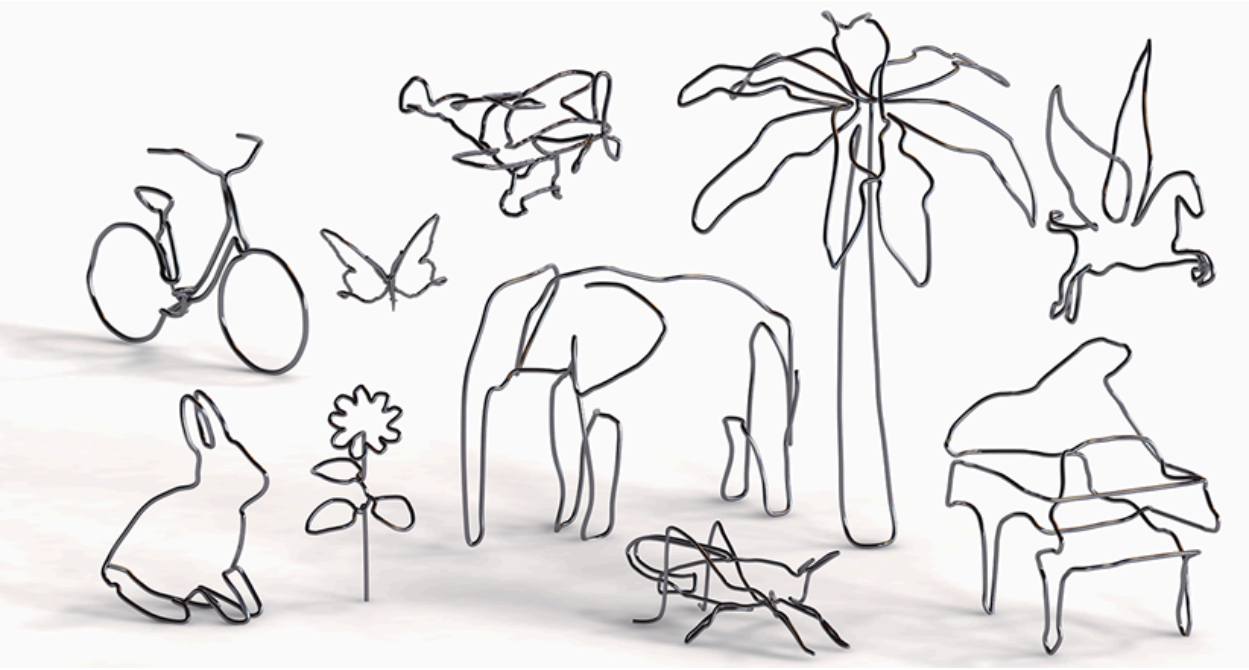
WireSculpt

Maya Plug-in Design Document



Wire art from “WireRoom: Model-guided Explorative Design of Abstract Wire Art”

Claire Lu Neha Thumu

Based on:

WireRoom: Model-guided Explorative Design of Abstract Wire Art

Zhijin Yang, Pengfei Xu, Hongbo Fu, and Hui Huang

CIS660: Advanced Topics in Computer Graphics and Animation
Instructor: Dr. Stephen Lane

PROJECT SUMMARY

Production/Development Need

Wire art is utilized in sculpture pieces, jewelry making, and many more creative projects. Abstract wire art design can be difficult because artists need to shape the wire into a visually aesthetic form that can be viewed from different angles. This often requires many iterations, time, effort, and resources. Our tool, WireSculpt, allows artists to quickly develop and iterate on a prototype that aligns with their vision much more effectively.

Design Goals

Given a 3D model, this tool should allow for easy generation of a set of candidate wires that is visually pleasing and captures the shape and significant features of the model.

Target Audience

Our audience is mainly wire sculpture artists, enthusiasts, and hobbyists. The tool should facilitate users who may have little experience or struggle with developing wire art designs to create their own wire art. It can also be used for more experienced artists for quickly generating and iterating on prototypes to fit their intended look.

Tool Functionality

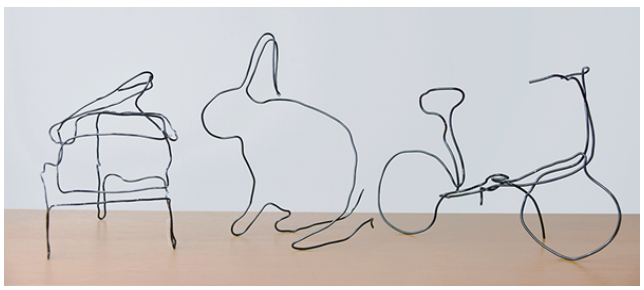
This tool can generate a set of candidate wire shapes for a given 3D model from which the user can choose from. The user can change the appearance of the wire shape by adjusting the parameters that are used for the wire generation algorithm. They can also edit a given wire shape.

Implementation Details

The algorithm for generating wire shapes involves using the dynamic traveling salesman problem (TSP) to construct a set of near-optimal paths. Specific points on the mesh are chosen as landmark sites and paths are generated simultaneously from each of the landmark sites. Path generation incorporates weighting to encourage feature attraction and discourage path repetition. Only a subset of paths are kept to avoid exponential generation of paths. At the end, an error calculation is computed to determine the final set of paths presented as the candidate wire shapes.

Development Schedule

The project will be divided into various milestones. For the Alpha version, the tool should utilize a basic TSP algorithm that incorporates landmark site selection. For the Beta version, all major features should be implemented, including feature attraction, path repulsion, and error calculation. The Final version will focus on enhancing aesthetics, including extruding curves to mimic wires, bug testing, and final polishing.



1. AUTHORING TOOL DESIGN

1.1. Significance of Problem or Production/Development Need

Wire prototyping is a common technique for sculptors prior to starting their pieces. To create prototypes, artists often follow a reference and replicate it to the best of their ability. The wire sculptures can also be artworks on their own. The process of creating these sculptures can be an arduous one as it requires time and effort to bend wire in the desired shapes. It is also common to create iterations of wire art before finding a version that fits with the artist's intention.

WireSculpt facilitates an easier wire art process for artists! In case they desire to create a prototype, our product will ensure that the resulting wire resembles the key structure of the input reference. If they simply wish to iterate upon their idea, our model will provide various versions of wires based on the input 3D model. The wire art generated by our model is guaranteed to be fabricated in real life.

1.2. Technology

This tool is based on the 2021 SIGGRAPH paper titled "WireRoom: Model-guided Explorative Design of Abstract Wire Art" by Zhijin Yang, Pengfei Xu, Hongbo Fu, and Hui Huang. WireRoom proposes a framework for designing abstract 3D wire art that is composed of a single-wire structure. Their method is made with the intention of being able to be reproduced by artists through traditional means. This paper is a part of a series of papers that focus on creating computational tools for designing wire sculptures; however their methodology is unique due to the wide range of papers that they cite and their goal of creating an abstract single-wire output.

The basis of the algorithm for this paper relies on techniques from the following papers:

- Automatic landmark extraction technique from Au et al. 2011
- Heat method for distance computation from Crane et al. 2017
- Traveling salesman problem from Punnen 2007
- Apparent ridges from Judd et al. 2007
- Suggestive contours from DeCarlo et al. 2003

We chose this paper because we found the problem space to be intriguing and wished to delve deeper into this topic. This functionality (creating an abstract wire representation of a 3D model) also does not currently exist in Maya so this could be a valuable contribution as a plug-in. We also wanted to create a tool that would benefit artists and liked the fabrication element of this paper.

1.3. Design Goals

1.3.1 Target Audience.

The target audience for our tool is mainly artists who are interested in creating wire art using a 3D model as a basis.

1.3.2 User Goals and Objectives

The user can easily prototype and automatically create abstract wire versions of a given 3D model. The resulting wire is also able to be fabricated in real life due it being a single connected wire.

1.3.3 Tool Features and Functionality

The primary function of the tool is for the user to be able to generate an abstract wire representation of a given 3D model. They are able to change the results by toggling the following parameters.

They can adjust the feature attraction weights via the range parameter (a), which controls the strength of the attraction, and the steepness parameter (b), which controls the range/locality of the attraction.

They can adjust the path repulsion weights via a* and b* which are similar to their attraction counterparts but affect the strength and locality of repulsion.

The user can also affect the multi-path construction by changing K, the number of paths that are expanded in an iteration, and M, the number of paths to keep in an iteration. A lower number would be computed more quickly but would give less variety and a higher number would do the opposite.

The user can also pass in a custom λ value which would affect the ranking of the resultant wires (the order they appear in the UI).

Finally, the user can specify a thickness value for the NURBs curve extrusion (for when they pick a wire from UI and it is created in Maya).

The user can choose from a series of possible desired wires in the UI by cycling through them via a slider.

Time permitting, the user will be able to see an estimated time until the process is completed from the UI.

1.3.4 Tool Input and Output

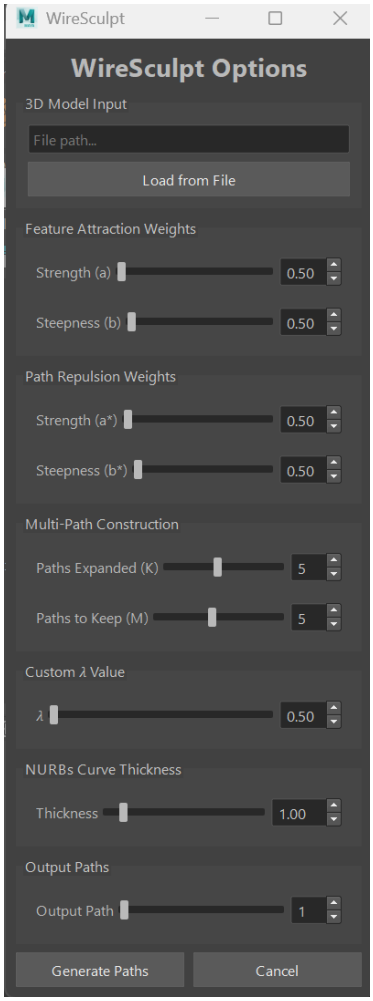
Input: 3D model in an .obj format

Output: a set of candidate wires in the shape of the mesh (specifically will be a NURBs curve)

1.4. User Interface

The user interface will contain a place to load in a 3D model and various parameters that the user can adjust. The user can begin the wire creation process in this interface. At the end of the process, the different resultant wires will be visible in the UI and the user can pick their desired output through a button press. This output will then be visible in Maya as a NURBs curve. A mockup of this interface can be seen below.

1.4.1 GUI Components and Layout



The main window proposed GUI contains the following:

The user can load in their 3D model by clicking on the ‘load from file’ button and by finding their file path. This file can only be of type .obj. The file path will be added to the input field.

The user can adjust the various parameters by moving the sliders or typing values into the slots. These parameters will adjust the weights, path construction, ranking of the paths, and the thickness of the output wire.

When the user clicks on the ‘generate paths’ button, it will begin the process of creating the paths on the algorithmic side.

The output from this process will be visible from the output paths slider. The user can move this slider in order to view the various wire outputs. Each wire will be constructed in the Maya window.

