# APPROXIMATION AND VISUALIZATION OF SETS DEFINED BY ITERATED FUNCTION SYSTEMS

A Thesis
Submitted to the Faculty of Graduate Studies and Research
In Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in
Computer Science
Faculty of Science
University of Regina

By
Daryl H. Hepting
Regina, Saskatchewan
March, 1991

# Abstract

An iterated function system (IFS) is defined to be a set of contractive affine transformations. When iterated, these transformations define a closed set, called the attractor of an IFS, which has fractal characteristics. Fractals of any sort are currently a topic of great popular appeal, largely due to the exciting images to which they lend themselves. Iterated function systems represent one of the newest sources of fractal images. Research to date has focused on exploiting IFS techniques for the generation of fractals and for use in modelling applications. Both areas of this research are well suited to computer graphics, and this thesis examine the IFS techniques from a computer graphics perspective.

As a source of fractals, iterated function systems have some relationship to other methods of fractal generation. In particular, the relationship between IFS attractors and Julia sets will be examined throughout the thesis. Many insights can be gained from the previous work done by Peitgen, Richter and Saupe [32, 33] both in terms of methods for the generation of the fractal sets and methods for their visualization. The differences between the linear transformations which compose an IFS and the quadratic polynomials which define Julia sets are significant, but not moreso than their similarities.

This thesis deals with the related questions of approximation and visualization. The method of constructing the approximating set of points is dependent upon the visualization method in use. Methods have been developed both to visualize the attractor and its complement. The two techniques used to examine the complement set are based on the distance and escape-time functions.

The modelling power of standard IFS techniques is limited in that they cannot be

used to model any object which is not strictly self-affine. To combat this, methods for controlling transformation application are examined which allow objects without strict self-affinity to be modelled.

As part of this research, an extensible software system was developed to allow experimentation with the various concepts discussed. A description of that system is included in Chapter 6.

# Acknowledgments

I would like to thank my supervisor, Dr. P. Prusinkiewicz for guiding me along many interesting paths during my studies. I would also like to thank my thesis committee members: Dr. Alan G. Law, especially for providing so many opportunities for building my character; Dr. J. Chris Fisher for his patience in helping to develop some geometrical intuition in a computer science student; and the external examiner, Dr. Harley Weston, for his insightful comments on this thesis. Although Dr. Dietmar Saupe was not an official committee member, I benefitted greatly from our collaboration, including the time spent at the University of Bremen.

I would like to acknowledge the support of the Natural Sciences and Engineering Research Council through its postgraduate scholarship program and the Faculty of Graduate Studies and Research for its support in providing a Teaching Fellowship.

Working in the graphics lab at the University of Regina was not a solitary pursuit and I want to acknowledge Dave Fracchia, Rob Morris, Allan Snider, and Richard Smith for making the long hours of work less painful. Dr. Norma Fuller was a great help, especially in preparing some of the illustrations for this thesis. I cannot forget Jim Hanan, Debbie Fowler, and Lynn Mercer with whom I have shared many interesting experiences as members of the Graphics Group.

I would like to thank Ken Musgrave and Craig Kolb for being my "friends at Yale". Closer to home, I want to thank Nalin Wijesinghe, with whom I have always shared "office" space; Judy Chapman, a fellow graduate student with whom I have shared many of these trials and tribulations; the Computer Science department that has been my home; and those friends that I still have outside of the university.

Finally, I would like to thank my parents for all their love and support, without

which none of this would have been possible.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

The topic of iterated function systems (IFS) gained a great deal of popularity with computer graphicists following the work of Barnsley and Demko [4, 11]. This topic continues to receive a great deal of research partly because of the intricate shapes which they define and partly because of the potential that they represent as a solution for many modelling problems. An IFS is simply a set of transformations which shrink the space around them. When these transformations are iterated, a closed set with fractal characteristics called the *attractor of the IFS* results. Some of the theory involved with iterated function systems was presented as early as 1971 by Williams [45] and formalized by Hutchinson [23] in 1981 but the concepts of function iteration go back to the turn of the century. As such, there exists a strong connection between IFS attractors and Julia sets, which are fractal sets arising from iteration of quadratic polynomials on the complex plane.

Mandelbrot gave a "tentative" definition for fractals as those sets whose Hausdorff dimension is strictly greater than their topological one. There have been other definitions for fractal, but those which rely only on a single criterion will inevitably leave out many examples which should be considered as fractals. As Falconer [16] states:

> My personal feeling is that the definition of a 'fractal' should be regarded in the same way as the biologist regards the definition of 'life'.

> There is no hard and fast definition, but just a list of properties characteristic of a living thing, ... Most living things have most of the characteristics on the list, though there are living objects that are exceptions to each of them. ...
>
> When we refer to a set $F$ as a fractal, therefore, we will typically have the following in mind.
>
> (i) $F$ has a fine structure, i.e. detail on arbitrarily small scales.
>
> (ii) $F$ is too irregular to be described in traditional geometric language, both locally and globally.
>
> (iii) Often $F$ has some form of self-similarity, perhaps approximate or statistical.
>
> (iv) Usually, the 'fractal dimension' of $F$ (defined in some way) is greater than its topological dimension.
>
> (v) In most cases of interest $F$ is defined in a very simple way, perhaps recursively.

It is not always clear whether a square defined by an iterated function system is a fractal. Rather, this thesis is concerned with algorithms to generate the attractor efficiently and methods to visualize the attractor and its complement which will provide information about its structure. In both respects, a great deal can be gained from the computer graphical study of Julia sets undertaken first by Peitgen and Richter at the University of Bremen, described in the book *The Beauty of Fractals* [32]. The relationship between Julia sets and IFS attractors suggests that it would be possible to develop visualization techniques for IFS's analogous to those which already exist for Julia sets. Work in this direction was begun by Barnsley [2] and Prusinkiewicz [37] and continued in the paper by Hepting et al. [22]. Important differences do exist, stemming from the non-linearity of the maps used in Julia sets, but they do not detract from the insights which can be gained by study of both.

The work presented here is part of a continuing research effort at the University of Regina in the study of fractals with computer graphical techniques. Some of the

results in the areas of approximation, escape-time, and control of transformation application represent continuations of the work done by Sandness [40] whereas the area of distance computation is a new focus of study.

The thesis begins in Chapter 2 with a discussion of methods to approximate the attractor of an IFS. The challenge is to generate the approximation both efficiently and accurately. Several variations of techniques for approximating an attractor have been put forward in the literature. Barnsley proposed a stochastic algorithm [4] in which transformations are applied based on associated probabilities. A deterministic approach was presented by Reuter [39] and Prusinkiewicz [37], which represented an extension to computer graphics of Hutchinson's method [23]. Dubuc [13] presented two additional methods which are based on a criterion for measuring the accuracy of the approximation. The first of these methods closely resembles the **Adaptive-cut** method first presented in the paper by Hepting et al. [22]. Dubuc's second method includes additional tests at the screen level to improve the efficiency of the approximation. The method presented by Williams [45] differs from all of others in that it takes the route of direct computation of a representative set of points of the attractor, rather than to merely approximate such a set. The **Adaptive-cut** method will be contrasted to the stochastic and other deterministic methods. An interesting by-product of the **Adaptive-cut** method in three-dimensions is the ability to produce concise models for purposes of ray-tracing.

The methods for approximating the attractor do not provide any information about points in the complement of the attractor. It has been demonstrated [32, 33] that intriguing images of Julia sets may be obtained by making use of the points which lie outside of the attractor. Therefore, in addition to the methods designed to model the attractor $\mathcal{A}$, it is possible to develop methods which examine its complement $X \setminus \mathcal{A}$. The two methods presented here are based on the distance and escape-time functions.

Chapter 3 discusses the question of distance calculation. The ability to construct an accurate approximation of a set is essential for the effective computation of the distance from that set. Several algorithms for computing distance are reviewed. The problem of finding the point in the attractor closest to an arbitrary point has been

studied widely in a much larger context and can be considered to be part of the class of "range searching" problems. Specifically, methods which use the structure of the attractor to reduce the number of computations required will receive a great deal of attention. It is interesting to note that there also exists a distance estimator method for Julia sets. Because it is only an estimate, it is not as accurate as the calculations performed for IFS's.

Chapter 4 discusses the escape-time function. The escape-time function can be defined for IFS as well. A discrete function has been introduced by Prusinkiewicz [37] and Barnsley [2]. A continuous extension of this function was made in the paper by Hepting et al. [22]. This version of the escape-time function differs from that for Julia sets in that it lacks meaning as a potential function. Regardless, the escape-time function for IFS has proven to be a valuable tool for the study of attractors.

Barnsley introduced the idea of addresses for points in attractors, according to the sequence of transformations applied to determine its location. Although the distance and escape-time functions do not model the attractor directly, both of these functions return values associated with a point in the attractor approximation. In this sense, points in the complement of the attractor also receive a coding, which may or may not be unique, and this information may be used to gain further insights into the complement set of the attractor.

Chapter 5 discusses ways to control transformation application. Control may be applied using either deterministic or stochastic methods, which are similar but not identical. The desire to attempt control is to allow IFS to represent more complex structures than otherwise possible. Particularly, one wishes to break the strict self-affinity which is present in standard IFS attractors. The ability to represent such complex structures with IFS enhances the modelling capabilities of IFS techniques. The Black Spleenwort fern [2] shows the possibility of using IFS for the modelling of natural structures. Further indications of this come from the relationship between Lindenmayer systems, fractals and plants. Some aspects of this relationship were analysed by Prusinkiewicz [36] to enhance the understanding of the fractal geometry of nature [29].

In order to experiment with the concepts introduced throughout the thesis, an

extensible software system has been developed, which is described in Chapter 6. It also represents an extension of Sandness' work in developing an environment in which transformations and their interpretation can be easily specified.

Traditionally, problems in computer graphics have aspects of both modelling and rendering. Although rendering is a consideration when dealing with problems of visualization, the majority of time is given to the study and improvement of algorithms dealing strictly with the problems of modelling. As a study in the area of computer graphics, considerable effort has been placed in the generation of images. However, the goal has always been to combine the aesthetics of the final image with an understanding of the underlying mathematics of fractals.

# Chapter 2

# Approximating the Attractor

The study of algorithms and equations for generating sets with fractal characteristics has been a topic of study since the turn of the century. The close relationship between Koch constructions and IFS attractors was demonstrated by Prusinkiewicz [37]. There is also a strong relationship between Julia sets and IFS attractors which shall be exploited throughout the thesis.

The intricate sets described in this thesis exist in a structured environment, called a complete metric space. The following definitions [26, 43, 2] outline the properties of such spaces.

## 2.1 Background

**Definition 2.1 (Metric space)** *Let $X$ be a set and let $d$ be a nonnegative real-valued function defined on $X \times X$ that satisfies the following conditions: For all $x$, $y$, and $z$ in $X$,*

- $d(x, y) = 0$ *if and only if* $x = y$,
- $d(x, y) = d(y, x)$,
- $d(x, y) + d(y, z) \geq d(x, z)$ *(triangle inequality).*

*The function $d$ is called a distance function or a metric for $X$ and $(X, d)$ is called a metric space.*

The metric space familiar to most people is the $R^2$, the plane, with the Euclidean distance function, which is defined as:

$$d(x,y) = \sqrt{(y_1 - x_1)^2 + (y_2 - x_2)^2}.$$

However, the Euclidean distance function is not the only possible choice for a metric on this space. A somewhat less familiar, yet equally valid, choice for a metric on $R^2$ is the so-called Manhattan distance function, defined as:

$$d(x,y) = \mid y_1 - x_1 \mid + \mid y_2 - x_2 \mid .$$

**Definition 2.2 (Equivalent metrics)** *Two metrics $d_1$ and $d_2$ on a space $X$ are said to be equivalent if there exist constants $0 < c_1 < c_2 < \infty$ such that*

$$c_1 d_1(x,y) \leq d_2(x,y) \leq c_2 d_1(x,y) \ \forall \ (x,y) \in X \times X$$

When $d_1$ is the Euclidean metric and $d_2$ is the Manhattan metric, it is a routine exercise to show that these are equivalent using $c_1 = 1$ and $c_2 = 2$.

**Definition 2.3 (Equivalent metric spaces)** *Two metric spaces $(X_1, d_1)$ and $(X_2, d_2)$ are equivalent if there is a function $h : X_1 \mapsto X_2$ which is one-to-one and onto such that the metric $\tilde{d_1}$ on $X_1$ defined by*

$$\tilde{d_1}(x,y) = d_2(h(x), h(y)) \ \forall \ x, y \in X_1$$

*is equivalent to $d_1$.*

**Definition 2.4 (Cauchy sequence)** *A sequence $\{a_n\}$ in a metric space $(X,d)$ is called a Cauchy sequence if, for any given number $\varepsilon > 0$, there is an integer $N > 0$ such that for all $n, m > N$,*

$$d(a_n, a_m) < \varepsilon$$

A metric space in which Cauchy sequences converge is called a *complete* metric space. The $n$-dimensional real space with either the Euclidean or Manhattan distance function is an example of a complete metric space. This thesis will study the iteration of affine transformations within these spaces.

**Definition 2.5 (Affinity)** *An affinity is any transformation from a linear space to itself preserving collinearity. An affinity transforms parallel lines into parallel lines, and preserves ratios of distances along parallel lines.*

Each affinity is the product of a linear transformation and a translation. The distinction between the two components of an affinity is made on the basis of the following definition [1].

**Definition 2.6 (Linear transformation)** *If $T$ is a function on a vector space, it is called a linear transformation if*

1. *$T(\vec{u}+\vec{v}) = T(\vec{u}) + T(\vec{v})$ for all vectors $\vec{u}$ and $\vec{v}$ in the space.*

2. *$T(k\vec{u}) = kT(\vec{u})$ for all vectors $\vec{u}$ and all scalars $k$.*

For geometric investigations, a magnitude ( either a "length" or "norm") and a direction are employed for a vector. In many applications, there is an underlying norm which can be used to induce a metric:

$$d(x, y) = \| x - y \| .$$

Two vectors are equal if they have the same magnitude and direction. However, two points are equal only if they coincide. Translation then is an important component of an affine transformation. In order to include the translation as part of a general affine transformation, $R^n$ is imbedded in $R^{n+1}$ by a restriction of homogeneous coordinates [18]. In $R^2$, a 3x3 matrix is used to represent the transformations and a row vector is used to represent the points. The notation $T(x) = xT$ is used to represent application of the transformation $T$ to the point $x$. The following describes the process in more detail.

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} a & c & 0 \\ b & d & 0 \\ l & m & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0.290 & 0.155 & 0.000 \\ -0.078 & 0.580 & 0.000 \\ 1.000 & 0.000 & 1.000 \end{bmatrix}$$

Table 2.1: Matrix for an affinity which rotates by 15 degrees, scales by 0.3 in 'x' and 0.6 in 'y' and translates by 1 along the 'x' axis.

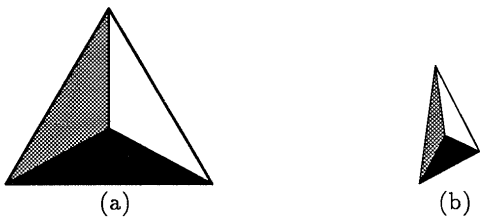



Figure 2.1: Application of the affinity given in Table 2.1 to the triangle in (a) will result in the triangle shown in (b).

Here, the parameters {a, b, c, d} represent the linear transformation and coefficients {l, m} represent the translation. In this example, the transformation maps the *preimage* $(x, y, 1)$, to the *image* $(x', y', 1)$, which is determined by the following formula.

$$\begin{aligned} x' &= ax + by + l \\ y' &= cx + dy + m \end{aligned}$$

An affinity is uniquely determined by its effect on any one triangle. Figure 2.1 shows the effect of applying the affinity given in Table 2.1.

Just as it is possible to measure the magnitude of a vector by computing its norm, one can measure the corresponding properties for a matrix. One very useful way of generating norms for matrices $A$ is by means of norms on vectors. In fact,

$$\| A \| \stackrel{\triangle}{=} max_{\|\vec{x}\|=1} \| \vec{x}A \|$$

defines a matrix norm and it is said to be *subordinate* to the corresponding vector

norm [20]. Construction of this pairing provides an exceptionally useful inequality:

$$\| \vec{x}A - \vec{y}A \| \leq \| A \| \| \vec{x} - \vec{y} \|$$

for any vectors $\vec{x}$ and $\vec{y}$. Thus, for any subordinate matrix norm, $\| A \|$ serves as the *Lipschitz constant* for the affine transformation $A$. It may be shown that the value of the matrix norm which is subordinate to the Manhattan norm for vectors [7] can be computed as

$$\| A \| = max_{1 \leq j \leq n} \sum_{i=1}^{n} | a_{ij} | .$$

for any $n^{\text{th}}$ order matrix $A$. Similarly, a matrix norm which is subordinate to the Euclidean norm for vectors, often called the *spectral norm* [20, 27], can be computed as

$$\| A \| = (\text{maximum eigenvalue of } A^T A)^{\frac{1}{2}} \tag{2.1}$$

If the Lipschitz constant for a transformation is strictly less than 1, the transformation is said to be contractive. A collection of contractive affine transformations has special meaning in this thesis, as determined by Definition 2.7.

**Definition 2.7 (Iterated function system)** *An iterated function system (IFS) is a finite set, denoted by $\mathcal{T}$, of contractive affine transformations.*

In order to provide further insight into construction of fractal sets using iterated function systems, it is instructive to examine analogous concepts for Julia sets.

## 2.2 Julia sets

When considering images of fractals, one is most likely to recall images of Julia sets. Notably different from IFS's, Julia sets are defined in terms of a single quadratic polynomial:

$$f_c : C \mapsto C, \; f_c(z) = z^2 + c, \; c \in C. \tag{2.2}$$

The action of the transformation varies depending upon the location of the preimage. Each equation of this form has two finite solutions which are the fixed points of the equation [33]. Each fixed point $f_c(z) = z$ can be classified as one of the following types:

- $z$ is attracting if $\mid f_c'(z) \mid < 1$.
- $z$ is repelling if $\mid f_c'(z) \mid > 1$.
- $z$ is indifferent if $f_c'(z) = e^{2\pi i\theta}$.

Initial values of $c$ which are close to attracting fixed points will tend towards infinity under iteration. The set of all points whose orbits tend to infinity is called the basin of attraction for infinity.

$$A_c(\infty) = \{z_0 \in C : f_c^k(z_0) \to \infty \text{ as } k \to \infty\}$$

The Julia set $J_c$ for a particular $f_c$ is defined to be the boundary of the corresponding set $A_c(\infty)$. The set $K_c$ as the complement to $A(\infty)$, sometimes called the filled-in Julia set, is defined to be the set of points whose orbits remain bounded. If $z_0$ is an arbitrary point in $J_c$ then

$$J_c = closure\{z : f_c^k(z) = z_0 \text{ for some integer } k\}.$$

The Julia set for a particular $f_c$ can be approximated by starting with the repelling fixed point $u_0$ and iterating the function $f_c$. This method is called the Inverse Iteration Method (IIM) [33]. A sample of a Julia set generated using the IIM method is shown in Figure 2.2(a).

In the study of Julia sets in relation to iterated function systems, it is sometimes useful to adopt an alternative form of the original quadratic equation in terms of the pair of non-linear transformations:

$$\begin{aligned} F_{1_c} &= +\sqrt{u-c} \\ F_{2_c} &= -\sqrt{u-c}. \end{aligned}$$

## 2.3 IFS attractors

An IFS may contain several distinct affine transformations, the iteration of which involves the operation of matrix multiplication. Although the procedure is straightforward in itself, the order in which the matrices are multiplied is important since the

multiplication is generally not commutative. The ideas are clarified in the following definition:

**Definition 2.8 (Composite transformation)** *A composite transformation is the product of any number of elements from $\mathcal{T}$. The elements from $\mathcal{T}$ may be identified by their indices. The order of elements in the composite transformation indicates the order of multiplication, from left to right. Hence, $xT_1T_2T_3$ corresponds to $T_3(T_2(T_1(x)))$.*

The notion of the attractor of an IFS mentioned earlier is made more precise in the following definition from Hutchinson [23]. Figure 2.2(b) shows an example of the attractor of an IFS.

**Definition 2.9 (Attractor)** *Let $X = (X, d)$ be a complete metric space. Let $\mathcal{T} = (T_1, T_2, \ldots, T_n)$ be a finite set of contraction maps on $X$. Under these conditions, there will exist a unique closed and bounded set, $\mathcal{A}$ called the attractor of the IFS $\mathcal{T}$, such that $\mathcal{A} = \bigcup \mathcal{A}T_i$.*

The set $\mathcal{A}$ has the following properties:

- $\mathcal{A}$ is the closure of the set of fixed points of finite composition of members of $\mathcal{T}$.
- $\mathcal{A}$ is invariant with respect to $\mathcal{T}$.
- $\mathcal{A}$ is the union of limit points from all sequences constructed from $\mathcal{T}$.

**Definition 2.10 (Compactness)** *A subset $S$ of $X$ is compact if and only if it is closed and bounded.*

The common characterization of the attractors in both the linear and non-linear cases as collections of fixed points represents a connection between the iterated function systems and Julia sets.

Each composite transformation has an associated, not necessarily unique, string of transformations which can be used to identify it. Barnsley [2] has formalized this notion into a scheme for addressing points in the attractor. For an IFS with $N$ transformations, there is an associated *code space* $\sigma$ on $N$ symbols. Addresses of

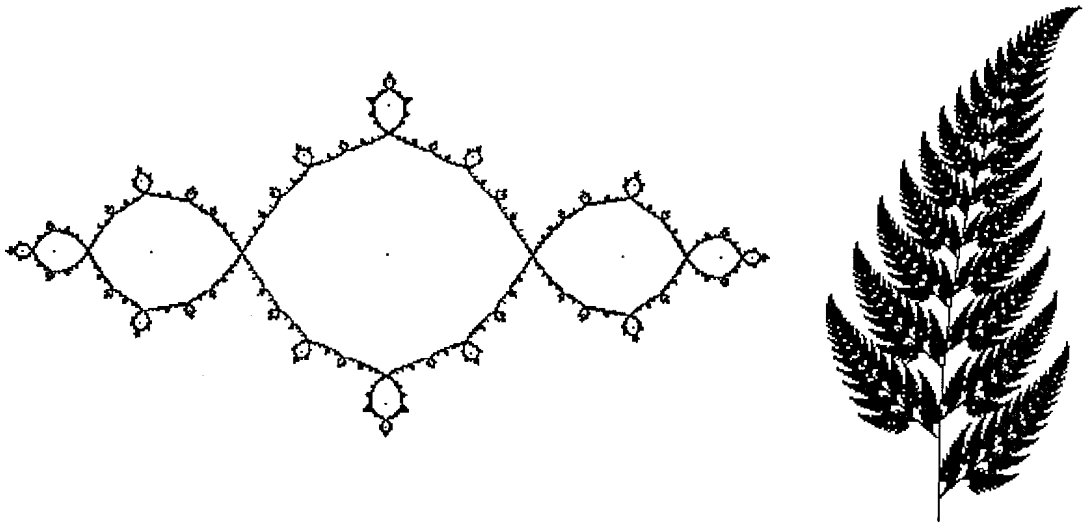



Figure 2.2: The Julia set corresponding to $z^2 - 1$ is shown in (a) and the attractor of the Black Spleenwort fern IFS given in Table 2.4 is shown in (b).

---

points in the attractor are determined by a continuous function which maps elements of the code space to the attractor. It is determined by the sequence of transformations, applied to the preimage, required to reach the point in question. This idea of addresses can be used to categorize three different types of attractors [2, Chapter 4], which are illustrated in Figure 2.4.

**Definition 2.11 (Attractor types)** *An attractor can be classified as being of one of three types, depending on the addresses of points in the attractor:*

1. *totally disconnected: each point of the attractor has a unique address. The attractor is totally disconnected if and only if*

$$T_i(\mathcal{A}) \bigcap T_j(\mathcal{A}) = \emptyset \ \forall \ i, j \in \{1, 2, \ldots, N\} \ \ with \ \ i \neq j$$

2. *just-touching: the attractor is not totally disconnected yet it contains a nonempty set $\mathcal{O}$ which is open in the metric space of the attractor and which satisfies the*

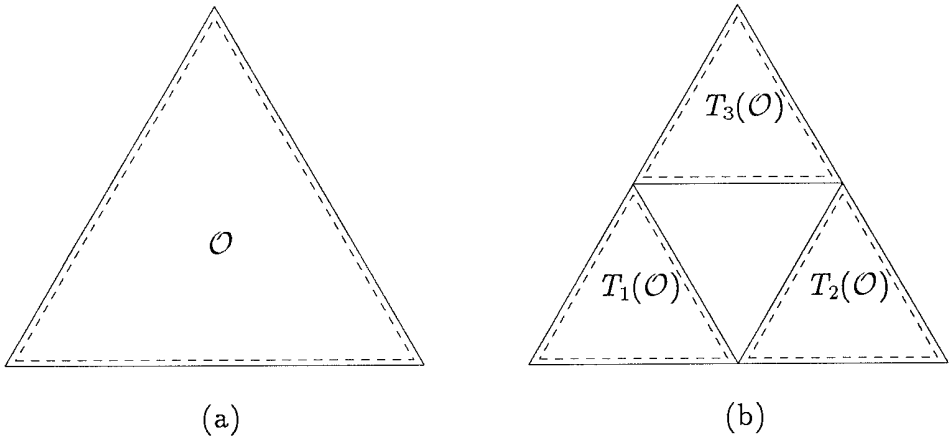



Figure 2.3: A schematic representation of the open-set condition. The open set $\mathcal{O}$ shown in (a) and its images under the transformations from $\mathcal{T}$ shown in (b) satisfy the criteria set out in Definition 2.11. The dashed lines indicate the boundaries of the open sets, excluding any of the limit points indicated by the solid lines.

$$T_1 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.250 & 0.433 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.500 & 0.000 & 1.000 \end{bmatrix}$$

Table 2.2: IFS for Sierpiński gasket.

*open set condition, given below. This condition is illustrated for the Sierpiński gasket in Figure 2.3.*

- $T_i(\mathcal{O}) \bigcap T_j(\mathcal{O}) = \emptyset \ \forall \ i, j \in \{1, 2, \ldots, N\}$ *with* $i \neq j$
- $\bigcup_{i=1}^{N} T_i(\mathcal{O}) \subset \mathcal{O}$

3. *overlapping: the attractor is neither disconnected nor just-touching.*

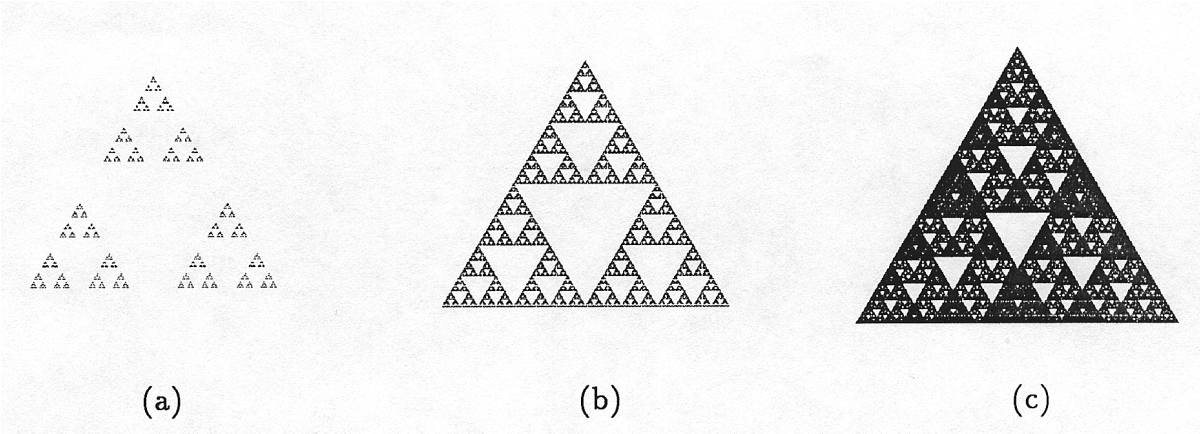



Figure 2.4: The three possible types of attractor: (a) totally disconnected, (b) just-touching, and (c) overlapping.

## 2.4 Algorithms for approximating the attractor

Following Hutchinson's study [23] of several fractal curves, Barnsley [4] was able to produce colourful images of the attractors using computer graphics techniques. From Definition 2.9, the attractor is the limit of all sequences which can be constructed from the elements of $\mathcal{T}$. In order to construct a picture of the attractor it is necessary to construct sequences of composite transformations from $\mathcal{T}$, and to plot the images of some initial point under these sequences of composite transformations. The choice for the initial point is important, as it could adversely affect the viewers' perception of the attractor. Figure 2.5 shows two pictures of the Sierpiński Gasket attractor differing only in the choice for the preimage. Since $\mathcal{A}$ is the closure of the fixed points of the finite composition of transformations from $\mathcal{T}$, the simplest approach is to choose a fixed point of one of the transformations in $\mathcal{T}$.

There are several means for constructing the actual transformation sequences since as mentioned, the attractor is the limit of all sequences which approximate it. Along with a discussion of the different techniques used for construction of these sequences, a criterion for measuring the accuracy of an approximation is also discussed.

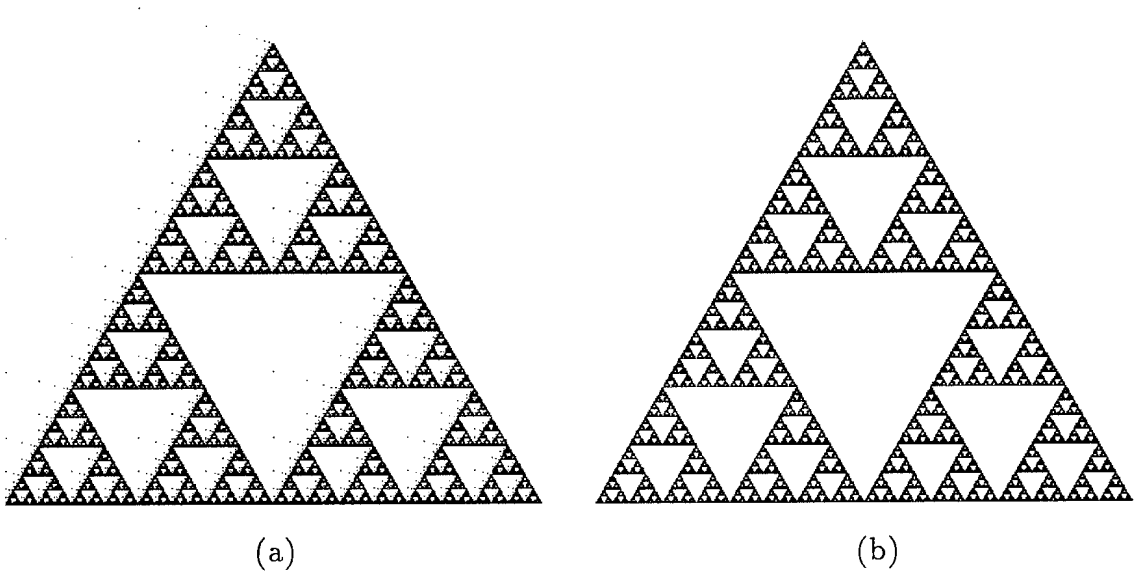



Figure 2.5: Illustration of the choice of preimage: (a) shows the attractor when the preimage $x_0 \notin \mathcal{A}$ and (b) shows the attractor with $x_0 \in \mathcal{A}$.

### 2.4.1 Enumeration method

The recursive expression for the attractor found in Definition 2.9 provides the basis for an algorithm, first described by Hutchinson [23], to generate all possible composite transformations of some finite length. The problem of finding the permutations of length at most $n$ in the set $\mathcal{T}$ is most easily handled by a recursive construction of a tree of transformations. A straightforward algorithm is to apply transformations based on an ordered traversal of this tree. The tree traversal will allow every possible sequence to be enumerated. Although other variations are possible, only breadth-first and depth-first traversal strategies are considered here. Computation is stopped when the search tree has achieved a specified depth, which is analogous to specifying a maximum length for the transformation sequences. This technique is similar to the IIM algorithm [32] for creating images of Julia sets. A variation on the deterministic approach, described by Williams [45], involves direct computation of the fixed points of the composite transformations.

Separate from the method of tree traversal is that of matrix composition. The

first element of all sequences of transformations is the identity transformation. In the process of constructing some finite composite transformation as an element of an approximating sequence, intermediate elements of the sequence may be composed using either premultiplication or postmultiplication. The image under these two different sequences will generally not be the same, since matrix multiplication is not commutative. Figure 2.6 illustrates the four variations possible. The approximation produced by any of the variations will contain the same points, only the order in which the points are generated will be different.

#### Breadth-first traversal

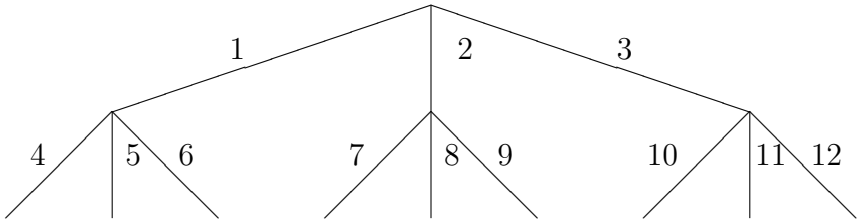
A breadth-first traversal visits all the siblings at level $n$ before proceeding to level $n+1$. The details of the method are given in Algorithm 2.1. The storage requirements for the tree are substantial, since whole levels of the tree must be stored in memory. Specifically, if a maximum depth of $N$ is selected, then both levels $N-1$ and $N$ will require storage. This approach is conceptually equivalent to a parallel construction of the transformation sequences. Since these transformation sequences can be generated independently of one another, the high cost in storage may not be justified.

#### Depth-first traversal

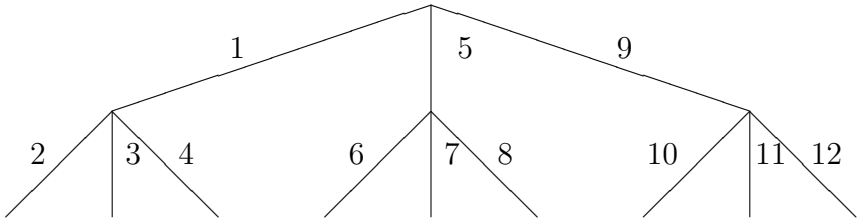
A depth-first traversal strategy is preferrable because the costs in storage are much lower, since only a single transformation sequence needs to be kept in memory at any one time. The details of this method are given in Algorithm 2.2.

### 2.4.2 Random iteration

Another possible method for generating an approximation to the attractor was proposed by Barnsley [4]. It uses the equivalent of a random traversal of the tree of transformations, by associating a set of probabilities $\mathcal{P} = \{p_1, \ldots, p_N\}$ with $\mathcal{T}$ to control transformation application. This method is explained more fully in Algorithm 2.3. Elton [14] proved that if the number of iterations is sufficiently high, the shape will converge to the attractor. The speed at which this process converges




(a)




(b)

| Composite Transformation | (a)Breadth-first PREMULT | POSTMULT | (b) Depth-first PREMULT | POSTMULT |
|---|---|---|---|---|
| 1 | $T_1$ | $T_1$ | $T_1$ | $T_1$ |
| 2 | $T_2$ | $T_2$ | $T_1T_1$ | $T_1T_1$ |
| 3 | $T_3$ | $T_3$ | $T_2T_1$ | $T_1T_2$ |
| 4 | $T_1T_1$ | $T_1T_1$ | $T_3T_1$ | $T_1T_3$ |
| 5 | $T_2T_1$ | $T_1T_2$ | $T_2$ | $T_2$ |
| 6 | $T_3T_1$ | $T_1T_3$ | $T_1T_2$ | $T_2T_1$ |
| 7 | $T_1T_2$ | $T_2T_1$ | $T_2T_2$ | $T_2T_2$ |
| 8 | $T_2T_2$ | $T_2T_2$ | $T_3T_2$ | $T_2T_3$ |
| 9 | $T_3T_2$ | $T_2T_3$ | $T_3$ | $T_3$ |
| 10 | $T_1T_3$ | $T_3T_1$ | $T_1T_3$ | $T_3T_1$ |
| 11 | $T_2T_3$ | $T_3T_2$ | $T_2T_3$ | $T_3T_2$ |
| 12 | $T_3T_3$ | $T_3T_3$ | $T_3T_3$ | $T_3T_3$ |

Figure 2.6: Construction of trees using (a) breadth-first and (b) depth-first traversal strategies. The labels on the trees indicate the order in which the paths are traversed and the table shows the composite transformation generated by each variation at that point.

| **ALGORITHM 2.1** | **Breadth-first**$(x_0, n, order, \mathcal{T}, N)$ | |
|---|---|---|
| **PURPOSE** | Compute an approximation to $\mathcal{A}$ | |
| Arguments | $x_0$ | the starting point |
| | $n$ | number of levels in tree |
| | *order* | order of multiplication |
| | $\mathcal{T}$ | the finite set of affine transformations |
| | $N$ | the number of transformations |
| Local variables | *List* | a list containing the current composite transformations |
| Global symbols | $Id$ | identity transformation |
| | POSTMULT | constant to test if *order* is set to postmultiplication |
| | $i$ | loop index |
| | $s$ | number of elements in list |

```
BEGIN
    List = Id
    s = 1
    FOR i = 1, ..., n DO
        List = Process-level(List, s, x_0, order, T, N)
        s = s^N
    ENDFOR
END
```

to the attractor is determined by the set of probabilities chosen. For an IFS with $N$ transformations, it is reasonable to assign each a probability of $\frac{1}{N}$ if each has the same Lipschitz constant. When the Lipschitz constants differ, the probabilities should be selected according to the contribution of each transformation to the area of the attractor (Equation 2.3).

$$p_i \approx \frac{\mid detA_i \mid}{\sum_{i=1}^{N} \mid detA_i \mid} \text{ for } i = 1, 2, \ldots, N. \tag{2.3}$$

| **Procedure** | **Process-level**$(List, s, x_0, order, \mathcal{T}, N)$ | |
|---|---|---|
| **Purpose** | Compute the image points in the next level of tree | |
| Arguments | $List$ | list of the composite transformations |
| | $s$ | number of points in the list |
| | $x_0$ | the starting point |
| | $order$ | order of multiplication |
| | $\mathcal{T}$ | the finite set of affine transformations |
| | $N$ | the number of transformations |
| Output | $\widehat{List}$ | modified list containing new transformations |
| Local variables | $i, j, k$ | loop indices |
| | $\widehat{List}$ | new list for composite transformations |
| Functions | plot-point() | plot a point on the screen |

```
BEGIN
    FOR i = 1, ..., s DO
        FOR j = 1, ..., N DO
            k = (i − 1)N + j
            IF (order == POSTMULT) THEN
                List^[k] = List[i]T_j
            ELSE
                List^[k] = T_j List[i]
            ENDIF
            plot-point(List[i](x_0))
        ENDFOR
    ENDFOR
    RETURN (List^)
END
```

**ALGORITHM 2.2** **Depth-first**$(T, x_0, n, order, \mathcal{T}, N)$
**PURPOSE** Compute an approximation to $\mathcal{A}$

| | | |
|---|---|---|
| Arguments | $T$ | current transformation matrix |
| | $x_0$ | preimage |
| | $n$ | number of levels in tree |
| | *order* | multiplication order |
| | $\mathcal{T}$ | the finite set of affine transformations |
| | $N$ | the number of transformations |
| Local variables | $i$ | loop index |
| | *comp* | composite transformation |
| Global symbols | POSTMULT | constant to test if *order* is set to postmultiplication |
| Functions | plot-point() | plot a point on the screen |

```
BEGIN
    IF (n > 0) THEN
        FOR i = 1, ..., n DO
            IF (order == POSTMULT) THEN
                comp = T T_i
            ELSE
                comp = T_i T
            ENDIF
            plot-point(comp(x_0))
            Depth-first(comp, x_0, n - 1, order, T, N)
        ENDFOR
    ENDIF
END
```

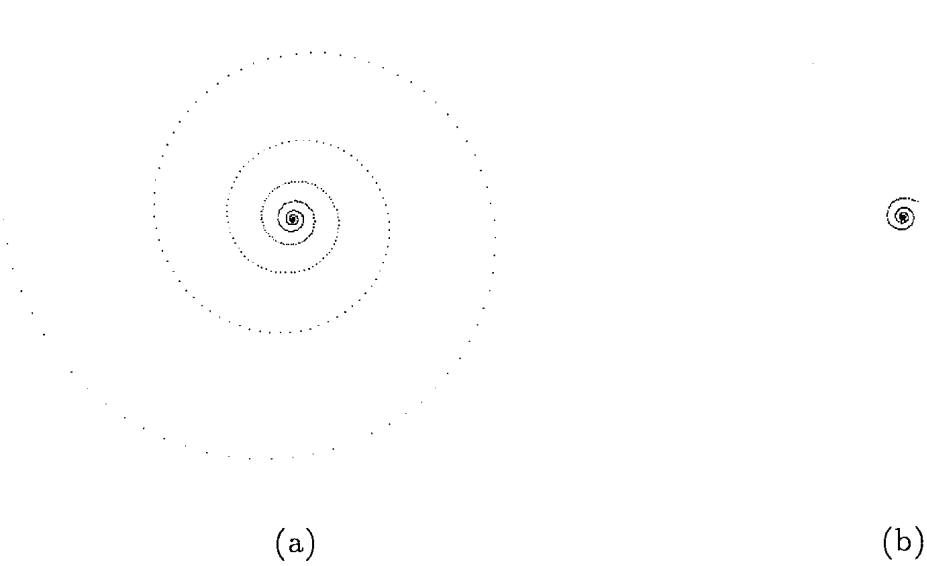



Figure 2.7: Two different sequences of image points under repeated application of the transformation in Table 2.3 which both converge to the fixed point of that transformation. The preimage for the sequence in (b) was selected to be closer to that fixed point than the one in (a).

