Knotty: Knot Generator

CS285 Final Project

Andrew Lee, Brandon Wang

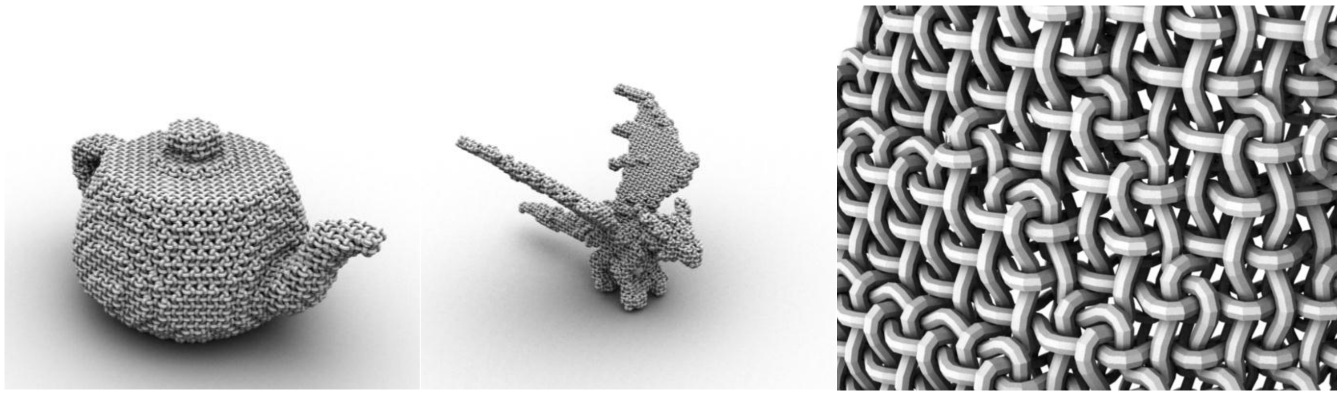

Figure 1: *Example outputs of our algorithm (left and middle) and a close-up of the resulting weave texture (right). More examples are shown on the last page.*

Abstract

We present a tool for the automatic procedural generation of a single-string Chinese-style knot that resembles any arbitrary input 3D model. We call this tool Knotty.

1 Introduction

Chinese knotting is a decorative folk art form that creates interesting shapes and patterns by knotting together cord. Most examples of Chinese knots are abstract and symbolic. However, there are few examples of knots that are designed to resemble actual objects like animals. The only examples we could find were of dragonflies, fish, and turtles.

We wish to procedurally generate a knotted representation of an arbitrary object, and fabricate a model using a Fused Deposition Modeling (FDM) machine.

2 Workflow

To create our knot, we take in an arbitrary object as an OBJ file. The OBJ is then voxelized into axis-aligned voxels, with a user-specified resolution. We then find the outer faces of the voxelized object, and find the faces connected to each face. From this network of interconnected faces, we find an Eulerian cycle to determine a path for a single string. The resulting path across each face is matched to one of three cases of weaving, from which we derive control points for a spline. We create a physical object by creating a sweep along the spline, converting this sweep into polygons, and exporting the resulting b-rep into STL or OBJ.

2.1 Voxelization

Our first step is to voxelize the object, into axis-aligned cubes. This is done to create more uniform face sizes, so that in a later stage, each weave will have approximately the same amount of space.

This is done for b-reps in one of three ways. We found positives and negatives of each approach attempted.

Our first attempt was to use a 3D winding number. We create a 2D x,y plane outside the object, and trace rays in the z direction. For each triangle intersection, we maintain a count of the winding number, increasing the counter if the dot product of the normal of the triangle with our ray is positive, and decreasing otherwise. The interior of the object is defined to be regions with a positive winding number counter.

Our second attempt was essentially the same idea as our first. Instead of using a winding number test, we used the common in-out "exclusive or" boundaries. That is, every intersection changes our orientation from inside to outside, or vice versa. This is useful for objects with incorrect normal information.

Although our first two methods outline voxelization methods for closed objects with no overlapping polygons, we found that the objects we encountered are not always as "well-behaved" as we would like them to be. For example, we would frequently encounter objects with overlapping polygons. Thus, for naughty objects, we trace rays from three orthogonal axis-aligned planes, and mark every intersection as a voxel. Though this only gives us boundary voxels, we only need the boundary voxels for our algorithm. However, this method is three times slower than our first two methods, so we reserve this option only for naughty objects. The winding number test is chosen by default for our voxelization, but the user may manually specify which of the three tests to take.