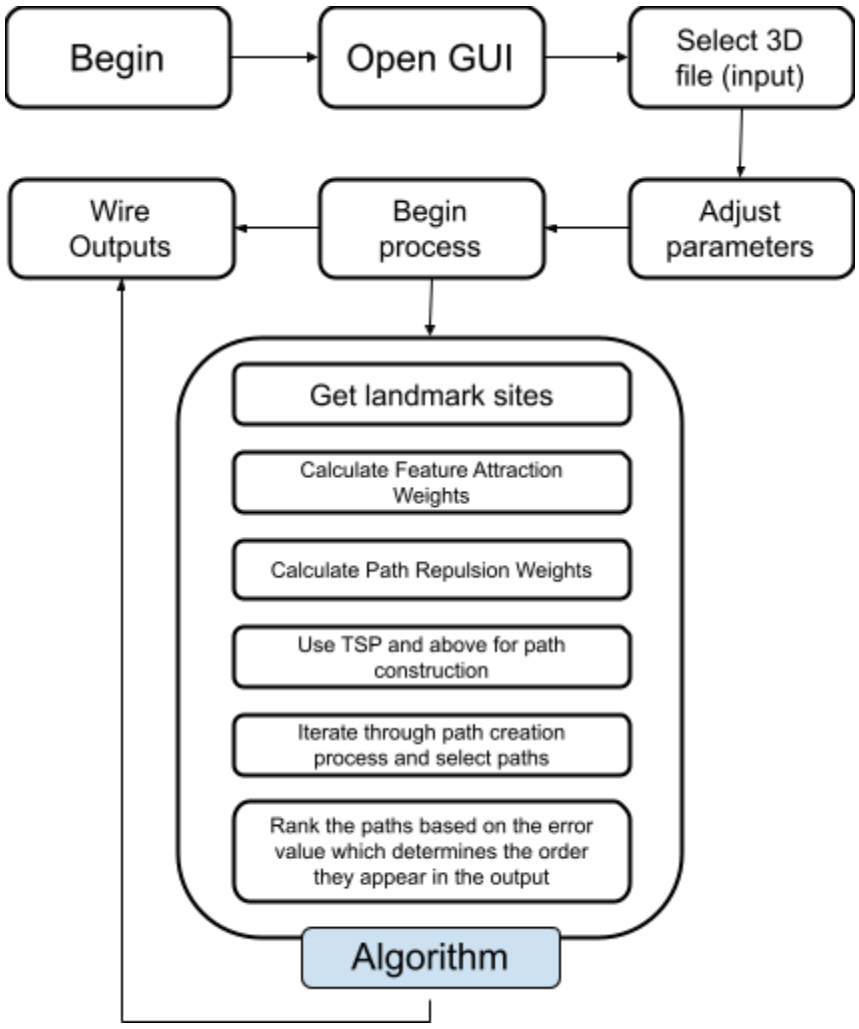
We plan to iterate upon this UI design as we continue working on this project to better suit what the tool supports.

1.4.2 User Tasks

Our goal is for this tool to be artist friendly and not require any technical prior knowledge. There is no need for the user to have any prior knowledge of how the wires are constructed. In the interest of time it may take to process the model, it would be useful for the user to be aware that high poly models would take longer to generate wires due to the number of vertices.

The user may need some artistic prior knowledge when choosing their desired wire output from the series of possible wires constructed from their 3D model; however that is only necessary if they are intending to use our tool for their own work/fabrication process. They may need some familiarity with Maya if they wish to examine their desired wire output within the Maya interface. This would involve knowledge of Maya controls such as zoom, pan, and rotate.

1.4.3 Work Flow



Prior to using this tool, the user needs to load in the plugin to Maya (which we will provide a tutorial for) and the node that has our algorithm in the same scene. They also need to find a 3D model that they wish to turn into wire art. After the algorithm returns paths, the user can cycle through the paths in the UI window and see the wires in the Maya scene. They can then export the wire mesh if they so choose using Maya's builtin export logic.

This workflow is a straightforward approach for the user to be able to create an abstract wire representation of their given 3D model. The tool is also easy to use and requires no prior knowledge from the user other than knowing the basic control scheme for Maya. This would also be an easy way to iterate wire prototypes for artists seeking to create wire sculptures by hand.

2. AUTHORING TOOL DEVELOPMENT

2.1. Technical Approach

2.1.1 Algorithm Details

Wire Generation Overview

Our generator has several key components: loading in the mesh, processing the extreme points of the mesh, using the Traveling Salesman Problem in conjunction with the extreme points, and utilizing various line drawing algorithms to calculate feature weights and make our resulting wire look more aesthetically pleasing.

To load in the mesh, we will only be allowing .obj files as they have all the data that we require for this tool (vertices and normals). We will be making use of third party software to aid us in the processing of these files.

To acquire the extreme points of the mesh, we will mirror the WireRoom author's strategy in using the automatic landmark extraction algorithm outlined by Au et al. 2011.

We need to calculate feature attraction weights to ensure that the path goes through key points of the model. These key points can be acquired using the apparent ridges and suggestive contours methods proposed by Judd et al. 2007 and DeCarlo et al. 2003 respectively. These weights are defined using a distance field, proposed by Crane et al. 2017, and a modified logistic function.

We also need to calculate path repulsion weights in order to ensure that the same features are not visited multiple times resulting in an unsatisfactory path. These weights are calculated by using the distance value of the vertex and using a range and steepness parameters.

We will then use the Traveling Salesman Problem based on the extremities and the weights. TSP will merely be used as a way to traverse the graph composed of the vertices of the mesh. We will begin the path from one of the extreme points and create a series of paths based on vertices that have not been traversed yet and the repulsion and attraction weights mentioned above. In each iteration of the path creation, the paths are assessed and discarded based on their feature coverage. When the iterations are complete and we have a set of complete paths, we will assess these paths using gaussian density fields to determine how well the paths align with our desired features and rank them in the UI so the user may choose their desired wire art.

Input: 3D model as a .obj in the form of triangle meshes
Output: a set of candidate 3D wire shapes
Procedure

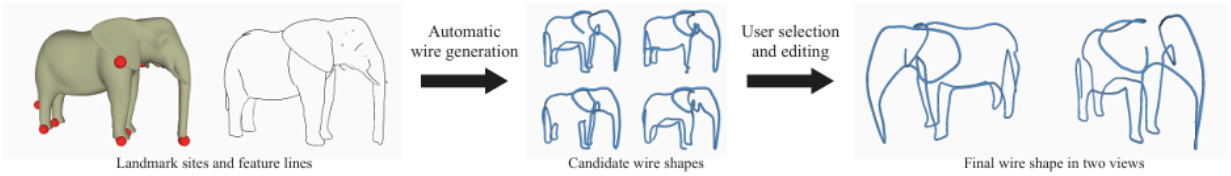


Fig. 3. The workflow of our framework. Given an input model, we first extract the landmark sites and the feature lines (left). Then our automatic wire generation algorithm returns a set of wire shapes (middle). The user may select the most desired one and perform subsequent edits to obtain the final wire shape (right, displayed in two views).

1. User inputs 3D model and chooses a camera view for the desired shape
    a. Process geometry and initialize vertices and edges
2. Extract landmark sites following algorithm Mesh Segmentation (Au, 2011) and define them as extreme points
    a. We plan to implement this using the provided implementation cited in section **2.1.3 Software Design and Development**
3. Implicitly warp the triangle surface using feature attraction and path repulsion weights *Feature attraction weights shorten the triangle edges near feature lines to encourage more accurate capture of desired feature lines of a target model; Path repulsion weights lengthen triangle edges near existing paths to discourage going through the same features multiple times. This effectively reduces and increases, respectively, the distance in the DTSP problem run later.*
    a. Compute feature attraction weights:
        i. Extract 3D feature lines on the surface of the model following line drawing rendering techniques (Decarlo, 2003; Judd 2007)
            1. We plan to implement this using the provided implementation cited in section **2.1.3 Software Design and Development**
        ii. Build a distance field on the surface using the heat method, using the feature lines as sources (Crane, 2017)

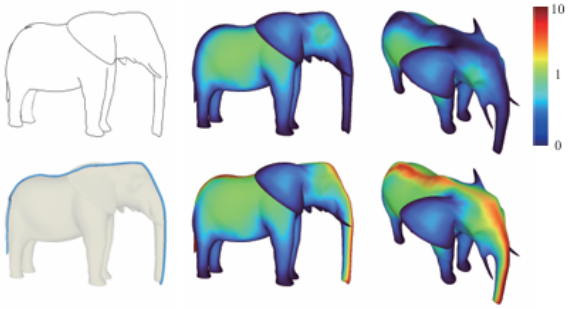


Fig. 4. The visualization of the weights for implicitly warping the surface model. Top: the visualization of the warping effect with the feature attraction weights. Bottom: the visualization of the warping effect with both the feature attraction weights and the path repulsion weights. Left: the feature lines (top) and an existing path (bottom) for computing the weights. Middle and right: the visualization of the warping effects in two views.

1.
    a. Blue represents shorter distances; Red represents longer distances

2. We plan to implement this using the provided implementation cited in section **2.1.3 Software Design and Development**

iii. Define feature attraction weights using the modified logistic function presented in WireRoom (below)

1. The feature attraction weight $\boldsymbol{w_i}$ of each vertex $\boldsymbol{v_i}$ of the surface mesh with the distance $\boldsymbol{x_i}$ (from heat method, Crane 2017) is defined by $\boldsymbol{w_i} = \mathrm{L}(\boldsymbol{x_i})$

2. $$L(x) = \frac{a}{1 + \exp(-bx/\bar{l})} + (1 - a),$$

   a. Where a is the range parameter (controls the range of this function, i.e., $L(x) \in [1 - a/2, 1)$ when $x \in [0, +\infty)$)

   b. And b is the steepness parameter, which controls the rate of change of $L(x)$.

iv. Adjust edge lengths in the graph such that each edge, $\boldsymbol{l_{ij}}$ is recomputed as the following

1. $$l'_{ij} = \frac{w_i + w_j}{2} l_{ij}$$

The effect of the feature attraction weights is visible in the below figure presented in WireRoom

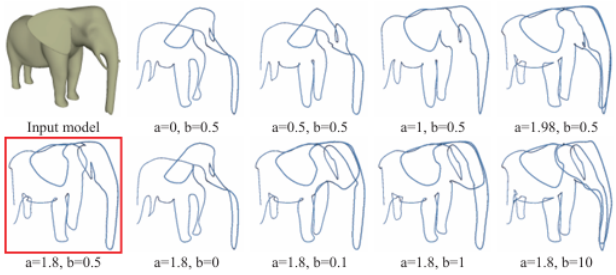


Fig. 6. The wire shapes generated with different combinations of a and b. The red rectangle highlights the wire shape generated with the default parameters. When $a = 0$, i.e., the first wire shape in the top row, the generated path approximates to the geodesic path on the unwarped surface.

v.

b. Compute path repulsion weights

i. Build a distance field by setting the existing path as sources

ii. Define a weight function in the form of the function presented in feature attraction weights, such that:

1. The path repulsion weight $\boldsymbol{w^*_i}$ of each vertex $\boldsymbol{v_i}$ of the surface mesh with the distance $\boldsymbol{x^*_i}$ (from heat method, Crane 2017) is defined by $\boldsymbol{w^*_i} = \mathrm{L}(\boldsymbol{x^*_i})$

2. $$L(x) = \frac{a}{1 + \exp(-bx/\bar{l})} + (1 - a),$$

   a. where $x = \boldsymbol{x^*_i}$, $a = \boldsymbol{a^*}$, $b = \boldsymbol{b^*}$

iii. Adjust edge lengths in the graph such that each edge, $\boldsymbol{l_{ij}}$ is recomputed as the following

1. $l'_{ij} = \frac{w_i + w_j}{w_i^* + w_j^*} l_{ij}$

iv. The below image shows the warping effect visualization with respect to an existing path (in blue); taken from Figure 4 in WireRoom

1. 

v. The below image illustrates the effect of parameter $a*$ and $b*$, taken from Figure 8 from WireRoom

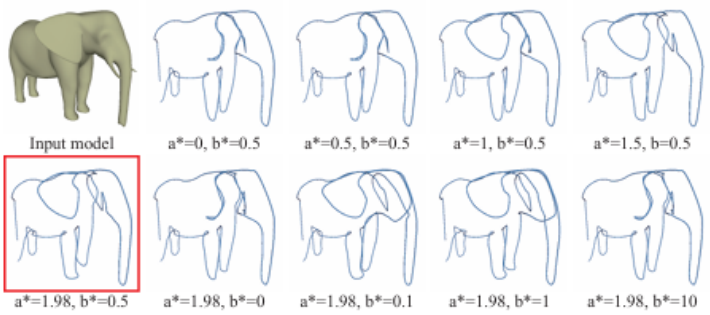
1. 

