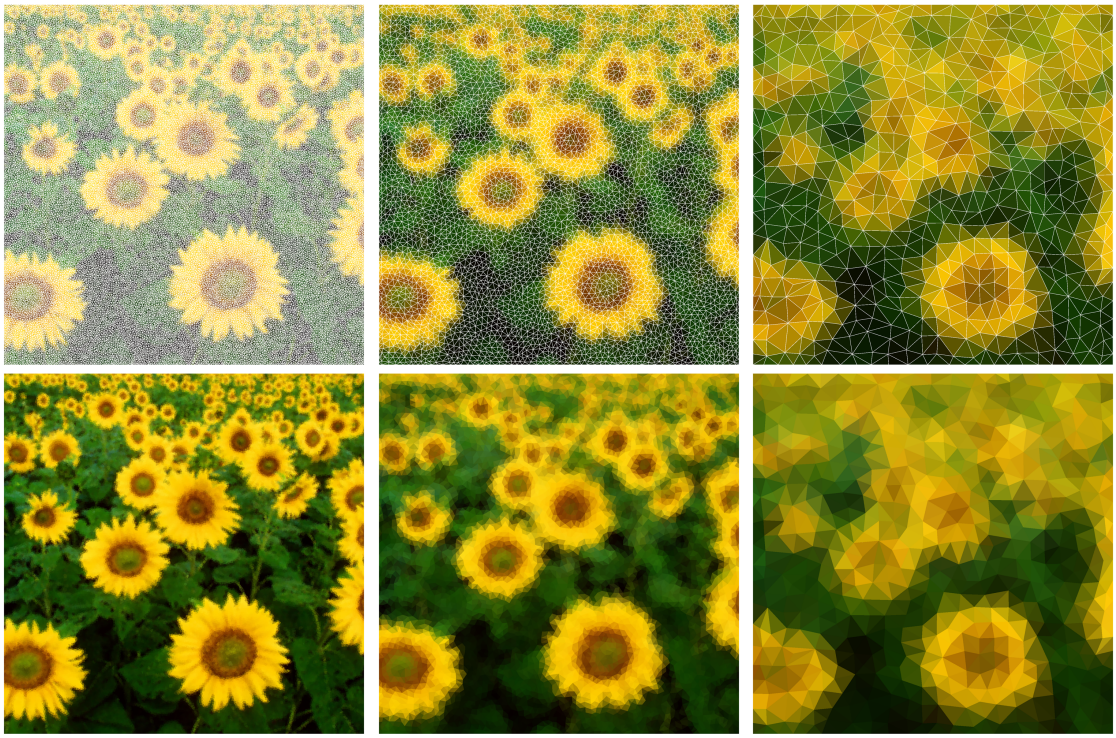
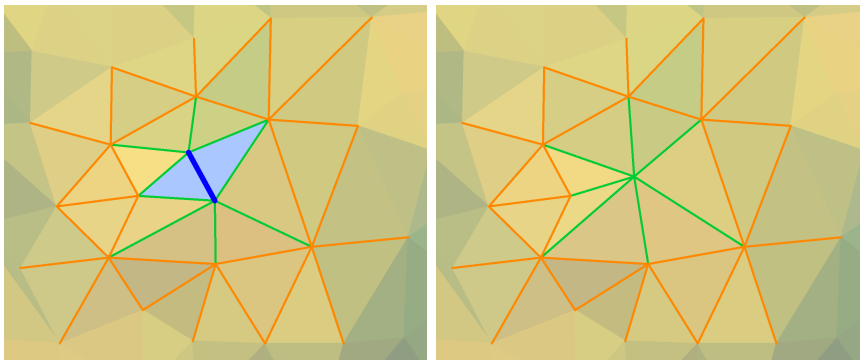
CSCI-1200 Data Structures — Fall 2017
Homework 9 — Priority Queues for Mesh Simplification

In this assignment we will work with a 2D mesh or graph of vertices, edges, and triangles. At the vertices we store colors loaded from an image in `.ppm` format. The goal in this assignment is to reduce the total size of the mesh, but still provide a reasonable approximation of the original image. The images below show meshes with approximately 80,000 triangles, 10,000 triangles, and 1,000 triangles (from left to right), displayed with and without the edges between the triangles drawn with white lines. *Be sure to read the entire handout before beginning the assignment.*



