REMESHING EULERIAN-ON-LAGRANGIAN STRANDS

An Undergraduate Research Scholars Thesis

by

KEVIN JIANG

Submitted to the Undergraduate Research Scholars Program at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by Research Advisor: Dr. Shinjiro Sueda

May 2020

Major: Computer Science

TABLE OF CONTENTS

ABSTRACT

Remeshing Eulerian-on-Lagrangian Strands

Kevin Jiang
Department of Computer Science and Engineering
Texas A&M University

Research Advisor: Dr. Shinjiro Sueda
Department of Computer Science and Engineering
Texas A&M University

Physics-based simulation of strands is an important and well-studied topic within the field of computer graphics. One particular case, in which a strand is bending and sliding around a sharp corner, has been a challenge to simulate due to the additional constraints that are involved. Many of the previous methods for simulating strands do not perform well with strands crossing and sliding with respect to each other. In this research, we have developed a formulation that is capable of simulating and remeshing bending and sliding strands that combine the traditional Lagrangian method of physics-based simulation with an Eulerian approach. We found that our program is able to support dynamic remeshing of an Eulerian-on-Lagrangian strand when it is bending around a sharp corner, which provides a more accurate simulation. This could be extended to more complex simulations, such as the simulation of Ayatori (string art or string figures). Additionally, there are potential engineering applications involving cables.

ACKNOWLEDGMENTS

I would like to thank my research advisor, Dr. Sueda, for his patience and guidance throughout the year. He has always been willing to answer my questions and point me in the right direction when I was unsure. Without him, none of this work would have been possible.

CHAPTER I

INTRODUCTION

Simulating strands is one topic of interest within the field of computer graphics. One method in which this can be achieved is defined as Eulerian-on-Lagrangian strands, which we will refer to as EOL strands, by Sueda et. al. [2011]. These EOL strands build off the traditional Lagrangian description of a strand by adding an Eulerian component to each node, which provides a better simulation of a strand stretching and bending. Our research focuses on accurately simulating the case in which an EOL strand bends and slides around a sharp corner. One potential application of this program is to simulate ayatori. Ayatori (also known as string art or string figures) is a game in which one or more people use their hands to manipulate a loop of string to form various designs. The patterns created can range from simple shapes to complex patterns that require many steps to execute. As our research deals with bending strands, it could be extended to deal with ayatori, which would involve a strand bending and crossing around itself. There are also engineering applications involving cables.

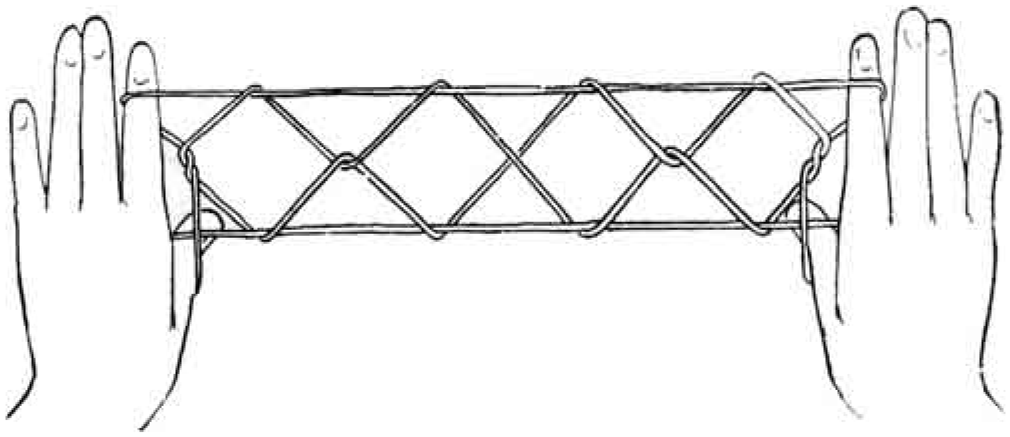

Figure 1: One pattern that could be made in a game of Ayatori [1]

Previous Work

There are many pertinent research materials within this field. Many of these deal with elastic rods, which are curve-like elastic bodies that have a length much larger than its cross-section. Bergou et al. [2008] presented an approach for modeling elastic rods using quasistatics and constraints [2]. Batty et al. [2012] later extended this method to simulate the dynamics of thin sheets of viscous incompressible liquid [3]. Bertails et al. [2006] showed that the equations for dynamic, inextensible elastic rods can be used for predicting hair motion [4]. Umetani et al. [2014] introduced a new technique to simulate elastic rods in Position-Based Dynamics framework [5].

