pftrail v1.01 manual

Herman Haverkort

31 March 2020

pftrail is a software package to render plane-filling curves and traversals as three-dimensional digital models, that are output in COLLADA format. These models can subsequently be imported in software such as Blender, to study them and to create photo-realistic images (due to fragile or degenerate features, the models produced by the current version of the software may not be suitable for 3D printing). The software is provided as a c++-source which can be compiled and run from the command line. It has been tested on Linux and MacOS systems. The package includes the following files: `pftrail.cpp`, `ifs-classics`, `ifs-inventions`, `generate-preamble.cpp`, `colours`, `postamble`, and this manual. New versions of the package may be available from http://spacefillingcurves.net or http://herman.haverkort.net.

Contents

1 Underlying concepts

1.1 Plane-filling curves

A plane-filling curve is a curve that twists such that it fills an entire two-dimensional shape, for example a square. The reader may now be tempted to think of the path of a child's colouring pencil as it fills a square in a drawing. However, a pencil stroke has some width, whereas a true plane-filling curve is infinitely thin. No matter how many infinitely thin lines one puts next to each other on paper, if they are infinitely thin, they will never fill the square entirely—yet infinitely thin curves exist that do fill the square. The secret of such plane-filling curves is that they are not only infinitely thin, but also infinitely crinkly.

Famous examples of such curves include Pólya's triangle-filling curve (also known as Sierpiński curve), and square-filling curves by Peano and Hilbert, but there are many other plane-filling curves with beautiful recursive structures. One may also define "curves" that fill the plane with crinkles but also contain "jumps", that is, parts where one travels directly from one point to another without covering any area in between. We will use the term *plane-filling traversals* to denote plane-filling curves with or without jumps. Lebesgue's plane-filling traversal, also known as the Z-order curve, is a well-known example. For references to the literature in which these curves were first described, see the file `ifs-classics` that is included in the package.

The order in which plane-filling traversals visit points in the plane can be exploited to design efficient algorithms and data structures for many purposes. Typically, they are useful for their *locality-preserving properties*: points that are close to each other in the traversal tend to be close to each other in the plane, and vice versa. However, two-dimensional sketches of plane-filling traversals often do not show this well: they are hard to read on different levels of detail and it is hard to see how far apart points are along the traversal. The pftrail package can be used to produce compelling visualisations of the traversals that give us more insight in their structure.

1.2 Defining plane-filling curves

Before we can discuss how to visualise a plane-filling traversal, we first need to discuss how one can define one. Consider, for example, Pólya's curve. We start with a single line segment (see Figure 1a). We refine this simple drawing as follows: let p and r be the first and the second endpoint of the original line segment. Imagine a circle with centre line pr and draw another point q halfway on the circle as we follow it clockwise from p to r. Erase the original line segment pr and replace it by two smaller segments pq and qr (Fig. 1b). Next, refine the drawing again by applying the same refinement procedure to each segment, but this time changing the orientation: to find the new intermediate points, we now follow the circles in counterclockwise direction. To indicate this change in orientation, we add an arrow head to pr on the left side, and put the arrow heads for pq and qr on the right side. In this way, two line segments become four segments (Fig. 1c). Note that the middle two segments lie on top of each other, but they have different directions. If we repeat this refinement process three more times, alternating clockwise and counterclockwise, and move all points slightly so that the curve does not back up on top of itself, we get Fig. 1d. If we continue ad infinitum, the curve fills a right isosceles triangle.

Mathematically, the plane-filling curve is a continuous, surjective mapping f from the unit interval (all numbers $t \in [0, 1]$) to an image (a set of points in the plane) $\cup_{t\in[0,1]} f(t)$, such that the image has non-zero area (formally defined by the concept of *Jordan content*). The image may be a simple shape such as a square or a triangle, as in the case of the Pólya curve, but it could also be something more adventurous,

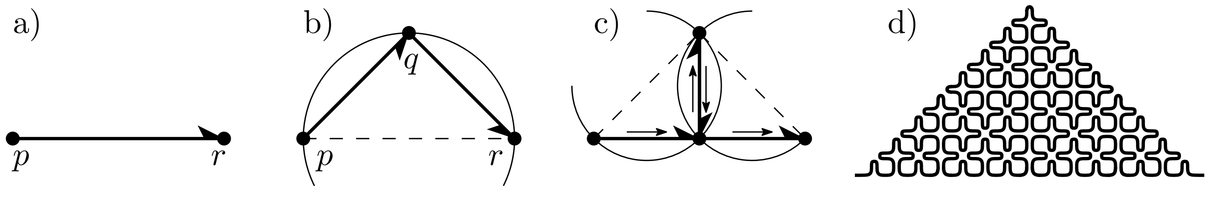


Figure 1: Pólya's triangle-filling curve.

for example a shape bounded by a fractal. We can trace the curve by going through the points $f(t)$ as t increases from 0 to 1. The mapping is usually chosen such that it is measure-preserving, that is, if we let t go from 0 to 1 at constant speed, then we fill the plane at constant speed. More precisely: for any interval $[a, z] \subset [0, 1]$, its image $\cup_{t\in[a,z]} f(t)$ has measure (area) $z - a$, where we take the area of the complete image of the curve as the unit of area. Thus, $f(t)$ is the point where the curve has arrived after filling a fraction t of the complete image. The reader may now verify that for Pólya's curve, $f(0)$, $f(1/2)$, and $f(1)$ are the points p, q and r in Figure 1: the curve starts at $f(0) = p$; the curve then leads to q over the infinite refinement of the first of two segments, which fills 1/2 of the complete triangle, hence $f(1/2) = q$, and it ends at $f(1) = r$.

Plane-filling curves usually have a self-similar structure: the curve consists of a finite number of parts $f_1, \ldots, f_n$, each of which are similar to the curve as a whole. Each part covers a part of the pre-image (the unit interval) and the image of the curve, such that together, the parts cover, that is, tessellate, the complete pre-image and the complete image. To define a particular plane-filling curve, one has to specify, for each part f_i of the curve, what similarity transformation maps the whole curve to the part f_i. Above, we did this with a figure that defines these similarity transformations implicitly by showing how they affect a line segment with an arrowhead.

