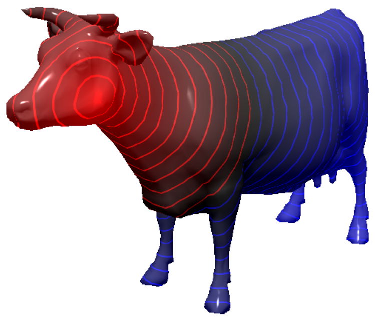
COMP 557 - Fall 2022 - Assignment 3
Mesh Processing, GLSL, Heat Diffusion and Geodesics

Getting Started



In this assignment you will load meshes and run geometry processing algorithms to compute geodesic distances on the mesh manifold (a geodesic is the shortest distance on the manifold between two points). You will also use GLSL shaders for per fragment lighting, and to visualize the geodesic distances with stripes.

While the assignment is based on an ACM Transactions on Graphics article by Crane et al., Geodesics in Heat: A New Approach to Computing Distance Based on Heat Flow, you do not need to read or understand the paper in detail to complete the assignment.

Provided Code

Download the sample code from MyCourses. It is a working program that will load a polygon soup. Note that in class, the mesh data structures material put some emphasis on memory usage. However, in this assignment you will note that we will store some extra information in various half-edge data structure classes for various reasons.

The provided code contains a number of files and classes:

- main - Contains the main function and creates the view.
- HalfEdge - Half edge class, much like what was seen in the lectures, and including code to draw the half edge to help with debugging. Feel free to add members as you see fit (e.g., cache important values for expensive computations).
- Face - Face class, much like what we have seen in the lectures (e.g., having an example half edge that has this face as its leftface), but with additional members to store the center position, the area, and the gradient of the heat value across the mesh.
- Vertex - Vertex class, much like what we have seen in the lectures (e.g., having an example half edge that has this vertex as its head), but with additional members for storing a "smooth" surface normal, Laplacian coefficients, heat values, initial conditions, etc.
- PolygonSoup - Simple parser for obj files, which you will modify to scale and center vertices after loading.
- HEDS - The half edge data structure (to construct from a PolygonSoup), which also contains a number of method stubs for you to complete the geometry processing portion of the assignment objectives..
- MeshDrawHeatGeo - This class sets up native IO vertex and index buffers for drawing. It uses the `heat_*` vertex and fragment program for drawing a shaded mesh that also reveals the heat diffusion solution.
- MeshDrawPicking - Much like the MeshDrawHeatGeo class, this class sets up a `pick_*` GLSL program, and likewise sets up buffers to draw every triangle in a unique colour to allow selection of mesh faces. While vertex picking is needed in the assignment, the barycentric coordinates of a the picked location in a triangle allows for the closest vertex to the click point to be selected.

The application uses mouse right clicking to select points on the mesh, and has a simple keyboard interface for loading different meshes, testing the half edge data structure. See the `key_callback` function in `main.cpp` for a reference of all the commands in the provided code, and feel free to add any others you find useful.

The provided code includes `//todo` comments. Use control-shift-f in visual studio (or the equivalent in your IDE) to find all these comments in the code, but likewise it is useful to read through all the provided files.

Steps and Objectives

In each step, you should be able to visually verify that your implementation is working correctly.

1. ***Scale and center meshes (1 mark)***

   Compute an axis aligned bounding box for the loaded mesh vertices, and then modify the vertex positions so that they bounding box is centered at the origin and has a length of 10 in the largest dimension.

We apply the scale directly to the mesh because this can help with validating geodesic calculations later (e.g., for the sphere). Notice how small meshes, such as headtri.obj, fill the screen nicely once you've completed this objective.

2. ***Half Edge Data Structure and Triangulate non-triangular faces (2 marks)***

   Create the half edge and face objects to make up your half edge data structure. See the `createHalfEdge` method in `HEDS` which will help you create twin pointers in your half edge (note that this createHalfEdge code will not be efficient for large meshes).

   Triangulate any faces in the soup which do not have 3 sides (note that the soup wireframe geometry will not draw properly when it contains non-triangular faces, and you do not need to worry about this).

   Take care to ensure that you set all the necessary pointers, and test your half edge data structure by walking around the mesh with the keyboard controls. Set "draw test half edge" in the controls to true, and then use space, n, and w keys on the keyboard. You may need to rotate your mesh or view in wireframe to see the half edge!

3. ***Vertex normals (1 marks)***

   In this assignment, you can assume that the meshes we use are approximating smooth surfaces. As such, add code to your HEDS constructor to compute a per vertex normal as the average of the normals of adjacent faces. You should observe that your objects appear smooth and correctly lit once you have completed this objective.

4. ***GLSL Lighting (2 marks)***

   The provided code uses buffer objects to provide vertex and index arrays to OpenGL for efficient drawing, and these are drawn with the `heat_*`GLSL vertex and fragment programs. Various "uniform" parameters are already set up for you in main. Use the `materialShininess` uniform and note that the value is already set using an appropriate glUniform call in `MeshDrawHeatGeo::drawVBOs`.