Importantly, none of the previous research mentioned do well with cables crossing and sliding with respect to each other. Simulating cables or strings can be difficult, since the traditional Lagrangian method cannot fully account for the case where the string must bend around another object. As seen in the image below, the part of the string which bends around the corner of the table is not entirely natural, because it lies in between the nodes for which motion is accounted for. Remeshing the strand also leads to undesirable artifacts, since the remeshed node cannot slide around the corner without producing more interpenetrations. Therefore, another method must be used in order to properly simulate strings sliding and bending.

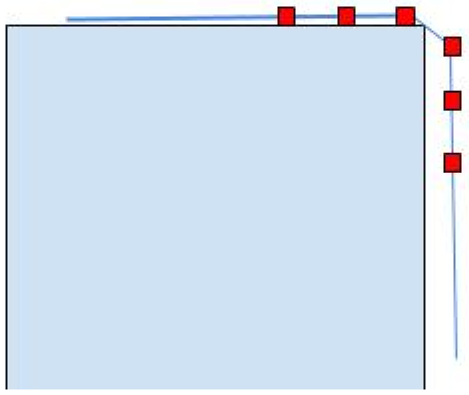

Figure 2: Example of a simulation of a string sliding over a table using the Lagrangian method

One important research material to note is the Large-Scale Dynamic Simulation of Highly Constrained Strands by Sueda et al. [2011], in which they introduce a new framework to overcome difficulties in simulating highly constrained strand-like physical curves. In summary, the method combines both the traditional Lagrangian approach and an Eulerian approach, which results in "robust, efficient, and accurate simulations of massively constrained systems of rigid bodies and strands [6]." We extend this approach, adding support for dynamic remeshing.

CHAPTER II

METHODS

We developed our program in C++, using the OpenGL API. Additionally, we used other libraries including Eigen and GLM to perform the necessary matrix calculations. To start off, we decided to use code from an existing cloth simulator provided by Dr. Shinjiro Sueda, so that we could have a framework of code to start developing in.

Nodes and Edges

For our program, we use nodes to represent the different points of a strand. Each node has its own mass, position, and velocity, which we will refer to as m_i, $\mathbf{x}_i$, and $\mathbf{v}_i$ respectively. A strand can be drawn by simply using linear interpolation to create lines between each node. By storing the positions and velocities of each node in a vector, we can also calculate their positions over time by solving the matrix equation $\mathrm{M}\dot{v} = \mathbf{f}$. Using the method of Sueda et al. [2011], we also add an s component to each of the nodes. This s component, called the Eulerian coordinate, represents the coordinates of the material of the strand; in the code, it is essentially the texture coordinates. This allows the material to move independently of the node's world coordinates.

We then define two types of nodes. A Lagrangian node, or L-node, refers to a node in which s has been constrained, allowing only the $\mathbf{x}$ to change. We call it this because its motion is calculated using the Lagrangian description, in which the behavior of particles within a system is calculated by solving equations related to their position and velocity. The second type of node we will define is the Eulerian node, or E-node. For this node, the $\mathbf{x}$ is constrained, leaving the s to freely move. This is similar to the Eulerian specification in fluid dynamics, which focuses on observing the flow over time at a specific point.

To simulate a strand that properly stretches and bends, we require additional forces aside from gravity. We achieve this by defining a spring between each node, with a given tension to control how much the strand is allowed to stretch. This introduces spring forces between each of the nodes. We can calculate these spring forces by looking at the current length of the edge compared to the original length of the edge and apply these forces to each node. Adding the springs allows the strand to stretch and bounce, creating a more realistic simulation.

Texturing

To create the initial visual for the strand, we simply drew a line between each node. However, this was not enough when we began to implement the s components. Since the s components describe the reference coordinates of the material, we needed a proper texture so that we could see the changes in s. To create a surface for texturing, we draw a cylinder in between each node, rotated so that the flat faces correspond to the positions of the nodes. This creates a smooth, three-dimensional strand with proper shading and appearance. To texture it, we use a purple and black checkerboard pattern, to make it easier to debug changes in s. Since the texture coordinates are directly related to the s components, it is easy to notice changes through visual inspection.

Finally, we add spheres at each node so that it is easy to tell where each node is. We use red spheres to denote E-nodes and blue spheres to denote L-nodes. We also add smaller spheres along the strand at specific intervals of s. This allows us to determine whether an issue with the program is related to the texture or the s components by checking whether the texture behaves the same way as the spheres. Figure 3 shows what the strand looks like with the debug features.

