Lofting 3D Shapes

Robby Prescott
Department of Computer Science
University of Wisconsin – Eau Claire
Eau Claire, Wisconsin 54701
robprescott715@gmail.com

Chris Johnson
Department of Computer Science
University of Wisconsin – Eau Claire
Eau Claire, Wisconsin 54701
johnch@uwec.edu

Abstract

On the most fundamental level, the computer understands 3D objects as cluster of connected triangles. In fact, the first half of the widely used OBJ file format simply consists of a list of vertices (as x-, y-, z-coordinates), and the second half of the file follows with a list of triangular faces (as indices into the vertex list). In theory, it is possible to construct OBJ file for a torus within a text editor, but this would be a tremendous waste of time. Approaching the task programmatically, on the other hand, creating a torus, along with an extensive list of other shapes, can be completed using a few lines of code and some basic geometric intuition. Although, this hinges on the assumption that the proper solidifying functions exist.

For this research project, we are designing a function titled *loft* that provides method to solidify objects from a list of polygons or *closed vertex paths*. The goal of loft is best described in an inquisitive manner outlined in the paragraph that follows.

Imagine a triangle and a five-pointed star of similar size are plane parallel and not coplanar. How would we connect the vertices of the triangle and the five-pointed star to create a closed 3D shape? The issue here is that the choice isn't obvious, especially for a computer, for these two shapes differ in vertex cardinality. Furthermore, as the problem is generalized allowing various shapes, sizes, orientations, and nonplanar paths, choices for these vertex connections become increasingly less obvious. This is the exact problem the loft function aims to solve. In order to complete this task of generating a 3D object from an arbitrary contour, we must remove all ambiguity by defining the aesthetically desired properties of a general 3-dimensional shape. Creating these definitions lends itself to some interesting questions that blur the line between beauty and mathematics.

1 Introduction & Motivation

Like any function, loft is given parameters and completes a certain task with the said parameters. Before we unravel the parameters of the loft function, we must become acquainted with a few terms. A *vertex* consists of an x-, y-, and z-coordinate. These vertices are often strung together into an ordered list called a *path*. Lastly, an *array* is an ordered list of elements, typically of an arbitrary yet uniform type. Thus, the parameters of the loft function are simply an array of paths.

Once given the data as an array of paths, the loft function creates connections between the vertices of the consecutive elements within the array of paths. In turn, these connections generate the triangular faces that will constitute the final 3-dimensional solid. Although the task seems rather simplistic, this function requires a generous amount of optimization due to the unobvious nature of the decisions creating each of these vertex connections. In fact, if the computer were to perform the algorithmically undesired "brute force" method, examining every possible combination of vertex connections between two paths, path *a* (p_a) and path *b* (p_b), the number of combonations would increase exponentially as the number of vertices in either of the paths increases. Since the two paths can each be abstracted to a resemble a disjoint set within a bipartite graph, we can can determine the total number of possible vertex combinations between two paths by calculating the total number of unique bipartite graphs given by the expression:

$$\binom{|p_a|\,|p_b|}{0}+\binom{|p_a|\,|p_b|}{1}+\cdots+\binom{|p_a|\,|p_b|}{|p_a|\,|p_b|}$$

Expression 1

where $|p|$ denotes the number of vertices in the disjoint set or path p. Each term in the expression resembles the entries within the $(|p_a||p_b|)^{th}$ row of Pascal's Triangle. Since the the sum of all the entries in the n^{th} row of Pascals Triangle is equal to 2^n, the following equation yields the total number of possible vertex combinations between p_a and p_b:

$$\sum_{j=0}^{|p_a||p_b|}\binom{|p_a|\,|p_b|}{j}=2^{|p_a||p_b|}$$

Equation 1

If we were to extend this notion to account for an array containing *n* paths, the following expression would yield the total number of possible vertex combinations for an array containing *n* paths:

$$\prod_{i=0}^{n-2}2^{|p_i||p_{i+1}|}$$

Expression 2

where p_i is the path at the i^{th} index in the array. The total number of combinations exponentially increases as n increases or any given p_i increases, and therefore the computation time reflects these increases. For this reason, the loft function's algorithm utilizes inherent geometric properties of the paths in question to create the optimal connections to solidify any given shape in linear computation time.

