Synthesis of a Dihedral Animation "origami"

Alexandra Lamtyugina

16 December 2015

Abstract

Folding along a dihedral angle is a phenomenon observed in various disciplines, which range from mathematics (folding of polyhedral nets) to structural biology (folidng of proteins). My semester project focused on creating a versatile dihedral animation that can potentially be applied to more complicated (and thus more interesting) situations in order to visually demonstrate folding-related concepts.

Introduction

Origami

Many classical three-dimensional polyhedra can be described in terms of what are known as "polyhedron nets." The nets can be thought of to originate as a result of making a series of incisions along several edges of the polyhedron, and then unraveling the faces such that they lie on the same plane. By definition, there are several requirements that have to be met before an unfolding can be considered to be a net [1]:

1. The net has to be planar.
2. The net has to be in a single piece (i.e. all faces are connected by at least one edge to the rest of the net).
3. The net has to result from an unfolding done as a result of "incisions" made along the edges (i.e. an edge unfolding).
4. The net has to be non-self-overlapping within the plane that it occupies.

Such nets are frequently used as a starting point to create decorative models of polyhedra, which allow for a tangible exploration of the concepts to be presented.

Chemistry

Not surprisingly, many molecules exhibit geometry which mirrors that of polyhedra. For instance, the buckminsterfullerene molecule is in fact a truncated icosahedron. Both the molecule and the geometrical model can be assembled from sixty identical units (the latter of which I have done with paper on a previous occasion). Therefore, although molecules don't necessarily "fold" from a two dimensional net, the same geometrical constraints still apply.

Much like a net is a two dimensional representation of a polyhedron, the primary sequence of a protein is a two dimensional representation of a protein. The folding mechanism of a protein can be mirrored to the one of the polyhedra, given that the dihedral angles between each amino acid residue are known from previously gathered experimental data (input). Although my project did not concern itself with animating the folding of polypeptides into their tertiary structures, it is an idea that is worth considering in the future.

Goals

While at first glance folding a two-dimensional net into its respective three-dimensional polyhedron is a fairly straightforward concept, I have found that programming such an animation requires an understanding of the mathematics behind translations and rotations of objects within a reference frame. One of my objectives was to become familiar with the concepts behind manipulating the spatial positions of the faces of a net. The ultimate purpose of my project was to create a programmed animation that demonstrates the folding of a two-dimensional polyhedron net into its respective three-dimensional polyhedron. Although initially I planned on creating a program that would calculate the internal dihedral angles necessary to fold a given net into its respective polyhedron, I have settled for making a program that has all of the dihedral angles pre-programmed due to time and ability constraints.

Methods

The programming aspect of my project was completed using OpenGL. Although the original plan was to do the entire project in C, some of the preliminary work (e.g. the convex hull animation shown during my seminar presentation) was done in Python due to the fact that it required quick implementation of ideas into code, which is more easily done in Python rather than C. However, due to the fact that many of the OpenGL reference texts use C as the language shown in examples, my final RTICA is a C file that can be complied by the `winaid` compiler provided in the repository, which is intended to run from a PC command line.

In a nutshell, the mechanics of the dihedral animation that I have completed can be described as follows. After defining the vertices of the individual polygons making up the net to be folded in arrays, a series of `glPushMatrix()`

and `glPopMatrix()` commands will be used in order to animate the folding of the individual polygons of the net around a defined axis and in a defined angle. By using the built-in function `glNewList()`, it was possible to pre-compile the faces of the polygons, and then call the faces for their respective positional transformations one at a time.

Based on the method described above, all of the commands are carried out at the same time and all of the faces of the polyhedron will fold simultaneously. I have personally chosen to distinguish this type of folding from sequential folding by invoking the term "heat" in the title of the file of the program. Although initially the title is not entirely intuitive, I decided to describe this simultaneous folding as heat folding as it reminds me of the way that plastic becomes deformed when it is heated. That is, sections that have more plastic do not contract as quickly as sections that have a thinner layer of plastic. In parallel, the folding and unfolding of a net can be thought of being carried out under high temperatures and resulting in the contraction and expansion of the empty spaces of the net to yield a 3-dimensional polygon.