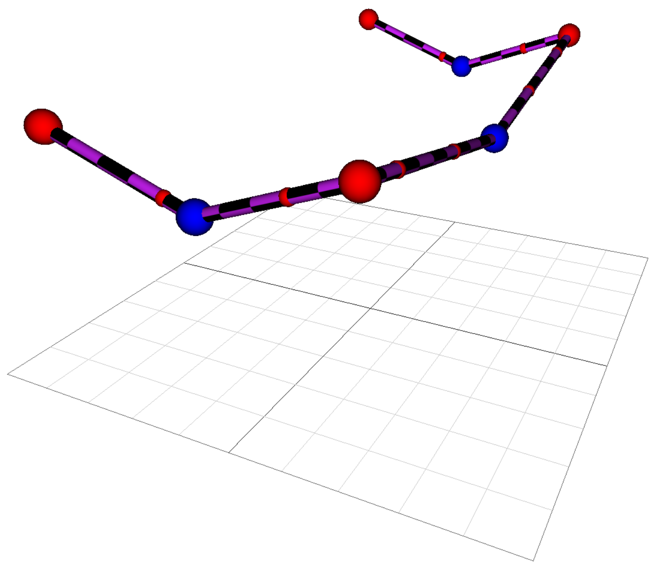

Figure 3: Debug texture of strand with L-nodes, E-nodes, and s markers. The red nodes represent E-nodes, while the blue nodes represent L-nodes.

Matrix Equations

To determine the motion of each node in the strand, we must solve the following matrix equation:

$$\mathbf{A}\mathbf{v} = \mathbf{b} \tag{Eq. 1}$$

Using implicit Euler, we get:

$$\mathbf{A} = \mathbf{M} + \mathbf{D}$$

$$\mathbf{b} = \mathbf{M}\mathbf{v}_0 + h\mathbf{f}$$

$\mathbf{M}$ is the mass matrix and $\mathbf{D} = \alpha h\mathbf{M} + \beta h^2\mathbf{K}$ is the Rayleigh damping matrix. α and β are the damping coefficients, h is the time step, and $\mathbf{K}$ is the stiffness matrix for the springs. Given n nodes in a strand, $\mathbf{M}$, $\mathbf{D}$, and $\mathbf{K}$ are all size $4n$ by $4n$. Similarly, $\mathbf{v}$ and $\mathbf{f}$ are also of length $4n$.

$\mathbf{f}$ is a vector containing all the forces that act on each node. For our simulation, we only deal with the spring force from each edge and gravity to make it simpler. The force due to gravity is easy to calculate. The only addition we must make is the Jacobian below.

$$\mathbf{J}_i = \left(\mathbf{I_3} - \mathbf{F}_i\right) \tag{Eq. 2}$$

With this addition, the vector we add to $\mathbf{f}$ becomes $\mathbf{J}^T\mathbf{f}_g$. $\mathbf{J}_i$ is the Jacobian for the nodes on each end of the spring. To get the $\mathbf{F}$ that is in the Jacobian for a particular node p, we calculate the $\mathbf{F} = (x_1 - x_p)/(s_1 - s_p)$ for each node incident to p, then take the weighted average of the two (or one if the node is on the end) values. Forming the Jacobian this way allows us to factor s into our calculations. Similarly, the spring force is calculated as below:

$$\mathbf{f}_s = E(l - L)\frac{\Delta x}{l}$$

$$\mathbf{f} = \begin{pmatrix} \mathbf{J}_0^T\mathbf{f}_s \\ -\mathbf{J}_1^T\mathbf{f}_s \end{pmatrix}$$

The spring force is then added to $\mathbf{f}$ in the same manner.

We can fill $\mathbf{M}$ simply by setting the diagonal elements to the mass of each node. $\mathbf{K}$, however, is more complex. Below is the formulation of $\mathbf{K}$:

$$\mathbf{K}_{local} = \begin{pmatrix} \mathbf{J}_0^T\mathbf{K}_s\mathbf{J}_0 & -\mathbf{J}_0^T\mathbf{K}_s\mathbf{J}_1 \\ -\mathbf{J}_1^T\mathbf{K}_s\mathbf{J}_0 & \mathbf{J}_1^T\mathbf{K}_s\mathbf{J}_1 \end{pmatrix} \tag{Eq. 3}$$

where

$$\mathbf{K}_s = \frac{E}{l^2}\left(\left(1 - \frac{l-L}{L}\right)(\Delta x \Delta x^T) + \frac{l-L}{l}(\Delta x^T \Delta x)\,\mathbf{I}_3\right)$$

$$\mathbf{J}_i = \begin{pmatrix} \mathbf{I_3} & -\mathbf{F}_i \end{pmatrix}$$

$E =$ modulus of elasticity

$l =$ current length of spring

$L =$ unstretched length of spring

$\Delta x = \mathbf{x}_1 - \mathbf{x}_0$

To fill $\mathbf{K}$, we loop through each edge and compute the $\mathbf{K}_{local}$ matrix each time. Each edge is attached to two nodes, so we can obtain the spring length by subtracting the positions of the two nodes. The Jacobian is used in the same way as in Eq. 2, allowing us to factor s into our calculations. Once we have the value of $\mathbf{K}_{local}$, we add it to the global $\mathbf{K}$ matrix by adding it at the indices corresponding to the nodes on each end of the spring.

