X2012
29/11/2014

ÉCOLE POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

Exact Geodesic Algorithm Implementation

INF555 : Digital Representations and Analysis of Shapes

Gurvan L'Hostis

gurvan.lhostis@polytechnique.edu

Abstract: This project aims at implementing an algorithm for exact geodesic computing on triangular meshes. This algorithm has a first phase of distance field propagation from a source vertex that is done thanks to a simple window data structure and the idea of light rays emanating from this source. The second part of the algorithm consists in computing the actual path from the second vertex back to the source thanks to the information that was propagated across the mesh.
I present the implementation, its results on simple meshes, its difficulties in handling the floating point precision and possible extensions.

Keywords: Triangular mesh • Geodesics • Fast-marching

1. Formal description of the algorithm

The algorithm that is studied was formally conceived in 1987 [1] but the first practical implementation of this algorithm was only described by [2] in 2005. More precisely, what we seek to reproduce is only a part of their implementation: the exact algorithm for propagating the distance field from a source vertex, and backtrack the geodesic to that source from any other vertex on a triangular mesh. We do not try to code the approximate version that is presented in the paper.

1.1. Propagating the distance field

The first part of the algorithm consists in propagating the distance field across the mesh. We have to know the geodesic distance so as to be able later to compute the path to the source from any point of the edges. This is done with a simple **window** data structure. We see the origin vertex as a light source that illuminates the edges of the mesh. More specifically, we use the fact that geodesic paths are straight lines when we unfold the triangles they go through, thus the edges that are closer to the source behave like windows that define where the light can go in the edges that are further away from the source (see figure 1 (a))
Such a window on an edge consists in a simple 6-tuple $(b_0, b_1, d_0, d_1, \sigma, \tau)$. The first two coordinates are the bounds of the window on the edge, the two that follow are the distances to the source of each of these bounds,

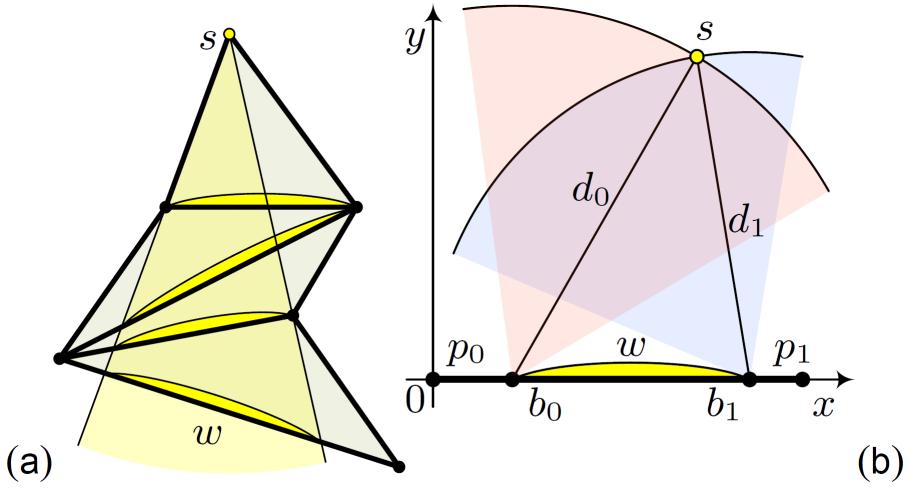

Figure 1. (a) Diagram illustrating how the origin vertex acts a light source through the edges. (b) How to compute source coordinates with window data. *(source: [2])*

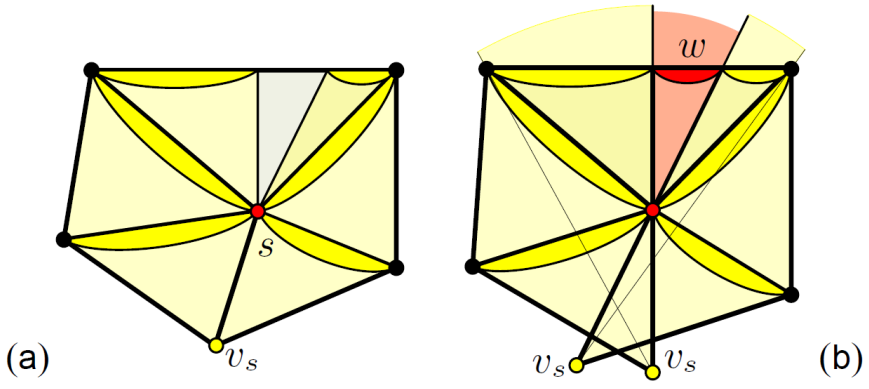

Figure 2. (a) Orthogonal projection of faces surrounding a saddle point. The grey area is not directly illuminated by the source v_s. (b) Two-way unfolding of the faces starting from the upper face. The shortest paths to the red area pass through s which acts as a pseudosource. *(source: [2])*

which allow to compute the position of the source (see 1 (b)) and τ is a boolean that indicates on which side of the edge the source is.

