CZECH TECHNICAL UNIVERSITY IN PRAGUE
Faculty of Nuclear Sciences and Physical Engineering

Department of Mathematics
Branch of Studies: Applied Information Technology

Visualization of fractal sets in multi-dimensional spaces

Bachelor‘s Degree Project

Author: Petr Šupina
Advisor: Doc.Dr.Ing. Michal Beneš
Academic Year: 2005/2006

Acknowledgments:

I would like to thank Doc.Dr.Ing. Michal Beneš and Ing. Vladimír Chalupecký for their expert guidance, and express my gratitude to Prom. fil. Irena Dvořáková for grammar corrections.

Declaration:

I declare that this bachelor project is all my own work and I have cited all sources I have used in the bibliography.

In Prague, May 29th 2006 Petr Šupina

Název práce:
Vizualizace fraktálních množin ve vícerozměrných prostorech

Autor: Petr Šupina

Obor: Inženýrská informatika
Druh práce: Bakalářská práce

Vedoucí práce: Doc.Dr.Ing. Michal Beneš. Katedra matematiky, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze

Konzultant: Ing. Vladimír Chalupecký. Katedra matematiky, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze

Abstrakt: Práce seznamuje čtenáře s algebraickými základy pro rozšíření komplexních čísel jako jsou kvaterniony a Davenportova hyperkomplexní čísla. Tato rozšíření jsou dále použita pro generování escape-time fraktálů a počítání fraktálních dimenzí. Práce je doprovázena softwarovým projektem, který implementuje jednotlivé vizualizační nástroje, a výsledky výpočtů. Vizualizační nástroje zahrnují dvourozměrná a třírozměrná zobrazení aproximací podmnožin hyperkomplexních fraktálů. Je také poskytnuto srovnání Ray-tracingu s Marching Cubes algoritmem. Dále jsou zmíněny možné problémy související s numerickou aproximací fraktálních dimenzí.

Klíčová slova: fraktál, hyperkomplexní, Davenportova čísla, kvaternion, vizualizace.

Title:
Visualization of fractal sets in multi-dimensional spaces

Author: Petr Šupina

Abstract: The project familiarizes the reader with algebraic principles of extensions to complex numbers such as quaternions and Davenport hypercomplex numbers. These extensions are then used to generate escape-time fractals and to calculate fractal dimensions. The project is accompanied by a software project implementing various visualization tools, and visualization results. The visualization tools include 2-dimensional and 3-dimensional views of approximations of hypercomplex fractal subsets. A comparison between Ray-tracing and Marching Cubes algorithms is also included. Possible problems related to numerical approximation of fractal dimensions are mentioned as well.

Key words: fractal, hypercomplex, Davenport numbers, quaternion, visualization.

Table of Contents

1. Preface

Everyone has seen some type of a fractal. It may have been a picture, and there are many fractals in nature. Soon, people started to realize that some of the natural shapes can be expressed mathematically. After a few centuries, the first complex fractal was introduced. Such a discovery could not have been achieved without the means to visualize such a fractal. The greatest visual aid was, and still is, a computer. Even though it has very limited precision, several algorithms were devised to make it as accurate as possible. Even nowadays, it is a big task to compromise between accuracy and computation time.

It may seem unlikely, but the main purpose of complex fractals is generating pictures. There are millions of fractal pictures in thousands of picture galleries all over the Internet. It has become an art lately.

Some interesting extensions to complex fractals were invented to expand the already infinite varieties of fractals. Typically, these extensions are based on algebras that extend complex numbers to more dimensions preserving complex numbers as their subset. Such algebras are usually difficult to visualize since they are more than 3-dimensional. The task of a visualization tool is to choose a subset of a fractal set and calculate an approximation having limited complexity. The most popular are 3D subsets because 2D subsets are very similar to complex fractals.

Our goal in this project is to develop algebras extending complex numbers and use them to generate various fractals similar to complex fractals in definition.

An additional task is to devise a method to determine the complexity of a set and distinguish between a regular object and a fractal. Such a method involves calculating the fractal dimension.

The work belongs to the framework of the research project “Applied Mathematics in Technical and Physical Sciences” of the Ministry of Education, Youth and Sport of the Czech Republic, Project Number 6840770010.

2. Algebras

2.1. Complex Numbers

2.1.1. Properties

All complex numbers are of the form $a+ib$, where a and b are real numbers, and i represents an imaginary number with $i^2=-1$, that is, i is the square root of -1. If b is 0, then the complex number corresponds to the real number a. In some fields, especially in electrical engineering, these numbers are written as $a+jb$.

There are other ways to express these numbers. One of them is called mod-arg form which uses the complex modulus and argument. The third form uses Euler's formula $e^{ix}=\cos x+i\sin x$:

$$z=a+ib=r(\cos\varphi+i\sin\varphi)=r\,\mathrm{e}^{i\varphi}, \tag{2.1}$$

where $r=|z|$ and $\varphi=\arg z$.

Real numbers are a subset of complex numbers if the imaginary part is zero.

Since Gauss proved the Fundamental Theorem of Algebra [1], we know that every polynomial equation having complex coefficients and a degree ≥ 1 has at least one complex root.

All complex numbers are usually denoted by $\mathbb{C}$. Associative, commutative, and distributive laws of algebra can be applied to them.

A complex number can be viewed as a point or a vector in a two-dimensional Cartesian coordinate system called the complex plane or Gauss plane. Therefore, we can use the xy-plane to display complex numbers. Thus, we can imagine arithmetic operations and some of their properties.

It is not possible to order complex numbers because they cannot be turned into an ordered field. This property disallows some operations such as $<, >, \leq,$ or $\geq$.

2.1.2. Conjugate

The conjugate of a complex number $x=a+ib$ is defined as:

$$x^*=a-ib, \tag{2.2}$$

inverting the sign of the imaginary part.

The complex conjugation is a very "transparent" operation. It commutes with all the arithmetic operations: sum, difference, product, or quotient in the sum… Such an operation is called a field isomorphism.

2.1.3. Addition and Subtraction

To add or subtract two complex numbers, add or subtract the corresponding real and imaginary parts.

Addition can be interpreted as a transformation of the plane $\mathbb{C}$. Every point is moved in the same direction and distance. We can say that addition gives a translation of the plane $\mathbb{C}$.

The negation of $a+ib$ is $-a-ib$, so the negation of a complex number will be located just opposite 0 and in the same distance from 0. Negation can be interpreted as a transformation of the plane $\mathbb{C}$, too. A rotation by 180° around 0 gives the negation of the entire set $\mathbb{C}$.

2.1.4. Absolute Value

The absolute value $|x|$ of a real number x is the number itself if it is positive or zero, but if x is negative, then its absolute value $|x|$ is its negation $-x$, that is, the corresponding positive value. For a complex number $z=a+ib$, we define the absolute value $|z|$ as being the distance from z to 0 in the plane $\mathbb{C}$:

$$|z|=\sqrt{a^2+b^2}. \qquad (2.3)$$

Note that this operation maps $\mathbb{C}$ into $\mathbb{R}$ (real numbers), so it can be used as a replacement for some comparison operations.

2.1.5. Multiplication

Let us take 2 complex numbers $a+ib$ and $c+id$, multiply them, and extract the real and imaginary parts, which will give the result:

$$(a+ib)(c+id)=(ac-bd)+i(bc+ad). \qquad (2.4)$$

To multiply a complex number by a real number, set d to 0:

$$(a+ib)c=ca+icb. \qquad (2.5)$$

Let us consider multiplying of an arbitrary complex number $z=a+ib$ by i:

$$(a+ib)i=-b+ia. \qquad (2.6)$$

If we interpreted this statement geometrically, we would discover that the multiplication by i gives a 90° counterclockwise rotation around 0.

The absolute value of x times y is the absolute value of x times the absolute value of y. In fact, this is true in general:

$$|xy|=|x||y|. \qquad (2.7)$$

In order to prove this, we shall prove that it is true for the squares, so we do not have to deal with square roots. We will show that $|xy|^2 = |x|^2 |y|^2$. Let x be $a+ib$, and let y be $c+id$. According to the formula for multiplication, xy equals $(ac-bd)+i(bc+ad)$, therefore:

$$|xy|^2 = (ac-bd)^2 + i(bc+ad)^2. \tag{2.8}$$

In order to show that $|xy|^2 = |x|^2 |y|^2$, all we have to do is show that:

$$(ac-bd)^2 + (bc+ad)^2 = (a^2+b^2)(c^2+d^2). \tag{2.9}$$

We have seen two special cases of multiplication: one by a real number which leads to scaling, the other by i which leads to rotation. The general case is a combination of scaling and rotation.

Let x and y be points in the complex plane $\mathbb{C}$. The lengths of lines from 0 to x and 0 to y are the absolute values $|x|$ and $|y|$. We already know the length of the line from 0 to xy is going to be the absolute value $|xy|$ which equals $|x||y|$. What we do not know is the direction of the line from 0 to xy. The answer is that "angles add". We will determine the direction of the line from 0 to x by a certain angle called the argument of x, usually denoted $\arg x$. This is the angle between the positive real axis and the line from 0 to x. The other point y has the angle $\arg y$. Then the product xy will have an angle which is the sum of the angles $\arg x + \arg y$.

Function $\arg x$ can be calculated using the expression $arctg\,\frac{b}{a}$, but it is important to correctly determine the quadrant of the vector (a,b), because, for example, $\frac{-b}{-a} = \frac{b}{a}$.

2.1.6. Powers of i

Another special case of multiplication is calculating various powers of the imaginary unit i. We started with the assumption that $i^2 = -1$. After a brief investigation, we can conclude that the following sequence is repeating:

$$\begin{aligned} i^{4n} &= 1 \\ i^{4n+1} &= i \\ i^{4n+2} &= -1 \\ i^{4n+3} &= -i, \end{aligned} \tag{2.10}$$

where n is an integer.

The sequence also gives the reciprocal i^{-1} of i, which is $-i$. This means that it is a number whose reciprocal is its own negation. It is easy to check that i times $-i$ is 1, so i and $-i$ are reciprocals.

2.1.7. Reciprocal Value

Our goal is to find x^{-1} given a complex number x. In other words, given a complex number $x = a + ib$, find another complex number $y = c + id$ such that $xy = 1$. We will use the product formula we developed in the section on multiplication:

$$(a+ib)(c+id) = (ac-bd) + i(bc+ad). \tag{2.11}$$

If two complex numbers are equal, their real parts are equal and their imaginary parts as well. Because we need $xy = 1$, we can write:

$$(ac-bd) + i(bc+ad) = 1. \tag{2.12}$$

This gives two equations:

$$\begin{aligned} ac - bd &= 1 \\ bc + ad &= 0. \end{aligned} \tag{2.13}$$

In our case, x was given and y was unknown, so in these two equations a and b are given, and c and d are the unknowns to be solved. Since the above is a simple system of linear equations, we can conclude that:

$$\begin{aligned} c &= \frac{a}{a^2+b^2} \\ d &= \frac{-b}{a^2+b^2}. \end{aligned} \tag{2.14}$$

Note that $a^2 + b^2 = |x|^2$.

This gives the reciprocal of x, which is $y = c + id$.

Complex conjugates can be used to interpret reciprocals. We can easily check that a complex number $x = a + ib$ multiplied by its conjugate $a - ib$ is the square of its absolute value $|x|^2$.

Therefore, $\frac{1}{x}$ is the conjugate of x divided by the square of its absolute value $|x|^2$:

$$\frac{1}{x} = \frac{x^*}{|x|^2} = \frac{x^*}{x^* x}. \tag{2.15}$$

However, the simplest way to calculate the reciprocal is to use the dot product:

$$\frac{1}{x} = \frac{x^*}{\langle x|x\rangle} = \frac{x^*}{a^2+b^2}. \tag{2.16}$$

2.1.8. Division

Just as subtraction can be compounded from addition and negation, division can be compounded from multiplication and reciprocation. Being given x, we want to find $\frac{1}{x}$.

We can safely say that division exists for all complex numbers except for dividing by 0, because $zx^{-1} = x^{-1}z = \frac{z}{x}$, that is, the commutative law applies.

2.1.9. Logarithm

The logarithm of a complex number x is defined as every complex number y which satisfies the equation: $e^y = x$.

This is denoted by $\ln x := y$.

The solution is obtained by using the form $e^y = re^{i\varphi}$, where $r = |x|$, and $\varphi = \arg x$, so the result is:

$$y = \ln x = \ln |x| + \ln e^{i \arg x} = \ln |x| + i \arg x. \qquad (2.17)$$

The complex logarithm is defined for all $x \neq 0$, and is multi-valued.

2.1.10. Exponential Function

Let $x = a + ib$ and $y = c + id$ be from $\mathbb{C}$.

The exponential function retains the important properties:

$$\begin{aligned} e^{x+y} &= e^x e^y \\ e^0 &= 1 \\ e^x &\neq 0. \end{aligned} \qquad (2.18)$$

We can write: $e^{a+ib} = e^a e^{ib}$.

The expression e^a is the exponential function for real numbers, and, to calculate e^{ib}, we can use the above mentioned Euler's formula. First, let us prove the formula using section 2.1.6. and the Taylor series expansions of the real functions e^x, $\cos x$, and $\sin x$. Instead of using real arguments, we will use iz, where z is a real number. This is possible, because the radius of convergence of all three series is infinite.

$$\begin{aligned}e^{iz} &= 1+iz+\frac{(iz)^2}{2!}+\frac{(iz)^3}{3!}+\frac{(iz)^4}{4!}+\ldots=\\&=1+iz-\frac{z^2}{2!}-i\frac{z^3}{3!}+\frac{z^4}{4!}+i\frac{z^5}{5!}-\frac{z^6}{6!}-i\frac{z^7}{7!}+\frac{z^8}{8!}+\ldots=\\&=(1-\frac{z^2}{2!}+\frac{z^4}{4!}-\frac{z^6}{6!}+\frac{z^8}{8!}-\ldots)+i(z-\frac{z^3}{3!}+\frac{z^5}{5!}-\frac{z^7}{7!}+\ldots)=\\&=\cos z+i\sin z.\end{aligned} \tag{2.19}$$

The rearrangement of terms is possible, because the series are absolutely convergent.

We have proved that $e^{ib}=\cos b+i\sin b$, therefore:

$$e^x = e^a(\cos b+i\sin b), \tag{2.20}$$

where all functions are real.

De Moivre's formula can also be used:

$$(\cos z+i\sin z)^n = \cos nz+i\sin nz, \tag{2.21}$$

for z from $\mathbb{R}$, and n from $\mathbb{Z}$.

The same formula can be used with z and n from $\mathbb{C}$. In this case, $(\cos z+i\sin z)^n$ is a multi-valued function while $\cos nz+i\sin nz$ is not, and one can state that the latter is a value of the first.

We often desire a more generalized form of the function with a variable base. This can be achieved simply, thanks to the fact that $\ln x$ is the inverse function to e^x:

$$x^y = e^{y\ln x}, \tag{2.22}$$

where all numbers are complex.

2.2. Quaternions

2.2.1. Properties

Quaternions are members of a non-commutative division algebra. A division algebra is a ring, in which every nonzero element has a multiplicative inverse.

They are an extension of complex numbers first invented by Sir William Rowan Hamilton in 1843 [2]. Even though their main disadvantage is that they do not obey the commutative law, they are used in particular for intuitive three-dimensional rotations.

The quaternions form a 4-dimensional division algebra over the real numbers, which is also a normed vector space often denoted by H (for Hamilton).

The numbers are obtained by adding the elements i, j, and k to the real numbers, which satisfy the following relations:

$$i^2 = j^2 = k^2 = ijk = -1. \tag{2.23}$$

The multiplication is assumed to be associative, therefore we can immediately write:

$$\begin{aligned} ij &= k, ji = -k \\ jk &= i, kj = -i \\ ki &= j, ik = -j. \end{aligned} \tag{2.24}$$

To get the first equation, we use the definition $ijk = -1$ right-multiplied on both sides by k:

$$\begin{aligned} ijkk &= -k \\ ij(-1) &= -k \\ ij &= k. \end{aligned} \tag{2.25}$$

By left-multiplying and right-multiplying on both sides by i, j, or k, we get the remaining equations.

The equations prove that the quaternions are not commutative, for example, $ij = k \neq ji = -k$. This has some unexpected consequences, for instance, polynomial equations can have more distinct solutions than the degree of the polynomial. The equation $z^2 + 1 = 0$ has infinitely many solutions $z = bi + cj + dk$ with $b^2 + c^2 + d^2 = 1$.

Every quaternion is a real linear combination of the basis quaternions 1, i, j, and k:

$$a + bi + cj + dk. \tag{2.26}$$

The scalar part of the quaternion is a while the remainder is the vector part.

In the following sections, quaternions defined as:

$$\begin{aligned} p &= a + \vec{u} = a + bi + cj + dk \\ q &= t + \vec{v} = t + xi + yj + zk, \end{aligned} \tag{2.27}$$

where $\vec{u}$ represents the vector (b, c, d), and $\vec{v}$ represents the vector (x, y, z), will be used.

