Domain-Specific Languages for Efficient Composition of Paths in 3D

Anton Bakker[1] and Tom Verhoeff[2]

[1]Norfolk, Virginia, USA; antonbakker.com, antonbakker30@gmail.com
[2]Department of Mathematics and Computer Science, Eindhoven University of Technology, Netherlands; T.Verhoeff@tue.nl

Abstract

We explore some domain-specific languages to describe (families of related) paths in 3D space. In particular, we explain the syntax, semantics, and implementation of Anton's Path Language. The syntax resembles that of file paths with wildcards, and the semantics of a path expression is the set of all paths that satisfy the constraints imposed by that expression. Anton's Path Language incorporates a mechanism to define patterns, based on repeated and transformed motifs. Through these patterns, various symmetries can be enforced. The implementation is an efficient engine running in Rhinoceros with GrassHopper, that quickly finds paths satisfying a given path expression and some global constraints. Anton's Path Language underlies a powerful toolbox for artists, who can express their ideas in a path expression, feed it into the engine, and then post-process this through some beauty filters or by visual inspection. We provide three example artworks and how they were discovered using this toolbox.

Introduction

Koos Verhoeff (1927–2018, [12]) is well known for his mathematical art involving piecewise linear paths in 3D space, realized by beams connected through miter joints (see Figure 1, left). It is straightforward to make the longitudinal beam edges connect properly across all miter joints in an open path, But in a *closed* path, the closing joint is in general problematic [14]. To ensure proper connection across *all* joints in a closed path, Koos invented various mathematical techniques [14][15]. His selection of 3D paths was mostly guided by his phenomenal geometric insight, which he had developed during his high-school years by solving geometry problems in bed in the dark in his head (his parents forbade the use of light after going to bed).

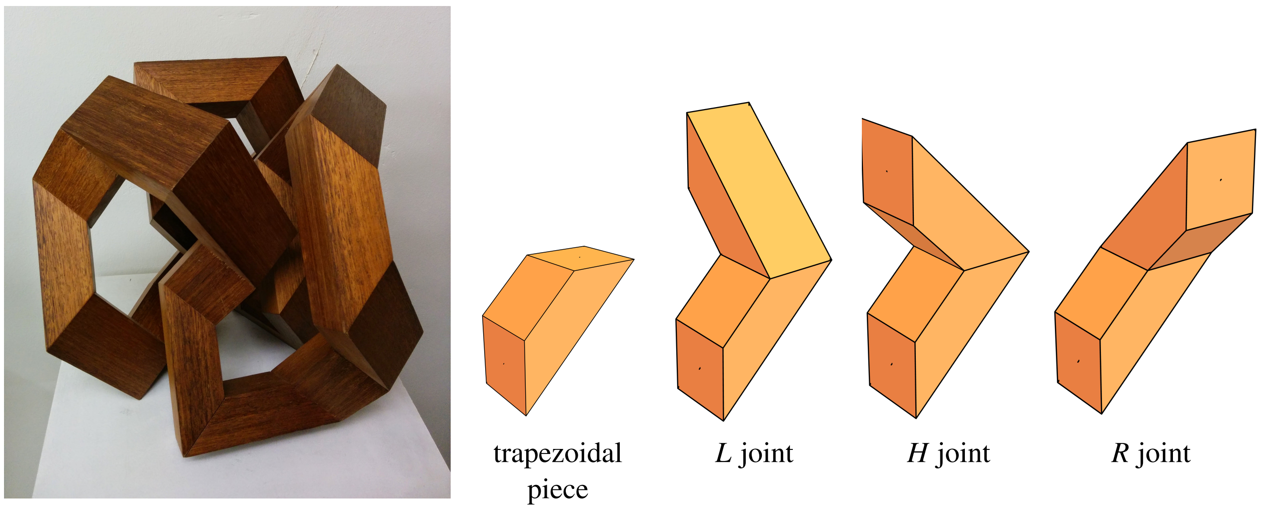

Figure 1: *Sculpture by Koos Verhoeff (left, wood, 40 cm) from 24 trapezoidal pieces with square cut faces.*

