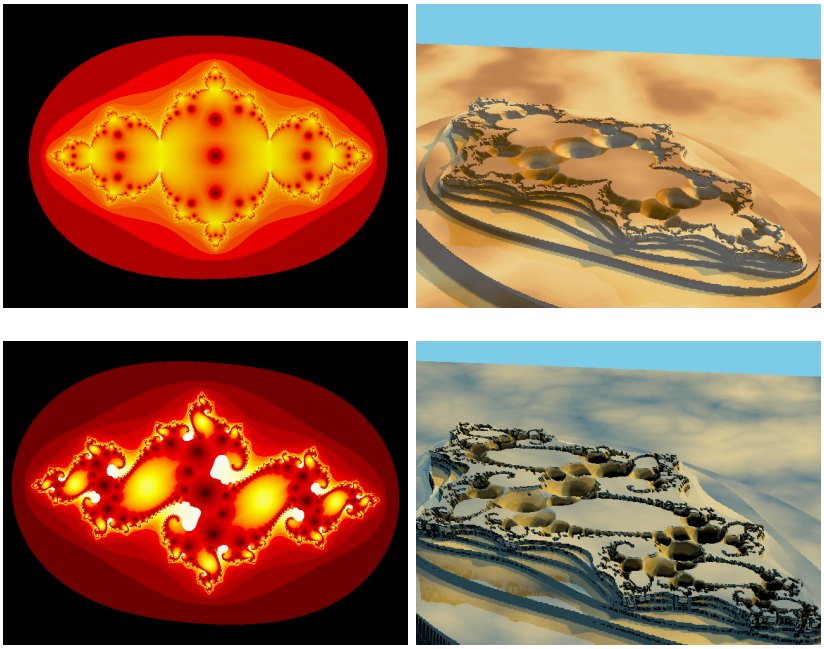


Fig.14 - Julia landscapes generated as a whole by POV-Ray 3.7 "height field" function starting from the $2D$ color level images on the left

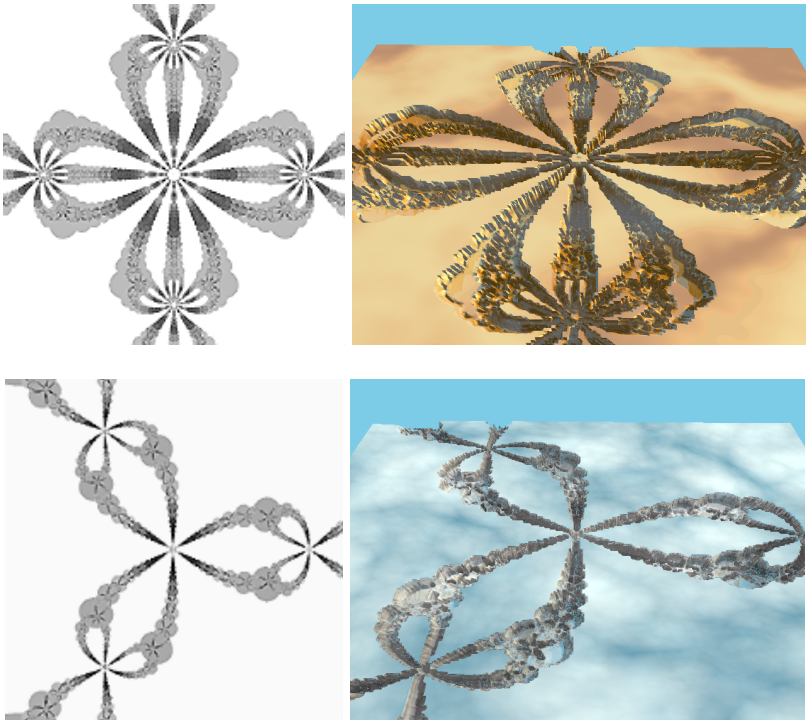


Fig.15 - Newton's method landscapes generated as a whole by POV-Ray 3.7 "height field" function starting from the $2D$ gray level images on the left (*seahorse Julia set*)

4.2.3 Sequential rendering

In the present subsection we present an ordered *sequential procedure,* implemented by *POV-Ray 3.7,* to construct simple fractal landscapes comparable with the examples presented in the previous paragraphs obtained employing different methods.

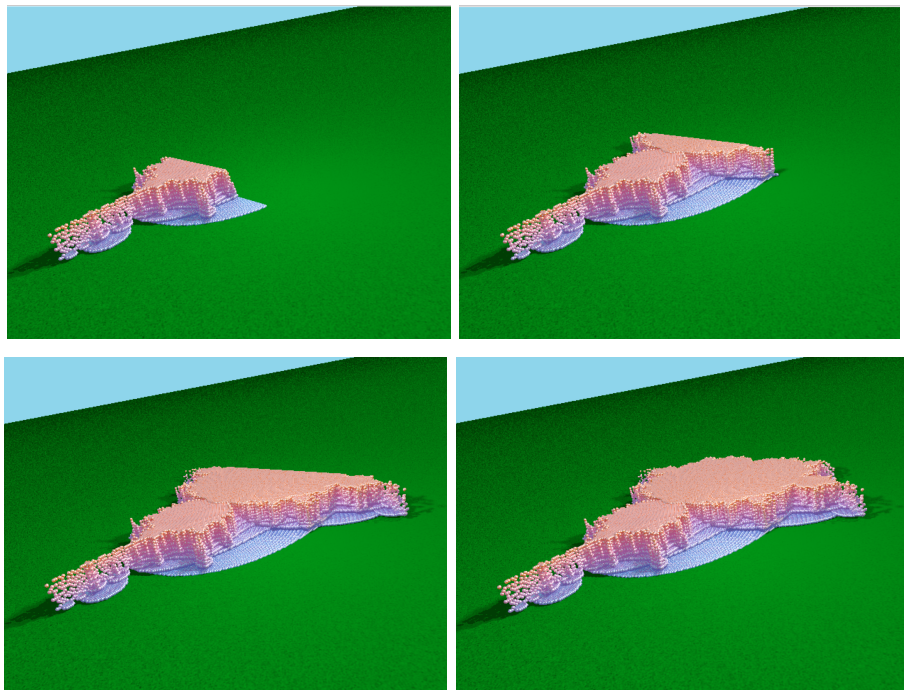


Fig.16 - Mandelbrot mountain landscape generated sequentially by small spheres (POV-Ray 3.7)

VIEW ANIMATION (requires internet connection)

POV-Ray 3.7 code to generate fig 16

```
//=========================================
// Sequential Mandelbrot mountain landscape
// generation (POV-Ray 3.7 small spheres)
//=========================================

#include "colors.inc"   // Standard Color definitions
#include "glass.inc"    // Glass pigments
#include "metals.inc"  // Metal pigments

global_settings {assumed_gamma 1.0}     // set display gamma
#declare Th = -40;    // set rotation angles
#declare Ph = 10;

background { color red 0.2 green 0.6 blue 0.8 }
plane { < 0.0, 1, 0.0 >, 1    //  normal vector to palne | -1 is the height of the floor
```

```
   pigment { color .3*Green }
   normal { bumps 0.5 scale 0.6}  // grass effect
   rotate < 0, Th, Ph >}

 camera { // set view point (camera) location
    location <0, 100, -220>
    look_at  <20, 10,  0>
  }

light_source   // create a regular point light source
{
  0*x // light's position (translated below)
  color Gold // light's color
  translate <1000, 1000, -100>
}

light_source    // Area light source (creates soft shadows)
{
  0*x // light's position (translated below)
  color Silver  // light's color
  // <widthVector> <heightVector> nLightsWide mLightsHigh
  area_light
  <8, 0, 0> <0, 0, 8> // lights spread out across this distance (x * z)
  4, 4                // total number of lights in grid (4x*4z = 16 lights)
  adaptive 0          // 0,1,2,3...
  jitter              // adds random softening of light
  translate <40, 80, -40>   // <x y z> position of light
}

 #declare R = 0.5;      // set radius value
 #declare L = 2;   // set square area side
 #declare X = 0;  // set co-ordinate x initial value
 #declare Y = 0;  // set co-ordinate y initial value
 #declare n = 200;   // set number of pixels per area side
 #declare N = 20;   // set number of cycles

//  replace -n, n by -n*clock, n*clock for animation
#for (p, -n, n, 1)  // partial values 0, n/4, n/2  //
    #for (q, -n, n, 1)
       #declare X = 0;
       #declare Y = 0;
       #for (k,0,N)
       #declare XX = X*X - Y*Y  + p*L/n - 1;
       #declare YY = 2*X*Y + q*L/n;
       #declare X = XX;
       #declare Y = YY;

         #if (X*X+Y*Y < R+.01)
     sphere {
       < p, k, q >, 1
          hollow
      radiosity { importance 1.0 }
          texture {
          pigment { color rgb < abs(sin(.5*pi*k/N)), 0.3, abs(cos(.5*pi*(k/N))) > }
                    }
          finish { ambient rgb <0.3,0.1,0.2>
            diffuse .3
            reflection .3
            specular 1
           }

rotate < 0, Th, Ph >
```

```
translate < 0, 0, 0 >  }

#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
```

A second example, as usual, is offered by a sequence of steps of a process building a *Julia set*.

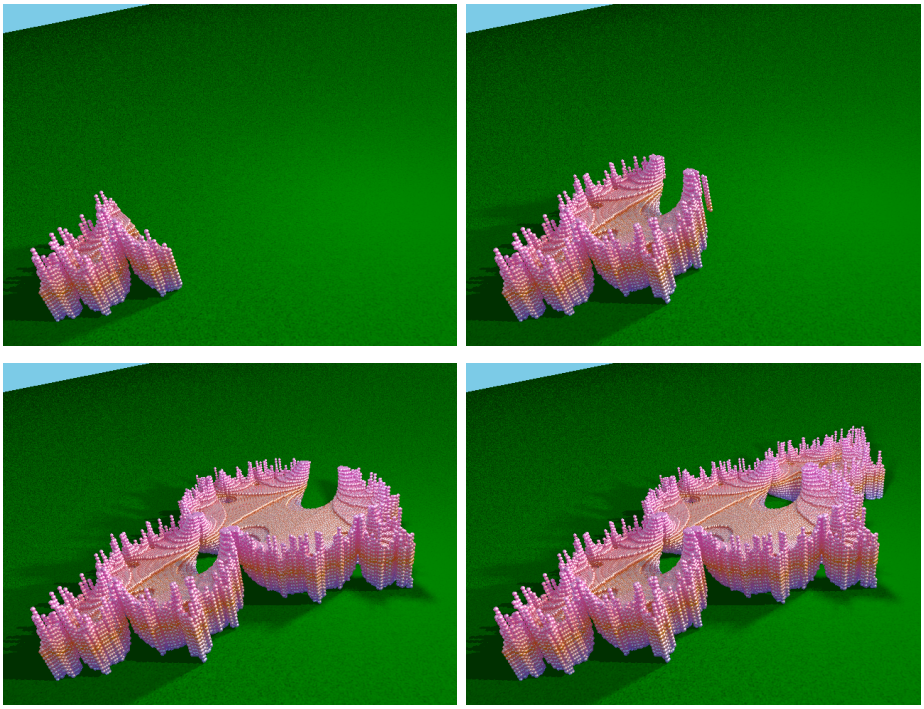


Fig.17 - Julia ($c = -0.745429$) mountain landscape generated sequentially by small spheres (POV-Ray 3.7)

VIEW ANIMATION (requires internet connection)

POV-Ray 3.7 code to generate fig 17 and 18

```
//==========================================
// Sequential Julia mountain landscape
// generation (POV-Ray 3.7 small spheres)
//==========================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"    // Glass pigments
#include "metals.inc"  // Metal pigments
```

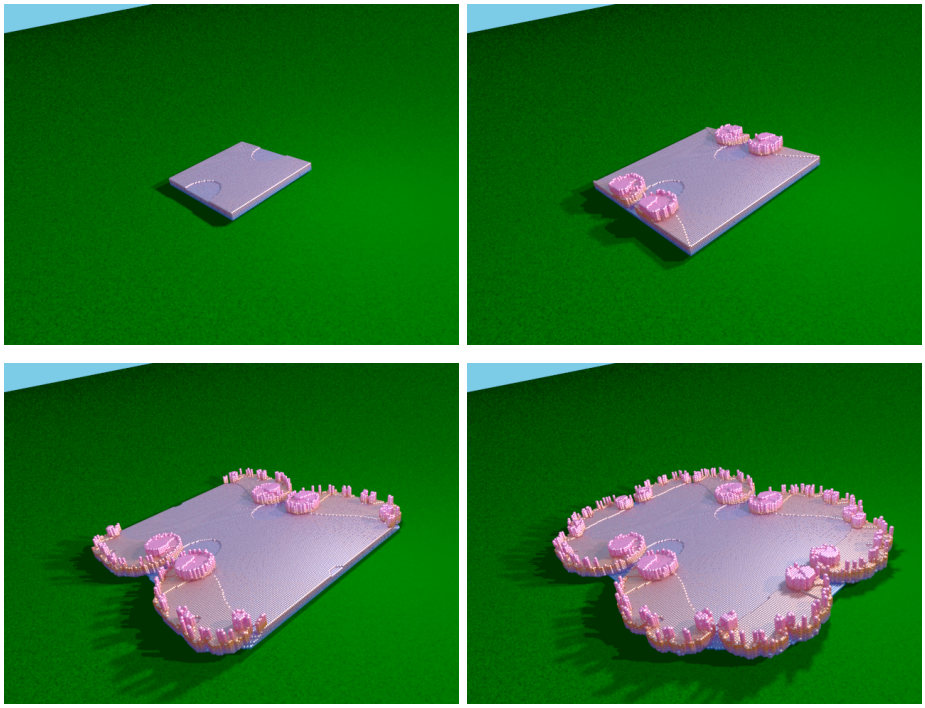


Fig.18 - Julia ($c = 0.27334$) mountain landscape generated sequentially by small spheres (POV-Ray 3.7)

VIEW ANIMATION (requires internet connection)

```
global_settings {assumed_gamma 1.0}
#declare Th = -50;    // set rotation angles
#declare Ph = 10;

background { color red 0.2 green 0.6 blue 0.8 }
plane { < 0.0, 10, 0.0 >, 1    //  normal vector to palne | -1 is the height of the floor
   pigment { color .3*Green }
   normal { bumps 0.5 scale 0.6}  // grass effect
   rotate < 0, Th, Ph >}

 camera {{ // set view point (camera) location
    location <0, 120, -200>
    look_at  <-35, 1,  0>
  }

light_source    // create a regular point light source
{
  0*x // light's position (translated below)
  color Gold // light's color
  translate <1000, 1000, -100>
}

light_source    // Area light source (creates soft shadows)
{
  0*x // light's position (translated below)
  color Silver  // light's color
  // <widthVector> <heightVector> nLightsWide mLightsHigh
```

```
  area_light
  <8, 0, 0> <0, 0, 8> // lights spread out across this distance (x * z)
  4, 4                // total number of lights in grid (4x*4z = 16 lights)
  adaptive 0          // 0,1,2,3...
  jitter              // adds random softening of light
  translate <40, 80, -40>   // <x y z> position of light
}

 #declare R = 0.13;   // set radius value
 #declare L = 2;   // set square area side
 #declare X = 0;  // set co-ordinate x initial value
 #declare Y = 0;  // set co-ordinate y initial value

 // set parameter c values
 #declare Cx = -0.7454294;  // alternative value c = 0.27334
 #declare Cy = 0;
 #declare n = 200;  // set number of pixels per area side
 #declare N = 40;   // set number of cycles

//  replace -n, n by -n*clock, n*clock for animation
#for (p, -n, n, 1)  // partial values -n/4, 0,  n/4
     #declare IncX = p*L/n;
  #for (q, -n, n, 1)
       #declare IncY = q*L/n;
       #declare X = IncX;
       #declare Y = IncY;
       #for (k,0,N)
       #declare XX = X*X - Y*Y  + Cx ;
       #declare YY = 2*X*Y + Cy;
       #declare X = XX;
       #declare Y = YY;

       #if (X*X+Y*Y < R+.002)
     sphere {
       < p, k-3, q >, 1
          hollow
      radiosity { importance 1.0 }
          texture {
          pigment { color rgb < abs(sin(.5*pi*k/N)), 0.3, abs(cos(pi*(k/N))) > }
                    }
          finish { ambient rgb <0.3,0.1,0.2>
             diffuse .3
             reflection .3
             specular 1
           }

rotate < 0, Th, Ph >
translate < 0, 0, 0 >  }

#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
```

A Newton's method example of similar landscape is presented in fig. 19 and the related code is also provided.

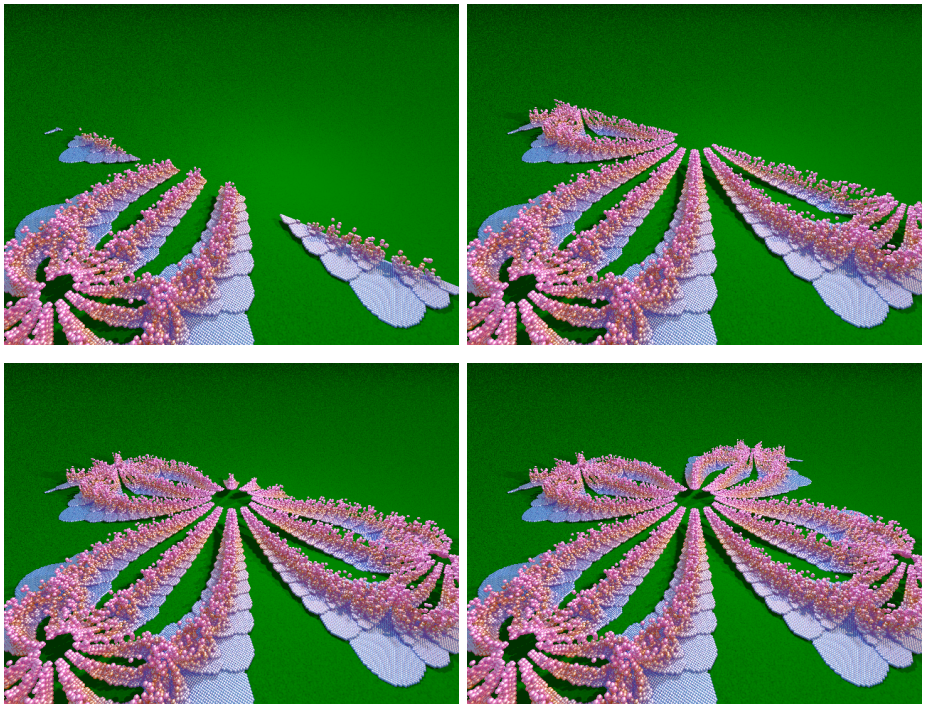


Fig.19 - Newton's method fractal mountain landscape generated sequentially by small spheres (POV-Ray 3.7)

VIEW ANIMATION (requires internet connection)

POV-Ray 3.7 code to generate fig 19

```
//=====================================================
// Sequential Newton's method set mountain landscape
// generation (POV-Ray 3.7 small spheres)
//=====================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigments
#include "metals.inc"  // Metal pigments

global_settings {assumed_gamma 1.0}     // set display gamma
#declare Th =  -60;    // set rotation angles
#declare Ph = 0;

background { color red 0.2 green 0.6 blue 0.8 }
plane { < 0.0, 10, 0.0 >, 1    //  normal vector to palne | -1 is the height of the floor
   pigment { color .3*Green }
   normal { bumps 0.5 scale 0.6}  // grass effect
   rotate < 0, Th, Ph >}

 camera {{ // set view point (camera) location
    location < 0, 120, -180>
    look_at  < 0, -20, 0>
    angle 90}
```

```
light_source   // create a regular point light source
{
  0*x // light's position (translated below)
  color Gold // light's color
  translate <1000, 1000, -100>
}

light_source    // Area light source (creates soft shadows)
{
  0*x // light's position (translated below)
  color Silver  // light's color
  // <widthVector> <heightVector> nLightsWide mLightsHigh
  area_light
  <8, 0, 0> <0, 0, 8> // lights spread out across this distance (x * z)
  4, 4                // total number of lights in grid (4x*4z = 16 lights)
  adaptive 0          // 0,1,2,3...
  jitter              // adds random softening of light
  translate <40, 80, -40>   // <x y z> position of light
}

 #declare R = 0.9;   // set radius value
 #declare L = 1.0;   // set square area side
 #declare X = 0.01;  // set co-ordinate x initial value
 #declare Y = 0.01;  // set co-ordinate y initial value
 #declare Cx = 0.0;   // set x shift
 #declare Cy = 0.0;   // set y shtift
 #declare n = 200;  // set number of pixels per area side
 #declare N = 20;   // set number of cycles

//  replace -n, n by -n*clock, n*clock for animation
#for (p, -n, n/4, 1)  // partial values -n/4, 0,  n/4
     #declare IncX = p*L/n;
   #for (q, -n, n, 1)
       #declare IncY = q*L/n;
       #declare X = IncX;
       #declare Y = IncY;
       #for (k,0,N)
       #declare XX = 3*X/4 - X*(X*X - 3*Y*Y)/((X*X + Y*Y)*(X*X + Y*Y)*
       (X*X + Y*Y)*4);
       #declare YY = 3*Y/4 - Y*(Y*Y - 3*X*X)/((X*X + Y*Y)*(X*X + Y*Y)*
       (X*X + Y*Y)*4);
       #declare X = XX;
       #declare Y = YY;

       #if ((X - Cx)*(X-Cx) + (Y - Cy)*(Y - Cy) < R)
     sphere {
       < p, k, q >, 1
          hollow
      radiosity { importance 1.0 }
          texture {
          pigment { color rgb < abs(sin(.5*pi*k/N)), 0.3,
          abs(cos(pi*(k/N))) > }
                      }
          finish { ambient rgb <0.3,0.1,0.2>
            diffuse .3
            reflection .3
            specular 1
           }

rotate < 0, Th, Ph >
translate < 0, 0, 0 >  }
```

```
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
```

4.2.4 Random rendering

In this subsection the same landscapes are obtained starting from *random initial conditions,* applying the same *algorithm* (*law*) to each of them. Note how order arises step after step.

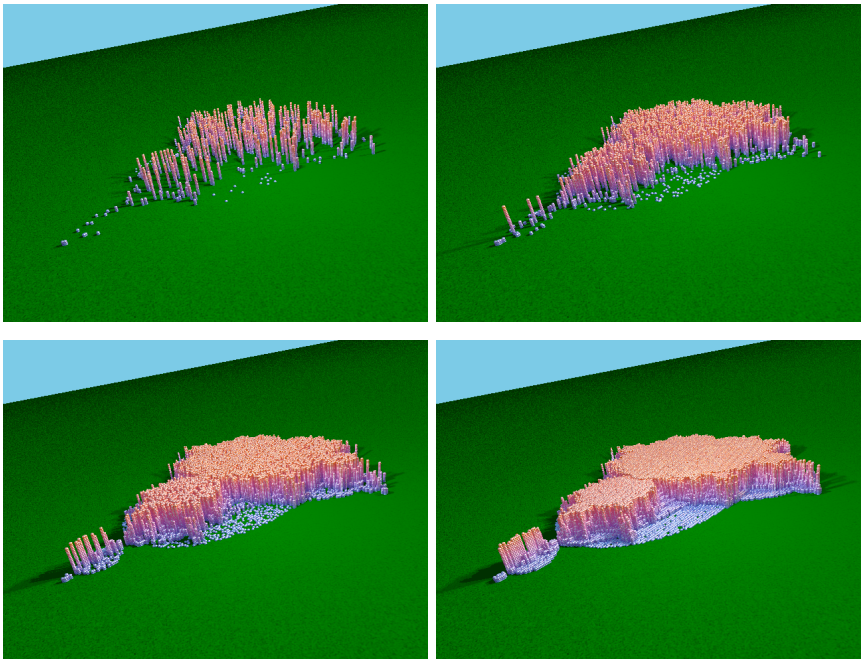


Fig.20 - Mandelbrot mountain landscape built randomly by small spheres (POV-Ray 3.7)

VIEW ANIMATION (requires internet connection)

In fig. 20 a picture of a random Mandelbrot landscape is shown. One can recognize how the shape of the resulting structure refines as the number of points increases.

POV-Ray 3.7 code to generate fig 20

```
//========================================
// Random Mandelbrot mountain landscape
// generation (POV-Ray 3.7 small spheres)
//========================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigments
```

```
#include "metals.inc"  // Metal pigments

global_settings {assumed_gamma 1.0}   // set display gamma
#declare Th = -40;    // set rotation angles
#declare Ph = 10;

background { color red 0.2 green 0.6 blue 0.8 }
plane { < 0.0, 1, 0.0 >, 1    //  normal vector to palne | -1 is the height of the floor
   pigment { color .3*Green }
   normal { bumps 0.5 scale 0.6}  // grass effect
   rotate < 0, Th, Ph >
    }

 camera {{ // set view point (camera) location
    location <0, 100, -220>
    look_at  <20, 10,  0>
  }

light_source   // create a regular point light source
{
  0*x // light's position (translated below)
  color Gold // light's color
  translate <1000, 1000, -100>
}

light_source    // Area light source (creates soft shadows)
{
  0*x // light's position (translated below)
  color Silver  // light's color
  // <widthVector> <heightVector> nLightsWide mLightsHigh
  area_light
  <8, 0, 0> <0, 0, 8> // lights spread out across this distance (x * z)
  4, 4                // total number of lights in grid (4x*4z = 16 lights)
  adaptive 0          // 0,1,2,3...
  jitter              // adds random softening of light
  translate <40, 80, -40>   // <x y z> position of light
}

#declare R = 0.5;      // set radius value
#declare L = 2;   // set square area side

#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;  // set co-ordinate y initial value

#declare n = 400;   // alternative value n = 200    // set number of pixels per area side
#declare N = 20;  // set number of cycles
#declare K = array[n+1][n+1];    // image point matrix

#for (p, 0, n, 1)  // partial values 0, n/4, n/2
   #for (q, 0, n, 1)
       #declare X = 0;
       #declare Y = 0;
       #declare K[p][q] = 0;

       #for (k,0,N)
       #declare XX = X*X - Y*Y  + 2*p*L/n - L - 1;
       #declare YY = 2*X*Y + 2*q*L/n - L;
       #declare X = XX;
       #declare Y = YY;

      #if (X*X+Y*Y < R+.01)
      #declare K[p][q] = k;
```

```
      #end  // end if
      #end  // end for k
      #end  // end for q
      #end  // end for p

#declare Rnd_1 = seed (1153);     // random number generators
#declare Rnd_2 = seed (553) ;

//  replace n*n by n*n*clock for animation
#for(j,0,n*n)    // partial values n*n/64, n*n/16, n*n/4
#declare p = int(n*rand(Rnd_1));
#declare q = int(n*rand(Rnd_2));

#for(i,0,K[p][q])
     sphere {
       < p-n/2, i, q-n/2 >, 1
          hollow
      radiosity { importance 1.0 }
          texture {
            pigment { color rgb < abs(sin(.5*pi*i/N)), 0.3,
        abs(cos(.5*pi*i/N)) > }
                    }
            finish { ambient rgb <0.3,0.1,0.2>
               diffuse .3
              reflection .3
              specular 1
             }

rotate < 0, Th, Ph >
translate < 0, 0, 0 >  }

#end  // end for i
#end  // end for j
```

While fig. 21 shows a picture of a random *Julia* landscape.
Here is the related *POV-Ray 3.7* code.

POV-Ray 3.7 code to generate fig 21

```
//=========================================
// Random Julia mountain landscape
// generation (POV-Ray 3.7 small spheres)
//=========================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigments
#include "metals.inc"  // Metal pigments

global_settings {assumed_gamma 1.0}   // set display gamma

#declare Th = -50;    // set rotation angles
#declare Ph = 10;

// Set a color of the background (sky)
background { color red 0.2 green 0.6 blue 0.8 }
```

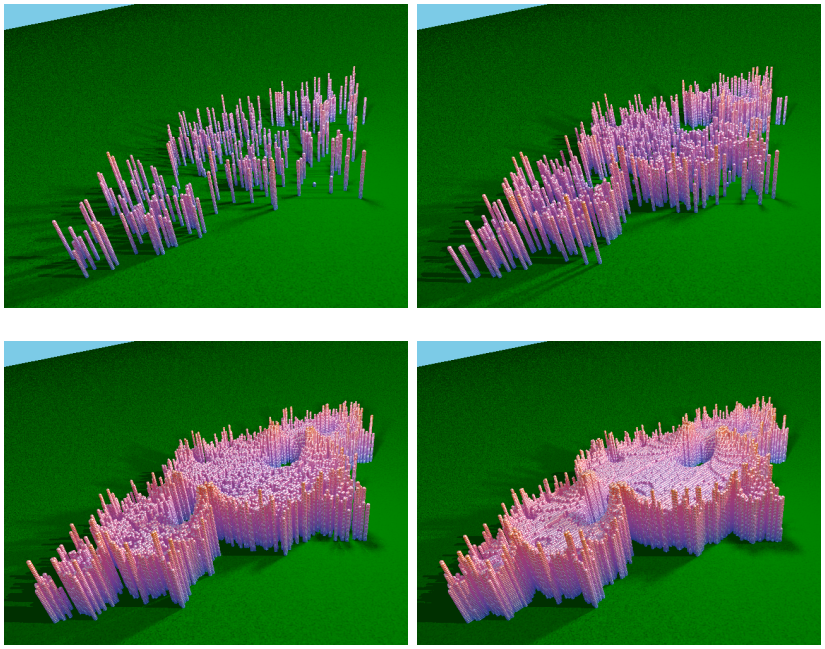


Fig.21 - Julia mountain landscape generated radndomly by small spheres (POV-Ray 3.7)

VIEW ANIMATION (requires internet connection)

```
plane { < 0.0, 1, 0.0 >, 1    //  normal vector to palne | -1 is the height of the floor
   pigment { color .3*Green }
   normal { bumps 0.5 scale 0.6}  // grass effect
   rotate < 0, Th, Ph >}


 camera {{ // set view point (camera) location
    location <0, 120, -200>
    look_at  <-35, 1,  0>
  }

light_source   // create a regular point light source
{
  0*x // light's position (translated below)
  color Gold // light's color
  translate <1000, 1000, -100>
}

light_source    // Area light source (creates soft shadows)
{
  0*x // light's position (translated below)
  color Silver  // light's color
  // <widthVector> <heightVector> nLightsWide mLightsHigh
  area_light
  <8, 0, 0> <0, 0, 8> // lights spread out across this distance (x * z)
  4, 4                // total number of lights in grid (4x*4z = 16 lights)
  adaptive 0          // 0,1,2,3...
  jitter              // adds random softening of light
```

```
  translate <40, 80, -40>   // <x y z> position of light
}

#declare R = 0.13;      // set radius value
#declare L = 2;  // set square area side
#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;  // set co-ordinate y initial value

#declare Cx = -0.7454294;   // set parameter c values
#declare Cy = 0;

#declare n = 400;  // alternative value n = 300    // set number of pixels per area side
#declare N = 40;   // set number of cycles

#declare K = array[n+1][n+1];   // image point matrix

//  replace n by n*clock for animation
#for (p, 0, n, 1)  // partial values 0, n/4, n/2
   #declare IncX = -L + 2*p*L/n;
   #for (q, 0, n, 1)
       #declare IncY = -L + 2*q*L/n;
       #declare X = IncX;
       #declare Y = IncY;
       #declare K[p][q] = 0;

    #for (k,0,N)
       #declare XX = X*X - Y*Y + Cx;
       #declare YY = 2*X*Y + Cy;
       #declare X = XX;
       #declare Y = YY;

    #if (X*X+Y*Y < R+.002)
       #declare K[p][q] = k;

#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p

#declare Rnd_1 = seed (1153);    // random numbers generator
#declare Rnd_2 = seed (553);

//  replace n*n by n*n*clock for animation
#for(j,0,n*n)    // partial values n*n/64, n*n/16, n*n/4
#declare p = int(n*rand(Rnd_1));
#declare q = int(n*rand(Rnd_2));

 #for(i,0,K[p][q])
     sphere {
       < p-n/2, i, q-n/2 >, 1
          hollow
      radiosity { importance 1.0 }
          texture {
          pigment { color rgb < abs(sin(.5*pi*i/N)), 0.3, abs(cos(.5*pi*i/N)) > }
                    }
          finish { ambient rgb <0.3,0.1,0.2>
            diffuse .3
            reflection .3
            specular 1
           }

rotate < 0, Th, Ph >
```

```
translate < 0, 0, 0 >  }

#end  // end for i
#end  // end for j
```

And four steps of the Newton's method landscape random process are illustrated inf fig. 22.

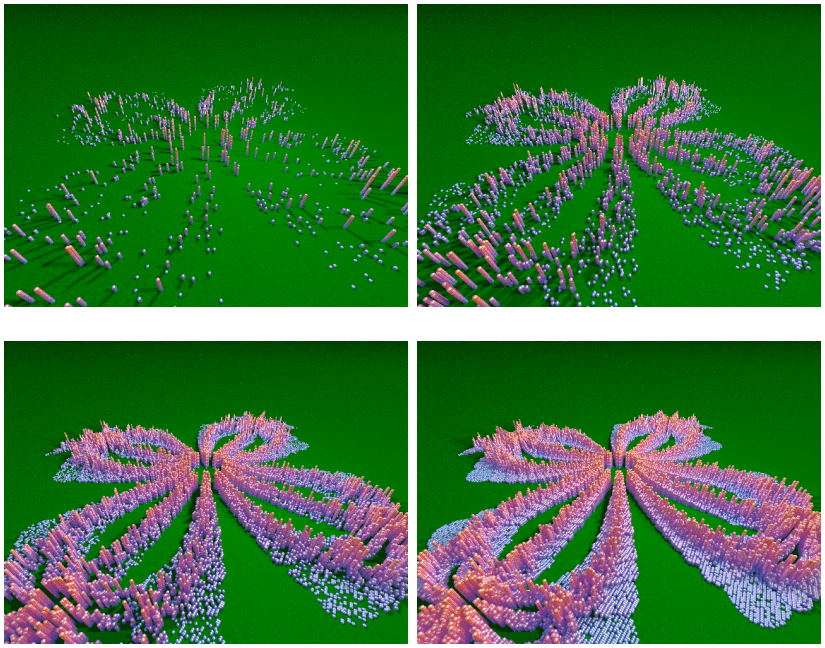


Fig.22 - Newton's method mountain landscape generated randomly by small spheres (POV-Ray 3.7)

VIEW ANIMATION (requires internet connection)

POV-Ray 3.7 code to generate fig 22

```
//================================================
// Random Newton's method mountain landscape
// generation (POV-Ray 3.7 small spheres)
//================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigments
#include "metals.inc"  // Metal pigments

global_settings {assumed_gamma 1.0}   // set display gamma
```

```
#declare Th =  -60;    // set rotation angles
#declare Ph = 0;

// Set a color of the background (sky)
background { color red 0.2 green 0.6 blue 0.8 }
plane { < 0.0, 10, 0.0 >, 1    //  normal vector to palne | -1 is the height of the floor
   pigment { color .3*Green }
   normal { bumps 0.5 scale 0.6}  // grass effect
   rotate < 0, Th, Ph >}


 camera {{ // set view point (camera) location
    location < 0, 120, -180>
    look_at  < 0, -20, 0>
    angle 90}

light_source   // create a regular point light source
{
  0*x // light's position (translated below)
  color Gold // light's color
  translate <1000, 1000, -100>
}

 light_source    // Area light source (creates soft shadows)
{
  0*x // light's position (translated below)
  color Silver  // light's color
  // <widthVector> <heightVector> nLightsWide mLightsHigh
  area_light
  <8, 0, 0> <0, 0, 8> // lights spread out across this distance (x * z)
  4, 4                // total number of lights in grid (4x*4z = 16 lights)
  adaptive 0          // 0,1,2,3...
  jitter              // adds random softening of light
  translate <40, 80, -40>   // <x y z> position of light
}

#declare R = 0.9;      // set radius value
#declare L = 1.0;   // set square area side
#declare X = 0.01;  // set co-ordinate x initial value
#declare Y = 0.01;  // set co-ordinate y initial value
#declare Cx = 0.0;   // set x shift
#declare Cy = 0.0;   // set y shift

#declare n = 400;  // alternative value  n = 300      // set number of pixels per area side
#declare N = 20;   // set number of cycles

#declare K = array[n+1][n+1];   // image point matrix

#for (p, 0, n, 1)  // partial values -n/4, 0,  n/4
    #declare IncX = -L + 2*p*L/n;
    #for (q, 0, n, 1)
       #declare IncY = -L + 2*q*L/n;
       #declare X = IncX;
       #declare Y = IncY;
       #declare K[p][q] = 0;

   #for (k,0,N)
     #declare XX = 3*X/4 - X*(X*X - 3*Y*Y)/((X*X + Y*Y)*(X*X + Y*Y)*
       (X*X + Y*Y)*4);
    #declare YY = 3*Y/4 - Y*(Y*Y - 3*X*X)/((X*X + Y*Y)*(X*X + Y*Y)*
    (X*X + Y*Y)*4);
       #declare X = XX;
```

```
       #declare Y = YY;

        #if ((X - Cx)*(X-Cx) + (Y - Cy)*(Y - Cy) < R)
           #declare K[p][q] = k;
       #end  // end if

#end  // end for k
#end  // end for q
#end  // end for p

#declare Rnd_1 = seed (1153);    // random numbers generator
#declare Rnd_2 = seed (553);

//  replace n*n by n*n*clock for animation
#for(j,0,n*n)    // partial values n*n/64, n*n/16, n*n/4
  #declare p = int(n*rand(Rnd_1));
  #declare q = int(n*rand(Rnd_2));

 #for(i,0,K[p][q])
     sphere {
       < p-n/2, i, q-n/2 >, 1
          hollow
      radiosity { importance 1.0 }
          texture {
          pigment { color rgb < abs(sin(.5*pi*i/N)), 0.3,
          abs(cos(.5*pi*(i/N))) > }
                     }
          finish { ambient rgb <0.3,0.1,0.2>
            diffuse .3
            reflection .3
            specular 1
           }

rotate < 0, Th, Ph >
translate < 0, 0, 0 >  }

#end  // end for i
#end  // end for j
```

Chapter 5

Three-dimensional fractals with cylindrical symmetry

Rendering structures either as wholes or by sequential or by random processes

Landscape rendering, we saw in the previous chapter, is not the only way to extend $2D$ fractals to a $3D$ space. In fact a proper $3D$ fractal body should be generated in such a way that its boundary is a fractal surface and not only a curve on a plane with segments adjoined along the third dimension of space. But if one attempts to extend to $3D$ a complex fractal as a Mandelbrot, or a Julia or Newton's method set one encounters soon at least three serious problems.

1. A first, not irrelevant, problem is related to the computation time which increases now according to power 3 and no longer to power 2. So calculations required to build some plots may take even more than 24 hours to be developed.

2. A second problem arises because complex numbers possess only two components (*i.e.,* a *real* and an *imaginary* part), while in three space dimension we need three components (co-ordinates) in order to plot a point.

3. A third problem, which involves both complex and real fractals, arises because, in three dimensions, the level curves (painted in a same color) become fractal level surfaces, one including and hiding the others. So the fractal structure may be hidden by external non-fractalized surfaces.

The first problem is successfully solved when a $2D$ fractal, like *e.g.,* Mandelbrot and some of the Julia sets exhibit a symmetry axis around which it can be revolved to generate a body characterized by cylindrical symmetry. It must be emphasized that, in this case, the degree of complexity of the fractal structure is not increased respect to the two dimensional fractal, since no further control parameter is introduced accompanying the third dimension.

The second problem, when no symmetry axis exists for the $2D$ fractal, is usually solved recurring to *quaternions,* which generalize complex numbers to more dimensions. Since the simplest kind of *quaternions* exhibit four components, only three of them may be used to represent a point within the $3D$ space. So the fourth component of quaternions can be set to a fixed value, as *e.g.,* zero. Now the introduction of a new parameter accompanying the third space dimension actually involves a new degree of freedom which increases the degree of complexity of the fractal.

The third problem may be solved imposing suitable maximum and minimum threshold values to the recursive function characterizing the fractal, so fixing a visible fractal boundary shell to the body and avoiding to plot external points.

In the present chapter we will examine $3D$ fractals with *cylindrical symmetry,* obtained revolving $2D$ *Mandelbrot, Julia* and *Newton's method* sets, introducing also threshold values to improve the calculation speed. While in the next chapter we will examine some generalized *Mandelbrot, Julia* sets generated employing *quaternions* and *hypercomplex numbers.*

We will use, in both chapters, only *POV-Ray 3.7* for better rendering effects, constructing the images *point by point* in order to be able to control the constructive process at each step.

5.1 *POV-Ray 3.7* $3D$ fractal structures with cylindrical symmetry as wholes

As a first step we need to start building two dimensional fractals endowed with a symmetry axis. And as a second step we will revolve each $2D$ set around its symmetry axis in order to obtain a $3D$ fractal body characterized by a cylindrical symmetry.

5.1.1 Generalized Mandelbrot set with cylindrical symmetry

The $2D$ starting Mandelbrot set

The ordinary $2D$ Mandelbrot set which provides the basis for the process yielding to the desired $3D$ plot could be, *e.g.,* the one represented in the next fig. 1.

We point out that in order to build $3D$ bodies it is not suitable to involve color map very rich of different colors, since now we are not intended to mark any *escape velocity.* Rather we are interested to metal or glass material rendering better effects.

POV-Ray 3.7 code to generate fig. 1

```
//================================
// Mandelbrot 2D set as a whole
// (POV-Ray point by point plot)
//================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <0.0,0.0,0.0> }   // set black background
```

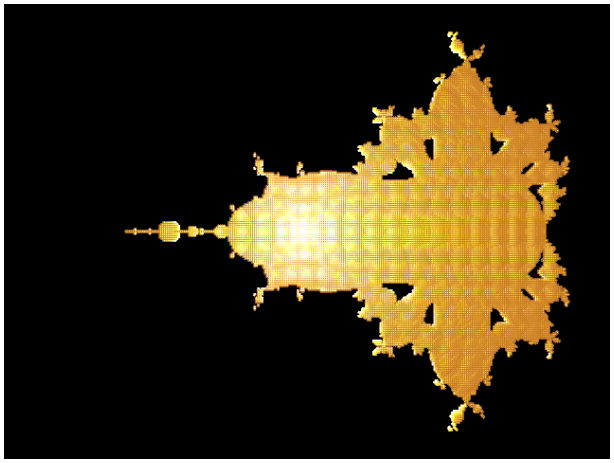


Fig.1 - POV 3.7 two dimensional Mandelbrot set as a whole

```
camera { // set view point (camera) location
    location <0, 0, -200>
    look_at  <1, 0,  0>
    angle 80
}

light_source { // set point light sources location
< -2.0, 0, -5>
rgb <1.000000, 1.000000, 1.000000> * 10.0     // set white light color and intensity
}

#declare R = .1;     // set radius value
#declare L = 2;  // set square area side
#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;   // set co-ordinate y initial value
#declare n = 200;    // set number of pixels per area side
#declare N = 50;   // set number of cycles

  #for (p, -n, n, 1)
  #for (q, -n, n, 1)
       #declare X = 0;
       #declare Y = 0;
       #for (k,0,N)
       #declare XX = X*X - Y*Y  + p*L/n - 1;
       #declare YY = 2*X*Y + q*L/n;
       #declare X = XX;
       #declare Y = YY;

   #if (X*X+Y*Y < R+.01)
   sphere {
     < p, q, 0 >, 1
      hollow
radiosity { importance 1.0 }
     texture {
       pigment { color Col_Glass_Yellow
     }
               }
```

```
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .1
       reflection .1
       specular 1
      }
    }

#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
```

Moreover, for faster rendering, it will be suitable to impose both a maximum and a minimum value threshold to the recursive function $X^2 + Y^2$ in the previous program.

Fig.2 - POV 3.7 two dimensional carved Mandelbrot set as a whole

The minimum threshold adds also the beautiful effect of carving the interior structure of the Mandelbrot set which will be especially relevant in the $3D$ rendering of open sections of the fractal bodies.

POV-Ray 3.7 code to generate fig. 2

```
//====================================
// Mandelbrot 2D carved set as a whole
// (POV-Ray point by point plot)
//====================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <0.0,0.0,0.0> }   // set black background

camera { // set view point (camera) location
```

```
    location <0, 0, -200>
    look_at  <1, 0,  0>
    angle 80
  }

 light_source { // set point light sources location
< -2.0, 0, -5>
rgb <1.000000, 1.000000, 1.000000> * 10.0     // set white light color and intensity
 }

#declare R = .1;     // set radius value
#declare L = 2;  // set square area side
#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;   // set co-ordinate y initial value
#declare n = 200;    // set number of pixels per area side
#declare N = 50;  // alternative   N = 100;     // set number of cycles

  #for (p, -n, n, 1)
  #for (q, -n, n, 1)
       #declare X = 0;
       #declare Y = 0;
       #for (k,0,N)
       #declare XX = X*X - Y*Y  + p*L/n - 1;
       #declare YY = 2*X*Y + q*L/n;
       #declare X = XX;
       #declare Y = YY;

   #if (X*X+Y*Y > R)
        #if (X*X+Y*Y < R+.01)
   sphere {
     < p, q, 0 >, 1
      hollow
radiosity { importance 1.0 }
     texture {
       pigment { color Col_Glass_Yellow
     }
              }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .1
       reflection .1
       specular 1
      }
    }

#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
```

The $3D$ generalized Mandelbrot set

Revolving the carved $2D$ set one may obtain, finally, a $3D$ body endowed of cylindrical symmetry (see fig. 3).

Manifestly adding a third dimension sensibly hides the fractalization of the boundary surfaces respect to what appears in two dimensions. The result will be of more impact if the rotation of the $2D$ figure is not continuous, but discrete, so that some insight into the interior

of the fractal body becomes more evident (see fig. 4), or if one observes an animation of the continuous revolution process.

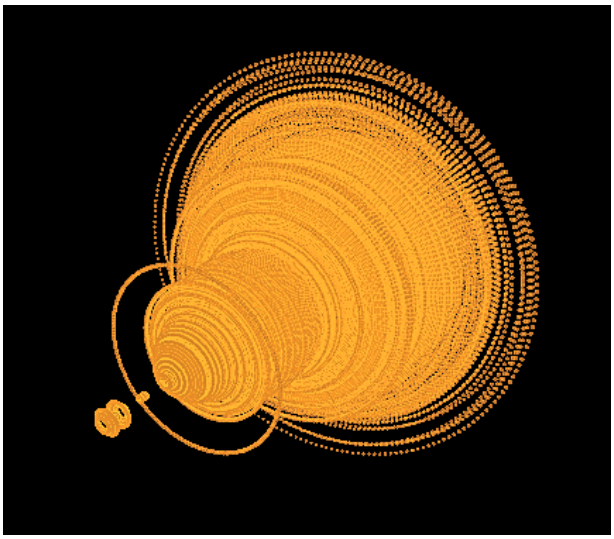


Fig.3 - POV 3.7 3*D* Mandelbrot set with cylindrical symmetry

VIEW ANIMATION (requires internet connection)

POV-Ray 3.7 code to generate fig. 3 and 4

```
//===================================
// Mandelbrot 3D set as a whole
// (POV-Ray cylindrical symmetry)
//===================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect

global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }    // set black background

camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

 light_source { // set point light sources location
< -3.0, 10, -5>
rgb <1.000000, 1.000000, 1.000000> * 2.0     // set white light color and intensity
 }

 light_source {
< 5.0, 10, -10>
```

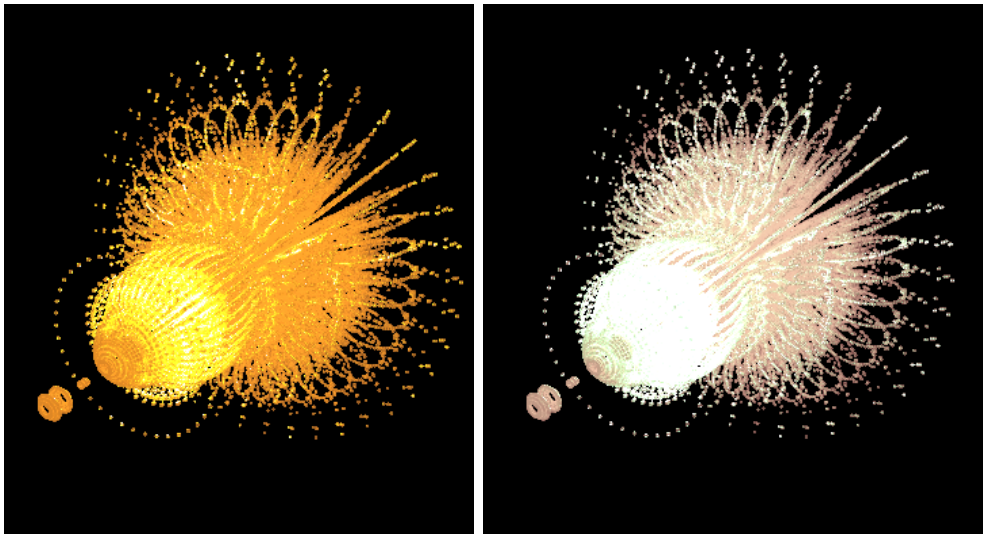


Fig.4 - POV 3.7 3*D* Mandelbrot set with cylindrical symmetry
(two different glass renderings)

```
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R = .1;     // set radius value
#declare L = 2;  // set square area side
#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;  // set co-ordinate y initial value
#declare Z = 0;  // set co-ordinate z initial value

#declare n = 200;    // set number of pixels per area side
#declare N = 10;   // set number of cycles
#declare Nr = 100;  // alternative   Nr = 20;   // set number of sections
#declare Th = -45;
#declare Ph = 30;

union{ #for (p, -n, n, 1)
    #for (q, -n, n, 1)
       #declare X = 0;
       #declare Y = 0;
       #for (i,1,N)
       #declare XX = X*X - Y*Y  + p*L/n - 1;
       #declare YY = 2*X*Y + q*L/n;
       #declare X = XX;
       #declare Y = YY;
       #if (X*X+Y*Y > R)
        #if (X*X+Y*Y < R+.01)
        #for (k,0,Nr)  //  replace Nr by Nr*clock for animation
    sphere {
     < p, q*cos(3.14*k/Nr), q*sin(3.14*k/Nr) >, 1
     texture {
       pigment { color Col_Glass_Yellow }
// alternative pigment { color Col_Glass_Old }
        }
```

```
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for i
#end  // end for q
#end  // end for p
rotate < 0, Th, Ph >}
```

An open view of the same $3D$ Mandelbrot set allows to examine also its internal structure in detail. In fig. 5 some interesting glass color rendering pictures are presented.

POV-Ray 3.7 code to generate fig. 5

```
//===============================================
// Mandelbrot 3D set as a whole
// (POV-Ray cylindrical symmetry open section)
//===============================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect

global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }    // set black background

camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
}

light_source { // set point light sources location
< -3.0, 10, -5>
rgb <1.000000, 1.000000, 1.000000> * 2.0     // set white light color and intensity
}

light_source {
< 5.0, 10, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

#declare R = .1;     // set radius value
#declare L = 2;  // set square area side
#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;  // set co-ordinate y initial value
#declare Z = 0;  // set co-ordinate z initial value

#declare n = 200;    // set number of pixels per area side
#declare N = 10;   // set number of cycles
#declare Nr = 40;   // set number of sections
#declare Th = -45;
#declare Ph = 30;
```

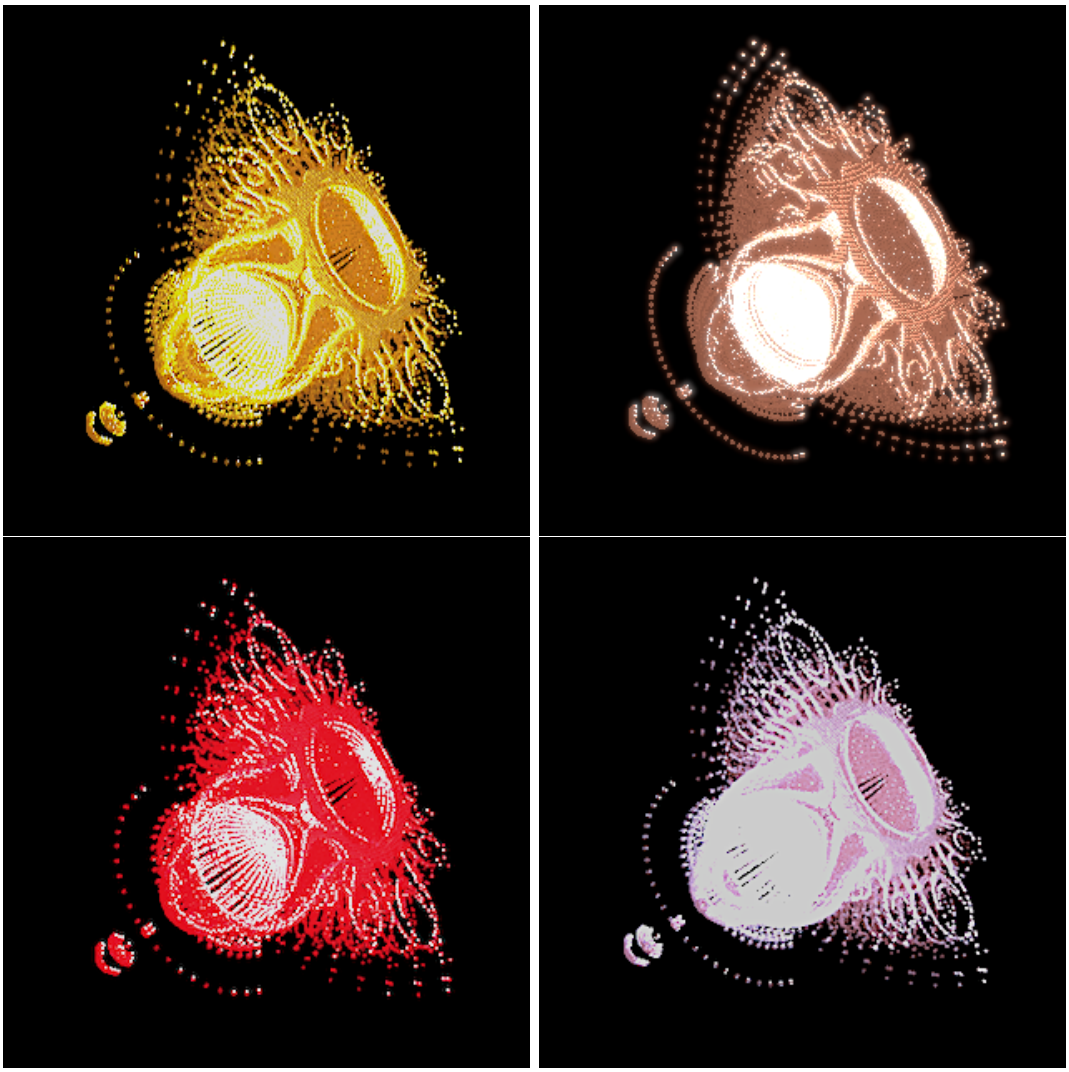


Fig.5 - POV 3.7 3D Mandelbrot set with cylindrical symmetry open section
(four different glass renderings)

```
union{ #for (p, -n, n, 1)
    #for (q, -n, n, 1)
       #declare X = 0;
       #declare Y = 0;
       #for (i,1,N)
       #declare XX = X*X - Y*Y  + p*L/n - 1;
       #declare YY = 2*X*Y + q*L/n;
       #declare X = XX;
       #declare Y = YY;
       #if (X*X+Y*Y > R)
       #if (X*X+Y*Y < R+.01)
       #for (k,1,Nr)
    sphere {
     < p, abs(q)*cos(3.14*k/Nr), abs(q)*sin(3.14*k/Nr) >, 1
     texture {
       pigment { color P_Copper4 }
        }
        finish { ambient rgb <0.1,0.1,0.1>
       diffuse .3
       reflection .3
       specular 1 }
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for i
#end  // end for q
#end  // end for p
rotate < 0, Th, Ph >}
```

5.1.2 Generalized Julia set with cylindrical symmetry

The $2D$ starting Julia set

The same constructive procedure can be profitably applied to the generation of a $3D$ generalized *Julia set* with cylindrical symmetry. As before we start from a $2D$ *Julia set*.

POV-Ray 3.7 code to generate fig. 6

```
//===========================================
// Julia 2D set as a whole (full and carved)
// (POV-Ray point by point plot)
//===========================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect

global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <0.0,0.0,0.0> }   // set black background

camera { // set view point (camera) location
    location <0, -50, -300>
    look_at  <-5, 0,  0>
```

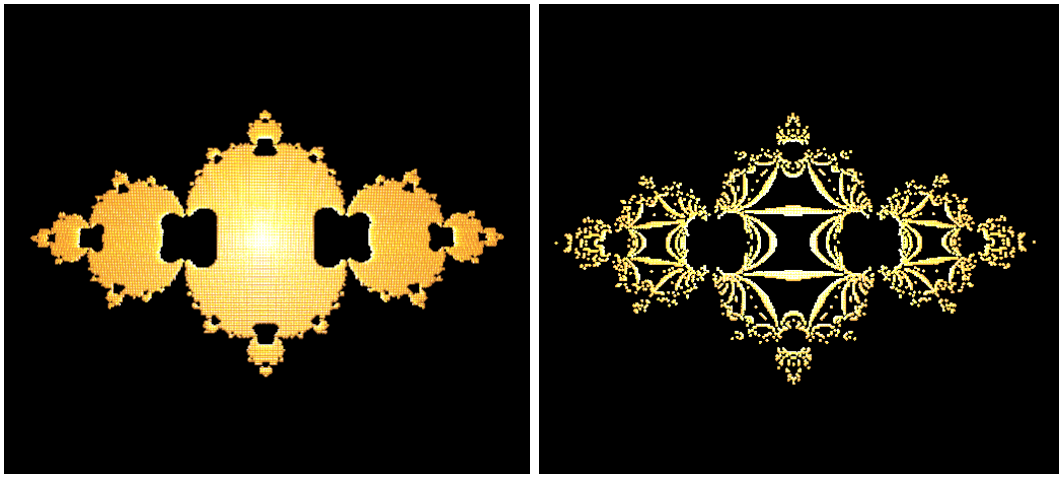


Fig.6 - POV 3.7 two dimensional Julia set ($c = -0.7454294$) as a whole (full and carved)

```
}

light_source { // set point light sources location
< -2.0, -1, -5>
rgb <1.000000, 1.000000, 1.000000> * 10.0     // set white light color and intensity
}

#declare R = .1;     // set radius value
#declare L = 2;  // set square area side
#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;  // set co-ordinate y initial value

#declare n = 200;    // set number of pixels per area side
#declare N = 30;   // set number of cycles

#declare Cx  = -0.7454294;    // set parameter c values
#declare Cy = 0;

#for (p, -n, n, 1)
    #declare IncX = p*L/n;
    #for (q, -n, n, 1)
    #declare IncY = q*L/n;
    #declare X = IncX;
    #declare Y = IncY;
       #for (k,0,N)
       #declare XX = X*X - Y*Y  +  Cx;
       #declare YY = 2*X*Y + Cy;
       #declare X = XX;
       #declare Y = YY;
       #if (X*X+Y*Y > R)   // comment for full set
       #if (X*X+Y*Y < R+.002)
   sphere {
     < p, q, 0 >, 1
      hollow
radiosity { importance 1.0 }
     texture {
       pigment { color Col_Glass_Yellow
```

```
    }
            }
       finish { ambient rgb <0.3,0.1,0.1>
      diffuse .1
      reflection .1
      specular 1
     }
   }

#end  // end if  // comment for full set
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
```

The $3D$ generalized Julia set

Revolving the carved $2D$ *Julia set* we generate a $3D$ set with cylindrical symmetry.