For instance, a rotation around the unit vector $\vec{n}$ by an angle θ can be computed using the quaternion $p = (a, \vec{u}) = (\cos\frac{1}{2}\theta, \vec{n}\sin\frac{1}{2}\theta)$.

2.2.2. Conjugate

The conjugate p^* of the quaternion p is defined as:

$$p^* = a - bi - cj - dk. \tag{2.28}$$

Note that $(pq)^* = q^* p^*$, which is not generally equal to $p^* q^*$.

2.2.3. Addition and Subtraction

The addition of two quaternions p and q is equivalent to summing each of the elements together:

$$p + q = (a + t) + (b + x)i + (c + y)j + (d + z)k. \tag{2.29}$$

Subtraction is computed by negating all the elements of q and using addition.

Addition follows the commutativity and associativity rules of real and complex number.

2.2.4. Absolute Value

The absolute value of p is the non-negative real number defined by:

$$|p| = \sqrt{p^* p} = \sqrt{p \cdot p} = \sqrt{a^2 + b^2 + c^2 + d^2}. \tag{2.30}$$

We can also write $|pq| = |p||q|$ for all quaternions p and q. The proof is similar to the one for complex numbers.

If we define the distance function (metric) $d(p, q) = |p - q|$, the quaternions form a metric space.

2.2.5. Multiplication

The non-commutative multiplication between two quaternions p and q is sometimes called the Grassmann product. The complete form is described here:

$$pq = at - \vec{u} \cdot \vec{v} + a\vec{v} + t\vec{u} + \vec{u} \times \vec{v} \tag{2.31}$$

or

$$pq = (at - bx - cy - dz) + (bt + ax + cz - dy)i + (ct + ay + dx - bz)j + (dt + az + by - cx)k. \quad (2.32)$$

Notice the difference between pq and qp:

$$qp = at - \vec{u} \cdot \vec{v} + a\vec{v} + t\vec{u} - \vec{u} \times \vec{v}. \quad (2.33)$$

The product of a quaternion and its conjugate is a scalar.

2.2.6. Dot-product

The dot-product is equivalent to a 4-vector dot product. The dot-product is the sum of each element of p multiplied by each element of q. It is a commutative product between quaternions, and returns a scalar value:

$$p \cdot q = at + \vec{u} \cdot \vec{v} = at + bx + cy + dz. \quad (2.34)$$

The dot-product can be rewritten using multiplication:

$$p \cdot q = \frac{pq + qp}{2}. \quad (2.35)$$

It is also useful to isolate an element from a quaternion. For example, the k term can be isolated from p: $p \cdot k = d$.

2.2.7. Reciprocal Value

The inverse of a quaternion is defined so that the equation $p^{-1}p = 1$ is satisfied. It is derived in the same way that the complex inverse is found:

$$p^{-1} = \frac{p^*}{p^* p}. \quad (2.36)$$

The division of a quaternion by a scalar is equivalent to multiplication by the scalar inverse, such that each element of the quaternion is divided by the divisor.

Note that division in the form $\frac{p}{q}$ is not defined for quaternions, because $p^{-1}q \neq qp^{-1}$.

2.2.8. Sign

The sign of a complex number returns the complex number of the same direction found on the unit circle. The quaternion sign also produces the unit quaternion:

$$\operatorname{sgn} p = \frac{p}{|p|}. \tag{2.37}$$

2.2.9. Argument

The argument finds the angle between the 4-dimensional quaternion vector and the unit scalar $1 + 0i + 0j + 0k$. It returns the scalar angle:

$$\arg p = \arccos \frac{a}{|p|}. \tag{2.38}$$

2.2.10. Logarithmic and Exponential Function

Exponential and logarithmic functions can be defined because quaternions have a division algebra.

$$\begin{aligned} \ln p &= \ln|p| + \operatorname{sgn}\vec{u} \arg p \\ e^p &= e^a (\cos|\vec{u}| + \operatorname{sgn}\vec{u} \sin|\vec{u}|) \\ p^q &= e^{q \ln p}. \end{aligned} \tag{2.39}$$

These equations can be derived similarly like for the complex numbers.

2.3. Hypercomplex Numbers

There are multiple distinct definitions of hypercomplex numbers, which leads to a certain confusion. Higher dimensional numbers in the Clifford Algebra are often called hypercomplex even though they do not have all the properties of complex numbers and no classical function theory can be constructed over them.

A more general definition says that a hypercomplex number is a number having properties departing from those of the real and complex numbers [3]. The most common examples are 4-dimensional tessarines, coquaternions, and above mentioned quaternions, 8-dimensional biquaternions and octonions, and 16-dimensional sedenions.

There is another definition of "the" hypercomplex numbers brought to us by Clyde Davenport [4]. These 4-dimensional numbers will be described in more detail in the following paragraphs, referred to as *the* hypercomplex numbers.

Unlike quaternions, the hypercomplex numbers satisfy the commutative law of multiplication. The law which does not apply is the field property that states that all non-zero elements of a field have a multiplicative inverse, so it is not a division algebra. For a non-zero hypercomplex number p, the multiplicative inverse p^{-1} does not always exist.

We will define multiplication using the basis vectors 1, i, j, and k as with quaternions, and the following applies:

$$\begin{aligned} ij &= ji = k \\ jk &= kj = -i \\ ki &= ik = -j \\ ii &= jj = -kk = -1 \\ ijk &= 1. \end{aligned} \tag{2.40}$$

Note that now $ij = k$ and $ji = k$, and similarly for other products of pairs of basis vectors, so the commutative law holds.

Providing that $p = a + bi + cj + dk$ and $q = t + xi + yj + zk$ are two hypercomplex numbers, the following functions and properties can be defined.

2.3.1. Addition and Subtraction

The addition of two hypercomplex numbers p and q is equivalent to the definition for quaternions:

$$p + q = (a+t) + (b+x)i + (c+y)j + (d+z)k. \tag{2.41}$$

Subtraction is computed by negating all the elements of q and using addition.

Commutativity and associativity are preserved.

2.3.2. Absolute Value

The absolute value of p is the non-negative real number defined by:

$$|p| = \sqrt{a^2 + b^2 + c^2 + d^2}, \tag{2.42}$$

because the hypercomplex numbers can be represented as a 4-dimensional Euclidean space.

2.3.3. Multiplication

By multiplying p by q, we get the following commutative expression:

$$pq = (at - bx - cy + dz) + (bt + ax - dy - cz)i + (ct - dx + ay - bz)j + (dt + cx + by + az)k. \tag{2.43}$$

2.3.4. Reciprocal Value

To compute reciprocal, the matrix representation of a hypercomplex number is necessary. The matrix representation can be created, for instance, by trial and error:

$$\begin{aligned} p = a\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} + b\begin{pmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix} + c\begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} + d\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \\ = \begin{pmatrix} a & -b & -c & d \\ b & a & -d & -c \\ c & -d & a & -b \\ d & c & b & a \end{pmatrix}. \end{aligned} \tag{2.44}$$

The matrix on the right side has the usual matrix inverse. It is remarkable that the matrix inverse of the typical matrix element is another matrix having the same pattern of entries:

$$p^{-1} = \begin{pmatrix} a & -b & -c & d \\ b & a & -d & -c \\ c & -d & a & -b \\ d & c & b & a \end{pmatrix}^{-1} \tag{2.45}$$

$$\begin{aligned} p^{-1} = 1\left[a(a^2 + b^2 + c^2 + d^2) - 2d(ad - bc)\right]/\det p \\ + i\left[-b(a^2 + b^2 + c^2 + d^2) - 2c(ad - bc)\right]/\det p \\ + j\left[-c(a^2 + b^2 + c^2 + d^2) - 2b(ad - bc)\right]/\det p \\ + k\left[d(a^2 + b^2 + c^2 + d^2) - 2a(ad - bc)\right]/\det p \end{aligned} \tag{2.46}$$

The reader may verify that if one multiplies p by p^{-1}, or vice-versa, one obtains a unity result.

It fails to be a field only because the denominator in the vector form inverse is the determinant of the 4-D element in matrix form:

$$\det p = \left[(a-d)^2 + (b+c)^2\right]\left[(a+d)^2 + (b-c)^2\right] \tag{2.47}$$

which is zero under the conditions $(a = d \wedge b = -c) \vee (a = -d \wedge b = c)$, therefore the ring is not defined under these conditions.

2.3.5. Generalization

The hypercomplex numbers have a generalization of any unary complex valued function defined on the complex numbers. Note that a hypercomplex number p can be represented as a pair of complex numbers in the following way:

$$\begin{aligned} r &= (a-d) + i(b+c) \\ s &= (a+d) + i(b-c). \end{aligned} \tag{2.48}$$

The numbers r and s are complex numbers.

Conversely, if we have a hypercomplex number in the form (r, s), we can solve a, b, c, and d. The solution is:

$$\begin{aligned} a &= \frac{\operatorname{Re} r + \operatorname{Re} s}{2} = \frac{a-d+a+d}{2} = a \\ b &= \frac{\operatorname{Im} r + \operatorname{Im} s}{2} \\ c &= \frac{\operatorname{Im} r - \operatorname{Im} s}{2} \\ d &= \frac{\operatorname{Re} s - \operatorname{Re} r}{2}. \end{aligned} \tag{2.49}$$

We can now, for example, compute e^p. First, the two complex numbers r and s are computed as above, then $r = e^r$ and $s = e^s$ is calculated, where e^x is the complex version of the exponential function. The equations above are used to solve a, b, c, and d. The hypercomplex number thus obtained is $p = e^p$.

These equations can be clarified by introducing yet another representation of the algebra:

$$p = \frac{\left[(a-d) + i(b+c)\right](1-k) + \left[(a+d) + i(b-c)\right](1+k)}{2}. \tag{2.50}$$

Let us substitute $r = (a-d) + i(b+c), s = (a+d) + i(b-c), \varepsilon = \dfrac{1-k}{2}$, and $\eta = \dfrac{1+k}{2}$.

So $p = r\varepsilon + s\eta$, where r and s are obviously complex variables, and for a positive integer n:

$$\begin{aligned} \varepsilon^n &= \varepsilon \\ \eta^n &= \eta \\ \varepsilon\eta &= (0,0,0,0) \\ \varepsilon \cdot \eta &= 0 \text{ (vector dot-product)} \\ \varepsilon + \eta &= 1 \\ -\varepsilon + \eta &= k. \end{aligned} \tag{2.51}$$

Consequently, the algebra operations can be written as:

$$\begin{aligned} p + q &= (r_1 + r_2)\varepsilon + (s_1 + s_2)\eta \\ p - q &= (r_1 - r_2)\varepsilon + (s_1 - s_2)\eta \\ pq &= (r_1 r_2)\varepsilon + (s_1 s_2)\eta \\ p^{-1} &= r^{-1}\varepsilon + s^{-1}\eta. \end{aligned} \tag{2.52}$$

Because of the way that 4D analytic functions are defined, they have the same properties as for the corresponding complex-valued functions and we can use the same notation as for the complex variables. We have extended the complex analysis to treat a 4D variable. This result is not possible with non-commutative quaternions.

3. Fractals

The word "fractal" has two related meanings. It denotes a shape that is recursively constructed or self-similar, that is, a shape that appears similar at all scales of magnification and is therefore often referred to as "infinitely complex". In mathematics a fractal is a geometric object or quantity that satisfies a specific condition, namely having a Hausdorff dimension greater than its topological dimension. A plot of the quantity on a log-log graph versus the scale then gives a straight line whose slope is the Hausdorff dimension. An example of a fractal is the length of a coastline measured with different length rulers. The shorter the ruler, the longer the length measured, a paradox known as the coastline paradox. The term fractal was coined by Benoît Mandelbrot [5], from the Latin word fractus, meaning "broken" or "fractured".

Objects that are now called fractals were discovered and explored long before the word was coined.

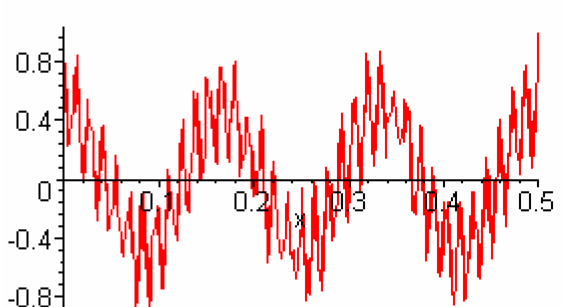


Figure 3.1

The idea of "recursive self similarity" was originally developed by the philosopher Leibniz and he even worked out many of the details. In 1872, Karl Weierstrass [6] found an example of a function with the non-intuitive property that it is everywhere continuous but nowhere differentiable - the graph of this function would now be called a fractal:

$$f(x) := \lim_{n\to+\infty} \sum_{k=1}^{n} a^k \cos b^k \pi x, \tag{3.1}$$

where a is a real numbers in the interval $(0,1)$, and b is an integer such that $ab > 1 + \frac{3}{2}\pi \sim 5.712...$, for instance, Figure 3.1 shows a graph for $a = \frac{1}{2}, b = 12$, and n limited to 20.

In 1904, Helge von Koch [7], dissatisfied with Weierstrass's very abstract and analytic definition, gave a more geometric definition of a similar function, which is now called the Koch snowflake (Figure 3.2). The idea of self-similar curves was taken further by Paul Pierre Lévy, who described a new fractal curve, the Lévy C curve (Figure 3.3).

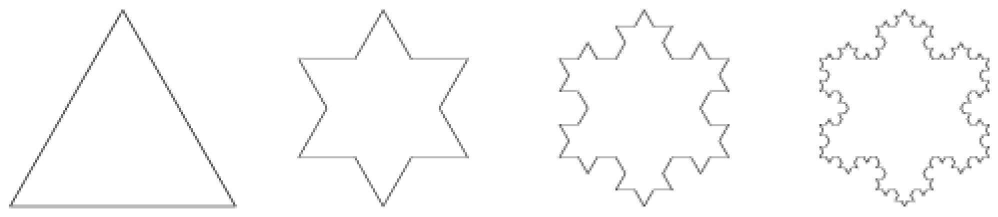


Figure 3.2: Koch snowflake

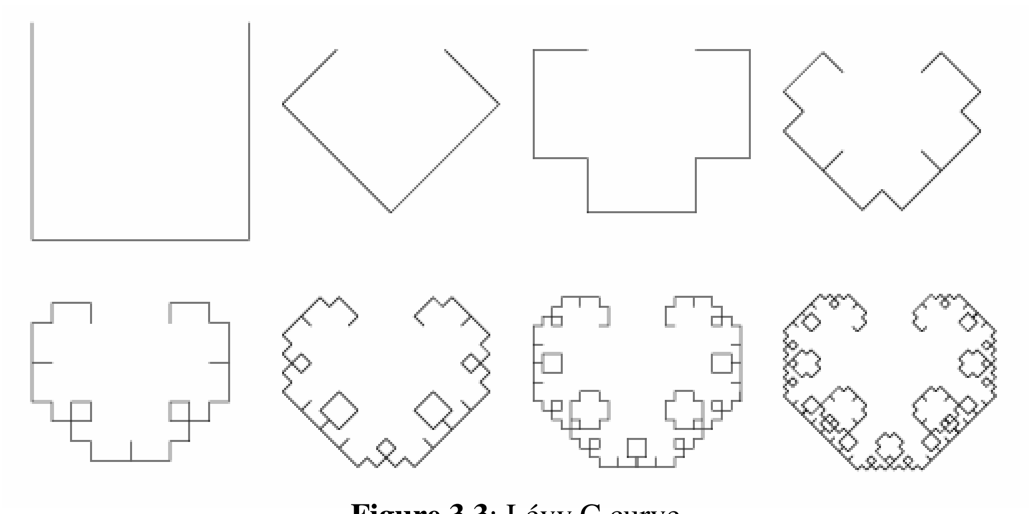


Figure 3.3: Lévy C curve

Georg Cantor gave examples of subsets of the real line with unusual properties - these Cantor sets are now also recognised as fractals.

Iterated functions in the complex plane had been investigated in the late 19th and early 20th centuries by Henri Poincaré, Felix Klein, Pierre Fatou, and Gaston Julia [7]. However, they lacked the means to visualize the beauty of many of the objects that can now be displayed with the aid of modern computer graphics.

In the 1960s, Benoît Mandelbrot started investigating self-similarity in papers such as "How Long Is the Coast of Britain?". In 1975, he coined the word fractal to denote an object whose Hausdorff-Besicovitch dimension is greater than its topological dimension. He illustrated this mathematical definition using computer visualizations.

Additional examples of fractals include the Lyapunov fractal simulating population growth, Sierpiński triangle (Section 3.4.2) and carpet, Menger sponge, space-filling curve (Figure 3.4), dragon curve (Figure 3.5), or the Koch curve (Section 3.3.1).

Fractals can be deterministic or stochastic, that is, non-deterministic.