The last coordinate σ is the distance of the source indicated by the window, or *pseudosource*, to the real source. The need for pseudosources stems from the existence of saddle points[1] in meshes; a simple light propagation from the main source does not illuminates all of it, as can be seen in figure 2. When we are confronted to a saddle point, the shortest paths to the areas that are not illuminated by the main source go through edges or vertices of the mesh, and because these cases are not taken into account by the light ray propagation idea, we need to

[1] *A saddle point in a mesh can be defined as a vertex around which the angles of the adjacent faces sum above* 2π.

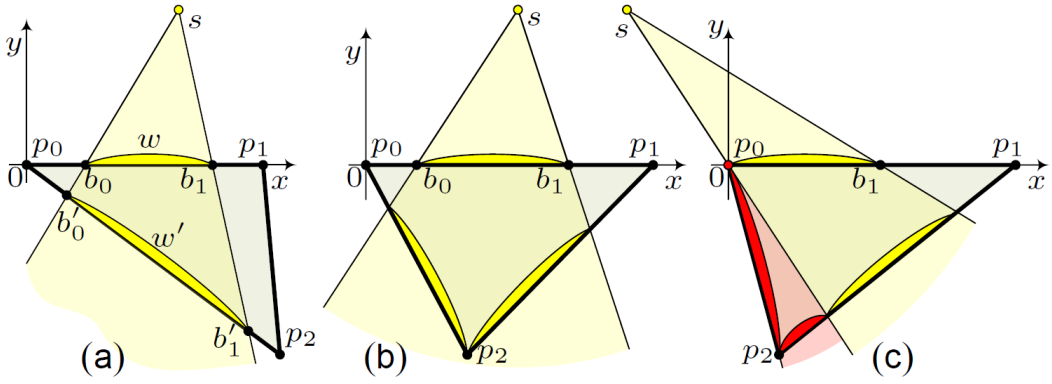

Figure 3. Different propagation cases from window w. (a) w propagates fully in one edge. (b) w propagates in two edges. (c) The presence of a saddle point in p_0 requires creating two pseudo-windows in addition to the classical window. *(source: [2])*

consider pseudosources.

We will see in the backtracking part that such windows contain all the information that is needed to compute a geodesic path back to the source.

Windows are gradually propagated from edge to edge as shown in figure 3 similarly to the Dijkstra algorithm on a graph. We maintain a priority queue of unpropagated windows that tells what is the next window to propagate. We order it by minimizing the distance to the main source.

When we add a new window to an edge, it may already contain windows that overlap with it. We want to keep only the windows that track the shortest path to the source, and this indicates how to cut or remove the windows and address the overlapping issues (the process is detailed in the implementation section).

1.2. Backtracking

The second part of the algorithm consists in finding the shortest path back to the source thanks to the information that was propagated throughout the mesh.

For that, we jump from window to window by computing which of the two other edges of the triangle intersects with the segment formed by the current point and the source. When we arrive to a pseudosource we continue to navigate from an adjacent window until finding the main source.

2. Implementation

2.1. Class architecture

The class architecture of my implementation relies on the JCG library architecture (see figure 4). The main class of the implementation, which describes the distance field propagation algorithm, is coded in `ExactAlgorithm`. This is where the window queue is held, as well as pointers to the two points of the geodesics we compute.

