Computer Design of Open Tiling Materials

January 8, 2016

Vincent Galissard de Marignac
EPFL, IN
vincent.demarignac@epfl.ch

Supervisor:
Mina Aleksandra Konakovic

Professor:
Prof. Dr. Mark Pauly

Abstract

The aim of this project is to implement mechanisms allowing the manipulation of discrete open surfaces expanded with physical qualities. Recursively searching intersections for the detection of freely-drawn or clamped cutting curves, one can introduce holes in a surface to intentionally alter its topology. The inclusion of the *ShapeOp* library for shape optimisation under constraints reproduces physical properties that allow the opening, stretching and bending of simulated sheets of materials with varying stiffness. I observed how cutting through a surface modified its physical behaviour when reshaped because of the dependence on mesh connectivity. The result is an *OpenFipper* plugin able to generate, cut, combine and deform meshes purposefully for applications such as the simulation of paper garlands or soft pastry.

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Contents

Introduction 3

I Cutting mesh plugin 4

1 Plugin primitives 4

1.1 Definitions . . . 4

1.2 Two-step cutting . . . 4

1.2.1 Cutting along a single edge . . . 4

1.2.2 Connecting cuts . . . 4

1.3 Merging elements . . . 4

2 User input 5

2.1 Clicking the mesh . . . 5

2.2 Selecting elements . . . 5

3 Curve drawing 5

3.1 Discrete world . . . 6

3.2 Intersecting the mesh . . . 6

3.3 Searching for the next intersection . . . 6

3.4 Avoiding same intersection loop . . . 7

3.5 The edge cases . . . 8

3.6 Clamping . . . 8

II Physical behaviour 10

4 Domain of application 10

5 Constraint exploration 10

5.1 Closeness constraints . . . 10

5.2 Edge strain constraints . . . 11

5.3 Triangle strain constraints . . . 11

5.4 Area constraints . . . 11

5.5 Bending constraints . . . 12

5.6 Rectangle constraints . . . 13

III Cutting patterns and applications **14**

6 Mesh generation **14**

6.1 Hand-made meshes 14

6.2 Computer-generated meshes 15

7 Cutting patterns and deformation **15**

7.1 Garlands 15

7.2 Other objects 17

7.3 Hinged tessellation 17

8 Multiple meshes **18**

8.1 Garlands 18

8.2 Lantern 18

Conclusion **19**

Introduction

The explicit polygon mesh structure is a widely used modeling of surfaces, appreciated for its ease of use and convenient versatility. This project covers triangle and quadrilateral meshes in their halfedge representation, in particular using the *OpenMesh* implementation. Given this context, two-dimensional meshes embedded in three dimensions can be made to depict real-world surfaces. Furthermore, open surfaces have particular characteristics, providing them with extra mobility and shape diversity.

Physical materials can be modelled precisely from polygonal meshes with the addition of globally effecting local constraints. A surface can be cut open, affecting the topology of the underlying object, and can have great impact on its physical behaviour when handled strategically. *OpenFlipper* is an open source educational application and programming framework that relies on plugins for the processing and rendering of geometry elements. I will use it to create a plugin, the purpose of which is to merge the physical and topological aspects in a set of useful tools.

The task of gaining ability to create and manipulate open surfaces can be broken down into smaller essential components. Cutting *primitives* will make up the building blocks for larger algorithms as basic operations on single mesh elements such as the splitting and merging of edges and vertices. With the addition of the *ShapeOp* library for global optimisation of surfaces under local constraints it is possible to simulate the behaviour of physical objects guided towards a specific shape. Several configurations will allow to explore various types of materials, such as stiff paper or stretchy cloth. With this toolbox in place, one can create objects such as garlands and lanterns, from the combination of multiple cut surfaces, enabling the study of the deformation of complex topologies augmented with physical properties.

Part I

Cutting mesh plugin

1 Plugin primitives

1.1 Definitions

I will refer to *cutting primitives* as basic operations that will be used on single elements of a mesh, which include the cutting of a single edge and the split of one vertex in two.
A *cut* is similar to a hole in a surface. It is defined by two or more connected boundary edges. More specifically, contrary to a hole which can also be the border of a surface, is introduced by the user on a mesh using the MeshCut plugin.
When drawing a path using the mouse, the cursor can move too fast with respect to the rate at which OpenFlipper registers individual path points. I call this phenomenon a *jump* and it typically spans multiple faces and edges, but always more than two in the case of faces, and more than one in the case of edges.

1.2 Two-step cutting

Cutting takes effect in two steps and therefore makes use of two separate primitives, namely cutting at edges and then opening the cut at vertices. More specifically cutting intends to actually split individual edges in two to create an incision in the surface and opening the cut allows to widen the extend of a single cut to consecutive cuts. Note that the process is the same for triangle and quadrilateral meshes.

1.2.1 Cutting along a single edge

The first primitive duplicates the selected edge and sets their facing halfedges as boundaries, effectively creating a cut. Connectivity is taken care of by copying pointers to adjacent elements of the new halfedges[1]. The orientation of halfedges circling around faces is maintained by connecting halfedges in a counter-clockwise manner, maintaining the same adjacent faces orientation. We can see on Figure 1 a triangle mesh being cut at a single selected edge, where the red edge is the initial one and the green edge is the new one.

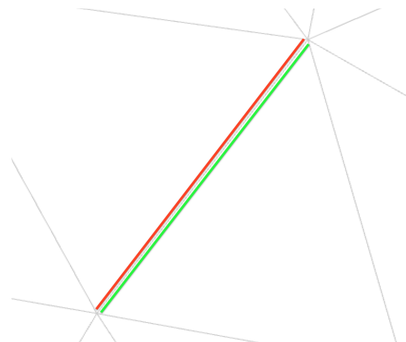


Figure 1: A single edge split into two edges. The red edge depicts the original and the green the new one.

1.2.2 Connecting cuts

The second primitive splits vertices in two and rewires attached edges. Consider two cuts sharing a vertex at one of their extremities. We can connect them by "opening" the mesh at the common vertex, or more specifically by splitting it in two. Assigning outgoing edges to either "left" or "right" side we can group them with respect to their future vertex connection. Pointers are adjusted to their corresponding neighbouring elements, keeping the right orientation.

1.3 Merging elements

It is also useful to be able to perform the reverse operation to cutting, namely to merge vertices together. Doing so can be regarded as "stitching" the surface at a cut. The merging primitive of a pair of vertices sees the edges of one of them attributed to the other and the lone vertex deleted. As

[1] Florian Brandherm, on the OpenMesh mailing list, helped by hinting about the pointer adjustments needed to wire an additional edge and effectively marking the boundary.

this operation is effectively stitching, the openings on both sides of the merged vertex are preserved by adapting boundaries.
As we will see in Part III, we also need a tool to combine meshes together. This tool does not modify the geometry of the target components but rather simply combines them into one mesh for convenience.