---

In those cases when the IFS contains singular transformations, Barnsley[5] suggests an empirical approach to the assignment of probabilities. With the nondeterminism of this method there is no option for the order of matrix composition as postmultiplication must be used. Of the approximation methods considered, this method is most efficient in terms of memory requirements, as it requires storage only for the current transformation.

None of the methods presented thus far provide a criterion for measuring the accuracy with which the attractor can be approximated. Figure 2.7 shows two approximating sequences for the attractor defined by the IFS given in Table 2.3.

Regardless of the choice for an initial point, both sequences converge to $\mathcal{A}$. Since the IFS contains only one transformation, the attractor is the single point which is the fixed point of that transformation. The two spirals shown in Figure 2.7 are a result of different choices for the preimage, but it is not immediately clear which of these two spirals better approximates the attractor. In the development of a method for determining the accuracy of an approximation, the Euclidean distance function is

**ALGORITHM 2.3** **Random-iteration**$(x_0, n, \mathcal{T}, \mathcal{P}, N)$
**PURPOSE** Compute an approximation to $\mathcal{A}$

| | | |
|---|---|---|
| Arguments | $x_0$ | preimage |
| | $n$ | number of points |
| | $\mathcal{T}$ | the finite set of affine transformations |
| | $\mathcal{P}$ | the finite set of probabilities associated with $\mathcal{T}$ |
| | $N$ | the number of transformations |
| Local variables | $x, y$ | points |
| | $i, j$ | loop indices |
| | *rand* | storage for random number |
| | *psum* | sum of probabilities |
| | $k$ | index of selected transformation |
| Functions | rnd() | return a pseudo-random number $\in [0, 1]$ |
| | plot-point() | plot a point on the screen |

```
BEGIN
    x = x_0
    FOR i = 1, ..., n DO
        rand = rnd()
        psum = 0
        FOR j = 1, ..., N DO
            psum = psum + P_j
            IF (rand < psum) THEN
                k = j
                BREAK
            ENDIF
        ENDFOR
        y = T_k(x)
        plot-point(y)
        x = y
    ENDFOR
END
```

$$\begin{bmatrix} 0.986 & 0.086 & 0.000 \\ -0.086 & 0.986 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix}$$

Table 2.3: IFS for the singleton shown in Figure 2.7

Figure 2.8: An illustration of the asymmetry of the two parts of Definition 2.12. The black line represents the closed interval [0,1] (call it $A$), and the grey dots represent a set of points (call it $B$) which approximate the set $A$. Each point in $B$ corresponds to a point in $A$, so Formula 2.4 is 0. However, every point in $A$ does not correspond to a point in $B$, so Formula 2.5 is $\frac{1}{2}$.

used. It is necessary to measure the distance from each point in the approximation to each point in the attractor, using the Hausdorff distance.

**Definition 2.12 (Hausdorff distance)** *For any two compact sets $A$ and $B$, the Hausdorff distance between them $d_H(A, B)$ is defined to be:*

$$\max\left\{\max_{x\in A}\ \min_{y\in B} \parallel x-y \parallel, \max_{y\in B}\ \min_{x\in A} \parallel x-y \parallel\right\}.$$

Using this criterion, Figure 2.7(b) can be said to more accurately approximate the attractor, since all of its points lie closer to the attractor than many points of Figure 2.7(a). If the initial point $x_0 \in \mathcal{A}$ then the result is a set $\mathcal{A}_\varepsilon \subset \mathcal{A}$ which leads to

$$\max_{x\in\mathcal{A}_\varepsilon}\min_{y\in\mathcal{A}} \parallel x-y \parallel = 0\ . \tag{2.4}$$

The accuracy depends upon the second half of the formula, and therefore it can be said that a precision of $\varepsilon$ is attained if

$$\max_{y\in\mathcal{A}}\min_{x\in\mathcal{A}_\varepsilon} \parallel x-y \parallel < \varepsilon. \tag{2.5}$$

The idea involved in the computation is illustrated in Figure 2.4.2.

Following Definition 2.12, it would seem to be a simple matter to measure the accuracy with which the attractor of an IFS is approximated by a given point set.

Aside from the lengthy process of evaluating the distance from the definition, this technique does not provide any criterion for terminating the computations as it can only give the knowledge of what precision was reached by the computations.

This difficulty could be removed if it were possible to measure the precision according to some local criterion, without having to examine the entire set. It is possible to measure the distance between each image point under a particular transformation and the fixed point of that transformation but this involves the burden of first solving the system of equations to determine the fixed point and then computing the distance to that point. A further difficulty is that if a point $x_n$ is at some distance $\mu$ away from its limit, there is no guarantee that the point $x_{n+1}$ will be closer than $\mu$. Fortunately, if the transformations are composed using premultiplication, one can be guaranteed that each point in the sequence will be closer to its limit than the previous.

**Theorem 2.1** *The images of $x_0$ under a sequence of contractive composite transformations, each composed using premultiplication, form a Cauchy sequence.*

**Proof:** In order to show that this sequence is indeed Cauchy, it is necessary to show that each image point is contained in a disk which is contained in all the previous disks.

Consider the minimum enclosing disk $D$ for the attractor $\mathcal{A}$ of some IFS $\mathcal{T}$, and choose the point $x_0$ as the center of this disk. Since $D$ encloses all points in $\mathcal{A}$, all images of $x_0$ under the composite transformations from $\mathcal{T}$ will also be included in $D$. By definition of a contractive transformation [12], image points under such a transformation are moved nearer to the fixed point of the transformation by a factor determined by the Lipschitz constant for the transformation. Since $D$ is the minimum enclosing disk for the set $\mathcal{A}$, the disks $T_i(D)$ must be the minimum enclosing disks for the sets $T_i(\mathcal{A})$, and $T_i(D) \subset D$. Let $S = T_{i_n} \cdots T_{i_1}$ be a composite transformation and let $S(D)$ represent the disk $D$ under the transformation $S$. The disks $(T_iS)(D) \subset S(D)$ in similar fashion to the first case. Progressing from this case, let $R_j = T_jS$. In this way, the disks $T_iR_j \subset (T_iR_j)(D)$ since the final transformation, which determines the position most strongly, remains fixed. Pre-multiplication of transformations is required since the disks $ST_i(D)$ are pulled towards the respective fixed points of the

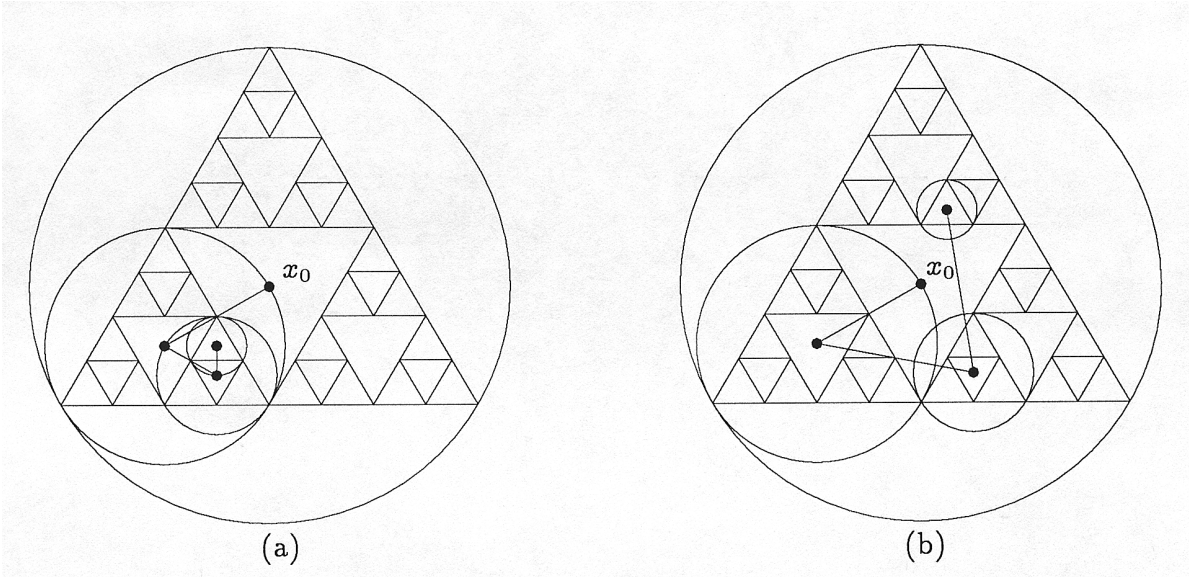



Figure 2.9: An examination of the sequences of images under (a) premultiplication and (b) postmultiplication of transformation matrices shows that both approach the fixed point of the composite transformation. Using premultiplication, successive images of the bounding disk $D$ are contained within one another and the distance between successive image points is strictly decreasing. This is not the case when post-mulitplication is used.

---

individual transformations. In this case, the distance between image points is not bounded by any single disk at any particular level.
The property discussed in Theorem 2.1 is illustrated in Figure 2.9.

### 2.4.3 Adaptive-cut method

It would be very time consuming to measure the distance between every two image points to measure the accuracy with which each particular sequence approximated its limit point. The decreasing values provided by Cauchy sequences provides an easier way. The approach is to find an upper bound, $D_{max}$, for the distance between the preimage and all limit points of the finite compositions from $\mathcal{T}$.

$$D_{max} = \max\{\| x_0 - y \| :\ y \in \mathcal{A}\}$$

Since $D_{max}$ cannot be computed directly, its value is estimated from a low resolution approximation to the attractor. This distance can be used in conjunction with the

Lipschitz constant of the composite transformation, $\lambda_{comp}$, to define a criterion which indicates when the desired precision has been reached.

$$D_{max} \times \lambda_{comp} < \varepsilon \tag{2.6}$$

This condition is both necessary and sufficient to guarantee the desired precision. If $x_0 \in \mathcal{A}$ then $D_{max}$ can be estimated easily by the diameter of the attractor, which is the approach adopted by Hepting et al [22]. A similar criterion was employed by Dubuc [13]:

$$diam(\mathcal{A}) = \max\{\| x - y \|: x, y \in \mathcal{A}\}.$$

In using a single distance estimate to measure the precision, some sequences may be computed to a higher degree of precision than desired. In fact, the initial estimate of $D_{max} = diam(\mathcal{A})$ can be decreased if the requirement that $x_0 \in \mathcal{A}$ is relaxed. The goal is to minimize the maximum distance between the preimage $x_0$ and any of the limit points of the finite compositions from $\mathcal{T}$. By definition, the radius of the minimum disk enclosing $\mathcal{A}$ satisfies this criterion. Therefore, $D_{max} = R$ where $R$ is the radius of this minimum enclosing disk is the optimum value. The difference in the choice of preimages is illustrated in Figure 2.10.

Given this measure for the accuracy of approximation, it is straightforward to determine the number of transformation applications required to achieve a desired accuracy of approximation. In the simple case when all transformations have the same Lipschitz constants, the following equation can be used to determine the number of transformation applications needed to achieve precision $\varepsilon$.

$$N = \lceil \frac{\log(\frac{\varepsilon}{D_{max}})}{\log(\lambda)} \rceil$$

However, when the contraction ratios of the different transformations are not all equal, a new problem arises. In this case, different numbers of transformation applications are required, depending on the contraction ratio of each component transformation. Upper and lower bounds on the number of transformations can be determined using the following relation, which follows from the work done by Reuter [39].

$$\lceil \frac{\log(\frac{\varepsilon}{D_{max}})}{\log(\lambda_{min})} \rceil \leq N \leq \lceil \frac{\log(\frac{\varepsilon}{D_{max}})}{\log(\lambda_{max})} \rceil$$

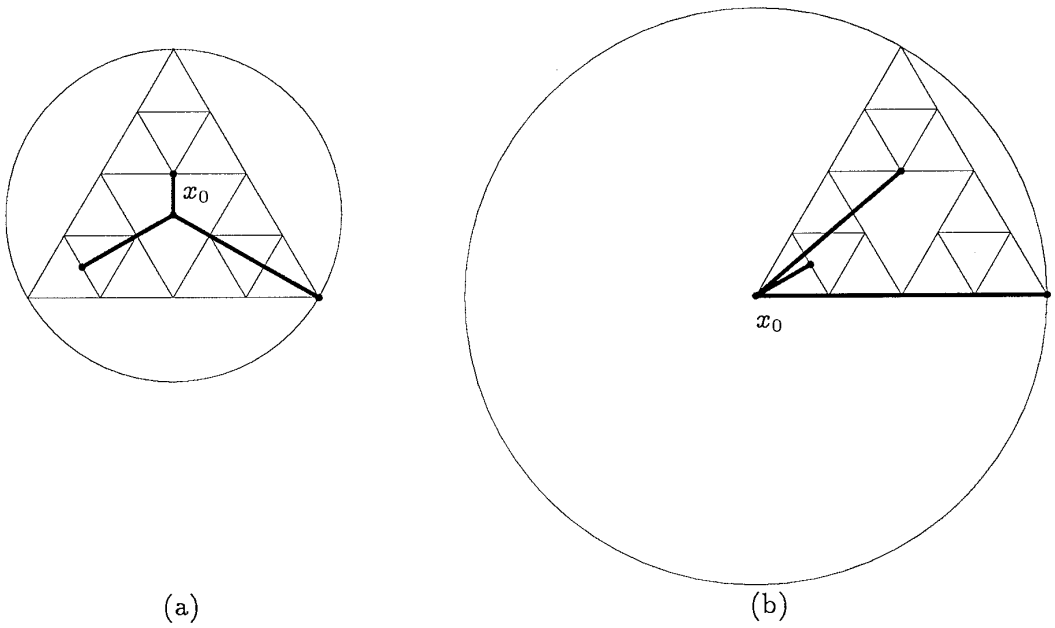



Figure 2.10: An illustration of the importance of selecting the starting point for computations. The circles are drawn with radius equal to the maximum of $d(x_0, \mathcal{A})$. The case when $x_0$ is chosen to be the center of the minimum enclosing disk is shown in (a). The case when $x_0$ is chosen as one of the fixed points of the transformations is shown in (b). Since the maximum distance is decreased in (a), there is a smaller variance in the distances to all points in the attractor when compared to (b).

---

The Black Spleenwort fern attractor approximation shown in Figure 2.11 generated using the enumeration method reveals the problems for approximation when transformations with different scaling ratios are applied equally.

The maximum detail that can be perceived in any picture of an attractor is a function of the resolution of the output device. As such, computations should be stopped when this precision is reached. To improve the quality of image at this level, oversampling may be employed. Alternatively, for a complex IFS, the time to generate an image to the level of full screen resolution may be unpleasantly long and it would be preferable to compute an approximation only to a much lower precision. The following section presents a method which employs the ideas discussed here in order to compute an approximation to the attractor with a desired level of precision and a minimum number of points.

Figure 2.11: The enumeration method is used to generate an approximation to the Black Spleenwort fern attractor, shown in Figure 2.2(b).

$$T_1 = \begin{bmatrix} 0.000 & 0.000 & 0.000 \\ 0.000 & 0.160 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.200 & 0.230 & 0.000 \\ -0.260 & 0.220 & 0.000 \\ 0.000 & 1.600 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} -0.150 & 0.260 & 0.000 \\ 0.280 & 0.240 & 0.000 \\ 0.000 & 0.440 & 1.000 \end{bmatrix} \quad T_4 = \begin{bmatrix} 0.850 & -0.040 & 0.000 \\ 0.040 & 0.850 & 0.000 \\ 0.000 & 1.600 & 1.000 \end{bmatrix}$$

Table 2.4: IFS for Black Spleenwort fern.

The **Adaptive-cut** uses a depth-first traversal of the transformation tree and premultiplication of matrices. Rather than evaluate each sequence to a specified length, the approximation criterion in Formula 2.6 (see page 27) is applied to each sequence of transformations to determine when the computation of each sequence may be terminated. Let $\mathcal{L} = \{\lambda_1, \ldots, \lambda_N\}$ be the contraction ratios for the affine transformations in $\mathcal{T}$. The sets $T_k(\mathcal{A})$ cover $\mathcal{A}$:

$$\mathcal{A} = \bigcup_{k=1}^{N} \mathcal{A}T_k \ .$$

Let $R$ denote the radius of the minimum disk enclosing $\mathcal{A}$ and let $x_0$ be the centre of the minimum enclosing disk. The covering sets $T_k(\mathcal{A})$, centered at $T_k(x_0)$ have radii of $\lambda_k R$. Each set $T_k(\mathcal{A})$ in turn is covered by sets $T_k(T_l(\mathcal{A})), l = 1, \ldots, N$ having radii $\lambda_l \lambda_k R$, and so forth. Proceeding in this manner, it is possible to cover the whole attractor $\mathcal{A}$ by sets of radii less than $\varepsilon$. These covering sets are given in the form

$$\mathcal{A}T_{k_n} \cdots T_{k_1} T_{k_0}$$

centered at:

$$x_0 T_{k_n} \cdots T_{k_1} T_{k_0}$$

with

$$\lambda_{k_n} \cdots \lambda_{k_1} \lambda_{k_0} < \frac{\varepsilon}{R} \ .$$

An important benefit of this method over the others is that since the accuracy of approximation can be measured, one only needs to examine the image under the final composite transformation. It is essential that premultiplication of transformations is used in constructing this final composite matrix, since the use of postmultiplication would not yield a Cauchy sequence. Although the computations for measurement are identical regardless of the method of composition, the effect is to distribute the points in the approximation evenly across the whole attractor. Figure 2.9 indicates the problem, as the location of the each image point is determined by the last transformation applied to it. Although this difference between orders of multiplication is not evident for those IFS like the Sierpiński gasket, it becomes very clear for those IFS like the Black Spleenwort fern. The transformations for this attractor are given

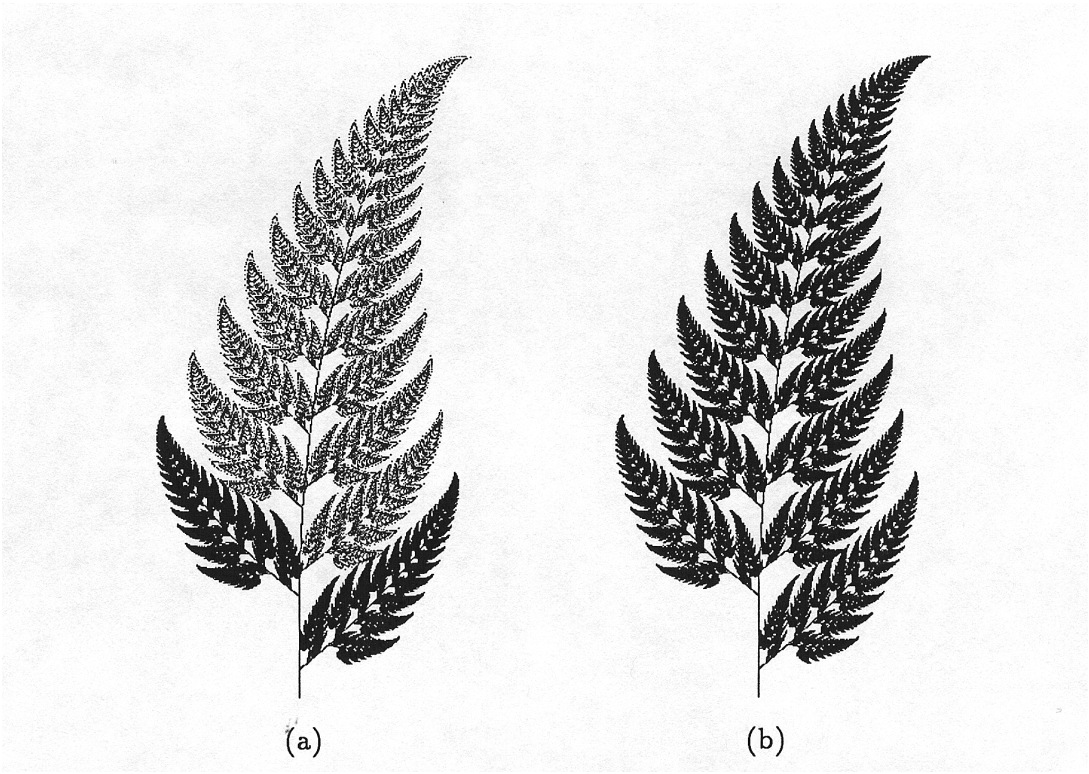



Figure 2.12: The different point densities which result from (a) postmultiplication and (b) premultiplication of transformations when using the termination criterion in Formula 2.6.

---

in Table 2.4. In this case, the Lipschitz constants for the transformations differ considerably, between 0.01 and 0.85. One might rightly expect the least contractive transformation to require more applications than the others in order to achieve the desired precision. This is in fact the case using premultiplication of transformations and the adaptive termination criterion. As an illustration of the failure of postmultiplication of matrices, examine Figure 2.12(a). Figure 2.12(b) shows the corrected version using premultiplication of matrices.

Any implementation of Algorithm 2.4 relies on the precision with which it is possible to estimate certain quantities, namely the size of the attractor and the Lipschitz constants of each transformation.

To estimate the size of the attractor, an approximation to the attractor must be generated and measured. The radius of the minimum enclosing disk is taken to

| **ALGORITHM 2.4** | **Adaptive-cut**$(T, x_0, \mathcal{T}, R, \mathcal{L}, \varepsilon, N)$ | |
|---|---|---|
| **PURPOSE** | Calculate an approximation $\mathcal{A}$ with precision $\varepsilon$ | |
| Arguments | $T$ | current transformation matrix |
| | $x_0$ | preimage, centre of minimum disk enclosing $\mathcal{A}$ |
| | $\mathcal{T}$ | the finite set of affine transformations |
| | $R$ | radius estimate for $T(\mathcal{A})$ |
| | $\mathcal{L}$ | the finite set of contraction ratios |
| | $\varepsilon$ | precision of approximation |
| | $N$ | the number of transformations |
| Local variables | $i$ | loop index |
| Functions | plot-subset() | plot a subset of $\mathcal{A}$ with specified radius on the screen |

```
BEGIN
        IF (R > ε) THEN
                FOR i = 1, ..., N DO
                        Adaptive-cut(T_i T, x_0, T, λ_i R, L, ε, N)
                ENDFOR
        ELSE
                plot-subset(T(x_0), ε)
        ENDIF
END
```

be one half of the diagonal of the bounding box. Although this represents a rather unsophisticated approach to the problem, satisfactory results have been obtained. The minimum enclosing disk could be found more effectively with the use of the algorithm described by Elzinga [15].

The contraction ratios of the composite transformations could be computed directly each time, but the cost of evaluating the spectral norm described in Equation 2.1 is very high. A more practical approach is to multiply the contraction ratios of the component matrices. Although this approach is good for similarities, it may fail for general affinities. The paper by Hepting et al. [22] contains a discussion of

$$T_1 = \begin{bmatrix} 0.001 & 0.000 & 0.000 \\ 0.000 & 0.999 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.999 & 0.000 & 0.000 \\ 0.000 & 0.001 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix}$$

Table 2.5: A sample pathological IFS for which the current techniques of estimating the Lipschitz constants fail.

$$T_1 = \begin{bmatrix} 0.333 & 0.000 & 0.000 \\ 0.000 & 0.333 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.667 & 0.000 & 0.000 \\ 0.000 & 0.333 & 0.000 \\ 0.333 & 0.000 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.667 & 0.000 & 0.000 \\ 0.000 & 0.667 & 0.000 \\ 0.333 & 0.333 & 1.000 \end{bmatrix} \quad T_4 = \begin{bmatrix} 0.333 & 0.000 & 0.000 \\ 0.000 & 0.667 & 0.000 \\ 0.000 & 0.333 & 1.000 \end{bmatrix}$$

Table 2.6: IFS for square using affinities.

other techniques for arriving at an estimate.

The simplest method for estimating compound contraction factors relies on the property $\lambda(ST) \leq \lambda(S)\lambda(T)$ and estimates the contraction factor of a composite affine transformation $T_{k_1} \cdots T_{k_n}$ as the product of the individual contraction factors. Thus, $\lambda(T_{k_1} \cdots T_{k_n}) \leq \lambda(T_{k_1}) \cdots \lambda(T_{k_n})$. However, this formula may grossly overestimate the actual value of the contraction ratio of the composite transformation in the case of certain affine transformations.

Referring to the IFS in Table 2.5, since $\lambda(T_1) = \lambda(T_2) = 99/100$, the product $\lambda(T_1)\lambda(T_2)$ yields $99^2/100^2$. On the other hand, $\lambda(T_1T_2) = 99/100^2$. Thus, the use of the product $\lambda(T_1)\lambda(T_2)$ overestimates the actual value of $\lambda(T_1T_2)$ by the factor of 99.

The **Adaptive-cut** method, employing the above method of estimation is illustrated in Figure 2.13. The transformations in the IFS for this square are given in Table 2.6. The termination criterion ($\varepsilon = \frac{1}{3}$) is such that a single application of $T_1$ is sufficient. Notice that computation of certain squares is continued beyond the point where this criterion is actually met.

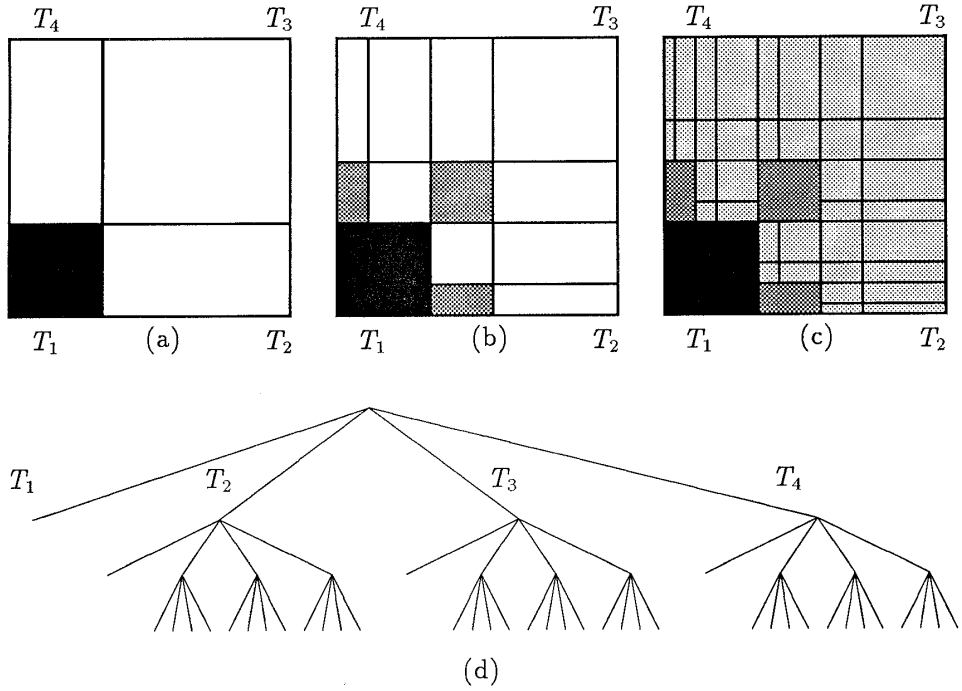



Figure 2.13: Operation of the **Adaptive-cut** method on the square defined by the IFS given in Table 2.6, using a precision value $\varepsilon = \frac{1}{3}$. Three levels of transformation application are required to compute all parts to the desired precision. Each version of the square in (a), (b) and (c) corresponds to a level in the tree of transformations, shown in (d). As each section of the square reaches the final precision, it is coloured to indicate that no further computations are performed.

---

The **Adaptive-cut** method also suggests an empirical approach to the selection of probabilities in the **Random-iteration** method since it produces an estimate of the relative areas covered by each transformation. This area information is accessible by computing the ratios of the transformation application, determined by the rightmost element of each sequence.

## 2.5 Magnification of the attractor

Reuter [39] presented an algorithm for the magnification of sections of the attractor using the concept of addresses on the attractor. Subsets of the attractor

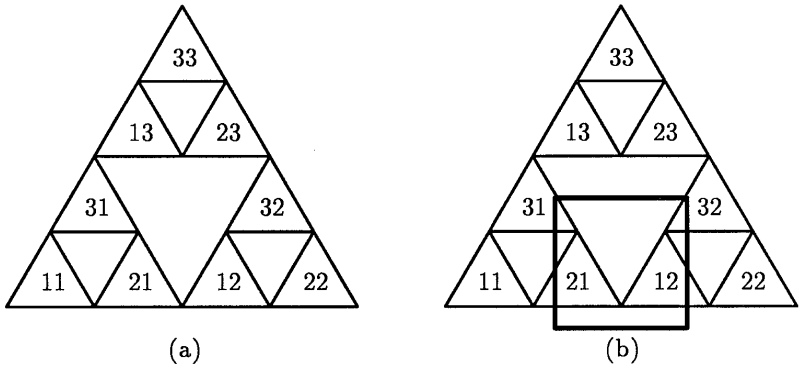



Figure 2.14: The subsets which result from considering the last two component transformations.

can be distinguished from others by examining the last $m$ elements in each composite transformation. For example, using the last 2 transformations in the Sierpiński gasket example will divide the attractor into 9 subsets. This idea is illustrated in Figure 2.14(a). Using the window shown in Figure 2.14(b), none of the transformation sequences ending in 11, 22, 13, 23, or 33 are visible within that window, and hence should not be used.

Her approach involved cataloguing the various postmultiplication sequences which identified all visible subsets of the attractor. Attractor generation then proceeded by random iteration of points, with each one being postmultiplied by a randomly selected sequence found during the cataloguing phase.

This cataloguing phase is not required if the **Adaptive-cut** is used to construct the attractor. Computation of any sequence of transformations is stopped once its bounding disk is found to be completely outside of the window of interest. There is some overhead associated with this approach, since the preimage must be transformed and tested with all the intermediate elements of the sequence. The accuracy of this method is demonstrated in Figure 2.15.

In later chapters, methods for calculating fields of data about the attractor will be examined. In these cases, it will not be correct to only compute the visible portion of the attractor.

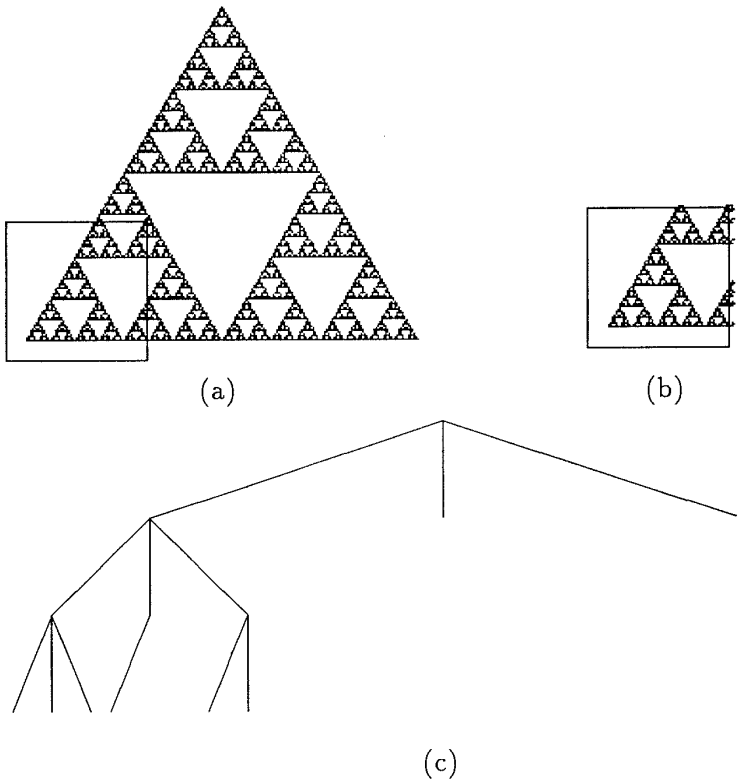



Figure 2.15: Application of the **Adaptive-cut** method to the efficient magnification of regions in the attractor. A sample region for magnification is indicated by the square drawn on (a), and the points actually computed for that magnification are shown in (b). The first levels of the corresponding tree of transformation applications are shown in (c).

## 2.6 Probabilities and measures

All approximations to the Black Spleenwort fern attractor shown are correct in some sense of the word. The attractor underlying each approximation is the same, but each approximation may converge to the attractor at different rates. Even after each process converges to the same final shape of the attractor, the relative densities of points will remain different. The mass of points at each pixel indicates the underlying measure on the set.

Different sets of probabilities change the perception of the attractor. Barnsley suggests the set of probabilities $\{T_1 = 0.01, T_2 = 0.07, T_3 = 0.07, T_4 = 0.85\ \}$ be

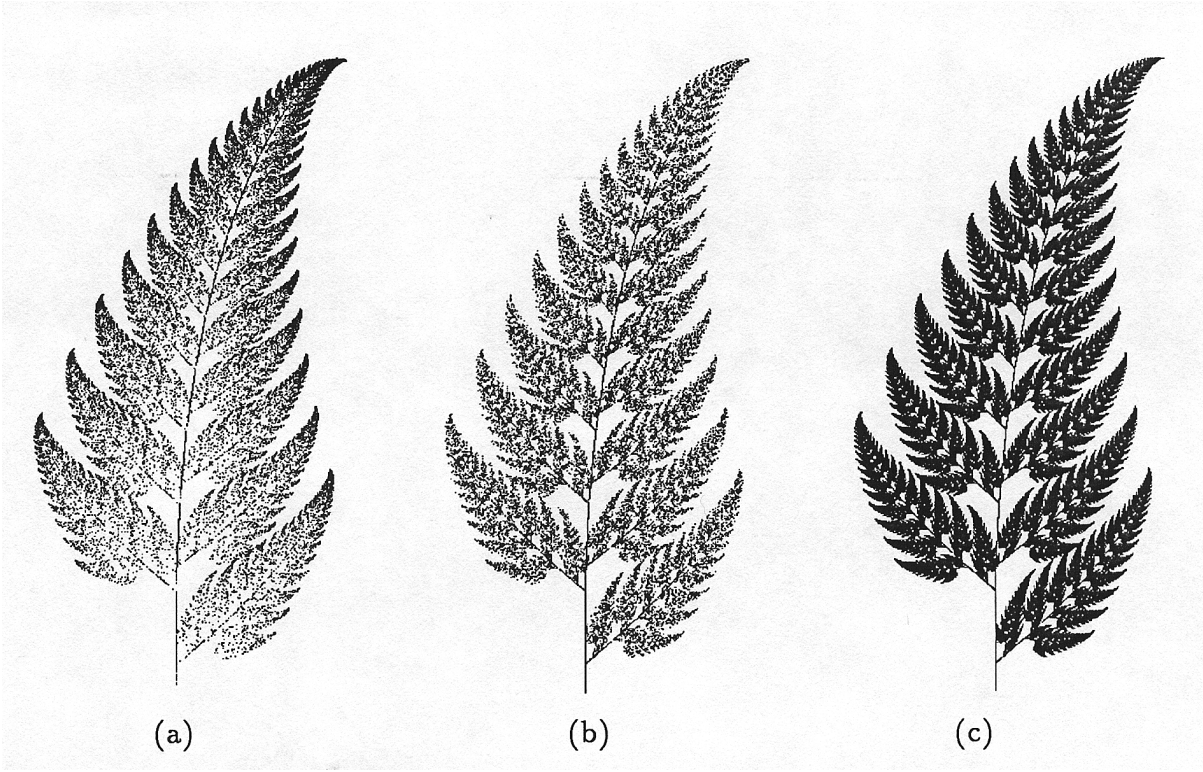



Figure 2.16: Three approximations to the Black Spleenwort fern attractor where (a) uses the random-iteration method with probabilities suggested by Barnsley, (b) uses the random-iteration method with probabilities derived from operation of the **Adaptive-cut** method and (c) uses the **Adaptive-cut** method directly.

---

used in generating the Black Spleenwort fern attractor. A set which provides a more uniform covering of points on the attractor is $\{T_1 = 0.03, T_2 = 0.11, T_3 = 0.13, T_4 = 0.73\}$. The attractor approximations resulting from each of these sets of probabilities are shown in Figure 2.16(a) and (b) respectively.

Hutchinson introduced the idea of measures as a means of differentiating among the approximations of an attractor produced by different IFS's. For example, there are an infinite number of transformations with contraction ratios $0.5 \leq \lambda < 1$ that will generate the line segment [0,1]. The use of transformations with $\lambda < 0.5$ might mean that the attractor would be disconnected. The difference between these sets is the invariant measure which arises from the IFS. Figure 2.17 demonstrates the idea for two different IFS's.

