# A recursive algorithm for the generation of space-filling curves

**Article** · January 1998

**2 authors:**

Greg Breinholt
6 PUBLICATIONS   103 CITATIONS

SEE PROFILE

Christoph Schierz
Technische Universität Ilmenau
60 PUBLICATIONS   816 CITATIONS

SEE PROFILE

# A Recursive Algorithm for the Generation of Space-Filling Curves

*Greg Breinholt and Christoph Schierz*

Institute for Hygiene and Applied Physiology

Swiss Federal Institute of Technology Zürich (ETHZ)

CH-8092 Zürich, Switzerland

email: breinholt@computer.org schierz@iha.bepr.ethz.ch

## ABSTRACT

Space-filling curves have intrigued both artists and mathematicians for a long time. They bridge the gap between aesthetic forms and mathemtical geometry. To enable construction by computer, an efficient recursive algorithm for the generation of space-filling curves is given. The algorithm is elegant, short and considerably easier to implement than previous recursive and non-recursive algorithms, and can be efficiently coded in all programming languages that have integer operations. The algorithmic technique is shown applied to the generation of the Hilbert and a form of the meandering Peano curve. This coding technique could be successfully applied to the generation of other regular space-filling curves.
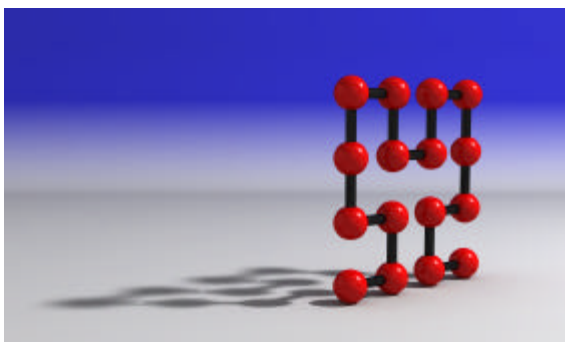
## 1. INTRODUCTION

In mathematics, space-filling curves are commonly used to reduce a multi-dimensional problem to a one-dimensional problem; the curve is essentially a linear transversal of the discrete multi-dimensional space. Although it was Peano [10] that produced the first space-filling curves, it was Hilbert [5] who first popularized their existence and gave an insight into their generation. Hilbert and Peano curves belong to the class of *FASS* curves; an acronym for space-filling, self-avoiding, simple and self-similar [11]. FASS curves can be thought of as finite, self-avoiding approximations of curves that pass through all points of a square. The Hilbert curve is a particular form of the FASS curve that scans a $2^m \times 2^m$ array of points while never maintaining the same direction for more than three consecutive points. The Peano curve scans a $3^m \times 3^m$ array of points while never traveling in the same direction for more than five consecutive points. Figure 1 shows an example of a Hilbert curve, and a meandering Peano curve.
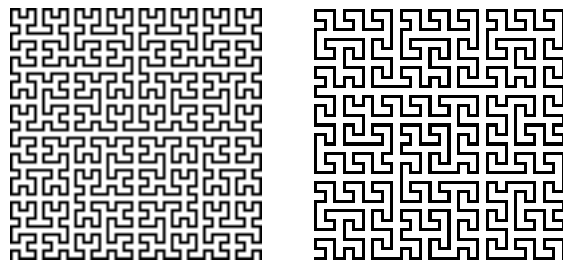


**Figure 1. Hilbert curve (32 × 32 points) and meandering Peano curve (27 × 27 points).**

Though space-filling curves were discovered over a century ago their use has been sporadic but

varied. Interesting examples of their use have been in data structures [1], traveling salesman problems [9], image analysis [6], digital halftoning [12], pattern and texture analysis [7], cryptology [2] and data compression [3].

Most of the techniques for curve generation are interpretations of Hilbert's original suggestion to continually divide a plane into four parts, each of these parts into four parts, and so on, calculating the necessary plotting points as the division proceeds. This continual dividing of the plane continues until the required curve resolution is obtained, i.e. in a typical recursive procedure. The exact method to determine the points was not given by Hilbert (who worked before the invention of computers) and this has lead to the many different programming code solutions to the problem [4, 8, 13].

This paper presents a simple algorithm that can quickly and efficiently generate the points of the Hilbert and Peano curves using the simplest recursive technique.

## 2. IMPLEMENTATION

The basis of this implementation is the deconstruction of the curve into a set of unit shapes, as shown in Figure 2 for the Hilbert curve.