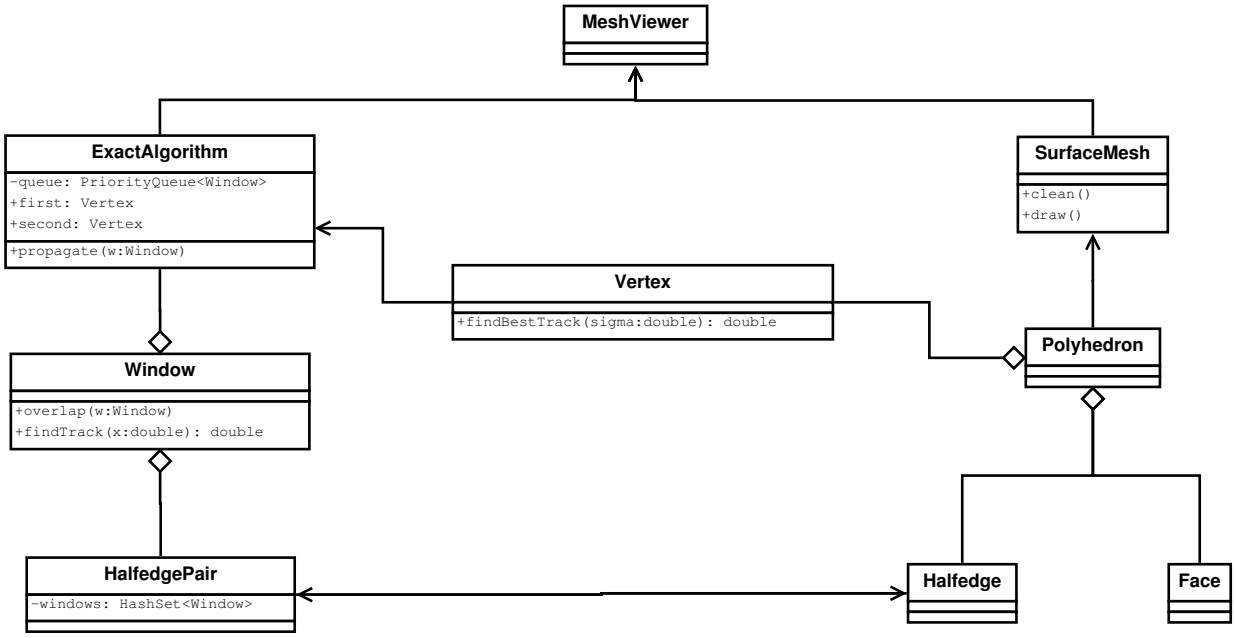

Figure 4. **Class diagramm of the implementation.**

The class `HalfedgePair`, which is used in JCG for mesh loading, is developed as a structure to represent the edges; it is more convenient to store the windows here than in the halfedges. The class `Window` implements the 6-tuple described in the previous section. It also contains the functions that deal with overlapping issues.

The backtracking part, which is simpler, is implemented in two classes. The iterative hops from window to window are realized with a recursive function in the class `Window`, and the other important function of finding the right track from a vertex is implemented in `Vertex`.

Utility functions are implemented in `SurfaceMesh` and `Face` for managing the geodesics data and graphics, user-interaction is implemented in `MeshViewer` and `ArcBall`. Also, the loop subdivision classes from TD5 are integrated to this architecture for testing reasons.

2.2. Window propagation and overlapping

The propagation functions are quite straightforward. The main difficulty consists in successfully breaking down the problem in sub-cases depending on point orientations. A special case I had not taken care of in the very beginning is when one of the vertices of the triangle is the source, in which case the orientation functions returns unsure results. Also, the propagation of windows through a new pseudosource needs careful thinking while implementing.

What is critical in the algorithm and that is not deeply described in [2] is window overlapping. When we add a new window to an edge (or halfedge pair), we check for overlap with every previously existing window.

First of all, window orientation must be properly taken care of while making computations, which often becomes

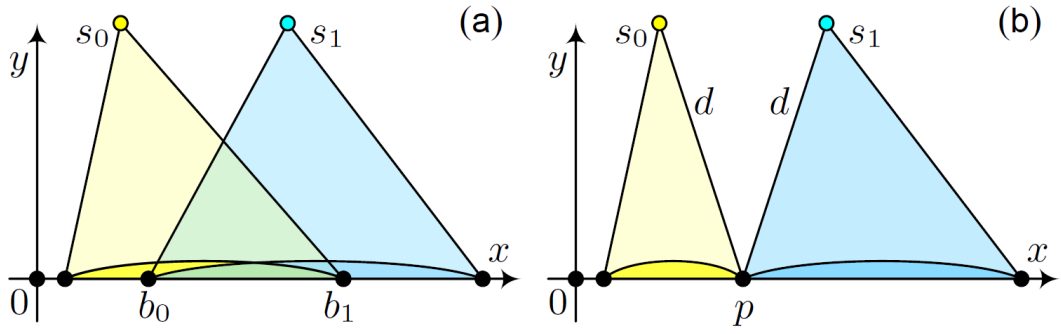

Figure 5. Diagram illustrating how the overlap between two windows is dealt with in the special case where $\sigma_0 = \sigma_1$. *(source: [2])*

a puzzling problem.

Also, successfully finding out which window is the best requires a lot of calculus, which leads to a lot of possible bugs. In order to avoid making duplicate computations and to have a synthetic organization of data, an inside class `OverlapData` is defined in `Window`, containing information such as important point coordinates and distances. There are five main tasks to take care of in the overlapping process:

- compute the bounds of the region in which the two windows overlap;
- compute the distance to the source of these bounds;
- decide whether there is a full cut or not (i.e. whether there is a window that is better than the other on the full overlapping region);
- in the case where the cut is partial, compute the separation point of the two windows (i.e. the point that is equidistant to both sources);
- effectively reformat the windows.

Whilst the first two tasks are basic, the third one requires some mathematical reasoning. We use the following lemma to discriminate the three possible cases that are partial cut, full cut from one window or the other.