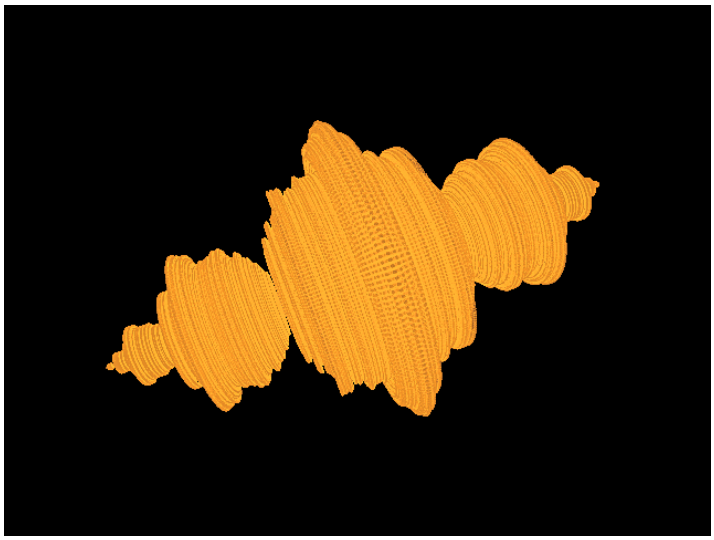


Fig.7 - POV 3.7 $3D$ Julia set with cylindrical symmetry ($c = -0.7454294$)

VIEW ANIMATION (requires internet connection)

Also for the *Julia set* a better rendering effect arises when discrete steps replace continuous rotation, or viewing an animation of the revolution process.

The insight into the interior structure of the body improves the quality of the picture.

POV-Ray 3.7 code to generate fig. 7 and 8

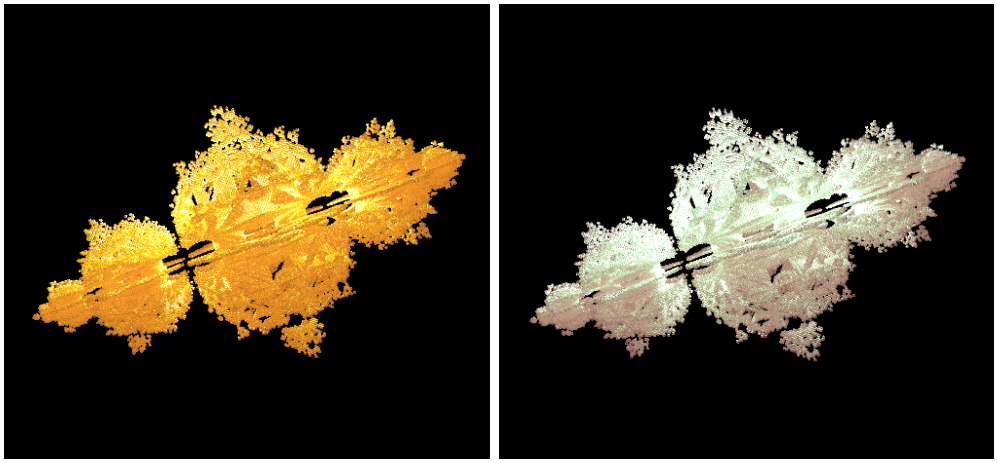


Fig.8 - POV 3.7 $3D$ Julia set ($c = -0.7454294$) with cylindrical symmetry
(two different glass renderings)

```
//================================
// Julia 3D set as a whole
// (POV-Ray cylindrical symmetry)
//================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }    // set black background

camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0     // set white light color and intensity
}

light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

#declare R = .1;     // set radius value
#declare L = 2;  // set square area side
#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;  // set co-ordinate y initial value
#declare Z = 0;  // set co-ordinate z initial value

#declare n = 200;    // set number of pixels per area side
#declare N = 30;   // set number of cycles
#declare Nr = 100;   // alternative  Nr = 25;   // set number of sections
#declare Th = -5;
```

```
#declare Ph = 20;

#declare Cx  = -0.7454294;    // set parameter c values
#declare Cy = 0;

union{ #for (p, -n, n, 1)
  #declare IncX = p*L/n;
  #for (q, -n, n, 1)
      #declare IncY = q*L/n;
      #declare X = IncX;
      #declare Y = IncY;
      #for (i,1,N)
      #declare XX = X*X - Y*Y  + Cx;
      #declare YY = 2*X*Y + Cy;
      #declare X = XX;
      #declare Y = YY;
       #if (X*X+Y*Y > R)
       #if (X*X+Y*Y < R+.005)
       #for (k,0,Nr)    //  replace n by n*clock for animation
    sphere {
     < p, q*cos(3.14*k/Nr), q*sin(3.14*k/Nr) >, 1
   // hollow
   // radiosity { importance 1.0 }
     texture {
       pigment { color Col_Glass_Yellow }
        }
       finish { ambient rgb <0.3,0.1,0.1>
       diffuse .3
      reflection .3
      specular 1 }
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for i
#end  // end for q
#end  // end for p
rotate < 0, Th, Ph >}
```

The open sections of the Julia set with $c = -0.7454294$ offer also very nice images (see fig. 9).

POV-Ray 3.7 code to generate fig. 9

```
//======================================
// Julia 3D set as a whole
// (POV-Ray open section)
//======================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }    // set black background

camera { // set view point (camera) location
    location <-20, 20, -300>
```

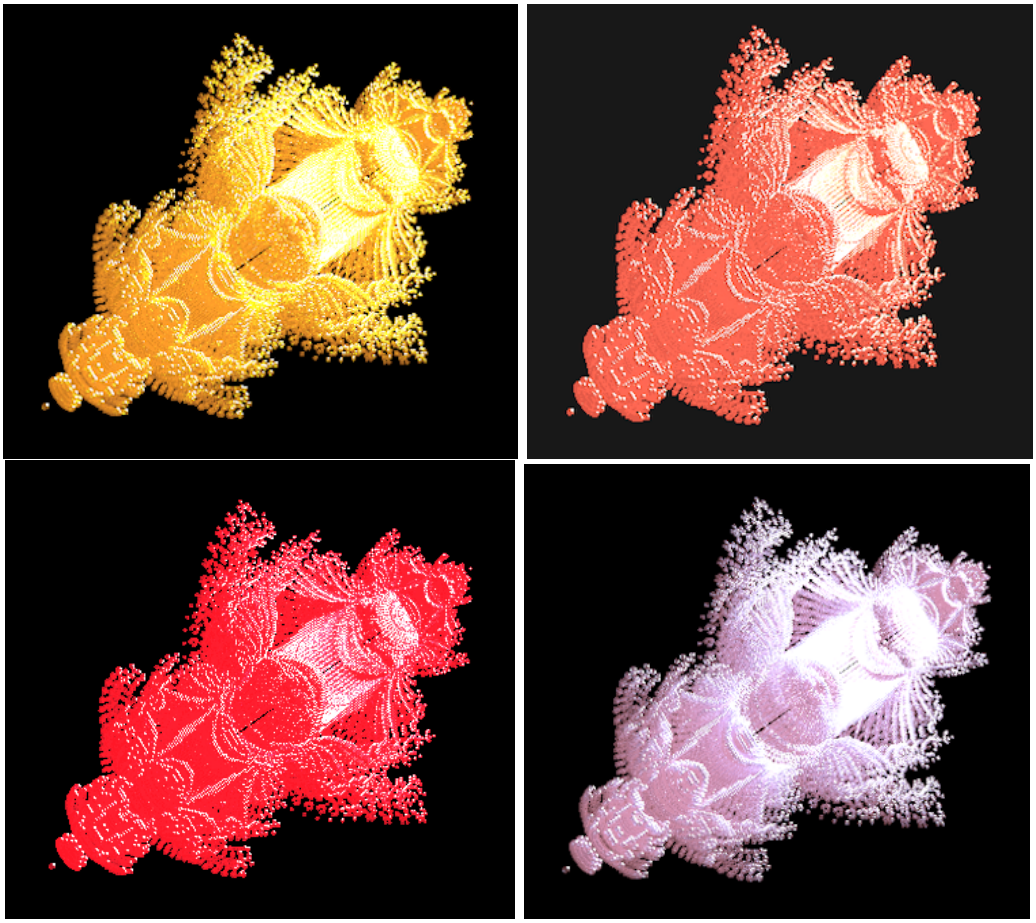


Fig.9 - POV 3.7 $3D$ Julia set with cylindrical symmetry open section
($c = -0.7454294$ – four different glass renderings)

```
    look_at  <-5, 0,  0>
}

light_source { // set point light sources location
< -3.0, 10, -5>
rgb <1.000000, 1.000000, 1.000000> * 2.0     // set white light color and intensity
}

light_source {
< 5.0, 10, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

#declare R = .1;     // set radius value
#declare L = 2;  // set square area side
#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;  // set co-ordinate y initial value
#declare Z = 0;  // set co-ordinate z initial value

#declare n = 200;    // set number of pixels per area side
#declare N = 30;   // set number of cycles
#declare Nr = 25;   // set number of sections
#declare Th = -45;
#declare Ph = 30;

#declare Cx  = -0.7454294;    // set parameter c values
#declare Cy = 0;

union{ #for (p, -n, n, 1)
  #declare IncX = p*L/n;
    #for (q, -n, n, 1)
      #declare IncY = q*L/n;
      #declare X = IncX;
      #declare Y = IncY;
      #for (i,1,N)
      #declare XX = X*X - Y*Y  + Cx;
      #declare YY = 2*X*Y + Cy;
      #declare X = XX;
      #declare Y = YY;
       #if (X*X+Y*Y > R)
       #if (X*X+Y*Y < R+.002)
       #for (k,0,Nr)
    sphere {
     < p, abs(q)*cos(3.14*k/Nr), abs(q)*sin(3.14*k/Nr) >, 1
     hollow
     radiosity { importance 1.0 }
     texture {
       pigment { color Col_Glass_Ruby }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for i
#end  // end for q
#end  // end for p
rotate < 0, Th, Ph >}
```

5.1.3 Generalized Newton's method set with cylindrical symmetry

The $2D$ starting Newton's method set

Differently respect to Mandelbrot and Julia sets which fill bounded regions of space, the Newton's method sets extend towards infinity. Therefore we need consider only a portion of a $2D$ set to be revolved in order to construct a $3D$ body.

As an example we will examine a part of a Newton's method $2D$ set related to the polynomial $f(z) = z^4 + 1$.

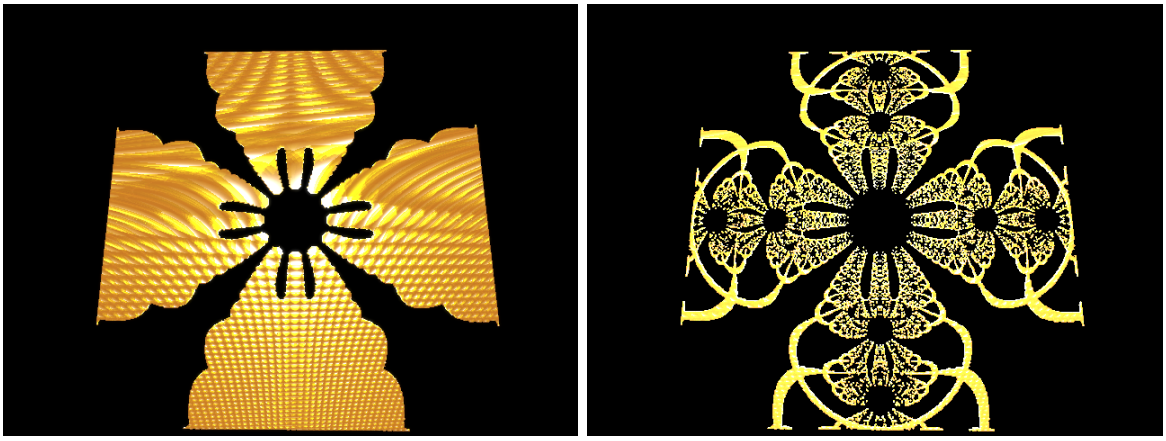


Fig.10 - POV 3.7 two dimensional Newton's method set $[f(z) = z^4 + 1]$ as a whole (full and carved)

POV-Ray 3.7 code to generate fig. 10

```
//=========================================================
// Newton's method 2D set as a whole (full and carved)
// (POV-Ray point by point plot)
//=========================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <0.0,0.0,0.0> }   // set black background

camera { // set view point (camera) location
    location <0, -50, -300>
    look_at  <-5, 0,  0>
    angle 90
}

light_source { // set point light sources location
< 0.0, 0, -10>
```

```
rgb <1.000000, 1.000000, 1.000000> * 10.0     // set white light color and intensity
}

#declare R = 0.9;     // set radius value
#declare L = 1.4;  // set square area side
#declare X = 0.01;  // set co-ordinate x initial value
#declare Y = 0.01;  // set co-ordinate y initial value
#declare n = 200;    // set number of pixels per area side
#declare N = 10;   // set number of cycles

#for (p, -n, n, 1)
    #declare IncX = p*L/n;
    #for (q, -n, n, 1)
    #declare IncY = q*L/n;
    #declare X = IncX;
    #declare Y = IncY;
       #for (k,0,N)
       #declare XX = 3*X/4 - X*(X*X - 3*Y*Y)/(X*X + Y*Y)/(X*X + Y*Y)/
       (X*X + Y*Y)/4;
       #declare YY = 3*Y/4 - Y*(Y*Y - 3*X*X)/(X*X + Y*Y)/(X*X + Y*Y)/
       (X*X + Y*Y)/4;
       #declare X = XX;
       #declare Y = YY;
      #if (X*X+Y*Y > R)  // comment for full set
      #if (X*X+Y*Y < R+.05)
   sphere {
     < p, q, 0 >, 1
      hollow
   radiosity { importance 1.0 }
     texture {
       pigment { color Col_Glass_Yellow
     }}
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .1
       reflection .1
       specular 1
      }}

#end  // end if   // comment for full set
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
```

The $3D$ generalized Newton's method set

Revolving the carved $2D$ *Newton's method set* we generate a $3D$ set with cylindrical symmetry.

POV-Ray 3.7 code to generate fig. 11 and 12

```
//====================================
// Newton's method 3D set as a whole
// (POV-Ray cylindrical symmetry)
//====================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
```

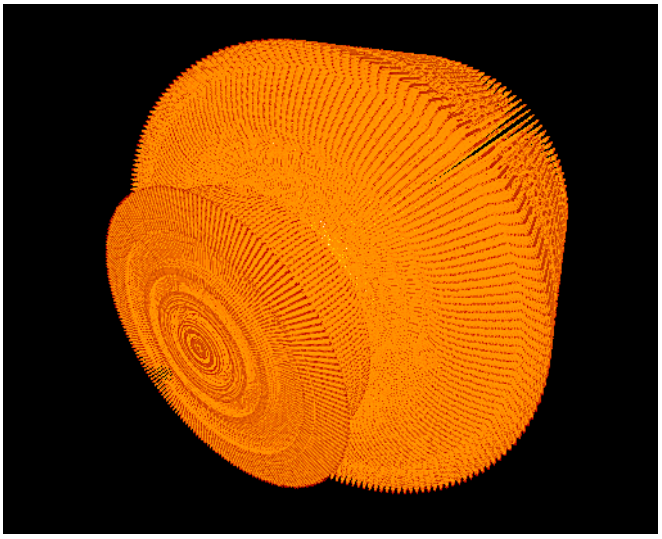


Fig.11 - POV 3.7 $3D$ Newton's method set with cylindrical symmetry $[f(z) = z^4 + 1]$

VIEW ANIMATION (requires internet connection)

```
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <0.0,0.0,0.0> }   // set black background

 camera { // set view point (camera) location
    location <-20, 20, -400>
    look_at  <-35, 1,  0>
    angle 100
  }

  light_source   // set area light sources location
{
  0*x // light's position (translated below)
  color Silver  // light's color
  // <widthVector> <heightVector> nLightsWide mLightsHigh
  area_light
  <8, 0, 0> <0, 0, 8> // lights spread out across this distance (x * z)
  4, 4                // total number of lights in grid (4x*4z = 16 lights)
  adaptive 0          // 0,1,2,3...
  jitter              // adds random softening of light
  translate <40, 80, -40>   // <x y z> position of light
}

#declare R = 0.9;     // set radius value
#declare L = 1.10;  // set square area side
#declare X = 0.01;  // set co-ordinate x initial value
#declare Y = 0.01;  // set co-ordinate y initial value

#declare n = 200;    // set number of pixels per area side
#declare N = 10;   // set number of cycles
 #declare Nr = 100;  // alternative  Nr = 25;   // set number of sections
 #declare Th = -45;
  #declare Ph = 30;
```

```
  #for (p, -n, n, 1)
    #declare IncX = p*L/n;
    #for (q, -n, n, 1)
    #declare IncY = q*L/n;
    #declare X = IncX;
    #declare Y = IncY;
       #for (i,1,N)
       #declare XX = 3*X/4 - X*(X*X - 3*Y*Y)/(X*X + Y*Y)/(X*X + Y*Y)/
       (X*X + Y*Y)/4;
       #declare YY = 3*Y/4 - Y*(Y*Y - 3*X*X)/(X*X + Y*Y)/(X*X + Y*Y)/
       (X*X + Y*Y)/4;
       #declare X = XX;
       #declare Y = YY;
   #if (X*X+Y*Y > R)
#if (X*X+Y*Y < R+.05)
#for (k,0,Nr)   //  replace Nr by Nr*clock for animation
   sphere {
      < p, q*cos(3.14*k/Nr), q*sin(3.14*k/Nr) >, 1
     hollow
   radiosity { importance 1.0 }
     texture {
       pigment { color Col_Glass_Yellow
     }
               }
        finish { ambient rgb <0.3,0.1,0.1>
 diffuse .3
       reflection .3
       specular 1
      }
    }

#end  // end if
#end  // end if
#end  // end for k
#end  // end for i
#end  // end for q
#end  // end for p
rotate < 0, Th, Ph >
translate <50,20,0> }
```

Also for the *Newton's method set* a better rendering effect arises when discrete steps replace continuous rotation or while observing an animation of the revolution process. The insight into the interior structure of the body improves the quality of the picture.

The open sections reveal more explicitly the fractal structure as one is able to recognize in the following images (see fig. 13).

POV-Ray 3.7 code to generate fig. 13

```
//================================================
// Newton's method 3D set as a whole
// (POV-Ray cylindrical symmetry open section)
//================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
```

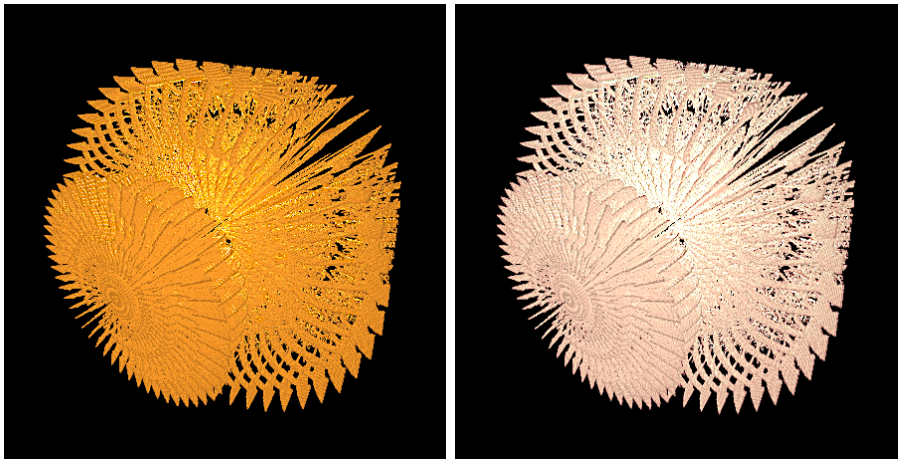


Fig.12 - POV 3.7 $3D$ Newton's method set $[f(z) = z^4 + 1]$ with cylindrical symmetry (two different glass renderings)

```
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }    // set black background

camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-35, 1,  0>
    angle 100
}

light_source    // Area light source (creates soft shadows)
{
  0*x // light's position (translated below)
  color Silver  // light's color
  // <widthVector> <heightVector> nLightsWide mLightsHigh
  area_light
  <8, 0, 0> <0, 0, 8> // lights spread out across this distance (x * z)
  4, 4                // total number of lights in grid (4x*4z = 16 lights)
  adaptive 0          // 0,1,2,3...
  jitter              // adds random softening of light
  translate <40, 80, -40>   // <x y z> position of light
}

#declare R = 0.9;     // set radius value
#declare L = 1.10;  // set square area side
#declare X = 0.01;  // set co-ordinate x initial value
#declare Y = 0.01;  // set co-ordinate y initial value
#declare Z = 0;   // set co-ordinate z initial value
#declare n = 200;    // set number of pixels per area side
#declare N = 15;  // set number of cycles
#declare Nr = 25;   // set number of sections
#declare Th = -35;
#declare Ph = 30;

 union{ #for (p, -n, n, 1)
  #declare IncX = p*L/n;
```

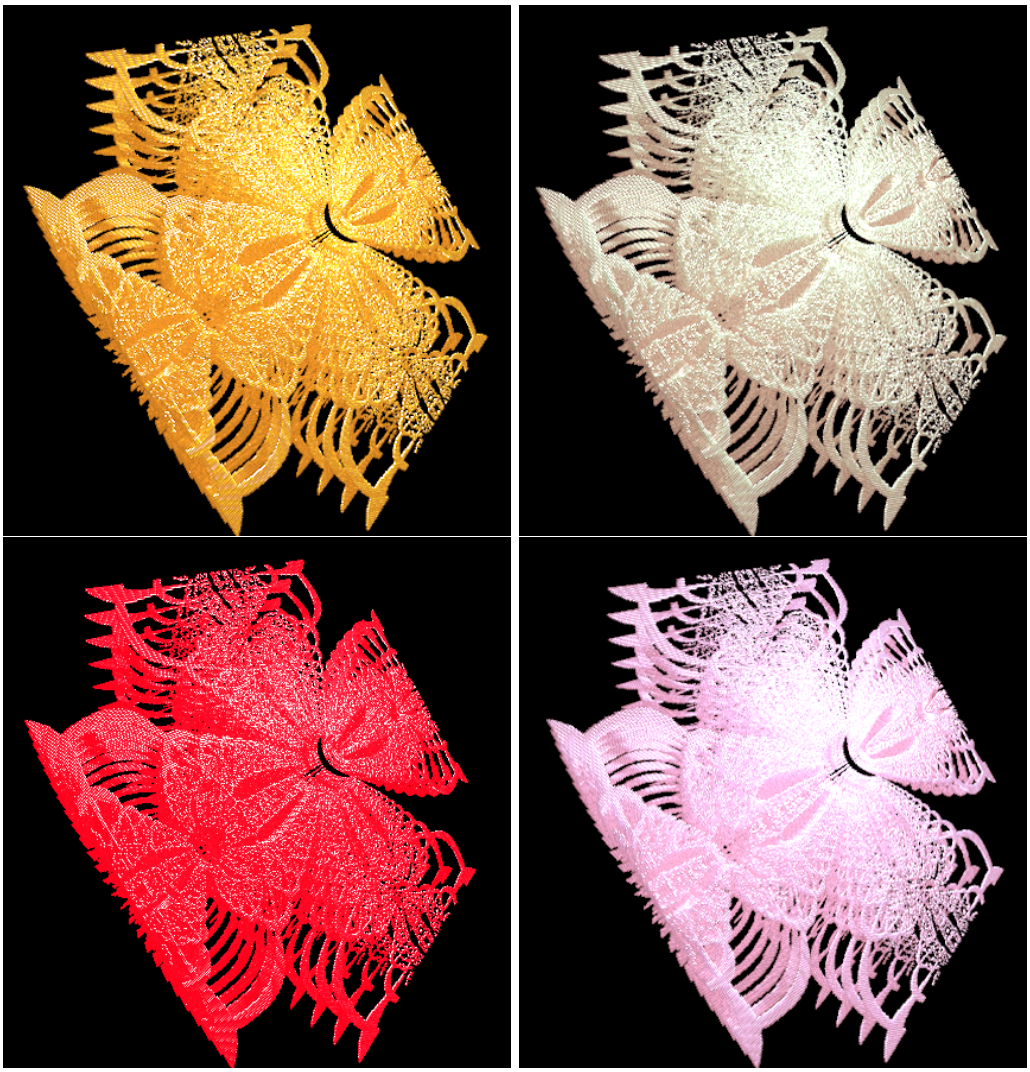


Fig.13 - POV 3.7 $3D$ Newton's method set with cylindrical symmetry open section $[f(z) = z^4 + 1]$
(four different glass renderings)

```
    #for (q, -n, n, 1)
      #declare IncY = q*L/n;
      #declare X = IncX;
      #declare Y = IncY;
       #for (i,1,N)
       #declare XX = 3*X/4 - X*(X*X - 3*Y*Y)/(X*X + Y*Y)/(X*X + Y*Y)/
       (X*X + Y*Y)/4;
       #declare YY = 3*Y/4 - Y*(Y*Y - 3*X*X)/(X*X + Y*Y)/(X*X + Y*Y)/
       (X*X + Y*Y)/4;
       #declare X = XX;
       #declare Y = YY;
       #if (X*X+Y*Y > R)
       #if (X*X+Y*Y < R+.05)


        #for (k,0,Nr)
    sphere {
     < p, abs(q)*cos(3.14*k/Nr), abs(q)*sin(3.14*k/Nr) >, 1
     hollow
     radiosity { importance 1.0 }
     texture {
      pigment { color Col_Glass_Yellow }
     // alternative  pigment { color Col_Glass_Old }
     // alternative  pigment { color Col_Glass_Red }
     // alternative  pigment { color Col_Glass_Bluish }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for i
#end  // end for q
#end  // end for p
rotate < 0, Th, Ph >
translate <50,20,0> }
```

5.2 Sequential rendering of fractal structures with cylindrical symmetry

Following the same exposition order as in the previous chapter we now examine the process of generation of $3D$ fractal structures with cylindrical symmetry, no more limiting ourselves to view the final result of the process, *i.e.,* the structure built as a whole. As before we will consider generalized $3D$ *Mandelbrot, Julia and Newton's method* sets, according to two generation processes.

1. In this section (§5.2) we present a *sequentially ordered* rendering process.

2. While in the next section (§5.3) we will follow a *random* process from which an ordered structure will emerge as an attractor.

5.2.1 *POV-Ray 3.7* sequential rendering of $3D$ Mandelbrot set with cylindrical symmetry

We begin showing the images of four sequential steps of a generalized $3D$ *Mandelbrot set* with cylindrical symmetry (see fig. 14).

POV-Ray 3.7 code to generate fig. 14

```
//================================================
// Mandelbrot 3D set generated sequentially
// (POV-Ray cylindrical symmetry open section)
//================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }    // set black background

camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
}

light_source { // set point light sources location
< -3.0, 10, -5>
rgb <1.000000, 1.000000, 1.000000> * 2.0     // set white light color and intensity
}

light_source {
< 5.0, 10, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

#declare R = .1;     // set radius value
#declare L = 2;  // set square area side
#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;  // set co-ordinate y initial value
#declare Z = 0;  // set co-ordinate z initial value

#declare n = 200;    // set number of pixels per area side
#declare N = 10;   // set number of cycles
#declare Nr = 50;     // set number of sections
#declare Th = -45;
#declare Ph = 30;

union{ #for (p, -n, n, 1)
    #for (q, 0, n, 1) // alternative n/8, n/4, n/2
       #declare X = 0;
       #declare Y = 0;
       #for (i,1,N)
       #declare XX = X*X - Y*Y  + p*L/n - 1;
       #declare YY = 2*X*Y + q*L/n;
       #declare X = XX;
       #declare Y = YY;
       #if (X*X+Y*Y > R)
        #if (X*X+Y*Y < R+.01)
        #for (k,0,Nr)
    sphere {
```

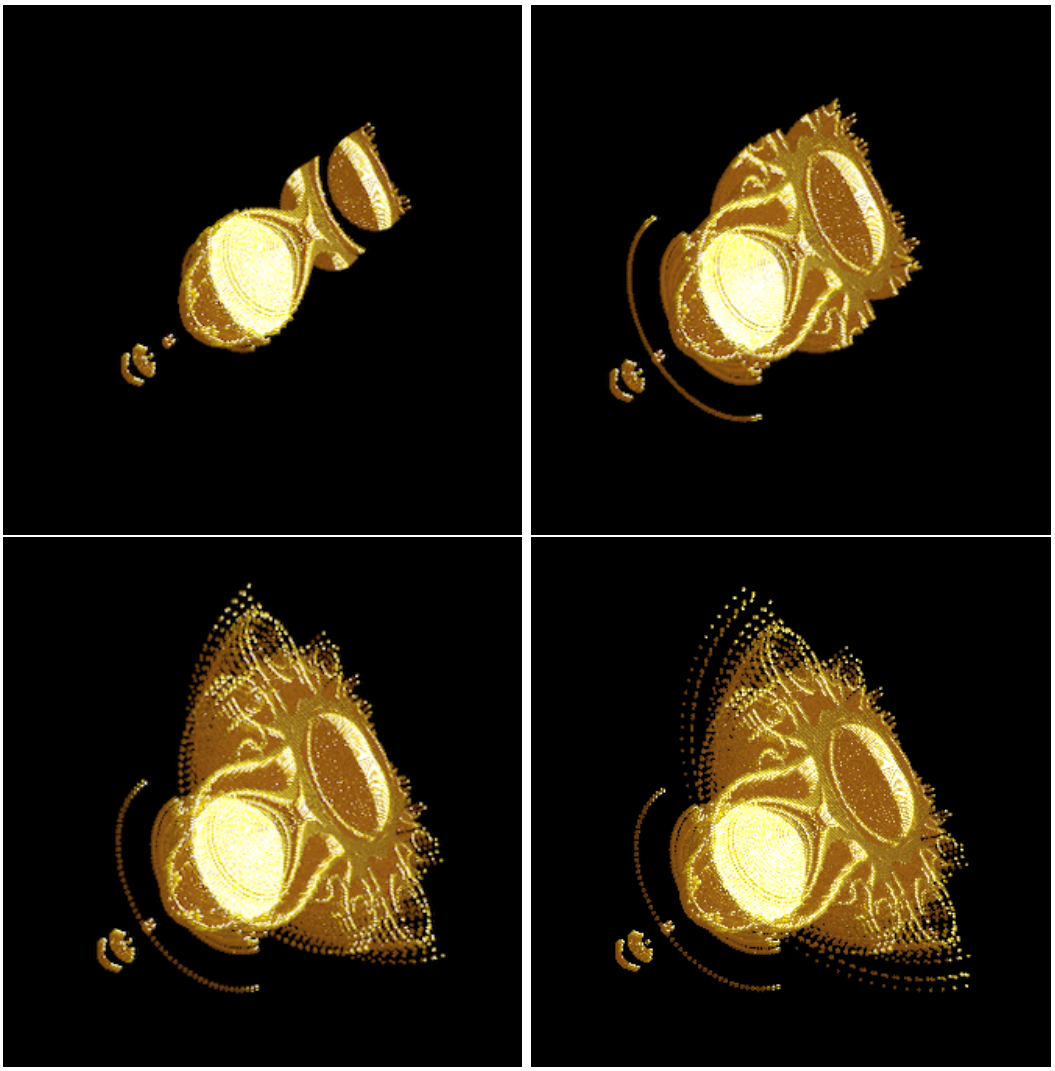


Fig.14 - POV 3.7 $3D$ Mandelbrot set with cylindrical symmetry open section generated sequentially

```
     < p, q*cos(3.14*k/Nr), q*sin(3.14*k/Nr) >, 1
     texture {
      pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for i
#end  // end for q
#end  // end for p
rotate < 0, Th, Ph >}
```

5.2.2 *POV-Ray 3.7* sequential rendering of $3D$ Julia set with cylindrical symmetry

After examining the sequential code generating a *Mandelbrot set* with cylindrical symmetry we see, now, how in a similar way one can obtain a *Julia set* by rotation around to a symmetry axis (see fig. 15).

POV-Ray 3.7 code to generate fig. 15

```
//=================================================
// Julia 3D set generated sequentially
// (POV-Ray cylindrical symmetry open section)
//=================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }    // set black background

camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

 light_source { // set point light sources location
< -3.0, 10, -5>
rgb <1.000000, 1.000000, 1.000000> * 2.0     // set white light color and intensity
 }

 light_source {
< 5.0, 10, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 #declare R = .1;     // set radius value
 #declare L = 2;  // set square area side
 #declare X = 0;  // set co-ordinate x initial value
```

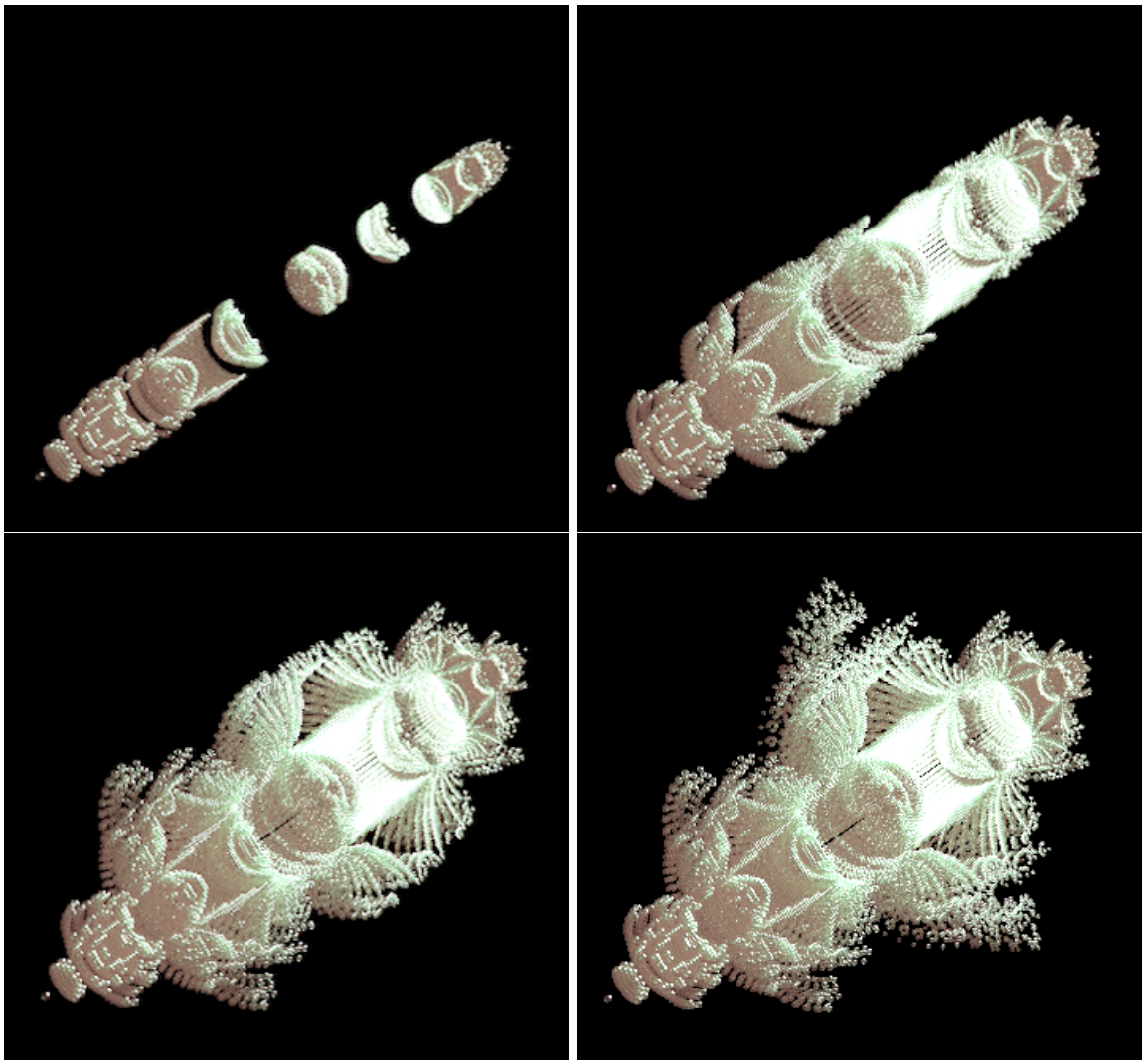


Fig.15 - POV 3.7 3*D* Julia set with cylindrical symmetry open section generated sequentially

```
#declare Y = 0;  // set co-ordinate y initial value
#declare Z = 0;  // set co-ordinate z initial value
#declare n = 200;    // set number of pixels per area side
#declare N = 30;   // set number of cycles
#declare Nr = 25;    // set number of sections
#declare Th = -45;
 #declare Ph = 30;
#declare Cx  = -0.7454294;    // set parameter c values
#declare Cy = 0;

union{ #for (p, -n, n, 1)
 #declare IncX = p*L/n;
   #for (q, 0, n, 1)    // alternative n/16, n/8, n/4
      #declare IncY = q*L/n;
     #declare X = IncX;
     #declare Y = IncY;
      #for (i,1,N)
      #declare XX = X*X - Y*Y  + Cx;
      #declare YY = 2*X*Y + Cy;
      #declare X = XX;
      #declare Y = YY;
      #if (X*X+Y*Y > R)
       #if (X*X+Y*Y < R+.002)
       #for (k,0,Nr)
   sphere {
    < p, q*cos(3.14*k/Nr), q*sin(3.14*k/Nr) >, 1
    hollow
  radiosity { importance 1.0 }
     texture {
       pigment { color Col_Glass_Old }
        }
       finish { ambient rgb <0.3,0.1,0.1>
       diffuse .3
       reflection .3
       specular 1 }
   }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for i
#end  // end for q
#end  // end for p
rotate < 0, Th, Ph >}
```

5.2.3 *POV-Ray 3.7* sequential rendering of $3D$ Newton's method set with cylindrical symmetry

Finally we show a genralized $3D$ *Newton's method* set with cylindrical symmetry (see fig. 16).

POV-Ray 3.7 code to generate fig. 16

```
//================================================
// Newton's method 3D set generated sequentially
// (POV-Ray cylindrical symmetry open section)
//================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
```

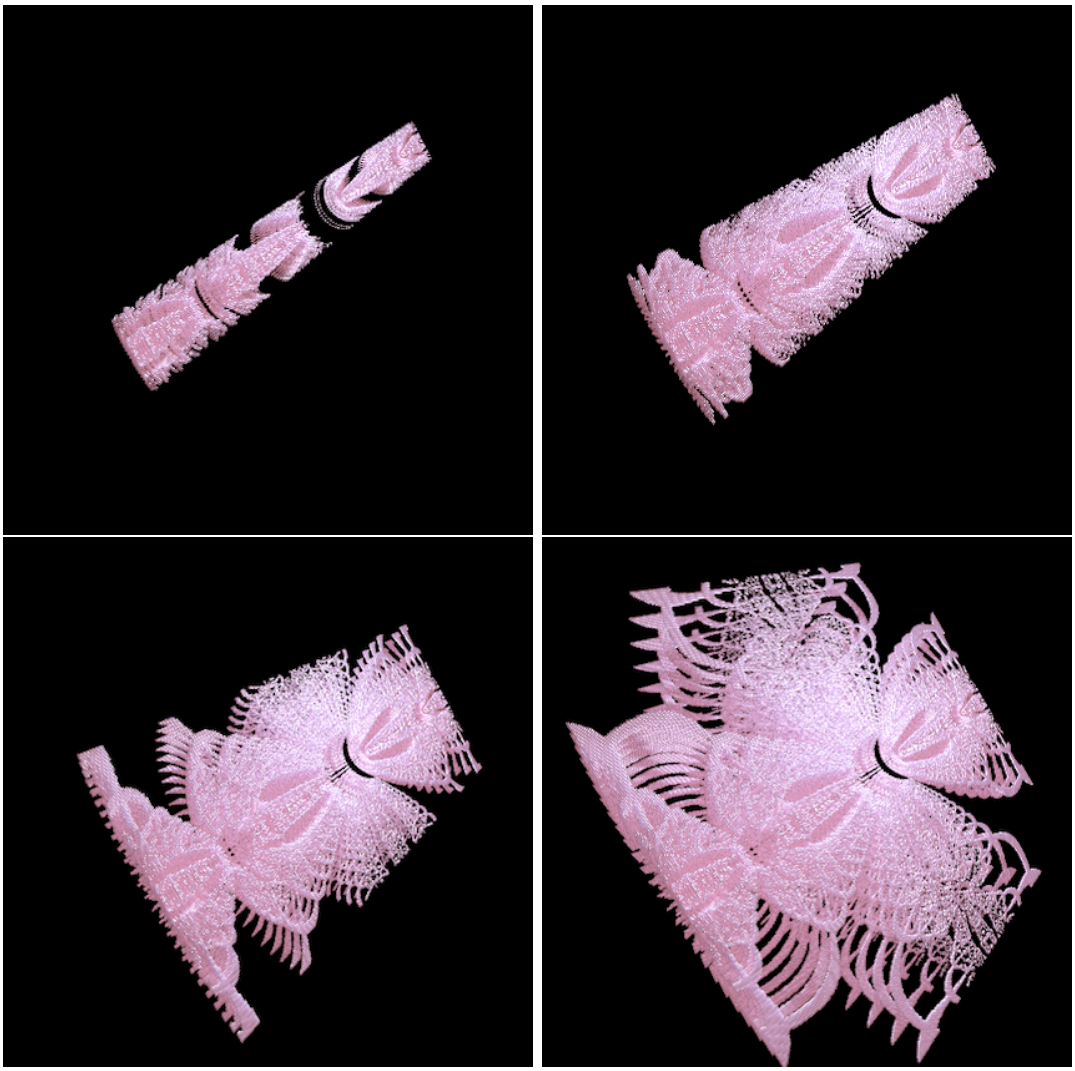


Fig.16 - POV 3.7 3*D* Newton's method set with cylindrical symmetry open section generated sequentially

```
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }    // set black background

camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-35, 1,  0>
    angle 100
  }

light_source    // Area light source (creates soft shadows)
{
  0*x // light's position (translated below)
  color Silver  // light's color
  // <widthVector> <heightVector> nLightsWide mLightsHigh
  area_light
  <8, 0, 0> <0, 0, 8> // lights spread out across this distance (x * z)
  4, 4                // total number of lights in grid (4x*4z = 16 lights)
  adaptive 0          // 0,1,2,3...
  jitter              // adds random softening of light
  translate <40, 80, -40>   // <x y z> position of light
}

#declare R = 0.9;     // set radius value
#declare L = 1.10;  // set square area side
#declare X = 0.01;  // set co-ordinate x initial value
#declare Y = 0.01;  // set co-ordinate y initial value
#declare Z = 0;  // set co-ordinate z initial value

#declare n = 200;    // set number of pixels per area side
#declare N = 15;   // set number of cycles
#declare Nr = 25;  // set number of sections
#declare Th = -35;
#declare Ph = 30;

union{ #for (p, -n, n, 1)
  #declare IncX = p*L/n;
      #for (q, 0, n, 1)  // alternative n/8,  n/4, n/2
      #declare IncY = q*L/n;
      #declare X = IncX;
      #declare Y = IncY;
      #for (i,1,N)
       #declare XX = 3*X/4 - X*(X*X - 3*Y*Y)/(X*X + Y*Y)/(X*X + Y*Y)/
       (X*X + Y*Y)/4;
       #declare YY = 3*Y/4 - Y*(Y*Y - 3*X*X)/(X*X + Y*Y)/(X*X + Y*Y)/
       (X*X + Y*Y)/4;
       #declare X = XX;
       #declare Y = YY;
       #if (X*X+Y*Y > R)
        #if (X*X+Y*Y < R+.05)
        #for (k,0,Nr)
     sphere {
     < p, q*cos(3.14*k/Nr), q*sin(3.14*k/Nr) >, 1
     hollow
     radiosity { importance 1.0 }
     texture {
       pigment { color Col_Glass_Bluish }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }
```

```
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for i
#end  // end for q
#end  // end for p
rotate < 0, Th, Ph >
translate <50,20,0> }
```

5.3 Random rendering of fractal structures with cylindrical symmetry

The random rendering of ordered structures of any kind of systems – included fractals – is very interesting since it allows to show how a random distribution of initial points, which does not exhibit any apparent order, evolves assuming a gradually more and more ordered configuration thanks to the *information* coded into the *law* governing the evolution process.

In the present section we will show four steps of the evolution of each one of our test fractals, *i.e.,* generalized $3D$ *Mandelbrot, Julia* and *Newton's method* sets endowed with cylindrical symmetry.

5.3.1 *POV-Ray 3.7* random rendering of $3D$ Mandelbrot set with cylindrical symmetry

As usual we start with *Mandelbrot set* (see fig. 17)

POV-Ray 3.7 code to generate fig. 17

```
//================================================
// Mandelbrot 3D set generated randomly
// (POV-Ray cylindrical symmetry)
//================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }    // set black background

camera { // set view point (camera) location
    location <-20, 20, -150>
    look_at  <-5, 0,  0>
  }

 light_source { // set point light sources location
< -3.0, 10, -5>
rgb <1.000000, 1.000000, 1.000000> * 2.0     // set white light color and intensity
 }

 light_source {
< 5.0, 10, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }
```

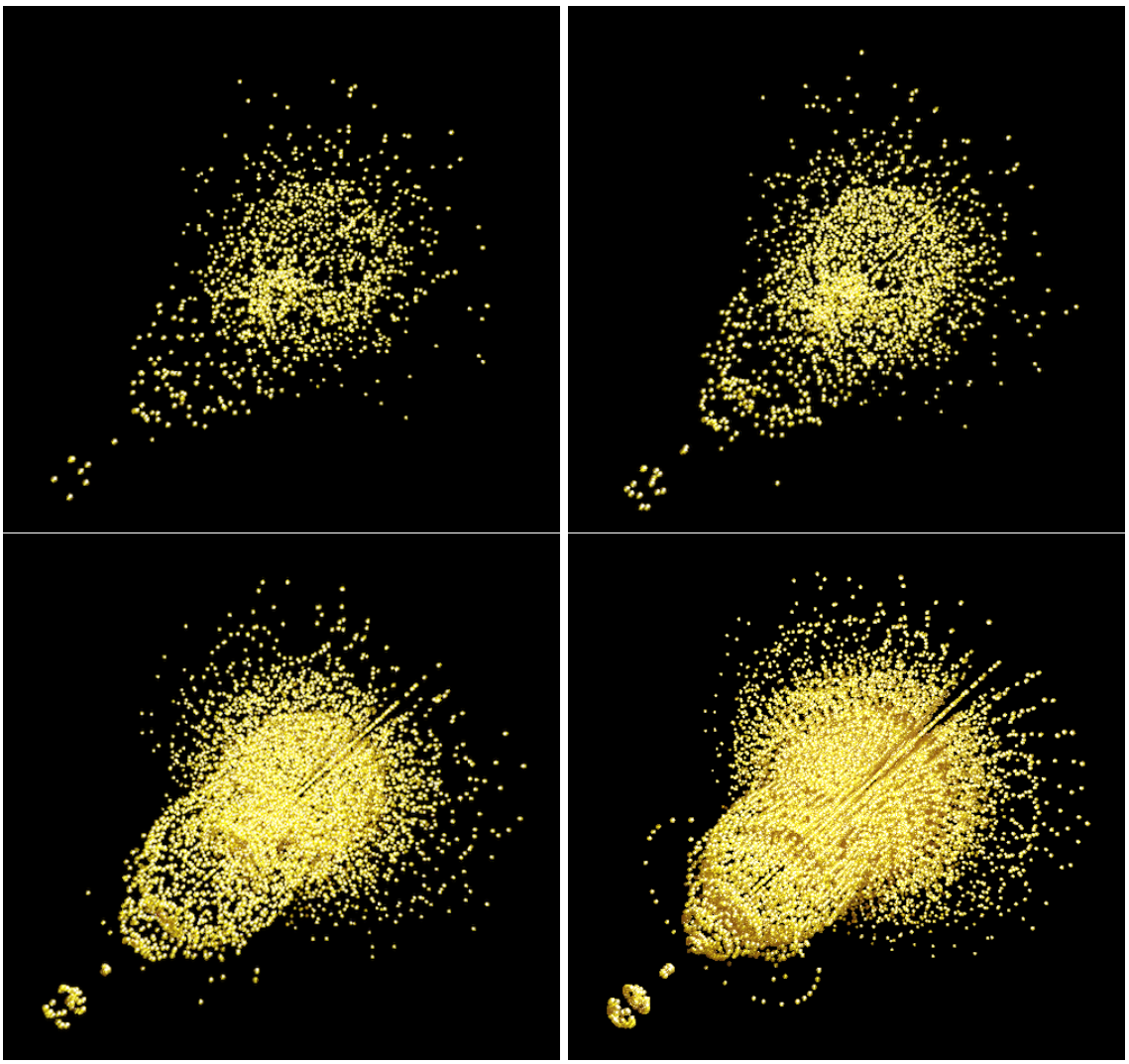


Fig.17 - POV 3.7 3*D* Mandelbrot set with cylindrical symmetry generated randomly

VIEW ANIMATION (requires internet connection)

```
#declare R = .1;     // set radius value
#declare L = 2;  // set square area side
#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;  // set co-ordinate y initial value
#declare Z = 0;  // set co-ordinate z initial value

#declare n = 200;    // set number of pixels per area side
#declare N = 10;   // set number of cycles
#declare Nr = 50;   // set number of sections
#declare Th = -45;
#declare Ph = 30;

#declare P = array[2*n+1][2*n+1];    //  P matrix
#declare Q = array[2*n+1][2*n+1];   // Q matrix

union{ #for (p, -n, n, 1)
    #for (q, -n, n, 1)
       #declare X = 0;
       #declare Y = 0;
       #declare P[p+n][q+n] = 0;
       #declare Q[p+n][q+n] = 0;

       #for (i,0,N)
       #declare XX = X*X - Y*Y + p*L/n;
       #declare YY = 2*X*Y + q*L/n;
       #declare X = XX;
       #declare Y = YY;
       #if (X*X+Y*Y > R)
        #if (X*X+Y*Y < R+.01)
        #declare P[p+n][q+n] = p+n;
        #declare Q[p+n][q+n] = q;
#end  // end if
#end  // end if
#end  // end for i
#end  // end for q
#end  // end for p

#declare Rnd_1 = seed (1153);
#declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (758);

//  replace Nr by Nr*clock for animation
#for(j,0,n*n*Nr)    // partial values n*n/32, n*n/16, n*n/4
#declare p = int(2*n*rand(Rnd_1));
#declare q = int(2*n*rand(Rnd_2));
#declare k = int(2*Nr*rand(Rnd_3));
     sphere {
     <  P[p][q], Q[p][q]*cos(6.28*k/Nr), Q[p][q]*sin(6.28*k/Nr) >, 1
      texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }
    }

#end  // end for j
rotate < 0, Th, Ph >
translate < -80, - 80, 0 >
  }
```

5.3.2 *POV-Ray 3.7* random rendering of $3D$ Julia set with cylindrical symmetry

The second example is provided by a *Julia set* ($c = -0.7454294$) with cylindrical symmetry (see fig. 18).

POV-Ray 3.7 code to generate fig. 18

```
//=======================================================
// Julia 3D set ($c = -0.7454294$) generated randomly
// (POV-Ray cylindrical symmetry)
//=======================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }    // set black background

camera { // set view point (camera) location
    location <-20, 20, -150>
    look_at  <-5, 0,  0>
}

light_source { // set point light sources location
< -3.0, 10, -5>
rgb <1.000000, 1.000000, 1.000000> * 1.0     // set white light color and intensity
}

light_source {
< 5.0, 10, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

#declare R = .1;     // set radius value
#declare L = 2;  // set square area side
#declare X = 0;  // set co-ordinate x initial value
#declare Y = 0;  // set co-ordinate y initial value
#declare Z = 0;  // set co-ordinate z initial value

#declare n = 200;    // set number of pixels per area side
#declare N = 30;   // set number of cycles
#declare Nr = 100;   // se number of sections
#declare Th = -35;
#declare Ph = 30;

#declare Cx  = -0.7454294;    // set parameter c values
#declare Cy = 0;

#declare P = array[2*n+1][2*n+1];    //  P matrix
#declare Q = array[2*n+1][2*n+1];    //  Q matrix

union{ #for (p, -n, n, 1)
  #declare IncX = p*L/n;
    #for (q, -n, n, 1)
      #declare IncY = q*L/n;
      #declare X = IncX;
      #declare Y = IncY;
      #declare P[p+n][q+n] = 0;
```

Fig.18 - POV 3.7 $3D$ Julia set ($c = -0.7454294$) with cylindrical symmetry generated randomly

VIEW ANIMATION (requires internet connection)

```
       #declare Q[p+n][q+n] = 0;

       #for (i,0,N)
       #declare XX = X*X - Y*Y  + Cx;
       #declare YY = 2*X*Y + Cy;
       #declare X = XX;
       #declare Y = YY;
       #if (X*X+Y*Y > R)
        #if (X*X+Y*Y < R+.01)
        #declare P[p+n][q+n] = p+n;
        #declare Q[p+n][q+n] = q;
#end  // end if
#end  // end if
#end  // end for i
#end  // end for q
#end  // end for p

#declare Rnd_1 = seed (1153);
#declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (758);

//  replace Nr by Nr*clock for animation
#for(j,0,n*n*Nr)    // partial values n*n/64, n*n/16, n*n/4
#declare p = int(2*n*rand(Rnd_1));
#declare q = int(2*n*rand(Rnd_2));
#declare k = int(2*Nr*rand(Rnd_3));
    sphere {
     < P[p][q], Q[p][q]*cos(6.28*k/Nr), Q[p][q]*sin(6.28*k/Nr) >, 1
    texture {
     pigment { color Col_Glass_Old }
        }
        finish { ambient rgb <0.3,0.1,0.1>
       diffuse .3
       reflection .3
       specular 1 }
    }

#end  // end for j
rotate < 0, Th, Ph >
translate < -110, - 80, 0 >}
```

5.3.3 *POV-Ray 3.7* random rendering of $3D$ Newton's method set with cylindrical symmetry

Finally we show a portion of a *Newton's method* set generated starting from random initial points (see fig. 19). Here is the related program code.

