分形算法与程序设计

——用 Visual C++实现

孙博文 编著

科 学 出 版 社

北 京

内 容 简 介

本书从实用的角度出发，论述了分形图形的生成算法与程序设计。内容包括分形图的递归算法、文法构图算法、迭代函数系统算法、逃逸时间算法、分形演化算法，以及分形图的放大、分形图的动画、分形图的立体化和利用分形算法实现自然景物的模拟等内容。

本书共分 10 章，图文并茂，集中介绍了近年来分形图形学的研究成果，给出了相应的算法和 Visual C++程序设计源代码，使读者易学、易掌握、易用。只要具备高中的数学知识和 Visual C++程序设计的能力，便可轻松阅读此书。

本书可供数学、物理、计算机、艺术设计、工业造型、影视动画制作等专业的本专科学生阅读学习，也可供从事计算机绘图、数字图像处理等领域的研究人员和工程技术人员参考，还可供广大分形爱好者参考阅读。

图书在版编目(CIP)数据

分形算法与程序设计：用 Visual C++实现/孙博文编著. —北京：科学出版社，2004

ISBN 7-03-014542-9

Ⅰ. 分… Ⅱ. 孙… Ⅲ. ①分形理论-算法分析 ②C 语言-程序设计 Ⅳ.①0189.3 ②TP312

中国版本图书馆 CIP 数据核字（2004）第 111603 号

责任编辑：万国清 丁 波／责任校对：耿 耘

责任印制：吕春珉／封面设计：飞天创意

科 学 出 版 社 出版

北京东黄城根北街 16 号

邮政编码：100717

http://www.sciencep.com

印刷

科学出版社发行 各地新华书店经销

*

2004 年 11 月第 一 版 开本：787×1092 1/16
2004 年 11 月第一次印刷 印张：20 插页：2
印数：1－4 000 字数：456 000

定价：38.00 元（含光盘）

（如有印装质量问题，我社负责调换〈 〉）

前　言

“事实上，无论是从美学的观点还是从科学的观点，许多人在第一次见到分形时都有新的感受”（曼德勃罗语）。确实如此，这句话不仅说出了笔者的亲身体验，也说出了许许多多分形爱好者的体验。

分形图的玄妙与优美让笔者为之倾心十几年，恐怕今后的岁月中也很难摆脱它的“诱惑”。它自然而优雅，纷繁而又有序，在绚丽的色彩变化背后透露着几分神秘。正如曼德勃罗所说，“在外行看来，分形艺术似乎是魔术。但不会有任何数学家疏于了解它的结构和意义”。笔者不是数学家，但同样对分形结构十分着迷，作为一名计算机图形学教师，笔者更关心的是这些玄妙的图形是如何构造出来的？事实上，几乎所有喜欢分形的人都曾提出过这样的问题，本书便是笔者对这一问题的部分解答。

本书图文并茂，浅显易懂。全书共分 10 章，第 1 章为分形简介，力图回答这样一个问题：分形是什么？主要介绍了分形的概念与定义、分形的特征与测量、分形的方法论意义及其与自然的关系，以及分形与计算机图形学之间的关系等；第 2 章介绍构造分形图的递归算法，以丰富的实例体现递归在分形图中的妙用；第 3 章为文法构图算法，主要介绍 LS 文法的构图原理与规则实践；第 4 章为迭代函数系统算法，主要介绍相似变换与仿射变换及利用仿射变换的原理构造生成分形图的算法;第 5 章为逃逸时间算法，这一算法所产生的丰富而美丽的图形是分形打动人心的秘密武器；第 6 章介绍分形显微镜；第 7 章为分形演化算法，重点介绍两个生成分形图的演化模型，一个是元胞自动机模型，另一个是扩散有限凝聚模型（DLA 模型）；第 8 章介绍分形动画，以动画的形式表现分形的玄妙，同时阐述了分形动画的基本原理与算法；第 9 章介绍三维空间中的分形，将分形绘图投入到三维空间之中，重点介绍了 OpenGL 函数库的功能与用法以及如何利用 OpenGL 函数库构造三维空间中的分形；第 10 章为分形自然景物模拟算法，利用分形构图方法，我们可以构造逼真的自然景象。

当然，分形图不只是用来欣赏的，它代表着几何学的一个新的研究方向，即对非规整几何对象的研究。这一任务是传统几何学所不能胜任的，所以诞生了分形几何学。因为大自然中存在着大量的非规整几何对象，而分形几何又能很好地表达和模拟这些自然景物，因此，分形几何学也被称为大自然的几何学。由于分形几何对象是不规整的，所以借助三角板和圆规实现手绘几乎是不可能的（极其简单的分形图除外），因此要想研究分形图，必须要得到计算机的帮助，而本书所介绍的分形算法，恰恰是利用计算机生成这些复杂分形图的必备工具。同时，利用分形算法所生成的自然景物，也已经或将在科幻影片和电子游戏中得到应用。

学习分形需要较深的数学基础，这使许多人望而却步。而本书恰恰是照顾到有不同知识背景的读者，有意回避分形的数学问题，将重点放在分形图的计算机算法构造和实现上，所以读者只要具备高中的数学知识，就可以看懂书中的所有算法。如果同时读者又具备了 Visual C++的编程能力，便可以看懂书中算法所携带的 Visual C++程序。另外，

由于有了算法的源代码，可以调试和修改程序中的参数，从而产生许多意想不到的美丽图形，大大增加了读者的参与感和创新性，并满足了部分读者的适用需求。本书的算法和程序设计是笔者多年学习和研究的结果，书中的许多内容在其他的分形类书籍和计算机图形学书籍中很少出现，这将成为读者深入研究分形理论的一个很好的台阶。

本套书共 4 本，分别是“分形算法与程序设计（Visual Basic 版）”、“分形算法与程序设计（Visual C++版）”、“分形算法与程序设计（Delphi 版）”、“分形算法与程序设计（Java 版）”，本书是其中的一本。本套书的写作特点是：所设计的分形算法基本相同，所举实例也大体类似，只是分别用各自的编程语言来实现这些实例，请读者参考阅读。

本书的问世，不仅仅是笔者本人努力的结果，而且还凝结了许多人的心血。书中的所有算法均在哈尔滨理工大学的分形图形学讨论班中宣读，得到讨论班部分成员：孙百瑜、张海波、周烨、马强、赵衍鑫、龚宗耀、李文利和潘艺民等人的许多中肯的意见与建议，而且，张海波、周烨、马强和刘天立等还参与了部分算法的编写或调试，在此向他们表示深深的谢意。另外，科学出版社的编辑为此书的出版费尽心血，在此也要向他们表示感谢。最后还要感谢我的家人，由于他们的理解和支持，使我得以安心此书的写作，并完成它。

由于笔者水平有限，书中错误和疏漏在所难免，敬请广大读者批评指正。

孙博文

2004 年 3 月 31 日于哈尔滨

目　　录

第 1 章　分 形 简 介

分形是自然界的几何学。

——曼德勃罗（分形理论创始人）

分形作为现代科学的“时髦”概念，已经广泛应用于物理、化学、生物、医学、地理、地质、材料科学、计算机科学以及经济学、哲学、社会学等诸多领域。它之所以“时髦”，是因为它使我们看到了一些以前没被注意到的东西，或者说，它让我们用另一种眼光看世界。

1.1　分形概念的提出与分形理论的建立

分形在英文中为 fractal，是由美籍法国数学家曼德勃罗（Benoit Mandelbrot）创造出来的。此词源于拉丁文形容词 fractus，对应的拉丁文动词是 frangere（破碎、产生无规碎片）。此外，它与英文的 fraction（碎片、分数）及 fragment（碎片）具有相同的词根。在 20 世纪 70 年代中期以前，曼德勃罗一直使用英文 fractional 一词来表示他的分形思想。因此，取拉丁词之头，撷英文之尾所合成的 fractal，本意是不规则的、破碎的、分数的。曼德勃罗是想用此词来描述自然界中传统欧氏几何学所不能描述的一大类复杂无规的几何对象，例如，蜿蜒曲折的海岸线、起伏不定的山脉，粗糙不堪的断面，变幻无常的浮云，九曲回肠的河流，纵横交错的血管，令人眼花缭乱的满天繁星等。它们的特点是，极不规则或极不光滑。直观而粗略地说，这些对象都是分形。