Lemma 2.1.
Given an overlapping region $[\delta_0, \delta_1]$ and distances to the sources of windows a and b that we call d_0^a, d_1^a, d_0^b and d_1^b, we have

$$d_0^a < d_0^b \text{ and } d_1^a < d_1^b \iff a \text{ is better on the full interval}$$

The fourth task is dealt with by solving the quadratic problem as described in the paper (see 5).

The last task is more complex than it seems: there are fourteen possible cases depending on whether the overlapping region fully covers one of the edges, whether there is a fullcut, whether windows are cut in two separate

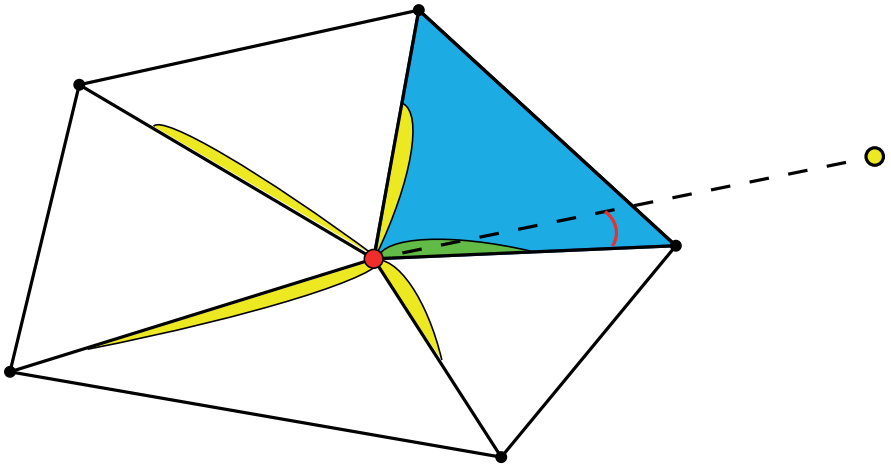

Figure 6. Illustration of the best window search. The source is in yellow, the vertex from which we do the search is in red. The best window to pursue with is the green one because it minimizes the angle represented in red.

windows. The contingency of a window that is cut in two separate valid windows, which not even hinted in the paper, is something that my early implementation did not take care of and that was a source of bugs.

When windows are created during overlapping computations, we put them in a special queue to be added like a new window and we repeat the process with them afterwards. When windows must be deleted, they are removed from the set of windows after the traversal and they are marked as invalid so that they will not be propagated in later iterations without having to remove them from the priority queue which would take linear time in size of the queue.

At last, if a window survives this overlapping deletion process, it is added to the set of the edge and to the priority queue.

2.3. Backtracking

The backtracking process is done with two main functions.

The first called function is used to start a geodesic path from a vertex. It is called on the geodesic end but also when pseudosources are encountered.

This method traverses all the adjacent edges and retrieves their windows that lie next to the vertex. All these windows give the same geodesic distance and path, what we seek is the one that advances to the next point, which is the one for which the source is the corresponding face's angular sector (see figure 6). When the right window is found, a call to the other function is made.

The other function is the one that is used to perform the hops between windows. It is a window method which takes a coordinate on the edge as a parameter. This method computes the positions of the current point, the source and the opposite vertex on the triangle so as to know what is the next edge to hop on. Once the edge is identified, we compute its intersection with the source-current point segment so as to have the next position. The

coordinate of the point on the next edge is then computed, which allows the method to retrieve the corresponding window from the halfedge pair and ultimately to call itself on the next window.
There is a special case for when the source is found to be one of the edge ends: we call the first function, and we continue the process.

2.4. Float-point number comparison

The biggest challenge of such an implementation, as I realized while debugging, is handling the rounding errors. At first, I naively considered that ϵ comparisons applied only for equality comparisons. I soon realized that rounding errors had to be taken care of for every comparison, but the trick was to determine if the rounding errors were "inclusive" or "exclusive".
For instance, when checking whether two windows overlap, we don't want to consider the case when the overlap is smaller than ϵ. This is what I call an exclusive comparison.

```
!(b1 <= w.b0 - ExactAlgorithm.epsilon || w.b1 <= b0 - ExactAlgorithm.epsilon)
```

On the other hand, during backtracking, let us consider the time when we compute the coordinate of the next point on the next edge. If the point is close to one the bounds of the edge, we might end up with a coordinate that is < 0 or $> edge.length$, and this is why we need to do an inclusive comparison.

```
(x >= b0 - ExactAlgorithm.epsilon && x <= b1 + ExactAlgorithm.epsilon)
```

For case separations such as during partial cut when we need to know which window does better on the left side of the separation for example, considering rounding errors makes no sense because of the symmetry.

```
data.wD0 < data.oD0
```

I used an error bound of $\epsilon = 10^{-8}$, which I chose experimentally: I diminished it until my code assertions did not hold.