POV-Ray 3.7 code to generate fig. 19

```
//=================================================================
// Newton's method 3D set [$f(z) = z^{4} + 1$] generated randomly
// (POV-Ray cylindrical symmetry)
//=================================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
```

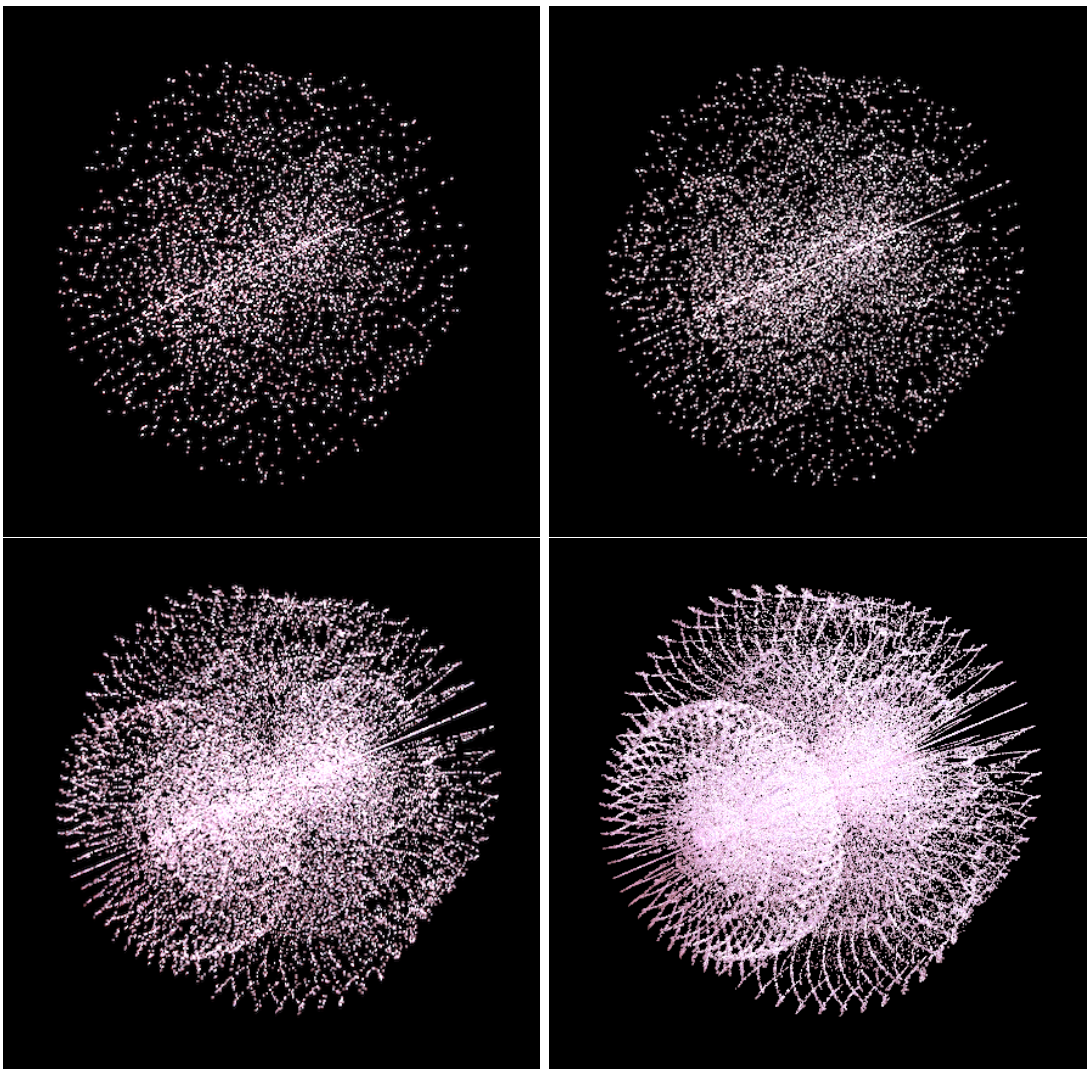


Fig.19 - POV 3.7 $3D$ Newton's method set $[f(z) = z^4 + 1]$ with cylindrical symmetry generated randomly

VIEW ANIMATION (requires internet connection)

```
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }    // set black background

camera { // set view point (camera) location
    location <-20, 20, -450>
    look_at  <-35, 1,  0>
}

  light_source    // Area light source (creates soft shadows)
{
  0*x // light's position (translated below)
  color Silver  // light's color
  // <widthVector> <heightVector> nLightsWide mLightsHigh
  area_light
  <8, 0, 0> <0, 0, 8> // lights spread out across this distance (x * z)
  2, 3                // total number of lights in grid (4x*4z = 16 lights)
  adaptive 0          // 0,1,2,3...
  jitter              // adds random softening of light
  translate <40, 80, -40>   // <x y z> position of light
}

#declare R = 0.9;     // set radius value
#declare L = 1.10;  // set square area side
#declare X = 0.01;  // set co-ordinate x initial value
#declare Y = 0.01;  // set co-ordinate y initial value
#declare Z = 0;  // set co-ordinate z initial value

#declare n = 200;    // set number of pixels per area side
#declare N = 30;   // set number of cycles
#declare Nr = 50;   // set number of sections
#declare Th = -60;
#declare Ph = 20;

#declare P = array[2*n+1][n+1];    //  P matrix
#declare Q = array[2*n+1][n+1];    //  Q matrix

union{ #for (p, -n, n, 1)
  #declare IncX = p*L/n;
  #for (q, 0, n, 1)
      #declare IncY = q*L/n;
      #declare X = IncX;
      #declare Y = IncY;
      #declare P[p+n][q] = 0;
      #declare Q[p+n][q] = 0;

       #for (i,1,N)
       #declare XX = 3*X/4 - X*(X*X - 3*Y*Y)/(X*X + Y*Y)/(X*X + Y*Y)/
       (X*X + Y*Y)/4;
       #declare YY = 3*Y/4 - Y*(Y*Y - 3*X*X)/(X*X + Y*Y)/(X*X + Y*Y)/
       (X*X + Y*Y)/4;
       #declare X = XX;
       #declare Y = YY;
       #if (X*X+Y*Y > R)
        #if (X*X+Y*Y < R+0.02)
         #declare P[p+n][q] = p+n;
        #declare Q[p+n][q] = q;
#end  // end if
#end  // end if
#end  // end for i
#end  // end for q
#end  // end for p
```

```
#declare Rnd_1 = seed (1153);
#declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (758);

#for (j,0,n*n*Nr)      // partial values n*n/64, n*n/16, n*n/4
#declare p = int(2*n*rand(Rnd_1));
#declare q = int(n*rand(Rnd_2));
#declare k = int(Nr*rand(Rnd_3));
    sphere {
     < P[p][q], Q[p][q]*cos(2*pi*k/Nr), Q[p][q]*sin(2*pi*k/Nr) >, 1
     hollow
  radiosity { importance 1.0 }
     texture {
       pigment { color Col_Glass_Bluish }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }
    }

#end  // end for j
rotate < 0, Th, Ph >
translate <-150,-50,0> }
```

Chapter 6

Three-dimensional fractals from quaternions and hypercomplex numbers

Rendering whole structures, sequential and random processes

The $3D$ fractals we dealt with in the previous chapters either as *landscapes* or as *structures* with *cylindrical symmetry,* generated by revolution around a symmetry axis, possess a *complexity level* depending only on one complex parameter variation, just like the $2D$ fractals starting from which they are built. In fact *Mandelbrot, Julia* and *Newton's method* fractals are constructed employing complex numbers ($z = a + ib$) which are characterized only by two real components (a, b) identifying two space dimensions (Gauss plane).

In order to raise the degree of complexity, people thought to generalize fractal generation from usual complex numbers to multidimensional generalized complex numbers, like *e.g., quaternions* and some kind of *hypercomplex numbers.*

a) *Quaternions* are a sort of generalization of scalar complex numbers $z = a + ib$ where the imaginary unit i solves the algebraic equation:

$$z^2 = -1, \tag{6.1}$$

to new entities $q = a + ib + jc + kd$ involving further imaginary units j, k.

Each *quaternion* is equivalent to a $2D$ matrix solving the matrix equation:

$$M^2 = -U \quad \Longleftrightarrow \quad \begin{pmatrix} M_{11} & M_{22} \\ M_{21} & M_{22} \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{6.2}$$

The previous equation has three independent solutions:

$$I = \begin{pmatrix} i & 0 \\ 0 & -i \end{pmatrix}, \quad J = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \quad K = \begin{pmatrix} 0 & i \\ i & 0 \end{pmatrix}, \tag{6.3}$$

which are the matrix representation of the extended imaginary units i, j, k, obeying, consequently, together to the usual imaginary unit i, to the algebraic rules:

$$\begin{aligned} & i^2 = -1, \quad j^2 = -1, \quad k^2 = -1, \\ & ij = -ji = k, \quad jk = -kj = i, \quad ki = -ik = j. \end{aligned} \tag{6.4}$$

But so proceeding we have too many components to be represented in ordinary $3D$ space. The problem may be solved projecting *quaternions* onto a $3D$ hyperplane, *i.e.* onto the usual $3D$ space, choosing *e.g.,* always vanishing fourth components ($d = 0$). Therefore we may consider simply generalized complex numbers involving two imaginary units i, j as:

$$z = a + ib + jc, \tag{6.5}$$

with the algebraic rules:

$$i^2 = -1, \quad j^2 = -1, \quad ij = ji = 0. \tag{6.6}$$

So we will have three components generalized complex numbers and three real parameters involved in generalizing *Mandelbrot, Julia* and *Newton's method* sets.

b) *Hypercomplex numbers* have been obtained as an alternative possibile generalization of usual complex numbers.[1]

A type of hypercomplex numbers due to Davenport (1996) and sometimes called "*the* hypercomplex numbers" are defined according to the following multiplication rules:

$$\begin{aligned} & i^2 = j^2 = -k^2 = -1, \\ & ij = ji = k, \quad jk = kj = -i, \quad ki = ik = -j. \end{aligned} \tag{6.7}$$

c) An interesting alternative to *quaternions* and *hypercomplex* numbers, for our purposes, is offered by a simple combination of two usual complex numbers:

$$z_1 = x_1 + iy_1, \qquad z_2 = x_2 + iy_2, \tag{6.8}$$

[1] *See, e.g., Wolfram MathWorld,* "Hypercomplex Number" (mathworld .wolfram.com/Hyper complexNumber.html).

when one equates *e.g.,* their real parts so that one obtains:

$$z_1 = x + iy_1, \qquad z_2 = x + iy_2, \tag{6.9}$$

and plots the three real numbers x, y_1, y_2 along the Cartesian axes of the $3D$ space. Following the latter way we will obtain shapes similar to the ones provided by *hypercomplex* numbers, but more detailed, as we will see later, in this chapter. In the following we will refer the sets generated by such a procedure as *double complex fractals.*

6.1 The POV-Ray 3.7 "Julia_fractal" built-in function

The rendering software *POV-Ray 3.7* includes a built-in function to generate generalized $3D$ *Julia sets* employing quaternions as wholes. The process is fast until we require a relative small number of iterations, *i.e.,* a relative small computing precision. The graphical rendering is interesting even if it does not reveal significant fractal details and it does not allow – at least in a simple way – to investigate internal sections with minimum and maximum thresholds as we have shown in the previous chapter examining fractals with a cylindrical symmetry. Moreover the "Julia_fractal" built-in function does not seem to allow to investigate the generation process during its evolution steps. An aspect which, as we have already repeatedly emphasized, is relevant in order to biological applications. Here are few examples.

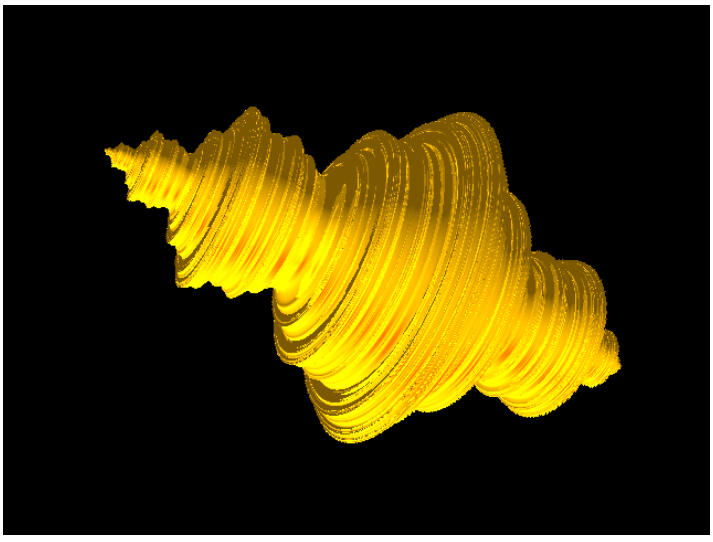


Fig.1 - POV 3.7 $3D$ Julia set ($c = -0.745429$) as a whole
(by "Julia_fractal" function [quaternion])

POV-Ray 3.7 code to generate fig 1

```
//================================================
// Julia 3D set ($c=-0.745429$) as a whole
// (POV-Ray Julia_fractal function [quaternion])
//================================================

#include "colors.inc"  // Standard Color definitions
#include "metals.inc"  // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }      // set black background

camera { // set view point (camera) location
perspective
location < 1.0, 1.0, -3>
right x * 1
up y * 3/4
angle 50
look_at < -0.2, 0.0, 0.0>
}

light_source { // set point light sources location
< 0.0, 10, -10>
rgb <1.000000, 0.500000, 0.000000> * 2.0
}

julia_fractal {        //  apply julia_fractal fucntion
  <-0.745429,0.0,0.0,0.0>
  hypercomplex
  sqr
  max_iteration 10
  precision 80
  radiosity { importance 1.0 }
  pigment {BrightGold scale 1}
   finish {
      ambient .1
      diffuse .5
      reflection 1
      specular .5
      metallic 2}
 rotate y*137
 rotate z*20}
```

The method has the relevant advantage to allow to build $3D$ structures even if they do not exhibit a cylindrical symmetry as it is shown in the picture in fig. 2

POV-Ray 3.7 code to generate fig 2

```
//=======================================================
// Julia 3D set ($c=-0.745429+j0.25$) as a whole
// (POV-Ray Julia\_fractal function [quaternion])
//=======================================================

#include "colors.inc"     // Standard Color definitions
#include "metals.inc"    // Metal pigment effect
```

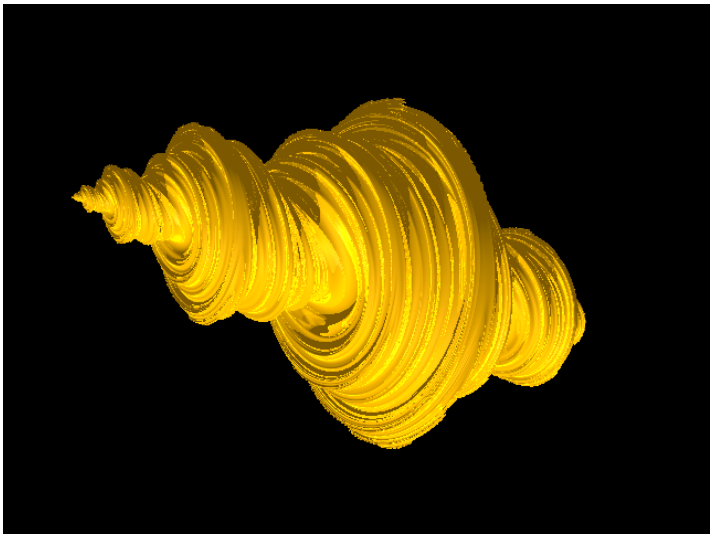


Fig.2 - POV 3.7 $3D$ Julia set ($c = -0.745429 + j\,0.25$) as a whole
(by "Julia_fractal" function [quaternion])

```
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }      // set black background

camera { // set view point (camera) location
perspective
location < 1.0, 1.0, -4>
right x * 1
up y * 3/4
angle 45
look_at < -0.1, 0.0, 0.0>
}

light_source { // set point light sources location
< 0.0, -10, -10>
rgb <1.000000, 1.000000, 1.000000> * 1.0
}

julia_fractal {        //  apply julia_fractal fucntion
  <-0.745429,0.0,0.25,0.0>
  quaternion
  sqr
  max_iteration 10
  precision 50
  pigment { Orange  scale 1}
   finish {
      ambient .2
      diffuse 1
      reflection 1
      specular .3
      metallic 1}

rotate <50,135,-20>}
```

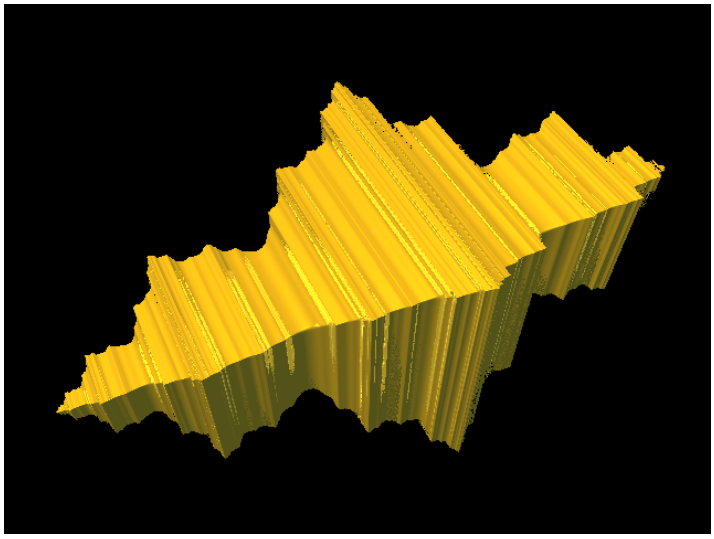


Fig.3 - POV 3.7 $3D$ Julia set ($c = -0.745429$) as a whole
(by "Julia_fractal" function [hypercomplex])

POV-Ray 3.7 code to generate fig 3

```
//=====================================================
// Julia 3D set ($c=-0.745429$) as a whole
// (POV-Ray Julia_fractal function [hypercomplex])
//=====================================================

#include "colors.inc"     // Standard Color definitions
#include "metals.inc"    // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }      // set black background

camera { // set view point (camera) location
perspective
location < 1.0, 1.0, -3>
right x * 1
up y * 3/4
angle 50
look_at < -0.2, 0.0, 0.0>
}

light_source { // set point light sources location
< 0.0, 10, -10>
rgb <1.000000, 0.500000, 0.000000> * 2.0
}

julia_fractal {        //  apply julia_fractal fucntion
  <-0.745429,0.0,0.0,0.0>
  hypercomplex
  sqr
  max_iteration 10
  precision 80
```

```
 radiosity { importance 1.0 }
 pigment {BrightGold scale 1}
  finish {
     ambient .1
     diffuse .5
     reflection 1
     specular .5
     metallic 2
   }
rotate y*137
rotate z*20
 }
```

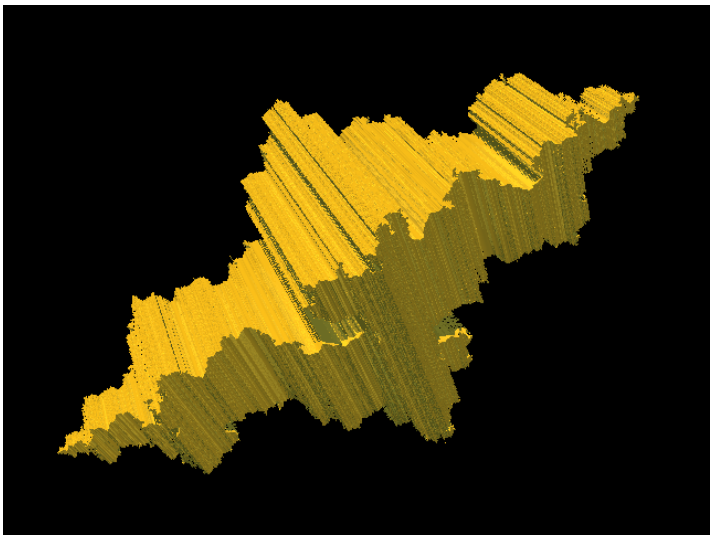


Fig.4 - POV 3.7 $3D$ Julia set ($c = -0.745429 + i0.113089 + j0.113089$) as a whole (by "Julia_fractal" function [hypercomplex])

POV-Ray 3.7 code to generate fig 4

```
//====================================================================
// Julia 3D set ($c=-0.745429+i0.113089+j0.113089$) as a whole
// (POV-Ray Julia_fractal function [hypercomplex])
//====================================================================

#include "colors.inc"     // Standard Color definitions
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }      // set black background

camera { // set view point (camera) location
perspective
location < 1.0, 1.0, -3>
right x * 1
```

```
up y * 3/4
angle 50
look_at < 0.0, 0.0, 0.0>
}

light_source { // set point light sources location
< 0.0, 15, -10>
rgb <1.000000, 0.500000, 0.000000> * 2.0
}

julia_fractal {        //  apply julia_fractal fucntion
  <-0.745429,0.113089,0.113089,0.0>
  hypercomplex
  sqr
  max_iteration 15
  precision 80
  radiosity { importance 1.0 }
  pigment {BrightGold scale 1}
   finish {
      ambient .1
      diffuse .5
      reflection .5
      specular .5
      metallic 2
    }
 rotate x*(-10)
 rotate y*150
 rotate z*30}
```

6.2 Point by point generation of $3D$ quaternion fractal sets

Starting from the present section we abandon both a cylindrical symmetry approach to $3D$ fractals and the *built-in* "Julia_fractal" function, which provide final stages of structures *as wholes* without revealing the different stages of the process according to which the images themselves are produced. Being interested, on the contrary, in viewing in detail the entire process of generation of the ordered structures, we need, as we have seen in the previous chapters, to build each $3D$ fractal following a *point by point* image generation method. Of course a longer machine time will be required[2]
which will increase according to power 3, being 3 the space dimensions to be scanned. Each one by a sequential cycle, like, *e.g,* a *for-next loop.* We will consider, following our usual approach, both *point by point* quaternion and respectively hypercomplex fractal structures generated:

1. either as *wholes*;
2. or by a *sequential* process;
3. or by a *random* process.

[2] A time interval that often may take even several hours, depending on the machine clock speed and memory.

In the following sections we will test what happens employing the recurrence law:

$$Z = Z^2 + C, \tag{6.10}$$

where, now Z, C involve only three parts of *quaternions*:

$$Z = X + iY + jZ, \qquad C = C_X + iC_Y + jC_Z, \tag{6.11}$$

since the ordinary space has only three dimensions. Taking into account the rules previously established we have:

$$Z^2 = X^2 - Y^2 - Z^2 + 2\,iXY + 2jXZ, \tag{6.12}$$

and the resulting recurrence rules:

$$\begin{aligned} X_{n+1} &= X_n^2 - Y_n^2 - Z_n^2 + C_X, \\ Y_{n+1} &= 2X_nY_n + C_Y, \\ Z_{n+1} &= 2X_nZ_n + C_Z. \end{aligned} \tag{6.13}$$

We point out that several combinations are possible since each co-ordinate axis of the space may be scanned either setting the initial value of a co-ordinate (X, Y, Z) equal to zero[3] varying the related C_X, C_Y, C_Z or, on the contrary, setting a constant value for a C component and varying the starting value of the related X, Y, Z co-ordinate.

In the former case the fractal structure will exhibit a Mandelbrot-like behavior along the related co-ordinate axis, while in the latter case it will behave like a Julia set. So the following interesting distinct combinations will arise.[4]

	X_0	Y_0	Z_0	C_X	C_Y	C_Z
Mand.-Mand.-Mand.	0	0	0	var	var	var
Mand.-Mand.-Julia	0	0	var	var	var	const
Mand.-Julia-Julia	0	var	var	var	const	const
Julia-Julia-Julia	var	var	var	const	const	const

We will examine only some of the resulting shapes with their related *POV-Ray 3.7* codes.

6.2.1 Generalized $3D$ quaternion fractals sets as wholes

Mandelbrot-Mandelbrot-Mandelbrot quaternion fractal set

Manifestly the structure of this $3D$ fractal structure exhibits a cylindrical symmetry as it is better shown by sections along the co-ordinate planes.

[3] Or to any other stated value.

[4] The remaining combinations like *Julia-Mand-Mand, Julia-Julia-Mand,* etc. result simply by co-ordinate axes rotation.

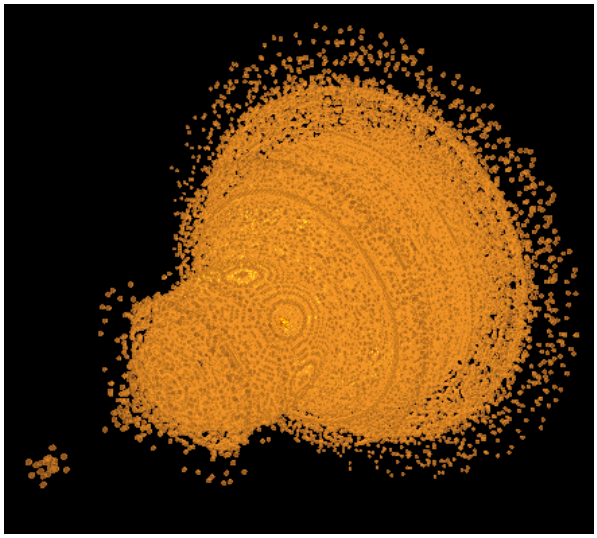


Fig.5 - POV 3.7 3*D* Quaternion Mandelbrot-Mandelbrot-Mandelbrot set as a whole
(point by point generation)

POV-Ray 3.7 code to generate figs 5, 6 and 7

```
//================================================
// Quaternion 3D Mand-Mand-Mand set as a whole
// (POV-Ray point by point generation)
//================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"     // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }      // set black background

camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
}

light_source { // set point light sources location
< -20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 3.0
}

#declare R =  .1;    // set radius value
#declare L = 2;      // set square area side
#declare n = 200;     // set number of pixels per area side
```

Fig.6 - POV 3.7 Open section of a $3D$ quaternion Mandelbrot-Mandelbrot-Mandelbrot set as a whole (point by point generation)

```
#declare st = 1;    // set increment step
#declare N = 40;     // set number of cycles
#declare Th = -45;    //  set rotation angles
#declare Ph = 30;

union{ #for (p, -n, n, st)  //  replace   n  by  n*clock for animation
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -n, n, st)
       #declare IncY = q*L/n;    // Mand1 Y1 increment

    #for (r, -n, n, st)  // alternative     #for (r, 0, n, st)
      #declare IncZ = r*L/n;  // Mand2 Y2 increment

      #declare X = 0;   // start MandX
      #declare Y = 0;   // start MandY
      #declare Z = 0;   // start MandZ

     #for (k,1,N)
       #declare XX = X*X - Y*Y - Z*Z + IncX;  // cycle MandX
       #declare YY = 2*X*Y + IncY;  // cycle MandY
       #declare ZZ = 2*X*Z + IncZ;   // cycle MandZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;
       #declare W = X*X +Y*Y + Z*Z;

   #if ( W > R)       // escape if
      #if (W < R + 0.0015)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }}
```

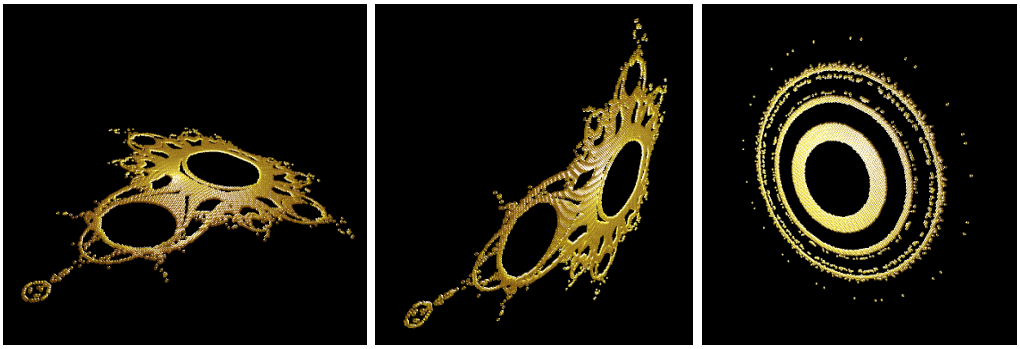


Fig.7 - Cartesian plane sections of the POV 3.7 3D Quaternion Mandelbrot-Mandelbrot-Mandelbrot set as a whole
(point by point generation)

```
       finish { ambient rgb <0.3,0.1,0.1>
       diffuse .3
      reflection .3
      specular 1 }   // plot sphere
   }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 30, 15, 0 >
rotate < 0, Th, Ph > }
```

Julia-Julia-Julia quaternion fractal sets

Here is a quaternion *Julia-Julia-Julia* set ($C = -0.7454294$) which exhibits a cylindrical symmetry the C parameter being a real number.

POV-Ray 3.7 code to generate fig 8, 9 and 10

```
//=========================================================
// Quaternion 3D Julia-Julia-Julia set ($C = -0.7454294$)
// as a whole (POV-Ray point by point generation)
//=========================================================

#include "colors.inc"     // Standard Color definitions
#include "glass.inc"      // Glass pigment effect
#include "metals.inc"    // Metal pigment effect
global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }       // set black background
```

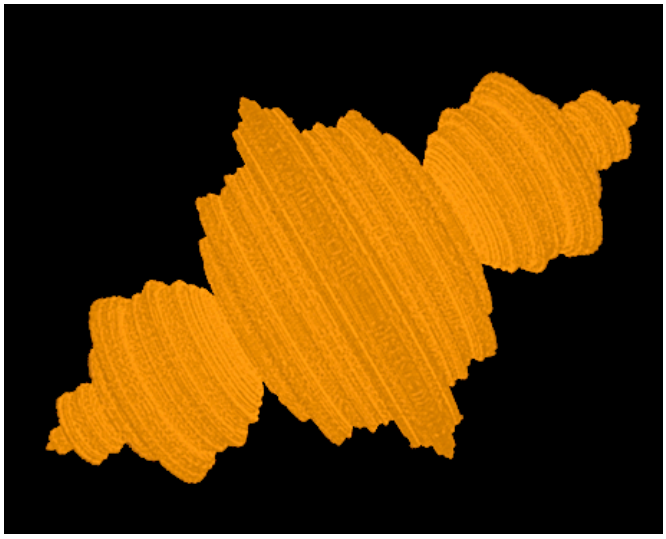


Fig.8 - POV 3.7 $3D$ Quaternion Julia-Julia-Julia set ($C = -0.7454294$) set as a whole (point by point generation)

```
 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
}

light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st = 1;    // set increment step
#declare Nr = 30;   // set number of cycles
#declare Th = 0;    //  set rotation angles
#declare Ph = 30;

#declare Cx  = -0.7454294;   // JuliaX parameter
#declare Cy = 0;  // JuliaY parameter
#declare Cz  = 0;   // JuliaZ parameter

union{
   #for (p, -n, n, st)  //  replace  n by n*clock for animation
     #declare IncX = p*L/n;   // JuliaX increment

    #for (q, -n, n, st)
    #declare IncY = q*L/n;    // JuliaY increment
```

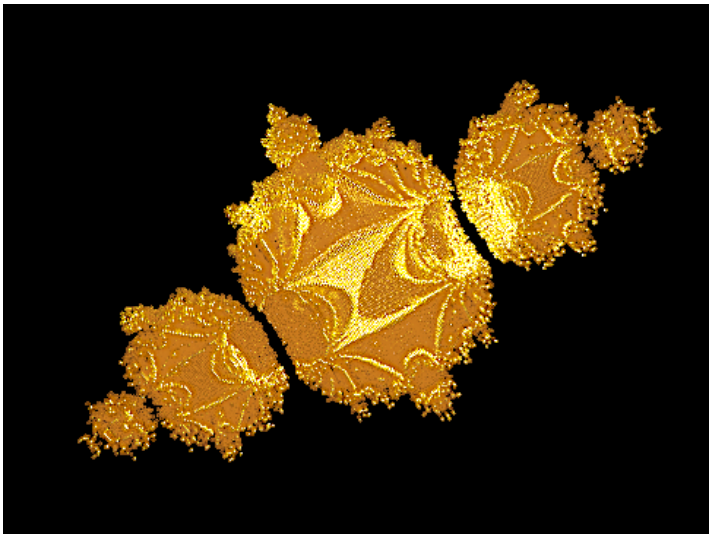


Fig.9 - POV 3.7 Open section of a $3D$ quaternion Julia-Julia-Julia set ($C = -0.7454294$) as a whole (point by point generation)

```
    #for (r, -n, n, st)  // alternative   #for (r, 0, n, st)
    #declare IncZ = r*L/n;  // JuliaZ increment

      #declare X = IncX;   // start JuliaX
      #declare Y = IncY;   // start JuliaY
      #declare Z = IncZ;   // start  JuliaZ

      #for (k,1,Nr)
      #declare XX = X*X - Y*Y - Z*Z + Cx;   // cycle JuliaX
     #declare YY = 2*X*Y + Cy;              // cycle JuliaY
     #declare ZZ = 2*X*Z + Cz;              // cycle JuliaZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;
       #declare W = X*X +Y*Y + Z*Z;
   #if ( W > R)       // escape if
      #if (W < R + 0.02)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
}
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
```

Fig.10 - Cartesian plane sections of the POV 3.7 $3D$ Quaternion Julia-Julia-Julia set ($C = -0.7454294$) as a whole (point by point generation)

```
#end  // end for r
translate < 0, -10, 0 >
rotate < 0, Th, Ph > }
```

And here is the structure of a d generalized quaternion "sea-horse" Julia set characterized by the parameters $c_1 = c_2 = -0.7454294 + i0.113089$ (no cylindrical symmetry).

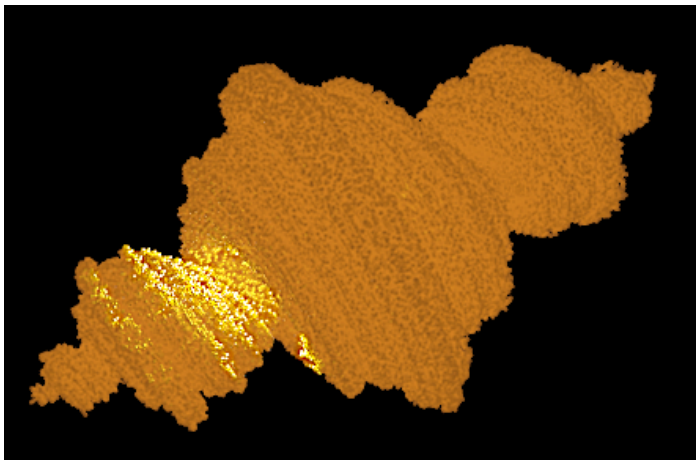


Fig.11 - POV 3.7 $3D$ Quaternion "sea-horse" Julia-Julia-Julia set ($C = -0.7454294 + i0.113089 + j0.00642$) as a whole (point by point generation)

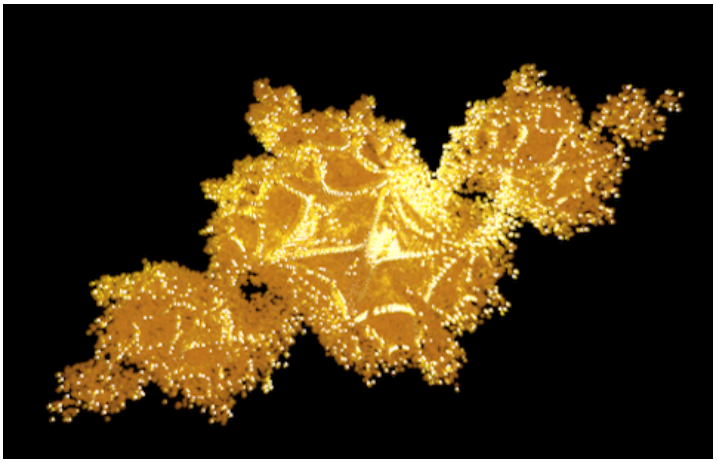


Fig.12 - POV 3.7 Open section of a $3D$ quaternion "sea-horse" Julia-Julia-Julia set
$(C = -0.7454294 + i0.113089 + j0.00642)$ as a whole
(point by point generation)

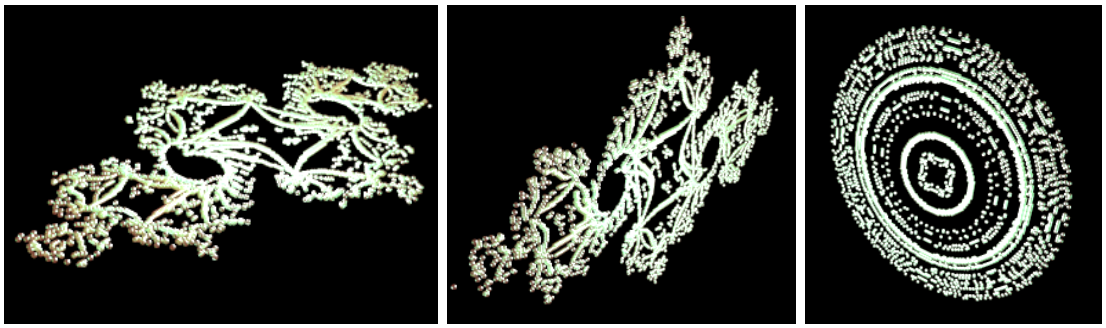


Fig.13 - Cartesian plane sections of the POV 3.7 $3D$ Quaternion "sea-horse" Julia-Julia-Julia set
$(C = -0.7454294 + i0.113089 + j0.00642)$ as a whole
(point by point generation)

POV-Ray 3.7 code to generate fig 11, 12 and 13

```
//=====================================================
// Quaternion 3D Julia-Julia-Julia set
// ($C = -0.7454294 + i 0.113089 + j 0.00642$)
// as a whole (POV-Ray point by point generation)
//=====================================================

#include "glass.inc"     // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }      // set black background

camera { // set view point (camera) location
```

```
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
}

light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

#declare R =  .1;    // set radius value
#declare L = 2;      // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st1 = 1;      // set increment steps
#declare st2 = 3;
#declare st3 = 3;
#declare Nr = 30;     // set number of cycles
#declare Th = 0;    //  set rotation angles
#declare Ph = 30;
#declare Cx  = -0.7454294;   // JuliaX parameter
#declare Cy = 0.113089;      // JuliaY parameter
#declare Cz  = 0.00642;        // JuliaZ parameter

union{
 //  replace  n by n*clock for animation
   #for (p, -n, n, st1)
     #declare IncX = p*L/n;   // JuliaX increment

    #for (q, -n, n, st2)
    #declare IncY = q*L/n;    // JuliaY increment

    #for (r, -n, n, st3)  // alternative #for (r, 0, n, st3)
    #declare IncZ = r*L/n;  // JuliaZ increment
    #declare X = IncX;   // start JuliaX
    #declare Y = IncY;   // start JuliaY
    #declare Z = IncZ;   // start  JuliaZ

  #for (k,1,Nr)
     #declare XX = X*X - Y*Y - Z*Z + Cx;   // cycle JuliaX
     #declare YY = 2*X*Y + Cy;              // cycle JuliaY
     #declare ZZ = 2*X*Z + Cz;              // cycle JuliaZ
     #declare X = XX;
     #declare Y = YY;
     #declare Z = ZZ;
     #declare W = X*X +Y*Y + Z*Z;

   #if ( W > R)       // escape if
      #if (W < R + 0.001)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
    }
```

```
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 0, -10, 0 >
rotate < -20, Th, Ph > }
```

Julia-Mandelbrot-Mandelbrot quaternion fractal set

Among all the possible combinations of generalized Mandelbrot and Julia sets an elegant one is obtained assigning a constant value to only one of components of the parameter C, say C_X, varying the remaining C_Y, C_Z and the co-ordinate X.

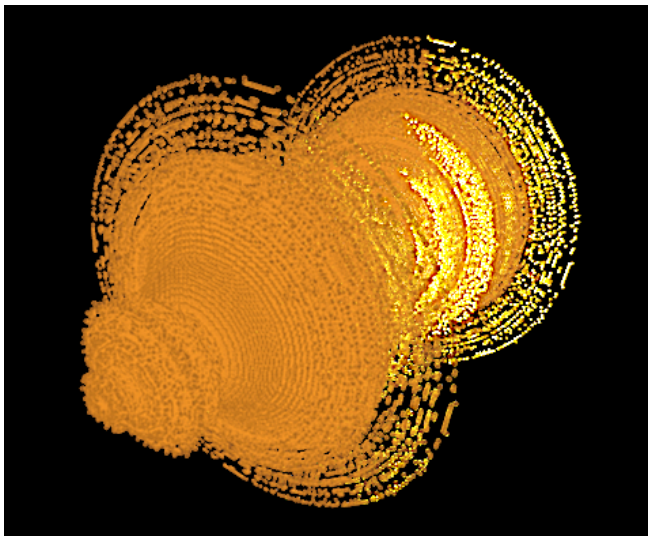


Fig.14 - POV 3.7 3D Quaternion Julia-Mandelbrot-Mandelbrot set as a whole
(point by point generation)

POV-Ray 3.7 code to generate figs 14, 15 and 16

```
//=================================================
// Quaternion 3D Julia-Mand-Mand set
// (C = -0.7454294, Y(0) = 0, Z(0) = 0) as a whole
// (POV-Ray point by point generation)
//=================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

global_settings {assumed_gamma 1.0}   // set display gamma
```

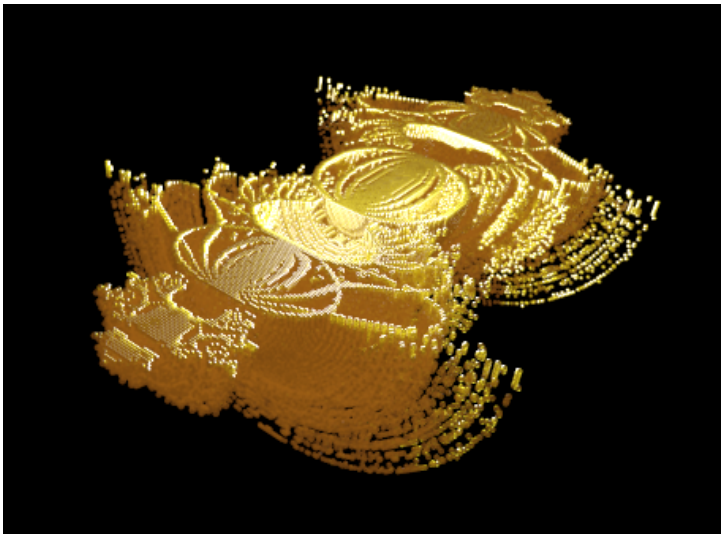


Fig.15 - POV 3.7 Open section of a $3D$ quaternion Mandelbrot-Mandelbrot-Mandelbrot set as a whole (point by point generation)

```
background { color rgb <00,0.0,0.0> }      // set black background

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 30, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 3.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st1 = 1;      // set increment steps
#declare st2 = 1;
#declare st3 = 1;
#declare N = 20;     // set number of cycles
#declare Th = -45;    //  set rotation angles
#declare Ph = 30;

#declare Cx = -0.7454294;  // set C parameter values

union{ //  replace  n by n*clock for animation
     #for (p, -n, n, st1)
     #declare IncX = p*L/n;   // Julia X increment
```

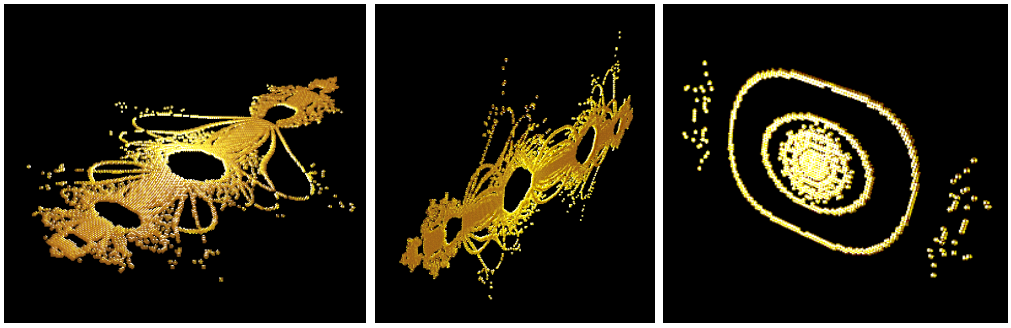


Fig.16 - Cartesian plane sections of the POV 3.7 3D Quaternion Julia-Mandelbrot-Mandelbrot set ($C_X = -0.7454294, Y_0 = 0, Z_0 = 0$) as a whole (point by point generation)

```
    #for (q, -n, n, st2)
       #declare IncY = q*L/n;    // Mand Y increment

    #for (r, -n, n, st3)    //  replace  n by n*clock for animation
      #declare IncZ = r*L/n;  // Mand Z increment

       #declare X = IncX;   // start JuliaX
       #declare Y = 0;   // start MandY
       #declare Z = 0;   // start MandZ

     #for (k,1,N)
       #declare XX = X*X - Y*Y - Z*Z + Cx;  // cycle JuliaX
       #declare YY = 2*X*Y + IncY;  // cycle MandY
       #declare ZZ = 2*X*Z + IncZ;   // cycle MandZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;
       #declare W = X*X +Y*Y + Z*Z;

   #if ( W > R)       // escape if
      #if (W < R + 0.01)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 30, 15, 10 >
rotate < Th, Th, Ph > }
```

6.2.2 Generalized $3D$ quaternion fractals generated sequentially

In the present section we show four steps of the ordered sequential generation of each kind of the generalized combinations of the $3D$ *Mandelbrot* and *Julia* sets examined previously. In the related *POV-Ray 3.7* codes the number of points of each step is denoted by ns.

Mandelbrot-Mandelbrot-Mandelbrot quaternion fractal set

We remember that we have denoted as *Mandelbrot-Mandelbrot-Mandelbrot* $3D$ fractal set obtained starting from the initial values $X_0 = Y_0 = Z_0 = 0$ and variable increments C_x, C_y, C_z.

POV-Ray 3.7 code to generate figs 17 and 18

```
//=========================================================
// Quaternion 3D Mand-Mand-Mand set generated sequentially
// (POV-Ray point by point generation)
//=========================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

global_settings {assumed_gamma 1.0}   // set display gamma
background { color rgb <00,0.0,0.0> }      // set black background

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 3.0
 }

#declare R =  .1;   // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare ns = 200;  // partial values ns = 50, 75, 100
#declare st = 1;
#declare N = 40;  // N = 40
#declare Th = -45;
#declare Ph = 30;

//  replace  ns by ns*clock for animation
 union{ #for (p, -ns, ns, st)
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -ns, ns, st)
       #declare IncY = q*L/n;    // Mand1 Y1 increment

    #for (r, -ns, ns, st) // alternative #for (r, 0, ns, st) [open sets]
```

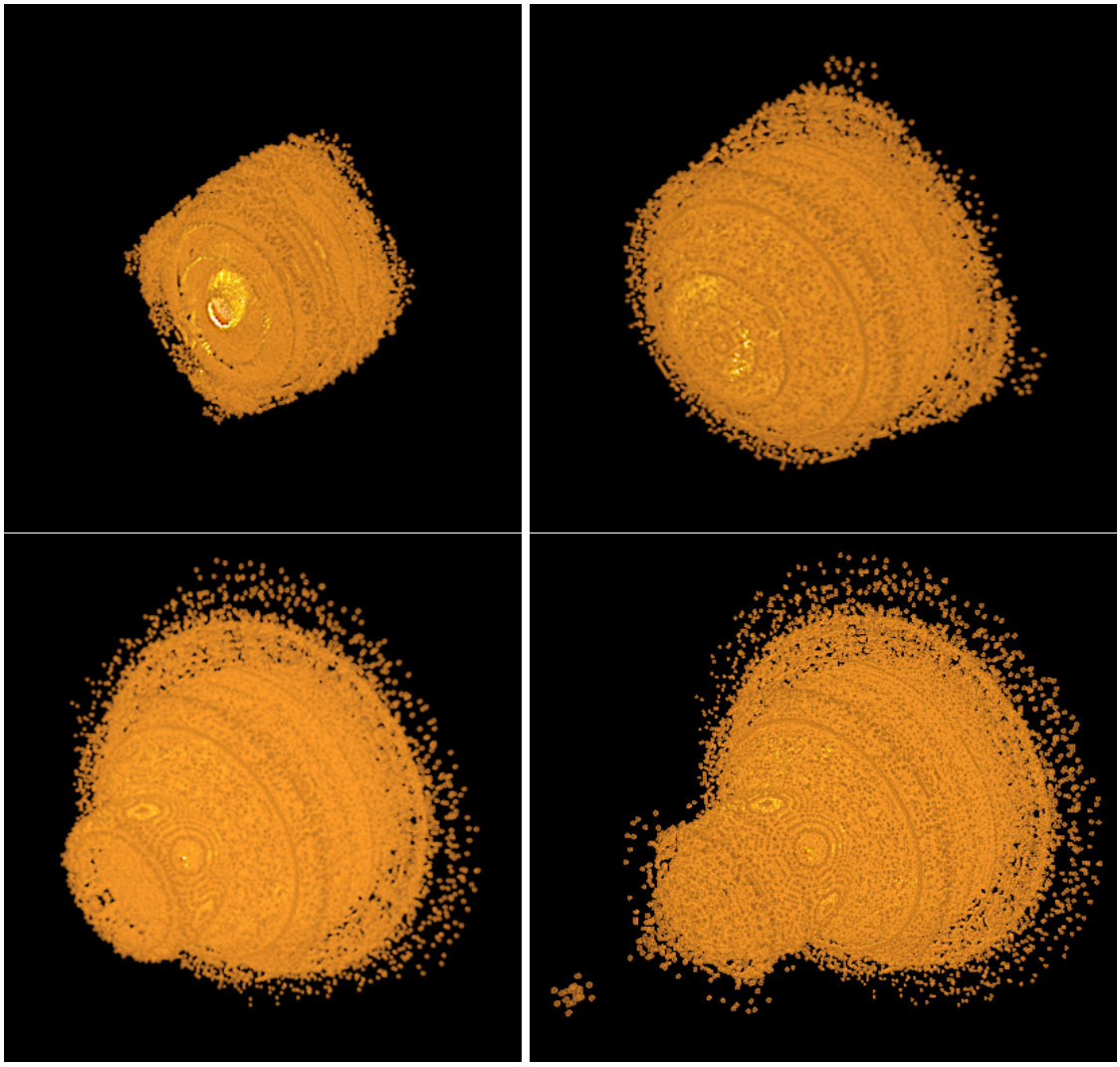


Fig.17 - POV 3.7 3*D* Quaternion Mandelbrot-Mandelbrot-Mandelbrot set generated sequentially

VIEW ANIMATION (requires internet connection)

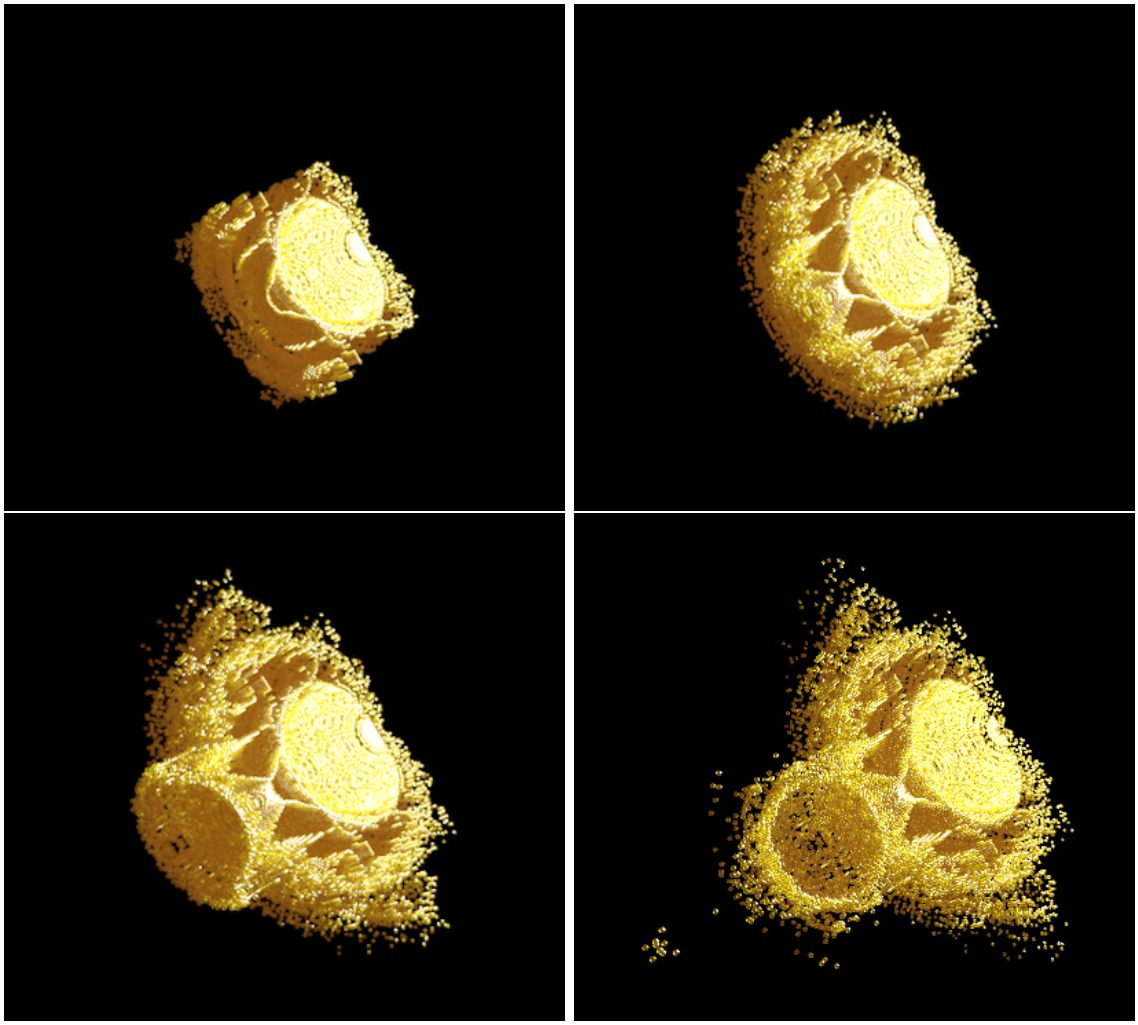


Fig.18 - POV 3.7 3*D* Open section of a quaternion Mandelbrot-Mandelbrot-Mandelbrot set generated sequentially

```
      #declare IncZ = r*L/n;  // Mand2 Y2 increment

      #declare X = 0;   // start MandX
      #declare Y = 0;   // start MandY
      #declare Z = 0;   // start MandZ

     #for (k,1,N)
       #declare XX = X*X - Y*Y - Z*Z + IncX;  // cycle MandX
       #declare YY = 2*X*Y + IncY;  // cycle MandY
       #declare ZZ = 2*X*Z + IncZ;   // cycle MandZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;
       #declare W = X*X +Y*Y + Z*Z;

   #if ( W > R)       // escape if
      #if (W < R + 0.0015)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 30, 15, 0 >
rotate < 0, Th, Ph > }
```

Julia-Julia-Julia quaternion fractal set ($C = -0.7454294$)

The second example is provided by a *Julia-Julia-Julia* quaternion fractal set obtained assigning to the parameters $C = C_x + iC_y + jC_z$ the real value $C = -0.7454294$.

POV-Ray 3.7 code to generate figs 19 and 20

```
//=========================================================
// Quaternion 3D Julia-Julia-Julia set ($C = -0.7454294$)
// generated sequentially
// (POV-Ray point by point generation)
//=========================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"
#include "metals.inc"

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
```

```
    look_at  <-5, 0,  0>
  }



   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;      // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st = 1;
#declare Nr = 30;  // Nr = 40
#declare Th = 0;
#declare Ph = 30;

#declare Cx  = -0.7454294;   // JuliaX parameter
#declare Cy = 0;  // JuliaY parameter
#declare Cz  = 0;   // JuliaZ parameter

 union{
 //  replace  n by n*clock for animation
   #for (p, -n, n, st)
     #declare IncX = p*L/n;   // JuliaX increment

    #for (q, -n, n, st)
    #declare IncY = q*L/n;    // JuliaY increment

    #for (r, -n, n, st)
    #declare IncZ = r*L/n;  // JuliaZ increment

      #declare X = IncX;   // start JuliaX
      #declare Y = IncY;   // start JuliaY
      #declare Z = IncZ;   // start  JuliaZ

      #for (k,1,Nr)
      #declare XX = X*X - Y*Y - Z*Z + Cx;   // cycle JuliaX
     #declare YY = 2*X*Y + Cy;              // cycle JuliaY
     #declare ZZ = 2*X*Z + Cz;              // cycle JuliaZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;

       #declare W = X*X +Y*Y + Z*Z;

   #if ( W > R)       // escape if
      #if (W < R + 0.02)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
```

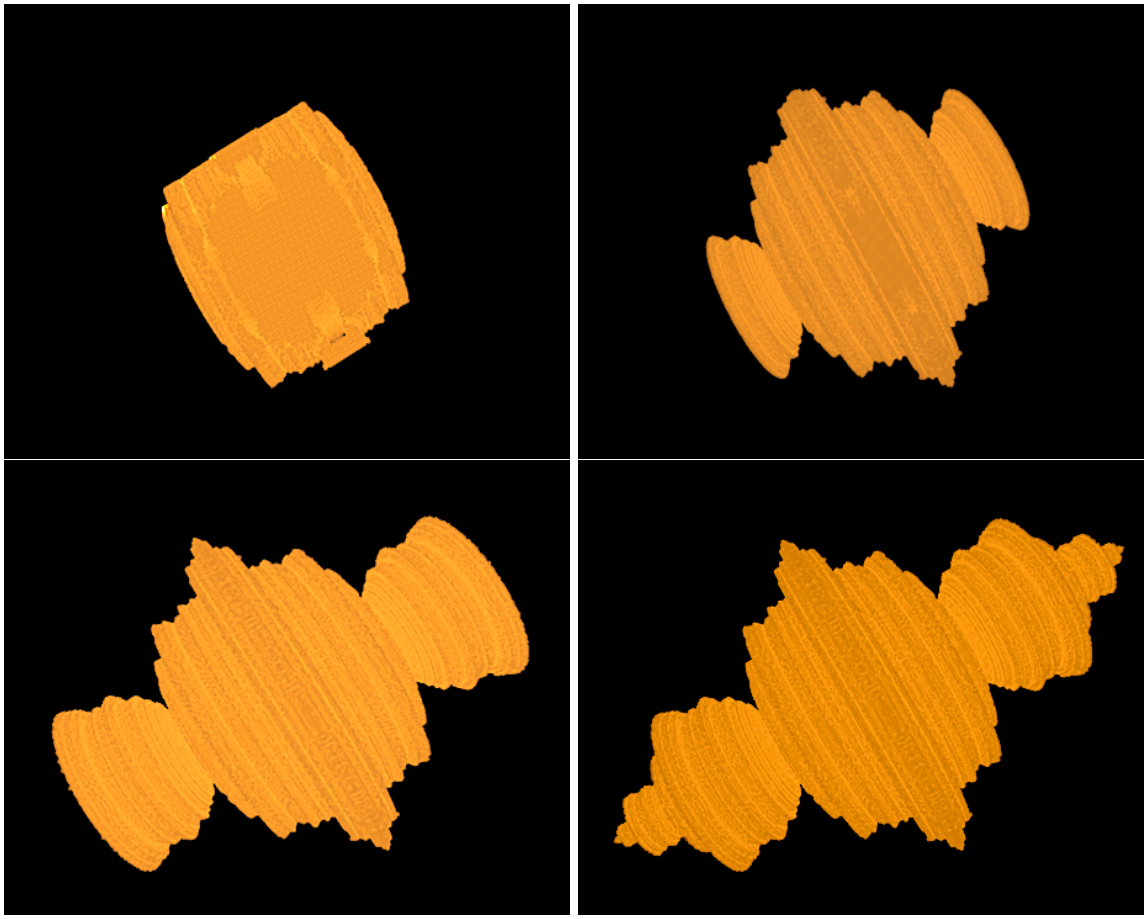


Fig.19 - POV 3.7 $3D$ Quaternion Julia-Julia-Julia set ($C = -0.7454294$) generated sequentially

VIEW ANIMATION (requires internet connection)

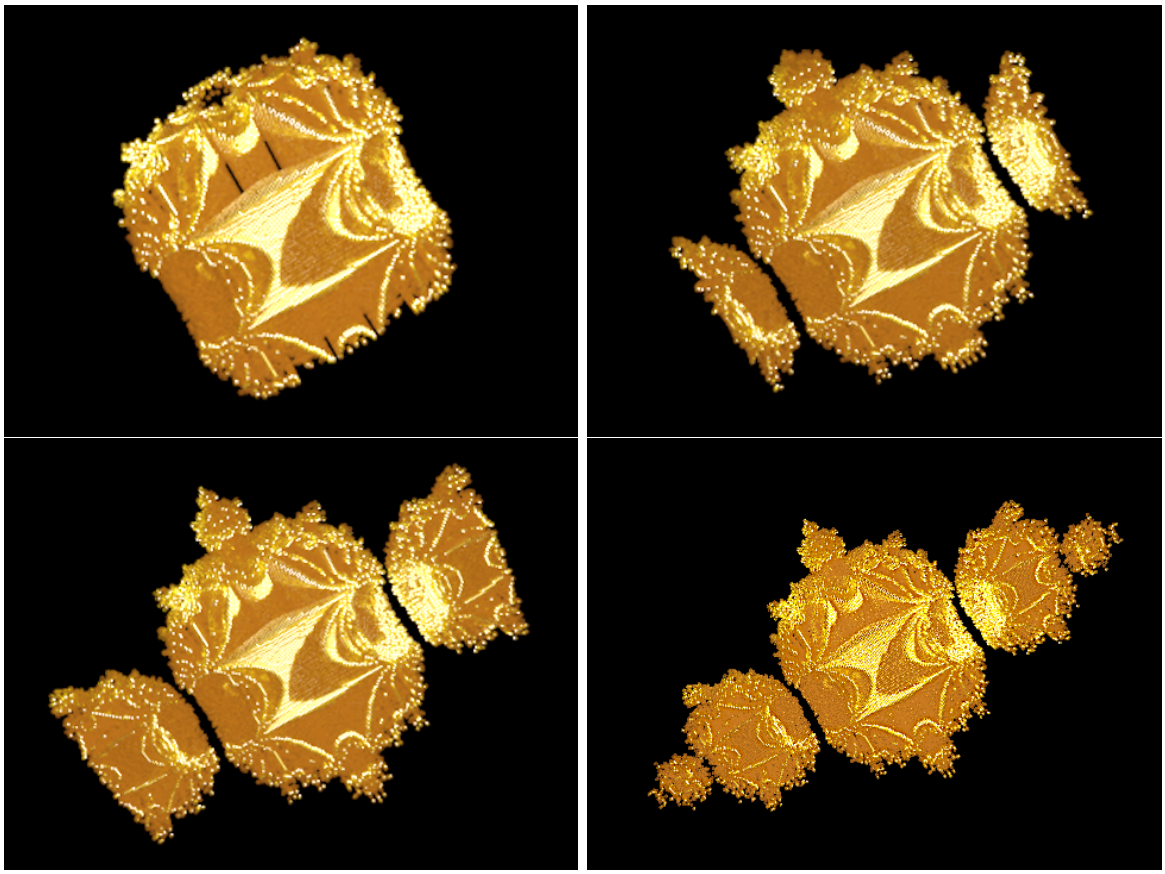


Fig.20 - POV 3.7 $3D$ open section of a quaternion Julia-Julia-Julia set
($C = -0.7454294$) generated sequentially

```
      specular 1 }   // plot sphere
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 0, -10, 0 >
rotate < 0, Th, Ph > }
```

Julia-Julia-Julia "sea-horse" quaternion fractal set ($C = -0.7454294 + i0.113089 + j0.00642$)

A third example is rperesented by a gemeralized $3D$ *"sea-horse" Julia-Julia-Julia* quaternion fractal set obtained assigning to the parameters $C = -0.7454294 + i0.113089 + j0.00642$, which does not exhibit a cylindrical symmetry.

