Implémentation

Environment

We chose Unity to implement the heat method in order to better visualize the result. All codes are written in C#, and the scene is built with Unity Editor. For the main problem, we use the library ALGLIB to do sparse matrix operations and to solve linear equations. In addition, we use the C5 Generic Collection Library for the priority queue implementation.

Mesh representation

We obtain ordinary triangle meshes via various ways. Those meshes are represented by a vertex array (array of Vector3) and a triangle array (sets of 3 indexes stored in an int array). We first wrote a method to convert them to half-edge representation, defined in Geometry.cs. The conversion can be done in time of O(nd2) where n is the number of vertices and d is the maximum degree of vertices. There are however 2 important things to consider:

1. The half-edge representation is not well defined when using meshes with boundaries. In order to incorporate with other methods built on this representation, we decided to add a new face to cap each boundary. Those faces are marked as “boundary faces”, and all vertices around them are marked as “boundary vertices”. This way we can easily implement the boundary conditions in the heat method.
2. Many 3D models obtained from the internet have UV mappings, and thus have UV seams. This means that at the same position there can be 2 separate points having different UV coordinates. So the geometry we built may have seam-like boundaries blocking the way. To cope with this, we implemented a method to weld all overlapping vertices with a complexity of O(nlogn), based on kdTree range searching.

Matrix precalculation

For each mesh loaded, we calculate in the first place its discrete unweighted laplacian matrix -Lc, and the matrices A-tLc adapted to 2 different boundary conditions (if there are boundaries).

Dirichlet condition – We set all elements in the rows/columns of the boundary vertices to 0, except the diagonal elements. This way the heat value will always be 0 at the boundary.

Neumann condition – The original laplacian matrix described in the paper satisfies Neumann condition.

We also build an Vector3 array of size (3 \* triangle count) keeping all the values of cot(angle) \* opposite edge vector, in order to accelerate the calculation of divergence.

For the first time, we skipped the Cholesky decomposition step since the overall performance without it is still reasonable. However, because of the numerical problems that we will explain afterwards, we finally implemented the Cholesky decomposition. It is applied on all precalculated matrices.

Main calculation

For single source problem, we follow these steps:

1. Calculate the heat flow u by solving the heat equation (A-tLc)u = delta(source).
2. Calculate X, the normalized gradient of the heat flow, on every triangle.
3. Calculate DivX on every vertex using the value of X on its surrounding triangles.
4. Calculate the distance field Phi by solving the Poisson equation LcPhi = DivX
5. We then calculate the gradient of the distance field gradPhi on every triangle which can be used to calculate the shortest paths

At first, without Cholesky decomposition, we used LinCG (Linear Conjugate Gradient) solver to solve linear equations. This solver solves symmetric positive definite problems, but it works fine even with our second semi-definite problem (Poisson equation). We just need to shift the result such that the distance value at the source is equal to zero.

However, from the result we obtained, we observed that LinCG solver is sensible to numerical errors. The solution of the heat equation contains values ranging from 1 to 10^-20 or smaller. The longer the distance to the source, the smaller the heat value is. LinCG returns values of 0 when smaller than ~10^-12, so the heat gradient of the farther area cannot be computed. We can only improve the solution by increasing the time step t, which increases the heat value but creates a smoothed distance field.

Therefore, we chose to implement the Cholesky decomposition of matrices. It consists of decomposing a symmetric positive matrix M into LL^T, where L is a lower triangular matrix. This means the linear equation LL^Tx = y becomes two basic triangular systems, that can be solved by simple substitution. This improves greatly the calculation time when switching sources (The decomposition is only calculated once for every mesh). And since we calculate the exact solution this way, we managed to have much smaller numerical errors, so as to eliminate the problems above.

We overcame 2 difficulties regarding the Cholesky decomposition:

* While M is a sparse matrix, L isn’t necessarily sparse, especially when the entries of M are mostly far away from the diagonal. This comes to one of our model which has randomly ordered vertices – it costs more than 4G RAM space to store the matrix L. After we reorder the vertices of the mesh along a certain axis, L becomes much sparser and takes much less time to compute.
* We have to add a small regularization term to the diagonal entries of the laplacian matrix to get strict positive-definiteness (needed for the Cholesky decomposition of ALGLIB).

Result

Additional work

Multisource

We also tried calculating the geodesics on surfaces when multiple vertices are marked as sources. However, by simply changing the initial heat vector to delta(sources) we generates incorrect heat field and distance field – the solved heat field ut cannot guarantee equal values at each source vertex, and thus not every source vertex has a distance of zero. Typically, the vertex surrounded by more source vertices has a higher temperature and a negative distance.

[tuuuuuuuuuuuuuuuuuuuuuuuu]

We solved this by imposing the constraints ut = 1 at every source vertex in the heat equation, and also the constraint phi = 0 at every source vertex in the Poisson equation. (The first constraint cannot imply the second since the distance field we get is only the closest potential of the heat gradient field)

[tuuuuuuuuuuuuuuuuuuuuuuu]

One downside, however, is that the matrices need to be modified to contain these extra constraints related to the source. For exemple, to have ut = 1 at the source we put 1 at the diagonal entries of the sources, and 0 everywhere else in the same lines and columns. We then compensate the terms we deleted by adding an additional vector at the right side of the equation. Because of this, the Cholesky decomposition need to be carried out each time we change the source. This slows down the calculation a lot.

Navigation

Now that we have the distance field, we would like to calculate the path from a given point to the source point. We achieved this by first calculating the gradient of the distance field (step 5 of the main calculation) which is considered uniform on every triangle, then trace a trajectory by recursively following the gradient in every triangle and entering the next one.

We visualized this process by moving a walking man on the surface of the mesh. His position is defined by barycentric coordinates of the triangle he is standing on. The distance gradient on it is also converted to barycentric coordinates, but with a sum of 0 (In this way, we can add it to the position coordinates and still get a barycentric coordinate with a sum of 1). A walk function takes a distance as its argument and first moves on the triangle on which he is standing. By finding the first coordinate that reaches zero we find the edge the man will come across. It then recursively calls itself at the next triangle he enters, until walking the given distance or reaching the source.

There are however 2 extra cases to consider:

* The gradient is pointing to a boundary. In this case, when reaching a boundary, we walk alongside it towards the smaller-distance vertex.
* 2 neighboring triangles have opposing gradient. In this case, when reaching the middle edge, we chose to follow the edge instead.

Mapping

In order to better visualize the distance field, we chose to map a striped texture onto the surface of the mesh. It is actually quite simple: We create some striped textures with color gradient that are only one pixel tall, assign it to the mesh, and for each vertex we set the U value of its UV coordinates to its distance to the source. And that’s it! When rendering the texture, each point on a triangle has its U coordinate interpolated from the 3 vertices of the triangle, and it is exactly the interpolated distance of this point – so it will be colored by the pixel of the striped texture representing this distance.

To make it even better, in addition to the main texture, we also apply generated normal map, specular map and emission map to each model, making it especially realistic. We just need to set the tangent vector of each vertex to the average distance gradient on it to make the normal map work.

Dijkstra

We implemented a simple Dijkstra shortest path algorithm on the graph made by the vertices and edges of the mesh. This allows us to easily create a line of source vertices connecting 2 selected vertices (In the demo, press shift and click on the surface of the mesh).