Alexei Baboulevitch

CS284 - Final Report

The main goal of this project was to take a "regular map", which is a symmetric 2D tesselation of a 3D object, and turn it into its corresponding 3D shape. Many regular maps have been discovered algorithmically, but we have no idea what most of them look like in 3D. Folding them up by hand, especially for higher genus, is unfeasible, which is why we need a program like this to do it for us. Ideally, it would create an aesthetically pleasing representation, but its main benefit would be to provide a rough idea of the mapping.

My approach consisted of taking the regular map and transforming it according to an algorithm from a topology textbook by William Massey (which involves non-linear distortions of the map, and cuts between arbitrary vertices), which eventually recombines it into the topological representaiton of the sum of several tori. These can then be mapped onto carefully designed torroidal "fold-outs", which would be re-folded into the 3D object and maybe even subdivided.

My initial goal was to enable the cuts and distortions utilized by the topology algorithm. I settled on using a compatible-triangulation-based transformation, as described in "As Rigid as Possible Shape Deformation" by Marc Alexa et al. (The paper was more about creating smooth transitions between the two shapes, which I didn't need. I got the compatible triangulation algorithm itself from "On Compatible Triangulations of Simple Polygons" by B. Aronov et al.). The algorithm transitions from one polygon into another by tesselating both of them, mapping the tesselations onto a symmetric n-gon, intersecting the tesselations, and moving the vertices back. I picked this approach because it was adaptive to some extent and didn't require solving complex equations -- the distortion was approximated by the triangulation.

I started out by scavanging parts of my genus 4 homework assignment to create an OpenGL applet. I decided to use OpenGL instead of other environments because it had hardware-accelerated triangle support (which is what the algorithm uses) and because it allowed for both 2D and 3D applications -- which is great for constructing a 2D surface that folds into a 3D object. (I simply set the z-axis coordinates of every point to 0 and the projection to orthogonal.)

I decided to use a texture to represent my regular map because I figured it would be easier for a first draft of the application (less things to keep track of, such as inner edges and vertices), and because the barycentric mapping is already taken care of by OpenGL's texture support. The texture is imported using PyImage, and the edges and vertices are parsed from a text file. Each edge and vertex is assigned a label in accordance with the regular map, even though their positions may be different.

Next, I created a set of classes to represent the shape: a texture class (representing the main shape) with a list of polygons, a polygon class with a list of vertices, and a vertex class with a set of coordinates and texture coordinates. These classes were modified substantially as I worked my way through the problem. For instance, after I realized that cutting and moving polygons wouldn't work with vertex-based texture coordinates (since neighboring polygons could have different texture coordinates at their common vertices after a cut), I gave each polygon texture coordinates instead. In order to utilize OpenGL shading, I initialized a normal vector with each polygon. The texture class got a list of all the vertices (so that I wouldn't have to iterate through every polygon), and optionally allowed initializaiton without specifying the polygons. (Instead, the triangulation routine triangulated all the starting vertices.)

I made the applet interactive from the start, allowing the user to drag the vertices with their mouse. Since the projection is orthogonal, mapping from a pixel on the screen to the actual coordinates is as simple as applying a linear transformation. The manual control was intended mainly for demo purposes.

The topology algorithm begins with a possibly concave polygon, defined by a set of outer connected vertices. I didn't initially want to create a separate edge representation, so I gave each vertex a "left" and "right" neighboring vertex, which are set to None by default (this is preserved for inner vertices). Anything to the right of a vertex/right-neighboring-vertex edge is on the outside of the polygon, and vice versa.

In order to triangulate the shape, I started with a Delaunay triangulation, as specified in the shape deformation paper. In order to do this, I tried using several libraries, including scikits and Delny. However, they used unconstrained Delaunay triangulations, meaning that the all the vertices were treated as a point cloud and the outer edges were ignored.

At first, I tried to remove the "outside" polygons algorithmically, eventually settling on a graph search over the polygons. I started with a polygon I knew for sure was "inside" because it had a vertex/right-neighboring-vertex edge going counter-clockwise (and thus making the polygon "inside"), and then traversed its neighboring polygons. Each time I crossed an outer edge, I knew I was opposite of where I had just been -- "inside" became "outside" and vice versa. (I had no edge data structure and my vertices did not contain neighboring polygon data, so edges were first mapped to polygons using a search over all the polygons. This is a very inefficient way of doing things, and I intend to add additional structure to fix this.)

Unfortunately, this method did not prove successful, since there exist cases where a Delaunay triangulation avoids creating polygons with some of the outer edges, thus making culling the "outside" polygons impossible. After some looking, I discovered constrained Delaunay triangulation, and used the CGAL Python module to do it.