To facilitate calculations, we can express the similarity transformations explicitly using matrix multiplications, as follows. We regard a point $f(t) = (x, y)$ of the curve as a four elements' column vector $(x, y, t, 1)$. For a part f_i of the curve, we specify the corresponding similarity transformation as a 4×4 matrix $M(i)$, such that if and only if $v = (x, y, t, 1)$ is a point of the curve, then $M(i)\,v$ is a point of the curve. The matrix $M(i)$ is of a restricted type: it scales down and possibly rotates, reflects and translates the x and y components of v to map the image of f to that of f_i, and it scales down and possibly reflects and translates the t-component of v to map the pre-image of f to that of f_i. Since each $M(i)$ induces scaling down, any product of an infinite sequence of matrices $M(i_1), M(i_2), M(i_3), \ldots$ with v converges to a vector that is independent of v. In the end, the curve consists of the relations $f(t) = (x, y)$ that are described by the vectors of convergence for all infinite sequences of matrices chosen from $M(1), \ldots, M(n)$.

For example, if we place Pólya's curve for an isosceles right triangle such that it starts at the point with coordinates $(0, 0)$ and ends at $(2, 0)$, then it is described by the following matrices:

$$M(1) = \begin{bmatrix} 1/2 & 1/2 & 0 & 0 \\ 1/2 & -1/2 & 0 & 0 \\ 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad M(2) = \begin{bmatrix} 1/2 & -1/2 & 0 & 1 \\ -1/2 & -1/2 & 0 & 1 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If we take, for example, $i_1, i_2, i_3, i_4, i_5, \ldots = 1, 2, 1, 2, 1, \ldots$, then the product $M(i_1)M(i_2)M(i_3)\ldots$ converges to a matrix whose last column is $(\frac{4}{5}, \frac{2}{5}, \frac{1}{3}, 1)$ and all other entries are zero; thus, $f(\frac{1}{3}) = (\frac{4}{5}, \frac{2}{5})$ is a point of the curve.

Any notation system for plane-filling curves and traversals specifies the matrices $M(i)$ in one way or another. Actually writing out 16-element matrices would, usually, be unnecessarily cumbersome. Our initial description of Pólya's curve defined the similar transformations implicitly by showing, for each part of the curve, the image of a line segment with an asymmetric arrowhead. However, input for computer software requires a textual format. The notation system used by pftrail is a textual format that is essentially a concise method to describe the line segments with arrowheads. The format basically stems from the book *Brainfilling curves: a fractal bestiary* by Jeffrey Ventrella; all curve definitions from that book can be processed directly by pftrail. For further details, see Section 4.1.

1.3 Visualising plane-filling curves as plane-filling trails

Sketches such as Figures 1b and 1d do not make it clear in an instant in what order a traversal fills exactly what parts of the plane—not to mention giving an impression of the traversal's locality-preserving properties and violations thereof. The purpose of pftrail is to clarify this using a three-dimensional approach. pftrail reads a definition of a plane-filling traversal and produces a *plane-filling trail*, a model of the traversal on a three-dimensional landscape, in which each point $f(t) = (x, y)$ of the traversal is

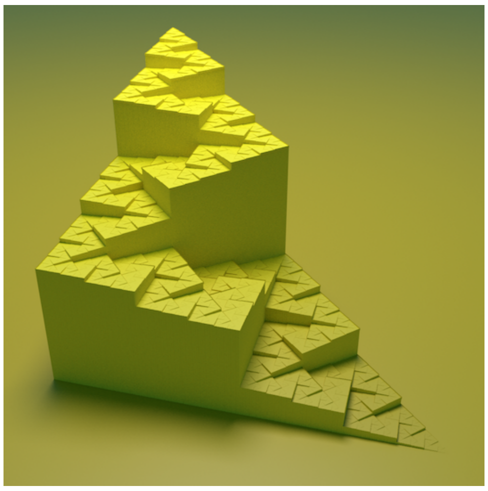
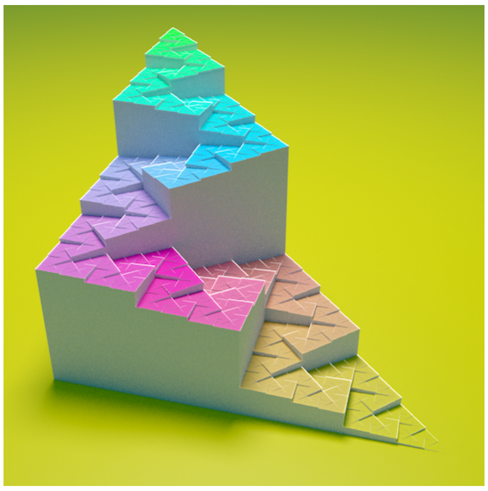
a)  b) 

Figure 2: a) Pólya's curve. b) Pólya's curve with a colour gradient.

rendered as a point (x, y, t). Thus the traversal becomes a steadily ascending path in the landscape, see Figure 2. High, steep slopes now reveal pairs of points that are close in the plane but far apart along the traversal. Narrow corridors would reveal sections between points that are relatively close to each other along the traversal but far apart in the plane. Wide corridors show sections of the traversal that have good locality-preserving properties in both directions. The global course of the traversal is easy to follow, but the image also facilitates studying the curve in more detail. Moreover, the visualisation is independent of what definition of the traversal is used, out of multiple equivalent definitions. The visualisation gives the user the possibility to study the traversal without any bias towards an arbitrary choice of an underlying tessellation.

Alternative visualization methods that come closest to meeting the same goals may render the t-coordinate as values on a grey or colour scale instead of elevation. For complicated, self-crossing curves, it may be useful to combine both approaches. The pftrail software can also do this, as in Figure 2b.