1975 年，曼德勃罗出版了他的法文专著《分形对象：形、机遇与维数》（Les objects fractals: forme, hasard et dimension），标志着分形理论正式诞生。1977 年他又出版了该书的英译本。1982 年曼德勃罗的另一部历史性著作《大自然的分形几何学》与读者见面，该书虽然是前书的增补本，但在曼德勃罗看来却是分形理论的“宣言书”，而在分形迷的眼中，它无疑是一部“圣经”。该书旁征博引、图文并茂，从分形的角度考察了自然界中的诸多现象，引起了学术界的广泛注意，曼德勃罗也因此一举成名。

此后，一直持续的分形热吸引了全世界众多科学家和学者，他们在各自领域中的研究工作，使分形理论遍地开花。

1.2　分形的几何特征

分形作为几何对象，首先是破碎的、不规则的，但不是所有破碎的、不规则的形状都是分形。曼德勃罗（1986 年）曾经给分形下过这样一个定义：组成部分与整体以某种方式相似的形。也就是说，分形一般具有自相似性。但分形理论发展到今天，已经不仅

限于研究对象的自相似性质了，如果一个对象的部分与整体具有自仿射变换关系，我们也可以称它为分形。今后，条件可能还会进一步拓宽，只要是部分与整体以某种规则联系起来，通过某种变换使之对应，我们都可以将其看成是分形，因为分形的本质就是标度变换下的不变性，而这层意思是可以拓展的。

1. 自相似性

自相似便是局部与整体的相似，或者说，局部是整体的缩影等。下面举几个典型的例子来使读者更好地理解对象的自相似性。

（1）Cantor 三分集

集合论创始人，德国数学家康托（G.Cantor，1845～1918 年）在 1883 年曾构造了一种三分集，其几何表示如下：

取一条欧氏长度为 L_0 的直线段，L_0 叫做初始操作长度。将这条直线段三等分之后，保留两端的线段，将中间的一段扔掉，如图 1.1 所示 n=1 的操作；再将剩下的两条直线段分别三等分，然后将其中间部分扔掉，如图 1.1 所示 n=2 的操作，以此类推，直至无穷，便形成了无数个尘埃似的点，这便是 Cantor 三分集。它们的数目无穷多，但长度为零。这种构造的自相矛盾性质曾使 19 世纪的数学家感到困惑。但我们从几何关系来看，最终生成点的分布是局整相似的，甚至，这个过程中每一步图形之间也是局整相似的，这便是自相似。

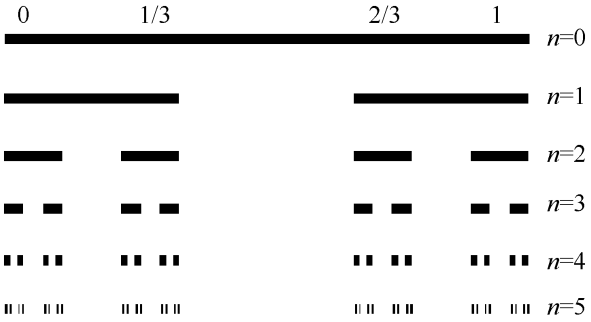

图 1.1　Cantor 三分集

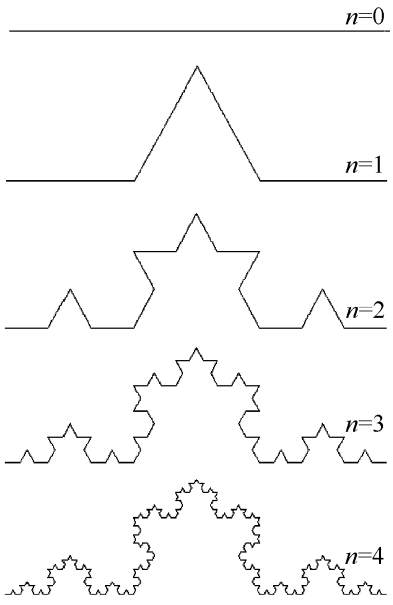

图 1.2　Koch 曲线

（2）Koch 曲线

1904 年, 瑞典数学家科赫（H.von Koch，1870～1924 年）构造了一种“妖魔曲线”，被称为 Koch 曲线。其构造过程如下：

取一条欧氏长度为 L_0 的直线段，将其三等分，保留两端的线段，将中间的一段改换成夹角为 60° 的两个等长的直线，如图 1.2 中 n=1 的操作。将长度为 $L_0/3$ 的 4 个直线段分别进行三等分，并将它们中间的一段均改换成夹角为 60° 的两段长为 $L_0/9$ 的直线段，得到图 1.2 中 n=2 的操作。重复上述操作直至无穷，便得到一条具有自相似结构的折线，这便是我们所说的 Koch 曲线。

（3）Sierpinski 垫片

以上两个自相似图形都是基于一条欧氏直线段生成的，Cantor 三分集是将线段删去一部分，最终得到的是一个离散的点集，而 Koch 曲线是将线段增加一部分，最终得到的是一个处处不光滑的折线集。波兰数学家谢尔宾斯基（W.Sierpinski，1882～1969 年）于 1915 年给出了一个从平面上的二维图形出发做曲线的有趣例子。所构造的 Sierpinski 垫片相当于将上述构造方法推广到平面上，其初始图形是一个等边三角形面，构造过程如下：

首先，我们将这个等边三角形面四等分，得到 4 个小等边三角形面，去掉中间一个。将剩下的 3 个小等边三角形面分别进行四等分，再分别去掉中间的一个。重复以上操作直至无穷，可以得到如图 1.3 所示的图形，可以看出它的每一小部分在结构上都与整体相同，这也是一个典型的自相似图形。

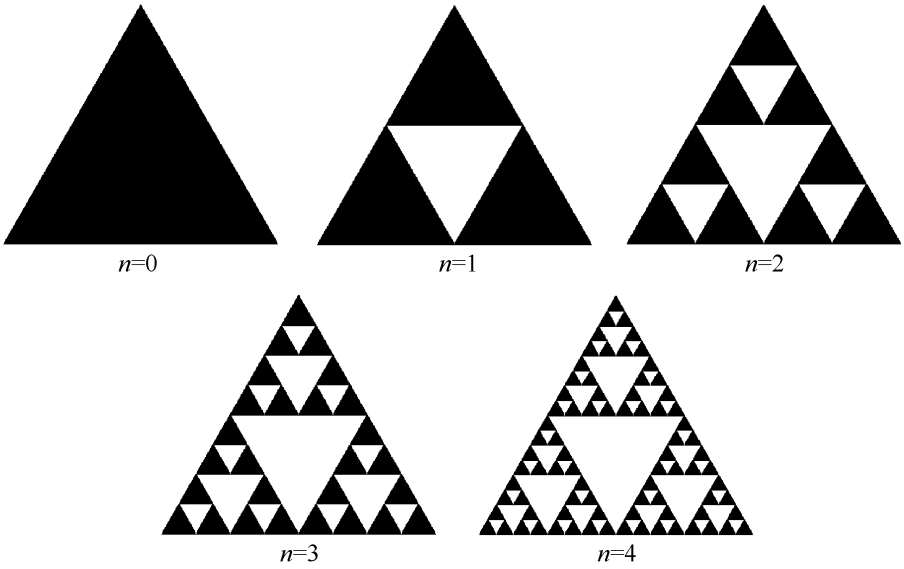

图 1.3　Sierpinski 三角形

2. 自仿射性

自仿射性是自相似性的一种拓展。如果将自相似性看成是局部到整体在各个方向上的等比例变换的结果的话，那么，自仿射性就是局部到整体在不同方向上的不等比例变换的结果。前者称为自相似变换，后者称为自仿射变换。图 1.4 表示的是相似变换与仿射变换的不同。