2 The Algorithm

The loft function only focuses on two consecutive paths in the array at one time. It connects the paired paths until it reaches the final pair in the array. For example, an array of size n with paths at a given index i denoted p_i would begin by lofting the paths p_0 and p_1, then p_1 and p_2, and so on until it reaches the final pair p_{n-2} and p_{n-1}. For each of these pairs, the process for creating connection between the paired paths' vertices can be broken into a series distinct steps that follow this basic outline:

1) Make copies of the paths to manipulate in 3D space
2) Map nonplanar paths to an appropriate plane
3) Perform linear transformations to make the current pair of planar paths coplanar
4) Center both paths over the origin
5) Reverse the order of the indices if path traversal is clockwise
6) Determine which path has less vertices (p_S) and which has more (p_L)
7) Scale p_L to completely encompass p_S
8) Generate rays from the origin that extend through the edge midpoints of p_S
9) Connect vertices on p_L to the vertices on p_S in the sections created by the rays
10) Find appropriate connections for unconnected vertices in p_L if they exist
11) Find appropriate connections for unconnected vertices in p_s if they exist
12) Make the final connections that cross each of the rays
13) Generate a connected list of triangles

2.1 Make Copies of the Paths to Manipulate in 3D Space

This step seems somewhat obvious, yet it is worth noting. In order to exploit the intrinsic geometric properties of the pair of paths in question, several alterations must occur to the vertices' coordinates of the original path. The vertex connections are made with the altered, duplicate versions of the paths, and when exporting the final structure as a list of triangles, these connections are mapped back to the original paths using the vertex indices

from the corresponding altered path. Therefore, we must preserve the original two paths in some form. This is easily completed by creating deep copies of the pair of paths in question.

2.2 Map Nonplanar Paths to an Appropriate Plane

Sometimes the vertices within a particular path are not coplanar. In which case, it is appropriate to perform the 3-Dimensional analogue of the least squares regression algorithm over the set of vertices within the path to generate a plane that is suitable 'average' for the vertices of the path. Each of the vertices in the path are then projected normally onto the newly generated plane, in turn making each of the vertices in the path coplanar.

2.3 Make the Current Pair of Planar Paths Coplanar

The entire algorithm for generating vertex connections is aiming for geometric simplicity. Removing a dimension from consideration is a logical step in this simplistic direction. Although it is arbitrary which plane is chosen, one elegant choice forces the paths to be parallel to the $z = 0$ plane. For when the planes of the path are oriented in this fashion, we can remove the z-coordinate from consideration for all vertices. This leaves every vertex with only an x- and y-coordinate, in turn reducing the 3-dimensional problem to a 2-dimensional problem. A mathematical method for rotating/orienting these planes is most easily completed with a matrix transformation.

2.4 Center Both Paths over the Origin

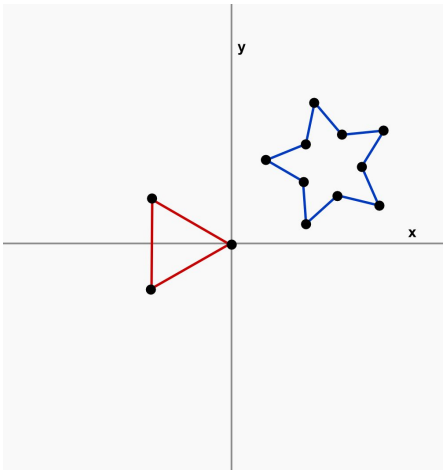

Figure 1: uncentered paths

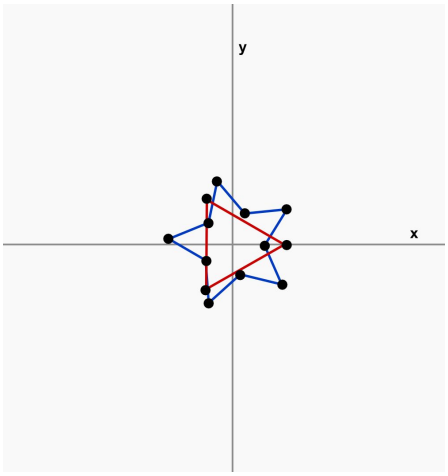

Figure 2: centered paths

Once the pair of paths are located on the xy-plane, it is useful to "pin them to the origin" so to speak. This informal term involves translating by the additive inverse of the center point for each path. For a given path, this center point is calculated by taking the mean of the x- and y-coordinates.