The pftrail tool works as follows. It reads the definition of a traversal in the format from Ventrella's book. In fact, pftrail reads an extended version of this format that supports traversals with jumps and traversals with multiple segment replacement rules (known as *generators*). Thus, the various traversals that have been proposed in the computer science literature can all be rendered and it is easy to explore new designs. The traversals are not confined to an integer grid, so we can also get a picture of interesting curves related to, for example, the Rauzy fractal (see Fig. 3). For rendering, the traversal is sampled and drawn on a grid of hexagonal cells (the necessary sampling density is determined automatically); thus pftrail operates without any knowledge of the shape of the part of the plane filled by the traversal (which can be a complicated fractal). With instructions in the input file and on the command line, one can control parameters such as camera angle and focus, resolution of the rendering grid, visualisation style, and the height of small "parapets" that can accentuate steep drops to enhance the perception of depth. The output is a COLLADA file that can be rendered with, for example, Blender; if the resolution is not too high, it is also possible to move the three-dimensional model around in Blender in real time.

Special features include "polynomial" close-up: given a focus point p and a zoom parameter α, any point q at distance x from p is moved to the point at distance $x^{1/\alpha}$ from p on the ray from p through q. Elevation differences are modified in a similar way. This allows us to zoom in on features that remain invisible in normal close-up views. For example, the Gosper curve (Fig. 3, bottom left) follows a tessellation with tiles arranged in a hexagonal grid pattern. At the vertices of this grid, the tiles wind around each other like logarithmic spirals that shrink by a huge factor of roughly $9 \cdot 10^7$ per revolution. No normal close-up view could show these spirals, but the polynomial close-up reveals them clearly, see Figure 3.

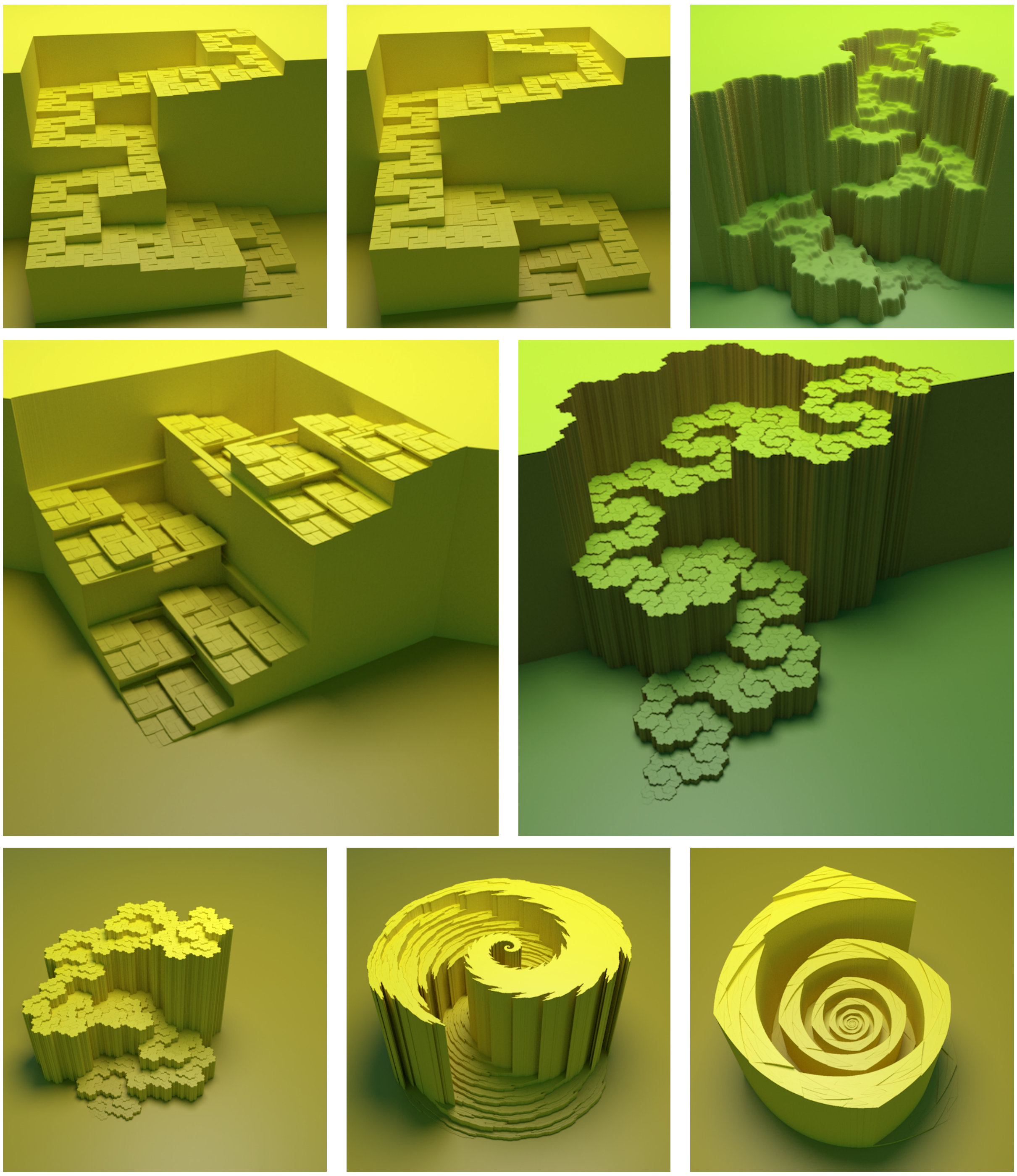


Figure 3: On top: Hilbert's curve, $\beta\Omega$, and a curve from Ventrella filling a fractal "pinwheel" tile, rendered "eroded". In the middle: Double-Gray-code and a curve filling half of a Rauzy fractal. Bottom: the Gosper curve, a close-up of the point at 2/7 of the curve where three tiles meet ($\alpha = 8$), and a close-up of the point at 1/3 of Pólya's curve ($\alpha = 5$). For references to the sources of the curves, see the file `ifs-classics` that is included in the package.