Fig. 8. The wire shapes generated with different combinations of a^* and b^*. The red rectangle highlights the wire shape generated with the default parameters.

c. The effect of parameters a, b, and a*, b* presented in the feature attraction and path repulsion weights sections above
   i. Effect of feature attraction weights, a and b
      1. Range parameter: ***a,*** $a \in [0, 2)$
         a. Determines how fitted the wires are to the feature lines (how much it modifies the lengths)
         b. Higher ***a*** values represent stronger attraction to the feature lines
      2. Steepness parameter: ***b,*** $b \in [0, \text{infinity})$
         a. Controls rate of change in attraction strength as distance from a feature line increases.
         b. Higher ***b*** values represent stronger attraction to the feature lines, i.e. minimizes the effective area. (A low value of b = 0 essentially spans effective area to whole model)
   ii. Effect of feature attraction weights, a* and b*
      1. Range parameter: ***a*,*** $a \in [0, 2)$
         a. A lower ***a**** reduces edge lengthening effect at the path region and can create effects with more redundancy
         b. A higher ***a**** increases edge lengthening effect at the path region and can cause less redundancy

c. a* increases the edge length based on the existing wire paths; bigger a* = more repulsion and such a longer wire (to ensure that the path is not crossed too many times)

2. Steepness parameter: ***b*,*** $b \in [0, \text{infinity})$
   a. Controls the rate of change of the repulsion effect as the distance increases
   b. Lower values = repulsion has a greater area and as such farther away paths can affect overlapping
   c. Higher values = only nearby paths can repel (more targeted range)

iii. By default, a = 1.8, b = 0.5, a* = 1.98, b* - 0.5

4. Compute multi-path expansion for near-optimal path

In order to generate a set of visually pleasing wire shapes to satisfy different user preferences, it is necessary to generate multiple near-optimal paths (rather than one optimal path). Thus, we will follow WireRoom's procedure of a greedy strategy with simultaneously multiple path expansion. (Shown in the image below)
**Note that the distances between pairs of landmark sites dynamically changes as each step involves recomputing path repulsion weights.*

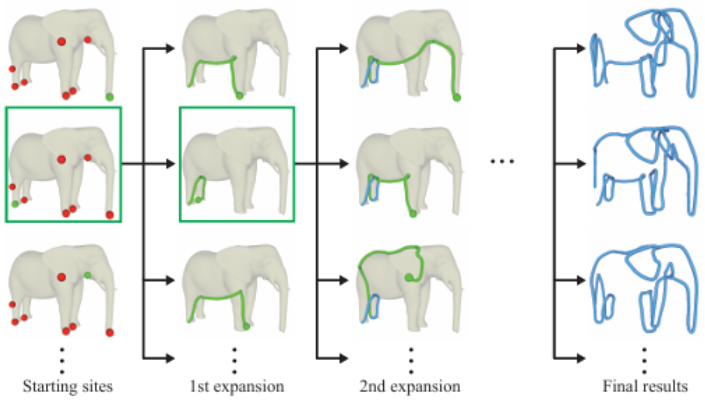


Fig. 5. The multi-path expansion procedure for near-optimal paths. For clear illustration, only one branch is shown in each path expansion. The green rectangles highlight the branches for path expansion.

a. Define the following notation
   i. Let p = a path, defined as a set of consecutive edges.
   ii. Let P = a set of paths, where a path $p \in$ P. P will hold all paths generated by our starting sites.
   iii. Let K = the number of nearest unvisited sites that a path p can expand to in each step. (By default, K = 5)
   iv. Let Δp_k = the expanded paths, i.e. the geodesic path between a current and next site of a path
   v. Let c = a feature coverage score, which defines the degree at which a path covers features, as described in the previous section. A small c_k means that the geodesic path Δp_k passed the region with uncovered

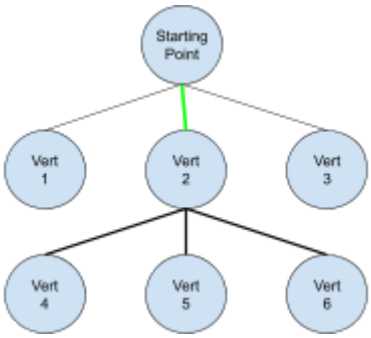
features, which most likely means the path has not looped back on itself and covered features redundantly, so Δp_k is more likely to be kept.

vi. Let M = the number of expanded paths that will be kept. (By default, M = 10)

b. Define a set of starting sites S that includes all landmark sites obtained in Step 2.

c. Let each starting site be a special edge element of a path p and add all paths p to the set of paths P – so that we have |S| paths in P.

i. In other words, each path will have the same starting edge from the starting site



The starting point is the landmark site. The green line is our special edge element (each of the paths from the landmark site start with this edge). Then, they each have their own paths that they take.

P = { p(start->v2, v2->v4),
p(start->v2, v2->v5),
p(start->v2, v2->v6) }

d. Compute the feature attraction weights from Step 3.

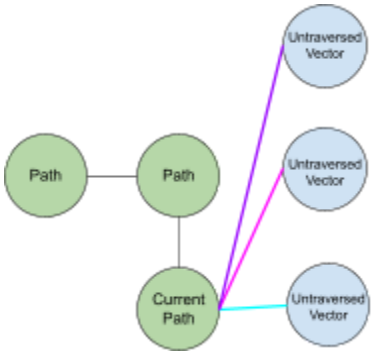
e. While there exists an unvisited site:

i. Let P' = the new tracked path set.

ii. (Re)compute path repulsion weights, as the existing path expands with each iteration.

iii. For each path in P:

1. Expand each path to K nearest unvisited sites, from the endpoint of the path that was most recently added. This yields K expanded paths as defined by Δp_k



The nodes in green are the vertices that compose the path so far. The nodes in blue are the expansion of each path to untraversed nodes/vertices.

In this case, K = 3.

2. Compute the feature coverage score c for each of the expanded paths.

a. $$c = \frac{\sum_{e \in \Delta p} l'(e)}{\sum_{e \in \Delta p} l(e)},$$ where $l'(e)$ = the edge length on the warped surface, $l(e)$ = the edge length on the original surface.

3. Rank all expanded paths by their feature coverage score c_k, this score is larger if features are traversed repeatedly.
4. Keep the top M ranked paths. Add the updated paths, which now include their expanded parts, to P'.

iv. Let P = P'

5. Rank all generated paths computed in Step 4 and select a subset of them: this happens when all our generate paths are complete
   a. Let $E(p)$ be a global error function defined by the following equation:
      i. $E(p) = E_c(p) + \lambda E_r(p),$ where
         1. E_c = a global feature conformity error
         2. E_r = a path redundancy error
      ii. *E(p)* describes the degree to which a path does not conform to the features lines and the amount of redundancy in our generated path
      iii. We will choose the top M paths with the smallest *E(p)* as our final candidate wires.
   b. Calculate the global feature conformity error *Ec*
      i. Measures the distribution similarity between the path and feature lines of the model
      ii. $E_c(p) = \|\mathbf{F}_p - \mathbf{F}_f\|$, where F*p* and F*f* are two Gaussian fields on the surface following Gaussian density function $g(x,\mu,\sigma)$:
      iii. We define Gaussian density function $g(x,\mu,\sigma)$, where x is the geodesic distance from a surface point to the path or feature lines

$$g(x, \mu, \sigma) = \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

         1.
      iv. Gaussian field generated from wire path: F_p = the sum of $g(x,\mu,\sigma)$ across all vertices, where
         1. x = geodesic distance from outputted wire path to each vertex, computed by using the heat map method from Crane et al. 2017
         2. $\mu = 0$, and $\sigma = 4^{-} l$ in WireRoom's implementation
      v. Gaussian field generated from feature lines: F_f is the sum of $g(x,\mu,\sigma)$ across all vertices, where
         1. x = geodesic distance from feature line to each vertex; these values are already computed by using the heat map method from Crane et al. 2017 in Step 3) a) ii)
         2. $\mu = 0$, and $\sigma = 4^{-} l$ in WireRoom's implementation
   c. Calculate the path redundancy error *E*r
      i. Measures repetition of path in capturing feature lines

ii. $E_r(p) = \left\| \mathbf{F}_p - \sum_{\Delta p \subset p} \mathbf{F}_{\Delta p} \right\|$, where F$p$ is the Gaussian field generated from the wire path and FΔp is the Gaussian fields where $\{\Delta p\}$ represents the set of all expanded paths during the expansion of path p, i.e, $\Delta p \subset p$.:

1. For a path p, this would involve recomputing the heat map from Crane for each of the path segments.

d. Compute $E(p)$ following the above steps and choose the top M paths with the smallest $E(p)$ as the candidate wires
e. These error term values will be used in ranking our generated paths to determine the order that they appear in the UI.

6. Perform spline curve fitting
    a. Perform a Bezier interpolation to make a fitted C^2 continuous curve.
    b. Select control points as the vertices on the path with the highest curvature values.
        i. Larger than $1/\bar{l}$
    c. Next, evenly select vertices, with an interval value of $5\bar{l}$ as remaining control points
    d. We plan to implement this using the provided implementation cited in section **2.1.3 Software Design and Development**

Assumptions/Simplifications

1. The paper cites three sources for computing feature attraction lines, Decarlo 2003 for Suggestive Contours, Judd 2007 for Apparent Ridges, and Haralick 1983 for Ridges and Valleys. We make the simplifying assumption that the first two sources, Decarlo and Judd will provide sufficient feature extraction. In our implementation, we plan to forgo Ridges and Valleys with the assumption that Suggestive Contours and Apparent Ridges will extract.
2. The paper mentions that the landmark has to be converted to a special edge when computing a path. We assume this means for a set of paths that start from a landmark, they would all have the same starting edge.
3. Although this is not explicitly mentioned in WireRoom, we are assuming that the geodesic distances needed in computing F_p and F_f are from the heat map method by Crane et al. 2017.

2.1.2 Maya or Houdini Interface and Integration

For this project, our UI components will be implemented using MEL or Pyqt. The user will be asked to provide a 3D model as an input along with a series of parameters or they can choose to pass in the default values of these parameters (listed in more detail in **section 2.1.3 Software Design and Development**).

Our algorithm, where we generate an abstract wire representation of the given input, will be done in C++. The parameters will be from MEL and will be passed along to the C++ code through a

node interface. Then we will iterate through our algorithm and there will be an output of a series of possible paths that the user can choose from for their desired wire art. These paths will be visible in our UI. Once the path is chosen, a NURBs curve will be returned and visible in Maya.

The creation of the NURBs curve and NURBs circle will rely on Maya MFNMesh objects. We will use both in order to have proper extrusions of the wire.

This NURBs curve can be given thickness to resemble a wire by extruding a NURBs circle along it. The user will be able to specify the thickness of the wire by controlling a slider UI component.

