Vertex-Labeling Algorithms for the Hilbert Spacefilling Curve

John J. Bartholdi, III [*] Paul Goldsman

January 11, 2000

Abstract

We describe a method, based on vertex-labeling, to generate algorithms for manipulating the Hilbert spacefilling curve in the following ways:

1. Computing the image of a point in R^1.
2. Computing a pre-image of a point in R^2.
3. Drawing a finite approximation of the curve.
4. Finding neighbor cells in a decomposition ordered according to the curve.

Our method is straightforward and flexible, resulting in short, intuitive procedures that are as efficient as specialized procedures found in the literature. Moreover, the same method can be applied to many other spacefilling curves. We demonstrate vertex-labeling algorithms for the Sierpinski and Peano spacefilling curves, and variations.

Key words: *Hilbert curve, Sierpinski curve, Peano curve, spacefilling curve*

[*]Both authors can be reached at the School of Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta, Georgia 30332-0205 USA. E-mail: john.bartholdi@isye.gatech.edu, and pgold@isye.gatech.edu

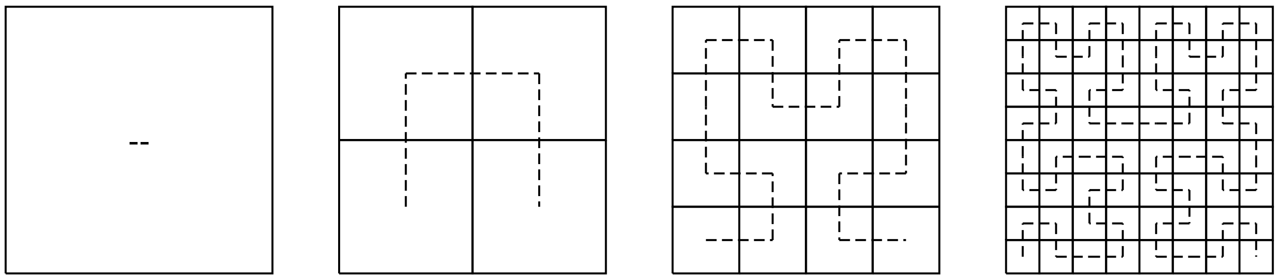

Figure 1.1: Generation of the Hilbert spacefilling curve

1 Introduction

Spacefilling curves, functions that continuously map the unit interval on to a bounded region of higher dimension, have stimulated increasing interest in recent years due to their pretty representations and their use in practical applications [5], [27]. Two of the most important functions required to use a spacefilling curve in practice are: 1. Computing the image of a point in R^1. 2. Computing a pre-image of a point in R^2. In other words, these are operations to converting between positions on a spacefilling curve and coordinates in a higher-dimensional space. We shall refer to these operations, in general, as *coding*.

The Hilbert curve (Figure 1.1) in particular appears to have useful characteristics (see for example: [11], [12], [13], [14], [17]). However, it is widely considered to be tricky to work with. Abel and Mark [1] found the Hilbert ordering to be very promising compared with a number of other orderings in certain application domains, but "the major weakness remains the lack of inexpensive algorithms to evaluate the key of a cell or subquadrant from its coordinates and vice versa." Nulty [23], working with geometric range searching problems, noted that the Hilbert curve has "excellent clustering properties", but found it less convenient than the N-Curve or the Sierpinski curve (pp. 26–27). Laurini and Thompson [18] found the Hilbert and Peano curves to be "more robust, that is better performers over a range of requirements", in spatial information systems. But, in the case of the Hilbert curve, "it is quite complex to get keys." In fact, they provide a coding algorithm expressly "to demonstrate the awkwardness in using Hilbert keys" (pp. 162–168). Samet [28] also reports difficulty working with the Hilbert curve, compared with several other spatial orderings.

Liu and Schrack echo these sentiments in presenting new coding algorithms for two and three dimensional three-dimensional Hilbert curves (they refer to the basic conversion operations as *decoding* and *encoding*). In fact such concerns motivated them to develop a new ordering that is simpler but discontinuous [19], [20], [21].

Algorithms for manipulating the Hilbert curve and other common spacefilling curves are typically based on a geometric analysis of how basic shapes decompose into multiple smaller versions properly transformed and linked together. Nulty [23] formalizes this approach in a generic function *SpaceKey*: 1. Find subcell containing the point of interest. 2. Update key value appropriately. 3. Scale, rotate and reflect subcell as necessary. 4. Continue until sufficiently precise. One finds this sort of analysis throughout the literature; two examples will suffice.

Abelson and diSessa's description of the *turtle graphics* approach to curve drawing is representative [2]. In this interesting system, the *turtle* operates by pointing in a certain direction and moving forward a given distance. The state of the turtle includes its current position and heading. They authors describe the reasoning behind their Hilbert curve drawing: "(W)e must be careful to keep track of how each level of the curve affects the state of the turtle. Assume that the turtle begins facing along the edge of the square that it traverses, and ends in the same direction. Then pasting together four level $n-1$ curves to form the level n curve requires interface turns as shown [in an accompanying figure].... A second tricky point is that there are really two different level $n-1$ Hilbert curves—one that traverses its square to the left and one that traverses to the right—and that the level n curve is made up of two of each parity." .

Sagan's [26] derivations are based on similar analysis. In the case of the Peano curve, the region "is just shrunk uniformly towards the origin to 1/9 its size and translated into positions 1,3,7,9 ..., shrunk and reflected on the y-axis, and translated into positions 2 and 8, shrunk and reflected on the x-axis and translated into positions 4 and 6, and finally, shrunk and reflected on the x and y-axis and translated into position 5." The Hilbert curve requires "in addition to shrinkings, rotations through 90^o of the first and -90^o of the last square before they are reflected and translated into their respective positions" .

Hilbert curve drawing algorithms are generally efficient, recursive procedures [2], [10], [15], [16], [29]. On the other hand, algorithms for point coding, the sorts of procedures needed when actually applying the Hilbert curve, tend to be more complex and ad hoc, generally unrelated to each other or to the drawing routines [6], [7], [8], [9], [11], [17] [18], [26], [27].

This state of affairs means that a promising tool may be underutilized. It also suggests that an important criterion for choosing a spacefilling curve for a practical application is usability. This is not surprising, but it means that certain spacefilling curves may not be given adequate attention, because of perceived difficulty or simply unfamiliarity. Indeed, empirical evidence suggests that researchers do tend to stick with the few well-known curves.

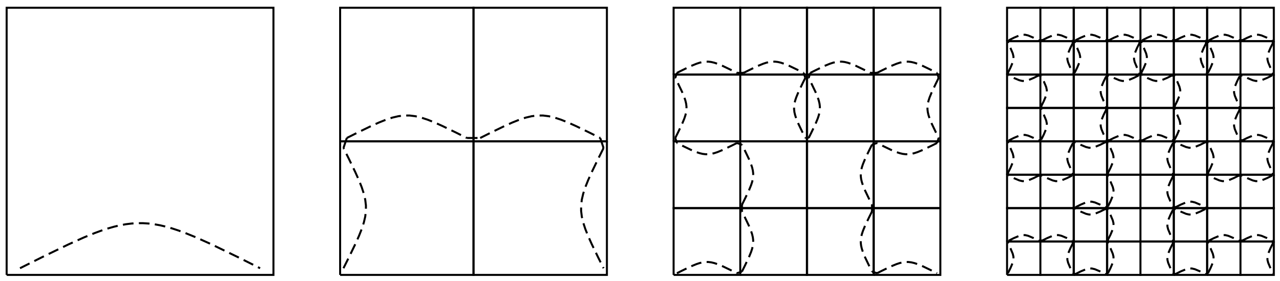

Figure 2.1: The Hilbert spacefilling curve, alternate view

We offer a new approach, based on vertex-labeling, for generating algorithms for the Hilbert curve. This approach leads to simple procedures that involve no complex mathematical or geometric operations. These procedures are also quite flexible. For example they handle regular and irregular geometric figures in the same way. This is important since many common geometric models are composed of irregular figures. Triangulations consisting of scalene triangles (that is, triangulated irregular networks or TINS) are often used to model surfaces. In other work, we discuss the creation of spacefilling curves on irregular triangulations and global data structures based on the hierarchical subdivision of spherical triangles [3], [4].