2 How to set-up the software

The package contains six files: `pftrail.cpp`, `generate-preamble.cpp`, `ifs-classics`, `ifs-inventions`, `colours`, and `postamble`. Unpack these files and compile the cpp-files; on a Linux or MacOS system with g++ installed, this should be possible with the commands:

```
g++ -o pftrail pftrail.cpp
g++ -o generate-preamble generate-preamble.cpp
```

3 How to use the software

The software produces models in COLLADA format that can be imported in, for example, Blender. The COLLADA files are assembled from three parts. The main part, the geometry of the model, is produced by pftrail. It needs to be composed with a *preamble* that defines the materials of the model, and a *postamble* that puts model, light and camera together. Preambles can be generated with generate-preamble (and edited further by hand, if desired); a fixed postamble comes with the package. For example, we can create a (low-resolution) model of a Pólya curve as follows:

```
./generate-preamble 1 colours:yellow >preamble
./pftrail ifs-classics:polya >geometry
cat preamble geometry postamble >polya.dae
```

The first line generates the preamble, using the `yellow` colour scheme from the file `colours` (included in the package), defining a single (1) colour for the surface of the path-filling trail. The second line generates the geometry, using the plane-filling traversal definition and the scene settings (viewing direction etc.) called `polya` from the file `ifs-classics` (also included in the package). The third line concatenates the preamble, the geometry, and the postamble, to produce a complete COLLADA (`.dae`) file.

The remaining sections of this document explain how to write files with definitions of plane-filling traversals and scene settings, what further options for pftrail can be specified on the command line, and how to use generate-preamble to produce different colour schemes. At the end, examples are given that can be used to produce the figures from the introduction.

4 Writing plane-filling traversal definitions

4.1 Defining a plane-filling curve

Basic plane-filling traversals can be defined using the format from the figures in Ventrella's book *Brain-filling curves: a fractal bestiary*. For example, here is the Pólya curve:

```
Square grid
2 segments

segment values:
1: 1, 1, 1,-1
2: 1,-1, 1,-1
```

One could, for example, make a file called `polya` with exactly these contents, and then call pftrail like this:

```
./pftrail polya >geometry
```

The basic format of such a plane-filling curve definition is as follows. The first line specifies the coordinate system. There are three options:

- `square` for Cartesian coordinates;
- `triangular` to put the positive y-axis at a 60 degrees' angle with the positive x-axis;
- `cubic` to use three coordinate axes at 120 degrees' angles to each other, in which vectors are treated as equivalent if their difference is a scalar times $(1, 1, 1)$. This is very similar to the second option, but more symmetric. Triangular coordinates (x, y) correspond to cubic coordinates $(x, -y, y - x)$.

The remaining lines define the rule for how to rewrite a line segment (as in Figure 1). For each segment, there are five numbers i: x_i, y_i, d_i, o_i, or, if cubic coordinates are used, six numbers i: x_i, y_i, z_i, d_i, o_i. These are interpreted as follows:

- i is just the rank of the segment in the sequence;
- x_i, y_i, z_i are the coordinates of the segment's end point relative to its starting point (the starting point is the end of the previous segment; the first segment starts in the origin of the coordinate system);
- d_i specifies on which end to put the arrowhead: `-1` to put the arrowhead at the starting point, or `1` to put it at the other end;
- o_i specifies whether to put the arrowhead on the left or on the right side of the segment (as seen from its starting point): `-1` to flip it to the right side; `1` to keep it on the left side.

For readers who want to know what goes on behind the scenes: the similarity transformations $M(i)$ are now derived as follows. For $i \in \{0, \dots, n\}$, define $X_i := \sum_{j \le i} x_i$ and $Y_i := \sum_{j \le i} y_i$ and $T_i := \sum_{j \le i} ||(x_i, y_i)||^2 / ||(X_n, Y_n)||^2$. The transformation matrix $M(i)$ is now derived as the unique transformation matrix that satisfies the following conditions: (1) if $d_i = 1$, then $f(0)$ is mapped to (X_{i-1}, Y_{i-1}) and $f(1)$ is mapped to (X_i, Y_i), whereas if $d_i = -1$, then $f(1)$ is mapped to (X_{i-1}, Y_{i-1}) while $f(0)$ is mapped to (X_i, Y_i); and (2) the transformation induces a reflection (in three-dimensional x, y, t-space) if and only if $o_i = -1$.

Any curve from Ventrella's book can be rendered by putting the definition from Ventrella's book in a file and giving this file as the command line argument to pftrail.

4.2 Organizing plane-filling curves in files

To avoid creating an unorganized pile of small files with incomprehensible sequences of numbers, pftrail allows the user to give curves names (or even multiple names), to include comments, and to put multiple curves in one file. For example, one could make a file `mycurves` that defines Pólya's curve and Gosper's flowsnake:

```
IFS polya
/* From G. Pólya: Über eine Peanosche Kurve.
Bull. Int. Acad. Sci. Cracovie, Ser. A, 1913, pp 305-313. */

Square grid
2 segments

segment values:
1: 1, 1, 1,-1 // fills triangle with corners (0,0),(1,0),(1,1)
2: 1,-1, 1,-1 // fills triangle with corners (1,1),(1,0),(2,0)

IFS gosper
/* From M. Gardner: Mathematical Games---In which
"monster" curves force redefinition of the word "curve".
Scientific American, 235(6):124-133 (1976) */

Cubic grid
7 segments

segment values:
1: 1, 0, 0, 1, 1
2: 0,-1, 0,-1,-1
3:-1, 0, 0,-1,-1
4: 0, 0, 1, 1, 1
5: 1, 0, 0, 1, 1
6: 1, 0, 0, 1, 1
7: 0, 0,-1,-1,-1
```

Now, on the command line for pftrail, one can specify the curve by giving the file name, followed by a colon and the name of the curve. For example:

```
./pftrail mycurves:gosper >geometry
```

Note that names are case-insensitive, and they cannot include spaces; quotes or escape characters are not recognized. Comments in the file come in two flavours. They either start with `//` and end at the end of the line, or they start with `/*` at the beginning of a line and end at the next occurrence of `*/`. Comments of the second type cannot be nested.