Anton (the first author) has collaborated with Koos for almost 40 years. Initially, this concerned the realization of designs by Koos. Later, Anton started exploring his own ideas. In particular, he moved away from beams and miter joints to curved renderings [4], focusing on the construction of 3D paths satisfying various constraints, such as symmetry and being knotted. Anton compares his approach to the composition of music: music for the eyes, based on repeated and transformed visual motifs. Anton, with help of Tom (the second author) and later also Dmitry Anikin, created a tool that evolved, over many years, into a powerful computational toolbox to explore customized search spaces for interesting paths.

Koos did occasionally use the computer to generate all possible closed paths using a particular building block [14]. He typically used plain backtracking followed by, partly automated, *beauty filtering*, as he called it. For example, the sculpture in Figure 1 was selected from all 62 882 closed paths with 24 copies of a trapezoidal piece (Figure 1), by filtering for order-3 rotational symmetry and lack of mirror symmetry. In this article, we focus on efficient computational techniques to compose interesting 3D paths, where some of the filtering is already handled in the generator. In the next section, we explain the various concepts behind the generation of 3D paths. Subsequent sections dive into Anton's Path Language, dealing with its syntax, semantics, implementation, and application.

Technical Preliminaries

A piecewise linear (geometric) path, from now on simply called a *path*, can be described by a sequence of waypoints, where each point is defined by its Cartesian coordinates. However, for many purposes this is an inconvenient description, since it is not invariant under rigid motions. Seymour Papert introduced a self-relative description format, called *Turtle Geometry* in [8], which was further developed in [1]. Tom investigated the generalization to 3D space and its application to mathematical art in [11].

In Turtle Geometry, a (virtual) turtle is commanded to draw a path. Thus, a path can be defined by a sequence of turtle commands, also known as a turtle program. At each moment, the turtle has a *state* captured by its *position* (a point) and its *attitude*. In 2D space, this attitude is fully defined by the turtle's *heading* (a vector). In 3D space, there is additionally the turtle's *normal* (another vector, perpendicular to its heading, defining its relative up direction). The plane perpendicular to the normal and containing the heading is called the turtle's *base plane*. The turtle commands are:

- $\mathit{Move}(d)$: move distance d forward in the direction of the current heading, changing only its position;
- $\mathit{Turn}(\varphi)$: turn (yaw) clockwise about the current normal by angle φ, changing only its heading;
- $\mathit{Roll}(\psi)$: roll clockwise about the current heading by angle ψ, changing only its normal (in 3D).

Turtle Geometry can be viewed as a special-purpose programming language, because it can only be used to compute paths. In contrast, a general-purpose programming language (GPL) can be used to solve arbitrary computational problems; a GPL is said to be universal or Turing complete. In the world of programming, it is nowadays common practice to solve a family of related programming problems through a domain-specific language (DSL). The DSL offers a notation that is easy to use for domain experts that are not programmers. A domain expert expresses the particular problem to be solved in the DSL, and the DSL's 'engine' (usually programmed in a GPL) then executes this DSL expression to obtain a solution.

Koos used Turtle Geometry extensively, and also invented further specializations, for instance, related to the trapezoidal piece of Figure 1 (cf. [17]). He described designs involving these pieces by recipes consisting of a string of symbols L (left), H (orthogonal), and R (right). These symbols correspond to the following turtle program fragments and to the three joints on the right in Figure 1:

