Project Pine: An Exploration of Algorithm-Driven Solid Generation

Alexander Ozer
thealexozer@gmail.com

1 Introduction

Computer Aided Design (CAD) is a widely used system for modeling three-dimensional objects in preparation for manufacturing. The advent of rapid-prototyping systems, such as 3D-printing, allows designs made in CAD software to be transformed from a digital format to a physical object in a robust, inexpensive manner. However, the immensely broad spectrum of objects 3D-printers are capable of producing is not reflected in the abilities of standard CAD software. In such software, only a limited number of tools are available to manipulate a 3D model, and each tool itself has finite capability. For instance, while an "Extrude" tool is capable of extending any arbitrary, two-dimensional shape out of its plane into a 3D prism, this tool may only be used manually and in finite increments. Each extrusion requires the user's manual activation of the tool. The same methodology applies to using tools such as Revolve, Cut, and Pattern. Although the multitude and variety of tools available in CAD software allows the user to designs a vast array of objects, not all objects can be efficiently modeled using this approach.

Generating solids with a significant degree of complexity, most notably fractals, is notoriously difficult using CAD programs. A fractal is a pattern that replicates itself within itself indefinitely. To design a fractal in traditional CAD software, an initial pattern must be designed and then replicated many times, scaling it down and repatterning as needed. However, this method is both repetitive and inflexible; software itself could easily perform this repetition, and if the initial pattern must be altered, the entire design must be remodeled. Through Project Pine, I attempt to bridge the gap between the mathematical realm of fractals and the physical realm. In this paper, I recount the development of a method of fractal solid generation, its implementation as the computer program *Pine*, and its results.

2 Programming Language Selection

In writing a computer program to generate 3D-printable fractals, the programming language to use must be considered. The selected language should have an efficient runtime, support concurrency, and support object-oriented programming. Prior to writing code for Pine, my experience was limited to the Java programming language. Despite its popularity and flexibility, Java is not compiled directly to Assembly language, meaning its runtime performance is hampered. Instead of using Java, I instead opted to learn a new language entirely.

Golang, or Go for short, is a new language developed by Google. It supports direct compilation to Assembly and features seamless support for concurrency, meaning complex fractals can be generated very quickly. Its learning curve is significantly less steep than that of other compiled

languages, such as C or C++, and its syntax feels like a dynamically-typed language even though it is statically typed. This allows efficient programs to be written with extensive functionality in little code. Therefore, I decided to learn it and use it to write Pine.

3 3D Formatting

Almost all 3D objects are represented by a mesh of triangular faces joined at their seams to make a bounding surface. CAD software exports models to triangle meshes, and common 3D-printing services, such as Shapeways.com, accept models in the form of triangle meshes. Part of the fundamental restriction of CAD programs is that they do not allow users to directly generate triangles; only their finite toolset can do so, and only in restrictive ways. Pine attempts to overcome this restriction, and thus must be capable of directly manipulating raw meshes of triangles.

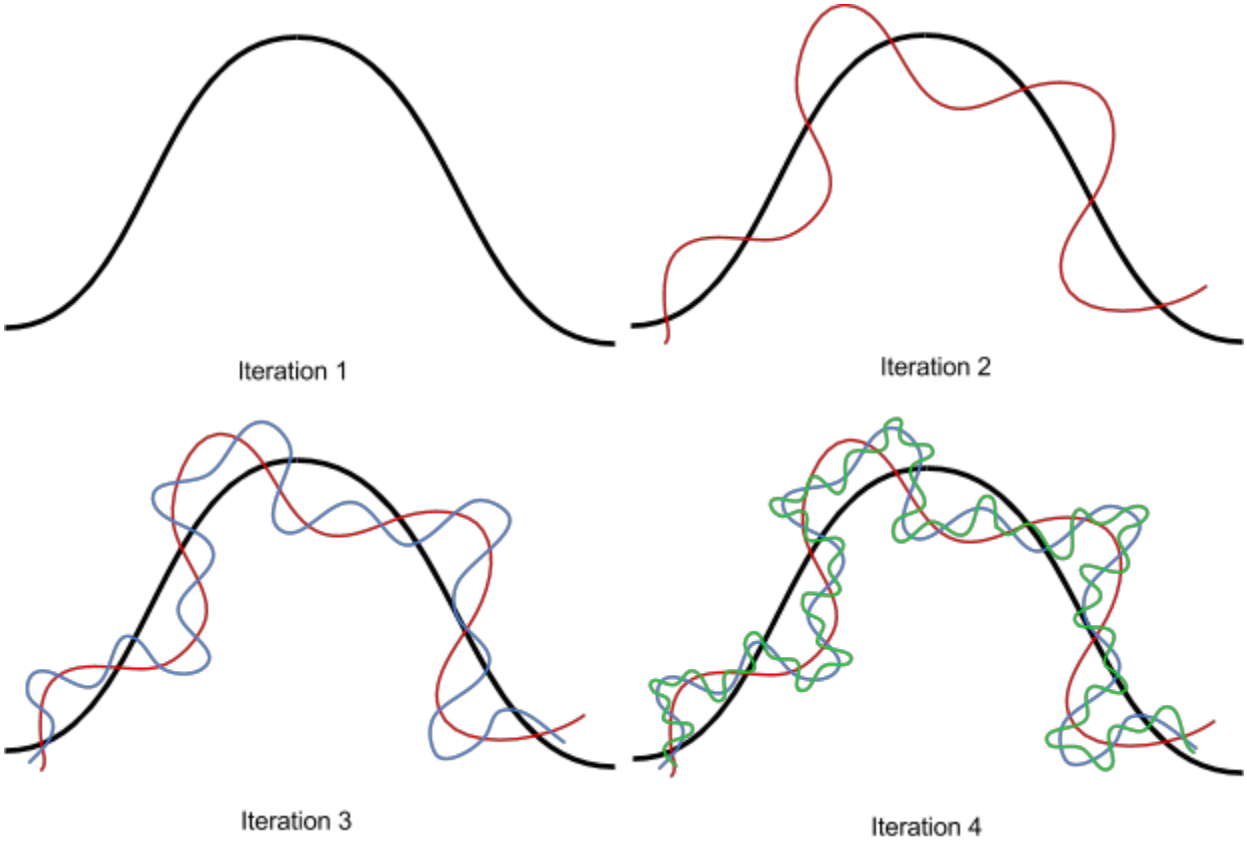
To work with triangle meshes, Pine must first be able to write to a file format which 3D-printing services are capable of handling. The most common format is known as STL, short for stereolithography. STL files contain a header section followed by a list of floating-point numbers. Each group of three floats represents a 3D-coordinate, and every three coordinates represents the vertices of a triangle. Because Pine works with triangles as a whole and not individual floats, it must streamline the conversion to this format.