My Delaunay triangulation algorithm first iterates through the shape's vertices and picks out the outer edges, adding them to a CGAL triangulation object if the shape has no polygons (as it does when the program is first started -- only the vertices are passed to the new texture object.) If the shape has polygons, these are iterated as well. All the outer edges are added to the CGAL triangulation object, as well as any other constrained edges. (To clarify, there are two iterations of adding constraints -- one for when the shape is initialized without polygons, and the other for when the shape has polygons. Also, the first iteration works by finding the first vertex on the outside of the polygon, and then following its "right" neighbors until it reaches the same vertex again. These are added to an array to form a consecutive list of outer edge vertices, which are used to remove concavities later on. The second iteration through the polygons would not find these edges in consecutive order. This system is somewhat inefficient, and I would like to contain my search to one loop if possible.) After this, I put the new polygons through my algorithm for removing concavities, leaving me with the final triangulated shape.

After the triangulation algorithm was in place, I added a nearly identical one to the polygon class. I did this so that I could have polygons with arbitrary numbers of vertices, instead of just triangles. As they stand now, the texture and polygon classes overlap quite a bit, and I don't think having a separate triangulation routine for the main texture object is necessary. I could just as easily store the untriangulated texture as an "edgeless polygon" and triangulate it on initialization.

Next, I needed a procedure that would let me transform the shape into a symmetrical n-gon. This was fairly easy to do by multiplying a vertical vector several times by a rotational matrix, and mapping the vertices to the corresponding points.

After this, I needed to be able to intersect edges at arbitrary points while keeping everything triangulated and without creating duplicate vertices. I did this at first by expanding my applet to create arbitrary cuts, but this was too complex for the algorithm

I was trying to implement. I only needed to cut across pre-existing vertices, so I modified it to do that.

The algorithm works by going through all the polygons and creating edges for each pair of vertices. Edges that have already been tested for intersections are stored in a set. Edges that include either of the two intersecting vertices aren't considered. Then, the intersection between the two intersecting vertices and the edge is calculated by creating the appropriate parametric equations and solving a matrix using numpy. The results of this are two values of t, one for the line between the intersecting vertices and the other for the edge. These values are used to calculate the location of the new, intersecting vertex and to interpolate the texture coordinates. Finally, the difference between the coordinates of the new vertex and both of the edge vertices are calculated; if it's below an error threshhold, the vertex is discarded. (This is done in order to avoid numerical error, which can create duplicate vertices.) The new vertex is then stored in the edge set. After this, the new vertices are inserted into the correct positions in the polygon's vertex array. In the end, the polygons are triangulated and added to the texture's polygon list.

Because the polygons are only intersected by one line at a time, there is guaranteed to be a straight line between the newly created vertices. When new vertices are created at intersections, they are set to move linearly along the line between the two intersecting "control" vertices. These dependent vertices are added to a separate list in the main texture object and updated whenever any of the control vertices are moved. One of the parametric solutions from the matrix solver is used to specify where along the line they should be. This is an easy way to take care of the inner vertices, but it causes problems later on down the line.

I then added "temporary coordinates" to each vertex, which can be positioned by the user. This does not move the actual vertices, but instead creates a "ghost" shape which the current shape is meant to transform into. I also added a "temp" boolean parameter to the triangulation algorithms, so that I could triangulate the temporary polygon without influencing the real one.

With triangulation, mapping to an n-gon, and intersection in place, I could easily implement the compatible triangulation and transformation algorithm. After the user moves the temporary vertices into the desired configureation, the temporary and actual polygons are triangulated using Delaunay (with constraints on the outer edges of the polygon, and also on edges between dependent vertices with the same control vertices).

The actual shape is transformed into an n-gon, and all the edges from the triangulation of the temporary shape are intersected with it. Because all the dependent vertices move along with the intersecting edges, the n-gon can then be linearly transformed into the temporary shape with no intersections.

Unfortunately, after one step, the inner dependent vertices create boundaries between their control vertices, thus creating hard-to-avoid "walls" inside the shape. This means that it can no longer be distorted arbitrarily, since crossing one of these lines violates the constraints. If I don't do this, then the inner vertices will have to be moved some other way, so that they don't end up outside the shape boundaries. The solution to this still eludes me, although I suspect it can be done by merging polygons.