$$\begin{aligned}
L &= \mathit{Move}(1); \mathit{Roll}(\psi); \mathit{Turn}(60^\circ); \mathit{Roll}(\psi - 180^\circ); \mathit{Move}(1) \\
H &= \mathit{Move}(1); \mathit{Roll}(90^\circ); \mathit{Turn}(90^\circ); \mathit{Roll}(-90^\circ); \mathit{Move}(1) \\
R &= \mathit{Move}(1); \mathit{Roll}(-\psi); \mathit{Turn}(-60^\circ); \mathit{Roll}(180^\circ - \psi); \mathit{Move}(1)
\end{aligned}$$

where $\psi = \arccos\sqrt{2/3} \approx 35.26°$. For example, the sculpture in Figure 1 is defined by the 24-symbol string $LHRLLRHLLHRLLRHLLHRLLRHL$. The regularity in this string can be made more explicit by allowing abbreviations and transformations in the language:

$$(Aa)^3 \text{ where } A = LHRL \text{ and } a \text{ is the reverse of } A$$

In [11][15][16][18] other specializations of Turtle Geometry are presented.

The relationship between the symmetries [5] of a path and the structure of its turtle program are investigated in [11]. For instance, a closed path has an order-n rotational symmetry when its program can be written as a sequence of n copies of the same program fragment. An inefficient way of generating paths with an order-n rotational symmetry would be to generate all paths and afterwards filter the ones with the desired symmetry. It is more efficient to customize the generator so that it only generates paths with the desired symmetry. One could construct a dedicated generator by hard-coding the restriction to order-n rotational symmetry, for a specific value of n. However, changing the generator for each new design is cumbersome.

In 2016, Anton got the idea for a generator where the built-in restrictions can be tweaked through input parameters that somehow describe the desirable patterns and possibly other constraints on the paths to be generated. He decided to focus on point lattices. Such lattices consist of points discretely and regularly spread throughout space. A lattice path can only connect nearest lattice points. We restrict ourselves here to the well-known cubic lattices [19]:

- simple cubic (SC), where each point has 6 neighbors,
- face-centered cubic (FCC), where each point has 12 neighbors, and
- body-centered cubic (BCC), where each point has 8 neighbors.

The sculpture in Figure 1 involves a path in FCC. Given a lattice, the options for paths are highly constrained. The goal is to generate paths that are unique (modulo rigid motions), closed, have no self intersections, and exhibit certain patterns and satisfy some additional selectable global constraints. The pattern can be used to enforce desired symmetries. The artist then chooses a particular pattern and sets constraints. When not satisfied with the generated result, the artist can adjust the input and try again. Note that there are also constraints that cannot be handled inside the generator, such as filtering for knots (considered out of scope).

The pattern input parameter is a path expression in what we will call *Anton's Path Language*. Such path expressions can be viewed as special turtle programs with *wildcards*, where the generator will try all possible values for the wildcards, while ensuring that the global constraints are met. These wildcards are similar to the asterisk wildcard in file paths (to match filenames) and the dot wildcard in regular expressions (to match strings). Global constraints that are supported in Anton's Path Language are: lattice type, joint constraints, maximum travel distance from the origin (useful when looking for knots), and avoidance of planar paths.

Anton's Path Language: Syntax

We first define the syntax of path expressions in a bottom-up fashion. A *token* is one of

- *wildcard*, which is one of `-`, `<`, `|`, `>`, `=`, and `_` (single lattice step, possibly angle constrained);
- single upper-case letter (refers to basic path defined earlier; see below);
- single lower-case letter (reversal of path defined by corresponding upper-case letter);
- single letter followed by `%` (reflection of path defined by that letter);
- single letter followed by a number (repeated path, all joined the same way);
- wildcard followed by a number, a dot, and another number (repeated and scaled path; use repeat count 1, if no repetition is desired).

A *basic path expression* is a sequence of tokens, not starting with <, |, >, =, or _. The language does not have parentheses for grouping. Instead one can define *abbreviations*, giving a single-letter name to a path:

- single upper-case letter followed by a colon (:) and a basic path expression.