POV-Ray 3.7 code to generate figs 21 and 22

```
//=================================================================
// Quaternion 3D ‘‘sea-horse’’ Julia-Julia-Julia set
// ($C = -0.7454294 + i 0.113089 + j 0.113089$) generated sequentially
// (POV-Ray point by point generation)
//=================================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare ns = 200;  // partial ns = 50, 75, 100
#declare st1 = 1;    // set increment steps
#declare st2 = 1;
#declare st3 = 1;
#declare Nr = 30;   // set number of cycles
#declare Th = 0;   //  set rotation angles
#declare Ph = 30;
```

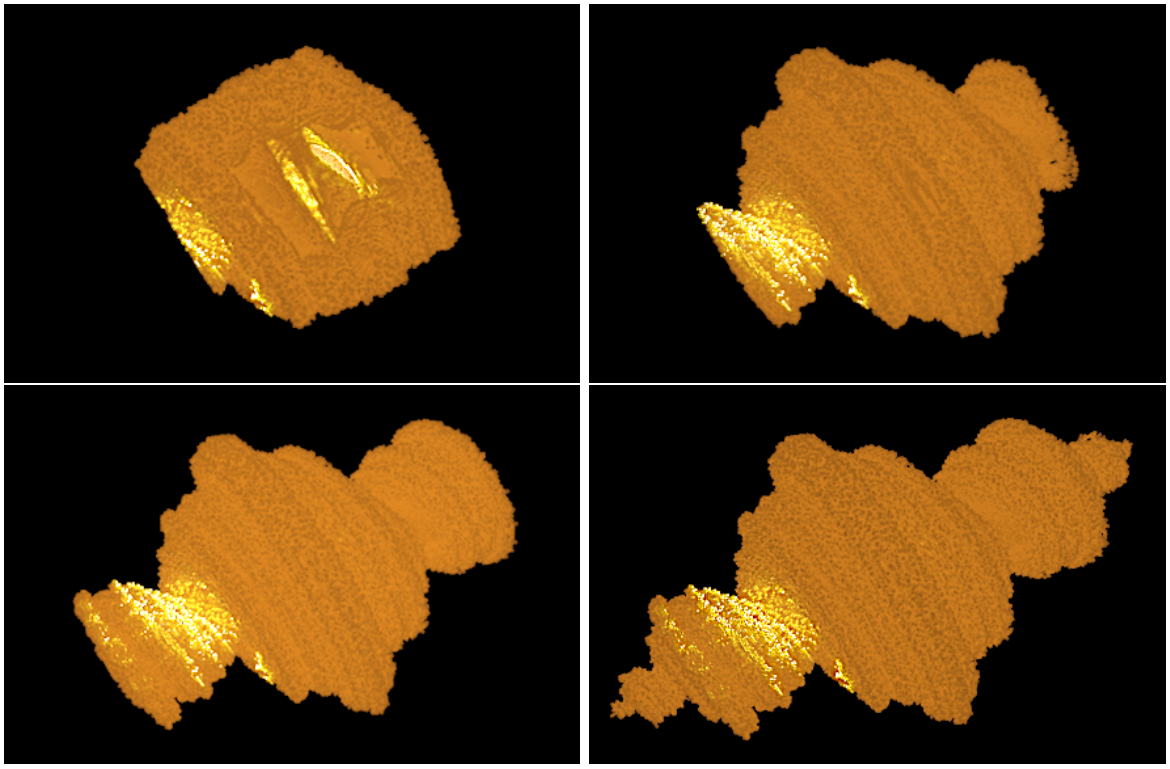


Fig.21 - POV 3.7 $3D$ Quaternion "sea-horse" Julia-Julia-Julia set ($C = -0.7454294 + i0.113089 + j0.00642$) generated sequentially

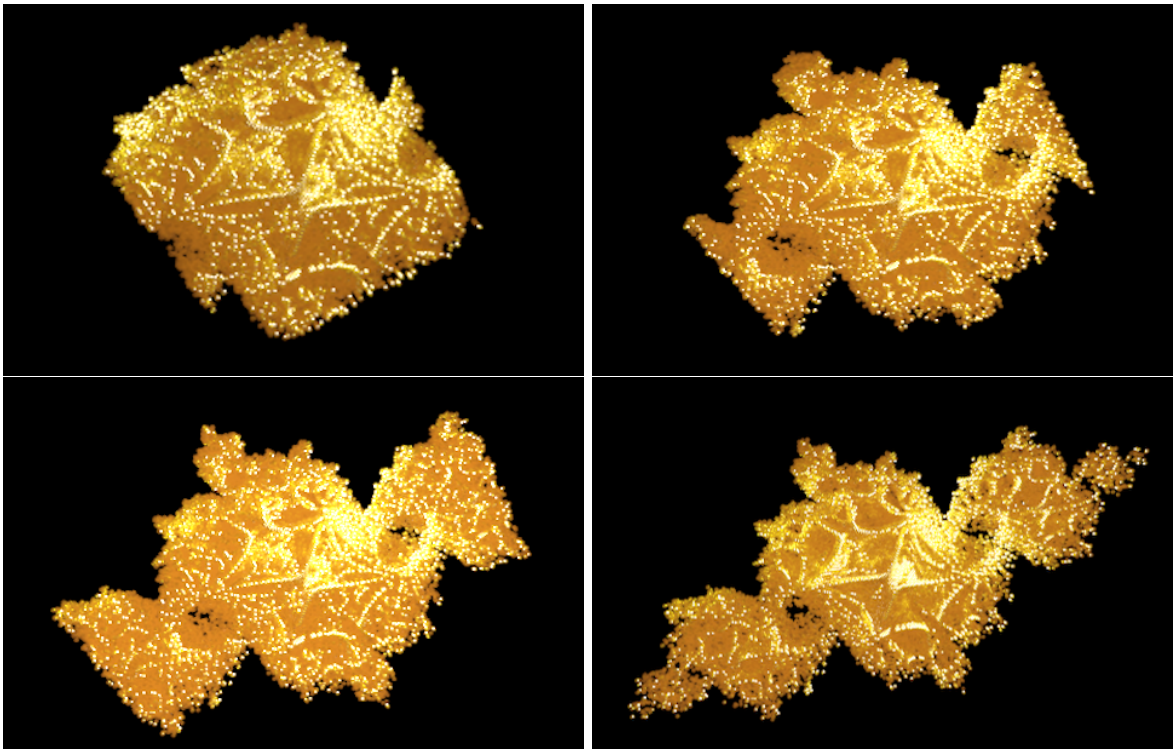


Fig.22 - POV 3.7 $3D$ open section of a quaternion "sea-horse" Julia-Julia-Julia set $(C = -0.7454294 + i0.113089 + j0.00642)$ generated sequentially

```
#declare Cx  = -0.7454294;   // JuliaX parameter
#declare Cy = 0.113089;  // JuliaY parameter
#declare Cz  =  0.113089;   // JuliaZ parameter

 union{
 //  replace  n by n*clock for animation
   #for (p, -ns, ns, st1)
     #declare IncX = p*L/n;   // JuliaX increment

    #for (q, -ns, ns, st2)
    #declare IncY = q*L/n;    // JuliaY increment

    #for (r, -ns, ns, st3)
    #declare IncZ = r*L/n;  // JuliaZ increment

      #declare X = IncX;   // start JuliaX
      #declare Y = IncY;   // start JuliaY
      #declare Z = IncZ;   // start  JuliaZ

      #for (k,1,Nr)
      #declare XX = X*X - Y*Y - Z*Z + Cx;   // cycle JuliaX
     #declare YY = 2*X*Y + Cy;              // cycle JuliaY
     #declare ZZ = 2*X*Z + Cz;              // cycle JuliaZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;

       #declare W = X*X +Y*Y + Z*Z;

   #if ( W > R)       // escape if
      #if (W < R + 0.0053)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 30, 15, 20 >
rotate < Th, Th, Ph > }
```

Julia-Mandelbrot-Mandelbrot quaternion fractal set ($C = -0.7454294$)

A fourth example is given by a *Julia-Mandelbrot-Mandelbrot* quaternion fractal set which is obtained setting $C_x = -0.7454294$ and variable C_y, C_z.

POV-Ray 3.7 code to generate figs 23 and 24

```
//=======================================================
// Quaternion 3D Julia-Mandelbrot-Mandelbrot fractal set
// ($C = -0.7454294$) generated sequentially
// (POV-Ray point by point generation)
//=======================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 30, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 3.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st1 = 1;    // set increment steps
#declare st2 = 1;
#declare st3 = 1;
#declare N = 20;   // set number of cycles
#declare Th = -45;    //  set rotation angles
#declare Ph = 30;

#declare Cx = -0.7454294;  // JuliaX parameter

 union{
 //  replace  n by n*clock for animation
 #for (p, -n, n, st1)
     #declare IncX = p*L/n;   // Julia X increment

    #for (q, -n, n, st2)
       #declare IncY = q*L/n;    // Mand Y increment

    #for (r, -n, n, st3)
      #declare IncZ = r*L/n;  // Julia Y increment

      #declare X = IncX;   // start JuliaX
      #declare Y = 0;   // start MandY
      #declare Z = 0;   // start MandZ

     #for (k,1,N)
       #declare XX = X*X - Y*Y - Z*Z + Cx;  // cycle JuliaX
       #declare YY = 2*X*Y + IncY;  // cycle MandY
```

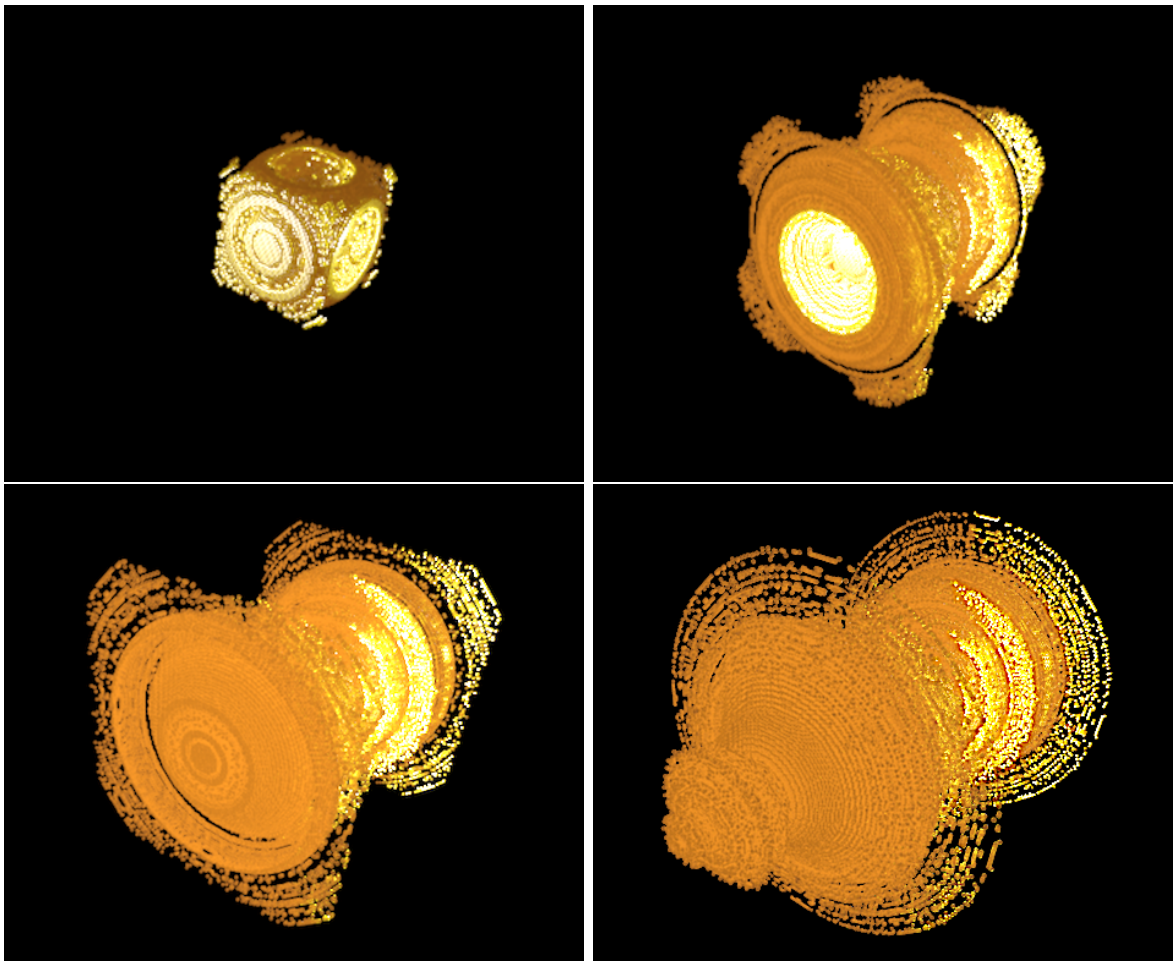


Fig.23 - POV 3.7 $3D$ quaternion Julia-Mandelbrot-Mandelbrot set ($C = -0.7454294$) generated sequentially

VIEW ANIMATION (requires internet connection)

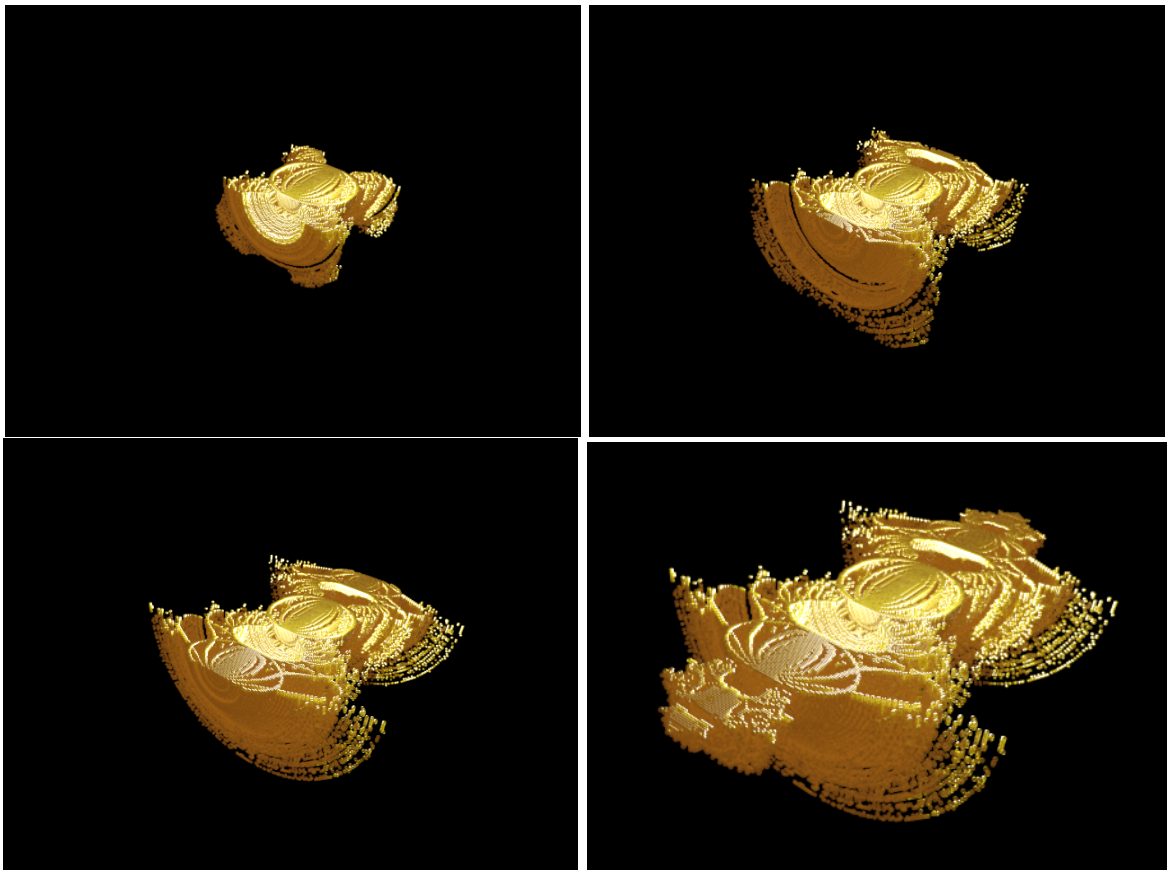


Fig.24 - POV 3.7 $3D$ Open section of a quaternion Julia-Mandelbrot-Mandelbrot set ($C = -0.7454294$) generated sequentially

```
      #declare ZZ = 2*X*Z + IncZ;   // cycle JuliaZ
      #declare X = XX;
      #declare Y = YY;
      #declare Z = ZZ;
      #declare W = X*X +Y*Y + Z*Z;

   #if ( W > R)       // escape if
      #if (W < R + 0.01)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 30, 15, 10 >
rotate < Th, Th, Ph > }
```

6.2.3 Generalized $3D$ quaternion fractal sets generated randomly

The main interest in our investigation on order emerging from random initial conditions driven by information is concerned on randomly generated structures. In this section we show some examples of quaternion fractals, *i.e.,* $3D$ Mandelbrot, Julia sets and mixed ones.

Mandelbrot-Mandelbrot-Mandelbrot quaternion fractal set

POV-Ray 3.7 code to generate figs 25 and 26

```
//============================================================
// Quaternion 3D Mandelbrot-Mandelbrot-Mandelbrot fractal set
//  generated randomly
// (POV-Ray point by point generation)
//============================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
```

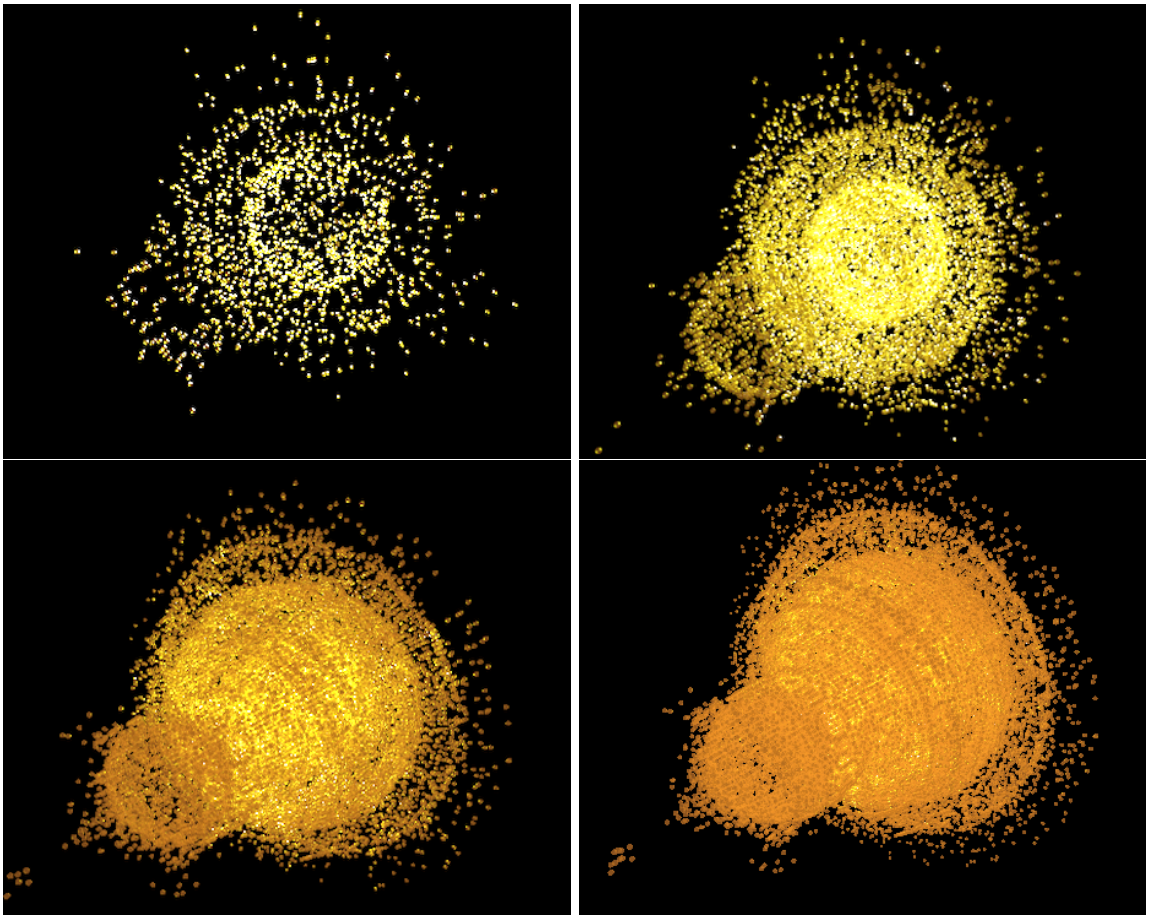


Fig.25 - POV 3.7 $3D$ quaternion Mandelbrot-Mandelbrot-Mandelbrot set generated randomly

VIEW ANIMATION (requires internet connection)

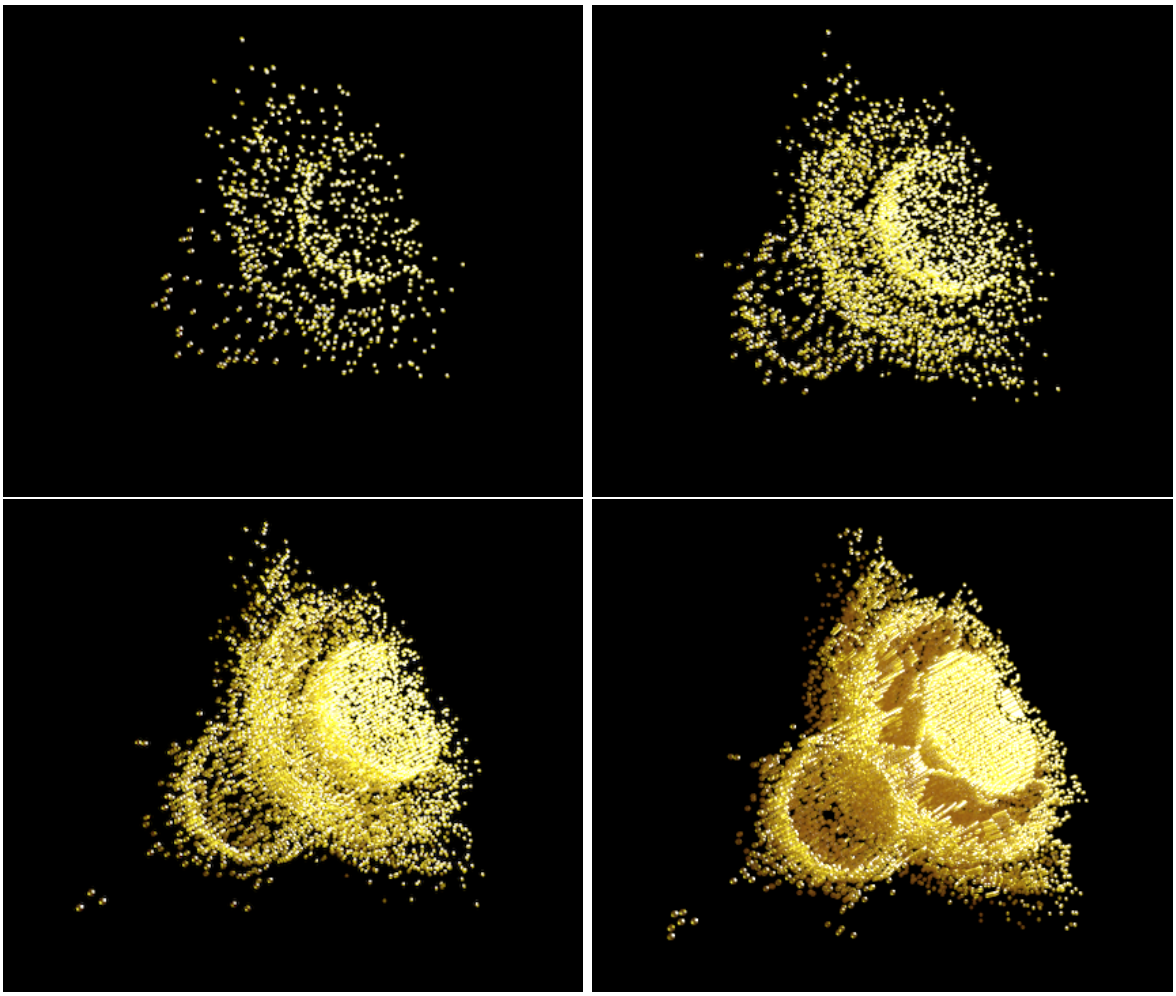


Fig.26 - POV 3.7 3*D* Open section of a quaternion Mandelbrot-Mandelbrot-Mandelbrot set generated randomly

```
< -20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 3.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;      // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st = 1;
#declare N = 40;  // N = 40
#declare Th = -45;
#declare Ph = 30;

#declare K = array[2*n+1][2*n+1][2*n+1];   // escape rate 3D matrix
  #for (p, -n, n, st)
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -n, n, st)
       #declare IncY = q*L/n;    // Mand1 Y1 increment

    #for (r, -n, n, st)
      #declare IncZ = r*L/n;  // Mand2 Y2 increment

      #declare X = 0;   // start MandX
      #declare Y = 0;   // start MandY
      #declare Z = 0;   // start MandZ
      #declare K[p+n][q+n][r+n] = 0;

     #for (k,0,N)
       #declare XX = X*X - Y*Y - Z*Z + IncX;  // cycle MandX
       #declare YY = 2*X*Y + IncY;  // cycle MandY
       #declare ZZ = 2*X*Z + IncZ;   // cycle MandZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;
       #declare W = X*X +Y*Y + Z*Z;

   #if ( W > R)       // escape if
      #if (W < R + 0.0015)     // escape if
       #declare K[p+n][q+n][r+n] = k;
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r

#declare Rnd_1 = seed (1153);
#declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (876);

union{
//  replace n*n*n by  n*n*n*clock for animation
    #for(j,0,n*n*n*64)
   #declare p = int(2*n*rand(Rnd_1));
   #declare q = int(2*n*rand(Rnd_2));
   #declare r = int(2*n*rand(Rnd_3));
    #if(K[p][q][r] > 0)
```

```
    sphere {
     < -n+p, -n+q, -n+r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

translate < 30, 15, 0 >
rotate < 0, Th, Ph > }
#end // end if
#end}  // end for j
```

Julia-Julia-Julia quaternion fractal set ($C = -0.7454294$)

Here is a $3D$ *Julia-Julia-Julia* quaternion fractal set ($C = -0.7454294$) generated randomly.

POV-Ray 3.7 code to generate figs 27 and 28

```
//=========================================================
// Quaternion 3D Julia-Julia-Julia fractal set
//  generated randomly
// (POV-Ray point by point generation)
//=========================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st = 1;
#declare Nr = 30;  // Nr = 40
#declare Th = 0;
#declare Ph = 30;

#declare Cx  = -0.7454294;   // JuliaX parameter
#declare Cy = 0;  // JuliaY parameter
```

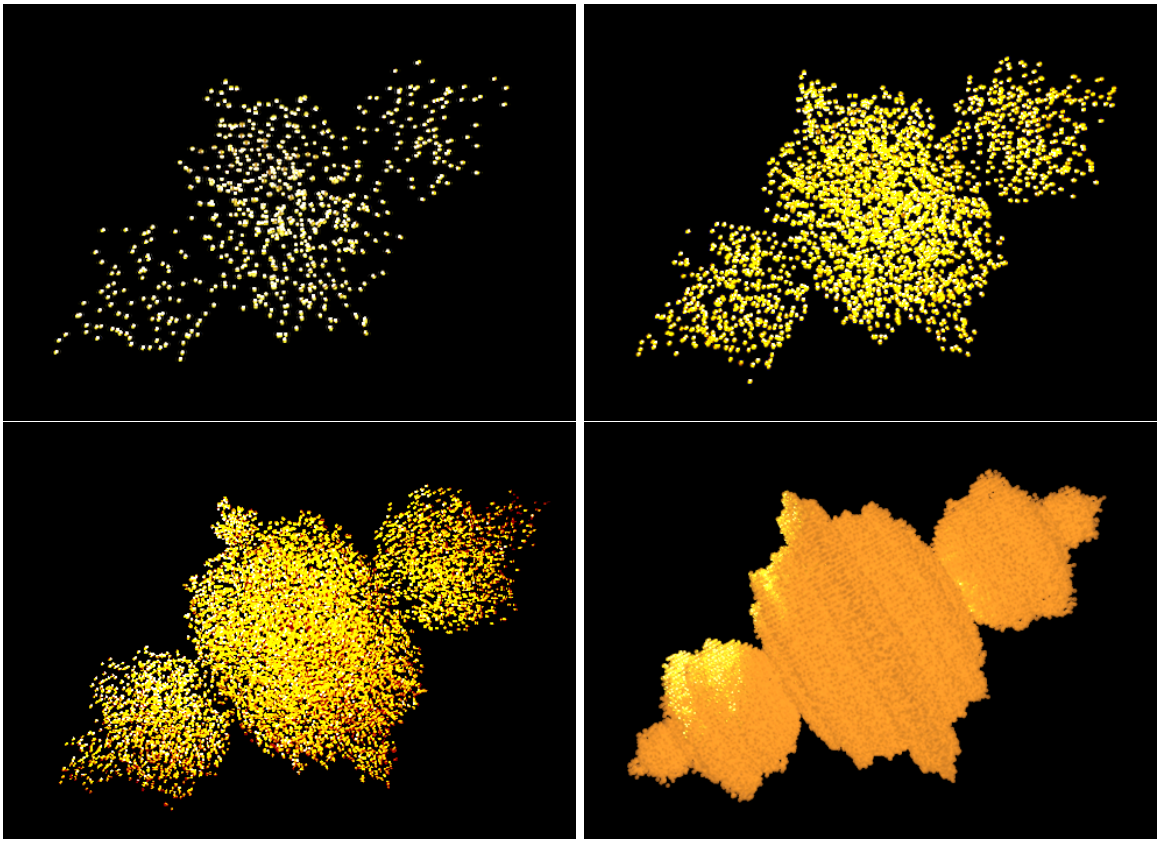


Fig.27 - POV 3.7 $3D$ quaternion Julia-Julia-Julia set ($C = -0.7454294$) generated randomly

VIEW ANIMATION (requires internet connection)

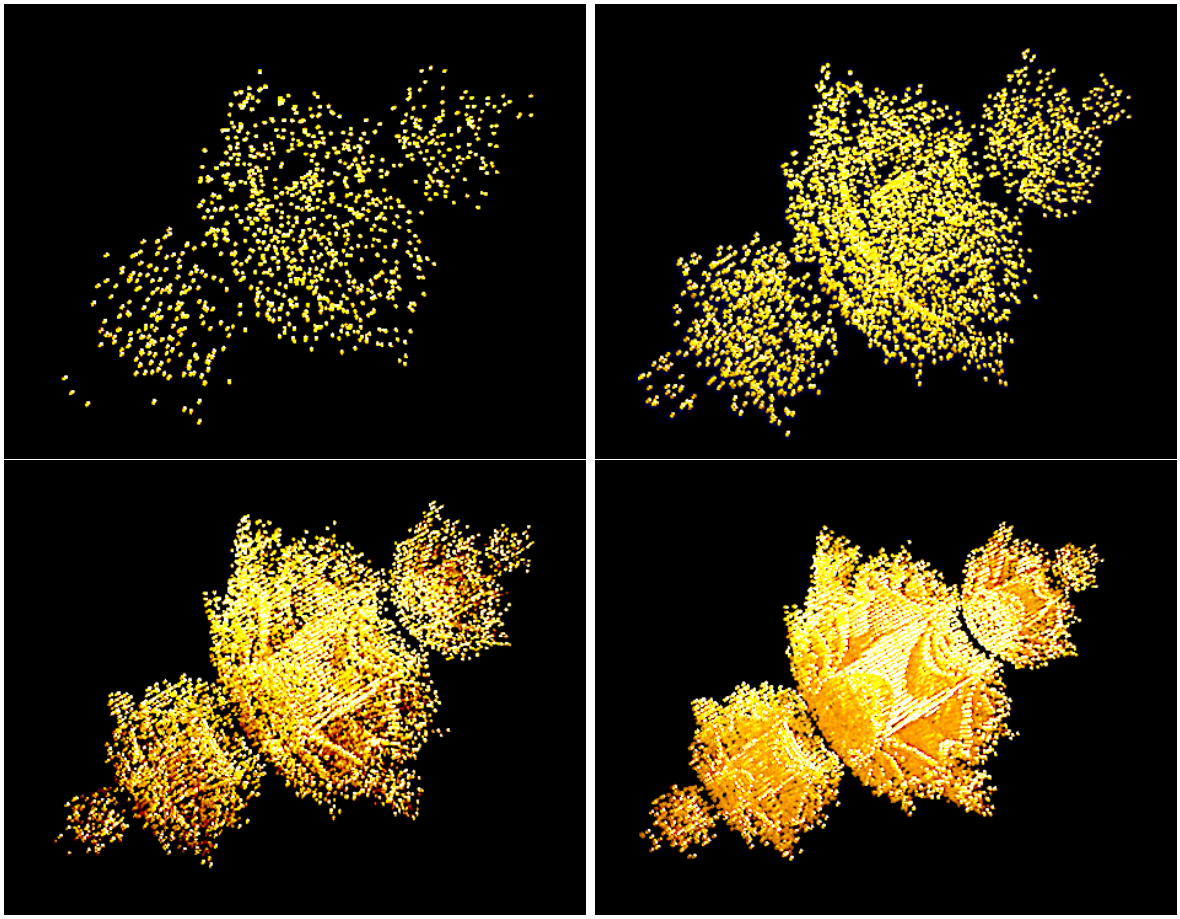


Fig.28 - POV 3.7 $3D$ Open section of a quaternion Julia-Julia-Julia set
($C = -0.7454294$) generated randomly

```
#declare Cz  = 0;   // JuliaZ parameter

#declare K = array[2*n+1][2*n+1][2*n+1];
   #for (p, -n, n, st)
     #declare IncX = p*L/n;   // JuliaX increment

    #for (q, -n, n, st)
    #declare IncY = q*L/n;    // JuliaY increment

    #for (r, -n, n, st)
    #declare IncZ = r*L/n;  // JuliaZ increment

      #declare X = IncX;   // start JuliaX
      #declare Y = IncY;   // start JuliaY
      #declare Z = IncZ;   // start  JuliaZ
     #declare K[p+n][q+n][r+n] = 0;

      #for (k,0,Nr)
      #declare XX = X*X - Y*Y - Z*Z + Cx;   // cycle JuliaX
     #declare YY = 2*X*Y + Cy;              // cycle JuliaY
     #declare ZZ = 2*X*Z + Cz;              // cycle JuliaZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;

       #declare W = X*X +Y*Y + Z*Z;

   #if ( W > R)       // escape if
      #if (W < R + 0.02)     // escape if
    #declare K[p+n][q+n][r+n] = k;
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r

#declare Rnd_1 = seed (1153);
#declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (876);

 union{
//  replace n*n*n by  n*n*n*clock for animation
 #for(j,0,n*n*n)  // alternative   n*n*n/16 n*n*n/64  n*n*n/256
   #declare p = int(2*n*rand(Rnd_1));
   #declare q = int(2*n*rand(Rnd_2));
   #declare r = int(2*n*rand(Rnd_3));
    #if(K[p][q][r] > 0)
    sphere {
     < -n+p, -n+q, -n+r >, 1
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

translate < 0, -10, 0 >
rotate < 0, Th, Ph > }
#end // end if
#end}  // end for j
```

Julia-Mandelbrot-Mandelbrot quaternion fractal set ($C = -0.7454294$)

POV-Ray 3.7 code to generate figs 29 and 30

```
//=========================================================
// Quaternion 3D Julia-Mand-Mand fractal set
// ($C = -0.7454294$) generated randomly
// (POV-Ray point by point generation)
//=========================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }
   light_source { // set point light sources location
< -20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }
 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 3.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare Cx = -0.7454294;
#declare n = 200;     // set number of pixels per area side
#declare st = 1;
#declare N = 40;  // N = 40
#declare Th = -45;
#declare Ph = 30;

#declare K = array[2*n+1][2*n+1][2*n+1];
     #for (p, -n, n, st)
     #declare IncX = p*L/n;   // Mand1 X increment
    #for (q, -n, n, st)
       #declare IncY = q*L/n;    // Mand1 Y1 increment
    #for (r, -n, n, st)
      #declare IncZ = r*L/n;  // Mand2 Y2 increment
      #declare X = IncX;   // start JuliaX
      #declare Y = 0;   // start MandY
      #declare Z = 0;   // start MandZ
      #declare K[p+n][q+n][r+n] = 0;

     #for (k,1,N)
       #declare XX = X*X - Y*Y - Z*Z + Cx;  // cycle JuliaX
       #declare YY = 2*X*Y + IncY;  // cycle MandY
       #declare ZZ = 2*X*Z + IncZ;   // cycle MandZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;
       #declare W = X*X +Y*Y + Z*Z;
```

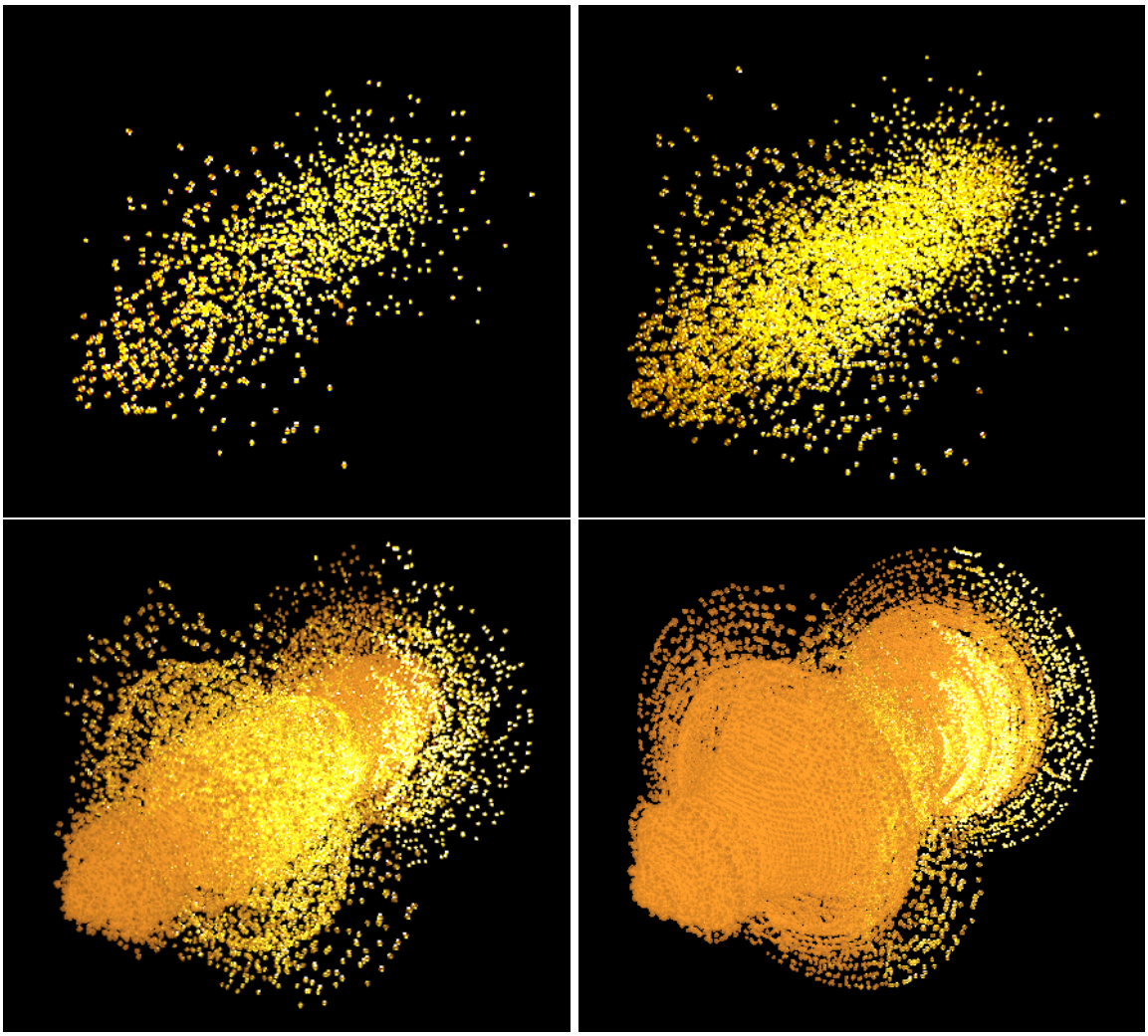


Fig.29 - POV 3.7 $3D$ quaternion Julia-Mand-Mand set ($C = -0.7454294$)
generated randomly

VIEW ANIMATION (requires internet connection)

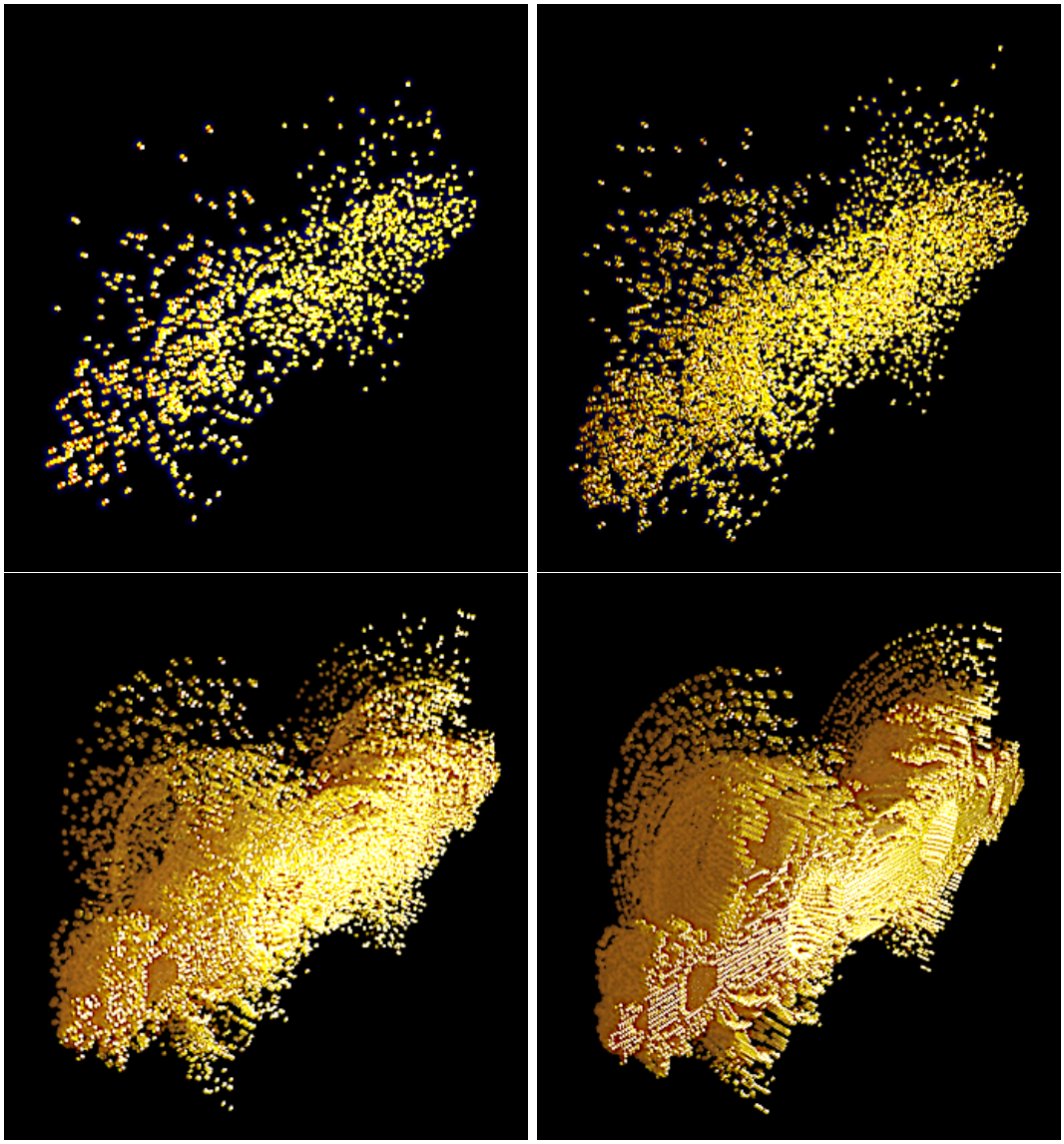


Fig.30 - POV 3.7 $3D$ Open section of a quaternion Julia-Mand-Mand set
($C = -0.7454294$) generated randomly

```
   #if ( W > R)       // escape if
      #if (W < R + 0.0015)     // escape if
    #declare K[p+n][q+n][r+n] = k;
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r

#declare Rnd_1 = seed (1153);
#declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (876);

 union{
//  replace n*n*n by  n*n*n*clock for animation
 #for(j,0,n*n*n)  // alternative   n*n*n/16 n*n*n/64  n*n*n/256
   #declare p = int(2*n*rand(Rnd_1));
   #declare q = int(2*n*rand(Rnd_2));
   #declare r = int(2*n*rand(Rnd_3));
    #if(K[p][q][r] > 0)
    sphere {
     < -n+p, -n+q, -n+r >, 1
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

translate < 30, 15, 10 >
rotate < Th, Th, Ph >}
#end // end if
#end}  // end for j
```

6.3 Point by point generation of $3D$ double complex fractal sets

The third main section of the present chapter will be devoted to *double complex* $3D$ generalized fractals. We will follow the same exposition scheme as in the previous sections dedicated to *quaternion* fractals.

1. First of all we consider generalized *Mandelbrot, Julia* and mixed sets as *wholes*.

2. Subsequently we will examine a sequential ordered process of generation of the same structures.

3. And finally we show how those ordered structures may arise according to a random process which assigns the initial conditions by chance.

6.3.1 Generalized $3D$ double complex fractal sets as wholes

Mandelbrot-Mandelbrot-Mandelbrot double complex fractal set

We start considering a generalized $3D$ *Mandelbrot* set obtained varying the parameters $c_1 = c_{1x} + ic_{1y}, c_2 = c_{2x} + ic_{2y}$ along assigned intervals.

POV-Ray 3.7 code to generate figs 31, 32 and 33

```
//=================================================
// Double complex 3D Mand-Mand-Mand set as a whole
// (POV-Ray point by point generation)
//=================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare X1 = 0;
#declare X2 = 0;
#declare Y1 = 0;
#declare Y2 = 0;
#declare n = 200;     // set number of pixels per area side
#declare st = 1;
#declare Nr = 20;  // Nr = 40
#declare Th = -45;
#declare Ph = 30;

 union{ #for (p, -n, n, st)
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -n, n, st)
       #declare IncY1 = q*L/n;    // Mand1 Y1 increment

    #for (r, 0, n, st)
      #declare IncY2 = r*L/n;  // Mand2 Y2 increment

      #declare X1 = 0;   // start Mand1
      #declare X2 = 0;   // start Mand1
```

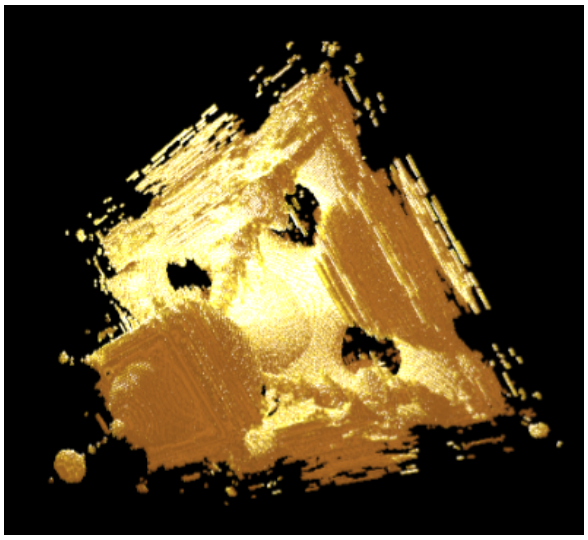


Fig.31 - POV 3.7 double complex $3D$ Mandelbrot-Mandelbrot-Mandelbrot set as a whole (point by point generation)

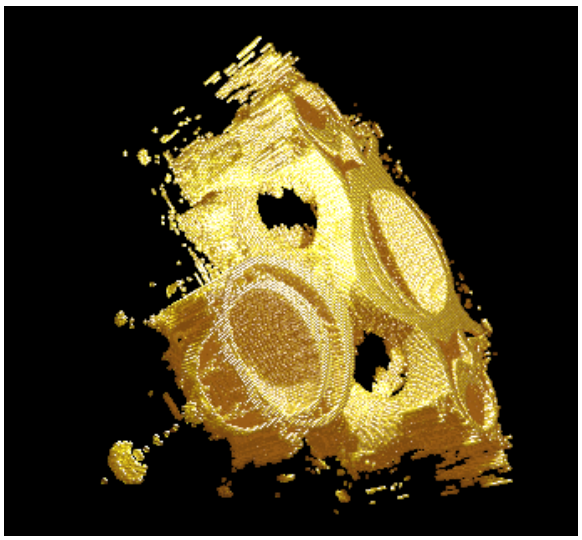


Fig.32 - POV 3.7 Open section of a $3D$ double complex Mandelbrot-Mandelbrot-Mandelbrot set as a whole (point by point generation)

Fig.33 - Cartesian plane sections of the POV 3.7 3*D* double complex Mandelbrot-Mandelbrot-Mandelbrot set as a whole (point by point generation)

```
      #declare Y1 = 0;   // start Mand1
      #declare Y2 = 0;   // start Mand2

     #for (k,1,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + IncX;  // cycle Mand1
       #declare XX2 = X2*X2 - Y2*Y2 + IncX;  // cycle Mand1
       #declare YY1 = 2*X1*Y1 + IncY1;   // cycle Mand1
       #declare YY2 = 2*X2*Y2 + IncY2;   // cycle Mand2
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

   #if ( W > R)       // escape if
      #if (W < R + 0.02)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 80, -10, 0 >
rotate < 0, Th, Ph > }
```

Julia-Julia-Julia double complex fractal set ($c_1 = c_2 = -0.745429$)

We now show what happens to a *Julia-Julia-Julia* double complex set identified by the parameters values $c_1 = c_2 = -0.745429$.

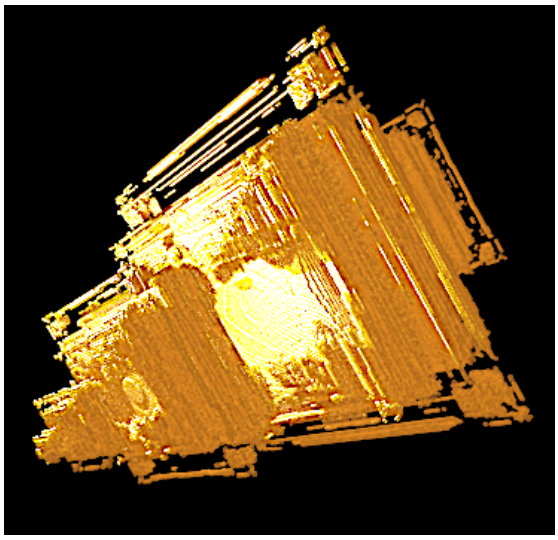


Fig.34 - POV 3.7 double complex $3D$ Julia-Julia-Julia ($c_1 = c_2 = -0.745429$) set as a whole (point by point generation)

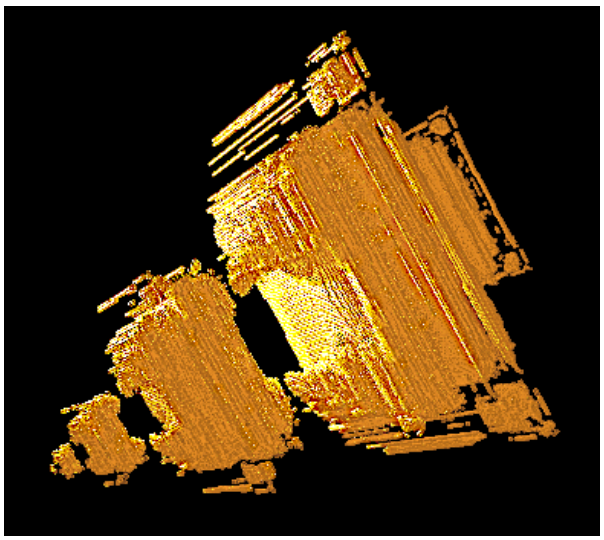


Fig.35 - POV 3.7 Open section of a $3D$ double complex Julia-Julia-Julia ($c_1 = c_2 = -0.745429$) set as a whole (point by point generation)

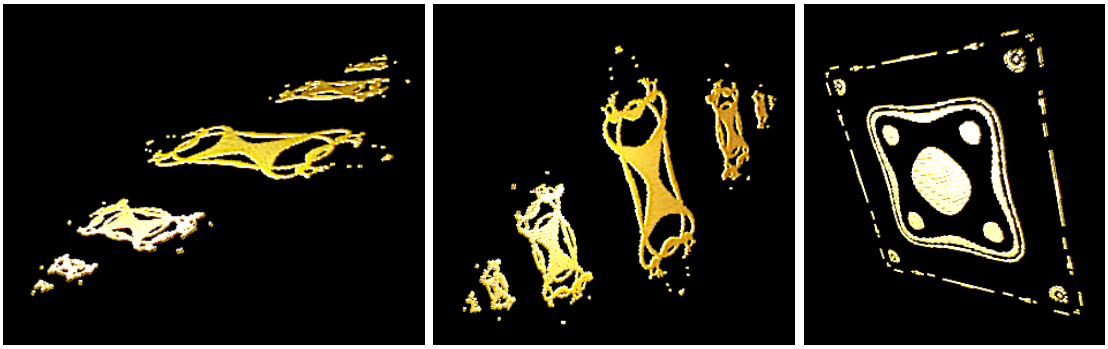


Fig.36 - Cartesian plane sections of the POV 3.7 $3D$ double complex Julia-Julia-Julia set ($c_1 = c_2 = -0.7454294$) as a whole (point by point generation)

POV-Ray 3.7 code to generate figs 34, 35 and 36

```
//=====================================================================
// Double complex 3D Julia-Julia-Julia set (c1 = c2 = -0.7454294)
//  as a whole (POV-Ray point by point generation)
//=====================================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }
   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }
 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;      // set number of pixels per area side
#declare st = 1;
#declare Nr = 20;
#declare Th = -30;
#declare Ph = 30;

#declare X1 = 0;
#declare X2 = 0;
#declare Y1 = 0;
#declare Y2 = 0;
```

```
#declare Cx1 = -0.7454294;
#declare Cx2 = -0.7454294;
#declare Cy1 = 0;
#declare Cy2 = 0;

 union{ #for (p, -n, n, st)
     #declare IncX = p*L/n;   // Julia1 X increment

    #for (q, -n, n, st)
       #declare IncY1 = q*L/n;    // Julia1 Y1 increment

    #for (r, -n, n, st)
      #declare IncY2 = r*L/n;  // Julia2 Y2 increment

      #declare X1 = IncX;   // start Julia1
      #declare X2 = IncX;   // start Julia2
      #declare Y1 = IncY1;   // start Julia1
      #declare Y2 = IncY2;   // start Julia2

     #for (k,1,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + Cx1;  // cycle Julia1
       #declare XX2 = X2*X2 - Y2*Y2 + Cx2;  // cycle Julia2
       #declare YY1 = 2*X1*Y1 + Cy1;   // cycle Julia1
       #declare YY2 = 2*X2*Y2 + Cy2;   // cycle Julia2
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

   #if ( W > R)       // escape if
      #if (W < R + 0.02)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 80, -10, 0 >
rotate < 0, Th, Ph > }
```

Julia-Julia-Julia "sea-horse" double complex fractal set ($c_1 = c_2 = -0.7454294 + i0.113089$)

And here is the structure of a double complex generalized *"sea-horse" Julia set* ($c_1 = c_2 = -0.7454294 + i0.113089$).