2 User input

2.1 Clicking the mesh

The action of clicking registers the 3D projection, as well as underlying face index, of the viewport 2D position on an existing visible object. The projection is called a *hit* and its position is the *hit point*, it can be used to compute the closest edge and vertex on the *hit face* as demonstrated in Algorithm 1, where target elements are edges and vertices.

Algorithm 1 Find closest element to hit point on hit face

$closest_element \leftarrow first_element$
$min_dist \leftarrow distanceFromTo(first_element, hit_point)$
for each element e on $hit_triangle$, starting from second element **do**
  $dist \leftarrow distanceFromTo(e, hit_point)$
  if $dist < min_dist$ **then**
    $min_dist \leftarrow dist$
    $closest_element \leftarrow e$
  end if
end for

2.2 Selecting elements

By clicking on the mesh, one can select single edges to be cut or vertices to be merged for example. Edge selection is convenient for discriminatory or precise picking on a mesh without modification to its geometry. The user can select or deselect individual edges for later cutting, as seen on Figure 2, which is particularly appropriate for joining curves or closing loop patterns.
To expand selection capabilities, I added the possibility to draw a path for free curve selection, as we will see in section 3.

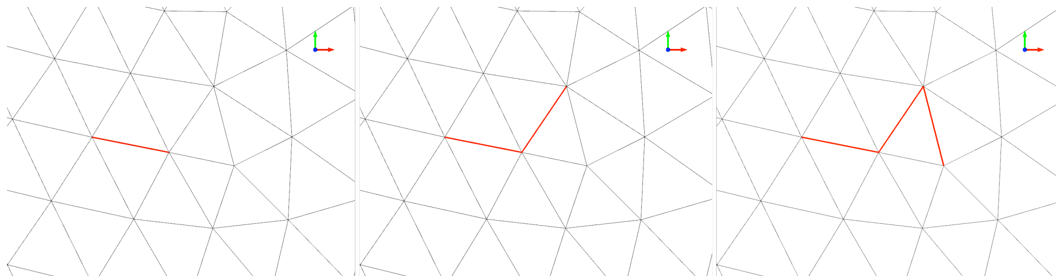
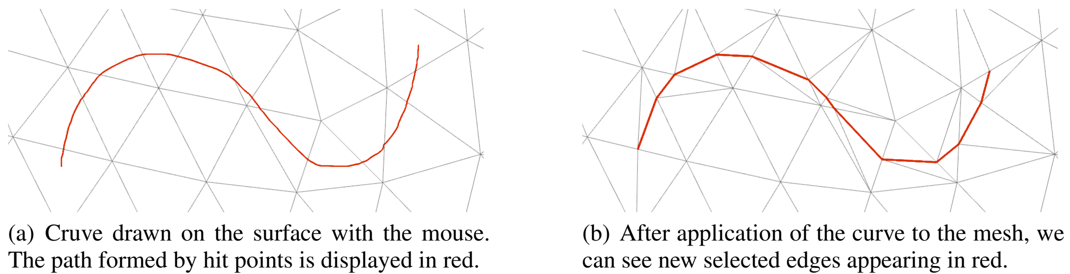


Figure 2: Selecting multiple edges, seen in red, one after the other from left to right.

3 Curve drawing

The mesh can be cut freely by drawing a curve on its surface. It can be performed using the mouse by clicking and, while holding, tracing a path through the surface. The path actually connects edge crossing positions whenever the cursor traverses from a face to another and as we will see all the complexity resides in finding these intersections. When drawing a curve as seen on Figure 3(a), the path composed by all hit points is displayed following the mouse cursor. After releasing the mouse button, the path is applied to the surface as seen on Figure 3(b). Free curve cutting first requires a

path to be applied to the surface. This is done during the selection process and will be explained in the next subsections.



(a) Cruve drawn on the surface with the mouse. The path formed by hit points is displayed in red.

(b) After application of the curve to the mesh, we can see new selected edges appearing in red.

Figure 3: Drawing freely a curve on a triangular mesh surface.

3.1 Discrete world

Computers work with discretised versions of real-world quantities. When sweeping over the mesh with the mouse, the registered path is not continuous but rather a sequence of points that, when linked together, represent an approximation of the continuous curve the user has drawn. Furthermore, the mesh on which to apply this curve is itself discretised through its explicit representation. In the case of the MeshCut plugin the mesh, composed of triangle or polygonal faces, can receive a cut by inserting new edges between existing or newly created vertices before splitting them, as depicted on Figure 3(b).

3.2 Intersecting the mesh

Let us consider a drawn path on a triangle mesh and assume there are at least two hit points per face, one on each side of every edge crossed. We will *walk* iteratively through all the registered hit points, considering the relevant positions, faces and closest edges along that path. Starting our walk on one face, the points of interest are exactly those on both sides of an edge, the first is the *current point* and the other the *next point*. We want to compute the intersection of the segment defined by the current and next points with the edge between them using Thales' intercept theorem by projecting each point on the line defined by the edge[2]. The intersection position and edge index are saved to a queue for the splitting phase. This is done recursively until the path has been covered completely and we can go through the saved queue to create new vertices at crossing points. The last step of this algorithm is to select, for later cutting, the edges between all the newly created vertices. This basic method is described in Algorithm 2.
Unfortunately in practice our assumption doesn't always hold. There is an obvious case for which the recorded mouse movement is not precise enough and yields points that are further apart than would be necessary for Algorithm 2 to work. Consider for example the triangle mesh on Figure 4(a) where small black circles represent the hit points and the red line is the deduced path. We can see that the blue outlined triangle does not hold any hit point. This situation is the jump mentioned in Subsection 1.1 and therefore needs further development.
To make up for the missing points, a possible strategy is to iteratively search for the next potential intersection and iterate until the next point has been reached.

3.3 Searching for the next intersection

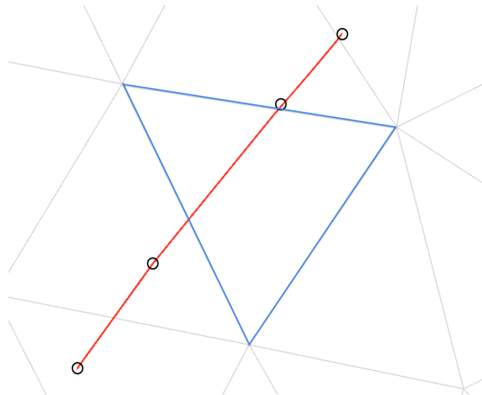
Let us define the jump as the segment designated by the current hit point and the next one, which is outside of the face the current point lies on. Let us call that segment an *outgoing segment*, because its starting point is inside the face and its end point is outside of it. We want to find the intersection of that segment with one of the edges of the current hit triangle. It is useful to first project the outgoing segment onto the plane defined by the face, because our surface might not be flat. Next we can iterate

[2]Thales' intercept theorem allows us to compute the intersection point P_i on a line knowing points P_0 and P_1 and their projections on that line P'_0 and P'_1. We have $\vec{e} = P'_1 - P'_0$. Then $P_i = P'_0 + \vec{e}\frac{\|P_0 - P'_0\|}{\|P_0 - P'_0\| + \|P_1 - P'_1\|}$.