An abbreviation must be defined before using it and cannot be recursive. A *full path expression* is a sequence of abbreviations followed by a basic path expression, known as *root*. An example of a full path expression is:

```
A:-|>
B:A>a>
B3
```

Its meaning will be explained in the next section. Further examples are given in the section on applications. Anton's Path Language has other features, but these fall outside the scope of this article.

Anton's Path Language: Semantics

The basic building block of a lattice path is a *step*, which is a directed edge (or arrow) between nearest lattice points. A *path* is a sequence of adjacent steps, connected head to tail, where all points, except possibly the first and last, are distinct (so there is no self intersection). When the first and last point are equal, the path is called *closed*; otherwise it is called *open*. Paths that can be transformed into each other by a rigid motion (i.e., an isometry) are considered equivalent, but keep in mind that direction matters.

We now define the semantics of path expressions, in a bottom-up and compositional fashion. Tokens other than wildcards and basic and full path expressions all have a semantics that is a set of paths. The semantics of a wildcard in isolation consists of just a single step, a path of length one, together with a possible *joint constraint*. Note that all paths of length one in a cubic lattice are equivalent. The difference in meaning between the various wildcards becomes clear when they are used in a basic path expression. The wildcard - has no joint constraint; <, |, >, and = "constrain the turn angle on the joint to the preceding step to be acute, right, obtuse, and straight respectively; _ constrains the joint to the preceding steps to be planar.

The semantics of an upper-case letter is the semantics of its corresponding basic path expression (see below). The semantics of a lower-case letter is the set of *reversed* paths for the corresponding upper-case letter. The semantics of a letter followed by % is the set of *reflected* paths for that letter. A letter followed by a number, say Ar, has (almost) the same semantics as A repeated r times, but there is an additional joint constraint (see below). Finally, a wildcard followed by a number, a dot, and a number, say $wr.s$ has the same semantics as w followed by $s-1$ copies of =, and this repeated r times; e.g., >3.2 is short for >=>=>=.

The semantics of a basic path expression, say consisting of the n tokens $t_1t_2 \dots t_n$, is the set of paths obtained by taking a path from the semantics of each token, say p_i for t_i, and joining those into a path $p = p_1p_2 \dots p_n$, heeding the following rules. (1) Paths p do not self-intersect. (2) Unless the basic path expression is the root, paths p are restricted to *open* paths. (3) Paths p_i and p_{i+1} $(1 \leq i < n)$ are to be joined in all possible ways, unless this freedom is restricted by a joint constraint. If t_{i+1} is a wildcard with joint constraint, then this constraint is applied. (4) Furthermore, if t_{i+1} came from a repeated letter (e.g., B3 being expanded to BBB), at position three or later, then there is no joint freedom and it is joined the same way as the preceding copies. (5) Paths that ultimately derive from the same abbreviation must all use the same instance.

The semantics of a full path expression is the set of all *closed* paths for its root. There are two properties that are easy to calculate for a path expression. The first is the *length* (total number of steps) of its paths: the sum of the lengths of its tokens, multiplied by any repetition and scaling factors. The second is how many *degrees of freedom* (DoF) it has, that is, how many independent opportunities for variation there are. This equals the total number of tokens minus the number of basic path expressions (its first step has no freedom), plus one for each token with a repeat count greater than one.

Let's see how the example path expression of the preceding section is handled in FCC (it yields no paths in SC because of >, and in BCC because of `|`). First, root `B3` means that all open paths are generated for `B` and 3 identical copies are joined together in the same way, where that way of joining varies over all possibilities. Since the path for `B3` must be closed, the third copy must end where the first starts (if not, then the this option is discarded). Moreover, by the *Looping Theorem* [13], the joint between the third and first copy will then also be the same as the other `BB` joints. Next, consider how open paths for `B:A>a>` are generated. For this, all open paths for `A` are generated, and each such path is joined in all possible ways to an obtuse step, a reversed copy of `A`'s path, and another obtuse step. Finally, to generate open paths for `A:-|>`, a first step is done (fixed), the next step must make a right angle with the first, and the final step an obtuse angle with the second. This example expression generates closed paths with an order-3 rotational symmetry consisting of 24 steps. It turns out that there are 360 such paths. One of them is the sculpture in Figure 1.