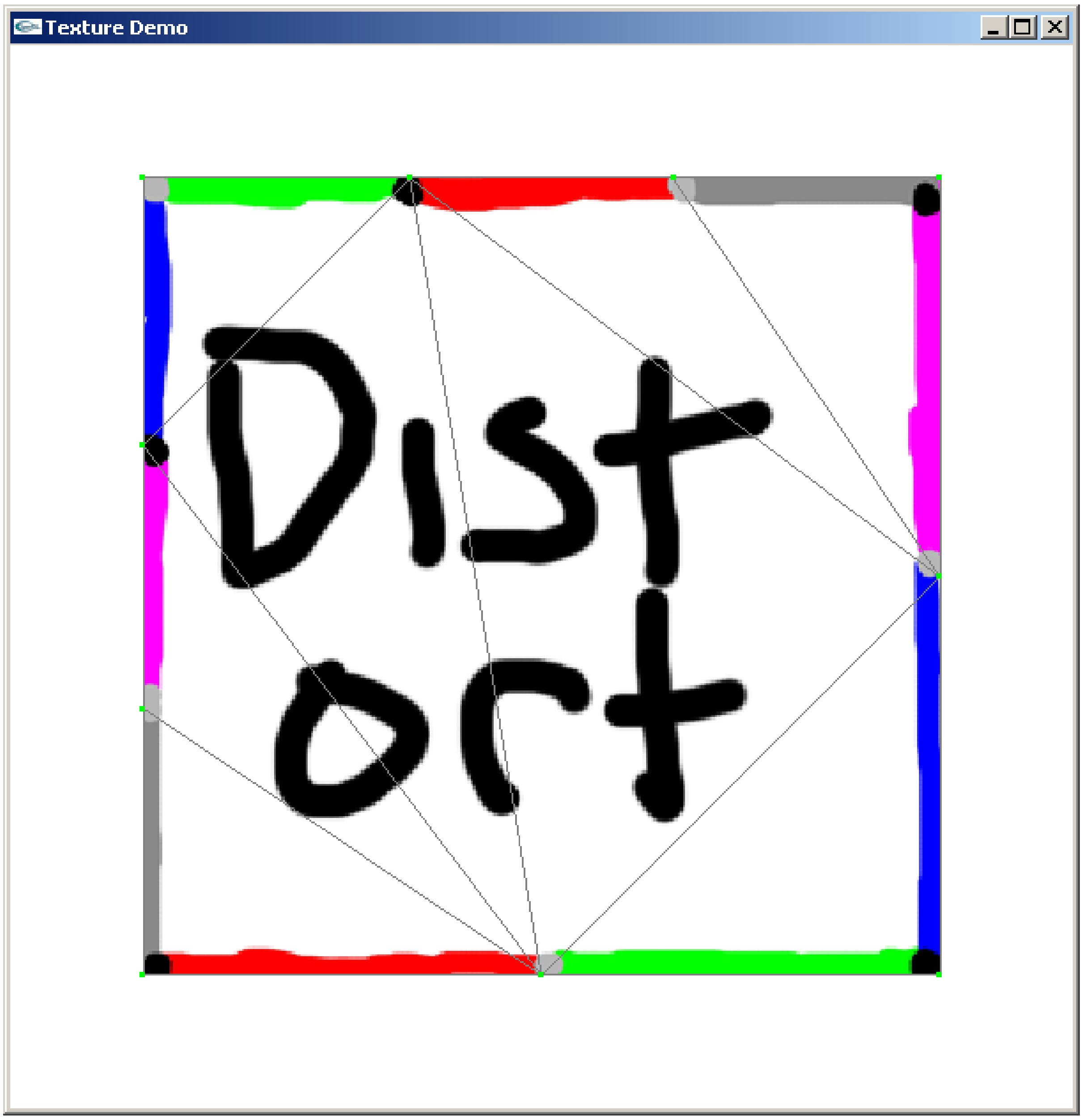
In order to be able to stick one part of the shape onto another, I need to be able to cut across an arbitrary vertex-to-vertex line, update the outer edge representation for each vertex along the cut (that is, update the "left" and "right" vertices for each vertex between the outer vertices along the cut), update any dependant vertices to follow the vertices along the cut, rotate and translate the non-merging vertices, and merge the merging vertices. This has not been implemented yet, but to make it easier, I'm considering adding an edge object. This would also prevent searching through all the polygons to find the edges, which occurs in several of the algorithms.