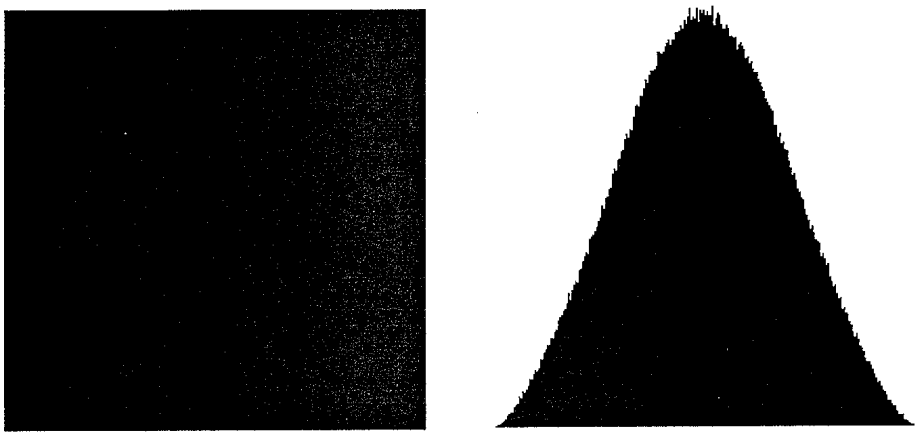



Figure 2.17: Normalized histograms of the invariant measures on the line [0,1] resulting from (a) the IFS in Table 2.7 and (b) the IFS in Table 2.8.

$$T_1 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.500 & 0.500 & 1.000 \end{bmatrix}$$

Table 2.7: IFS for the line segment [0,1] with scaling 0.5.

$$T_1 = \begin{bmatrix} 0.750 & 0.000 & 0.000 \\ 0.000 & 0.750 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.750 & 0.000 & 0.000 \\ 0.000 & 0.750 & 0.000 \\ 0.250 & 0.000 & 1.000 \end{bmatrix}$$

Table 2.8: IFS for the line segment [0,1] with scaling 0.75.

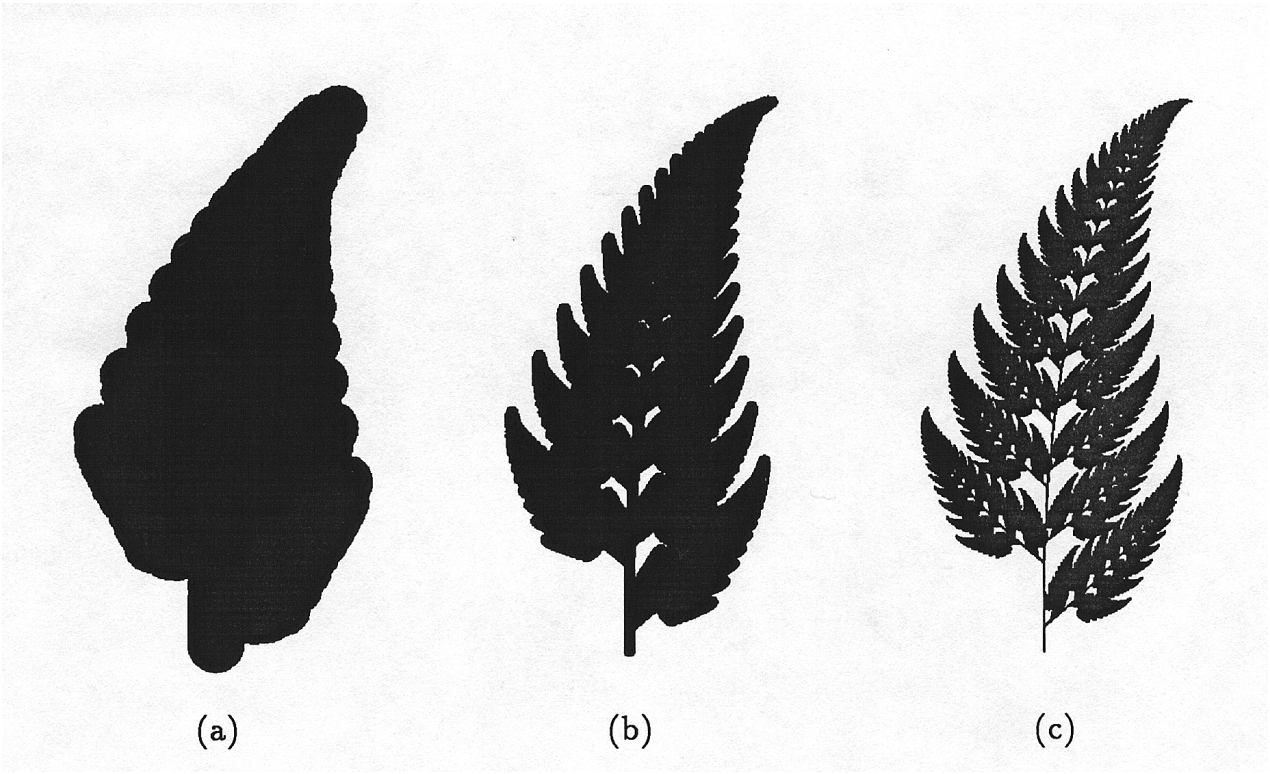



Figure 2.18: A progression of approximations to the Black Spleenwort fern attractor, where (a) has a low precision and (c) has precision equal to the size of pixels used in the computation.

---

It is not essential that the **Random-iteration** method be used to compute measures. The measure from a particular set of probabilities can be estimated accurately using the **Adaptive-cut** method [22]. The advantage of the deterministic approach is that far fewer computations may be required to achieve the desired approximation and a bound on the accuracy of the computation is also achieved.

## 2.7 Visualization techniques

The measure of accuracy which the **Adaptive-cut** method provides makes it possible capture the geometry of the attractor by rendering the attractor with disks of radius epsilon. An application of this technique is shown in Figure 2.18 which allows the geometry to be seen quickly and then refined to the final shape.

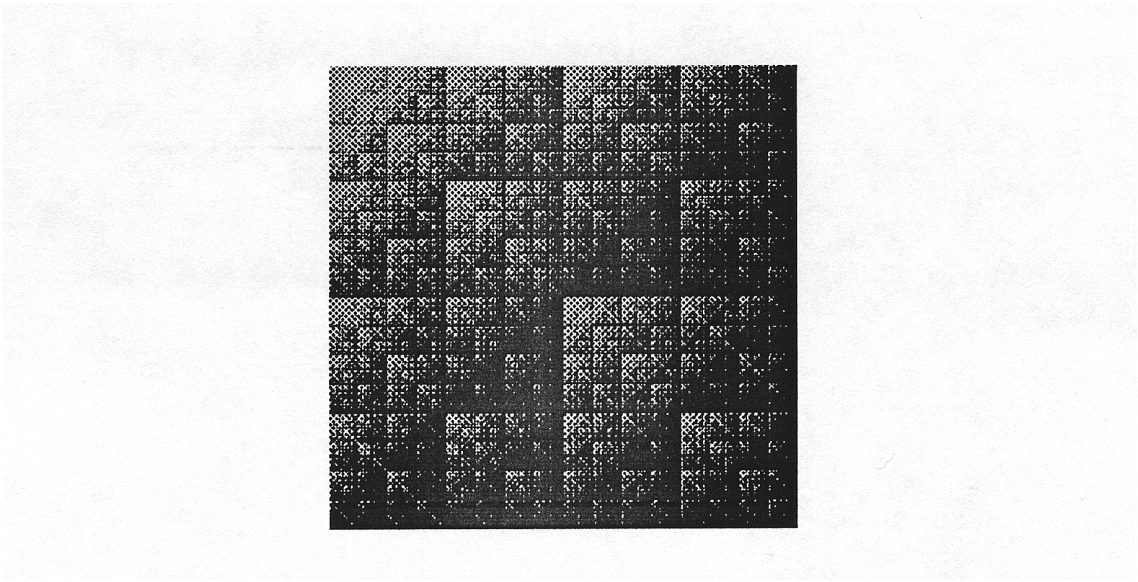


Figure 2.19: A square rendered using a non-uniform measure, defined with the probabilities $\{T_1 = 0.2, T_2 = 0.2, T_3 = 0.2, T_4 = 0.4\}$ associated with the transformations in Table 2.9.

$$T_1 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.500 & 0.000 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.500 & 0.500 & 1.000 \end{bmatrix} \quad T_1 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.500 & 1.000 \end{bmatrix}$$

Table 2.9: IFS for a square using similarities.

### 2.7.1 Invariant measures

The technique of using measures to render the set was proposed by Barnsley[4]. In general, a grid corresponding to the display resolution is used to store the number of times that each grid cell is hit. Once the computations have finished, each value is normalized and then used to assign colours. Figure 2.19 illustrates this technique.

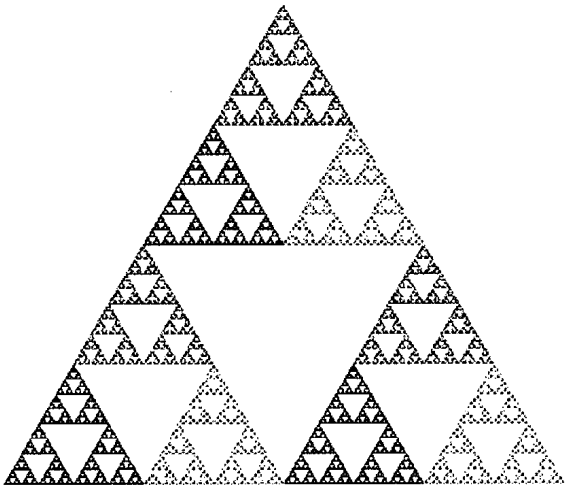


Figure 2.20: A Sierpiński gasket attractor rendered by examining the last two transformations for each point.

### 2.7.2 Transformation indices

Associated with each point $x \in \mathcal{A}$ is the sequence of transformations which were applied to the preimage to arrive at the point $x$. Barnsley calls this the code string for a particular point. Points in the attractor can be coloured separately according to the indices provided by the code string for the point. Each level of indexing divides the set into $N$ subsets, where $N$ is the number of transformations in the IFS. This approach was first described by Reuter [39]. A sample of this technique is shown in Figure 2.20.

### 2.7.3 Metrics

As the method for computing distance changes, so does the way points are related to one another. For example, the locus of points equidistant from a point is a circle in the Euclidean metric, but a diamond in the Manhattan metric (as Figure 2.21 shows).

Although no difference is visible when the resolution is very high, the difference in shape becomes apparent at low resolutions. This is illustrated with the Dragon

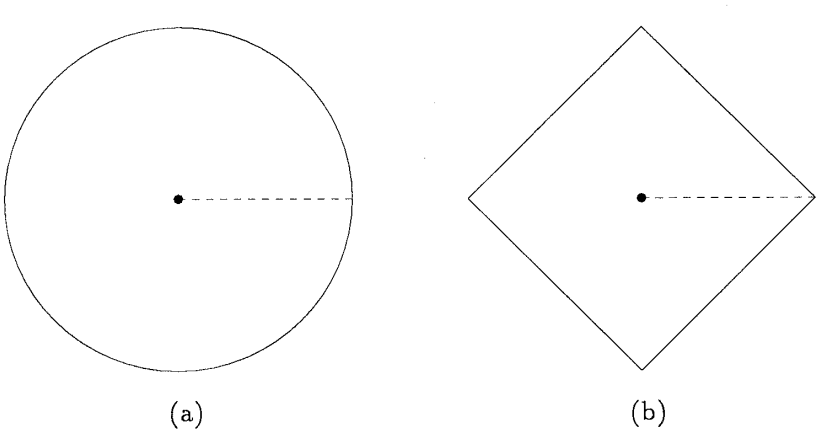



Figure 2.21: The locus of points which are a certain distance from another point changes depending on the metric being used. Here, those loci of points are illustrated for (a) Euclidean and (b) Manhattan metrics.

$$T_1 = \begin{bmatrix} 0.500 & 0.500 & 0.000 \\ -0.500 & 0.500 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} -0.500 & 0.500 & 0.000 \\ -0.500 & -0.500 & 0.000 \\ 0.000 & 1.000 & 1.000 \end{bmatrix}$$

Table 2.10: IFS for the Dragon curve.

curves in Figure 2.22. The IFS for that curve is given in Table 2.10.

### 2.7.4 Ray-tracing of three-dimensional IFS attractors

All algorithms described for generating approximations in two dimensions can easily be extended to the $n$-dimensional case. The **Adaptive-cut** method can be extended into three dimensions by the replacement of disks by spheres. Spheres are the simplest primitives and are available in all current ray-tracing programs. As done previously, the radius of the sphere is determined by the precision of the attractor generated. This approach seeks to build a model of the attractor with the desired precision using a collection of spheres. Through this method, the complex structure of the attractor is described easily, and in a form which requires no special processing.

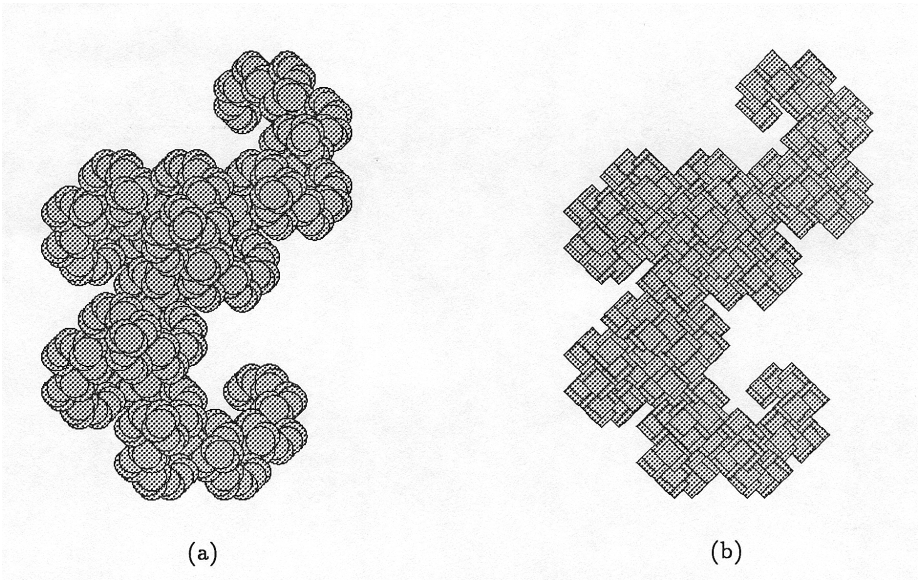



Figure 2.22: A low resolution approximation to the Dragon curve attractor, calculated using: (a) the Euclidean metric and rendered with disks, (b) the Manhattan metric and rendered with diamonds.

---

An example of this technique is given in Figure 2.23, whose attractor is defined by the IFS in Table 2.11. Hart [21] has done work in the area of ray-tracing deterministic fractals. The approach taken by Hart uses the technique of "unbounding volumes" to eliminate points from membership in the attractor. Although this method can proceed directly from the transformations of the IFS, it requires a method for computing distance to the attractor and a specialized ray-tracing program which can perform the required computations.

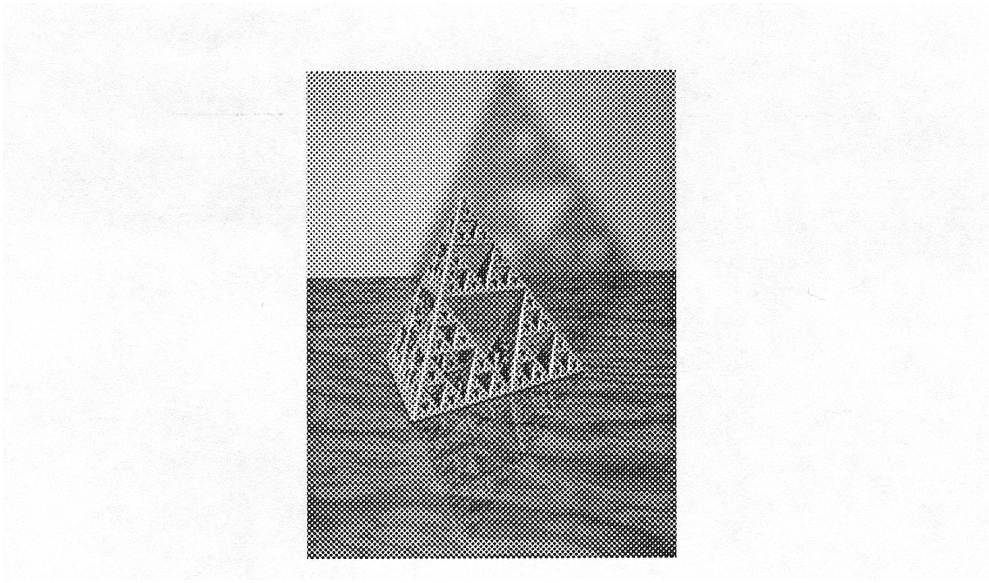


Figure 2.23: A three-dimensional attractor modelled as a set of spheres and ray-traced. A colour version of this image appears in the original of this thesis as Figure **??**.

$$T_1 = \begin{bmatrix} 0.500 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.500 & 0.000 \\ 0.000 & 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.500 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.500 & 0.000 \\ 0.500 & 0.000 & 0.000 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.500 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.500 & 0.000 \\ 0.250 & 0.433 & 0.000 & 1.000 \end{bmatrix} \quad T_4 = \begin{bmatrix} 0.500 & 0.000 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 & 0.000 \\ 0.000 & 0.000 & 0.500 & 0.000 \\ 0.250 & 0.145 & 0.409 & 1.000 \end{bmatrix}$$

Table 2.11: IFS for tetrahedron.

# Chapter 3

# Distance Calculation

The distance of a point from a fractal has been used as a means to render the fractal itself [21]. Knowledge of distance from a fractal provides an important characterization of the complement set.

Calculation of the distance of a point from the attractor of an IFS has a parallel to the distance estimation methods for Julia sets. The quadratic form of Julia set expressions lends itself well to an estimation formula. However, lacking such a formula for the IFS case, one is forced to compute the distance directly. Because Julia sets are defined as the boundary between two stable regions, estimation of distance to these sets has an added significance than in the case of IFS attractors where no distinctions can be made about the points which make up the attractor.

This chapter develops distance calculation as an effective tool for the study of points in the complement of an IFS attractor. Despite the differences already noted, it also provides a means to further study the relationship between IFS's and Julia sets.

## 3.1 Distance for Julia sets

Recall the definition of a Julia set, $J_c$, from Chapter 2 (see Formula 2.2 on page 10). It is possible to estimate the distance $d(z_0, J_c)$, for any point $z_0$, based on a potential function [33] for Julia sets. To perform the estimation, some radius $R$ is selected as a boundary for the target set about infinity and a maximum number of iterations $N$ is

chosen. The sequence of points $z_{k+1} = z_k^2 + c, k = 0, 1, 2, \ldots$ is computed until either the radius $R$ is exceeded or the maximum number of iterations $N$ is reached. If the images remain within the radius after $N$ iterations, $z_0$ is considered to be a member of the set $K_c$. Even if the point $z_k$ lies outside of $R$, it may still be close to the set. In this case, another sequence of points is computed in order to arrive at an estimate for the distance from the set. This accuracy of this estimate deteriorates rapidly as the distance from the set increases.

## 3.2 Distance for IFS attractors

The problem of calculating the distance of any point to the set $\mathcal{A}$ is more complex than it might first appear, since there is no simple formula which may be applied. The distance of the point $x$ to the set $\mathcal{A}$ is defined by Equation 3.1.

$$d(x, \mathcal{A}) = \min_{y \in \mathcal{A}} \parallel x - y \parallel \tag{3.1}$$

Unfortunately, direct computation of the distance in this manner is impossible, due to the infinite nature of the set $\mathcal{A}$. As an alternative, a finite approximation to the attractor can be made and subsequently used for the distance calculations. The accuracy of the distance calculation depends upon the accuracy with which $\mathcal{A}_\varepsilon$ approximates $\mathcal{A}$. Chapter 2 provided a discussion of techniques for constructing this point set. Using $\mathcal{A}_\varepsilon$ to approximate $\mathcal{A}$ with an accuracy of $\varepsilon$, it is possible to establish bounds for the distance of a point $x$ to the set $\mathcal{A}_\varepsilon$ with Formula 3.2:

$$d(x, \mathcal{A}_\varepsilon) \geq d(x, \mathcal{A}) \geq d(x, \mathcal{A}_\varepsilon) - \varepsilon. \tag{3.2}$$

This chapter shall examine methods for efficient computation of the lower bound for $d(x, \mathcal{A})$ and why the ability to perform this calculation is important.

## 3.3 Algorithms for computing distance

The optimal solution of the distance problem in one dimension is straightforward. On a line, the points can be sorted. Once a sorted list is found, a binary search can

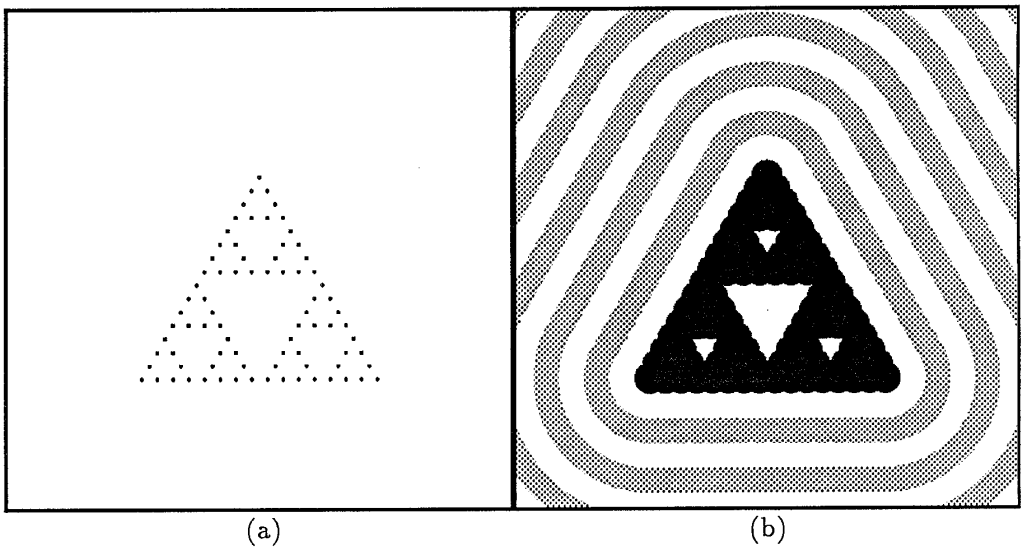



Figure 3.1: Distance to the Sierpiński gasket calculated with $\varepsilon = 0.0625$ where (a) shows the set of points $\mathcal{A}_\varepsilon$ which approximate the gasket and (b) shows the image calculated using $\mathcal{A}_\varepsilon$, with equidistant bands drawn surrounding the attractor approximation to illustrate the character of the distance values.

---

be used to locate the two points on either side of the point in question, one of which must be the closest to the sampling point. However, for two or more dimensions the problem becomes necessarily more complex, since there is no metric which can impose a total ordering on the points.

### 3.3.1 Basic method

After having constructed $\mathcal{A}_\varepsilon$, the simplest approach is to calculate the distance from $x$ to each point $y \in \mathcal{A}_\varepsilon$. The details of this computation method are given in Algorithm 3.1. Distance computation based on a finite approximation of $\mathcal{A}$ is simple to implement. However, the time needed to compute $d(x, \mathcal{A}_\varepsilon)$ is directly proportional to the number of points in $\mathcal{A}_\varepsilon$. As the precision of $\mathcal{A}_\varepsilon$ increases, the number of points in $\mathcal{A}_\varepsilon$ and consequently the amount of storage required grow exponentially.

Algorithm 3.1(**Basic-distance**) is very general, as it makes no assumption about

| **ALGORITHM 3.1** | **Basic-distance**($PtList$,$n$,$x$) | |
|---|---|---|
| **PURPOSE** | Estimate the distance of a point $x$ to the attractor | |
| Arguments | $PtList$ | list of points in $\mathcal{A}_\varepsilon$ |
| | $n$ | number of points in $PtList$ |
| | $x$ | point |
| Output | $d_{min}$ | distance estimate with $\mid d_{min} - d(x, \mathcal{A}) \mid \leq \varepsilon$ |
| | $y$ | a point in $\mathcal{A}_\varepsilon$ with $d(x, y) = d_{min}$ |
| Local variables | $i$ | index into the list $PtList$ |
| | $dist$ | distance between $x$ and $PtList[i]$ |
| Global symbols | HUGE | a sufficiently large number |

```
BEGIN
        d_min = HUGE
        FOR i = 1, ..., n DO
                dist = d(x, PtList[i])
                IF (dist < d_min) THEN
                        d_min = dist
                        y = PtList[i]
                ENDIF
        ENDFOR
        RETURN(d_min,y)
END
```

the existence of any structure in the set. In order to improve the efficiency of the computation, some method of placing structure on the point set must be developed. Range searching, as discussed by Sedgewick [41], is a general approach for finding records in a set satisfying some search criterion based on the attributes of the records. Distance computation is merely one application of that technique. An efficient implementation of a range-searching method generally involves the two separate phases of preprocessing and the range-searching itself. The preprocessing phase may employ any of several strategies [41] to structure the point set for more efficient searching. One method for structuring the set is to construct a grid and divide the points into

smaller sets corresponding to the cell to which they belong. Notably, the grid method can be used to capture proximity relations in the point set while limiting the amount of searching required. The time required for this preprocessing phase may be substantial when the number of points in the approximation is high.

Following the technique outlined in Algorithm 2.4, the points in $\mathcal{A}_\varepsilon$ can be generated in a fashion which provides proximity information about the points without the need of a separate preprocessing phase.

**Definition 3.1 (Proximal disk)** *Given some composite, contractive affine transformation $S$ and a preimage $x_0$, with $D_{max} = \max\{\| x_0 - y \|: \ y \in \mathcal{A}_\varepsilon\}$ then if subsequent transformations $T_i \in \mathcal{T}$ are composed using premultiplication, the image points $T_{i_n} \cdots T_{i_1} S(x_0)$ will lie within a disk of radius $r$. This disk, centered at $S(x_0)$, is called the proximal disk for $S$. The value of $r$ is determined as $\lambda_S \times D_{max}$. The existence of these disks follows directly from Theorem 2.1.*

Contained within each disk is a set of one or more points. Let $M$ be the number of transformation applications required to achieve precision $\varepsilon$ in the approximation, and $m$ be the number required to achieve precision $\mu, \mu > \varepsilon$. The proximal disk of radius $\mu$ will contain $N^{(M-m)}$ points in $\mathcal{A}_\varepsilon$, where $N$ is the number of transformations in $\mathcal{T}$. Specifically in the case of the Sierpiński gasket (see Figure 3.2), if 3 transformation applications are required to achieve precision $\varepsilon$ and 2 transformation applications are required to achieve precision $\mu$, then each disk will contain 3 points.

If the attractor is disconnected in the sense of Definition 2.11, these proximal disks fulfill the requirements of the *loci of proximity* discussed by Preparata and Shamos [34].

Distance computations can be improved by the use of the proximal disks to discard those subsets which cannot contain the point with minimum distance to $x$. The proximal disk defined by the composite transformation $S$ may be excluded in distance computations if, when computing distance $d(x, \mathcal{A}_\varepsilon)$, the following relation holds:

$$d(x, S(x_0)) - r > d_{min} \tag{3.3}$$

This concept is illustrated in Figure 3.3. Two different algorithms have been implemented based on this principle.

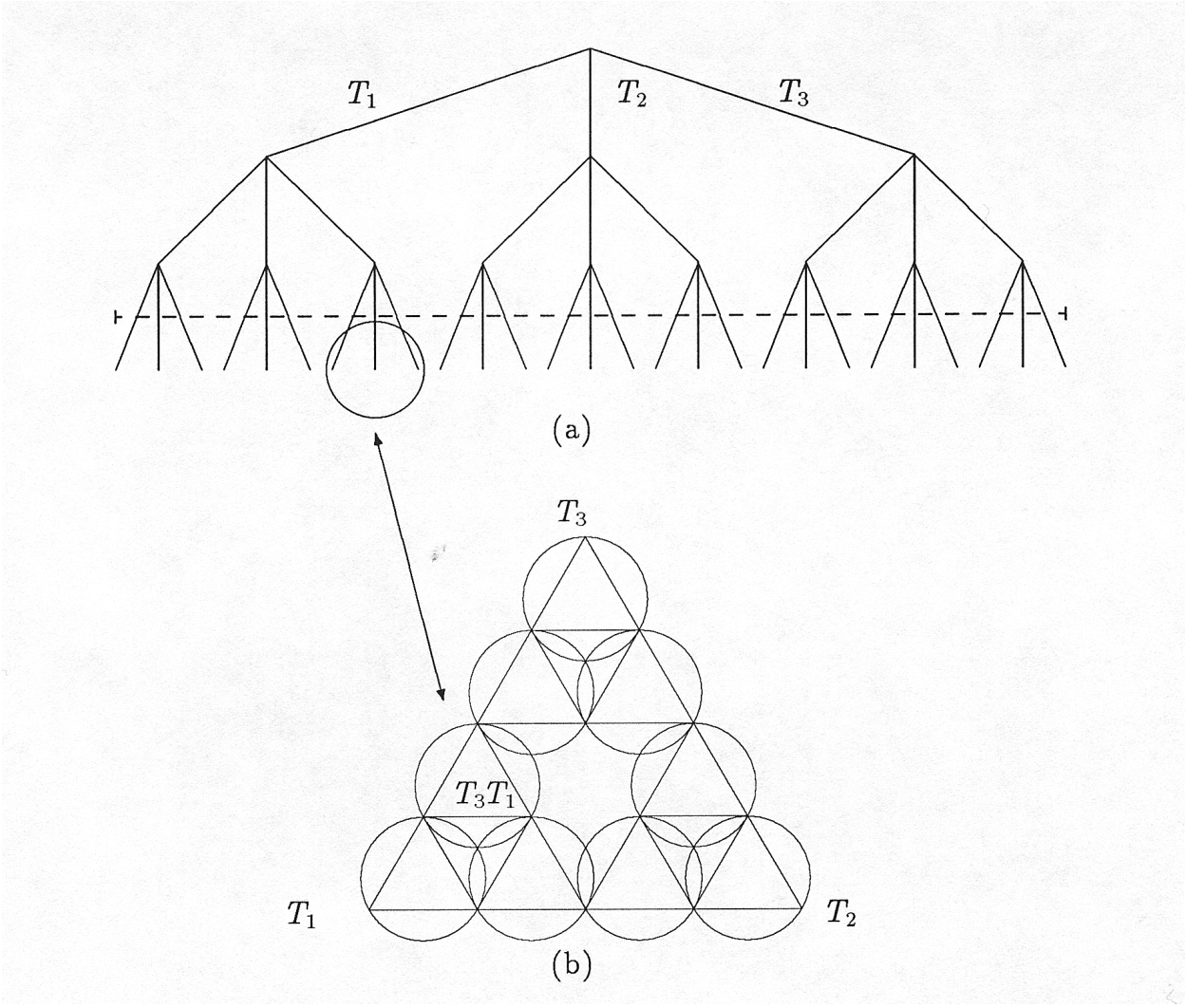



Figure 3.2: The tree of transformations associated with the generation of the Sierpiński gasket is shown in (a) where the dashed line indicates the level at which the points are stored into subsets. The partitioning of the attractor indicated by those subsets is illustrated by the proximal disks in (b).

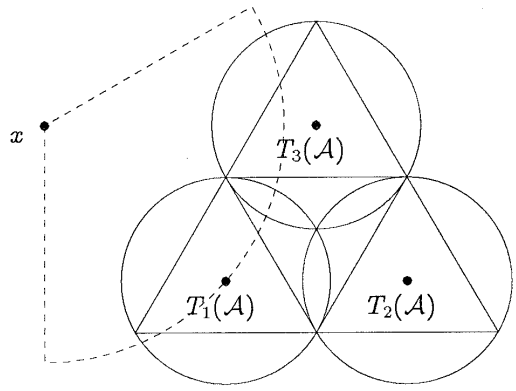



Figure 3.3: When computing $d(x, \mathcal{A}_\varepsilon)$, all points in the subset $T_2(\mathcal{A})$ can be discarded, since it fails the test of Formula 3.3 , as no point in its proximal disk is within the minimum distance indicated by the dashed arc.

### 3.3.2 Fixed subset method

The first approach is closely related to the grid method described by Sedgewick [41] for range searching problems. As points in the attractor approximation are generated, each is stored in relation to its proximal disk. Figure 3.2 illustrates how an attractor can be partitioned according to these proximal disks. The details of this method are given in Algorithm 3.2.

The value of $\mu$ upon which the partition is based is an important choice, since the number of points in each subset is a crucial factor in computations. If the subdivision is too coarse, each subset will contain too many points and the time spent searching in the subsets will be disproportionately high. However if the subdivision is too fine, the number of subsets will be too large and a disproportionate amount of time will be spent selecting the subsets to be searched. For his purposes, Sedgewick [41] advocates choosing a size based on a constant ratio of the number of items to the number of subdivisions. Tables 3.1 and 3.2 show some experimental results in this area. The subdivision criterion is expressed as a value of computation precision ( column 1 in Tables 3.1 and 3.2 ) rather than a number of points. This approach was found to be much more appropriate for the purpose at hand, especially in the case of the

| Attractor: Sierpiński gasket<br>Precision of computation: 0.0078 | | | |
|---|---|---|---|
| Precision threshold | Number of subsets | Points per subset | Total time (minutes) |
| 0.0156 | 729 | 3 | 11.73 |
| 0.0312 | 243 | 9 | 4.33 |
| 0.0624 | 81 | 27 | 2.37 |
| 0.1248 | 27 | 81 | 3.31 |

Table 3.1: Experimental results obtained using different subset sizes with the **Fixed-subset** method for computation of distance to the Sierpiński gasket attractor.

| Attractor: Black Spleenwort fern<br>Precision of computation: 0.0547 | | | |
|---|---|---|---|
| Precision threshold | Number of subsets | Points per subset | Total time (minutes) |
| 0.1641 | 3300 | 11.79 | 61.33 |
| 0.2188 | 1994 | 19.51 | 38.75 |
| 0.4376 | 634 | 61.36 | 22.63 |
| 0.8752 | 148 | 262.85 | 44.84 |

Table 3.2: Experimental results obtained using different subset sizes with the **Fixed-subset** method for computation of distance to the Black Spleenwort fern attractor.

Black Spleenwort fern attractor. Based on the experimental results, a default value of 8 times the precision of computation was chosen for as the threshold for subset creation.

Rather than compute the distance $d(x, \mathcal{A}_\varepsilon)$ directly, the **Fixed-subset** method performs the calculation in two steps. First, compute $d(x, \mathcal{A}_\mu)$ where $\mu > \varepsilon$. On the basis of the initial computation, only those subsets which pass the test given in Formula 3.3 will be retained for further examination at precision $\varepsilon$.

**ALGORITHM 3.2** **Fixed-subset(*SubsetList*,*n*,*x*)**
**PURPOSE** Estimate the distance of a point $x$ to the attractor

| | | |
|---|---|---|
| Arguments | *SubsetList* | list of subsets contained in $\mathcal{A}_\varepsilon$ where each element is composed of a point $spt[i]$ in the subset; the distance $sd[i] = d(x, spt[i])$; an estimate $sr[i]$ of the radius of the disk enclosing the subset; a list of points SFA[$i$] contained in the subset; and the number of elements $sn[i]$ in the subset. |
| | $n$ | number of elements in *SubsetList* |
| | $x$ | point of interest |
| Output | $d_{min}$ | distance estimate with $\mid d_{min} - d(x, \mathcal{A}) \mid \leq \varepsilon$ |
| | $y$ | a point in $\mathcal{A}_\varepsilon$ with $d(x, y) = d_{min}$ |
| Local variables | HUGE | a sufficiently large number |
| | $i$ | list index for *SubsetList* |
| | $j$ | list index for each SFA[$i$] |
| | $sd_{min}$ | minimum of all $sd[i]$ |
| | *dist* | distance between $x$ and SFA[$i$][$j$] |

```
BEGIN
    d_min = HUGE
    sd_min = HUGE
    FOR i = 1,...,n DO
        sd[i] = d(x, spt[i])
        IF (sd[i] < sd_min) THEN
            sd_min = sd[i]
        ENDIF
    ENDFOR
    FOR i = 1,...,n DO
        IF (sd[i] - sr[i] ≤ d_min) THEN
            FOR j = 1,...,sn[i] DO
                dist = d(x, SFA[i][j])
                IF (dist < d_min) THEN
                    d_min = dist
                    y = SFA[i][j]
                ENDIF
            ENDFOR
        ENDIF
    ENDFOR
    RETURN(d_min, y)
END
```

### 3.3.3 Adaptive subdivision method

This method applies the idea of proximal disks in an incremental way, such that a series of progressively more accurate approximations are calculated and measured. At each step, a distance calculation of the form $d(y, \mathcal{A}_{\mu_i})$, where $\mu_i > \mu_{i+1} > \ldots > \varepsilon$, is performed and all possible proximal disks are eliminated before proceeding to the next step. In this way, only those points in the attractor which are required to evaluate $d(y, \mathcal{A}_\varepsilon)$ are ever computed. This method has a parallel with $n$-dimensional trees applied to range-searching problems, but instead of searching a static tree each time, the **Adaptive-subdivision** method rebuilds the tree for each point $y$.

The selection of the starting point $x_0$ and hence the initial distance estimate for the attractor is crucial. The distinction between $x_0 \in \mathcal{A}$ and $x_0 \notin \mathcal{A}$ is very important because the minimal selection for $D_{max}$ will minimize computations and affect how those computations are actually performed.

At each level of the tree, image points are calculated and the minimum distance is calculated from that set of points in order to make comparisons and to discard as many points as possible according to Formula 3.3. Since the points computed as part of the earlier approximations will generally not be sufficiently close to the set $\mathcal{A}_\varepsilon$ to be considered as members, it is important that the minimum distance is reset before the next approximation is calculated. Figure 3.4 illustrates the errors which would result if this minimum distance were not reset.

For this same reason, points which reach the desired precision are maintained until all the leaf nodes have that precision. This is important in the case when the IFS contains transformations with different contraction ratios. The procedure is presented in Algorithm 3.3. Hepting et al. [22] described the simpler, more restrictive case with the starting point $x_0 \in \mathcal{A}$.