Anton's Path Language: Implementation

Anton's Path Language is embedded in Rhino via GrassHopper, through the GhPython script component. The actual generator is written in Cython (closed source), which is a Python superset compiled to C. The generator uses backtracking to find all paths (modulo rigid motions), but it prunes as early as possible when global constraints (such as being closed, free of self-intersection, and travel distance limit) are violated.

To get an impression of performance, consider the earlier example, in FCC, describing paths of length 24. Plain backtracking would consider on the order of $11^{23} \approx 10^{24}$ paths, because each step, except the first, can choose from 11 neighbors in FCC. The example path expression, however, has fewer degrees of freedom (DoF), viz. six (2 in `A`, 3 in `B`, and 1 in `B3`). In FCC a right-angle step (`|`) has 2 possibilities and an obtuse-angle step (>) has 5 possibilities (see supplementary material for more information). Each unconstrained joint before an abbreviation or inside a repeated token adds a factor 11. So, the total (worst-case) search space is now only $\approx 2\times5\times11^4 = 146\,410$. The generator actually finds 360 FCC paths for this expression in 0.8 s on a 10-core 2021 M1 MacBook Pro with Rhino 7 (all times reported below concern this configuration). The path expression consisting of 10 unconstrained steps in FCC, produces 47 440 (unique closed non-self-intersecting) paths in 26.4 seconds.

Anton's Path Language: Applications

We present some details for the (re)discovery of three artworks through the application of Anton's Path Language. How the designs are dressed up (profiled and rendered with material properties) is beyond the scope of this article (cf. [4]). Also see [2] for more artworks by Anton.

Opus 951465 — A knot

The search for Opus 951465 starts in the SC lattice with the following path expression:

```
A:--1.5-1.6-1.6-1.3
B:AA%
B3
```

This expression describes paths with $(1 + 5 + 6 + 6 + 3) \times 2 \times 3 = 126$ steps and 6 DoF (search space size $5^6 = 15\,625$). These paths have an order-3 rotational symmetry (because of `B3`), and an order-6 roto-reflection symmetry (because of `B:AA%`). Without a travel distance limit, the generator yields 521 paths in 7 s, and 72 paths in less than 1 s when the travel distance limit is set to 16%.

Next, Anton's custom suspected-knot beauty filter (implemented in GrassHopper), finds 17 candidates among 521 (15 among 72). Finally, Anton visually inspected these candidates and preferred the design in Figure 2. Koos discovered this design in the 1990s, through geometric insight (cf. [6, Figures 4 and 5]).