The portion of Pine dedicated to exporting internal triangle data to the STL format is titled *go-mesh* and is available in my Github repository, located at *github.com/alexozer/go-mesh*.

4 Generator Design

Now that Pine can export triangle meshes to a format which may be 3D-printed, the problem of developing a system of generating the triangles arises. With such a wide range of possibilities that can stem from this, I decided to remain focused on one application while simultaneously generalizing my solution so that it may apply to many other applications.

I developed Pine's initial system with the goal of of generating a specific fractal I call a *helical fractal*. It is composed of a helical curve which is recursively rewrapped in helical curves. Below is a two-dimensional equivalent. The first iteration is merely the initial curve, the second iteration (marked in red) "wraps around" the first curve, the third iteration (marked in blue) wraps around the second curve, and the fourth iteration (marked in green) wraps around the third curve.



This fractal serves as a simple starting point because each iteration forms a single curve which does not self-intersect.

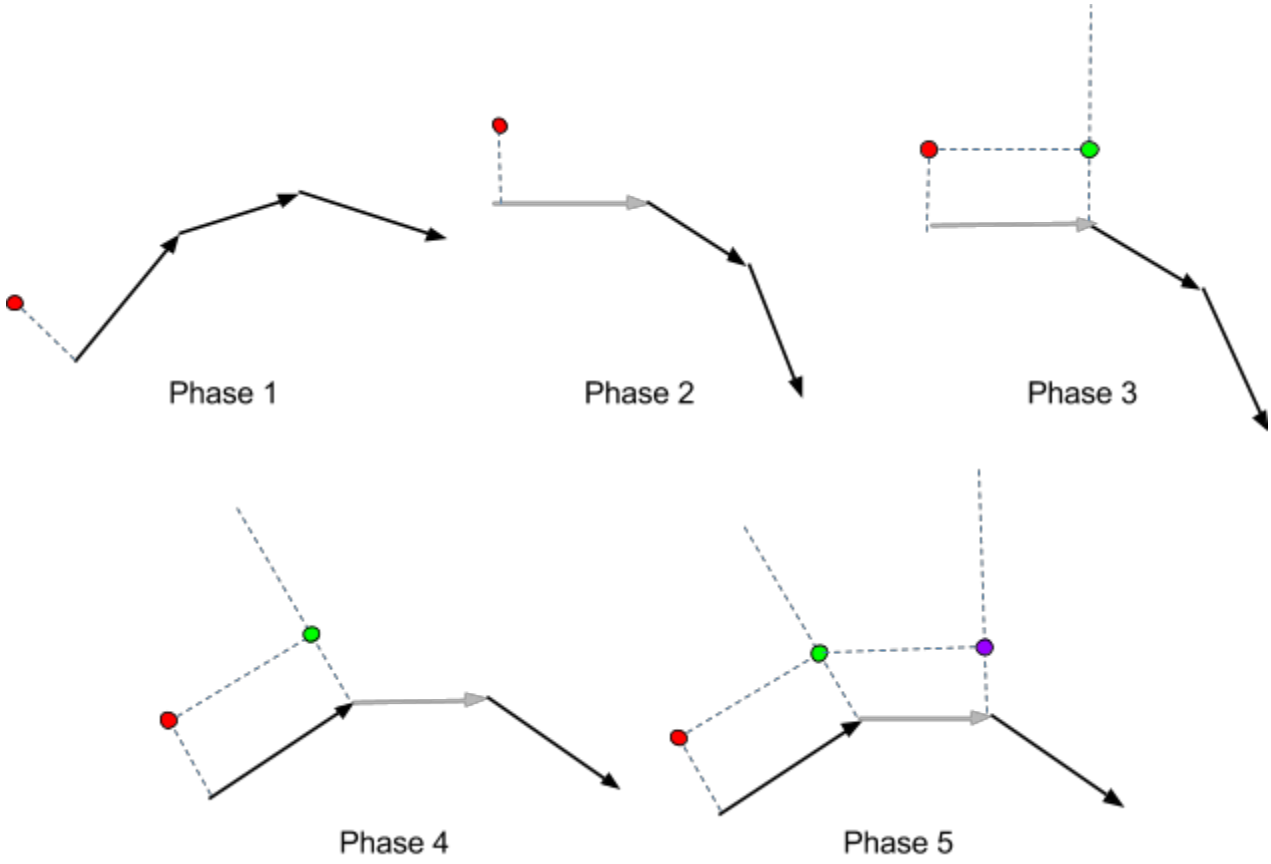
There are two key components of the helical fractal which Pine must handle to generate it: the "backbone" curve, and a 3D tube of triangles which wraps around the backbone. Without the 3D tube, the resultant model would only be an infinitely-thin curve which cannot be 3D-printed. The 3D tube exists in fragments because it is composed of triangles; therefore, it is not required for the backbone curve to be continuous; it may be composed of distinct line segments. This is precisely how Pine represents the curvature of the backbones.