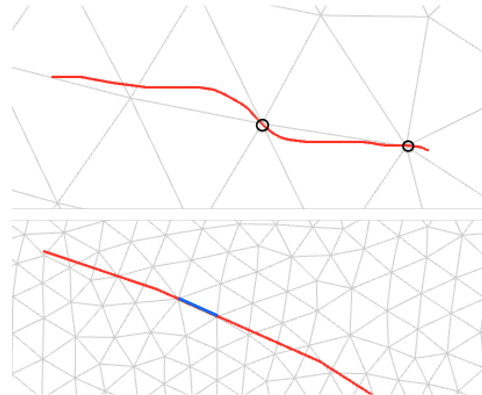
Algorithm 2 Apply and select drawn curve to mesh

Input: R a list of triples containing hit positions, faces and edges
Q is a queue of pairs of intersection points and edge indices
$next_point \leftarrow R.pop()$
while R has more than one element **do**
  repeat
    $current_point \leftarrow next_point$
    $next_point \leftarrow R.pop()$
  until $current_point.face \neq next_point.face$
  $i_pos \leftarrow$ intersection between $(current_point.pos, next_point.pos)$ and $next_point.edge$
  insert $pair(next_point.edge, i_pos)$ at the end of Q
end while
$next_crossing \leftarrow Q.pop()$
$current_vertex \leftarrow$ split $next_crossing.edge$ at $next_crossing.i_pos$
while Q is not empty **do**
  $current_crossing \leftarrow next_crossing$
  $next_crossing \leftarrow Q.pop()$
  $next_vertex \leftarrow$ split $next_crossing.edge$ at $next_crossing.i_pos$
  set $edge$ between $current_vertex$ and $next_vertex$ as selected
  $current_vertex \leftarrow next_vertex$
end while

over the face's edges to find an intersection using an adapted version of the method "Intersection of two lines in three-space" from *Graphics Gems* 1st ed., p.304 [1]. This method allows to verify whether a segment intersects, is parallel to, is collinear to or does not intersect another segment, which is useful for the discrimination of edge cases. Once the intersection has been found, we can mark it similarly to the method in Algorithm 2 and continue on to the next one. Because this is a recursive process, we must uniquely identify all the intersections between the two points of the jump to avoid looping indefinitely.



(a) Path over a surface, spanning multiple triangles. Circles denote hit points and red segments between them the interpolated path. We can see that the blue triangle does not encompass any hit point, making the path over it a jump.



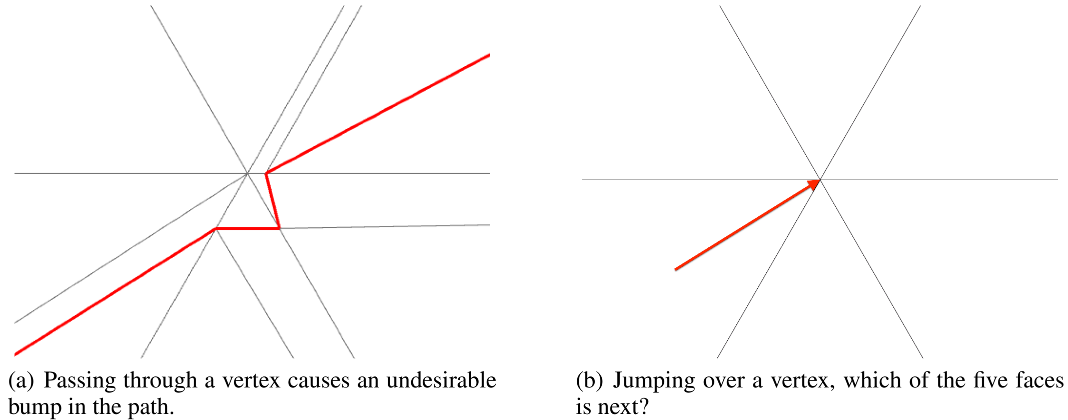
(b) Top: crossing over consecutive existing vertices should yield a path consisting of the edge between them. Bottom: a particular case of the top one, namely that a jump includes the crossing of two or more aligned vertices.

Figure 4: Jumps and vertex crossings are particular cases that require further attention.

3.4 Avoiding same intersection loop

We have now found an intersection and position the new beginning of our walk on it to find the next, on the opposite face. This scenario shows that when searching for a crossed edge it is possible that the same intersection is found again, looping forever on this same position.

A way to get around this situation is to push the new starting point inside the face in order to step off of the edge. We can find an intersection of the outgoing segment with one of the triangle's medians and push the starting point to this new intersection. This ensures the new starting point to be inside the triangle never to be detected as an intersection again.



(a) Passing through a vertex causes an undesirable bump in the path.

(b) Jumping over a vertex, which of the five faces is next?

Figure 5: Intersecting an existing vertex.

3.5 The edge cases

Finding an intersection and going further means leaving a face to enter another, analogous to crossing an edge. Unfortunately a path does not always cross an edge. Consider the paths on Figure 4(b), observe how both traverse existing vertices and how no edge was crossed. There are two problems depending on whether we are in the top or bottom case. The first might cause curve glitches, shown in Figure 5(a), altering the curve's smoothness. A possible strategy would be to avoid splitting at the intersection found and save the closest vertex for later path selection. The second, as the walk includes a jump, causes the next face to be unknown. Indeed which one of the faces adjacent to the center vertex on Figure 5(b) should be the next step? To answer this we could iterate over the faces to find a new intersection for the path starting at the vertex.