#### The use of point-to-point coherence

Let $y_{min}(x)$ denote a point $y \in \mathcal{A}_\varepsilon$ closest to $x$. Although the point $y_{min}(x)$ may change as $x$ is slightly perturbed, the distance $d(x, \mathcal{A})$ is a continuous function of $x$. If $x'$ is a new point which lies in the vicinity of $x$, the value $d(x', y_{min}(x))$ provides a

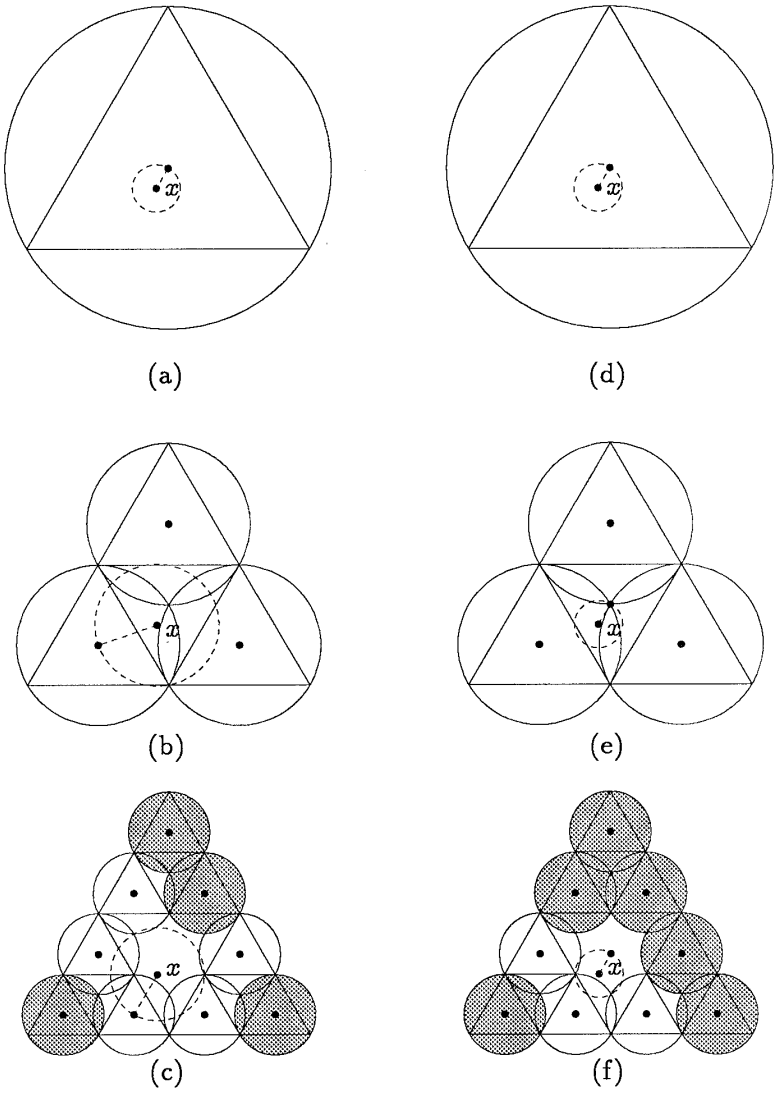



Figure 3.4: Illustration of the computation of $d(x, \mathcal{A}_\varepsilon)$ using the **Adaptive-subdivision** method with 3 levels of approximations containing 1, 3, and 9 points respectively. At each stage, the dashed line is drawn between $x$ and the point in $\mathcal{A}_{\mu_i}$ which is closest to $x$ and anything contained within the dashed circle requires further consideration. Those circles shaded grey are not considered. The first column shows proper operation of the method as the minimum distance is reset after each level. The second column illustrates how the incorrect point is selected if the minimum distance is maintained throughout the computations.

| **ALGORITHM 3.3** | **Adaptive-subdivision**$(x, \varepsilon, x_0, \mathcal{T}, \mathcal{L}, N, R)$ | |
|---|---|---|
| **PURPOSE** | Estimate the distance of a point $x$ from the attractor | |
| Arguments | $x$ | point |
| | $\varepsilon$ | precision of computation |
| | $x_0$ | preimage for attractor approximation |
| | $\mathcal{T}$ | finite set of affine transformations,$T_i$ |
| | $\mathcal{L}$ | array of Lipschitz constants, $\lambda_i$, associated with the transformations |
| | $N$ | the number of transformations |
| | $R$ | radius of $\mathcal{A}$ |
| Output | $d_{min}$ | distance estimate with $\mid d_{min} - d(x, \mathcal{A}) \mid \leq \varepsilon$ |
| | $y$ | a point in $\mathcal{A}_\varepsilon$ with $d(x, y) = d_{min}$ |
| Global symbols | $Id$ | identity transformation |
| | INCLUDE | keep the point for further calculation |
| | EXCLUDE | discard the point |
| | HOLD | copy the point to further levels |
| Local variables | $List$ | a list of elements, each of which contains an affine transformation $T[i]$; an estimate $r[i]$ for the radius of $T[i](\mathcal{A})$; the distance $d[i] = d(x, T[i](x_0))$ and a flag $f[i]$ to indicate its status. |
| | $n$ | number of elements in $List$ |
| | $i$ | list index |

BEGIN

$List = \{Id, R, d(x, x_0)$,INCLUDE$\}$

$n = 1$

$d_{min} = d(x, x_0)$

$y = x_0$

$List =$**Process-list**$(List, n, \mathcal{T}, \mathcal{L})$

**Process-elements**$(List, n, \mathcal{T}, \mathcal{L})$

RETURN $(d_{min}, y)$

END

| **Procedure** | **Process-list**$(List, n, \mathcal{T}, \mathcal{L})$ | |
|---|---|---|
| **Purpose** | Create a new list for the subsets $(T_j T_i)(\mathcal{A})$ of $T_i(\mathcal{A})$ | |
| Arguments | $List$ | a list of elements describing $T_i(\mathcal{A})$ |
| | $n$ | number of elements in $List$ |
| | $\mathcal{T}$ | finite set of contractive affine transformations |
| | $\mathcal{L}$ | set of contraction factors for the transformations |
| Output | $\widehat{List}$ | new list of elements |
| Local variables | $i, j, k$ | list indices |

```
BEGIN
    k = 0
    FOR i = 1, ..., n DO
        IF (f[i] == INCLUDE) THEN
            FOR j = 1, ..., N DO
                T̂[k] = T_j T̂[i]
                r̂[k] = λ_j r̂[i]
                d̂[k] = d(x, T̂[k](x_0))
                k = k + 1
            ENDFOR
        ENDIF
    ENDFOR
    RETURN (List^)
END
```

good estimate of the distance $d(x', \mathcal{A})$. The use of this estimate for the initial distance in the **Adaptive-subdivision** algorithm can greatly speed the algorithm operation. The improved accuracy of $d_{min}$ will allow some regions to be rejected earlier than if $d_{min}$ was set to $d(x, x_0)$. It is essential that $d_{min}$ is calculated as $d(x', y_{min}(x))$ rather than set to $d(x, \mathcal{A})$, since that value might be less than $d(x', \mathcal{A})$.

The operation of the **Adaptive-subdivision** method with and without point-to-point coherence is illustrated in Figure 3.5. Notice that even fewer points need to be generated as a result of using the better initial distance estimate. It is not possible to apply this technique to the **Fixed-subset** method since all computations in the

| **Procedure** | **Process-elements**$(List, n, \mathcal{T}, \mathcal{L})$ | |
|---|---|---|
| **Purpose** | Process the tree | |
| Arguments | $List$ | a list of elements describing $T_i(\mathcal{A})$ |
| | $n$ | number of elements in $List$ |
| | $\mathcal{T}$ | finite set of contractive affine transformations |
| | $\mathcal{L}$ | set of contraction factors for the transformations |
| Output | $d_{min}$ | distance estimate with $\mid d_{min} - d(x, \mathcal{A}) \mid \leq \varepsilon$ |
| | $y$ | a point in $\mathcal{A}_\varepsilon$ with $d(x, y) = d_{min}$ |
| Local variables | $i, j, k$ | list indices |

```
        DO
                FOR i = 1, ..., n DO
                        IF (d[i] < d_min) THEN
                                d_min = d[i]
                                y = T[i](x_0)
                        ENDIF
                ENDFOR
                FOR i = 1, ..., n DO
                        IF (r[i] < ε) THEN
                                f[i] = HOLD
                        ENDIF
                        IF (d_min ≤ d[i] − r[i]) THEN
                                f[i] = EXCLUDE
                        ENDIF
                ENDFOR
                List =Process-list(List, n, T, L)
                IF (n > 0) THEN
                        d_min = d(x, x_0) /* reset the distance */
                ENDIF
        WHILE (n > 0)
END
```

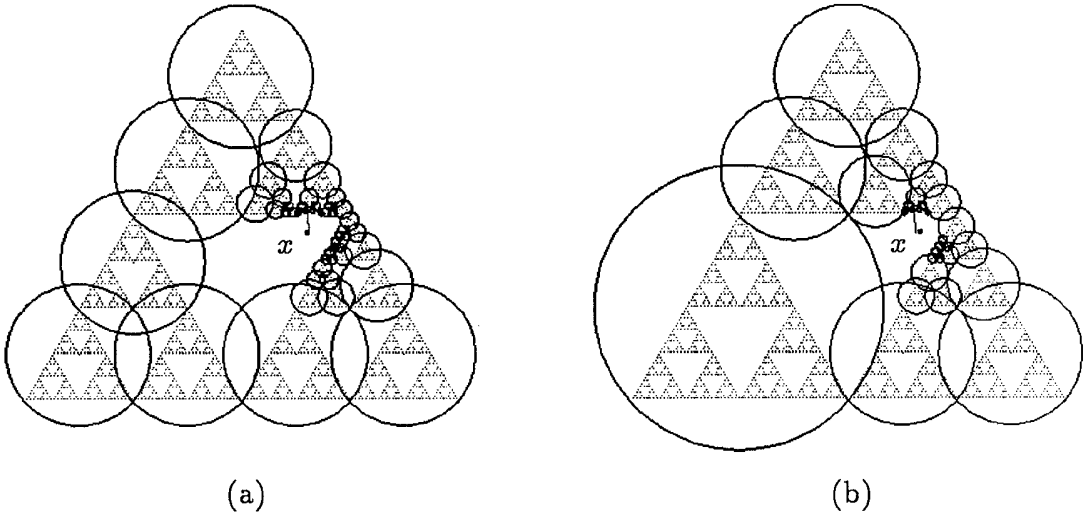



Figure 3.5: Computing distance from a point $x$ to the Sierpiński gasket, using the adaptive subdivision algorithm: (a) without point-to-point coherence and (b) with point-to-point coherence.

first phase are always required.

These algorithms were tested [1] for performance on a variety of attractors with varying degrees of precision. It is difficult to make comparisons between different attractors, due the the wide variety of parameters including size of the attractor, the number of transformations, and the contractivity of the transformations. Comparisons, however, can be made on the basis of computation time and memory for the algorithms on the various attractors. The test images for the attractors are shown in Figures 3.6,3.7,3.8,and 3.9. Three of the attractors have been used previously in examples, the fourth attractor, resembling a star, was chosen because it is overlapping. The transformations which define it are given in Table 3.3.

The experimental results clearly indicate that it is possible to significantly decrease computation times from those required with the **Basic-distance** method ( Tables 3.4, 3.5, 3.6, and 3.7 ). Both methods based on the proximal disk approach fared very well. The **Fixed-subset** method has the advantage that all computations required to build the approximation are performed only once, at the beginning. It has the

[1]The experiment was performed on a Silicon Graphics 4D-25G computer with 12Mb of RAM.

$$T_1 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.00 & 0.500 & 0.000 \\ -0.500 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.500 & 0.000 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.250 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_4 = \begin{bmatrix} 0.354 & 0.354 & 0.000 \\ -0.177 & 0.177 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix}$$

$$T_5 = \begin{bmatrix} 0.354 & \text{-}0.354 & 0.000 \\ 0.177 & 0.177 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix}$$

Table 3.3: IFS for the Star attractor.

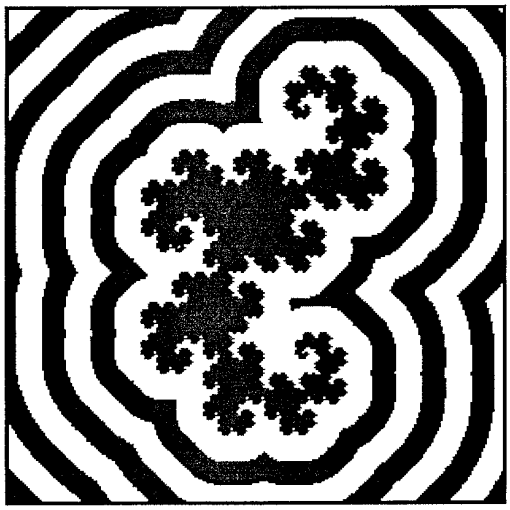


Figure 3.6: Distance computed to an approximation of the Dragon curve attractor with $\varepsilon = 0.0078$.

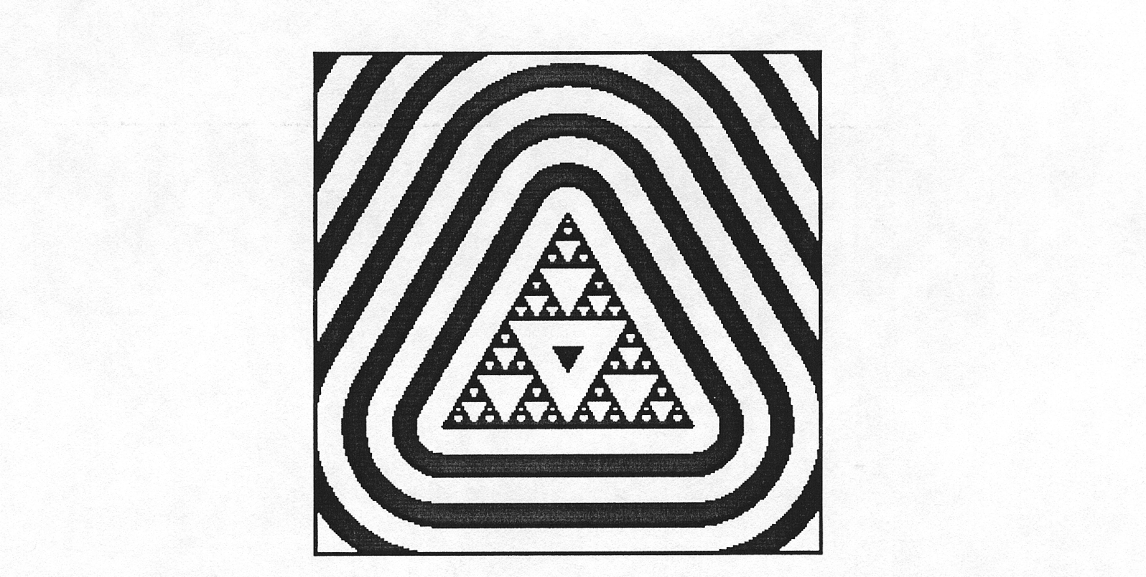


Figure 3.7: Distance computed to an approximation of the Sierpiński gasket attractor with $\varepsilon = 0.0078$.

Figure 3.8: Distance computed to an approximation of the Black Spleenwort fern attractor with $\varepsilon = 0.0547$.

| Attractor: Dragon curve<br>$D_{max}$: 0.9060<br>Grid size: 256x256 | | | |
|---|---|---|---|
| Precision of calculation | Computation method | Total Time (minutes) | Memory (Kbytes) |
| 0.0078 | basic | 221.35 | 512.00 |
| | fixed subset | 9.01 | 512.00 |
| | adaptive | 17.06 | 6.28 |
| | coherence | 9.75 | 5.53 |
| 0.0156 | basic | 55.37 | 128.00 |
| | fixed subset | 4.68 | 128.00 |
| | adaptive | 13.67 | 5.72 |
| | coherence | 8.08 | 5.16 |
| 0.0313 | basic | 12.52 | 32.00 |
| | fixed subset | 3.20 | 32.00 |
| | adaptive | 10.54 | 5.72 |
| | coherence | 6.60 | 4.41 |
| 0.0625 | basic | 3.18 | 8.00 |
| | fixed subset | 2.20 | 8.00 |
| | adaptive | 7.59 | 5.53 |
| | coherence | 5.15 | 4.03 |

Table 3.4: Experimental results for computation of distance to the Dragon curve attractor, in terms of time and space.

drawback of substantial memory requirements. The **Adaptive-subdivision** has the clear advantage of significantly reduced memory requirements and has proven to be much faster when the precision of the calculations make the amount of available memory a very real concern.

## 3.4 Visualization techniques

Distance provides a means for examining the complement of the attractor, $X \setminus \mathcal{A}$, and the visualization techniques discussed here will focus on that problem. The technique of distance estimation has been used successfully to create images of Julia

| Attractor: Sierpiński gasket<br>$D_{max}$: 0.6663<br>Grid size: 256x256 | | | |
|---|---|---|---|
| Precision of calculation | Computation method | Total Time (minutes) | Memory (Kbytes) |
| 0.0078 | basic | 29.53 | 68.34 |
| | fixed subset | 2.37 | 68.34 |
| | adaptive | 8.72 | 9.75 |
| | coherence | 7.50 | 8.63 |
| 0.0156 | basic | 8.66 | 22.78 |
| | fixed subset | 1.35 | 22.78 |
| | adaptive | 6.21 | 7.50 |
| | coherence | 5.43 | 6.94 |
| 0.0313 | basic | 2.90 | 7.59 |
| | fixed subset | 0.88 | 7.59 |
| | adaptive | 4.34 | 6.66 |
| | coherence | 3.87 | 4.97 |
| 0.0625 | basic | 1.10 | 2.53 |
| | fixed subset | 0.64 | 2.53 |
| | adaptive | 3.01 | 5.25 |
| | coherence | 2.81 | 3.84 |

Table 3.5: Experimental results for computation of distance to the Sierpiński gasket attractor, in terms of time and space.

sets using the DEM/J method [33].

Characteristic to a field of distance values is a collection of equidistant lines. The most useful methods for interpreting this information exploit this property. Examples of this idea include selection of grey values to highlight contour lines in the data, or to create bands to indicate equal distance intervals. An example of this technique is shown with the flame attractor in Figure 3.10, which is defined by the IFS given in Table 3.8. Another effective technique is to select grey scales which flow smoothly together, giving a sense of the continuity in the distance values. Figure 3.11 shows a sample of an image created using this method. A further useful interpretation of the field data is to interpret each distance as a height. This technique was used in

| Attractor: Black Spleenwort fern<br>$D_{max}$: 5.5394<br>Grid size: 256x256 | | | |
|---|---|---|---|
| Precision of calculation | Computation method | Total Time (minutes) | Memory (Kbytes) |
| 0.0547 | basic | 527.48 | 1215.69 |
| | fixed subset | 22.21 | 1215.69 |
| | adaptive | 36.27 | 100.41 |
| | coherence | 26.83 | 100.41 |
| 0.1094 | basic | 130.23 | 296.66 |
| | fixed subset | 11.89 | 296.66 |
| | adaptive | 27.11 | 33.19 |
| | coherence | 21.37 | 33.19 |
| 0.2188 | basic | 34.09 | 78.03 |
| | fixed subset | 7.46 | 78.03 |
| | adaptive | 20.55 | 11.81 |
| | coherence | 17.07 | 11.53 |
| 0.4375 | basic | 8.00 | 19.81 |
| | fixed subset | 4.11 | 19.81 |
| | adaptive | 13.85 | 8.91 |
| | coherence | 12.28 | 8.91 |

Table 3.6: Experimental results for computation of distance to the Black Spleenwort fern attractor, in terms of time and space.

Figure 3.12 to create a relief image of the attractor against its surrounding area.

An alternative use of distance from a set is to exclude points outside the attractor from consideration for membership in the attractor. Simply, if $r = d(x, \mathcal{A})$ then a disk of radius $r$ about the point $x$ can be drawn which is completely outside of $\mathcal{A}$. Any points contained in the disk need not be considered further. All points in the area of interest are considered in this manner. This technique is an adaptation of the method for computing images of Julia sets described by Fisher [17]. The effect of this procedure is to create a quick outline of the set, which is refined until the desired precision is reached. The radius of each disk is determined by the distance from its center to the attractor. For aesthetic considerations, a scaling function may

Figure 3.9: Distance computed to an approximation of the Star attractor with $\varepsilon = 0.0098$.

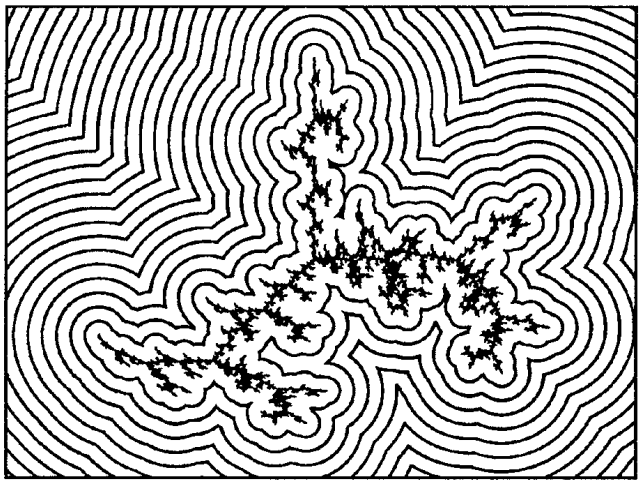


Figure 3.10: The Flame attractor and its complement rendered using contour lines.

| Attractor: Star<br>$D_{max}$: 1.0615<br>Grid size: 256x256 | | | |
|---|---|---|---|
| Precision of calculation | Computation method | Total Time (minutes) | Memory (Kbytes) |
| 0.0098 | basic | 1059.43 | 2441.41 |
| | fixed subset | 30.03 | 2441.41 |
| | adaptive | 33.03 | 1215.94 |
| | coherence | 17.59 | 1215.94 |
| 0.0195 | basic | 214.47 | 488.28 |
| | fixed subset | 19.95 | 488.28 |
| | adaptive | 26.94 | 393.20 |
| | coherence | 14.19 | 393.20 |
| 0.0391 | basic | 42.64 | 97.66 |
| | fixed subset | 18.40 | 97.66 |
| | adaptive | 20.86 | 123.47 |
| | coherence | 11.02 | 123.47 |
| 0.0781 | basic | 7.46 | 19.53 |
| | fixed subset | 6.52 | 19.53 |
| | adaptive | 15.13 | 40.78 |
| | coherence | 8.69 | 39.84 |

Table 3.7: Experimental results for computation of distance to the Star attractor, in terms of time and space.

$$T_1 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.309 & -0.394 & 0.000 \\ 0.394 & 0.309 & 0.000 \\ 1.000 & 0.000 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.458 & -0.202 & 0.000 \\ 0.202 & 0.229 & 0.000 \\ -0.710 & -0.710 & 1.000 \end{bmatrix} \quad T_4 = \begin{bmatrix} 0.150 & 0.300 & 0.000 \\ 0.000 & 1.000 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix}$$

Table 3.8: IFS for the Flame attractor.

Figure 3.11: The Black Spleenwort fern and its complement rendered by assigning a smooth ramp of grey scales to the distance values.

Figure 3.12: Distance to the Sierpiński gasket computed and interpreted as height values. A colour version of this image appears in the original of this thesis as Figure **??**.

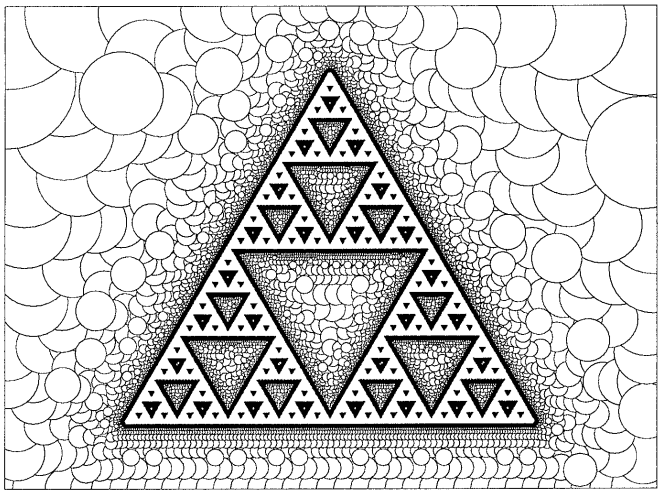


Figure 3.13: Sierpiński gasket with disks where the radii are scaled to $\frac{1}{4}$ of the distance from the center of each disk to the attractor.

be applied to reduce the radii of the disks. The order in which the disks is computed is indicated by the tiling of the disks, as in Figure 3.13.

The visual appeal of the images is enhanced by interpreting the disks as spheres, as done in Figures 3.14, 3.15 and 3.16.

### 3.4.1 Transformation indices

Each point in the attractor is defined by a sequence of transformations, applied to the starting point. Although this transformation index sequence is unique only for disconnected attractors, it can still produce interesting results. When $d(x, \mathcal{A})$ is calculated, a point in the attractor along with its sequence of transformations is associated with that distance value. As an alternative, it is possible to view the plane with values determined by the defining transformations of the points associated with each distance value, as in Figure 3.17.

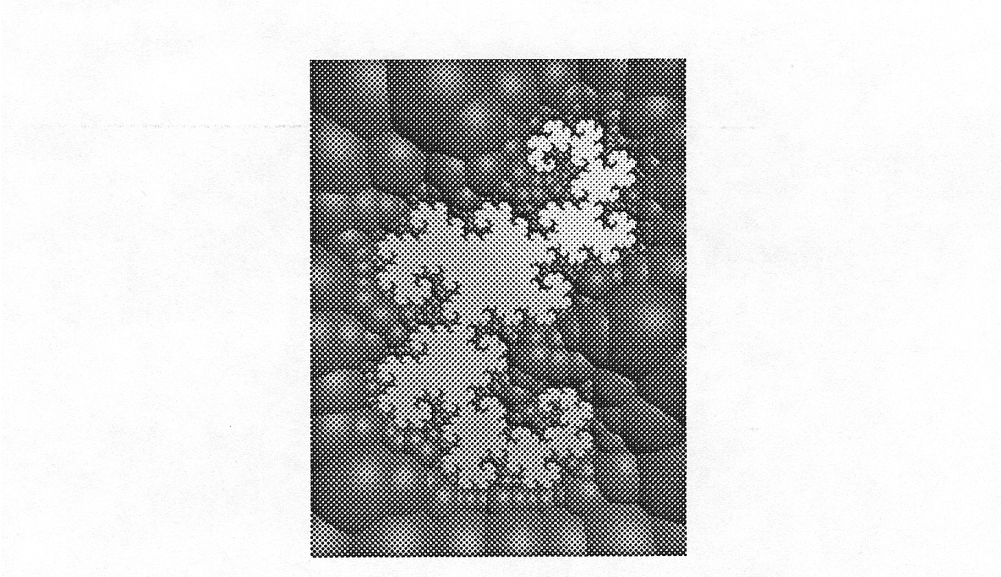


Figure 3.14: The Dragon curve visualized by covering the complement with spheres scaled exponentially. A colour version of this image appears in the original of this thesis as Figure **??**.

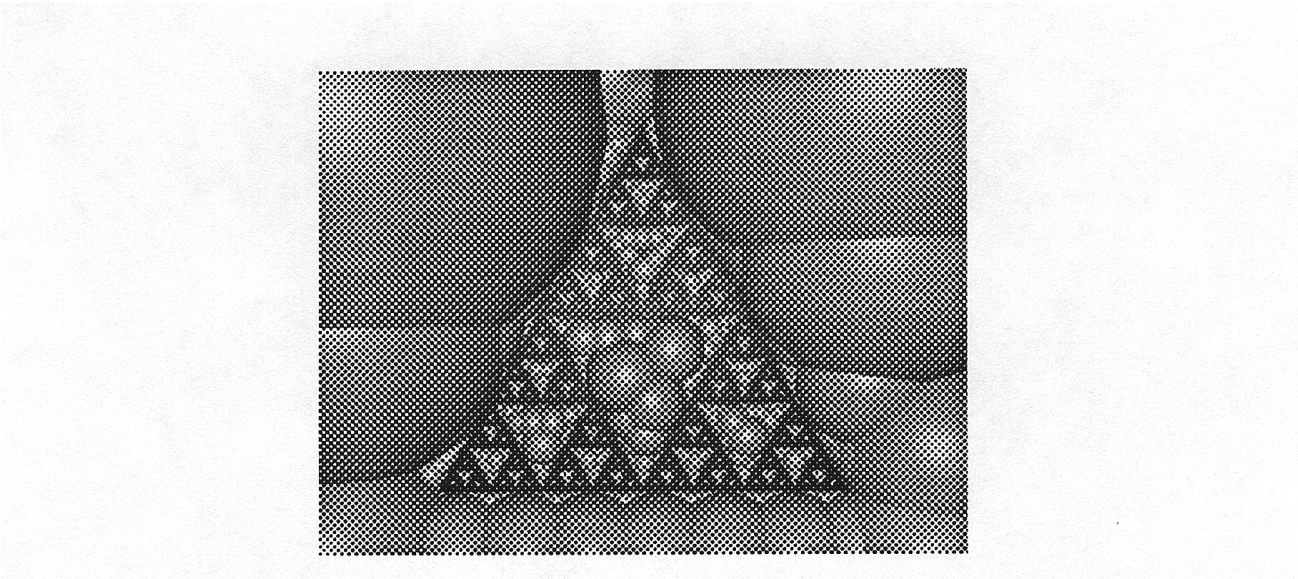


Figure 3.15: The Sierpiński gasket visualized by covering the complement with spheres of radii equal to the distance from their centers to the attractor. A colour version of this image appears in the original of this thesis as Figure **??**.

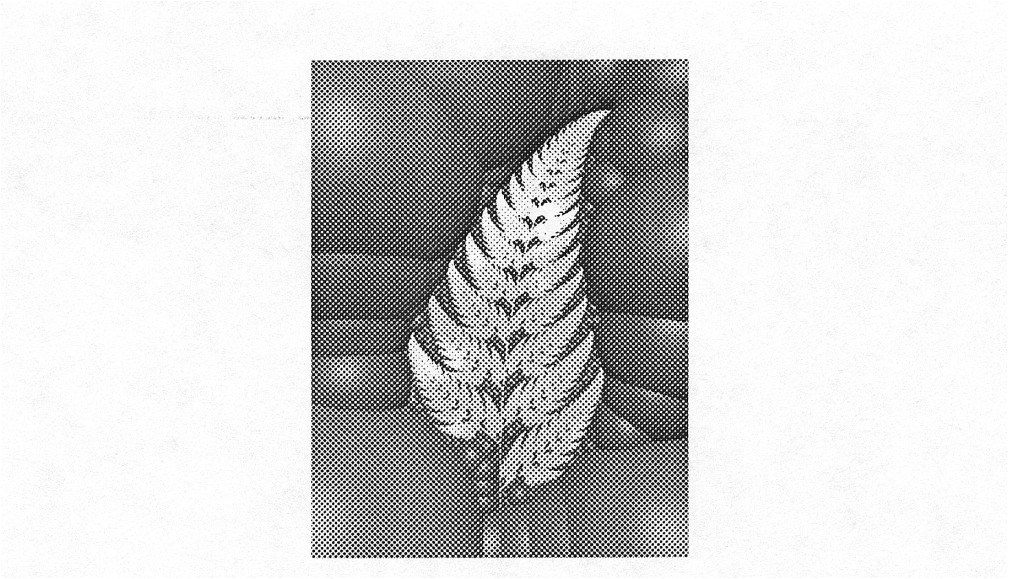


Figure 3.16: The Black Spleenwort fern attractor visualized by covering the complement with spheres. A colour version of this image appears in the original of this thesis as Figure **??**.

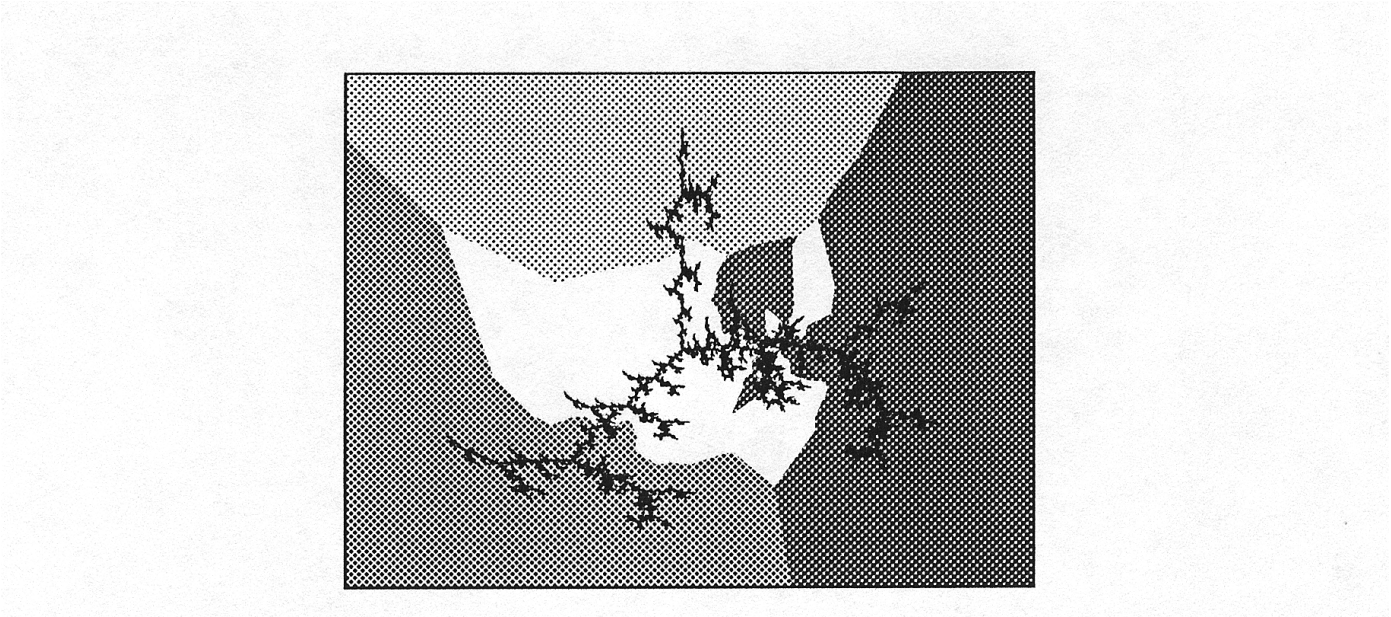


Figure 3.17: The Flame attractor and its complement visualized by examining the last transformation associated with each distance value.

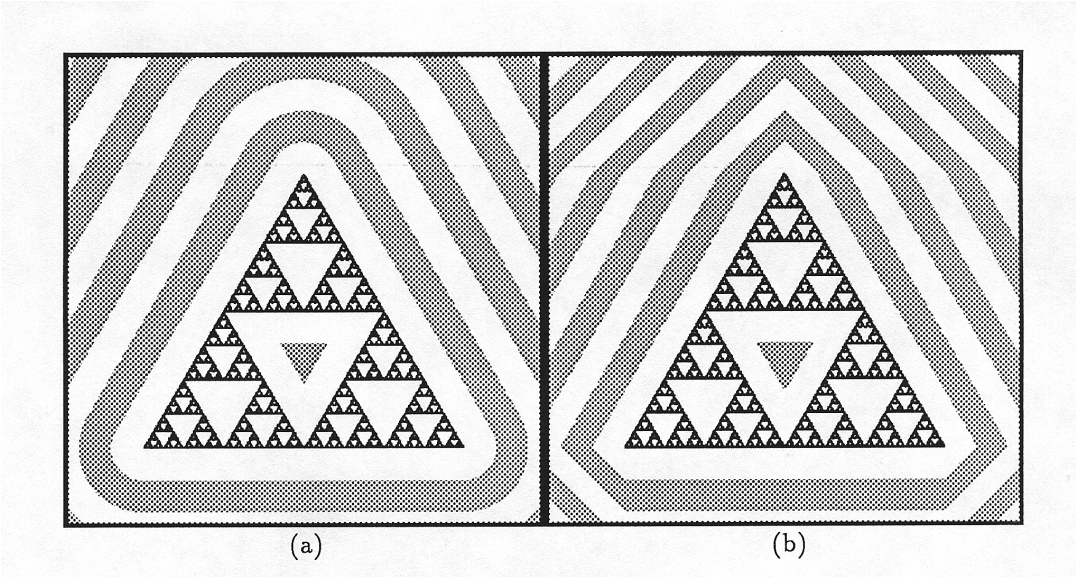



Figure 3.18: A comparison of the distance to the Sierpiński gasket using (a) the Euclidean metric and (b) the Manhattan metric.

### 3.4.2 Metrics

As an insight into how distance is actually measured, the distance may be computed and visualized using a distance function other than the standard Euclidean. Figure 3.18 shows the difference in the complement of the attractor which is seen between the Euclidean and Manhattan metrics. Figure 3.18(b) lacks the smooth curves which are associated with the Euclidean metric. Figure 3.19 illustrates the effect of using the Manhattan metric to eliminate points from the complement of the attractor. It is analogous to Figures 3.14, 3.15 and 3.16. Although the spheres have been replaced by pyramids, the meaning is still the same.

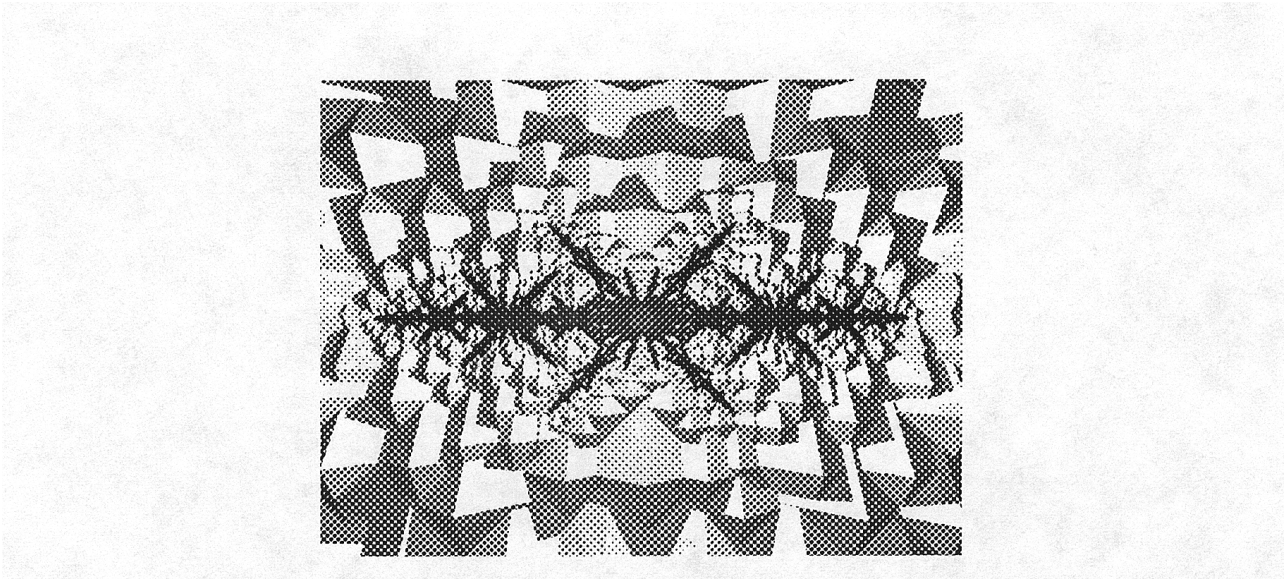


Figure 3.19: The Star attractor visualized by covering the complement with pyramids, which with the Manhattan metric perform the equivalent function to spheres in the Euclidean metric. A colour version of this image appears in the original of this thesis as Figure **??**.

# Chapter 4

# Escape-time Calculation

The escape-time function described in this chapter differs fundamentally from all other methods for visualizing the attractor of an IFS. Whereas the concept of distance is concrete and well understood, the notion of escape-time is more open to interpretation. This chapter shall deal with ways in which the escape-time may be interpreted. Unlike the methods previously described, the calculation of the escape-time function requires the inversion of the original contractive affine transformations. The visualization process is necessarily different, since an attractor, in the sense of Definition 2.9, does not exist for the strictly expansive inverse transformations. Each of the expansive transformations has a fixed point that is shared with its contractive inverse. Given the characterization of the attractor as a collection of the fixed points from $\mathcal{T}$, the escape-time method works by locating the fixed points of the composite transformations from $\mathcal{T}^{-1}$. As a by-product of the root searching process, many details of the complement $X \setminus \mathcal{A}$ can be visualized. The escape-time function for iterated function systems has a close analogue with the function defined for Julia sets. As a means of introduction, this function will be studied along with differences which exist between it and the IFS version. Additionally, this chapter shall examine methods for the computation and interpretation of the escape-time function.

## 4.1 Escape time for Julia sets

Returning to the earlier characterization, the Julia set, for the quadratic mapping $f_c(z) = z^2 + c$ can be viewed as the attractor defined by the nonlinear transformations:

$$\begin{aligned} F_{1_c}(z) &= \sqrt{z - c} \\ F_{2_c}(z) &= -\sqrt{z - c} \end{aligned}$$

Given a point $z$ not in the Julia set, repeated application of the inverse mappings $F_{1_c}^{-1}, F_{2_c}^{-1} : z \mapsto z^2 + c$ will eventually force the image points of $z$ to infinity. The number of iterations necessary to force the point outside of a large disk of radius $R$ is called the escape-time of $z$. In order to make evaluation of the escape-time computationally feasible, a maximum number of iterations, $k$, is defined. A continuous and differentiable extension of the integer valued escape-time $E_c(z)$ function is given by means of the potential function [33]:

$$g_c(z) = \lim_{k \to \infty} \frac{\log \mid f_c^k(z) \mid}{2^k}$$

and the formula

$$E_c(z) = -\log_2 g_c(z) \ .$$

## 4.2 Escape time for IFS attractors

Following the development of the escape-time function for Julia sets, the concept of this function was extended to IFS in the discrete case. It was introduced independently by Barnsley [2, chapter 7] and Prusinkiewicz [37].