POV-Ray 3.7 code to generate fig 37, 38 and 39

```
//=====================================================================
// Double complex 3D Julia-Julia-Julia set
// (c1 = c2 = -0.7454294 + i 0.113089)
// as a whole (POV-Ray point by point generation)
/=====================================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 30, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;      // set number of pixels per area side
#declare st = 1;
#declare Nr = 50;
#declare Th = -10;
#declare Ph = 30;

#declare X1 = 0;
#declare X2 = 0;
#declare Y1 = 0;
#declare Y2 = 0;

#declare Cx1 = -0.7454294;
#declare Cx2 = -0.7454294;
#declare Cy1 = 0.113089;
#declare Cy2 = 0.113089;

 union{ #for (p, -n, n, st)
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -n, n, st)
       #declare IncY1 = q*L/n;    // Mand1 Y1 increment

    #for (r, -n, n, st)
      #declare IncY2 = r*L/n;  // Mand2 Y2 increment

      #declare X1 = IncX;   // start Julia 1
      #declare X2 = IncX;   // start Julia 2
      #declare Y1 = IncY1;   // start Julia 1
```

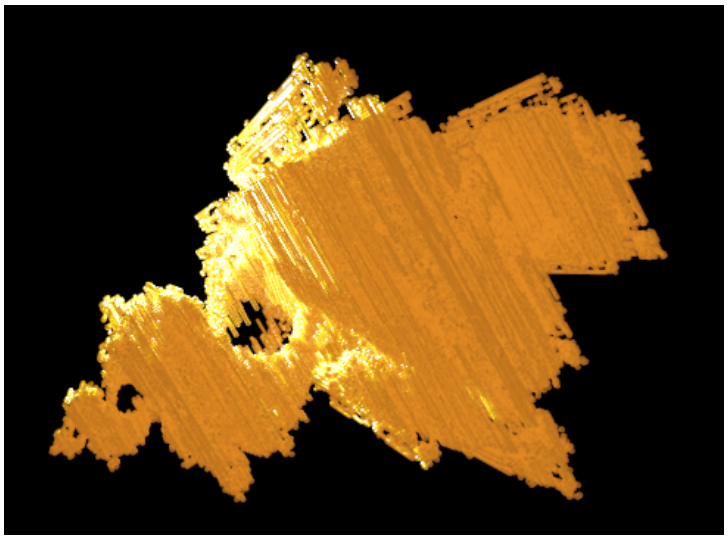


Fig.37 - POV 3.7 $3D$ Double complex Julia-Julia-Julia set ($c_1 = c_2 = -0.7454294 + i0.113089$) set as a whole (point by point generation)

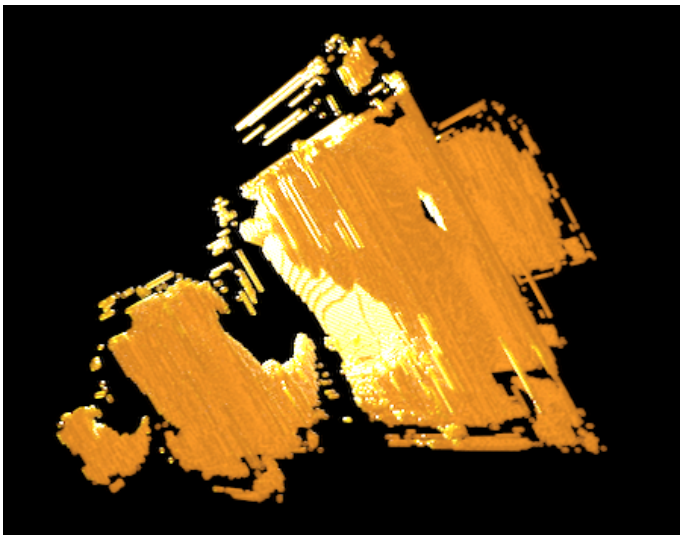


Fig.38 - POV 3.7 Open section of a $3D$ double complex Julia-Julia-Julia set ($c_1 = c_2 = -0.7454294 + i0.113089$) as a whole (point by point generation)

Fig.39 - Cartesian plane sections of the POV 3.7 $3D$ double complex Julia-Julia-Julia set ($c_1 = c_2 = -0.7454294 + i0.113089$) as a whole (point by point generation)

```
      #declare Y2 = IncY2;   // start Julia 2

     #for (k,1,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + Cx1;  // cycle Julia 1
       #declare XX2 = X2*X2 - Y2*Y2 + Cx2;  // cycle Julia 2
       #declare YY1 = 2*X1*Y1 + Cy1;   // cycle Julia 1
       #declare YY2 = 2*X2*Y2 + Cy2;   // cycle Julia 2
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

   #if ( W > R)       // escape if
      #if (W < R + 0.05)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 30, -20, 0 >
rotate < 0, Th, Ph > }
```

Julia-Mandelbrot-Mandelbrot double complex fractal set ($c_{1x} = c_{2x} = -0.7454294$)

The last example is given by a *Julia-Mandelbrot-Mandelbrot* obtained setting the parameters $c_{1x} = c_{2x} = -0.7454294$ and varying c_{1y}, c_{2y}.

POV-Ray 3.7 code to generate figs 40, 41 and 42

```
//============================================================
// Double complex 3D Julia-Mand-Mand (c = -0.7454294) set
// as a whole (POV-Ray point by point generation)
//============================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st = 1;

#declare Nr = 20;  // Nr = 40
#declare Th = -45;
#declare Ph = 150;

#declare X1 = 0;
#declare X2 = 0;
#declare Y1 = 0;
#declare Y2 = 0;
#declare Cx = -0.7454294;

 union{ #for (p, -n, n, st)
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -n, n, st)
       #declare IncY1 = q*L/n;    // Mand1 Y1 increment

    #for (r, -n, n, st)
      #declare IncY2 = r*L/n;  // Mand2 Y2 increment
```

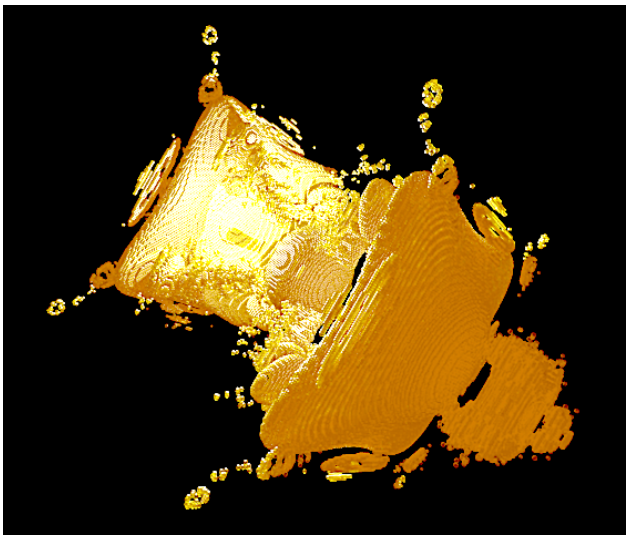


Fig.40 - POV 3.7 double complex $3D$ Julia-Mandelbrot-Mandelbrot set
as a whole (point by point generation)

Fig.41 - POV 3.7 Open section of a $3D$ double complex Julia-Mandelbrot-Mandelbrot ($c = -0.7454294$) set
as a whole (point by point generation)

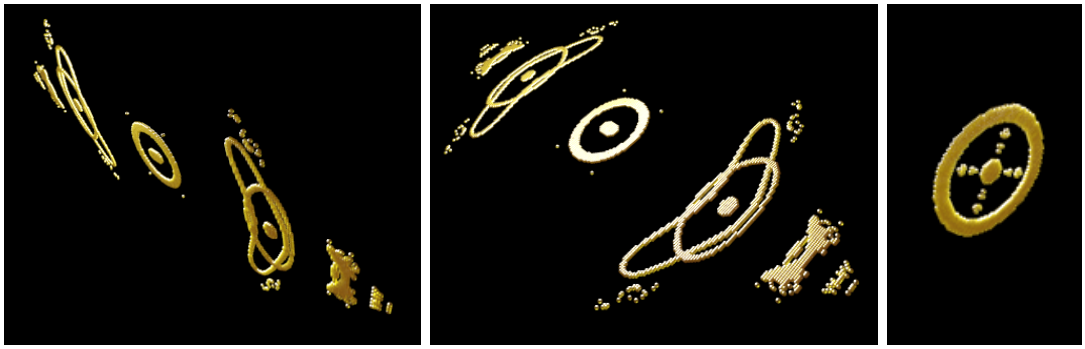


Fig.42 - Cartesian plane sections of the POV 3.7 3D double complex Julia-Mandelbrot-Mandelbrot set ($c = -0.7454294$) as a whole (point by point generation)

```
    #declare X1 = IncX;   // start Julia1
    #declare X2 = IncX;   // start Julia2
    #declare Y1 = 0;   // start Mand1
    #declare Y2 = 0;   // start Mand2

   #for (k,1,Nr)
     #declare XX1 = X1*X1 - Y1*Y1 + Cx;  // cycle Julia1
     #declare XX2 = X2*X2 - Y2*Y2 + Cx;  // cycle Julia2
     #declare YY1 = 2*X1*Y1 + IncY1;   // cycle Mand1
     #declare YY2 = 2*X2*Y2 + IncY2;   // cycle Mand2
     #declare X1 = XX1;
     #declare X2 = XX2;
     #declare Y1 = YY1;
     #declare Y2 = YY2;
     #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

 #if ( W > R)       // escape if
    #if (W < R + 0.02)     // escape if
  sphere {
   < p, q, r >, 1   //  adding 3d axis
     texture {
     pigment { color Col_Glass_Yellow }
      }
      finish { ambient rgb <0.3,0.1,0.1>
      diffuse .3
     reflection .3
     specular 1 }   // plot sphere
  }

#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r

rotate < 0, Th, Ph >
translate < 0, -15, -5 >}
```

6.3.2 Generalized $3D$ double complex fractals generated sequentially

Mandelbrot-Mandelbrot-Mandelbrot double complex fractal set

Now it is time to consider the *double complex Mandelbrot and Julia fractals* generated sequentially. Here are the codes and resulting pictures.

POV-Ray 3.7 code to generate figs 43 and 44

```
//=========================================================
// Double complex 3D Mand-Mand-Mand fractal set
//  generated sequentially (POV-Ray point by point generation)
//=========================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare ns = 75;
#declare st = 1;
#declare Nr = 20;  // Nr = 40
#declare Th = -45;
#declare Ph = 30;

#declare X1 = 0;
#declare X2 = 0;
#declare Y1 = 0;
#declare Y2 = 0;

 union{ #for (p, -ns, ns, st)
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -ns, ns, st)
       #declare IncY1 = q*L/n;    // Mand1 Y1 increment

    #for (r, -ns, ns, st)
      #declare IncY2 = r*L/n;  // Mand2 Y2 increment
```

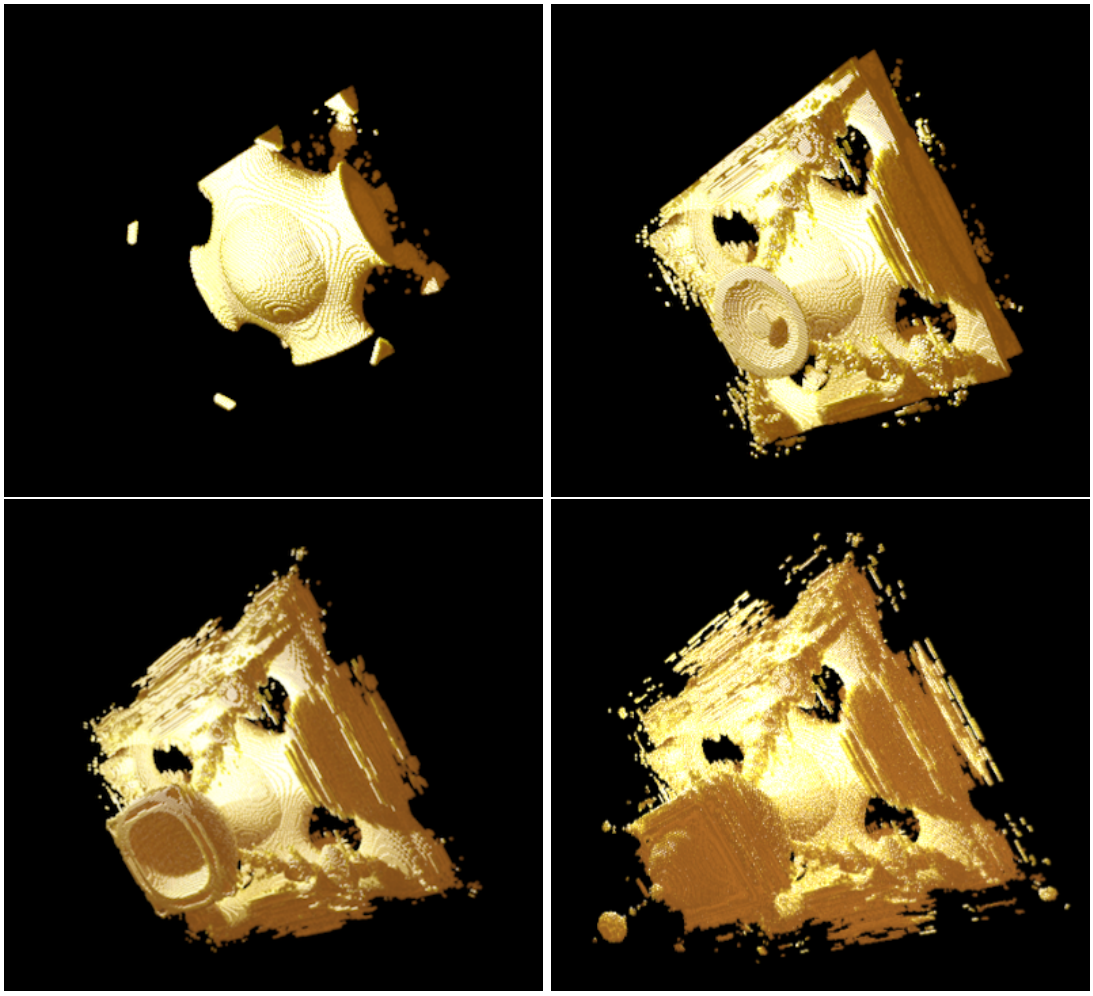


Fig.43 - POV 3.7 $3D$ double complex Mandelbrot-Mandelbrot-Mandelbrot set generated sequentially

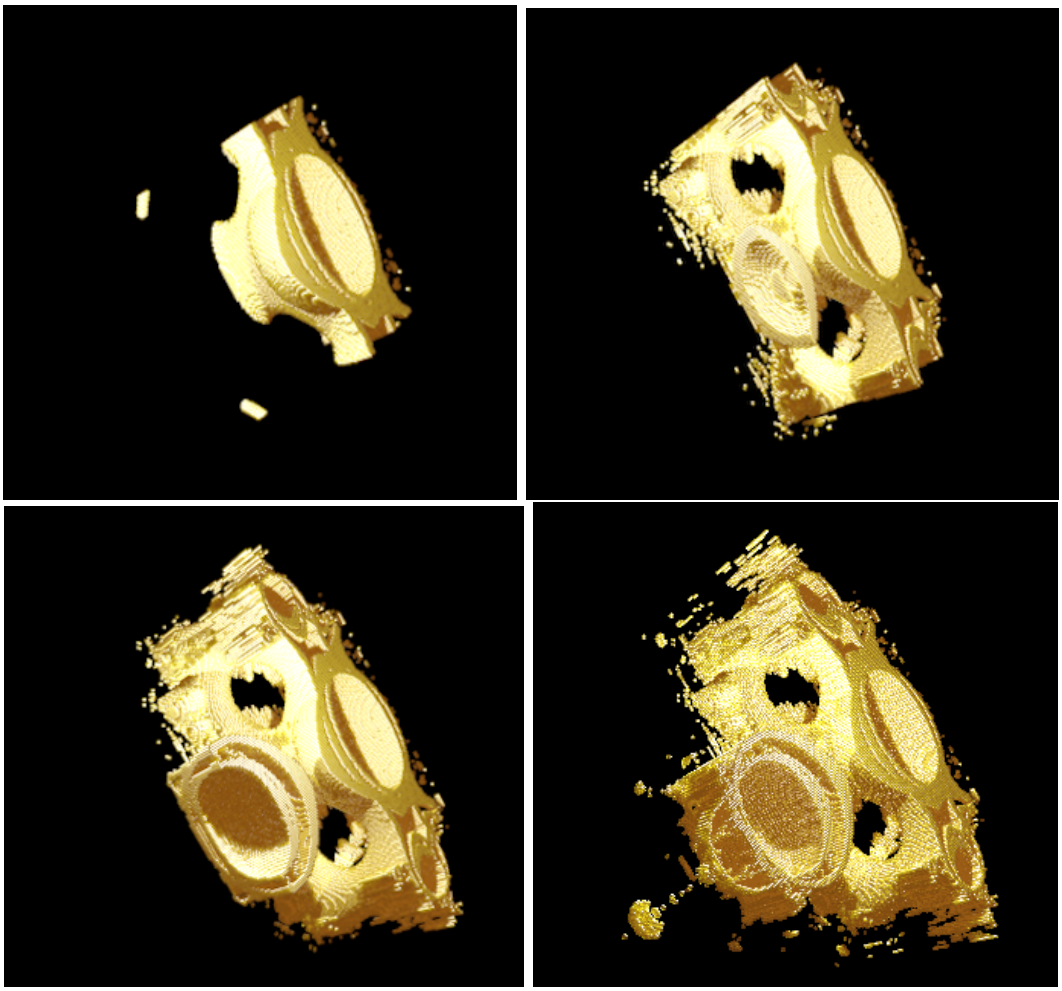


Fig.44 - POV 3.7 3*D* Open section of a double complex
Mandelbrot-Mandelbrot-Mandelbrot set generated sequentially

```
      #declare X1 = 0;   // start Mand1
      #declare X2 = 0;   // start Mand1
      #declare Y1 = 0;   // start Mand1
      #declare Y2 = 0;   // start Mand2

     #for (k,1,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + IncX;  // cycle Mand1
       #declare XX2 = X2*X2 - Y2*Y2 + IncX;  // cycle Mand1
       #declare YY1 = 2*X1*Y1 + IncY1;   // cycle Mand1
       #declare YY2 = 2*X2*Y2 + IncY2;   // cycle Mand2
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

   #if ( W > R)       // escape if
      #if (W < R + 0.02)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 80, -10, 0 >
rotate < 0, Th, Ph > }
```

Julia-Julia-Julia double complex fractal sets ($c_1 = c_2 = -0.7454294$)

Here is a *Julia-Julia-Julia* double complex fractal set ($c_1 = c_2 = -0.7454294$).

POV-Ray 3.7 code to generate figs 45 and 46

```
//=========================================================
// Double complex 3D Julia-Julia-Julia fractal set
// ($c_{1} = c_{2} = -0.7454294$) generated sequentially
// (POV-Ray point by point generation)
//=========================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
```

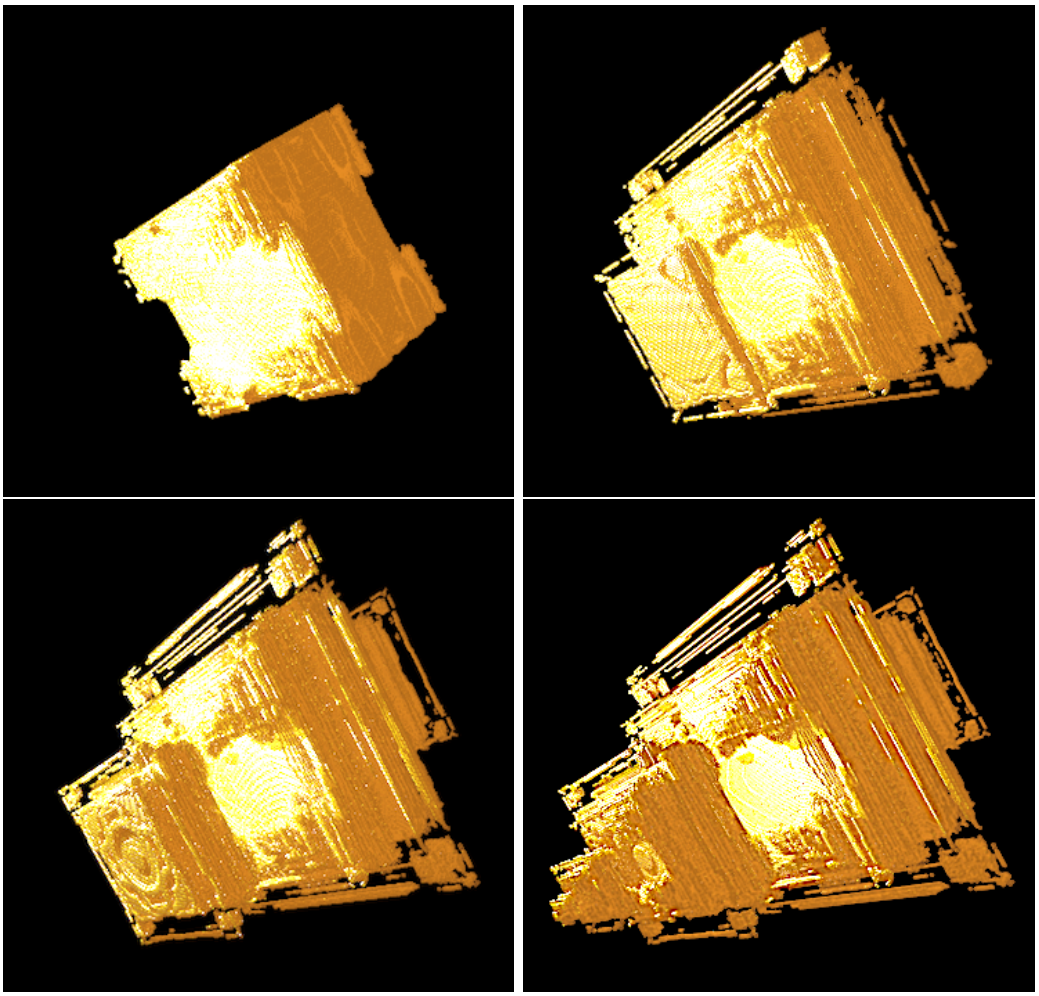


Fig.45 - POV 3.7 $3D$ double complex Julia-Julia-Julia set ($C = -0.7454294$)
generated sequentially

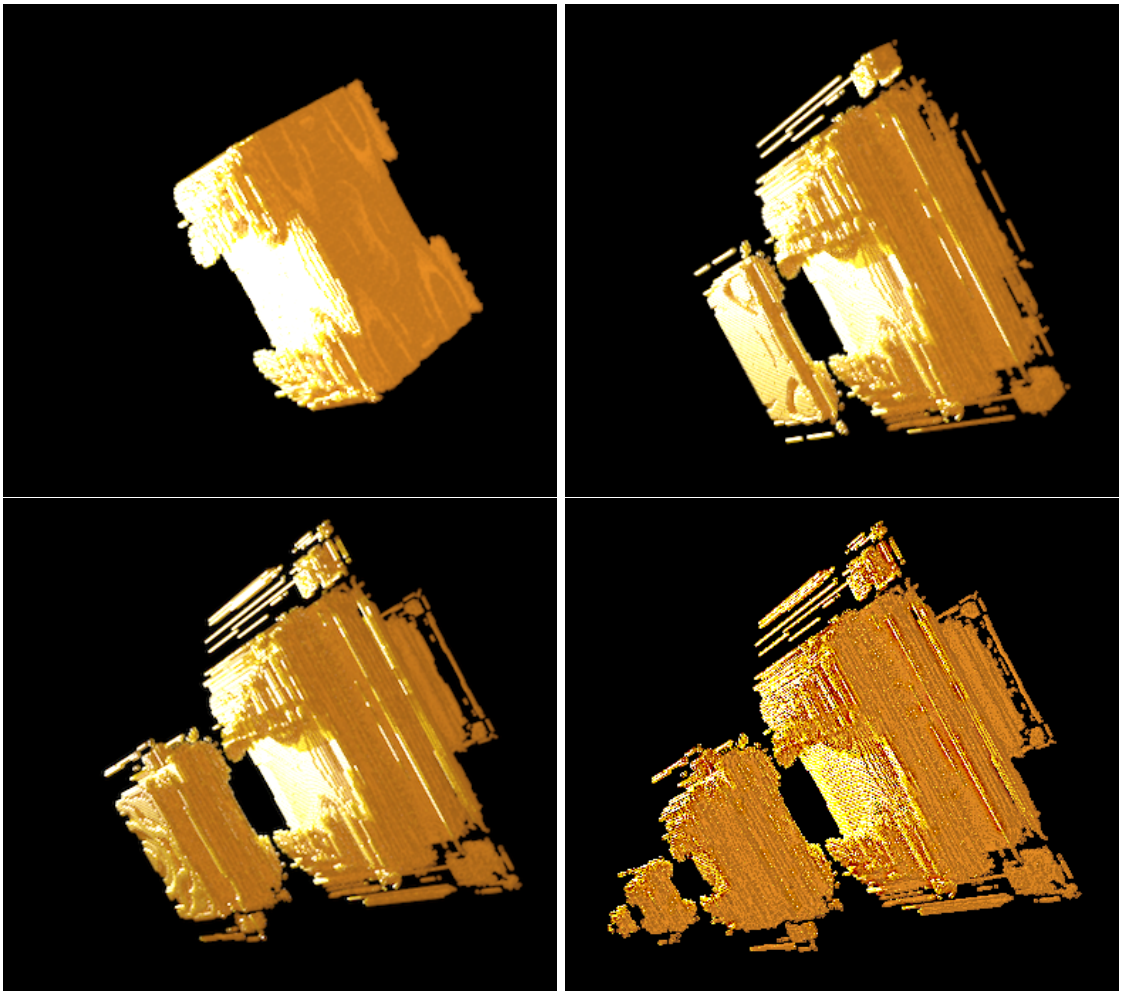


Fig.46 - POV 3.7 $3D$ open section of a double complex Julia-Julia-Julia set
($C = -0.7454294$) generated sequentially

```
  }

   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare ns = 50;
#declare st = 1;
#declare Nr = 20;  // Nr = 40
#declare Th = -30;
#declare Ph = 30;

#declare X1 = 0;
#declare X2 = 0;
#declare Y1 = 0;
#declare Y2 = 0;

#declare Cx1 = -0.7454294;
#declare Cx2 = -0.7454294;
#declare Cy1 = 0;
#declare Cy2 = 0;


 union{ #for (p, -ns, ns, st)
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -ns, ns, st)
       #declare IncY1 = q*L/n;    // Mand1 Y1 increment

    #for (r, -ns, ns, st)
      #declare IncY2 = r*L/n;  // Mand2 Y2 increment

      #declare X1 = IncX;   // start Mand1
      #declare X2 = IncX;   // start Mand1
      #declare Y1 = IncY1;   // start Mand1
      #declare Y2 = IncY2;   // start Mand2

     #for (k,1,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + Cx1;  // cycle Mand1
       #declare XX2 = X2*X2 - Y2*Y2 + Cx2;  // cycle Mand1
       #declare YY1 = 2*X1*Y1 + Cy1;   // cycle Mand1
       #declare YY2 = 2*X2*Y2 + Cy2;   // cycle Mand2
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

   #if ( W > R)       // escape if
      #if (W < R + 0.02)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
```

```
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 80, -10, 0 >
rotate < 0, Th, Ph > }
```

Julia-Julia-Julia "sea-horse" double complex fractal set ($c_1 = c_2 = -0.7454294 + i0.113089$)

Here is also the *Julia-Julia-Julia "sea horse"* double complex fractal set ($c_1 = c_2 = -0.7454294 + i0.113089$).

POV-Ray 3.7 code to generate figs 47 and 48

```
//=========================================================
// Double complex 3D Julia-Julia-Julia ‘‘sea-horse’’ fractal set
// ($c_{1} = c_{2} = -0.7454294 + i 0.113089$$) generated sequentially
// (POV-Ray point by point generation)
//=========================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare ns = 100;
#declare st = 1;
#declare Nr = 20;  // Nr = 40
```

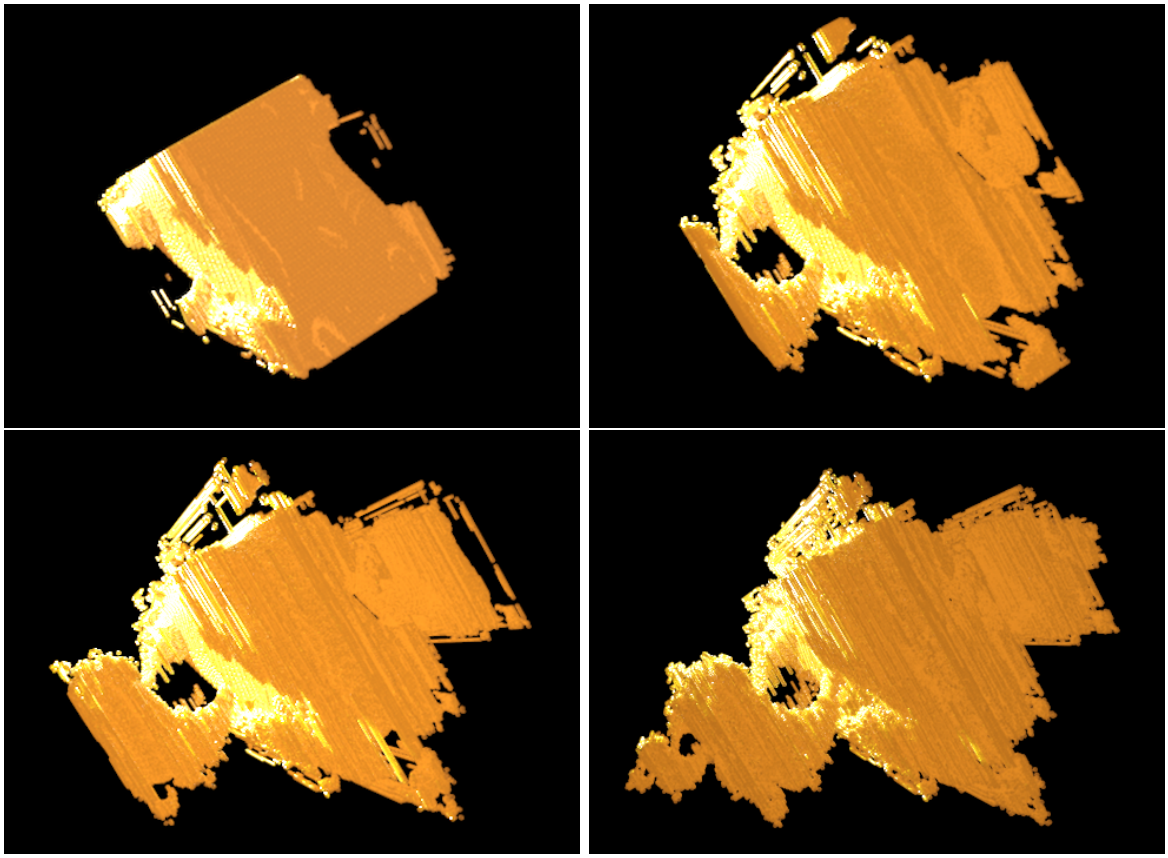


Fig.47 - POV 3.7 $3D$ Double complex "sea horse" Julia-Julia-Julia set ($c_1 = c_2 = -0.7454294 + i0.113089$) generated sequentially

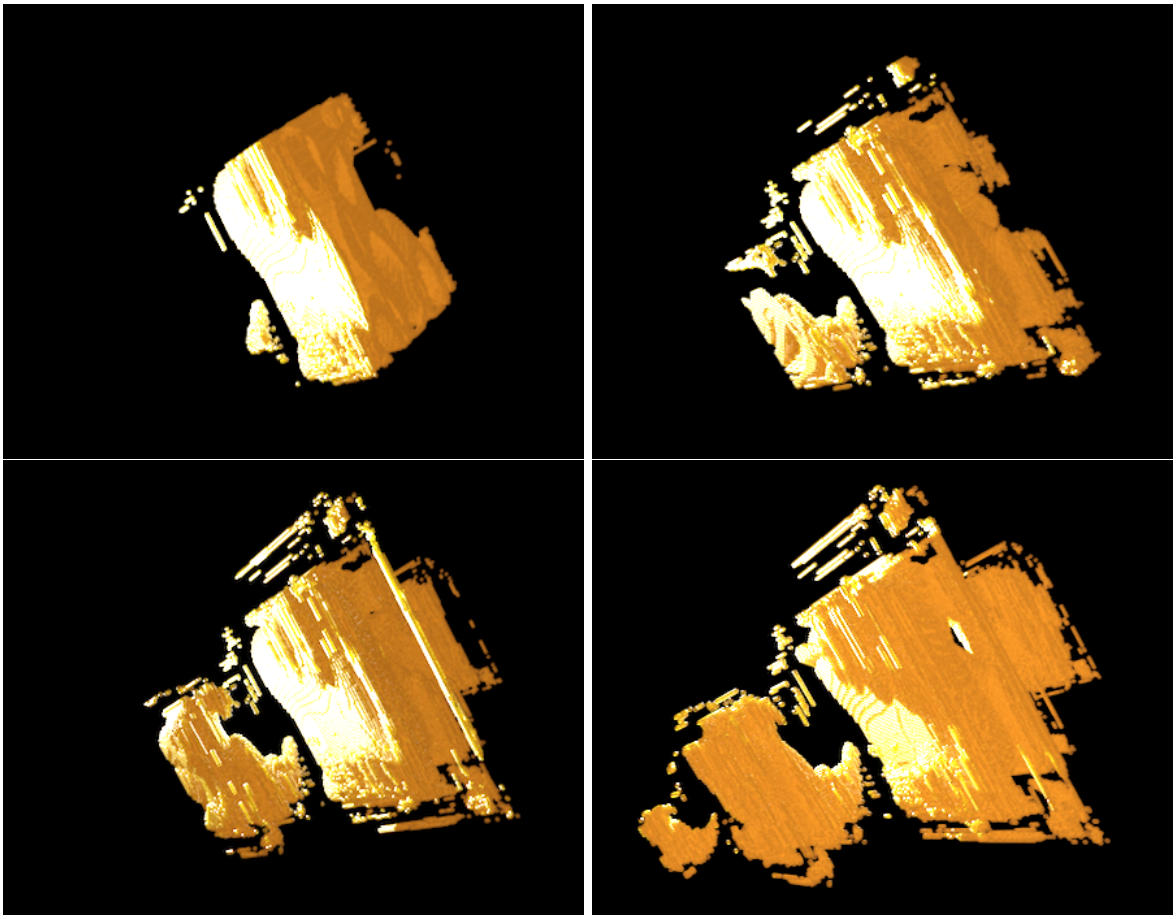


Fig.48 - POV 3.7 $3D$ open section of a Double complex "sea horse" Julia-Julia-Julia set $(c_1 = c_2 = -0.7454294 + i0.113089)$ generated sequentially

```
#declare Th = -10;
#declare Ph = 30;

#declare X1 = 0;
#declare X2 = 0;
#declare Y1 = 0;
#declare Y2 = 0;

#declare Cx1 = -0.7454294;
#declare Cx2 = -0.7454294;
#declare Cy1 = 0.113089;
#declare Cy2 = 0.113089;


 union{ #for (p, -ns, ns, st)
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -ns, ns, st)
       #declare IncY1 = q*L/n;    // Mand1 Y1 increment

    #for (r, -ns, ns, st)
      #declare IncY2 = r*L/n;  // Mand2 Y2 increment

      #declare X1 = IncX;   // start Mand1
      #declare X2 = IncX;   // start Mand1
      #declare Y1 = IncY1;   // start Mand1
      #declare Y2 = IncY2;   // start Mand2

     #for (k,1,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + Cx1;  // cycle Mand1
       #declare XX2 = X2*X2 - Y2*Y2 + Cx2;  // cycle Mand1
       #declare YY1 = 2*X1*Y1 + Cy1;   // cycle Mand1
       #declare YY2 = 2*X2*Y2 + Cy2;   // cycle Mand2
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

   #if ( W > R)       // escape if
      #if (W < R + 0.03)     // escape if
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere
    }
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r
translate < 30, -20, 0 >
rotate < 0, Th, Ph > }
```

6.3.3 Generalized $3D$ double complex fractal sets generated randomly

The last step of this chapter is concerned with random generation process of double complex $3D$ generalized Mandelbrot, Julia and mixed fractal sets.

Mandelbrot-Mandelbrot-Mandelbrot quaternion fractal set

As usual we begin with *Mandelbrot-Mandelbrot-Mandelbrot* set.

POV-Ray 3.7 code to generate figs 49 and 50

```
//=========================================================
// Double complex 3D Mand-Mand-Mand set generated randomly
// (POV-Ray point by point generation)
//=========================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st = 1;
#declare Nr = 20;  // Nr = 40
#declare Th = -45;
#declare Ph = 30;
#declare X1 = 0;
#declare X2 = 0;
#declare Y1 = 0;
#declare Y2 = 0;

#declare K = array[2*n+1][2*n+1][2*n+1];   // escape rate 3D matrix
 #for (p, -n, n, st)
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -n, n, st)
       #declare IncY1 = q*L/n;    // Mand1 Y1 increment

    #for (r, -n, n, st)
```

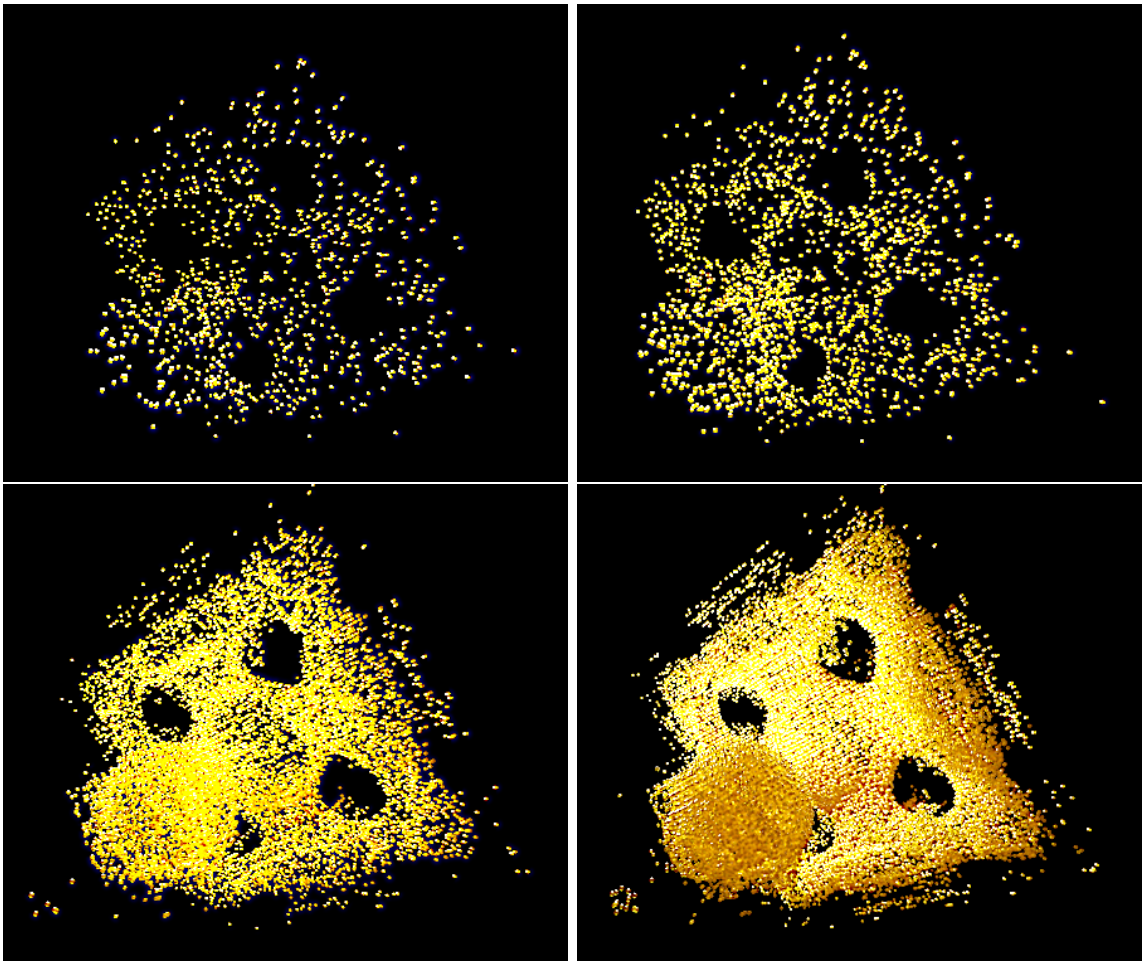


Fig.49 - POV 3.7 3*D* Double complex Mandelbrot-Mandelbrot-Mandelbrot set generated randomly

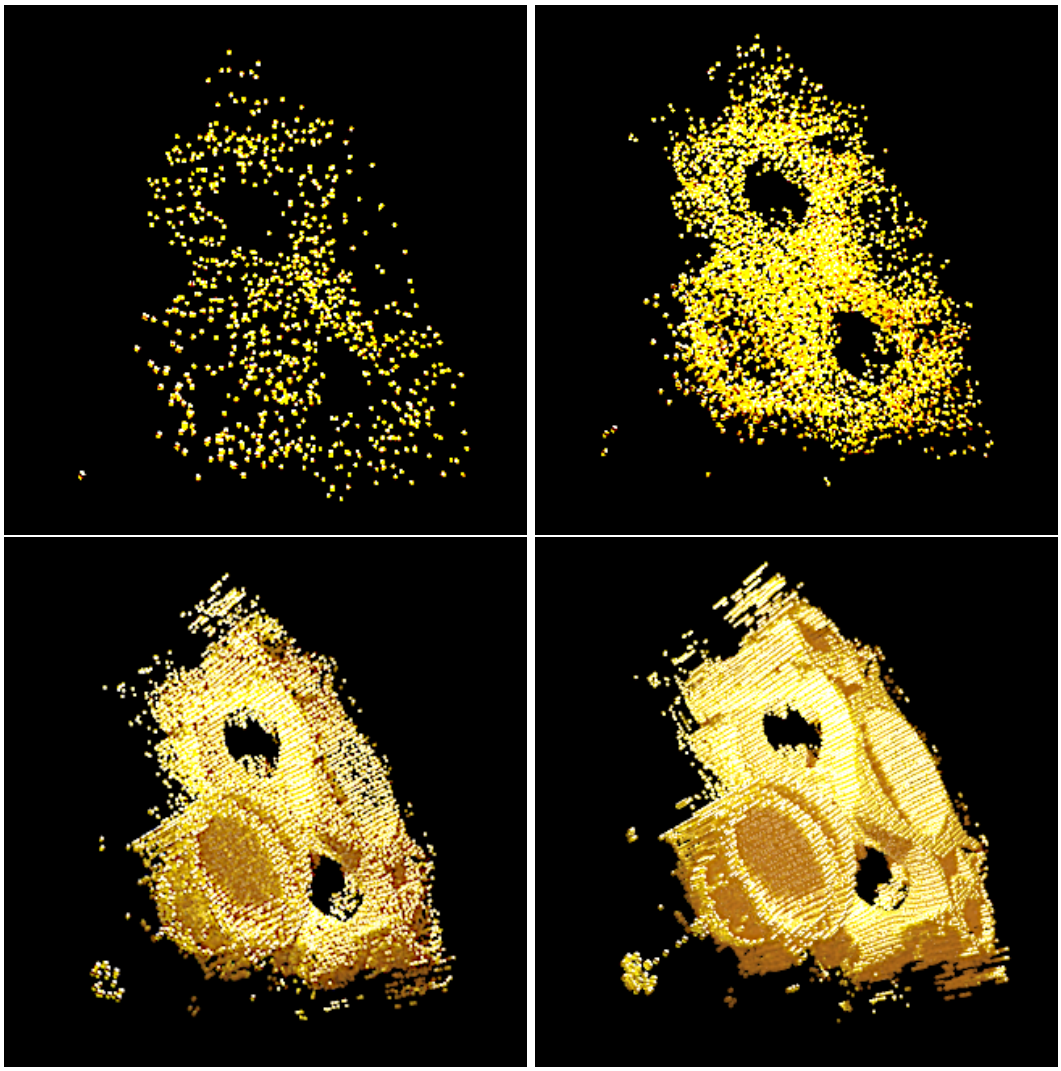


Fig.50 - POV 3.7 3D Open section of a double complex Mandelbrot-Mandelbrot-Mandelbrot set generated randomly

```
      #declare IncY2 = r*L/n;  // Mand2 Y2 increment

      #declare X1 = 0;   // start Mand1
      #declare X2 = 0;   // start Mand1
      #declare Y1 = 0;   // start Mand1
      #declare Y2 = 0;   // start Mand2
      #declare K[p+n][q+n][r+n] = 0;

     #for (k,0,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + IncX;  // cycle Mand1
       #declare XX2 = X2*X2 - Y2*Y2 + IncX;  // cycle Mand1
       #declare YY1 = 2*X1*Y1 + IncY1;   // cycle Mand1
       #declare YY2 = 2*X2*Y2 + IncY2;   // cycle Mand2
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

   #if ( W > R)       // escape if
      #if (W < R + 0.02)     // escape if
      #declare K[p+n][q+n][r+n] = k;
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r

 #declare Rnd_1 = seed (1153);
 #declare Rnd_2 = seed (553);
 #declare Rnd_3 = seed (876);
 union{ #for(j,0,n*n*n/4)    // partial values  n*n*n,
  n*n*n/4, n*n*n/32, n*n*n/64
   #declare p = int(2*n*rand(Rnd_1));
   #declare q = int(2*n*rand(Rnd_2));
   #declare r = int(2*n*rand(Rnd_3));
    #if(K[p][q][r] > 0)
    sphere {
     < -n+p, -n+q, -n+r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

translate < 80, -10, 0 >
rotate < 0, Th, Ph > }
#end // end if
#end}  // end for j
```

Julia-Julia-Julia double complex fractal set ($c_1 = c_2 = -0.745429$)

As a second example we show a Julia-Julia-Julia double complex fractal set the parameters of which are given by $c_1 = c_2 = -0.745429$.

POV-Ray 3.7 code to generate figs 51 and 52

```
//=========================================================
// Double complex 3D Julia-Julia-Julia set (c1 = c2 =  -0.745429)
// generated randomly (POV-Ray point by point generation)
//=========================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st = 1;
#declare Nr = 20;  // Nr = 40
#declare Th = -30;
#declare Ph = 30;

#declare X1 = 0;
#declare X2 = 0;
#declare Y1 = 0;
#declare Y2 = 0;

#declare Cx1 = -0.7454294;
#declare Cx2 = -0.7454294;
#declare Cy1 = 0;
#declare Cy2 = 0;

#declare K = array[2*n+1][2*n+1][2*n+1];   // escape rate 3D matrix

#for (p, -n, n, st)
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -n, n, st)
       #declare IncY1 = q*L/n;    // Mand1 Y1 increment

    #for (r, -n, n, st)
      #declare IncY2 = r*L/n;  // Mand2 Y2 increment

      #declare X1 = IncX;   // start Mand1
      #declare X2 = IncX;   // start Mand1
```

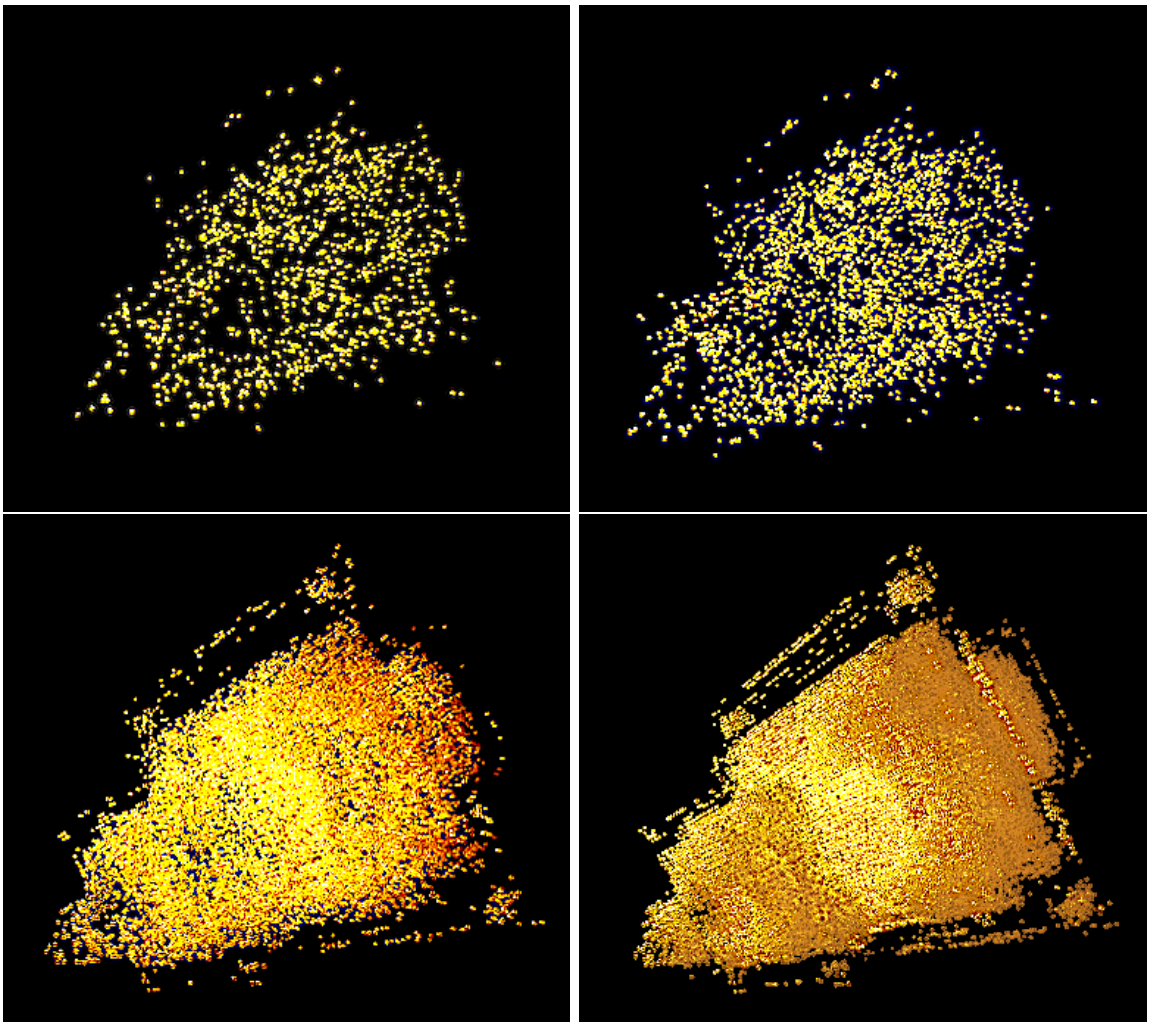


Fig.51 - POV 3.7 $3D$ Double complex Julia-Julia-Julia set ($c_1 = c_2 = -0.745429$) generated randomly

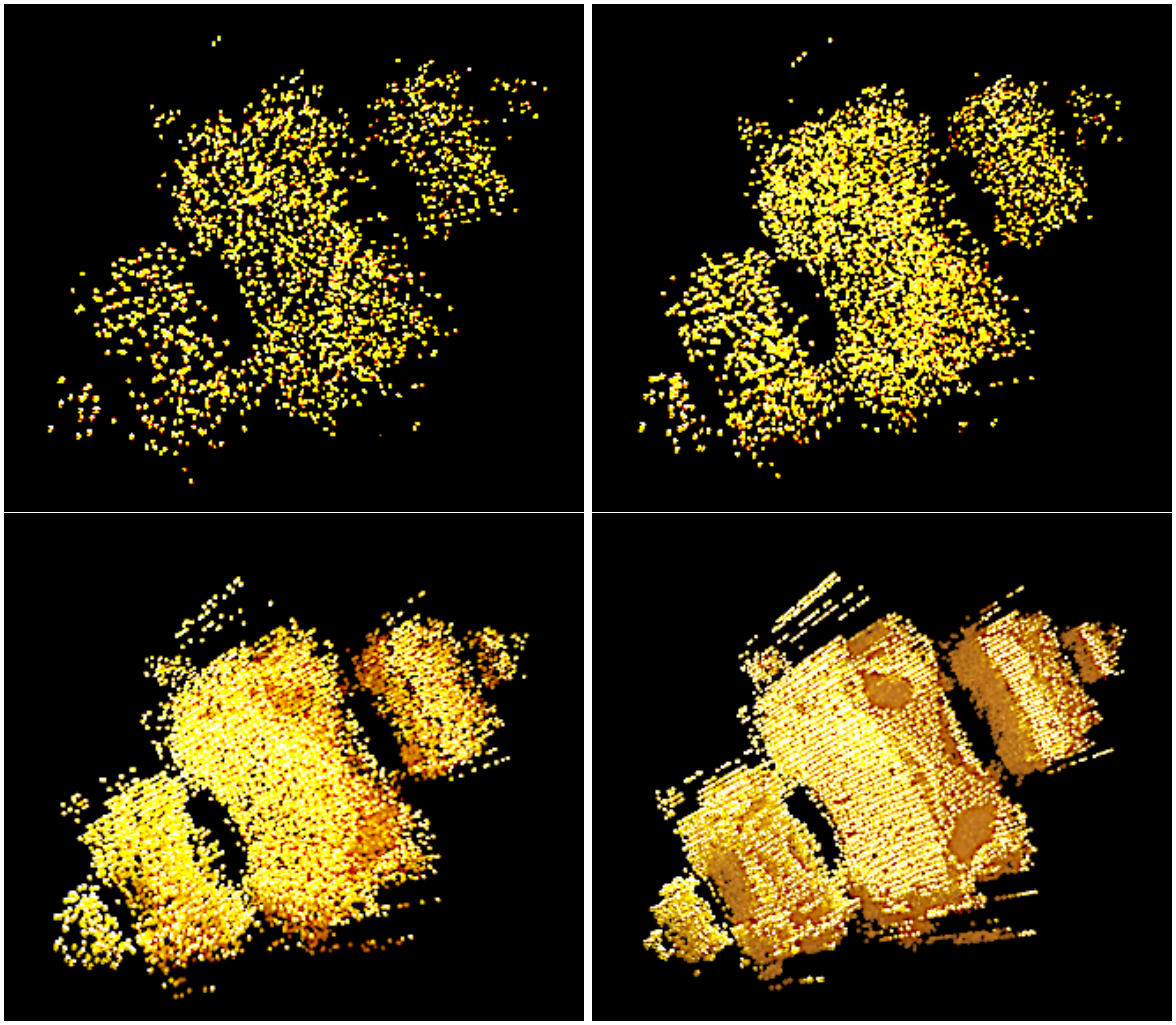


Fig.52 - POV 3.7 $3D$ Open section of a double complex Julia-Julia-Julia set
($c_1 = c_2 = -0.745429$) generated randomly

```
      #declare Y1 = IncY1;   // start Mand1
      #declare Y2 = IncY2;   // start Mand2
      #declare K[p+n][q+n][r+n] = 0;

        #for (k,1,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + Cx1;  // cycle Mand1
       #declare XX2 = X2*X2 - Y2*Y2 + Cx2;  // cycle Mand1
       #declare YY1 = 2*X1*Y1 + Cy1;   // cycle Mand1
       #declare YY2 = 2*X2*Y2 + Cy2;   // cycle Mand2
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

   #if ( W > R)       // escape if
   #if (W < R + 0.02)     // escape if
   #declare K[p+n][q+n][r+n] = k;
#end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r

 #declare Rnd_1 = seed (1153);
 #declare Rnd_2 = seed (553);
 #declare Rnd_3 = seed (876);

 union{ #for(j,0,n*n*n/4)    // partial values n*n*n*4, n*n*n/4, n*n*n/32, n*n*n/64
   #declare p = int(2*n*rand(Rnd_1));
   #declare q = int(2*n*rand(Rnd_2));
   #declare r = int(2*n*rand(Rnd_3));
    #if(K[p][q][r] > 0)
    sphere {
     < -n+p, -n+q, -n+r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

translate < 80, -10, 0 >
rotate < 0, Th, Ph > }
#end // end if
#end}  // end for j
```

Julia-Mandelbrot-Mandelbrot double complex fractal set ($c_{1x} = c_{2x} = -0.7454294$)

The last example is a *Julia-Mandelbrot-Mandelbrot* obtained setting the parameters $c_{1x} = c_{2x} = -0.7454294$ and varying c_{1y}, c_{2y} generated starting from *random* initial conditions.