The final part we add to the equation is a method to constrain certain nodes. This allows us to specify which nodes should be constrained, as well as what components should be fixed. We could simply skip the calculation for each node that we want to be fixed; however, this method is not easily extensible. A more general constraint method is needed: one that can specify a node to be either fixed or moving at a constant velocity. To achieve this, we form a KKT system [7]:

$$\begin{pmatrix} \mathbf{A} & \mathbf{G}^T \\ \mathbf{G} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{v} \\ \lambda \end{pmatrix} = \begin{pmatrix} \mathbf{b} \\ \mathbf{0} \end{pmatrix} \qquad \text{(Eq. 4)}$$

The first thing to notice is the addition of the $\mathbf{G}$ matrix. We formulate $\mathbf{G}$ in such a way that, after solving the matrix equation, the velocities of the constrained nodes equal the ones that we specified. For example, if our $\mathbf{G}$ matrix looks like this:

$$\mathbf{G} = \begin{pmatrix} \mathbf{I}_4 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I}_4 \end{pmatrix} \qquad \text{(Eq. 5)}$$

then solving the matrix equation gives us a velocity of 0 for the first and the third nodes (the order of the rows in $\mathbf{G}$ do not matter). We could also constrain the velocity to be a constant by filling the bottom half of $\mathbf{b}$ with nonzero values. This would move the particular nodes at a constant velocity.

To solve this equation, we used the SparseQR solver from the Eigen library, which implements a left-looking rank-revealing QR decomposition of sparse matrices. Even though it is slower than other sparse solvers, the result it produced was more stable, which is why we decided to use it. To get the new position of each node, we multiplied the time step h by the velocities obtained from the solver and set those as the new positions in the nodes.

When implementing this in the program, we used a sparse matrix for the left side of the equation. Since all of this has to be calculated for each frame, it is important for the process of solving the matrix equation to be efficient. Although building a sparse matrix is computationally expensive, it is often more efficient to solve matrix equations using sparse matrices.

Remeshing

In order to have a strand that behaves properly, we must remesh the strand when two nodes are too close to each other. For example, assume we have an L-node p that is being dragged towards an E-node q that is adjacent to it. Since the position of q is constrained, the material of the strand is essentially being dragged through q. Eventually, p will be dragged very close to q. If the strand is not remeshed, then p will end up bouncing around q, but never actually passing through q like it should, producing something similar to the image below:

