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 laplacien 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 laplacien 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 added a small regularization term to the diagonal entries of the laplacien matrix to get strict positive-definiteness (needed for ALGLIB).

Result