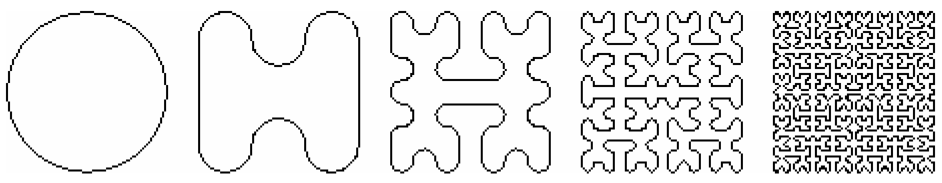


Figure 3.4: Space-filling Curve (Peano Curve)

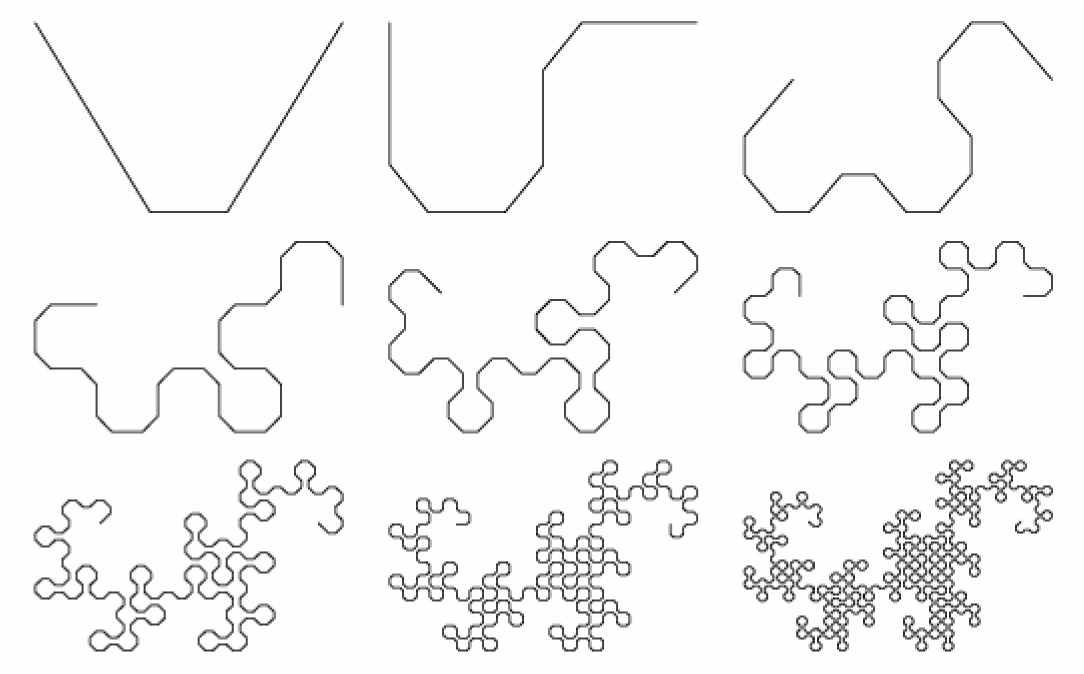


Figure 3.5: Dragon Curve

3.1. Techniques for Generating

Three common techniques for generating fractals are:

- Iterated function systems - These fractals have a fixed geometric replacement rule. Cantor set, Sierpiński carpet, Sierpiński gasket, Peano curve (Figure 3.4), Koch snowflake (Figure 3.2), Harter-Heighway dragon curve (Figure 3.5), T-Square (Figure 3.6), and Menger sponge are some examples of such fractals.
- Escape-time fractals – Fractals that are defined by a recurrence relation at each point in a space such as the complex plane. Examples of this type are the Mandelbrot set, Julia sets (Section 3.4.1), and the Lyapunov fractal.
- Random fractals – Fractals that are generated by stochastic rather than deterministic processes, for example, fractal landscapes (Perlin Noise), Lévy flight and the Brownian tree.

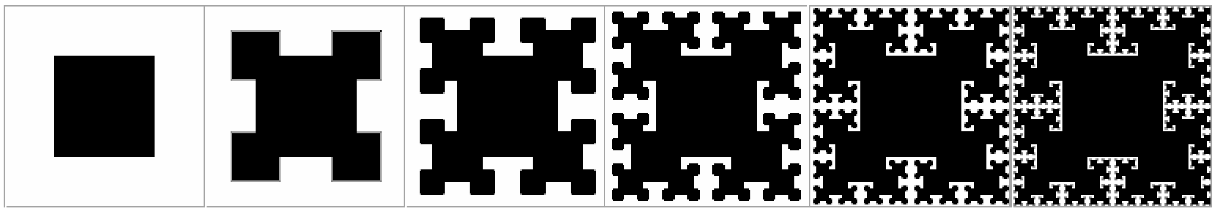


Figure 3.6: T-Square

3.2. Self-similarity

Fractals can also be classified according to their self-similarity. There are three types of self-similarity:

- Exact self-similarity - This is the strongest type of self-similarity; the fractal appears identical at different scales. Fractals defined by iterated function systems often display exact self-similarity.
- Quasi-self-similarity - This is a loose form of self-similarity. The fractal appears approximately (but not exactly) identical at different scales. Quasi-self-similar fractals contain small copies of the entire fractal in distorted and degenerated forms. Fractals defined by recurrence relations are usually quasi-self-similar but not exactly self-similar.
- Statistical self-similarity - This is the weakest type of self-similarity. The fractal has numerical or statistical measures which are preserved across scales. Many definitions of "fractal" imply some form of statistical self-similarity. The fractal dimension itself is a numerical measure which is preserved across scales. Random fractals are examples of fractals which are statistically self-similar, but neither exactly nor quasi-self-similar.

It should be noted that not all self-similar objects are fractals, for example, the real line (a straight Euclidean line) is exactly self-similar, but since its Hausdorff dimension and topological dimension are both equal to one, it is not a fractal.

3.3. Iterated Function Systems (IFS)

Iterated Function Systems are a finite set of contraction maps w_i for $i = 1, 2, ..., n$, each with a contractivity factor $r_i < 1$, which map a compact metric space onto itself. It is the basis for fractal image compression and interpolation techniques.

An IFS fractal is a solution to a recursive set equation. The fractal is made up of the union of several copies of itself, each copy being transformed by a function (hence "function system"). The functions are normally contractive, which means they bring points closer together and make shapes smaller. The shape of an IFS fractal is made up of several possibly-overlapping smaller copies of itself, each of which is also made up of copies of itself. This is the source of its self-similar fractal nature [8].

The most common algorithm to compute IFS fractals is called the chaos game. It consists of choosing a random point in the plane, then iteratively applying one of the functions chosen randomly from the function system and drawing the point. An alternative algorithm is to generate each possible sequence of functions up to a given maximum length, and then to plot the results of applying each of these sequences of functions to an initial point or shape.

3.3.1. Koch Curve

A unit line is the initial state before iterating. The first step involves removing the middle one-third of the unit line and replacing it by two line segments, each one-third long, creating a bottomless triangle in the middle. The object now contains four equal line segments. This is the first iteration. In the next iteration, each of the four lines is replaced by another four lines using the same replacement rule, and so on. The procedure is repeated endlessly to generate the Koch curve, a classical example of an IFS fractal.

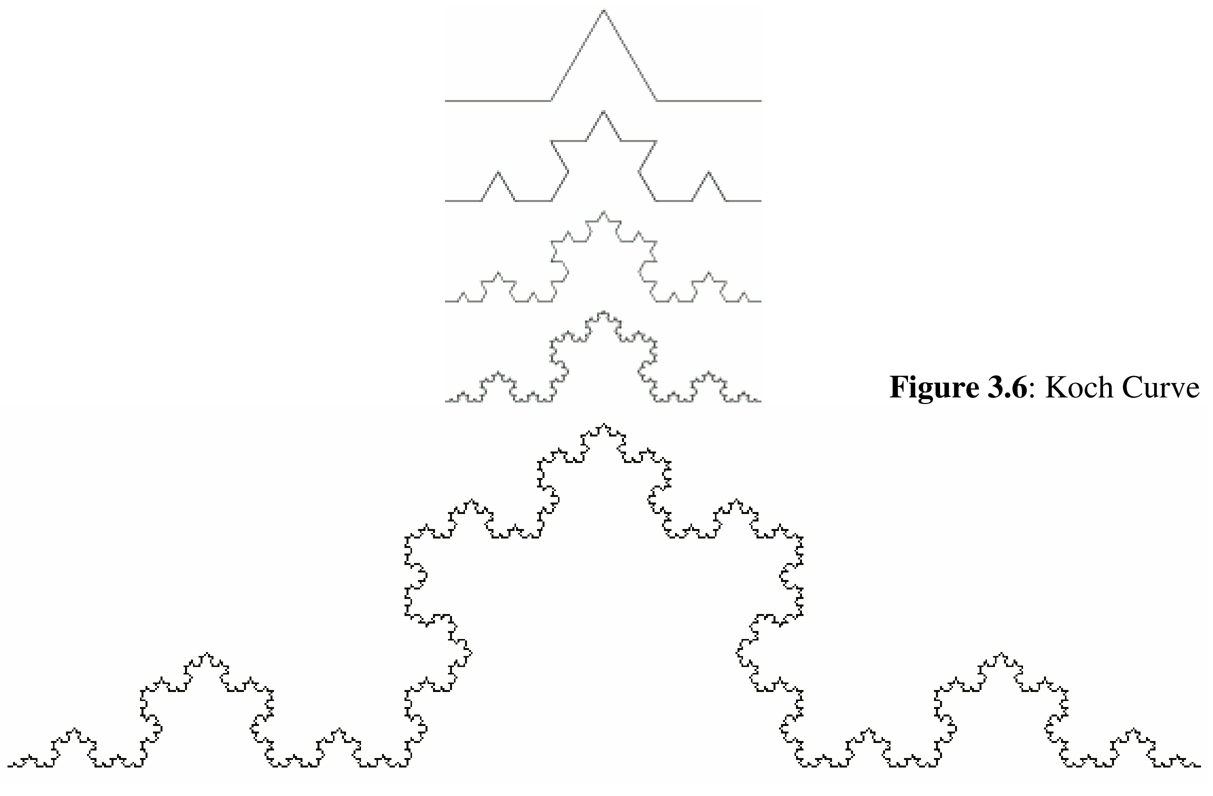


Figure 3.6: Koch Curve

3.4. Escape-time Fractals

As stated above, these fractals are defined by a recurrence relation, which is a system of non-linear processes in n-dimentional spaces. The processes predict the next (n+1) values in terms of the present (n) values. Then, the new values are applied to the equation.

The shape of the fractal is determined by the constant coefficients and the initial values.

If we iterate these processes (equations), it does not take long for the solution to display one of four behaviors:

- it converges to a single fixed point
- it produces a periodic cycle of repeating values
- it diverges to infinity
- it exhibits chaos.

The fixed point that the system approaches is called the attractor. As the function is iterated, the sequence of sets created converges to the attractor. Usually, there is an area around the point called the finite basin of attraction, which bounds the solutions. In other words, within the basin of attraction, there is a range of initial values that will eventually produce the same pattern. Points outside of the basin are not bounded by it, and tend to move ever further from the attractor. Because they escape the vicinity of the attractor, fractals that have this particular characteristic are called escape-time fractals. Repellers are points outside of the basin to which points may slowly escape. Strange attractors differ from regular attractors in that it is impossible to tell where they will be. Two points on the same attractor can be very close to one another at one point in time, and at another far apart. Thus, strange attractors behave chaotically.

The following fractals in this section can be defined for complex numbers, quaternions, and the hypercomplex numbers. It is possible to define them for even higher-dimensional spaces, but the benefit is minimal, considering that we cannot efficiently visualize nor imagine more that 3-dimensional objects.

Since the hypercomplex numbers and quaternions are merely extensions to complex numbers, the relations defined for these 4-dimensional spaces can also be used to generate complex fractals, using only 1 and i basis vectors, and zeroing j and k.

3.4.1. Mandelbrot and Julia Sets

This is undoubtedly the best known family of fractals which is defined extremely simply by the relation:

$$z_{n+1} = z_n^2 + c, \tag{3.2}$$

where c is a constant.

The behavior of the sequence depends upon the following data: the parameter c and the initial point z_0. Julia sets are defined by fixing c and letting z_0 vary in the field of complex numbers, while the Mandelbrot set is obtained by fixing $z_0 = 0$ and varying the parameter c.

If z_0 is taken far from 0, then the sequence tends very quickly towards infinity. There are values of z_0 for which the sequence remains bounded. For a given c, these values form the Julia set of the polynomial $z := z^2 + c$. The corresponding Julia set consists only of the boundary point of the "filled-in" set [12].

Obviously, the Julia set depends on the choice of the parameter c, but the surprise is that it depends heavily on it, so by varying c, an incredible variety of Julia sets can be obtained. Some look like a cloud, others like sparks, or a rabbit, and many have sea-horse tails…

Although the Mandelbrot set is just a single set, resembling a cardioid in the complex plane, there are two major classes of Julia sets. Some are in one piece – we say they are connected, while the others are just a cloud of points, which we call a Cantor set.

We can define a new set of values of c for which the Julia set is connected. This set is called the Mandelbrot set, since Benoît Mandelbrot was the first to produce pictures of it, using a computer. As mentioned above, the set can also be defined as the set of values of c for which, starting with $z_0 = 0$, the sequence z_n remains bounded. The equivalence of these two definitions was proved in 1919, independently, by Fatou and Julia.

The relationship between the Mandelbrot and Julia sets is the underlying principle for many other fractals.

The most frequent extension to the provided formula is replacing the exponent 2 with an arbitrary integer exponent, or even using a real, complex, or hypercomplex exponent. We have shown that the exponential function is defined in all of these cases. The resulting relation is:

$$z_{n+1} = z_n^p + c, \tag{3.3}$$

where p is a real, complex, or hypercomplex number. The relation behaves identically to the original relation for $p = 2 + 0i + 0j + 0k$.

This extension of the complex Mandelbrot set is characteristic of generating multiple fractal-shaped circles that are connected in 0. The circles are surrounded by areas called buds. For positive integer values of p, there are exactly $p-1$ circles. We can say that the same formula applies to a positive real exponent because it really looks like there are 4 and a half of circles for $p = 5.5$ (Figure 3.8). Setting other basis vectors i, j, and k to a non-zero value leads to other less predictable effects.

A similar behavior can be observed in the extended Julia sets. For connected fractals, the number of branches again connected in 0 is given by a positive integer or a positive real exponent, and is therefore equal to p.