2.5 Reverse the Order of the Indices if Path Traversal is Clockwise

When making connections, algorithmic efficiency is optimized if the paths traverse around the origin in the same direction. Again, it is arbitrary which direction is chosen, but by convention, counterclockwise is often chosen to be the direction of positive traversal. For the algorithm, a test is performed to determine if a given path traverses clockwise around the origin. If this is the case reverse the order of the vertex indices within the path. Do the same list reversal operation to the corresponding preserved path in order to maintain the index mapping between paths.

2.6 Determine p_S and p_L

This step simply involves comparing the sizes of the paths and assigning each path to the appropriate variable. Note that this assignment is a shallow copy of the path and points to the same memory address. Although if a deep copy is made, the algorithm still functions as expected, it will just use unnecessary memory. In the event where both path's sizes are equal, p_S and p_L are arbitrarily assigned.

2.7 Scale p_L to Completely Encompass p_S

This step is not required per say. However, for debugging purposes it is imperative to have a 2-dimensional visual of the current pair of paths in order to immediately and plainly discern where the connections are behaving unpredictably. Scaling the larger path lends itself to a cleaner image with minimal line path intersections and vertex connections from p_S to p_L oriented radially from the origin. Scaling the path will not affect the overall outcome of the vertex connections. This is explained in the following non-rigorous, yet intuitive manner.

Since the vertices in a path consist of only x- and y-coordinates, we may think of a path as a set of 2-dimensional vectors. If you scale a set of vectors the angle between them will not change. Furthermore, since the path is centered on the origin, scaling the center vector will not change its value, and therefore, will not translate the overall path. As a consequence of this, the vertices of the scaled path will remain between the same rays that emanate from the origin before and after scaling. Thus the connections generated by

the algorithm are unaffected by the scaling of the paths at this stage. Before scaling, however, it is important to make a deep copy of p_L at this stage denoted $p_{|L|}$. This p_L duplicate preserves relative distances to p_S that will become useful in later steps.

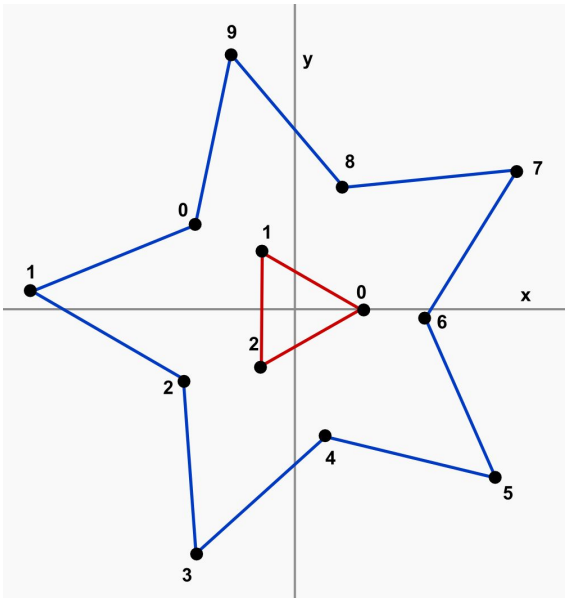

Figure 3: scaled large path encompassing the small path

2.8 Generate Rays from the Origin that Extend through the Edge Midpoints of p_S

Ray generation is the essential step to the loft function's algorithm. It is the geometric exploit that determines most of the vertex connections. To generate these rays, calculate the midpoints for each of the edges on p_S. These *edges* are simply the line segments between the vertices on p_S. For each of these midpoints, create a line segment that emanates from the origin, intersects the given midpoint, and extends an arbitrarily far distance away from the origin.