Pine's source code defines objects with a specific *type*; each type serves a role in generating the fractal, and can be repurposed to build other arbitrary designs. The types involved in generating the helical fractal's backbone are named *CurveMaker* and *CurveTaker*. A CurveMaker object outputs a stream of connected vertices which form a backbone curve, whereas a CurveTaker receives a segmented curve from a CurveMaker. Some objects could be both a CurveTaker and a CurveMaker.

Pine uses two implementations of CurveMaker and CurveTaker to generate the backbone of the helical fractal: *Line* and *Helix*. Line is a CurveMaker and outputs a series of collinear line segments. Helix is a CurveTaker and a CurveMaker; it receives line segments from another CurveMaker and outputs line segments that form a helix about the inputted segments. The beauty of this system is that each individual Helix does not know exactly what it is generating itself

about; this decouples each iteration of the fractal from each other, which strongly aligns with the mathematical nature of fractals themselves.

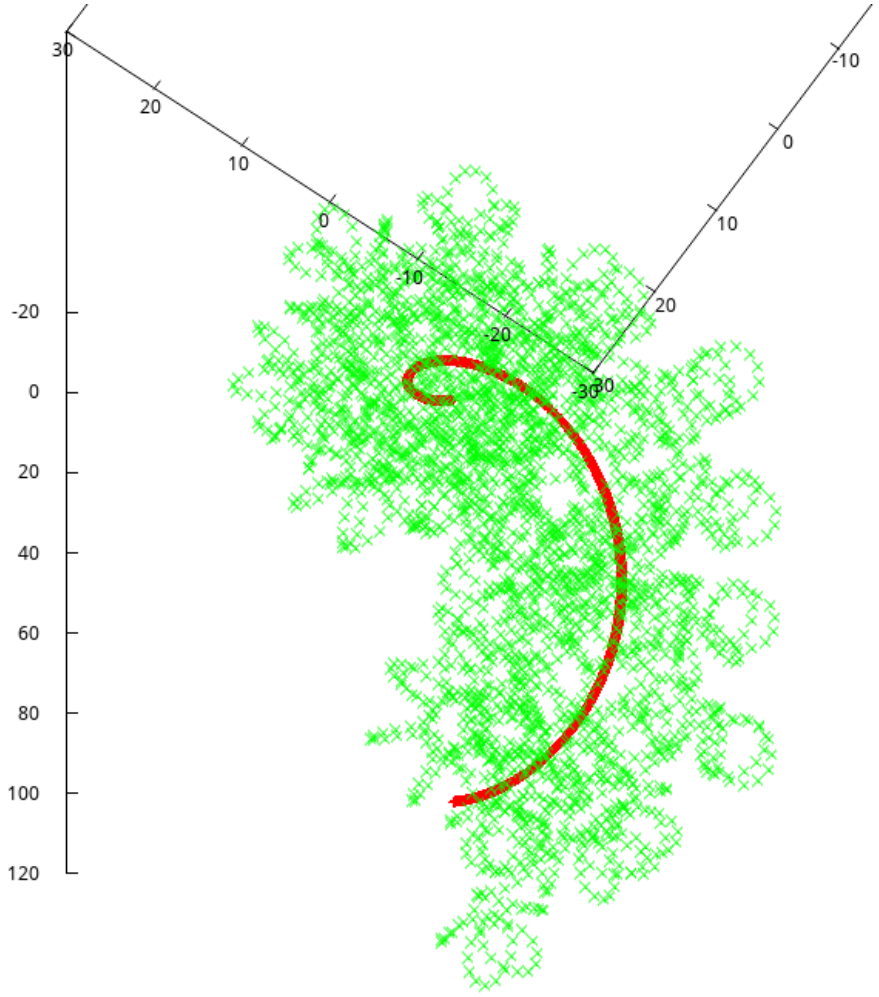
Generating a helical curve about a Line parallel to an axis is simple. If a Line parallel to the Y-axis, then as line segments progressively more negative in the Y direction are passed to a Helix, the Helix simply outputs line segments that connect vertices with decreasing Y-coordinates, equal distance from the Line, and increasing offset angle. However, because a Helix object may read a curve from another Helix object and not a Line, it cannot assume that its input curve is linear. Therefore, the Helix object's generating algorithm must be more involved. Pine implements Helix's curve generation algorithm using spatial reorientation. The general method is outlined below.



The black arrows represent the line segments received by a Helix object. The Helix object first plots an initial points (shown in red) perpendicular to the first received segment. In the second phase, the most recently received line is aligned with a coordinate axis; in this 2D representation, the first arrow (now grey) becomes aligned with the X-axis. In the third phase, the Helix object places a point on the plane which the oriented line segment is normal to (shown in green). In 3D, this point would be rotated about the oriented line segment a bit; this creates the curvature of the outputted helix. In phases four and five, the process is repeated using the next available line segment and the most recently placed point. In 3D, it is essential that the last point (green) is used to align the new point (purple) on the plane because the last point's rotation is used as a reference for the new point; the new point is rotated in its plane slightly more than the projected

last point would appear to be. This isn't pictured in these 2D diagrams. The entire process continues until the Helix object reaches the end of the output from the CurveMaker object it was reading from.

