# Fast and exact geodesic distance on triangular mesh

## Motivation

Given a 3D mesh, one could want to be able to compute distance on the mesh. One possible application for an exact computation of geodesic distance on 3D mesh would be the ability to draw Beziers and Splines curves on those meshes.

## Existing methods and limitations

We could easily approximate distance between two points by limiting ourselves to edges, using Djikstra algorithm. Using this method, the shortest path between two given points would mainly consist of a walk along the edge of the mesh. This approximation might hold for graph that are sufficiently tight and regular, but the error with this method is arbitrary high if the mesh or the points are not well chosen.

## Exact computation of geodesic distance

Even if Djisktra’s method is not adapted to the problem we are trying to tackle, we can use the same paradigm to compute the exact geodesic distance : propagating information step by step, keeping at any given point in time the optimal information, and stopping when there is no information left to propagate.

In this case, the information is a “window” on an edge h. This window describes a way of reaching a subset of h from a point S called the source or the pseudo-source. Furthermore, the window can be reached from S using only straight lines in the planar unfolding of the faces that lie between s and the edge.

On one window, the distance to point S on the mesh is completely known, and if we know the distance from S to the starting point, we can derive the distance field over the window through S.

In a Djikstra-like manner, we would now like to propagate this information to edges that are across a new adjacent face. We can do this by adding the new face to the planar unfolding, then drawing straight lines from S through the window. This will define a certain number of new windows on which the distance field through S is known.

However, the new windows might intersect with existing windows that were already here. We then need to merge the existing windows and the new window so as to keep only the minimum distance to the starting point. We can then carry on propagating the resulting new windows.

We will continue this propagation-merge loop until no more windows are left to propagate. Barring any unreachable points, the distance field should then be known over the entire mesh.

If the method is pretty simple in theory, there are a few problems that arise on edge cases. Propagating the windows is done according to a set of rules described in the article, which we will explain in the second part. Merging was more loosely described, and we derived our own algorithm that did a correct, albeit not really computationally efficient job.

## Our Implementation

### Main Objective

Even if the implementation of the algorithm described in the paper provided is a goal in itself, we wanted to go a little further by creating a small video-game based on geodesic distance.

Given a complicated and big 3D mesh, the game objective is to find the “source” using only information given by the distance field (i.e. You’re getting closer or farther from the source). This information is represented using colours on the mesh. We take advantage of the exact nature of the algorithm using subdivision to represent the information on a much more detailed level that what could be achieved with an approximate Djikstra algorithm on the initial mesh.

### Data Structure

We chose to use the halfedge representation of graphs. This allows us to achieve balance between the size of the object we’re manipulating and the ease with which we can traverse the graph.

The only extensive custom data structure we’re implementing I the window we described in the first paragraph. A window is an object that can locally describe the distance field, and possesses method to be able to propagate itself along neighbouring edges.

### Propagation

Given a window, we first need to be able to propagate it along the neighbouring edge. We identified 6 main cases and a few sub-cases leading to different results.

In order to distinguish between those cases, we will introduce a few notations. Given a window (B0, B1) on an edge h, and knowing the source S associated to the window, we can identify M0, M1, M2 and M3 as the intersection between 4 pairs on lines. We then define a point as *valid* if it lies on one on the triangle edge. We distinguish between our cases depending on which of the previously mentioned points are valid.

#### Case 0

#### Case 1

#### Case 2

#### Case 3

### Merging

As we propagated a window on a new edge,

### Shortest distance of an arbitrary point

One we computed and propagated all the windows, barring any mistakes or error in the propagation, we can then compute the distance to the source of any point P on the mesh. We extracted three cases that we will describe here:

#### P is on an edge of the mesh

If P is on an edge e of the mesh, computing the distance is easy. We go through all the windows covering that edge, find the one containing P and return the distance of P according to the window.

#### P is a vertex of the mesh

If P is a vertex, we can also compute the distance by examining all edges containing P. For each one of those edges, one window will contain P, and define a possible distance. We simply choose the minimum of those distances as the correct value for P.

#### P lies inside a face

If P lies inside a face of the mesh, we can go through all the windows that cover the edges of the face. For each window, we can compute the minimum distance we can achieve by going through this window immediately as the formula is derived from the Pythagorean theorem. Therefore, we only need to minimize this distance by going through each window once.

### Representation of the distance field and subdivision

We now have the ability to compute the distance from any point to the source. To represent the information, we can use the tools Processing offers to colour each face according to the distance. For each face, we compute the mean distance of its vertex, and choose a colour according to this distance.

However, this method of representation doesn’t really reflect the work we done. Even if the distance field we represent is accurate, we can display much more information without any new distance field computation. For that, we subdivide each face using a variant of loop-subdivision, where no vertex are moved and the new edge vertex lies exactly at the centre of each edge. This allows us to keep the same overall geometry while multiplying the number of faces by four.

We can now colour each new face with the same method. As we can compute the distance field for each vertex, and not only the initial vertex, we are able to represent the distance field with an arbitrary degree of precision.

## Results

### Efficiency

### Accuracy

The limitations of our method mainly come from rounding errors. As we propagate windows, errors that appear when computing square roots and divisions pile up and decrease the precision of our algorithm. We identified two main issues:

First of all, there is error in the distance field of each window. Small mistakes quickly add up when propagating and result in windows with incorrect distance field. However, those errors are not extremely problematic when representing the results, because they are continuous with respect to the real distance field, and when representing with colour on each face, the differences are minor.

Then, some windows are incorrectly placed. Rounding errors when defining the starting and ending point of each window can result in bigger errors when propagating along a face. This leads to edges that are covered by windows that are not optimal, due to the fact that the optimal window propagation was stopped due to a rounding error. This kind of errors explains the red bleeding we can observe on figure with a lot of edges. Some edges also have gaps or overlaps in their window coverage. This might explain the bluer shapes that sometimes appear when subdividing the polygon.

## Extensions

### Bezier Curves

One of the application for our algorithm would be the representation of Bezier Curves and Splines on a 3D Mesh. The definition of Bezier Curves and Splines only rely on distance computation. And we could efficiently compute the geodesic distance between two given points using our algorithm and stopping before the complete distance field has been computed. Using this method, we could draw curves on 3D shapes while retaining the control simplicity and properties of Bezier curves.

### Find the Source!

The little game we presented in the introduction could be expanded to a full-fledged video game. The computation of the real distance field is quite fast for simple meshes, and we can efficiently achieve a smooth gradient for the colours by subdividing our initial mesh. By allowing the player to walk on the mesh instead of “flying” above, the game would be a little bit harder.

### Procedural texturing

We could use the algorithm we implemented for procedural texturing for computer graphics, video game and else. For example, if the mesh represents a part of the world, and the source is a water point in the desert, the concentration of vegetation is a decreasing function of the distance to the water point. We could use the distance field we computed to apply different textures depending on how far we are from a certain points, and dynamically generate environnments.