Furthermore, we supersample the object to remove aliasing artifacts.

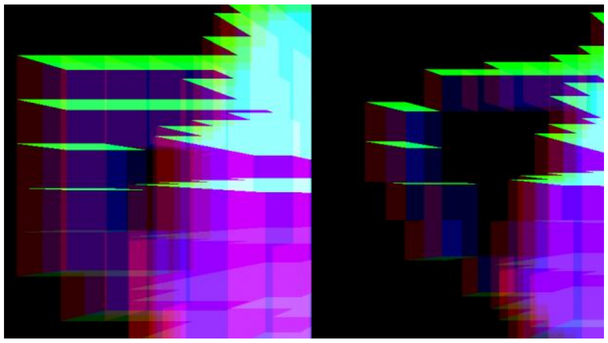

Figure 2: *Supersampled voxelization to remove aliasing (left) and a single sample voxelization (right)*

2.2 Graph

With the voxels, we can extract a surface composed of quadrilaterals. Namely, the outside-facing square faces of the voxels.

To find this surface, we first determine the set of all square faces of the voxels that touch exactly one voxel. If the voxels form a hollow shape, this set would include faces that are not visible, so we cannot stop here. Next, we trace a ray from the outside inward to find a face that is certainly touching the outside. From that starting face, we crawl the surface by finding the faces that share an edge with the faces we have already determined to be surface faces.

There is a tricky case where one edge is shared by exactly two voxels. In this situation, where all four faces contain that edge, we pick the face that would yield a concave bend.

From this surface, we derive a graph simply by making each face a vertex, and connecting the vertices the same way each face is connected to the other faces. In other words, the edges in the graph correspond to the edges that the surface faces share as figure 3 illustrates.

2.3 Eulerian Path

We want to generate a knot that is made from a single length of string, which is the case for most Chinese knots. To do that, we find an Eulerian path on the graph we had just generated. Every vertex on this graph has exactly 4 edges, so it is always possible to find an Eulerian path at this point. We used Hierholzer's algorithm to find such a path, outlined here:

```
start with a vertex **V** on graph **G**
find cycle **C** by traversing **G** until arrive at **V**
while **G** has unused edges:
        traverse **C** until arrive at **V'** with unused edges
        find subcycle **C'** starting and ending with **V'**
        splice **C'** into **C** at **V'**
```

This algorithm has a linear running time with respect to the number of edges in the graph, so it is pretty fast. We made one addition to the algorithm: At a vertex where we have multiple unused edges, which gives us a choice in which direction to take next, we choose the edge that would correspond to going in the current direction. This will correspond to a more aesthetically pleasing knot.

2.4 Weaving

Now, we must consider weaving. In order to have a good knot, the string should weave over and under itself in a consistent manner.

As illustrated in Figure 4, when we consider a surface face, the Eulerian path on that face can be one of three cases. For each case, we define how the two string segments interweave and the sequence of controls points that would yield these patterns (not shown).

Upon closer inspection, we noticed an important phenomenon. If we imagine cutting each string segment in half, and then labelling each piece either "over" or "under" (shown as green and purple, respectively), it turns out that each case follows the same pattern. Namely, the pieces starting at opposite edges are both "over" and the other two opposite edges are both "under".

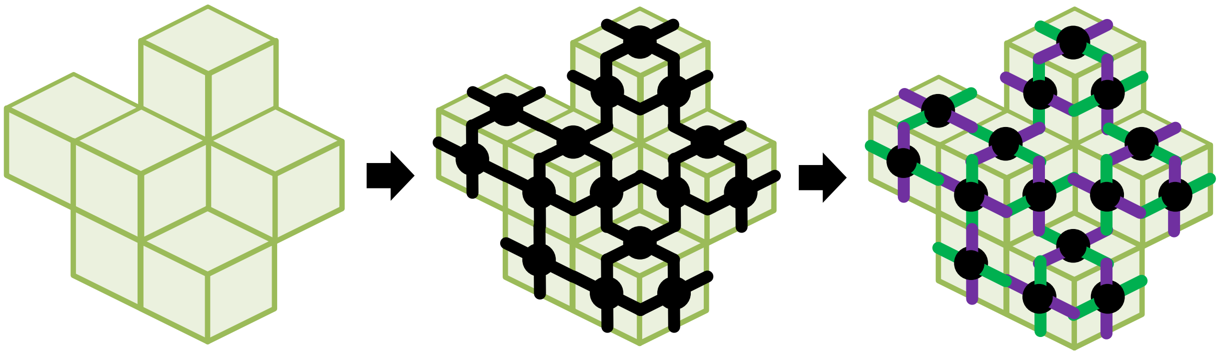

Figure 3: *The voxel surface (left), the graph derived from the surface faces (middle), and the graph colored to represent which segments should be "over" and "under".*