3.6 Clamping

When drawing a curve, clamping to existing edges without modification to the mesh fills the gap between the single edge selection tool and the curve drawing tool. This method's design is similar to the latter in the way it reviews elements in the queue and only keeps those of interest. This time, a change of vertex is indicative of an element to record. If two consecutive vertices are different and their associated edge is the same, then we need to register them as the edge between is to be selected in an upcoming method. If they do not share the same edge, however, we can consider this step as a jump. Jumping is taken care of by breadth-first search from the current vertex with a cost incentive. This incentive allows to visit only the vertices with smallest distance to the end point, so the found path will be traced by reducing the distance from the newly found vertices to the destination, the end point of the jump. There are two precautions to be taken for this tool to work correctly.
The first is to take into account edges whose endpoints are twice the distance away than other potential vertices, or more. It is the case for a triangle whose hypothenuse you want to register and the opposite angle is right or obtuse, and we see that the opposite vertex will always register even if the path is closer to the hypothenuse. Because of this, we need to verify that each registered vertex is connected to the corresponding edge, if not we simply skip this vertex and go on to the next one in the queue.
The second issue that needs to be addressed is in the case of a jump. As previously stated, BFS is used to find a path from the beginning of the jump to the end, decreasing the distance at every step. To prevent cases where the same vertex is visited twice we need to memorise and avoid them during the same jump. This is easily resolved by remembering all vertices already visited, and clearing this memory once the jump is over. Doing this prevents the algorithm from indefinitely

oscillating between two vertices and guarantees termination. A way to improve this tool would be to use Dijkstra's algorithm for shortest path finding, which could be part of a topic for future work.

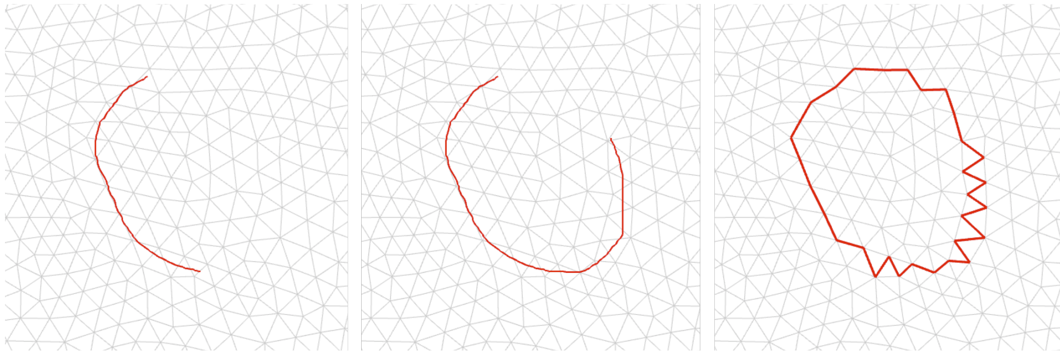


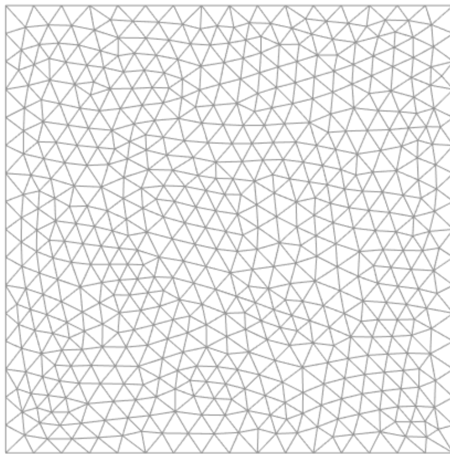
Figure 6: The path drawn with the mouse is clamped to the closest continuous edge path.

Part II

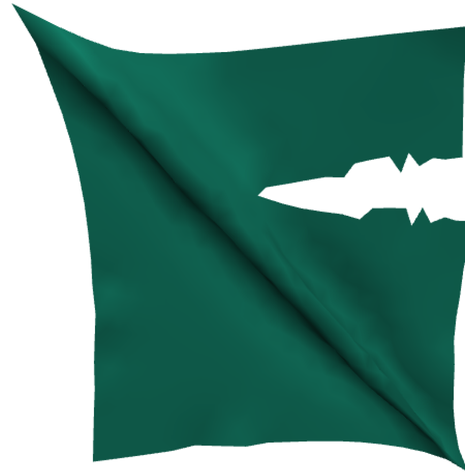
Physical behaviour

This part is dedicated to the use of the ShapeOp library, which is a set of tools for static and dynamic geometry processing for optimisation under constraint [2]. Without focusing on its mechanism too much, the main concern will be the possible applications.

4 Domain of application



(a) Base mesh for constraint testing, a square of triangles with apparently random structure.



(b) Cut and deformed mesh. Attached at the top left corner and pulled from bottom right one.

Figure 7: Base square mesh used in Part II.

Surfaces can be given physical properties to mimic real-life materials. We will use ShapeOp constraints to give the studied meshes such qualities. For the purpose of demonstrating the shape-altering tools I will mainly use a flat sheet composed of 568 vertices created from a square made up of two right triangles. The initial square was remeshed multiple times with the smallest edge as target length, using the *Isotropic Remesher* plugin from OpenFlipper. The result is a flat square mesh made up of roughly equilateral triangles arranged in no particular order. We can see this mesh on Figure 7(a). On Figure 7(b) the same mesh, coloured in green, is cut near the upper right corner and stretched in the top left to bottom right direction. In the next part I will use more types of meshes, triangular and quadrilateral, of different shapes, connectivity and scale for the purpose of the specific application.

5 Constraint exploration

5.1 Closeness constraints

This sort of constraints is useful for setting vertices to a desired position. In particular we can fix certain vertices and set handles on others to be able to move them around at will. In MeshCut we can select any set of vertices and fix them using a closeness constraint with a relatively high weight. The weights in ShapeOp are a measure that assign relative importance to all constraints. Therefore the fixed vertices are given the highest weight to ensure they really do remain motionless. To allow for interactivity, it is possible to set handles on vertices. These are, in the likes of fixed vertices, assigned a highly weighted closeness constraint. However, unlike their fixed counterparts the importance of respecting the mouse motion is lower, so the weights are set to half of the maximum range defined in MeshCut.

5.2 Edge strain constraints

For the simulation of plausible material behaviour such as paper for example, we can set constraints on edges to retain their length close to their original value. On the right side of Figure 8 we can see how the sheet behaves when pulled to the right with the top line vertices fixed. Notice how the object seems to remain rigid compared to the image on the left. Of course the stiffness only holds when using triangles, as a quadrilateral mesh for example would shear. Edge strain will be used a lot in Part III because it is able to simulate paper well.

Figure 8: Two configurations of a sheet fixed at the top and pulled from the bottom right corner. Left: triangle strain; right: edge strain.

5.3 Triangle strain constraints

Instead of restraining the edges to a set length, we can drive triangles to retain their original shapes. While the result is not too dissimilar to using edge strain constraint, we notice that in this case the material is more supple, which is a reminder of stretchable cloth fabric, as mentioned in [3]. An indicator of the difference between these two instances is the way the left border bends. On the left it does so toward the inside of the object while the more stiff sheet on the right appears not to bend much or even to do so in the opposite direction, although this is due to the surface bending in depth rather than stretching.