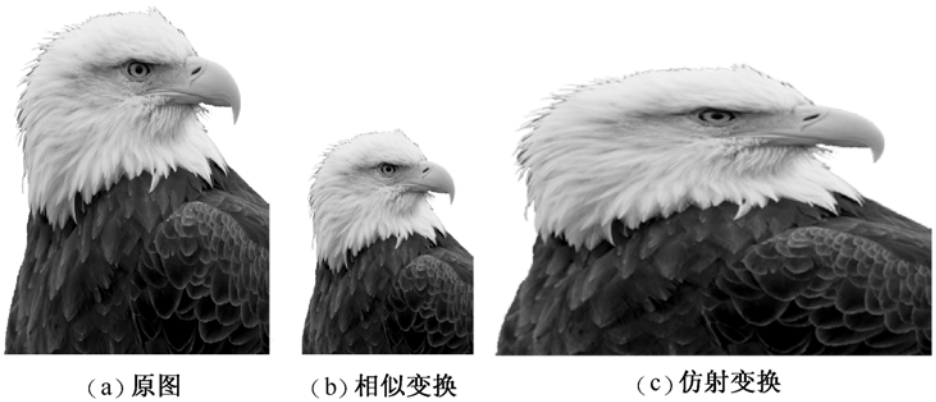

图 1.4　相似变换与仿射变换的不同

3. 精细结构

分形还有一个更重要的特征，即精细结构。在理论上，Koch 曲线是按一定规则无限变化的结果，所以，假如用一个数学放大镜来看 Koch 曲线的话，无论放大多少倍，都能看到里面还有与整体相似的结构。这一点非常接近于自然界中的对象，也符合中国古代的哲学思想：“一尺之捶，日取其半，万世不竭”（庄子《天下篇》）。在这里，我们不打算讨论物质是否无限可分，我们只是注意到分形和自然对象都具有极多层次的结构，这是分形体最基本的特征。但是，自然界中的对象与数学中的分形还是有所不同。在自然界中，对象即使存在自相似性，也是在有限层次区间中，所以在对自然界的分形研究中，一般我们只关心有限层次的自相似性或自仿射性结构。

4. 分形与欧氏几何图形的区别

可以看出，分形与普通的欧氏几何图形有明显的区别：

①欧氏图形是规则的，而分形是不规则的，也就是说，欧氏图形一般是逐段光滑的，而分形往往在任何区间内都不具有光滑性。

②欧氏图形层次是有限的，而分形从数学角度上讲，层次是无限的。

③欧氏图形一般不会从局部得到整体的信息，因为它们不强调局部与整体的关系，而分形强调这种关系，所以，分形往往可以从局部“看出”整体。

④欧氏图形越复杂，其背后的规则也必定越复杂，而对于分形图形，虽然看上去十分复杂，但其背后的规则却是相当简单的。

因此，我们要构造、绘制的图形是与中学学到的欧氏图形完全不同的，首先必须找出分形对象“不规则”的规则，然后在计算机的帮助下才能将其生成出来。

1.3　分形的测量

分形既然不是传统意义上的图形，我们如何来把握它的性质呢？这是一个较艰深的数学问题，已经超出本书的范围。通俗地讲，用传统几何学的测量方法是无法把握住分形中的不变量的，因为它既不是一些点的轨迹，也不是某一方程或方程组的解，最主要的是它没有确定的标度。

1. 标度

标度，简单地说，就是计量单位的定标。比如米尺的标度是米或分米；学生用尺的标度是厘米或毫米；卡尺的标度是毫米或微米；磅秤的标度是公斤；天平的标度是毫克等。

对于不同的被测对象，可选用不同的测量工具。我们不会用卡尺去测量人的身高，也不会用天平去称大象的重量，这说明人的身高和大象的重量都是有确定标度的。

而分形则不然，由于自相似性，当变化尺子的标度时，我们看到的是相同或相似的图形。这类对象是没有确定标度的，反过来说，在标度变化下是不变的。从这个角度看，分形的本质是标度变化下的不变性。

那么，如何来把握这种不变性呢？现在我们知道了，只有分维才能担当此任。

2. 分维

分维是分形很好的不变量，用它可以把握住分形体的基本特征。那么，什么是分维呢？

我们还记得在欧氏几何学中所学的维数吗？点是 0 维，线是 1 维，平面是 2 维，立体是 3 维。好像维数一定是整数，其实不然。

让我们来看这样一个例子：

图 1.5（a）是边长为 1 的正方形，当边长变成原来的 1/2 时，原正方形中包含 4 个小正方形，如图 1.5（b），而 $4=2^2$。

图 1.5（c）是边长为 1 的正立方体，当边长变成原来的 1/2 时，原正立方体中包含 8 个小正立方体，如图 1.5（d），而 $8=2^3$。

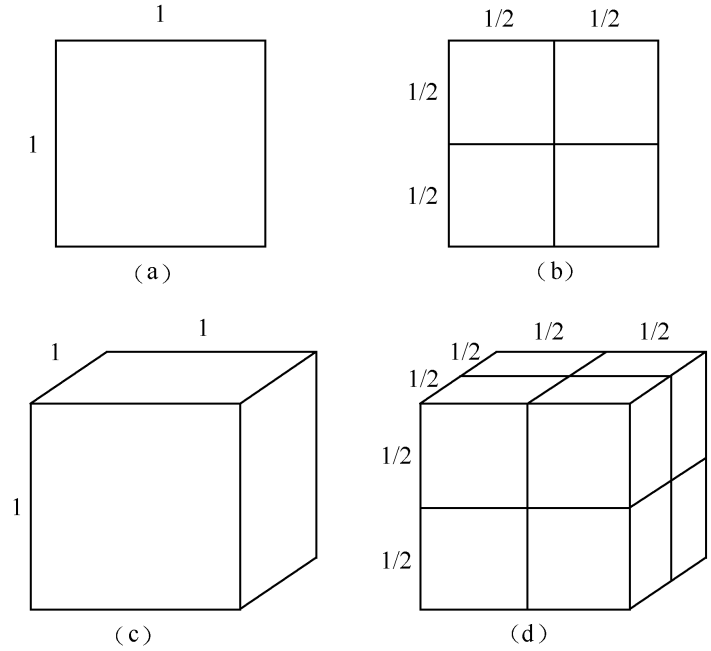

图 1.5　相似维数的示例

这里我们发现一个小秘密，表达式 $4=2^2$ 和 $8=2^3$ 的 2 上面的幂，恰好是相应的正方形和正立方体的维数。

如果将上面的关系写成通式，则有

$$N=k^D \tag{1.1}$$

其中，k 为边长缩小的倍数，N 为边长缩小 k 倍后新形体的个数，则 D 便是形体所具有的维数。

如果我们将式（1.1）的两边同时取对数，可得

$$\lg N=D\lg k$$

则

$$D=\frac{\lg N}{\lg k} \tag{1.2}$$

从式（1.2）可以看出，维数 D未必一定是整数。所以说，分数维是有权利存在的。

可是分数维有必要存在吗？我们来看看下面的例子：

一条线段，如果我们用 0 维的点来测量它（数学中的测量可以看成是一种覆盖，即用测量尺子去覆盖被测对象），得到的结果是无穷大，因为线段中包含无穷多个点。如果

用 2 维的单位小平面来测量此线段，得到的结果将是 0，因为线段中不包含平面。那么，用什么样的尺子测量它才能得到一个确定大小的有限值呢？看来只有用 1 维的单位线段来测量它才能得到有限值。

于是，可以得出一个结论：

若测量尺子的维数小于被测对象的维数时，其测量结果是无穷大；若测量尺子的维数大于被测对象的维数时，其测量结果为 0；只有测量尺子的维数与被测对象的维数相等时，其结果才是有限值，并且这个维数一定在上述两个维数之间。

那么，用什么样的尺子来测量 Koch 曲线才会得到有限值呢？用 1 维的线来测量它，其结果是无穷大，因为当 Koch 曲线迭代到无穷多次时，Koch 曲线将是无限长；若用 2 维的小平面来测量它，其结果将是 0，因为 Koch 曲线中没有平面。看来只好选一个大于 1 维且小于 2 维的尺子来测量它，才会得到有限值。如此说来，Koch 曲线自身的维数是一个分数，实际上，它的维数 D=1.26186。