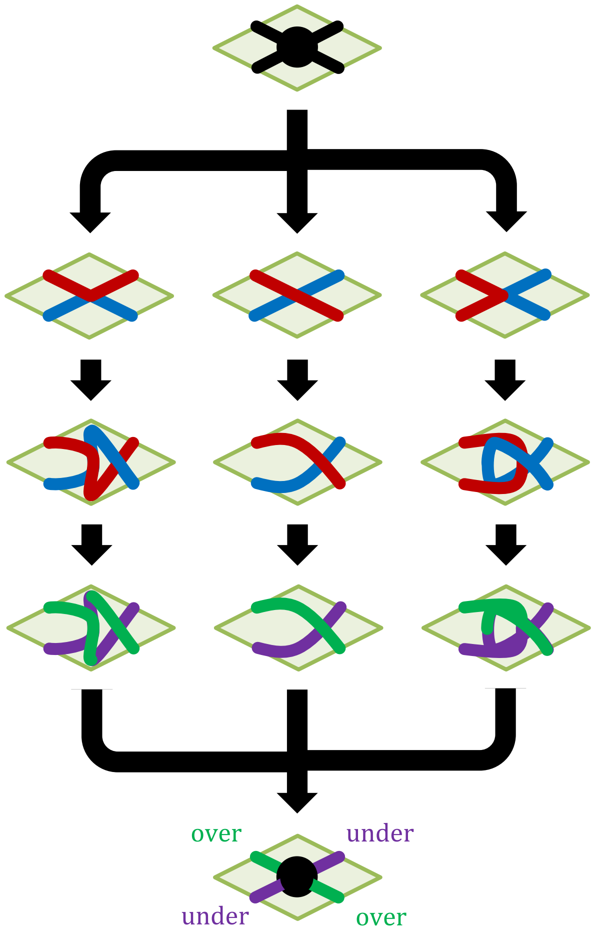

Figure 4: *The weaving cases to consider.*
(top) The portion of the graph on a surface face
(1st row) The three cases that the Eulerian path yields
(2nd row) The manner in which each case is realized
(3rd row) Recoloring the segments to highlight which portions are "over" and which are "under"
(bottom) No matter the case, we can assign each piece to be "over" and "under" the same way

This property lets us "texture" each face of the surface as shown in the right of Figure 3, labeling which segments go over and under. Notice that when we follow a segment from one face to the next, it always switches between "over" and "under". It can be shown that any surface composed of quadrilaterals can be textured in this way, where the label of a segment switches across an edge of the surface.

Therefore, when we traverse the Eulerian path, the segments will consistently alternate between "over" and "under", leading to a good weave.

At this point, we can generate a sequence of control points for the B-spline that becomes the knot. To do this, we simply traverse the Eulerian path, and at each vertex, we determine which of the cases in Figure 4 is satisfied. We then append to our sequence the control points that correspond to that case.

2.5 B-Spline Generation

Given a B-spline's control points, we sample the function at a rate of $2 \cdot |control\ points|$. We found this to be a good approximation, given the large number of control points we deal with. More samples would greatly increase the output file size, and less samples produce artifacts in the final object.

We use De Casteljau's algorithm to sample our b-spline. De Casteljau's algorithm is a recursive function to evaluate B-spline curves in Bernstein form. The Bernstein form of a given segment of our B-spline is defined as a function of its control points. For Knotty, we use cubic B-splines:

$$B_i(t) = (1-t)^3 \cdot CP_i + 3(1-t)^2 \cdot t \cdot CP_{i+1} + 3(1-t) \cdot t^2 CP_{i+2} + t^3 CP_{i+3}$$

$t \in [0,1]$

Given this Bernstein polynomial, De Casteljau's algorithm first creates line segments between sequential control points (i.e. 0,1 and 1,2). For our cubic case, this creates three lines. Each of these three lines is subdivided with a ratio of $t:(1-t)$, and the points are connected. This subdivision step is repeated for the resulting line segments, until we have a single line segment, where the subdivision point is our sample result.