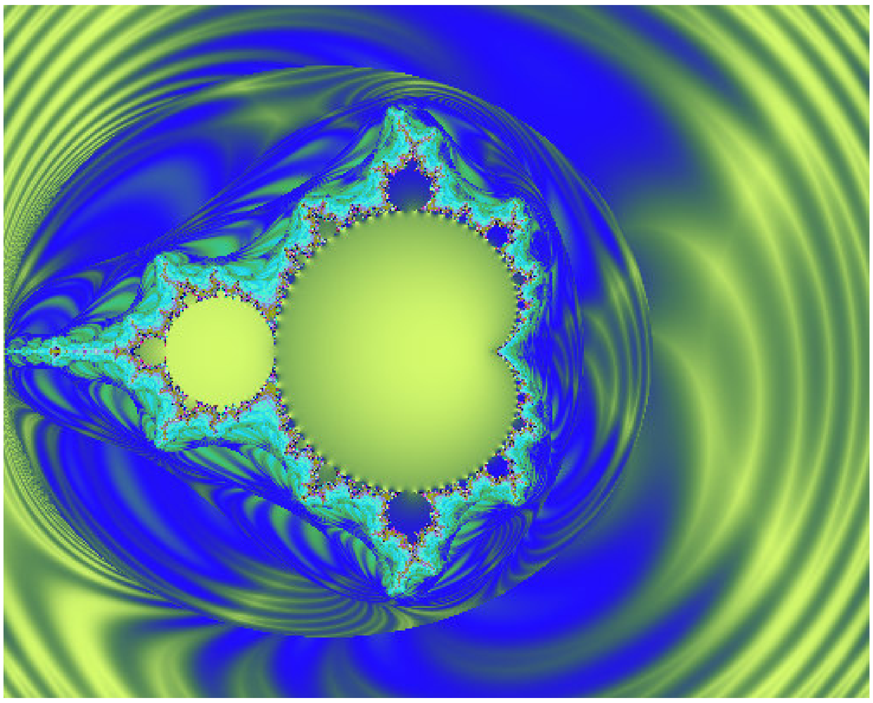


Figure 3.7: Complex Mandelbrot Set: $z_{n+1} = z_n^2 + z_0$

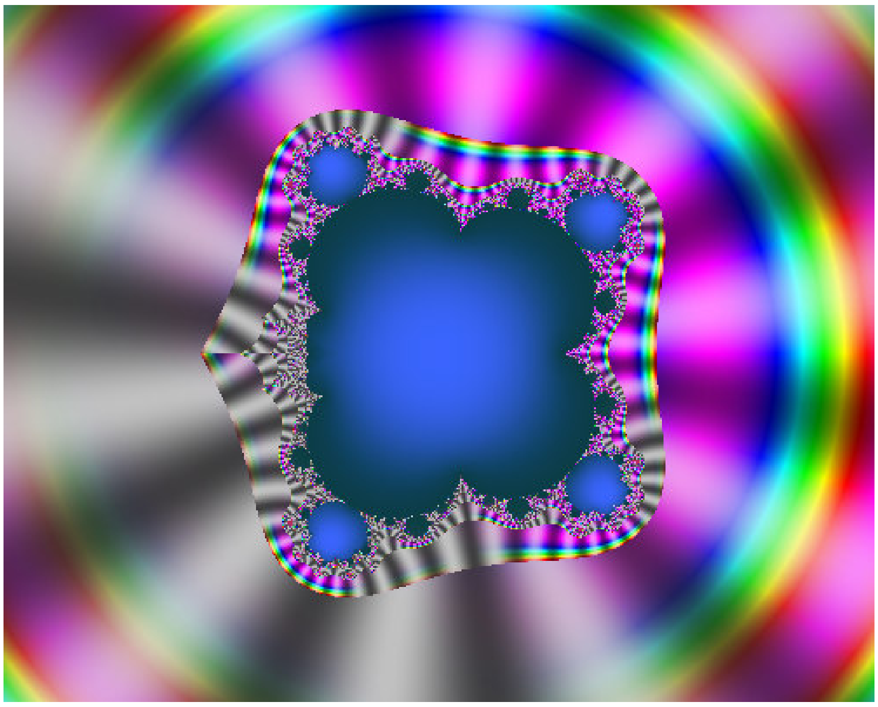


Figure 3.8: Complex Mandelbrot Set: $z_{n+1} = z_n^{5.5} + z_0$

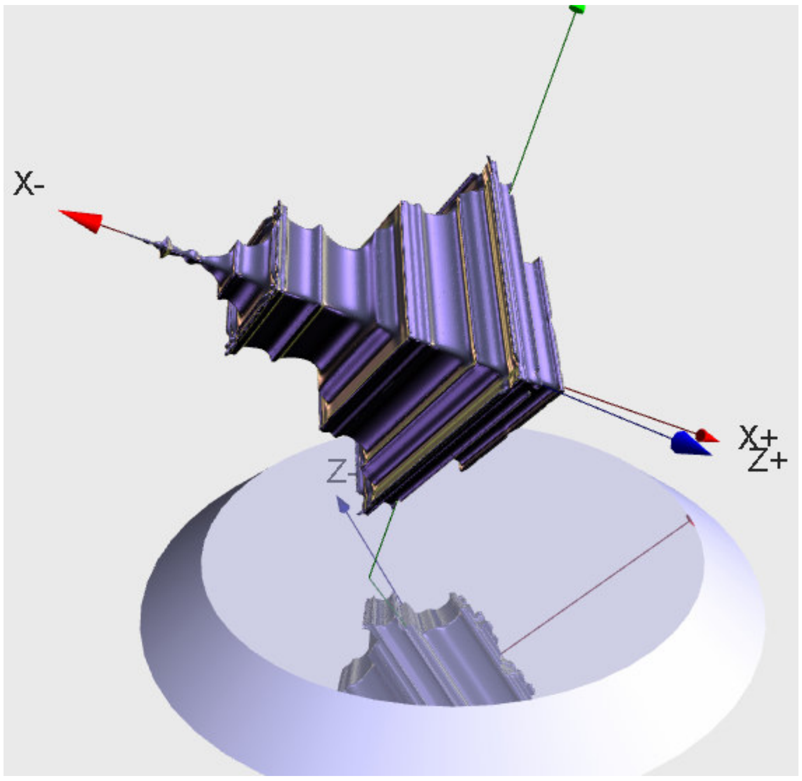


Figure 3.9: 3D Subset of Hypercomplex (Davenport) Mandelbrot Set (max. 8 iterations):

$$z_{n+1} = z_n^2 + z_0$$

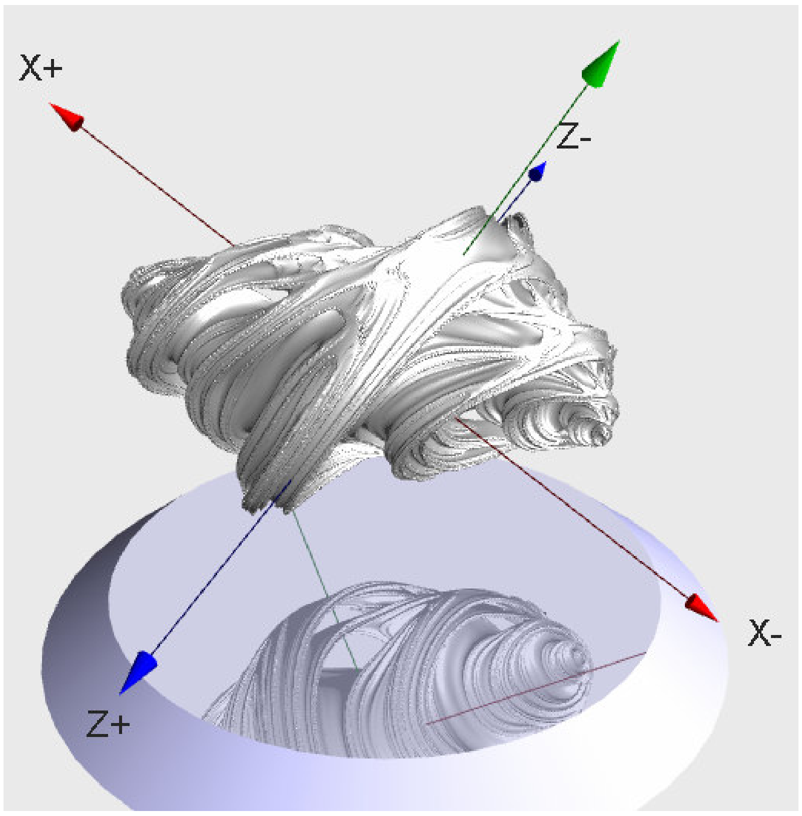


Figure 3.10: 3D Subset of Quaternion Julia Set (max. 12 iterations):

$$z_{n+1} = z_n^2 - 0.08 - 0.8j - 0.03k$$

3.4.2. Sierpiński Triangle

This fractal is just one of many fractals originally developed by W. Sierpiński. However, most of his fractals follow the same pattern which is making holes into a regular geometric object using its own shape. This type of a fractal is typical of the Iterated Function Systems, but it is possible to write a recurrence relation and make it an escape-time fractal.

In order to maintain the desired shape of the fractal, which is either a triangle, a pyramid, or a 4D pyramid, we cannot use the magnitude of a hypercomplex number to determine if the sequence converges or diverges. Hypercomplex and lower-dimensional spaces use the Euclidean norm, which would generate a spherical appearance. For this purpose, I created a special norm for any hypercomplex number $z = a + bi + cj + dk$:

$$\|z\| = \begin{cases} a+b+c+d & a \geq 0 \wedge b \geq 0 \wedge c \geq 0 \wedge d \geq 0 \\ +\infty & \text{else} \end{cases} \tag{3.4}$$

The norm uses the Manhattan norm only for numbers that have all basis vectors positive, and otherwise returns infinity. This means that all numbers z for which $\|z\| = +\infty$ are forced to diverge in the first iteration.

The relation for a complex number $z = a + ib$:

$$z_{n+1} = \begin{cases} 2z_n - i & b > \tfrac{1}{2} \\ 2z_n - 1 & a > \tfrac{1}{2} \\ 2z_n & \text{else} \end{cases} \tag{3.5}$$

The relation for a hypercomplex number $z = a + bi + cj + dk$:

$$z_{n+1} = \begin{cases} 2z_n - k & d > \tfrac{1}{2} \\ 2z_n - j & c > \tfrac{1}{2} \\ 2z_n - i & b > \tfrac{1}{2} \\ 2z_n - 1 & a > \tfrac{1}{2} \\ 2z_n & \text{else} \end{cases} \tag{3.6}$$

It is also important to use 1 as the bailout value to determine divergence, otherwise the pyramids have improper sizes.

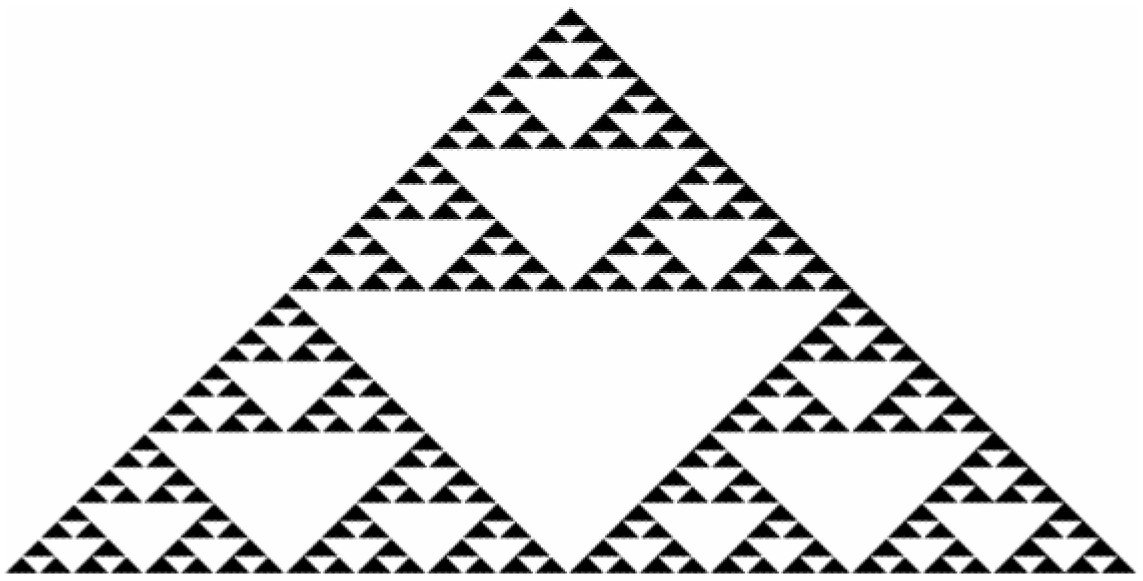


Figure 3.11: Sierpiński Triangle (6th iteration)

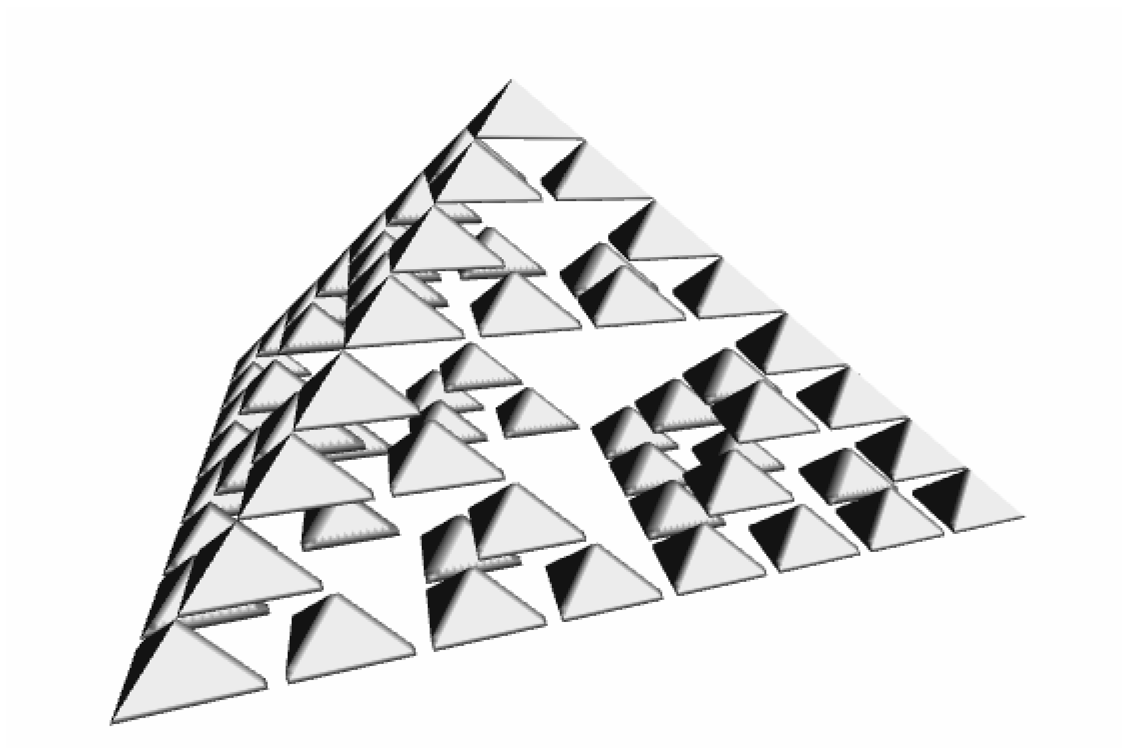


Figure 3.12: Approximation of Sierpiński Pyramid (4th iteration)

3.4.3. Newton Fractal

The Newton fractal is a complex boundary set which is characterized by applying Newton's method to a polynomial $p(z)$. It divides the complex plane into regions, each of which is associated with a root of the polynomial, $k = 1, 2, ..., \deg p$. It is significant for numerical analysis because it shows that Newton's method can be very sensitive to the choice of the start point.

Each point of the complex plane is associated with one of the $\deg p$ roots of the polynomial in the following way: the point is used as the starting value z_0 for Newton's iteration:

$$z_{n+1} = z_n - \frac{p(z)}{p'(z)} \tag{3.7}$$

generating a sequence of points $z_1, z_2, ...$ If the sequence converges to the root, then z_0 is an element of the region.

The classical example of a polynomial used is $p(z) = z^3 - 1$ which solves the equation $z^3 = 1$ having one real and two complex roots. Therefore:

$$z_{n+1} = z_n - \frac{z_n^3 - 1}{3z_n^2}. \tag{3.8}$$

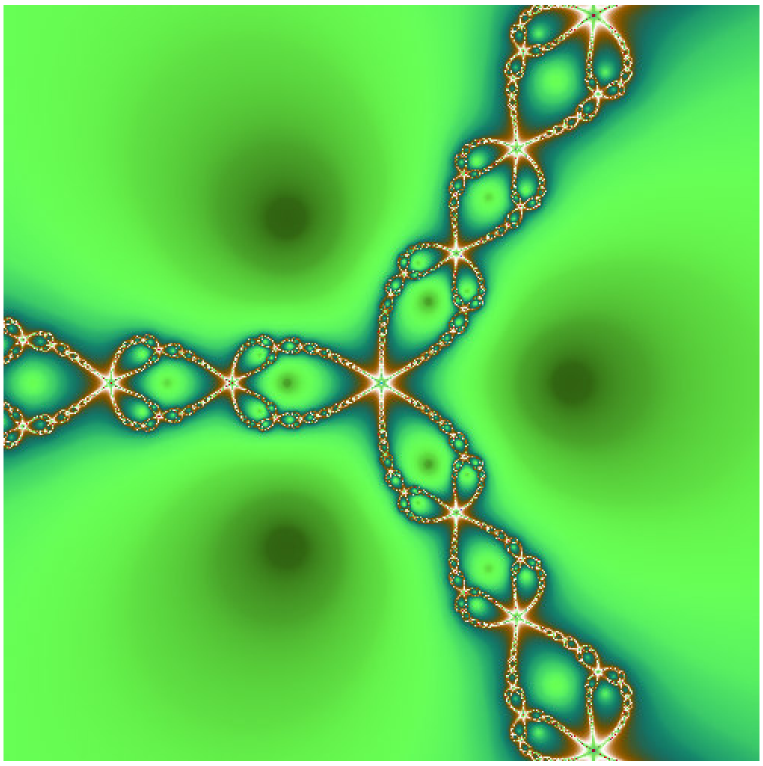


Figure 3.13: Complex Newton Fractal: $z_{n+1} = z_n - \frac{z_n^3 - 1}{3z_n^2}$

3.4.4. Other Escape-time Fractals

- Breeder is similar to the Mandelbrot's main body, but it is inverted on the i-axis:
  $z_{n+1} = z_n(1 - z_n) + c,$
  where c is either equal to z_0 (Mandelbrot) or a hypercomplex parameter (Julia).

- Phoenix is one of the most beautiful complex fractals (Figure 3.14). Using proper parameters ($p_1 = 0.567, p_2 = -0.5$), it can resemble a butterfly:
  $z_{n+1} = z_n^2 + p_1 + p_2 y_n$
  $y_{n+1} = z_n,$
  where p_1 and p_2 are hypercomplex numbers, and $y_0 = z_0$.

- Spider is another fractal designed for complex numbers (Figure 3.15). Obviously, it resembles a spider:
  $z_{n+1} = z_n^2 + y_n$
  $y_{n+1} = \frac{y_n}{2} + z_{n+1},$
  where $y_0 = z_0$.