POV-Ray 3.7 code to generate figs 53 and 54

```
//=========================================================
// Double complex 3D Julia-Mand-Mand set (c1x = c2x = 0.745429)
// generated randomly (POV-Ray point by point generation)
//=========================================================

#include "colors.inc"    // Standard Color definitions
#include "glass.inc"    // Glass pigment effect
#include "metals.inc"   // Metal pigment effect

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera { // set view point (camera) location
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
  }

   light_source { // set point light sources location
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

 light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#declare R =  .1;    // set radius value
#declare L = 2;     // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st = 1;
#declare Nr = 20;  // Nr = 40
#declare Th = -45;
#declare Ph = 150;

#declare X1 = 0;
#declare X2 = 0;
#declare Y1 = 0;
#declare Y2 = 0;

#declare Cx = -0.7454294;

#declare K = array[2*n+1][2*n+1][2*n+1];   // escape rate 3D matrix

     #for (p, -n, n, st)
     #declare IncX = p*L/n;   // Mand1 X increment

    #for (q, -n, n, st)
       #declare IncY1 = q*L/n;    // Mand1 Y1 increment

    #for (r, -n, n, st)
      #declare IncY2 = r*L/n;  // Mand2 Y2 increment

      #declare X1 = IncX;   // start Julia1
      #declare X2 = IncX;   // start Julia2
      #declare Y1 = 0;   // start Mand1
      #declare Y2 = 0;   // start Mand2
      #declare K[p+n][q+n][r+n] = 0;
```

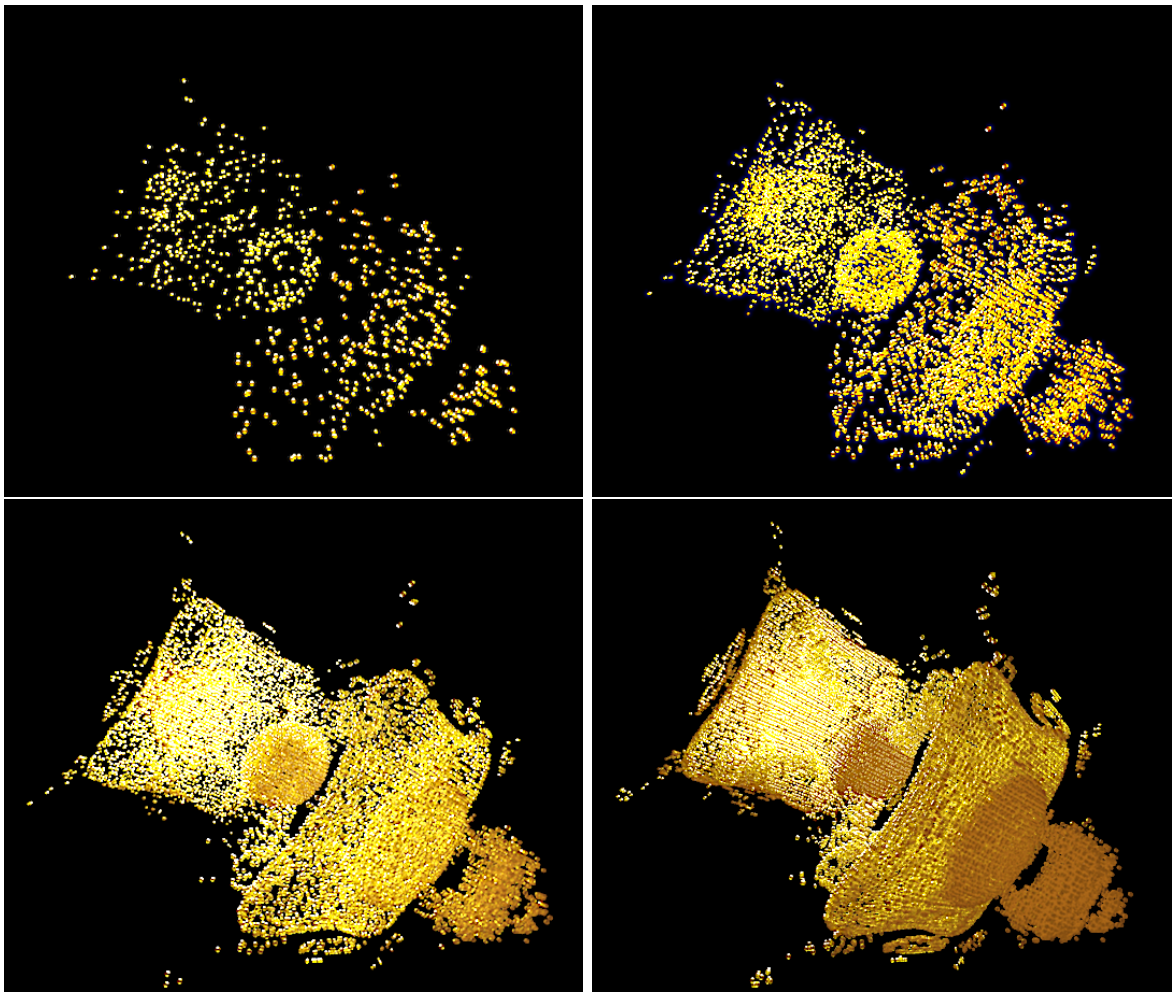


Fig.53 - POV 3.7 $3D$ Double complex Julia-Mandelbrot-Mandelbrot set ($c_{1x} = c_{2x} = 0.745429$) generated randomly

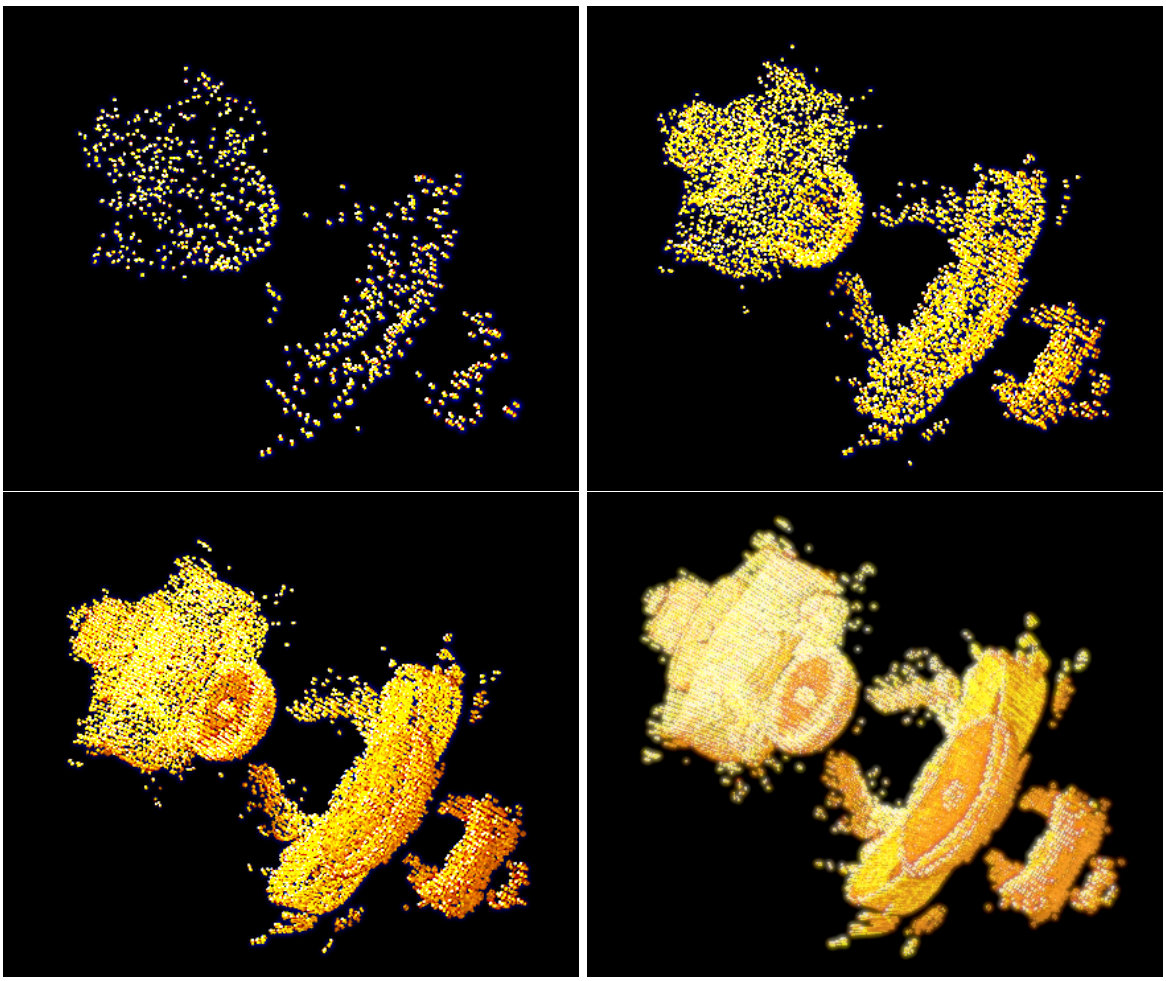


Fig.54 - POV 3.7 $3D$ Open section of a double complex Julia-Mandelbrot-Mandelbrot set ($c_{1x} = c_{2x} = 0.745429$) generated randomly

```
     #for (k,1,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + Cx;  // cycle Julia1
       #declare XX2 = X2*X2 - Y2*Y2 + Cx;  // cycle Julia2
       #declare YY1 = 2*X1*Y1 + IncY1;   // cycle Mand1
       #declare YY2 = 2*X2*Y2 + IncY2;   // cycle Mand2
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

   #if ( W > R)       // escape if
      #if (W < R + 0.02)     // escape if
      #declare K[p+n][q+n][r+n] = k;
      #end  // end if
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r

 #declare Rnd_1 = seed (1153);
 #declare Rnd_2 = seed (553);
 #declare Rnd_3 = seed (876);

 union{ #for(j,0,n*n*n/4)    // partial values n*n*n*4, n*n*n, n*n*n/4, n*n*n/32
  #declare p = int(2*n*rand(Rnd_1));
   #declare q = int(2*n*rand(Rnd_2));
   #declare r = int(2*n*rand(Rnd_3));
    #if(K[p][q][r] > 0)
    sphere {>
     < -n+p, -n+q, -n+r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

rotate < 0, Th, Ph >
translate < 0, -15, -5 >}
#end // end if
#end}  // end for j
```

Chapter 7

Fractal structures from cellular automata

Rendering random processes

7.1 Introduction

Biological living systems are *organized ordered structures* which arise and develop according to cell reproduction. A process which requires *spatial contiguity* between generating and generated cells. An interesting drastic simplified model scheme of contiguity generation (and possible suppression) processes is offered by *cellular automata,*[1] which operate under the following assumptions.

- Each cell is represented schematically as a point, or a disk, or a box in a $2D$ *cellular automaton,* or a small sphere or cube in $3D$, disregarding any internal structure.
- Each cell is allowed to occupy a single place within a grid covering some region of plane or space.
- A *rule* (*law*) is assigned to each cell according to which it is allowed to generate another cell in a specific contiguous place and not elsewhere. Moreover also a second rule may be stated according to which an existing cell can survive or not (or assume a special *state* among a set of admissible ones) in the next step of the process.
- The initial conditions: number and position of the starting cell(s) may be assigned or random and the *law* governing the process generation is allowed to involve even random numbers.

White color on a black background denotes alive cells, while black color denotes no cell or a dead cell. More colors may be introduced to denote more possible states of a cell.

[1]The idea of firstly conceiving *cellular automata* is commonly attributed to Stanisław Ulam and John von Neumann. It was proposed in 1948. Later Stephen Wolfram applied the same idea in its more recent fascinating book *A new kind of science* (www.wolframscience.com/nks/). For more details see, *e.g.,* natureofcode.com/book/chapter-7-cellular-automata/.

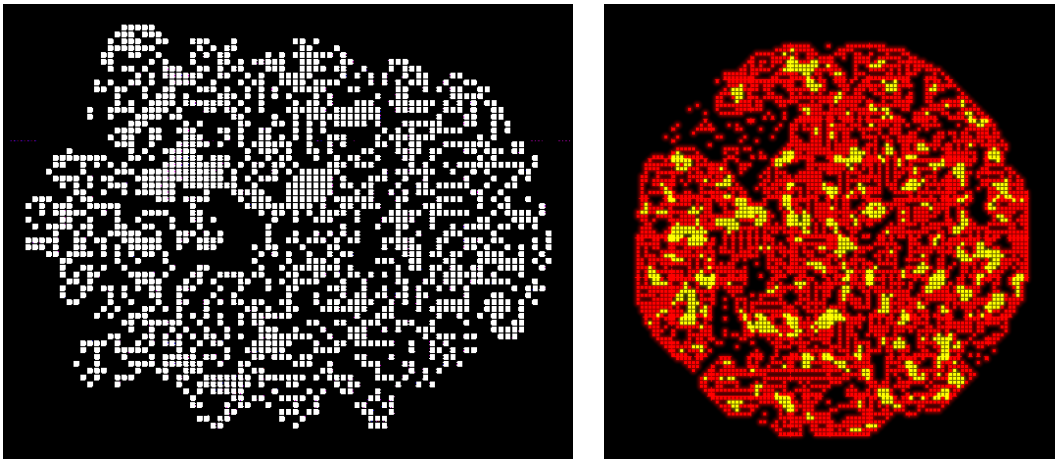


Fig.1 - Examples of cellular automata

VIEW ANIMATION 1 (requires internet connection) **VIEW ANIMATION 2**

Among the most famous computer dedicated software to program *cellular automata,* we must mention *Golly*[2] which operates assigning the *generating* new cells according to a suitable code of instructions which will not be discussed here. And among the *cellular automata* one of the most known is *The game of life* (by John Conway) the name of which suggests, by itself, some application to a biological context.

Even if it was originally created as a game for recreation, it revealed later some relevance for possible mathematical applications to more serious models of scientific interest. Notwithstanding it is based on very simple rules of generation/cancellation of cells it is able to produce unpredictable configurations, depending on the initial conditions choice.

The elementary rules are the following.

1. If a cell is white (alive) it will become black (dead) if and only if

   (a) it has four or more white neighbors;

   (b) it has one or fewer white neighbors.

2. If a cell is black (dead) it will come white (alive) if it has precisely three white neighbors.

On July 2018 the 3.2 version of Golly was released which implements also a first script allowing to test $3D$ cellular automata (*see* fig. 3). Of course a more impressive effect appears when one looks at a movie showing the dynamical behavior of the automata which blink continuously from black to white or colors and viceversa.

[2] *Golly* is an open source software which may be downloaded for several platforms either as a compiled binary or as a source code (golly.sourceforge.net).

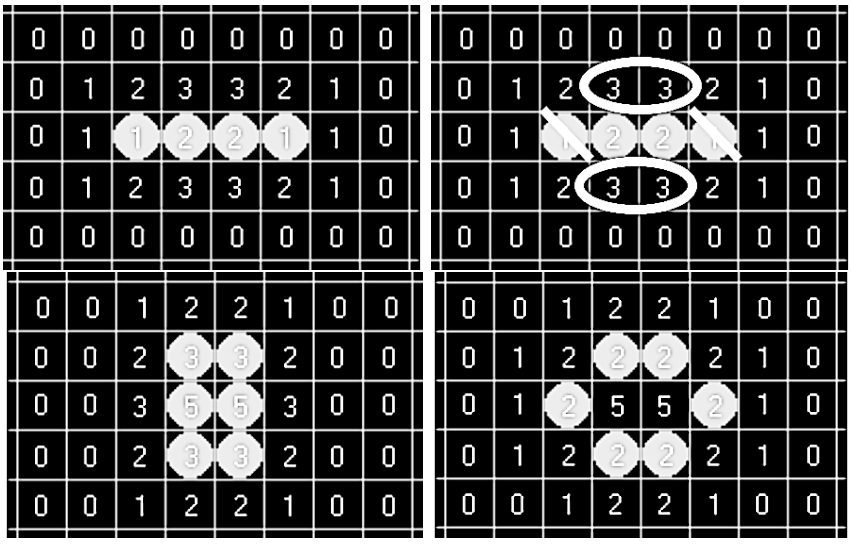


Fig.2 - The Game of life elementary rules

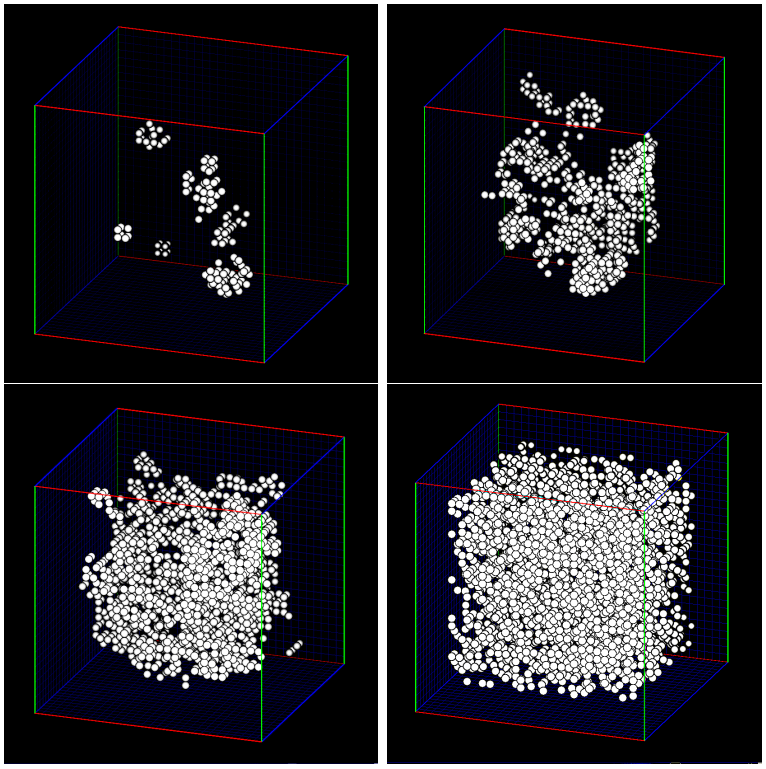


Fig.3 - Generation steps of a $3D$ cellular automaton (Golly 3.2)

VIEW ANIMATION (requires internet connection)

7.2 Fractals generated randomly by cellular automata

In the present chapter we are interested to apply the methods of *cellular automata* to the generation of *fractal structures* according to a *random process*. Instead of plotting each fractal image choosing each test point in a totally random way, any point will be generating randomly with a contiguity constraint like a *contiguous cell* of an *automaton*. The rule in order to decide if a nearby point is alive and which is its color will be decided by the respective fractal generation *law,* like, *e.g.,* $z_{n+1} = z_n^2 + c$ for *Mandelbrot* and *Julia* sets. In order to reach our goal we will suitably adapt the *Python 3* and *POV-Ray 3.7* codes presented in the previous chapters.

7.2.1 $2D$ Mandelbrot, Julia and Newton's method sets generated randomly by cellular automata

We can implement a *Python 3.7* algorithm for *cellular automata* generating *Mandelbrot, Julia* and *Newton's method* two-dimensional fractals according to the following strategy.[3]

- The generation process starts by choosing a random initial cell (plot as a small square or a small disk) in some point of a plane area.
- The recursion cycle, typical of the fractal one likes to build, is applied to the co-ordinates of the chosen point, in order to evaluate the related *escape rate*.
- A cell is painted, centered in the same point, the color of which corresponds, according to some color map, to the *escape rate*.
- A new point is chosen *randomly* near to the previous one, so that the cell centered in that point results contiguous to the previous cell. The allowed positions of the cells are determined by a suitable grid on the plane. The more refined is the grid, the higher the resolution of the image will result.

We will begin considering $2D$ fractal examples and later $3D$ fractals.

Python 3 codes to generate Figs 4, 5 and 6

```
############################################################
# 2D Mandelbrot set generation by a cellular automaton
# (graphics module)
############################################################

# mod graphics in /Users/strumia/Library/Python/3.6/site-packages/graphics/
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")
from graphics import *      # import graphics module

Radius = 10    # set escape rate threshold
```

[3]The algorithm can be improved in order to avoid that an already plotted cell is plotted again, but such a condition will render the code more complicated and perhaps not so much faster.

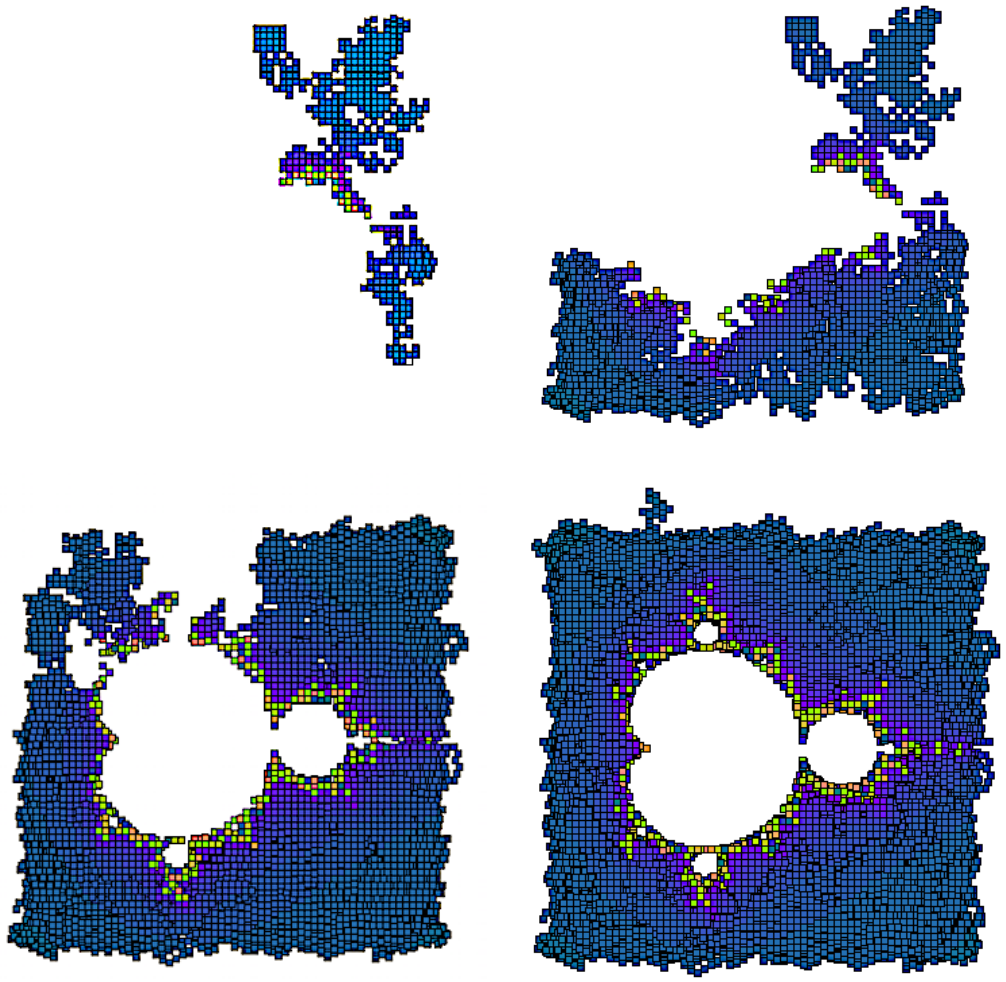


Fig.4 - Rough scheme of a cellular automaton randomly generating a Mandelbrot set

VIEW ANIMATION (requires internet connection)

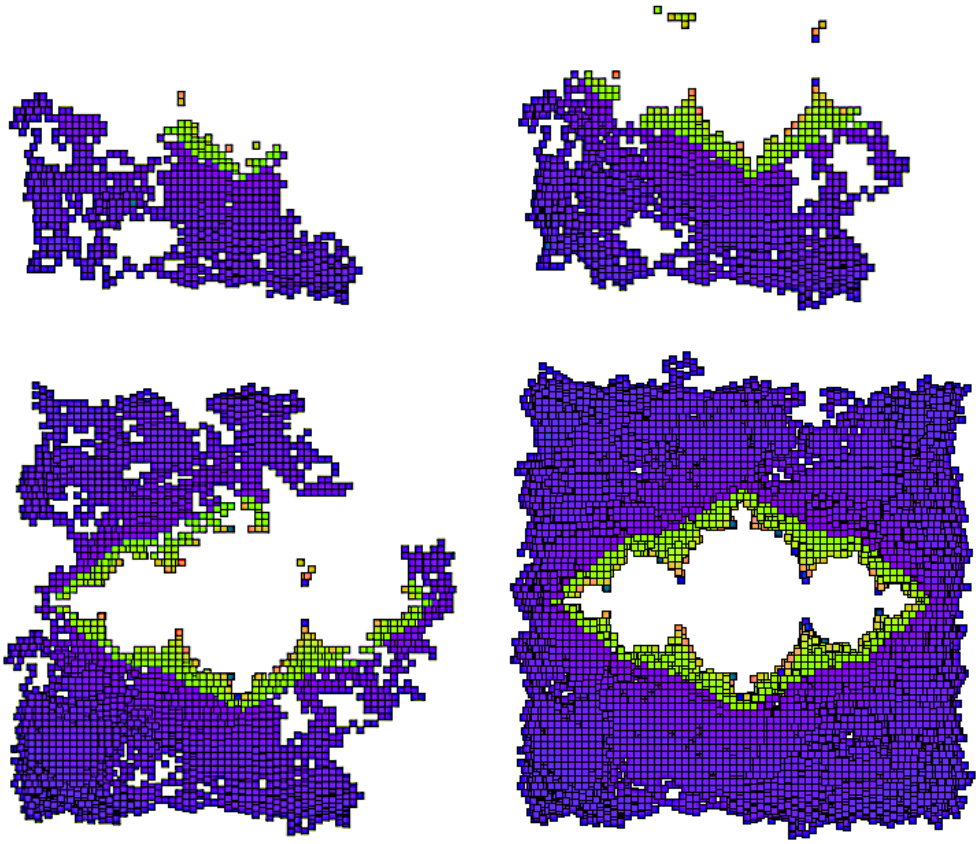


Fig.5 - Rough scheme of a cellular automaton randomly generating a Julia set ($c = 0.7454294$)

VIEW ANIMATION (requires internet connection)

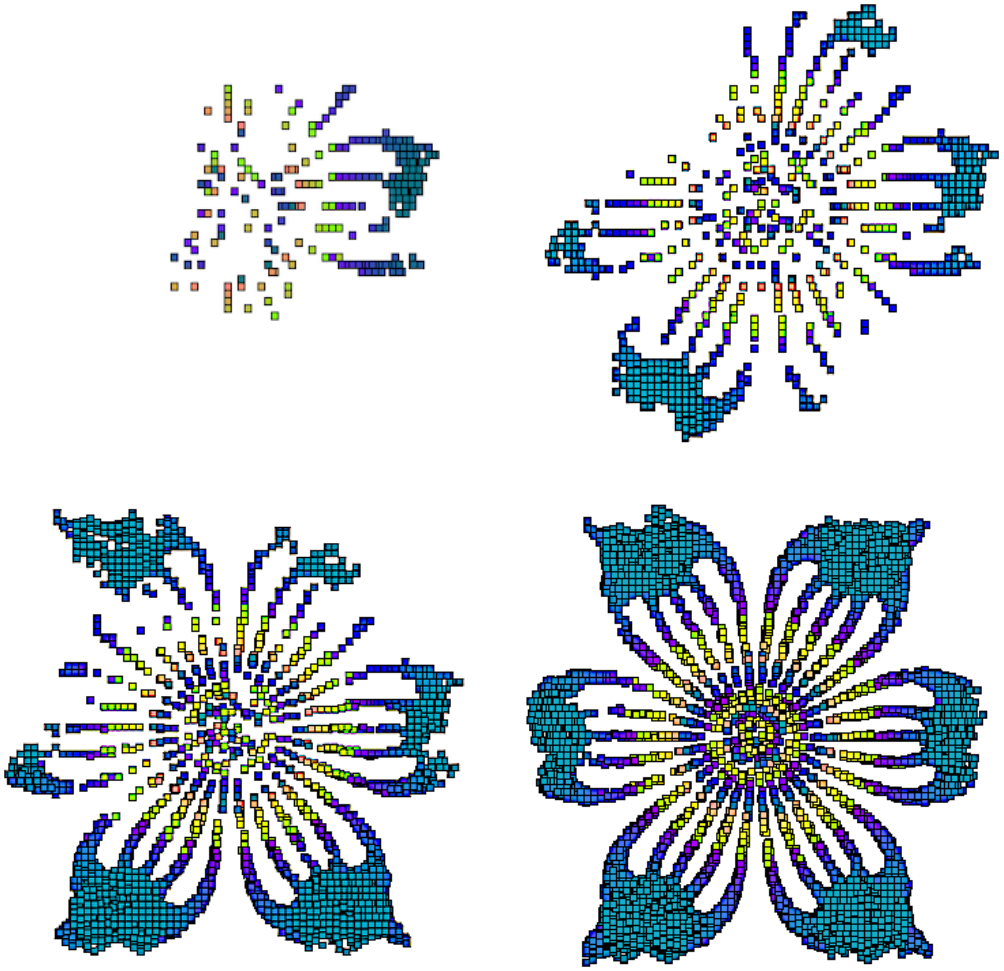


Fig.6 - Rough scheme of a cellular automaton randomly generating a Newton's method set $[f(z) = z^6 + 1]$

VIEW ANIMATION (requires internet connection)

```
Cx = .5     # set initial x parameter shift
Cy = 0.0   # set initial y parameter shift
Side = 1.3     # set square area side
M = 300   # set side number of elementary squares
N = 1   # set color map scale factor
Num = 256*N   # set number of cycles
sT=5    #  set step jump
w = 0   # set starting escape modulus value

import random  # import random module

p = M/2+random.randrange(-M/2,M/2)
q = M/2+random.randrange(-M/2,M/2)

win = GraphWin("Mandelbrot set",int(5*M/3),int(5*M/3))

def rectCol(p,q,w):
   Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
   Point(int(p+sT/2),int(q+sT/2)))
   Rect.draw(win).setFill(color_rgb(int(10*w%255),
   int((128-10*w)%255),int((128+10*w)%255)))

i = 1   # set non-zero index value
while i > 0:    # set random co-ordinates choice cycles
   p = p+random.randrange(-1,2)*sT
   q = q+random.randrange(-1,2)*sT
   if p < 0:
      p = p+2
   elif p > M:
     p = p-2

   if q < 0:
      q = q+2
   elif q > M:
      q = q-2

   Incx = Cx - Side + 2*Side/M*q
   Incy = Cy - Side + 2*Side/M*p
   x = 0.0
   y = 0.0
   w = 0
   for n in range(1,Num):
      xx = x*x - y*y - Incx
      yy = 2*x*y - Incy
      x = xx
      y = yy
      if x*x + y*y > Radius:    # escape rate condition
         w = n/N
         rectCol(int(M/3+q),int(M/3+p),int(w))
         break

win.getMouse()
win.close()


#####################################################################
# 2D Julia set generation by a cellular automaton (c = 0.7454294)
# (graphics module)
#####################################################################

# mod graphics in /Users/strumia/Library/Python/3.6/site-packages/graphics/
import sys
```

```
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")
from graphics import *      # import graphics module

Radius = 30   # set escape rate threshold
Cx = 0.7454294     # set c  parameter real part value
Cy = 0.0      # set c  parameter real part value
Side = 1.7     # set square area side
M = 300   # set side number of elementary squares
N = 1   # set color map scale factor
Num = 256*N   # set number of cycles
sT=5    #  set step jump
w = 0   # set starting escape modulus value
x = 0.0  #set x co-ordinate initial value
y = 0.0  #set y co-ordinate initial value

import random   # import random module

p = M/2+random.randrange(-M/2,M/2)
q = M/2+random.randrange(-M/2,M/2)

win = GraphWin("Julia set", int(5*M/3),int(5*M/3))

def rectCol(p,q,w):
   Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
   Point(int(p+sT/2),int(q+sT/2)))
   Rect.draw(win).setFill(color_rgb(int(10*w%255),
   int((128-10*w)%255),int((128+10*w)%255)))

# Alternative values Cx 0.1747, 0.1747 Cy -.072,-1.072
# Side 0.0015, 0.00015 Num 1024

i = 1   # set non-zero index value
while i > 0:    # set random co-ordinates choice cycles
   p = p+random.randrange(-1,2)*sT
   q = q+random.randrange(-1,2)*sT
   if p < 0:
     p = p+2
   elif p > M:
     p = p-2

   if q < 0:
     q = q+2
   elif q > M:
     q = q-2

   Incx = - Side + 2*Side/M*q
   Incy = - Side + 2*Side/M*p
   x = Incx
   y = Incy
   w = 0
   for n in range(1,Num):
      xx = x*x - y*y - Cx
      yy = 2*x*y - Cy
      x = xx
      y = yy
      if x*x + y*y > Radius:    # escape rate condition
         w = n/N
         rectCol(int(M/3+q),int(M/3+p),int(w))
         break

win.getMouse()
win.close()
```

```
################################################################
# 2D Newton's method set generation by a cellular automaton
# (graphics module)
################################################################

# mod graphics in /Users/strumia/Library/Python/3.6/site-packages/graphics/
import sys
sys.path.append("/Users/strumia/Library/Python/3.6/site-packages/graphics/")
from graphics import *      # import graphics module

Radius = .5     # set escape rate threshold
Cx = 0.0    # set initial x parameter shift
Cy = 0.0    # set initial y parameter shift
Side = .8       # set square area side
M = 300   # set side number of elementary squares
N = 1   # set color map scale factor
Num = 256*N   # set number of cycles
sT= 5    #  set step jump
w = 0   # set starting escape modulus value

import random    # import random module
p = M/2+random.randrange(-M/2,M/2)
q = M/2+random.randrange(-M/2,M/2)

win = GraphWin("Mandelbrot set", int(5*M/3),int(5*M/3))

def rectCol(p,q,w):
   Rect = Rectangle(Point(int(p-sT/2),int(q-sT/2)),
   Point(int(p+sT/2),int(q+sT/2)))
   Rect.draw(win).setFill(color_rgb(int(10*w%255),
   int((128-10*w)%255),int((128+10*w)%255)))

i = 1   # set non-zero index value
while i > 0:    # set random co-ordinates choice cycles
   p = p+random.randrange(-1,2)*sT
   q = q+random.randrange(-1,2)*sT
   if p < 0:
       p = p+2
   elif p > M:
      p = p-2

   if q < 0:
      q = q+2
   elif q > M:
      q = q-2

   Incx = - Side + 2*Side/M*q
   Incy = - Side + 2*Side/M*p
   x = Incx
   y = Incy
   w = 0
   for n in range(1,Num):
      xx = 5*x/6.0 - x*(x*x*x*x - 10*x*x*y*y +5*y*y*y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/6.0
     yy = 5*y/6.0 + y*(5*x*x*x*x - 10*x*x*y*y + y*y*y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/(x*x+y*y)/6.0
    x = xx
    y = yy
    if (x-Cx)*(x-Cx) + (y-Cy)*(y-Cy) < Radius:     # escape rate condition
       w = n/N
       rectCol(int(M/3+q),int(M/3+p),int(w))
```

```
win.getMouse()
win.close()
```

7.2.2 $3D$ Mandelbrot and Julia sets generated randomly by cellular automata

Generation of $3D$ fractals by cellular automata implementing a *POV-Ray 3.7* code requires some care. In fact, the code, needs to test sequentially first of all if each point in a suitable interval of $3D$ space belongs the fractal set or it does not, so obtaining a $3D$ matrix a values. Then the co-ordinates of each point belonging to the fractal set have to be extracted starting from an initial condition (*e.g.,* the origin of the $3D$ interval), in such a way that each point is contiguous to the previous one. In other words the difference between each co-ordinate of the $(n+1)$-*th* point from the co-ordinates of the n-*th* point is either ± 1 or 0.

Cellular automata generating quaternion fractal sets

The following images show, each one, six steps of generation of respectively:

- $3D$ quaternion Mandelbrot-Mandelbrot-Mandelbrot fractal set
- $3D$ quaternion Julia-Julia-Julia fractal set (C = -0.7454294)
- $3D$ quaternion Julia-Mandelbrot-Mandelbrot fractal set (C = -0.7454294)

The related *POV-Ray 3.7* code is also provided.

Pov-Ray 3.7 codes to generate figs 7, 8, 9

```
//=========================================================
// 3D quaternion Mandelbrot-Mandelbrot-Mandelbrot set generation
// by a cellular automaton (POV-Ray 3.7)
//=========================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}
background { color rgb <00,0.0,0.0> }

camera {
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
}

light_source {
< -20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

light_source {
```

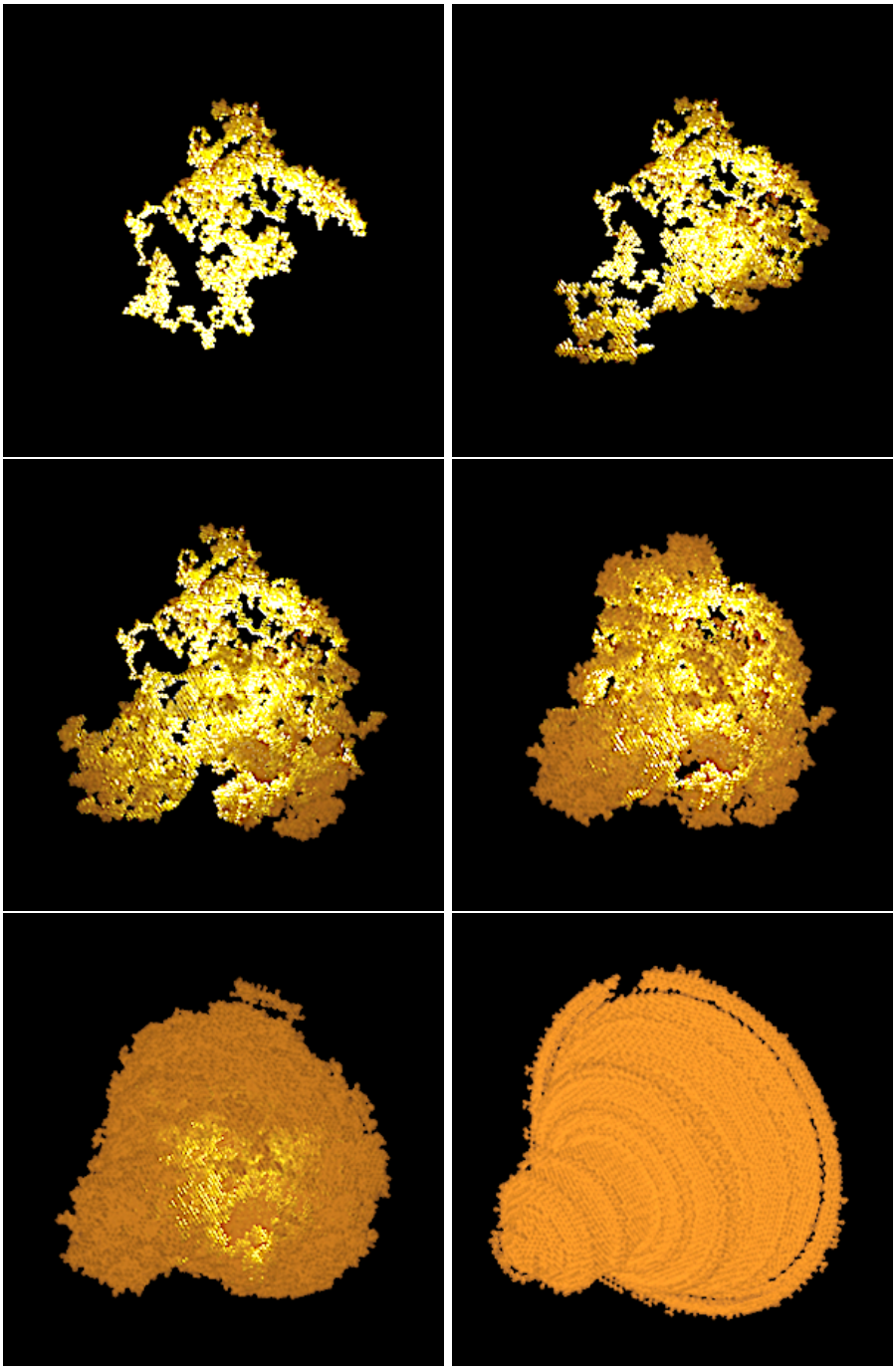


Fig.7 - POV 3.7 3*D* Quaternion Mandelbrot-Mandelbrot-Mandelbrot set generated by a cellular automaton

VIEW ANIMATION (requires internet connection)

```
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 3.0
}

#declare R =  0.1;     // set radius value
#declare L = 2;  // set square area side
#declare n = 200;     // set number of pixels per area side
#declare st = 1;   // set increment step
#declare N = 40;    // set number of cycles
#declare Th = -45;   // set rotation angles
#declare Ph = 30;

#declare K = array[2*n+1][2*n+1][2*n+1];   // escape rate 3D matrix
  #for (p, -n, n, st)
  #declare IncX = p*L/n;   // Mand1 X increment
  #for (q, -n, n, st)
  #declare IncY = q*L/n;    // Mand1 Y1 increment
    #for (r, -n, n, st)
      #declare IncZ = r*L/n;  // Mand2 Y2 increment
      #declare X = 0;   // start MandX
      #declare Y = 0;   // start MandY
      #declare Z = 0;   // start MandZ
      #declare K[p+n][q+n][r+n] = 0;

     #for (k,0,N)
       #declare XX = X*X - Y*Y - Z*Z + IncX;  // cycle MandX
       #declare YY = 2*X*Y + IncY;  // cycle MandY
       #declare ZZ = 2*X*Z + IncZ;   // cycle MandZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;
       #declare W = X*X +Y*Y + Z*Z;

caption(W < R)     // escape if   0.0015
       #declare K[p+n][q+n][r+n] = k;
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r

#declare Rnd_1 = seed (1153);
#declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (876);
#declare pp = 0;
#declare qq = 0;
#declare rr = 0;

union{
#for(j,0,pow(n,3))    // partial values n*n/4, n*n, n*n*n/64, n*n*n/16, n*n*n
   #declare p = pp + st*pow(-1,int(n*rand(Rnd_1)));
   #declare q = qq + st*pow(-1,int(n*rand(Rnd_2)));
   #declare r = rr + st*pow(-1,int(n*rand(Rnd_3)));

   #if ( abs(p) < n)
   #if( abs(q) < n)
   #if (abs(r) < n)
   #if(K[p+n][q+n][r+n] > 0)
   #declare pp = p;
   #declare qq = q;
   #declare rr = r;
    sphere {
```

```
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

translate < 30, 15, 0 >
rotate < 0, Th, Ph > }
#end // end if
#end // end if
#end // end if
#end // end if


//=========================================================
// 3D quaternion Julia-Julia-Julia set generation
// by a cellular automaton (POV-Ray 3.7)
//=========================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}
background { color rgb <00,0.0,0.0> }

camera {
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
}

light_source {
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

#declare R =  .5;      // set radius value
#declare L = 2;   // set square area side
#declare n = 150;     // set number of pixels per area side
#declare st = 1;   // set increment step
#declare Nr = 30;   // set number of cycles
#declare Th = 0;   // set rotation angles
#declare Ph = 30;
#declare Cx  = -0.7454294;   // JuliaX parameter
#declare Cy = 0;  // JuliaY parameter
#declare Cz  = 0;   // JuliaZ parameter

#declare K = array[2*n+1][2*n+1][2*n+1];   // escape rate 3D matrix
     #for (p, -n, n, st)
     #declare IncX = p*L/n;   // JuliaX increment
     #for (q, -n, n, st)
     #declare IncY = q*L/n;    // JuliaY increment

    #for (r, -n, n, st)
    #declare IncZ = r*L/n;  // JuliaZ increment
```

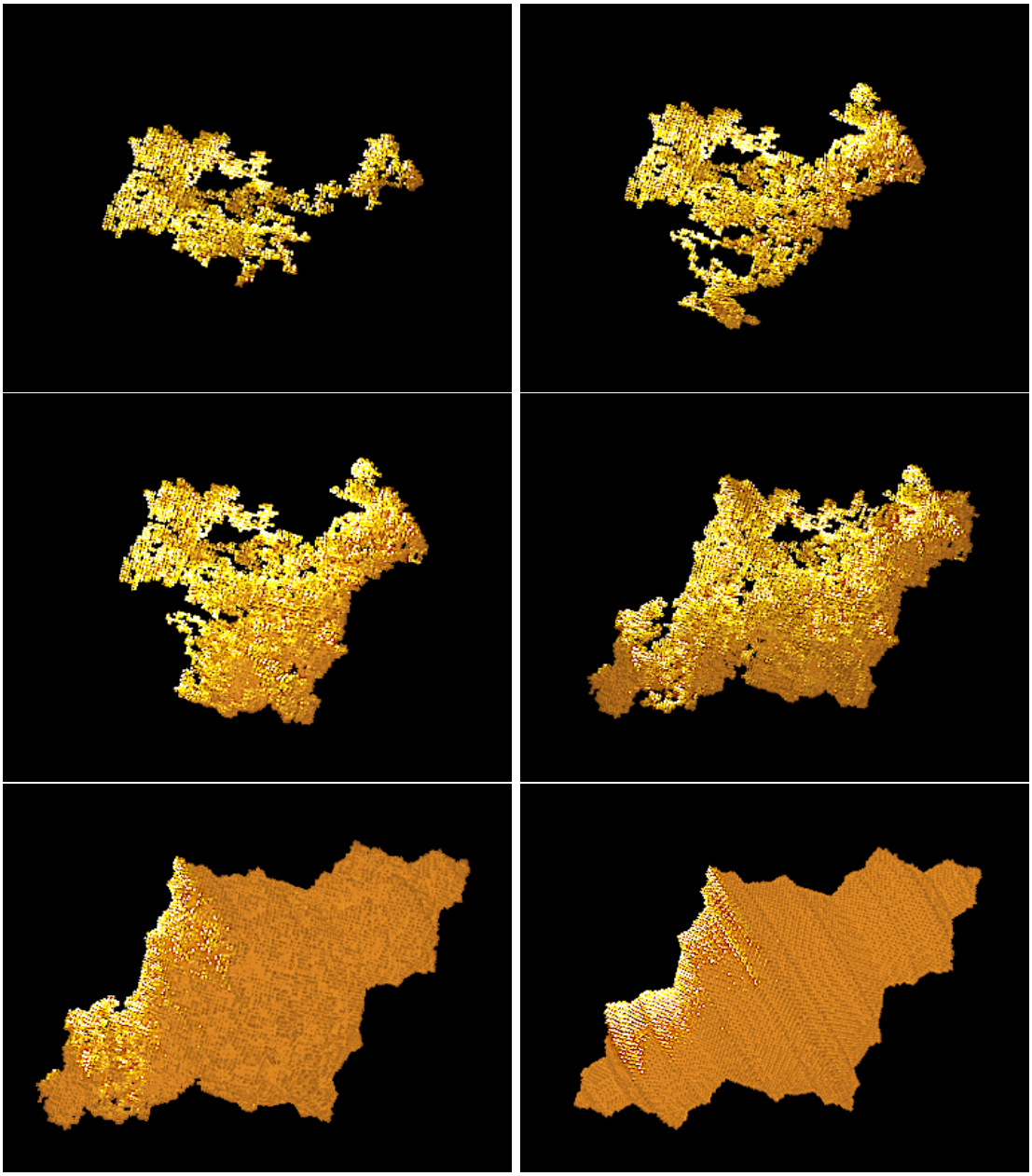


Fig.8 - POV 3.7 3*D* Quaternion Julia-Julia-Julia set
generated by a cellular automaton

VIEW ANIMATION (requires internet connection)

```
      #declare X = IncX;   // start JuliaX
      #declare Y = IncY;   // start JuliaY
      #declare Z = IncZ;   // start  JuliaZ
      #declare K[p+n][q+n][r+n] = 0;

      #for (k,0,Nr)
       #declare XX = X*X - Y*Y - Z*Z + Cx;   // cycle JuliaX
       #declare YY = 2*X*Y + Cy;              // cycle JuliaY
       #declare ZZ = 2*X*Z + Cz;              // cycle JuliaZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;

       #declare W = X*X +Y*Y + Z*Z;

   #if (W < R)     // escape if
   #declare K[p+n][q+n][r+n] = k;
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r

#declare Rnd_1 = seed (1153);
#declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (876);

#declare pp = 0;
#declare qq = 0;
#declare rr = 0;

union{ #for(j,0,n*n*n/16)  // partial  n*n/4, n*n, n*n*n/64, n*n*n/16,   n*n*n
#declare p = pp + st*pow(-1,int(n*rand(Rnd_1)));
#declare q = qq + st*pow(-1,int(n*rand(Rnd_2)));
 #declare r = rr + st*pow(-1,int(n*rand(Rnd_3)));

 #if ( abs(p) < n)
 #if( abs(q) < n)
 #if (abs(r) < n)
 #if(K[p+n][q+n][r+n] > 0)
    #declare pp = p;
    #declare qq = q;
    #declare rr = r;
    sphere {
     < p, q, r >, 1
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

translate < 0, -10, 0 >
  rotate < 0, Th, Ph > }

  #end // end if
  #end // end if
  #end // end if
  #end // end if
  #end}  // end for j
```

```
//=========================================================
// 3D quaternion Julia-Mandelbrot-Mandelbrot (C = -0.7454294)
//  set generation by a cellular automaton (POV-Ray 3.7)
//=========================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}
background { color rgb <00,0.0,0.0> }

camera {
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
}

light_source {
< -20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

light_source {
< 30, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 3.0
}

#declare R =  .5;      // set radius value
#declare L = 2;   // set square area side
#declare Cx = -0.7454294;
#declare Cz = 0;
#declare n = 150;     // set number of pixels per area side
#declare st = 1;   // set increment step
#declare N = 40;    // set number of cycles
#declare Th = -45;   // set rotation angles
#declare Ph = 30;

#declare K = array[2*n+1][2*n+1][2*n+1];   // escape rate 3D matrix
 #for (p, -n, n, st)
     #declare IncX = p*L/n;   // Julia X increment

    #for (q, -n, n, st)
       #declare IncY = q*L/n;    // Mand Y increment

    #for (r, -n, n, st)
    #declare IncZ = r*L/n;  // Julia Y increment

      #declare X = IncX;   // start JuliaX
      #declare Y = 0;   // start MandY
      #declare Z = 0;   // start MandZ
      #declare K[p+n][q+n][r+n] = 0;

     #for (k,1,N)
       #declare XX = X*X - Y*Y - Z*Z + Cx;  // cycle JuliaX
       #declare YY = 2*X*Y + IncY;  // cycle MandY
       #declare ZZ = 2*X*Z + IncZ;   // cycle JuliaZ
       #declare X = XX;
       #declare Y = YY;
       #declare Z = ZZ;
       #declare W = X*X +Y*Y + Z*Z;

      #if (W < R)     // escape if
      #declare K[p+n][q+n][r+n] = k;
```

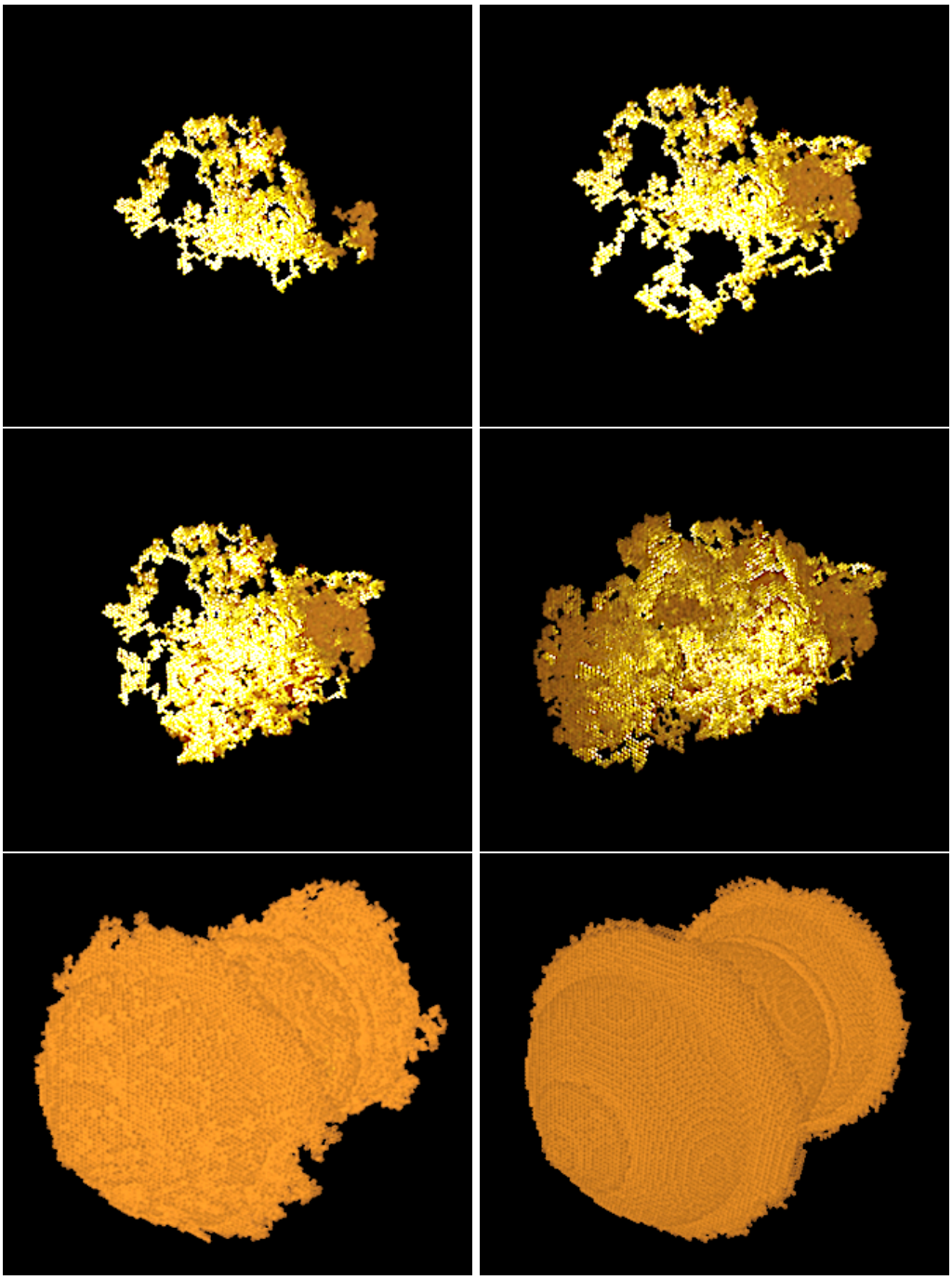


Fig.9 - POV 3.7 3*D* Quaternion Julia-Mandelbrot-Mandelbrot set
set (C = -0.7454294) generated by a cellular automaton

```
  #end  // end if
  #end  // end for k
  #end  // end for q
  #end  // end for p
  #end  // end for r

  #declare Rnd_1 = seed (1153);
 #declare Rnd_2 = seed (553);
 #declare Rnd_3 = seed (876);

 #declare pp = 0;
 #declare qq = 0;
 #declare rr = 0;

// repalce   n*n*n/8   by    n*n*n/8*clock   for animation
  union{#for(j,0,n*n*n/8) // partial values n*n/4, n*n/2, n*n, n*n*n/64,
   n*n*n/8, n*n*n

  #declare p = pp + st*pow(-1,int(n*rand(Rnd_1)));
   #declare q = qq + st*pow(-1,int(n*rand(Rnd_2)));
   #declare r = rr + st*pow(-1,int(n*rand(Rnd_3)));

   #if ( abs(p) < n)
    #if( abs(q) < n)
    #if (abs(r) < n)
 #if(K[p+n][q+n][r+n] > 0)
  #declare pp = p;
  #declare qq = q;
  #declare rr = r;
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

    translate < 30, 15,10 >
  rotate < Th, Th, Ph > }
 #end // end if
  #end // end if
  #end // end if
  #end // end if
    #end}  // end for j
```

Cellular automata generating double complex fractal sets

The further following images show, each one, six steps of generation of respectively:

- $3D$ double complex Mandelbrot-Mandelbrot-Mandelbrot fractal set
- $3D$ double complex Julia-Julia-Julia fractal set ($c_{1x} = c_{2x} = -0.7454294$)
- $3D$ double complex Julia-Mandelbrot-Mandelbrot fractal set ($c_x = -0.7454294$)

And the related *POV-Ray 3.7* code is also given.