To perform this simplification, we will use a standard mesh processing operation, the *edge collapse*, which is illustrated below. First, we identify an edge in the mesh that should be removed (drawn in blue). Next, we locate the two triangles on either side of that edge (drawn in light blue), and remove these triangles from the mesh. Finally, we locate all triangles using exactly one of the two endpoints of the edge and change that endpoint coordinate to the midpoint of the collapsed edge. Note: We reuse one of the endpoints of the collapsed edge, while the other is deleted. After the edge collapse is finished, the mesh has 1 fewer vertex, 2 fewer triangles, and 3 fewer edges (the blue edge and 2 green edges are removed). Note that if the edge to be collapsed is on the boundary of the image/mesh, the collapse will result in a mesh with 1 fewer vertex, 1 fewer triangle, and 2 fewer edges.



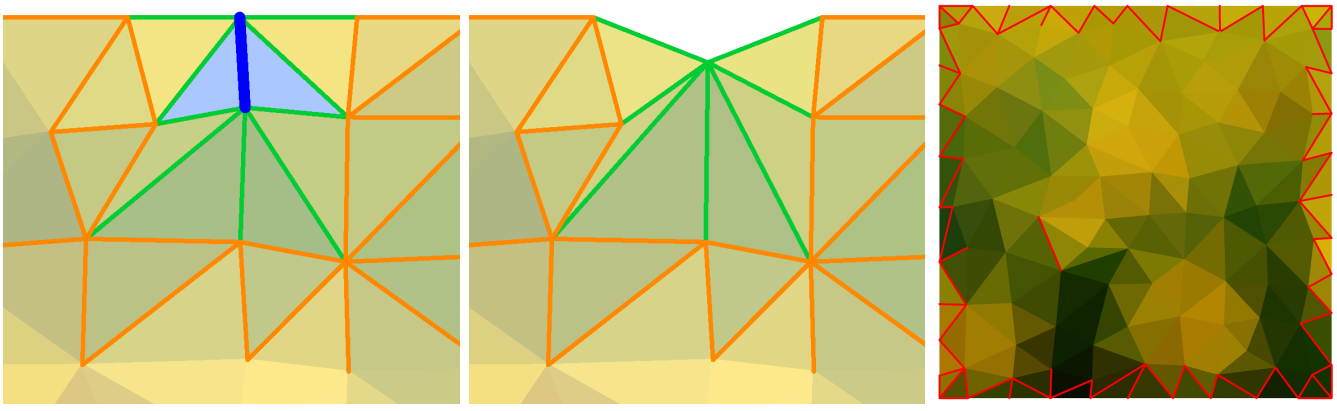
We provide a significant amount of code for this assignment. *Be sure to review all of the provided code before starting your implementation.* Search for the word "`ASSIGNMENT`" within these files to see the sections you need to fill in. The provided code should compile and run on your machine with no edits. The program has reasonable default values for all parameters, but you can override the defaults by providing any or all of these command line arguments (in any order):

`-image <filename>`	*The image file must be a* `.ppm`
`-dimensions <cols> <rows>`	*The size of the grid for the original mesh*
`-target <count>`	*The target number of triangles after simplification*
`-shortest`	*Choose the shortest edge to collapse*
`-random`	*Choose a random edge to collapse*
`-color`	*Choose an edge collapse with least impact on the overall appearance*
`-preserve_area`	*Disallow edge collapses that affect the total area of the mesh*
`-debug`	*Save intermediate meshes & perform extra error checking*
`-linear`	*Perform linear sweep over all edges to find the best edge collapse*
`-priority_queue`	*Use a priority queue to find the best edge collapse*

As the parameters above indicate, we have some choices when selecting the next edge to collapse. Always choosing the shortest edge (which is our default) is known to be a very good algorithm for producing high quality meshes. The resulting triangles in the final mesh are roughly all the same area and many are approximately equilateral in shape.