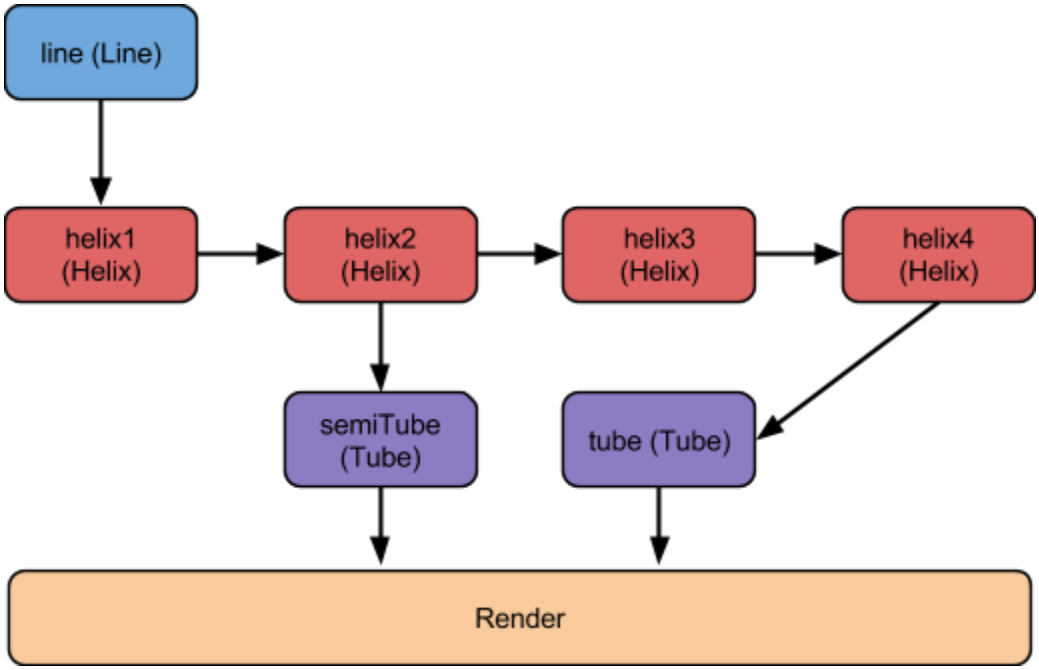
This algorithm is quite effective. Using it, Pine can successfully generate the helix fractal's backbone. The following is a plot of points generated using Pine's implementation of the algorithm.



The red curve represents a first-iteration Helix object that simple revolves around a Line object. The green points are the output of a much higher-order Helix object, e.g. one that has generated itself around the output of many more sequential Helix objects.

After the helical fractal backbone is generated, it must be wrapped with a tube of triangles to give it volume. It is quite simple for Pine to accomplish this by aligning rings of triangles with each line segment of the final Helix object's output. A *Tube* object implements this.

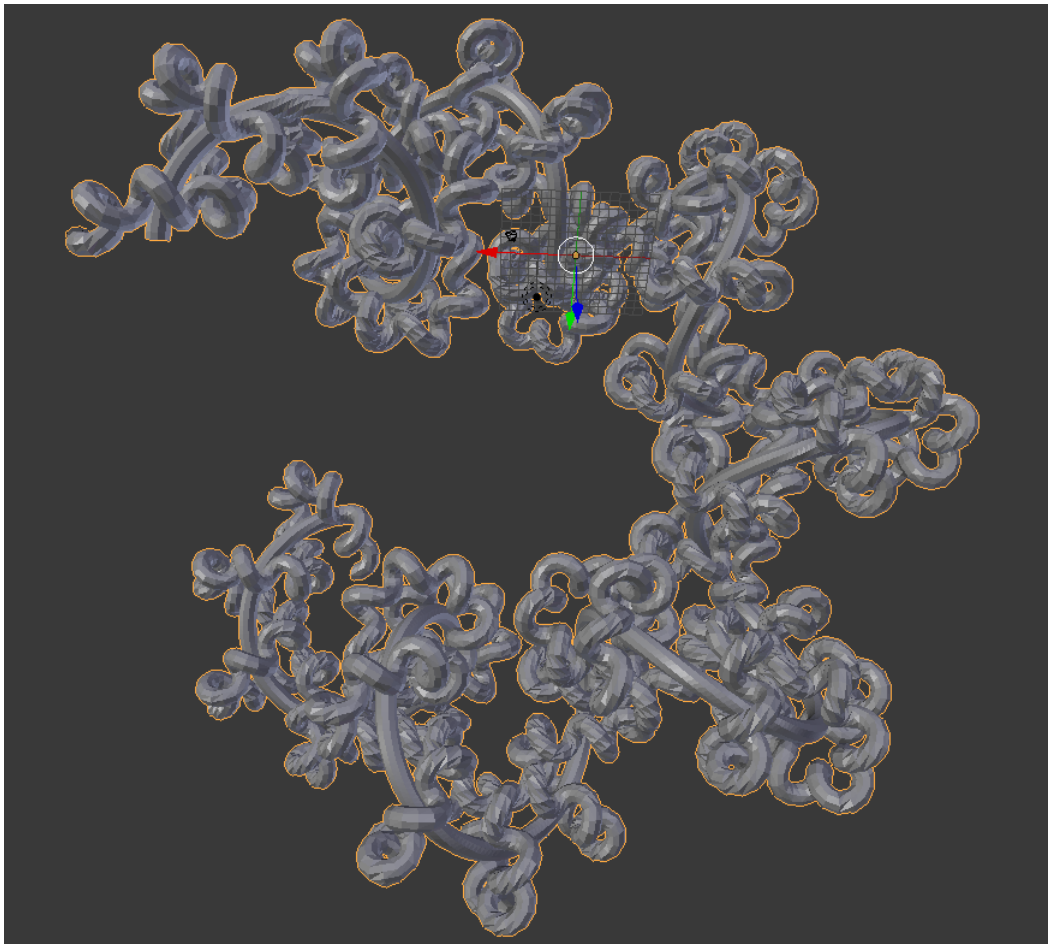
Line, Helix, and Tube objects abstract away the raw generation of triangles to a set of modular, "fractal building blocks". The diagram below outlines the hierarchy of the generation of a helical fractal containing two tubes.