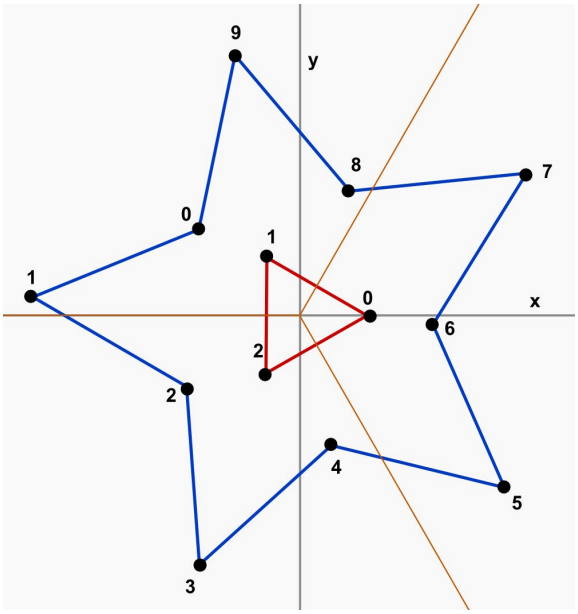

Figure 4: rays from the origin through the midpoints of the small path

2.9 Connect Vertices on p_L to the Vertices on p_S in the Sections Created by the Rays

Generally speaking, most of the connections will be created in this step. Note that each section generated by the rays contains only one point on p_S. Within each of these sections, iterate through the points on p_L that fall within the current section, and connect them to the single point on p_S.

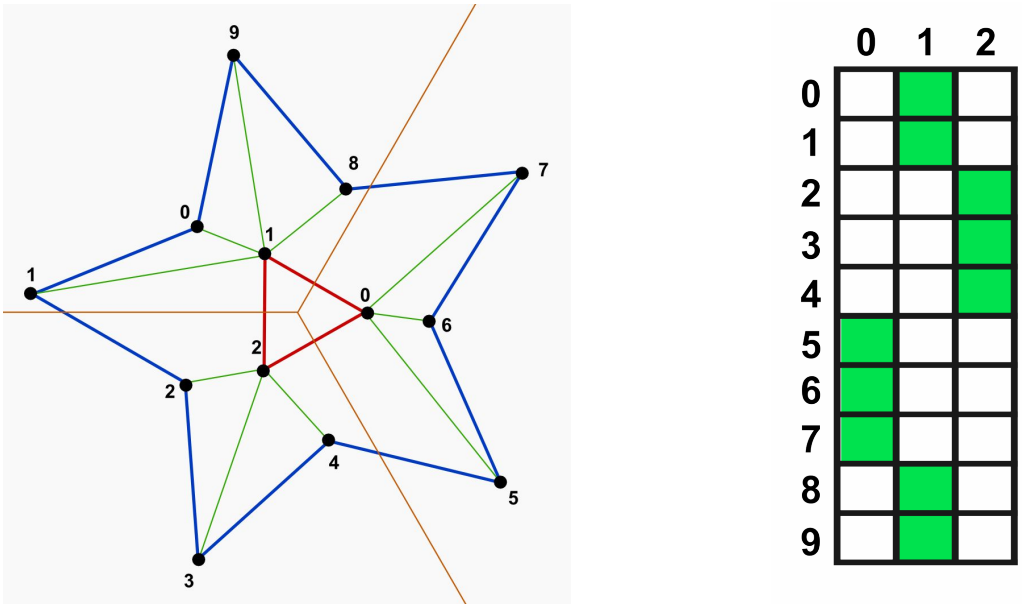

Figure 5: vertex connections seen in both the graphical form (left) and the adjacency matrix form (right) in green

Storing the connections can be handled in a variety of ways. The method I propose is an adjacency matrix. This adjacency matrix provides a tabular visualization for the connections between indices, as well as a clean way to export the triangles later on. In the adjacency matrix, the rows represent the indices on p_L, and the columns represent the indices on p_S . Although, this row/column assignment is completely arbitrary since it is equally valid viceversa. The values within the matrix are of boolean type, where *true* represents a connection between indices and *false* represents no connection between indices. Speaking in terms of the adjacency matrix, a valid and exportable list of connections can be visualized by an unbroken path of true values that wraps around from the right edge to the left edge, and similarly from the bottom edge to the top edge. Furthermore, this path should never bifurcate. An equally descriptive definition for the unbroken path involves every true cell having exactly two true cells to its immediate right, left, top, or bottom (wrapping included).