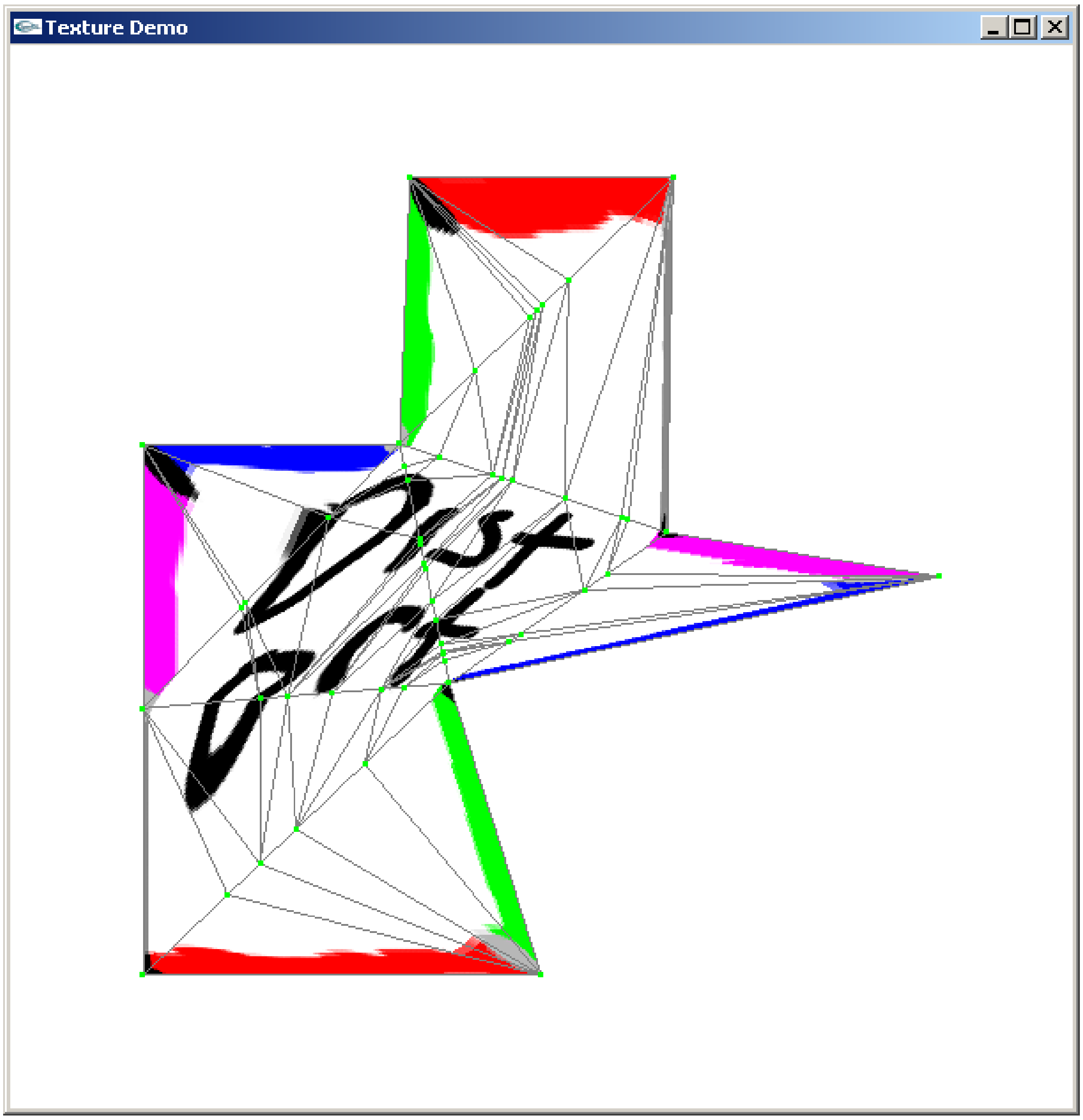
Over the course of this project, I learned about how malleable the topology of a shape can be. It's surprising how easily a regular map can be cut and distorted without the final result losing any coherence, and also how a flat, continuous 2D representation can be folded up into a 3D object with openings. I also learned about various different kinds of triangulations, including Delaunay and constrained Delaunay, and about how triangulations can be intersected in order to create a smooth transition from one shape to another. Before settling on the compatible triangulation method, I explored various methods of distorting a texture inside an arbitrary polygon, and it wasn't nearly as easy as I thought it would be. Unfortunately, I ran into several problems (namely, the inability to perform a compatible triangulation reliably after one iteration) and was unable to finish the project as I had specified it. However, I created a good framework for continuing to whittle away at this problem, and I intend to continue working on it over the next semester.