- ManOWar is a peculiar name of a fractal with characteristics of Mandelbrot-Julia sets (Figure 3.16):
  $z_{n+1} = z_n^2 + y_n + c$
  $y_{n+1} = z_n,$
  where $y_0 = z_0$.

- Lambda fractal was originally developed by P. F. Verhulst in 1845:
  $z_{n+1} = cz_n(1 - z_n),$
  where c is either equal to 0.5 (Mandelbrot) or a hypercomplex parameter (Julia)

- There are many fractals created by M. Barnsley, but only a few can be easily redefined for hypercomplex spaces. This is one of them (Figure 3.17):
  $$z_{n+1} = \begin{cases} (z_n - 1)c & \operatorname{Re} z_n \geq 0 \\ (z_n + 1)c & \text{else} \end{cases},$$
  where c is either equal to z_0 (Mandelbrot) or a hypercomplex parameter (Julia).

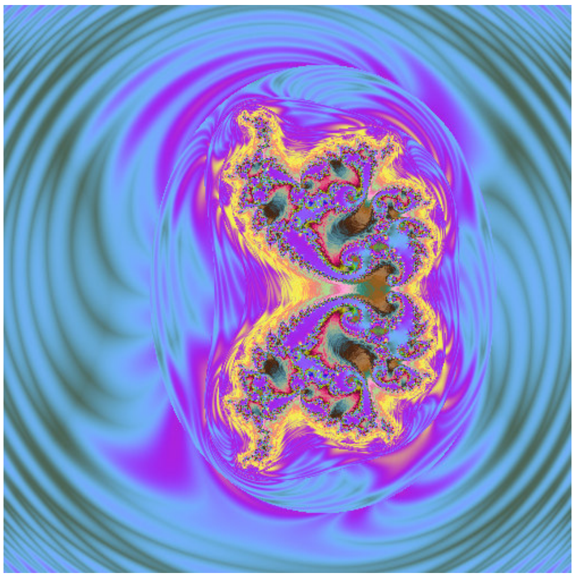


Figure 3.14: Complex Phoenix Fractal: $p_1 = 0.567, p_2 = -0.5$

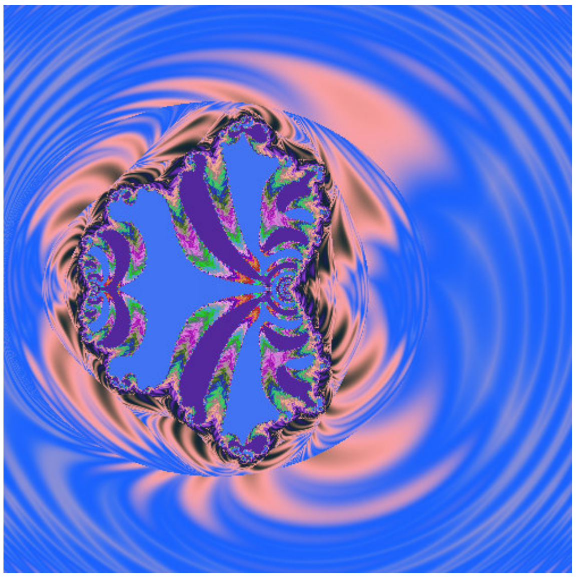


Figure 3.15: Complex Spider Fractal

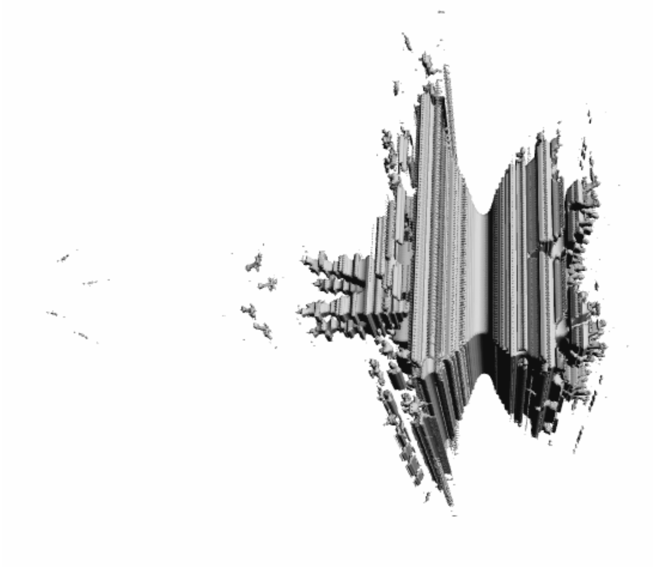


Figure 3.16: 3D Subset of Hypercomplex (Davenport) ManOWar Fractal (max. 20 iterations): $c = 0$

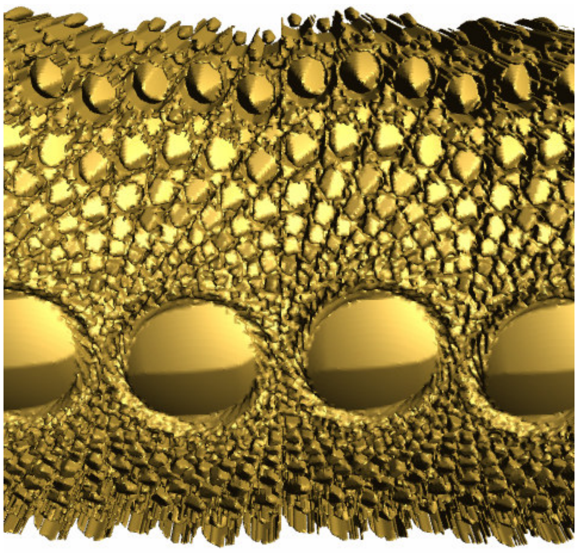


Figure 3.17: 3D Subset of Quaternion Barnsley-Julia Fractal (max. 80 iterations): $c = 1 + 0.1i$

3.5. Random Fractals

Although random fractals are neither the biggest nor the most popular group of fractals, it is a group with many practical applications.

Natural objects do not contain identical scaled down copies within themselves, and so sometimes are not considered fractals at all. However, natural objects can often be classified as random fractals, that is, each smaller part is statistically similar to the whole. Random fractals can be generated by modifying the iteration process to include a probabilistic element.

If we modify the Koch curve algorithm (Section 3.3.1) in such a way that the triangle in the middle points randomly (for example, determined by a coin toss) to either side of the original line, the final fractal shape looks very irregular compared to the exact Koch curve, but is closer to the shape of natural objects such as coastlines.

3.5.1. Perlin Noise

Perlin noise is one of many functions to generate landscapes, clouds, fire, smoke, and many other approximations of natural objects.

Perlin noise is widely used in computer graphics, and it resulted from the work of Ken Perlin, who used it to generate textures.

Many people use random number generators in their programs to create unpredictability, make the motion and behavior of objects appear more natural, or generate textures. Random number generators (RNG) themselves have certain properties that make them useless for generating fractal structures. An ideal uniform RNG would generate a completely new random value in a specific range of values each time called, which would result in completely random structures.

Each landscape contains wide variations in height (mountains), medium variations (hills), small variations (boulders), tiny variations (stones), and so on. The same pattern of wide and small variations can be observed almost anywhere. The Perlin noise function recreates this by simply adding up noise functions at a range of different scales.

To create a Perlin noise function, two things are needed, that is, a noise function and an interpolation function.

A noise function is based on a random number generator with the following properties. In addition to a typical RNG, it requires one numeric parameter. It has to return the same random number each time called with the same parameter, but a different number each time called with a different parameter. This way, we can generate a set of random numbers corresponding to the input parameters. But, we do not have to, because the RNG returns a corresponding random value any time called. That is why it does not require much of the computer memory to execute.

The generated values are only discrete random values, and we need to define a function. One way of doing so is to interpolate the values between these random values. The original Ken Perlin's algorithm uses a simple linear interpolation, which in most cases seems to be enough. Of course, we can use better interpolation functions such as cubic splines, a windowed sinc function, or the Gaussian function, but it certainly is not recommended to use point sampling, although it may generate interesting unnatural shapes.

The noise function is now connected, and we can define parameters like amplitude or frequency. The amplitude represents the difference between the minimum and maximum values the function can have. The period, which remains constant for the function, is the distance between two neighboring random values. The frequency is defined as 1/period.

If we take many of these interpolated functions with various frequencies and amplitudes and add them all together, another noise function is created. This is the Perlin noise function.

The chosen frequencies and amplitudes are not exactly arbitrary. The most common advice is to use twice the frequency and half the amplitude for each successive noise function added. Sometimes, this is not convenient, for example, to create smooth rolling hills, we could use a Perlin noise function with large amplitudes for the low frequencies, and very small amplitudes for the higher frequencies. We could also make a flat, but very rocky plane choosing low amplitudes for low frequencies.

To make it simpler, a single number is used to specify the amplitude of each frequency. This value is known as the Persistence. The term was originally coined by Mandelbrot. He defined noise with a lot of high frequencies as having a low persistence.

Each successive noise function added is known as an octave. The reason for this is that each noise function is twice the frequency of the previous one. In music, octaves also have this property.

This one-dimensional function can be easily extended to a higher-dimensional function by using higher-dimensional interpolations and adding the interpolated values together.

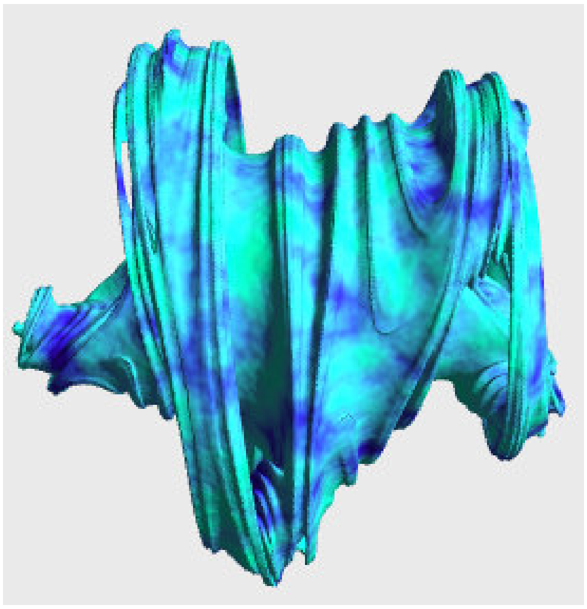


Figure 3.18: Perlin Noise Texture on top of Quaternion Julia Fractal (max. 5 iterations): $z_{n+1} = z_n^5 + 0.6 + 0.8i$

3.6. Fractal Dimensions

The actual definition of a fractal has not been furnished yet, although there have been a few attempts to do so (see the beginning of this chapter). The most frequently used definition is the one given by Mandelbrot. It states that a fractal is an object whose Hausdorff-Besicovitch dimension is greater than its topological dimension. The Hausdorff-Besicovitch dimension is often called the Hausdorff dimension.

3.6.1. Hausdorff Dimension

In mathematics, the Hausdorff dimension is a non-negative extended real number in the closed infinite interval $\langle 0,+\infty\rangle$ which is associated with any metric space. It was introduced in 1918 by the mathematician Felix Hausdorff [9]. Many of the techniques used to compute the Hausdorff dimension for highly irregular sets were created by Abram Samoilovitch Besicovitch [9]. It is also less frequently called the capacity dimension or fractal dimension.

Intuitively, the dimension of a set, for example, a subset of Euclidean space, is the number of independent parameters needed to describe a point in the set. The topological dimension is one mathematical approach that models this idea. For instance, a point in the plane is described by two independent parameters (the Cartesian coordinates of the point), so we can say that the plane is two-dimensional. As can be expected, the topological dimension is always a natural number.

However, the topological dimension behaves in quite unexpected ways on certain highly irregular sets such as fractals. The Hausdorff dimension gives another way how to define dimension, which takes the metric into account. It extends the concept of the topological dimension by relating it to other properties of the space such as area or volume.

Let us take a closer, yet still very brief, look at the theory behind the Hausdorff dimension. The theory of the Hausdorff dimension is based upon the theory of measure developed earlier by Henri Lebesgue and Constantin Caratheodory.

Suppose (X,d) is a metric space. In order to deal with the problems of this approach, Hausdorff defined an entire family of measures on subsets of X, one for each possible dimension $s \in \langle 0,+\infty)$.

For example, if $X = \mathbb{R}^3$, this construction assigns an s-dimensional measure H^s to all subsets of $\mathbb{R}^3$. For $s = 2$, it is expected that:

- for the unit segment along the x-axis: $H^2(\langle 0,1\rangle \times \{0\} \times \{0\}) = 0$
- for the unit square on the xy-plane: $0 < H^2(\langle 0,1\rangle \times \langle 0,1\rangle \times \{0\}) < +\infty$
- for the unit cube: $H^2(\langle 0,1\rangle \times \langle 0,1\rangle \times \langle 0,1\rangle) = +\infty$.

The above example suggests that we can define a set A to have Hausdorff dimension s if its s-dimensional Hausdorff measure is positive and finite. This is not exactly true. The Hausdorff dimension of A is the threshold value s where below s the s-dimensional Hausdorff measure is ∞ and above s it is 0. It is possible for the s-dimensional Hausdorff measure of an s-dimensional set to be 0 or ∞. For instance, $\mathbb{R}$ has dimension 1 and its 1-dimensional Hausdorff measure is infinite.

Let C be the class of all subsets of X. For each positive real number s, let p_s be the function $A \to \operatorname{diam}(A)^s$ on C. The Hausdorff outer measure of dimension s, denoted H^s is the outer measure corresponding to the function p_s on C. Thus, for any subset E of X:

$$H_\delta^s(E) = \inf\{\sum_{i=1}^{\infty} \operatorname{diam}(A_i)^s\} \tag{3.9}$$

where the infimum is taken over sequences $\{A_i\}_i$ which cover E by sets, each with diameter $\leq \delta$. Then:

$$H^s(E) = \lim_{\delta \to 0} H_\delta^s(E). \tag{3.10}$$

It is possible to create a formula to calculate the Hausdorff dimension for IFS fractals. Let us revise what an IFS fractal is:

Iterated Function Systems are a finite set of contraction maps w_i for $i = 1, 2, ..., n$, each with a contractivity factor $r_i < 1$, which map a compact metric space onto itself.

Then, there is a unique non-empty compact set A such that:

$$A = \bigcup_{i=1}^{n} w_i(A). \tag{3.11}$$

In certain cases, to determine the dimension of the set A, we need the set to be open on the sequence of contractions w_i. For this purpose, we can use the following set V:

$$\bigcup_{i=1}^{n} w_i(V) \subset V \tag{3.12}$$

where the sets in the union on the left are disjoint.

Then the unique fixed point of w is a set whose Hausdorff dimension is s where s is the unique solution of:

$$\sum_{i=1}^{n} r_i^s = 1. \tag{3.13}$$

For r independent of i, we can write directly $nr^s = 1$.

One of the simplest examples to derive the formula for calculating the Hausdorff dimension is this one: Let us divide a unit segment into n equally long pieces, each of which is $r := \frac{1}{n}$ in length. If we were to divide a square into n equal sub-squares, each having r long sides, the length would be defined as $r := \frac{1}{n^{\frac{1}{2}}}$. Obviously, the formula for a cube would be $r := \frac{1}{n^{\frac{1}{3}}}$. Therefore, we can write that generally $r := \frac{1}{n^{\frac{1}{s}}}$ where s is the dimension of an object. If we solved for s by means of the equation $r = \frac{1}{n^{\frac{1}{s}}}$, the result would be:

$$D_H := \frac{\ln n}{\ln \frac{1}{r}} \tag{3.14}$$

where n is the factor of the change in length, and $\frac{1}{r}$ is the factor of the change in scale. D_H denotes the Hausdorff dimension, but basically $D_H := s$.

The calculation of the topological dimension by this formula is clear. Let us see what happens if we apply the formula to a fractal.

The Koch curve is a perfect example. In each iteration, the length is reduced to $\frac{1}{3}$ of the length in the previous iteration, and the number of self-similar segments is 4. Hence, we can substitute these numbers into the formula, which gives the Hausdorff dimension of the curve:

$$D_H = \frac{\ln 4}{\ln 3} \sim 1.2618... \tag{3.15}$$

Other examples of Hausdorff dimensions are:

- The Euclidean space $\mathbb{R}^n$ has $D_H = n$.
- The circle S^1 has $D_H = 1$.
- Countable sets have $D_H = 0$.
- The Cantor set (a zero-dimensional topological space) is a union of two copies of itself, each copy shrunk by a factor $\frac{1}{3}$. This implies the Hausdorff dimension to be $\frac{\ln 2}{\ln 3}$, which is approximately 0.63.
- The Sierpiński triangle is a union of three copies of itself, each copy shrunk by a factor of $\frac{1}{2}$, therefore $\frac{\ln 3}{\ln 2}$, which is approximately 1.58.
- The Mandelbrot set (the border) has the Hausdorff dimension equal to 2. A difficult proof was given by Mitsuhiro Shishikura [11].

Often, it is very difficult to calculate the Hausdorff dimension for fractals such as the Mandelbrot set. Therefore, various closely related definitions of fractional dimensions were invented. For example, the box-counting dimension (see 3.6.2.).