We showing how our vertex-labeling approach leads to algorithms of similar structure for a number of key operations on spacefilling curves: 1. Drawing a finite approximation of the curve. 2. Computing the image of a point in R^1. 3. Computing a pre-image of a point in R^2. 4. Finding neighbor cells in a decomposition ordered according to a spacefilling curve.

The same approach carries over to a large number of other spacefilling curves. We demonstrate the Sierpinski and Peano curves and variations.

2 The Hilbert Curve

Figure 2.1 shows an alternate way to illustrate the Hilbert curve, in which the line segments of Figure 1.1 have been replaced by arcs. The arc notation conveys more information, since it (broadly) indicates how space is ordered within each cell. Loosely speaking, the arcs are meant to show that the curve enters each cell at a vertex, visits every point in the cell, exits at a different vertex, then enters the next cell and so on.

One observes immediately the self-similar nature of such curves: Each level of the curve is composed

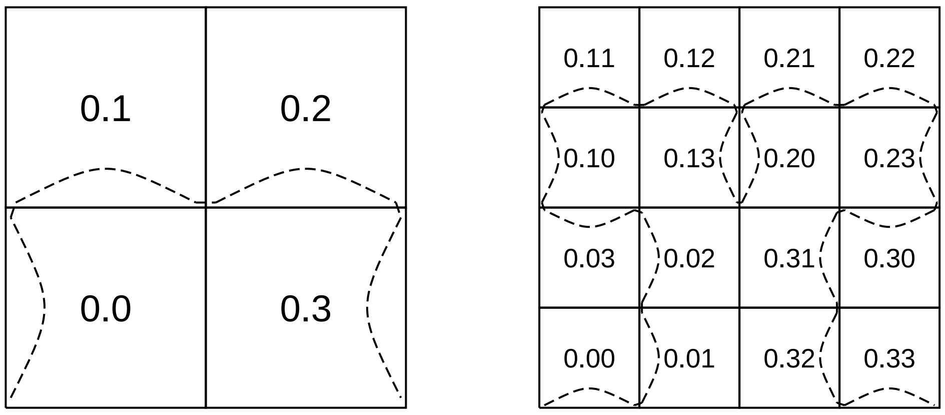

Figure 2.2: Level 1 and 2 Hilbert codes

of multiple, smaller copies of previous levels, oriented in various ways. Components of a curve expand in a predictable way to multiple components at the next level. For example, in the curve illustrated, it is easy to see that each arc expands into four connected arcs. Most spacefilling curve algorithms are based on a geometric analysis of how the shapes expand at each level. Our approach is more topological, more concerned with how the curve relates subcells of a cell to each other.

Each cell at the same level of the decomposition is equivalent in size and shape and has a pre-image in R^1 consisting of an equal-size subinterval of the unit interval. For example, Level 1 of the decomposition has four cells, whose pre-images are: $[0.0, 0.1]$, $[0.1, 0.2]$, $[0.2, 0.3]$, and $[0.3, 1.0]$ (in base 4). We will follow a common convention and assign the lower bound of each interval as the *index value* or *Hilbert code* of the corresponding cell. Thus, the four cells of Level 1 have codes of 0.0, 0.1, 0.2, and 0.3, respectively (Figure 2.2). We refer to codes as index values since they can be sorted to order the cells. Such an ordering is continuous, due to the continuity of spacefilling curves. Cells adjacent in the ordering are adjacent in space; cells near in space *tend* to be near in the ordering.

The curve imposes an order on points located in cells in the following way: A point assumes the Hilbert code of its (smallest) enclosing cell. In other words, points are ordered according to the sequence in which the curve visits their cells. Therefore, if we want to sort a set of points according to the curve, we need to find a decomposition in which each point is located in a different cell, and then the ordering of the points is established by their Hilbert codes.

We convert from a point in R^2, to its Hilbert code (in R^1), at a given precision, by finding a sufficiently small cell enclosing the point. We are indifferent to the exact position of the point within the cell. If we

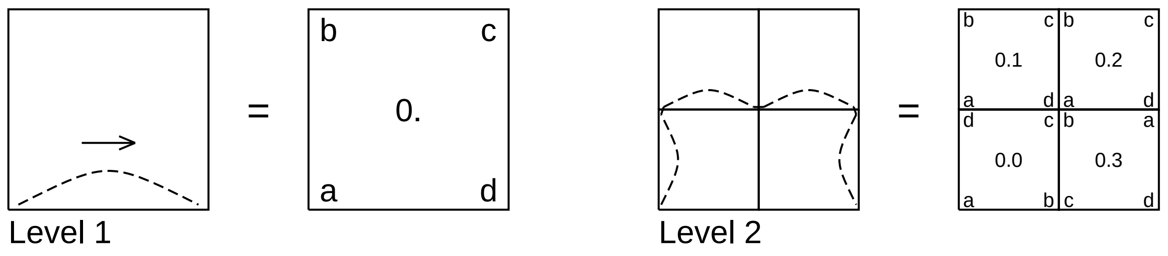

Figure 2.3: Correspondence between vertex labels and arcs

later want to recover the coordinates of a point, we will have to choose some point within the enclosing region. With care, this can be performed multiple times without loss of precision.

2.1 Labeling procedure

The Hilbert curve orders the space within each cell in such a way that the lowest point occurs at one *entry* vertex, and the highest occurs at a different *exit* vertex. The arcs in Figure 2.1 indicate this ordering by beginning at one vertex and ending at another vertex. Each time the cell is divided, the spatial ordering is refined to a greater level of detail. Our approach labels cell vertices in sequence, to indicate the order in which regions of the cell are visited by the curve. The vertex where the curve *enters* the cell is labeled a. The vertex where the curve *exits* the cell is labeled d. The intermediate vertices are labeled, in order, b and c.

The curve notation is a kind of shorthand for the vertex labels. Figure 2.3 shows how the systems correspond. The arc notation is more visually appealing and reveals the spatial ordering at a glance. In addition to its vertex labels, each cell has a base 4 Hilbert code between 0 and 1 indicating its place in the cell ordering. An observation now leads us to the labeling approach to spacefilling curve algorithms: Referring to nothing more than the Hilbert codes and vertex information (labels and coordinates) of a cell, we can generate the Hilbert codes and vertex information of all of its subcells.

The labeling process is summarized in Figure 2.4: At the top level, the Hilbert code is initialized to ‘0.’, and the vertices are labeled. At the next level, the square region is divided into four equivalent square subcells. The Hilbert code of each cell is updated by right-appending a digit as follows: ‘0’ for the cell containing vertex a; ‘1’ for the cell containing vertex b; ‘2’ for the cell containing vertex c; ‘3’ for the cell containing vertex d.

The new vertices are labeled in accordance with the basic Hilbert curve flow pattern. It is very

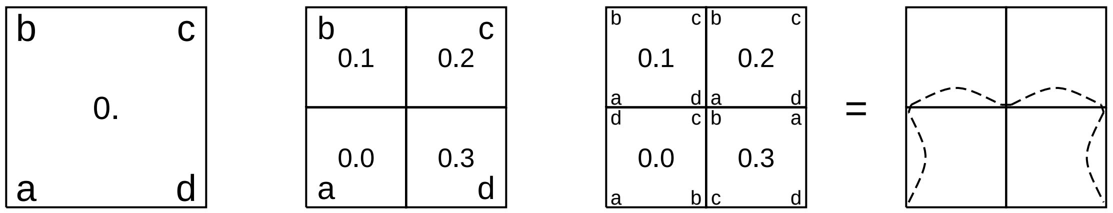

Figure 2.4: Hilbert curve label updating

helpful to refer to the arc diagrams in determining the details of vertex relabeling. As an example, we examine how the vertices in cell 0.0 (containing vertex a) are updated. Recall that an arc enters a cell at vertex a, exits at vertex d, and the remaining vertices are labeled, in sequence, c and d. We refer to the vertices of cell 0.0 as: a_{new}, b_{new}, c_{new}, and d_{new}.