Because the generation of a final triangle mesh has been abstracted into such simple steps, the code required to generate the final model is very readable and self-documenting.

```
line := Line{
    First:    vec3.T{0, 0, 0},
    Final:    vec3.T{0, 50, 0},
    Segments: 3750,
}

helix1 := Helix{
    Radius: 25,
    Pitch:  50,
}
helix1.SetParent(&line)

helix2 := Helix{Radius: 10, Pitch: 25}
helix2.SetParent(&helix1)

helix3 := Helix{Radius: 5, Pitch: 10}
helix3.SetParent(&helix2)
```

```
helix4 := Helix{Radius: 1.5, Pitch: 5}
helix4.SetParent(&helix3)

tube := Tube{Radius: 1, Sides: 10}
tube.SetParent(&helix4)

semiTube := Tube{Radius: 1, Sides: 10}
semiTube.SetParent(&helix2)

mesh := MeshRender{}
mesh.SetParent(&tube)

buffer := mesh.Render()

mesh.SetParent(&semiTube)
line.Segments = 1000
buffer2 := mesh.Render()
```

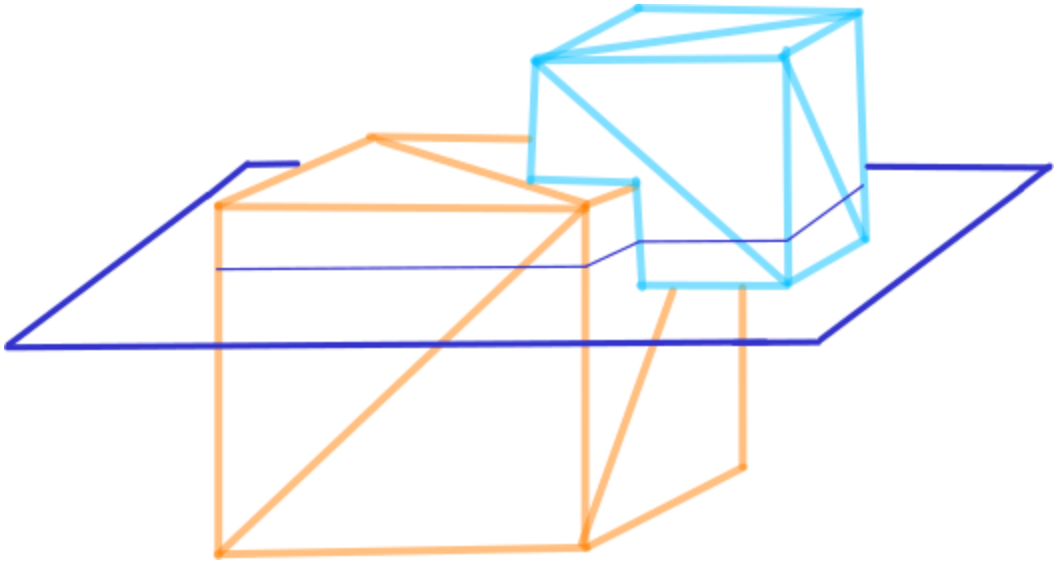


This is the result after Pine generates the solid, exports it to a STL file, and the file is imported into a triangle mesh viewer. Clearly, the model is quite complex and beyond the reasonable capabilities of typical CAD software. The model is ready to 3D-print.

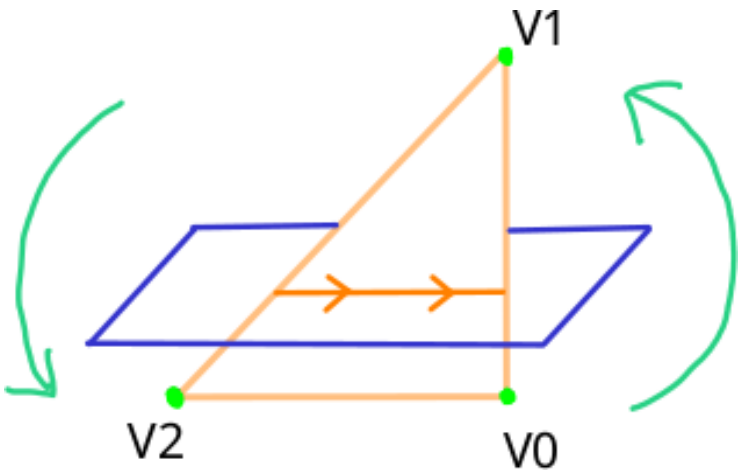
5 Intersecting Meshes

Although Pine's modular mesh generator can already create fairly complex designs, it is incapable of handing self-intersecting meshes. Most 3D-printing services require user-submitted triangle meshes to contain no intersecting triangles. It is possible to repair intersecting meshes through retriangulation, but this is complicated to implement in a robust manner. Instead, I propose a system for converting intersecting triangle meshes directly to voxels (3D pixels), a format also accepted by most 3D-printing services.