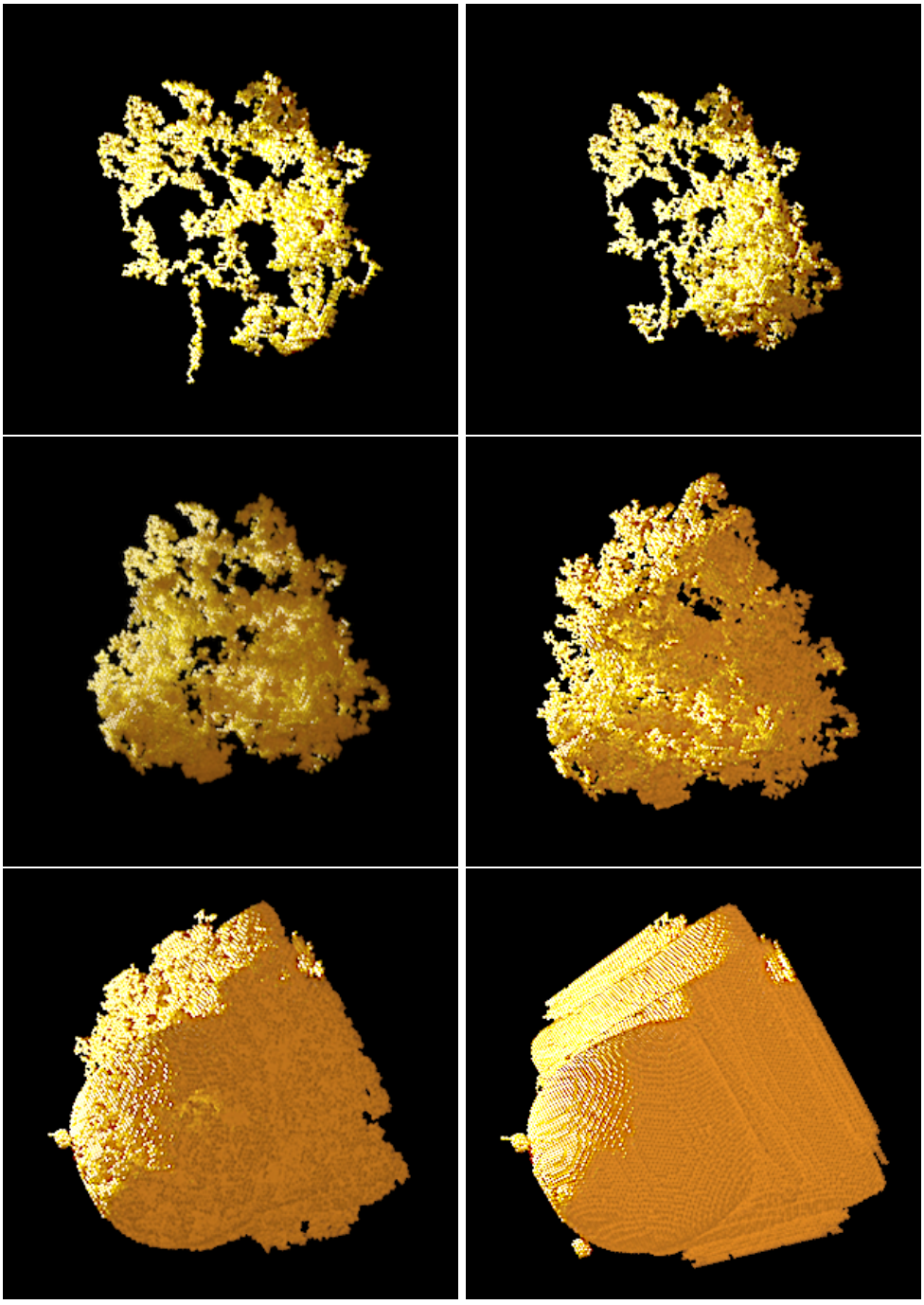


Fig.10 - POV 3.7 3*D* Doble complex Mandelbrot-Mandelbrot-Mandelbrot set generated by a cellular automaton

VIEW ANIMATION (requires internet connection)

Pov-Ray 3.7 codes to generate figs 10, 11, 12

```
//=========================================================
// 3D double complex Mandelbrot-Mandelbrot-Mandelbrot set generation
// by a cellular automaton (POV-Ray 3.7)
//=========================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}
background { color rgb <00,0.0,0.0> }

camera {
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
}

light_source {
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

#declare R =  0.5;      // set radius value
#declare L = 2;   // set square area side
#declare X1 = 0;   // set co-ordinates x initial value
#declare X2 = 0;
#declare Y1 = 0;   // set co-ordinates y initial value
#declare Y2 = 0;
#declare n = 150;     // set number of pixels per area side
#declare st = 1;   // set increment step
#declare Nr = 20;    // set number of cycles
#declare Th = -45;  // set rotation angles
#declare Ph = 30;

#declare K = array[2*n+1][2*n+1][2*n+1];   // escape rate 3D matrix
   #for (p, -n, n, st)
   #declare IncX = p*L/n;   // Mand1 X increment
   #for (q, -n, n, st)
   #declare IncY1 = q*L/n;    // Mand1 Y1 increment

   #for (r, -n, n, st)
   #declare IncY2 = r*L/n;  // Mand2 Y2 increment
   #declare X1 = 0;   // start Mand1
   #declare X2 = 0;   // start Mand1
   #declare Y1 = 0;   // start Mand1
   #declare Y2 = 0;   // start Mand2
   #declare K[p+n][q+n][r+n] = 0;

   #for (k,0,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + IncX;  // cycle Mand1
       #declare XX2 = X2*X2 - Y2*Y2 + IncX;  // cycle Mand1
       #declare YY1 = 2*X1*Y1 + IncY1;   // cycle Mand1
       #declare YY2 = 2*X2*Y2 + IncY2;   // cycle Mand2
```

```
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

     #if (W < R)     // escape if
     #declare K[p+n][q+n][r+n] = k;
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r

 #declare Rnd_1 = seed (1153);
 #declare Rnd_2 = seed (553);
 #declare Rnd_3 = seed (876);

 #declare pp = 0;
 #declare qq = 0;
 #declare rr = 0;

 // replace n*n*n by n*n*n*clock  for animation
union{
#for(j,0,n*n)    // partial values n*n/2, n*n, 2*n*n, 3*n*n, 4*n*n, 16*n*n,  n*n*n

   #declare p = pp + st*pow(-1,int(n*rand(Rnd_1)));
   #declare q = qq + st*pow(-1,int(n*rand(Rnd_2)));
   #declare r = rr + st*pow(-1,int(n*rand(Rnd_3)));

   #if ( abs(p) < n)
    #if( abs(q) < n)
    #if (abs(r) < n)
   #if(K[p+n][q+n][r+n] > 0)
   #declare pp = p;
   #declare qq = q;
   #declare rr = r;
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

translate < 80, -10, 0 >
rotate < 0, Th, Ph > }
#end // end if
#end // end if
#end // end if
#end // end if
#end}  // end for j
```

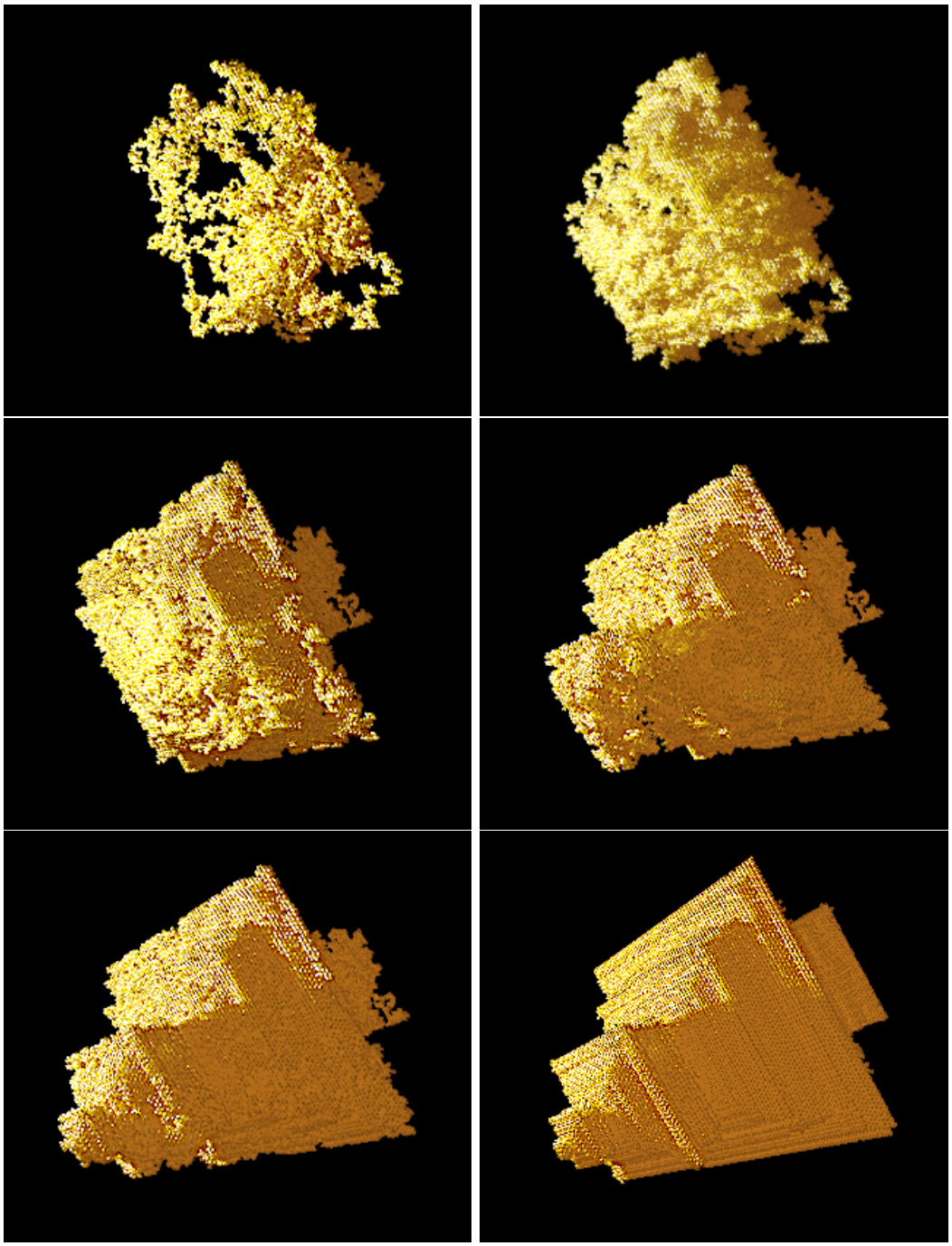


Fig.11 - POV 3.7 $3D$ Double complex Julia-Julia-Julia set ($c_{1x} = c_{2x} = -0.7454294$) generated by a cellular automaton

```
//=======================================================
// 3D double complex Julia-Julia-Julia set
// (c1x = c2x = -0.7454294)
// generation by a cellular automaton (POV-Ray 3.7)
//=======================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
global_settings {assumed_gamma 1.0}
background { color rgb <00,0.0,0.0> }

camera {
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
}

light_source {
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

#declare R =  .5;      // set radius value
#declare L = 2;   // set square area side
#declare X1 = 0;   // set co-ordinates x initial value
#declare X2 = 0;
#declare Y1 = 0;   // set co-ordinates y initial value
#declare Y2 = 0;
#declare Cx1 = -0.7454294;
#declare Cx2 = -0.7454294;
#declare Cy1 = 0;
#declare Cy2 = 0;
#declare n = 150;     // set number of pixels per area side
#declare st = 1;   // set increment step
#declare Nr = 20;    // set number of cycles
#declare Th = -30;   // set rotation angles
#declare Ph = 30;

#declare K = array[2*n+1][2*n+1][2*n+1];   // escape rate 3D matrix

#for (p, -n, n, st)
     #declare IncX = p*L/n;   // Mand1 X increment
     #for (q, -n, n, st)
     #declare IncY1 = q*L/n;    // Mand1 Y1 increment
     #for (r, -n, n, st)
     #declare IncY2 = r*L/n;  // Mand2 Y2 increment
     #declare X1 = IncX;   // start Mand1
     #declare X2 = IncX;   // start Mand1
     #declare Y1 = IncY1;   // start Mand1
     #declare Y2 = IncY2;   // start Mand2
     #declare K[p+n][q+n][r+n] = 0;

       #for (k,1,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + Cx1;  // cycle Mand1
       #declare XX2 = X2*X2 - Y2*Y2 + Cx2;  // cycle Mand1
       #declare YY1 = 2*X1*Y1 + Cy1;   // cycle Mand1
       #declare YY2 = 2*X2*Y2 + Cy2;   // cycle Mand2
```

```
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

#if (W < R + 0.02)     // escape if
#declare K[p+n][q+n][r+n] = k;
#end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
#end  // end for r

#declare Rnd_1 = seed (1153);
#declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (876);

#declare pp = 0;
#declare qq = 0;
#declare rr = 0;

union{ #for(j,0,2*n*n)    // partial values n*n, 2*n*n, 3*n*n, 8*n*n, n*n*n/8, n*n*n
#declare p = pp + st*pow(-1,int(n*rand(Rnd_1)));
#declare q = qq + st*pow(-1,int(n*rand(Rnd_2)));
#declare r = rr + st*pow(-1,int(n*rand(Rnd_3)));

   #if ( abs(p) < n)
   #if( abs(q) < n)
   #if (abs(r) < n)
   #if(K[p+n][q+n][r+n] > 0)
   #declare pp = p;
   #declare qq = q;
   #declare rr = r;
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

translate < 80, -10, 0 >
rotate < 0, Th, Ph > }
#end // end if
#end // end if
#end // end if
#end // end if
#end}  // end for j


//============================================================
// 3D double complex Julia-Mand-Mand set
// (c1x = c2x = -0.7454294)
// generation by a cellular automaton (POV-Ray 3.7)
//============================================================

#include "colors.inc"  // Standard Color definitions
#include "glass.inc"   //  Glass pigment effect
#include "metals.inc"   // Metal pigment effect
```

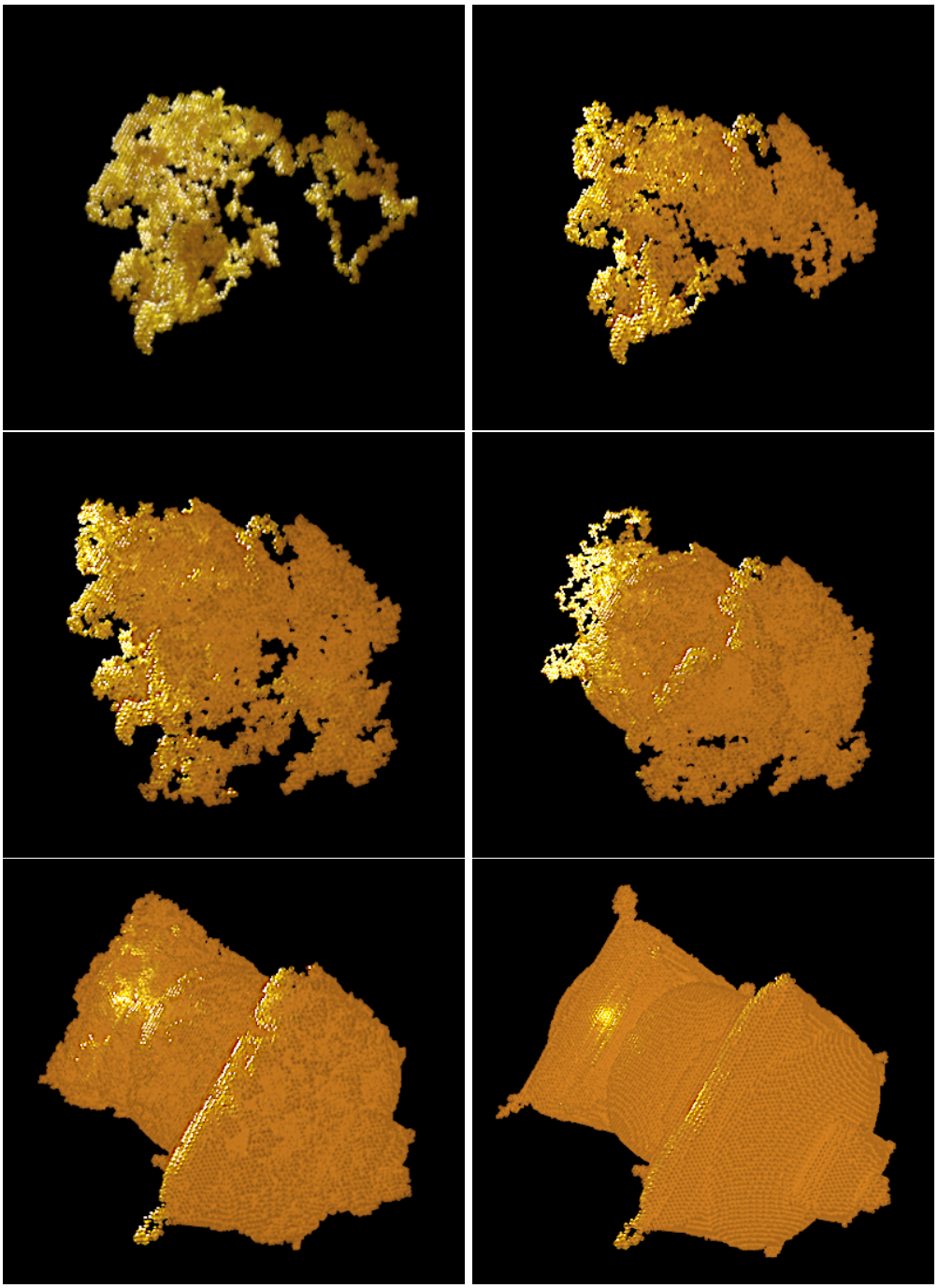


Fig.12 - POV 3.7 $3D$ Double complex Julia-Mand-Mand set ($c_{1x} = c_{2x} = -0.7454294$) generated by a cellular automaton

```
global_settings {assumed_gamma 1.0}
background { color rgb <00,0.0,0.0> }

camera {
    location <-20, 20, -300>
    look_at  <-5, 0,  0>
}

light_source {
< -50, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

light_source {
< 20, 20, -10>
rgb <1.000000, 1.000000, 1.000000> * 2.0
}

#declare R =  .5;      // set radius value
#declare L = 2;   // set square area side
#declare X1 = 0;   // set co-ordinates x initial value
#declare X2 = 0;
#declare Y1 = 0;   // set co-ordinates y initial value
#declare Y2 = 0;
#declare Cx = -0.7454294;
#declare n = 150;     // set number of pixels per area side
#declare st = 1;   // set increment step
#declare Nr = 20;    // set number of cycles
#declare Th = -45;   // set rotation angles
#declare Ph = 150;
#declare K = array[2*n+1][2*n+1][2*n+1];   // escape rate 3D matrix

    #for (p, -n, n, st)
    #declare IncX = p*L/n;   // Mand1 X increment
    #for (q, -n, n, st)
     #declare IncY1 = q*L/n;    // Mand1 Y1 increment
    #for (r, -n, n, st)
      #declare IncY2 = r*L/n;  // Mand2 Y2 increment

      #declare X1 = IncX;   // start Julia1
      #declare X2 = IncX;   // start Julia2
      #declare Y1 = 0;   // start Mand1
      #declare Y2 = 0;   // start Mand2
      #declare K[p+n][q+n][r+n] = 0;

     #for (k,1,Nr)
       #declare XX1 = X1*X1 - Y1*Y1 + Cx;  // cycle Julia1
       #declare XX2 = X2*X2 - Y2*Y2 + Cx;  // cycle Julia2
       #declare YY1 = 2*X1*Y1 + IncY1;   // cycle Mand1
       #declare YY2 = 2*X2*Y2 + IncY2;   // cycle Mand2
       #declare X1 = XX1;
       #declare X2 = XX2;
       #declare Y1 = YY1;
       #declare Y2 = YY2;
       #declare W = X1*X1 +X2*X2 +Y1*Y1 + Y2*Y2;

     #if (W < R + 0.02)     // escape if
      #declare K[p+n][q+n][r+n] = k;
      #end  // end if
#end  // end for k
#end  // end for q
#end  // end for p
```

```
#end  // end for r

#declare Rnd_1 = seed (1153);
#declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (876);

#declare pp = 0;
#declare qq = 0;
#declare rr = 0;

 union{ #for(j,0,n*n/2)    // partial values n*n/2, n*n, 2*n*n,
  n*n*n/32, 16*n*n, n*n*n

   #declare p = pp + st*pow(-1,int(n*rand(Rnd_1)));
   #declare q = qq + st*pow(-1,int(n*rand(Rnd_2)));
   #declare r = rr + st*pow(-1,int(n*rand(Rnd_3)));
   #if ( abs(p) < n)
    #if( abs(q) < n)
    #if (abs(r) < n)
  #if(K[p+n][q+n][r+n] > 0)
  #declare pp = p;
  #declare qq = q;
  #declare rr = r;
    sphere {
     < p, q, r >, 1   //  adding 3d axis
       texture {
       pigment { color Col_Glass_Yellow }
        }
        finish { ambient rgb <0.3,0.1,0.1>
        diffuse .3
       reflection .3
       specular 1 }   // plot sphere

rotate < 0, Th, Ph >
translate < 0, -15, -5 >}
#end // end if
#end // end if
#end // end if
#end // end if
#end}  // end for j
```

Chapter 8

Heart structure models from cellular automata

Sequential and random rendering processes

8.1 Introduction

All the previous chapters of our report are to be considered, in the same time, both as autonomous – since each one is self-consistent – and as introductory to the present chapter. In fact here we attempt an application of the previous methodology and results to biology. The aim of the whole report is that of showing how ordered material systems (*i.e.,* bodies of any kind) can be modeled as *attractors* towards which simpler elementary structures (particles, cells, etc.), even starting from random initial conditions, are driven by a suitable *information* (*law*, or sequence of *laws*). So order appears to arise from chance, while some suitable *information* is hidden within the random process itself.

In principle one could guess that the whole material universe may be modeled by a set of nested attractors generating one from the other according to a non-completely stochastic process. In the present chapter we will test, as an example, some models generating the ordered structure of the external surface of a human heart. As we will see very rough models can be obtained assigning a relatively compact mathematical law, while a more realistic model seems to require to provide the *uncompressed string* of all the components of the structure, as if the sequence of the respective numbers (co-ordinates of representative points of each elementary component) were non-computable.

8.2 Python 3 rendering of a heart-like external structure

8.2.1 The heart as a whole

We will attack the problem of modeling a complex *structure* starting from a simpler one of which we know a mathematical generating law. As a second step we will try to slightly

modify the known law in order to obtain a structure which results to be more similar to the required complex one. In order to observe the generation process of the complex structure we will need not to build it instantly *as a whole,* but *step by step,* placing *point after point,* or at least *small parts after small parts.*

Approaching a $2D$ heart profile (continuous paths)

We start with the simpler problem of approaching a $2D$ heart profile by modifying the parametric equations of a circular path:

$$x = \pm R \sin\theta, \qquad y = R\cos\theta, \quad \theta \in [0, \pi]. \tag{8.1}$$

A negative exponential deformation factor for y seems to be enough for our purposes:

$$x = \pm R \sin\theta, \qquad y = Re^{-a\theta}\cos\theta, \quad \theta \in [0, \pi]. \tag{8.2}$$

where a is a suitable damping factor. The sequence of deformations is shown as overlapped in fig. 1 and as separate images in fig. 2.

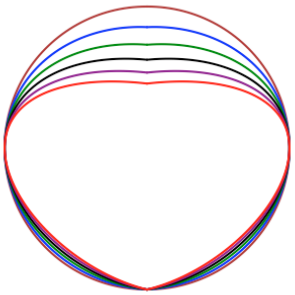


Fig.1 - Rough heart profile obtained by deformation of a circular path ($a = 0 \div .3$, overlapped profiles)

Python 3 codes to generate figs 1 and 2

```
########################################
# Heart 2D profile generation
# by deformation of a circular path
# overlapped images
# (matplotlib module)
########################################

import matplotlib as mpl
import numpy as np
import matplotlib.pyplot as plt
```

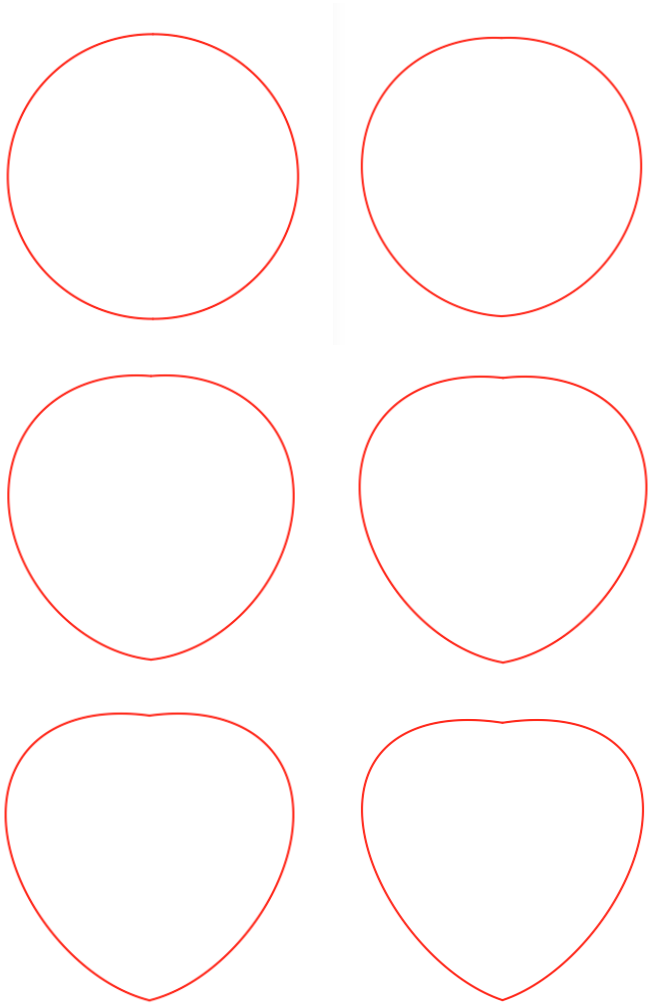


Fig.2 - Rough heart profile obtained by deformation sequence ($a = 0 \div .3$, sequential deformation steps)

```
r = 4.0
n = 100
theta = np.linspace(0.0, np.pi, n)

x = r * np.sin(theta)
y = r * np.cos(theta)
X1 = x + 0.0008*y*y
X2 = -x - 0.0008*y*y
Y = [0,0,0,0,0,0]
c = ['brown','blue','green','black','purple','red']

fig = plt.figure(figsize=(4,4), dpi=100)
for k in range(6):
   Y[k] = 2*r*np.cos(theta)*np.e**(-.3*(np.pi-theta)*k/6)

    plt.axis('off')
    plt.plot(X1, Y[k], c[k])
    plt.plot(X2, Y[k], c[k])
 # plt.plot(X1, Y[5], c[k])
 # plt.plot(X2, Y[5], c[k])

plt.show()


#######################################
# Heart 2D profile generation
# by deformation of a circular path
# sequential images
# (matplotlib module)
#######################################

import matplotlib as mpl
import numpy as np
import matplotlib.pyplot as plt

r = 4.0
n = 100
theta = np.linspace(0.0, np.pi, n)

x = r * np.sin(theta)
y = r * np.cos(theta)
X1 = x + 0.0008*y*y
X2 = -x - 0.0008*y*y

for k in range(6):
  Y = 2*r*np.cos(theta)*np.e**(-.3*(np.pi-theta)*k/6)
  fig = plt.figure(figsize=(4,4), dpi=100)
  plt.axis('off')
  plt.plot(X1, Y, 'r')
  plt.plot(X2, Y, 'r')

plt.show()
```

Approaching a $3D$ heart external surface (mesh plots)

Following a similar procedure we perform the passage from two to three dimensions.

a) In the case of a biological organ like a heart we could start, *e.g.,* with a simple spherical surface. In chapter**??** we have seen how to model and render a spherical shell, *e.g.,* recurring to *parametric equations*:

$$x = R\sin\theta\sin\phi, \qquad y = R\cos\theta\sin\phi, \qquad z = R\cos\phi, \tag{8.3}$$

as *information* (*law*) governing the process of its generation, starting from elementary components like dots or tiny spheres.

b) A second step will be that of modifying the parametric equations of the sphere so that we may obtain a suitable deformation of its surface, adapting its shape to a heart-like one. Of course the result will be rough when very irrelevant modification are applied, while it will be more refined if more appropriate changes will be imposed. Modified parametric equations as the following will be enough suitable for our didactical purposes, even if, of course, better ones can be found:[1]

$$\begin{aligned} x &= \pm R\sin\theta\sin\phi + ay^2, \\ y &= \pm b\,R\cos\theta\sin\phi, \qquad\qquad \theta,\phi \in [0,\pi], \\ z &= c\,R\cos\phi e^{d\phi}. \end{aligned} \tag{8.4}$$

Python 3 code to generate fig. 3

```
########################################
# Heart 3D shape generation
# by deformation of a spherical surface
# sequential mesh plots
# (matplotlib module)
########################################

from mpl_toolkits import mplot3d  # imports 3D matplotlib modulus
import numpy as np
import matplotlib.pyplot as plt

r = 4.0

theta = np.linspace(0.0, 2*np.pi, 40)
phi = np.linspace(0.0, np.pi, 40)
theta, phi = np.meshgrid(theta, phi)

x = r * np.sin(theta)*np.sin(phi)
y = r * np.cos(theta)*np.sin(phi)

for k in range(6):
 X = x + 0.0008*y*y*k/6
 Y = y - .3*y*k/6
 Z = 2*r*np.cos(phi)*np.e**(-.3*(np.pi-phi)*k/6)

 fig = plt.figure(figsize=(4,4), dpi=100)
 ax = plt.axes(projection='3d')
```

[1] An interesting challenge for the reader could be that of experiencing how to improve our equations.

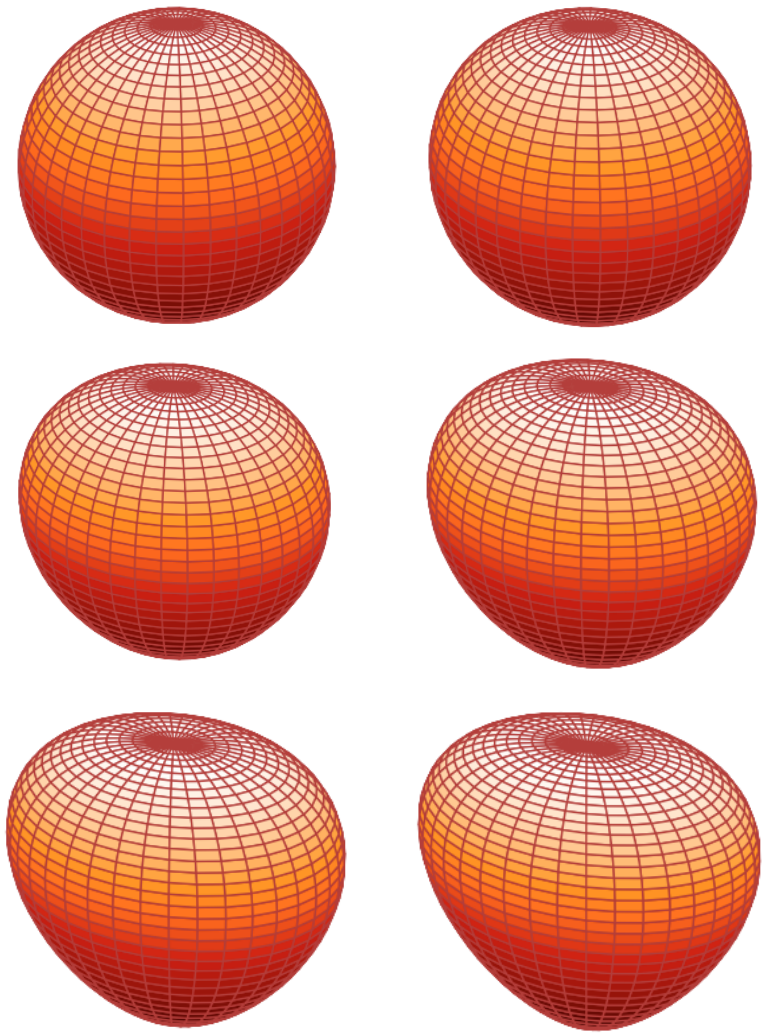


Fig.3 - Heart shape as a whole mesh plots obtained by deformation sequence of a sphere
$(a = 0 \div .0008, b = 1 \div .7, c = 2, d = 0 \div .3)$

```
    ax.set_xlim3d([-2.5, 2.5])
    ax.set_ylim3d([-2.5, 2.5])
    ax.set_zlim3d([-5.0*(1-k/24), 5.0*(1-k/24)])
    ax.set_axis_off()

    ax.plot_surface(X, Y, Z, cmap="gist_heat", edgecolor='brown')

plt.show()
```

c) A third step consists in obtaining a rendering of the heart-like structure by means of small spots, resembling biological cells, instead of usual $3D$ plot meshes. The smaller the spots are the more detailed and realistic the image results. Of course, in our model, we cannot attain to the true biological scale of cells respect to the model extension, since it would require so a huge number of dots that the computing time and power would be enormous.

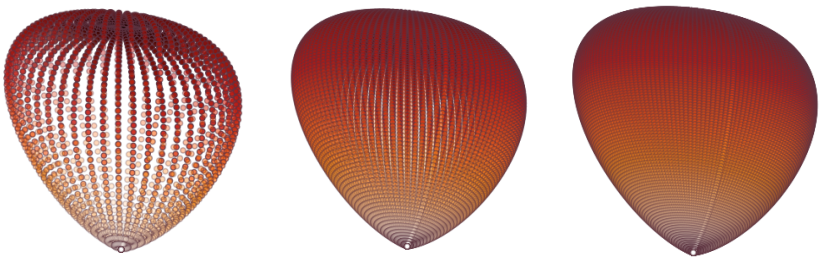


Fig.4 - Heart shape as a whole scatter plot sequence (n = 50, 100, 150 spots)

Python 3 code to generate fig. 4

```
########################################
# Heart 3D shape generation
# scatter (spot) image
# (matplotlib module)
########################################

from mpl_toolkits import mplot3d  # imports 3D matplotlib modulus

import numpy as np
import matplotlib.pyplot as plt

r = 5.0
n = 150  #alternative 50, 100

theta = np.linspace(0.0, 2*np.pi, n)
phi = np.linspace(0.0, np.pi, n)
theta, phi = np.meshgrid(theta, phi)

x = r * np.sin(theta)*np.sin(phi)
y = r * np.cos(theta)*np.sin(phi)
z = r*np.cos(phi)
```

```
X = x + 0.0008*y*y
Y = .7*y
Z = 2*z*np.e**(-.3*(np.pi-phi))

ax = plt.axes(projection='3d')

ax.set_xlim3d([-4.5, 4.5])
ax.set_ylim3d([-5.0, 5.0])
ax.set_zlim3d([-5.0, 4.0])
ax.set_axis_off()

C = np.arange(n*n)

ax.scatter(X, Y, Z, c=C, cmap = "gist_heat", marker="o",
edgecolor=[.3,.15,.2])

plt.show()
```

8.2.2 Sequential rendering of a heart-like model

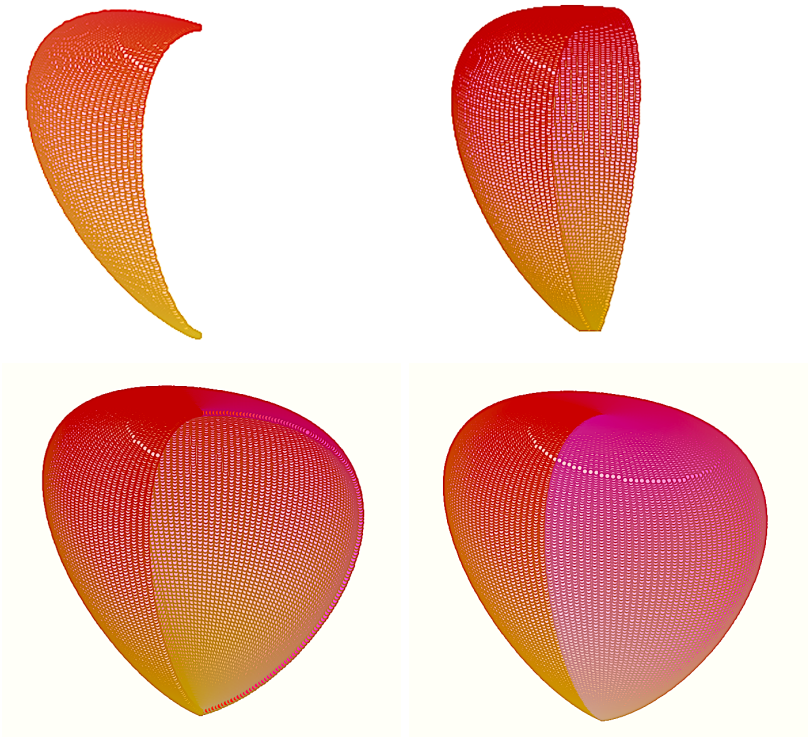


Fig.5 - Heart shape as a whole scatter plot sequence
(vertical sections: n = 50, 100, 150 spots)

Python 3 code to generate fig. 5

```
########################################
# Heart 3D scatter sequential generation
#  vertical sections
# (matplotlib module)
########################################

from mpl_toolkits import mplot3d  # imports 3D matplotlib modulus

import numpy as np
import matplotlib.pyplot as plt

r = 5.0
n = 150

theta = np.linspace(-np.pi, np.pi, n)
phi = np.linspace(0.0, np.pi, n)
theta, phi = np.meshgrid(theta, phi)

x = r * np.sin(theta)*np.sin(phi)
y = r * np.cos(theta)*np.sin(phi)
z = r*np.cos(phi)

X = x + 0.0008*y*y
Y = .7*y
Z = 2*z*np.e**(-.3*(np.pi-phi))

ax = plt.axes(projection='3d')

ax.set_xlim3d([-4.5, 4.5])
ax.set_ylim3d([-5.0, 5.0])
ax.set_zlim3d([-5.0, 4.0])
ax.set_axis_off()

C = np.arange(n*n)

for u in range(0,np.int(n/4),1):  #alternative n/4, n/2, 3*n/4, n
 for t in range(0,n,1):
  ax.scatter(X[t,u], Y[t,u], Z[t,u], s = 12,
  c = [1,.7*np.abs(np.cos(2*np.pi*(1+u/n))),
  np.abs(np.sin(np.pi*(1+t/n)))], marker="o",
  edgecolor=[.7,np.abs(t/255),np.abs(u/(300))] ,zorder=2)
  plt.pause(0.001)

plt.show()
```

Python 3 code to generate fig. 6

```
########################################
# Heart 3D scatter sequential generation
#  horizontal sections
# (matplotlib module)
########################################

from mpl_toolkits import mplot3d  # imports 3D matplotlib modulus

import numpy as np
```

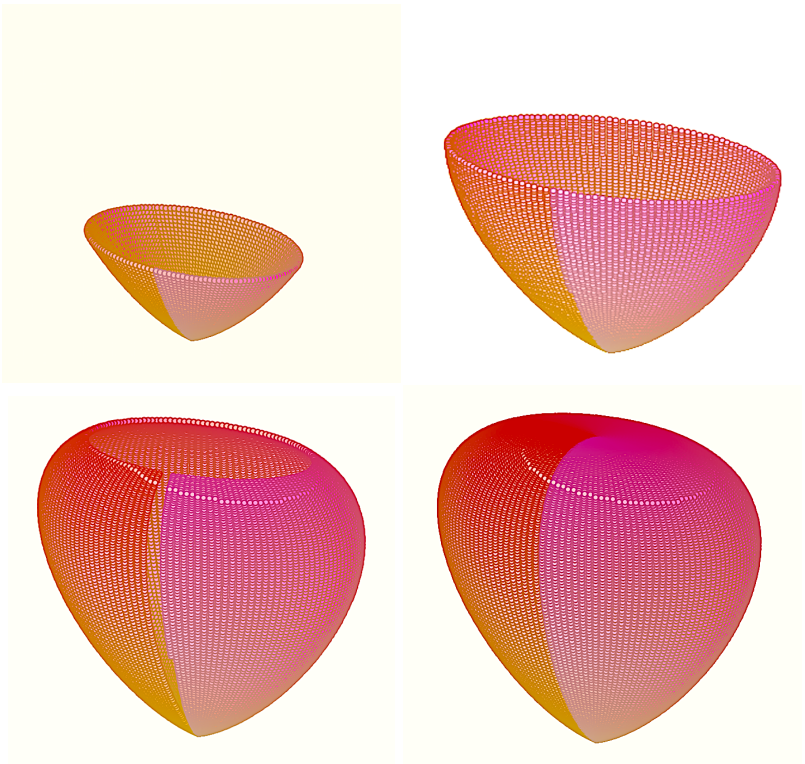


Fig.6 - Heart shape as a whole scatter plot sequence
(horizontal sections: n = 50, 100, 150 spots)

```
import matplotlib.pyplot as plt

r = 5.0
n = 150

theta = np.linspace(-np.pi, np.pi, n)
phi = np.linspace(0.0, np.pi, n)
theta, phi = np.meshgrid(theta, phi)

x = r * np.sin(theta)*np.sin(phi)
y = r * np.cos(theta)*np.sin(phi)
z = r*np.cos(phi)

X = x + 0.0008*y*y
Y = .7*y
Z = 2*z*np.e**(-.3*(np.pi-phi))

ax = plt.axes(projection='3d')

ax.set_xlim3d([-4.5, 4.5])
ax.set_ylim3d([-5.0, 5.0])
ax.set_zlim3d([-5.0, 4.0])
ax.set_axis_off()
C = np.arange(n*n)
```

```
for u in range(n-1,1,-1):
 for u in range(1,n,1):  #alternative  3*n/4, n/2, n/4, n
  ax.scatter(X[u,t], Y[u,t], Z[u,t], s = 12,
  c = [1,.7*np.abs(np.cos(2*np.pi*(1+u/n))),
  np.abs(np.sin(np.pi*(1+t/n)))], marker="o",
  edgecolor=[.7,np.abs(t/255),np.abs(u/(300))] ,zorder=2)
  plt.pause(0.001)
plt.show()
```

8.2.3 Random rendering of a heart-like model

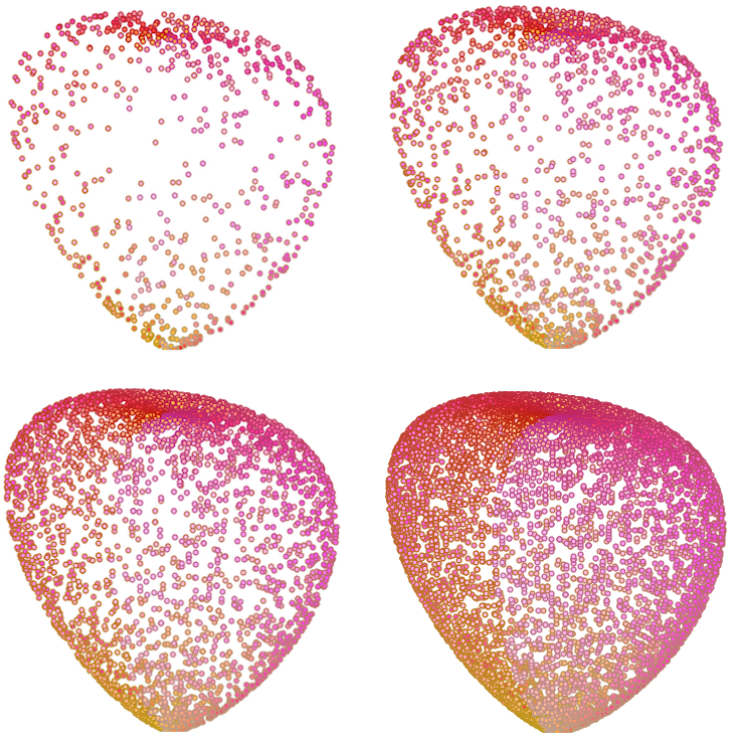


Fig.7 - Heart shape random scatter plot sequence

VIEW ANIMATION (requires internet connection)

Python 3 code to generate fig. 7

```
########################################
# Heart 3D scatter random generation
# (matplotlib module)
########################################

from mpl_toolkits import mplot3d  # imports 3D matplotlib modulus
```

```
import numpy as np
import matplotlib.pyplot as plt
import random as rd

r = 6.0
n = 150

theta = np.linspace(-np.pi, np.pi, n)
phi = np.linspace(0.0, np.pi, n)
theta, phi = np.meshgrid(theta, phi)

x = r * np.sin(theta)*np.sin(phi)
y = r * np.cos(theta)*np.sin(phi)
z = r*np.cos(phi)

X = x + 0.0008*y*y
Y = .7*y
Z = 2*z*np.e**(-.3*(np.pi-phi))

ax = plt.axes(projection='3d')

ax.set_xlim3d([-4.5, 4.5])
ax.set_ylim3d([-5.0, 5.0])
ax.set_zlim3d([-6.0, 3.0])
ax.set_axis_off()

while r > 0:
 t = np.random.choice(X.shape[0], 1, replace=False)
 u = np.random.choice(Y.shape[1], 1, replace=False)

 ax.scatter(X[t,u], Y[t,u], Z[t,u], s = 12,
 c = [1,.7*np.abs(np.cos(2*np.pi*(1+u/n))),
  np.abs(np.sin(np.pi*(1+t/n)))], marker="o",
  edgecolor=[.7,np.abs(t/255),np.abs(u/(300))] ,zorder=2)
 plt.pause(0.001)

plt.show()
```

8.2.4 Cellular automaton rendering of a heart-like model

A model attempting to describe, at least qualitatively, biological cell replication generating (or regenerating) an organ like a heart, requires space contiguity between any replicant and replicated cell. Therefore generically random replication cannot be suitable. Rather we need recur to a cellular automata based scheme. So we will examine, now, a python 3 code implementing an algorithm which generates a heart-like $3D$ structure:

- starting from an initial condition (cell co-ordinates) chosen by chance;
- replicating into a contiguous cell the position of which (co-ordinates) is shifted randomly aside ($x_i \to x_i$ or $x_i \to \pm 1, i = 1, 2, 3$).

The figs 8 amd 9 show two examples obtained starting from a cell located respctivly in two different initial positions chosen by chance. The *information* (*law*) driving the generation process being the same set of parametric equations, in both examples, manifestly leads cells to tend to the same structure (*attractor*).

Fig.8 - Heart shape random cellular automaton plot (first example)

VIEW ANIMATION (requires internet connection)

Fig.9 - Heart shape random cellular automaton plot (second example)

VIEW ANIMATION (requires internet connection)

Python 3 code to generate figs 8 and 9

```
########################################
# Heart 3D scatter cellular automaton
# random generation
# (matplotlib module)
########################################

from mpl_toolkits import mplot3d  # imports 3D matplotlib modulus

import numpy as np
import matplotlib.pyplot as plt
import random as rd

r = 6.0
n = 150

theta = np.linspace(0.0, 2*np.pi, n)
phi = np.linspace(0.0, np.pi, n)
theta, phi = np.meshgrid(theta, phi)

x = r * np.sin(theta)*np.sin(phi)
y = r * np.cos(theta)*np.sin(phi)
z = r*np.cos(phi)

X = x + 0.0008*y*y
Y = .7*y
Z = 2*z*np.e**(-.3*(np.pi-phi))

ax = plt.axes(projection='3d', aspect=.85)
ax.plot_wireframe(X, Y, Z, edgecolor='pink',zorder=1,alpha=0.08) # background trnaslucent image

ax.set_aspect='equal'
ax.set_xlim3d([-4.5, 4.5])
ax.set_ylim3d([-5.0, 5.0])
ax.set_zlim3d([-6.5, 2.5])
ax.set_axis_off()

M = n
Step = 1
C = np.arange(n*n)

t = np.random.choice(X.shape[0], 1, replace=False)
u = np.random.choice(Y.shape[1], 1, replace=False)

while r > 0:
 if t <= 0:
  t = t + 2*Step
 elif t >= M:
  t = t - 2*Step

 if u <= 0:
  u = u + 2*Step
 elif u >= M:
  u = u - 2*Step

 t = t + Step*(-1)**rd.randrange(0,2)
 u = u + Step*(-1)**rd.randrange(0,2)

 if t <= 0:
```

```
  t = t + 2*Step
 elif t >= M:
  t = t - 2*Step

 if u <= 0:
  u = u + 2*Step
 elif u >= M:
  u = u - 2*Step

 ax.scatter(X[t,u], Y[t,u], Z[t,u], s = 12,
 c = [1,.7*np.abs(np.cos(2*np.pi*(1+u/n))),
  np.abs(np.sin(np.pi*(1+t/n)))], marker="o",
  edgecolor=[.7,np.abs(t/255),np.abs(u/(300))] ,zorder=2)

 plt.pause(0.001)

plt.show()
```

8.3 POV-Ray 3.7 rendering of a heart-like external structure

Rendering quality is significantly increased, also modeling an heart-like shape, while employing a ray tracing software as *POV-Ray 3.7*.

We now examine the results when the heart model

- either is built as a whole;
- or is generated by means of a random process;
- or as a cellular automaton.

We will examine all those approaches.

8.3.1 The heart as a whole

POV-Ray 3.7 code to generate figs 10

```
//======================================
// Heart 3D as a whole
// (POV-Ray 3.7 rendering)
//======================================

 #include "colors.inc"

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera {
    location <-20, 20, -250>
    look_at  < 0, 0,  0>
  }

   light_source {
 < 20, 40, -120>
 rgb <1.000000, 1.000000, 1.000000> * 4.0
```

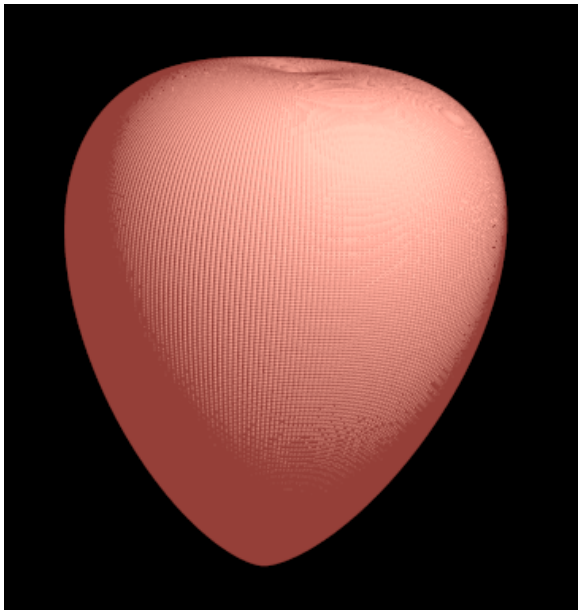


Fig.10 - Heart as a whole (POV-Ray 3.7 rendering)

```
}

#declare R = 60;

#declare n = 300;
#declare st = 1;
#declare S = 1;  // scale factor

 #declare Theta = -110;
 #declare Phi = 0;
 #declare Psi = -2;

union{ #for (p, 0, n, st)
  #declare th = p*2*pi/n;

   #for (q, 0, n, st)
      #declare ph = q*pi/n;

     #declare XX = S*R*cos(th)*sin(ph);
     #declare YY = S*R*sin(th)*sin(ph);
     #declare ZZ = R*cos(ph);

     #declare X = XX + 0.0008*YY*YY;
     #declare Y = 0.7*YY;
     #declare Z = 2*ZZ*exp(-.3*(pi-ph));

   sphere {
     < X, Y, Z >, 1   //  adding 3d axis
        texture {
        pigment { color < 1.0, 0.5, 0.4 > }
        }
        finish { ambient rgb <0.3,0.1,0.1>
```

```
         diffuse .3
        reflection 0.0
        specular 0.0 }   // plot sphere
    }

#end  // end for q
#end  // end for p

 translate < 0, 30, 20 >
 rotate < Theta, Phi, Psi > }
```

8.3.2 Sequential rendering of a heart-like model

We show in the following pictures the sequential sections of a progressive longitudinal and latitundial generation of the heart shape. We note that now a very refined resolution (very small cell dimensions has been chosen).

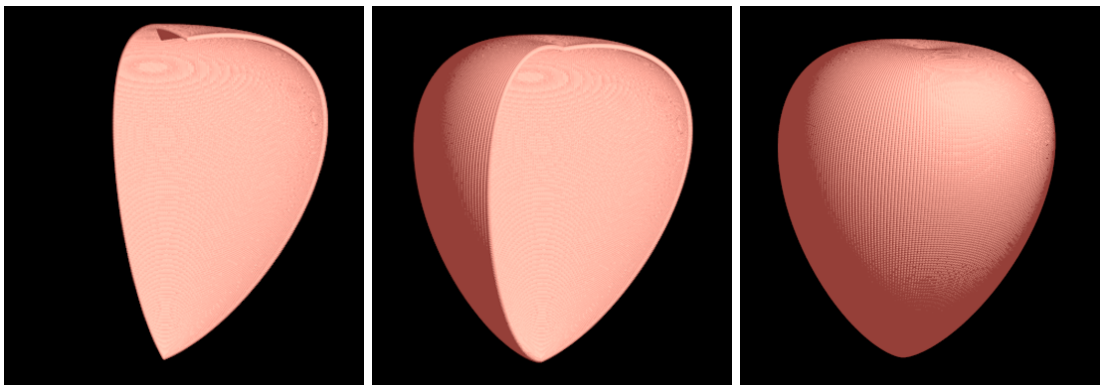


Fig.11 - Heart sequential vertical sections (POV-Ray 3.7 rendering)

VIEW ANIMATION (requires internet connection)

POV-Ray 3.7 code to generate figs 11 and 12

```
//============================================================
// Heart 3D sequential vertical/horizontal sections
// (POV-Ray 3.7 rendering)
//============================================================

 #include "colors.inc"

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

 camera {
    location <-20, 20, -250>
    look_at  < 0, 0,  0>}

   light_source {
```

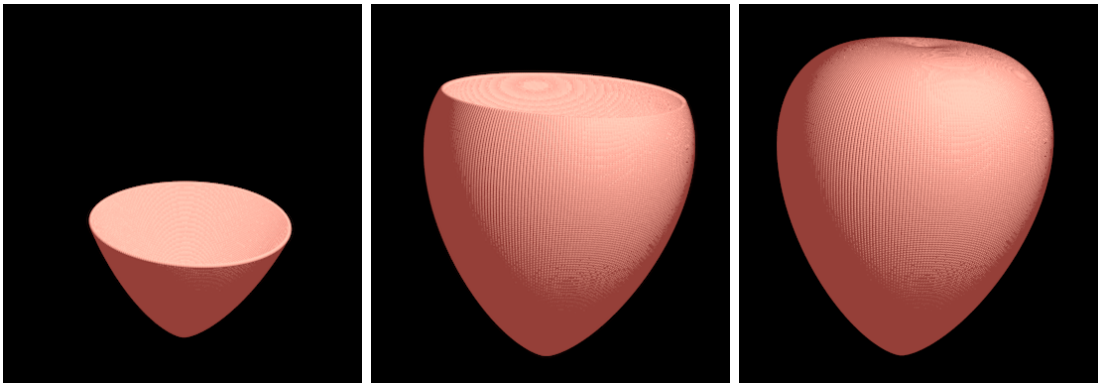


Fig.12 - Heart sequential horizontal sections (POV-Ray 3.7 rendering)

VIEW ANIMATION (requires internet connection)

```
< 20, 40, -120>
rgb <1.000000, 1.000000, 1.000000> * 4.0
}

#declare R = 60;

#declare n = 300;
#declare st = 1;
#declare S = 1;  // scale factor

 #declare Theta = -110;
 #declare Phi = 0;
 #declare Psi = -2;

// replace n by n*clock for vertical sections animation
 union{ #for (p, 0, n, st)  // vertical section steps 2*n/3-n,n/3-n, 0-n
   #declare th = -p*2*pi/n;

// replace n by n*clock for horizontal sections animation
    #for (q, 0, n, st)  // horizontal section steps 2*n/3-n,n/3-n, 0-n
       #declare ph = pi - q*pi/n;

      #declare XX = S*R*cos(th)*sin(ph);
      #declare YY = S*R*sin(th)*sin(ph);
      #declare ZZ = R*cos(ph);

      #declare X = XX + 0.0008*YY*YY;
      #declare Y = 0.7*YY;
      #declare Z = 2*ZZ*exp(-.3*(pi-ph));

    sphere {
      < X, Y, Z >, 1   //  adding 3d axis
         texture {
         pigment { color < 1.0, 0.5, 0.4 > }
         }
         finish { ambient rgb <0.3,0.1,0.1>
          diffuse .3
         reflection 0.0
```

```
        specular 0.0 }   // plot sphere
    }

 #end  // end for q
 #end  // end for p

  translate < 0, 30, 20 >
  rotate < Theta, Phi, Psi > }
```

8.3.3 Random rendering of a heart-like model

Random rendering steps of our rough algorithmic heart model are illustrated in the following fig. 13

POV-Ray 3.7 code to generate fig. 13

```
//=========================================================
// Heart 3D random generation steps
// (POV-Ray 3.7 rendering)
//=========================================================

#include "shapes.inc"
#include "glass.inc"
#include "colors.inc"
#include "metals.inc"

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

camera {
    location <-20, 20, -250>
    look_at  < 0, 0,  0>
  }

   light_source {
 < 20, 40, -120>
 rgb <1.000000, 1.000000, 1.000000> * 4.0
 }

 #declare R = 60;
 #declare n = 300;
 #declare st = 1;
 #declare S = 1;  // scale factor

  #declare Theta = -110;
  #declare Phi = 0;
  #declare Psi = -2;

 #declare X = array[n+1][n+1];
 #declare Y = array[n+1][n+1];
 #declare Z = array[n+1][n+1];


#for (p, 0, n,st)
  #declare th = p*2*pi/n;

    #for (q, 0, n,st)
       #declare ph = q*pi/n;
```

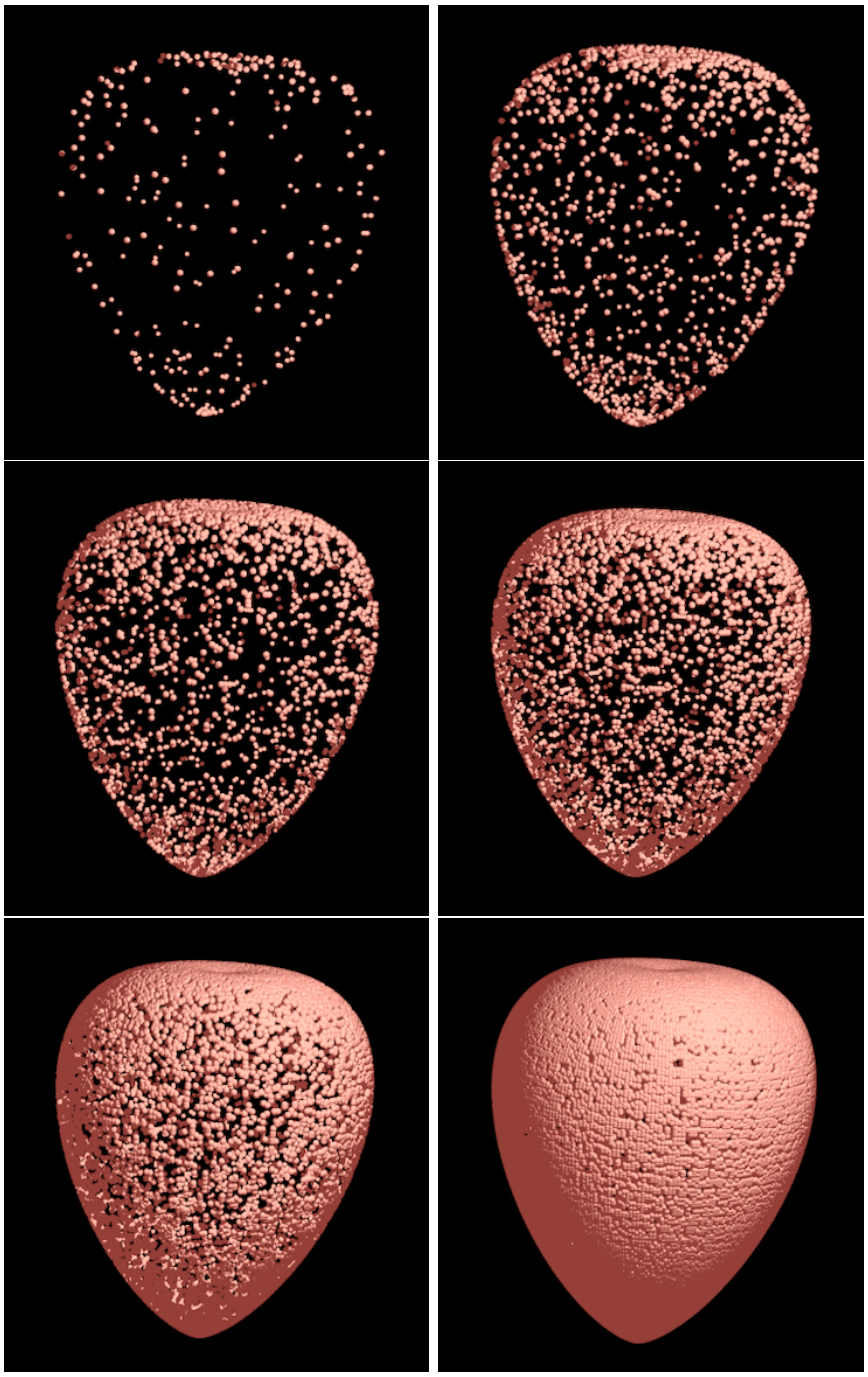


Fig.13 - Heart random generation steps (POV-Ray 3.7 rendering)

VIEW ANIMATION (requires internet connection)

```
       #declare XX = S*R*cos(th)*sin(ph);
      #declare YY = S*R*sin(th)*sin(ph);
      #declare ZZ = R*cos(ph);

      #declare X[p][q] = XX + 0.0008*YY*YY;
      #declare Y[p][q] = 0.7*YY;
      #declare Z[p][q] = 2*ZZ*exp(-.3*(pi-ph));

  #end  // end for q
  #end  // end for p

 #declare Rnd_1 = seed (1153);
 #declare Rnd_2 = seed (553) ;

 // replace n by n*clock for animation
  union{ #for(j,0,200*n)   // steps n, 5*n, 10*n, 20*n, 50*n, 200*n
  #declare p = int(n*rand(Rnd_1));
  #declare q = int(n*rand(Rnd_2));
    sphere {
       < X[p][q], Y[p][q], Z[p][q] >,1
    texture {
     pigment { color < 1.0, 0.5, 0.4 > }
     }
     finish { ambient rgb <0.3,0.1,0.1>
     diffuse .3
     reflection 0.0
     specular 0.0 }   // plot sphere  }

  translate < 0, 30, 20 >
  rotate < Theta, Phi, Psi > }
    #end}  // end for j
```

8.3.4 Cellular automaton rendering of a heart-like model

POV-Ray 3.7 code to generate fig. 13

```
//====================================
// Cellular automata 3D Heart shape generation
// by POV-Ray 3.7
//====================================

#include "colors.inc"

  global_settings {assumed_gamma 1.0}
  background { color rgb <00,0.0,0.0> }

camera {
    location <-20, 20, -250>
    look_at  < 0, 0,  0>
  }

   light_source {
 < 20, 40, -120>
 rgb <1.000000, 1.000000, 1.000000> * 4.0
 }

 #declare R = 60;
 #declare n = 300;
```

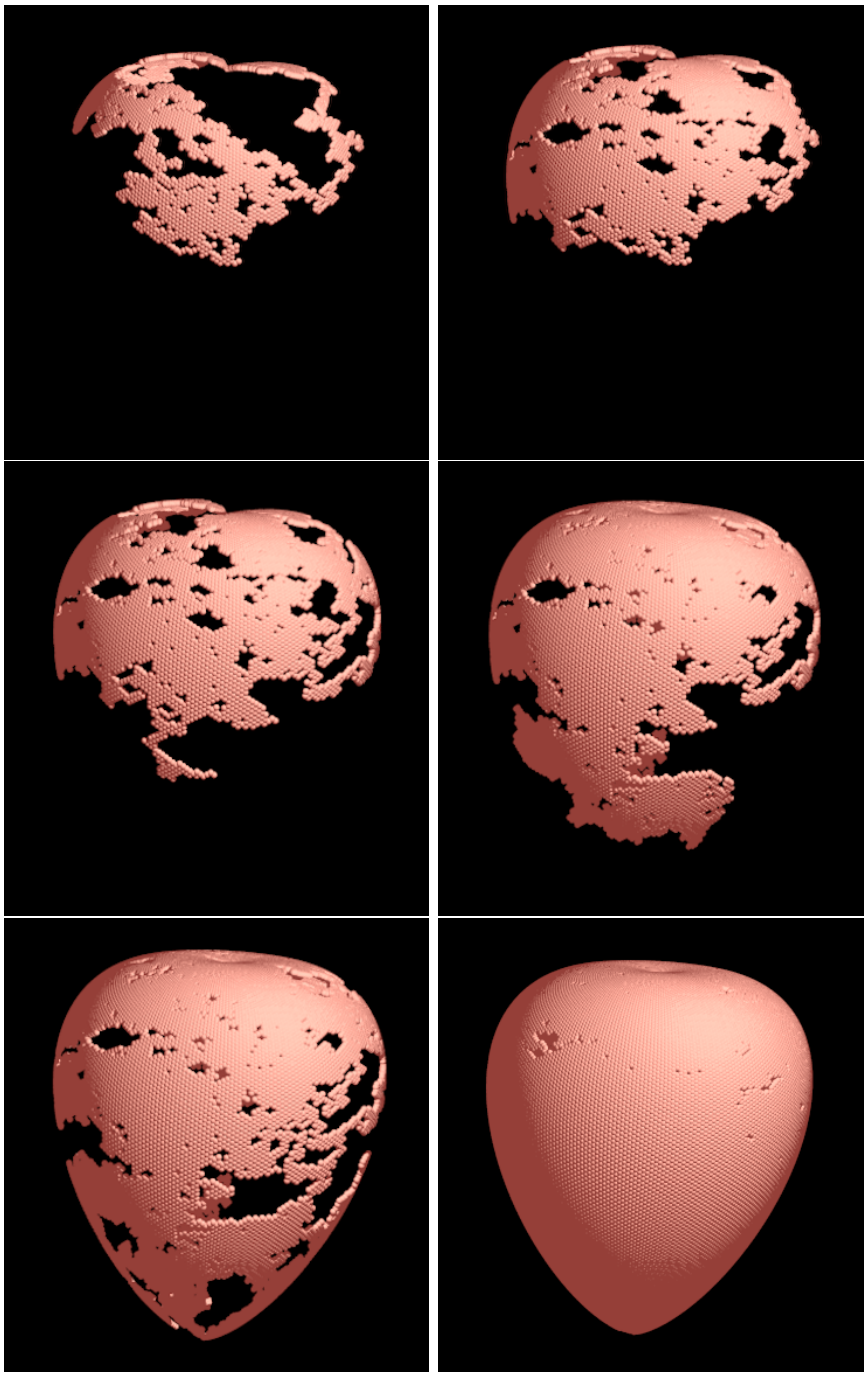


Fig.14 - Heart generation steps by a cellular automaton (POV-Ray 3.7 rendering)

VIEW ANIMATION (requires internet connection)

```
 #declare st = 1;
 #declare S = 1;  // scale factor

  #declare Theta = -110;
  #declare Phi = 0;
  #declare Psi = -2;

 #declare X = array[n+1][n+1];
 #declare Y = array[n+1][n+1];
 #declare Z = array[n+1][n+1];


#for (p, 0, n,st)
  #declare th = p*2*pi/n;

    #for (q, 0, n,st)
       #declare ph = q*pi/n;
       #declare XX = S*R*cos(th)*sin(ph);
      #declare YY = S*R*sin(th)*sin(ph);
      #declare ZZ = R*cos(ph);

      #declare X[p][q] = XX + 0.0008*YY*YY;
      #declare Y[p][q] = 0.7*YY;
      #declare Z[p][q] = 2*ZZ*exp(-.3*(pi-ph));

  #end  // end for q
  #end  // end for p

 #declare Rnd_1 = seed (1153);
 #declare Rnd_2 = seed (553);
 #declare pp = 0;
 #declare qq = 0;

  union{ #for(j,0,5*n*n)   // steps 50*n, 100*n, 110*n, 200*n, 370*n, 5*n*n
  #declare p = pp + st*pow(-1,int(n*rand(Rnd_1)));
   #declare q = qq + st*pow(-1,int(n*rand(Rnd_2)));

   #if ( abs(p) < n)
   #if( abs(q) < n)
   #declare pp = p;
  #declare qq = q;

    sphere {
       < X[abs(p)][abs(q)], Y[abs(p)][abs(q)], Z[abs(p)][abs(q)] >,1
    texture {
     pigment { color < 1.0, 0.5, 0.4 > }
     }
     finish { ambient rgb <0.3,0.1,0.1>
     diffuse .3
     reflection 0.0
     specular 0.0 }   // plot sphere  }

  translate < 0, 30, 20 >
  rotate < Theta, Phi, Psi > }

   #end // end if
   #end // end if
  #end}  // end for j
```

Chapter 9

A more realistic model of heart structure

Sequential and random rendering processes

9.1 Introduction

9.2 Heart model based on POV-Ray 3.7 *smooth_triangle* object

Bob Hughes has written, in year 2000, a 3*D POV-Ray* heart model code based on a list of data from a real human heart representation obtained by decomposition of the external heart surface into small *smooth triangles*.[1]

We point out that the author does not start from a mathematical law (*compressed string*) to calculate the data identifying the *smooth triangles*.[2]

So he needs to provide a full data list of the co-ordinates and local normal components of each individual *smooth triangle*. The data file (*heart.inc*) involves more than 1600 *smooth triangles*, the related data being listed sequentially one after the other as if the information were at all *irreducible*.