2.10 Find appropriate connections for unconnected vertices in p_L if they exist

Seldomly, a vertex on p_L will fail to get connected to any vertex on p_S. The test for this case simply involves checking for an empty row in the adjacency matrix. This skipped connection usually occurs when the the vertex intersects a midpoint ray from p_S. In this instance, determine the closest vertex on p_S to the vertex on $p_{|L|}$ whose index corresponds to the unconnected vertex on p_L. Add the connection between these indices to the adjacency matrix.

2.11 Find appropriate connections for unconnected vertices in p_S if they exist

More commonly, a vertex on p_S will fail to get connected to any vertex on p_L. The test for this case simply involves checking for an empty column in the adjacency matrix. This skipped connection occurs when no vertices on p_L fall within the section outlined by the rays on either side of the unconnected vertex on p_S. In this instance, determine the closest distance from the unconnected vertex on p_S to one of the two vertices on p_L that located radially immediately before and after the p_L verticeless section outlined by the rays. Add the connection between these indices to the adjacency matrix.

2.12 Make the Final Connections that Cross Each of the Rays

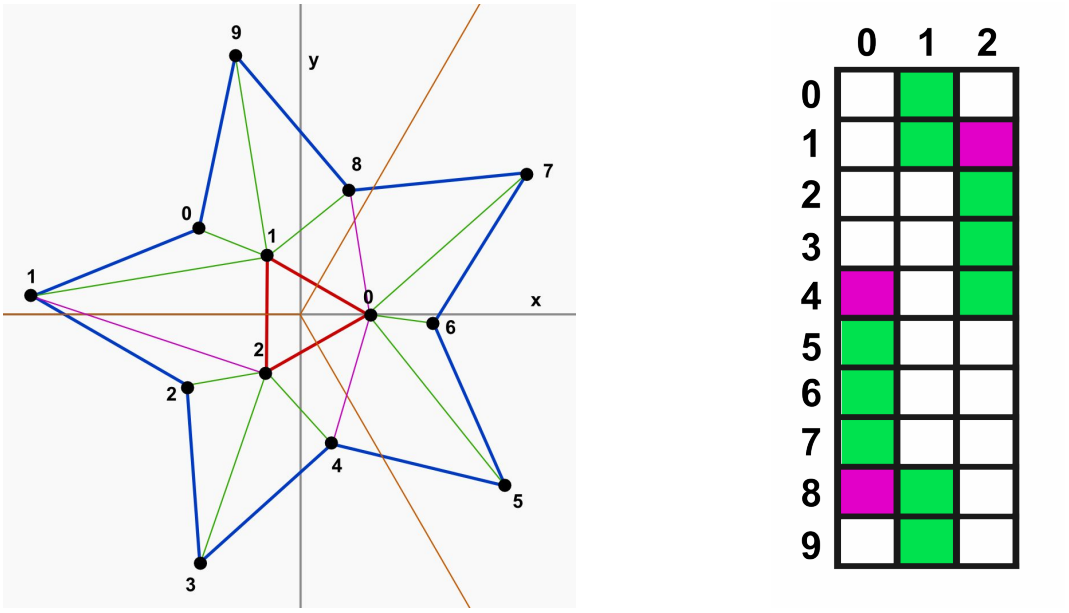

Figure 6: vertex connections for the quadrilateral/tetrahedral cases seen in both the graphical form (left) and the adjacency matrix form (right) in magenta

Notice the vertices' indices that are separated by the rays. Quadrilaterals are still present between the connections. Since the OBJ file format requires a list of triangular faces, we need to make a decision on a connection between the two kitty-corner options. This connection will divide a given quadrilateral into two triangles, making the the desired 'unbroken path' in our adjacency matrix. In our original 3-dimensional paths, the quadrilateral is actually four points floating in 3D space. This may also be thought of as a tetrahedron. Unless the four points in our original paths are coplanar, one of of the two options for the connection will result in a concave surface for that portion, the other will result in a convex surface for that portion. Since—arguably—3D solids look better when they are entirely convex, you may wish to set the decision to automatically choose the convex option by default. However, giving the loft function an optional parameter that signifies the convex/concave decision to make for these tetrahedral cases gives the user more control over the final shape they desire to create.