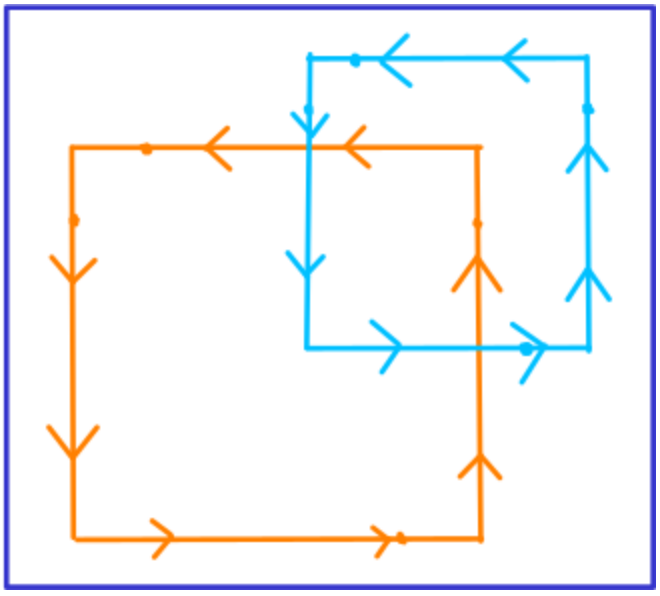
A 3D grid of voxels is represented by a group of images where each represents a cross-section of the mesh at a particular point. The horizontal and vertical position of each pixel in each image defines the remaining two coordinates of each voxel. The image "slices" are obtained by intersecting the mesh with a plane at each elevation required.



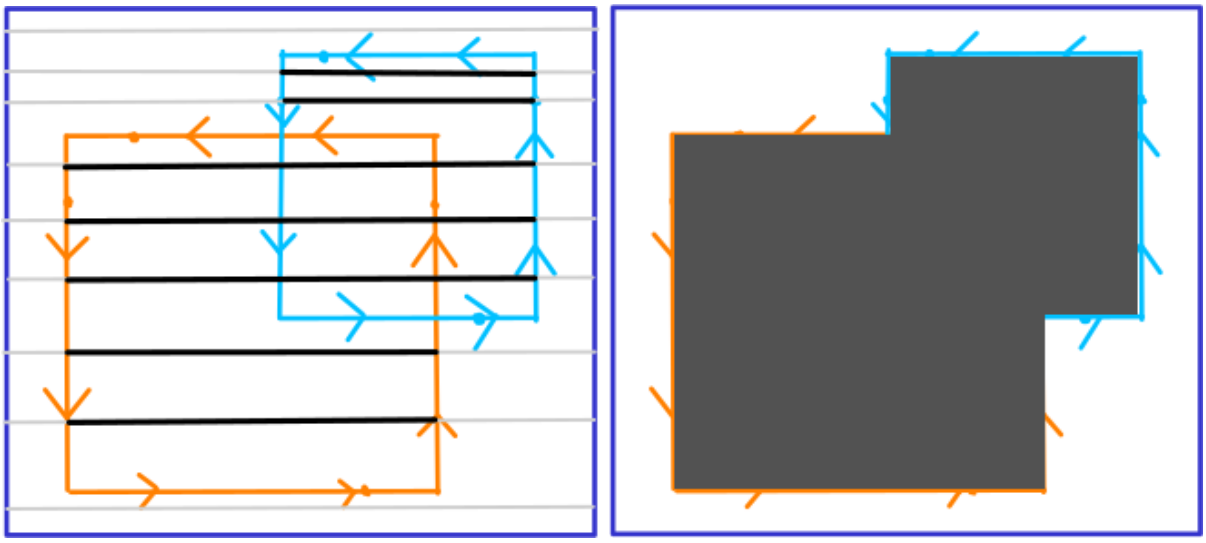
Each intersection line on the plane is assigned a direction based on the order of its vertices. If the order appears clockwise facing the viewer, a right-directed line is drawn on the plane.