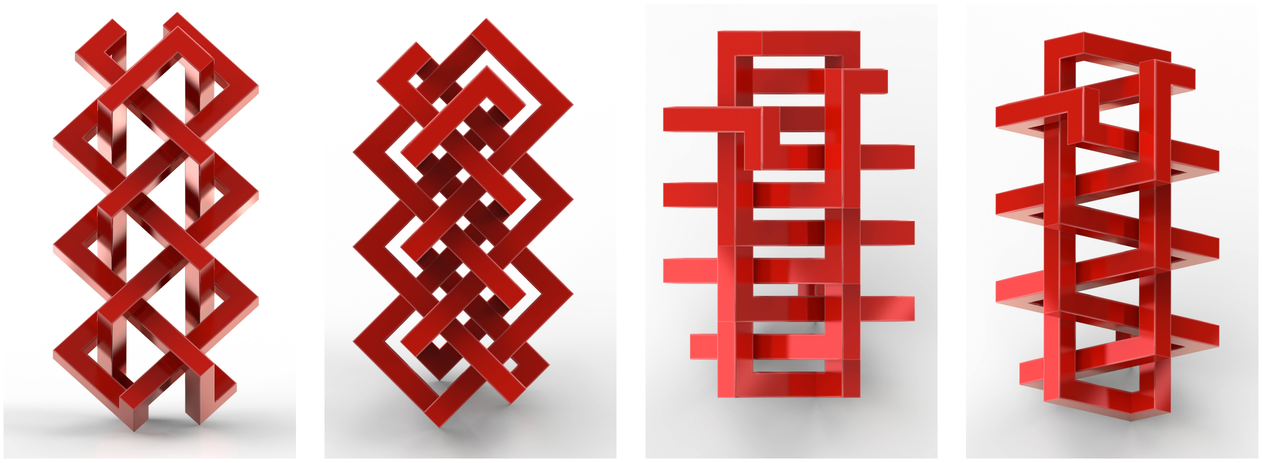

Figure 2: *Opus 951465, four perspectives.*

Opus 185131 – A spiral

The search for Opus 185131 starts in the FCC lattice with the following path expression:

```
A:--1.2_1.2_1.3_1.3_1.3_1.4_1.4_1.4_1.5
AA%
```

This expression describes paths with $(1+2+2+3+3+3+4+4+4+5)\times 2 = 62$ steps and 10 DoF (search space size $11 \times 3^8 \times 11 = 793\,881$). The scaling factors and planar joint constraints (_) in named path `A` encourage the creation of planar spirals. This spiral is then joined to a reflected copy (`AA%`), so that the final design lies in two planes. With the travel distance limit set to 13%, the generator yields 4 321 paths in 25.6 s.

The suspected-knot beauty filter identifies 46 candidates. After visual inspection, Anton preferred the design in Figure 3, which is actually a trefoil knot.

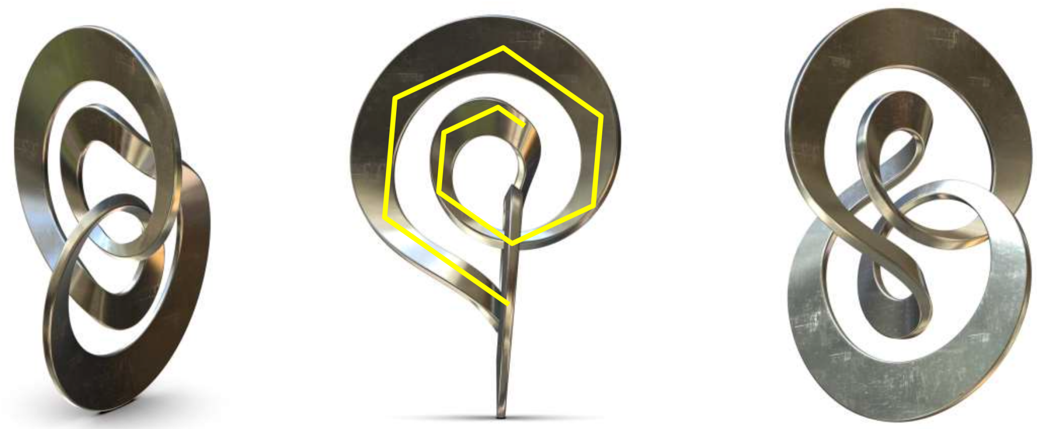

Figure 3: *Opus 185131: three perspectives; middle: plus overlay of FCC spiral* `A`.

Opus 125707 — An optical illusion

The goal behind this design (also see [3][7]) is that the path must have 2D projections that seem topologically contradictory. The search takes place in the BCC lattice with the following path expression

```
A:-8
B:Aa
BB
```

This expression describes paths with $8\times2\times2 = 32$ steps and 9 DoF (search space size $7^7\times7\times7 = 40\,353\,607$). With the travel distance limit set to 13%, the generator yields 39 521 paths in 108 s.