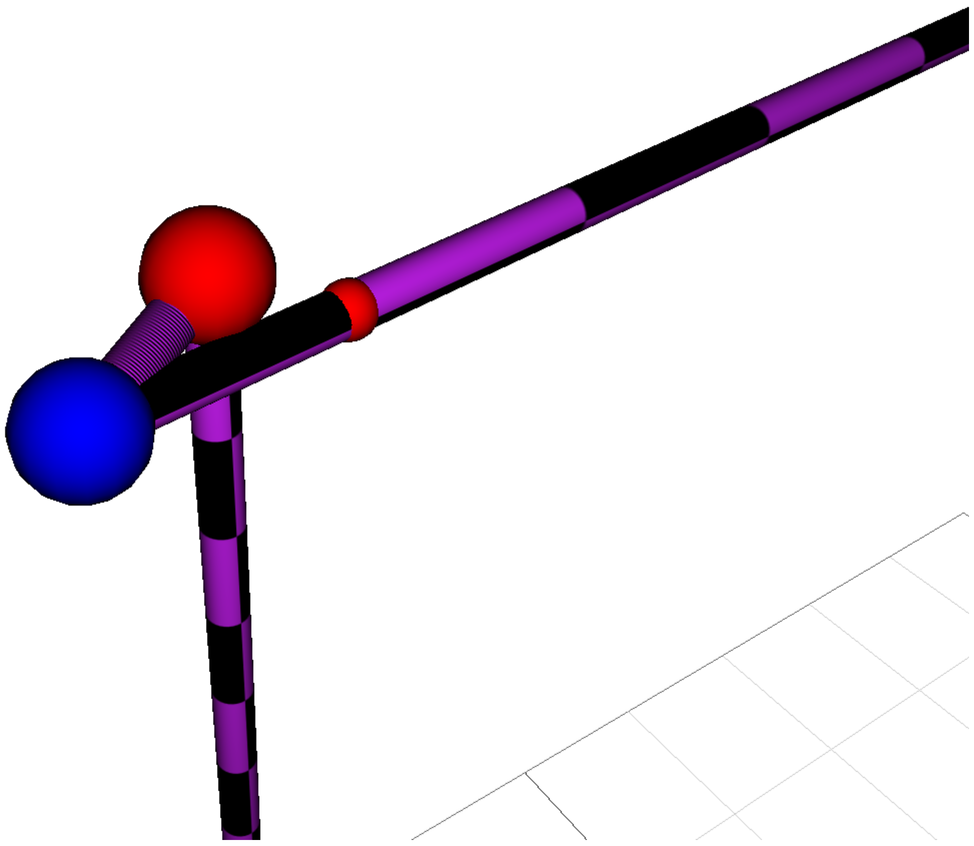

Figure 4: Result of two nodes that are too close without remeshing

To check if remeshing is required, we first check the s values for each node after each step of the program. if an L-node has an s value that is too close to one of an E-node, then that node is disabled. During the process of disabling the node, we must also remove it from all the edges that contain that node, as well as from the force and velocity vectors. After removing the node, we must insert it back in the correct position when it is far enough away from the original node that caused it to be disabled. Since the s for an L-node is constant, we can tell how far away it is from the E-node by calculating the difference in s for the disabled L-node and the E-node. Once the difference reaches the threshold, we re-insert the node into the system matrices and vectors.

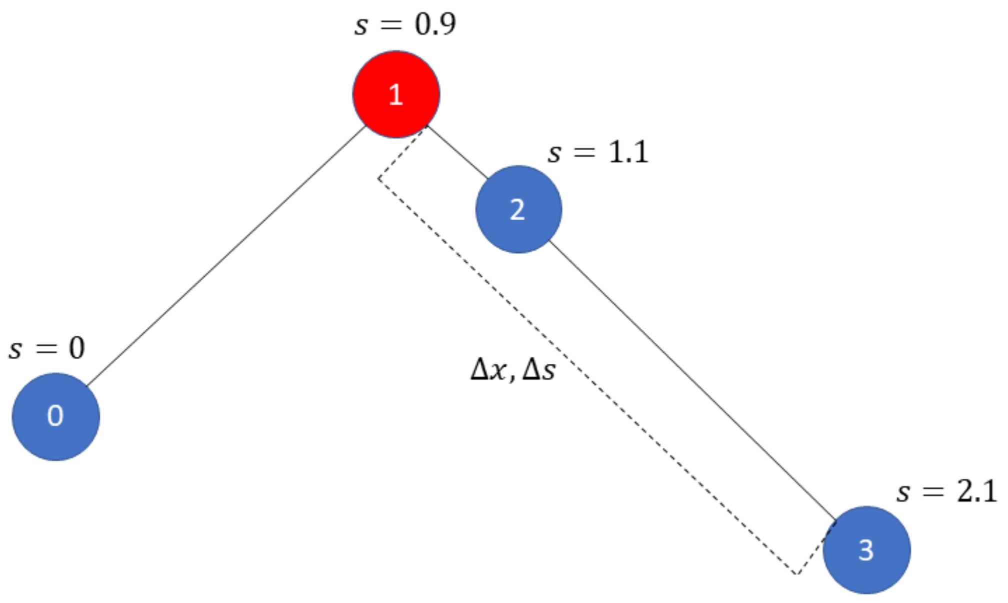

Figure 5: Diagram of node insertion for remeshing.

The last property to calculate before inserting the node back into the strand is the velocity. Since the values of the node have not changed since being disabled, we need to calculate what the velocity should be for the node that is being inserted. Figure 5 shows a diagram of a possible remeshing scenario. Nodes 0, 2, and 3 are L-nodes while node 1 is an E-node. We are inserting node 2 in between nodes 1 and 3, so its velocity should be a linear interpolation of the velocities of its neighbors. We can calculate this using the equation:

$$v_2 = (1 - \alpha)v_{w1} + \alpha v_{w3} \qquad \text{(Eq. 6)}$$

where

$$\alpha = \frac{s_2 - s_1}{\Delta s}$$

$$\Delta s = s_3 - s_1$$

$$\Delta x = x_3 - x_1$$

$$v_w = \begin{pmatrix} \mathbf{I} & -\mathbf{F} \end{pmatrix} \begin{pmatrix} \mathbf{v_x} \\ \mathbf{v_s} \end{pmatrix} \qquad \text{(for E-nodes)}$$

$$v_w = v_x \qquad \text{(for L-nodes)}$$

$$F = \frac{\Delta x}{\Delta s}$$

Using this, we can set node 2's velocity and direction before insertion, so that it behaves properly.