Your first task is to complete the implementation of the edge collapse. The provided code (which should be rewritten/replaced) simply deletes the two triangles and leaves a hole. This should work "ok" for at least a few collapses, but if you enable debugging and the extra error checking the program will eventually crash with an error. Make sure your completed implementation of the edge collapse works robustly, and can aggressively simplify a large mesh down to a small number of triangles. The `Mesh::Check` function (and later the `PriorityQueue::check_heap` function) are used to sanity check the state of your data as you implement and debug. Use a memory debugger and make sure you have no errors or memory leaks.

Once collapse is working, you can also experiment with the `-preserve_area` command line option. Certain edges should not be collapsed because they change the shape of the boundary of the mesh (as shown below left and middle), and decrease the overall area. The image below right visualizes these *illegal edges* in red. Note that most of these edges are along the boundary of the mesh, but sometimes they appear in the interior. Sometimes an edge collapse will cause triangles in the neighborhood to twist or flip upside down and overlap other triangles. These situations can be detected by calculating the total area before & after the proposed collapse or checking the clockwise/counter-clockwise orientation of the vertices. We provide finished code to detect illegal edge collapses.



In performing an edge collapse, a small neighborhood of the mesh is modified. Triangles change area and edges change length. In the diagrams above we see that the green edges (the edges that were touching one of the endpoints of the collapsed edge) change length. The orange edges (which share a vertex with one or more of the green edges) do not change length; however, their status as *legal* or *illegal* may change. After performing an edge collapse, you should recalculate both the length and the legal/illegal status (stored as `Edge` class member variables for efficiency) of the green & orange edges. Carefully study the `Mesh`, `Triangle`, `Edge`, and `Vertex` classes for helper functions that will help you identify these edges efficiently (without doing an expensive linear sweep through *all* edges in the mesh!).

Even if you are efficient about recalculating the edge length and legal/illegal status, the simplification process using the default `-linear` algorithm for finding the next edge is slow. **Your second major task for this assignment is to modify the code to add a priority queue to the `Mesh` representation.** This priority queue will store all edges in the mesh, organized by priority for collapse. When the `-shortest` option is chosen the priority value is simply the edge length – except edges with *illegal* status are assigned a very large number to ensure they fall to the bottom of the heap.

Note: The use of a priority queue for this problem is somewhat tricky because *the length and/or legal/illegal status of an edge will change as the algorithm progresses.* Therefore, we cannot use the STL Priority Queue and we instead need a custom `PriorityQueue` implementation that allows fast access to elements in the middle of the heap. You'll need to complete the implementation of several functions in this file.

Performance Analysis

Let's assume that the input mesh has v_0 vertices, e_0 edges, and t_0 triangles. Further let's define k to be the number of edges connected to a vertex. For a triangular mesh in 2D, $k = 6$ averaged across the entire mesh. We note that the relative counts of elements in the mesh is a well-studied problem in mathematics. You can read more about the *Euler characteristic* here: https://en.wikipedia.org/wiki/Euler_characteristic.

If we use the `-shortest` criteria for selecting an edge, analyze the performance of the overall program using the `-linear` vs. `-priority_queue` command line options. What is the running time of the program to reduce the mesh to contain the target $= t_{final}$ or fewer triangles? Does the command line argument `-preserve_area` change the answer? Separately analyze the different key functions in the program and justify your answer with a clear and concise writeup. Additionally, using the UNIX `time` command, test your program on different size inputs and target output sizes for the two different command line arguments. Create a neat table summarizing these timing results in your README.txt file. Does it match your theoretical analysis?

Viewing the Output Meshes

The program outputs `.html` files using SVG (Scalable Vector Graphics) format. You should be able to view these files in a modern web browser on your laptop and use the small checkboxes at the top to toggle on & off the visualizations of the edges.

Extra Credit: Prioritizing Edge Collapse by Color

Explore alternate edge collapse criteria that consider not just the length and legality of a collapse, but also determine the relative impact the collapse will have on the overall appearance of the image by analyzing the colors of the vertices. Implement this variation with the optional `-color` command line option. Discuss the quality of your results in your README.txt and include screenshots of your more impressive results.

Submission

Use the provided template `README.txt` file for your algorithm analysis and any notes you want the grader to read. **You must do this assignment on your own, as described in the "Academic Integrity for Homework" handout. If you did discuss the problem or error messages with anyone, please list their names in your `README.txt` file.**