CS 294-74

Mesh Generation and Geometry Processing in Graphics, Engineering, and Modeling

Second Project: Mesh Generator, Surface Reconstructor, or Mesh Processing Program

40% of final grade

Implement a two-dimensional mesh generator or a project of equal complexity, closely related to the material discussed in this class.

You may choose a project from the following list, or devise one of your own in conjunction with me. You are welcome and encouraged to design a project that is related to your research outside this course. (However, please be honorable and don't suggest a project that you've *already* implemented as part of your research.)

- A triangular Delaunay refinement mesh generator.
- An anisotropic triangular bubble mesh generator.
- A triangular advancing front mesh generator.
- A quadtree-based triangular mesh generator.
- A quadrilateral mesh generator based on paving.
- A triangular mesh improver, based on optimization-based smoothing and topological transformations.
- An implementation of isosurface stuffing.
- A surface reconstructor for three-dimensional point sets.
- A surface mesh processing program for two-dimensional surface meshes embedded in three-dimensional space.

To maintain variety, I will allow at most two students to choose any one item from the list above. The sooner you make your choice, the more likely you'll get the project you want. Details on each of these choices appear later in this document.

If you choose a project from this list, you should also choose some aspect of the algorithm as a point of experimentation. (Some suggestions appear in the individual sections on each choice later in this document, although you may also try an idea of your own.) For instance, with an advancing front mesh generator, you might experiment with various methods of handling front collisions, and determine which works best. With a mesh improver, you might experiment with specific ways of finding good transformations or choosing the next transformation. With a surface simplifier, you might experiment with different utility functions. The experiments need not be elaborate.

Deadlines

April 2: By today, you should have discussed with me (in person) what project you are undertaking, and what approach you plan to take to filling in and experimenting with the details of the algorithm.

April 30: You should have arranged a demonstration of your project, to take place no later than today. Contact me in advance for an appointment, which may be in any on-campus location of your choosing. Your program can still be unfinished and very buggy, but I want to see graphical evidence that your project has partial functionality. The demo also presents an opportunity to discuss the details of your project, and whether the direction or scope of the work still seems reasonable. The demo is worth 10% of your score—4 points. If you demo your program by today, you will probably receive the full four points. For each day your demo is late, you'll lose 10% of those four points.

May 7: The source code for your project, including instructions for compilation and running, are due today. This portion is worth 80% of your score: 32 points. For each day your project is late, you'll lose 10% of your gross assigned score out of 32.

May 9: The HTML public documentation for your project is due today. The public documentation is worth 10% of your score—4 points—and will become a permanent part of the course web site. For each day your documentation is late, you'll lose 10% of those four points.

Language: If you write in any language other than C, you are required to give me very complete instructions on how to compile and run your code. You may also be required to obtain for me an account with which I can run your code. If your submission does not run on a Mac, I may need even more help.

Interface: If it's appropriate, your program should use the same file formats as the program Triangle. If it is a two-dimensional mesh generator, it should read a file with the suffix `.poly`, and write a file with the suffix `.ele` or `.edge`. (The latter is a better choice for quadrilateral meshes if you want to view them.) If it is a surface reconstructor for three-dimensional point sets, it should read a file with the suffix `.node`. All the file formats are documented on the Triangle web pages.

An advantage of using these file formats is that you (and I) can use the Show Me visualization program to view and print your input and output. (If you need a three-dimensional version of Show Me, please ask.)

Geometric primitives and data structures: You may find that you have to create a selection of new geometric predicates and constructors for your project. Since this can occasionally be a surprisingly difficult task, please feel free to ask me for help.

You are at liberty to use any geometric data structures you wish. For some projects, you may find it useful to build on the triangulation dictionary you wrote for Project I.