These definitions (topological dimension, Hausdorff dimension, box-counting dimension) give the same value for many shapes. However, they give different values for some highly irregular curves. These curves were originally called "monster curves" because they seemed so bizarre and non-intuitive at the time.

3.6.2. Box-counting and Cube-counting Dimension

The box-counting dimension, or the Minkowski-Bouligand dimension as it is often referred to, is another way of determining the fractal dimension of a set in a metric space.

In order to calculate the box-counting dimension of a set in a metric space, we need to create a grid of equally-sized boxes (squares), and count, how many boxes are required to cover the set. The box-counting dimension is calculated by determining how this number changes as we make the boxes smaller, and thus the grid finer.

Let us suppose that $n(r)$ is the number of boxes of side length r required to cover a set. Then the box-counting dimension is defined as:

$$D_B := \lim_{r\to 0}\frac{\ln n(r)}{\ln\frac{1}{r}}. \tag{3.16}$$

If the limit does not exist, we can still determine the upper box dimension $D_{\overline{B}}$ and the lower box dimension $D_{\underline{B}}$ which correspond to the upper limit and lower limit, respectively. In other words, the box-counting dimension is defined only if the upper and lower box dimensions are equal. The upper box dimension is sometimes called the entropy dimension, Kolmogorov dimension, Kolmogorov capacity, or upper Minkowski dimension, while the lower box dimension is also called the lower Minkowski dimension.

There are some interesting properties of these dimensions. Both box dimensions are finitely additive, that is, if $\{A_1, A_2, ..., A_n\}$ is a finite collection of sets then:

$$D_B(A_1 \cup A_2 \cup ... A_n) = \max\{D_B(A_1), D_B(A_2), ..., D_B(A_n)\}. \tag{3.17}$$

However, the dimensions are not countably additive, so the equality does not hold for an infinite sequence of sets. For example, the box dimension of a single point is 0, but the box dimension of the collection of rational numbers in the interval $\langle 0,1\rangle$ has dimension 1. The Hausdorff dimension, by comparison, is countably additive.

Both dimensions are strongly related to the more popular Hausdorff dimension. For many fractals all these dimensions are equal. For example, the Hausdorff dimension, lower box dimension, and upper box dimension of the Cantor set are all equal to $\frac{\ln 2}{\ln 3}$. However, the definitions are not equivalent.

The box dimensions and the Hausdorff dimension are related by the inequality:

$$D_H \le D_{\underline{B}} \le D_{\overline{B}}.$$

It is possible to extend the box-counting dimension to a higher dimension than 2D. The definition remains the same, except that this time we use cubes instead of boxes, and r is just the length of an edge of a cube. The value n is the number of cubes required to cover a set. Cubes are usually objects in a 3D space, but they can very well represent objects in a 4D space, or other spaces.

The main advantage of the box/cube-counting dimension is that it can be used in computer algorithms. Since it is technically impossible to exactly calculate the limit from the definition, we have to approximate. First, we have to choose just one value of r, and compute the number of boxes/cubes. This should be a simple enough task, because we just count the number of boxes/cubes which contain any part of the set measured. It may be important to note that a fractal is usually only the border of an object. The second thing to be done is to compute the number of boxes/cubes with a different value r. This is all we need to calculate a rough approximation of D_B of the object since D_B represents the direction of a line in a log-log graph, where $\ln\frac{1}{r}$ is on the x-axis and $\ln n(r)$ is on the y-axis. Knowing that it is a line, we can calculate the direction from the two computed values of $n(r)$ by calculating the direction of the vector between these two points in the graph:

$$D_B \cong \frac{\ln n(r_1) - \ln n(r_2)}{\ln\frac{1}{r_1} - \ln\frac{1}{r_2}}. \tag{3.18}$$

To calculate a more accurate approximation of D_B, we simply compute more than just two points in the graph, and then use a convenient function to extrapolate the line to get a more precise result.

Another advantage of this algorithm is that it can be used for pictures or other data inputs.

Several examples of approximations of the box/cube-counting dimension impossible to calculate by the Hausdorff dimension are:

- a coastline: 1.26
- the surface of a human brain: 2.76
- non-eroded rocks: 2.2 – 2.3
- the circumference of a cloud’s projection: 1.33

3.6.3. Correlation dimension

Yet another measure of fractal dimension is called the correlation dimension. It is designed specifically to measure dimensions of sets containing random points in space, or rather chaotically placed points in space.

The real feature of the correlation dimension is in determining the dimensions of fractal objects. This method has the advantage of being straightforward and quick to calculate, and it often produces the same results as with other methods.

Given a set of n points in an s-dimensional space:

$$\vec{x}(i) = (x_1(i), x_2(i), ..., x_s(i)) \qquad (3.19)$$

where $i = 1, 2, ..., n$, the correlation integral $C(\varepsilon)$ is calculated by:

$$C(\varepsilon) = \lim_{n \to \infty} \frac{g}{n^2} \qquad (3.20)$$

where g is the total number of pairs of points having a distance between them less than or equal to distance ε. As the number of points tends to infinity, and the distance between them tends to zero, the correlation integral, for small values of ε, has the form:

$$C(\varepsilon) \cong \varepsilon^{\nu}. \qquad (3.21)$$

If the number of points is sufficiently large, and evenly distributed, a plot of the correlation integral versus ε will show an estimate of ν. This idea can be better understood by realizing that for higher-dimensional objects there will be more ways for points to be close to each other, so the number of pairs close to each other will rise more rapidly for higher dimensions.

This method can be used to distinguish between completely random sets and chaotically random sets having a specific pattern.

The disadvantage is that it is rather difficult to use for numerically generated fractals in Euclidean spaces if we do not know the exact locations of the points of a fractal set, especially if we want to use computers to do calculations. Without knowing the location of points, it is statistically impossible to try and hit a single point in the set by neither random nor systematic testing of points in such a set, because this is exactly what we have to do. If we, indeed, do not hit a single point, it is as if we were calculating the dimension of an empty set.

4. Visualization of Escape-time Fractal Sets

Although IFS fractals can be used for fractal compression and random fractals can generate realistic landscapes, escape-time fractals produce the most beautiful images.

It is necessary to color fractals in order to visualize them, but no one is to say how to do that. This gives vast possibilities to do so. Visualization techniques have greatly evolved recently with the improvements in graphic cards which can now display a high number of colors, and also thanks to the imagination of several authors of programs generating fractals.

4.1. Complex Fractal Sets

A complex number can be represented by a point in a plane where the x and y coordinates are the real and imaginary parts of the number. For the fractals produced by the iteration of a complex recurrent relation, each point of the part of the plane displayed on the screen is iterated, and the value of the iterated point is monitored during the successive iterations. The positions of these points showing the successive values plot an orbit. For some initial complex values, that is, points of the plane, the orbit is enclosed in a limited region of the plane, or it can even follow a cyclic trajectory. In other cases, the orbit escapes towards the infinity after a few iterations, which means that the sequence of values diverges. The question is how to determine that the sequence diverges. For this purpose we usually test the sequence of magnitudes of the iterated complex point. Of course, there are other ways to determine the behavior of a complex point that lead to a different graphical output. An infinite number of iterations would be necessary for us to be certain of the behavior of only one point, which is something we cannot afford in an algorithm. However, for most of the points it is possible to detect divergence after only an acceptable number of iterations.

There are a few ways to show the result of a computation concerning a point. The first one is to use a third coordinate axis obtaining a fractal relief. The other more common method lies in the use of a scale of colors to display this value. The second coloring technique simply assigns a color according to the number of iterations needed to determine the convergence of a point. This is the most classic technique. It used to be very popular, because it does not require a high-color display. It is often enough to use only two colors, representing divergence and convergence. The boundary between these two colored areas is the actual approximation of a fractal set.

Such techniques do not generally allow for a smooth enough transition between adjacent colors. However, it is desirable for many fractal images to have clear limits between some areas of different colors, for example, an area of red shades and an area of green shades.

The desire for smoother transitions leads to algorithms that can make use of the full potential of true-color graphic cards that are able to display more than 16 million colors, which is the maximum the human eye can distinguish. Note that it is a standardized range of colors missing some colors that are not present in nature.

Linas Vepstas [10] has created an interesting algorithm giving a variation of colors by a modification of the above method. Basically, its principle is to compute not only the number of iterations needed to detect the divergence, but, moreover, how distant the point is from the limit between two successive iterations.

Other methods, known for a longer time, may also be useful. For example, we can use the angle between the orbit and the x-axis at the time when the divergence is detected. Or, we can use the potential calculated by the formula $\frac{\log|z|}{2^{iterations}}$.

Points can be colored not only by determining the divergence. For example, Paul Carlson has developed a few “orbit trap” algorithms and adapted them to the legendary program “Fractint” and another well known program called “Ultra Fractal”. The principle is to test if the orbit of the point falls into a circle, a square, etc., and to color the point according to the distance between the point and the centre of such a figure. The images obtained show fractal structures consisting of spheres, cylinders, or cones with a very characteristic "solid" appearance.

4.2. Hypercomplex Fractal Sets

Visualizing hypercomplex fractals is different from the methods developed for complex fractals. The main issue is that such multi-dimensional sets are impossible to visualize at once. Only one subset can be visualized at a time. Obviously, the subset can be 3-dimensional at most. This means that we have to create a mapping from the higher-dimensional space to a lower-dimensional space. One way of doing so is to hold one or more dimensions constant, depending on the type of algebra used.

If we wanted to display only a 2D subset of such a set, we could use the visualization techniques for complex fractals.

Visualization of 3D subsets is quite different, and we have a few options to do so. Note that we can generate multiple 3D fractal subsets varying one of the other dimensions (previously constant) according to a time rule, which creates a time-dependent animation. There are various techniques to make such visualization efficient, but let us focus on generating a single 3D subset.

Beside the following methods, there are many other 3D rendering methods, such as radiosity or photon maps, that have to deal with very similar problems generating approximations of fractal sets.

4.2.1. Ray Tracing

Ray Tracing is a rendering method based on global illumination. It traces rays of light from the eye back through the image plane (screen) into the scene. Then the rays are tested against all objects in the scene to determine if they intersect any objects. If the ray misses all objects, then that pixel is set to the background color. Such an object can be a fractal set. Ray tracing handles shadows, multiple specular reflections, and texture mapping in a very easy straight-forward way.

Note that ray tracing is a point sampling algorithm. Like all point sampling algorithms, this leads to the potential problem of aliasing, which is manifested in computer graphics by jagged edges or other visual artifacts.

As already mentioned, a ray is projected through each pixel and tested for intersection against all objects in the scene. If there is an intersection with an object, then several other rays are generated. Shadow rays are sent towards all light sources to determine if any objects occlude the intersection point. If the surface is reflective, then a reflected ray is generated. If the surface is not opaque, then a transmitted ray is generated. Each of the secondary rays is tested against all the objects in the scene.

The reflective and/or transmitted rays are continually generated until the ray leaves the scene without hitting any object or a maximal recursion level has been reached. This creates a ray tree.

A local illumination model is applied at each level and the resultant intensity is passed up through the tree, until the primary ray is reached. Thus we can modify the local illumination model at each tree node by:

$$I = I_{local} + K_R R + K_T T, \tag{4.1}$$

where R is the intensity of light from the reflected ray and T is the intensity of light from the transmitted ray. K_R and K_T are the reflection and transmission coefficients. For a very specular surface, such as plastic, we sometimes do not compute the local intensity I_{local}, but only use the reflected/transmitted intensity values.

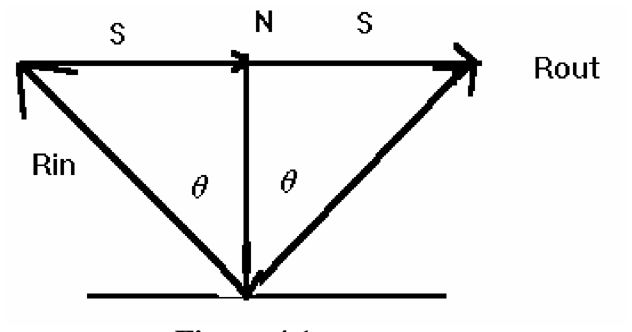


Figure 4.1

The reflected ray is calculated by projecting R_{in} onto N, which is $N\cos\theta$ so:

$$\begin{aligned} R_{out} &= N\cos\theta + S \\ R_{in} + S &= N\cos\theta \\ S &= N\cos\theta - R_{in} \end{aligned}$$

$$R_{out} = 2N\cos\theta - R_{in} = 2N(N\cdot R_{in}) - R_{in}.$$

The general idea behind ray-object intersections is to insert the mathematical equation for the ray into the equation for the object and determine if there is a real solution. If there is a real solution, then there is an intersection, and we must return the closest point of intersection and the normal N at the intersection point. For a shadow ray we must determine whether any ray-object intersection is closer than the ray-light intersection. For a ray tested against a boundary object, we just return a simple hit or no hit. For texture mapping we need the intersection point relative to some reference frame for the surface [13].

The problem with fractal sets is that we do not know the equation for the boundary set to be intersected. We merely know the recurrence relation which can be used to create many boundary sets (approximations of the real set). For complex fractals, we had to test each point in a bounded image plane, that is, $O(n^2)$ pixels. However, we have to test $O(n^3)$ points in order to isolate the boundary set in 3D, which is a major performance issue.

A second major issue involves the way lighting is computed in 3-dimensional computer graphics. To calculate smooth shading, the angle between a line from the light source to the surface of the object and the normal to the surface of the object is needed. Since fractals are infinitely "complex", they do not have normals. In 1989, John Hart generalized an equation from complex numbers that gives a distance estimate between a point and the surface of a Julia set in the quaternions. By using this formula, a vast majority of tests in the 3D space can be skipped and only the points near the quaternion Julia set are tested. In addition, this distance estimation can be used to calculate an approximate normal, so that lighting can be calculated. The existence of this formula is the main reason why there are so many computer programs that can generate only quaternion Julia sets. The formula:

$$d(z) = \frac{|z_n|}{2|z_n'|}\log|z_n| \tag{4.2}$$

requires to calculate additional sequence of points alongside z_n:

$$z'_{n+1} = 2z_n z'_n. \tag{4.3}$$

The first point in this sequence should always be $z'_0 = 1$.

The idea is simple. At any time during ray tracing we can query the distance estimator, and it will tell us the distance from our ray's origin to the closest point in the Julia set (in any direction - not necessarily in the direction of the ray). We can then safely take a step of the same distance along the ray direction, because we are guaranteed not to skip over any points in the set.

There are many other general improvements of Ray tracing to make it faster.

The main characteristics are:

- the most frequently used algorithm to visualize hypercomplex fractals
- relatively quality images – The algorithm calculates a color for each pixel. It is often desirable to calculate the colors of sub-pixels in order to implement full-screen anti-aliasing.
- unnatural appearance – Each pixel color is calculated with great precision considering every aspect of a scene based on a mathematical model. Since the generated fractals are only approximations, they do not exhibit this feature.
- slow generation – This is especially true if we want to generate more views from multiple observation angles.
- no available hardware support

4.2.2. Cube Marching

Cube Marching, or Marching cubes, is a computer graphics algorithm, published at the 1987 SIGGRAPH conference by Lorensen and Cline [16], for extracting a polygonal mesh of an isosurface from a 3D scalar field (sometimes called voxels).

The algorithm proceeds through the scalar field, taking eight points at a time (thus forming an imaginary cube), and determines the polygons that must be created (if any) to represent the part of the isosurface that intersects this cube.

This is done by creating an index to a pre-calculated array of 256 possible polygon configurations $(2^8 = 256)$ within the cube, and by treating each of the 8 scalar values as a bit in an 8-bit integer. If the scalar's value is higher than the iso-value, that is, it is inside the surface, then the appropriate bit is set to one, while if it is lower (outside), it is set to zero. There are no scalar values for fractal isosurfaces, but we can use a bitwise value indicating whether a point is in the set or not. The final value, after all 8 scalars are checked, is the actual index to the polygon configuration array.

Normally, each vertex of the generated polygons is placed at the appropriate position along the cube's edge by linearly interpolating the two scalar values connected by that edge. However, this is not possible with fractal sets because we do not know the exact mathematical formula, such as with quadrics, to directly generate a fractal set. To go around this problem, we can introduce the successive approximation algorithm.

Successive approximation has the advantage of not having to know much about a set. We still only need to be able to test the divergence of a point in a fractal set. Since we know that one of the ends of the edge is inside the shape and one is outside, we can check an additional point between these two. We should now have a region half the size in which we know the surface resides. Repeating this process recursively only a few more times gives a very accurate result, but introduces additional slowdown. It is better to use this algorithm rather than decrease the sizes of cubes, making more polygons because the algorithm of Marching cubes produces high numbers of polygons.

The pre-calculated array of 256 cube configurations can be obtained by reflections and symmetrical rotations of 15 unique cases (Figure 4.2).