It is important to recognize the differences which exist between the quadratic polynomial defining the Julia set and the set of affine transformations used in the IFS and which cause significant differences in the computation and interpretation of the two functions:

1. The union of the two maps $\{ F_{1_c}^{-1}, F_{2_c}^{-1} \}$ which define the Julia set is the single function $f_c(z) = z^2 + c$. Every sequence of transformations for an IFS must be considered separately, since each transformation is generally unique.

2. Affine transformations scale linearly, whereas the quadratic polynomials do not.

Prusinkiewicz [37] showed that given an IFS $\mathcal{T}$ with attractor $\mathcal{A}$, for any point $x \in \mathcal{A}$ there exists at least one sequence of inverse mappings $T_{i_1}^{-1}T_{i_2}^{-1}T_{i_3}^{-1}\ldots$ which will not take $x$ to infinity. In contrast, every sequence of inverse mappings applied to a point $x \notin \mathcal{A}$ will eventually repel it to infinity. These facts allow the definition of a discrete escape-time function for iterated function systems [37].

**Definition 4.1 (Discrete escape-time)** *For a point $x$, the discrete escape-time function [37] is the maximum number of iterations of inverse transformations in $\mathcal{T}^{-1}$ necessary to take the image of $x$ outside of a large disk of radius $R$:*

$$E(x) = \begin{cases} 0 & \text{if } \| x \| \geq R \\ \max_{k=1,\ldots,N} \{E(T_k^{-1}(x)) + 1\} & \text{if } \| x \| < R \end{cases} \tag{4.1}$$

*For points in $\mathcal{A}$, the function value is formally set to $E(x) = \infty$.*

A general method proposed by Prusinkiewicz [37], based on Equation 4.1, is to recursively construct the tree of image points which:

- result from the application of a sequence of inverse transformations $T_{i_1}^{-1}T_{i_2}^{-1}\cdots T_{i_n}^{-1}$ to the sampling point $x$, and
- remain within the large circle of radius $R$.

Consider the example of the IFS of a single transformation, as given in Table 4.1. The image produced after applying the integer-valued escape-time function to the inverse of this transformation is shown in Figure 4.1. The values are assigned according to the following relation, where $\lambda$ is the contraction factor for the transformation.

$$E(x) = k \quad \text{if} \quad \| \lambda^{-(k-1)}x \| < R \leq \| \lambda^{-k}x \|$$

or in fact:

$$E(x) - 1 < \frac{\log(\frac{R}{\|x\|})}{\log \frac{1}{\lambda}} \leq E(x)$$

$$\begin{bmatrix} \text{-}0.521 & 0.293 & 0.000 \\ -0.477 & -0.319 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix}$$

Table 4.1: IFS for the singleton illustrated in Figure 4.1.

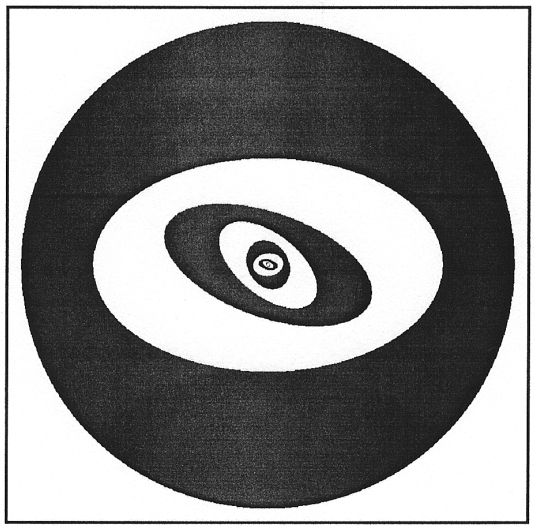


Figure 4.1: A picture of the attractor of an IFS with a single transformation, rendered using the escape-time function.

and this can be rewritten in terms of the component transformations of the particular IFS.

$$\frac{\log R - \log \parallel x \parallel}{\log \parallel T_k^{-1}(x) \parallel - \log \parallel x \parallel}$$

The continuous extension of the escape-time function which follows from this was made by Prusinkiewicz in the paper by Hepting et al. [22].

**Definition 4.2 (Continuous escape-time)** *Given an IFS* $\mathcal{T} = \{T_1, \ldots, T_n\}$*, the*

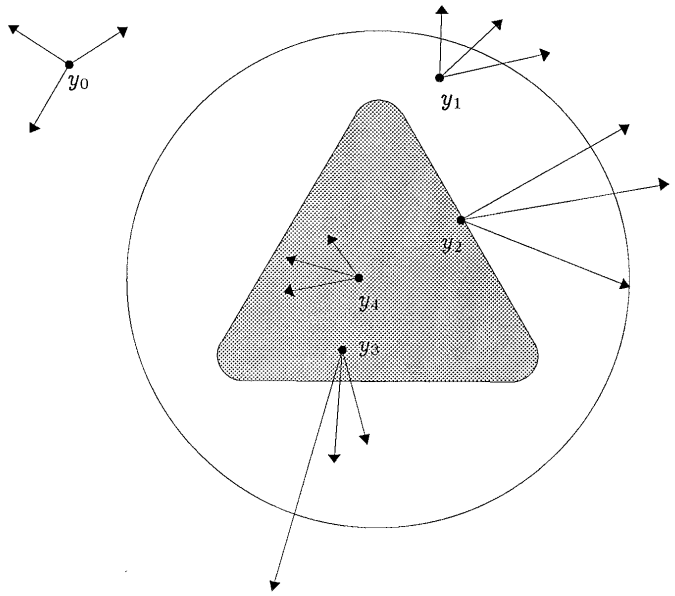



Figure 4.2: In this schematic diagram five points $y_0, \ldots, y_4$ and their images are shown with respect to the inverse transformations of an IFS $\mathcal{T}^{-1}$. The point $y_0$ satisfies the inequality $\parallel y_0 \parallel \geq R$, thus $E_c(y_0) = 0$. Furthermore, $\parallel y_1 \parallel < R$, but all images of $y_1$ have norms greater than $R$, thus, $0 < E_c(y_1) < 1$. Point $y_2$ has two images outside of the disk and one on the boundary of the disk, hence $E_c(y_2) = 1$. The closed curve, of which $y_2$ is a point, collects all points with escape time equal to 1. Hence, $1 < E_c(y_3) < 2$. For $y_4$, $E_c(y_4) > 2$.

---

*continuous escape-time function $E_c(x)$ is defined as follows:*

$$E_c(x) = \begin{cases} 0 & \text{if } \parallel x \parallel \geq R \\ \max_{k=1,\ldots,N} \frac{\log R - \log\|x\|}{\log\|T_k(x)\| - \log\|x\|} & \text{if } \parallel T_k^{-1}(x) \parallel \geq R \ \forall\ k = 1, \ldots, N \\ \max_{k=1,\ldots,N} \{E(T_k^{-1}(x)) + 1\} & \text{if } \parallel x \parallel < R \end{cases}$$

**Theorem 4.1** *The escape-time function $E_c(x)$ depends continuously on the point $x$ and on the parameter $R$.*

**Proof:** proceeds by examining cases shown in Figure 4.2. Only the proof for the dependence on the point $x$ is presented, because the proof for the continuity with

respect to $R$ is similar. Outside the disk $D_R$ the escape-time function $E_c(x)$ is equal to 0, thus it is continuous. On the boundary of the disk continuity holds since the expression:

$$\max_{k=1,\ldots,N} \frac{\log R - \log \parallel x \parallel}{\log \parallel T_k^{-1}(x) \parallel - \log \parallel x \parallel} \tag{4.2}$$

is also equal to 0 for $\parallel x \parallel = R$. If $\parallel x \parallel < R$ and $\parallel T_k^{-1}(x) \parallel$ is strictly greater than $R$ for all $k = 1, \ldots, N$, then $E_c(x)$ is equal to Expression 4.2 in a neighborhood of $x$, therefore also continuous. Due to the recursive nature of the definition of $E_c(x)$ it remains to check that $E_c(x)$ is continuous at points $x$ with $\parallel T_k^{-1}(x) \parallel \geq R$ for all $k = 1, \ldots, N$, and where equality holds for at least one index $k$ (see point $y_2$ in Figure 4.2). Consider a sequence of points $x_i$, $i = 1, 2, \ldots$ such that $x_i \to x$ as $i \to \infty$. From the definition of the escape time,$E_c(x) = 1$ while for sufficiently large indices, $i$, $E_c(x_i)$ is equal to either

$$\max_{k=1,\ldots,N} \frac{\log R - \log \parallel x_i \parallel}{\log \parallel T_k^{-1}(x_i) \parallel - \log \parallel x_i \parallel} \tag{4.3}$$

or

$$1 + \max_{k=1,\ldots,N} E_c(T_k^{-1}(x_i)) \tag{4.4}$$

depending on whether all preimages of $x_i$ have norms greater or equal to $R$ or not. Clearly, Expression 4.3 tends to $E_c(x) = 1$ as $i \to \infty$. To investigate Expression 4.4, note that $\parallel T_k^{-1}(x) \parallel \geq R$, $k = 1, \ldots, N$. Moreover, we already know that $E_c$ is continuous at the points $T_k^{-1}(x)$ for $k = 1, \ldots, N$. Thus, one can conclude that Expression 4.4 must tend to

$$1 + \max_{k=1,\ldots,N} E_c(T_k^{-1}(x))$$

as $i \to \infty$. The value of this limit is equal to 1. □

Associated with the escape-time value for a point $x$ is the point $y$ from which it is repelled most slowly. Intuitively, $y$ is the point closest to $x$. Although this is not true in the general case, Saupe has shown in the paper by Hepting et al. [22] that in the case when all transformations are similarities with the same Lipschitz constant, the escape-time function is actually the logarithm of the distance function. Figure 4.3 shows the field about the Sierpiński gasket computed using first the escape-time function and then the distance function.

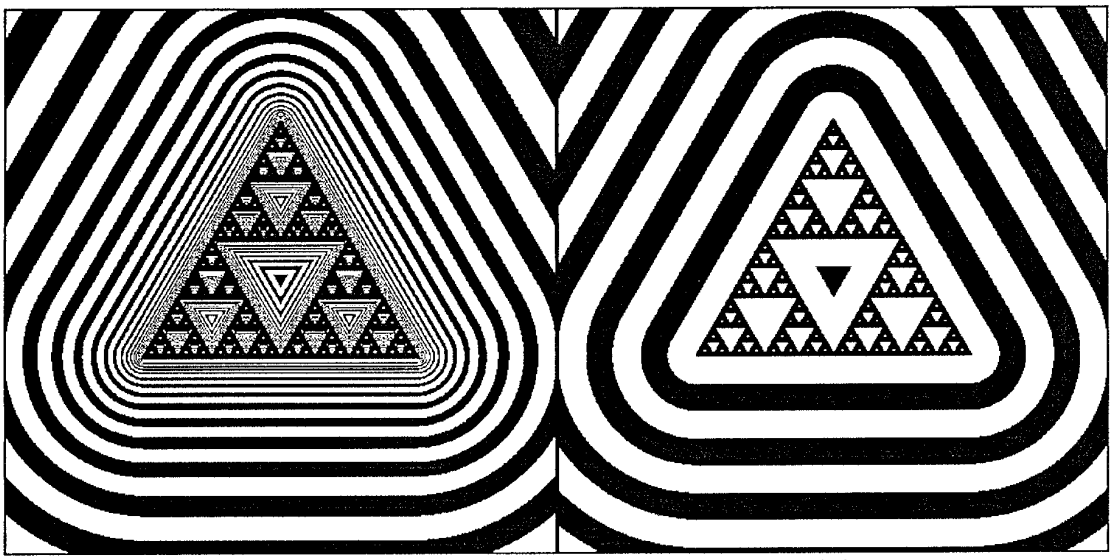



Figure 4.3: The difference between the escape-time function in (a) and the distance function in (b) can be seen easily by comparing the size and spacing of the bands in both images.

## 4.3 Algorithms for computing escape-time

In practical terms, the definition of the escape-time function requires modifications to allow for computation. The original definition does not guarantee that the function can be evaluated in a finite number of steps. Although all points $x_i \notin \mathcal{A}$ will escape from the infinity circle, if $x \in \mathcal{A}$, there will exist one path of infinite length. The collection of all possible paths forms a familiar tree structure with each leaf node representing a composite transformation under which the preimage escapes. Empirical results indicate that only a relatively small number of iterations is required to achieve an adequate approximation to the limit curve. This section will discuss several methods which can be used to compute some finite approximation to the escape-time values for points in X.

### 4.3.1 Basic method

The formula for computation of the escape-time function is modified in two ways to include a finite termination criterion:

1. the tree is only computed to a specified maximum depth. Points which remain inside the infinity circle after this maximum depth are not excluded from the attractor.

2. if the maximum depth is achieved anywhere in the tree, computations are halted. In the case when there may be several paths in the tree which do not repel the preimage, this condition asserts that location of the first such path is sufficient.

The length of the longest branch in this tree is the escape-time value in the discrete case. The general purpose **Basic-escape** method is described in Algorithm 4.1.

As observed with the other visualization methods, the final image produced using the escape-time function is dependent upon the accuracy with which the attractor can be approximated. Although specification of a maximum number of levels for computation can be an effective termination criterion when all of the transformations are similarities with the same Lipschitz constant, it does not provide any measure of the accuracy of the resulting approximation. In order to avoid artifacts in the final picture, it is important that computations not exceed the precision of the output device. The problem becomes more acute when the transformations scale differently as different parts of the attractor may appear with very different resolutions. An imperfect solution to this problem is to select the number of transformation applications $n$ as the minimum number required to achieve a desired precision. Adapting the method used in Chapter 2, it is possible to compute the number of levels using the following formula, where $R_I$ is the radius of the infinity circle and $\lambda_{max}$ is the maximum Lipschitz constant of all the transformations in $\mathcal{T}$.

$$n = \lfloor \frac{\log \frac{\varepsilon}{R_I}}{\log \lambda_{max}} \rfloor$$

The accuracy of the approximation obtained in this case is determined as $\mu = {\lambda_{min}}^n R_I$, which may be considerably less than the originally specified value of $\varepsilon$. The

**ALGORITHM 4.1** **Basic-escape**$(x, n, M, \mathcal{T}^{-1}, N)$
**PURPOSE** Compute the escape-time of a point $x$ from $\mathcal{A}$

| | | |
|---|---|---|
| Arguments | $x$ | point |
| | $n$ | number of levels |
| | $M$ | maximum number of levels |
| | $\mathcal{T}^{-1}$ | inverse transformations |
| | $N$ | the number of transformations |
| Output | $E$ | escape time for point |
| Local variables | $i$ | loop index |
| Functions | inside-infinity() | determine if a point is within the infinity circle |
| | escape-fraction() | compute the fractional part of the escape-time, according to Definition 4.2 |

```
BEGIN
    IF (n == 0) THEN
        E = 0
    ELSE
        E = 0
        tmp = 0
        FOR i = 1, ..., N DO
            x' = T_i^{-1}(x)
            IF (inside-infinity(x')) THEN
                tmp = Basic-escape(x', n - 1, M, T^{-1}, N) + 1
            ELSE
                tmp = escape-fraction(x, x')
            ENDIF
            IF (tmp > E) THEN
                E = tmp
            ENDIF
            IF (E >= (M - n)) THEN
                BREAK
            ENDIF
        ENDFOR
    ENDIF
    RETURN(E)
END
```

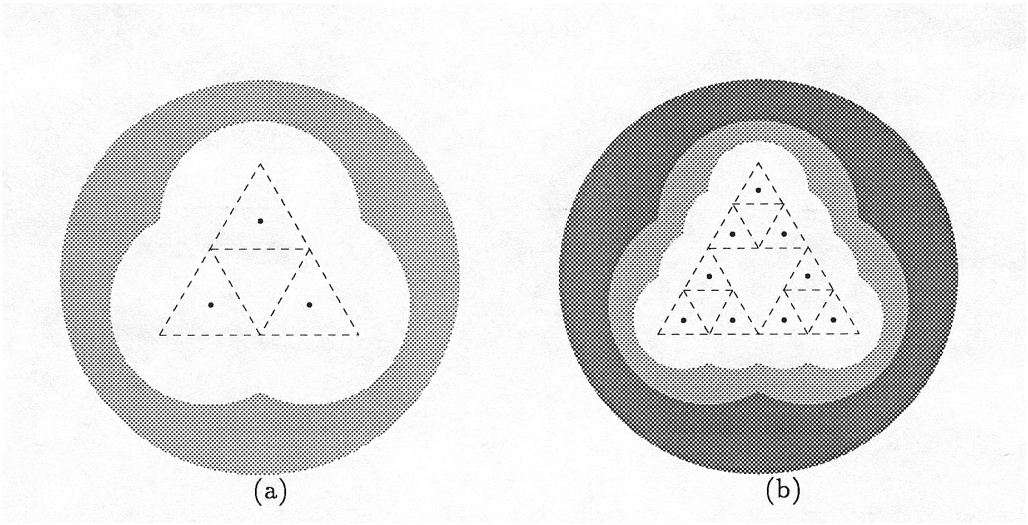



Figure 4.4: The shaded area in (a) represents those points with $0 < E_c \leq 1$, while the points in white are not excluded from the attractor. In (b), the lightly shaded area indicates those points with $1 < E_c \leq 2$. The process of refinement continues until the maximum number of levels is reached.

value of $R_I$ here plays an equivalent role to the value of $D_{max}$ in the **Adaptive-cut** method.

When approximating the attractor with the methods of Chapter 2, pre-multiplication of the contractive maps is used. In order to find the correct sequence of transformations using the inverses, each is applied in reverse order using postmultiplication. The postmultiplication is more efficient than premultiplication, since it does not require the composition of matrices prior to their application.

For any point under consideration, only the minimum number of transformation applications are made in order to resolve the status of the point. In this way, an escape-time image can be thought of as containing as many as $n$ different approximations to the attractor, where $n$ is the maximum number of levels allowed. Figure 4.4 illustrates how higher level approximations are contained within lower ones as more transformation applications are required to resolve points closer to the attractor. The value assigned by $E_c(x)$ to each point in the field is associated with a point in an approximation to the attractor at level $n$. Associated with this point $y \in \mathcal{A}_{\mu_n}$ is a code string of length $n$.

Although the **Basic-escape** method may provide an efficient means to construct the tree of transformations required to evaluate the escape-time function, the overall computation times are still relatively long. By examining the manner in which the transformations are applied to determine the escape-time value for any particular point, methods can be devised which eliminate whole subtrees from consideration.

### 4.3.2 Domain method

It is possible to speed computation of the escape-time function by dividing the plane into *domains* [37, 2].

**Definition 4.3 (Domain)** *A domain is a partition of the plane, $D_i$, such that for any $x \in \mathcal{A} \bigcap D_i$, the transformation $T_i \in \mathcal{T}$ takes the point $x$ to some point of $\mathcal{A} : T_i^{-1}(x) \in \mathcal{A}$.*

The transformation applied to the point is selected based on the following rule:

$$\text{if } x_n \in D_i \text{ then } x_{n+1} = T_i^{-1}(x_n)$$

The success of this method depends on two things:

- *a priori* knowledge of the domain boundaries.
- a simple method to determine current position with respect to those boundaries.

The determination of domain boundaries is a definite problem in this approach. At times the boundary between regions may itself be fractal which makes construction of the boundary very difficult. Although the domain boundaries for the Sierpiński gasket are easy to determine, as shown in Figure 4.5, the boundary for the dragon curve shown in Figure 4.6 may be very difficult to construct. In the case when the attractor is overlapping, the domain divisions may vary depending on the order in which the transformations are applied. In any case, the domain divisions may not be arbitrary. The problem of domain selections is illustrated in Figure 4.7. When feasible, this technique can greatly speed computations since only the subtree for $T_i$ needs to be evaluated, providing a reduction of $\frac{1}{N}$.

One alternative is to use the properties of the inverse, expansive transformations to devise a method for *approximating* values of escape-time function.

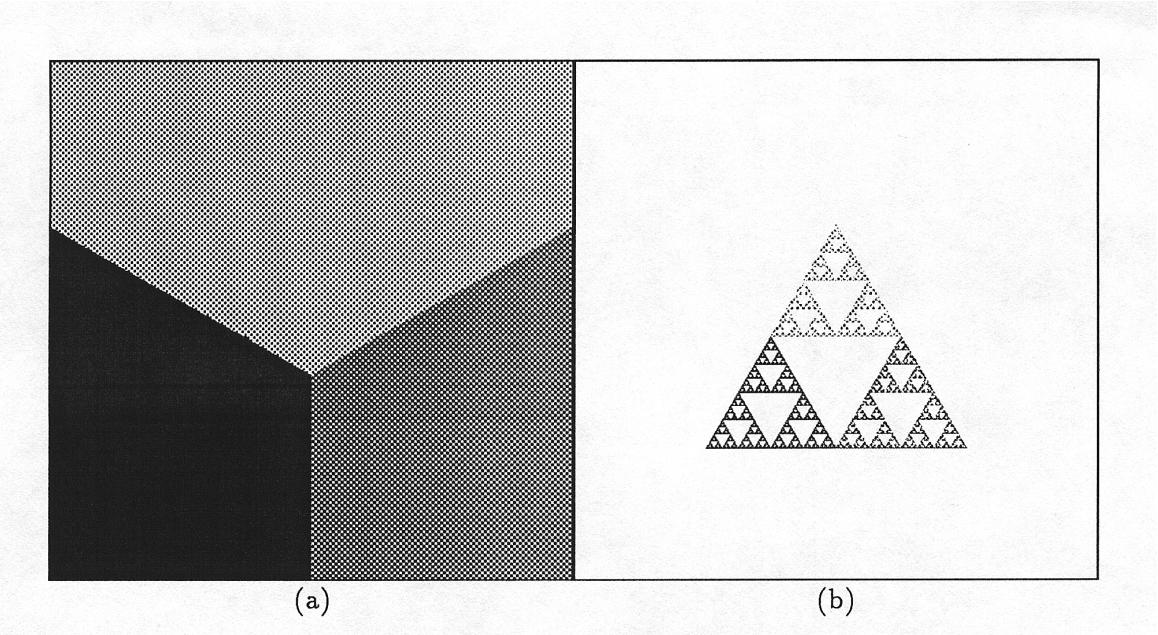



Figure 4.5: The domain divisions for the Sierpiński gasket are shown in (a). The attractor in (b) is rendered using the last transformation index of the points to show the correspondence with the domain divisions.

### 4.3.3 Grid approximation method

All points not in $\mathcal{A}$ will be repelled along different paths towards the boundary of the infinity circle. If there are two points, $x, y$, inside the window of interest such that $y = T^{-1}(x)$, then that path will have an accumulated escape-time equal to $E_c(y) + 1$, due to the fact that the image of a transformation is unique. The rest of the branches may be calculated in similar fashion. The strategy of this computation is illustrated in Figure 4.8.

Since it is not practical to store values for every position inside of the infinity circle, only points inside of the window are used for interpolation. Image points which fall outside of the window while remaining inside of the infinity circle must be computed directly. The escape-time function is evaluated for grid points starting on the outside of the window and moving towards the centre in a spiral fashion. The potential for time savings is significant since the evaluation of entire, large subtrees may be avoided. It is unlikely that the situation will arise that $y = T^{-1}(x)$ will be a grid

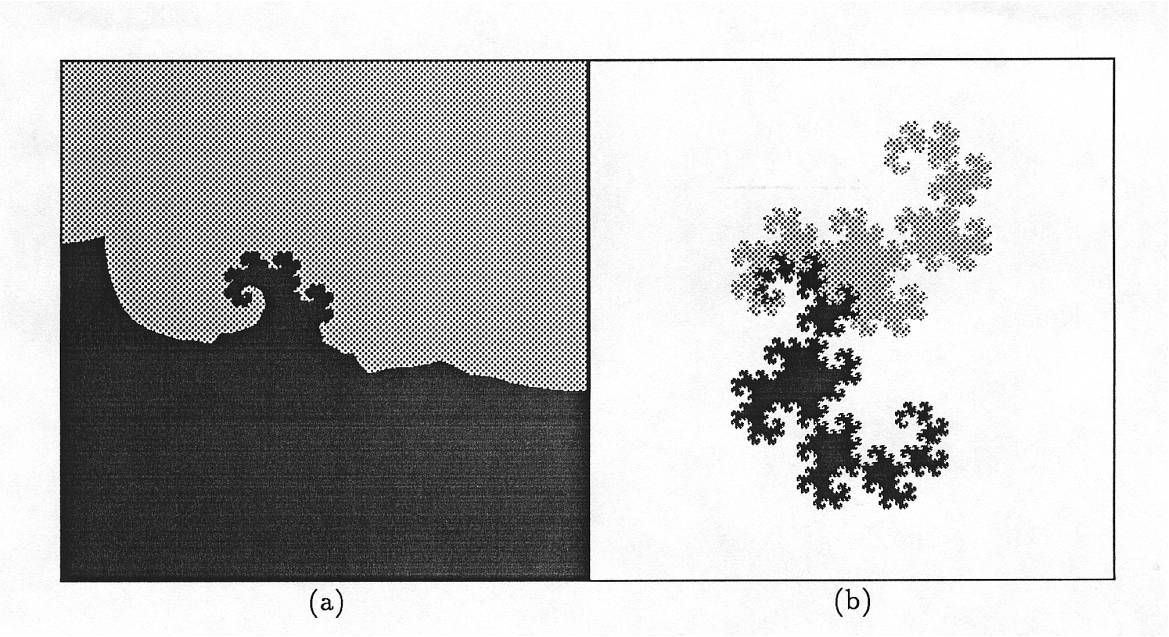



Figure 4.6: The domain divisions for the Dragon curve are shown in (a). The attractor in (b) is rendered using the last transformation index of the points to show the correspondence with the domain divisions.

---

location. The interpolation scheme subdivides each grid cell into two "interpolating" triangles. If the point $y$ lies inside the grid, it will intersect one of triangles. The idea is illustrated in Figure 4.9.

An interpolation is made only if all three vertices have been computed. Interpolation is carried out according to the procedure outlined by Snyder and Barr [42] who use the barrycentric coordinates [10] of the intersection point as weights for interpolating the values at the vertices of the triangle. The computation of the barrycentric coordinates is performed according to Equation 4.5 where $B_i$ is the coordinate for vertex $i$ (assuming modulo 3 arithmetic), $V_i$ is the vertex $i$ and P is the point of intersection.

$$B_i = (V_{i+2} - V_{i+1}) \times (P - V_{i+1}) \tag{4.5}$$

If some or all of the vertices are unresolved, then these may be computed recursively until a value is obtained. However, if any of the vertices are in the process of

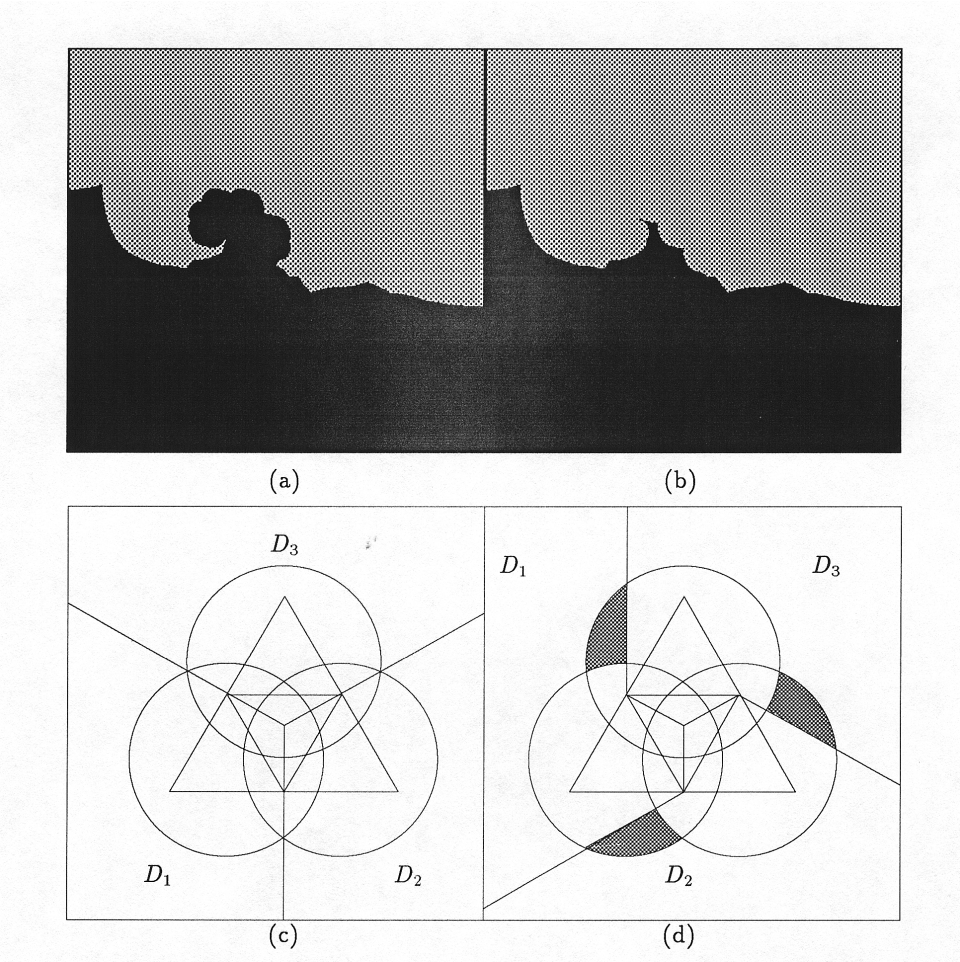



Figure 4.7: The domain divisions in the case of the dragon curve may vary at low resolutions depending on the order in which the transformations are applied, as indicated in (a) and (b). The divisions are not arbitrary, as indicated in the case of the gasket shown in (c) and (d). The divisions in (d) will cause incorrect transformation application in the shaded areas.

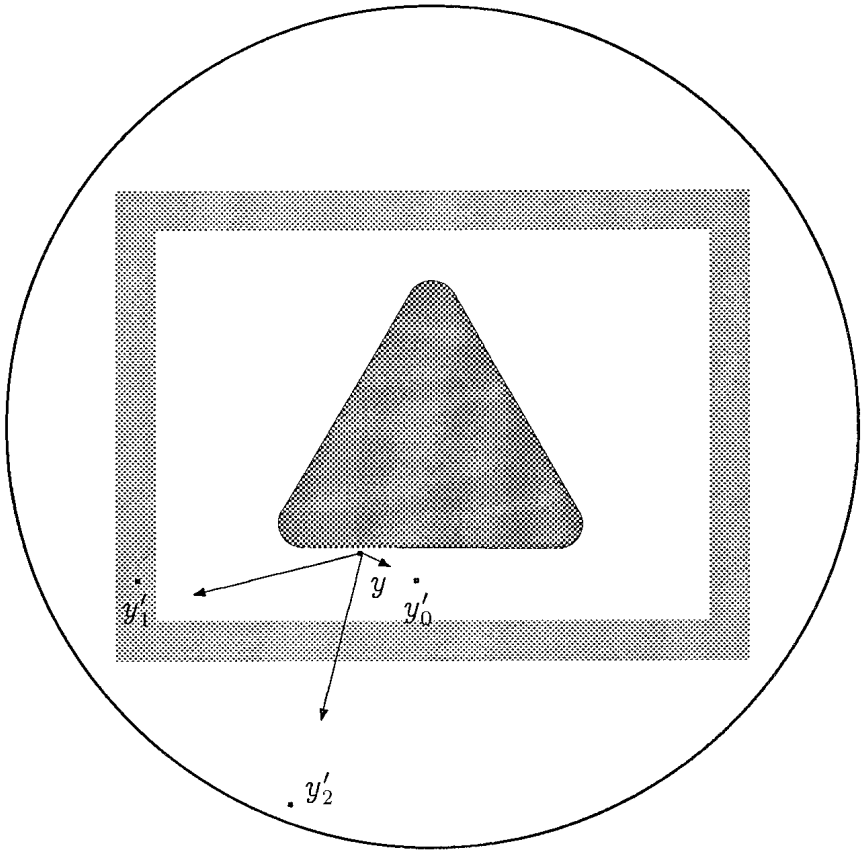



Figure 4.8: The strategy for approximating escape-time values for the point $y$. The image point $y'_0$ falls at a grid position which has not yet been evaluated. Computation of images of this point must continue until an image point reaches location at which direct computation is required or else a grid position at which a value can be approximated. The image point $y'_1$ falls at a grid position which has already been computed, so a value at this point may be approximated directly, without further iteration. The image point $y'_2$ falls completely outside of the grid, so its value must be computed directly.

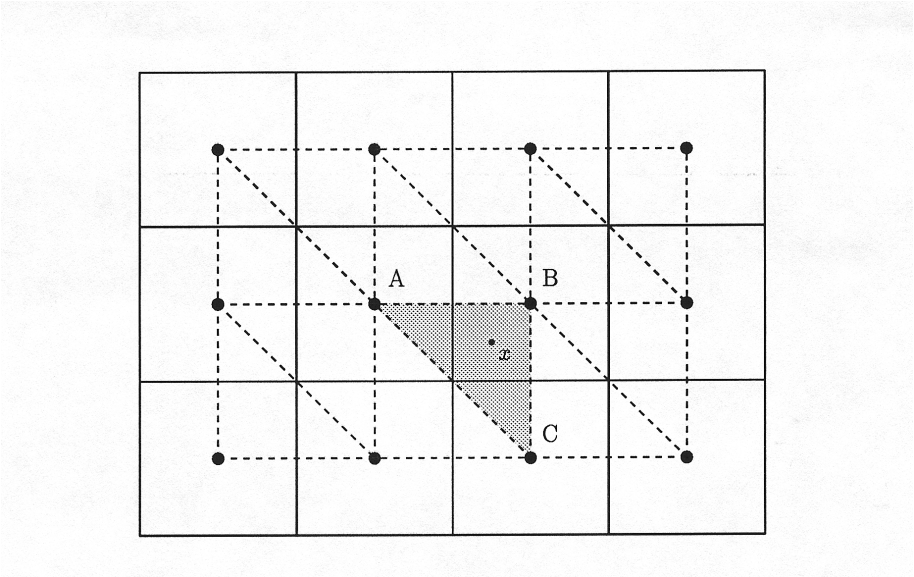



Figure 4.9: This figure illustrates a 4x3 region of the grid used in the approximations. The solid lines represent pixel boundaries and computations are made at the centers of these. The interpolation grid is built on these pixel centers, as indicated by the dashed lines. In order to approximate a value for the point $x$, the values stored at vertices A, B, and C are used.

being computed, a value cannot be interpolated and the function must be evaluated directly. This method is described in Algorithm 4.2.

These steps have allowed this technique to provide quite accurate approximations to the direct computation, and provide a sizeable improvement in computation speed, as indicated in Tables 4.2, 4.3 and 4.4. A comparison of the test images produced using both small and large radius infinity circle are shown in Figures 4.10, 4.11 and 4.3.3. In determining the error estimates for the approximation method, the values computed by the **Approximate-escape** method were compared on a pixel basis with those computed by the **Basic-escape** method. The absolute values of the differences were used to compute both the average error and standard deviation for the approximated computations.

**ALGORITHM 4.2** **Approximate-escape**$(x, n, M, \mathcal{T}^{-1}, N)$
**PURPOSE** Approximate the escape-time of a point $x$ from $\mathcal{A}$

| | | |
|---|---|---|
| Arguments | $x$ | point |
| | $n$ | levels remaining to compute |
| | $M$ | maximum number of levels |
| | $\mathcal{T}^{-1}$ | transformations |
| | $N$ | number of transformations |
| Output | $E$ | escape time for $x$ |
| Local variables | $i$ | loop index |
| Functions | inside-infinity() | determine if a point is within the infinity circle |
| | escape-fraction() | compute the fractional part of the according to Definition 4.2 |

```
BEGIN
    IF (n == 0) THEN
        E = 0
    ELSE
        E = 0
        tmp = 0
        FOR i = 1, ..., N DO
            x' = T_i^{-1}(x)
            IF (inside-infinity(x')) THEN
                tmp = Next-escape-level(x', n - 1, M, T^{-1}, N)
            ELSE
                tmp = escape-fraction(x, x')
            ENDIF
            IF (tmp > E) THEN
                E = tmp
            ENDIF
            IF (E >= (M - n)) THEN
                BREAK
            ENDIF
        ENDFOR
    ENDIF
    RETURN(E)
END
```

| **Procedure** | **Next-escape-level**$(x, n, M, \mathcal{T}^{-1}, N)$ | |
|---|---|---|
| **Purpose** | Compute the next escape level | |
| Arguments | $x$ | point |
| | $n$ | levels remaining to compute |
| | $M$ | maximum number of levels |
| | $\mathcal{T}^{-1}$ | transformations |
| | $N$ | number of transformations |
| Output | $E$ | escape time for $x$ |
| Local variables | $i$ | loop index |
| Functions | inside-grid() | test if a point is inside the grid |
| | grid-pos() | convert a point into a grid cell position |
| | triangle-unref() | determine if a triangle is referenced |
| | grid-approx() | approximate a value from surrounding grid points |

```
BEGIN
        IF (inside-grid(x')) THEN
                IF (triangle-unref(grid-pos(x'))) THEN
                        tmp = grid-approx(grid-pos(x')) +1
                ELSE
                        tmp = Basic-escape(x', n - 1, M, T^-1, N)
                ENDIF
        ELSE
                tmp = Basic-escape(x', n - 1, M, T^-1, N)
        ENDIF
        RETURN(E)
END
```

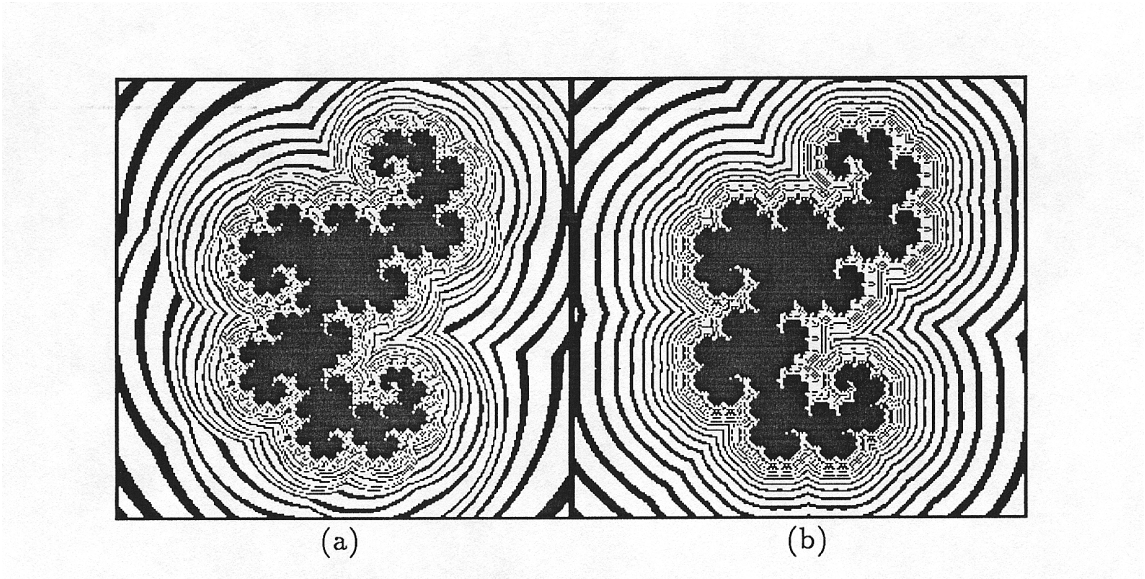



Figure 4.10: Two different escape-time renderings of the Dragon curve with different sizes for the infinity circle. In (a), the infinity circle has a radius of 1.52 which is just large enough to circumscribe the viewing area and in (b), a radius of 8 times that is used.

| Infinity circle radius | Number of levels | Total time (minutes) | | Average error | Standard deviation |
|---|---|---|---|---|---|
| | | Basic method | Approximate method | | |
| 1.52 | 15 | 2.04 | 0.86 | 0.004774 | 0.045037 |
| 3.04 | 17 | 8.16 | 2.17 | 0.003038 | 0.028273 |
| 6.08 | 19 | 29.75 | 7.22 | 0.002784 | 0.026316 |
| 12.16 | 21 | 114.43 | 25.84 | 0.002732 | 0.025799 |

Table 4.2: Experimental results for escape-time calculations on the Dragon curve which compare the basic and approximate evaluation techniques.