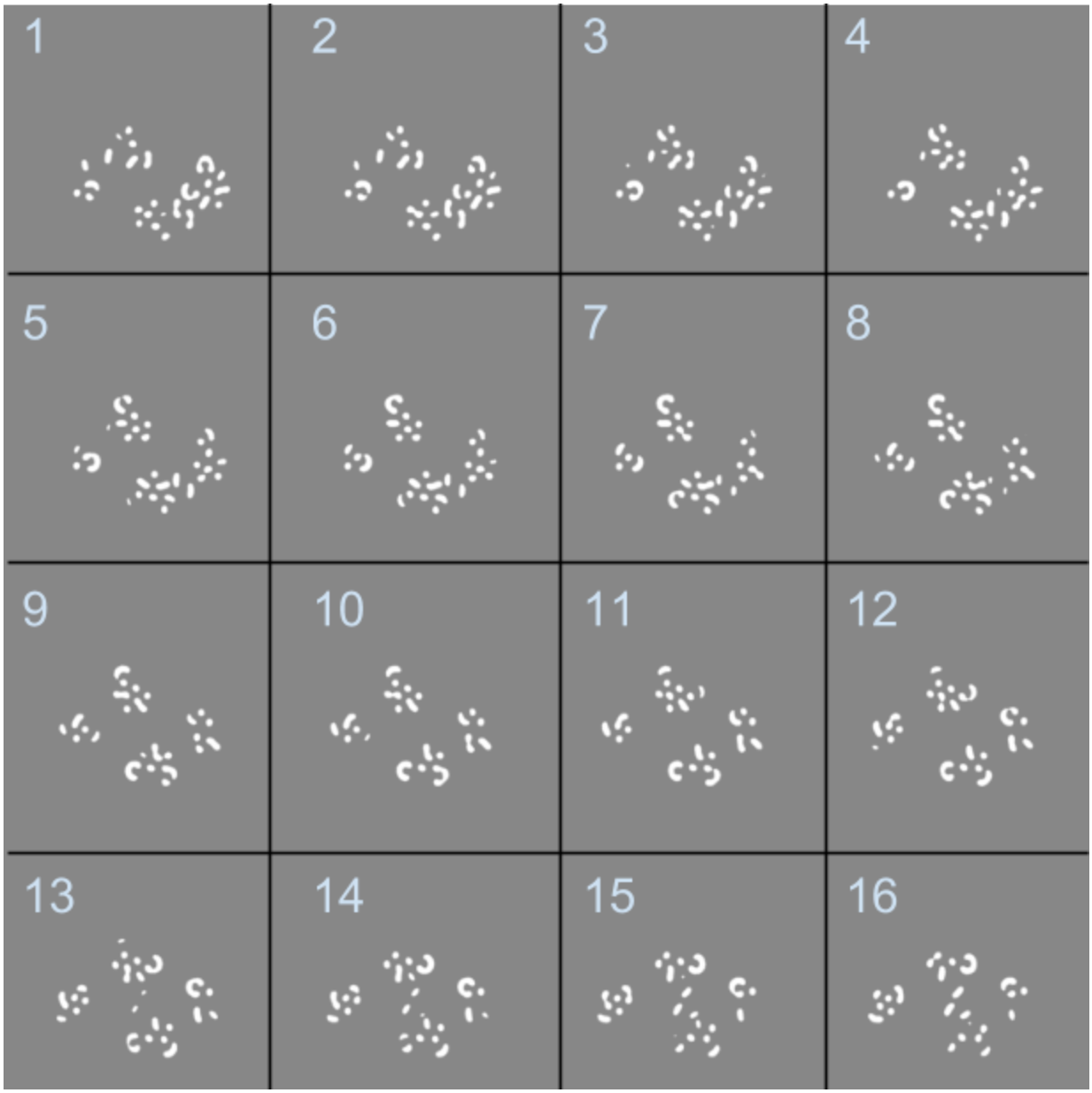
The resulting intersection pattern contains counter-clockwise loops of intersection lines.



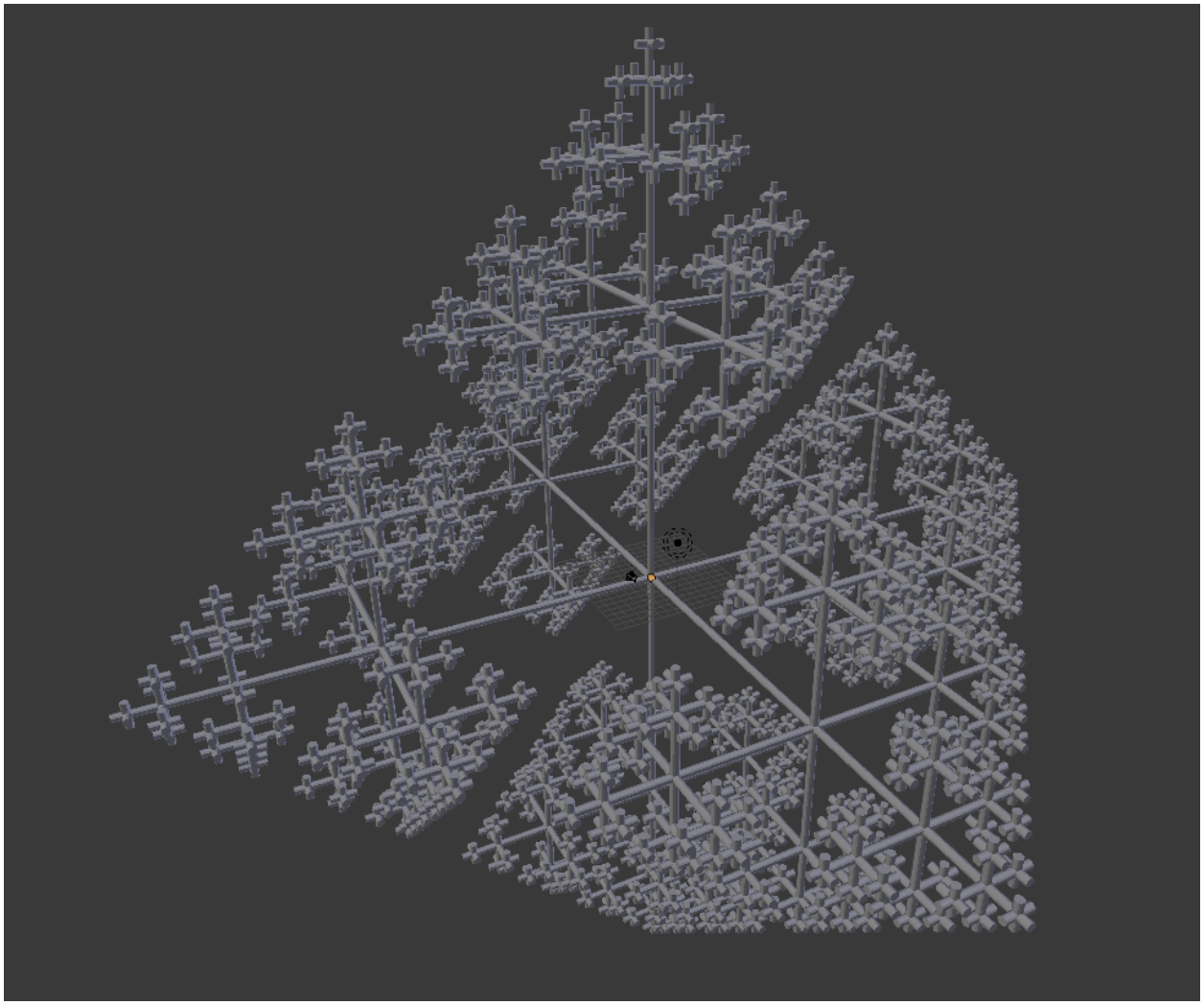
The pattern is then scan-converted into pixels. As each scanning segment (shown in grey and black) approaches from the left, for every down-facing arrow it crosses, its “depth” is increased, and for every up arrow it crosses, its “depth” is decreased. If the depth is given to be zero outside all loops, for every region where the depth is greater than zero, pixels are drawn.