CHAPTER III

RESULTS

To test the performance of the simulation, we chose to run it multiple times, increasing the number of nodes each time and measuring the simulation time (time taken to calculate one step). Simulation time was chosen over frames per second (FPS) because FPS includes the time taken to render the strand. Since we are only interested in how long it takes to calculate the step, the simulation time is a better choice. To provide a baseline for comparison, the same tests were performed for a purely Lagrangian version of the strand in which there are only L-nodes included. To differentiate the two, we will call this method the "Lagrangian method" and the version with E-nodes and L-nodes the Eulerian-on-Lagrangian method, or EOL method.

The tests were performed on a desktop computer running Windows 10, with an AMD Ryzen 5 1600 six-core processor and a NVIDIA GeForce GTX 1060 6GB graphics card. The code was compiled using Visual Studio 2017 in release mode. The simulation time was measured using the chrono C++ library to measure the time between the start and end of each step. For each trial, we ran the simulation for 100 steps and took the average of the simulation times for all 100 steps. This was repeated for both Lagrangian and EOL strands with 7, 40, 100, 200, 400, and 800 nodes. Each trial was done with the same configuration of E-nodes, which can be seen below in Figure 6.

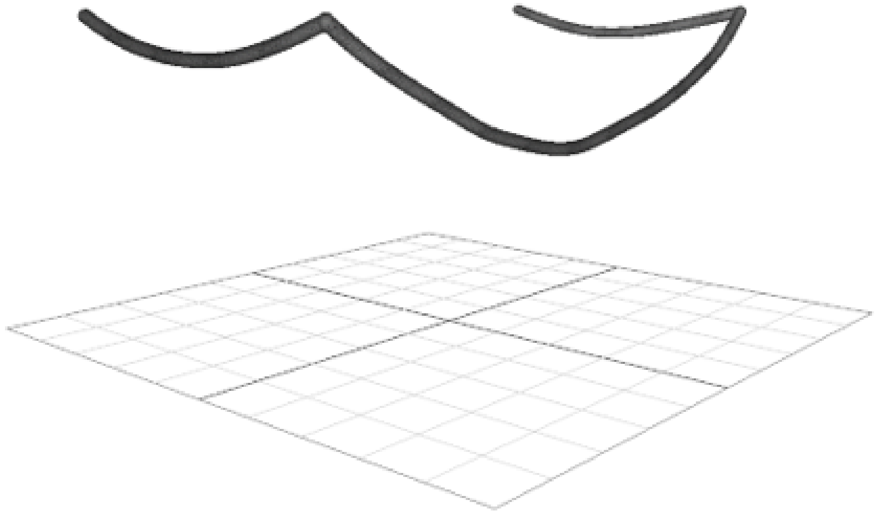

Figure 6: Configuration of E-nodes for tests.

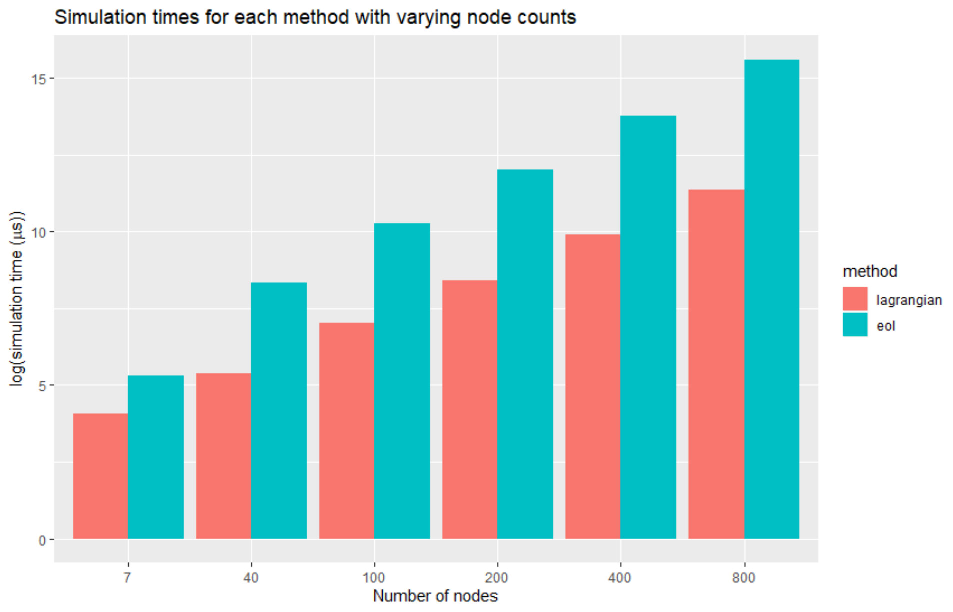

Figure 7: Log scaled graph of simulation times.