The pftrail package includes two files, `ifs-classics` and `ifs-inventions`, that contain definitions of many well-known and some lesser-known plane-filling traversals.

4.3 Traversals with jumps

To model traversals with jumps, we may include segments without arrowheads, that is, with $d_i = o_i = 0$. These will not be refined recursively, but rendered as straight line segments, modelling a jump from one end to the other. Often, these segments may end up looking like bridges or tunnels in the rendering; the details of this can be controlled by various command line options. For example, here is the Lebesgue curve (Z-order):

```
Square grid
7 segments

segment values:
1: 1, 1, 1, 1
2: 0,-1, 0, 0
3: 1, 1, 1, 1
4:-2, 0, 0, 0
5: 1, 1, 1, 1
6: 0,-1, 0, 0
7: 1, 1, 1, 1
```

If refinement of a traversal results in sequences of consecutive jumps, they are shortcut: only one line segment is rendered, that starts at the beginning of the sequence and ends at the end of the sequence. To render consecutive jumps separately, one can prevent the shortcutting by inserting non-jump segments of zero length. For example, here is U-order, with zero-length non-jump segments added so that each jump is rendered by two consecutive axis-parallel segments that form an L-shape, instead of single diagonal segments:

```
IFS u-with-L-jumps

Square grid
10 segments

segment values:
 1: 1, 0, 1, 1
 2: 0, 1, 0, 0  // first leg of jump
 3: 0, 0, 1, 1  // zero-length non-jump segment prevents
                // the two legs from being joined into one
 4:-1, 0, 0, 0  // second leg of jump
 5: 1, 0, 1, 1
 6: 1, 0, 1, 1
 7:-1, 0, 0, 0  // first leg of jump
 8: 0, 0, 1, 1  // zero-length non-jump segment prevents
                // the two legs from being joined into one
 9: 0,-1, 0, 0  // second leg of jump
10: 1, 0, 1, 1
```

4.4 Traversals with multiple generators

Traversals with multiple generators (rules to replace a segment by a chain of smaller segments) are also supported. For example, here are the rules for the Ω- and β-sections of the $\beta\Omega$-curve:

```
IFS beta-omega(omega)

Square grid
2 generators

generator A: // Omega
4 segments

segment values:
1: 2, 1, B,-1,-1
2: 1, 2, B,-1, 1
3: 1,-2, B, 1, 1
4: 2,-1, B, 1,-1

generator B: // Beta
4 segments

segment values:
1: 0, 3, A, 1, 1
2: 1, 2, B,-1, 1
3: 1,-2, B, 1, 1
4: 2,-1, B, 1,-1
```

The second line states how many generators there are. The generators are then specified one by one. They need to be indexed alphabetically as indicated. In the line for each segment, between the end point coordinates and the specification of its arrowhead, one finds a letter that says which generator (refinement rule) is to be used for that segment.

To specify jumps in a multi-generator traversal, do not specify zero direction and orientation for the arrowhead. Instead specify a segment with generator `X` and omit the arrowhead specification (see Section 4.5 for an example).

Rendering always starts with generator A.

Sometimes pftrail needs a hint For traversals defined by multiple generators, pftrail needs to know, for each generator, how much area is filled by the traversal that results from expanding that generator—otherwise elevations could not be calculated correctly. Under the assumption that the traversal is indeed plane-filling, pftrail can usually deduce this automatically from the dependencies between the generators. However, one can define plane-filling traversals for which this is not possible. For example, if a curve is a concatenation of two unrelated plane-filling curves, pftrail has no way of calculating the ratio between the area filled by the second part and the area filled by the first part. In that case, these areas should be specified with `fills`, as in the following example:

```
IFS concatenate-hilbert-and-polya

Square grid
3 generators

generator A:
2 segments

segment values:
1: 1, 0, B, 1, 1 // Hilbert curve section (scaled, fills area 1)
2: 1, 1, C, 1, 1 // Polya curve section (scaled, fills area 1/2)
```

```
generator B: // Hilbert curve
fills 4 // total area filled by the segments specified below
4 segments

segment values:
1: 0, 1, B, 1,-1
2: 1, 0, B, 1, 1
3: 1, 0, B, 1, 1
4: 0,-1, B, 1,-1

generator C: // Polya curve
fills 1 // total area filled by the segments specified below
2 segments

segment values:
1: 1, 1, C, 1,-1
2: 1,-1, C, 1,-1
```

4.5 Traversals with coinciding end points

Some plane-filling curves in the literature are closed, that is, they end where they begin. In such cases, the choice of starting point for a closed curve is arbitrary, and the elevation difference between points along the trail only corresponds to their distance along the curve as long as the shortest connection along the curve does not pass (or is not allowed to pass) through the arbitrary starting point. For this reason, I do not recommend visualizing such curves with pftrail.

To render a closed plane-filling curve with pftrail nonetheless, we need a little trick, since Ventrella's notation system is fundamentally unable to support curves with coinciding endpoints[1]. Fortunately, no plane-filling curve consists exclusively of sections with coinciding endpoints, since such curves would not be able to move around to fill the plane. So here is the trick we can use: one can cut up the closed curve into parts that are not closed, write generators for those parts, and write a special first generator that produces: (1) a jump to the starting point of the closed curve, (2) the closed curve (in two or more parts). In the model that is computed, the initial jump will be omitted.

For example, here is Moore's closed variation of Hilbert's curve.

```
Square grid
2 generators

generator A: // Loop
5 segments

segment values:
1: 0,-2, X   // sentinel initial tunnel (not rendered)
2: 0, 1, B, 1, 1
3: 0, 1, B, 1, 1
4: 0,-1, B, 1, 1
5: 0,-1, B, 1, 1

generator B: // Hilbert curve
4 segments

segment values:
1: 0, 1, B,-1,-1
2: 1, 0, B, 1, 1
3: 1, 0, B, 1, 1
4: 0,-1, B,-1,-1
```

1. The similarity transformations that map the curve as a whole to its individual segments, are determined by mapping the arrow for the curve as a whole to the arrows for the individual segments. For this reason, the arrow for the curve as a whole must have non-zero length.