Calculating normals as gradients of the scalar field at each grid point is not possible for the same reason as described above. We can however calculate normals of the generated polygons (flat shading), and compute the average of these normals at each of the vertices of the polygons (smooth shading), or use any other technique of generating normals from connected polygons. Normals are essential for shading the resulting mesh with some illumination model.

The applications of this algorithm are mainly medical visualizations such as CT and MRI scan data images, and special effects, or 3D modeling.

Note that the algorithm was patented for 20 years until 2005.

The main characteristics are:

- scale independence – Polygons unlike pixels can be exactly rendered at any scale. Pixels have to be interpolated.
- fixed subdivision – Every cube has the same size, so even for flat surfaces too many polygons are generated. This problem can be overcome by replacing the polygons having approximately equal normals by larger polygons. This is not efficient if we require only triangles which are the best polygons to render. Moreover, fractals are too complex to even bother with this issue.
- lesser precision – If the polygons are larger than pixels, certain artifacts can exhibit themselves in the form of an edgy surface.
- easy processing – Created polygons are easy to transform and render.
- hardware support – Almost all graphic cards have 3D acceleration to render polygons quickly, especially triangles.

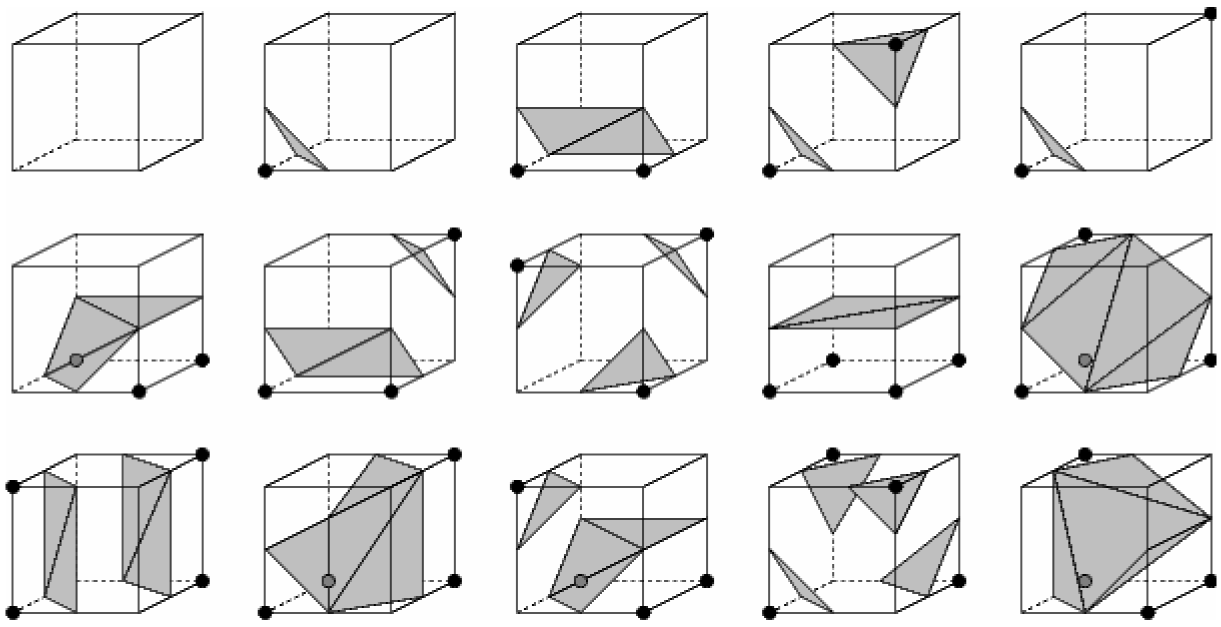


Figure 4.2: 15 Unique Cases (Cube Marching)

5. Application Environment

Several of my programs were used to generate pictures and calculate dimensions. Complex fractals were rendered using the program "Flashlight" even though my other program "QuadFractal" could have been used for this task. "Flashlight" can generate many more variations of complex fractals, but it lacks the support of hypercomplex fractals. Let us focus on "QuadFractal".

5.1. QuadFractal

"QuadFractal" is a multi-purpose open-source application under GNU GPL for generating fractals in Euclidean spaces. It was developed in C++. The GUI was created only for a Win32 environment.

The main features are:

- it generates and renders 3D models from hypercomplex (quaternion and Davenport) fractals
- it implements a 2D slice view to visualize 2D subsets of hypercomplex fractals
- it contains a dimension calculator
- it uses OpenGL with extensions
- it handles several visualization modes including reflections
- it enables the use of GLSL shaders
- it implements intuitive rotation using quaternion arcball
- it contains procedural texture generator
- it includes a solid material editor
- it provides an internal file format (INV3)
- it imports from 3ds and Zipper models
- it exports to X3D, VRML97, and C/C++ header

Updates of this program are available at [14].

The entire program is multi-threaded, but the use of mutex objects is avoided wherever possible to eliminate unnecessary waiting states. One way to do this is to use test and set functions, or make local copies of variables/classes. Of course, stability of the program is not compromised in any way. Threads in the application are designed to leave the GUI thread unoccupied and to perform simultaneous calculations and rendering. Currently, they are not "optimized" for multi-core CPUs, which requires dividing of individual calculations into threads. Usually, this can be implemented easily, and it is likely to be added in the future versions.

5.1.1. Generation of 3D Models

Visualization of 3D models is implemented using OpenGL. In order to use OpenGL, points, lines, or polygons have to be generated. The best way to visualize 3D models is to use triangles. Cube Marching was used to generate an isosurface consisting of triangles. Note that the program can visualize the edges of triangles (wire-frame), or single points (vertices of triangles) as well.

In order to use Cube Marching, we have to be able to determine if a point in a 3D space is a part of a fractal set. Since hypercomplex fractals are 4-dimensional sets, a simple method was devised to select a 3D subset of a hypercomplex space. One of the four dimensions is held constant, for instance, the one corresponding to the basis vector *k*:

```
void Permute4(Quad &ret, float x, float y, float z, float c) {
      // a point (x,y,z) is mapped into a hypercomplex space
      // using the constant c
      ret.r = x; ret.i = y; ret.j = z; ret.k = c;
}
```

There are three more similar functions. All of them return the result in a variable "ret" instead of using the return statement. The reason for this is performance because by using the constructor of "Quad", one more copy operation would be required when the function returns. The advantage of using constructors is a neat code:

```
void Permute4(float x, float y, float z, float c) {
      return Quad(x, y, z, c);
}
```

Since all 4 functions have the same arguments, a single pointer is used to reference the one chosen in a GUI dialog. Instead of using a complicated statement to call the function like this one:

```
      switch(permutation) {
            case 0:
                  Permute1(f4thParam, x, y, z);
                  break;
            case 1:
                  Permute2(x, f4thParam, y, z);
                  break;
            case 2:
                  Permute3(x, y, f4thParam, z);
                  break;
            case 3:
                  Permute4(x, y, z, f4thParam);
                  break;
            default: QFASSERT(false);     // invalid permutation
      }
```

a single line of code is used:

```
setpoint(cube[0], x, y, z, f4thParam);
```

providing we have initiated the function in the setup:

```
      switch(permutation) {
            case 0:
                  setpoint = Permute1;
                  break;
            case 1:
                  setpoint = Permute2;
                  break;
            case 2:
                  setpoint = Permute3;
```

```
				break;
			case 3:
				setpoint = Permute4;
				break;
			default: QFASSERT(false);
	}
```

Similar techniques are used throughout the program. Note that these 4 functions are static functions. For member functions of a class, a different type of call has to be made:

```
(this->*TestPoint)(q);
```

where TestPoint is a pointer to one of the functions testing the divergence of a point in a hypercomplex space, such as:

```
unsigned int Fractal::TestPointMandelbrot(const Quad &z0) {
	Quad z = z0, t;
	int numIter = 0;
	bool bout = true;
	do {	// z = z^2 + z0
		square(t, z);
		add(z, t, z0);
	} while(++numIter < nMaxIter && (bout = norm2(z) < fBailout2));
	return bout ? INF : numIter;
}
```

The macro “INF” represents infinity by a large integer. The function `norm2(z)` calculates the squared Euclidean norm of a hypercomplex number z. These test point functions are defined only once for both implemented algebras (quaternion and Davenport’s), which means that analytic functions are also implemented using pointers if they differ in definition. In the case of the Mandelbrot set, `square(z)` function can be calculated faster than `multiply(z, z)`, and it is much faster than `pow(z, 2)`. The implementation through pointers prevents the inline function expansion, so the use of the “ret” variable is beneficial.

The iterated relations could be implemented more conveniently using C++ operators sacrificing a few clock cycles:

```
z = z * z + z0;
```

Obviously, the iteration loop is the innermost loop in algorithms, which means that most CPU cycles are burnt in this loop. Unfortunately, personal computers are not designed for accelerated calculations of analytic functions. They are designed to handle simple vector operations, such as multiplying two 2 vectors, each element individually. So the use of extended instruction sets is pointless.

However, there are other ways to accelerate calculations. Since the OpenGL’s internal format of numbers is single-precision floating-point, the program uses this format as well. Moreover, it enables the lower precision floating-point calculations that are accurate enough for single-precision numbers.

The main recursive procedure that steps through a 3D grid follows:

```
void Fractal::SubDivide(int level, float x, float y, float z, float size) {
      if(bAbort) return;      // a simple stop flag
      if(level <= nMaxLevel) {
            if(level > nMinLevel) {
                  if(CanSkip(x, y, z, size)) return;
            }
            level++;
            size *= 0.5f;
            SubDivide(level, x + size, y, z, size);
            SubDivide(level, x + size, y + size, z, size);
            SubDivide(level, x, y, z, size);
            SubDivide(level, x, y + size, z, size);
            SubDivide(level, x, y, z + size, size);
            SubDivide(level, x, y + size, z + size, size);
            SubDivide(level, x + size, y, z + size, size);
            SubDivide(level, x + size, y + size, z + size, size);
            if(level == 4) { // 100 / (8 ^ (4 - 2)) = 1.5625f
                  PostMessage(whandle3, WM_USER_PROGRESS,
                              round(++nProgress * 1.5625f), NULL);
            }
      } else {
            Quad cube[8];
            unsigned int vals[8];
            TestCubePoints(cube, vals, x, y, z, size);
            MarchCube(cube, vals);
      }
}
```

The variable “nMaxLevel” defines the precision of a grid, thereby the number of generated triangles and their size. The argument “size” defines the initial size of the entire grid, a cube with the center at (x, y, z).

The function divides a cube into 8 half-sized cubes, and, after the maximal recursion level is reached, eight vertices of a cube are tested in `TestCubePoints()` and triangles are generated in `MarchCube()`.

In the fourth recursion level, a message is posted into the GUI thread stating the progress of a calculation in percents. This can be done because we know how many cubes are tested in the fourth level.

Note that `PostMessage()` is a safe way to deliver the message. We could use `SendMessage()`, which would work fine until we tried waiting for the thread from the GUI thread when it exits. This could produce a deadlock.

Notice that the function `round()` is used to round a float number to the closest integer. This function is not implemented by any standard C/C++. There are many custom functions that can be used. The actual function used was taken from [15]:

```
inline int round(double x) {
      const int p = 52;
      const double c_p1 = static_cast<double>(1L << (p / 2));
      const double c_p2 = static_cast<double>(1L << (p - p / 2));
      const double c_mul = c_p1 * c_p2;
      const double cs = 1.5 * c_mul;
      x += cs;
      const int a = *(reinterpret_cast<const int *>(&x));
      return a;
}
```

A clever compiler would compile this code as only one addition. The recommended way of rounding on Intel compatible processors is as follows:

```
inline int round(double x) {
      int a;
      __asm {
            fld x
            fistp a
      }
      return a;
}
```

which is only slightly slower because it uses FPU unlike the previous method.

The disadvantage of these two methods is that they both round 0.5, 1.5, … to 0, 1, … instead of 1, 2, …, but, practically, there are no consequences of this behavior in the application.

The big advantage is that the described functions are many times faster than the C/C++ truncation implemented through casting or constructors.

Let us return to the procedure `SubDivide()`. After "nMinLevel" is reached, a cube is tested whether it can be skipped or not, that is, whether it contains any parts of a set or not. The function `CanSkip()` is rather straightforward:

```
bool Fractal::CanSkip(float x, float y, float z, float size) {
      Quad q;
      setpoint(q, x, y, z, f4thParam);
      if((this->*TestPoint)(q) < nIterThr) {
        setpoint(q, x + size, y, z, f4thParam);
        if((this->*TestPoint)(q) < nIterThr) {
          setpoint(q, x + size, y, z + size, f4thParam);
          if((this->*TestPoint)(q) < nIterThr) {
            setpoint(q, x, y, z + size, f4thParam);
            if((this->*TestPoint)(q) < nIterThr) {
              setpoint(q, x, y + size, z, f4thParam);
              if((this->*TestPoint)(q) < nIterThr) {
                setpoint(q, x + size, y + size, z, f4thParam);
                if((this->*TestPoint)(q) < nIterThr) {
                  setpoint(q, x + size, y + size, z + size, f4thParam);
                  if((this->*TestPoint)(q) < nIterThr) {
                    setpoint(q, x, y + size, z + size, f4thParam);
                    if((this->*TestPoint)(q) < nIterThr) return true;
                  }
                }
              }
            }
          }
        }
      }
      return false;
}
```

It gradually tests all 8 vertices and determines the divergence. The function `TestPoint()` returns the number of iteration after which the divergence was determined. If this number is a very small number, such as 1 (nIterThr), it is very unlikely that the vertex would represent a point near the actual fractal set. Only if all 8 vertices are determined to be far from the fractal set, the tested cube can be skipped. Of course, this is not completely safe, but it can usually increase the performance by 50% - 90%. The variable "nIterThr" can be changed in the GUI. This improvement can also be used to find a fractal set inside a bigger part of a 3D space.

Another recursive algorithm called successive approximation was used to increase the precision of the position of a point located on an edge of a cube:

```
void Fractal::ApproxPoint(int level, tVertex &vert, const Quad &a, const
Quad &b, unsigned int vala) {
      Quad t = {(a.r + b.r) * 0.5f, (a.i + b.i) * 0.5f, (a.j + b.j) * 0.5f,
(a.k + b.k) * 0.5f};
      if(level < nApprox) {
            const bool tp = (this->*TestPoint)(t) >= INF;
            if((vala >= INF && !tp) || (vala < INF && tp)) {
                  ApproxPoint(level + 1, vert, a, t, vala);
            } else {
                  ApproxPoint(level + 1, vert, t, b, vala);
            }
      } else {
            getpoint(vert, t);
      }
}
```

A 3D vertex between points `a` and `b` is returned. The argument "vala" is the result of `TestPoint(a)`.

The function `getpoint()` maps a hypercomplex point back to a 3D vertex according to the choice of a 3D subset.

5.1.2. Generation of 2D Views

2D views are basically approximations of 2D subsets of hypercomplex fractals. Only a very simple method based on the number of iterations is used to color the generated pictures. Additionally, full-screen anti-aliasing is implemented to eliminate jaggy areas.

This is a part of the code used to generate 2D views:

```
      Quad q;
      tVertex c, rot;
      getpoint(c, center);    // get the center of a 3D area

      unsigned int dt = GetTickCount();   // a timer
      float y = c.y + asize;
      unsigned int cy, _cy;
      cy = _cy = nSize;
      do {
            RGBQUAD *p = ptr + (cy - 1) * nSize;
            float x = c.x - asize;
            unsigned int cx = nSize;
            do {
                  unsigned int sum[3] = {0, 0, 0};    // RGB for FSAA
```

```
                    float yAA = y;
                    unsigned int cyAA = nAA;
                    do {  // FSAA inner loop
                          float xAA = x;
                          unsigned int cxAA = nAA;
                          do {
                                rotate2d(rot, xAA, yAA, rotmatrix);
                                setpoint(q, rot.x, rot.y, rot.z, f4thParam);
                                xAA += fstepAA;
                                unsigned ni = (this->*TestPoint)(q) & 0xFF;
                                sum[0] += palette[ni].rgbBlue;
                                sum[1] += palette[ni].rgbGreen;
                                sum[2] += palette[ni].rgbRed;
                          } while(--cxAA);
                          yAA -= fstepAA;
                    } while(--cyAA);
                    RGBQUAD a = {sum[0] >> bitshift, sum[1] >> bitshift,
                                      sum[2] >> bitshift, 0};
                    *p++ = a;   // store a pixel
                    x += fstep;
              } while(--cx);
              y -= fstep;
              if(GetTickCount() - dt >= 40) {
                    if(bAbort) break;
                    PostMessage(hwnd, WM_USER_REDRAW, nSize-_cy, nSize-cy+1);
                    _cy = cy - 1;
                    dt = GetTickCount();
              }
        } while(--cy);
```

Every 40 ms, which corresponds to 25 FPS, a changed part of the window is redrawn. This is optimal because the human eye can distinguish only 24 FPS.

The function `rotate2d()` uses a 3D rotation matrix to rotate a point $(xAA, yAA, 0)$. The returned 3D point is then used in `setpoint()` to get the corresponding hypercomplex point. The 3D matrix is automatically created by rotating an object in the application's 3D view.