5.4 Area constraints

For stretchy materials, area constraints are useful. They preserve the area of triangles, which makes it possible to stretch edges independently. Furthermore, by setting the minimum and maximum ratios with respect to the initial area, one can make the surface stretch inward or swell outward. An example of inward stretching can be seen on Figure 9. The sheet has a cut at its center, we can see the opening reaching outward as if it was compelled to expand, and the outer borders stretching inward. This is because the sheet itself, in particular every face independently, is shrinking and pulling on its fixed extremities. This effect is obtained in several steps, at each of which the upper bound of the area constraint is lowered and the solver positions reset.

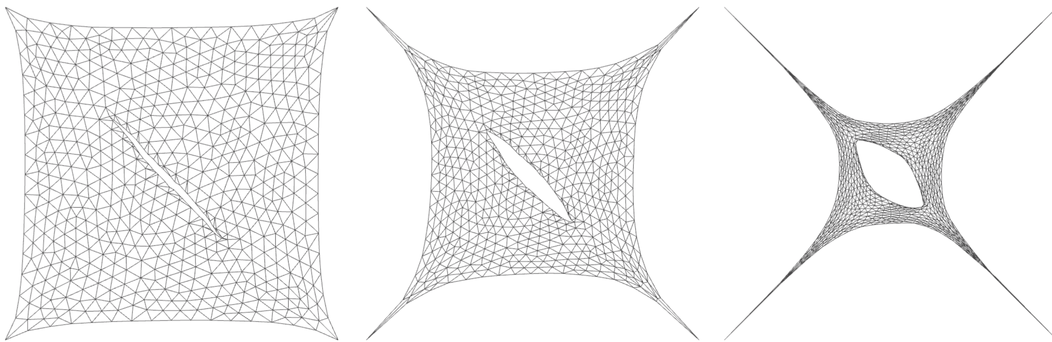


Figure 9: Gradually lowering area constraint's upper bound ratio stretches the surface inward.

It is also possible to combine constraints together in order to obtain more complex material properties. In the following example I used both triangle and area constraints to model an elastic material. Introducing a lateral cut accross its center and fixed by three of its corners, the sheet is pulled by its last corner in the direction perpendicular to the cut, deforming into the result seen on Figure 10. Notice how wrinkles form around the cut and extremities extend, just as we would expect from a material such as a rubber sheet for example.

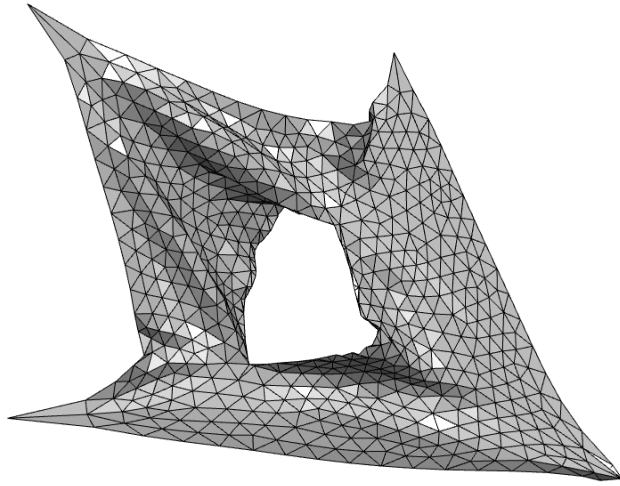
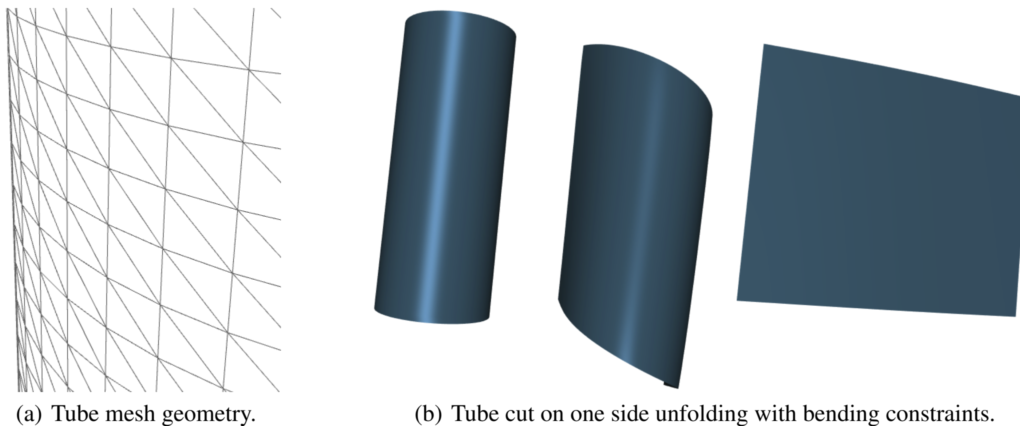


Figure 10: Pulling on the corner of a cut sheet, fixed by its other extremities.

5.5 Bending constraints



(a) Tube mesh geometry.

(b) Tube cut on one side unfolding with bending constraints.

Figure 11: A tube mesh.

Some materials are easier to bend and fold than others. Certain types of paper will oppose more resistance to bending and others, maybe thiner ones, less. This can be modelled with bending constraints. On Figure 11(b), we can see a tubular object, sliced along its length and fixed near the cut. Bending constraints with a maximum curvature of zero are then applied all over the mesh and we can observe it unfolding into a flat sheet over several iterations.

Notice that as the mesh unfolds it bends down. This is because of connectivity, and shows the importance of the geometry of the mesh for constrained surfaces. Indeed as seen on Figure 11(a) the mesh is composed of triangles with hypothenuses all pointing in the same direction. The orientation here is critical, as during the unfolding process, because of the global effect of local constraints, some of the initially unbent elements will do so to make up for the excess of already bent elements.

5.6 Rectangle constraints

Contrary to triangles with constrained edges, rectangles do not have fixed angles. As we will see in Part III when manipulating quadrilateral meshes it is useful to use rectangle constraints to preserve the right angles on each face, in other words to prevent shearing as demonstrated on Figure 12. This is particularly interesting when applied to a tiled quad mesh with special connectivity such as hinges.

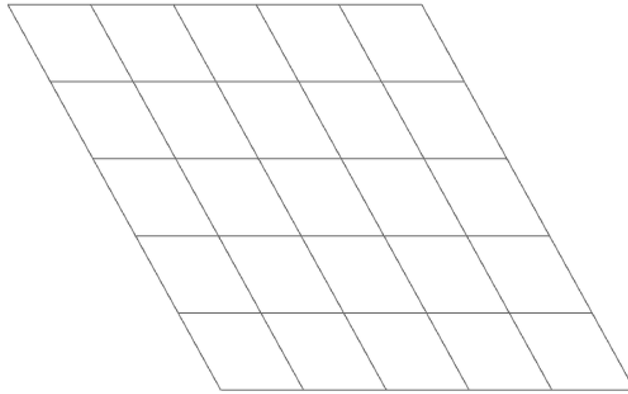


Figure 12: Quadrilateral mesh, sheared by pulling bottom vertices to the right.