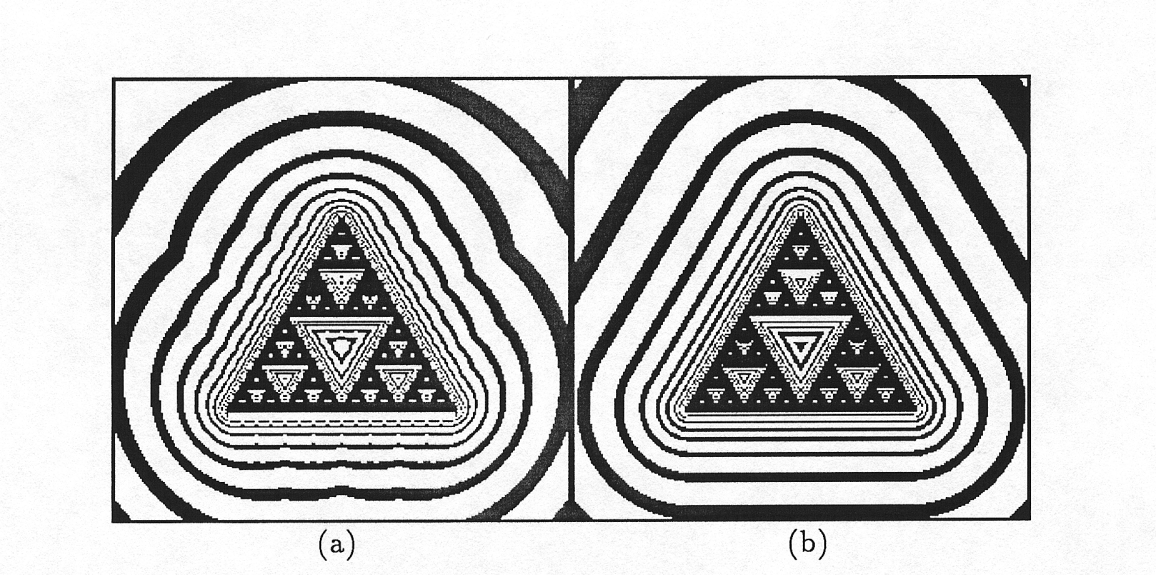



Figure 4.11: Two different escape-time renderings of the Sierpiński gasket with different sizes for the infinity circle. In (a), the infinity circle has a radius of 1.57 which is just large enough to circumscribe the viewing area and in (b), a radius of 8 times that is used.

| Infinity circle radius | Number of levels | Total time (minutes) | | Average error | Standard deviation |
|---|---|---|---|---|---|
| | | Basic method | Approximate method | | |
| 1.57 | 7 | 1.33 | 0.76 | 0.000157 | 0.004794 |
| 3.14 | 8 | 4.91 | 1.80 | 0.000183 | 0.004083 |
| 6.28 | 9 | 12.57 | 5.97 | 0.000191 | 0.003828 |
| 12.56 | 10 | 37.76 | 18.31 | 0.000174 | 0.003677 |

Table 4.3: Experimental results for escape-time calculations on the Sierpiński gasket which compare the basic and approximate evaluation techniques.

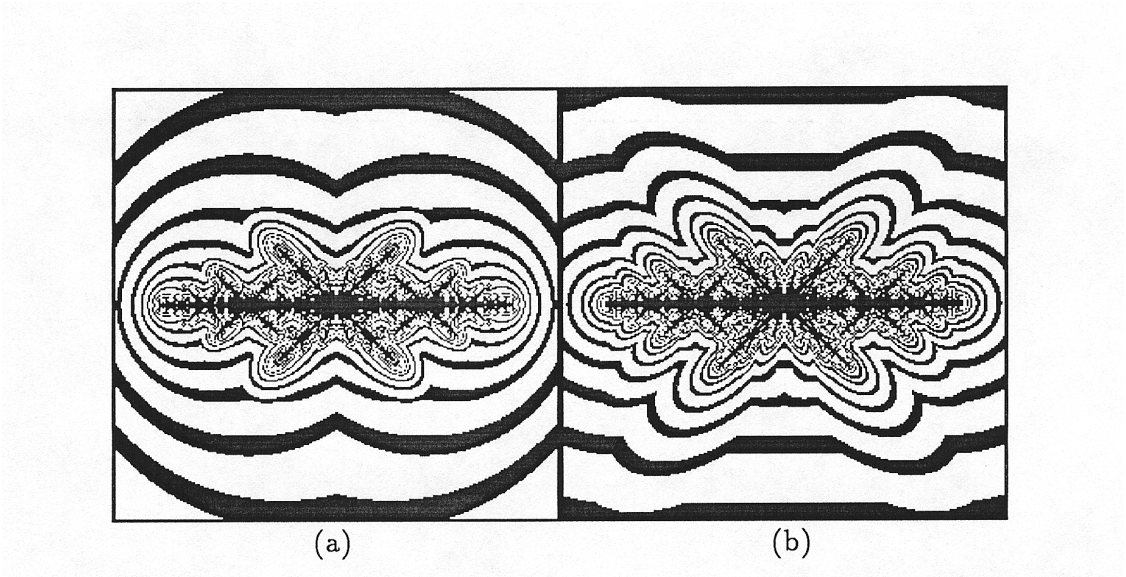



Figure 4.12: Two different escape-time renderings of the Star with different sizes for the infinity circle. In (a), the infinity circle has a radius of 1.77 which is just large enough to circumscribe the viewing area and in (b), a radius of 8 times that is used.

| Infinity circle radius | Number of levels | Total time (minutes) | | Average error | Standard deviation |
|---|---|---|---|---|---|
| | | Basic method | Approximate method | | |
| 1.77 | 7 | 2.30 | 1.38 | 0.000956 | 0.015160 |
| 3.54 | 8 | 7.53 | 4.24 | 0.000708 | 0.013155 |
| 7.08 | 9 | 31.45 | 15.94 | 0.000734 | 0.012774 |
| 14.16 | 10 | 96.35 | 52.79 | 0.000692 | 0.012562 |

Table 4.4: Experimental results for escape-time calculations on the Star which compare the basic and approximate evaluation techniques.

## 4.4 Selection of the infinity circle

The experimental results given in Tables 4.2, 4.3 and 4.4 indicate the importance of the infinity circle in any computation. The infinity circle represents a boundary outside of which all points tend to infinity. In practice, it defines the maximum area of interest for a particular attractor and its complement. Since the escape-time function is always zero outside of the infinity circle, the circle should at least enclose the attractor and the viewing window.

Figure 4.1 illustrates the results of an escape-time calculation for a simple example given by the IFS in Table 2.3. The integer values of the escape-time function are indicated by points on the boundaries of the black and white bands. Each boundary in Figure 4.1 is an image of the original infinity circle, under the contractive inverse of the current composite transformation. Points which remain inside the boundary have not been disproved of membership in the attractor. In the case of IFS's which contain more than 1 transformation, the boundary is actually the union of the copies of the infinity circle under the individual transformations. Those points which remain inside the boundary at some level require further computation to determine their membership in the attractor. The size of the infinity circle determines the degree of computation required, since any overlap between boundary circles means that the points contained within those regions are not be repelled by more than 1 transformation. Figure 4.13 illustrates the increased computational overhead when the radius of the infinity circle is increased.

The size of the infinity circle is important for its effect on the appearance of the final image. Notice the differences which occur in the images of Figures 4.10, 4.11 and 4.3.3 as the radius is increased. The effect is particularly important in the case when the transformations do not all scale the same. In the case of the star attractor, very different features of the image are highlighted as the radius increases. As the size of the infinity circle increases, the more expansive inverse transformations are selected with decreasing frequency which has the effect of reducing their contribution to the final image.

The center of the infinity circle is also important, especially when the radius of the

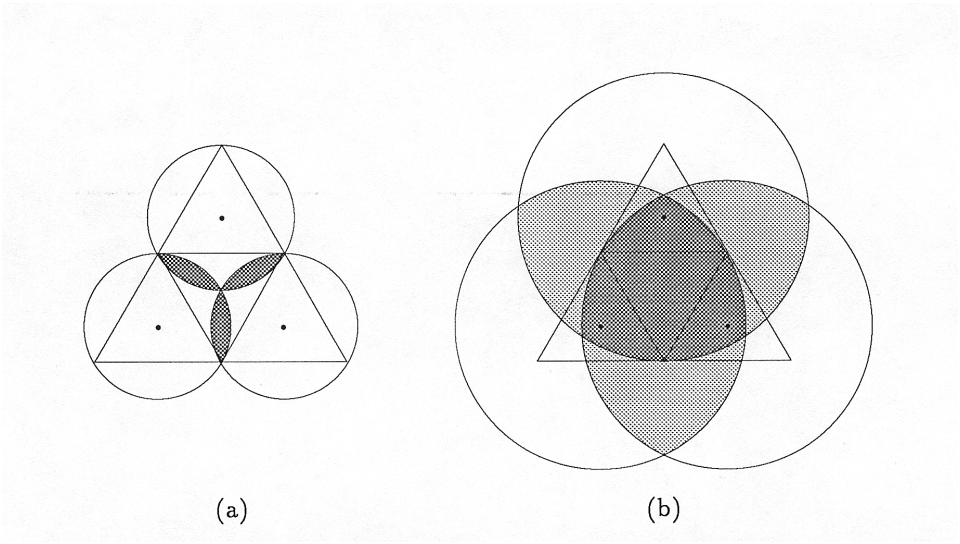



Figure 4.13: An illustration of how the size of infinity circle affects the amount of computations which must be performed. In (a), with a small infinity circle, there is little duplicated effort since the overlap is small. However, in (b), there is a great deal of overlap, which indicates that points in the central portion will not be repelled by any of the three transformations.

infinity circle is small. Since intermediate resolutions of the attractor approximation are used, an asymmetrical image may be produced if the infinity circle is not centered on the attractor. As an approximation to the center of the attractor, one may select the center of the infinity circle to be the center of the minimum disk enclosing the attractor. Figure 4.14 demonstrates the variations which are possible depending on the selection for the center of the infinity circle.

## 4.5 Visualization techniques

The simplest method for visualizing the escape-time function computed for an array of points associates a different grey level with each interval of escape-time values, as shown in Figure 4.15). If the bands indicate integer value boundaries then this corresponds to the discrete escape-time function. A second possibility is to highlight contours in the image as is shown in Figure 4.16.

Instead of varying colors, one can vary the height of each point according to its value of the escape-time function, as shown in Figure 4.17.

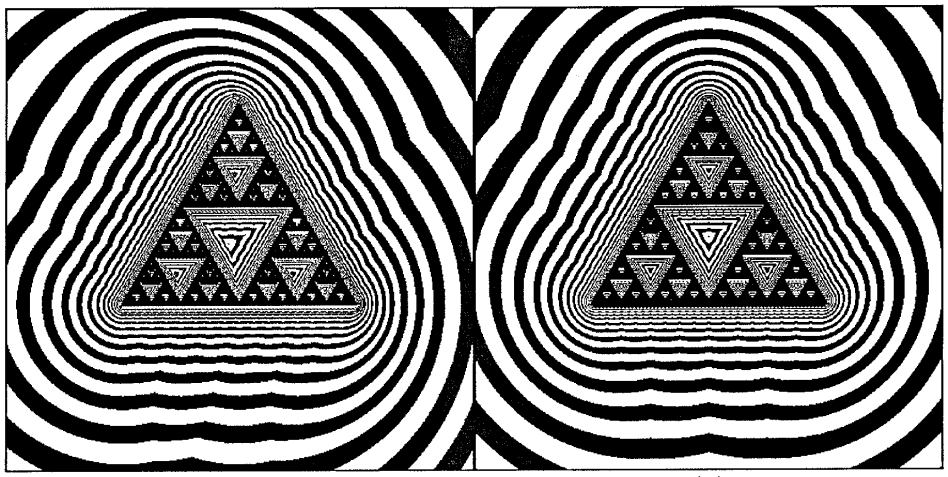



Figure 4.14: The final image may vary depending on the selection of the center for the infinity circle. In (a), it is centered at the origin but in (b) it is centered at the center of the minimum enclosing disk for the attractor.

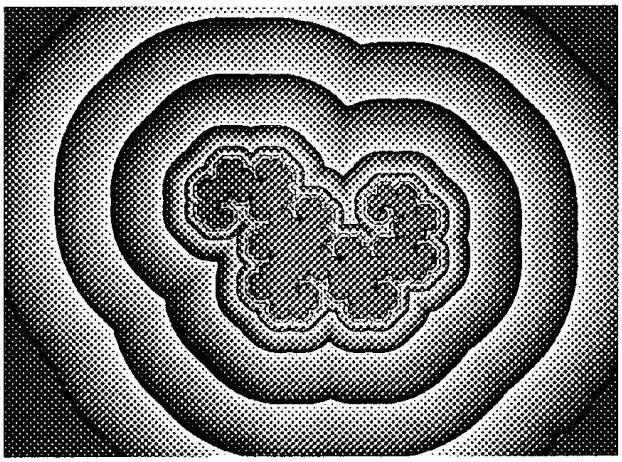


Figure 4.15: The Dragon curve and its complement visualized by applying a ramp of grey to the escape-time function values.

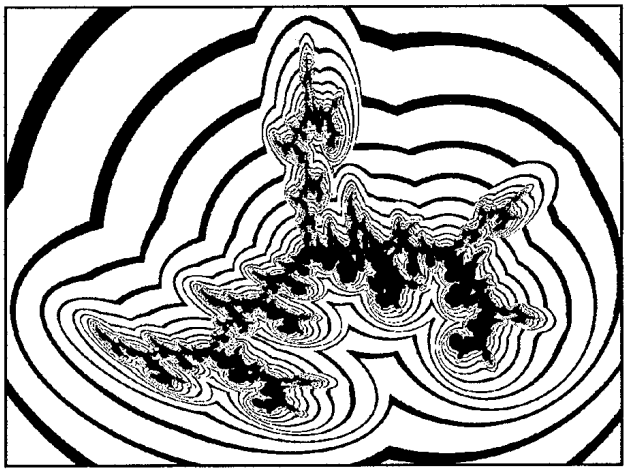


Figure 4.16: The Flame attractor and its complement visualized by higlighting contour lines in the escape-time function values.

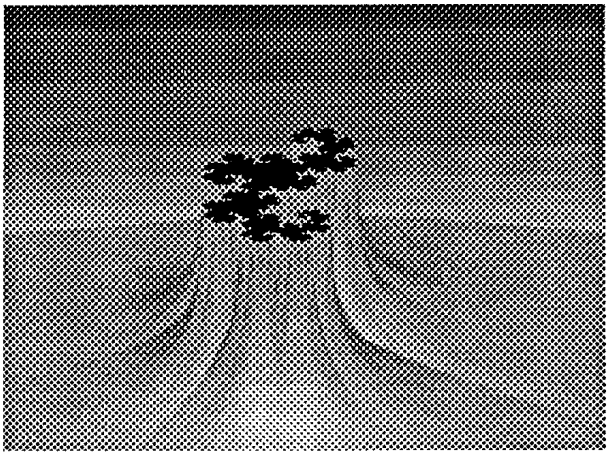


Figure 4.17: The Dragon curve and its complement visualized by interpreting escape-time function values as heights. A colour version of this image appears in the original of this thesis as Figure **??**.

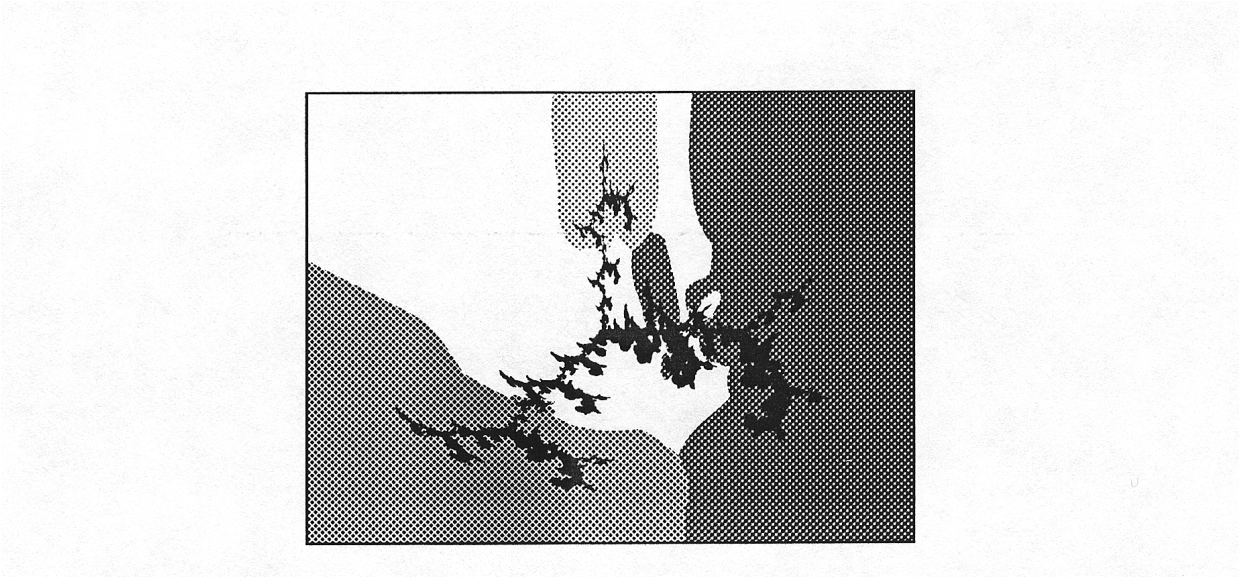


Figure 4.18: The Flame attractor overlaid on a background determined by the first transformation index. The different grey levels indicate different initial transformations.

### 4.5.1 Transformation indices

Each value in the escape time function has associated with it a string of transformations. An interesting variation on the theme of the escape time is the one where one examines what type of divisions of the plane occur using this various components of the history. Note that by considering only the first index, one can see the domain divisions for the attractor. A sample of this type of image is shown in Figure 4.18.

### 4.5.2 Metrics

As with the distance calculations of Chapter 3, it is possible to use different metrics to measure the relative distances of points from the attractor. In the case of the escape-time function, the shape of the infinity circle changes to a diamond, as illustrated in Figure 2.21. Figure 4.19 shows the escape-time function computed for a dragon curve using the Manhattan metric. Figure 4.20 shows a heightfield interpretation of the escape-time function for the Sierpiński gasket calculated using the Manhattan metric.

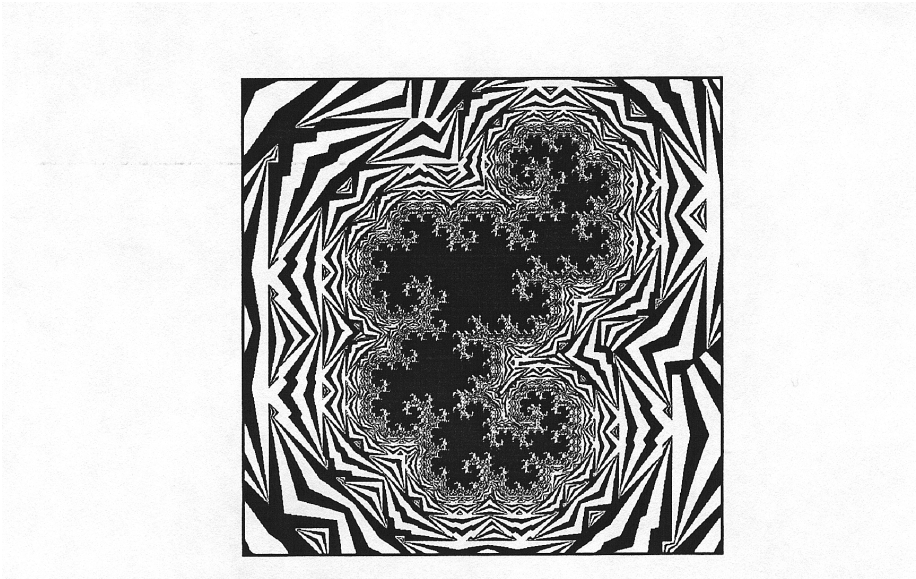


Figure 4.19: Escape-time for the Dragon curve computed with the Manhattan metric.

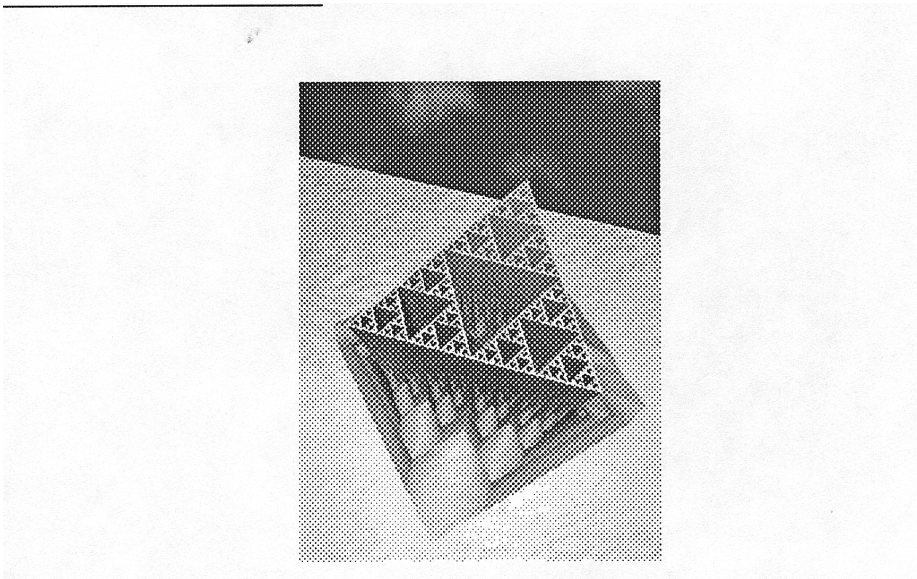


Figure 4.20: The escape-time function for the Sierpiński gasket computed using the Manhattan metric where each value is interpreted as a height. A colour version of this image appears in the original of this thesis as Figure **??**.

# Chapter 5

# Controlled Iterated Function Systems

The preceding chapters have discussed methods by which the attractor of an IFS can be accurately approximated and visualized. As demonstrated by the examples throughout this thesis, standard iterated function systems provide a concise description of fractal objects which have the property of self-affinity. However, there exist many objects which can be considered as fractals yet lack this property. Mandelbrot [29] used the tree shown in Figure 5.1 to illustrate the class of "non-uniform" fractals. This object cannot be represented with a standard IFS with invertible transformations, since the thin straight stem could not be produced in this way. Barnsley has shown that if singular transformations, called *condensation maps*, are permitted the attractor can be represented by the IFS given in Table 5.1. The effect of these maps is to shrink distances to zero is some particular direction.

$$T_1 = \begin{bmatrix} 0.000 & 0.000 & 0.000 \\ 0.000 & 0.277 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.000 & 0.000 & 0.000 \\ 0.000 & 0.277 & 0.000 \\ 0.000 & 1.000 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.354 & -0.354 & 0.000 \\ 0.354 & 0.354 & 0.000 \\ 0.000 & 2.000 & 1.000 \end{bmatrix} \quad T_4 = \begin{bmatrix} 0.354 & 0.354 & 0.000 \\ -0.354 & 0.354 & 0.000 \\ 0.000 & 2.000 & 1.000 \end{bmatrix}$$

Table 5.1: IFS with condensation for the tree attractor.

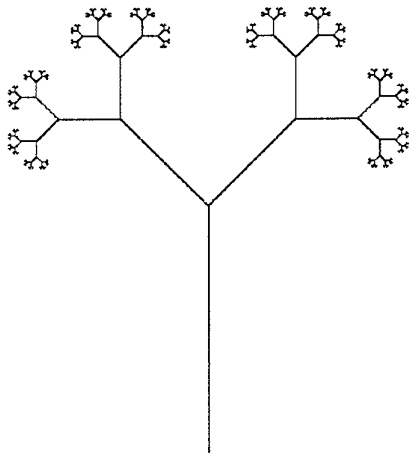


Figure 5.1: The attractor of the tree IFS with condensation maps.

The ability to represent complex attractors is important for both theoretical and practical reasons. Apart from the mathematical curiosity in expanding the class of fractals which can be generated using IFS techniques, the subsequent theory has many applications for modelling of natural structures. There is evidence to indicate that plants have fractal characteristics, not the least of which is the abundance of fractals which *look* like plants. As Prusinkiewicz [36] notes, the use of fractals as models for plants can be an effective tool for revealing important information about the self-similarity of their structure.

The approach of condensation maps alone is not adequate to represent attractors with more complex features. In order to increase the ability of standard IFS techniques to represent objects which are not strictly self-affine, methods for expanding the class of fractals definable by IFS techniques must be examined. The approach examined here is to control transformation application.

The notion of control yields restriction in transformation application. States are introduced in order to add memory to the system. Application of a transformation implies a transition between states. A directed graph can be used to specify which transformations may be applied in any given state. States are represented by nodes

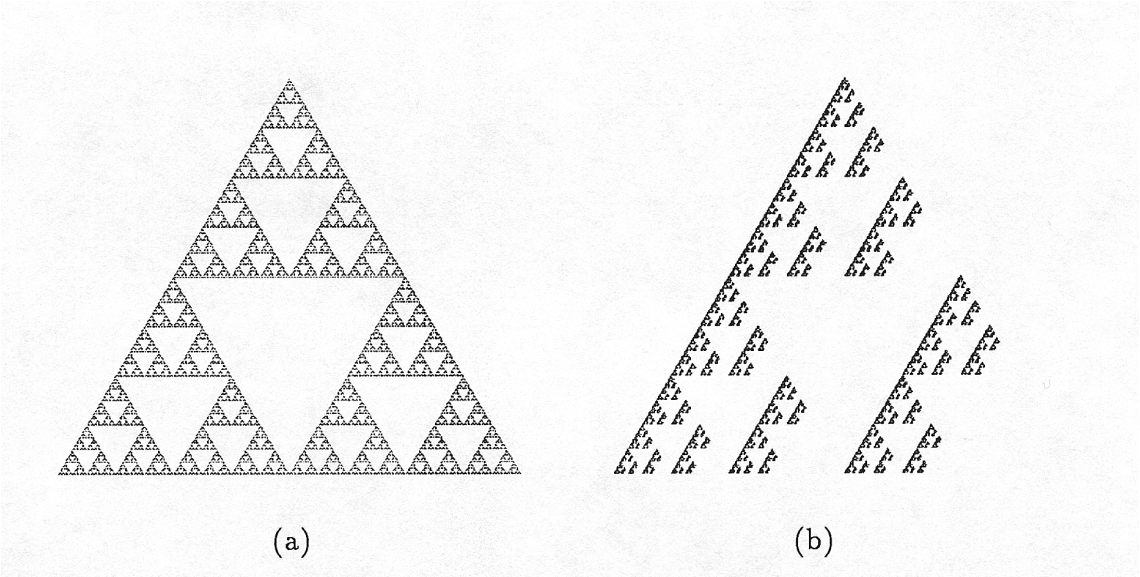



Figure 5.2: A standard gasket shown in (a) contains a version of the attractor (b) produced by applying a control mechanism to the transformations.

---

in the graph, and the edges are labelled with the allowable transformations. A short arrow is used to point to the initial state for transformation application. For any ordinary IFS, the system is always in the state when all transformations are applicable. Conversely, if conditions were added to the generation of the Sierpiński gasket which stated that that $T_2$ could not follow $T_2$, separate states would be required. This condition is expressed in the transition graph shown in Figure 5.3. Figure 5.2 shows the standard gasket and the one that is generated using the control mechanism. Since only a subset of the possible transformation sequences are applied to generate the attractor of controlled IFS, each controlled IFS attractor is a subset of the corresponding uncontrolled IFS attractor.

The transition information can form the basis of either a stochastic or deterministic mechanism to control the application of the transformations in $\mathcal{T}$.

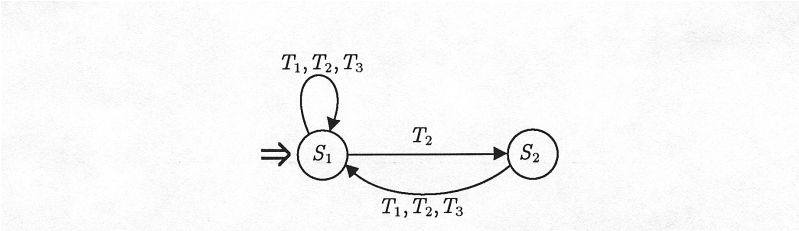



Figure 5.3: Transition graph for the restricted gasket of Figure 5.2(b).

$$\begin{bmatrix} 0.33 & 0.34 & 0.33 \\ 0.50 & 0 & 0.50 \\ 0.33 & 0.34 & 0.33 \end{bmatrix}$$

Table 5.2: Stochastic matrix for the restricted gasket.

## 5.1 Stochastic control

There are several methods by which memory may be added to a stochastic process. The reader is referred to Molloy [31] for further detail on this subject.

The simplest form which remembers only the last state is called a Markov chain.

**Definition 5.1 (Markov chain)** *A stochastic process which restricts the time between state changes to be memoryless, but allows any new state given the current state, is called a Markov chain.*

A Markov chain is *irreducible* if all states are reachable from all other states. Barnsley applied these notions to iterated function systems and developed the concept of Recurrent IFS (RIFS) [3]. The rules for transformation application are expressed in a *stochastic matrix*, which can be constructed from the information in the transition graph. Each transition in the graph is assigned a probability. Each row of the matrix corresponds to the total probability of some transition occuring given the current state. If there is no transition defined between a pair of states, then a probability of zero is assigned. Table 5.2 gives the stochastic matrix associated with the attractor shown in Figure 5.2(b).

$$T_1 = \begin{bmatrix} 0.000 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.000 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.130 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.297 & 0.297 & 0.000 \\ -0.297 & 0.297 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_4 = \begin{bmatrix} 0.262 & -0.262 & 0.000 \\ 0.262 & 0.262 & 0.000 \\ 0.000 & 0.140 & 1.000 \end{bmatrix}$$

$$T_5 = \begin{bmatrix} 0.000 & 0.200 & 0.000 \\ -0.200 & 0.000 & 0.000 \\ -0.050 & 0.050 & 1.000 \end{bmatrix} \quad T_6 = \begin{bmatrix} 0.200 & 0.000 & 0.000 \\ 0.000 & 0.200 & 0.000 \\ -0.050 & 0.050 & 1.000 \end{bmatrix}$$

$$T_7 = \begin{bmatrix} 0.740 & 0.000 & 0.000 \\ 0.000 & 0.740 & 0.000 \\ -0.078 & 0.078 & 1.000 \end{bmatrix} \quad T_8 = \begin{bmatrix} 0.000 & -0.172 & 0.000 \\ 0.172 & 0.000 & 0.000 \\ 0.050 & 0.190 & 1.000 \end{bmatrix}$$

$$T_9 = \begin{bmatrix} 0.172 & 0.000 & 0.000 \\ 0.000 & 0.172 & 0.000 \\ 0.050 & 0.190 & 1.000 \end{bmatrix} \quad T_{10} = \begin{bmatrix} 0.740 & 0.000 & 0.000 \\ 0.000 & 0.740 & 0.000 \\ 0.069 & 0.106 & 1.000 \end{bmatrix}$$

$$T_{11} = \begin{bmatrix} 0.740 & 0.000 & 0.000 \\ 0.000 & 0.740 & 0.000 \\ 0.000 & 0.260 & 1.000 \end{bmatrix}$$

Table 5.3: IFS for the fern leaf shown in Figure 5.4.

With this degree of control, it is possible to construct an IFS generating an attractor which is not strictly self-similar. For example, the attractor shown in Figure 5.4 was produced with the transformations in Table 5.3 using the transition information given by the stochastic matrix in Table 5.4 and illustrated in Figure 5.5. The use of condensation maps in this IFS is required in order to preserve the correct shape while maintaining the criterion that the Markov chain be irreducible. The procedure which makes use of the Markov chain to control the transformation application is given in Algorithm 5.1.

Returning to the example of the tree, Figure 5.6 illustrates the effect of allowing

$$\begin{bmatrix}
0.20 & 0.20 & 0.05 & 0.05 & 0.05 & 0.05 & 0 & 0.05 & 0.05 & 0 & 0.30 \\
0.20 & 0.20 & 0.05 & 0.05 & 0.05 & 0.05 & 0 & 0.05 & 0.05 & 0 & 0.30 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & 0 & 0.75 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0.75 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & 0 & 0.75 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & 0 & 0.75 \\
0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0 & 0 & 0 & 0.75 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0.75 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0.75 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.25 & 0.75 \\
0.15 & 0.15 & 0 & 0 & 0.05 & 0.05 & 0 & 0.05 & 0.05 & 0 & 0.50
\end{bmatrix}$$

Table 5.4: Stochastic matrix for the fern example.

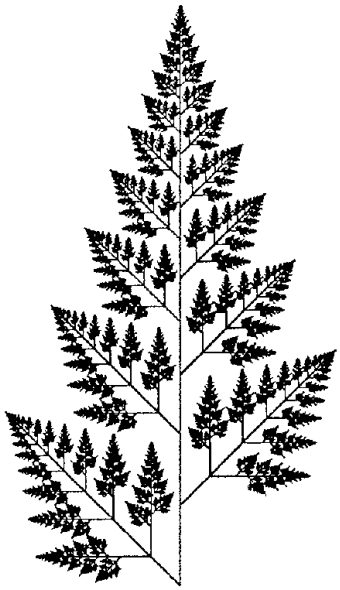


Figure 5.4: An attractor which breaks the strict self-similarity by creating an alternating branching pattern at alternate levels.

**ALGORITHM 5.1** **Markov-iteration**$(x_0, s_0, n, \mathcal{T}, \mathcal{M}, N)$
**PURPOSE** Compute an approximation to $\mathcal{A}$

| | | |
|---|---|---|
| Arguments | $x_0$ | preimage |
| | $s_0$ | initial state |
| | $n$ | number of points |
| | $\mathcal{T}$ | the finite set of affine transformations |
| | $\mathcal{M}$ | stochastic matrix associated with $\mathcal{T}$ |
| | $N$ | the number of transformations |
| Local variables | $x, y$ | points |
| | $i, j$ | loop indices |
| | *rand* | storage for random number |
| | *psum* | sum of probabilities |
| | $k$ | index of selected transformation |
| Functions | rnd() | return a pseudo-random number $\in [0, 1]$ |
| | plot-point() | plot a point on the screen |

```
BEGIN
    x = x_0
    cur = s_0
    FOR i = 1, ..., n DO
        rand = rnd()
        psum = 0
        FOR j = 1, ..., N DO
            psum = psum + M[cur][j]
            IF (rand < psum) THEN
                k = j
                BREAK
            ENDIF
        ENDFOR
        y = T_k(x)
        Plot-point(y)
        x = y
        cur = k
    ENDFOR
END
```

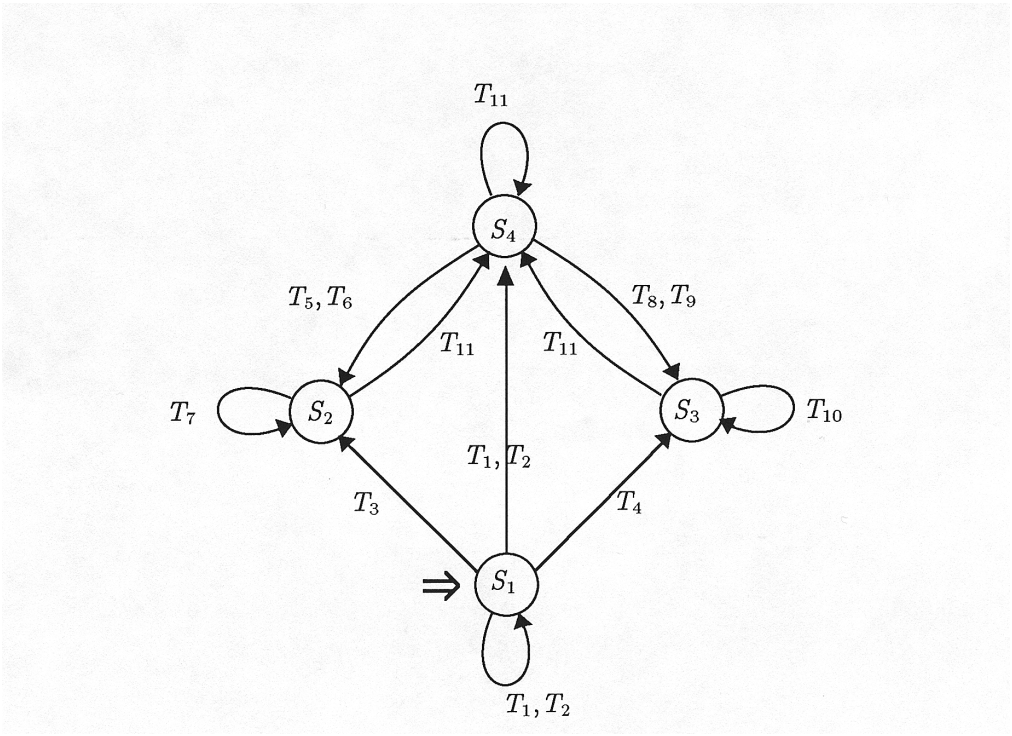



Figure 5.5: Control graph for the fern example.

the replacment of the condensation maps by ones which scale uniformly. The modified IFS is given in Table 5.5. The extraneous branching on the stem of the tree is caused by the application of the branching transformations $\{T_3, T_4\}$ to the stem area $\{T_1, T_2\}$. Clearly, if $\{T_1, T_2\}$ need to follow $\{T_3, T_4\}$, they must use condensation (see page 100). However, if the transformations are restricted so that $\{T_1, T_2\}$ cannot follow $\{T_3, T_4\}$, as indicated by the graph in Figure 5.7, the Markov chain becomes reducible and the attractor in Figure 5.8 is produced.

## 5.2 Deterministic control

The stochastic method of control with the idea of Markov chains is widely used in simulation and modelling as those disciplines are concerned with the limiting behaviour of a system. The deterministic method of control has its origin in the theory of machines and languages [25].