Note that $>>$ is the right logical shift, which can be used as a fast replacement for divisions by a power of 2: $\frac{x}{2^y} = x >> y$.

5.1.3. Calculating Box/Cube-counting Dimension

Technically, box/cube-counting dimension calculations are very similar to generating visualizations. In the case of the box-counting dimension, let us take the 2D view code as a template. If we replace the anti-aliasing by a grid refinement and add cube skipping, we will get almost what we need.

The innermost code for the box-counting dimension:

```
        rotate2d(rot, xi, yi, rotmatrix);
        setpoint(q, rot.x, rot.y, rot.z, f4thParam);
        if((this->*TestPoint)(q) >= INF) bIn = true;
        else bOut = true;
        if(bIn && bOut) {
              ncount++;
              goto loop_out;
        }
```

A grid refinement is used to better determine if any part of a fractal set is covered by a box, that is, if there is a point in the grid refinement that diverges, and another point that converges, we can say that the box probably covers a part of a fractal set (convergence is not exactly determined). In such a case, we add 1 to a counter.

In the end, we use the definition from section 3.6.2.

Cube-counting dimensions are constructed similarly, but they take much longer time to compute.

6. Results

6.1. Visualizations

Using the program “QuadFractal”, we managed to visualize 4 main approximations of 3D subsets of hypercomplex fractal sets, and many 2D subsets of these 3D subsets. Of course, there are more than four 3D subsets. Additional 3D subsets are not very interesting because they are created by flipping and rotating these four subsets. However, potentially interesting 3D subsets can be created by 4D rotations. The disadvantage is that 4D rotations are hard to setup so they are not implemented, but they may find their way into the program in the upcoming versions.

The best way to learn more about these fractals is to explore them in the program which provides a simple to use graphic interface. Generally, quaternion fractals have round shapes and hypercomplex (Davenport) fractals have a typical box appearance.

Implemented fractals (all are described in section 3.4.):

- Mandelbrot (including the variable exponent version)
- Julia (including the variable exponent version)
- Barnsley (Mandelbrot and Julia)
- Lambda (Mandelbrot and Julia)
- ManOWar (Mandelbrot and Julia)
- Newton
- Phoenix
- Sierpiński
- Spider
- Breeder (Mandelbrot and Julia)

Additionally, various procedural textures can be used for coloring generated isosurfaces. This includes techniques such as bump-mapping and displacement mapping (not involving OpenGL).

For advanced lighting, coloring, and vertex transformations GLSL shaders were added. They work as a replacement for the fixed functionality inside the graphic card’s GPU so the possibilities are endless.

6.2. Box/Cube-counting Dimensions

We have described how to calculate the box/cube-counting dimension in section 3.6.2.

The program "QuadFractal" was used to calculate the following dimensions of a few approximations of hypercomplex fractals and their subsets such as complex numbers.

The numbers of sub-cubes represents the number of cubes in a grid refinement.

6.2.1. Mandelbrot Set

Complex Mandelbrot set - 10 iterations, sub-boxes: 4^2

r	N(r)	ln(1/r)	ln(N(r))	D_B between 2 points
0.1000000	89	2.3025851	4.4886364	1.2037467
0.0500000	205	2.9957323	5.3230100	1.0853915
0.0250000	435	3.6888795	6.0753460	1.0695995
0.0125000	913	4.3820266	6.8167359	1.0351228
0.0062500	1871	5.0751738	7.5342283	1.0237077
0.0031250	3804	5.7683210	8.2438084	1.0355869
0.0015625	7798	6.4614682	8.9616226	0.9808172
0.0007813	15390	7.1546154	9.6414732	
			Arithmetic Mean:	1.0619960

Complex Mandelbrot set - 100 iterations, sub-boxes: 4^2

r	N(r)	ln(1/r)	ln(N(r))	D_B between 2 points
0.1000000	82	2.3025851	4.4067192	1.2644998
0.0500000	197	2.9957323	5.2832037	1.3027607
0.0250000	486	3.6888795	6.1862086	1.4523980
0.0125000	1330	4.3820266	7.1929342	1.5364269
0.0062500	3858	5.0751738	8.2579042	1.6723813
0.0031250	12297	5.7683210	9.4171106	1.6990221
0.0015625	39926	6.4614682	10.5947830	1.7032665
0.0007813	130014	7.1546154	11.7753974	
			Arithmetic Mean:	1.5186794

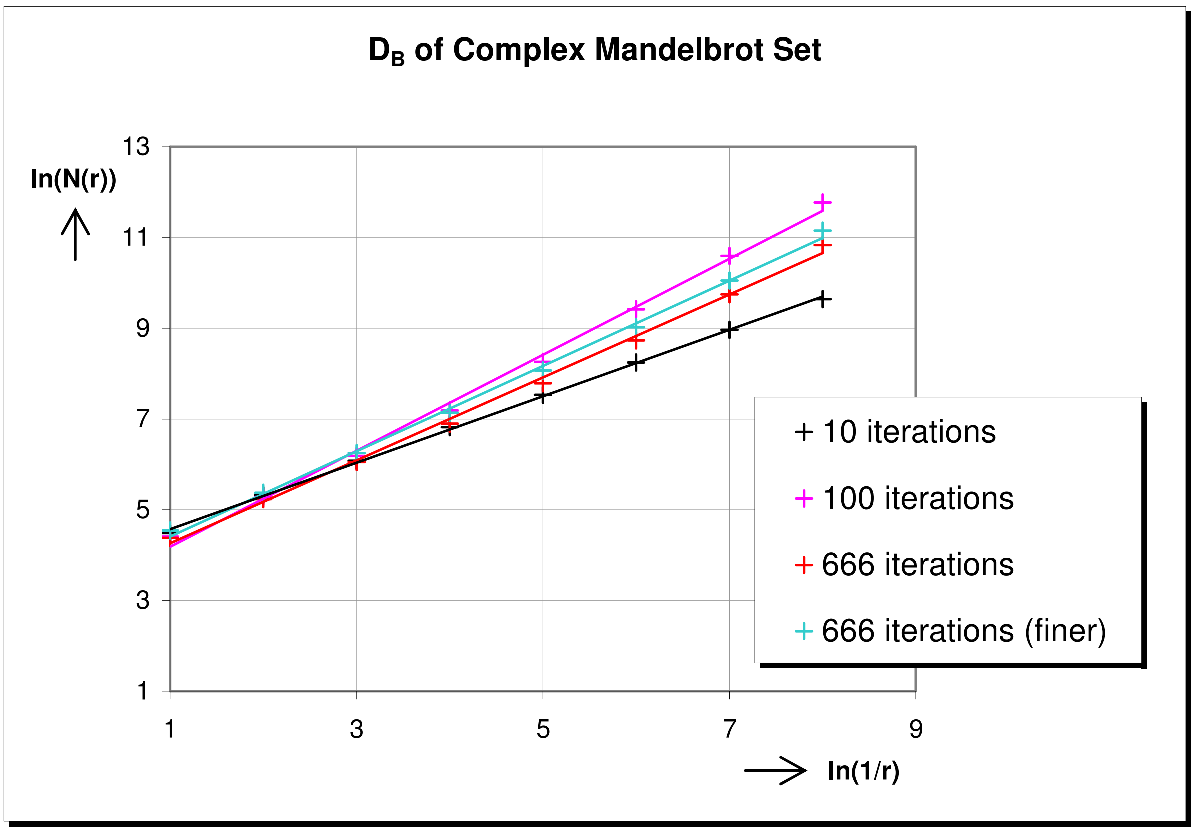
Complex Mandelbrot set - 666 iterations, sub-boxes: 4^2

r	N(r)	ln(1/r)	ln(N(r))	D_B between 2 points
0.1000000	80	2.3025851	4.3820266	1.2249664
0.0500000	187	2.9957323	5.2311086	1.1776194
0.0250000	423	3.6888795	6.0473722	1.2180007
0.0125000	984	4.3820266	6.8916259	1.2988729
0.0062500	2421	5.0751738	7.7919360	1.3615434
0.0031250	6221	5.7683210	8.7356859	1.4618962
0.0015625	17137	6.4614682	9.7489951	1.5625074
0.0007813	50617	7.1546154	10.8320428	
			Arithmetic Mean:	1.3293438

As we can see, the calculated dimensions are not quite what we would expect since we know that the Hausdorff dimension of the complex Mandelbrot set is equal to 2, but we can observe some characteristics. For a very small number of iterations (typical of hypercomplex visualizations), the fractal set is more similar to a regular object so the dimension is closer to 1. As we increase the number of iterations, the dimension increases. However, at some point, it decreases again. This is caused by the constant number of sub-boxes that we chose for all three tables. As the number of iterations increases, we have to increase the grid refinement. Note that the definition of the box-counting dimension does not cover this problem. It assumes that all boxes covering a fractal set are exactly identified. Let us see what happens if we increase the number of sub-boxes.

Complex Mandelbrot set - 666 iterations, sub-boxes: 8^2

r	N(r)	ln(1/r)	ln(N(r))	D_B between 2 points
0.1000000	94	2.3025851	4.5432948	1.1936040
0.0500000	215	2.9957323	5.3706380	1.2602358
0.0250000	515	3.6888795	6.2441669	1.2919239
0.0125000	1261	4.3820266	7.1396603	1.3471059
0.0062500	3208	5.0751738	8.0734030	1.3690017
0.0031250	8286	5.7683210	9.0223226	1.4803313
0.0015625	23119	6.4614682	10.0484101	1.5865841
0.0007813	69435	7.1546154	11.1481463	
			Arithmetic Mean:	1.3612552



Indeed, the dimension increased, but only slightly. This leads to the conclusion that much more precise results can be achieved by greatly increasing the grid refinement. Unfortunately, this is a great disadvantage of the box-counting dimension, because it means that the calculated dimension converges to the exact box-counting dimension very slowly. Of course, we cannot be sure that the dimension will converge to some larger number from the tables above. However, in the case of the complex Mandelbrot set, we can. We know that the Hausdorff dimension D_H of the set is equal to 2, and we also know that $D_H \le D_{\underline{B}} \le D_{\overline{B}}$, where D_B exists if $D_B = D_{\underline{B}} = D_{\overline{B}}$. The topological dimension of the set is equal to 1, so D_B can be 2 at the most. This leaves only one option: $D_B = 2$, or, it does not exist, which would be unfortunate.

Note that many of these calculations take minutes or hours to complete, so some finer grids were skipped in the following tables.

Quaternion Mandelbrot set - 100 iterations, sub-cubes: 4^4

r	N(r)	ln(1/r)	ln(N(r))	D_B between 2 points
0.1000000	14920	2.3025851	9.6104579	3.3607182
0.0500000	153266	2.9957323	11.9399303	3.5197450
0.0250000	1757902	3.6888795	14.3796316	
			Arithmetic Mean:	3.4402316

3D subset of hypercomplex Mandelbrot set - 100 iterations, sub-cubes: 4^3

r	N(r)	ln(1/r)	ln(N(r))	D_B between 2 points
0.1000000	910	2.3025851	6.8134446	2.2550857
0.0500000	4344	2.9957323	8.3765509	2.3808855
0.0250000	22626	3.6888795	10.0268550	2.3965923
0.0125000	119139	4.3820266	11.6880462	
			Arithmetic Mean:	2.3441878

Hausdorff dimensions of hypercomplex Mandelbrot sets are unknown up to now.

6.2.2. Sierpiński Triangle and Pyramid

Sierpinki triangle - 10 iterations, sub-boxes: 8^2

r	N(r)	ln(1/r)	ln(N(r))	D_B between 2 points
0.0100000	2259	4.6051702	7.7226775	1.58496250072116
0.0050000	6777	5.2983174	8.8212898	

The dimension of Sierpiński triangle was calculated exactly, because:

$$D_B = \frac{\ln \frac{6777}{2259}}{\ln \frac{0.01}{0.005}} = \frac{\ln 3}{\ln 2} = D_H. \tag{6.1}$$

Let us try the same parameters for Sierpiński pyramid:

Sierpinki pyramid - 10 iterations, sub-cubes: 8^3

r	N(r)	ln(1/r)	ln(N(r))	D_B between 2 points
0.2000000	28	1.6094379	3.3322045	2.0913183
0.0100000	14724	4.6051702	9.5972341	2.2947081
0.0050000	72244	5.2983174	11.1878046	

We know that $D_H = \frac{\ln 4}{\ln 2} = 2$ because there are 4 new half-sized pyramids in each pyramid. Obviously, the calculation was not so exact this time. After an infinite number of iterations, the fractal becomes a network of lines whose topological dimension is 1. This is the reason why we have to choose parameters for calculations very carefully. Too high number of iterations causes the algorithm to fail in finding the fractal set because it cannot find any convergent points by systematic testing of the points in a grid.

6.2.3. Exponential Julia Set

3D subset of quaternion Julia set $z_{n+1}=z_n^5 + 0.7 + 0.8i$:
100 iterations, sub-cubes: 4^3

r	N(r)	ln(1/r)	ln(N(r))	D_B between 2 points
0.1000000	2110	2.3025851	7.6544432	2.3634435
0.0500000	10858	2.9957323	9.2926574	2.3760412
0.0250000	56365	3.6888795	10.9396037	2.3722518
0.0125000	291829	4.3820266	12.5839233	
			Arithmetic Mean:	2.3705788

Not much has been known about dimensions of exponential Julia sets, but it can be observed that by increasing the exponent of z_n, the fractal becomes more complex. This means that the refinement grid would have to be much more precise to get a reasonably accurate result. This is no problem in theory, but it takes a lot of computation time in practice.

7. Conclusion

We have developed basic mathematical principles and functions in hypercomplex spaces, applied these functions to escape-time fractal sets, and used these fractals to generate various visualizations and calculate dimensions.

Although the developed application is intended only for Windows^{TM} based systems, it provides an extensive environment capable of visualizing approximations of subsets of hypercomplex fractals in a user friendly way, without the need to learn a script language. The disadvantage of such an implementation is a limited support of fractal formulas, that is, only hard-coded fractals are available. The advantage is the increased performance of such an approach, avoiding dynamic compilations.

Another advantage is that the application can generate 3D fractal models, and export them to various 3D file formats. Formats X3D and VRML97 are standardized formats for web deployment. Although they are often a few MiB big, they provide the best way to deploy 3D models for web since they are text formats similar to XML (X3D is XML). There is a way to load exported models in a different application without any support for parsing of 3D formats, because my application can produce a C/C++ header file with all necessary data and functions ready to use in OpenGL. The ability to import 3ds and Zipper (Stanford) models provides a way to convert 3D files.

From my point of view, the visualization using the Marching Cubes algorithm is more beneficial than Ray-tracing which cannot produce 3D models and is not supported by hardware.

The implemented dimension calculator is sometimes useful, and, sometimes, it produces inconclusive results due to limited computation time.

8. Bibliography

[1] Gauss C. F. (1799), 'New Proof of the Theorem That Every Algebraic Rational Integral Function In One Variable can be Resolved into Real Factors of the First or the Second Degree', Helmstedt.

[2] Hamilton W. R. (1853), 'Lectures on Quaternions', Royal Irish Academy.

[3] van der Waerden B. L. A (1985), 'History of Algebra from al-Khwarizmi to Emmy Noether', New York, Springer-Verlag.

[4] Davenport C. M. (1991), 'A Commutative Hypercomplex Calculus with Applications to Special Relativity', Knoxville, Tennessee.

[5] Mandelbrot B. B. (1967), 'How Long is the Coast of Great Britain, Statistical Self Similarity and Fractional Dimension', Science.

[6] Kolwankar K. M., Gangal A. D. (1996), 'Fractional differentiability of nowhere differentiable functions and dimensions', India, Pune.

[7] Peitgen H.-O., Jurgens H., Saupe D. (1992), 'Chaos and Fractals: New Frontiers of Science', Springer-Verlag, New York.

[8] Peitgen H.-O., Saupe D., (eds.) (1988), The Science of Fractal Images, Springer Verlag, New York.

[9] Edgar G. (1989), 'Measure, Topology and Fractal Geometry', Springer-Verlag, Berlin.

[10] Vepstas L. (1997), 'Renormalizing the Mandelbrot Escape', http://linas.org/art-gallery/escape/escape.html

[11] Shishikura M. (1994), 'The Boundary of the Mandelbrot Set has Hausdorff Dimension Two', Astérisque.

[12] Peitgen H.-O., Richter P. H. (1986), 'The Beauty of Fractals', Springer-Verlag, Berlin Heidelberg.

[13] Shirley P., Morley K. R. (2001), 'Realistic Ray Tracing, 2nd edition', A.K. Peters.

[14] Šupina P. (2005-2006), 'QuadFractal', http://flashlight.slad.cz

[15] de Soras L. (2004), 'Fast Rounding of Floating Point Numbers in C/C++ on Wintel Platform', http://ldesoras.free.fr

[16] Lorensen W. E., Cline H. E. (1987), 'Marching cubes: A high resolution 3D surface construction algorithm', ACM Press, New York.