2.13 Generate a Connected List of Triangles

Once the desired 'unbroken path' of connections within the adjacency matrix is complete, the triangles can be generated. Traverse the path of true values on the adjacency matrix and map the row and column numbers of the current cell to the indices in the original path. The current two indices refer to the first two vertices on the original paths for the

triangle to be created. The last vertex of the triangle to be created is determined by looking at the next cell in the traversal and discerning which index changed from the previous cell. Format the final list of triangles so that it matches the OBJ file format and export the file.

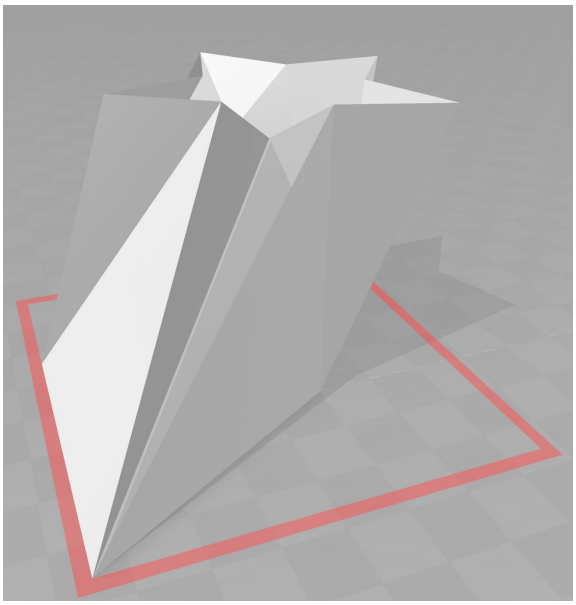

Figure 7: the final 3D shape generated by the vertex connections from the previous steps

3 Conclusion & Inspiration

Once completed, various shapes of complex geometries can be made with ease. Illustrated below are a few examples.

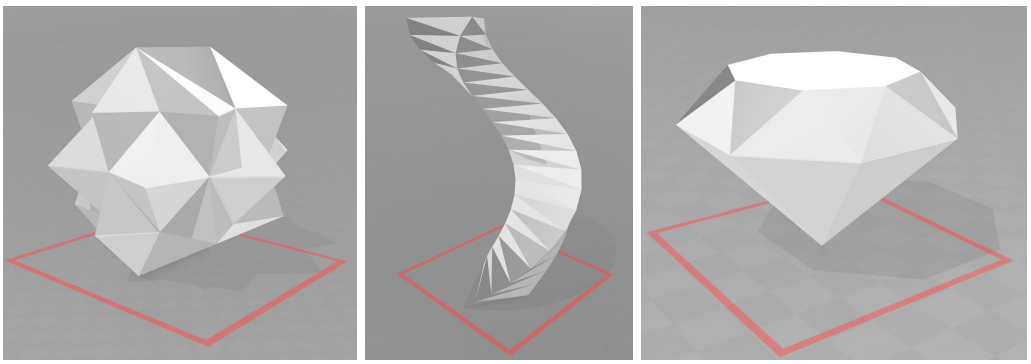

Figure 8: examples showing off the aesthetic capabilities of the loft function.

The current algorithm still could use improvement. Although it performs splendidly for all convex path cases, it only works for a subset of concave path cases. The concave path

cases that cause problems are characterized by the following property. When the center of the path is located at the origin and the vertices of the path are traversed in the positive polar coordinate fashion, the theta value decreases at least once between two consecutive vertices in the path. This type of concavity makes the ray approach invalid. A proposed possible solution and method of implementation involves finding the convex hull of the problematic path. This divides the vertices of the path into two sets: the vertices that are apart of the convex hull (*H*) and the vertices that are not part of the convex hull (*N*). Remap the vertices in *N* that occur in consecutive ordered subsets to the edge of the convex hull between the two surrounding vertices on the path in *H*. Including this as part of the loft function procedure between steps 5 and 6 could potentially account for a wide variety of concave path cases.

No matter how many improvements the algorithm undergoes, some paths can never be properly accounted for. Such paths have properties of self intersection, knots, and other complex geometries.

Lastly, the current algorithm only accounts for only genus-0 surfaces (topologically equivalent to a sphere), and genus-1 surfaces (topologically equivalent to a torus). Interesting variations implemented in the future may be generalized to include genus-*n* surfaces.