2.1.3 Software Design and Development

- WireNodeUI
    - Described in **1.4. User Interface**
- Custom MPx WireNode
    - Input:
        - Link to obj file path with in directory
        - a (range for attraction)
        - b (steepness for attraction)
        - a* (range for repulsion)
        - b* (steepness for repulsion)
        - λ for error
        - K (number of paths to expand in an iteration)
        - M (number of paths to keep in an iteration)
        - Thickness value of NURBs curve extrusion
    - Output:
        - Extruded NURBs Curve that mimics a wire
- Global Variables
    - Feature line **`Path`**
    - Set of all generated paths (**`Path`**s) P
    - Set of landmark sites (**`Landmarks`**)
    - Vector of candidate wires **`Path`**s
- Vertex Class
    - *Represents the vertices of the input 3D model*
    - Attributes
        - Position
        - Normal
        - Identifier (bool to show it is a landmark)
        - Map of edges for neighboring vertices
            - currentVertex = {neighbor1: edge obj between current and n1, neighbor2: edge obj between current and n2, …}
- Landmarks Class
    - *Represents the landmark sites (extreme points from Au et al. 2011)*
    - **Derived from Vertex class**

    - Attributes
        - Reference to the vertex endpoint of the special edge it represents
- Non-Landmarks Class
    - *Represents the non-landmark sites (not selected as extreme points from Au et al. 2011)*
    - **Derived from Vertex class**
    - Attributes
        - Weights: feature attraction weight ***w,*** and path repulsion weight ***w****
        - Geodesic distance from feature lines
- Paths Class
    - *Stores our paths*
    - Attributes
        - Vector of vertices (**`Vertex`**) that make up the path
        - Map of all the vertices in the path between the pairs of endpoints (**`Landmark`** vertices)
            - PathSegments = {key: [v0=LM1, v1, …, vn=LM2], ..}
            - PathSeg[0] = [v0=LM1, v1, …, vn=LM2]
            - PathSeg[1] = [v0=LM2, v1, …, vn=LM3]
- Edge Class
    - Attributes
        - Original edge length between two vertices
        - Warped edge length between two vertices

Third Party Software:

- Tinyobjloader: https://github.com/tinyobjloader/tinyobjloader
- Libigl (processing 3D models): https://libigl.github.io/
- Eigen Math Library (performing matrix operations): Eigen
- CGAL – Computational Geometry Algorithms Library
    - Surface Mesh Topology https://doc.cgal.org/latest/Manual/packages.html#PkgSurfaceMeshTopology
    - Approximation of Ridges and Umbilics on Triangulated Surface Meshes https://doc.cgal.org/latest/Manual/packages.html#PkgRidges3
- Imgui (C++ GUI): https://github.com/ocornut/imgui
- Common Test 3D Models: https://github.com/alecjacobson/common-3d-test-models

Other Resources:

- Authoring Tool Resources
    - Links to presentation & demo video: WireRoom: model-guided explorative design of abstract wire art
- Suggestive Contours details (including source code) from the authors
    - Source Code: https://gfx.cs.princeton.edu/gfx/proj/sugcon/
- Apparent Ridges
    - Implemented by someone else: https://github.com/yunjay/Apparent-Ridges
- Mesh Segmentation with Concavity-Aware Fields

    - Implemented by someone else: https://github.com/FlorianTeich/concavity-aware-fields
        - We will be using/referencing their automatic landmark extraction method
- Heat Method Implementations
    - https://github.com/kyle-rosa/differentiable_heat_method?tab=readme-ov-file based on Crane et al. 2017
    - https://github.com/CHoudrouge4/HeatMethodForGeodesicDistance?tab=readme-ov-file based on Crane et al. 2017
- C2 interpolating splines for spline curve fitting
    - Implemented by someone else: https://github.com/tanganke/YukselC2Interpolation
- Extruding along a curve to create thickness:
    - Autodesk Maya docs for manual extrusion: https://download.autodesk.com/us/maya/2010help/index.html?url=Creating_NURBS_surfaces_Sweep_a_profile_curve_along_a_path_curve.htm,topicNumber=d0e229867
- Similar Wire Art Implementations
    - Fabricable 3D Wire Art
        - Website: https://cdl.ethz.ch/publications/2024/WireArt/
        - Source code: https://github.com/kenji-tojo/fab3dwire
    - Multi-View Wire Art
        - Source code: https://github.com/KaiWenHsiao/Multi-view-Wire-Art
    - Rod Design (can be useful for how they process curves and .obj file format)
        - Source code: https://github.com/Mukidashi/RodDesign
- Estimated time in Maya
    - https://medium.com/@nicholasRodgers/making-a-time-tracking-tool-in-maya-7025ba308fc7

2.2. Target Platforms

2.2.1 Hardware

This was determined based on the minimum requirements stated by Autodesk for Maya 2022.

- CPU: 64-bit Intel® or AMD® multi-core processor with SSE4.2 instruction set
- Graphics Hardware: DirectX 11 or 12 compatible graphics card
- RAM: 8 GB (16 GB recommended)
- Disk Space: at least 6 GB for installing Maya

2.2.2 Software

This plug-in is developed for Maya and as such the user must have Maya installed to use this tool. This was determined based on the minimum requirements stated by Autodesk for Maya 2022.

- Operating system: Microsoft Windows 10
- Autodesk Maya 2022

2.3. Software Versions

2.3.1 Alpha Version Features (first prototype)

Setup: We will first prioritize setting up the project and ensuring it compiles and runs properly in Maya. This process also includes creating the necessary classes and data structures. If we use PyQt in creating our UI, we will also need to set up swig to ensure our python code can refer to our C++ logic.

Algorithmic Functionality: We will be incorporating the TSP with Au's method (to get the extremities). This will result in a wire mesh (but it will not be aesthetically pleasing).
We will also start working on incorporating the attraction and repulsion weights but are aiming to finish this part of the implementation by the beta version deadline. We are planning to use C++ to create our algorithm.

UI: We will create a basic UI framework that we expect to expand on as we continue our work on this tool. This framework will have the key features that we need (ability to adjust parameters, a way to load in the model, etc.). However, this framework will not be connected to the node-this step will happen in the beta version. This framework will be written in MEL.

2.3.2 Beta Version Features

Algorithmic Functionality: Our goal for this version is to finish implementing the attraction and repulsion weights as well as incorporating them into our path logic (TSP). We will also implement C_k, the feature coverage term, as this relies on the weights. We will also be implementing the error calculation to rank the generated paths to display in our UI.

Other Improvements: As part of this version, we will continue bug testing and ensure our previous functionality is still working. We will also continue to refine our UI.

2.3.3 Final Version Features

We will focus on debugging and testing our code to ensure our wire output is aesthetically pleasing. We will also focus on ensuring that we can properly extrude/sweep the output curve so it resembles a wire.
We will ensure that our UI is polished and is properly connected to our plug-in.
If time permits, we will also add spline curve fitting to make our resulting paths smoother.
Another stretch goal is adding an 'estimated time until completion' to our node.

2.3.4 Description of any demos or tutorials

Our final version code will be available on a public GitHub repository along with a tutorial on how to load our plug-in into Maya so users can easily try it out. We will provide a detailed user manual on how to use our tool. Time permitting, we will also have a general breakdown on how we implemented our plug-in.

Our repository will have a video demonstration going through the features in our tool. We will also have a link to the slides that are part of our final presentation which will contain additional information on the project.

3. WORK PLAN

3.1. Tasks and Subtasks

Task 1 – *Setup the Github Repository and Barebones UI*

Name: Neha
Duration: 1 Day

- **1.1. Github project repository setup**
  A public github repository with a visual studio solution and an appropriate .gitignore file. The repository should be with both members. This repo will also have files like our UI script.
- **1.2. Change the Python based UI to MEL**
  Will convert the current UI schematic to MEL with the help of GPT.

Task 2 – *Node Setup Code*

Name: Neha
Duration: 1 Day

- **2.1. Create a Node script (derived from MPxNode)**
  Ensure that there is a unique ID for the node, an initializePlugin function, uninitialize Plugin function, a way to make the menu pop up, and a process Node function.
- **2.2. Ensure that parameters are being passed in**
  Add print statements to make sure that the UI is connected to the Node and the parameter values are read in properly.

Task 3 – *Setup class structures*

Name: Claire
Duration: 2 Day

- **3.1. Create all classes**
  Create classes for Vertex, Landmark, Non-Landmark, Path, etc. with relevant variables and function headers.
- **3.2. Create global variables**
  This would include the user parameters along with variables like feature line, a set of generated paths, a set of landmark sites, and a vector of candidate wires.

Task 4 – *Process 3D Model*

Name: Neha
Duration: 2 Day

- **4.1. Use third party software to process**
  Will use either Tinyobjloader or Libgl to process the input 3D model based on the given file path. Will also need to only accept .obj files.
- **4.2. Organize model into class structure**
  We need to sort the information from the 3D model into our class hierarchy. This means creating vertex objects and calculating edge lengths, for example.

Task 5 – *Incorporate Au method*

Name: Claire
Duration: 2 Day

- **5.1. Incorporate code implementation for Au method**
  Using the implementation provided in section 2.1.3 for finding landmark sites
- **5.2. Update the relevant member variables**
  Update the weight members in the Non-Landmark vertex variables. Create landmark objects based on this data.

Task 6 – *Implement the A* path algorithm*

Name: Neha
Duration: 3 Day

- **6.1. Implement the A* path algorithm for traversal**
  Using the implementation provided in section 2.1.3 for finding landmark sites and use the A pseudocode on Wikipedia.*
- **6.2. Create paths in iterations**
  Follow the implementation in section 2.1.3 and the parameters to traverse the model and create a set of paths. These paths also need to be evaluated in each iteration.

Task 7 – *Implement the heat map distance method*

Name: Claire
Duration: 3 Day

- **7.1. Implement the heat map method**
  This method is necessary for getting the distances of the vertices based on the sources. We will be using the implementation provided in section 2.1.3.
- **7.2. Update the non-landmark class objects**
  The distances from this method should be updated as the geodesic distance between feature lines.

Task 8 – *Implement the feature attraction weights*

Name: Neha
Duration: 1 Day

- **8.1. Calculate the weights**
  Using the implementation provided in section 2.1.3 and the parameters for computing the weights.
- **8.2. Update variables**
  These weights would need to be updated for the non-landmark objects.

Task 9 – *Implement the path repulsion weights*

Name: Claire
Duration: 1 Day

- **9.1. Calculate the weights**
  Using the implementation provided in section 2.1.3 and the parameters for computing the weights.
- **9.2. Update variables**
  These weights would need to be updated for the non-landmark objects.

Task 10 – *Implement code for computing feature coverage score*

Name: Neha
Duration: 1 Day

- **10.1. Compute the feature coverage score c for each of the expanded paths.**
  Using the implementation provided in section 2.1.3 for finding landmark sites

Task 11 – *Implement error calculation computation*

Name: Claire
Duration: 2 Day

- **11.1. Implement computation for Ec**

Implement the code for computing Ec, involving the Gaussian field calculation for Ff and Fp

- **11.2. Implement computation for Er**
  Implement the code for computing Er, involving the Gaussian field calculation for Fp and F△p
- **11.3. Implement computation for E**
  Implement the code for computing E, using the error for Ec and Er
- **11.4. Return wires in a ranked order**
  This order is based on each resultant wire's E value.