The relative position and rotation of each unit shape is defined by its sequential position in the curve generation. As the resolution of the curve increases, more unit shapes are required for its description, but the principle remains true to Hilbert's original proposition of dividing each part into smaller parts.
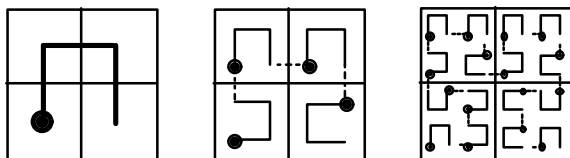


**Figure 2. De-constructed unit shapes for different curve resolutions. The circles represent the start points.**

As the curve progresses, the rotation of each unit shape follows a pattern that must be efficiently coded to describe both the rotation of the unit shape and its start and end points, so that the points of the curve can be plotted in the correct position and the correct order. The scheme chosen codes the rotation of each unit shape into two variables: $i1$ and $i2$, which represent the starting point and end point respectively, as shown in Figure 3.
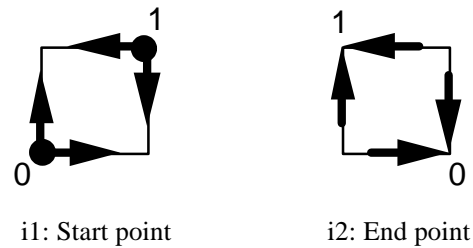


i1: Start point        i2: End point

**Figure 3. Coding system to describe the start and end points of the unit shape.**

Variable $i1$ is given the value 0 when the starting point is in the lower left corner of the unit shape, and the value 1 when the starting point is in the upper right corner. Variable $i2$ is given the value 0 when the end point of the unit shape is in the lower right corner, and the value 1 when the end point is in the upper left corner. Figure 4 illustrates this coding system when applied to four of the possible orientations of the unit shape used to construct the curve.
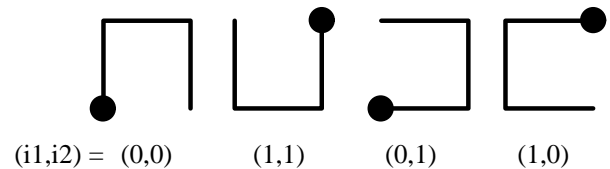


(i1,i2) =  (0,0)        (1,1)        (0,1)        (1,0)

**Figure 4. The coding system applied to four of the possible orientations of the unit shape.**

## 3. IMPLEMENTATION RESULTS

Program code is given for the generation of the Hilbert and Peano curves in Listing 1. The procedures work by recursively calling themselves, modifying their parameters with each call, until they reach the smallest unit shape for the next section of the curve to be plotted. The recursive algorithms could be modified so that the last recursive calls (i.e. those with lg = 1) directly produce the next four points, rather than calling the recursive functions again. This would greatly enhance the efficiency of the algorithms, but the performance gain is not so great during runtime, as the extra recursive calls made in the original algorithms are relatively time inexpensive.

The number of calls to the *Hilbert* procedure can be easily calculated:

$$\text{Number Calls} = W^2 + \frac{1}{3}\left(W^2 - 1\right)$$

$W$ = curve width ($2^m$; e.g. 1, 2, 4, 8, 16, 32, 64, ...)
1st term: calls that actually produce a point,
2nd term: due to recursive calling to find the points.

This gives the ratio:

$$\frac{\text{Points}}{\text{Calls}} = \frac{3W^2}{4W^2 - 1} \rightarrow 75\% \text{ for } W >= 8$$

At curve widths greater than 8, the algorithmic calling efficiency (number of points generated/number of calls to the recursive procedure) approximates to 75%. This is a relatively high ratio and demonstrates an efficient recursive procedure.

The recursive algorithm could be modified so that the last recursive calls (i.e. those with lg = 1) directly produce the next four points, rather than calling the *Hilbert* function again. This would greatly enhance the efficiency of the algorithm, but the performance gain will not be so great during runtime, as the extra recursive calls made in the original algorithm are relatively time inexpensive. This optimizing technique will speed the performance of the algorithm, but detracts slightly from its clarity, and so the algorithm given is the simplest, while not the most efficient.

## 4. SUMMARY

The same technique could be applied to other space-filling curves by identifying their unit shape, describing the points that make this unit shape, then developing a simple recursive procedure to generate all the points that make this curve from the unit shape. Another way to describe the process, as in the Hilbert curve, is that the square space is divided into a 2 x 2 array (or a 3 x 3 array for the Peano curve), and then further divided until the required resolution of curve is obtained. It would be possible to divide the same square space into a 4 x 4, or higher, and then by a simple modification of the recursive algorithm (requiring more calls for larger arrays), produce the required points. This would enable the construction of other forms of regular space-filling curves.

## 5. REFERENCES

[1] Asano, T., Ranjan, D., Roos, T., Welzl, E., and Widmayer, P., "Space Filling Curves and their Use in the Design of Geometric Data Structures," presented at LATIN '95: Theoretical Informatics, 1995.

[2] Bertilsson, M., Brickell, E., and Ingemarsson, I., "Cryptanalysis of Video Encryption Based on Space-Filling Curves," presented at Advances in Cryptology - EUROCRYPT '89, 1989.