By observation (Figure 2.4), the arc enters cell 0.0 at vertex a, and exits at a vertex on the midpoint of edge (a, b). Following the curve in sequence, vertex b_{new} is located at the midpoint of edge (a, d), and vertex c_{new} is at the midpoint of edge (a, c). This leads to the following update rules for cell 0.0:

$$\begin{aligned} a_{new} &= a \\ b_{new} &= (a+d)/2 \\ c_{new} &= (a+c)/2 \\ d_{new} &= (a+b)/2. \end{aligned}$$

Cells 0.1, 0.2, and 0.3 are updated in a similar way; their relabeling rules can be read directly from the diagram.

Cell 0.1:

$$\begin{aligned} a_{new} &= (a+b)/2 \\ b_{new} &= b \\ c_{new} &= (b+c)/2 \\ d_{new} &= (b+d)/2. \end{aligned}$$

Cell 0.2:

$$\begin{aligned} a_{new} &= (a+c)/2 \\ b_{new} &= (b+c)/2 \end{aligned}$$

$$\begin{aligned} c_{new} &= c \\ d_{new} &= (c+d)/2. \end{aligned}$$

Cell 0.3:

$$\begin{aligned} a_{new} &= (d+c)/2 \\ b_{new} &= (b+d)/2 \\ c_{new} &= (d+a)/2 \\ d_{new} &= d. \end{aligned}$$

2.2 Summary of Hilbert curve labeling procedure

1. *Initialization.*

   We initialize the vertex labels of a square bounding region, so that the entry vertex of an arc is labeled a, the exit vertex is labeled d, and the intermediate vertices are labeled in order b and c. We assign Hilbert code '0.' to the region (Figure 2.4).

2. *Iterative step, I: Finding Hilbert codes of subcells.*

   To create the next level of the decomposition, we divide the square into four subsquares, by adding one edge from the midpoint of edge (a,b) to the midpoint of edge (c,d), and another edge from the midpoint of edge (b,c) to the midpoint of edge (a,d). We first update the codes of each of the four subcells as follows. The subcell containing vertex a is the first *visited* by the curve, so we right-append the digit '0' to the code of its supercell. Following the path of the curve, we right-append the digit '1' for the subcell containing vertex b, '2' for the subcell containing c, and '3' for the subcell containing d.

3. *Iterative step, II: relabeling vertices of subcells.*

   Each arc expands into four arcs, one in each new subcell; we relabel vertices in such a way that each new arc progresses from vertex a to d within its cell, with the intermediate vertices labeled, in order, b and c. Vertices already labeled a, b, c and d do not change. The thirteen new vertices are labeled as detailed above (and in the figure).

4. *Decision step.*

   Continue the iteration steps until sufficient precision is reached.

Therefore, given initial vertex information and index value of a square region, creating iterates of a decomposition according to the Hilbert curve is a purely mechanical process. It can proceed with no additional information, and no need to further analyze the geometry. By recursively applying this procedure, we may label cells of the decomposition to any depth.

2.3 Computing a pre-image of a point in R^2

We wish to convert from a point in a square bounding region to its corresponding Hilbert code.

A point located in a square region takes the Hilbert code of its smallest enclosing cell. A point can be coded to a desired precision by enclosing it in a small enough region, and we can find an arbitrarily small enclosing square by preceding to the necessary depth in the hierarchical decomposition. The result is a labeled cell around the point, and a base 4 code on the interval $[0, 1]$. In determining the code, we need only label one subsquare at each level; that is, we just traverse the sequence of nested subcells in which the point falls. Therefore the amount of work required is proportional to the precision of the result. Summarizing the procedure to find a Hilbert code of a point (refer to Figure 2.4):

1. *Initialization.*
   Given a point P in a square bounding region, label the four vertices of the region a, b, c and d (in sequence), and initialize the Hilbert code of P to '0.'.

2. *Iteration.*
   Divide the current enclosing square (a, b, c, d) into four equivalent subsquares.
   If P is nearest vertex a, then update its Hilbert code by right-appending the digit '0', and update vertex labels as follows:

$$\begin{aligned} a_{new} &= a \\ b_{new} &= (a+d)/2 \\ c_{new} &= (a+c)/2 \\ d_{new} &= (a+b)/2. \end{aligned}$$

   If P is nearest vertex b, right-append '1' to its code, and update labels as follows:

$$\begin{aligned} a_{new} &= (a+b)/2 \\ b_{new} &= b \\ c_{new} &= (b+c)/2 \end{aligned}$$

$$d_{new} = (b+d)/2.$$

If P is nearest vertex c, right-append '2' to its code, and update labels as follows:

$$\begin{aligned} a_{new} &= (a+c)/2 \\ b_{new} &= (b+c)/2 \\ c_{new} &= c \\ d_{new} &= (c+d)/2. \end{aligned}$$

If P is nearest vertex d, right-append '3' to its code, and update labels as follows:

$$\begin{aligned} a_{new} &= (d+c)/2 \\ b_{new} &= (b+d)/2 \\ c_{new} &= (d+a)/2 \\ d_{new} &= d. \end{aligned}$$

3. *Decision.*

   Repeat iteration section until Hilbert code is sufficiently precise.

Point P is assigned the code of its smallest enclosing cell. It is *sufficiently precise* in the sense that we are indifferent to the exact position of P within the cell. Figure 2.5 shows an example find a code to Level 3.

2.4 Computing the image of a point in R^1

We wish to convert from a Hilbert code on the unit interval, to its corresponding point on the square bounding region. The procedure to find the image of a point in R^1 is a straight reversal of the procedure to find the pre-image of a point in R^2 (Section 2.3).

We begin with a Hilbert code, consisting of a string of base 4 digits. At each iteration, instead of right-adding a digit to the code, we left-remove a digit (the digit specifying the next subcell). We continue in this way, narrowing down to a smaller and smaller enclosing cell, until no more digits remain. We are left with a square cell (a, b, c, d), and we choose our point from this region according to some rule, such as: *Take the average of the vertices.* We label just one subsquare at each level of the subdivision. Therefore, the amount of work required is proportional to the precision of the result (that is, the length of the Hilbert code). We briefly summarize the procedure:

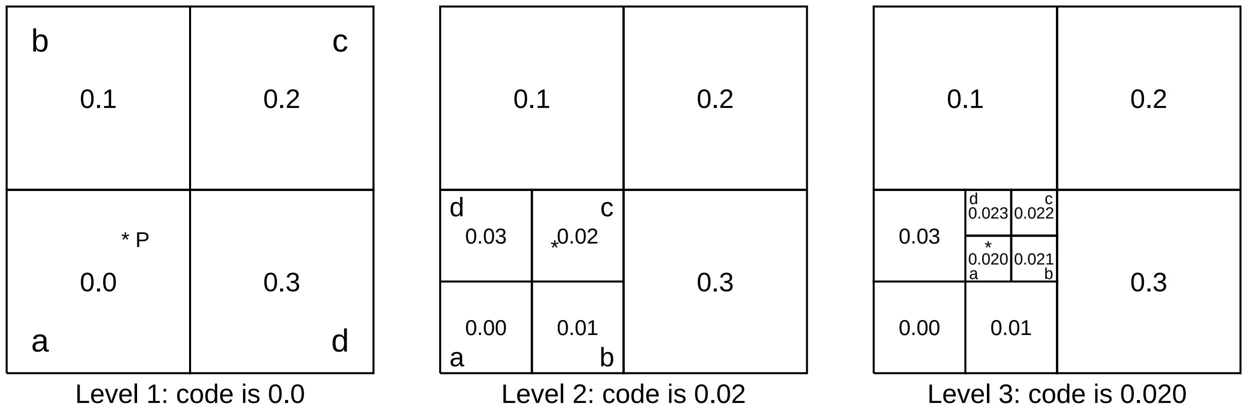

Figure 2.5: Hilbert curve: Point P is coded to Level 3

1. *Initialization.*

   Given a Hilbert code and a square bounding region, label the vertices a, b, c and d.