Task 12 – *Implement spline curve fitting*

Name: Neha
Duration: 3 Day

- **12.1. Perform a Bezier interpolation to make a fitted curve**
  Use the provided implementation in section 2.1.3 to incorporate a Bezier interpolation
- **12.2. Implement code for extruding NURBs circle along a NURBs curve**
  Write a MEL script to extrude the circle along the curve, similar to HW2
- **11.3. Connect extrusion to the UI**
  Connect to the UI so user can control thickness of wire

3.2. Milestones

3.2.1 Alpha Version

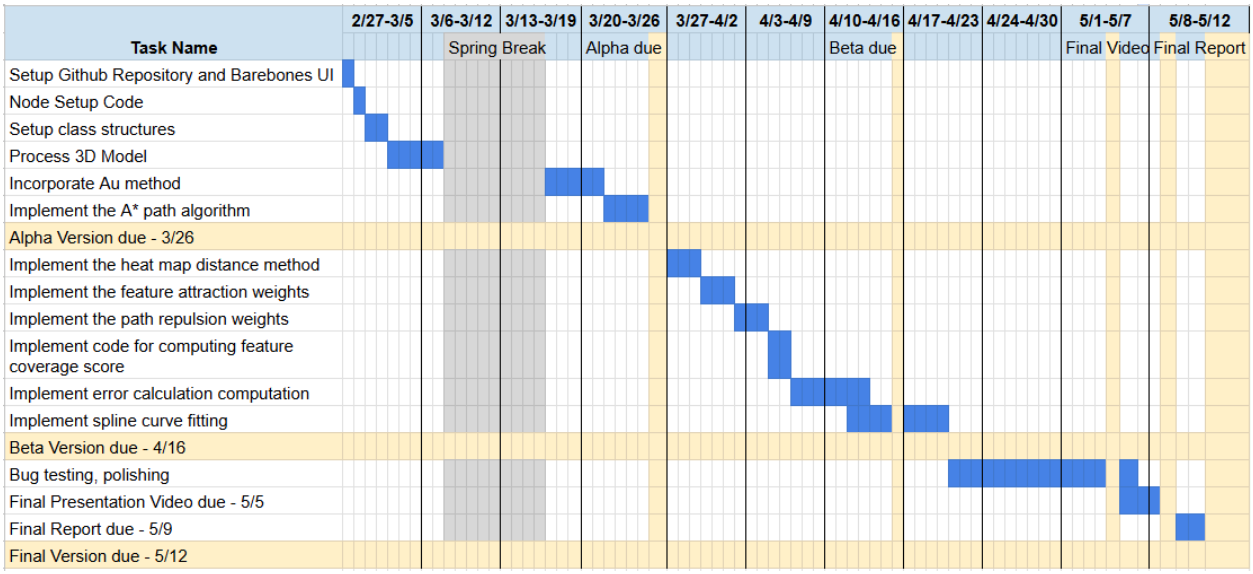
1. *Setup the Github Repository and Barebones UI*
2. *Node Setup Code*
3. *Setup class structures*
4. *Process 3D Model*
5. *Incorporate Au method*
6. *Implement the A* path algorithm*

3.2.2 Beta Version

1. *Implement the heat map distance method*
2. *Implement the feature attraction weights*
3. *Implement the path repulsion weights*
4. *Implement code for computing feature coverage score*
5. *Implement error calculation computation*
6. *Implement spline curve fitting*

3.3. Schedule

GANTT Chart


Task Name
2/27-3/5
3/6-3/12
3/13-3/19
3/20-3/26
3/27-4/2
4/3-4/9
4/10-4/16
4/17-4/23
4/24-4/30
5/1-5/7
5/8-5/12
Spring Break
Alpha due
Beta due
Final Video
Final Report
Setup Github Repository and Barebones UI
Node Setup Code
Setup class structures
Process 3D Model
Incorporate Au method
Implement the A* path algorithm
Alpha Version due - 3/26
Implement the heat map distance method
Implement the feature attraction weights
Implement the path repulsion weights
Implement code for computing feature coverage score
Implement error calculation computation
Implement spline curve fitting
Beta Version due - 4/16
Bug testing, polishing
Final Presentation Video due - 5/5
Final Report due - 5/9
Final Version due - 5/12

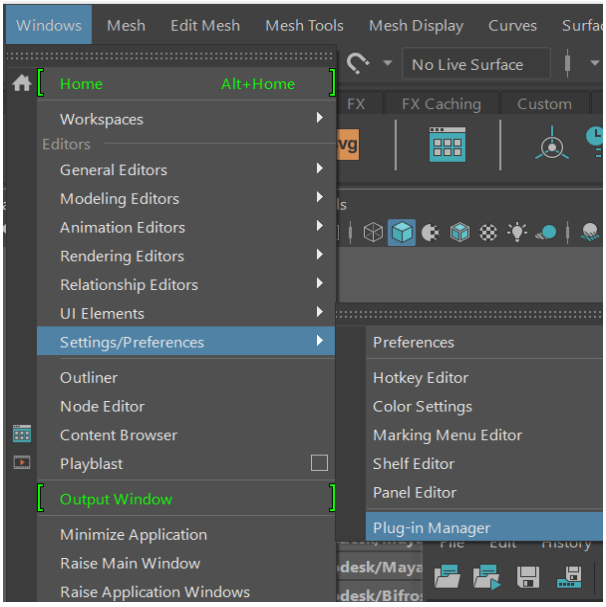
4. FINAL REPORT

4.1. Results

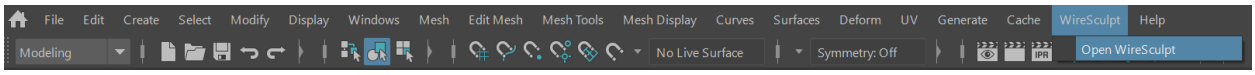
4.1.1 Documentation/Tutorial

In order to load in the plugin, first you need to make sure you have Maya 2022 or Maya 2024 installed (we have not tested this plug in for other versions). We also recommend Visual Studio 2022 to build the plugin.

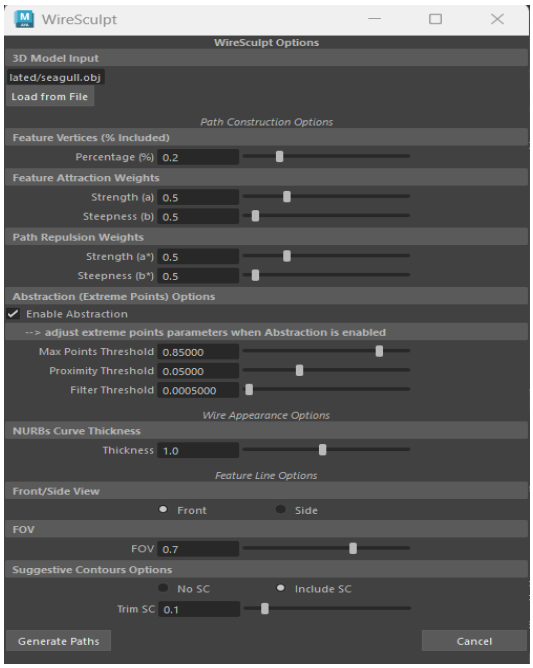
Clone our repo, then open it in Visual Studio in admin mode (without this, the project will not build properly). Once you build the project, the .mll file will be added to the debug folder of the project. To load the plugin, open your version of Maya and go to Windows->Settings/Preferences->Plug-in Manager and browse to open the file.



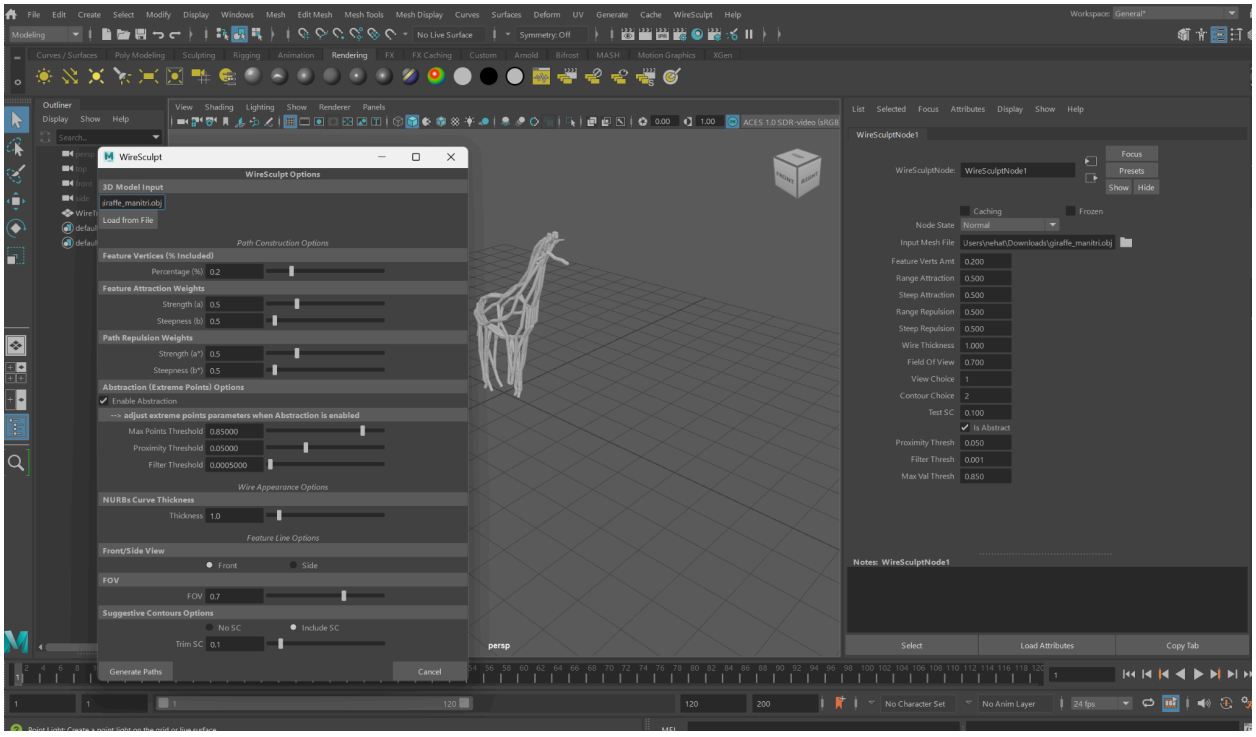
You can launch our GUI through the menu at the top of the Maya UI or by typing 'CreateWireSculpt;' into the script editor.



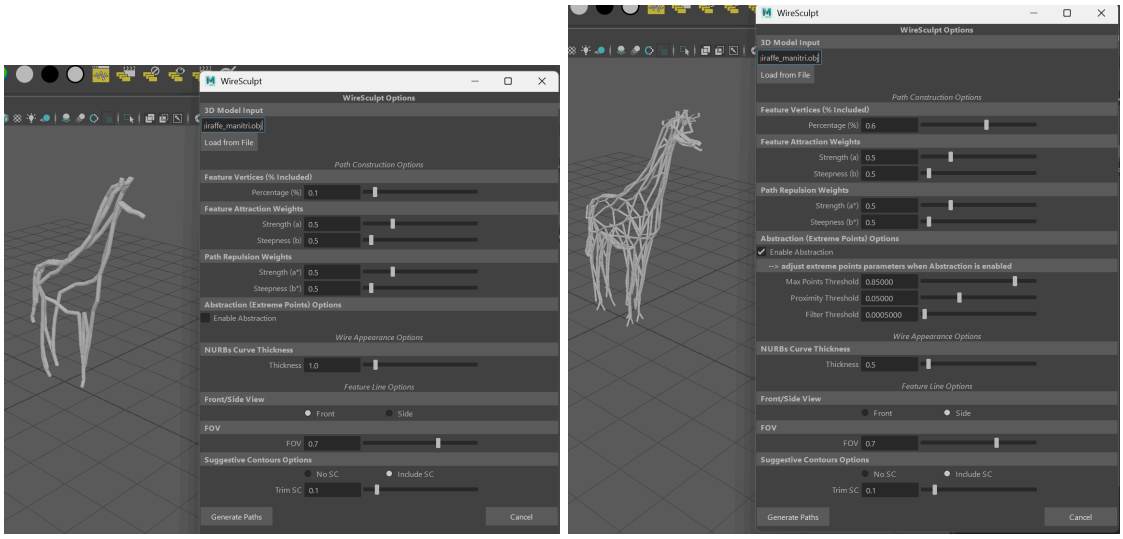
This will launch the GUI.