Borrowed code: You are welcome to use publicly available libraries or implementations of the following, so long as none of them was produced by any of your classmates: sorting, selection, binary heaps, balanced search trees, other fundamental non-geometric data structures, command-line switch processing, and geometric primitives like the orientation and incircle predicates. You must write the geometric algorithms all by yourself, unless you've obtained express permission from me to include an external geometric library. Such permission will normally be granted if your project still entails a good deal of geometric algorithm implementation on your part. If your project requires two-dimensional Delaunay triangulations, you may use Triangle. If your project requires three-dimensional Delaunay triangulations, ask me for a fast code.

Speed: You should strive to be able to deal with meshes or triangulations comprising 10,000 elements in a reasonable amount of time. Beyond this, please concentrate on functionality, not optimality. Your first priority is to implement a working algorithm, however slow.

Your first submission: the project. My preferred submission method is that you email me an archive of the code or the secret URL of a file or directory containing all your code. If your code doesn't run on a Mac, special arrangements will definitely need to be made. (Note that Macs run Unix, so if you've written your code in a reasonably portable manner on a Linux or other Unix machine, it will probably work.)

With your code, please send instructions for compiling and running your code. If you use C and Unix, relatively rudimentary instructions will probably do. Be sure to document how to specify the input and output files, and how to choose between any options you have provided. Please also provide example inputs (including some on which you think your program performs well, and perhaps some that show the algorithm's flaws), and let me know which options work best with these inputs, and what features you think these examples demonstrate.

Your second submission: public documentation. Provide, in HTML format, a self-contained web page or set of web pages that discusses your implementation in a manner accessible to outsiders who have some familiarity with mesh processing. Your public documentation will be copied to become a permanent part of the class web site. Therefore, please use relative URLs (rather than absolute URLs) to link your pages and figures together, but use absolute URLs for links to your personal home pages, research pages, and so forth. You may copyright your public documentation, so long as you offer me perpetual permission to serve your documentation from my web sites.

Your documentation should include:

- A description of the algorithm you implemented, with citations of relevant papers. Details that are discussed in the cited papers require only a brief description and a mention of where to find the details.
- Any noteworthy details of your implementation (especially details not discussed in the cited paper or papers), including extra or unusual features, and any departures from the standard algorithms. Where there are several standard algorithms, or the algorithm is ambiguous, how did you resolve the ambiguity? For instance, in an advancing front algorithm, how do you choose the apex of the next triangle?
- A detailed discussion of your experiments, and your conclusions as to which approach is best.
- Lots of figures demonstrating your algorithm in various circumstances. If possible, include figures that show the different results achieved by different choices in your experiments. (To generate figures, I recommend using `xv` to convert Show Me's PostScript output to GIF format, or to grab images from your screen. Let me know if you need help.) A small part of your score will depend on how glitzy/impressive your figures (especially meshes or triangulations) look.

The following things are optional, and will not affect your grade.

- Timings.
- A link to the source code. If you do wish to make your source code public, it is up to you whether to copyright it, release it to the public domain, distribute it under the GNU Public License, or make some other choice.

Project: Delaunay refinement mesh generator. Implement Ruppert's Delaunay refinement algorithm for triangular mesh generation. A working basic implementation (with an experimental component) would merit an A−. For an A+, also implement the modifications for handling small angles described by Miller, Pav, and Walkington. (In neither case do you need to use constrained Delaunay triangulations. If you do wish to implement CDTs, I recommend inserting segments incrementally rather than implementing Chew's algorithm.)

For simplicity, you may want to augment the .poly file format so that each segment includes with one or two extra fields, which indicate whether each side of the segment is interior or exterior.

Possible points of experimentation: How much do Alper Üngör's "off-centers" help reduce the number of triangles in the mesh? How does the order in which you split skinny triangles affect the number of triangles in the mesh?

Project: Anisotropic bubble meshing.

Implement a two-dimensional bubble mesher, perhaps based on the paper by Bossen and Heckbert, the paper by Shimada, or some combination thereof. If you wish, you may assume that the input is a polygon with holes, and the segments are specified in counterclockwise order about the outer boundary, then clockwise about each hole. Allow the use of a space-varying density function (ideally implemented as a linkable C function call) which dictates the approximate desired edge lengths or element areas at each point in the mesh. A working isotropic implementation (with an experimental component) would merit an A−. For an A+, also implement support for generating anisotropic meshes using a space-varying (but not too-quickly varying) ellipticity tensor in the place of the density function.