5. ***Vertex Valence (1 marks)***

   Implement the valence method in the `Vertex` class, to compute the number of adjacent edges around a vertex.

6. ***Mesh Laplacian(2 marks)***

   Compute the Laplacian for the mesh (see `HEDS::computeLaplacian`).

   Following the explanation in class, keep the vertex area separate from the cotangent operator. The diagonal elements of the cotangent operator should be placed into `Vertex` member `LCii`, and the weights for the adjacent vertices placed in `LCij`, where the weights stored in the `LCij` member of a vertex are only for the neighboring vertices, in either a counter clockwise or clockwise order (for you to choose, and to be consistent).

   You may find it useful to compare the values you compute for the icosahedron in a debugger to see that you get approximately -2.88 for the diagonal (LCii), and approximately 0.57 for the neighbours (LCij).

7. ***Compute heat solution with projected Gauss Seidel(2 marks)***

   See how the `mouseDown_callback` function in `main` sets up a selection request, which is processed in the render function. This sets the selected vertex to be "constrained", with the initial condition u=1. That is the heat at time zero is set to 1 by setting u0 for the selected vertex to 1 and setting all others to zero. We will assume the solution value at this vertex never changes (i.e., treat it as a constraint). As such, the selected vertex will act like a heat source.

   See how the `render` function in `main` calls solveHeatFlowStep with a requested number of steps and a solution t value. Implement this method of the half edge data structure, using the cached Laplacian weights to do a projected Gauss-Seidel (PGS) solve of the heat values, i.e., solve (A - t LC) ut = u0, where A is a diagonal matrix of vertex areas, and LC is the cotangent operator you computed previously. The "projected" part of this PGS solve is easy, as it simply means you should not change the value of ut for the constrained vertex. For all other vertices, however, you should update the ut value assuming all the neighboring vertices have a fixed value.

Your heat solution is set (copied) into the dynamic vertex attributes on each display call, and will be visible when you modify your `heat_frag.GLSL` fragment program to modulate the diffuse material property with `utv`, the varying heat values interpolated across the triangles.

8. ***Compute and visualize heat gradient at faces(1 marks)***

   See that the `render` function in `main` calls `heds->updateGradu()`. Compute the gradient of the heat across faces as described on the top of page 4 of the Geodesics in Heat paper and as covered in class. That is, for each vertex compute the cross product of the opposite side and the normal and scale by the heat value. Add these values and normalize the vector. Note that you do not need to divide by 2 times the area of the face, as you would if you wanted to compute the non-normalized gradient. Check your gradients by toggling the draw grad u boolean with the keyboard controls. These gradients should convincingly point away from the selected heat source vertex.

9. ***Compute normalized gradient divergence at vertices(1 marks)***

   Following the call to `updateGradu` is a call to `updateDivx`, which follows the computation which is also at the top of page 4 of the Geodesics in Heat paper (and discussed in class). In this case, the divergence at a vertex will be a sum of scalar values that are computed at each adjacent face. At each face it is a dot product of the normalized gradient vector with each of the outgoing edge vectors, scaled by the cotangent of the opposite angle.

10. ***Compute distance with Gauss Seidel(1 marks)***

    Solve distance step with Gauss Seidel, and note that there is no constraint used in the solve. Instead we must subtract the minimum value of the solution to align the zero distance with the heat source vertex.

    In this solve you are using just the cotangent operator part of the Laplacian, i.e., LC without the vertex areas, and solving the system LC phi = divx. Recall, as discussed in class, that the inner loop of the Gauss Seidel solve treats all other phi values fixed and updates the phi value of a given vertex based on the value of its neighbors.

    Your distance solution is set into the dynamic vertex attributes on each display call, and will be visible if you modify your per fragment lighting program to modulate the material by the distance values interpolated across the triangles.

11. ***GLSL stripes(1 marks)***

    Adjust your `heat_frag.GLGL` program to add a colour to the lighting based on the heat solution value. You may design your own colourmap (e.g., take inspiration from the paper), or create one similar to the image at the top of this assignment: 1 maps to red, 0.5 maps to black, and 0 maps to blue, with linear interpolation in between.

    Further adjust your fragment program to draw distance stripes. See the `smoothstep` GLSL built in function, which maps an input value to 0 or 1, but with a smooth region in between. Multiplying a smooth step with one minus another smooth step lets you go smoothly from 0 to 1 and then back to 0 at the intervals you choose. Using this step up and down with a modulus (remainder on division) of your distance value (phi) will let you produce stripes of equal distance at the fragment level.

Finished?

Great! Be sure your name and student number is in the window title, in your readme.txt (should you have additional information to submit), and in the top comments section of each of your source files.

Submit your source code as a zip file via MyCourses. Be sure to check that your submission is correct by downloading your submission from MyCourses. You cannot receive any marks for assignments with missing or corrupt files!

Note that you are encouraged to discuss assignments with your classmates, but not to the point of sharing code and answers. All code and written answers must be your own.