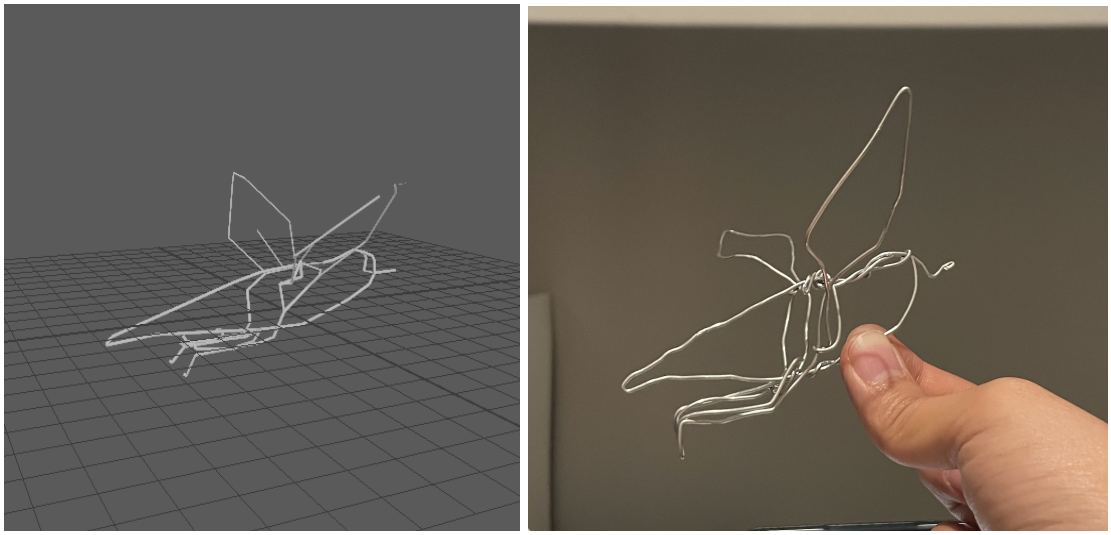
For an artist who seeks to use our tool for prototyping, they can input a 3D model reference, alter the parameters (more information in section 4.1.2), and they can click generate to see the wire output.



Due to the speed at which the wire outputs can be generated, an artist can quickly see multiple iterations. Here are some examples for the giraffe model:



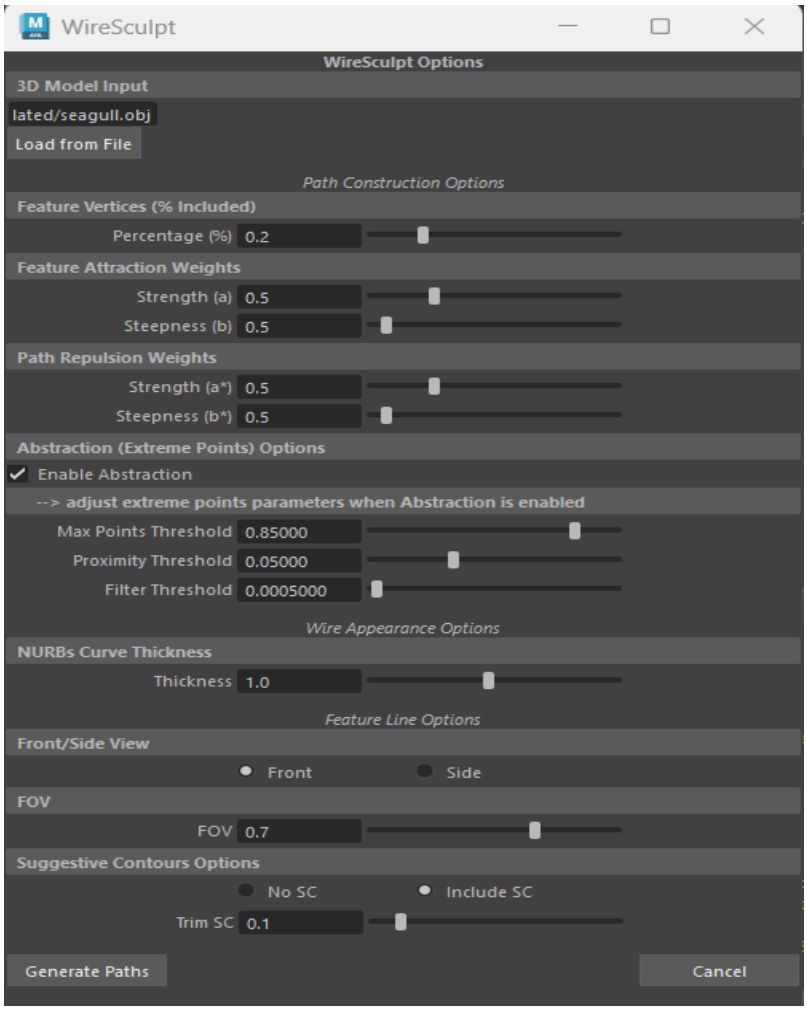
They can then use this as a basis for creating their own wire sculpture in real life! Like so:



Created by Claire and based on the seagull model.

4.1.2 Content Creation

To load a model into the plugin, click the load from file button. The 3D mesh has to be triangulated, manifold, and obj, or the plugin will not be able to process it.



The Abstraction option enables the wireframe sculpture to have a more abstract aesthetic by enabling more extreme points to be chosen as landmark vertices that get visited by the wire path algorithm. The extreme points parameters can be adjusted to change the vertices that are selected as extreme points.

- Proximity: Controls how close new extreme points can be to existing ones
- MaxVal: threshold for determining whether a vertex qualifies as an extreme point based on its Laplacian value relative to its neighbors
- Filter threshold: Controls how many areas are considered concave (concave vertices are not considered extreme)

The feature attraction weights determine how closely the path follows the features.

- Range (a): controls the strength of the attraction
- Steepness (b): controls the range/locality of the attraction

The path repulsion weights aid in avoiding repetitions of vertices in the path.

- Range (a*): controls the strength of the repulsion

- Steepness (b*): controls the locality of repulsion

NURBs Curve Thickness parameter adjusts the width of the output line**.**

Contours are view dependent so it's shaped by the perspective.

- Front: captures contours from the front of the model
- Side: captures contours from the side of the model

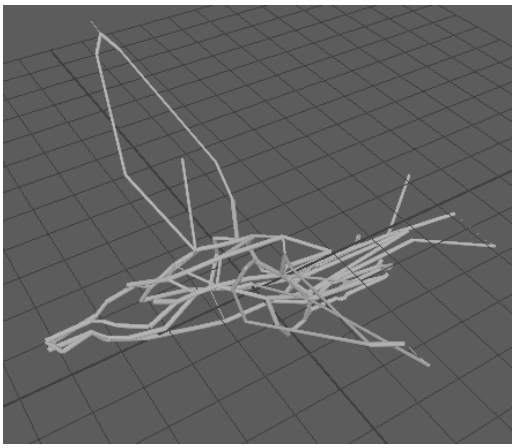
FOV (field of vision) is another parameter for affecting the look of the contours. It controls the field of view for which the occluding and suggestive contours get captured. By default, it is set at 0.7 which has worked well for most models.

Suggestive Contours can be switched on/off depending on wire output preference. By default, occluding contours and suggestive contours are used. Occluding contours capture the view-dependent silhouette of the model and suggestive contours capture finer details.

- Trim SC: the amount that the suggestive contours are trimmed.

The Feature Vertices option allows control over how much of the model's contours are captured in the wire path. A value of 1 adds all vertices from the contours to the path, making the wire sculpture match more closely with the model and appear more intricate, while 0 will cause the wire path to be solely constructed by the chosen extreme points, making the wire sculpture more minimalistic.

This would be an example of an output for the above parameters. Due to the inclusion of randomness for the feature vertices that are included there can be slight variations for different outputs of the same parameters.



4.2. Discussion

4.2.1 Accomplishments

Features implemented:

- Node setup code and GUI
- Set up all the necessary classes/data structures
- Processed the 3D model (read from the file and stored the necessary values: vertices and normals)
- Path traversal (A*)
- 2-Opt Approximation for Traveling Salesman Parameter
- Extreme points method integration
- Feature Lines/Contour Lines extraction
- Heat Map method integration
- Visual debugging (heat map colors, extreme points vertex highlighting, etc.)
- Aesthetic parameters (determining level of abstraction, feature line similarity)
- Wire output

All the features listed above work.

Features not implemented:

- Path traversal parameters
- Outputting multiple paths
- Spline curve fitting

The reason why we decided to omit outputting multiple paths is because we wanted to focus on making a single wire output look appealing to the user rather than dividing our attention and output a collection of wires that didn't look as aesthetic/clean. We also felt that by playing with the parameters we created above, the user could already achieve pretty different looks with just one single wire output and that having a collection of outputs might be more overwhelming for the user. Since the target user is an artist that works with traditional media like wire, they might not necessarily want too many parameters. The path traversal parameters are the parameters that control the output of multiple paths, so naturally we omitted that as well.

The reason why we omitted spline curve fitting is because we felt the wires looked nice enough that spline curve fitting wasn't as important as some of the other features we wanted to implement, like the abstraction and feature line similarity parameters. We think these parameters helped round out the tool to be able to capture a wider range of potential wire sculpture aesthetics, allowing more flexibility to artists in their creation/iteration process.

4.2.2 Design Changes

We originally planned to implement the heat map code by integrating source code from a github repository we found. However, after attempting to integrate it and finding some issues with the source code itself, we realized we needed a different approach. At this point, we found that libigl has a heat map implementation so we used this instead.

We also originally planned to just use A* to do all the path traversal steps but we realized that we wanted the output wire to be a single wire output that didn't have loose wire ends. This meant that we needed an algorithm to create a cycle so we decided to implement an approximation of the traveling salesman problem.

We had to make a slight adjustment to the extreme points method. The original method did not get all of the extreme points we wanted for some models (for example, missing the ears of the fox or the beak of the seagull). As such, we had to parameterize key parts of the algorithm: namely how vertices are classified as extreme points (max number and proximity) and determining which vertices are concave (if they are concave, they cannot be extreme).

4.2.3 Total Man-Hours worked

Claire: ~106
Neha: ~97

4.2.4 Future Work

We believe that the plug-in in its current state can be used by an artist to create wire sculptures with relative ease. However, future work could help in improving our plugin to be more versatile.

One possible idea that we had was to make a step by step process for how to create a wire sculpture based on the wire output. This would be automated and we could potentially show the path growing piece by piece from start to finish.
Another area for future work would be incorporating elements of the paper that we did not get a chance to address. This would be making a more interactive UI where the user can add extreme points manually as well as custom contours and these would be used in computing the output.
One other concept that could be worth addressing is making extreme points more generalized. The paper's implementation seemed to have found extreme points for any model without modifications to the algorithm while ours requires fine tuning parameters to get different sets of extreme points.

4.2.5 Third Party Software

Libigl

- We used the matrix/solver implementations as part of the extreme points implementation
- We used the heat map method from this library
- We used the read file method for extreme points and heat map

Extreme points implementation

- We used the repository in order to detect extreme points of meshes that are inputted by the user:
- We altered the code to parameterize the extreme points as the values used in the original implementation did not generalize to all inputs.

Feature lines (contours) detection

- We used the occluding contours and suggestive contours computation to determine the feature lines on the model.