Part III

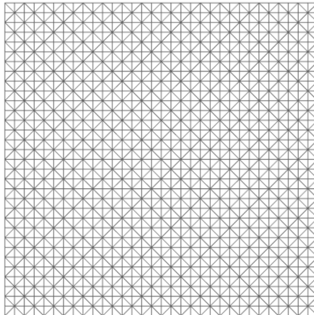
Cutting patterns and applications

In this part we will put to practice what has been developed so far to see what cuts can be applied to a mesh and observe interesting behaviours.

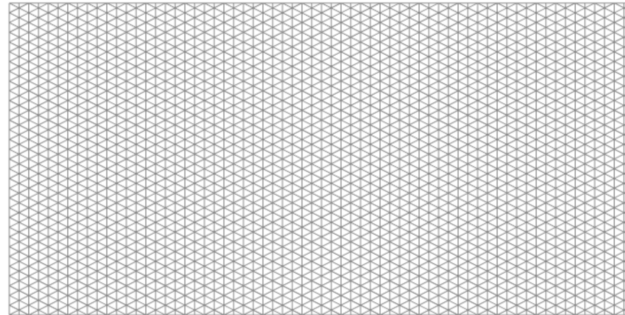
6 Mesh generation

6.1 Hand-made meshes

The following meshes have been created by hand from a simple square mesh, composed of four right triangles, in a similar manner as the base mesh in Part II. The square has been subdivided and manipulated deliberately to obtain these results. These geometries where intentionally created for the subsequent cuts to follow existing edges, to be meaningful to both the topology and geometry of the deformed mesh.

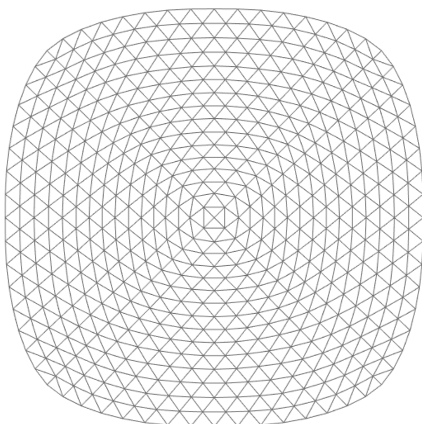


(a) Square mesh evenly tiled with right triangles.

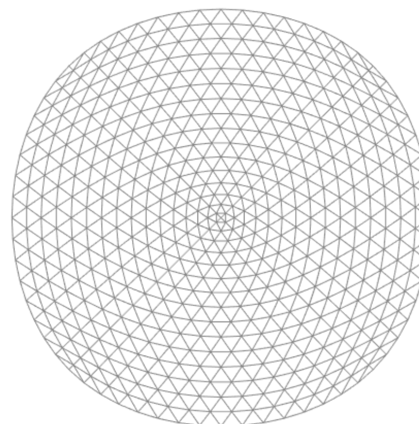


(b) Rectangle mesh evenly tiled with isosceles triangles.

Figure 13: Base rectangular triangle meshes, generated by subdivision. They will be used for the creation of garlands.



(a) Mesh obtained from rounding a square mesh by subdivision using *Modified Butterfly* method.

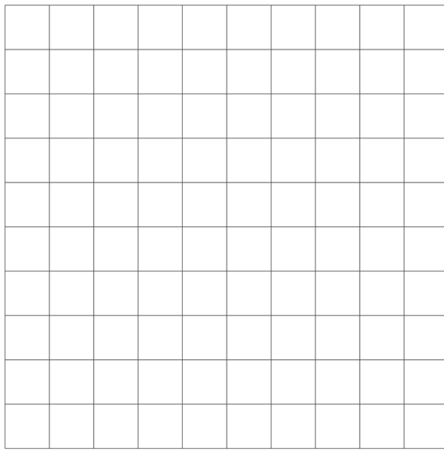


(b) Round mesh obtained through subdivision using the *Loop* method.

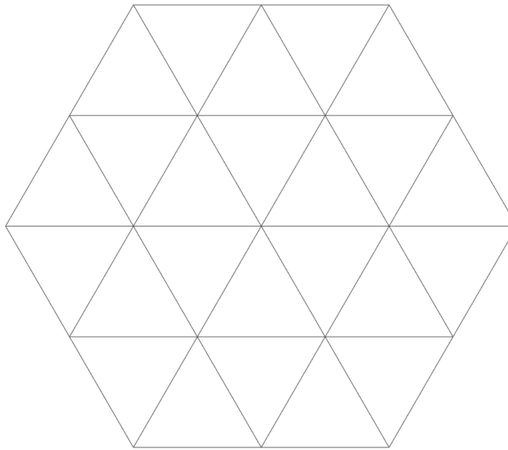
Figure 14: Meshes obtained from a simple square composed of four right triangles subdivided with OpenFlipper's *Subdivider* plugin's methods.

6.2 Computer-generated meshes

The MeshCut plugin provides the user with a mesh generation tool to create surfaces of two types. The first, an example of which can be seen on Figure 15(a), is a rectangular quad mesh, formed of squares, with given width and height that dictate the aspect and therefore the total number of faces ($= width \times height$).
The second is hexagonal in shape and consists of equilateral triangles, as seen on Figure 15(b). Its size is proportional to the argument given by the user as input. This length can be viewed as the hexagon's radius, or the minimum number of edges necessary to traverse from the center to the border. An input of 0 generates a single triangle and a radius of 1 or more create triangle tilings such as the one seen in Figure 15(b). The generation process is iteratively circular, as it builds up counter-clockwise, layer by layer. The top vertices of a layer are memorised until completion of that layer for the next, which allows future triangles to connect correctly to their sublayer counterparts.



(a) Quadrilateral tiling of a rectangular mesh.



(b) Triangle tiling of a hexagonal mesh.

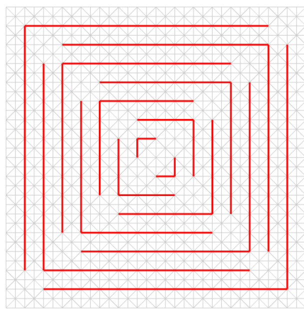
Figure 15: Meshes that can be generated in MeshCut.