其实，分形的维数一般都是分数（我们叫它为分形维数，或分维），当然也有例外，比如说 Peano 曲线，它的分维数是 2。

那么，Koch 曲线的分维数是怎样算出来的呢？

根据 Koch 曲线的生成规则可知，单位 1 的线段被分解为原来的 1/3 后，得到 4 条小线段。将其代入式（1.2）中，式中的 k 在这里等于 3；式中的 N，在这里等于 4，则

$$D=\frac{\lg 4}{\lg 3}=1.26186$$

用同样方法，我们可以计算出 Cantor 三分集的分维数为 0.6309，Sierpinski 垫片的分维数为 1.58496，你不妨自己计算一下。

当然，分维数的计算还有很多方法，不同的方法适用于测量不同类型的分形体。上面提到的计算分维数的方法，实际上是计算分形的相似维数，只适用于具有严格自相似的分形体，而对那些具有统计自相似性的分形，比如海岸线的分维数，只能寻找其他方法来计算。

对于具有统计自相似性的分形来说，盒维数计算法是一个不错的方法，它出现在 20 世纪 20 年代并被冠以种种名称，如度量维、信息维、Kolmlgorov 熵、熵维、容量维等。由于它是 1929 年首先被 Bouligand 引入的，故又称为 Bouligand 维。

其具体算法如下：

①在被测图上覆盖边长为 ε 的小正方形，统计一下有多少个正方形中含有被测对象，记入 N 中。

②缩小正方形，使 $\varepsilon=\varepsilon/2$，再统计一下有多少个正方形中含有被测对象，记入 N 中，以此类推。

③统计不同的 ε 值下记入的 N 值，然后分别取对数，在双对数坐标系中画出统计曲线，如图 1.6 所示。

④计算曲线中部近似直线的斜率，便是我们要找的分维数 D。

曼德勃罗曾经计算过英国西海岸线的分形维数，其值 D=1.25。

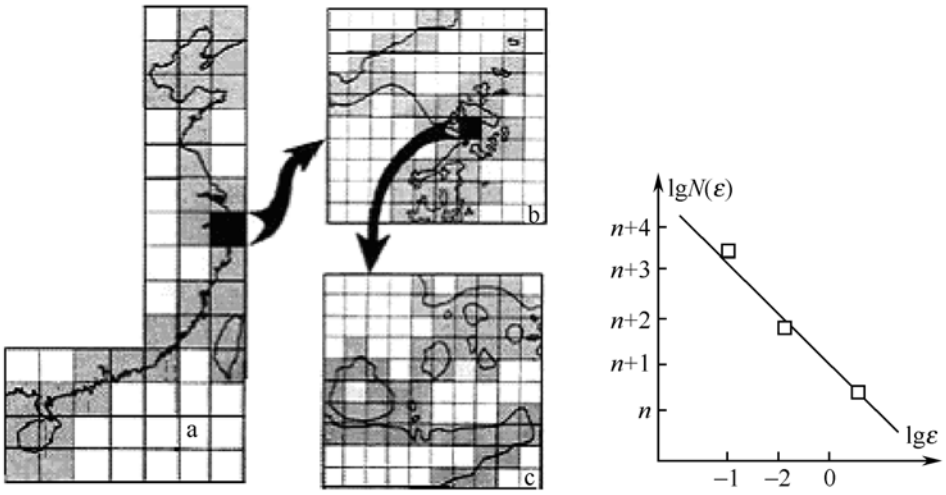

图 1.6　对海岸线的盒维数测量

1.4　自然界中的分形

《大自然的分形几何学》一书的出版，使我们看到分形几何与大自然中各种形态的联系，我们甚至惊呼分形无处不在。

大自然中到处隐藏着分形的奥秘：从我们餐桌上的菜花（图 1.7）到植物叶脉的纹理（图 1.8）；从大树枝干的分叉结构（图 1.9）到黑夜中闪电的痕迹（图 1.10）；从流向大海的江河（图 1.11）到错落起伏的山岭（图 1.12）等。这些用欧氏几何学无法描述的现象，用分形几何学可以很容易构造出相应的模型，并获得其模拟的几何性质。然而，自然界中并不存在严格意义上的分形，就像自然界中找不到真正的欧氏圆和直线一样。因为从严格意义上说，分形都是变化到无穷小尺度时才产生的，而自然形态只是停留在一定层次范围内才可以用合理的分形模式来考虑。也就是说，自然界中的分形特征只存在于某段层次区间内，而且这段区间内的形态结构也不是完全自相似的，它只具有统计意义上的自相似性。尽管如此，这已足以使我们获得更多有关自然形态的知识，并对自然界有了更深刻的理解。毕竟所有的理论都是对现实的一种抽象，分形也不例外，它是自然形态的几何抽象。“云团不是球形，山峦不是锥形，海岸线不是圆的，树皮不是光的，闪电不会沿直线行进。所有这些自然结构都具有不规则形状”（曼德勃罗）。所以，用传统的欧氏几何学描述它们是十分困难的。从这个意义上说“分形是自然界的几何学”一

图 1.7　菜花的剖面

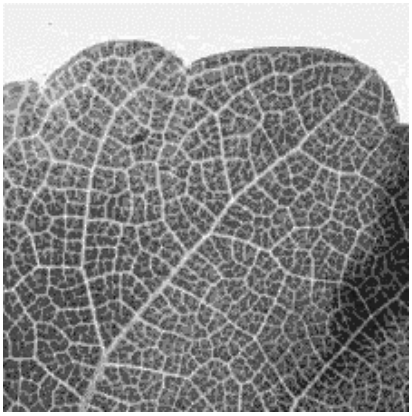

图 1.8　植物的叶脉

图 1.9　树木的枝干分叉显现自相似性

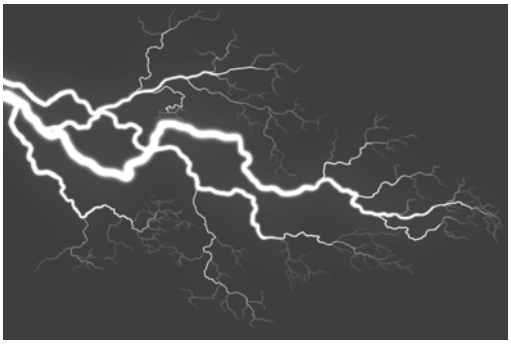

图 1.10　闪电的自相似结构

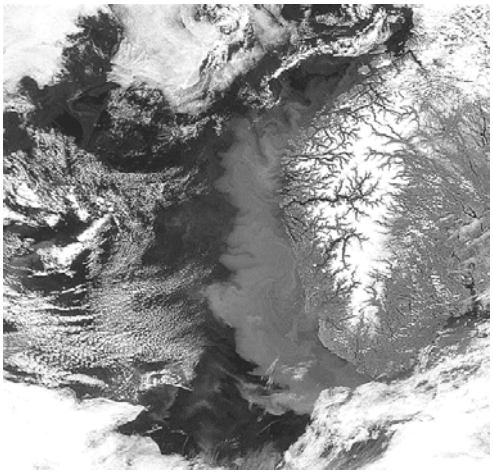

图 1.11　海岸的卫星照片

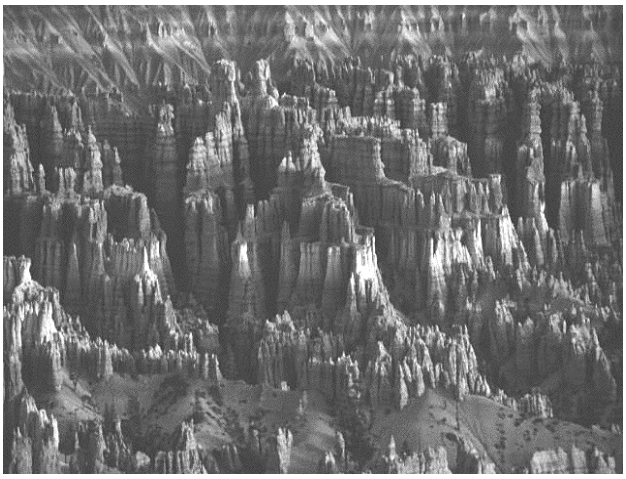

图 1.12　山脉

点也不过分，因为迄今为止，还没有一种几何学对自然形态描述得这样好。

1.5　分形是一种方法论

分形是一类形状，更是一种方法。从分形的角度看世界，我们可以发现很多未曾注意的现象与规则。正如沃尔夫奖（Wolf Prize）在颁发给分形理论创始人曼德勃罗时的评语所说的，“分形几何改变了我们对世界的看法”。