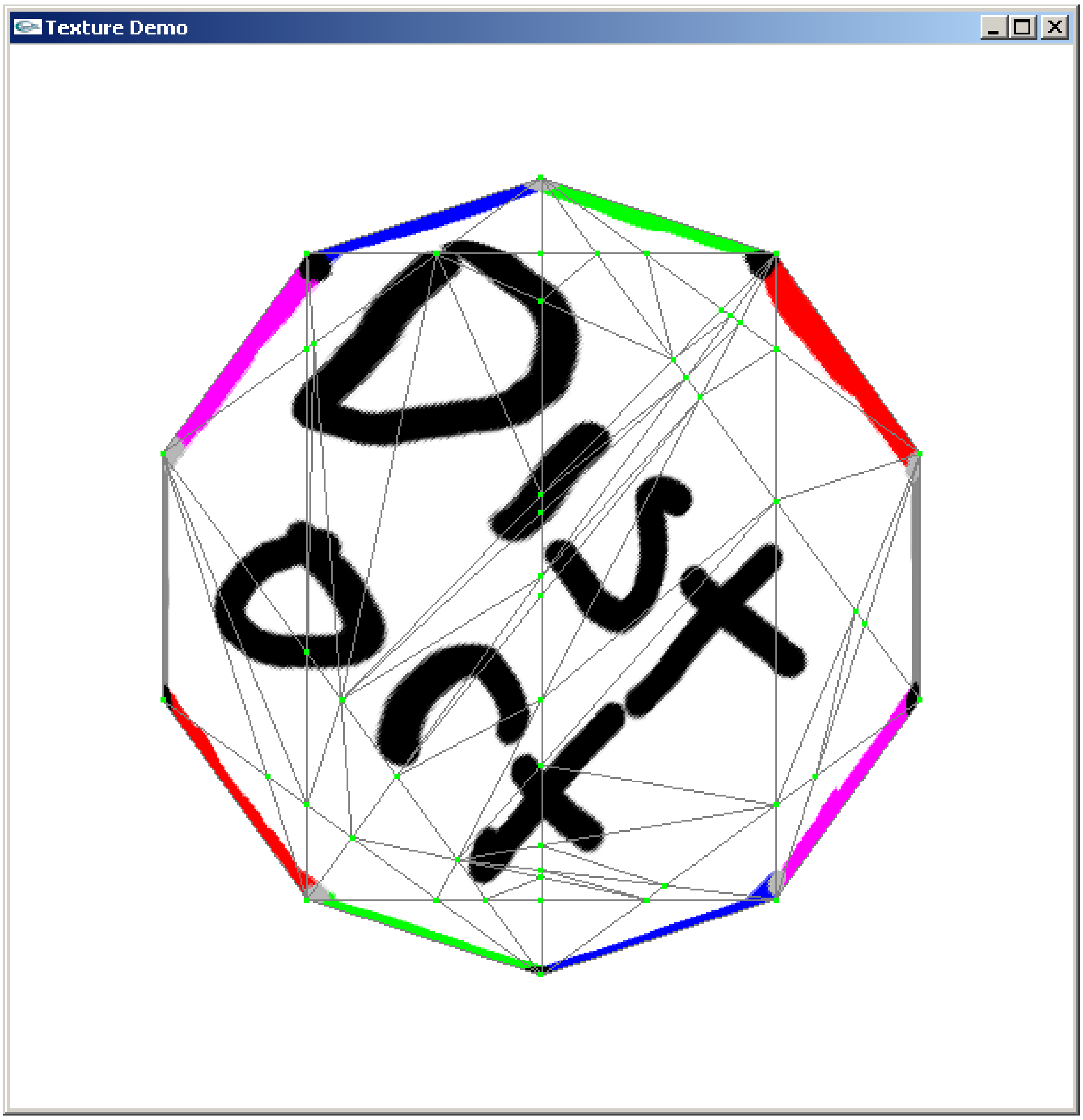
Manual:

* In order to run this script, you will need Python 2.4 installed, as well PyOpenGL, ctypes, numpy, PyImage, and CGAL for Python -- all for version 2.4.
* Run the program by running " "[Python Directory]\python.exe" texture-distort.py image text ", where image is the name of a square texture, and text is a text file representing the regular map.
* Format of the text file: "x1: n, c1 c2" for each line, where x is either v (for vertex) or e (for edge), n is a name for the vertex or edge, and c1 and c2 are either the x and y coordinates for the veretx, or the numbered vertices for the edge.
* Move the vertices around by dragging them with your mouse.
* To move the temporary (blue) vertices, Shift+drag them.
* In order to transform the current shape into the temporary shape using compatible triangulation, press "i".
* In order to triangulate the current shape, press "t".
* In order to map the current shape onto an n-gon, press "m".
* In order to map the temporary vertices onto an n-gon, press "M".

Examples:


Texture Demo
Dist
ort


Texture Demo
Dist
ort


Texture Demo
Dist
oct