2. *Iteration.*

   Divide the current square cell into four equivalent subsquares, as in the procedure to find the code of a point. Remove the first digit r to the right of the decimal point from the Hilbert code. If $r = 0$ then the subcell containing vertex a becomes the current cell. If $r = 1$ then the subcell containing vertex b becomes the current cell. If $r = 2$ then the subcell containing vertex c becomes the current cell. Otherwise, $r = 3$ and the subcell containing vertex d becomes the current cell. Relabel as in the labeling and point coding procedures.

3. *Decision.*

   If no digits remain, then we are done. Otherwise, continue with the *Iteration* step.

Point P is assigned coordinates based on the final enclosing cell.

2.5 Curve drawing

The labeling method leads directly to a curve drawing procedure. This procedure is similar to the procedures for computing images and pre-images of points. In coding points, we search for particular cells; in drawing curves, we traverse, in a sequence governed by the spacefilling curve, all cells at a particular level of the decomposition. We present a general, recursive curve-drawing procedure based on the labeling method:

1. *Initialization.*

   Specify the number of levels n, and the positions of the vertices of the bounding region. We begin at the top level of the decomposition. Initially set the current cell equal to the bounding region.

2. *Recursive step.*

   If we are at the Level n of the decomposition, draw the spacefilling curve within the current cell, and end the current recursive step.

   Otherwise, subdivide the current cell, and for each subcell (taken in order by index value): relabel its vertices, and send it to the recursive step at the next deeper level.

If each cell divides (recursively) into c subcells, then the amount of work required is proportional to c^n.

2.6 Neighbor-finding

The labeling method leads directly to a neighbor-finding procedure. This procedure has a structure similar to the procedures for finding images and pre-images of points. In coding a point, we search for the cell containing that point. In searching for the neighbors of a cell, we begin by identifying points on the interior of the edges of the cell. Then we search for all cells containing any of those points, a straightforward recursive task. This is a general neighbor-finding procedure based on the labeling method:

1. *Initialization.*

   Specify the positions of vertices of the bounding region, and of vertices of the cell of interest. Specify one point on the interior of each edge of the cell of interest. Initially set the current cell equal to the bounding region.

2. *Recursive step.*

   If we are at the deepest level, then return the current cell (unless it is the same as the cell of interest), and end the current recursive step.

   Otherwise, subdivide the current cell, and for each subcell containing one of the edge-interior points: relabel its vertices, and send it to the recursive step at the next deeper level.

Given a cell with e edges, this procedure traverses at most $2e$ paths through the decomposition. For a decomposition with n levels, the amount of work required to traverse each path is $O(n)$, just as in coding procedures. Therefore, the work to find the neighbors is also $O(n)$.

This procedure finds all edge-adjacent neighbors of a region. To find all vertex-adjacent neighbors, instead of searching for cells containing points on the interior of the edges of the cell of interest, search instead using the vertices of the cell.

2.7 Moore's Hilbert curve and other variations

Procedures for a number of Hilbert curve variations can be created as straightforward modifications of standard Hilbert curve procedures based on the labeling method. As an example, we briefly describe a labeling procedure for Moore's version of Hilbert's spacefilling curve (this variation is discussed in Sagan [27]).

The first iterates of Moore's variation are shown in Figure 2.6, in both arc and vertex label forms. In this variation, each subsquare of Level 1 is filled with a little Hilbert curve, but their initial orientations are different, so that, unlike the standard Hilbert curve, Moore's variation describes a *circuit* in the square.

The only change required in our labeling procedure for this curve is in the initialization section. This section is now responsible for labeling the first level of subcells, and generating the first digit of the code. The labeling method is identical in the iteration section, since the curve follows the standard Hilbert curve within each cell.

Figure 2.7 shows other Hilbert-type spacefilling curves, based on a 2x2 grid. (We omit cases which are reflections or rotations of other cases.) These curves are related to the Hilbert curve since the curve follows the Hilbert ordering within each subcell.

2.8 A three-dimensional Hilbert curve

Sagan [27] describes the first two levels of a three-dimensional Hilbert curve. We modify his labeling diagram so that cell *vertices* are numbered, rather than cell *centerpoints*. Figure 2.8 and Figure 2.9 show how Level 1 and Level 2 vertices are ordered by the three-dimensional Hilbert curve.

As in the two-dimensional case, the curve orders the space within each cell beginning at one vertex, and ending at another vertex. In the figures, we use vertex numbering rather than arcs to show how space is ordered; arcs would be too cumbersome and confusing. In the three-dimensional case, each cell is a cube, rather than a square, and we label eight vertices, rather than two. (The number of vertices labeled can be reduced; we label all eight for simplicity.) At each new level, each cell subdivides into eight subcells (or *octants*), each of which is a cube.