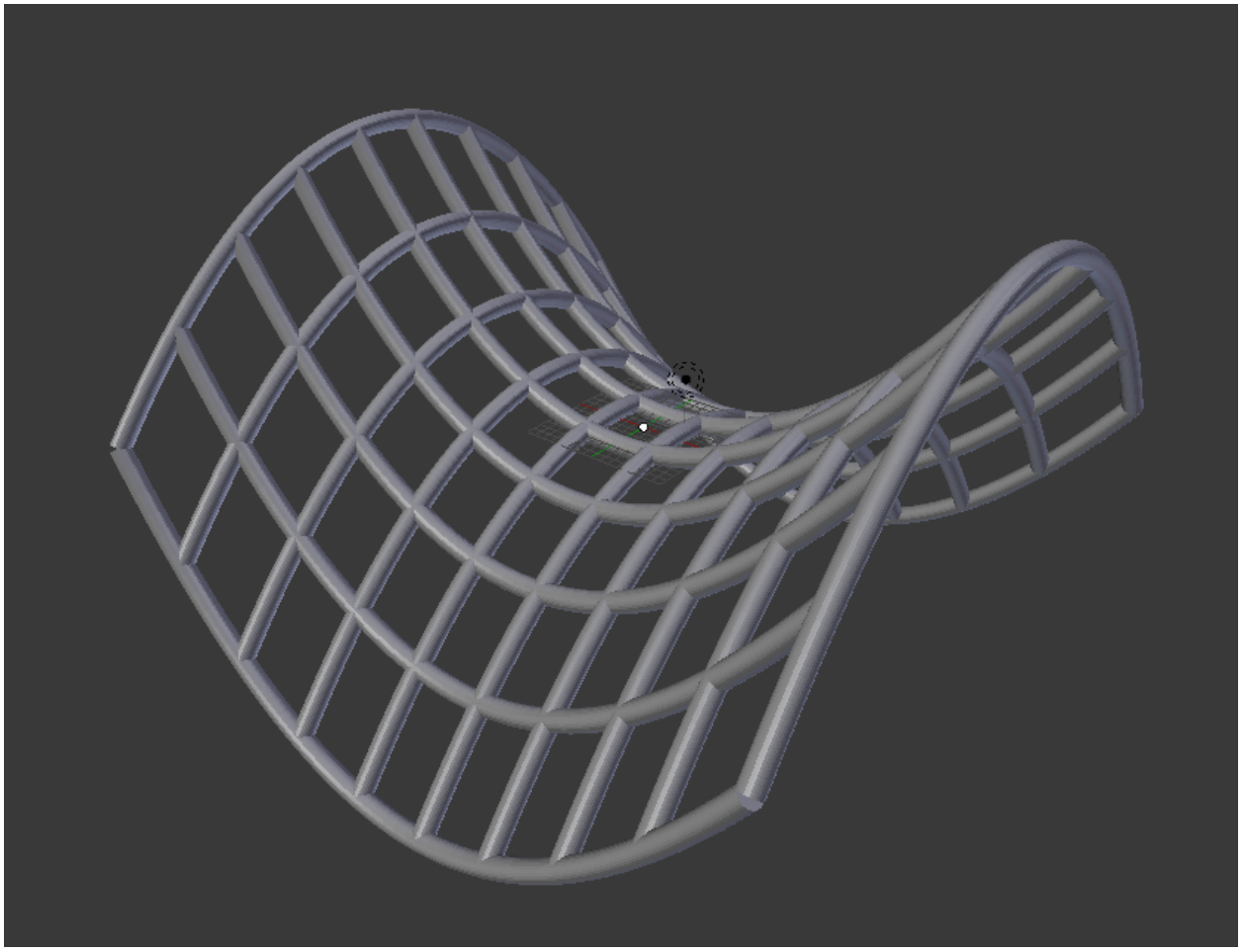
This technique can be used to “voxelize” triangle meshes. The following shows the progressive voxelization of part of the helical fractal mesh.



Now that Pine can handle intersecting meshes, it can freely generate fractals and other patterns which contain intersections. One example which demonstrates this is a six-way tree fractal. By recursively generating Tubes of triangles, Pine can create the following fractal design.



As another example, Pine can generate plots of 3D mathematical functions again using intersecting Tubes. The following represents a section of a hyperbolic paraboloid.



6 Conclusion

Through its robust and flexible method of solid generation, Pine can create physical models whose construction is beyond the capabilities of traditional CAD software. Pine allows fractals, 3D plots, and other abstract mathematical phenomena to be manufactured.