5 Defining scene settings

5.1 Writing view configurations

An IFS specification may be followed by instructions that define the scene. Many IFS specifications in the files `ifs-classics` and `ifs-inventions` contain such instructions. There are six types of instructions, which can be given in any order. None of them are mandatory. The six possible instructions are the following:

- `Render` x, y, d, o
  Specifies with what line segment to start the expansion of the traversal. The format is independent of the traversal definition: it is always in Cartesian coordinates, and, for multiple-generator curves, no generator is specified (the expansion always starts with the first generator). With default camera settings (see below), the positive x-axis points away from the viewer, the positive y-axis points to the left, and d and o specify the orientation as with segments in IFS definitions. In particular, with $d = 1$, $o = 1$ the traversal is drawn as normal from $(0, 0)$ to (x, y); with $d = -1$ the traversal is drawn from (x, y) to $(0, 0)$, reflected in the line through those points, and descending instead of ascending; with $o = -1$, the traversal is reflected in the line through $(0, 0)$ and (x, y). The default values are 1,0,1,1.
- `Ridge` x
  As reference points for elevation, a lower reference plane is generated at the level of the starting point of the traversal, and an upper reference (half-)plane in the back is generated at the level of the end point of the traversal. The upper plane starts at a ridge/fault line parallel to the y-axis, that is, running from left to right. Parts of the traversal that lie behind that line, are cut out from the reference plane (see the first five examples in Figure 3). With `Ridge`, one specifies the distance from the starting point to the fault line, where the unit of distance is the distance between the end points of the traversal. Note that that does not mean that with $x = 1$, the fault line passes through the end point. If, for example, the traversal is rendered starting from the segment from $(0, 0)$ to $(3, 4)$, then the unit distance for the fault line placement is 5, and to make the ridge pass through the end point of the traversal, we would have to specify `Ridge 0.6`. To omit the upper reference plane altogether, specify a negative value for x. By default, the ridge is place in the middle between the end points of the traversal.
- `Altitude` α
  Specifies the altitude of the camera in degrees: 0 is level with the point at which it is directed; 90 is straight above it, pointing down. The default value is 18.5.
- `Azimuth` α
  Specifies the direction of the camera in the projection on the horizontal plane, in degrees: 0 is the direction of the positive x-axis; positive values specify a view from the right to the left; negative values specify a view from the left to the right. The default value is 0.
- `Centre` a
  With `Centre` one can specify the point at which the camera is directed (ignoring elevation) and the point on which a close-up view (if applied) zooms in. Because successful close-ups depend on highly accurate centre coordinates, which can be cumbersome to calculate by hand, the centre point is not specified by Cartesian coordinates, but by specifying the traversal segment whose starting point it is. For this purpose, indexing starts at zero. For example, `403` specifies, in the expansion of the starter line segment, the 5th segment; within its expansion, the 1st segment, and within the expansion of that one, the 4th segment. To specify the endpoint of the traversal, we write the number of segments of the traversal. For example, for a traversal whose first generator has seven segments, `7` would specify the end point of the traversal. If the traversal specification or the `Render` line contains negative d-values, then one should be careful: segments are always counted in direction of increasing elevation, which, depending on the context, may not be the direction of increasing index in the IFS definition.
  Note that if the traversal has just one generator with n non-jump segments that all have the same size, then a will just be the fractional part of the elevation of the centre point in base-n notation. For example, the Pólya close-up in Figure 3 is made with:
  `Centre 01010101010101010101010101010101010101010101010101010101010101`
  which specifies the point $f((\text{binary})0.010101...) = f(1/3)$, at 1/3 of the length of the curve.

By default, the point at which the camera is directed is calculated such that the whole traversal can be seen from the smallest possible distance, given the altitude and azimuth settings.

- `Distance` x
  Specifies the distance of the camera to the centre of the drawing grid, relative to the minimum distance from which it can see the whole traversal within a square viewing window with a 37 degrees' field of view (this matches the viewing window as defined in preambles generated by generate-preamble). The default value is 1.1.

5.2 Selecting view configurations

To facilitate the use of different view configurations for the same traversal, or the use of the same view configuration for different traversals, view configurations can also be defined separately from the traversal. To read the view configuration from a file `filename`, run pftrail with the option `-c filename`. For example, we could make a file `topview` with the following contents:

```
ridge -1
altitude 90
```

Now, to create a scene in which the Pólya curve is seen straight from above, we run:

```
./pftrail ifs-classics:polya -c topview >geometry
```

Instead of the configuration that follows the definition of the Pólya curve in `ifs-classics`, pftrail now uses the configuration from the file `topview`.

It is also possible to give view configurations a name, so that multiple view configurations can be put in one file (even mixed with curve definitions) and selected by name. For example, `ifs-classics` contains the following view configuration:

```
view polya-f(1/3)

Render 1,0,1,1
Altitude 60
Azimuth -30
Centre 010101010101010101010101010101010101010101010101010101010101
```

Now the following line creates a scene in which the camera is directed at the specified centre point, at 1/3 of the length of the curve:

```
./pftrail ifs-classics:polya -c "ifs-classics:polya-f(1/3)" >geometry
```

Note the quotes: these may be necessary in this case, because otherwise the shell may choke on the parentheses in the name of the view configuration.