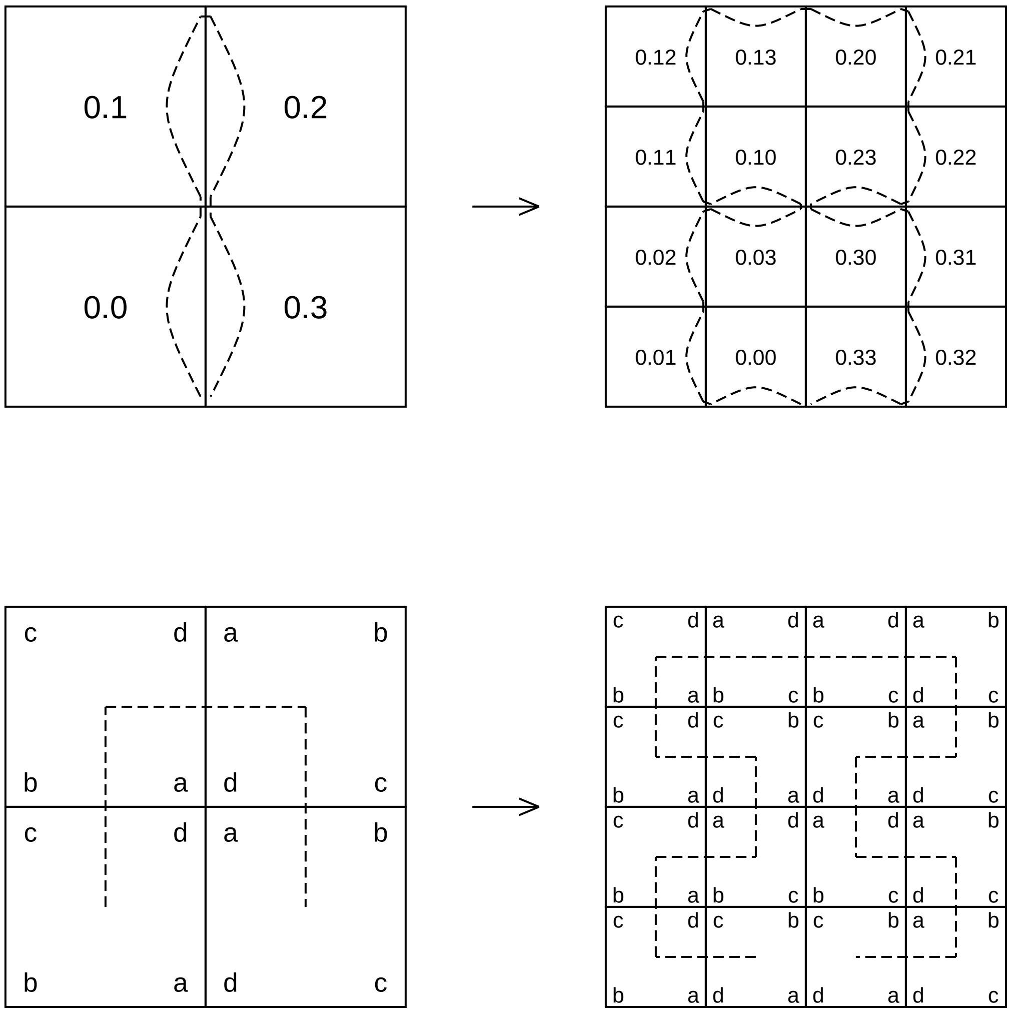

Figure 2.6: Moore's variation of the Hilbert curve

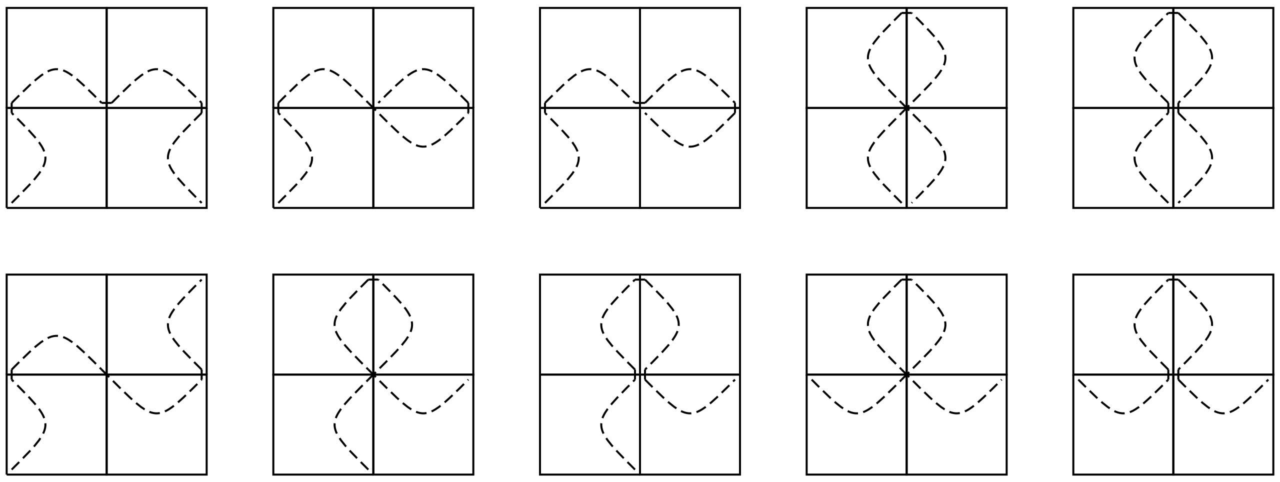

Figure 2.7: Hilbert curve variations

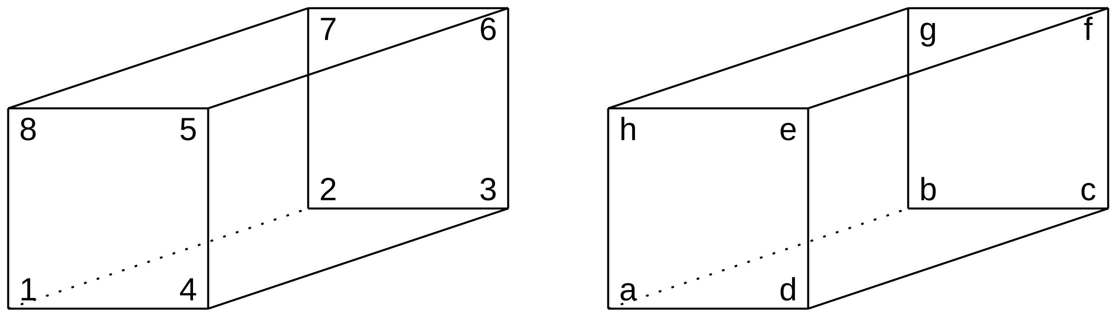

Figure 2.8: Three-dimensional Hilbert curve, Level 1

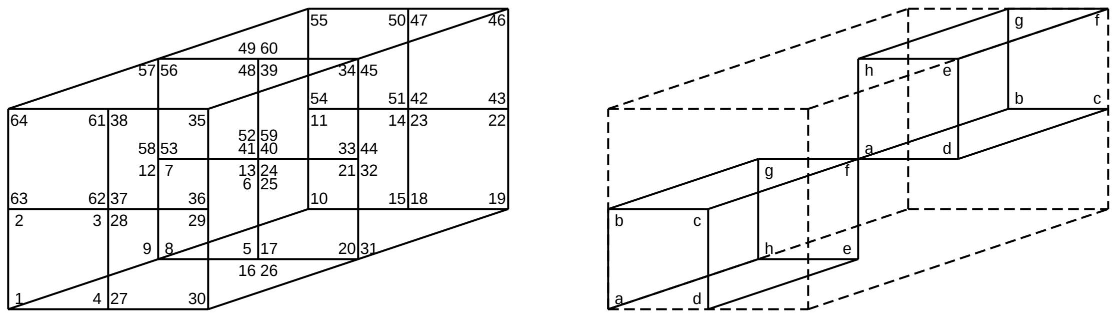

Figure 2.9: Three-dimensional Hilbert curve, Level 2, Octants 1 and 6

Figure 2.8 shows how the eight vertices of Level 1 are labeled. Figure 2.9 shows Level 2 ordering, and in particular how Octants 1 and 6 of Level 2 are labeled. The vertex labels can be read directly from the figure by following the vertex orderings shown. In Octant 1, vertex labels a-g follow vertices 1-8, in order. In Octant 6, vertex labels a-g follow vertices 41-48, in order.

Likewise, the relabeling instructions of the iteration step can be read directly from the figure. For example, we label the front lower left octant (Octant 1, vertices 1-8) as follows:

$$\begin{aligned}
a_{new} &= a \\
b_{new} &= (a+h)/2 \\
c_{new} &= (a+e)/2 \\
d_{new} &= (a+d)/2 \\
e_{new} &= (a+c)/2 \\
f_{new} &= (a+f)/2 \\
g_{new} &= (a+g)/2 \\
h_{new} &= (a+b)/2.
\end{aligned}$$

We label the rear upper right octant (Octant 6, vertices 41-48) as follows:

$$\begin{aligned}
a_{new} &= (g+d)/2 \\
b_{new} &= (g+c)/2 \\
c_{new} &= (f+c)/2 \\
d_{new} &= (f+d)/2 \\
e_{new} &= (f+e)/2 \\
f_{new} &= f \\
g_{new} &= g \\
h_{new} &= (g+h)/2.
\end{aligned}$$

This example involved the subdivision of a cube, but as this algorithm is topologically based, the same approach works for decompositions of other hexahedrons dividing in a similar way into eight smaller hexahedrons. Of course, calculating the positions of the new vertices (as a function of the old vertices) would be more complicated.

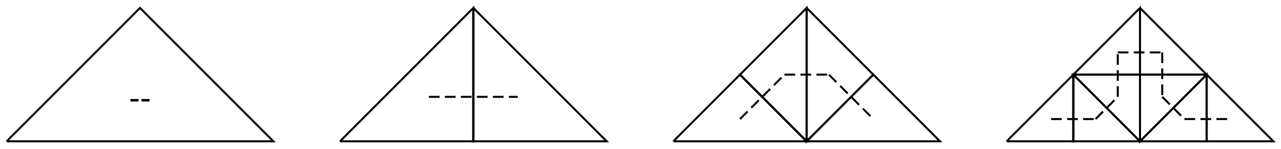

Figure 3.1: Generation of the Sierpinski spacefilling curve

2.9 Running time efficiency

When converting between the coordinates of a point in R^2, and an index value in R^1, procedures based on the labeling method need only consider the sequence of cells containing the point. Therefore, the work involved in these operations is $O(n)$, where n is the number of digits of accuracy (equivalently, the number of levels in the decomposition). Neighbor-finding is also $O(n)$, since it involves a constant number of conversion operations. Drawing procedures based on the labeling approach are $O(c^n)$, where c is the number of subcells within a subdivided cell. This is similar to the most efficient of the published coding and drawing algorithms. Almost all of the published coding algorithms we have discussed require $O(n)$ work. One exception is the $O(n^2)$ $R^1 \rightarrow R^2$ Hilbert ("Pi order") algorithm presented in [11] and [18].

Coding algorithms can also be used to draw curves, by using them to generate a sequence of coordinates in order. Sagan's drawing code is of this form. This is less efficient than the specialized routines, since calculating each point itself requires, in most cases, $O(n)$ work. Nevertheless, such solutions may be useful. Fisher [9] makes this point in describing how his essentially coding approach leads to a somewhat flexible procedure for Hilbert curve drawing.