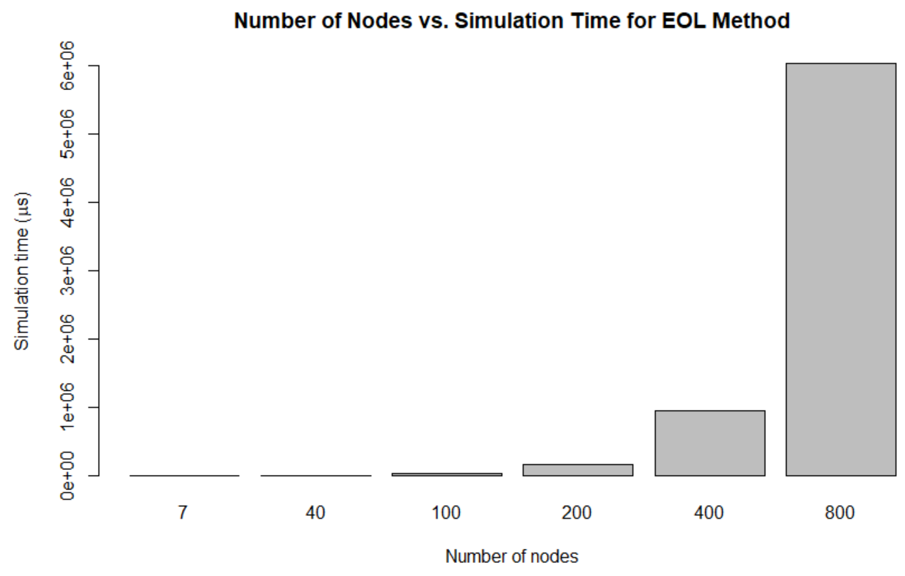

Figure 8: Simulation times for EOL method.

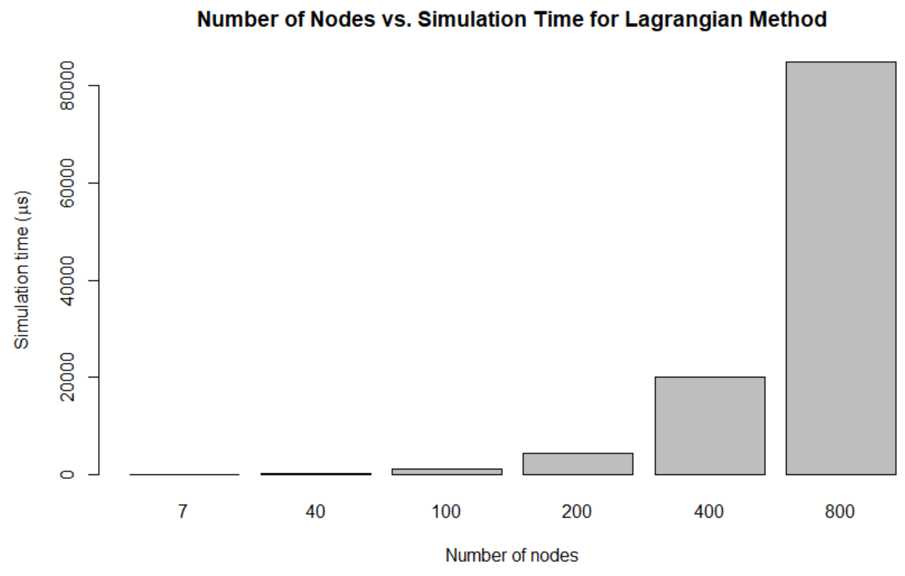

Figure 9: Simulation times for Lagrangian method.

Simulation Time

In terms of simulation time, the EOL method performed worse than the Lagrangian method. Figure 7 shows the comparison of simulation times, with a log scale for the y-axis to make the comparison easier to see. The simulation time for the EOL method was higher than that of the Lagrangian method in every case. However, the strand simulated with the Lagrangian method was unable to smoothly slide across sharp corners.

Figures 8 and 9 show the approximate simulation times for both methods. For the EOL method, simulation times ranged from around 100 microseconds to almost 6 seconds per step. The Lagrangian method had much lower times comparatively, ranging from around 10 microseconds to 0.08 seconds. For both methods, the simulation time scaled up drastically from 400 to 800 nodes, with a similar but smaller increase from 200 to 400 nodes.