Next the suspected-knot beauty filter reduces the number of candidates (exact number was not recorded). And then another automated custom beauty filter considers 2D projections in various directions, comparing these topologically for crossing patterns, looking for paths where these 2D projections differ 'significantly' (details omitted). Finally, by visual inspection, Anton picked the design in Figure 4, where you can see how these three 2D projections are topologically contradictory: on the left, one clearly sees two linked unknots; in the middle, a single knot; on the right, two linked 8-shaped figures.

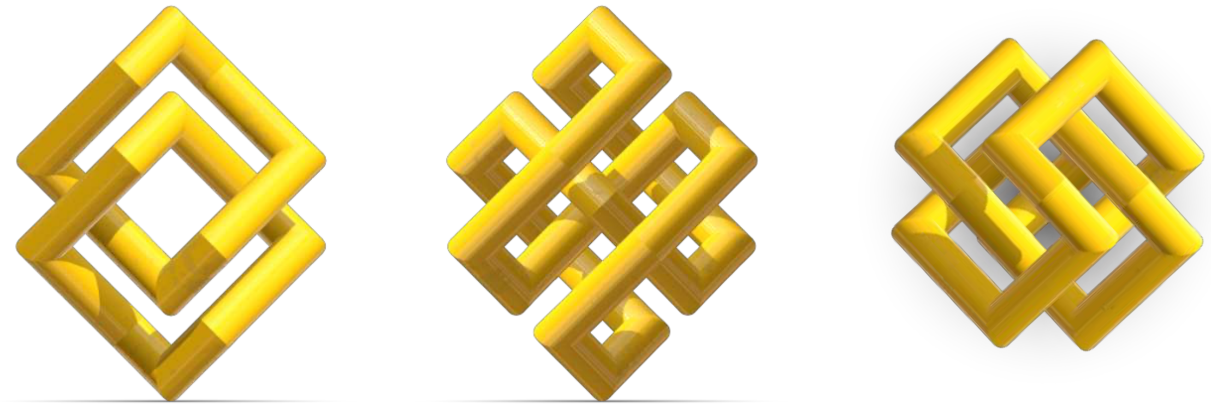

Figure 4: *Opus 125707: three perspectives.*

Conclusion

We described Anton's Path Language, which is like a regular expression language for 3D paths to help designers do focused searches for interesting paths. Use of this language leads to huge time savings when exploring design alternatives. The language still evolves and has been extended with numerous features. One of these features is support for generating (boundaries of) 2D tiles that satisfy the Conway criteria [9], so that they can yield Escher-like tessellations. Another extension involves loops and branching (also see branching in the language of [18]). Moreover, additional beauty filters have been implemented, for instance, to help find all possibilities for linking multiple copies of the same path.

It must be emphasized that Anton uses his Path Language merely as a tool in the search for sculptures that fit his philosophy. His artwork is intended to loosen our grip on a singular reality and bridge our differences of perspective. Viewing his sculptures in the round is intrinsic to the meaning of his art.

It would be interesting to consider language extensions that go outside lattices, as in [16][10]. A completely different angle is to explore the similarity with music composition, where a melody is like a path in tonal space, constructed from motifs with transformations like time reversal, pitch-interval reflection, and time scaling.

Acknowledgements

We would like to thank Dmitry Anikin for work on the implementation of the generator for Anton's Path Language and Sten Wessel for helping investigate its semantics.

References

[1] H. Abelson and A. diSessa. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics*. Cambridge, MA: MIT Press, 1980. https://mitpress.mit.edu/9780262510370/turtle-geometry

[2] A. Bakker. *Virtual MoMath Exhibition: Alternative Perspective*. 2020. https://www.antonbakker.com/momath (Accessed 25 Apr. 2023).