3 The Sierpinski Curve

Figure 3.1 shows the first levels of a curve, based on the recursive subdivision of an isosceles triangle, whose limit is the Sierpinski spacefilling curve, The path is drawn to indicate how the curve orders cells at each level. Figure 3.2 shows an alternate depiction of the Sierpinski curve, in which the line segments have been replaced by arcs. As with the Hilbert curve (Section 2), the arc notation broadly indicates how space is ordered within each cell, and is useful in developing the vertex-labeling approach.

The Sierpinski spacefilling curve is considered fairly easy to work with, due in large part to its high degree of symmetry. For drawing algorithms, see: [2], [10], [15], [16], [22], [29], [30]. Most of these are similar, well-constructed recursive procedures. These algorithms depend on curve regularity. For

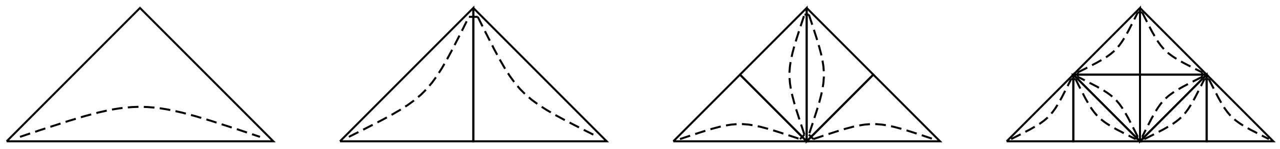

Figure 3.2: Sierpinski curve, alternate view based on the same decomposition

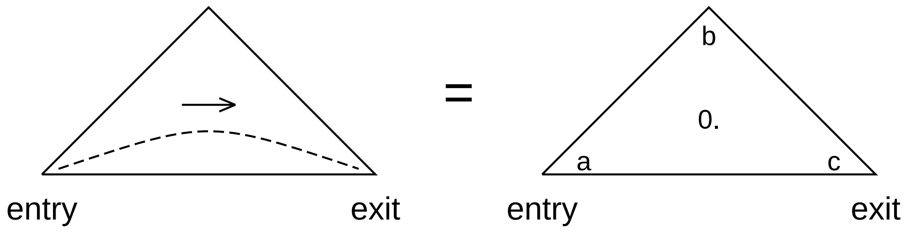

Figure 3.3: Correspondence between vertex labels and arcs

example, angles in the figures are all multiples of 45^o, and all line segments are one of the two lengths used in drawing a Sierpinski curve iterate. For coding algorithms, see: [5], [24], [27].

We present a labeling procedure for the Sierpinski code. Other procedures—coding, neighbor-finding, drawing—are derived from this, as with the Hilbert curve (Section 2).

3.1 Labeling procedure

The Sierpinski curve orders the space within a triangular cell in such a way that the lowest point occurs at an *entry* vertex, and the highest occurs at a different *exit* vertex. We indicate the ordering of space within a triangular cell by labeling its vertices in sequence. The vertex where the curve enters the cell is labeled a; the exit vertex is labeled c; the intermediate vertex is labeled b. Figure 3.3 shows how the arc notation and the vertex labels correspond.

1. *Initialization.*

   We initialize the vertex labels of a triangular bounding region as in Figure 3.4 (left), so that the entry vertex of an arc is labeled a, the exit vertex is labeled c, and the intermediate vertex is labeled b. We assign a Sierpinski code of '0.' to the region.

2. *Iterative step, I: Finding Sierpinski codes of subcells.*

   To create the next level of the decomposition, we bisect the triangle, by adding an edge from vertex b

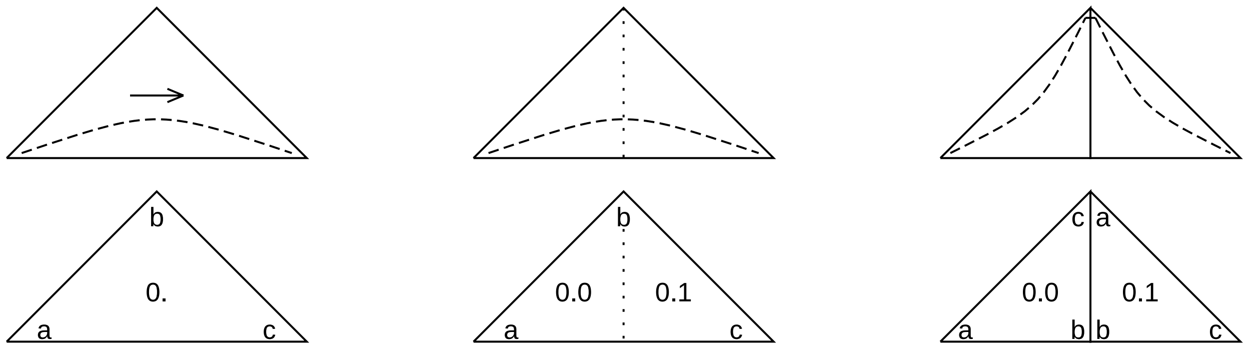

Figure 3.4: Label updating

to the midpoint of the opposite edge (a, c), creating two new isosceles triangles (Figure 3.4, middle). We first update the codes of each of the two subcells. The subcell containing vertex a is the first *visited* by the curve, so we right-append the digit '0' to the code of its supercell; for the other subcell, containing vertex b, we right-append the digit '1'.

3. *Iterative step, II: Relabeling vertices of subcells.*

   Each arc expands into two arcs, one in each new subcell;. We relabel vertices in such a way that each new arc progresses from vertex a to c within its cell, with the intermediate vertex labeled b (Figure 3.4, right): Vertices already labeled a and c do not change. The two new unlabeled vertices, located at the midpoint of edge (a, c), are both labeled b. The vertex originally labeled b is relabeled c in the subtriangle containing a, and a in the other subtriangle.

4. *Decision step.*

   Continue with the iteration steps until sufficient precision is reached.

Figure 3.5 shows a Sierpinski curve variation based on a scalene triangle, rather than the usual isosceles. This sort of variation, as well as variations in orientation and scale, are handled by the labeling method without modification. Many other variations are possible requiring no modification, or straightforward modification.

4 The Peano Curve

To round out our examples of well-known spacefilling curves, we describe a labeling procedure for the Peano curve (Figure 4.1). The standard Peano curve is relatively easy to generate from Peano's mapping

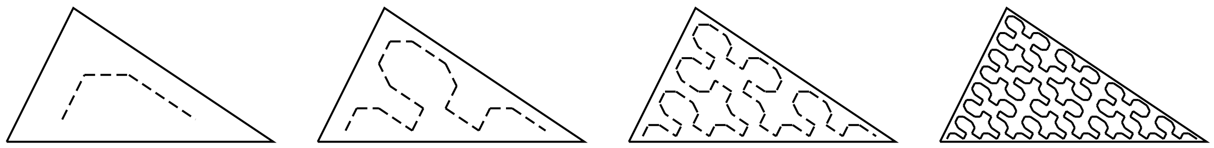
Figure 3.5: Sierpinski curve within a scalene triangle

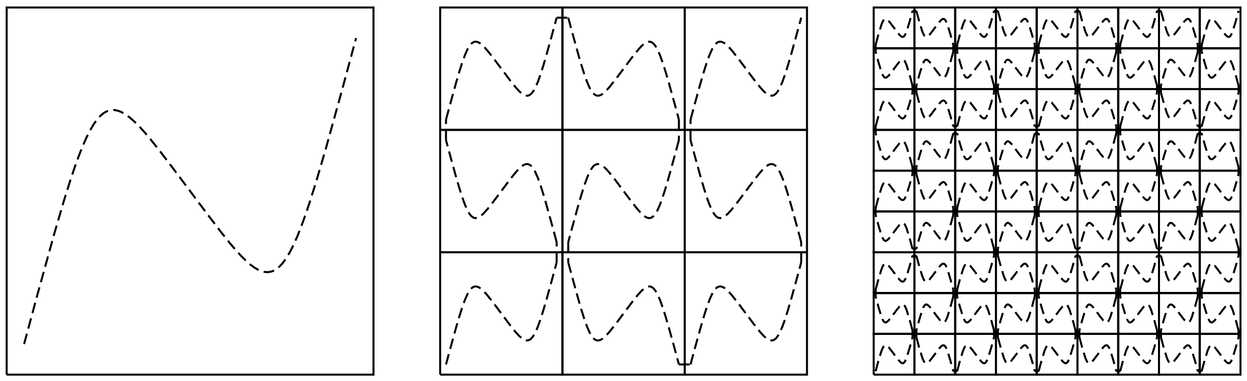
Figure 4.1: The Peano curve

function [25], [27]. Drawing algorithms for the Peano curve, or close variations, are found in: [2], Griffiths [16], [22].