Given sampled coordinates of the B-spline, we create a physical 3D object by sweeping a 2D cross section along the polyline. The orientation of the 2D cross section is defined by a rotation minimizing frame (RMF). Three vectors in 3D space: normal, tangent, and bitangent vectors, define our RMF. The RMF is used to minimize the rotation of our frame across the entire spline.

We used a double reflection method [1] to compute the RMF. The double reflection method can be thought of as projecting the spline onto a sphere, and maintaining the RMF of the spline by using the normal of the sphere, tangent of the spline function by using the next sample, and the bitangent as a cross product of the previous two vectors. We also checked if our normal vector would flip abruptly, and subsequently negate the normal vector. We find that the double reflection method generates very smooth sweeps from our given polylines.

The resulting vertices of each cross section are connected to their corresponding vertex on the previous cross section.

2.6 Exporting

Using the connected objects, we can render our shape to a screen using OpenGL, creating a strip of quadrilaterals formed by two vertices of a cross section to the corresponding two vertices on the previous slice, for all sequential vertices.

To export our shape to STL or OBJ, we trivially split each quadrilateral into two triangles, and write the vertex information in the respective format.

2.7 Caching

The generation of each step requires a significant amount of computation. We cached both the voxelized representation of the object, as well as a complete representation of our sweep's vertices to allow us to complete the program and test it in a reasonable amount of time.

For the generated high-resolution Stanford dragon in Figure 5, our entire process took about 51 minutes, whereas the winged dragon in Figure 5 took about 3 minutes.

3 Results

A video illustrating the Knotty workflow can be seen at:

http://www.youtube.com/watch?v=cKD7Vz54RK4

A video illustrating the Eulerian cycle our algorithm finds can be seen at:

http://www.youtube.com/watch?v=vYV00L6PanE

3.1 GitHub

Our project is open source and can be seen on GitHub at:

https://github.com/bmwang/knotty

4 Future Work

Because we were busy undergraduates in the midst of other projects and finals during the development of Knotty, we were not able to implement various features that we believe are natural extensions to our existing code.

4.1 Big Loops

Traditional Chinese knots often include big loops that flare out. In existing animal Chinese knots, these loops have been used to model the head and fins of a turtle and the wings of a dragonfly. Additional features these loops may model are feathers, horns, ears, and tails.

In our workflow, this extension is conceptually simple: Take a face, grab the section of string on that face, and pull out that length of string into a loop.

4.2 Path Finding

To find an Eulerian path for the knot, we simply took an existing algorithm that was simple and efficient. However, the path we get is often infeasible to physically realize with actual string. We believe that further research into actual Chinese knotting patterns will allow us to design an Eulerian path finding algorithm that obeys the traditions of the folk art.

Fortunately, as we have seen, the graph that this step uses is very simple; every vertex has valence 4, so this fact could possibly be exploited in such an algorithm.

4.3 Surface Generation

In Knotty, the way we generated a surface that approximates the original model was by fitting voxels to the model and then extracting the surface from the voxels. While this yields a simple surface to work with, it ultimately yields a bit of a boxy look. Additionally, the axis-aligned voxels may not obey the "grain" of the input model. For example, on the Stanford dragon, it would be very aesthetically pleasing if the knot's grain follows along the path of the dragon's body, instead of along the 3D lattice we impose by the voxels.

In our workflow, the steps starting with the graph only require a surface composed of quadrilaterals. Therefore, the voxelization and surface finding steps can be replaced by any algorithm that produces a surface composed entirely of quadrilaterals that approximates the shape of the input model. For a good knot, these quadrilaterals should be roughly the same size; quadrilaterals with wildly differing sizes would lead to an uneven knot.

4.4 User Interaction

Currently, Knotty has no user interaction. Adding user interaction could make it a powerful tool for artists. For example, the ability to adjust the surface could be useful if the automatically generated quadrilateral surface has issues. Also, to make the big loops extension, the artist should have the ability to select which faces to make big loops out of.

5 Conclusion

We have presented an algorithm for generating knotted representations of arbitrary objects. Our algorithm creates valid STL files for fabrication, as stated in our original project proposal. We utilize an Eulerian path algorithm to enforce a weave of an outer surface using a single string.