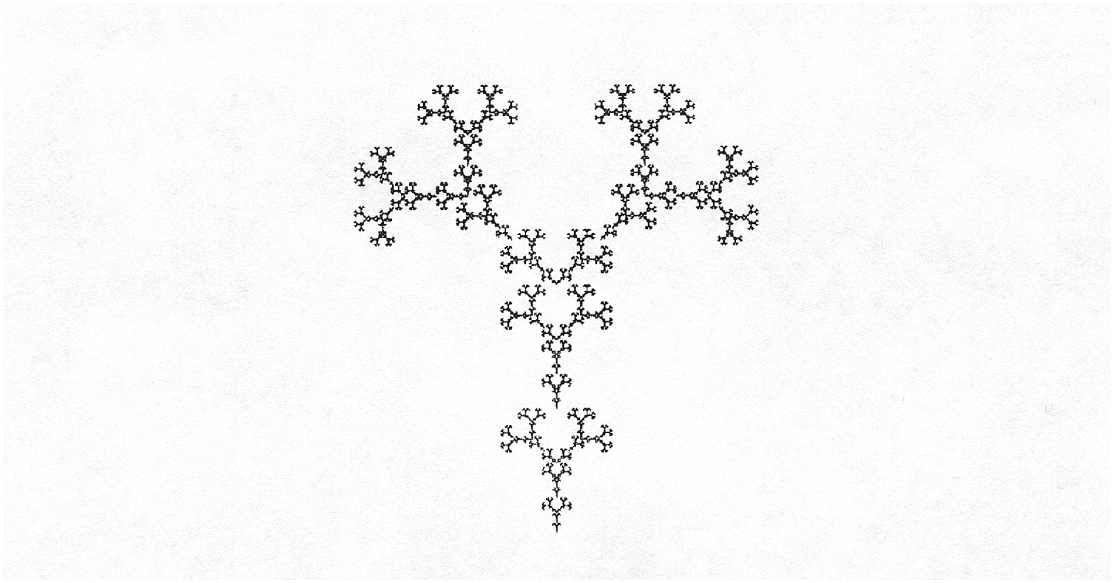


Figure 5.6: The attractor of the dichot produced without condensation maps.

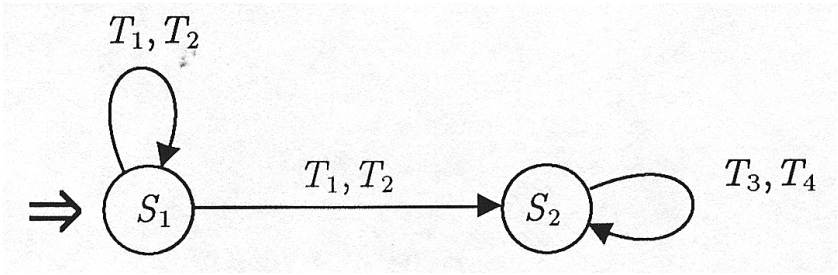



Figure 5.7: Possible transition graph to generate the tree attractor using the Markov chain approach.

$$T_1 = \begin{bmatrix} 0.277 & 0.000 & 0.000 \\ 0.000 & 0.277 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.277 & 0.000 & 0.000 \\ 0.000 & 0.277 & 0.000 \\ 0.000 & 1.000 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.354 & -0.354 & 0.000 \\ 0.354 & 0.354 & 0.000 \\ 0.000 & 2.000 & 1.000 \end{bmatrix} \quad T_4 = \begin{bmatrix} 0.354 & 0.354 & 0.000 \\ -0.354 & 0.354 & 0.000 \\ 0.000 & 2.000 & 1.000 \end{bmatrix}$$

Table 5.5: IFS without condensation for the tree attractor, which produces the attractor illustrated in Figure 5.6.

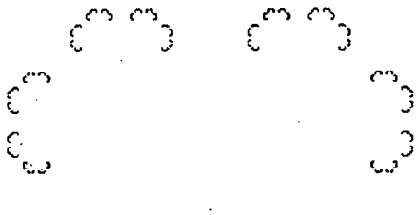


Figure 5.8: The attractor produced from the reducible Markov chain, described by the graph in Figure 5.7.

---

**Definition 5.2 (Language)** *A language is a set of strings of symbols, where:*

- *a string is a finite ordered sequence of symbols.*
- *the length of a string is the number of symbols in a string.*
- *a symbol is a distinguishable object used in constructing the strings of a language.*
- *the alphabet of a language is the set of all symbols which can appear in the strings of the language.*

In the case of an IFS, one can think of each transformation in $\mathcal{T}$ as a symbol in the language. Using the state transition information, it is possible to construct a machine called a finite state acceptor which will recognize strings in the language.

**Definition 5.3 (Finite-state acceptor)** *A finite-state acceptor is a quintuple:*

$$\mathcal{F} = \langle I, S, S_0, f, F \rangle$$

*where:*

*$I$ is a finite set of input symbols*

*$S$ is a finite set of* states*,*

*$s_0 \in S$ is a distinguished element of the set $S$, called the* initial state*,*

*$f \subset I \times S \rightarrow S$ is the next-state function,*

*$F \subset S$ is a distinguished subset of $S$, called the set of* final states*.*

Transitions are written as $(a, s_i) \rightarrow s_j$. This means that a point $a$, in state $s_i$ is taken to state $s_j$. Adding to the graphical notation introduced earlier, if $F \neq S$, the final states are distinguished by double circles. Prusinkiewicz [37] provided an example of this method in creating an IFS which describes the dichot attractor without the use of condensation maps. This example used the transformations listed in Table 5.6 controlled by the automaton $\mathcal{F}$ shown in Figure 5.9. An IFS, along with an automaton $\mathcal{F}$ is called a Controlled IFS (CIFS) [35].

With the automaton $\mathcal{F}$, it is possible to generate all strings with length less than some $n$ in the language defined by the automaton. Each of the strings accepted by the automaton $\mathcal{F}$ represents a composite transformation which is applied to some preimage $x_0 \in \mathcal{A}$ to construct an approximation to the attractor. Each point is attributed a state from the set of all states for the acceptor. To begin, the starting point is associated with the initial state for the automaton $\mathcal{F}$. Generation proceeds by applying all allowable transformations to the point in state $s_i$, which result in image points having states $s_{j_1} \ldots s_{j_n}$ where $n$ is the number of transitions allowed from state $s_i$. The details of this method are provided in Algorithm 5.2. The CIFS technique allows condensation maps to be eliminated from the examples under consideration, which allows the escape-time method to be used for rendering of the attractors.

Although it is possible to retain much from the algorithms for standard IFS's when dealing with CIFS's, non-trivial modification of the existing algorithms is required.

An illustration of distance calculation for a CIFS is shown in Figure 5.2. The transformations for this image are given in Table 5.7 and the automaton is shown in Figure 5.2.

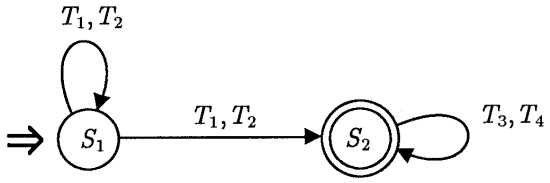



Figure 5.9: Transition graph for the tree IFS which defines the finite-state acceptor.

$$T_1 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.500 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.354 & -0.354 & 0.000 \\ 0.354 & 0.354 & 0.000 \\ 0.000 & 1.000 & 1.000 \end{bmatrix} \quad T_4 = \begin{bmatrix} 0.354 & 0.354 & 0.000 \\ -0.354 & 0.354 & 0.000 \\ 0.000 & 1.000 & 1.000 \end{bmatrix}$$

Table 5.6: IFS for the tree attractor where the transformations are to be applied according to the automaton specified in Figure 5.9.

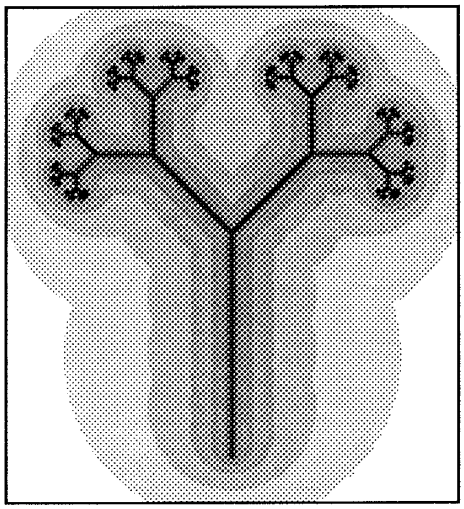


Figure 5.10: An escape-time rendering of the tree attractor.

| **ALGORITHM 5.2** | **Finite-state-attr**$(s, T, x_0, n, \mathcal{T}, N)$ | |
|---|---|---|
| **PURPOSE** | Compute an approximation to $\mathcal{A}$ | |
| Arguments | $s$ | current state |
| | $T$ | current transformation matrix |
| | $x_0$ | preimage |
| | $n$ | number of levels in tree |
| | $\mathcal{T}$ | the finite set of affine transformations |
| | $N$ | the number of transformations |
| Local variables | $i$ | loop index |
| | $t$ | new state after transition |
| | *comp* | composite transformation |
| Functions | transition() | determine if a transformation may be applied from a given state |
| | nxtstate() | return the state resulting from application of a particular transition |
| | finalstate() | determine if the state is a final state |
| | Plot-point() | plot a point on the screen |

```
BEGIN
    IF (n > 0) THEN
        FOR i = 1, ..., N DO
            IF (transition(i, s)) THEN
                comp = TT_i
                t = nxtstate(i, s)
            ENDIF
            IF (finalstate(t)) THEN
                Plot-point(comp(x_0))
            ENDIF
            Finite-state-attr(t, comp, x_0, n − 1, order, T, N)
        ENDFOR
    ENDIF
END
```

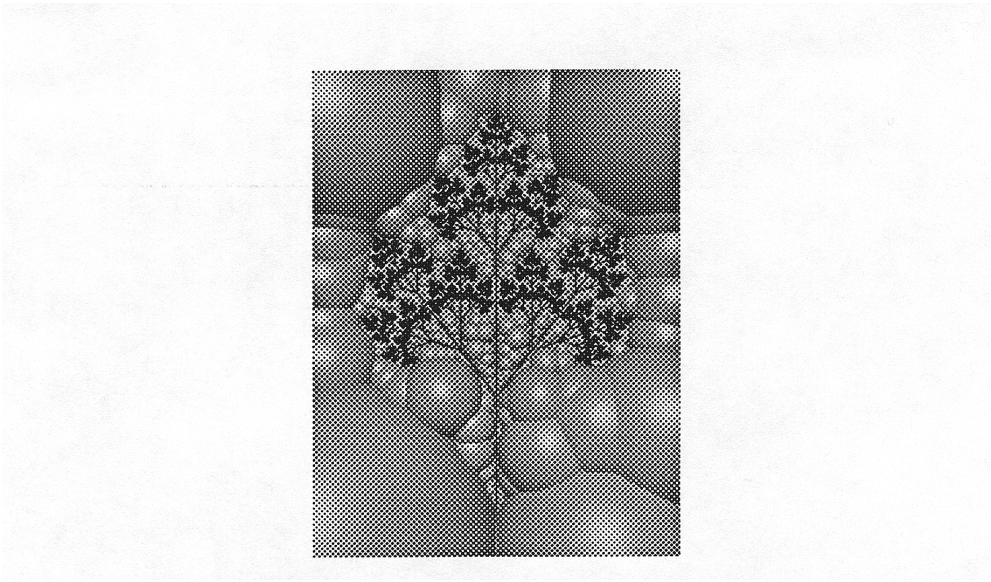


Figure 5.11: A carrot leaf surrounded by distance spheres.

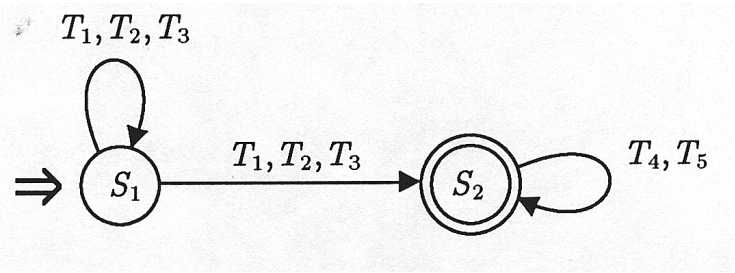



Figure 5.12: Control graph for the carrot leaf.

An illustration of escape-time calculation for a CIFS is shown in Figure 5.2. The transformations for this image are given in Table 5.8 and the automaton is shown in Figure 5.2.

$$T_1 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.250 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.000 & 0.500 & 1.000 \end{bmatrix} \quad T_4 = \begin{bmatrix} 0.405 & 0.294 & 0.000 \\ -0.294 & 0.405 & 0.000 \\ 0.000 & 0.290 & 1.000 \end{bmatrix}$$

$$T_5 = \begin{bmatrix} 0.405 & -0.294 & 0.000 \\ 0.294 & 0.405 & 0.000 \\ 0.000 & 0.290 & 1.000 \end{bmatrix}$$

Table 5.7: IFS for the carrot leaf attractor.

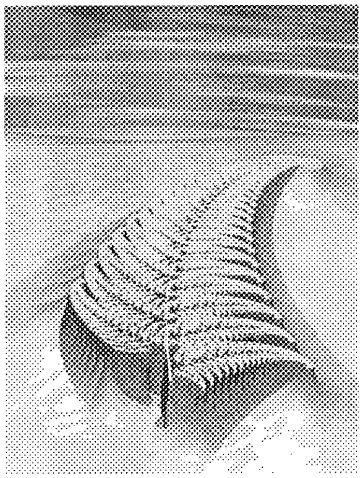


Figure 5.13: An fern-like attractor and its complement visualized using the escape-time function and interpreting the function values as heights. A colour version of this image appears in the original of this thesis as Figure **??**.

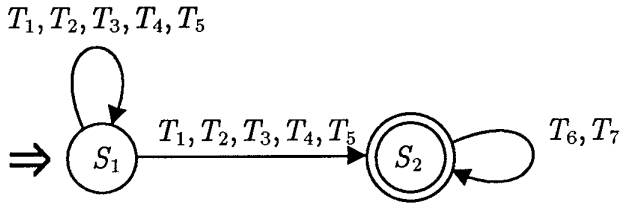



Figure 5.14: Control graph for the fern.

$$T_1 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ -0.010 & 0.000 & 1.000 \end{bmatrix} \quad T_2 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.010 & 0.000 & 1.000 \end{bmatrix}$$

$$T_3 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ -0.010 & 0.800 & 1.000 \end{bmatrix} \quad T_4 = \begin{bmatrix} 0.500 & 0.000 & 0.000 \\ 0.000 & 0.500 & 0.000 \\ 0.010 & 0.800 & 1.000 \end{bmatrix}$$

$$T_5 = \begin{bmatrix} 0.898 & -0.047 & 0.000 \\ 0.047 & 0.898 & 0.000 \\ 0.000 & 1.200 & 1.000 \end{bmatrix} \quad T_6 = \begin{bmatrix} -0.150 & 0.260 & 0.000 \\ -0.260 & 0.150 & 0.000 \\ 0.000 & 1.200 & 1.000 \end{bmatrix}$$

$$T_7 = \begin{bmatrix} -0.150 & -0.260 & 0.000 \\ -0.260 & 0.150 & 0.000 \\ 0.000 & 0.440 & 1.000 \end{bmatrix}$$

Table 5.8: IFS for controlled fern of Figure 5.2.

# Chapter 6

# Software Environment

An experimental software environment has been developed in order to test the algorithms put forth in this thesis and to provide a means for efficient specification and rendering of IFS attractors. It consists of the programs *gti* (Graphical Transformation Interpreter) and *goc* (Gti Output Converter). Information about the IFS is expressed using TSL (Transformation Specification Language), originally proposed by Sandness [40]. *Gti* interprets a TSL file to produce either an image interactively or create a binary file to be converted by *goc* into one of:

- a colour mapped image [9] which is directly viewable.
- a PostScript [44] file suitable for printing or display in a NeWS [38] environment.
- an input file for a rendering program like *rayshade* or *gout*.

## 6.1 Transformation Specification Language

Given the infinite variety of possible IFS attractors, it is important to have a convenient formalism for their specification. TSL was developed to serve the dual purpose of specifying the IFS along with its interpretation. The current version has enhanced functionality over the original version by Sandness, resulting from the addition of several commands, and reimplementation using the *lex* [28] and *yacc* [24] utilities of Unix to provide a compact, extensible specification of the language.

The attractor $\mathcal{A}$ may be visualized according to any of the methods described in the thesis. In a manner consistent with the ideas presented in the introduction, the term *model* refers to the set of points or values which is generated by an IFS in either two or three dimensions. The term *image* refers to the two dimensional representation of the model which is finally displayed.

### 6.1.1 Source file format

A complete TSL file includes both an IFS specification and a list of commands which define its interpretation. The order of commands in the file is not important. A sample is given in Table 6.1.2. All TSL commands consist of one or more keywords followed by a variable number of arguments. Arguments in a list must be separated by any of the following characters: blanks, tabs, newlines, commas or semicolons. Numerical values may be specified in arbitrary floating point format and written with or without decimal points. Any numerical value may be written as an expression, simply by enclosing it in braces. Any such expression is passed to *bc* [8] for evaluation. Functions available from *bc* include:

| Function name | Operation |
|---|---|
| sqrt(x) | square root |
| s(x) | sine |
| c(x) | cosine |
| e(x) | exponential |
| l(x) | log |
| a(x) | arctangent |

The additional variable 'p' is defined by *gti* and is set to 3.14159265358979323846 $\approx \pi$. Each source file is processed by the C language preprocessor so that support of the `#include` and `#define` directives from C is provided. Alphanumerical strings must begin with a character and may not contain any separation characters. Also as in C, comments may be included in the source file by placing them between the '/*' and '*/' delimiters but comments may not be nested. Any syntax errors encountered will cause the processing to stop.

This reference is organized into two main sections describing the specification and interpretation commands. The specification commands are applicable in all cases, but the interpretation commands are dependent upon the type of interpretation. Commands not applicable to the current method of interpretation are ignored. When describing commands, the following notation is observed:

| Notation | Meaning |
|---|---|
| **keyword** | the start of a section describing the keyword |
| `keyword` | an example using the keyword |
| *parameter* | a parameter for a keyword |
| `[]` | enclosed string is optional. If a string is not enclosed in these, then it is required. |
| ⟨⟩ | enclosed string is required |
| `\|` | a choice between items which it separates |

For example `[ a | b ]` means that either parameter *a* or keyword `b` may be used whereas ⟨ `a | b` ⟩ means that either parameter *a* or keyword `b` must be used.

### 6.1.2 IFS specification

Throughout the text, examples of IFS have been given in matrix form. Although this is effective for determining the formula of a transformation, it is sometimes more convenient to specify the rules which generate the transformation matrix rather than write down the matrix directly. The format of the corresponding commands is given below:

**xform**

specify the start of a new transformation. Individual transformations are described by scalings, rotations, and translations. Commands may be included to give control information for either deterministic or stochastic interpretation of the IFS, as described in Chapters 2 and 5. Processing begins with the transformation matrix set to the identity transformation. Each component is postmultiplied with the accumulated matrix.

`xform` [*qualifying statements*]

where *qualifying statements* can be any number of transformation or control qualifiers, described below.

```
/**
 *        TSL file for 2D Sierpinski Gasket - attractor
 **/
type attractor 2D          /* attractor interpretation in 2D */

/*
 * interpretation: how to produce an image from specified model type
 */
mode adaptive              /* generate points using adaptive method */
precision pixel            /* calculate using pixel size precision */
render constant            /* assign the same colour to all points */
window -0.3 1.3 -0.15 1.05 /* specify a viewing window on R^2 */

/*
 * transformations: specify affine transformations comprising IFS
 */
xform
        probability 0.33
        scale 0.5

xform
        probability 0.34
        translate 1.0 0.0, scale 0.5

xform
        probability 0.33
        translate {c(p/3)} {s(p/3)}, scale 0.5 /* use bc for trig */
```

Table 6.1: A sample TSL file describing an interpretation of the Sierpiński gasket.

### Transformation qualifiers

#### rotate

specify a rotation transformation.

`rotate` *angle*

where *angle* is a real number. The matrix constructed for the rotation in two dimensions is:

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`rotate` *angle axis*

where *angle* is a real number and *axis* is either "x", "y", or "z". The matrices constructed for the rotations about each of the axes respectively are:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**scale**

specify a scaling transformation.

`scale` *s*

where *s* is a real number. The matrices for uniform scaling, for two and three dimensions are generated as follows:

$$\begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

`scale` *sx sy*

where *sx, sy* are real numbers. With these different scaling factors, the following matrix is constructed for two dimensions:

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

`scale` *sx sy sz*

where *sx, sy, sz* are real numbers. With these different scaling factors, the following matrix is constructed for three dimensions:

$$\begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

**translate**

specify a translation transformation.

`translate` *dx dy*

where *dx, dy* are real numbers. The translation matrix generated for two dimensions is given below:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ dx & dy & 1 \end{bmatrix}$$

`translate` *dx dy dz*

where *dx, dy, dz* are real numbers. The translation matrix generated for three dimensions is given below:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ dx & dy & dz & 1 \end{bmatrix}$$

**matrix**

specify a transformation matrix directly. The last column of the matrix in either dimension is constant, since no perspective transformations are used.

`matrix` *matrix*

where *matrix* is a list of either 9 or 16 real numbers, depending on the dimension, specified in row major order as shown below:

$$\begin{matrix} a & c & 0 \\ b & d & 0 \\ l & m & 1 \end{matrix} \qquad \begin{matrix} a & d & g & 0 \\ b & e & h & 0 \\ c & f & i & 0 \\ l & m & n & 1 \end{matrix}$$

### Control qualifiers

#### accept/produce

specify how the current transformation is to be used in transitions between states of an automaton associated with the IFS.

`accept` *statelist* `produce` *statelist*

where *statelist* is a list of one or more non-negative integers.

#### probability

specify the probability associated with a transformation. The probability is used in the stochastic mode of attractor generation, both to determine which transformation to apply next and to specify an invariant measure supported by the attractor. The probabilities are also used in the adaptive mode to approximate this invariant measure. The probability of any one transformation being applied is calculated as a fraction of the total specified probabilities. If no probabilities are given, they are all assigned an equal value.

`probability` *value*

where *value* is a positive real number.

**probability vector**

specify the probabilities of transitions occuring from the current state. The collection of these for all transformations forms the $N \times N$ stochastic matrix used to specify the Markov chain associated with the IFS. If no vector for a transformation is specified, then all transitions are assigned equal probabilities.

`probability vector` $p_1 \ldots p_N$

where $p_1, \ldots, p_N$ is a list of probabilities.

Table 6.2 gives an example of how to specify an IFS with finite state control and Table 6.3 provides a sample of how to use control from a Markov chain.

### 6.1.3 IFS interpretation

Interpretation of an IFS can be carried out according to any of the four methods described in this thesis namely attractor, distance, disks, and escape-time. The attractor type, where only the points of the attractor are computed is most flexible, since a wide variety of algorithms are available to generate an appropriate point set, as described in Chapter 2. The remaining three types are more restrictive as they are implemented only for two dimensions. Acceptable algorithms for visualizing the volumes which would be created in three dimensions have not yet been explored.

Interpretation commands can be broken into groups for computation, viewing and rendering. Since each command is generally applicable only in a subset of all possible cases, the description indicates, using a tabular form, which form of the command is applicable to each type. If a particular command is used by all types, it is indicated by the word ALL in the type description for that command.

```
#define STEM 1
#define BRANCH 2

initial state STEM
final state BRANCH

xform
        accept STEM produce STEM, BRANCH
        scale 0.5

xform
        accept STEM produce STEM, BRANCH
        translate 0.0, 1.0 scale 0.5

xform
        accept BRANCH produce BRANCH
        scale 0.5, rotate -45.0 translate 0.0, 1.0

xform
        accept BRANCH produce BRANCH
        scale 0.5, rotate 45.0 translate 0.0, 1.0
```

Table 6.2: TSL description of the dichot IFS using finite state control.

---

**type**

specify the type of interpretation to be applied to the IFS.

`type attractor [2D|3D]`

specify an attractor-type interpretation, optionally specify dimension. Two dimensional attractor type is the default. See Table 6.4 for an example.

`type distance [2D]`

specify a distance-type interpretation, optionally specify dimension. See Table 6.5 for an example.

```
initial state 0

xform
        scale 0.5
        probability vector 0.33 0.34 0.33

xform
        translate 1. 0 scale 0.5
        probability vector 0.50 0.00 0.50

xform
        translate {c(p/3)} {s(p/3)} scale 0.5
        probability vector 0.00 1.00 0.00
```

Table 6.3: TSL description of the Sierpiński gasket IFS with a variation provided using a Markov probability transition matrix.

---

`type disks [2D]`

specify a disks-type interpretation, optionally specify dimension. See Table 6.6 for an example.

`type escape [2D]`

specify an escape-type interpretation, optionally specify dimension. See Table 6.6 for an example.

### Computation

#### control

Specify if control information should be used in interpreting IFS.

| *type* | *command* |
|---|---|
| ALL | `control` ⟨ `on\|off` ⟩<br>Default is to use control information if it is present. |

```
/**
 * 3D Sierpinski gasket
 **/

type attractor 3D


mode adaptive
precision 0.01
render constant
basecolour 256

start 1.0 10.0 5.0

xform
        scale 0.5

xform
        translate 1.0 0.0 0.0, scale 0.5

xform
        translate 0.5 0.8660 0.0, scale 0.5

xform
        translate 0.5 0.2887 0.8165, scale 0.5
```

Table 6.4: TSL for three-dimensional extension to the Sierpiński gasket.

```
/**
 * Barnsley's Black Spleenwort Fern
 **/

type distance 2D
mode coherence
window -6. 6. -1.0 11.0
gridsize 512 512

xform
        matrix  0.00    0.00    0.00
                0.00    0.16    0.00
                0.00    0.00    1.00

xform
        matrix  0.20    0.23    0.00
                -.26    0.22    0.00
                0.00    1.60    1.00

xform
        matrix  -.15    0.26    0.00
                0.28    0.24    0.00
                0.00    0.44    1.00

xform
        matrix  0.85    -.04    0.00
                0.04    0.85    0.00
                0.00    1.60    1.00
```

Table 6.5: TSL file for Black Spleenwort Fern

```
/**
 * Star attractor
 **/
type disks 2D
precision 0.1
traversal recursive scanline
radius exponential 0.67
window -1.25 1.25 -1.25 1.25
gridsize 512 512

xform
        scale 0.5, translate -0.5 0.0

xform
        scale 0.5, translate 0.5 0.0

xform
        scale 0.5 0.25

xform
        scale 0.5 0.25, rotate 45.

xform
        scale 0.5 0.25, rotate -45.
```

Table 6.6: TSL file for the star attractor.

**final state**

Specify the final states for the automaton. If none are specified explicitly, then are all taken to be final.

| *type* | *command* |
|---|---|
| ALL | `final state` *statelist*<br>where *statelist* is a list of one or more non-negative integers. |

```
/**
 * Dragon Curve
 **/

type escape
mode grid continuous
infinity circle 5.0 0.2 0.3
window -1.2, 1.2, -1.3, 1.9
gridsize 384 512

xform
        scale 0.7071, rotate 45.0

xform
        scale 0.7071, rotate 135.0, translate 0.0 1.0
```

Table 6.7: TSL for a dragon curve with escape-time interpretation.

**gridsize**

Specify the size of grid to use for non-interactive computations.

| *type* | *command* |
|---|---|
| distance<br>disks<br>escape | `gridsize` *x y*<br>where *x, y* are positive integers. The display size for an IRIS 3130 is 1024 x 768 and for an IRIS 4D it is 1280 x 1024. If this statement is not used, a default of 256 x 192 is assumed. |

**infinity circle**

Specify the radius and position of the infinity circle. The actual shape of the infinity circle will depend upon the metric used.

| *type* | *command* |
|---|---|
| escape | `infinity circle` ***`radius`*** `[`***`x y`***`]`<br>where *radius* is some positive real number and *x, y* are real coordinates. If the position of the circle is not given, the circle is centered on the bounding box for the the attractor. If no *radius* is specified, it is calculated for the smallest circle which will cover the window. |

**initial state**

Specify the initial state for the control mechanism. For the Markov chain, since states are associated with transformations, the effect is to specify the initial transformation, where transformations are numbered beginning with zero.

| *type* | *command* |
|---|---|
| ALL | `initial state` *state*<br>where *state* is a non-negative integer. |

**levels**

Specify the number of levels to use in constructing the transformation tree.

| *type* | *command* |
|---|---|
| attractor<br>escape | `levels` *number*<br>where *number* is some positive integer. For type attractor, it is only recognized when using enumeration mode. The default is 6 levels. |

**metric**

Specify the distance function to be used in computations. The choice of metric will also alter the shape of disks in the image.

| *type* | *command* |
|---|---|
| ALL | `metric euclidean`<br>Distance between the points $(x_0, y_0)$ and $(x_1, y_1)$ is calculated as $\sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$. The Euclidean distance function is used by default.<br>`metric manhattan`<br>Distance between the points $(x_0, y_0)$ and $(x_1, y_1)$ is calculated as $\mid x_1 - x_0 \mid + \mid y_1 - y_0 \mid$. |

**mode**

Specify the mode of computation.

| *type* | *command* |
|---|---|
| attractor | `mode enumeration [breadth\|depth] [premult\|postmult]`<br>Use an enumeration method (either Algorithm 2.1 or 2.2). If no options are specified, the tree is traversed depth-first and transformations are composed using premultiplication.<br>`mode stochastic`<br>Use the random iteration method (Algorithm 2.3).<br>`mode adaptive`<br>Use the adaptive-cut method (Algorithm 2.4). This is the default mode for attractor generation. |

| *type* | *command* |
|---|---|
| distance disks | `mode basic`<br>Use the basic distance method (Algorithm 3.1).<br>`mode subset [`*`size`*`]`<br>where *size* is some positive real number. Use the fixed subset method (Algorithm 3.2). If the *size* of the subsets is not given, a value equal to 8 times the precision of calculation is used.<br>`mode adaptive`<br>Use the adaptive subdivision method (Algorithm 3.3) without use of point-to-point coherence.<br>`mode coherence`<br>Use the adaptive subdivision method (Algorithm 3.3) with coherence. This is the default mode for distance calculations. |
| escape | `mode basic [discrete\|continuous]`<br>Use the direct computation method (Algorithm 4.1) to evaluate a form of the escape-time function. If neither is specified, the continuous form is used.<br>`mode grid [discrete\|continuous]`<br>Use the grid approximation method (Algorithm 4.2) to evaluate a form of the escape-time function. If neither is specified, the continuous form is used. This is the default mode for escape-time calculation. |

**points**

Specify the number of points to be used in attractor generation.

| *type* | *command* |
|---|---|
| attractor | `points` $n$<br>where $n$ is some positive integer. This statement is recognized only when stochastic mode is used. The default value is 10000. |

**precision**

Specify the accuracy of the approximation desired. The accuracy is measured by the product $D_{max} \times \lambda_{comp}$.

| *type* | *command* |
|---|---|
| ALL | `precision` ⟨*epsilon*\|`pixel`⟩ [*levels*]<br>where *epsilon* is a positive real number and *levels* is a positive integer. In place of *epsilon*, the keyword `pixel` may be used to indicate that the pixel size should be used. The value of *levels* can be used to limit computations. If it is not specified, a value of 20 is used. For type attractor, this statement is only recognized in adaptive mode. |

**radius**

Specify a scaling function for the radii of disks.

| *type* | *command* |
|---|---|
| disks | `radius` ⟨`linear`\|`exponential`⟩ [*scale*]<br>where *scale* is a positive real number not greater than one. Scaling can done by either a linear or exponential function. With radius $r$, linear scaling returns ($scale \times r$) and exponential scaling returns ($scale \times re^{-r}$). The default is to apply no scaling (equivalent to linear scaling with $scale = 1$. |

**start**

Specify a starting point for attractor generation.

| *type* | *command* |
|---|---|
| attractor | `start` ***x y*`[`*z*`]`<br>where *x y z* are real values. The default is to calculate the center of bounding box for the attractor. |

**traversal**

Specify the method for traversing the grid for disks calculation.

| *type* | *command* |
|---|---|
| disks | `traversal` ⟨`recursive|iterative`⟩ `[spiral|scanline]`<br>The recursive mechanism will continue computation along a particular path until the specified precision is reached, whereas the iterative mechanism will make one only computation before proceeding to the next grid cell, according to either a spiral or scanline path. By default, the grid is traversed in a spiral pattern, using a recursive mechanism. |

**Viewing in two dimensions**

**window**

Specify a window in world coordinates.

| *type* | *command* |
|---|---|
| ALL | `window` *xmin xmax ymin ymax*<br>Default is to calculate window just big enough to hold the attractor. |

### Viewing in three dimensions

#### aspect

Specify the aspect ratio.

| *type* | *command* |
|---|---|
| attractor | `aspect` *x y*<br>where *x, y* are positive integers. The default is 4:3<br>The horizontal view angle is computed from the aspect ratio and the vertical viewing angle. |

#### eye

Specify the eye location.

| *type* | *command* |
|---|---|
| attractor | `eye` *x y z*<br>The default eye location is (0.0, 0.0, -1.0). |

#### farplane

Specify the far clipping plane.

| *type* | *command* |
|---|---|
| attractor | `farplane` *distance*<br>where *distance* is a real value. Default is 100.0<br>Objects farther than this distance will not be mapped to the screen. |

**fovy**

Specify the vertical viewing angle.

| *type* | *command* |
|---|---|
| attractor | `fovy` *angle*<br>where *angle* is a real number. The default vertical viewing angle is 25 degrees. The angle is measured from the eye position, with the line of sight bisecting the angle. The horizontal viewing angle is calculated from this and the aspect ratio. |

**nearplane**

Specify the near clipping plane.

| *type* | *command* |
|---|---|
| attractor | `nearplane` *distance*<br>where *distance* is a real number. Default is 0.1 objects nearer than this will not be seen. |

**twist**

Specify an angle for rotation about the line of sight.

| *type* | *command* |
|---|---|
| attractor | `twist` *angle*<br>where *angle* is a real number. The default angle of 0 degrees will cause the positive 'y' direction in the world space to be aligned with the positive 'y' direction in image space. |

**vrp**

Specify the view reference point.

| *type* | *command* |
|---|---|
| attractor | `vrp` *x y z*<br>where *x y z* are real numbers. Default is to set vrp to the center of the bounding box containing the attractor. The vector from the eye position to the view reference point defines the line of sight. |

## Rendering

**basecolour**

Specify the base colourmap index.

| *type* | *command* |
|---|---|
| attractor<br>disks | `basecolour` ***index*** `[`*r g b*`]`<br>where *index* is an integer in the range [0,4095] and *r, g, b* are integers in the range [0,255]. If *r, g, b* are specified, that colour will be placed into the colour lookup table at location *index*. When the graphics window is opened, it will be cleared to the colour stored at that location. All other colours are referenced from this position. The default is 256. This statement is only recognized when the program is run interactively. |

**ignore**

Specify the number of image points to ignore before starting to plot points.

| *type* | *command* |
|---|---|
| attractor | `ignore` $n$<br>where $n$ is a positive integer. By default, no points will be ignored. This statement is only recognized in the enumeration and stochastic modes of attractor generation. |

**pointsize**

Specify size of disk to use when rendering the attractor interactively.

| *type* | *command* |
|---|---|
| attractor | `pointsize` ⟨*size*\|`pixel`⟩<br>where *size* is a real number in the range (0,1]. The keyword `pixel` indicates that each point should be drawn as a pixel. Each point, in adaptive mode, represents a set with radius equal to the precision of computation and if a value of *size* is specified, it determines how the radius is to be scaled before drawing. The default is to draw points as pixels. |

**render**

Specify the rendering method to be used. Unless otherwise noted these statements are only recognized when the program is run interactively.

| *type* | *command* |
|---|---|
| attractor | `render constant`<br>All points will be drawn with the colour stored at location basecolour+1. This is the default.<br>`render depthcue [`*range*`]`<br>where *range* is a positive integer. Points will be coloured based on their distance from the eyepoint. The point closest to the eye will be assigned basecolour+1 and the one furthest will be assigned basecolour+range. This statement is only recognized for 3D.<br>`render index [`*n*`]`<br>where $n$ is a positive integer. Points will be coloured based on the last $n$ transformations applied to the point. Colour map indices are computed according to the formula basecolour+$t_{i_n}N^{n-1} + \ldots + t_{i_1}$, where $N$ is the number of transformations and $t_i \in (0, \ldots, N-1)$.<br>`render measure [`*range*`]`<br>where *range* is a positive integer. Points will be coloured according to the accumulated "mass" there. The value of *range* indicates of colour map indices to use when the program is run interactively. If it is not specified, a value of 256 is used. This statement is recognized only in stochastic and adaptive modes, but can be used when the program is run non-interactively. |

| *type* | *command* |
|---|---|
| attractor | `render state`<br>Points will be coloured according to their final state. Colours are assigned from consecutive locations beginning at basecolour+1. This statement is only recognized in enumeration and adaptive modes. |
| disks | `render constant`<br>All disks will be drawn with the colour stored at location basecolour+1. This is the default.<br>`render index [`*n*`]`<br>where $n$ is a positive integer. Disks will be coloured based on the last $n$ transformations applied to the point. Colour map indices are computed according to the formula basecolour + $t_{i_n} N^{n-1} + \ldots + t_{i_1}$, where $N$ is the number of transformations and $t_i \in (0, \ldots, N-1)$.<br>`render size [`*range*`]`<br>where *range* is a positive integer. Disks will be coloured according to their size. Colours are assigned from the specified *range*. Any disk greater than or equal to the diagonal of the viewing window will be assigned basecolour+range. If no *range* is specified, a value of 256 is used. |

**setcolour**

Specify RGB colours to map, starting at basecolour.

| *type* | *command* |
|---|---|
| attractor disks | `setcolour` $r_1 g_1 b_1 \ldots r_n g_n b_n$<br>where $r_i g_i b_i$ are integer values in the range [0,255]. These colours are loaded into the colour lookup table at consecutive locations following the basecolour index.<br>`setcolour ramp` [$r_s g_s b_s r_e g_e b_e$] [*range*]<br>where $r_s, g_s, b_s, r_e, g_e, b_e$ are integer values in the range [0,255] and *range* is a positive integer. Create a ramp of colours starting at basecolour+1 with $r_s g_s b_s$ and ending at basecolour+range with $r_e g_e b_e$. The default is to create a ramp from 0, 0, 0 (black) to 255, 255, 255 (white) over a range of 256 entries. This statement is recognized only when the program is run interactively. |

## 6.2 GTI

As the name *Graphical Transformation Interpreter* implies, this program is designed to read and interpret an input file containing a TSL description of an IFS. The output generated by the processing of the file may be either an image on a graphics display or data in a binary file. Although a large part of this program is designed for interaction on a graphics workstation screen, much of the functionality of the program is maintained even on systems which do not have graphics support.

### 6.2.1 Usage

The following section provides an explanation of the invocation and options for the *gti* program. Processing occurs in stages:

- the input file is read and the interpretation parameters are set
- the diameter of the attractor is estimated according to the method described in Chapter 2, as required. Most processing options make use of this diameter estimate.
- interpretation is performed according to the parameter values and the selected command line options.

Usage: gti [options] TSLfile

where options can be:

**-b** open the graphics window without a border

This option is useful if a window is selected which has the same dimension(s) as the graphics screen, in which case the border would obscure part of the image displayed in the window.

**-c** run without interactive graphics

**Attractor** type files will write a binary file containing the list of points computed for the attractor. **Distance** type files will write a binary file containing the distance values for the entire field defined by the `window` statement.

**Disks** type files will write a binary file containing the list of disks computed for the complement of the attractor, enclosed in the region defined by the `window` statement. **Escape** type files will write a binary file containing the escape-time values for the entire field defined by the `window` statement. Note that there is no interactive processing option for files with this type.