2.5. Assertions

I used Java assertions all over my code. Not only are they easier to use than exception raising, they allow efficient debugging by having a breakpoint functionality in the Eclipse IDE.
What I found especially interesting with the assertion mechanism is that we can choose to enable it or not. At the beginning, I did not like having to put `if () raise new Exception()` tests everywhere because it made my implementation slower, thus I did not verify parameters assumptions enough for example. Thanks to assertions, there is no reason for being shy in testing.

2.6. Displaying and checking

I will now make a summary of the GUI additions I implemented.

There are two modes for propagating the distance field: either window by window or all in one time. The vertices can be chosen randomly or clicked on.

The faces are originally red; they become green when they are traversed by the distance field propagation. I removed the JCG light effects so as to be able to put information in shadows; the darker a face is, the more windows its edges contain.

The two vertices between which we want to compute the geodesic are colored in yellow whereas the other vertices are blue.

The faces that are traversed by the geodesic become blue and the exact path is drawn in magenta.

So as to gather information on how the distance field is propagating, I implemented a function that traverses all the edges and tests whether there are pseudosource windows (this should not happen on the convex region surrounding the source) or holes (an interval with no window). Pseudosource edges are colored in grey; porous edges are colored in cyan.

3. Results

3.1. Distance field propagation

The propagation part of the algorithm is definitely the hardest part. The floating point precision issues of the computations make my implementation faulty beyond approximately one thousand vertices.

Convex meshes

The first mesh that I worked with, because it is convex and thus does not involves tricky saddle points, is a sphere of 162 vertices.

Later on, thanks to the loop subdivision scheme implementation from practical sessions, I could evaluate the performances of the implementation on bigger meshes. The sphere with 642 vertices still gives good results but in the next iteration of 2562 vertices we first encounter holes (see cyan edges on figure 7).

General mesh

The mesh I mainly used for tests with saddle points is the torus. Under five hundred vertices, there are absolutely no holes on the mesh and the pseudo-source windows are where they are supposed to be.

Still, problems begin on the 528 vertices subdivision iteration with holes appearing (see 8).

In order to try a more complex mesh with few vertices, I used the "a" letter mesh which contains 276 vertices. The distance field propagation was successful on it.

The more complex meshes I tried such as the triceratops or the high genus mesh, which respectively contain 2832 and 1660 vertices, did not give perfect results with the propagation and holes remained wherever the source was

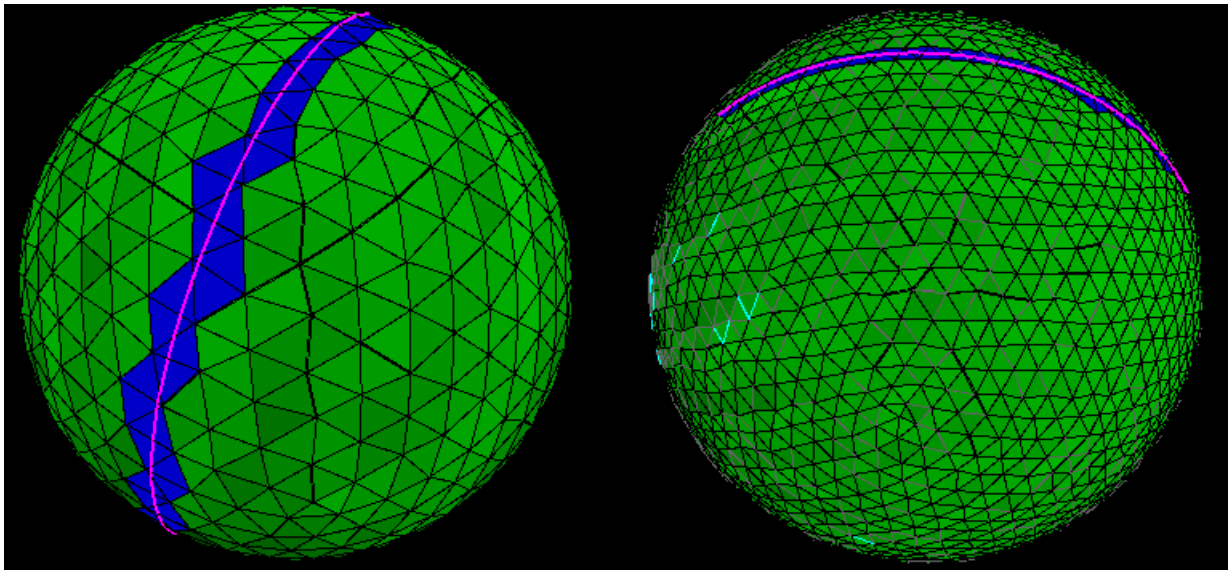

Figure 7. Images of implementation results on two sphere meshes. The simpler mesh on the left led to a propagation with no holes or pseudo-windows whereas the bigger mesh on the right has degenerated with abnormal pseudo-windows (in grey) and porous edges (in cyan).

placed (see 10).