7 Cutting patterns and deformation

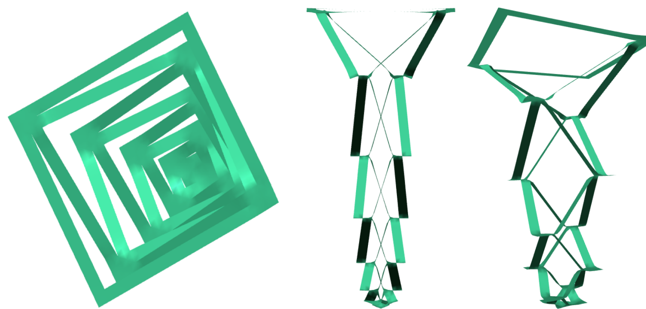
The meshes presented in Section 6 are now cut following patterns shown in this section, then deformed mainly constrained with edge strain on all edges, because of the paper-like aspect it single-handedly provides on triangle meshes. Quad meshes need additional restrictions to remain consistent, such as rectangle constraints.

7.1 Garlands

Garlands are decorative structures, sometimes made of paper, that can extend and be hanged on display. MeshCut allows to create parts of garlands from surfaces that can be cut and composed together to form larger strings. The process of creating those parts is made easy thanks to the plugin's various tools. A possible workflow is the following: after drawing and applying the desired cutting pattern, one could fix the base of the mesh, for example the outer border, and pull on the center to see it take its desired shape. There are infinite possibilities for base meshes, cutting patterns and deformations, of which here are a few.

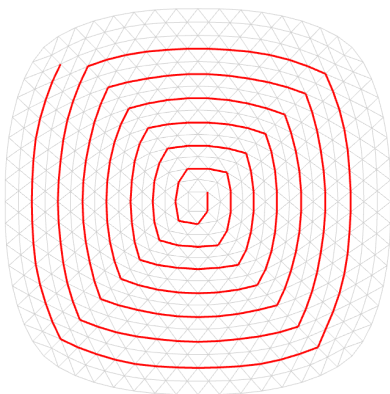


(a) A pattern is shown in red, drawn by hand and designed to cut the underlying mesh.

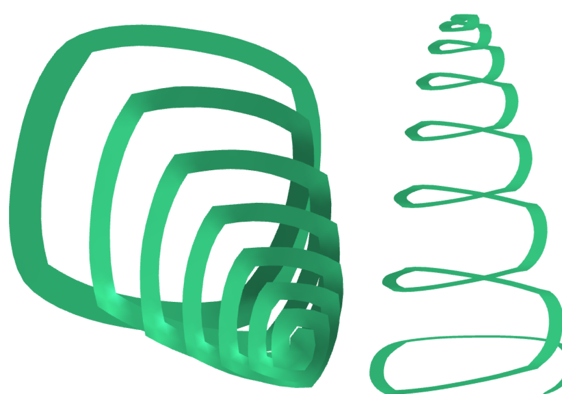


(b) Views of the component of a square garland. From left to right: slightly pitched view from the top, a side view and a tilted low side view.

Figure 16: The square triangular mesh was cut by hand in an alternatingly rotated right angles pattern on the left. On the right, different views of the resulting deformed mesh.

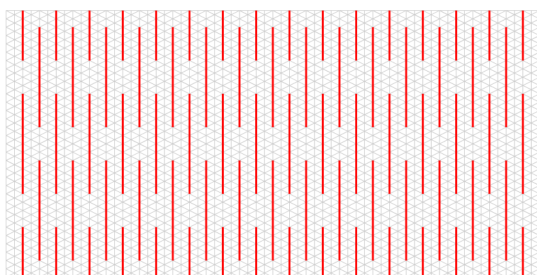


(a) Vortex pattern on a rounded square mesh.

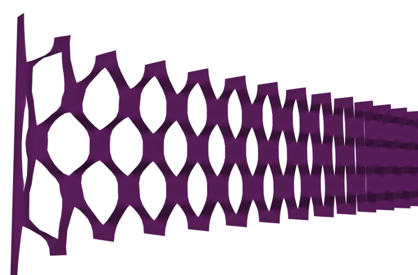


(b) Pulling center and border apart elongates the cut surface to produce a tree-like shaped object. Left: tilted view from the top. Right: side view.

Figure 17: A vortex pattern turning into a christmas tree.



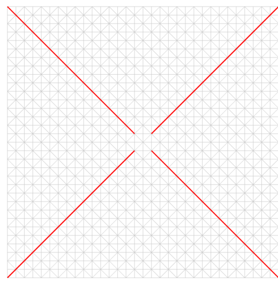
(a) Rectangle mesh with repeating cutting pattern.



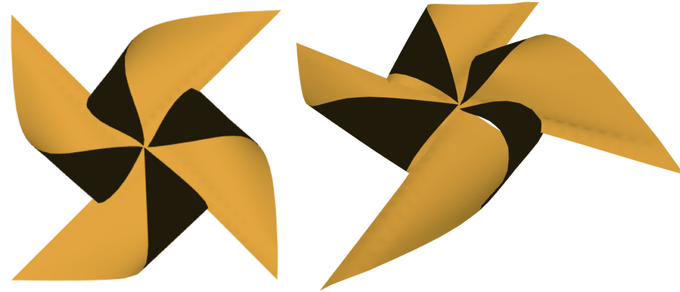
(b) Net garland, seen in depth at an angle.

Figure 18: A net garland obtained from the cut rectangular mesh by fixing the three leftmost vertex lines and pushing forward the handle made up of the three rightmost vertex lines.

7.2 Other objects

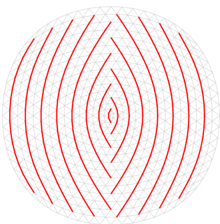


(a) A square cut from each corner towards the center.

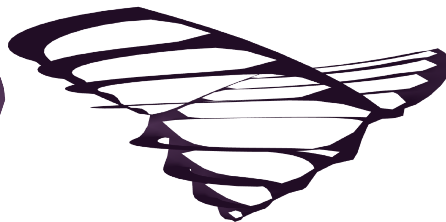


(b) Half of the branches are bent up towards the center. Curls are due to surface stiffness.

Figure 19: A *joulutorttu* is a Finnish christmas delicacy made with puff pastry and plum jam that looks like a pinwheel. The bent branches were handled using bending constraints to preserve some volume during the fold, which gives it its delicious aspect.



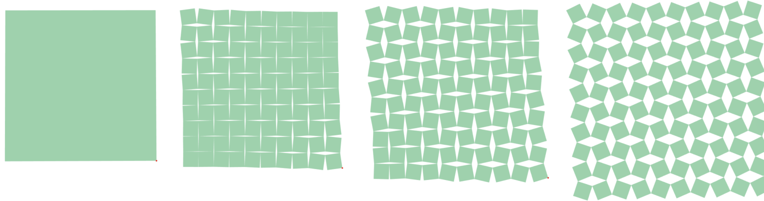
(a) Cutlines form a pattern that looks like fish gills.