[3] A. Bakker. *Opus 125707, Two Squares Optical Illusion*. Bridges 2023 Halifax Art Exhibition Catalog, https://gallery.bridgesmathart.org/exhibitions/2023-bridges-conference/anton-bakker

[4] A. Bakker and T. Verhoeff. "Artistic Rendering of Curves via Lattice Paths." *Bridges Conference Proceedings*, Waterloo, Ontario, Canada, July 27–31, 2017, pp. 447–450. http://archive.bridgesmathart.org/2017/bridges2017-447.html

[5] J. Conway, H. Burgiel, and C. Goodman-Strauss. *The Symmetries of Things*. CRC Press, 2008. DOI: https://doi.org/10.1201/b21368

[6] B. Ernst and R. Roelofs, Eds. *Koos Verhoeff – Jaarboek*. Ars et Mathesis, 2013.

[7] S. Macknik and S. Martinez-Conde. *Scientific American Mind*, Vol. 31, No. 6, Nov. 2020, pp. 32-33. https://doi.org/10.1038/scientificamericanmind1120-32

[8] S. Papert. *Mindstorms: Children, Computers, and Powerful Ideas*, 1st ed. Basic Books, 1980.

[9] D. Schattschneider. "Will It Tile? Try the Conway Criterion!" *Mathematics Magazine*, vol. 53, no. 4, 1980, pp. 224–233.

[10] M. van Veenendaal and T. Verhoeff. "Pretty 3D Polygons: Exploration and Proofs." *Bridges Conference Proceedings*, Virtual, Aug. 1–3, 2021, pp. 111–118. http://archive.bridgesmathart.org/2021/bridges2021-111.html

[11] T. Verhoeff. "3D Turtle Geometry: Artwork, Theory, Program Equivalence and Symmetry." *Int. J. of Arts and Technology*, vol. 3, no. 2/3, 2010, pp. 288–319. DOI: http://dx.doi.org/10.1504/IJART.2010.032569

[12] T. Verhoeff. "Some Memories of Koos Verhoeff (1927–2018)." *Bridges Conference Proceedings*, Stockholm, Sweden, July 25–29, 2018, pp. 3–6. http://archive.bridgesmathart.org/2018/bridges2018-3.html

[13] T. Verhoeff. "The Looping Theorem in 3D Turtle Geometry." *Bridges Conference Proceedings*, Halifax, Nova Scotia, Canada, July 27–31, 2023.

[14] T. Verhoeff and K. Verhoeff. "The Mathematics of Mitering and its Artful Application." *Bridges Conference Proceedings*, Leeuwarden, the Netherlands, July 24–29, 2008, pp. 225–234. http://archive.bridgesmathart.org/2008/bridges2008-225.html

[15] T. Verhoeff and K. Verhoeff. "Regular 3D Polygonal Circuits of Constant Torsion." *Bridges Conference Proceedings*. Banff, Canada, July 26–30, 2009, pp. 223–230. http://archive.bridgesmathart.org/2009/bridges2009-223.html

[16] T. Verhoeff and K. Verhoeff. "Lobke, and Other Constructions from Conical Segments." *Bridges Conference Proceedings*. Seoul, Korea, Aug. 14–19, 2014, pp. 309–316. http://archive.bridgesmathart.org/2014/bridges2014-309.html

[17] T. Verhoeff and K. Verhoeff. "Hopeless Love and Other Lattice Walks." *Bridges Conference Proceedings*. Waterloo, Ontario, Canada, July 27–31, 2017, pp. 197–204. http://archive.bridgesmathart.org/2017/bridges2017-197.html

[18] T. Verhoeff and K. Verhoeff. "The Obtetrahedrille as a Modular Building Block for 3D Mathematical Art." *Bridges Conference Proceedings*, Linz, Austria, July 16–20, 2019, pp. 407–410. http://archive.bridgesmathart.org/2019/bridges2019-407.html

[19] E.W. Weisstein. "Cubic Lattice." From *MathWorld–A Wolfram Web Resource*. https://mathworld.wolfram.com/CubicLattice.html