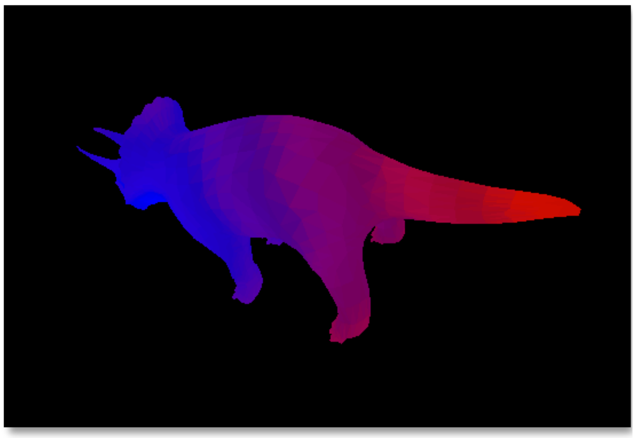
Fast and exact geodesic distance on triangular mesh

Adrien Matricon & Jean Caillé

I. Links

You can find the entire code on github : http://github.com/jcaille/INF555_Project

Larger versions of the images, as well as examples are available at http://bit.ly/YcUY4g



The best way to test the algorithm is to clone the src directory, choose the OFF file on which to compute the distance field in MeshViewer and run. Press ‘s’ to then subdivide the mesh. Blue is closer to source and red is farther. However, the colour function is not linear and small variations are introduced in order to better perceive isolines.

II. Motivation

Given a 3D mesh, one could want to be able to compute distance on the mesh. One simple application for this algorithm would be to colour the mesh according to the distance to a certain point.

A. Existing methods and limitations

We could easily approximate the distance between two points by limiting ourselves to following its edges, using Dijkstra algorithm. The shortest path between two given points would then mainly consist of a walk along the edges of the mesh. This approximation might hold for graphs that are sufficiently tight and regular, but the error on the results obtained with this method can be arbitrarily high if the mesh or the points are not well chosen.

B. Exact computation of geodesic distance

Even if Dijkstra’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 Dijkstra-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.

III. Our Implementation

A. 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 providing a nice way to visualize the information we computed. 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.

B. 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 go through the graph.

The only extensive custom data structure we're implementing is 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.

C. Propagation

Given a window, we first need to be able to propagate it along the neighbouring edges. 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 (B_0, B_1) on an edge h, we call P_0 and P_1 the extremities of h, P_2 the last vertex of the face and S the source associated to the window. We can then identify 4 noticeable points at the intersection between 4 pairs on lines: M_0(S-B_0, P_0-P_2), M_1(S-B_1, P_0-P_2), M_2 (S-B_0, P_1-P_2) and M_3(S-B_1, P_1-P_2).

We then define one of those intersection points as *valid* if it lies on one on the triangle edge. Knowing which point is valid and which is not allows us to distinguish between the various cases we have to consider (except case 0, which has to be tested first because otherwise considering intersection points makes no sense). You may want to visualize what the different cases actually look like. Drawings representing each case are available here (in the "Cases" folder): http://bit.ly/YcUY4g.

1. Case 0

If the source is on the edge, we add two new windows, each covering entirely one of the neighbouring edges.

2. Case 1 & 2 (only M0, M3 and either M1 or M2 valid)

If an extremity P of a window is also an extremity of the edge, then it can be a saddle point: after normally adding a new window on the opposite edge, we create new windows covering the rest of the neighbouring edges, with P as a pseudo-source.

3. Case 3 (only M0 and M3 valid)

Both neighbouring edges can be reached from the source through the window. We create new windows corresponding to the scopes on those edges that were reached.

4. Case 4 & 5 (either only M0 and M1 valid or only M2 and M3 valid)

Only one of the two neighbouring edge can be reached from the source through the window. We create one new window corresponding to the scope on this edge that was reached.

D. Merging

As we propagate windows, we have to make sure that windows don't overlap on a same edge, and also that we keep the distinction between windows already propagated and windows yet to be propagated.

The merge function is called to add a list of windows to a given edge. This list usually contains only one window, but can also contain two in cases 1 or 2 (see above).

We successively consider each "old window" (windows that were already on the edge before the merge) and keep track of two lists: one containing the new windows (*new list*) and one containing the old window currently considered (*old list*). The use of lists is justified by the fact that a window can be divided into more windows during the merging process, which can lead to many windows in both lists at the end of the merging process.