3.2. Backtracking

As regards convex geodesics, one can be convinced of the implementation's goodness simply by looking at the path from above the faces it goes through: it always form a straight line, even on complex meshes for which the propagation algorithm leaves holes.

Also, computing the geodesic distance between one point of the sphere and its opposite returns 3.11, which seems satisfying.

It is less easy to be convinced by geodesics that go through saddle points, especially on small meshes: what does looking from above mean when the geodesic runs over an edge?

Still, the results on meshes that are not too edgy, such as a subdivided torus mesh, appear to be good.

Last, we should remark than even though there are holes on the triceratops mesh propagation, we can compute a lot of geodesics and observe them (figure 12). They look very straight according to their unfolding.

4. Extensions

4.1. Optimizations

Window set datastructure

The main optimization that can be done is to switch from hashsets to interval trees to store the windows on an edge. This would allow log time window retrieving instead of current linear time for both window overlapping

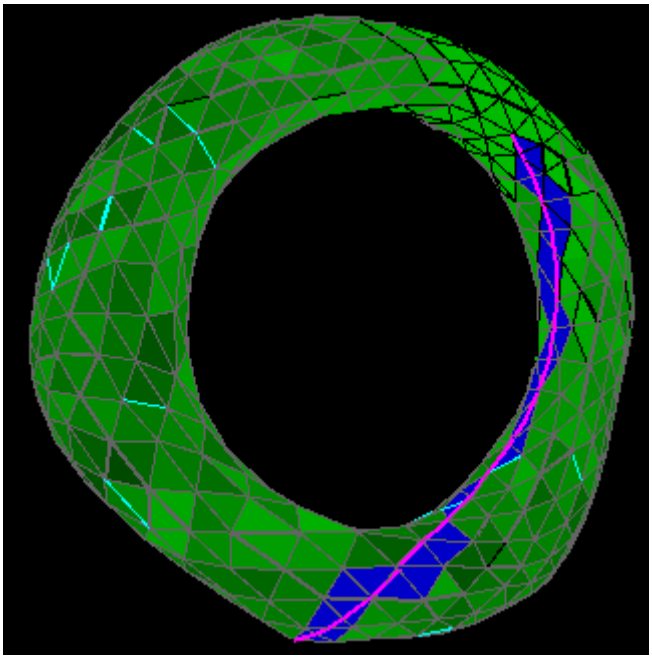

Figure 8. Image of the implementation result on a torus. We may observe that the convex area surrounding the source (upper end of the geodesic), that this area is surrounded by a larger area with regular pseudo-windows (grey edges) and that propagation has failed on parts of the lower half (cyan edges).

and backtracking. This optimization would be especially important to deal with larger meshes since the number of windows per edge is $O(n^{\frac{1}{2}})$.

Relative error bounding

Currently, the floating point rounding error is managed only through absolute error standards. A little research on this subject indicates that it is generally important to have both criteria: relative and absolute. Such improvements may very well improve the propagation performances and diminish the number of holes.

4.2. Other features

Smarter propagation data structure

There certainly are more evolved ways of dealing with rounding errors, and it would be interesting to see how the paper implementation handles them.

Approximation algorithm

The approximation algorithm described in the paper is an efficient tool to accelerate geodesic computations between two specific points as it bounds the region in which the exact algorithm should work. After having worked out precision issues, this would certainly be the most interesting feature to implement.