However, I also thought that it would be beneficial to view the folding of the faces one at a time. In that case, additional specifications are needed in the RTICA, which are modeled after the ones used in Mark Kilgard's airplane folding animation [2]. More specifically, a function that will check for the angle and direction that a given polyhedron face is folding can be added to the program, which will modify a `state` variable (as well as the direction of folding, if necessary) as each folding event occurs.

Additional Work

While working to create the dihedral animation, I have embarked on a couple of additional side projects that were related to my final program. The first involved creating a visualization of a convex hull, which I described in my seminar presentation. During my presentation, I stated that my project would only concern convex polygons, which are defined by their convex hulls that result from taking a set of points in 3-space and figuratively wrapping cling-wrap around those points. My convex hull animation (`hull.py`) accomplished this by generating a random set of points in 3-space and drawing lines between all of them. Initially, I attempted to draw polygons instead of lines, as that allows for a better visualization of the concept of convexity because the polygons drawn between points not on the surface would be theoretically obscured by those that do comprise the surface of the randomly generated solid. However, at the time that I wrote the program, I was not aware of the `GL_EXT_polygon_offset` function, without which the convex hull program was unviewable due to the flickering of overlapping polygons. If necessary, it would be trivial to add the polygon offset to `hull.py` such that the viewing problem is eliminated.

The other project, which was actually one of the ideas I was considering to be my main project at the beginning of the semester, involved the random generation of nets of a cube. This was done by assigning a grid of squares such

that

$$D = \begin{cases} x \in [-3, 3] \\ y \in [-3, 3] \end{cases} \tag{1}$$

describes the domain on which the 1 by 1 squares resided. The arrays of vertices of the individual possible faces of the net were then placed into another array, and the indices of that array were used to create a six-square net that was entirely connected. Figure 2 demonstrates the way that the indices of the array of arrays were visualized, with the origin being placed between 14, 15, 20, and 21.

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 & 28 & 29 \\ 30 & 31 & 32 & 33 & 34 & 35 \end{array} \tag{2}$$

The connectivity was ensured by first randomly selecting an index which corresponded to a “seed" square, and then randomly adding a square to one of the spaces immediately adjacent to the seed square by either adding or subtracting 1 (right or left) or 6 (below or above) from the index of the randomly picked array. Restrictions were placed such that squares that lie on one of the edges or corners only had additions that lie within the domain described by equation 1.

The result of these algebraic manipulations of the array indices are unique nets comprising of six squares. Unfortunately, the way in which the nets are generated makes it difficult to analyze them for foldability, and therefore this feature was not implemented into the `netgen.c` program. However, even without a computer's analysis of the net it is obvious that a large majority of teh random nets generated will not be able to fold into a closed cube. If net analysis was ever implemented, it would be interesting to anayze the ratio of foldable nets to the total number of randomly generated nets.

Conclusion

Although I initially intended my project to be entirely about generating unique nets from a variety of polygons and testing them for foldability, I lacked the proficiency in both programming and linear algebra in order to implement my ideas into code. Therefore, I focused on simply creating a dihedral animation, with the intent on using the concepts that I have learned in MATH 198 in the future, when I have gained more experience with the concepts necessary to make the program that I have originally envisioned. While by themselves `cubeC.c`, `tetrahedron.c`, and `netgen.c` may not be able to undergo any sig-

nificant applications, I believe that they have succeeded in serving as a starting point in my exploration of mathematics through computer animation.

References

[1] O'Rourke, J. *How to Fold It: The Mathematics of Linkages, Origami, and Polyhedra*, Cambridge University Press, Cambridge, UK, 2011.

[2] Kilgard, M. `origami.c` - https://www.opengl.org/archives/resources/code/samples/glut_examples/examples/origami.c