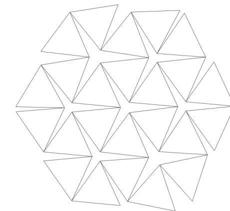
(b) Only edge strains were used to simulate paper properties. Left: tilted top view. Right: side view.

Figure 20: This basket-shaped object uses the round mesh as a base and is cut along rounded lines as depicted by the red pattern. After being cut, the right and left sides of the mesh are fixed and the center pulled downwards.

7.3 Hinged tessellation



(a) A hinged tessellated quad mesh opening under edge strain and rectangle constraints. The top left vertex is fixed and the bottom right pulled. Notice how the openings are making the surface wider.



(b) Triangular tiling requires only edge strain for the faces to retain their shape.

Figure 21: These meshes have been hinged tessellated upon generation and their connectivity did not require any manual intervention.

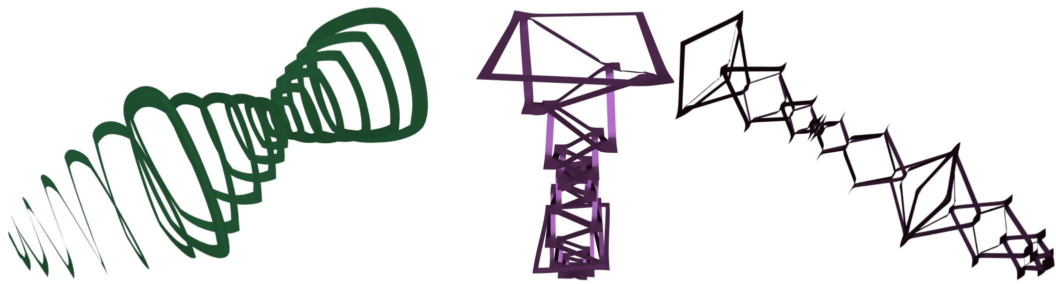
The quad mesh on Figure 21(a) was generated as a 10×10 grid of squares, each connected to its neighbours in an alternated fashion, more precisely each interior edge was alternatively connected

by only one of its vertices. The hexagon-shaped triangular mesh on Figure 21(b) could also be hinged tessellated in a similar way, being cautious about each layer's bottom linkage to the previous layer.

8 Multiple meshes

Combining multiple meshes together allows to create garlands out of smaller components such as those seen previously. In order to perform this task, the user must combine two or more meshes into a single one containing several components. These components can be attached together by their borders, effectively merging them into an individual object. Here are possible examples of such combinations.

8.1 Garlands



(a) Three vortex meshes bound together by the tip once and by the base another.

(b) Three square garland meshes glued together. This result is starting to look like a real garland.

Figure 22: Real garlands are made up of many similar components, such as the objects depicted here.

8.2 Lantern

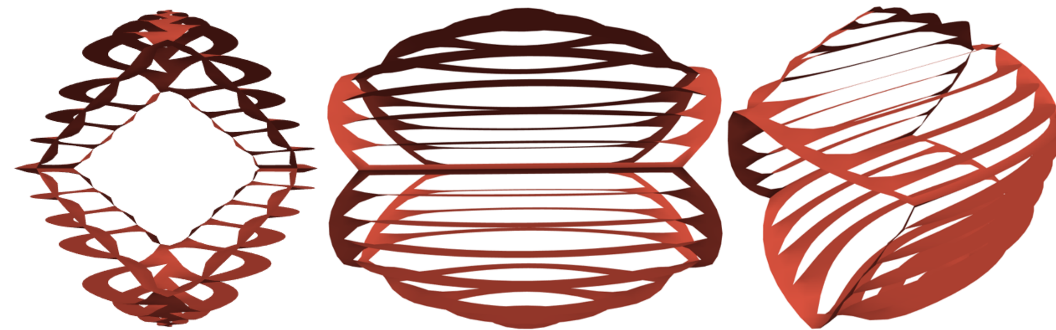


Figure 23: A paper lantern built from the combination of two round basket meshes, glued together at their left and right side borders, fixed at and pulled from their respective center.

Conclusion

The various tools created in this project allowed me to experiment with surfaces and cut properties of several tiling geometries. Cutting through a mesh gives it mobility, as is obvious from hinged tessellated surfaces. Complemented with shape constraints, surfaces become real-life materials that can stretch, bend and twist. Cuts can follow patterns that let three-dimensional shapes emerge, enabling the creation of more complex objects such as garlands and lanterns emerging from the combination of paper structures.

Future work

There are some possible developments to the current state of MeshCut, which I will briefly mention. Cutting the mesh can be improved in both free-drawing and clamping tools. The most problematic topic is robustness to jumps, as being able to precisely register a path requires precision analysis for free-form curves and graph methods such as Dijkstra's algorithm for shortest path finding.
Providing a more complete implementation of available ShapeOp constraints can also encourage the user to explore more physical properties. It would be interesting for example to expand to volumes.
Subdivision of hinged tessellated meshes is also a possible interesting addition. On a regularly tiled surface one could first subdivide each face and adjust connectivity to neighbouring elements. Using edge strain constraints would allow for length preservation and to converge to an open-tiled solution giving the impression of zooming out.

Documentation

The code for the plugin was written with the help of the following documentations:

OpenMesh: http://www.openmesh.org/media/Documentations/OpenMesh-Doc-Latest/index.html
Eigen: http://eigen.tuxfamily.org/dox/index.html
OpenFlipper: http://openflipper.org/Documentation/latest/index.html
ShapeOp: http://shapeop.org/ShapeOpDoc.0.1.0/index.html
C++ containers: http://www.cplusplus.com/reference/stl

Acknowledgements

The base structure of MeshCut was inspired by the *Topology* plugin, and in particular the split_edge function. Thanks to Florian Brandherm, who answered me on the OpenMesh mailing list, I was able to understand how splitting a single edge in two for the creation of a cut in between would be possible with the halfedge data structure of OpenMesh. I want to thank Alexandru Eugen Ichim for helping me understand aspects of ShapeOp better, and also for compilation tweaks. I want to especially thank Mina Aleksandra Konakovic for her help and patience throughout the project. I am also grateful to Prof. Mark Pauly and the LGG for allowing me to do this project.

References

[1] R. Goldman, "Intersection of two lines in three-space," in *Graphics Gems*, p. 304, Academic Press Professional, Inc., 1990.

[2] S. Bouaziz, M. Deuss, Y. Schwartzburg, T. Weise, and M. Pauly, "Shape-up: Shaping discrete geometry with projections," *Comput. Graph. Forum*, vol. 31, no. 5, pp. 1657–1667, 2012.

[3] S. Bouaziz, S. Martin, T. Liu, L. Kavan, and M. Pauly, "Projective dynamics: fusing constraint projections for fast simulation," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 154:1–154:11, 2014.