Heat map method

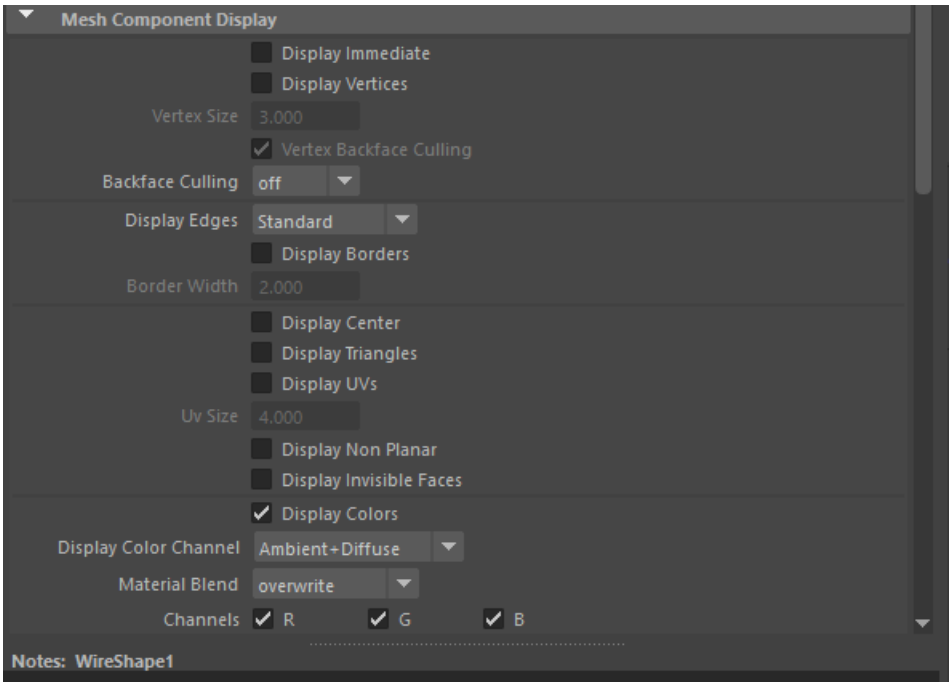
- We ultimately did not use this implementation due to finding a fully functional version of the method in Libigl; however the process of translating this to C++ was instrumental in understanding how the method worked.

2-Opt Traveling Salesman Approximation

- We referenced this repository for implementing a TSP approximation, although we did make changes to integrate it into our project.

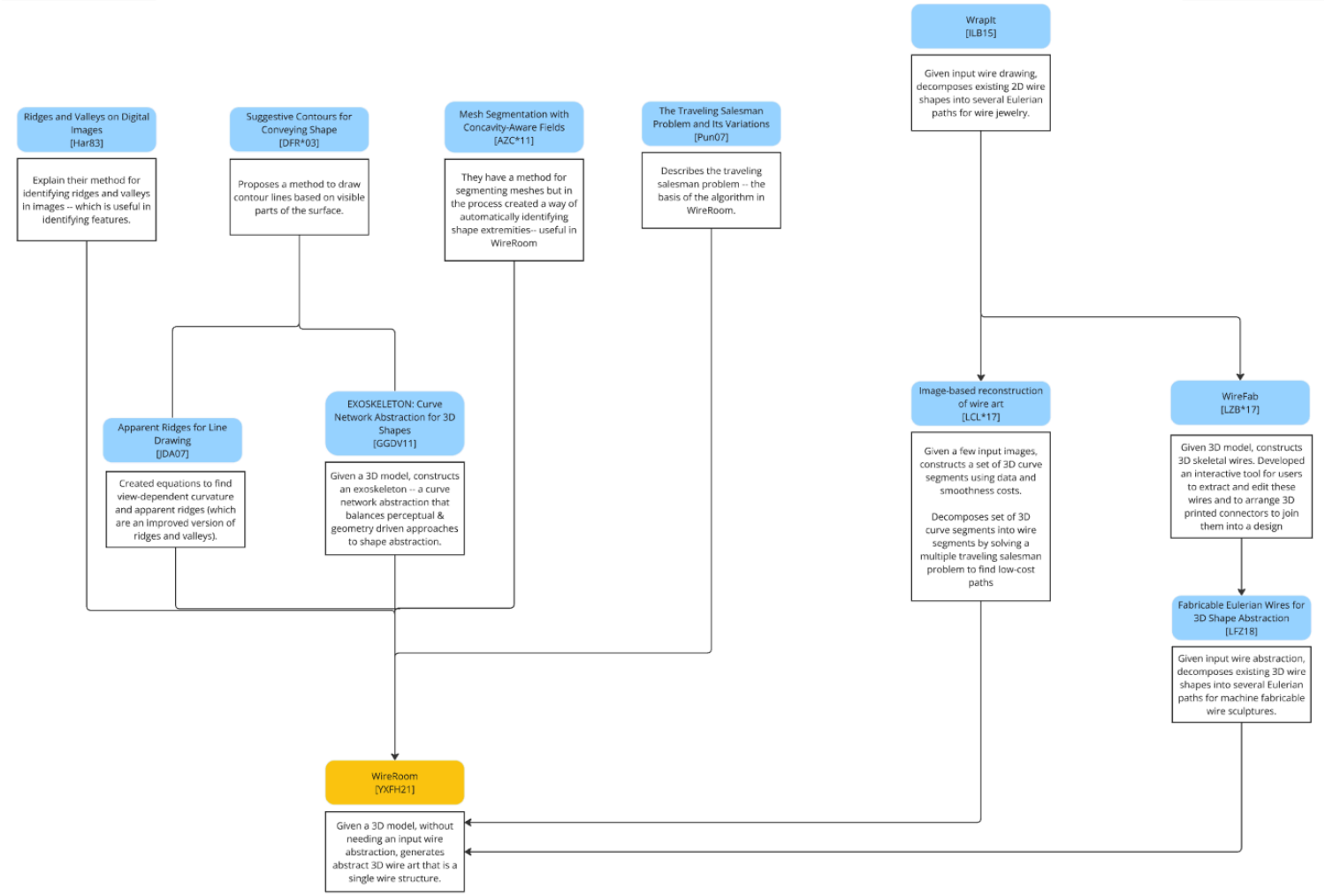
4.3 Lessons Learned

- One insight that would have been good to keep in mind is to check for necessary functionality in existing libraries before implementing it yourself. For example, the heat map method was available in libigl but we only realized this because we already integrated heat map and it wasn't outputting the expected results.
- Another insight is to check that the implementations you're integrating work properly. For example, at the beginning, we pulled the heat map code from a repository we found on Github, but after integrating it we found that the outputs gave us unexpected results. We tried building it and running it and found that the results weren't always consistent in the original project either.
- We would advise trying to visualize your implementation every step of the way. This helped us catch bugs early on before we implemented more of the next steps.
- For visualization, we found it helpful to display geometry in different colors. This thread gave some insight into the syntax for this.
    - Also keep in mind that certain Maya display options might need to be set in order to see colors. We needed to check 'Display Colors' in the Mesh Component Display options below.

- 
Mesh Component Display
Display Immediate
Display Vertices
Vertex Size
3.000
Vertex Backface Culling
Backface Culling
off
Display Edges
Standard
Display Borders
Border Width
2.000
Display Center
Display Triangles
Display UVs
Uv Size
4.000
Display Non Planar
Display Invisible Faces
Display Colors
Display Color Channel
Ambient+Diffuse
Material Blend
overwrite
Channels
R
G
B
Notes: WireShape1

4. RELATED RESEARCH

Evolution of Research



Description of Prior Research

[ILB15] In this seminal work, Iarussi and his team describe a computational design tool for novice wire-wrapped jewelry artists. Developed to empower artists to create wire jewelry from their own designs, the tool takes in a line drawing and generates a set of instructions to facilitate the execution of the design, as well as a custom 3D-printed tool for helping shape the wire. They

formulate their problem statement into two main challenges: wire decomposition and wire bending.

Ideally, wire designs would be decomposed into pieces that consist of single wires (wires that double back on themselves are not as aesthetically pleasing), as few wires as possible, and smooth bends (sharp angles are difficult to form). These ideas get carried through many subsequent abstract wire structure work involving the fabrication of 3D wire sculptures [LFZ18], [LZB*17], [LCL*17], and the basis of our authoring tool, [YXFH21]. To aim in wire bending, they generate custom *jigs* – tools used by jewelry makers for wrapping write, which helps enable the fabrication process. Similarly, many other sculpture works take into account fabrication and assembly constraints in developing their tools, such as [LFZ18] and [LZB*17].

While many ideas presented in this paper have influenced future works greatly, their main contribution is the wire decomposition algorithm that segments a drawing into a small number of fabricable wires. They do so by segmenting the input drawing into a number of smaller sub-graphs such that each sub-graph can be traversed by a wire exactly once, thus forming an Eulerian path. This involves abstracting the input shape to make it fabricable with wires. Although methods for shape abstraction existed, there had previously been no methods that factored in aesthetic and fabrication constraints in jewelry making.

Since Iarussi and his teammate's seminal work, many other computational wire sculpture tools have been proposed, proposing solutions to different problem spaces. Some concern the fabrication of 3D wire sculptures [LFZ18], developing an interactive tool for extracting wires and arranging connectors [LZB*17], while others center around the reconstruction of wire art from input images [LCL*17].

[LZB*17] WireFab is a prototyping system designed to facilitate the fabrications of poseable 3D models. In this paper, Liu and her team aim to lower the barrier to design and fabrication by enabling rapid prototyping, integrating wire bending software with personal fabrication technology such as laser cutting and 3D printing. In this way, they borrow ideas from WrapIt [ILB15], which generates 3D printed support structures to help users bend wires into shape.

The input to their system is a 3D model, from which a smooth curve gets abstracted. This curve is then used as a skeleton which is the basis for the rig. The input mesh gets further segmented manually by the user and joint placements for the model are specified. The end output is the generated connectors, joints, and appearances that align with the fabrication model.

These ideas, including the extraction of an abstract curve network from a 3D model and a modeling framework tool that considers physical fabrication constraints, reappear in subsequent wire sculpture works such as one paper concerning machine fabricable wire sculptures [LFZ18] and also the basis for our authoring tool, WireRoom [YXFH21].

[LFZ18] In this paper, Lira and his team draw from both of the previously mentioned works, WrapIt [ILB15] and WireFab [LZB*17]. They develop a system that automatically constructs a fabricable 3D wire sculpture consisting of a small number of wires with minimal overlap given an input wire abstraction. Inspired by the algorithm presented in WrapIt for decomposing a 2D wire piece into several segments, they extend this idea to their 3D wire abstraction, considering non-planar wires [ILB15]. The added dimension significantly increases the computational complexity. Similar to how WrapIt factors in fabrication in their tool by generating a 3D support system [ILB15], Lira and his team greatly consider fabrication, using a 3D wire bender that automatically bends the wire. In fact, this wire bender is the same one used in WireFab

[LZB*17]. However, the difference lies in the use of the wire bending machine – in WireFab, the machine was used to output a large number of wire segments that would be joined by connectors to allow for joint movement [LZB*17], whereas their use case involves an automatic method to compute near-Eulerian paths of a wire-sculpture design. Thus, they cite WrapIt as the most relevant previous work.

Their main contribution is the fully automatic method of constructing 3D wire sculptures. They are able to achieve a variety of 3D shapes by constructing Eulerian paths. The basis for our authoring tool, WireRoom, achieves a similar goal in generating abstract 3D wire shapes, requiring little user skill (users can manually refine generated wire art if they wish) [YXFH21]. However, WireRoom uses a completely different approach. Instead of taking in a wire abstraction, they take in a 3D model from which they determine landmark points and perform a dynamic traveling salesman problem [YXFH21]. Even so, WireRoom cites Fabricable Eulerian Wires [LFZ18] along with WrapIt [ILB15] as most relevant as they tackle the problem of decomposing 3D or 2D wire shapes into distinct paths for fabrication, signifying major developments in the line of abstract wire sculpture work.