人类通常习惯于从时间和空间两个方面来考察事物。沿着时间轴，我们可以看到事物的发展状况；面对空间轴，我们可以看到事物的形态和分布状况。而分形方法是人类观察事物的第三轴，它是从纵深的角度看世界。

如果我们将被观察对象看成系统的话，而系统又是由多个层次组成的，那么分形就是研究这些层次之间关系的一种方法。

曼德勃罗在研究多年来棉价变化时，从貌似随机的波动中发现了周波动、月波动、年波动之间的相似性；他在研究海岸线时，也发现其标度变化下的不变性等。

现在看来，分形理论至少会在 3 个方面改变我们对世界的认识。首先，自然界中许多不规则的形态其背后都有规则，比如山、云、海岸线和河流分布等，都可以用分形的方法建立模型并在计算机上构造出以假乱真的景象来。显然利用这套方法，我们可以把世界压缩到几个分形规则中，便于携带和传播。其次，许多以前被认为是随机的现象，从分形理论的角度看并不是随机的，比如布朗运动、股票价格的波动以及传染病的流行

传播等，这为我们控制这些貌似随机的现象奠定了理论基础。最后，分形理论中的分维概念为我们认识世界中的复杂形态提供了一个新的尺度。复杂性科学是现代科学的前沿，在这门科学的研究过程中，发现了许多符合分形规则的复杂形态，而分维是测量这些形态复杂程度的一种度量。也就是说，我们找到了对复杂性做定量分析的工具。

当然，分形理论不止在这 3 个方面改变我们。分形作为一种方法论，它既是认识世界和改造世界的工具，也是一种新的思维方式。由于自然界中普遍存在某种程度的自相似性，使得我们有可能从局部认识整体，从有限认识无限，从瞬间认识永恒。不仅如此，许多现象表明分形已经对文化产生了重要影响，而且已被看成是一种新的艺术形式。

1.6　分形与计算机图形学

计算机图形学是计算机科学的一个重要分支，现已成为用户接口、数据可视化、虚拟现实、电视广告以及许多应用领域的基础。

传统的计算机图形学以欧氏几何学为数学基础，构造规则的几何图形。主要研究点、线、面等基本图素的绘制算法以及对图形的平移、放缩、剪裁、填充等编辑功能的实现。近年来，计算机图形学的发展越来越多地融入物理原理构造三维图形算法，从而更好地模拟物体和相关环境之间的复杂交互以及各种节奏和样式的动画。

分形理论的发展离不开计算机图形学的支持，一个分形构造的表达不借助计算机的帮助是很难让人理解的。不仅如此，分形算法与现有计算机图形学的其他算法相结合，还会产生出非常美丽的图形，而且可以构造出复杂纹理和复杂形状，从而产生非常逼真的物质形态和视觉效果。

也正是看重这一点，计算机图形学也将分形算法纳入其中，近年来出版的计算机图形学方面的图书，几乎都有一些与分形有关的章节。

分形作为一种方法，在图形学领域主要是利用迭代和递归等技术来实现某一具体的分形构造。

分形几何学与计算机图形学相结合，将会产生一门新的学科——分形图形学。它的主要任务是以分形几何学为数学基础，构造非规则的几何图素，从而实现分形体的可视化以及对自然景物的逼真模拟。

第 2 章　分形图的递归算法

侯世达定律：做事所花费的时间总是比你预期的要长，即使你的预期中考虑了侯世达定律。

——侯世达《哥德尔·埃舍尔·巴赫——集异璧之大成》

小时候听过一个最无聊的故事，可直到现在仍然记忆犹新。

故事是这样讲的：从前有座山，山里有个庙，庙里有个老和尚在给一个小和尚讲故事，老和尚说：“从前有座山，山里有个庙，庙里有个老和尚在给一个小和尚讲故事，老和尚说：‘从前有座山，山里有个庙，庙里有个老和尚在给一个小和尚讲故事，老和尚说：……’”

这不过是大人被小孩缠得再也没有什么故事好讲时，所抛出的最后一招。然而，现在想来，它却是一个典型的递归过程。在日常生活中，我们常常会遇到递归的例子。比如，画中画、电影中的主人公正在演电影以及俄罗斯的套娃等。甚至，经典的艺术作品中也能找到递归的例子，如普罗科菲耶夫的第五钢琴协奏曲有递归的旋律，更直观的还要数艾舍尔的版画《鱼与鳞》（图 2.1），鱼身上每一个鳞片都是一条小鱼。

图 2.1　艾舍尔的版画《鱼与鳞》

在计算机程序设计中，递归是指一个过程直接或间接地调用其自身的一种算法。

①直接递归调用的例子如下：

```
void Recur(n)
{
      ……
     Recur(m);
      ……
     }
```

过程 Recur 的内部又调用了自身——Recur 过程。

②间接递归调用的例子如下：

```
void Recur_A(n)
```

```
{
      ……
      Recur_B(m);
      ……
}

void Recur_B(n)
{
      ……
      Recur_C(m);
      ……
      }

void  Recur_C(n)
{
      ……
      Recur_A(m);
      ……
      }
```

过程 Recur_A 的内部调用了过程 Recur_B；过程 Recur_B 的内部调用了过程 Recur_C；过程 Recur_C 的内部调用了过程 Recur_A，这相当于过程 Recur_A 间接地调用了过程 Recur_A 自身。

实质上，递归是利用计算机中压栈和出栈的功能，重复地运用某些规则来生成嵌套的结构。这里所谓的“压栈”，意思是暂时停止目前进行的操作，但并没有把当前的信息忘掉，然后去完成更低一层次的任务；而“出栈”则正好相反，是结束在这个层次上的操作回到更高的层次上来，重新开始因为“压栈”而中断的操作。

分形的自我相似、自我复制和自我嵌套，让人很自然地想到可以用计算机的递归算法来生成分形图。事实也是如此，对那些经典分形图的绘制，大多可采用递归算法。

2.1　Cantor 三分集的递归算法

按照 Cantor 三分集的生成规则，我们可以用图 2.2 来表示在绘图空间中的生成关系。其中，（*ax*,*ay*）-（*bx*,*by*）为初始线段，（*ax*,*ay*）-（*cx*,*cy*）和（d*x*,d*y*）-（*bx*,*by*）为初始线段三等分后，去掉中间线段所剩下的两个线段。按照递归的想法，以后每一次操作都将遵循这一规则。

图 2.2　Cantor 三分集构造示意图

1. 算法步骤

算法步骤如下：

①如图 2.2 所示，给定初始直线两个端点的坐标（*ax*,*ay*）和（*bx*,*by*），按 Cantor 三分集的生成规则计算出各关键点的坐标如下：

$$cx = ax + (bx - ax)/3$$
$$cy = ay - \mathrm{d}$$
$$dx = bx - (bx - ax)/3$$
$$dy = by - \mathrm{d}$$
$$ay = ay - \mathrm{d}$$
$$by = by - \mathrm{d}$$

其中，d 为一个常量，代表上下两层线段之间的距离。

②利用递归算法，将计算出来的新点分别对应于（*ax*,*ay*）和（*bx*,*by*），即

$$ax \leftarrow ax，\ ay \leftarrow ay，\ bx \leftarrow cx，\ by \leftarrow cy$$
$$ax \leftarrow \mathrm{d}x，\ ay \leftarrow \mathrm{d}y，\ bx \leftarrow bx，\ by \leftarrow by$$

