**Algorithms used in project**

**Delaunay Triangulation**:

This triangulation method was used for creating the proper triangles for the constraints on the texture and for the border points that connect to the mesh.

This works by creating all possible triangles for a set of points; however, we need to make sure that the vertices of the triangles are arrange in a clockwise order. We check this by taking a random point from the three vertices and calculate the cross-product that this point forms using the two vectors that it makes to the two other points. The product is positive if the vertices are already counterclockwise otherwise it will be negative.

Now that we have all possible triangles with clockwise oriented vertices, we remove all triangles that do not satisfy a Delaunay triangulation with respect to all other points that do not make up the triangle. The method to check is from Wikipedia (<http://en.wikipedia.org/wiki/Delaunay_triangulation>):

\begin{vmatrix}
A_x & A_y & A_x^2 + A_y^2 & 1\\
B_x & B_y & B_x^2 + B_y^2 & 1\\
C_x & C_y & C_x^2 + C_y^2 & 1\\
D_x & D_y & D_x^2 + D_y^2 & 1
\end{vmatrix} = \begin{vmatrix}
A_x - D_x & A_y - D_y & (A_x^2 - D_x^2) + (A_y^2 - D_y^2) \\
B_x - D_x & B_y - D_y & (B_x^2 - D_x^2) + (B_y^2 - D_y^2) \\
C_x - D_x & C_y - D_y & (C_x^2 - D_x^2) + (C_y^2 - D_y^2)
\end{vmatrix} > 0


Basically A, B, C are the vertices in counterclockwise orientation from the triangle we check against and D is vertex in the plane. If the determinant of the matrix on the right is bigger than 0 we know that this triangle does not satisfy a Delaunay triangle since D is inside the circumstances of the points A, B and C. Thus this triangle will be removed. In our case D, will be the vertices of all other points we used to generate triangles, except for the 3 points we used in the particular triangle. Thus, after checking all triangles for this condition, we are only left with triangles that together span the entire area without intersecting.

For creating the triangles that connect the mesh with the border points. We used a slightly smarter technique for efficiency reasons. We only created all possible triangles between the edge points and the border points, not between sets of edge points or all mesh points. Since that would generate too many cases to check.

**Matching**:

The matching was used to find what path on the mesh we need to go to connect the constraints in the mesh in same way we do for the texture triangulation.

This was done by first pairing up the constraints on the mesh and on the texture. Then we check what edges are formed on the texture between the constraints, which tells us what constraints on the mesh need a connected path. For every triangle on the texture we find the associated constraint vertices on the mesh and try find the path between them.

This is done by an algorithm called EdgeWalker, that accepts a starting vertex and destination vertex. From the starting vertex it checks what possible edges we can walk (that are connect to this vertex) and it chooses the edge it can walk by calculating the distance between the vertex at the end of the edge and destination vertex. In this case, we always walk the shortest path to the destination.

However, there are several exceptions in which we cannot choose an edge because it fails a certain of tests. The first test, is to check if another path by another constraint pair already used the vertex of this edge that we would walk to. To avoid intersections between paths, we cannot use this edge if that is the case. In addition, we cannot go back to a vertex that we already walked on otherwise we could end up in an infinite loop that usually happen around holes in the mesh. The last check we perform is if the orientation of the vertex to the destination and all other constraint points has the same sign as the orientation between the starting constraint and destination constraint and all other constraint points. This is done by creating a vector from the vertex to the destination and a vector from the other constraint points to the vertex and calculating the cross-product. The sign from the cross-product should agree with the sign of the cross-product for the starting and destination constraint point and the associated vectors. If the signs do not agree that means that a constraint’s orientation toward the path would change in such a way that it ends up in different sub-region. This means that this edge would block this path for this constraint; thus, this vertex can’t become part of the path to avoid blocking other paths in the future.

If it passes all the tests, we can walk on the edge and keep doing this until we reach the destination or we come to a vertex where we can’t choose any valid edge because they all fail the tests. In both cases, we terminate the algorithm and return all edges and vertices that we used for the path.