4.1 Labeling procedure

Figure 4.2 shows vertex labels corresponding to the arc notation. The curve notation shows the spatial ordering between cells and within cells. Within each cell, the arc indicates that the spacefilling curve enters the cell at one vertex (labeled a), then moves broadly towards another vertex (b), then diagonally towards a third vertex (c), finally exiting at a fourth vertex (d). We briefly describe a procedure to find the pre-image of a point in R^2 (Figure 4.3):

1. *Initialization.* Given a point P in a square bounding region, we wish to find its Peano code. Label the four vertices a, b, c and d. Initialize the Peano code to '0.'.

2. *Iteration.*

   Subdivide the cell into nine subsquares. Locate point P in one of the subsquares. Update the code, and label vertices of the subsquares as follows:

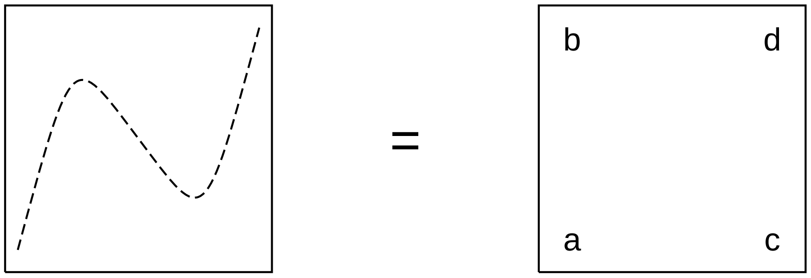

Figure 4.2: Peano curve - Correspondence between vertex labels and arcs

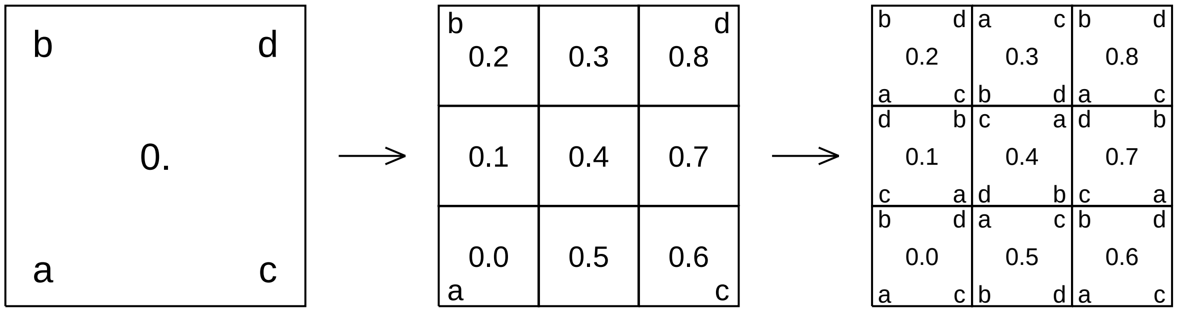

Figure 4.3: Peano curve - Initializing and updating

If P is in subsquare '0', right-append '0' to the Peano code, and update labels:

$$\begin{aligned} a_{new} &= a \\ b_{new} &= (2/3)a + (1/3)b \\ c_{new} &= (1/3)c + (2/3)a \\ d_{new} &= (1/3)d + (2/3)a. \end{aligned}$$

If P is in subsquare '1', right-append '1' to the Peano code, and update labels:

$$\begin{aligned} a_{new} &= (1/3)d + (2/3)a \\ b_{new} &= (2/3)b + (1/3)c \\ c_{new} &= (2/3)a + (1/3)b \\ d_{new} &= (1/3)a + (2/3)b. \end{aligned}$$

The remaining subsquares (2-8) follow a similar pattern.

3. *Decision.*

   Repeat until sufficient precision is reached.

Procedures for the Peano curve based on the labeling method have the same complexity as those for the Sierpinski curve (Section 3) and the Hilbert curve (Section 2), but the larger initial grid means that there is more detail. We based this procedure on four labeled vertices (a, b, c, and d), but other labeling systems are possible.

5 Discussion

The labeling approach to creating spacefilling curve algorithms is based on a recursive traversal of a hierarchical decomposition. Each cell at a given level of the decomposition maintains a local frame of reference, consisting of cell label and code information, and implicitly the spatial relationships of subcells. Given a cell with its local frame of reference, a simple set of labeling rules determines the labels and codes of each of its subcells. These rules can be determined by inspection from diagrams showing how the curve orders cells at successive levels of the decomposition. This approach allows us to avoid having to think in terms of geometric operations per se. In addition to conceptual simplification, this approach is flexible with respect to the variety of operations and the curve variations that can be supported.

While published drawing routines have stabilized to the point that a number of different curves can be drawn with similarly structured algorithms, the coding algorithms tend to be very different for different curves. Further, drawing and coding algorithms are generally quite different—although coding algorithms can be used as the basis for (inefficient) drawing procedures. Finally, both drawing and coding algorithms tend to focus on the few well-known curves.

Most of the published curve drawing algorithms are efficient but rigid, limited to drawing specific regular curves based on regular cells such as squares and right isosceles triangles (in the case of the Hilbert and Sierpinski curves, respectively), and with bounding regions of fixed (or restricted) size. Quantities such as angles and segment lengths are fixed. For the Hilbert curve and Peano curves, for example, all segment lengths are the same, and all angles are 90 degrees. For the Sierpinski curve, all angles are multiples of 45 degrees, and two fixed segment lengths are used. This simplifies the algorithms, but means that they are unsuited for irregular decompositions, since irregularly shaped cells require the ability to handle odd angle sizes and edge lengths. Likewise, the published procedures for computing images and pre-images of points (in R^1 and R^2 respectively) generally depend for their brevity on regularity and symmetry. They provide solutions for specific problems, with fixed initial position, scale, and orientation.

The labeling method handles a large set of variations with little or no extra work. The method makes no assumptions regarding location or scale of the bounding region, nor about the regularity of cell shape; it simply takes as input the vertices of the bounding region. For example, a Sierpinski-like curve based on scalene triangles poses no special problems. Similarly, asymmetric curves, and certain variations in cell orderings, are in principle no more complex than symmetric or familiar curves, and are handled no differently (calculations involved in computing points within irregular figures may of course be more involved).

Our labeling approach leads to very similar algorithms for a number of different operations on spacefilling curves. We have discussed converting between points in R^1 and points in higher dimensional space, as well as curve drawing and neighbor-finding. These operations are all handled with variations on the same basic algorithmic structure. Such functions have typically been implemented with quite different algorithms. In addition, the same basic approach can be applied to many different curves. The structure is the same in each case; the details differ.