然后利用步骤①中的计算关系计算出下一级新点（*cx*,*cy*）和（d*x*,d*y*），并压入堆栈。

③给定一个小量 *c*，当（*bx* −*by*）< *c* 时，被压入堆栈中的值依次释放完毕，同时绘制直线段（*ax*,*ay*）-（*bx*,*by*），然后程序结束。

2. 算法说明

①Cantor 集是被分解到无穷时的一种状态，但无穷只存在于想象之中，计算机无法做到，所以算法中设置了一个小量 *c*，即用 *c* 做一个指标，当被分解后的线段小于 *c* 时便停止递归。

②算法步骤②中的变量传递是重要的，因为只有这样我们才能使规则传递下去，即将每一个子过程都按照与父过程相同的规则操作，从而实现递归。

③算法步骤①中的常量 d 是为了使 Cantor 集每层操作的结果不重叠，这样可以使我们看清 Cantor 集变化过程的结构，它是自相似的。

3. 程序设计

以下是绘制 Cantor 三分集分形图（图 2.3）的算法过程，该图形的完整程序可在本书所附光盘中找到。

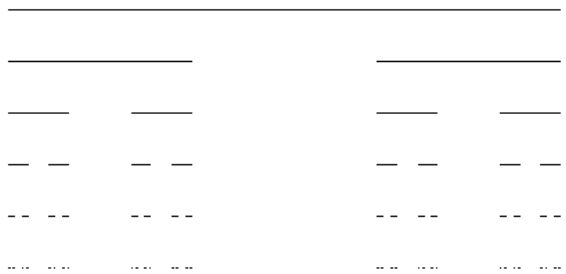

图 2.3　Cantor 三分集

程序 2.1　Cantor 三分集。

```
void CCantoView::canto(int ax, int ay, int bx, int by)
{
   CClientDC pDC(this);
   int c=10;// 三分集直线长度的最小精度，控制递归的深度

   if((bx-ax)<c)
   {
       pDC.MoveTo (ax,ay);//移到(ax,ay)
       pDC.LineTo (bx,by);//绘制(ax,ay)-(bx,by)的直线
   }
   else
   {
      int cx;
      int cy;
      int dx;
      int dy;

       pDC.MoveTo (ax,ay);
       pDC.LineTo (bx,by);

       cx = ax + (bx - ax) / 3;
       cy = ay + 50;
       dx = bx - (bx - ax) / 3;
       dy = by + 50;
       ay = ay + 50;
       by = by + 50;

       canto(ax, ay, cx, cy); //递归调用 canto()过程
       canto(dx, dy, bx, by); //递归调用 canto()过程
   }

}
```

4. 程序说明

程序说明如下：

①在实际编程中，你需要将此过程在下列过程中调用 canto()过程，才能绘出图来。

```
void CCantoView::OnDraw(CDC* pDC)
{
    ……
    canto(50,100,750,100);
}
```

②*c* 值越小，递归的层次越多，画出的线段也越多。但 *c* 值不能太小，因为递归的堆栈是很占内存空间的，*c* 过小容易产生溢出错误。

③由于 Cantor 三分集的规则是一段变成两段，所以在过程 CCantoView::canto()中用了两次递归调用，分别作为各自线段的递归。

④为了使 Cantor 三分集的每一层的线段都能看得见，则在程序设计时，不只是线段小于 *c* 值是才画图，而是每一次进入递归时都画图。

2.2　Koch 曲线的递归算法

图 2.4 是按照 Koch 曲线的生成规则设计的一个关系示意图，用来帮助建立 Koch 曲线的递归算法。其中，（*ax*,*ay*），（*bx*,*by*），（*cx*,*cy*），（*dx*,*dy*），（*ex*,*ey*）分别是各关键点在绘图区域中的坐标，*alpha* 为曲线隆起的角度，*L* 为曲线中每一线段的长度。

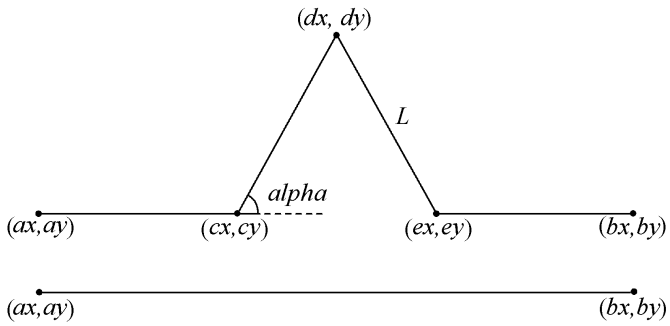

图 2.4　Koch 曲线示意图

1. 算法步骤

算法步骤如下：

①图 2.4 给定初始直线(*ax*,*ay*)-(*bx*,*by*), 按 Koch 曲线的构成原理计算出各关键点的坐标如下：

$$cx = ax + (bx - ax)/3$$
$$cy = ay + (by - ay)/3$$
$$ex = bx - (bx - ax)/3$$
$$ey = by - (by - ay)/3$$
$$L = \mathrm{Sqrt}((ex - cx)*(ex - cx) + (ey - cy)*(ey - cy))$$
$$alpha = \mathrm{atan}((ey - cy)/(ex - cx))$$
$$dx = cx + \cos(alpha + 3.14159/3)*L$$
$$dy = cy + \sin(alpha + 3.14159/3)*L$$

②利用递归算法，将计算出来的新点分别对应于（*ax*,*ay*）和（*bx*,*by*），即

$$ax \leftarrow ax，\ ay \leftarrow ay，\ bx \leftarrow cx，\ by \leftarrow cy$$
$$ax \leftarrow ex，\ ay \leftarrow ey，\ bx \leftarrow bx，\ by \leftarrow by$$
$$ax \leftarrow cx，\ ay \leftarrow cy，\ bx \leftarrow dx，\ by \leftarrow dy$$
$$ax \leftarrow dx，\ ay \leftarrow dy，\ bx \leftarrow ex，\ by \leftarrow ey$$

然后利用步骤①中的计算公式计算出下一级新点（*cx*,*cy*），（*dx*,*dy*），（*ex*,*ey*），并压入堆栈。

③给定一个小量 c，当 $L<c$ 时，被压入堆栈中的值依次释放完毕，同时绘制直线段 $(ax,ay)-(bx,by)$，然后程序结束。

2. 算法说明

算法说明如下：

①线段长度 L 由步骤①中的关系式给出，是考虑其递归过程的通用性。在整个的递归过程中，各个线段的方向是不一样的，所以线段端点的水平坐标和垂直坐标是变化的，因此，写成步骤①中的式子，是为了满足计算不同方向的线段长度的要求。

②在算法中，夹角 *alpha* 实际是一个绝对角度，即在整个坐标系中的角度。虽然在递归操作过程中，两条线段的相对夹角不变，但其绝对角度是变化的。所以，在算法中我们也是通过关系式计算出夹角 *alpha*，而不是给它一个固定值。

3. 程序设计

以下是绘制 Koch 曲线分形图（图 2.5）的算法过程，该图形的完整程序可在本书所附光盘中找到。

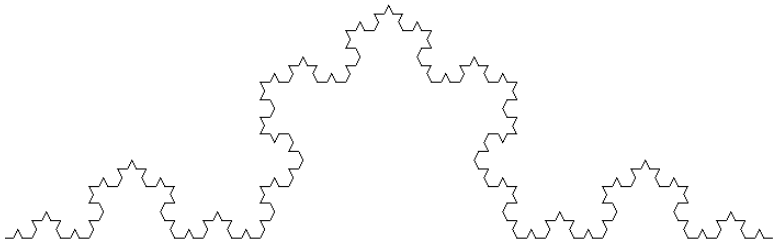

图 2.5　Koch 曲线

程序 2.2　Koch 曲线。

```
void CKochView::Fractal(double ax, double ay, double bx, double by)
 {
    CClientDC dc(this);
    int c=100; //控制 Koch 曲线的递归深度

    if(((bx-ax)*(bx-ax)+(by-ay)*(by-ay))<c)
    {
       dc.MoveTo(ax+100,600-ay);
       dc.LineTo(bx+100,600-by);
     }
      else
     {
         double cx;
         double cy;
         double dx;
         double dy;
         double ex;
         double ey;
```

```
            double l;
            double alfa;

            cx=ax+(bx-ax)/3;
            cy=ay+(by-ay)/3;
            ex=bx-(bx-ax)/3;
            ey=by-(by-ay)/3;

            Fractal(ax,ay,cx,cy);
            Fractal(ex,ey,bx,by);

            l=sqrt((ex-cx)*(ex-cx)+(ey-cy)*(ey-cy));
            alfa=atan((ey-cy)/(ex-cx));

            if((alfa>=0 && (ex-cx)<0) || (alfa<0 && (ex-cx)<0))
            alfa=alfa+PI;

            dx=cx+cos(alfa+PI/3)*l;
            dy=cy+sin(alfa+PI/3)*l;
            Fractal(cx,cy,dx,dy);
            Fractal(dx,dy,ex,ey);
        }
    }
```