An intriguing question is if similar biological data are truly non-computable (*incompressible string* o *irreducible information*) or we may guess that in future some shorter rule (*law*) could be discovered.

[1] The zipped code files can be downloaded at http://objects.povworld.org/cgi-bin/search.cgi?X=heart.

[2] A *smooth triangle* in *POV-Ray* is obtained interpolating a curved surface starting from its local gradient vectors assigned in each vertex of the triangle.

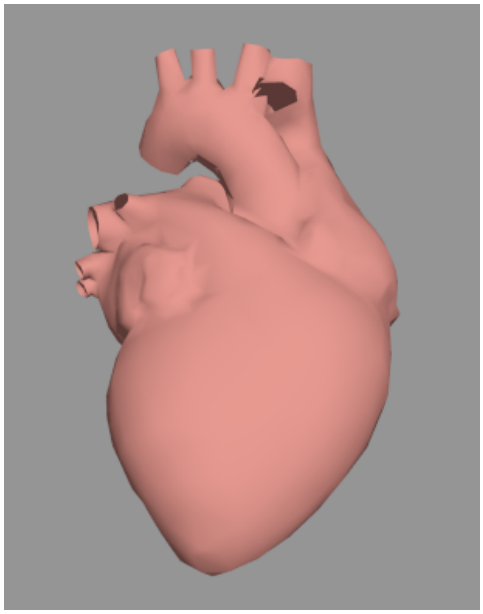


Fig.1 - Heart model based on a triangles data set by (Bob Hughes)

POV-Ray 3.7 code to generate fig. 1

```
//====================================
// 3D Heart shape generation as a whole
// by POV-Ray 3.7 (model by Bob Hughes)
//====================================

/*
POV-Ray version 1.0 include of a human heart.
by Bob Hughes
--
 omniVERSE: beyond the universe
  http://members.aol.com/inversez/POVring.htm
 mailto:inversez@aol.com?PoV

*/

camera {
   location <0,  0,  -10.0>
   direction <0.0, 0.0,  1.0>
   up  <0.0,  1.0,  0.0>
   right <1.33333, 0.0, 0.0>
   look_at <0, -1, 0>
}

// Light source
object { light_source {  <5, 10 , -30> color rgb 1  }}

/* Texture declarations for object 'HEART' */
#declare HEART_C1 = texture {
```

```
   finish{ ambient 0.1  diffuse 0.7}
//    phong 1.0
//    phong_size 70.0
    pigment{ color red 1.000 green 0.400 blue 0.3500}
}

object {  /* All Objects */
    #include "heart.inc"
#version 3.1
    rotate <-90, 180, 0>

rotate y*clock*360
    translate <0, -3, 0>
    /*
        Scene extents
        X - Min:  -2.4900  Max:   3.3300
        Y - Min:  -0.3300  Max:   5.7600
        Z - Min:  -2.5100  Max:   5.7900
    */
}

background{rgb .3}
```

The POV-Ray 3.7 *smooth_triangle* object operates on three couples of $3D$ vectors.

$$\begin{aligned}\text{smooth\_triangle } \{ \quad & < x_1 y_1 z_1 >< u_1 v_1 w_1 > \\ & < x_2 y_2 z_2 >< u_2 v_2 w_2 > \\ & < x_3 y_3 z_3 >< u_3 v_3 w_3 > \ \}\end{aligned}$$

– The former vectors of each couple $< x_i, y_i, z_i >, (i = 1, 2, 3)$ involve the *co-ordinates of the vertices* of a triangle.

– The latter vectors $< u_i, v_i, w_i >$ are the components of the local *gradient vector* in the related vertex, normalized to unity, which is orthogonal to the surface of the curved triangle itself.

```
//================================================
// Fragment example of the heart.inc data file
//================================================

#version 1.0
/* Object 'HEART' */
composite {
composite {
composite {
composite {
object {
union {
smooth_triangle { <-1.4100 0.9200 -0.7600> <-0.7857 -0.4896 -0.3782>
<-1.7200 1.7500 -0.8800> <-0.8391 -0.1947 -0.5079> <-1.7300 1.2500 -0.3800>  <-0.8751 -0.3883 -0.2886> }

smooth_triangle { <-1.3700 1.4600 -1.2200> <-0.8018 -0.2494 -0.5431>
 <-1.7200 1.7500 -0.8800> <-0.8391 -0.1947 -0.5079> <-1.4100 0.9200  -0.7600> <-0.7857 -0.4896 -0.3782> }
```

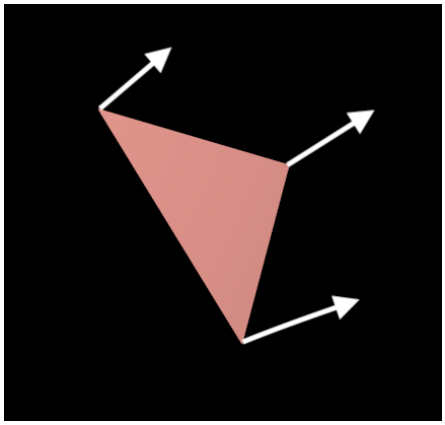


Fig.2 - Graphic representation of a smooth triangle

```
smooth_triangle { <-1.4100 0.9200 -0.7600> <-0.7857 -0.4896 -0.3782>
 <-1.1900 1.0100 -1.2200> <-0.7430 -0.3941 -0.5410> <-1.3700 1.4600  -1.2200> <-0.8018 -0.2494 -0.5431> }
.....................................................................

smooth_triangle {
<2.1600 4.0400 4.3000> <0.7212 0.2054 0.6616> <2.0500 3.4500 4.5200>
<0.9089 -0.1626 0.3840> <1.8200 3.4200 4.8000> <0.7830 0.1992 0.5892> }}
texture { HEART_C1 }
bounded_by { box { <1.45000 3.42000 4.27000> <2.16000 4.19000 5.10000> }}}
bounded_by { box { <1.41000 2.97000 4.27000> <2.16000 4.19000 5.69000> }}}
bounded_by { box { <0.57000 2.42000 4.27000> <2.16000 4.19000 5.69000> }}}
bounded_by { box { <0.26000 0.96000 1.54000> <3.30000 5.29000 5.69000> }}}
bounded_by { box { <-2.49000 -0.33000 -2.51000> <3.33000 5.76000 5.79000> }}}
```

The code offers a nice representation of a heart external shape as a whole, but here we are interested to control each *individual triangle* and possibly each *individual cell* (modeled by a *small sphere*) in order to build either an ordered *sequential* or *random* or *cellular automata* process, like those examined in the previous chapters.

9.2.1 Generating Hughes heart model by an ordered sequence of *smooth triangles*

As a first step we have modified the original *POV-Ray* code and *heart.inc* data file in order to obtain partial steps and related images showing how each *smooth_triangle* may be added according to an ordered sequence generating the heart structure.

POV-Ray 3.7 code to generate fig. 3

```
//================================================
// 3D Heart shape generation by an ordered sequence
// of smooth triangles (POV-Ray 3.7 code)
//================================================

#include "heart.inc"
#version 3.7

camera {
   location <0,  0,  -10.0>
   direction <0.0, 0.0,  1.0>
   up  <0.0,  1.0,  0.0>
   right <1.33333, 0.0, 0.0>
   look_at <0, -1, 0>
}

// Light source
object { light_source {  <5, 10 , -30> color rgb 1  }}

    /*
        Scene extents
        X - Min:  -2.4900  Max:   3.3300
        Y - Min:  -0.3300  Max:   5.7600
        Z - Min:  -2.5100  Max:   5.7900
    */

#declare n = 1639;

//  replace n by  n*clock for animation
#for (i, 1, n, 1)   // steps n/24  n/18  n/9  n/2    n
  object{
   T[i]
texture {
   finish{ ambient 0.1  diffuse 0.7 }
   pigment{ color red 1.000 green 0.400 blue 0.3500 filter 0 }
}
  rotate <-90, 180, 0>
// rotate y*clock*360
 translate <0, -3, 0>
  }

 #end

//=======================================================
// Fragment example of the modified heart.inc data file
//=======================================================

#version 1.0
#declare T = array[1800];

/* Object 'HEART' */
#declare T[1] = smooth_triangle {
<-1.4100 0.9200 -0.7600> <-0.7857 -0.4896 -0.3782> <-1.7200 1.7500 -0.8800>
<-0.8391 -0.1947 -0.5079> <-1.7300 1.2500 -0.3800> <-0.8751 -0.3883 -0.2886> };

#declare T[2] = smooth_triangle {
<-1.3700 1.4600 -1.2200> <-0.8018 -0.2494 -0.5431> <-1.7200 1.7500 -0.8800>
<-0.8391 -0.1947 -0.5079> <-1.4100 0.9200 -0.7600> <-0.7857 -0.4896 -0.3782> };
```

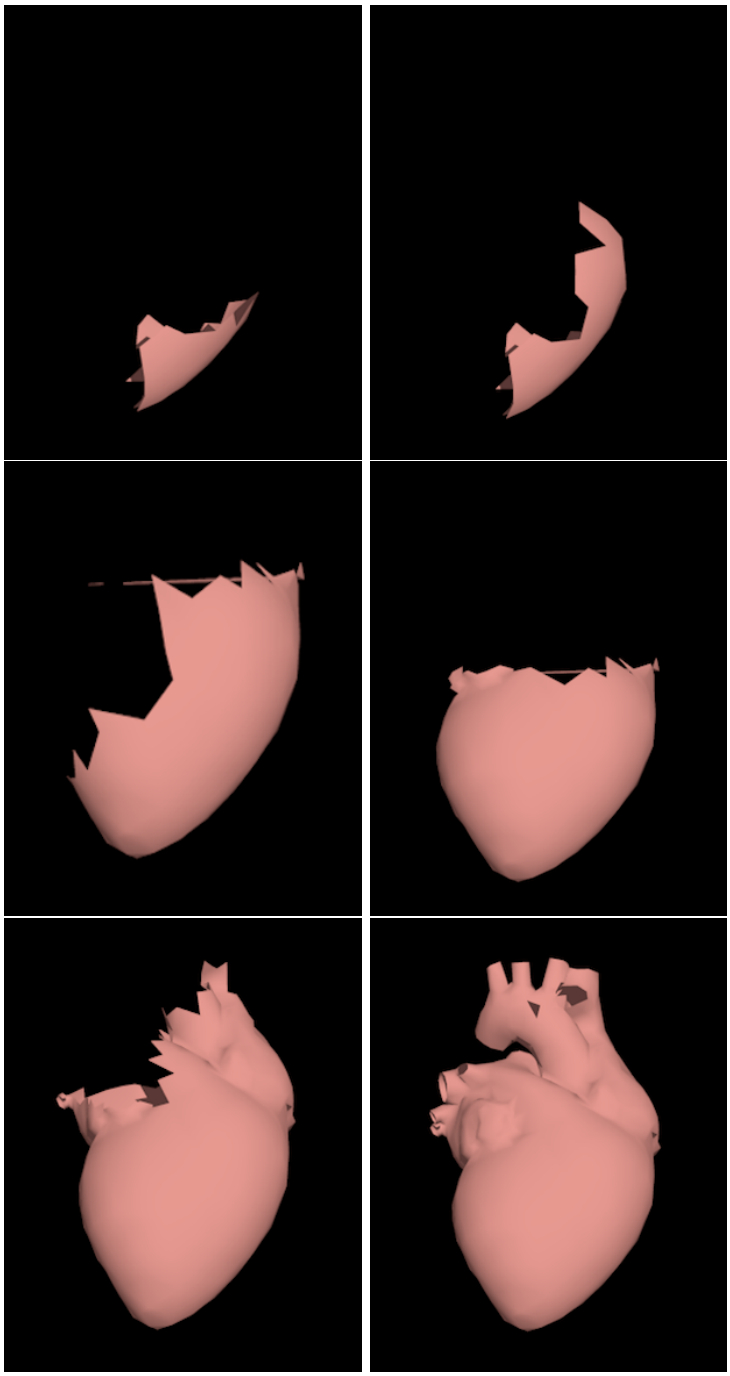


Fig.3 - Heart generation steps by an ordered sequence of smooth triangles (POV-Ray 3.7 rendering)

VIEW ANIMATION (requires internet connection)

```
#declare T[3] = smooth_triangle {
<-1.4100 0.9200 -0.7600> <-0.7857 -0.4896 -0.3782> <-1.1900 1.0100 -1.2200>
 <-0.7430 -0.3941 -0.5410> <-1.3700 1.4600 -1.2200> <-0.8018 -0.2494 -0.5431> };

.............................................................

#declare T[1638] = smooth_triangle {
<1.9700 4.1600 4.3600> <0.2145 0.6226 0.7526> <2.1600 4.0400 4.3000>
<0.7212 0.2054 0.6616> <1.7700 3.8300 4.6900> <0.2513 0.5910 0.7665>};

#declare T[1639] = smooth_triangle {
<2.1600 4.0400 4.3000> <0.7212 0.2054 0.6616> <2.0500 3.4500 4.5200>
<0.9089 -0.1626 0.3840> <1.8200 3.4200 4.8000> <0.7830 0.1992 0.5892>};
```

Since each *smooth triangle* is not very small, compared with a *biological cell,* the sequence results very rough and the images of each sequence step resembles a broken egg shell rather than a biological structure.

9.2.2 Generating Hughes heart model by random *smooth triangles*

The roughness of the images is further emphasized if we follow a *random* generation process by means of smooth triangles.

POV-Ray 3.7 code to generate fig. 4

```
//================================================
// 3D Heart shape generation by a random sequence
// smooth triangles (POV-Ray 3.7 code)
//================================================

camera {
 location <0,  0,  -10.0>
  direction <0.0, 0.0,  1.0>
    up  <0.0,  1.0,  0.0>
    right <1.33333, 0.0, 0.0>
    look_at <0, -1, 0>
  }

  light_source {
< 100, 120, -40>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#include "heart.inc"
 #version 3.1

    /*
        Scene extents
        X - Min:  -2.4900  Max:   3.3300
        Y - Min:  -0.3300  Max:   5.7600
        Z - Min:  -2.5100  Max:   5.7900
    */
// }

#declare n = 1638;
#declare i = 1;
```

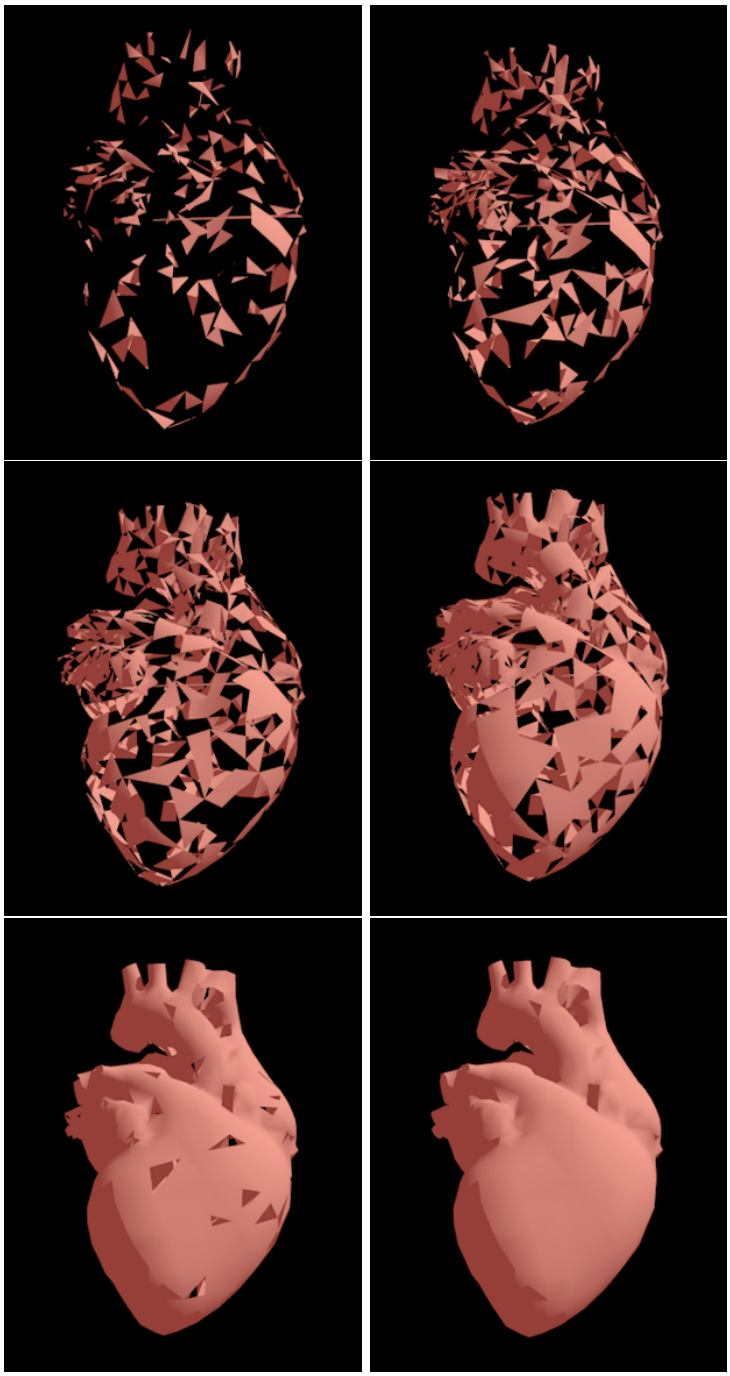


Fig.4 - Heart generation steps by a random sequence of smooth triangles (POV-Ray 3.7 rendering)

VIEW ANIMATION (requires internet connection)

```
#declare Rnd_1 = seed(128);

//  replace n by  n*clock for animation
union{ #for(i,0, n)  // steps  n/8, n/4, n/2, n, 3*n, 12*n
#declare ind = 1+int(n*rand(Rnd_1));
  object{
   T[ind]
  texture { pigment {color < 1.0, 0.5, 0.4 > }
   finish { ambient rgb <0.3,0.1,0.1>
     diffuse .3
     reflection 0.0
     specular 0.0 }
     }

  rotate <-90, 180, 0>
rotate 0*clock*360
 translate <0, -3, 0>
  }
#end}

// The heart.inc file is the same as for the ordered sequence code
```

9.2.3 Generating Hughes heart model by cellular automata in *smooth triangles*

A *random* sequence of *smooth triangles* may be treated even by *cellular automata,* simply imposing that each of them is generated randomly by a *contiguous smooth triangle.* The result is not very dissimilar to that obtained by an ordered sequence.

POV-Ray 3.7 code to generate fig. 5

```
//================================================
// 3D Heart shape generation by  cellular automata
// in smooth triangles (POV-Ray 3.7 code)
//================================================

camera {
 location <0,  0,  -10.0>
  direction <0.0, 0.0,  1.0>
    up  <0.0,  1.0,  0.0>
    right <1.33333, 0.0, 0.0>
    look_at <0, -1, 0>
  }

  light_source {
< 100, 120, -40>
rgb <1.000000, 1.000000, 1.000000> * 2.0
 }

#include "heart.inc"
 #version 3.1

    /*
        Scene extents
        X - Min:  -2.4900  Max:   3.3300
        Y - Min:  -0.3300  Max:   5.7600
        Z - Min:  -2.5100  Max:   5.7900
    */
```

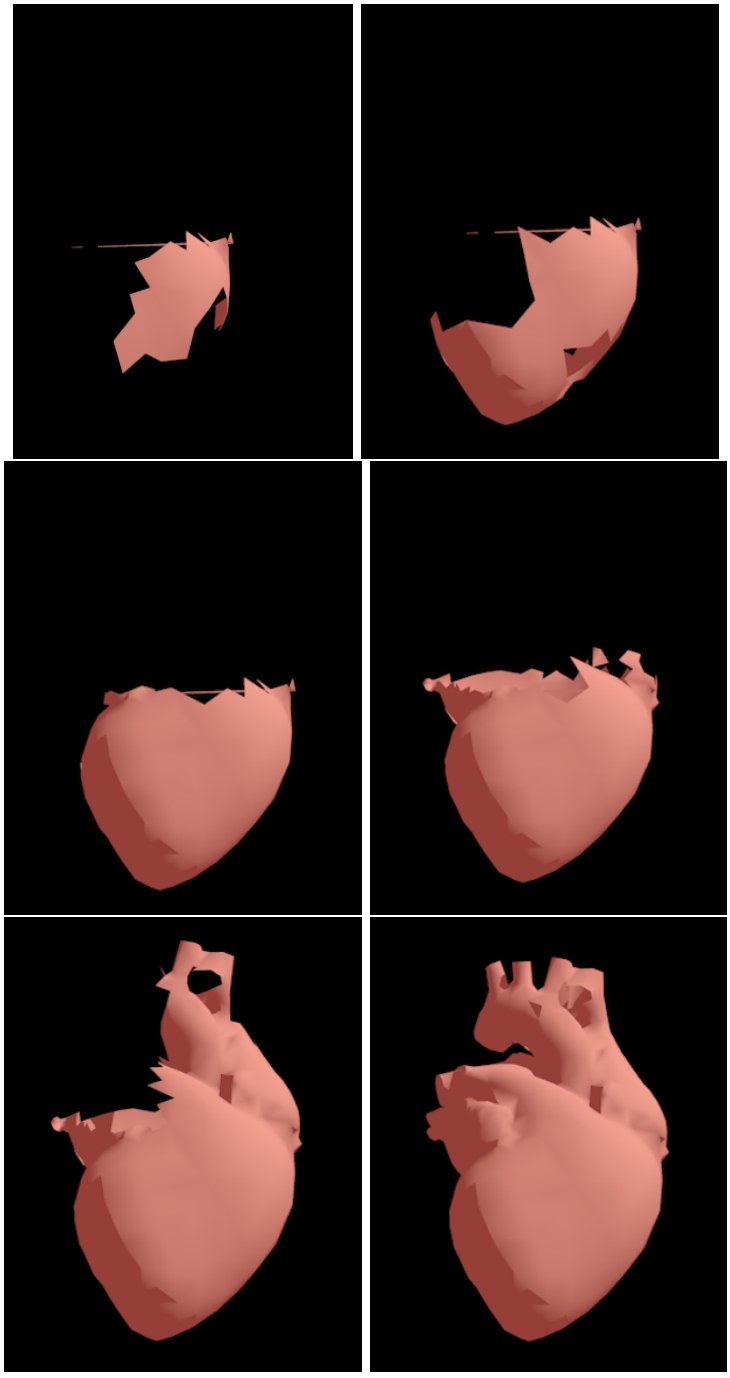


Fig.5 - Heart generation steps by smooth triangles cellular automata
(POV-Ray 3.7 rendering)

VIEW ANIMATION (requires internet connection)

```
// }

#declare n = 1638;
#declare i = 1;
#declare Rnd_1 = seed(128);
 #declare pp = 100;

//  replace n*n by  n*n*clock for animation
union{ #for(i,0, n*n)  // n, 8*n, 64*n, n*n/4, n*n/2, n*n
#declare p = pp + pow(-1,int(n*rand(Rnd_1)));

 #if (p > 0)
 #if ( p < n)
 #declare pp = p;
  object{
   T[p]
  texture { pigment {color < 1.0, 0.5, 0.4 > }
   finish { ambient rgb <0.3,0.1,0.1>
     diffuse .3
     reflection 0.0
     specular 0.0 }
     }

  rotate <-90, 180, 0>
rotate 0*clock*360
 translate <0, -3, 0>
  }
#end // end if
#end // end if
#end} // end i
```

9.3 Simplified heart model based on POV-Ray 3.7 single cells in *ordinary triangle* objects

In order to refine the modified heart model we have proposed in the previous sections so that *each cell* is represented by a *single point* belonging to some triangle we simplify the code by replacing *smooth_triangle* objects by ordinary *triangle* ones.

The heart external surface results no longer so perfectly smooth but we gain the advantage of more realistic *single cell* multiplication representation. The resulting effect appears in each image representing a step of the generation process and even more when the images are collected into a movie (see fig. 7).

POV-Ray 3.7 code to generate fig. 6

```
//================================================
// 3D Heart shape generation by single cells
// in ordinary triangles (POV-Ray 3.7 code)
//================================================

#include "heartPoints"
#version 3.7

  global_settings {assumed_gamma 1.0}
```

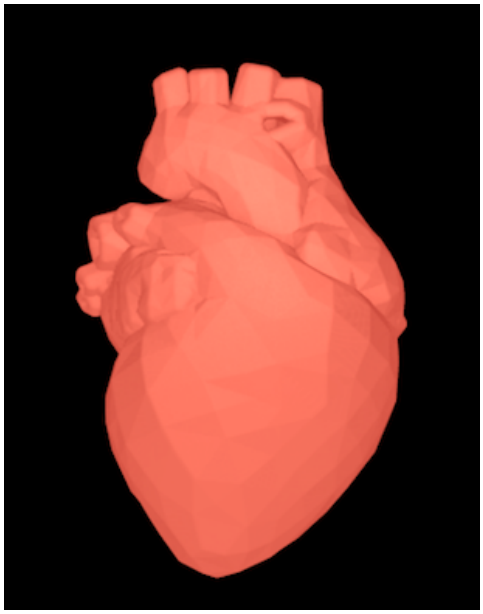


Fig.6 - Heart as a whole generated by single cells in ordinary triangles
(POV-Ray 3.7 rendering)

```
camera {
 location <0,  0,  -10.0>
  direction <0.0, 0.0,  1.0>
    up  <0.0,  1.0,  0.0>
    right <1.33333, 0.0, 0.0>
    look_at <0, -1, 0>
  }

 light_source {
< 7, 0, -40>
rgb <1.000000, 1.000000, 1.000000> * 1.0
fade_distance 3
fade_power 0.3
 }


 light_source{ <0,0,0>
              color rgb<1,1,1>
              area_light
              <5, 0, 0> <0, 0, 5>
              6,6 // numbers in directions
              adaptive 1  // 0,1,2,3...
              jitter // random softening
                       }//---- end of area_light

#declare st = 0.01;
#declare n = 1;

#declare N = 1647;
#for (i,0,N,1)
```

```
#for (p,0,n,st)
#for (q,0,p,st)
#declare X = a[i][3] - (a[i][3] - a[i][0])*p/n + (a[i][6] - a[i][0])*q/n;
#declare Y = a[i][4] - (a[i][4] - a[i][1])*p/n + (a[i][7] - a[i][1])*q/n;
#declare Z = a[i][5] - (a[i][5] - a[i][2])*p/n + (a[i][8] - a[i][2])*q/n;

 sphere { < X, Y, Z >, 0.1
   texture{
   pigment{color < 1.0, 0.3, 0.2 > }
   finish { ambient rgb <0.3,0.1,0.1>
     diffuse 0.3
     reflection 0.0
     specular 0.0}}
 rotate <-90, 180, 0>
rotate 0*clock*360
 translate <0, -3, 0>

 }

#end // i
#end  // p
#end  // q
```

9.3.1 Generating a heart model by an ordered sequence of cells in *ordinary triangles*

Ordinary triangles are characterized simply by the triplets of the co-ordinates of their vertices. So the "HeartPoints.inc" to be included by the *POV-Ray 3.7* code can be obtained by the original "heart.inc" file defining the *smooth triangles* data simply dropping the second vector of each couple, which is related to the local normal vector.

POV-Ray 3.7 code to generate fig. 7

```
//=====================================================
// 3D Heart shape generation by an ordered sequence
// of smooth triangles (POV-Ray 3.7 code)
//=====================================================

#include "heartPoints"

#version 3.7

  global_settings {assumed_gamma 1.0}

camera {
 location <0,  0,  -10.0>
  direction <0.0, 0.0,  1.0>
    up  <0.0,  1.0,  0.0>
    right <1.33333, 0.0, 0.0>
    look_at <0, -1, 0>
  }

 light_source {
< 7, 0, -40>
rgb <1.000000, 1.000000, 1.000000> * 1.0
fade_distance 3
fade_power 0.3
```

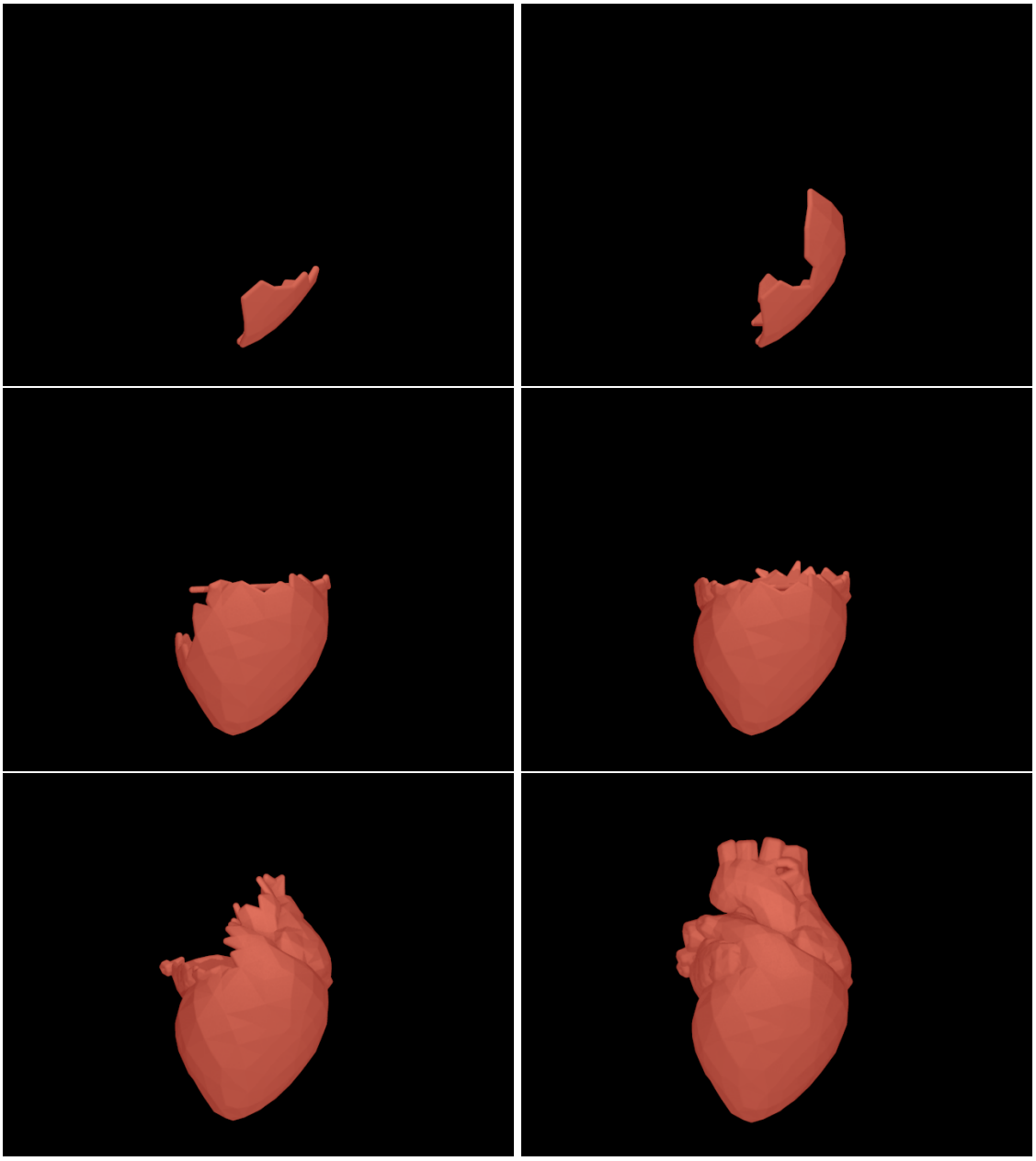


Fig.7 - Heart generation steps by an ordered sequence of cells in ordinary triangles (POV-Ray 3.7 rendering)

VIEW ANIMATION (requires internet connection)

```
 }

 light_source{ <0,0,0>
              color rgb<1,1,1>
              area_light
              <5, 0, 0> <0, 0, 5>
              6,6 // numbers in directions
              adaptive 1  // 0,1,2,3...
              jitter // random softening
                       }//---- end of area_light

#declare st = 0.01;
#declare n = 1;

#declare N = 1647;
#for (i,0,N,1)
#for (p,0,n,st)
#for (q,0,p,st)
#declare X = a[i][3] - (a[i][3] - a[i][0])*p/n + (a[i][6] - a[i][0])*q/n;
#declare Y = a[i][4] - (a[i][4] - a[i][1])*p/n + (a[i][7] - a[i][1])*q/n;
#declare Z = a[i][5] - (a[i][5] - a[i][2])*p/n + (a[i][8] - a[i][2])*q/n;

 sphere { < X, Y, Z >, 0.1
   texture{
   pigment{color < 1.0, 0.3, 0.2 > }
   finish { ambient rgb <0.3,0.1,0.1>
     diffuse 0.3
     reflection 0.0
     specular 0.0}}
 rotate <-90, 180, 0>
rotate 0*clock*360
 translate <0, -3, 0>}

#end // i
#end  // p
#end  // q

//=====================================================================
// Fragment example of the modified ‘‘heartPoints.inc’’ data file
//=====================================================================

#declare a = array[1648][9]{
{-1.4100,0.9200,-0.7600,     -1.7200,1.7500,-0.8800,     -1.7300,1.2500,-0.3800},
{-1.3700,1.4600,-1.2200,     -1.7200,1.7500,-0.8800,     -1.4100,0.9200,-0.7600},
{-1.4100,0.9200,-0.7600,     -1.1900,1.0100,-1.2200,     -1.3700,1.4600,-1.2200},



{1.8200,3.4200,4.8000,     1.7700,3.8300,4.6900,     2.1600,4.0400,4.3000},
{1.9700,4.1600,4.3600,     2.1600,4.0400,4.3000,     1.7700,3.8300,4.6900},
{2.1600,4.0400,4.3000,     2.0500,3.4500,4.5200,     1.8200,3.4200,4.8000}}
```

9.3.2 Generating a heart model by *random cells* in *ordinary triangles*

A *random* process generating the heart model *point by point* impressively shows how chance combined with information allows to build an ordered structure starting by initial conditions assigned randomly.

POV-Ray 3.7 code to generate fig. 8

```
//=================================================
// 3D Heart shape generation by random cells
// of smooth triangles (POV-Ray 3.7 code)
//=================================================

#include "heartPoints"

#version 3.7

  global_settings {assumed_gamma 1.0}

camera {
 location <0,  0,  -10.0>
  direction <0.0, 0.0,  1.0>
    up  <0.0,  1.0,  0.0>
    right <1.33333, 0.0, 0.0>
    look_at <0, -1, 0>
  }

 light_source {
< 7, 0, -40>
rgb <1.000000, 1.000000, 1.000000> * 1.0
fade_distance 3
fade_power 0.3
 }


 light_source{ <0,0,0>
              color rgb<1,1,1>
              area_light
              <5, 0, 0> <0, 0, 5>
              6,6 // numbers in directions
              adaptive 1  // 0,1,2,3...
              jitter // random softening
                       }//---- end of area_light

#declare st = 1;
#declare n = 100;

#declare N = 1647;
#declare Rnd_1 = seed (1153);
 #declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (876);

union{ #for(j,0,256*N,1)  // steps N/6  N/3  N  4*N  16*N  256*N
#declare i = int(N*rand(Rnd_1));
#declare p = int(n*rand(Rnd_2));
#declare q = int(p*rand(Rnd_3));

#declare X = a[i][3] - (a[i][3] - a[i][0])*p/n + (a[i][6] - a[i][0])*q/n;
#declare Y = a[i][4] - (a[i][4] - a[i][1])*p/n + (a[i][7] - a[i][1])*q/n;
#declare Z = a[i][5] - (a[i][5] - a[i][2])*p/n + (a[i][8] - a[i][2])*q/n;

 sphere { < X, Y, Z >, 0.05
   texture{
   pigment{color < 1.0, 0.3, 0.2 > }
   finish { ambient rgb <0.3,0.1,0.1>
     diffuse 0.3
```

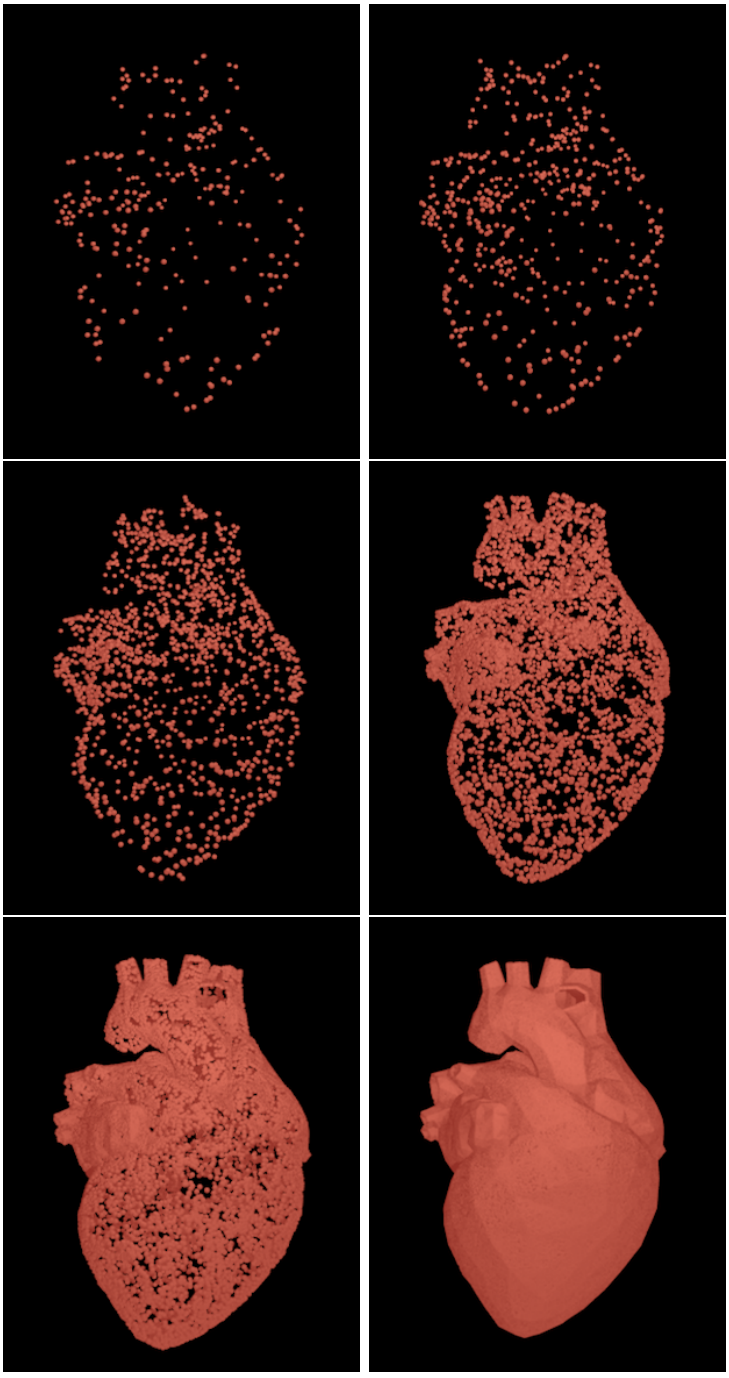


Fig.8 - Heart generation steps by random cells in ordinary triangles
(POV-Ray 3.7 rendering)

VIEW ANIMATION (requires internet connection)

```
     reflection 0.0
     specular 0.0}}
 rotate <-90, 180, 0>
# rotate 0*clock*360  // uncheck for animation
 translate <0, -3, 0>}
#end} // j
```

9.3.3 Generating a heart model by *cells* plotted by *cellular automata*

As a final step we implement a *random process* generating the heart model by *cellular automata*. Of course the *cell* dimension is not realistic respect to true biological conditions, in order to reduce heaviness and time duration of computer calculations. But the graphic result is enough to render the idea of what may happen in nature. Information seems just to drive chance towards order and organization.

POV-Ray 3.7 code to generate fig. 9

```
//====================================
// 3D Heart shape generation  by cellular automata
// in ordinary triangles POV-Ray 3.7
//====================================

#include "heartPoints"

#version 3.7

  global_settings {assumed_gamma 1.0}

camera {
 location <0,  0,  -10.0>
  direction <0.0, 0.0,  1.0>
    up  <0.0,  1.0,  0.0>
    right <1.33333, 0.0, 0.0>
    look_at <0, -1, 0>
  }

 light_source {
< 7, 0, -40>
rgb <1.000000, 1.000000, 1.000000> * 1.0
fade_distance 3
fade_power 0.3
 }


 light_source{ <0,0,0>
              color rgb<1,1,1>
              area_light
              <5, 0, 0> <0, 0, 5>
              6,6 // numbers in directions
              adaptive 1  // 0,1,2,3...
              jitter // random softening
                       }//---- end of area_light

#declare st = 1;
#declare n = 20;

#declare N = 1647;
```

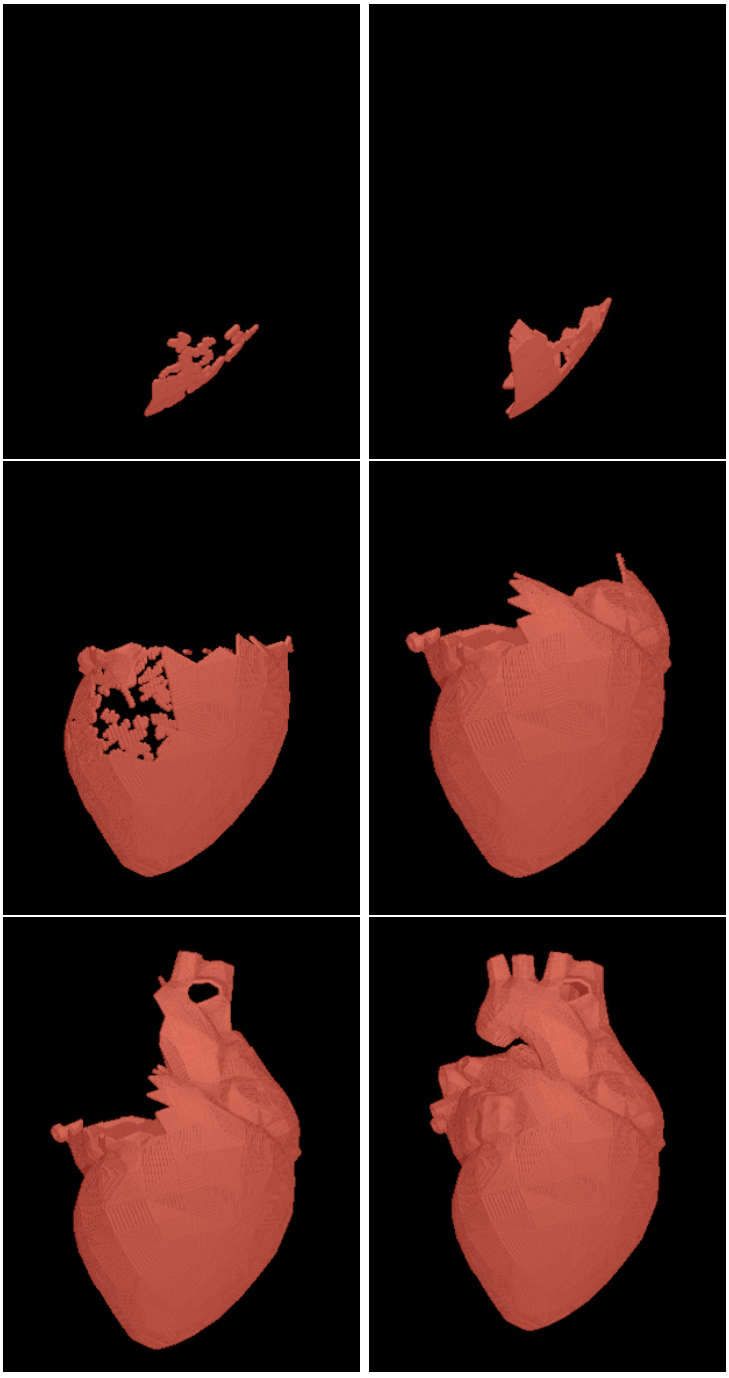


Fig.9 - Heart generation steps by cells as cellular automata belonging to ordinary triangles (POV-Ray 3.7 rendering)

VIEW ANIMATION (requires internet connection)

```
#declare Rnd_1 = seed (1153);
 #declare Rnd_2 = seed (553);
#declare Rnd_3 = seed (876);

 #declare ii = 0;
 #declare pp = 0;
 #declare qq = 0;

//  replace N by  N*clock for animation
union{#for(j,0,int(N/6),1)   //  steps N/6  N/2 24*N  256*N 512*N  630*N
#declare i = ii + st*pow(-1,int(N*rand(Rnd_1)));

#for(k,0,n,1)
#declare p = pp + st*pow(-1,int(n*rand(Rnd_2)));
#declare q = qq + st*pow(-1,int(n*rand(Rnd_3)));

#if (i > 0)
#if(i < N)
  #if (p > 0)
  #if (p < n)
      #if (q > 0)
    #if(q <= p)

#declare pp = p;
#declare qq = q;
#declare ii = i;

#declare X = a[i][3] - (a[i][3] - a[i][0])*p/n + (a[i][6] - a[i][0])*q/n;
#declare Y = a[i][4] - (a[i][4] - a[i][1])*p/n + (a[i][7] - a[i][1])*q/n;
#declare Z = a[i][5] - (a[i][5] - a[i][2])*p/n + (a[i][8] - a[i][2])*q/n;

 sphere { < X, Y, Z >, 0.05
   texture{
   pigment{color < 1.0, 0.3, 0.2 > }
   finish { ambient rgb <0.3,0.1,0.1>
     diffuse 0.3
     reflection 0.0
     specular 0.0}}
 rotate <-90, 180, 0>
rotate 0*clock*360
 translate <0, -3, 0>}
  #end // end if
  #end // end if
  #end // end if
  #end // end if
  #end // end if
  #end // end if
 // #end // end if
#end // k
#end} // j
```

Conclusion

In the present report we have attempted to show how *chance* and *information* need to work together so that ordered structures, like complex systems and in particular living bodies, are allowed to emerge *locally* starting from unorganized matter.

The role of some driving *information* has appeared essential in order to direct the evolutive trajectories of any system towards an organized ordered structure resulting as an *attractor*. *Matter* and *energy* alone prove to be not enough to generate order, because of the second law of thermodynamics, which compels any matter-energy system towards disorder and thermal equilibrium. Even if, locally, some partially ordered structures may emerge, by chance, the probability of such an occurrence is very very small and the entire age of the universe would not be sufficient to produce an organized system comparable to the living beings which actually we observe on our planet. The number of the ordered possible combinations of particles is too small compared with the huge number of disordered combinations. Moreover, even if an ordered configuration might occur by chance, its *stability* in time would be even more and more unlikely.

A further governing principle like *information,* which is *neither matter nor energy* (according to N. Wiener) seems to play an essential role in the process of order and organized systems emergence from matter.

We have shown how *algorithmic information* (in the sense we have proposed just from chapter 1) can operate in order to generate complex systems like *fractals* either starting from an already *ordered sequence of initial conditions* or starting from *random initial conditions,* leading to the same geometry of the resulting objects, as *attractors* towards which the evolutive trajectories are led thanks to information.

Of special interest, in relation to biological systems, has been revealed *cellular automata* since they add to the driving algorithmic information the constraint that any daughter cell is located in contiguity with its mother cell. No matter if the choice of the near location of the daughter cell is chosen by chance. What is relevant is the role of the *law* (*information*) according to which the daughter cell is born.

The techniques implemented to generate fractal shapes have been, finally, applied also to a biological system like a human organ, *e.g.,* the heart, in order to simulate the generation or regeneration of its tissue by a stem cell. We saw how a simple program (*compressed string*) allows to obtain only a rough model of an heart shape, while a true realistic anatomic shape seems to require to know the full (*uncompressed*) list of the co-ordinates localizing the single cells, even if they are schematically represented by small spheres. An intriguing question

arises if a biological organ belonging to a living body can be generated by an algorithm which can be compressed within a relatively short program string, or if an *incompressible* string of data is required to describe each single cell or constituent part of the whole system. Is the *DNA,* and more generally the biological code responsible of a living body generation, more resemblant to a compressible or to an incompressible string of code?

In principle one could guess to model the whole universe as a set of nested attractors.

In the present investigation we have limited ourselves only to attack the problem of the emergence of complex boundary geometrical *shapes* of bodies (like fractals and a living organ) thanks to the concurrence of some *information* (*i.e.,* something resembling an Aristotelian *form*). A More intriguing matter would be, beside that of the generation of the external and internal organized structure of complex systems, that of modeling their behavior along time, *i.e.,* their *dynamics.* So timidly approaching the matter of their *nature* (in the Aristotelian Thomistic sense of the word, *i.e.,* operational ability), together with the matter of their *essence* (*i.e.,* existing ability as organized ordered structures).

Further researches will be required in future to widen the present program of investigating the role of *information* (*form*) as an immaterial principle of *organization* and *activity* of matter-energy. We hope that the *INTERS project on "Form and information"* may offer a suitable context to develop such a stimulating search and will be albe to provide some more relevant results.

Bibliography

[1] Thomas Aquinas, *In Metaphys.,* lib. 2 l. 4 n. 8 in *Index Thomisticus* in www.corpus thomisticum.org/it/index.age.

[2] K. Gödel, "On formally undecidable propositions of *Principia Mathematica* and related systems I", in K. Gödel, *Collected Works,* vol. 1, Oxford University Press, New York (NY) 2001, pgs 144-195. K. Gödel, *Collected Works,* vol. 1, Oxford University Press, New York (NY) 2001.

[3] Thomas Aquinas, in https://dhspriory.org/thomas/QDdeVer2.htmSt. Thomas Aquinas' Works in English.

[4] G.Chaitin, *Information theoretic incomplitness,* World Scientific, Singapore 1992.

[5] S. Wolfram, *A new kind of science,* Wolfram Media Inc., Champaign (IL USA) 2002.

[6] G. Chaitin, *The unknowable,* Springer-Verlag, Singapore 1999.

[7] A. Strumia, *Dalla filosofia della scienza alla filosofia nella scienza,* Sisri-Edusc, Roma 2017.

[8] R.J. Marks II and Oth. (editors), *Biological information. New perspectives,* Proceedings of a Symposium held May 31 through June 3, 2011 at Cornell University, World Scientific, Singapore 2014 (www.worldscientific.com/worldscibooks/10.1142/8818#t=toc).

[9] P.G Gibson, J.R. Baumgardner, W.H. Brewer and J.C. Sanford, "Can Purifying Natural Selection Preserve Biological Information?", in [8], pgs 232-263.

[10] W. Ewert, W.A. Dembski and R.J. Marks II, "Tierra: The Character of Adaptation", in [8] pgs 105-138.

[11] C.W. Nelson and J.C. Sanford, "Computational Evolution Experiments Reveal a Net Loss of Genetic Information Despite Selection", in [8] pgs 338-368.

[12] N. Wiener, *Cybernetics: or the control and communication in the animal and the machine,* Technology Press, MIT, Cambridge (MA), 1965.

[13] W. Gitt, R. Compton and J. Fernandez, "Biological Information. What is It?", in [8], pgs 11-25.

[14] J.W. Oller, Jr., "Pragmatic Information", in [8], pgs 64-86.

[15] E. Sarti, "Information" (www.inters.org/information).

[16] A.Strumia, *Mechanics* (www.inters.org/mechanics).

[17] R. Thom, *Structural stability and morphogenesis: an outline of a general theory of models,* Perseus Books, Cambridge (MA) 1989.

[18] C. Shannon, "A Mathematical Theory of Communication", *The Bell System Technical Journal,* Vol. 27, pgs 379-423, 623-656 (1948).

[19] S. Kaufmann, "Evolution Beyond Entailing Law: The Roles of Embodied Information and Self Organization", in [8], pgs 521-522.

[20] W.F. Basener, "Limits of Chaos and Progress in Evolutionary Dynamics", in [8], pgs 87-104.

[21] G. Montaez, R.J. Marks II, J.Fernandez and J.C. Sanford, "Multiple Overlapping Genetic Codes Profoundly Reduce the Probability of Beneficial Mutation", in [8], pgs 139-167.

[22] J.C. Sanford, "Introduction to the 2^{nd} meeting session", in [8], pgs 203-209.

[23] J. Seaman, "dna.exe: A Sequence Comparison between the Human Genome and Computer Code", in [8], pgs 384-401.

[24] G. Sewell, "Entropy, Evolution and Open Systems", in [8], pg 168-178.