Not every recursive spacefilling curve generating process will support the labeling method. The spacefilling curve generating processes we have discussed have two important properties required by the labeling method. First, cells are well-ordered in the sense that the curve enters a cell exactly once,

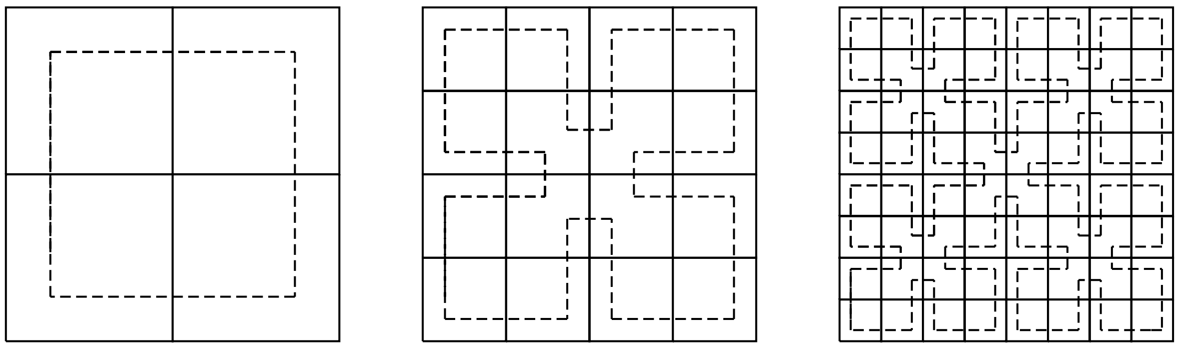

Figure 5.1: Sierpinski curve, alternate decomposition

traverses it and all of its subcells (if any), then leaves the cell, never to return. If this were not the case, and the curve entered a cell more than once, it would be hard to say where the cell appeared in the ordering. This is somewhat similar to Abel and Mark's concept of *quadrant-recursion* for quadtree structures [1]. Second, the generating processes are order-consistent: regions (or points) never reverse their ordering at different levels of the decomposition. In other words, if cell i precedes cell j, then all of i's subcells precede all of j's subcells. Two points located within the same cell at a given level have the same ordinality, that is to say, they are not ordered with respect to each other at that level. But if two points are in distinct cells at some level, then they are ordered with respect to each other, and will have the same relative ordering at all subsequent levels.

Figure 5.1 shows successive approximations to the Sierpinski spacefilling curve based on the recursive decomposition of square cells into equivalent subsquares (the figure is based on one in Bartholdi and Platzman [5]). The cells are not well-ordered since, starting with Level 2, certain cells appear more than once in the ordering. Therefore, this subdivision cannot support a labeling-based procedure.

6 Conclusions

We have described a vertex-labeling approach to creating algorithms for manipulating the Hilbert spacefilling curve. The labeling method manages to avoid explicit geometric analysis of spacefilling curve shapes. This leads to an intuitive and robust approach, with certain advantages over hitherto published procedures. In particular, the method requires little or no special handling for a number of variations in the two-dimensional (or higher) bounding region, including those related to scale, orientation, location, and shape regularity. Asymmetric curves are in principle no more difficult than symmetric curves.

This approach leads to similar algorithms for a number of common operations. We have demonstrated: 1. Computing the image of points in R^1. 2. Computing the pre-image of points in R^2. 3. Drawing representations of spacefilling curves. 4. Finding neighbors of cells in a decomposition, with respect to a spacefilling curve. The algorithms are short and efficient. Conversion and neighbor-finding functions are $O(n)$, where n is the number of digits of accuracy; Drawing procedures are $O(c^n)$, where c is the number of subcells within a subdivided cell. When the vertices are labeled, procedures for various functions can be derived from the figure basically by inspection. We demonstrated with the two-dimensional Hilbert curve and variations. We then showed that the method also leads directly to procedures for the three-dimensional case.

Our vertex-labeling approach can be applied to other spacefilling curves. We have shown additional examples involving the Sierpinski and Peano curves. The accessibility to a simple and general approach to creating spacefilling curve algorithms may encourage researchers to experiment with a greater variety of curves. This is useful as different curves have different characteristics, and may be more or less useful in particular applications.

References

[1] David J. Abel and David M. Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Systems*, 4(1):21–31, 1990.

[2] Harold Abelson and Andrea A. diSessa. *Turtle Geometry, the Computer as a Medium for Exploring Mathematics.* The MIT Press, Cambridge, MA, 1981.

[3] John J. Bartholdi, III and Paul Goldsman. Continuous indexing of hierarchical subdivisions of the globe. 1999. To be submitted.

[4] John J. Bartholdi, III and Paul Goldsman. A continuous spatial index of a triangulated surface. 1999. To be submitted.

[5] John J. Bartholdi, III and Loren K. Platzman. Heuristics based on spacefilling curves for combinatorial problems in Euclidean space. *Management Science*, 34(3):291–305, 1988.

[6] Theodore Bially. Space-filling curves: their generation and their application to bandwidth reduction. *IEEE Transactions on Information Theory*, it-15(6):658–664, Nov 1969.

[7] Arthur R. Butz. Convergence with Hilbert's space filling curve. *J. of Computer and System Sciences*, 3:128–146, 1969.

[8] Arthur R. Butz. Alternative algorithm for Hilbert's space-filling curve. *IEEE Transactions on Computers*, pages 424–426, Apr 1971.

[9] A. J. Fisher. A new algorithm for generating Hilbert curves. *Software—Practice and Experience*, 16(1):5–12, Jan 1986.

[10] Leslie M. Goldschlager. Short algorithms for space-filling curves. *Software—Practice and Experience*, 11:99, 1981.

[11] Michael F. Goodchild and Andrew W. Grandfield. Optimizing raster storage: An examination of four alternatives. In *Proc. Auto Carto 6*, volume 2, pages 400–407, Ottawa, 1983.

[12] C. Gotsman and M. Lindenbaum. On the metric properties of discrete space-filling curves. In *Proc Intl Conf on Pattern Recognition*, volume 3, pages 98–102, Piscataway, NJ, 1994. IEEE.

[13] C. Gotsman and M. Lindenbaum. Euclidean Voronoi labeling on the multidimensional grid. *Pattern Recognition Letters*, 16:409–415, 1995.

[14] C. Gotsman and M. Lindenbaum. On the metric properties of discrete space-filling curves. *IEEE Transactions on Image Processing*, 5(5):794–797, May 1996.

[15] J. G. Griffiths. Table-driven algorithms for generating space-filling curves. *Computer-Aided Design*, 17(1):37–41, Jan/Feb 1985.

[16] J. G. Griffiths. An algorithm for displaying a class of space-filling curves. *Software—Practice and Experience*, 16(5):403–411, May 1986.

[17] H. V. Jagadish. Linear clustering of objects with multiple attributes. *SIGMOD Record (ACM)*, 19(2):332–342, 1990.

[18] Robert Laurini and Derek Thompson. *Fundamentals of Spatial Information Systems*. Academic Press, Ltd., San Diego, CA, 1992.

[19] Xian Liu and Günther Schrack. Encoding and decoding the Hilbert order. *Software—Practice and Experience*, 26(12):1335–1346, Dec 1996.

[20] Xian Liu and Günther Schrack. An algorithm for encoding and decoding the 3-D Hilbert order. *IEEE Transactions on Image Processing*, 6(9):1333–1337, Sep 1997.

[21] Xian Liu and Günther Schrack. A new ordering strategy applied to spatial data processing. *International Journal of Geographical Information Science*, 12(1):3–22, 1998.

[22] Null. Space-filling curves, or How to waste time with a plotter. *Software—Practice and Experience*, 1:403–410, 1971.

[23] William Glenn Nulty. *Geometric Searching with Spacefilling Curves.* PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1993.

[24] William Glenn Nulty and John J. Bartholdi, III. Robust spatial searching with spacefilling curves. In Thomas C. Waugh and Richard G. Healey, editors, *Advances in GIS Research, Proceedings of the 6th International Symposium on Spatial Data Handling, Sept. 5th–9th 1994*, Edinburgh, Scotland, UK, 1994.

[25] G. Peano. Sur une courbe qui remplit toute en aire plaine. *Math. Ann.*, 36, 1890.

[26] Hans Sagan. On the geometrization of the Peano curve and the arithmetization of the Hilbert curve. *Int. J. Math. Educ. Sci. Tech.*, 23(3):403–411, 1992.

[27] Hans Sagan. *Space-Filling Curves.* Springer-Verlag, New York, 1994.

[28] Hanan Samet. Spatial databases, tutorial. In *SSD'95, Aug. 6. 1995*, Portland, Maine, USA, 1995.

[29] Niklaus Wirth. *Algorithms + Data Structures = Programs.* Prentice–Hall, Englewood Cliffs, NJ, 1976.

[30] Ian H. Witten and Brian Wyvill. On the generation and use of space-filling curves. *Software—Practice and Experience*, 13:519–525, 1983.