4. 程序说明

程序说明如下:

①要想使用本过程，需要在下列程序段调用。

```
void CKochView::OnDraw(CDC* pDC)
{
   ……
   Fractal(10,300,790,300);
  }
```

另外，程序的开始定义了 PI 值：

```
#define PI 3.1415926
```

这样后面的程序便可用 PI 了。

②此程序与程序 2.1 有所不同，程序 2.1 要画出 Cantor 三分集每一层次的结构，所以，它的绘图语句要放在 if 语句的 else 中。而此程序是绘制 Koch 曲线递归的最后一次结果，所以，它的绘图语句要放在 if 语句的满足条件结构中。

③由于 Koch 曲线（图 2.5）的规则是一段变成四段，所以在过程 Sub Koch()中用了 4 次递归调用，分别作为各自线段的递归。

2.3　Koch 雪花的递归算法

你已经看到，Koch 曲线的初始图元是直线，但最终结果却是一个参差不齐的曲线，很像雪花的边缘，我们不妨将 3 条这样的曲线围在一起，便会得到 Koch 雪花的图形。这样，初始图元就不能是一条直线了，而是一个三角形。

至于算法，我们可以利用 2.2 节介绍的 Koch 曲线递归算法，只要在程序设计中调用 3 次 Koch 递归过程，以实现三角形 3 条边各自的 Koch 递归生成。

程序 2.3　Koch 雪花。

```
void CSnowView::OnDraw(CDC* pDC)
{
    ……
    fractal(180, 120, 360, 432);
    fractal(360, 432, 540, 120);
    fractal(540, 120, 180, 120);
}
```

其中，该程序算法部分与程序 2.2 中的 Koch 曲线算法相同。

图 2.6 绘出了不同递归深度所表现出的 Koch 雪花图案。

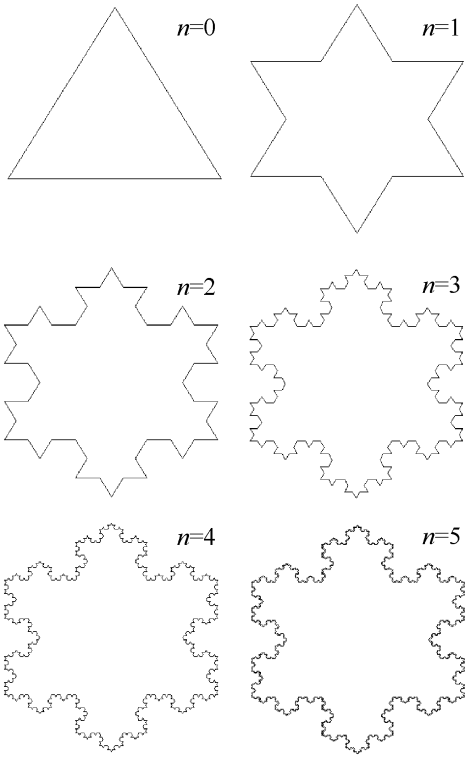

图 2.6　Koch 雪花

2.4　Arboresent 肺的递归算法

Arboresent 肺（图 2.7）也可以看成是一种 Koch 曲线，开始时是两个顶角为 $90^\circ+\varepsilon$ 的等腰三角形，它们有一个公共的锐角顶点，从这个顶点出发，两个三角形长边张成 2ε

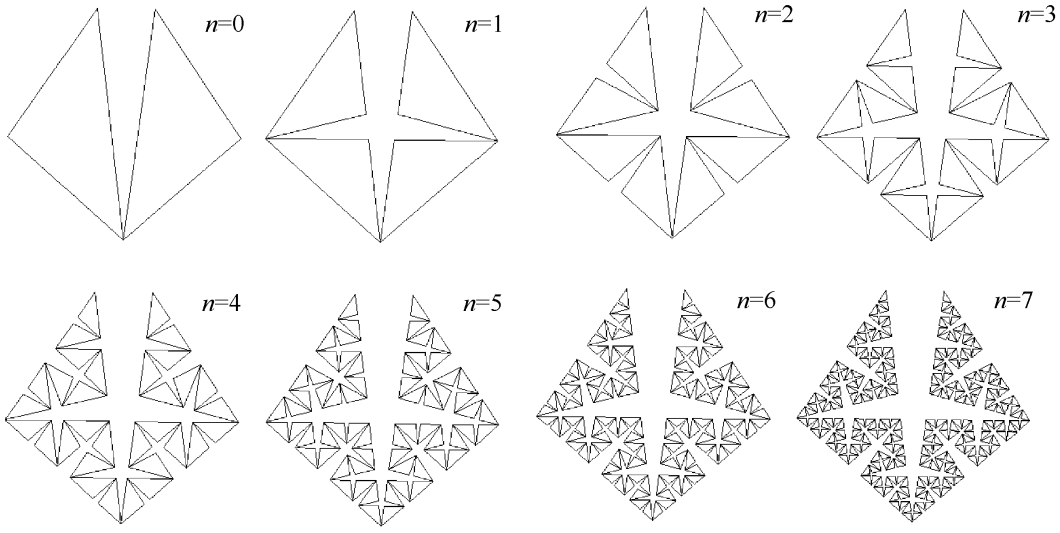

图 2.7　Arboresent 肺

的角，它的第一步操作是把每个三角形又分成两个顶角为 $90^{\circ}+\varepsilon$ 的小锐角等腰三角形，中间剪出角度为 2ε 的一个裂口并以此类推。

在窗体上放置两个按钮控件，一个为“生成”，另一个为“退出”。

程序 2.4　Arboresent 肺。

```
// Arboresent1.cpp : implementation file
void CArboresent::Draw(CDC *pDC)
{
   float prate;
   float theta;
   float alfa;
   float t;
   int level;
   float size;

   float x;
   float y;

   prate = 1.5;
   theta = atan(sqrt(4 / prate / prate - 1));
   t = pi - 4 * theta;
   alfa = 55 * pi / 180;
   level = 7;
   x = -2500;
   y = 0;
   size = 3000;

   m_pDC = pDC;
   DrawArboresent (x, y, size, prate, alfa, theta, t, level);
   DrawArboresent (x + size * prate * cos(alfa - theta - t),
                   y + size * prate * sin(alfa - theta - t),
                   size,prate,alfa+2*theta-t,-theta,-t,level);
   m_pDC = NULL;
}

void CArboresent::DrawArboresent(float x, float y,
                                 float size, float prate,
                                 float alfa, float theta,
                                 float t, float level)
{
   #define L_X 200
   #define L_Y 230
```

```
    float xe, ye;
    float xa, ya;
    float xb, yb;
    float xc, yc;
    float xd, yd;
    float l;

    l = size / prate;
    xe = x;
    ye = y;
    xa = x + size * cos(alfa);
    ya = y + size * sin(alfa);
    xb = x + l * cos(alfa - theta);
    yb = y + l * sin(alfa - theta);
    xc = x + l * cos(alfa - theta - t);
    yc = y + l * sin(alfa - theta - t);
    xd = x + size * cos(alfa - theta * 2 - t);
    yd = y + size * sin(alfa - theta * 2 - t);

    if (level <= 1)
    {
       m_pDC->MoveTo(L_X-xa/15, L_Y-ya/15);
       m_pDC->LineTo(L_X-xe/15, L_Y-ye/15);
       m_pDC->LineTo(L_X-xd/15, L_Y-yd/15);
       m_pDC->LineTo(L_X-xc/15, L_Y-yc/15);
       m_pDC->LineTo(L_X-xe/15, L_Y-ye/15);
       m_pDC->LineTo(L_X-xb/15, L_Y-yb/15);
       m_pDC->LineTo(L_X-xa/15, L_Y-ya/15);
       return;
    }

     DrawArboresent (xb, yb, l, prate, alfa - theta + pi,
                     theta, t, level - 1);
     DrawArboresent (xc, yc, l, prate, alfa - theta - t + pi,
                     -theta, -t, level - 1);
}
```