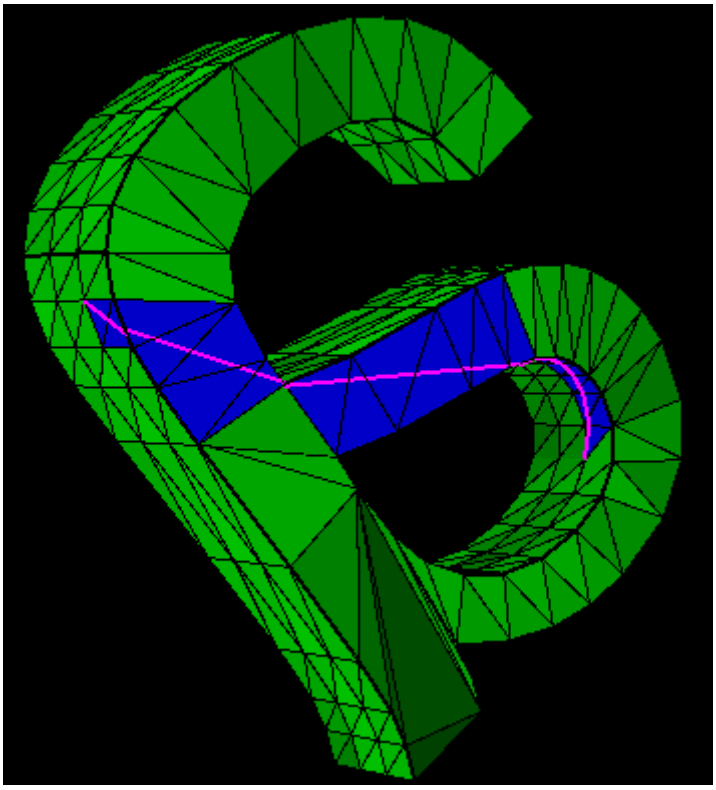

Figure 9. Image of the implementation result on a "a" letter mesh. The geodesic path passes through two saddle vertices (where the magenta line is broken). (Edge windows are not tested.)

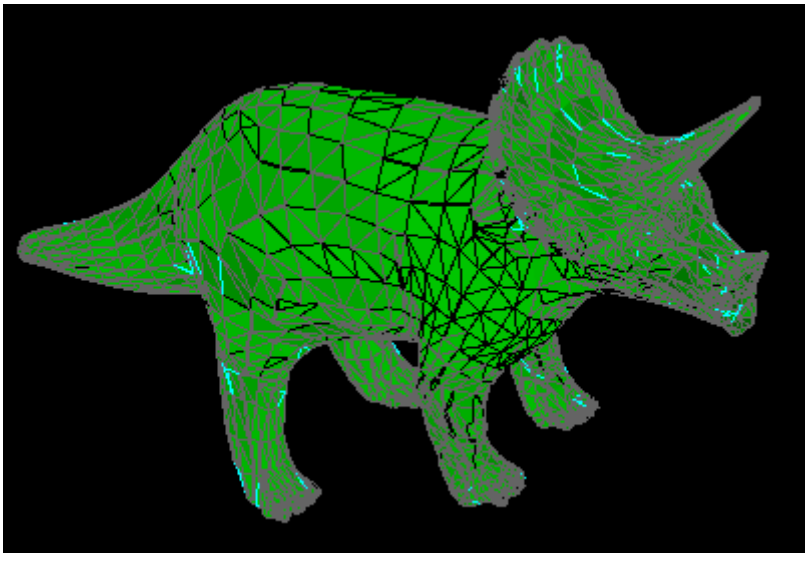

Figure 10. Image of the implementation result on a triceratops mesh with a source on the right shoulder. We may see that there are numerous holes on the edges.

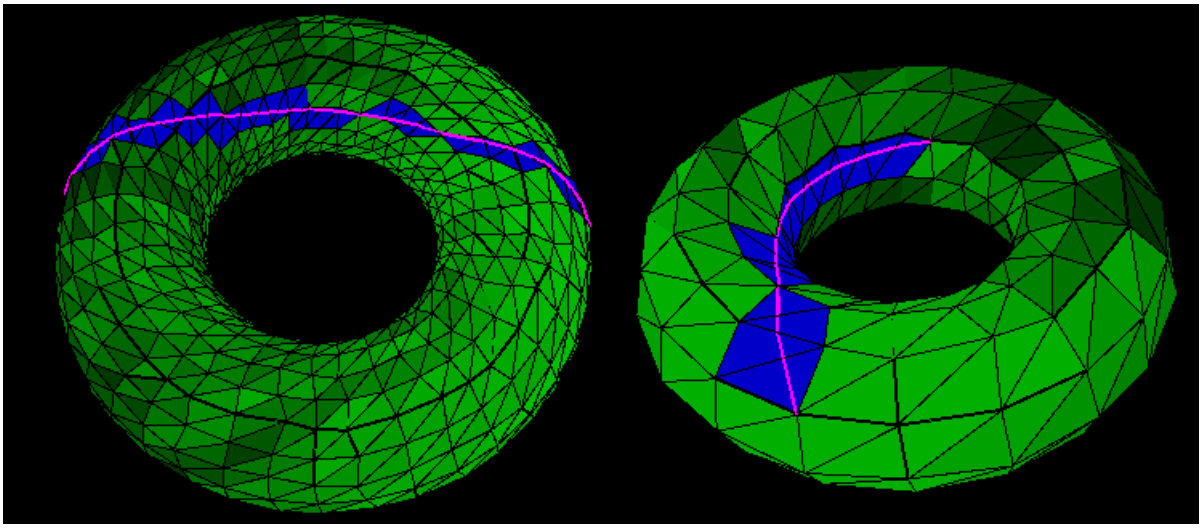

Figure 11. Image of implementation results on two tore meshes. We see on these meshes containing saddle points that the geodesic path is a straight line on the triangle unfolding. (Edge windows are not tested.)