For each window in the *new list*, we browse the *old list* in search of an overlap. When an overlap is found, we reduce the sides of the overlapping windows and/or divide them in the corresponding lists, in a way that each point of the edge that was covered by the two windows only stays covered by the window minimizing the distance from the source to this point through it (if the distance is the same, we give precedence to the old window to avoid infinite loops during the propagation).

After we finished browsing, we can replace each old window by the corresponding *old list* (in the list of windows on the edge and if necessary in the heap of windows yet to be propagated). After each old window has been considered, windows in the *new list* don't overlap anymore with those already on the edge, and we can add them to the list of windows on the edge and to the heap of windows that need to be propagated.

E. Shortest distance from the source to an arbitrary point

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

1. 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 two containing P (one on each halfedge), find the distance to P on these windows and return the lowest one.

2. P is a vertex of the mesh

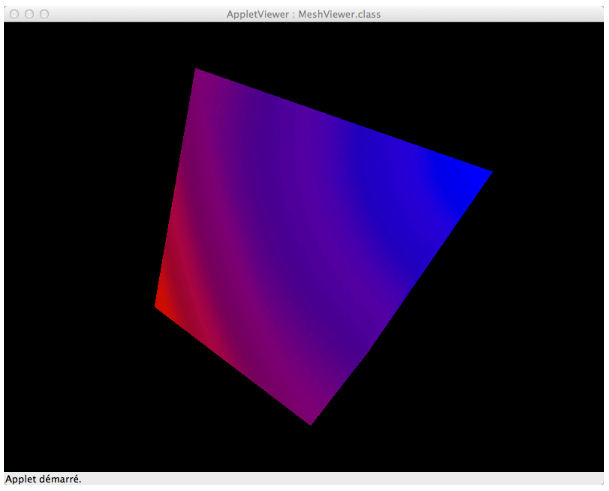
If P is a vertex, we compute the distance by examining all halfedges containing P. For each one of those edges, one window will contain P as its extremity with an associated distance to the source. We simply choose the minimum of those distances as the correct value for P.

3. P lies inside a face

If P lies inside a face of the mesh, we go through all the windows that cover the halfedges of this face. For each window, we compute the minimum distance from the source to P through this window in a single step with a formula derived from the Pythagorean theorem. We only need to go through each window once to minimize the distance from the source to P.

F. 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 field. For each face, we compute the mean distance from its vertices to the source, and choose a colour accordingly.



However, this method of representation doesn't really reflect the work we have done. Even if the distance field we represent is accurate, we can display much more information without any new distance field computation. A way to show that is to subdivide each face using a variant of loop-subdivision, in which no vertex is 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.

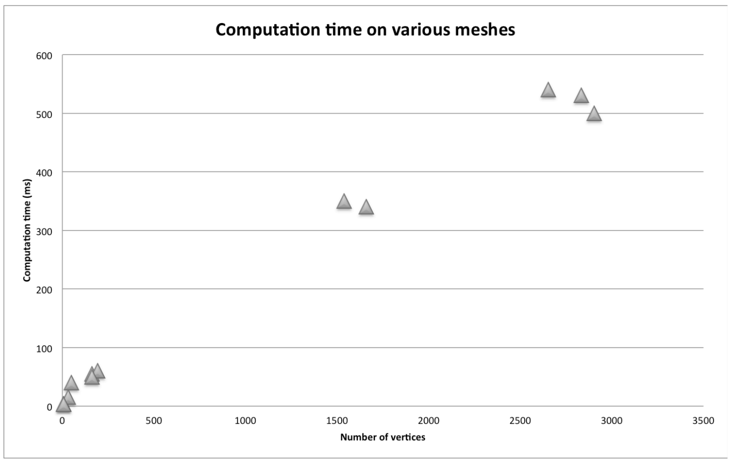
We can now colour each new face with the same method. As we can compute the distance field for each vertex, and not only the initial vertex, we are able to represent the distance field with an arbitrary degree of precision.

IV. Results

A. Efficiency

Overall, our implementation is pretty efficient, but this is partially due to the available computing power, as we didn't really focus on complexity. We did not study the complexity of our algorithm formally, however, different tests have shown a complexity approximately linear with respect to the number of vertices of the graph.

It is interesting to notice that once we have computed the distance field, we can subdivide our polygon without having to compute additional distance information.



B. Accuracy

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

First of all, there are errors in the distance field of each window. Small mistakes quickly add up when propagating and result in windows with incorrect distance fields. 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 in the definition of the extremities 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, because the propagation of the optimal was stopped due to rounding errors. This kind of errors explains the red bleeding we can observe on the figures that have many 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.

V. Extensions

A. Bezier Curves

One of the applications 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.

B. 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.

C. 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 environments.