Possible points of experimentation: What force law seems most effective? What rules for creating/destroying nodes seem most effective?

Project: Advancing front meshing. Implement an advancing front mesh generator for triangular meshes. If you wish, you may assume that the input is a polygon with holes, and the segments are specified in counterclockwise order about the outer boundary, then clockwise about each hole. A working basic implementation (with an experimental component) would merit an A−. For an A+, also implement support for generating anisotropic meshes using a space-varying (but not too-quickly varying) ellipticity tensor.

Possible points of experimentation: Experiment with different methods of choosing the best vertex to use when generating a new triangle. (When should you use an existing vertex, and when should you generate a new one? If you generate a new one, when is an equilateral triangle not ideal?) Experiment with different ways of deciding which edge of the front to advance next.

Project: Quadtree meshing. Implement a quadtree-based triangular mesh generator. A working basic implementation (with an experimental component) for polygons with holes would merit an A−. For an A+, also implement support for meshing general PSLGs. A mesh with no small angles is not always possible in this case, but do the best you can. It may take creativity to find warping rules that work well where multiple input segments intersect at a single vertex.

For simplicity, you may want to augment the .poly file format so that each segment includes one or two extra fields, which indicate whether each side of the segment is interior or exterior.

Possible points of experimentation: How finely does the quadtree need to be subdivided to ensure good element quality? (Hopefully, not as finely as Bern, Eppstein, and Gilbert refine it.) What warping rules tend to produce good elements?

Project: Quadrilateral meshing. Implement the paving algorithm for quadrilateral mesh generation. If you wish, you may assume that the input is a polygon with holes, and the segments are specified in counterclockwise order about the outer boundary, then clockwise about each hole. An implementation that includes enough basic features to support graded element sizes is complicated enough to merit an A+.

Possible points of experimentation: The paving algorithm has parameters that need to be set experimentally.

Project: Triangular mesh improvement. Implement a mesh improver that reads an existing triangular mesh, and uses optimization-based smoothing and topological transformations (including swapping and transformations that insert and perhaps delete vertices) to improve angles as much as possible. You may use a preexisting triangular mesh generator (like Triangle) to create input meshes. You may assume that boundary vertices are fixed and interior vertices are not, but it will improve your grade if you allow constrained smoothing of boundary vertices. (If the angle at a boundary vertex is sufficiently close to 180°, you may assume the vertex is allowed to move along the boundary segment.)

An aggressive and effective working implementation (with an experimental component) would merit an A−. For an A+, implement constrained smoothing of boundary vertices, **or** allow the use of a space-varying

density function (perhaps implemented as a linkable C function call) which dictates the approximate desired edge lengths or element areas at each point in the mesh, and use the transformations that insert or delete vertices to bring the element density close to the density specified by the function.

Possible points of experimentation: You will have to define a procedure that searches for good transformations and smoothing opportunities, and prioritizes them. What procedure gives a good combination of speed and high quality?

Project: Isosurface stuffing. Implement the isosurface stuffing algorithm. Users should be able to choose the α parameters (though I recommend having defaults taken from the paper). Use "continuation" to keep the memory requirements reasonable. An implementation with continuation for uniform meshes would merit an A$-$. For an A+, implement grading in the mesh interior.

Project: Surface reconstruction. Implement a surface reconstruction algorithm for three-dimensional point clouds that is relatively robust against noise. (Note that the three-dimensional Crust algorithm is neither robust against noise nor complicated enough to stand alone as a project.) Ask me if you want three-dimensional Delaunay triangulation code.

Project: Surface mesh processing. Implement a program for processing surfaces meshes, chosen from the many examples in the computer graphics literature. Possibilities include deforming a mesh under user control, remeshing, or surface simplification (although the latter is relatively easy, so you should discuss with me how to elaborate it enough to make it worth an A+).