[3] Bially, T., "Space-Filling Curves: Their Generation and Their Application to Bandwidth Reduction", *IEEE Trans. Information Theory*, vol. IT-15, pp. 658-664, 1969.

[4] Cole, A. J., "A Note on Space Filling Curves", *Software-Practice and Experience*, vol. 13, pp. 1181-1189, 1983.

[5] Hilbert, D., "Über die stetige Abbildung einer Linie auf ein Flächenstück", *Mathematische Annalen*, vol. 38, pp. 459-460, 1891.

[6] Lamarque, C.-H. and Robert, F., "Image Analysis using Space-filling Curves and 1D Wavelet Bases", *Pattern Recognition*, vol. 29, pp. 1309-1322, 1996.

[7] Lee, J.-H. and Hsueh, Y.-C., "Texture Classification Method Using Multiple Space Filling Curves", *Pattern Recognition Letters*, vol. 15, pp. 1241-1244, 1994.

[8] Musgrave, K., "A Peano Curve Generationm Algorithm," in *Graphics Gems II*, *The Graphic Gems Series*, J. Arvo, Ed. Boston: AP Professional, 1991, pp. 25.

[9] Norman, M. G. and Moscato, P., "The Euclidean Traveling Salesman Problem and a Space-Filling Curve", *Chaos, Solitons & Fractals*, vol. 6, pp. 389-397, 1995.

[10] Peano, G., "Sur une Courbe qui Remplit Toute une Aire Plane", *Mathematische Annalen*, vol. 36, pp. 157-160, 1890.

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

[12] Velho, L. and Gomes, J. d. M., "Digital Halftoning with Space Filling Curves", *Computer Graphics (SIGGRAPH '91)*, vol. 25, pp. 81-90, 1991.

[13] Witten, I. A. and Wyvill, B., "On the Generation and Use of Space-filling Curves", *Software-Practice and Experience*, vol. 13, pp. 519-525, 1983.

**Listing 1. Code in the C language to generate the Hilbert and Peano Curves.**

```c
#include <stdio.h>
#include <math.h>


void main(void) {
     int width = 64;                                   /* Curve width must be 2^m. */
     Hilbert(0, 0, width, 0, 0);                       /* Start Hilbert recursion. */
     width = 27;                                       /* Curve width must be 3^m. */
     Peano(0, 0, width, 0, 0);                         /* Start Peano recursion.   */
}


void Hilbert(int x, int y, int lg, int i1, int i2) {
  /* initial x, initial y, curve width, initial i1, initial i2 */
  if (lg == 1) {                                       /* Unit shape reached.      */
        printf("%d%c%d\n",x,',',y);                    /* Output coordinates.      */
        return;                                        /* Exit recursion.          */
  }
  lg >>= 1;                                            /* Divide by 2.             */
  Hilbert(x+i1*lg,        y+i1*lg,      lg,  i1,   1-i2);
  Hilbert(x+i2*lg,        y+(1-i2)*lg,  lg,  i1,   i2);
  Hilbert(x+(1-i1)*lg,    y+(1-i1)*lg,  lg,  i1,   i2);
  Hilbert(x+(1-i2)*lg,    y+i2*lg,      lg,  1-i1, i2);
}


void Peano(int x, int y, int lg, int i1, int i2) {
  /* initial x, initial y, curve width (3^m), initial i1, initial i2 */
  if (lg == 1) {
  printf("%d%c%d\n",x,',',y);                          /* Output coordinates.      */
  return;/* Exit recursion.                                */
  }
  lg = lg/3;                                           /* Divide by 3.             */
  Peano(x+(2*i1*lg),       y+(2*i1*lg),       lg,  i1,   i2);
  Peano(x+((i1-i2+1)*lg),  y+((i1+i2)*lg),    lg,  i1,   1-i2);
  Peano(x+lg,              y+lg,              lg,  i1,   1-i2);
  Peano(x+((i1+i2)*lg),    y+((i1-i2+1)*lg),  lg,  1-i1, 1-i2);
  Peano(x+(2*i2*lg),       y+(2*(1-i2)*lg),   lg,  i1,   i2);
  Peano(x+((1+i2-i1)*lg),  y+((2-i1-i2)*lg),  lg,  i1,   i2);
  Peano(x+(2*(1-i1)*lg),   y+(2*(1-i1)*lg),   lg,  i1,   i2);
  Peano(x+((2-i1-i2)*lg),  y+((1+i2-i1)*lg),  lg,  1-i1, i2);
  Peano(x+(2*(1-i2)*lg),   y+(2*i2*lg),       lg,  1-i1, i2);
}


  /*    Note. The multiplier 3 can be used to space the points aesthetically for plotting:
        printf("%d%c%d\n", x*3, ',', y*3);                              */
```