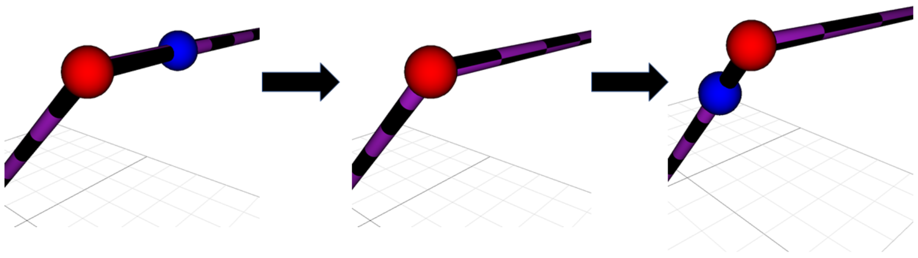

Figure 10: The remeshing process. The L-node (blue) approaches the E-node (red), is disabled when it gets close, and is re-inserted when it is far enough.

Remeshing Functionality

Figure 10 demonstrates the process of remeshing implemented in the simulation when a node approaches a bend in the strand. As before, the red node represents an E-node and the blue node represents an L-node. As the L-node is dragged closer to the E-node, it is disabled once it reaches a certain threshold. Eventually, once it is far enough away, the L-node is re-inserted into the strand in a configuration different from the original.

Occasionally during testing, we noticed that the velocity of the node after re-insertion was not consistent. The velocity ended up being much higher than we would expect from the simulation, which caused some nodes to heavily bounce back and forth along the strand, since the velocity was so high.

CHAPTER IV

CONCLUSION

In this paper, we demonstrated a program capable of simulating a string bending and sliding around sharp corners. The cylinders drawn between each node create the shape of a real-world string, especially when a realistic texture is applied. Several properties of the strand can be modified, including the configuration and position of nodes, as well as the mass and thickness of the strand. Basic physical forces of gravity and tension can be simulated, and the Eulerian components in each node allow for the manipulation of material coordinates. This enables the strand appear to stretch and bend by modifying the texture coordinates. Finally, the process of remeshing we implemented allows for an L-node to pass through other E-nodes without destabilization of the simulation.

Although the simulation time for higher node counts may be too slow to allow for a smooth simulation, it is more reasonable for lower node counts. As long as the node count remains at a moderate level, the simulator will run smoothly. A strand simulated with the Lagrangian method is faster, but it cannot slide smoothly across sharp corners like the one simulated with the EOL method.

Future Work

As mentioned earlier, this program could be extended to fully simulate ayatori. One major step towards such a simulator would be some sort of collision handling. When separate parts of the strand cross, then they should appear to bend around each other. One possible way this could be achieved is by first detecting when parts of the strand have crossed. Once it is determined where they have crossed, E-nodes could have been inserted at the points in which they cross. This could create a bend in both parts of the strand which accurately represents what would happen in the real world.

Another improvement that we could make to this simulator is to improve the efficiency.

Currently, the simulator takes six seconds or more to calculate one step with 800 nodes. To be able to properly simulate complex patterns, a large number of nodes will be required. Additionally, a higher node count means the strand looks more realistic. Improving the efficiency of the program to be able to handle high node counts would allow for simulation of more complex shapes.

Finally, we could improve the interface for which we define nodes and constraints. Currently, the positions and configurations of nodes are hard-coded into the program, and changing them would require someone to have knowledge of how the program works. Creating an easy to understand interface in which a user could define node positions, types, and constraints would allow for anybody to define their own shapes without having to know how the program works.

REFERENCES

[1] C. Jayne, *String Figures and how to Make Them: A Study of Cat's Cradle in Many Lands*. Master String Figures, Dover Publications, 1962.

[2] M. Bergou, M. Wardetzky, S. Robinson, B. Audoly, and E. Grinspun, "Discrete elastic rods," in *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, (New York, NY, USA), Association for Computing Machinery, 2008.

[3] C. Batty, A. Uribe, B. Audoly, and E. Grinspun, "Discrete viscous sheets," *ACM Trans. Graph.*, vol. 31, July 2012.

[4] F. Bertails, B. Audoly, M.-P. Cani, B. Querleux, F. Leroy, and J.-L. Lévundefinedque, "Super-helices for predicting the dynamics of natural hair," *ACM Trans. Graph.*, vol. 25, p. 1180–1187, July 2006.

[5] N. Umetani, R. Schmidt, and J. Stam, "Position-based elastic rods," in *ACM SIGGRAPH 2014 Talks*, SIGGRAPH '14, (New York, NY, USA), Association for Computing Machinery, 2014.

[6] S. Sueda, G. L. Jones, D. I. W. Levin, and D. K. Pai, "Large-scale dynamic simulation of highly constrained strands," *ACM Trans. Graph.*, vol. 30, July 2011.

[7] S. Boyd and L. Vandenberghe, *Convex Optimization*. USA: Cambridge University Press, 2004.