2.5　Sierpinski 垫片的递归算法

按照第 1 章中给出的 Sierpinski 垫片的生成规则，借用递归算法是比较容易实现的。但在具体设计算法时，还可以有多种不同的方案，这些方案各有优缺点。

2.5.1　算法一

该算法的步骤如下所示：

①如图 2.8 所示，给出初始三角形中心点的坐标（x,y）和三角形的边长 L，便可计算出三角形的 3 个顶点的坐标，并画出初始三角形。

$$x_1 = x - L/2, \qquad y_1 = y - L*\tan(3.14159/6)/2$$
$$x_2 = x + L/2, \qquad y_2 = y - L*\tan(3.14159/6)/2$$
$$x_3 = x, \qquad y_3 = y + L*\tan(3.14159/6)$$

②根据△*ABC* 中心点的坐标（x,y）计算出各边中点的坐标，并画出 3 条边中点之间的连线，从而形成 4 个小三角形△*ADE*, △*BEF*, △*CDF*, △*DEF*。

③设定递归深度，利用递归算法，分别对△*ADE*, △*BEF*, △*CDF* 执行步骤②，每次执行边长为上一步的一半，即 *L*=*L*/2，并压入堆栈，同时使递归深度值减 1，直到递归深度值等于 1 时，释放堆栈，并画出相应的三角形。

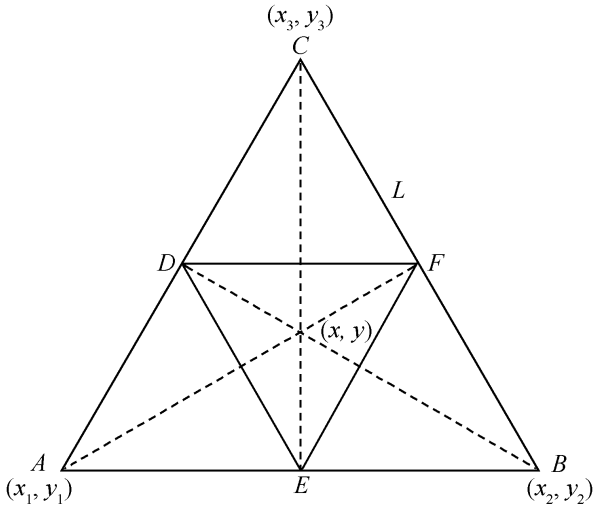

图 2.8　Sierpinski 垫片示意图

1. 算法说明

算法的说明如下所示：

①此算法一大优点是可以构造正三角形的 Sierpinski 垫片图形，但这也是它的弱点，即不能构造任意三角形的 Sierpinski 垫片图形。

②此算法采用的是设置递归深度变量初值的方法来控制其递归深度，这样可以更好地掌握递归的层次数。

2. 程序设计

在窗体上插入两个按钮控件 IDOK 和 IDCANCEL 分别作为“绘图”按钮和“退出”按钮。

程序 2.5　Sierpinski 垫片。

```
BOOL CSierpinskiDlg::OnInitDialog()
```

```
{
    CDialog::OnInitDialog();

    // Add "About..." menu item to system menu.

    // IDM_ABOUTBOX must be in the system command range.
    ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
    ASSERT(IDM_ABOUTBOX < 0xF000);

    CMenu* pSysMenu = GetSystemMenu(FALSE);
    if (pSysMenu != NULL)
    {
        CString strAboutMenu;
        strAboutMenu.LoadString(IDS_ABOUTBOX);
        if (!strAboutMenu.IsEmpty())
        {
            pSysMenu->AppendMenu(MF_SEPARATOR);
            pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX,
                                                strAboutMenu);
        }
    }

    // Set the icon for this dialog.
    //The framework does this automatically
    //  when the application's main window is not a dialog
    SetIcon(m_hIcon, TRUE);     // Set big icon
    SetIcon(m_hIcon, FALSE);    // Set small icon

    // TODO: Add extra initialization here

    CRect rt;
    GetClientRect(rt);
    x = rt.Width() / 2;
    y = rt.Height() / 2;
    l = 300;
    inti = 0;

    return TRUE;  // return TRUE  unless you set the focus to a control
}

void CSierpinskiDlg::OnPaint()
{
    if (IsIconic())
```

```
    {
        CPaintDC dc(this); // device context for painting

       SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

        // Center icon in client rectangle
        int cxIcon = GetSystemMetrics(SM_CXICON);
        int cyIcon = GetSystemMetrics(SM_CYICON);
        CRect rect;
        GetClientRect(&rect);
        int x = (rect.Width() - cxIcon + 1) / 2;
        int y = (rect.Height() - cyIcon + 1) / 2;

        // Draw the icon
        dc.DrawIcon(x, y, m_hIcon);
}
else
{
        //绘制初始三角形
        CClientDC dc(this);
        CPen pen(PS_SOLID,1,RGB(0,0,0));
        dc.SelectObject(&pen);

        dc.MoveTo (int(x - l / 2), int(y + l * tan(PI / 6) / 2));
        dc.LineTo (int(x + l / 2), int(y + l * tan(PI / 6) / 2));

        dc.MoveTo (int(x + l / 2), int(y + l * tan(PI / 6) / 2));
        dc.LineTo (x, int(y - l * tan(PI / 6)));

        dc.MoveTo (x, int(y - l * tan(PI / 6)));
        dc.LineTo (int(x - l / 2), int(y + l * tan(PI / 6) / 2));
        CDialog::OnPaint();
    }
}

void CSierpinskiDlg::OnOK()
{
    // TODO: Add extra validation here
    inti++;
    if(inti == 1 )//绘制第一级中点连成的三角形
    {
        CClientDC dc(this);
        l = l / 2;
```

```
            dc.MoveTo (int(x - l / 2), int(y - l * tan(PI / 6) / 2));
            dc.LineTo (int(x + l / 2), int(y - l * tan(PI / 6) / 2));

            dc.MoveTo (int(x + l / 2), int(y - l * tan(PI / 6) / 2));
            dc.LineTo  (x,             int(y + l * tan(PI / 6)));

            dc.MoveTo (x,int(y + l * tan(PI / 6)));
            dc.LineTo (int(x - l / 2), int(y - l * tan(PI / 6) / 2));
        }
        else//如果不是第一级，则调用 Sierpinski 垫片递归过程
        {
        sierpinski(x, y, l, inti);
        }

         //CDialog::OnOK();
    }

    void CSierpinskiDlg::sierpinski(int x,int y,int l,int i)
    {
        if( 1 == i )
        {
            CClientDC dc(this);
            dc.MoveTo (int(x - l / 2), int(y - l * tan(PI / 6) / 2));
            dc.LineTo (int(x + l / 2), int(y - l * tan(PI / 6) / 2));

            dc.MoveTo (int(x + l / 2), int(y - l * tan(PI / 6) / 2));
            dc.LineTo (x,              int(y + l * tan(PI / 6)));

            dc.MoveTo (x,int(y + l * tan(PI / 6)));
            dc.LineTo (int(x - l / 2),int(y - l * tan(PI / 6) / 2));
        }
        else
        {
            for(int j=0; j<=2; j++)
            {
               sierpinski(int(x + sqrt(3)* l * sin(j * 2 * PI / 3) / 3),
                          int(y - sqrt(3)* l * cos(j * 2 * PI / 3) / 3),
                          int(l / 2),
                          i - 1);
            }
        }
    }
```

另外，在头文件 SierpinskiDlg.h 中，进行了如下定义：