Because in this case, the view configuration comes from the same file as the traversal definition, the file name for the view configuration may be omitted (but do not omit the colon):

```
./pftrail ifs-classics:polya -c ":polya-f(1/3)" >geometry
```

6 pftrail command line options

6.1 Accuracy: `-a` x

pftrailsamples the plane-filling traversal such that for each point p on the traversal, a sample point p' is generated such that between p and p', the traversal stays within a distance of $x/4$ times a grid cell diameter from p'. By default, $x = 1$. This guarantees, among other things, that for any grid edge that is completely covered by the traversal, there is a sample point in at least one of the grid cells sharing that edge. With the `-a` option, one can change the value of x to sample less densely or more densely. In particular, lowering x to 1/2, or maybe even 1/4, could facilitate the generation of a quick sketch, especially when the sampling density determines the performance bottleneck, which is the case for polynomial-zoom views with high exponents. Raising x may help mitigating artefacts like Moiré patterns that may appear, especially, along some straight-edge boundaries of (parts of) the traversal.

6.2 Bridge width: `-b` x

Jumps (discontinuities) are rendered as bridges or tunnels whose width depends on the length of the jump. With the `-b` option, one can set the width of a jump that is almost as long as the diameter of the region filled by the traversal. The widths of shorter jumps are adapted accordingly. The default value is 0.015.

6.3 View configuration: `-c` filename[:viewid]

Specify the view configuration, see Section 5.2.

6.4 Rectangular bridges: `-f`

By default, bridges and tunnels that model jumps are rendered widest in the middle, whereas they start and end in a point. This is to minimize conflicts between bridges and the local traversals around their end points—in the middle, they typically pass parts of the traversals that are further away so that they are separated vertically and do not interfere with the bridge or tunnel. Alternatively, bridges and tunnels can be given a rectangular shape, that is, with the same width along their entire length. This is done with the `-f` option.

6.5 Colour gradient resolution: `-g` n

The three-dimensional model that is created, uses five types of materials: one for the lower reference planes, one for the upper reference plane, one for vertical walls, one for "parapets" (see the `-p` option), and n different materials for the surface of the path-filling trail itself (the default value of n is 1). More precisely, the section of the trail from elevation 0 to elevation i/n uses trail material number i. This can be used to simulate a colour gradient along the path, provided a matching preamble is used that defines n trail materials—see Section 7 for how to create such preambles.

6.6 Tunnel height: `-h` x

If the (default) `solid` drawing style is used (see the `-s` option below), this option sets the minimum height of open space (default: 0.01). If the distance between sample points in the same grid cell is too small to render the lower sample as a tunnel, cave or underpass with minimum height under the upper sample, then the space between the two sample points is filled up, or, if the lower sample has higher priority, the upper sample is omitted.

6.7 Kernel radius: `-k` n

If the `eroded` drawing style is used (see the `-s` option below), then the `-k` option specifies how many iterations are run in which each grid cell's elevation is replaced by the mean of itself and its neighbours. The default is 0.

6.8 Median strip width: `-m` x

If the distance between sample points in the same grid cell is too small to render the lower sample as a tunnel, cave or underpass with minimum height under the upper sample, then the space between the two sample points is filled up, or, if the lower sample has higher priority, the upper sample is omitted. The lower sample has higher priority if it is part of the "median strip" of a jump whereas the upper sample is not, or if both samples are part of the "median strip" of a jump, but the lower sample's jump is longer. The median strip of a jump is the part of the bridge or tunnel shape that lies within distance $x/2$ from its centre line. The default value for x is 0.01.

6.9 Resolution: `-r` n

Sets the resolution, defined as image diameter divided by grid cell diameter. pftrail may decide to use a higher resolution. In particular, pftrail bases the sampling density on a lower bound on the image diameter. Especially when using polynomial zooming, depending on the centre point, the lower bound is not very tight. In hindsight, the samples may then turn out to suffice for a higher resolution, and pftrail will use the higher resolution.

6.10 Parapet height: `-p` x

If the (default) `solid` or the `floating` drawing style is used (see the `-s` option below), then the `-p` option (default value 0.002) specifies the maximum height of little walls (parapets) that are placed at the top of large drops. These walls enhance the perception of depth when the edge of the drop is seen from behind (from the "mountain" side). For the outer ("valley") side of the parapets, the same material is used as for other vertical walls; for the inner ("mountain") side, the parapet material is used. Parapets have their maximum height when placed at the top of drop of at least 0.02 (this value is hard-coded in the section *Tunable constants etc.* in `pftrail.cpp`). For smaller drops, the parapet height decreases linearly to zero.

6.11 Quiet mode: `-q`

Quiet mode suppresses almost all progress reports and other output on the standard error stream.

6.12 Visualisation style: `-s` *style*

The option selects the visualisation style: possible values are `solid` (default), `floating`, and `eroded`. The traversal is drawn by drawing sample points on a hexagonal grid. In the `solid` style, when multiple samples fall into the same grid cell, the space between samples with a very small difference in elevation is filled or the upper sample is omitted (see the various other options that control this behaviour), and the space below the lowest sample and the ground plane is filled. The `floating` style is similar, but without filling the space down to the ground plane. In the `eroded` style, when multiple samples fall into the same grid cell, their mean elevation is used (with priority given to samples on longer jumps), and the space between it and the ground plane is filled. In the `eroded` style, the landscape may be smoothened further with `-k` option (see Section 6.7).

6.13 Verbose mode: `-v`

In verbose mode, pftrail gives more detailed output on standard error, especially about the properties of the iterated function system that is created from the input and about the parameter settings that are used. (Some of this information can also be found as comments in the output, regardless of whether verbose mode is used.)

6.14 Slab weight: `-w` x

If the (default) `solid` or the `floating` drawing style is used (see the `-s` option above), then the `-w` option sets the minimum weight (thickness) of the surface that supports the path-filling trail. The default value is 0.005.