References

[1] Mitchell, Joseph S. B. and Mount, David M. and Papadimitriou, Christos H., *The Discrete Geodesic Problem*, SIAM J. Comput., Aug. 1987

[2] Surazhsky, Vitaly and Surazhsky, Tatiana and Kirsanov, Danil and Gortler, Steven J. and Hoppe, Hugues, *Fast Exact and Approximate Geodesics on Meshes*, ACM SIGGRAPH 2005 Papers, 2005

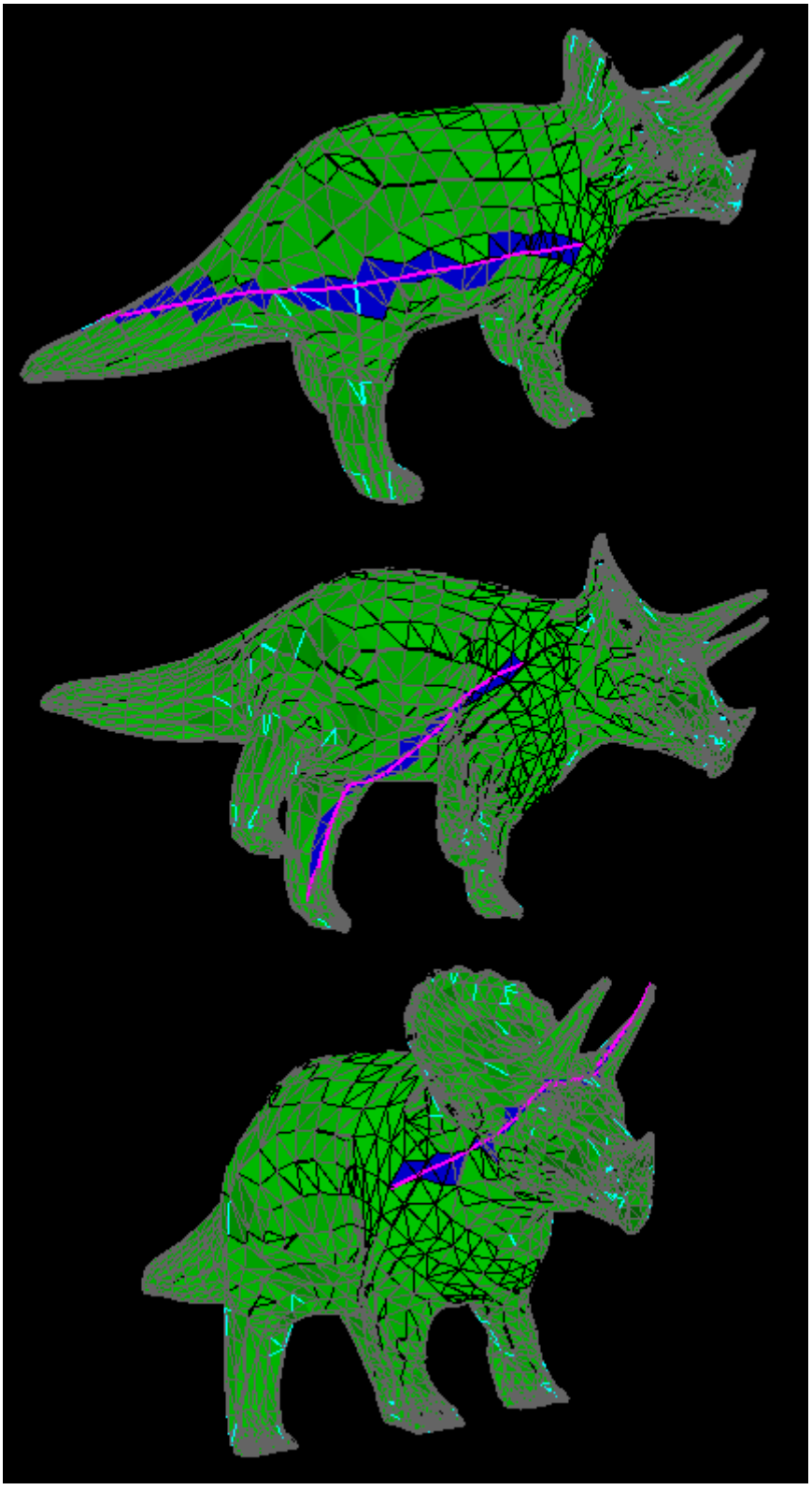

Figure 12. Geodesics over the triceratops mesh starting from the right shoulder and going respectively to the end of the tail, the rear left paw and the left horn.