**-i** include index information

Index information refers to the history of transformation applications associated with a particular point or field value. Due to storage constraints, there is a limit to the number of levels of index information which may be stored for any given attractor. This maximum number of levels is determined as: $\lfloor \log(65535)/\log(N+1) \rfloor$ where $N$ is the number of transformations in the IFS. When saving the binary file, index information is normally excluded because of the additional storage requirements. If it is included, the *goc* program is able to perform additional types of rendering with the data.

**-l** create a log file

Create a file which records all messages output during program execution, for example those which track the percentage completed during a lengthy computation. If no log file is specified, then all messages will be printed on standard error output.

**-o *file*** force the output to the named file

Normally, *gti* will create a filename based on the TSL input file. For example, output from "sample.tsl" would be stored in "sample.gti". In general, the base used in creating the output file name is the text which precedes the final period. If the name contains no periods, then the output file name is constructed by appending ".gti". If the output file name already exists, it is first copied to a filename with a ".bak" extension before being overwritten.

**-p** use external panels for interactive control

When this option is selected, the normal menus are replaced with a single, simple menu which facilitates interaction consistent with the approach of a

virtual laboratory [30].

**-q** run quietly

Do not output any messages during program operation.

**-s *xsize*** specify number of pixels in x

By specifying the number of pixels only in the horizontal direction, the proper aspect ratio can be maintained. In two dimensions, the number of pixels in the vertical direction is computed from the values specified in the `window` statement. In three-dimensions, the number of pixels in the vertical direction is computed from the value specified in the `aspect` statement.

**-v** select verbose output

Display additional information about various aspects of program execution.

### 6.2.2 Interactive operation

*Gti* is designed for interactive use with input files with type attractor, distance, and disks. The program can interact with the user either through menus or through external panels.

#### Menus

*Gti* employs a hierarchy of menus to provide access to a wide range of functions which can alter the screen image. This provides the user with the capability to manipulate parameters for interpretation on a strictly temporary basis in order to examine the effect of their changes.

**Attractor files** Interactive interpretation of these files will cause an image of the attractor to be drawn in the current graphics window. The menus provide the ability to alter the methods used for computation and rendering of the attractor.

#### Attractor specific functions

**Approximation** This menu allows selection of submenus which control parameters concerning the method used to approximate the attractor.

**Enumeration** Amongst the options for the enumeration mode of approximation, one may select between either breadth-first or depth-first traversal of the transformation tree, performed using either premultiplication or postmultiplication of matrices. It is also possible to select the number of levels to compute in this transformation tree.

**Stochastic** Select the number of points to use in the approximation, either 10,000, 100,000 or 1,000,000 point.

**Adaptive** Select the precision of computation, ranging form 0.1 to pixel precision.

**Point size** The point size determines the size with which each point is to be drawn. The only option which makes sense outside of adaptive is the pixel size, the other require adaptive mode of computation.

**Point rendering** Select the method in which the points of the attractor are to be rendered. Possible selections include measure, constant, index and state rendering. The number of index levels to be used is also selectable.

### General IFS functions

**Select control** Toggle the use of control information when interpreting the IFS. This will have no effect if there is no control information specified in the file. In the case when finite state control is specified, all transitions become possible. In the case of Markov chain control, the probability of application for each transformation reverts to the value specified by the `probability` statement. If these statements are not present, then each transformation is assigned an equal probability. This useful to examine the differences in structures which result with and without control.

**Select metric** Choose the metric function to be used for all distance computations. Regardless of the metric function used, the image of the attractor at the pixel

limit will be the same. However, at lower resolutions it is possible to see interesting differences based on the formula for the computation of distance. One may choose either Euclidean or Manhattan distance functions.

### General program functions

**Reread TSL file** Reread the input TSL file. This allows any changes made to the input file through editing to be applied without having to restart the program.

**Select window** Select the graphics window for *gti* and pop it in front of any other windows which may obscure it.

**Save screen image** Save the current screen image in an SGI imagelib format image. The filename is formed by appending ".*n*.ras" to the base name of the input file. For example, a screen image created from the IFS in "sample.tsl" is saved in "sample.0.ras". The number used in the file name is lowest one which will create a new name.

**Exit** Exit from the program, including the graphics environment.

**Distance files** Interaction with files with type distance permits the user to calculate the distance of selected points from the attractor, rather than compute distance for each point in the graphics window. The purpose in providing such an option is to illustrate the operation of the **Adaptive-subdivision** method (Algorithm 3.3). The point is selected by clicking the left mouse button somewhere inside the graphics window. The menus options described below provide the ability to control aspects of the way in which this procedure is visualized.

### Distance specific functions

**Computations** Select which part of the computations are visualized. Regardless of the selection made, the points which are computed in order to arrive at the distance are shown. When the distance is finally computed, a line is drawn

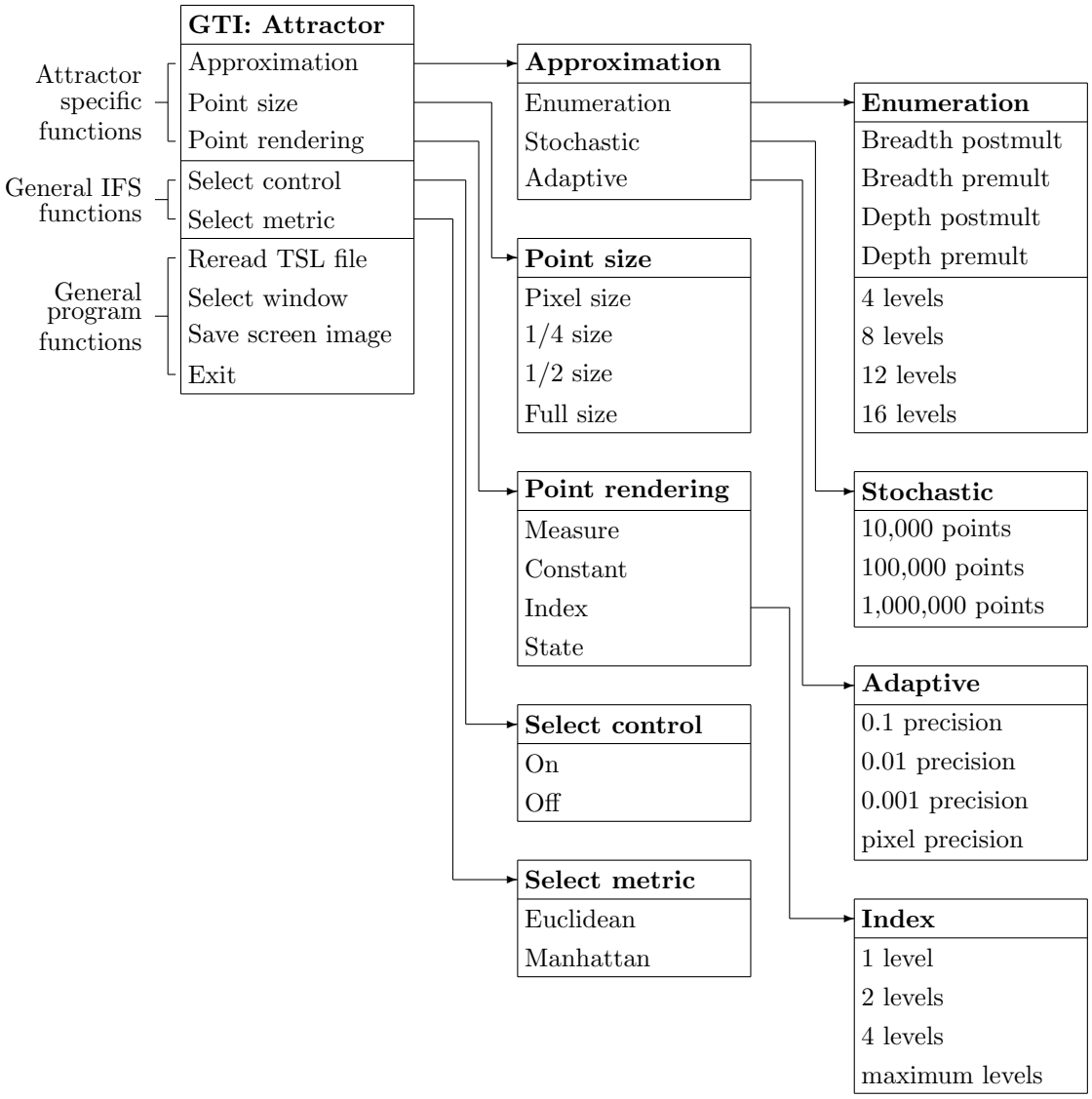



Figure 6.1: Hierarchy of menu selections for attractor files

between the selected point and the point in the attractor to which it is closest. This is the action carried out when the "Hide all computations" option is selected. The next level, selected by "Show eliminations", is to draw the disks surrounding those sets which have been removed from consideration. The last level, selected with "Show all computations", will show all disks as they are computed with the colour red used to indicate those that are eliminated, and blue to indicate the ones which cannot be eliminated at the current stage. Green is used to draw a circle about the selected point of radius equal to the distance to the nearest point in the attractor.

**Attractor** Select whether to show the attractor in its entirety or to display only those points which are used in the computations.

**Initial estimate** Select whether the initial estimate for distance should be made from the starting point or last point calculated (using coherence).

**Screen update** Select how the screen should be updated during computations. "Step and clear" will allow the user to step through computations by clicking the middle mouse button. Each click will cause the screen to be cleared and the next level of computations to be performed. "Step and clear at end" will use stepping but will not clear the screen until the end of the computations for the selected point. "Clear at end" will clear the screen at the end of all computations for the selected point but the computations will be performed without interruption.

**General IFS functions** See earlier description.

**General program functions** See earlier description.

**Disks files** Interactive interpretation of these files will cause an image of the attractor to be drawn in the current graphics window. The menus provide the ability to alter the methods used for computation and rendering of the attractor.

Interactive intepretation of these files will cause an image of the complement of the attractor to be drawn using disks inside the current graphics window. The menus

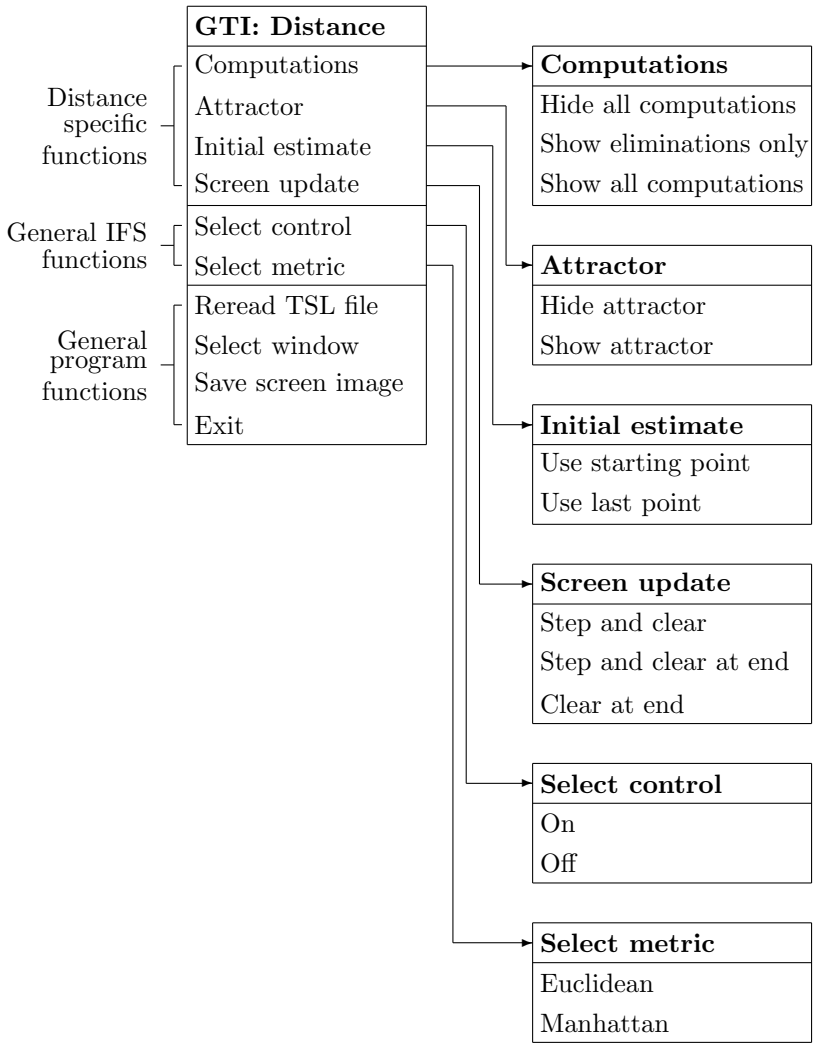



Figure 6.2: Hierarchy of menu selections for distance files

permit the alteration of the order in which disks are computed and the way in which they are rendered.

**Disks specific functions**

**Grid traversal** Select how the grid is to be traversed. The way in which the grid is traversed will have an effect on how the disks are tiled in the final image.

**Disk scaling** Select how the radii of the disks should be scaled. It is possible to use either a linear or an exponential function, additionally scaled by a variety of values.

**Disk rendering** Select how the disks to be rendered. If rendering is performed according to the size of radius, the maximum colour in the range is mapped to half the size of the window diagonal. Any radius which exceeds this will also receive this colour value. The minimum colour in the range is mapped to distances of zero. It is also possible to render the disks according to constant and index methods.

**General IFS functions** See earlier description.

**General program functions** See earlier description.

**Panels**

The second is consistent with the idea of the virtual laboratory [30] and expects the work of interaction to be done by the panels, external to *gti*. When GTI is invoked with the external panel option, only minimal menu support is provided from within *gti*.

### 6.2.3 Non-interactive operation

This mode of operation is provided so that the program can be used where direct support of graphics is not present. Also, the interactive modes are intended only

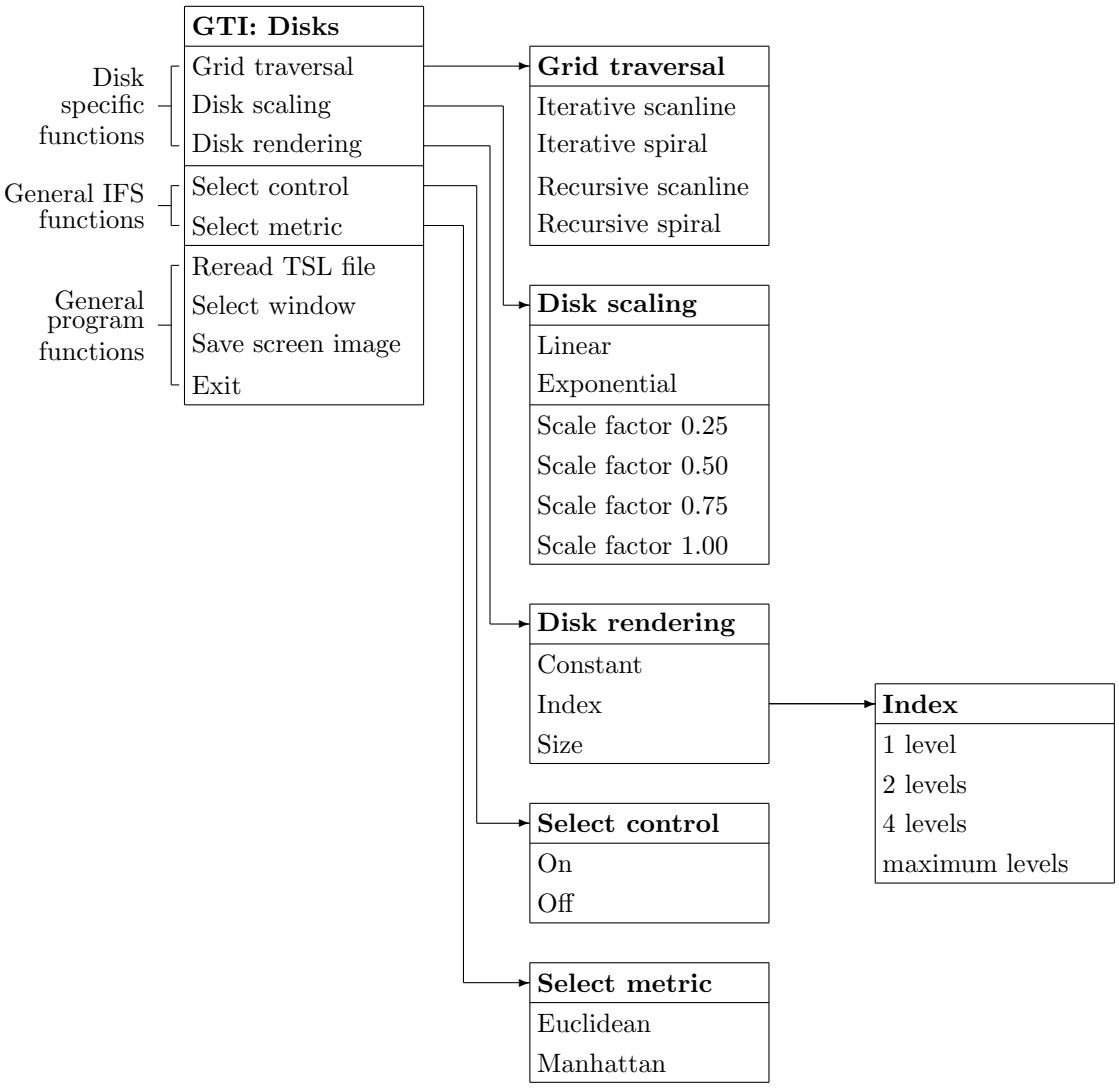



Figure 6.3: Hierarchy of selections for disks files

for simple screen graphics. For more sophisticated rendering, an intermediate binary output format has been developed to allow the data created by *gti* to be further processed into final form by other programs. This output file is only generated when the non-interactive mode is selected.

The binary file includes a section of information which precedes the data in order to provide information about how the data is structured and what functions can be performed on the data. This allows for a minimum of information to be specified on the command line of *goc* and it provides the user an additional means of identifying the contents of a file, aside from its name. Below is a description of some of the key elements in this heading section.

**type of interpretation** This describes which interpretation was used to generate the data in the file. This information is required in order to distinguish the meaning of the minimum and maximum values in the data.

**format** This describes the format of the data. Different formats exist for points, disks and field data depending on whether they also include index information. The format determines which conversion options may be applied to the data.

**metric** This describes which metric was used to compute the data. If the data is to be ray-traced, the metric information indicates which primitives are to be used in constructing the model from the data.

**number of transformations** This indicates how many transformations were contained in the IFS which generated the data file. This information is required in order to decode the index information properly.

**minimum data value** This indicates the minimum value stored in the file. It is used to flip the data before conversion, and indicate which values are part of the overlay for distance files.

**maximum data value** This indicates the maximum value stored in the file. It is used to normalize values, to indicate values which are part of the overlay for escape-time files, and to flip data before conversion.

Also stored are the extents of the bounding box, the size of grid used, and the dimension of the transformations which have value mostly as comments.

## 6.3 GOC

As the name *Gti Output Converter* implies, this program is designed to read the output file produced by *gti* and convert it into a variety of formats, either ready for display on a Silicon Graphics workstation or as input to other programs. The conversion process itself, however, need not be performed on such a workstation.

### 6.3.1 Usage

The following section provides an explanation of the invocation and options for the *goc* program. The inclusion in the binary output file from the *gti* program has allowed the amount of information required to be specified on the command line to be significantly reduced.

Usage: goc [options] infile outfile

where options can be:

**-f** flip data (min $\leftrightarrow$ max)

This option makes use of the minimum and maximum data values such that a new value $x' = max - x + min$. This option is used to reverse the orientation of the data in such a way that points in the attractor for distance files will occur at $max$ and points in the attractor for escape-time files will occur at $min$.

**-h** print header information

Print information about the size and the type of model which the file contains. If the verbose option is selected as well, then a detailed list of the file header contents are displayed, including minimum and maximum data values, the data storage format and any comments which may be present.

**-i *n*** select index mode for n levels

Use n levels of index information to create an image. This conversion is only possible if index information has been stored with the data. The value of n is limited by storage constraints. The maximum amount of index information which can be stored for a given IFS is determined as: $\lfloor \log(65535)/\log(N+1) \rfloor$ where $N$ is the number of transformations in the IFS. Colour map indices are assigned beginning at 257 with offsets from that position computed according to the formula $t_{i_n} N^{n-1} + \ldots + t_{i_1}$, where $N$ is the number of transformations in the IFS and $t_i \in (0, \ldots, N-1)$.

**-o** overlay the attractor on an index image

This conversion is only possible if index information has been stored with the data. When an index image is selected, positions which are determined to be in the attractor are assigned colour map index 256. The determination of attractor values is made by use of either the minimum or maximum data value, depending on the type of data stored (either distance or escape-time).

**-r *res*** select data resolution res

This value determines the coarseness with which the continuous field values are discretized by the colour mapping. This option need only be specified if the desired resolution is different from 1. Values of zero are mapped to colour map index 256. Other values are converted as $(\frac{(value-min)}{res} + 257)$.

**-v** verbose output

Print additional information about the program at various stages of operation.

**-F** *format* select output formats

| | |
|---|---|
| SGI | produce a Silicon Graphics imagelib format, colour mapped image file. |
| POSTSCRIPT | produce a PostScript file, suitable for printing or viewing under NeWS. |
| RAYSHADE | produce an input file for *rayshade*, a ray-tracing program written by Craig E. Kolb, Yale University. |
| GOUT | produce an input file for *gout*, a heightfield rendering program written by Dietmar Saupe, University of Bremen. |

### 6.3.2 Data conversions

The conversions performed by *goc* are listed in Table 6.8. The results of these conversions are not necessarily in a final format which can be directly examined or which can be output by the desired means. There exist many other utilities which can provide further conversion from these formats.

| image type | conversion | output format |
|---|---|---|
| ATTRACTOR 2D | disks | POSTSCRIPT |
| | spheres [a] | RAYSHADE |
| | spheres with index | RAYSHADE |
| ATTRACTOR 3D | spheres | RAYSHADE |
| | spheres with index | RAYSHADE |
| DISTANCE | colour mapped file | SGI |
| | heightfield data | RAYSHADE,GOUT |
| | heightfield data with index | RAYSHADE |
| DISTANCE | disks | POSTSCRIPT |
| | spheres | RAYSHADE |
| | spheres with index | RAYSHADE |
| ESCAPE | colour mapped file | SGI |
| | heightfield data | RAYSHADE,GOUT |
| | heightfield data with index | RAYSHADE |

Table 6.8: Conversions possible using *goc*

[a] assumes Euclidean metric by default. Pyramids are used in the case of Manhattan metric.

# Chapter 7

# Conclusions

In the study of fractals, computer graphics has been both an end and a means. The images produced can be easily appreciated without knowledge of the processes which created them. However, it is perhaps more significant that these images also ellucidate the mathematics which underlies each of them.

A study of fractals defined by iterated function systems is aided considerably by the intuition developed from the study of Julia sets by Peitgen [32] and others. The work done to explore the intricacies of the Julia sets has provided many parallels which have been profitably exploited. At the same time, the attempts to find these similarities have also brought many important differences to light. Each of the methods used to visualize the attractor and its complement has an analogue in Julia set theory.

Much of the popular appeal of iterated function systems comes from the "Chaos Game" which Barnsley uses to generate attractor approximations. Unfortunately, this "game" has also lead to the popular misconception that the attractors of IFS are random and that the probabilities must be associated with the transformations in order to define an attractor. Barnsley has admitted that in this case, "randomness is a red herring" [19]. To combat the misconception, this thesis has expounded the virtues of a wholly deterministic approach which still maintains many of the interesting properties of the stochastic approach. The work by Dubuc [13] has also favoured a deterministic approach.

In Chapter 2, the use of the Hausdorff metric was introduced in order to provide

a tool for measuring the accuracy with which an attractor is approximated. One can only measure the accuracy of an approximation if there exists something against which comparisons can be made. The **Adaptive-cut** method eliminated this difficulty by relying on the properties of Cauchy sequences to place a bound on the accuracy with which each sequence of transformations approximates its limit point. As an additional benefit, the **Adaptive-cut** method provides another solution to the problem of efficient magnification of the attractor, originally considered by Reuter [39].

The **Adaptive-cut** algorithm reopened the question of how to choose a starting point. It had been conventional wisdom to select the starting point to be a fixed point of one of the transformations in the IFS. By requiring only the last point of any sequence to be plotted, the condition that the starting point be a member of the attractor was removed, since only the accuracy of the final point is a concern. The accuracy of the approximation is measured by examining the maximum possible distance between the starting point and any point in the set. Rather than use an initial distance equal to the diameter of the set, the point at the center of the minimum enclosing disk is the better choice since it minimizes this maximum distance.

The ability to make an accurate approximation to an attractor proved to be an idea of fundamental importance in this thesis; not only for attractor generation, but also when examining the complement of the attractor, through the distance and escape-time methods.

Chapter 3 introduced the idea of computing the distance to the attractor. Although Barnsley [2] included images showing the distance to an attractor, neither the criterion used to determine the accuracy of the computations nor the method used to perform the computations are clear. However, with the ability to efficiently compute an approximation to a specified precison, it is a straightforward matter to maintain that precision in a final image produced by the distance calculations. Unlike the Julia set method which uses a function to *estimate* the distance, the distance to an IFS attractor is computed directly as the minimum distance to a set of points. The net effect is that the computation is potentially much more accurate, although considerably more time consuming. Once an accurate approximation to the attractor has been constructed, a variety of methods may be used to compute the distance of

a point to that approximating set. The **Basic-distance** method proved to be useful only when very low precision was required and its performance deteriorated exponentially as the precision increased. Again it was possible to employ the structure inherent in the attractors generated according to the **Adaptive-cut** method. The properties of Cauchy sequences were used to eliminate groups of points from further consideration. The two algorithms based on this approach were **Fixed-subset** and **Adaptive-subdivision**. The first computes all the points and divides them into subsets before any distance computations are made. This method requires large amounts of storage because all points are stored in memory, but it is computationally efficient because the points are only ever calculated once, at the beginning. The second approach performs the computations dynamically so the memory requirements are kept low but the same points need to be calculated each time. This method is much better when memory is a consideration.

The distance information was used to create two different models for the complement of the attractor: one as a field of values representing the distance of each point to the attractor and secondly as a set of disks which were each computed so as to include as much of the complement as possible. The method of disks was inspired by Fisher's method for Julia sets [17], although the accuracy of distance computations allows more accuracy in its implementation for the IFS case.

Chapter 4 discussed methods for computing the escape-time for IFS attractors. The concept of a discrete escape-time function for IFS was introduced in 1988 [37, 2]. Following that, a continuous version of the function was defined by Prusinkiewicz in the paper by Hepting et al. [22]. Although part of the original motivation for this function was to find a potential function for IFS, the absence of such a connection does not detract from the importance of the function. The escape-time function depends on several parameters which are not a concern in the case of distance. The selection of these parameters, as discussed in Chapter 4, is initially arbitrary. Interestingly enough, by considering the radius of the infinity circle as the maximum initial distance used in the methods of Chapter 2, that criterion can be applied to select the remaining parameters as a function of the accuracy of the approximation. There is a definite relationship between the distance and escape-time functions, as was proved

by Saupe in the paper by Hepting et al. [22]. In the cases when the transformations do not scale equally, the escape-time method provides a tool to examine the relative weight of each transformation in determining the shape of the attractor. The **Basic-escape** method, although reasonably efficient is still quite costly. Again, study of the properties of points leads to ways of speeding computation, either by using domains or approximating the escape-time values from a grid. The **Approximate-escape** does not suffer from the same lack of generality as the domain method, yet it is an approximation technique so the computed values are not exact.

Chapter 5 examined methods of controlling transformation application as a means of expanding the class of fractal shapes which are definable by iterated function systems. If IFS techniques are to prove useful in modelling natural structures, these techniques must include ways to capture shapes which are not strictly self-affine. Research into the control of transformation application is a relatively new addition to the study of IFS's. There are two basic directions which the research has followed in selecting a control mechanism, which parallels the standard IFS development: the stochastic case (RIFS [3]) and the deterministic case (CIFS [35]). As in the standard case, the stochastic control mechanism implemented using Markov chains is the more popular of the two research directions. The technique of *mixing* [6] has been applied to the Markov chain approach, to allow the attractors of several independent IFS's to be combined in one final image. The deterministic control mechanism implemented using a finite-state acceptor, appears to have similar advantages over the Markov chain approach as the deterministic method holds over the "Chaos Game" for standard IFS's. However, there are many problems which remain unanswered.

This work was carried out with the concept of scientific visualization in mind, which made the computer graphics aspect an essential element in this study. The software developed in conjunction with this thesis, described in Chapter 6 was designed to be flexible and allow for experimentation in a variety of ways.

This study has revealed several areas which remain open for further research, in both the areas of modelling and rendering:

- The criterion for terminating the **Adaptive-cut** method is not always optimal. Currently, the only available solution of computing the Lipschitz constant for

each composite transformation, by computing the eigenvalues for the matrix, would add substantially to the computation time. A question remains if there is a way to estimate the actual size of the sets without performing the direct computation. The evaluation of eigenvalues is particular to the Eucliean norm for matrices while other matrix norms are substantially easier to compute. Would it be possible to perform the computations using some other norm? If so, would it be as accurate as the Euclidean norm, and would it retain any geometric significance?

When the IFS is overlapping, portions of the attractor are computed more than once which leads to a waste in effort. Dubuc has done some work to eliminate this waste by allowing Boolean operations on the grid, but more study is required.

- Rendering of IFS attractors in three dimensions was examined briefly in Chapter 2. In order to generate precise images, ray-tracing was used. The approach to ray-tracing taken here is different than other current approaches to the ray-tracing of other deterministic fractals. A model which uses spherical primitives is easily specified to a ray-tracing program whereas these programs in general are not equipped to handle the unbounding volume calculations required by the method suggested by Hart [21]. In the naive implementation of the spherical approach used here, rendering an approximation of very high precision requires a tremendous amount of primitives. It should be possible to make use of the self-affinity in the attractor to create a more concise description of the model. It is not clear which approach for the ray-tracing of these attractors will ultimately prove to be more successful. Rendering of the field functions in three dimensions has not been explored at all, a question which belongs to the larger area of volume visualization.

- Since many improvements are possible for attractor calculation, it is not clear whether corresponding improvements could be possible in distance calculations. The index images for the distance calculations are reminiscient of the Voronoi partitions of the plane. Is it possible to use a Voronoi pre-processing phase to

get better performance, especially when the attractor is overlapping?

- The selection of parameters for the escape-time function has been performed in a rather arbitrary fashion. Is there some way to determine the optimal radius of the infinity circle for a given attractor? Is there an analytic way to determine from a particular IFS what the minimum and maximum values for the radius of the infinity circle should be?

- The **Adaptive-cut** method would seem to apply to the case of CIFS but a strong theoretical base still must be developed.

Overall, this thesis has succeeded in examining the relationship which exists between IFS attractors and Julia sets. Many of the insights and algorithms come from the cooperative research between the University of Bremen and the University of Regina, the results of which are also described in the paper by Hepting et al. [22]. Although the application of IFS techniques to the problems of modelling was not studied in this thesis to the depth which was first anticipated, the research which was completed hopefully provides some clues the many exciting problems which remain unanswered.

# Bibliography


[1] H. Anton. *Elementary Linear Algebra*. John Wiley and Sons, Toronto, 1984.

[2] M. F. Barnsley. *Fractals Everywhere*. Academic Press, 1988.

[3] M. F. Barnsley. Recurrent iterated function systems. *Constructive Approximation*, 1(1), 1989.

[4] M. F. Barnsley and S. G. Demko. Iterated function systems and the global construction of fractals. *Proceedings of the Royal Society of London, Series A*, (399):243–275, 1985.

[5] M. F. Barnsley and A. D. Sloan. A better way to compress images. *BYTE*, pages 215–233, 1988.

[6] M. A. Berger. Images generated by orbits of 2-d markov chains. *Chance*, 2(2):18–28, 1989.

[7] R. L. Burden and J. D. Faires. *Numerical Analysis*. PWS-KENT Publishing Company, Boston, fourth edition, 1989.

[8] L. Cherry and R. Morris. Bc - an arbitrary precision desk-calculator language. In *Unix User's Manual Supplementary Documents*, chapter USD:6. USENIX Association, 1986.

[9] R. Cowan et al. *Graphics Library User's Guide*. Silicon Graphics Incorporated, Mountain View, California, 1987.

[10] H. S. M. Coxeter. *Introduction to Geometry*. John Wiley and Sons, Toronto, 1980.

[11] S. Demko, L. Hodges, and B. Naylor. Construction of fractal objects with iterated function systems. *Computer Graphics*, 19:271–278, 1985.

[12] C. W. Dodge. *Euclidean Geometry and Transformations*. Addison-Wesley, Don Mills, Ontario, 1972.

[13] S. Dubuc and A. Elqortobi. Approximation of fractal sets. *Journal of Computational and Applied Mathematics*, 29:79–89, 1990.

[14] J. Elton. An ergodic theory for iterated maps. *Journal of Ergodic Theory and Dynamical Systems*, 7:481–488, 1987.

[15] D. J. Elzinga and D. W. Hearn. The minimum covering sphere problem. *Management Science*, 19(1):96–104, 1972.

[16] K. J. Falconer. *Fractal Geometry: Mathematical Foundations and Applications*. John Wiley and Sons, Toronto, 1990.

[17] Y. Fisher. Exploring the mandelbrot set. In H.-O. Peitgen and D. Saupe, editors, *The Science of Fractal Images*, chapter Appendix D, pages 287–296. Springer-Verlag, 1988.

[18] J. D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics*. Addison-Wesley, Don Mills, Ontario, 1982.

[19] J. Gleick. *Chaos: making a new science*. Penguin Books, Markham, Ontario, 1988.

[20] A. R. Gourlay and G. A. Watson. *Computational Methods for Matrix Eigenproblems*. John Wiley and Sons, Toronto, 1973.

[21] J. C. Hart, D. J. Sandin, and L. H. Kauffman. Ray tracing deterministic 3-d fractals. *Computer Graphics*, 23(3):289–296, 1989.

[22] D. Hepting, P. Prusinkiewicz, and D. Saupe. Rendering methods for iterated function systems. In *Fractal '90*, 1990.

[23] J. E. Hutchinson. Fractals and self-similarity. *Indiana University Journal of Mathematics*, 30(5):713–747, 1981.

[24] Stephen C. Johnson. Yacc: Yet another compiler-compiler. In *Unix Programmer's Manual Supplementary Documents 1*, chapter PS1:15. USENIX Association, 1986.

[25] R. Y. Kain. *Automata Theory: Machines and Languages*. McGraw-Hill Book Company, Toronto, 1972.

[26] R. H. Kasriel. *Undergraduate Topology*. Robert E. Krieger Publishing Company, Malabar, Florida, 1971.

[27] L. G. Kelly. *Handbook of Numerical Methods and Applications*. Addison-Wesley, Don Mills, Ontario, 1967.

[28] M. E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. In *Unix Programmer's Manual Supplementary Documents 1*, chapter PS1:16. USENIX Association, 1986.

[29] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman, New York, 1982.

[30] L. Mercer, P. Prusinkiewicz, and J. Hanan. The concept and design of a virtual laboratory. In *Graphics Interface '90 Conference Proceedings*, 1990.

[31] M. K. Molloy. *Fundamentals of Performance Modeling*. Macmillan Publishing Company, New York, 1989.

[32] H.-O. Peitgen and P. H. Richter, editors. *The Beauty of Fractals*. Springer-Verlag, Heidelberg, 1986.

[33] H.-O. Peitgen and D. Saupe, editors. *The Science of Fractal Images*. Springer-Verlag, New York, 1986.

[34] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, 1985.

[35] P. Prusinkiewicz and D. Hepting. Controlled iterated function systems. Manuscript, 1991.

[36] P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. The Virtual Laboratory. Springer-Verlag, New York, 1990.

[37] P. Prusinkiewicz and G. Sandness. Koch curves as attractors and repellers. *IEEE Computer Graphics and Applications*, 8(6):26–40, November 1988.

[38] R. Reimann et al. *4Sight Programmer's Guide*. Silicon Graphics Incorporated, Mountain View, California, 1990.

[39] L. Reuter. *Rendering and Magnification of Fractals Using Iterated Function Systems*. PhD thesis, Georgia Institute of Technology, 1987.

[40] G. Sandness. Iterative geometric transformations. Master's thesis, University of Regina, 1988.

[41] R. Sedgewick. *Algorithms*. Addison-Wesley, Don Mills, Ontario, 2 edition, 1988.

[42] J. M. Snyder and A. H. Barr. Ray tracing complex models containing surface tesselations. *Computer Graphics*, 21(4):119–128, 1987.

[43] D. L. Stancl and M. L. Stancl. *Real Analysis with Point-Set Topology*. Marcel Dekker, Inc., New York, 1987.

[44] Adobe Systems. *PostScript Language Reference Manual*. Addison-Wesley, Don Mills, Ontario, 1986.

[45] R. F. Williams. Compositions of contractions. *Bol. Soc. Brasil. Mat.*, 2:55–59, 1971.

# Appendix A

# Colour Plates

Colour plates are not included in this version. Please refer to the black and white versions within the text.

**Figure 2.23** Desktop Tetrahedron (Daryl Hepting, Allan Snider 1990). A collection of over 20,000 spheres was used to produce this approximation. The radius of spheres is equal to the precision of calculations. The fuzzy shadows and the wood grain desk were produced with the ray tracer.

**Figure 3.12** Sierpiński Ashtray (Daryl Hepting, Allan Snider 1989). This is a height-field interpretation of the distance to the Sierpinski gasket. Values were inverted to that the gasket would appear highest. With a marble texture applied to the heightfield, it is reminiscient of a large marble ashtray which might be found on a coffee table somewhere.

**Figure 3.15** Sierpiński Gasket in Bubbles (Daryl Hepting 1989). Distance function was used to create disks which could not possibly contain the attractor. For effect, these disks were converted to spheres and ray-traced.

**Figure 5.2** Fractal Carrot Leaf (Daryl Hepting, Przemyslaw Prusinkiewicz 1989). Spheres surround the carrot leaf shaped attractor. Each sphere touches some part of the attractor and they are used to exclude the points of the complement set so that only the attractor itself is shown in black.

**Figure 3.16** Reflecting Fern (Daryl Hepting, Allan Snider 1990). Barnsley's famous fern is done as a reflecting pond surrounded by lush green spheres which represent the complement of the set.

**Figure 3.14** Fiery Dragon (Daryl Hepting 1990). The dragon curve is highlighted in yellow, offset by the fiery red spheres which cover the complement set. This is an example of an exponential scaling function being applied to the spheres, as they are all of a more even size.

**Figure 3.19** Valley of the Stars (Daryl Hepting 1990). The Manhattan distance function was used to eliminate points from the complement of the attractor, leaving only the star itself in the valley. Because a different distance function was used, the pyramid shape replaces the sphere but it still represents the locus of points which are equidistant from its center.

**Figure 4.17** Dragon Mesa (Przemyslaw Prusinkiewicz, Dietmar Saupe, Daryl Hepting 1989) A sombre scene from another world. The dragon curve was created from escape-time data using a large infinity circle. The escape-time values were interpreted as a heightfield for rendering.

**Figure 4.20** Butte Manhattan (Daryl Hepting 1990). Ray tracing was used to achieve the sky and texture on the plane. The butte is actually the escape-time heightfield for the Sierpinski gasket, but using the Manhattan metric instead. A rather small infinity circle was used here.

**Figure 5.2** The Great Fern Dune of Goron III (Daryl Hepting, Przemyslaw Prusinkiewicz, Allan Snider 1990). The planet, moon and desert landscape were created by ray-tracing. The fern was created from escape-time data with a rather large infinity circle to give a smooth appearance to the dune.