6.15 Zoom exponent: `-z` x

This option selects polynomial-zoom view and sets the exponent for zooming. The centre point must be set with a `centre` instruction in the view configuration (see Section 5.1). Any point at horizontal distance h, vertical distance v from the centre will be moved to horizontal distance $h^{1/x}$, vertical distance $v^{1/\alpha(x)}$ from the centre, where $\alpha(x) = 1.5x - 0.5$. Thus, the vertical distortion is more extreme than the horizontal distortion. The reason for this is that in the representation of a plane-filling traversal by an ascending trail, subsections of the traversal inevitably look flatter than the traversal as a whole. Therefore, when we zoom in, we should somehow increase the elevation "contrast" to compensate for the flattening (the function α is hard-coded in the section *Tunable constants etc.* in `pftrail.cpp`). The resulting distances are scaled linearly so that the result fits on the rendering grid.

Note that high-resolution grids with high-exponent zooming may not always be feasible and/or suffer from visible artefacts near the centre point (if it is not occluded by other parts of the landscape). The first problem is caused by the fact that polynomial zooming distorts any region in the plane in such a way that it is stretched in the direction orthogonal to the ray to the centre. As a result, zooming in requires many more sample points, which may lead to memory problems for the current implementation. The second problem is that standard number types may not provide enough precision. Rounding errors of $1/2^{63}$ result in elevation differences of $1/2^{63/(1.5x-0.5)}$. For zoom exponents $x > 5$, this is more than $1/2^9 = 1/512$, which may lead to visible artefacts near the centre of zooming.

7 Using generate-preamble

generate-preamble takes two arguments: first, the number of trail surface materials that must be generated, and second, the file name of the colour scheme definition. The number of trail surface materials (colours) should match those used by pftrail, which uses 1 material by default and allows this to be changed with the `-g` option (see Section 6.5). Similar to traversal definitions (IFSs) and view configurations, it is possible to give colour schemes a name, put multiple named colour schemes in a file, and select them by appending a colon and the colour scheme name to the file name. Examples of colour schemes can be found in the file `colours` that is included in the package.

Colour schemes follow a fairly strict format. To give the scheme a name (this is the only optional part), start with `scheme`, followed by the name. After that, one needs to specify the materials for the trail surface, the lower reference plane, the upper reference plane, vertical walls, and the inside of parapets—in that order. Each of these materials is specified by a keyword (`trail`, `lowland`, `highland`, `walls`, or `parapets`), followed by five values that specify the material—except that for trail material, multiple of these five-tuples can be specified. Multiple materials for the trail surface are separated by commas; the last material of each category is followed by a semicolon, except for the last material of the scheme, which is followed by a full stop. The five values of a material specify the red component, the green component, and the blue component of diffusely reflected colour; how much of this colour is additionally emitted; and finally, the opacity of the material (0 is fully transparent, 1 is fully opaque). Any number of trail surface materials can be generated from any colour scheme: the required colour gradient is constructed by linear interpolation between the colours that are specified in the scheme.

For example, here is the definition of the `transparent` scheme:

```
scheme transparent

trail    0.75 0.66 0.00 0.00 1.00 ,
         0.80 0.45 0.11 0.00 1.00 ,
         0.87 0.23 0.24 0.00 1.00 ,
         0.93 0.00 0.37 0.00 1.00 ,
         0.51 0.27 0.68 0.00 1.00 ,
         0.00 0.49 0.87 0.00 1.00 ,
         0.00 0.75 0.44 0.00 1.00 ,
         0.00 1.00 0.00 0.00 1.00 ;

lowland  0.75 0.66 0.00 0.00 1.00 ;
highland 0.00 1.00 0.00 0.00 1.00 ;
walls    0.65 0.65 0.65 0.00 0.30 ;
parapets 1.00 1.00 1.00 1.00 1.00 .
```

8 Examples

The COLLADA files for the figures in this paper can be generated as follows:

Figure 2:

```
./generate-preamble 256 colours:yellows >polya-yellows.dae
./pftrail ifs-classics:polya -r 1000 -p 0.004 -g 256 >>polya-yellows.dae
cat postamble >>polya-yellows.dae

./generate-preamble 256 colours:multicolour >polya-multicolour.dae
./pftrail ifs-classics:polya -r 1000 -p 0.002 -g 256 >>polya-multicolour.dae
cat postamble >>polya-multicolour.dae
```

Figure 3:

```
./generate-preamble 256 colours:yellows >hilbert.dae
./pftrail ifs-classics:hilbert -r 1000 -p 0.004 -g 256 >>hilbert.dae
cat postamble >>hilbert.dae

./generate-preamble 256 colours:yellows >beta-omega.dae
./pftrail ifs-classics:beta-omega -r 1000 -p 0.004 -g 256 >>beta-omega.dae
cat postamble >>beta-omega.dae

./generate-preamble 256 colours:greens >fractal-pinwheel-eroded.dae
./pftrail ifs-classics:fractal-pinwheel -r 1000 -s eroded -k 25 -g 256 >>fractal-pinwheel-eroded.dae
cat postamble >>fractal-pinwheel-eroded.dae

./generate-preamble 256 colours:yellows >double-gray.dae
./pftrail ifs-classics:double-gray -r 1000 -p 0.004 -h 0.015 -b 0.027 -m 0 -g 256 >>double-gray.dae
cat postamble >>double-gray.dae

./generate-preamble 256 colours:greens >rauzy-triangle.dae
./pftrail ifs-classics:rauzy-triangle -r 1000 -p 0.004 -g 256 >>rauzy-triangle.dae
cat postamble >>rauzy-triangle.dae

./generate-preamble 256 colours:yellows >gosper.dae
./pftrail ifs-classics:gosper -c ":gosper-f(2/7)" -r 1000 -p 0.002 -g 256 >>gosper.dae
cat postamble >>gosper.dae

./generate-preamble 256 colours:yellows >gosper-zoom.dae
./pftrail ifs-classics:gosper -c ":gosper-f(2/7)" -r 500 -p 0.002 -g 256 -z 8 >>gosper-zoom.dae
cat postamble >>gosper-zoom.dae

./generate-preamble 256 colours:yellows >polya-zoom.dae
./pftrail ifs-classics:polya -c ":polya-f(1/3)" -r 500 -p 0.002 -g 256 -z 5 >>polya-zoom.dae
cat postamble >>polya-zoom.dae
```