[LCL*17] A slightly different problem setting is examined in this paper, where Liu and her team present an image based reconstruction process of wire art. They build on previous works related to modeling curve-based structures and modeling delicate structures from images. WrapIt proposes extracting and decomposing wire segments from an input drawing [ILB15]. However, Liu's image-based reconstruction differs in that they do not assume curves are planar or that the full view can be captured in a single image. Their main contribution is the ability to extract delicate and wiry topology from only several input images (as little as 3), and reconstruct a wire

sculpture. While the paper for our authoring tool utilizes 3D models rather than a series of images to form the basis of its abstract wire art, the 3D reconstruction of wire shapes presented in this work provides relevant context in the development of abstract wire art systems [YXFH21].

Notably, in order to decompose the set of 3D curve segments (obtained after several computational steps) into a set of distinct, continuous wires, they utilize the multiple traveling salesman problem. They define each of the curve segments as nodes in a directed graph and represent a continuous 3D wire by a path in the graph. They assign costs to edges and attempt to minimize the total cost of the edges and the total number of paths, observing that connecting many pieces of wires is nontrivial and bending wires at sharp angles is difficult, most of the time necessitating more than one wire.

WireRoom [YXFH21] similarly uses a variation of the traveling salesman problem, referencing [Pun07] as the basis for a dynamic travelling salesman problem. However, they use a different approach than the multiple traveling salesman problem in this image-based paper, first extracting landmark sites following [Au et al. 2011] and then defining an optimal path as one that passes all landmark sites. These ideas will be examined further in discussion of the traveling salesman problem presented in [Pun07] and the shape extremity extraction in [AZC*11].

[AZC*11] focuses on an automatic mesh segmentation method based on shape concavity. As a part of their algorithm, they create a method to automatically identify shape extremities which is instrumental in WireRoom's [YXFH21] process of creating abstract wires. Au and their team were inspired by a method from [ZSC*08] which computes an average squared geodesic distance field and the extremities in this method are found by using an iterative poisson disk

sampling scheme. However, they found this method to be too inefficient and instead they propagate a field from a point to the rest of the shape using BFS. They identify the points with local maximum/minimum field values as extremities. For models that do not have clear extrusions, they fall back to the geodesic method. WireRoom [YXFH21] utilized this method as a part of its workflow to determine the landmark sites which are then passed on to the TSP method.

[Pun07] thoroughly dissects the traveling salesman problem from the mathematical explanation, to variations of TSP, and even applications. In the context of WireRoom [YXFH21], the focus is on the problem itself. Punnen defines the traveling salesman problem as a graph where each edge has a cost/weight and the goal is to find a Hamiltonian cycle, a cycle where each of the nodes is visited only once, while keeping the sum of the costs as small as possible. This problem allows for one of the main criteria of [YXFH21] to be satisfied. That is to say, their goal is for the output to be a single connected wire in order to ensure feasibility in the fabrication process-if one were to choose to recreate the output wire in real life. TSP is rescoped in the context of wire generation within [YXFH21]. Namely, the nodes refer to the landmark sites from the algorithm in [AZC*11] and the edges are part of the input triangle mesh. However, they found that this method needs weights in order to ensure that their paths are attracted to desired feature lines of the input model. They utilized line drawing techniques in [Har83], [DFR*03], and [JDA07] in order to compute these weights.

[Har83] defines ridges as areas of an image where the surface normals are in the same direction as the light source and valleys as concavities that typically fall in shadow. These definitions are

both in the context of analyzing pixels in images as Haralick's algorithm was created with computer vision in mind. He identifies ridges and valleys by computing the directional derivative of pixel values with respect to neighboring gradients. Using the resultant matrices, it is then possible to extract ant draw lines based on these features.

[YXFH21] extends this concept by applying ridge and valley detection based on the surface of an input 3D model and utilizing the resultant lines for their weights.

[JDA07] proposes a non-photorealistic system for rendering lines. Namely, they introduce a new type of line called suggestive contours, a type of line drawn on clearly visible parts of a model, as opposed to true contours, which form on surfaces that bend sharply away from the viewer. They provide multiple definitions for suggestive contours. The first definition, which is derived in [Koe84], identifies them as curves where the radial curvature, the measurement of a curve in a radial plane, is zero and the surface bends away from the viewer. The second defines them as points on the surface that are nearly contours, which means that the dot product between the surface normal and the view vector is almost a zero (a local minimum). They also exclude suggestive contours that appear as contours from farther perspectives but lack corresponding contours in closer views.

[JDA07] proposes two main approaches for solving for suggestive contours: directly identifying regions in the mesh where the radial curvature is equal to 0 and analyzing a rendered image instead of using the mesh. For the purposes of [YXFH21], iterating through the mesh to identify suggestive contours would be more of a feasible approach in computing feature weights.

[DFR*03] introduces a method of line drawing where they render apparent ridges, which are a set of points where a maximum view-dependent curvature, how much a surface bends from the viewpoint, is locally maximal in the principal view-dependent curvature in direction t. To find apparent ridges, they iterate through each edge and adjust t such that it points towards the increasing view-dependent curvature. They then detect zero crossings and determine if it is a local maximum by checking if a vertex is perpendicular to the zero crossing line. If it is, then it is treated as an apparent ridge.

They compare their method to ridges and valleys, which they define earlier in the paper. Although they do not explicitly reference [Har83], their approach to deriving ridges and valleys is similar. The two methods share the same foundational definition, but apparent ridges yield different results for areas of the model that are turned away from the viewpoint. This approach also captures ridges that are not well-defined in the traditional ridges/valley method due to an absence of maximum curvature.

Additionally, they compare apparent ridges to suggestive contours from [JDA07]. While contours consider the direction of the view vector, apparent ridges rely on the direction of the maximal normal variation (t). If these two directions are the same, suggestive contours classify them as zero (and thereby removing them), whereas apparent ridges treat them as maxima (considering them as ridges). The researchers find that the two models are rather different from one another and as such each have their own strengths and weaknesses. This distinction likely influenced [YXFH21] to incorporate both methods in their own feature weight calculation process.

[GGDV11] This paper introduces the concept of an exoskeleton in the abstraction of the shape of a 3D model. It utilizes segmentation and shape approximation. Its major contribution is in developing a framework for capturing geometric structure and global appearance of 3D shapes. They cite previous work in 3D shape abstraction, such as non-photorealistic rendering techniques that utilize view-dependent features like suggestive contours in [DFR*03]. While this paper does not specifically use this approach, these ideas are built on further in WireRoom [YXFH21].

[YXFH21] WireRoom proposes a framework for designing abstract 3D wire art that is composed of a single-wire structure. Their method is made with the intention of being able to be reproduced by artists through traditional means.

WireRoom draws from many previous works and cites WrapIt [ILB15]and Fabricable Wires [LFZ18] as the most relevant previous works. These two works decompose existing 2D or 3D, respectively, wire shapes into a distinct set of Eulerian paths for the purpose of wire sculpture fabrication. These two works set a standard and are used as evaluation metrics for WireRoom's output. However, while WireRoom has a similar goal, they do not require an existing wire shape as input as these works do, and thus use a different approach.

In order to allow for ease in the fabrication process of wires generated by WireRoom, the algorithm is primarily based on the dynamic traveling salesman problem with dynamic distance measurement (DTSP) and references [Pun07]. This ensures that their resulting wire is one connected piece. They pull from the automatic landmark sites identifier algorithm in [AZC*11] to ensure that the path calculated by DTSP traverses the extreme points of the mesh and thus resembles the mesh. However, these two algorithms alone did not yield the desired wire output and they needed to add feature weights in order to ensure that the shape of the wire more closely

mirrored their feature lines. This was possible by drawing from the line drawing techniques in [Har83], [DFR*03], and [JDA07]. They can then combine these feature attraction weights with a set of path repulsion weights in their TSP. The resulting computation yields a set of desired abstract wire paths.

References

[AZC*11]

O. Kin-Chung Au, Y. Zheng, M. Chen, P. Xu and C. -L. Tai, "Mesh Segmentation with Concavity-Aware Fields," in IEEE Transactions on Visualization and Computer Graphics, vol. 18, no. 7, pp. 1125-1134, July 2012, doi: 10.1109/TVCG.2011.131.

[DFR*03]

Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. 2003. Suggestive contours for conveying shape. ACM Trans. Graph. 22, 3 (July 2003), 848–855. https://doi.org/10.1145/882262.882354

[GGDV11]

Fernando De Goes, Siome Goldenstein, Mathieu Desbrun, and Luiz Velho. 2011. Technical Section: Exoskeleton: Curve network abstraction for 3D shapes. Comput. Graph. 35, 1 (February, 2011), 112–121. https://doi.org/10.1016/j.cag.2010.11.012

[ILB15]

Emmanuel Iarussi, Wilmot Li, and Adrien Bousseau. 2015. WrapIt: computer-assisted crafting of wire wrapped jewelry. ACM Trans. Graph. 34, 6, Article 221 (November 2015), 8 pages. https://doi.org/10.1145/2816795.2818118

[Koe84]

Koenderink, Jan. J., 1984. What does the occluding contour tell us about solid shape? *Perception 13*, 321-330.

[JDA07]

Tilke Judd, Frédo Durand, and Edward Adelson. 2007. Apparent ridges for line drawing. ACM Trans. Graph. 26, 3 (July 2007), 19–es. https://doi.org/10.1145/1276377.1276401

[Har83]

Robert M Haralick. 1983. Ridges and valleys on digital images. Computer Vision, Graphics, and Image Processing, Volume 22, Issue 1, Pages 28-38. ISSN 0734-189X. https://doi.org/10.1016/0734-189X(83)90094-4.

[LCL*17]

Lingjie Liu, Duygu Ceylan, Cheng Lin, Wenping Wang, and Niloy J. Mitra. 2017. Image-based reconstruction of wire art. ACM Trans. Graph. 36, 4, Article 63 (August 2017), 11 pages. https://doi.org/10.1145/3072959.3073682

[LFZ18]

Wallace Lira, Chi-Wing Fu, and Hao Zhang. 2018. Fabricable eulerian wires for 3D shape abstraction. ACM Trans. Graph. 37, 6, Article 240 (December 2018), 13 pages. https://doi.org/10.1145/3272127.3275049

[LZB*17]

Min Liu, Yunbo Zhang, Jing Bai, Yuanzhi Cao, Jeffrey M. Alperovich, and Karthik Ramani. 2017. WireFab: Mix-Dimensional Modeling and Fabrication for 3D Mesh Models. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17). Association for Computing Machinery, New York, NY, USA, 965–976. https://doi.org/10.1145/3025453.3025619

[Pun07]

Punnen, A.P. (2007). The Traveling Salesman Problem: Applications, Formulations and Variations. In: Gutin, G., Punnen, A.P. (eds) The Traveling Salesman Problem and Its Variations. Combinatorial Optimization, vol 12. Springer, Boston, MA. https://doi.org/10.1007/0-306-48213-4_1

[ZSC*08]

Zhang, H., Sheffer, A., Cohen-Or, D., Zhou, Q., Van Kaick, O. and Tagliasacchi, A. (2008), Deformation-Driven Shape Correspondence. Computer Graphics Forum, 27: 1431-1439. https://doi.org/10.1111/j.1467-8659.2008.01283.x

[YXFH21]

Zhijin Yang, Pengfei Xu, Hongbo Fu, and Hui Huang. 2021. WireRoom: model-guided explorative design of abstract wire art. ACM Trans. Graph. 40, 4, Article 128 (August 2021), 13 pages. https://doi.org/10.1145/3450626.3459796