We believe our project to be a success, as we deliver on the main promises made in our project proposal.

References

[1] Wang, W., J¨uttler, B., Zheng, D., and Liu, Y. 2008. Computation of rotation minimizing frame. ACM Trans. Graph. 27, 1, Article 2 (March 2008), 18 pages.

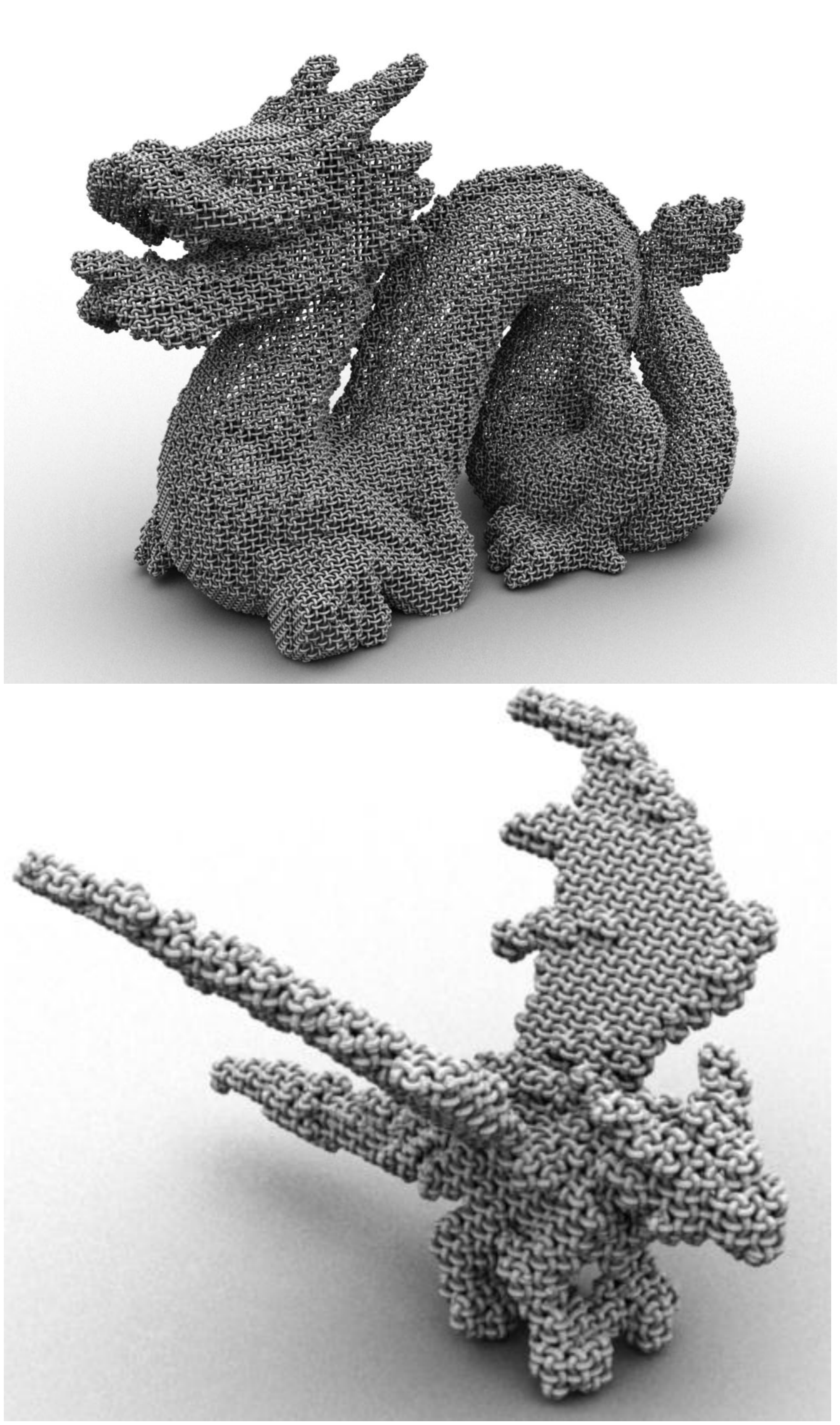

Figure 5: *High resolution Stanford dragon (above), knotted winged dragon (below)*